REIN - A Fast, Robust, Scalable REcognition INfrastructure

Marius Muja*, Radu Bogdan Rusu†, Gary Bradski†, David G. Lowe*
* University of British Columbia, Canada
{mariusm,lowe}@cs.ubc.ca
† Willow Garage, 68 Willow Rd., Menlo Park, CA 94025, USA
{rusu, bradski}@willowgarage.com

***Abstract*— A robust robot perception system intended to enable object manipulation needs to be able to accurately identify objects and their pose at high speeds. Since objects vary considerably in surface properties, rigidity and articulation, no single detector or object estimation method has been shown to provide reliable detection across object types to date. This indicates the need for an architecture that is able to quickly swap detectors, pose estimators, and filters, or to run them in parallel or serial and combine their results, preferably without any code modifications at all. In this paper, we present our implementation of such an infrastructure, ReIn (REcognition INfrastructure), to answer these needs. ReIn is able to combine a multitude of 2D/3D object recognition and pose estimation techniques in parallel as dynamically loadable plugins. It also provides an extremely efficient data passing architecture, and offers the possibility to change the parameters and initial settings of these techniques during their execution. In the course of this work we introduce two new classifiers designed for robot perception needs: BiGGPy (Binarized Gradient Grid Pyramids) for scalable 2D classification and VFH (Viewpoint Feature Histograms) for 3D classification and pose. We then show how these two classifiers can be easily combined using ReIn to solve object recognition and pose identification problems.**

I. INTRODUCTION

In this paper we focus our efforts on the design of a scalable, efficient, and modular architecture (**ReIn** - pronounced "reyn") for the problem of object recognition and pose estimation from 2D/3D imagery. **ReIn** is motivated by the recent advances in object recognition such as reported in the PASCAL VOC challenge [1]. The latest challenge achieved classification rates of 48.6-93.0% and detection rates of 10.2-55.3%. These results, while encouraging for computer vision algorithm research, are nowhere near acceptable for robotics. Missing even five percent of the objects on a table is unacceptable for a table clearing robot (one out of 20 objects is left on the table or is perhaps broken by the robot due to mis-detection). Where humans are involved, missing even 1 percent is unacceptable due to safety reasons. These results indicate a need to combine various detection, recognition and pose algorithms and to combine different sensing modalities in order to attain robust performance. Combining different algorithms and sensing modalities is a non-trivial task. Quite often, recognition performance is traded for speed, and tuning parameters can become complex. We typically might use one or more fast 2D algorithms with low threshold settings to over-detect objects over the whole scene in order to avoid missed detections ("propose") and then use one or more slower algorithms with higher recognition performance to filter out the correct objects from their proposed (sparse) locations ("dispose") followed perhaps by the use of 3D information to get 6 degree of freedom (DOF) object orientation ("6DOF pose"). Thus, recognition algorithms can be used as detectors, recognizers and as filters. Parameters for these algorithms must be adjusted over large amounts of data in order to play these roles.

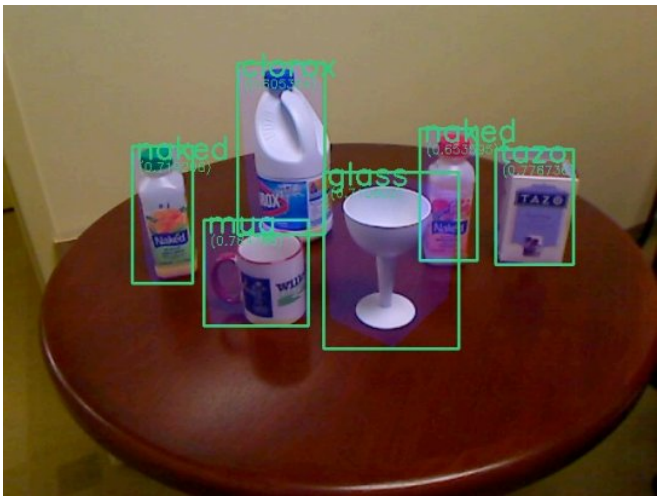

Fig. 1. Object recognition using BiGG and VFH within our **ReIn** infrastructure.

In order facilitate the above needs, we propose a new modular software architecture, **ReIn**[1], that lightly wraps existing detection, recognition and pose algorithms so that they may be used in parallel and in serial without the need to write further code. **ReIn** runs efficiently, taking advantage of shared memory where it is available to reduce data copying. In addition, the architecture automatically provides an online interface to allow changing/tuning parameters for each algorithm as it runs. We demonstrate this architecture showing experimental results combining two of our most recent detectors: BiGG and VFH [2] (see Figure 1). **ReIn** is an open source architecture that defines common interfaces that are shareable by a large pool of object detectors, and

[1] **ReIn** – **Re**cognition **In**frastructure – is a BSD licensed, open source project, available as part of ROS, the Robot Operating System (http://www.ros.org/wiki/rein).

creates an unified methodology for swapping these detectors at run-time using data passing redirections.

Though there are many object recognition architectures in the literature, there aren't too many *generalized* (or better said *standardized*) recognition infrastructures. This is mostly due to the fact that researchers usually insist on individual detectors in their publications, and though they compare them with other detectors, the incentive of combining them together is small.

