Segmentation and Background Elimination with a Multi-Flash Camera

Chris Brown, Randal Nelson
Matthew Parent, Anustup Choudhury, Craig Harman

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 882
October 2005

Abstract

Our goal is to isolate and ultimately identify objects in cluttered and possibly partially-obsuring background environments. Multiple images (usually six in our case) of a scene, including at least five flash images with different flash locations, locate depth edges in the scene. Morphological processing turns the collection of edges into a collection of regions. These regions are candidates for non-background status, and can be compared by color and shape analysis to background regions. Further, we think the edges, labeled with depth and color information, will provide a richer and more reliable set of inputs to an object recognizer like that of Nelson and Selinger, which so far has only had intensity edges to work with.

This research was supported by AppleAid Inc. through a NAVEOD grant.
## Contents

1 Overview and Summary 4

2 Hardware 5
   2.1 Communicating with the camera 6
   2.2 Sample code for interfacing with the camera 6
   2.3 Description of hardware 6
   2.4 Flash timing 8
   2.5 Code used 9

3 Depth Edge Processing 10
   3.1 Overview 10
   3.2 Image Capture 11
   3.3 Iterative Morphology and Region Labeling 11
   3.4 Morphological Tree 12
   3.5 Depth Edge Region Analysis 13
   3.6 Color Region Analysis 14
   3.7 Contour Analysis 14
   3.8 Processing Likely Regions 15
   3.9 Region Joining and Processing 16
   3.10 Results 17

4 Color Image Processing 19
   4.1 Visualizing Clusters in Color Space 19
   4.2 Classification Techniques 23
   4.3 Classifying Depth Regions 23
   4.4 Removing Background Bleed 26

5 Future Work 42

6 Conclusions 43

A Morphological Preprocessing 44

B Additional Depth Edge Techniques 44
C  Canon SDK 48

D  BasicStamp Code 51
1 Overview and Summary

A multi-flash camera originally proposed to create non-photorealistic output [4] does so by finding depth edges in the scene, allowing algorithms to isolate scene components that are likely connected and at different depths. This sort of operation is known as “segmentation” in computer vision; the goal is to find scene regions that correspond to “salient” objects or object parts. Since salience depends on intention, segmentation is not well-defined outside a particular visual task context, but in many tasks important 3-D objects cohere in depth and are separated in depth from other objects and from the background.

Our work so far is really a negative version of the problem. In many applications, there are salient objects that may vary considerably in appearance, with a background that varies less, more slowly, more predictably, or more detectably. Thus finding the background may be an easier job than finding objects directly. In fact this is the strategy investigated in this report, where the domain we are investigating is unexploded ordnance (UXO) on various outdoor backgrounds. Candidate object regions are found by morphological treatment of the depth edges, and they are then compared by traditional pattern classification techniques to background color features. This approach assumes that background features are more predictable than object features; in our domain this indeed appears to be the case. The idea of restricting color feature analysis to a small candidate region, rather than looking for “object-colored” or “not background colored” regions globally in the entire image, is a powerful one. It originated at the AppleAid laboratories, only AppleAid used the Nelson-Selinger object recognizer [1; 2; 5; 6; 3; 7] to find candidate areas of the image. In our experience, and that of AppleAid, it is easy for parts of the background to mimic the color of interesting objects, but the reverse is not true as often.

Along the way, we did a fairly extensive investigation of color spaces and tried several classification techniques for the two-class background-or-not task.

After filtering the candidate regions against the background, we are left with regions that may indeed correspond to objects. We may then wish to move beyond object detection to object recognition. The regions can be linked together, analyzed for shape, and generally treated as regions arising from shapes to be recognized. Also, and somewhat more interestingly, probabilistically labeled lines from the region boundaries (“probable color edge”, “probable depth edge (object occludes background)” “probable depth edge (background occludes object)”) can be passed to the Nelson-Selinger object recognizer. This paragraph’s idea has only been tried at AppleAid labs and is in our opinion a promising avenue for future work.

Our multi-flash camera is built with COTS components, which ironically had to be fairly high-end so that the usual automagic features that defeat consumer choice and control were augmented by features that actually let the user do creative and esoteric operations like disable the flash or change the f-stop and the exposure time. Some reverse engineering was required to fool the slave flashes. The result is a system controlled by a BasicStamp.

Another idea we implemented was to keep a historical tree of the ancestry of regions emerging from morphologically processing the depth edges. We thought that might provide more accurate boundaries for regions of interest, but in fact we have not used it yet.
As a sort of outline to the system we have developed so far, the block diagram of Fig. 1 shows the components of the system and their connections.

![Block Diagram](image.png)

Figure 1: System block diagram.

## 2 Hardware

Except for some editorializing, this section is by Craig Harman, who delivered the working multi-flash camera [4] in late July 2005. First we recall our goal, materials, and strategy.

**Goal:** with one remote wireless operation, trigger the taking of five images with a digital camera.

**Materials:** 1) rather advanced digital camera (we need the capability of manual overrides for many of its “smart” features). 2) remote flashes that are triggered by the camera’s “pre-flash and flash” sequence. 3) BasicStamp-style microcontroller to trigger LEDs and sense flashes.

**Strategy:** The idea is to sense the camera’s main flash, and to emit LED flashes into the receivers on each of four surrounding flashes that mimic the pre-flash-flash sequence, thus triggering a slave flash. We set off the sequence with a wireless remote control to avoid vibrations. We hope to be able
to sequence all five flash images (central, central+left, central+right, central+above, central+below) with one button-press.

### 2.1 Communicating with the camera

Canon provides an SDK for interfacing with their cameras. You can apply to receive the SDK by visiting this website:

http://consumer.usa.canon.com/ir/controller?
=SDKHomePageAct&keycode=Sdk_Lic&fcategoryid=314&modelid=9808&id=3464

or

http://tinyurl.com/af9wx.

Canon responded quickly to one group member’s SDK request, but never responded to another member’s request.

