The Cooper Union for the
Advancement of Science and Art

Interactive Foreground Extraction with Superpixels

by

Abrar Rahman

A thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Engineering

September 2014

Dr. Sam Keene
Advisor

The Cooper Union for the Advancement of Science and Art

This thesis was prepared under the direction of the Candidates Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Dr. Teresa Dahlberg
Dean, School of Engineering

Dr. Sam Keene
Candidate's Thesis Advisor

Acknowledgements

I would like to thank my advisor, Professor Keene, for his guidance and advice throughout the project. I would also like to thank my family and friend for their support and encouragement.

Abstract

Interactive segmentation plays a large part in image editing. Segmentation partitions an image into regions that share similarities. The GrabCut algorithm provides a way to get a segmentation of a target object with minimal input from the user and extracts it as foreground. In cases where the regions of the target object is missing from the extraction or there are unwanted background pixels, more user input is required to refine the segmentation. GrabCut performs this refinement by iteratively updating its model for the foreground and background. The novel contribution made is to simplify the refinement process by adding or removing pieces from an image split into atomic regions called superpixels. This reduces the number of interactions needed from the user to extract the desired object from the image.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the field of Computer Vision, segmentation is often used a preprocessing step to perform further processing. This is often used in applications of image matching and object recognition [1]. The task of segmentation is to partion an image into non-overlapping regions that are homogeneous over a set of features. The standards for performing the task include splitting and merging, region growing, and clustering, though clustering algorithms have become prevalent. Automatic segmentation partitions an image into a number of segments based on an algorithm and preset parameters. The pixels within these segments are assumed to have some similarity with each other, so automatic segmentation can reduce the processing needed when doing recognition or compression tasks. Procedures from automatic segmentation can also be used to aid in interactive segmentation, which is the focus of our efforts.

Interactive segmentation is commercially used primarily for photo editing tasks. The tools provided to do so vary in quality of the segmention and the effort required by the user to get the desired segment. All require some form of input from the user and this could be as involved as roughly tracing out the desired segment to as effortless at putting a box around desired segment [3]. The segmentation algorithm then labels the part of the image that it believes the user wants as foreground, while the rest of the image is labelled as background. The foreground is extracted from the image to perform further editing tasks.

Though there are many interactive segmentation tools available, GrabCut stands out because of its ease of use and segmentation performance compared to the competition[3]. Even so, there are cases in which extracting the desired object

from the image requires further supervision from the user. This is done by adding samples into the iterative learning model of GrabCut. If adding or removing regions of monotonous color, this works quite well. There are difficulties though in adding or removing certain types of regions, especially in portions of an image that is noisy or has textures.

To deal with these trouble regions, we group them together with a superpixel representation of the image. A superpixel representation reduces the complexity of an image by going from working with hundreds of thousands of pixels to hundreds of superpixels. A superpixel is perceptually meaningful unit of an image that is most likely uniform in color and texture[5]. There is also a regional constraint on superpixels which keep them small and compact. Regions that give GrabCut trouble in adding or removing are consolidated into one or more superpixels and can be manipulated as a unit. This makes it easier on the part of the user to make adjustments to an extraction.

Chapter 2

Automatic Segmentation

The background information is split into two chapters. This chapter includes commonly used segmentation algorithms that will aid in the understanding of the interactive segmentation algorithms in Chapter 3. The techniques in this chapter only segments the image and does not do anything toward selecting out an object or foreground region, though it does not preclude using them as a step in that process.

2.1 Segmentation Techniques

2.1.1 K-means Clustering

K-means clustering is a simple, but widely used clustering algorithm [6]. The user would need to specify a K for the number of desired segments for the image to be partitioned into. The segments are determined by grouping pixels by set of property values. In the simplest case, the property value is the grayscale intensity of the pixel. In color images the property values could be the color of the pixel represented in RGB or the HSV format. The segments need not be a coherent region unless some property values are included to encourage coherency, such as the pixel position.

To perform the clustering, K clusters are initialized with a random value as their mean property vector. A distance function determines how close the property vector of each pixel is to any of the cluster means. The pixel is associated with

the cluster that is the most similar to it, which corresponds to the least distance as calculated by the distance function. The cluster mean property vectors are updated to be the mean of the vectors associated with the cluster. The process begins again with the new mean property vector. The process continues until convergence, where pixels no longer change their association. After convergence each pixel is associated with K clusters.