While there are a some industrial object recognition systems such as Cognex's library [3] and Evolution Robotics ViPR [4], these tend to be domain specific to factory inspection and navigation respectively. There have also been attempts at cognitive perceptual architectures for robotics such as COG at MIT [5]. There are far fewer object recognition architectures devoted to general purpose robotics. Stanford has developed The STAIR Vision Library [6] which is centered around a sliding windows approach, now modifiable by masks, and CMU has produced a system for textured objects [7]. OpenCV [8] is a computer vision library containing many object recognition techniques including a feature detector-descriptor pipeline and OpenCV is in fact is called by the BiGGPy recognition routine described below. But, none of the above addresses run time configurable general object recognition and object pose systems in a generic way. And none does this in a way where we can have the reconfigurable benefits of message passing over a distributed system but still automatically take advantage of shared memory when possible.

The remaining of this paper is organized as follows: a brief description of the **ReIn** system architecture is presented in Section II. The two detectors used to demonstrate **ReIn**, namely BiGG and VFH are presented in Section III. We validate the framework and provide experimental results in Section IV, and conclude with hints towards our related work in Section V.

II. Architecture

To obtain reliable recognition for multiple types of objects we often need to combine different object detectors, each with its own strengths and weaknesses. These detectors can be combined in different configurations, in parallel, in cascade or some mixture of the two. Usually each algorithm has a different interface and combining any two of them involves converting between different data structures which can be inefficient. Also the task of integrating many different detection algorithms into a running system can be non-trivial.

We developed **ReIn**, a Recognition Infrastructure implemented on top of ROS (Robot Operating System), to address these concerns. In **ReIn** an algorithm is viewed as a black-box, with a well defined interface, that consumes a set of *inputs*, produces some *outputs* and is configured by a set of *parameters*. We define a set of interfaces shareable by a large number of object detectors (see Figure 5):

- **Attention operator**: Takes as input an image and/or a 3D point cloud and produces as output a mask, a region of interest (ROI) in the image or a segmentation of the point cloud. An attention operator is usually placed in front of a detector to find "interesting" regions in the image/point cloud where to perform the detection, thus reducing the detector's search space. For example an attention operator could use stereo 3D information to produce regions of interest in an image for a vision-only object detector. An attention operator could also be used to find interesting regions in the environment that the robot should examine in more detail.
- **Detector**: Takes as input an image, a 3D point cloud, a list of ROIs/masks or a list of detections, and produces as output a list of detections and potentially a list of poses. Since different detection algorithms may only require some of the inputs (for example some algorithms only use images, some don't take advantage of regions of interest in the image), the inputs can be used in any combination, configurable by a set of parameters.
- **Pose estimator**: Takes as input an image and/or a point cloud and a set of detections and computes the poses of the detected objects. A pose estimator is used when a pose is required for tasks such as grasping, but the detection algorithms used are not capable of computing the poses of the detected objects.

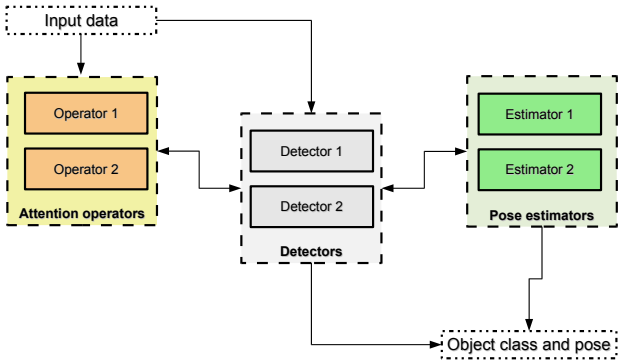

Fig. 2. An overall snapshot into the **ReIn** system architecture, together with its tree major components: attention operators, detectors, and pose estimators.

Adding existing attention, detection or pose estimation algorithms to this infrastructure is accomplished by lightly wrapping them so they implement the above interfaces. Once wrapped, they we can be freely combined in different configurations by redirecting their inputs and outputs.

An additional advantage of wrapping existing algorithms in our infrastructure is the fact that they automatically become plugins (ROS nodelets[2]), capable of being dynamically loaded/unloaded from a system. The plugin system allows for great flexibility, making possible for different algorithms to be loaded as part of the same process, part of different processes of even on different machines (on a compute cluster for example). When loaded as part of the same process, the data exchange between the different algorithms

[2]**nodelet**: a ROS plugin system that provides a way to run multiple algorithms as part of the same process with zero copy cost between them.

happens very efficiently with zero copying, by passing shared pointers.

Since **ReIn** is built on a distributed message passing architecture (that will take advantage of shared memory where available to avoid copying data), it is simple to configure the "roles" that classifiers will take. The configuration of BiGGPy as classifier and VFH as filter used in this paper is shown in Figure 3. Figure 4 shows how easy it is to reverse the roles so that VFH plays the main classifier and BiGGPy the filter. This is done by remapping expected message names such as "/bigg/image" to look for the raw "/image" and so on. In this example, the raw images and point clouds are messages produced by another launch file responsible for sensing (not shown).

```
<launch>
    <node pkg="nodelet" type="nodelet" name="bigg"
          args="standalone bigg_detector/BiGGNodelet" output="screen">
        <remap from="/bigg/image" to="/image" />
        <param name="db_type" value="filesystem" />
        <param name="connection_string" value="$(find bigg_detector)/database/models" />
        <rosparam>
            use_rois: False
            template_radius: 128
            magnitude_threshold: 200
            start_level: 2
            levels: 3
        </rosparam>
    </node>

    <node pkg="nodelet" type="nodelet" name="vfh_classifier" args="standalone
          vfh_classifier/VFHClassifier" output="screen">
        <remap from="~point_cloud"   to="/points2" />
        <remap from="~input_detections"   to="/bigg/detections" />
        <param name="dataset_location" value="$(find vfh_cluster_classifier)/data" />
        <rosparam>
            use_point_cloud: True
            use_input_detections: True
        </rosparam>
    </node>
</launch>
```