The Canon SDK is designed for use with Visual C++ 6.0, which was released in 1998. The current version of Visual C++ is part of Visual Studio.NET 2003, and Visual C++ will soon be upgraded again when Visual Studio.NET 2005 finishes its beta cycle. Canon’s decision to support only the seven year old version of Visual C++ is actually not unusual. Microsoft deliberately broke portions of Visual C++ when it first released Visual Studio.NET in an attempt to force developers to migrate from Visual C++ to Visual C#. Because so many developers revolted and never upgraded from Visual C++ 6.0, Microsoft allows customers to legally purchase a Visual Studio.NET license and then “downgrade” that license to a Visual C++ 6.0 license by installing the older version of the software.

### 2.2 Sample code for interfacing with the camera

The sample code that Canon provides for interfacing with their cameras uses the justly reviled Microsoft Foundation Class (MFC) library for its user interface. If you are fortunate enough to be unfamiliar with MFC, it can be difficult to extricate the Canon SDK code from the MFC code. We have provided some example code to illustrate how to use the Canon SDK from a command line program in Appendix C. The code also provides instructions for configuring Visual C++ 6.0 to use the Canon SDK libraries and header files.

### 2.3 Description of hardware

Our goal was to have our hardware prototype provide the same functionality as the prototype described in [4]. The original prototype is shown in Figure 5 of the paper, and described in this paragraph from pages 9 and 10:
Our basic prototype makes use of a 4 MegaPixel Canon Powershot G3 digital camera. The dynamic response in the images is linearized. The four booster (slaved Quantarray MS-1) 4ms duration flashes are triggered by optically coupled LEDs turned on sequentially by a PIC microcontroller, which in turn is interrupted by the hot-shoe of the camera. ([4], pp.9-10)

Our rig appears as in Fig. 2. The electronics fit on a small circuit board behind the central baffle, which is needed to defeat the flash when taking the $I_0$ image. The camera, with its central flash, lives behind the baffle as well. The baffle is lifted for the image used for region classification based on color. The rig is heavy, solid, and vibration-free. Its height and angles are configured to match the setup of a miniaturized version on Appleaid’s mobile robot.

There are several differences between our hardware configuration and theirs. While we have had experience programming the PIC microcontroller using CCS’s C compiler, we chose to use a BASIC Stamp microcontroller from Parallax, Inc. for this project. While using C (or assembly) on a PIC
would provide greater control over the hardware, we have found the Stamp to be easier to use for rapid prototyping on small projects like this that do not require sub-millisecond control of hardware or analog inputs. We initially purchased a BASIC Stamp 2, but found it necessary to purchase a Stamp 2px in order to get the functionality needed for our optical trigger.

While we are both using Canon Powershot cameras, they are using a higher-end G3 camera, while we are using an S60. Cameras in the ‘G’ series have a hot-shoe connection that can be used to directly connect and trigger an external flash. Cameras in the ‘S’ series, however, do not output any electrical trigger signals for an external flash. The original prototype connected the hot-shoe adapter to the microcontroller, but this was not an option for us. The only method we have to detect that the camera’s internal flash has gone off is to use an optical sensor that we connect to the microcontroller. This method requires that the camera’s internal flash goes off every time that we want one of the external flashes to go off. We position the light sensor over the Powershot’s internal flash, and cover the light sensor and internal flash with electrical tape to prevent light leakage.

We are using Canon HF-DC1 external flashes instead of the Quantarray MS-1 external flashes described in the paper.

The light detector that we are using was purchased from Radio Shack. The Radio Shack part number is part # 276-142. This package includes both an infrared detector and emitter.

The LEDs that we are using to trigger the external flashes are ”Super Red LEDs” from Lumex (Mfg. part # SSL-LX9053SRC/E, Digi-Key part # 67-1612-ND). We specifically chose an LED that emitted visible light to make it easier to detect problems with the LED wiring.

The LEDs are highly directional, and so need to be positioned as close to perpendicular as possible over the light sensors of the external flashes. We the LEDs are embedded in ”flower arrangement foam” that was purchased from our local fabric store. This foam is rigid, yet very easy to create holes in. The foam blocks are held in place using electrical tape.

Fig. 3 shows the circuit diagram for the multiflash camera.

2.4 Flash timing

The Powershot camera that we are using, like most cameras in its class, sets off its internal flash twice when taking a picture with the flash. The ”pre-flash” is used to illuminate the scene for a split-second so that the camera can determine proper exposure settings. The ”primary flash” is then used during the camera’s integration window.

We experimentally measured the duration between the start of the pre-flash and the start of the primary flash for our Powershot S60 to be 109 +/- 1 ms. The duration of the pre-flash was measured to be between 160 and 200 microseconds.

The Canon HF-DC1 external flashes that we are using go off in response to the primary flash. To simulate the camera’s built-in flash, we position a red LED over the HF-DC1’s light sensor. We blink the LED twice - the first time to simulate the pre-flash, and the second time to simulate the primary flash. The LED is turned on for 1 ms. This is the shortest amount of time that an LED can be turned off and then on using the BASIC Stamp.
2.5 Code used

The microcontroller code is very simple. Each of the four LEDs is directly connected to one of the
output pins on the Stamp. To blink the LED, we just set the pin to high, wait 1 millisecond, and then
set the pin back to low. For example:

```
HIGH 15 ' Pin 15
PAUSE
LOW 15
```

Detecting the Powershot’s internal flash using the light detector is only slightly more complicated.
The Stamp only gives us 1ms granularity when controlling the hardware, but the pre-flash event that
we want to detect has a duration of less than 200 microseconds. The BASIC Stamp family allows us
to detect sub-millisecond trigger signals using the POLL* family of commands. These commands are
not implemented in the Stamp 2, but can be used on any members of the Stamp 2p family.

Polling is set up using the POLLIN and POLLMODE commands:

```
' Polled input, looking for a high state (1) on pin 2
POLLIN 2, 1

' Enable polling
POLLMODE 2
```
The microcontroller can be put to sleep until the expected signal occurs using the following syntax:

```
POLLWAIT 8 ' Wait indefinitely for the desired poll state specified above
```

More information about the syntax of the POLL* commands can be found on the Parallax website. The BASIC Stamp code is in Appendix D.