The result of clustering an image with a K of 4 using color is shown in Figure 2.1. With a small K we see that all of the people get lumped together with the ground and roof. Furthermore, the parts of the building in shadow on the right fall into the same cluster as the ground and the trees rather than the rest of the building. To be useful for segmenting specific objects, many more segments are needed, and some consideration to regionality is needed to prevent the ground and roof being in the same cluster. Both these aspects can be found in the superpixel algorithm based on K-means in Chapter 4.

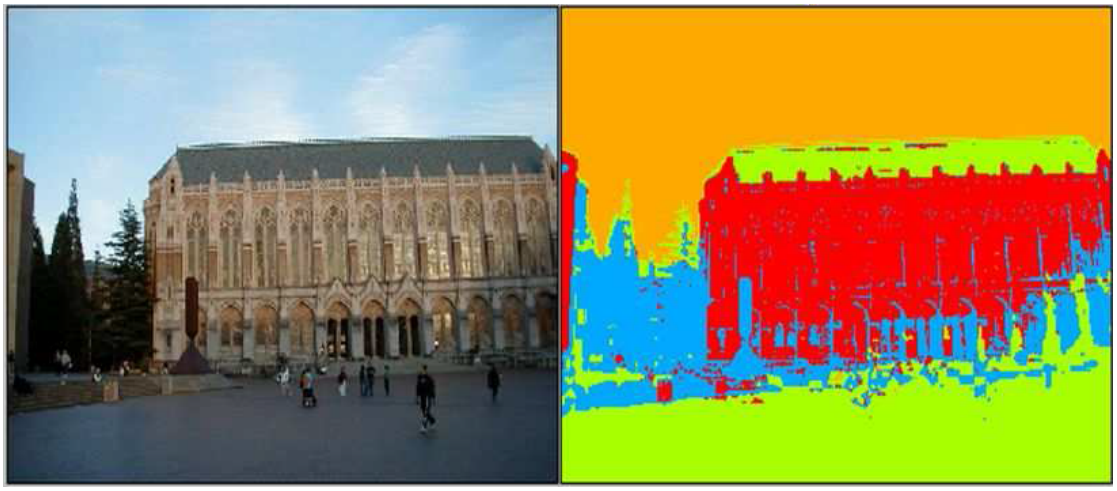


FIGURE 2.1: K-means segmentation into 4 clusters [1]

2.1.2 Mean-shift Clustering

The Mean-shift algorithm works to find dense clusters in a set of data [7]. The procedure for mean-shift first involves taking a histogram of property values of the pixels (e.g intensity) and a window size σ_s (in units of number of bins) is chosen. A pixel is chosen and the corresponding histogram bin set as the center of the window of size σ_s. The property value of the pixels in the bins are averaged and the window center is shifted to bin of that mean value. This averaging and shifting continues until convergence. The center moves toward dense groups of

values incrementally. This procedure is run for each pixel, and by the end, all the pixels will have converged toward local maxima in the histogram. The pixels that converged to the same points are grouped together as a cluster. If the distribution of the historgrom was Gaussian, you can expect that all the points would converge to the center of the curve. If histogram has multiple peaks, points will cluster to those peaks if the window size is not too small that it does not move from its local group of points, or too large so that points gather to intermediary points bewteen peaks.

The Figure 2.2, shows the result of mean-shift with $\sigma_s = 50$ and $\sigma_s = 5$ on the same image as in Figure 2.1. As mentioned before σ_s is the window size for averaging, and as seen in the example, the larger the window, the more clusters in the image. With this method we do see much of the building clustered together and has individually clustered the people in the image. In exchange it does seem to be sensitive to noise with its fuzzy adherence to edges.

(a) $\sigma_s = 50$ (b) $\sigma_s = 5$

FIGURE 2.2: Mean-shift example [1]

2.1.3 Automatic Segmentation Summary

If a user wanted to use the types of presented segmentation techniques to perform foreground extraction, it would be a long arduous process. These segmentation processes are slow to run because because multiple comparison operations are required for every single pixel, and that needs to be done multiple times because its an iterative process. So running the segmentation with different parameters is a

long process. Furthermore, since color is the only metric used for separation, disjoint parts of the image are considered one segment even if they are from different objects. These segmentation techniques were not designed to partition out specific portions of an image, so interactive segmentation tools leverage information from users to speed up segmentation and have a more accurate extraction of the desired object.