Fig. 3. Launch file for **ReIn** where BiGGPy classifies and VFH filters.

```
<launch>
    <node pkg="nodelet" type="nodelet" name="bigg"
          args="standalone bigg_detector/BiGGNodelet" output="screen">
        <remap from="/bigg/image" to="/image" />
        <remap from="/detections" to="/vfh/detections" />
        <param name="db_type" value="filesystem" />
        <param name="connection_string" value="$(find bigg_detector)/database/models" />
        <rosparam>
            use_rois: False
            template_radius: 128
            magnitude_threshold: 200
            start_level: 2
            levels: 3
        </rosparam>
    </node>

    <node pkg="nodelet" type="nodelet" name="vfh_classifier"
          args="standalone vfh_classifier/VFHClassifier" output="screen">
        <remap from="~point_cloud"   to="/points2" />
        <param name="dataset_location" value="$(find vfh_cluster_classifier)/data" />
        <rosparam>
            use_point_cloud: True
            use_input_detections: True
        </rosparam>
    </node>
</launch>
```

Fig. 4. Launch file example reversing the roles so that VFH is the classifier and BiGGPy the filter.

In addition to the features presented above, **ReIn** includes a framework for training object detectors. In order to use this framework, an object detector needs to implement the `Trainable` interface in addition to the `Detector` interface. The advantage of doing this is that all detectors implementing the `Trainable` interface can be trained in an uniform manner, using the same data formats (for example

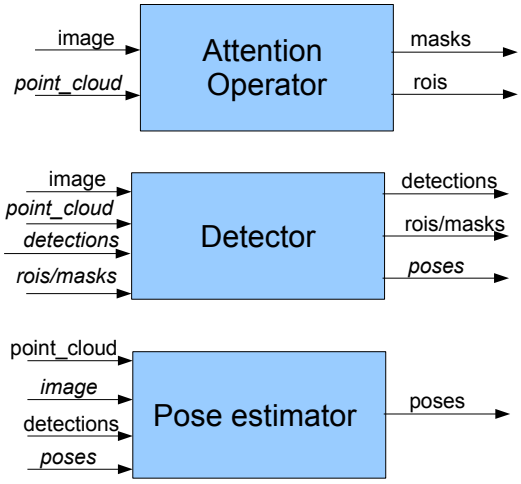

Fig. 5. The set of common interfaces and operators in **ReIn**.

bag files[3] or sets of annotated images) and the same tools. **ReIn** contains support for the saving and loading of the trained models, with the serialization backends configurable at launch time. The current available backends allow for serialization on the local filesystem, as either regular files or in a SQLite database, or on a centralized relational database such as MySQL, PostgreSQL or Oracle.

III. BiGG and VFH

An application example that we are currently pursuing is table clearing with our PR2 platform[4], which involves the recognition of plates, cups, and common household items. Many of these items have no internal texture and many of them are fairly confusable (different types of cups for example). For this task, it is convenient to use fairly dense stereo (using textured light projection) combined with 2D imagery. The addition of 3D information will help identify table planes as well as to verify objects and their pose.

Our object recognition and object pose strategy is to use a fast 2D classifier set at a low recognition threshold to rapidly over-detect objects in order to minimize mis-detections. We call this the object "Proposal" stage where the hope is that no object is missed and the correct object is identified in each location even if there are several false positives. We will then use a 3D object and pose detection algorithm to filter out the incorrect object proposals from the correct ones which we term the "Disposal" stage. Finally, the 3D data will also be used to give us object pose in 6DOF, called the "6DOF Pose" stage.

To validate **ReIn** we implement the above strategy using the following two algorithms: i) BiGG (Binarized Gradient Grids) and it's Pyramid extension (BiGGPy) and ii) VFH (Viewpoint Feature Histogram) which we previously introduced in [2].

[3]"Bag file" is the common format for storing and accessing ROS messages in an efficient way

[4]PR2 (Personal Robot 2) is a robotic platform developed by Willow Garage – `http://www.willowgarage.com`

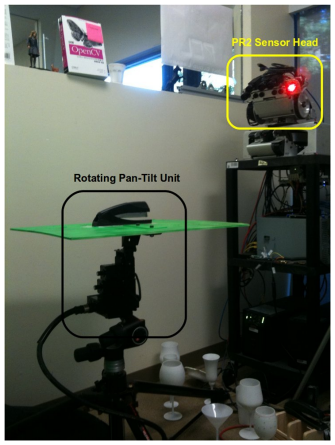

Fig. 6. Pan-Tilt rotating unit used for capturing ground truth for the object pose (both for BiGG and VFH).