Looking back on our hardware experiences, it seems to us that any group with minimal digital hardware experience should be able to build a nice cheap effective multi-flash camera out of parts like circuit-board-mounted cameras and simply-fired strobe units or maybe even bright LEDs. We bet that approach will present less trouble than we had with our COTS setup. We spent much effort reverse engineering and working around “smart” automatic features of the complex hardware-software system, and what is needed is really very simple. We are envisioning a system probably based around a Basic Stamp or similar microcontroller whose parameters (exposure time, say) are pre-set according to the local conditions (not unlike the days of actually setting speed, exposure and focus on your camera manually before shooting).

It is not impossible to imagine doing the processing locally (a “computational sensor”), not shipping images back to the main on-board computer.

In any event, the expense, complexity, and massiveness of our hardware should not, we believe, prejudice anyone against the possibilities here.

3 Depth Edge Processing

This section is by Matt Parent, and describes work to create and identify likely UXO regions from the depth-edge image produced by processing the multi-flash images. What seems interesting here is the reliable creation of regions out of edges using morphology, and their subsequent use as candidates to be screened against background by another cue, in our case color.

3.1 Overview

The general algorithm can be broken down into eight basic steps.

1. Capture 6 images
2. Iteratively perform morphological dilation and label regions of depth edge image
3. Build morphological tree
4. Analyze regions by their spatial properties and output most likely regions
5. Analyze regions by their color and output most likely regions
6. Contour Analysis
7. Send most likely regions on for further color processing
8. Join similar regions and process resulting regions

3.2 Image Capture

For our image processing we capture a total of six images. We capture the five images necessary for the depth edge processing described in [4] and an additional color image. The color image is taken using only the primary flash of the camera and is used for color analysis. Taking a color image proved necessary because of variations in the colors present in the $I_0$ image. We also tried to use the max image for color analysis but this image has a tendency to be overexposed (Fig. 4).

![Image](image_url)

Figure 4: On the upper left is an $I_0$ image taken in bright sunlight and on the upper right is an $I_0$ image taken in shade. The lower left is a typical result of the max image and the lower right image is what a typical color image looks like.

3.3 Iterative Morphology and Region Labeling

Some morphological preprocessing is done to separate regions that have bled together in the depth edge image and are considered one very large region (Fig. 5). This preprocessing simply performs three
morphological dilations followed by three morphological erosions. The number three was discovered experimentally and there is an inherent trade-off between the accuracy of the region size and accuracy of the region shape. If too many dilations and erosions are performed the end result is shapes that all look almost circular. Too few results in many regions that are still bleeding together. Region bleeding often causes the region containing the UXO to stand out more than the other regions but it does not accurately locate the UXO in the image.

![Image](image1.png) ![Image](image2.png)

Figure 5: On the left is the edge image without morphological preprocessing and on the right is the same image after morphological preprocessing.

After the morphological preprocessing is finished the following algorithm is performed to build the morphological tree.

**Algorithm 1 Build Morphological Tree**

```plaintext
while Number of regions > 0 do
    label regions in image
    save labeled image into array
    dilate image
end while
```

The structuring element used for the image dilation is a 3x3 array of ones and the region labeling is done by considering 4-connected neighbors to be contained in the same region. At the end of processing we have an array containing each labeled image that was produced by the algorithm which is used for building the morphological tree.

### 3.4 Morphological Tree

The morphological tree is built by analyzing the results of repeated morphological operations. It is worth noting that the process for building the tree assumes that the morphological operations performed never result in a region gaining any pixels. The tree is built using two images at a time, the image being constructed at the current level and the image at the level above that spawned it (now referred to as the image’s parent). Each region is represented as an object and contains information about the region such as its parent, children, centroid, size, compactness, coordinates, perimeter length,
and central moments (Table 1). Internally the tree is represented as a two dimensional array that can dynamically change size and each region object is stored at tree[image number][region’s label] where image number is the number of images analyzed prior to the current one.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>$\sum_{i=1}^{n} \sum_{j=1}^{m} f_R(i, j)$</td>
</tr>
<tr>
<td>Perimeter Length</td>
<td>$|{(i, j) \notin R</td>
</tr>
<tr>
<td>Centroid</td>
<td>$\frac{1}{\text{size}} \sum_{i=1}^{n} \sum_{j=1}^{m} (i, j) f_R(i, j)$</td>
</tr>
<tr>
<td>Compactness</td>
<td>$\frac{\text{Perimeter Length}^2}{\text{size}}$</td>
</tr>
<tr>
<td>Central Moment$_{pq}$</td>
<td>$\sum_{i=1}^{n} \sum_{j=1}^{m} (i - x_c)^p (j - y_c)^q f_R(i, j)$</td>
</tr>
</tbody>
</table>

Table 1: This table contains the formulae for various region statistics. They are computed from an image of size $m \times n$. $R$ is the region whose statistics are being calculated. $f_R(x_{ij}, y_{ij}) = 1$ if $(x_{ij}, y_{ij}) \in R$ and 0 otherwise. $\text{neighbor}(x, y)$ denotes any four-connected neighbor of $(x, y)$. $x_c$ and $y_c$ denote the $x$ and $y$ component of the centroid respectively.

The tree is built one level at a time. Each image is scanned and every pixel with an associated region object is updated. The only pixels that don’t have an associated region object are the pixels with a label of zero because they represent the edges. If a pixel has a label of $m \neq 0$ with coordinate $(x, y)$ then its size is incremented by one and $(x, y)$ is added to its list of coordinates. The region’s parent is represented by a pointer to the region object that contains the pixel located at $(x, y)$ in the parent image. When the pointer to the parent is created, the parent region also adds a pointer to the region as one of its children. The perimeter length of the region is increased by the number of four-connected neighbors that are labeled with a zero or not in the image. After the entire image has been analyzed the centroid, compactness, and moments for each region are calculated. Unfortunately the tree structure hasn’t been very useful for the depth edge images due to the preprocessing and region analysis but may be useful for texture or color. The preprocessing reduces the noise in depth edge image which only leaves fairly significant regions for further processing.