Chapter 3

Interactive Segmentation Tools

Interactive segmentation tools provides a way for a user to extract out an object from an image. The underlying algorithm uses user input to try and determine what region the user wants to extract. This region is labelled as the foreground, while the discarded region is taken to be background. Most tools allow for editing of the extraction if portions need to be added or removed.

3.1 Commercial Selection Tools

The most widely used and recognizable user selection tools are included in the Photoshop tool suite and the GNU Image Manipulation Program (Gimp). These tools, the Magic Wand, Magnetic Lasso, and Quick Selection, do not have their algorithms made public, but some details of their inner workings can be inferred.

3.1.1 Magic Wand

The Magic Wand (known as Fuzzy selection in the Gimp image editing program) is a color based selection tool [8]. Starting off with a point or region, adjacent points are added to the selected region if it is within a tolerable deviation from the color based statistics of the originally selected region. The tool works well in cases where the target selection is of a solid color and objects with clearly defined edges. Otherwise it might be impossible to get close to the selection you want, especially if the background has similar color statistics.

3.1.2 Magnetic Lasso

The Magnetic Lasso (similar to the Intelligent Scissors and Path tool in Gimp) finds a contour to select around an object [9]. The user selects points around the edge of the target object by placing nodes around the edge of the object. The algorithm traces out the object by performing edge detection between nodes and having the selection follow this detected edge. This tool works very well when the target object has a well defined edge. Otherwise if the area between two nodes is highly texturized or noisy it is unlikely that the edge detector will find a path that follows the contour well.

3.1.3 Quick Selection

Though the algorithm for Photoshop's Quick Selection tool has not been made public, there is another tool called Paint Selection [2] that was inspired by Quick Selection and has the same user interface. In both tools, the user clicks within the target selection and drags over the object of interest. The points passed over provides the initial seeding for the learning. The selection region grows from the initial seeding to encompass what it believes to be the object, as shown in Figure 3.1. When growing the region snaps to align with strong boundary edges. The selected foreground is modelled as a color Gaussian Mixture Model (GMM) with four components. An eight component GMM is learned from the random sampling from the background. When the selection is dragged to a new region, the backgound is resampled and recalculated. To actually cut out a foreground region, a graph-cut is used. The details of graph-cuts will be discussed in the following section.

3.2 Graph-cut Based Tools

Both the Foreground Selection tool in Gimp and the GrabCut algorithm uses a form of Graph-cut as the basis for their operation. Both require some kind of input for the region of interest (rough outline for the Foreground Selection tool and a bounding box for GrabCut) and the Froeground Selection tool requires some foreground labelling while GrabCut does not. Both use some form of the following Graph-cut algorithm to perform its segmentation and extraction.

FIGURE 3.1: Illustration of Paint Selection use [2]

3.2.1 Graph-cut

Given a greyscale image is given as the array $\boldsymbol{z} = (z_1, \ldots, z_n, \ldots, z_N)$, and that $\underline{\alpha} = (\alpha_1, \ldots, \alpha_N)$ for each image pixel to indicate a label for the pixel. For a hard segmentation (no variable opacity at the edges) $\alpha_n \in \{0, 1\}$, where 0 indicates a background label and 1 indicates foreground. The parameter $\underline{\theta}$ is a histogram of grey level values defined as:

$$\underline{\theta} = \{h(z; \alpha), \alpha = 0, 1\} \tag{3.1}$$

one for both background and foreground. The histograms are normalized such that they sum to 1: $\int_z h(z, \alpha) = 1$. The task is to learn the labels $\underline{\alpha}$ given the image $\boldsymbol{z}$ and model $\underline{\theta}$.

The energy function $\boldsymbol{E}$ is defined so that cutting out the minimum would result in a good segmentation of the object based on the observations of the foreground and background. This is written the form of a Gibbs energy function:

$$\boldsymbol{E}(\underline{\alpha}, \underline{\theta}, \boldsymbol{z}) = U(\underline{\alpha}, \underline{\theta}, \boldsymbol{z}) + V(\underline{\alpha}, \boldsymbol{z}) \tag{3.2}$$

where U evaluates whether the labels $\underline{\alpha}$ fits the data $\boldsymbol{z}$ given the histrogram models from $\underline{\theta}$. This is defined as:

$$U(\underline{\alpha}, \underline{\theta}, \boldsymbol{z}) = \sum_{n} -\log h(z_n, : \alpha_n) \tag{3.3}$$

and the smoothness term V is defined as:

$$V(\underline{\alpha}, \boldsymbol{z}) = \gamma \sum_{(m,n)\epsilon \boldsymbol{C}} dis(m,n)^{-1}[\alpha_n \neq \alpha_m] \exp -\beta(z_m - z_n)^2 \tag{3.4}$$

where $\boldsymbol{C}$ is the set of neighboring pixels and $dis(\bullet)$ is the Euclidean distance between adjacent pixels. This incentivises coherent regions minimizing the possibility of having holes in the segmentation. If the constant $\beta = 0$ then smoothness is encouraged everywhere (to a degree, controlled by the constant γ) and is called the Ising prior. β is chosen to be:

$$\beta = (2\langle (z_m - z_n)^2 \rangle)^{-1} \tag{3.5}$$

where $\langle \bullet \rangle$ is an expectation over an image sample. This value of β ensures that exponential term in 3.4 switches between high and low contrast. The constant γ was chosen to be 50 by experimental optimization.

With the energy model in 3.2 the selection is made by:

$$\hat{\underline{\alpha}} = \arg\min_{\underline{\alpha}} \boldsymbol{E}(\underline{\alpha}, \underline{\theta}) \tag{3.6}$$

Several changes are made to build upon this graph-cut to make it into the GrabCut algorithm.

3.2.2 GrabCut

The GrabCut algorithm is the one of the easiest methods for a user to extract a foreground object from an image as it requires the least interaction from the user[3]. In many cases, the only input required from the user is to draw a bounding box around the target object. If the selection is not ideal, the user could provide more information in form of adding samples to the foreground and background to get a more optimal cut. There are three main developments made in GrabCut to make improvements on Graph-cut [3].

The first is that the grayscale histogram is replaced with a color Gaussian Mixture Model (GMM). This takes into account multiple colors expected to be in the foreground and background. A GMM is a convex combination of Gaussians (a linear combination that sum to 1 and has non-negative coefficients). Samples from the image are used as observations to use the expection-maximization algorithm to learn the parameters of the GMM.

The next improvement made is that the one shot minimum energy cut is replaced with an iterative algorithm that alternately learns parameters and performs segmentation. The third is that the user can give an incomplete labelling; the foreground region does not need to be explicitly labelled.

The new model is a $K = 5$ component GMM; one for each the foreground and background. To introduce the GMM into the energy function, a vector $\boldsymbol{k} = \{k_1 \ldots, k_n, \ldots, k_N\}$ is made with $k_n \in \{1, \ldots, K\}$. Each pixel has a k_n value that assigns it to one of the mixture components from either the foreground or background GMM. $\alpha_n = 0$ or 1 determines whether it is a background or foreground component.

The new updated energy function from 3.2 is now:

$$\boldsymbol{E}(\underline{\alpha}, \boldsymbol{k}, \underline{\theta}, \boldsymbol{z}) = U(\underline{\alpha}, \boldsymbol{k}, \underline{\theta}, \boldsymbol{z}) + V(\underline{\alpha}, \boldsymbol{z}) \tag{3.7}$$

with the new U as:

$$U(\underline{\alpha}, \boldsymbol{k}, \underline{\theta}, \boldsymbol{z}) = \sum_n D(\alpha_n, k_n, \underline{\theta}, z_n) \tag{3.8}$$

where:

$$D(\alpha_n, k_n, \underline{\theta}, z_n) = -\log p(z_n|\alpha_n, k_n, \underline{\theta}) - \log \pi(\alpha_n, k_n) \tag{3.9}$$

with $p(\bullet)$ as a Gaussian probability distribution and $\pi(\bullet)$ are mixture weight coefficients, so up to a constant:

$$\begin{aligned} D(\alpha_n, k_n, \underline{\theta}, z_n) = &-\log \pi(\alpha_n, k_n) + \frac{1}{2} \log \det \Sigma(\alpha_n, k_n) \\ &+ \frac{1}{2}[z_n - \mu(\alpha_n, k_n)]^\top \Sigma(\alpha_n, k_n)^{-1} [z_n - \mu(\alpha_n, k_n)] \end{aligned} \tag{3.10}$$

The smoothness term V is unchanged from equation 3.4.