A. BiGG and BiGGPy

To develop a rapid object detector for our object proposal stage we drew on ideas from the HoG detector [9] which is essentially a grid of gradient histograms. For speed we adapted ideas from DOT [10] which binarized image gradients and used logical OR instead of histogram bins in each grid cell using a Cosine matching function. We use a simpler normalized count of matching gradients in BiGG described below. The stages of the resulting BiGG algorithm are shown in Figure 7 and described as follows:

1) Gradients are computed from a gray scale input image using OpenCV's [8] Scharr [11] gradient detector.
2) Small magnitude gradients are then removed by thresholding (we used a threshold of 200) and then the gradients are discretized into one of 8 directions ignoring direction of contrast (dark-light and light-dark are equivalent) since objects boundaries might be dark to light or light to dark depending on the background the robot observes them against.
3) To remove spurious gradients, a 3x3 filter is next run that eliminates binarized gradient directions that only appear once in a given 3x3 region.
4) We next compute a gradient "Summary Image" where in each n x n block (typically 7x7) we OR the gradients together to provide some generalization to exact alignment and pose.
5) In the training phase, the above summary gradients are recorded as a gradient template for each view of the object[5]. We used a template of 32x32 in the summary image. In test, the gradient templates are compared, one by one in a sliding window over the image scene. Matching is done by taking the logical AND of each memorized template with a given window of the summary image. Results are normalized between 0 and 1 by dividing the match result by the total number of non-zero gradients in the template. Results are then reported out (optionally with the 6DOF pose memorized with the matched view) and thresholded to declare recognition (thresholds from 0.7 to 0.85 work well).
6) Finally, the learned templates are stored in a database or on disk.

[5]More intuitive training view coverage may be done by mechanically or perspectively OR'ing together gradients at each view over a solid angular part of the view sphere.

Training BiGG is often done by presetting a threshold level, say 0.82, learning an initial view and only learning a new view when none of the set of existing templates for that object is above the threshold.

Some of the advantages of BiGG are that it can be trained at frame-rate. We use a precise pan-tilt turn table to learn views of the object together with a ground truth object pose (see Figure 6). About 350 views of the object are learned in a 15 second sweep of the object covering a half view sphere of the object. Since BiGG uses just grayscale gradients without regard to direction of contrast, it is very tolerant of lighting conditions. The summary image collects all gradients in each patch (here 7x7), and while testing we can sample every 7th pixel in each direction for a 49 times speedup without loss of accuracy. Because BiGG captures gradients in their context, it can take advantage of the interior texture where it can find it but can also recognize textureless and even transparent objects just from their outer contour. Finally, if cleanly written, BiGG can be quite fast and can take advantage of SSE or CUDA instructions to parallelize matching via parallel AND'ing of the summary image patch with the template.

The disadvantages of BiGG are: BiGG uses only logical matches of gradients in its template (zeros do not count) so highly textured scenes will cause many false positives. Although BiGG is quite fast, it still scales linearly with the number of objects learned and this will become a limiting factor for an autonomous robot.

In order to retain the advantages and minimize the disadvantages, we developed a pyramid form of BiGG, termed "BiGGPy". Figure 8 gives a flow chart of the change in moving from BiGG to BiGGPy. Instead of computing the summary image, we start with the full resolution binary gradient image at the bottom of the pyramid and go up the pyramid in each stage by logically OR'ing 2x2 gradient cells from the lower level together forming a pyramid level of half the size in each dimension. Typically we use a 4 level pyramid which forms our data structure to train and test against.

Training BiGGPy then goes from top to bottom of the pyramid. Templates at the top levels of the pyramid are typically associated with many objects, and those at the bottom level with one or a few objects. At the top level, we have a very blurred, non-discriminative, gradient summary image so we set a high threshold in order to break up the learned objects into many different subtrees, threshold that we decay as we descent the pyramid levels and the templates get more discriminative.

We learn a new top template every time no preexisting template matches the object. After the top level, learning proceeds recursively down the best matching template sub-

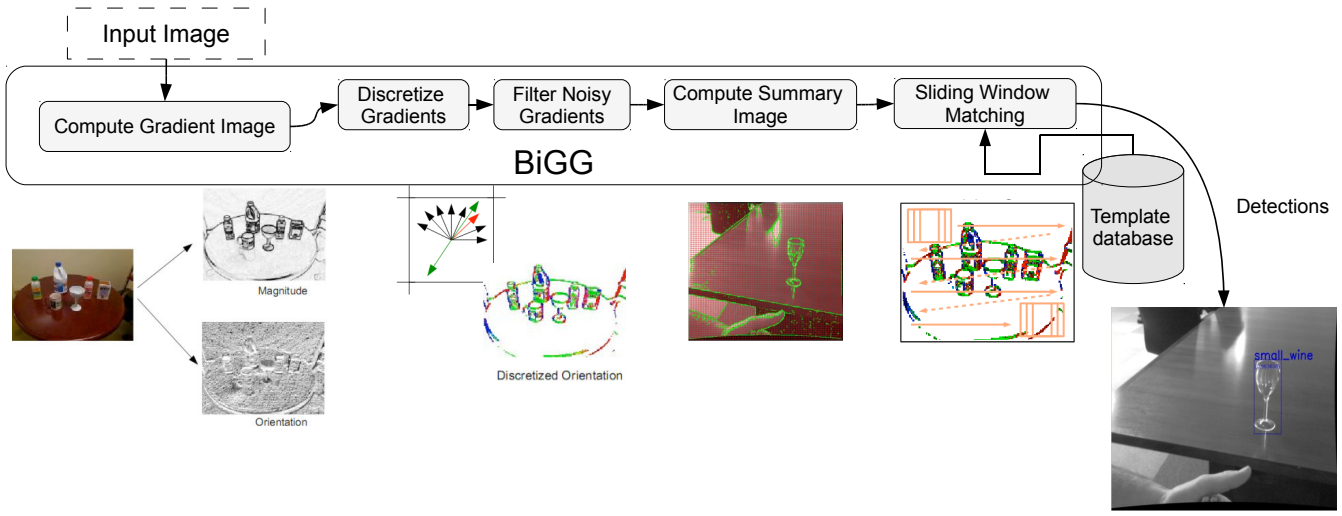

Fig. 7. BiGG (Binarized Gradient Grids) detector architecture. Starting from the left, gradients and their magnitudes of an image are computed. Small magnitude gradients are removed and the rest are binarized into 8 directions ignoring direction of contrast. Next, singleton (noisy) gradients are removed and a gradient summary image is creating by OR'ing gradients together over a local patch. Finally, in recognition mode, a sliding window search is used to find object in the scene.