### 3.5 Depth Edge Region Analysis

The region analysis is very simplistic but works surprisingly well. Our initial hope was that calculating central area moments and boundary moments would provide descriptors that could separate the UXO from the background. Unfortunately the background regions do not have consistent features that can be extracted to differentiate them from the UXO by calculating moments or other shape measures. The descriptors that differentiate the UXO from the background most successfully are size and compactness. (Due to the definition of compactness a higher number means that the object is less compact.) The UXO tend to be fairly large with relatively low compactness levels while the background regions tend to be small and compact or large and non-compact. As a result $\frac{\text{size}}{\text{compactness}}$ is a very useful de-
scriptor for finding the UXO. We keep the top 10% of regions ranked by this descriptor for color processing.

### 3.6 Color Region Analysis

The color analysis begins by performing the depth edge algorithm on a set of images of the background. We then take the average HSV color of each region and create a histogram of these values. We then take the likely regions produced by the depth edge analysis and calculate their average HSV value. All regions whose average HSV value is far enough away from its nearest neighbor in the background image (the threshold is currently .08) is kept for further processing. Occasionally some background regions are sent on for further processing but this could probably be avoided by creating the histogram out of multiple background images to increase the chance of including outliers in the background histogram.

![Figure 6](image.png)

Figure 6: On the left is the original color image. In the middle is the output from the depth edge analysis. On the right is the output from the color analysis.

### 3.7 Contour Analysis

After determining the regions that are most likely to be UXO we analyze the contour information to try to determine the source of a particular contour. A given depth edge may be the result of the object represented by the region or it may be caused by the occluding edge of another object (or background matter) between the object and the camera. To determine whether an edge belongs to the object or an occluding edge we use the four edge images produced by the depth edge algorithm and analyze them separately. For any given edgel we head in the direction of the flash that produced it since a left flash will produce edges on the right side of an object. If the region to the left of this edgel is the object then a corresponding counter is incremented. If the region to the left is a different region then the corresponding counter is decremented. After each set of edges is analyzed each edgel of the contours has a value associated with it. If the value is positive then it is considered to be caused by the object itself, a negative value indicates an occluding edge, and a zero indicates an unknown edge.
3.8 Processing Likely Regions

After the most likely regions are found they are saved and sent on for further processing. The relevant region information is passed on in a text file that stores the number of likely regions and the coordinates of every pixel in each region (Fig. 8). The first line of the file contains the number of regions. Each region takes up two lines in the file. The first of these lines states the number of pixels in the region and the next line is a list of each pixel’s coordinates as x,y pairs. These areas can then be analyzed in the max image produced by the depth edge algorithm for color or other measures.

Each region is sent on for further color processing to eliminate any region bleeding that may have occurred (Fig. 9).

```
3
4
1,2,1,3,2,3,2,4
3
5,9,5,10,5,11
1
20,32
```

Figure 8: This is an example text file that would be passed on for further processing. The first line indicates that there are 3 regions of interest. The first region consists of the four points \{(1, 2), (1, 3), (2, 3) (2, 4)\}. The second region consists of the three points \{(5, 9), (5, 10), (5, 11)\}. Lastly, the third region consists of the single point \{(20, 32)\}. 

Figure 7: This image shows the results of the contour labeling overlaid on to the original color image. Red indicates that the contour belongs to an occluding edge, blue indicates an edge belonging to the region, and green indicates an unknown edge.
Figure 9: On the left is the region prior to low level color processing and on the right is the same region after low level color processing.

3.9 Region Joining and Processing

As a result of occluding objects a region of interest may have been split into multiple pieces which we would like to view as part of the same object. The regions are joined by morphology and color. First three morphological dilations are performed on the regions, followed by six morphological erosions, followed by three more morphological dilations. This merges any regions that are very close to each other as well as eliminating any small extraneous pixels that may be left by the lower level color processing. The regions are then analyzed by color and if any regions are similar enough they are considered to be the same object and joined together.

Once the regions are joined the convex hull is computed and used for shape analysis. Using the convex hull results in shape features that are more noise invariant as well as providing a representation of the region that can later be used more easily for robotic manipulation (Fig. 10).

Figure 10: On the left are the likely regions prior to region joining. In the middle are the likely regions after morphological joining. On the right are likely regions after color joining and the convex hull.
3.10 Results

The results shown here show the original image on the left and the interesting regions on the right.
4 Color Image Processing

This section is by Anustup Choudhury.

4.1 Visualizing Clusters in Color Space

The 2D plots shown for the images in the previous report could not separate the clusters we hoped would exist for the background and the UXO. Hence, it was decided that probably 3-D plots might be useful. Therefore the 3D plots of the given picture was plotted for the 3 different color models - HSV, RGB and LST.

In order to clearly find the demarcation present in the UXO and the background - the image (Fig. 11) was manually split into the background and the UXO. For doing this, the `roipoly` function present in MatLab was used and we could get separate images for the background (Fig. 12) and (Fig. 13).

The 3D plots were then plotted. And the results are shown in Figs. 14, 15, and 16.

The red points stands for the pixels representing the values of the UXO and the green pixels stand for the pixels representing the values of the background.
Figure 13: The hand-segmented background region

Figure 14: 3-D plot of HSV histogram of original UXO image pixels.

Figure 15: 3-D plot of LST histogram of original UXO image pixels.
But, as we can see there is no clear separation between the two sets of points and hence it is difficult to use these values for classification. Hence, a new idea was proposed.

According to this new idea, the individual image is partitioned into a certain number of “tiles”. For every rectangular tile, the average values of the three parameters of the model is computed. We were of the opinion that while the background indeed is composed of quite a wide variety of individual pixels, the average properties of those pixels definitely had a different character between UXO and background. Thus we sought some sort of context or average property. Overlapping tiles would also work if we needed more samples. Part of the charm of this project, however, is that it is of low feature dimension and the idea of also keeping the number of samples down to a mentally manageable number appeals to us.

Hence, we divided the background and the UXO image into 300 parts and calculated the average values and plotted them. The results in those cases are shown in Figs. 17, 18, and 19.

As in the previous case, the red stands for the UXO and the green pixels stands for the background.
Figure 18: 3-D plot of LST histogram of original UXO image tiles.