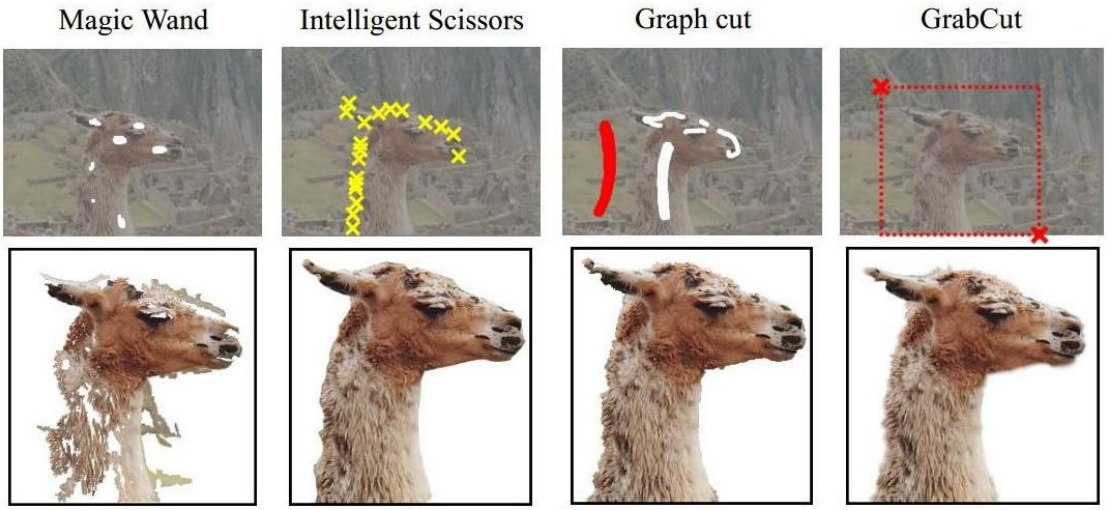


FIGURE 3.2: Algorithmic comparison of several tools. Red markings indicates labelling background while white markings indicates labelling foreground [3]

For the energy minimization, it is done iteratively. The initial labelling is done with a bounding box with the outside being labelled as background and the inside of the box labelled as foreground. The initial GMM's are learned from these regions and the first segmentation made. After the initial segmentation, the segmented section is taken as the foreground and the GMM's recalculated. A new segmentation is made, and the process is repeated until convergence.

3.2.3 User Interaction with GrabCut

Figure 3.2 shows a comparison of the relative effort needed to use some of the dicussed tools plus their segmentation results. Effort is shown by how many interactions the user needed to get the desired segmentation. It is difficult to have a direct comparison of which algorithm provides the best extraction result because it is very dependant on the skill of the user with the tool and how much user input is taken. Even so is evident that though GrabCut required the least user input out of the shown methods and closely matches the extraction results of using a graph-cut.

Though it works well in many cases, in some, additional input from the user is sometimes needed to improve the segmentation. Figure 3.3 shows an example where all that is needed from the user in GrabCut is the bounding box. In this