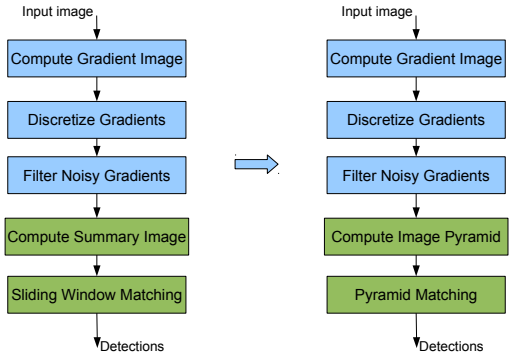

Fig. 8. BiGGPy: Moving from BiGG at left to Pyramid of BiGG (BiGGPy) on the right.

tree. If no existing template is found, another is learned and so on until the bottom of the tree. At the bottom, we record the object class, segmentation mask, (given by using depth cues plus GrabCut [12]), bounding box from the segmentation mask and object pose from the pan-tilt table. In this way, we learn a tree of BiGG masks whose search time is (on average) logarithmic in the number of objects learned.

In test mode for BiGGPy, we compute a pyramid summary image as above. At top we do a sliding window search over the smallest summary image. This produces candidate detection locations. In each candidate location, we descend to the next level of the pyramid and search with lowered threshold that vicinity (plus and minus a pixel in order to avoid misses due to slight misalignments). The search proceeds recursively until the bottom layer where recognitions are reported or until no template matches. Figure 10 depicts this process.

Note that, unlike BiGGPy's logarithmically growing recognition search time with each new object, the memory requirements of BiGGPy grow linearly with new objects. Memory is however much less of an issue than search time. We need the robot to remain rapidly responsive even as it learns a large numbers of objects. But in any given situation, such as clearing a kitchen table for example, we only need to load in the BiGGPy templates that we need. When object recognition search requires templates that are not in memory, the templates can be pulled in from disk. The initial recognition time may be slower but the robot will quickly come up to speed in that given context. Old templates or template trees that have gone stale can similarly be pruned from memory. Thus, memory requirements are much less of a problem than search times.

BiGGPy not only allows recognition to scale to many objects, but the more detailed gradients at the bottom levels of the pyramid are less likely to produce false positives. Although there are a fair number of parameters in BiGGPy such as pyramid levels, pyramid blur, gradient magnitudes etc., in practice we use the default values mentioned above which have performed well and mainly just tune the top threshold value and how fast it decays through lower pyramid levels.

B. VFH

VFH was already presented in our previous work [2] as a standalone 3D *meta-local* descriptor that is extremely efficient for object class recognition and pose estimation at high speeds. The meta locality of the descriptor comes from the fact that it is usually applied to a cluster of 3D points that contains the object to be recognized with a high probability. In our previous work, we assumed that the objects of interest are supported by horizontal planes, and used segmentation and clustering techniques to extract individual objects as

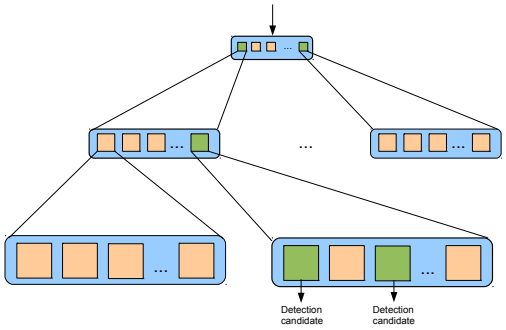

Fig. 9. BiGGPy Tree. The top level coarse templates index many objects, lower levels are more discriminative and index fewer and fewer objects. Thresholds start high at top and decrease as we go lower so that we still get recognitions if slightly misaligned at the more discriminative levels.

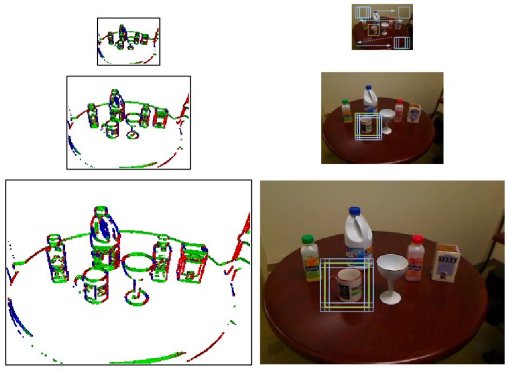

Fig. 10. Object recognition in BiGGPy. At the top of the gradient summary pyramid, a window based search is done using very generic BiGG templates. Where objects are found, that local region is searched at higher resolution by increasingly discriminant detectors. This is done recursively down the pyramid until at the highest resolution the templates are associated with recognized objects.

separate clusters. Another assumption that was made was that the objects are in light clutter (i.e., they can be segmented in a 3D Euclidean sense), as our main application was grasping with the PR2 robot.

Herein we relax these assumptions and use BiGG detectors to obtain segmentation bounds (i.e., masks) in the image space as *proposers*. This means that we no longer need the objects to be separable, as we use VFH to build signatures of the extracted 3D point clusters, and compare against trained models directly.