Figure 19: 3-D plot of pixel RGB histogram of original UXO image tiles.
Figure 20: Classified by SVM: UXO. Classified by nearest neighbor: UXO. Classified by nearest centroid: UXO.

After some analysis of the 3D plots of the different models, it was found that the LST color models results in better separation of the UXO and the background cluster and hence we have used the LST models for further analysis of the images.

4.2 Classification Techniques

If we were classifying purely on the basis of color, our job would be to differentiate between clusters of the type shown in Figs. 17, 18), and 19). That is, we have a binary classification problem.

In fact we may not actually use the classification technique in that manner, but it is interesting now to try the basic binary classification techniques on a few images.

We tried three different techniques for the binary classification problem: The Support Vector machine (SVM) (in particular, the freely-available SVMFU package from MIT Labs), and home-brewed nearest neighbor and nearest centroid techniques.

For the nearest neighbor technique, the euclidean distance is computed from the test point to every point present in either of the two “training” clusters; the minimum euclidean distance then tells us which class has the nearest neighbor to our test point.

Similarly, for the nearest centroid technique, the centroids of the clusters are found out first and then the euclidean distance of the test point is found out from the centroid of the clusters. The minimum euclidean distance determines the nearest cluster in the centroid sense.

Figs. 20 — 25 show the images and the results of the different classification techniques.

The above results were obtained using a Gaussian kernel for the SVM. A linear kernel resulted in proper classification of the penultimate image but unexpected classification of the last image. Finally, a polynomial kernel was used for classification of the images which gave us the expected results - that is, the last two images were classified as background by the SVM.

4.3 Classifying Depth Regions

First, we did some work on classifying regions produced by morphological processing using a two-class problem (UXO and background). We used several different classifiers and Figs. 26 — 28 give some good results.
Figure 21: Classified by SVM: background. Classified by nearest neighbor: background. Classified by nearest centroid: background.

Figure 22: Classified by SVM: background Classified by nearest neighbor: background Classified by nearest centroid: background
Figure 23: Classified by SVM: UXO. Classified by nearest neighbor: UXO. Classified by nearest centroid: UXO.

Figure 24: Classified by SVM: UXO. Classified by nearest neighbor: background. Classified by nearest centroid: background.

However, Fig. 29 shows what can go wrong when the true region is merged with a background region.

Results as shown in the two earlier figures show that if we can split the region in such a way that the region occupied by the UXO is more than the background, then we are pretty much assured of getting the right results. Thus we want to look into the behavior of the color features. Fig. 30 shows the appearance of the clusters for the two classes in LST color space. This plot gives us confidence that if the background region were not merged with the UXO region we would have no problem with the classification.

These results motivate the idea of using knowledge of background features to try to fix up the regions supplied by the depth-edge and morphology process. That is, to discard pixels from candidate regions that are classified as “background pixels”. Some of that work is described in the next section.
4.4 Removing Background Bleed

As we saw in Fig. 29 there are instances when a single depth-generated region contains both UXO and background pixels. During the classification of this regions there is a possibility that the classification can be incorrect due to this hybrid nature of the regions.

We devised a technique to postprocess the depth-generated regions using the prior knowledge of color features generated by supervised learning.

Using the color features we have found effective, we obtain clusters of the UXO and the background after training on the data from representative images. We process the depth-generated regions, which could have both UXO and background pixels. For every pixel of the region we check whether that pixel is closer to the UXO cluster or to the background cluster. If the pixel is closer to the background cluster then we eliminate the pixel and if the pixel is closer to the UXO cluster then we retain the pixel in the region. The data below shows the images obtained through this technique and demonstrate that this is a viable way to clean the depth images prior to computing their region or contour features for classification purposes.

Two different techniques were tried for the clean up of the regions. The first uses the nearest neighbor; every region pixel’s features are compared with every point present in both the clusters, and the classification is chosen that has a feature vector closest to the pixel features. The second uses the nearest centroid; every region pixel’s features are compared with the centroid of both the clusters and that class is chosen whose centroid is closest to the pixel features.

This operation is fairly effective, and should work well when the results are treated with some morphological processing to remove the sprinkling of small regions produced near boundaries. (Figs. 31 — 33).
Figure 26: Classifying an M42 against background. SVM classifies Region 5 as UXO and the other regions as background. Nearest Centroid classifies region 5 as UXO and all the other regions as background. Nearest neighbor classifies region 1 and region 5 as UXO and the other regions as background. The reason this technique classifies region 1 as UXO is because as we can see the color of that portion of the background is distinctively different from the other portions of the background and that the average L,S and T values of that region is closer to the UXO cluster than the background cluster. For the M42

Figure 27: SVM classifies Region 1, Region 2 and Region 3 as the UXO and the other regions as the background. Nearest centroid classifies region 1, region 2 and region 3 as UXO and the other regions as background. Nearest neighbor classifies region 1, region 2 and region 3 as UXO and the other regions as background. Here as expected we get the best classification. The reason for the best classification is due to the distinctive difference between the background and the UXO colors.
Figure 28: SVM classifies region 1 as UXO and all the other regions as background. Nearest centroid classifies region 1 as UXO and all the other regions as background. Nearest neighbor classifies region 1 as UXO and all the other regions as background.

Figure 29: SVM fails to classify any of the regions as the UXO. It classifies all the regions as background. In region 1, the true UXO region has merged with the background. Since the region occupied by the background is much larger than the UXO region, the classification result for region 1 is background. Nearest centroid classifies all the regions as background. Nearest neighbor classifies region 2 as UXO and all the other regions as background. From the images, we can see that due to the pronounced effect of the flash on region 2, it probably appears as bright as the UXO resulting in that point being closer to the UXO cluster than the background cluster. This is probably the reason that it classifies the UXO region as background.
Figure 30: Cluster UXO and background features in LST color space.
Figure 31: The original image. Region 1 consists of both the background and the UXO.

Figure 32: Nearest neighbor classification for background cleanup.

Figure 33: Nearest centroid classification for background cleanup.
Following are some related results for the different types of UXO against various backgrounds.

Case 1: When the environment is leaves (Figs 34 – 41).
Figure 36: Original Image.

Figure 37: Nearest neighbor cleanup.

Figure 38: Nearest centroid cleanup.
Figure 39: Original Image.

Figure 40: Nearest neighbor cleanup.

Figure 41: Nearest centroid cleanup.
Figure 42: Original image and nearest neighbor cleanup.

Figure 43: Nearest centroid cleanup.

Case 2: when the environment is rocks (Figs. 42 — 46).
Figure 44: Original Image.

Figure 45: Nearest neighbor cleanup.

Figure 46: Nearest centroid cleanup.
Having a look at the results we observe that the technique of "nearest centroid" SEEMS to perform better than the technique of "nearest neighbor". Also, the bad cases of classification occur when the color of the UXO resembles that of the background or in case of UXOs like the cluster UXO that have metallic color and the color of the environment reflects on the UXO. In these cases it mistakes part of the UXO as the background.

Though the clean up operation is effective in cleaning large chunks of the background regions it tends to leave the edges of the background. Thus we are motivated to attempt to remove the edges of the background which are left behind.

We are using the ("single pixel") L, S and the T parameters for the removal of the background. Now, we propose to use a fourth ("context") parameter for the detection of the concerned regions. We chose the fourth parameter to be the average value of the L or S or T or hue of all the pixels surrounding a pixel.

The next set of figures (Fig. 48 – 53) describes the effects of the anti-bleeding, background-cleaning algorithm using the more sophisticated context-aware four-dimensional feature space.

As expected, the context features that help most are those having to do with the hue of the pixel (Hue and T). The average L value was also a contender but did not perform as well as T on this image. A theoretical justification might be possible. Our feeling is that an independent feature like Hue might be best.

This discarding of single pixels leaves various clouds and ghost outlines of surrounding pixels. They are the type of thing morphology is good at eliminating, so an erosion-dilation pair is applied to remove the small regions and isolated pixels. Figs 54 – 57 illustrate the results.

A final stage of removing isolated pixels (in the four- or eight-neighbor sense) can clean up any stragglers, or this result can be accomplished via morphology as well.
Figure 48: The original “depth region” as delivered by the multi-flash camera and the morphological region-finder. It is polluted in the lower right by background.

Figure 49: Eliminating pixels of the original “depth region” that cluster with the background by the nearest centroid criterion, using only the three single-pixel LST color space features.
Figure 50: Eliminating pixels of the original “depth region” that cluster with the background by the nearest centroid criterion, using only three single-pixel LST color space features and the context-sensitive fourth feature: average L value of surrounding pixels.

Figure 51: Eliminating pixels of the original “depth region” that cluster with the background by the nearest centroid criterion, using only three single-pixel LST color space features and the context-sensitive fourth feature: average S value of surrounding pixels.
Figure 52: Eliminating pixels of the original “depth region” that cluster with the background by the nearest centroid criterion, using only three single-pixel LST color space features and the context-sensitive fourth feature: average T value of surrounding pixels.

Figure 53: Eliminating pixels of the original “depth region” that cluster with the background by the nearest centroid criterion, using only three single-pixel LST color space features and the context-sensitive fourth feature: average hue value of surrounding pixels.
Figure 54: One iteration of erosion-dilation performed on the result of background-elimination using
the average surrounding hue context feature (Fig 53).

Figure 55: Two iterations of erosion-dilation performed on the result of background-elimination using
the average surrounding hue context feature (Fig 53).
Figure 56: One iteration of erosion-dilation performed on the result of background-elimination using the average surrounding T context feature (Fig 52).

Figure 57: Two iterations of erosion-dilation performed on the result of background-elimination using the average surrounding T context feature (Fig 52).
### 5 Future Work

We like the idea of sending our cleaned-up “not background” regions (and convex hulls) directly into Nelson’s OR package. We have not done that yet but our idea is that we have attacked the obscuration and background clutter problems that Nelson’s algorithm was designed to handle. If we have succeeded, then we should be able to present reliable contours to his algorithm, and in fact should be able to supply labels to the contours we present. We think it is possible to label contours, at least with some probability, as object contour (caused by a depth edge with object occluding background), occluding edge (caused by a depth edge with background occluding object), and color edge.

We feel that Nelson’s OR algorithm could fairly easily be modified to take advantage of labeled edges, and that there will be many fewer edges to deal with since it is our job to create “non-background” regions. Thus the hope is for more robust and faster performance from the contour-based OR module.

The following points seem like reasonable things to do sometime when time and resources permit.

1. Make a good suite of test images with the multi-flash camera. Fix most of the relevant system parameters to good values and vary a few. Randal has done quite well quantifying his OR system with really thorough studies. It may be that the NAVEOD or other sponsors would like to see this sort of thing.

2. Develop more and more sophisticated shape features for regions and see if it is possible to classify non-background regions into UXO types. These shape features would be computed on the highly-processed regions emerging from depth-analysis, background-debleeding, smoothing, and convex-hullification.

3. Develop an enhanced version of Nelson’s OR program that can use information about the physical origin of contours – object occluding boundary versus internal object surface details, for instance. Send the contours from regions as above (after all cleanup and smoothing) to an enhanced version of Nelson’s OR program. Include information labeling each contour as to where it came from (region occludes background, background occludes region, color boundary) and the strength of belief in its origin. Quantify relative performance against baseline Nelson OR system using brightness edges. Our thought is that surface markings may vary enough to make the depth-edge contours more reliable for OR purposes.

4. Develop texture features. We’re not sure about this: texture is full of relevant-scale and thus resolution uncertainties. The background is likely to have more interesting texture than the UXO, so maybe texture can join color as a complementary cue to color and depth.

5. A simple extension would make the selection of relevant background (if there were several in the area) adaptive. Further, a robot could learn new backgrounds on command if no fitting ones were found in its library.
6 Conclusions

The multi-flash camera can indeed be built, and we detail our experiences and make some recommendations. It can be used for segmenting certain objects in depth. Besides the algorithms provided by the camera’s originators [4], only standard operations like morphological processing, are necessary to turn the camera’s output of depth edges into candidate depth regions.