For the sake of completeness we re-iterate the main steps of the VFH descriptor computation as used in this paper:

- given an image mask, the corresponding 3D points are first extracted as $\mathcal{P}$ (e.g., see Figure 11 top left);
- for each point $\boldsymbol{p}_i \in \mathcal{P}$, a surface normal $\vec{\boldsymbol{n}_i}$ is estimated as explained in [13] (e.g., see Figure 11 top right);
- the first component of the VFH signature is extracted as an extended FPFH descriptor from the centroid $\overline{\boldsymbol{p}_i}$ of $\mathcal{P}$ to each point $\boldsymbol{p}_i \in \mathcal{P}$ [2], [14] (e.g., see Figure 11 bottom):

$$\begin{aligned} \alpha &= \mathsf{v} \cdot \boldsymbol{n}_j \\ \phi &= \mathsf{u} \cdot \frac{(\boldsymbol{p}_j - \boldsymbol{p}_i)}{d} \\ \theta &= \arctan(\mathsf{w} \cdot \boldsymbol{n}_j, \mathsf{u} \cdot \boldsymbol{n}_j) \end{aligned} \tag{1}$$

- given a ray $\vec{\boldsymbol{r}_i}$ to each point $\boldsymbol{p}_i \in \mathcal{P}$, the second components of the VFH signature is extracted as the dot product between each normal $\vec{\boldsymbol{n}_i}$ and $\vec{\boldsymbol{r}_i}$:

$$\gamma = \vec{\boldsymbol{n}_i} \cdot \vec{\boldsymbol{r}_i} \tag{2}$$

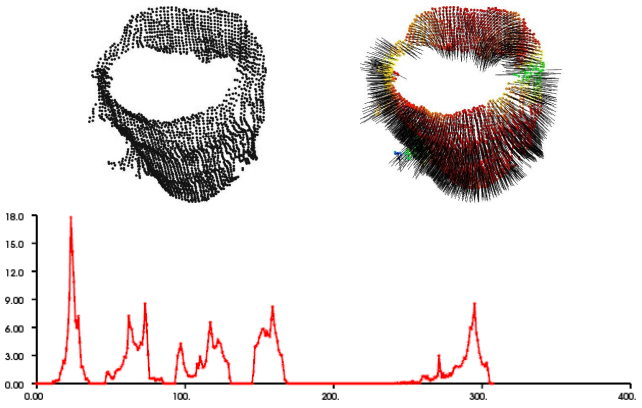

Fig. 11. Top left: point cloud cluster obtained from the BiGG mask; top right: estimated surface normals; bottom: its corresponding VFH signature.

To compare VFH signatures, we construct a kd-tree [15] in a χ^2 space using the following distance:

$$\chi^2(p, q) = \sum_i \frac{(p_i - q_i)^2}{p_i + q_i} \tag{3}$$

After the tree is built, a new cluster $\mathcal{P}$ can be queried and its closest k-nearest neighbors in χ^2 space can be extracted. Each neighbor has a corresponding distance $\boldsymbol{t}_h$ which allows easy filtering either through thresholding or statistics.

Figure 12 presents the closest 9 nearest neighbors for a query cluster representing a glass with a stem (located on the bottom left part of the image). The neighbors are sorted in ascending order from bottom left to top right (row wise), and their distances from the query cluster are shown in green. Please note that in this example, the query cluster was part of the "training" data – which consisted of 2720 datasets representing different objects in various poses – meaning that the distance from the query to itself should be 0. The distance threshold was set to 50 for the purpose of this example.

IV. Validation and Experimental Results

We have evaluated the detection architecture presented above on a set of household objects, some examples of which can be seen in Figures 1 and 13. We used the pan-tilt table shown in Figure 6 to obtain the different object views and provide ground truth for the experiments. For each object we collected about 350 views on the viewing hemisphere.

Some recognition results are shown in Figure 13. The first and third row present views of the different objects used,

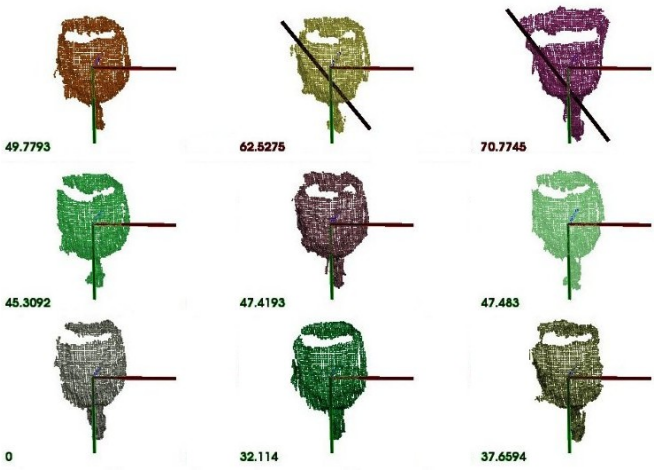

Fig. 12. Example query for a cluster (bottom left) in a χ^2 kd-tree. The closest 9 nearest neighbors are sorted according to their distance in ascending order from bottom left to top right, row-wise. Candidates crossed with a diagonal line are rejected for having their distances larger than t_h.

while the second and fourth row present the precision/recall curves for testing for those objects. In each case the combination of BiGGPy with VFH works much better than BiGGPy alone.