Backgrounds can be characterized by color features (and a simple pixel-color context feature) and candidate depth regions can be further classified as “probable background” and “probable object”. We investigated the RGB and LST color spaces, and some simple (nearest-neighbor, nearest-centroid) and more complex (SVM) classification algorithms.

Regions that are classified as object can be further processed to group them, extract basic shape features from the assembled regions. Object classification could occur at this stage, using cheaply-computed 2-D region information (shape and color).

We have pretty much abandoned the idea of just using the depth features along with color and texture features in a unified classification scheme. That might still be a useful idea but as of now it seems more natural to proceed through structured stages of “(pre)processing” rather than just to rely on independently-generated features produced from the raw images.

Last, sophisticated object recognition software [1; 2; 5; 6; 3; 7] could work even better if it were confident about the origins of its input edges (“probably an occluding boundary of the object” vs. “some intensity edge”). Thus we are encouraged that our work can possibly be seen as elaborate pre-processing for one of the world’s most successful object recognition paradigms.
A  Morphological Preprocessing

Fig. 58, which will not appear where we want it, shows the effects of successive stages of the morphological preprocessing that creates regions out of depth edges.

B  Additional Depth Edge Techniques

This appendix is devoted to a couple of additional techniques that have proven useful in producing better depth edges. The first of these techniques is to use the same algorithm using eight flashes instead of four. The additional flash locations are positioned at diagonals to the lens of the camera and the edges are found using the appropriate diagonal edge detector. This technique results in far fewer bleeding regions that also disappear much more quickly through morphology (Figs 59 and 60).

Another technique that produces better depth edges is to use not only the usual “horizontal and vertical edge” filters to detect edges parallel to the flash on the images produced with the standard four flashes, but to apply diagonal edge filters to the four-flash image sets. This technique is not as effective as the eight flash technique described but it is still better than the standard four flash algorithm. Both techniques require less morphological preprocessing resulting in more accurate contours of the regions.
Figure 58: Following the images counter-clockwise from the upper left shows the morphological pre-processing step by step. The left side shows the regions shrinking as a result of dilations on the edges while the regions grow on the right side due to erosions. Preprocessing reduces the amount of region bleeding and also eliminates many of the spurious regions.
Figure 59: On the upper left is the original edge image produced by a four flash setup. The upper right contains the four flash edge image after performing a morphological close operation. On the lower left is the original edge image produced by an eight flash setup. The lower right contains the eight flash edge image after performing a morphological close operation.
Figure 60: On the upper left is the original edge image produced with the standard algorithm. The upper right contains the edge image after performing a morphological close operation. On the lower left is the original edge image produced by the modified algorithm. The lower right contains the edge image after performing a morphological close operation.
This code for the multiflash camera must be slightly extended to allow for a sixth image, one correctly exposed by the central flash, with resulting minimal shadows, to be sent to the color analysis phase.

```
// snapshot.cpp

// A simple program to take a single picture from the command line.

// HOWTO Configure a Command-line Program That Uses the Canon SDK
// [CTH, 15 May 2005]

// Open up a new instance of Microsoft Visual Studio 6.0. Go to the ‘File’
// menu and select 'New...' Select 'Win32 Console Application' from the list
// of project templates, and provide a project name.

// You will need to add 'C:\CDSDK\INC' to the include path, and 'C:\CDSDK\LIB'
// to your library path. To do so, go to the 'Tools' menu, select 'Options',
// and then click on the 'Directories' tab. From the drop-down box entitled
// 'Show directories for:', select 'Include Files', and then add the path
// 'C:\CDSDK\INC' to the list of include directories. Go back to the drop-down
// box, and select 'Library Files', then add the path 'C:\CDSDK\LIB' to the list
// of library paths. If you don't enable multi-threaded support, you will get
// errors like this while compiling:

// Linking...
// nafxcwd.lib(thrdcore.obj) : error LNK2001: unresolved external symbol __endthreadex
// nafxcwd.lib(thrdcore.obj) : error LNK2001: unresolved external symbol __beginthreadex
// Debug/snapshot.exe : fatal error LNK1120: 2 unresolved externals

// In order to use the Canon SDK, your project will need to be compiled with
// multi-threaded support. To enable multi-threaded support, go to the
// 'Project' menu, select 'Settings...' and then click on the 'C/C++' tab.
// From the 'Category:' drop-down box, select 'Code Generation'. From the
// 'Use run-time library:' drop-down box, select 'Debug Multithreaded'
// (unless you are working on a release build, in which case you would just
// use 'Multithreaded').

// You will need to link against several libraries. Go to the 'Project' menu
// again, select 'Settings...' and then click on the 'Link' tab. In the
// text-box labeled 'Object/library modules', add the following libraries
// to the list:

// CDSDK.lib // (the Canon SDK library)
// winmm.lib // (the Multimedia timer library)

// You should now be able to compile this sample program. In order to actually
// run the program, you must copy ALL of the files in 'C:\CDSDK\REDIST' to the
// working directory or a directory in your path. If you do not, you will receive
// mysterious errors, and your program will not work.

// Required by 'cdapi.h', which uses HWND and perhaps other Windows
// datastructures, and required by 'mmsystem.h'
#include <afxwin.h>
#include <stdio.h>
#include <mmsystem.h>
#include "cdapi.h"
```

C Canon SDK

#define GETERRORID( x ) (cdERROR_ERRORID_MASK&x)

static cdUInt32 cdSTDCALL RelCallBackFunc(cdReleaseEventID EventID,
   const void *pData,
   cdUInt32 DataSize,
   cdContext Context );

cdUInt32 cdSTDCALL RelCallBackFunc(cdReleaseEventID EventID,
   const void *pData,
   cdUInt32 DataSize,
   cdContext Context)
{
   return cdOK;
}

int
main(void)
{
   // Timer variables
   DWORD t, t0, dt;

   // TODO: Elevate the priority of this thread to "real-time"
   // (But do we really want to do this? We know SDK library
   // is multi-threaded, so we want to give those threads time
   // to do their work...)

   // Use Multimedia timers with a resolution of 1ms
   if (TIMERR_NOERROR != timeBeginPeriod(1)) {
      printf("Error: Unable to set timer resolution to 1ms\n");
      return -1;
   }

   // Initialize SDK //
   cdVersionInfo SVersion;
   cdError err;
   memset( &SVersion, 0, sizeof(cdVersionInfo) );
   SVersion.Size = sizeof(cdVersionInfo);
   SVersion.MajorVersion = 7;
   SVersion.MinorVersion = 1;
   err = CDStartSDK( &SVersion, 0 );
   if( GETERRORID(err) != cdOK ) {
      printf("Error with CDStartSDK(): ErrorCode = 0x%08X", err );
      return -1;
   }

   // Obtain list of Sources //
   cdSourceInfo m_SelectedSrc;
   memset( &m_SelectedSrc, 0, sizeof(cdSourceInfo) );

   cdHEnum hEnumDevice;
   cdUInt32 count;
   cdHSource m_hSource;

   // Prepare to enumerate list of connected devices
   err = CDEnumDeviceReset(1, &hEnumDevice);
   if( err ) {
      printf("Error with CDEnumDeviceReset(): ErrorCode = 0x%08X", err);
      return -1;
   }
}
// Retrieve number of connected devices
err = CDGetDeviceCount(hEnumDevice, &count);
if (err) {
    printf("Error with CDEnumDeviceCount(): ErrorCode = 0x%08X", err);
    return -1;
}
else if (count == 0) {
    printf("No devices found.\n");
    return -1;
}
// lemma: count != 0

cdSourceInfo* pSourceInfo;
pSourceInfo = new cdSourceInfo;

// Find the first device; we only enumerate the first device
err = CDEnumDeviceNext(hEnumDevice, pSourceInfo);
if (err) {
    printf("Error with CDEnumDeviceNext(): ErrorCode = 0x%08X", err);
    return -1;
}

// Stop enumerating devices
err = CDEnumDeviceRelease(hEnumDevice);
if (err) {
    printf("Error with CDEnumDeviceRelease(): ErrorCode = 0x%08X", err);
    return -1;
}

// Connect to the camera source
err = CDOpenSource(pSourceInfo, &m_hSource);
if (err) {
    printf("Error with CDOpenSource(): ErrorCode = 0x%08X", err);
    return -1;
}
printf("Connected to camera %s\n", pSourceInfo->Name);

// We lock the camera’s UI so that the user can’t screw things up by pressing buttons
err = CDLockUI( m_hSource );
if (GETERRORID(err) != cdOK) {
    printf("Error with CDLockUI(): ErrorCode = 0x%08X", err);
    return -1;
}
printf("Camera UI has been locked\n");

// Enter the "Remote Release Control" mode
err = CDEnterReleaseControl( m_hSource, RelCallBackFunc, NULL );
if (GETERRORID(err) != cdOK) {
    printf("Error with CDEnterReleaseControl(): ErrorCode = 0x%08X", err);
    return -1;
}

// Specify how data will be stored
err = CDSelectReleaseDataKind(m_hSource, cdREL_KIND_PICT_TO_CAM);
if (GETERRORID(err) != cdOK) {
    printf("Error with CDSelectReleaseDataKind(): ErrorCode = 0x%08X", err);
    return -1;
}

err = CDSetFlashSetting(m_hSource, cdFLASH_MODE_OFF, cdCOMP_NA);
if (GETERRORID(err) != cdOK) {
    printf("Error with CDSetFlashSetting(): ErrorCode = 0x%08X", err);
    return -1;
}

// printf("Attempting to take picture...\n");
t0 = timeGetTime();

cdUInt32 NumData;
```c
err = CDRelease(m_hSource, FALSE, NULL, NULL, cdPROG_NO_REPORT, &NumData);
if (GETERRORID(err) != cdOK) {
    printf("Error with CDRelease(): ErrorCode = 0x%08X", err);
    return -1;
}

t = timeGetTime();
dt = t - t0;

printf("Time to take pictures: %d ms\n", dt);
// Some values for dt using the function parameters:
// CDRelease(m_hSource, FALSE, NULL, NULL, cdPROG_NO_REPORT, &NumData);
// without disabling the flash:
// 4109, 4252, 4154, 4255 ms
// after disabling the flash:
// 4573, 4549, 4633 ms
// TODO: Why does it ~0.5s longer after we disable the flash?

/// Cleanup ///
// TODO: Unlock camera UI
// TODO: Get out of release control mode
err = CDFinishSDK();
if (GETERRORID(err) != cdOK) {
    printf("Error with CDStartSDK(): ErrorCode = 0x%08X", err);
    return -1;
}

return 0;
```

### D BasicStamp Code

```
' {STAMP BS2px}
' {PBASIC 2.5}

DEBUG "Initializing", CR

counter VAR Word
counter = 0

' Polled input, looking for a high state (1) on pin 2
POLLIN 2, 1

' Enable polling
POLLMODE 2

Main:
' --- Pin 15 ---
POLLWAIT 8  ' Wait indefinitely for the desired poll state specified above
HIGH 15
PAUSE 1
LOW 15
PAUSE 40

POLLWAIT 8
HIGH 15
PAUSE 1
LOW 15

DEBUG "Pin 15", CR
PAUSE 100
```
' --- Pin 13 ---
POLLWAIT 8
HIGH 13
PAUSE 1
LOW 13
PAUSE 40
POLLWAIT 8
HIGH 13
PAUSE 1
LOW 13
DEBUG "Pin 13", CR
PAUSE 100

' --- Pin 11 ---
POLLWAIT 8
HIGH 11
PAUSE 1
LOW 11
PAUSE 40
POLLWAIT 8
HIGH 11
PAUSE 1
LOW 11
DEBUG "Pin 11", CR
PAUSE 100

' --- Pin 9 ---
POLLWAIT 8
HIGH 9
PAUSE 1
LOW 9
PAUSE 40
POLLWAIT 8
HIGH 9
PAUSE 1
LOW 9
DEBUG "Pin 9", CR
PAUSE 100

' --- Ignore flash ---
POLLWAIT 8
PAUSE 40
POLLWAIT 8
DEBUG "Ignore flash", CR
PAUSE 100

GOTO Main
END
References