The benefit of combining two objects detectors in a cascading manner in our architecture is also well illustrated by figure 14 showing the combined precision/recall curve for all the objects from figure 13. The first object detector (BiGG) is configured with low detection thresholds to obtain high recall at the cost of many false positive detections (see figure 15), detections which are then filtered by the second object detector (VFH). Using this combination we get nearly perfect precision out to about 85% recall on our test objects. Although the two methods presented above don't obtain perfect recognition (with recall being less than 90%), it is obvious that combining them results in much better performance than using them individually. Since **ReIn** allows for the detectors to be used together in a seamless and efficient manner, it is plausible to think that the precision/recall curve in figure 14 can be further improved by including additional object detectors in the system.

To evaluate the effectiveness of the **ReIn** data passing architecture, we built a simple attention operator that receives a 3D point cloud from the stereo camera at full 640x480, and returns the entire image as a ROI/mask. The point cloud was loaded once and sent in a loop to the operator in order to evaluate how fast we can process the data. By instantiating two copies of the same attention operator as standard processes, we achieved approximately 30Hz (this includes the serialization, memory copy, and deserialization of the data). Using **ReIn**, the two operator instances are sharing the same process as nodelets, thus obtaining a processing rate of approximatively 4400Hz. This numbers are just illustrating the effectiveness of the shared pointer passing architecture, which avoids copying and (de-)serializing the data.

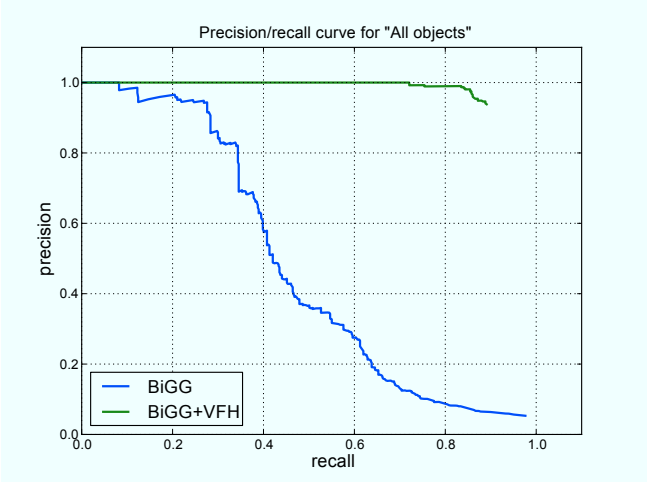

Fig. 14. Precision-recall curve for all the objects.

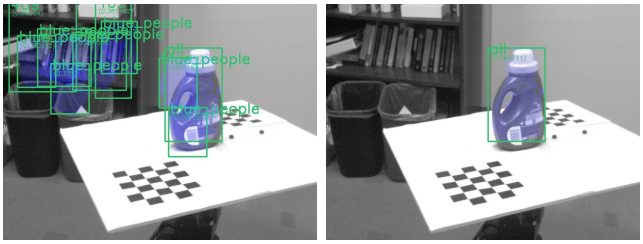

Fig. 15. Example where BiGGPy detects many false positives at left which are filtered out by VFH at right.

V. Conclusions and Future Work

This paper has introduced **ReIn** and demonstrated its use for a particular recognition task: tabletop objects. Our preliminary results are encouraging – not just for the high recognition rates on our combination of BiGGPy and VFH, but on the flexibility of **ReIn** to allow rapid reconfiguration and tuning of combinations of classifiers for a specific task. **ReIn** allows us to flexibly solve this task without a noticeable run time performance penalty. Our use of **ReIn** is expanding, currently we have wrapped several other algorithms: LARK [16], Latent-SVM [17] and HoG [9]. In the near future we will add feature detector-descriptor techniques for textured objects where a bag of words technique is used to propose objects and geometric RANSAC with a 3D feature point model is used to verify ("dispose") the proposed recognition while computing the 6DOF object pose similar to the work described in [7] which we call "TOD" for Textured Object Detector.

We are making **ReIn** available together with the BiGGPy and VFH detectors as a BSD-licensed, open source package in ROS. We encourage other researchers to download and evaluate **ReIn** in their own work.

As future plans, we are currently working on object picking tasks, with a goal to scale to 1000 objects or more. We plan to extend and use **ReIn** as our main recognition framework, this time using a cross-validated voting of

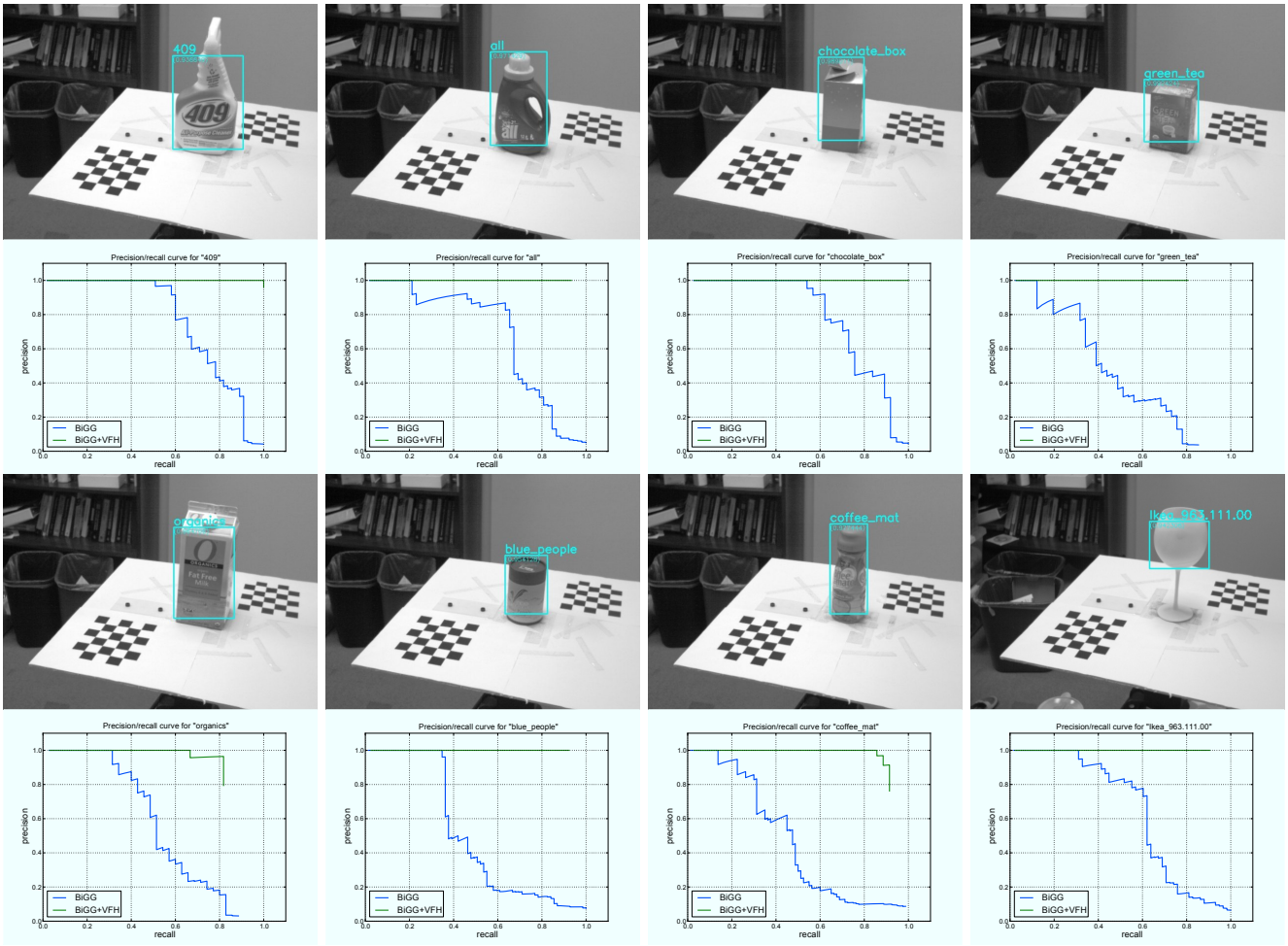

Fig. 13. The top rows (1 and 3) show some detection results while the bottom rows (2 and 4) shows associated precision-recall graphs for these objects. Notice how the combination of BiGGPy with VFH boosts the recognition result.

classifiers to handle this span of textured, transparent and untextured items.

REFERENCES

[1] PASCAL2, "Pascal visual object class challenge workshop," in *http://pascallin.ecs.soton.ac.uk/challenges/VOC/voc2010/workshop/index.html*, 2010.

[2] R. B. Rusu, G. Bradski, R. Thibaux, and J. Hsu, "Fast 3D Recognition and Pose Using the Viewpoint Feature Histogram," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010.

[3] C. Corp., "Visionpro tool library." [Online]. Available: http://www.cognex.com/ProductsServices/VisionSoftware/VisionTools.aspx?id=2406

[4] E. Robotics, "Vipr." [Online]. Available: http://www.evolution.com/core/ViPR/

[5] P. Fitzpatrick, "From first contact to close encounters: a developmentally deep perceptual system for a humanoid robot," in *PhD thesis and technical report AITR-2003-008, MIT EECS*, 2003.

[6] S. Gould, O. Russakovsky, I. Goodfellow, P. Baumstarck, A. Ng, and D. Koller, "The stair vision library (v2.4)," 2010. [Online]. Available: http://ai.stanford.edu/~sgould/svl

[7] A. Collet, D. Berenson, S. Srinivasa, and D. Ferguson, "Object recognition and full pose registration from a single image for robotic manipulation," in *ICRA*, 2009.

[8] G. Bradski and A. Kaehler, "Learning opencv, computer vision with the open source computer vision library," in *O'Reilly Press*, 2008. [Online]. Available: http://opencv.willowgarage.com

[9] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *In CVPR*, 2005, pp. 886–893.

[10] S. Hinterstoisser, V. Lepetit, S. Ilic, P. Fua, and N. Navab, "Dominant orientation templates for real-time detection of texture-less objects," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010.

[11] B. Jahne, H. Scharr, and S. Korke, "Principles of filter design," in *Handbook of Computer Vision and Applications, Academic Press*, 1999.

[12] C. Rother, V. Kolmogorov, and A. Blake, "Grabcut: Interactive foreground extraction using iterated graph cuts," *ACM Transactions on Graphics*, vol. 23, pp. 309–314, 2004.

[13] R. B. Rusu, Z. C. Marton, N. Blodow, and M. Beetz, "Learning Informative Point Classes for the Acquisition of Object Model Maps," in *In Proceedings of the 10th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, December 17-20 2008.

[14] R. B. Rusu, N. Blodow, and M. Beetz, "Fast Point Feature Histograms (FPFH) for 3D Registration," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 12-17 2009.

[15] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," *VISAPP*, 2009.

[16] H. Seo and P. Milanfar, "Training-free, generic object detection using locally adaptive regression kernels," vol. 32, no. 9, pp. 1688–1704, 2010.

[17] P. Felzenszwalb, D. Mcallester, and D. Ramanan, "A discriminatively trained, multiscale, deformable part model," in *In IEEE Conference on Computer Vision and Pattern Recognition (CVPR-2008*, 2008.