FIGURE 3.3: Easy extraction example

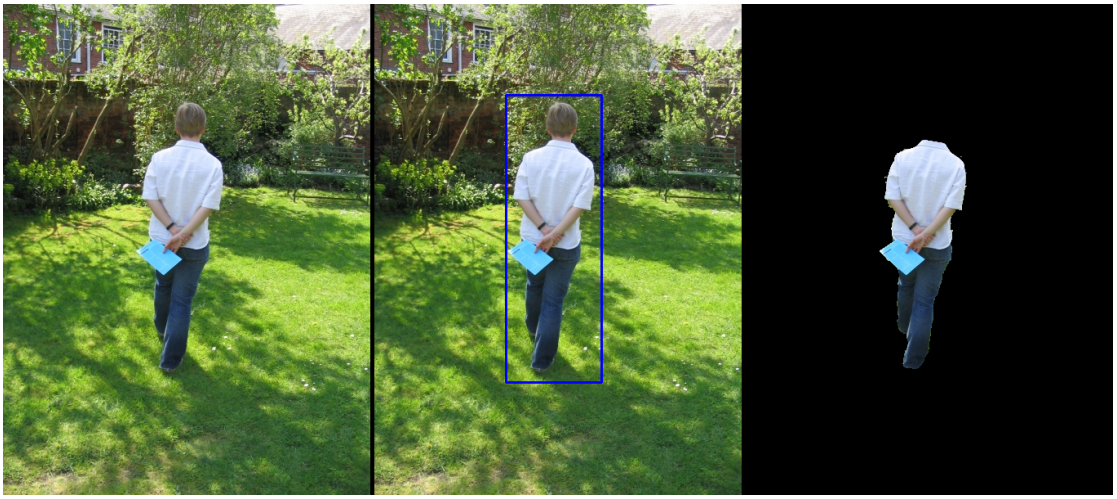


FIGURE 3.4: Harder extraction example

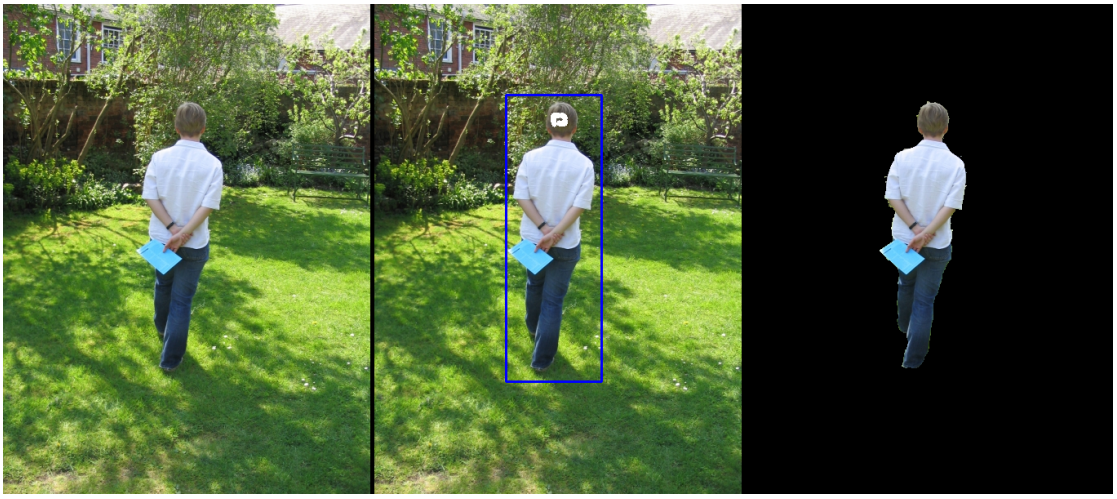


FIGURE 3.5: Harder extraction example with more user imput

example the taget object, the flowers, have clearly defined edges and have a sharp color contrast with the background. In Figure 3.4 using only a bounding box gives the majority of the target person, but their hair is missing from the selection. It is understandable that the hair would be labelled as background because the wall in the background is close in color. To remedy this, further unput from the user is needed to label the missing hair region as shown in Figure 3.5.

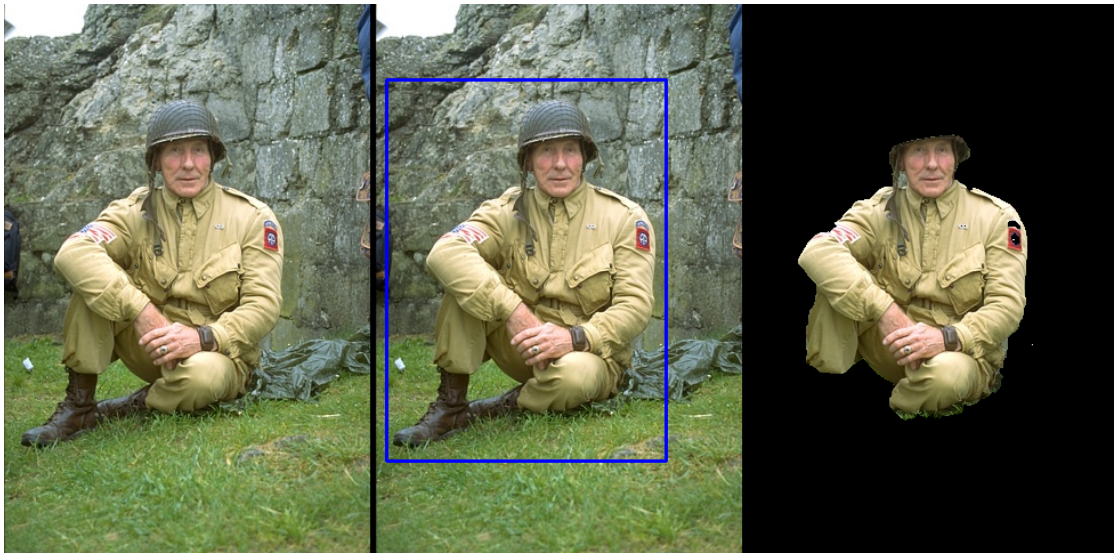


FIGURE 3.6: Harder extraction example

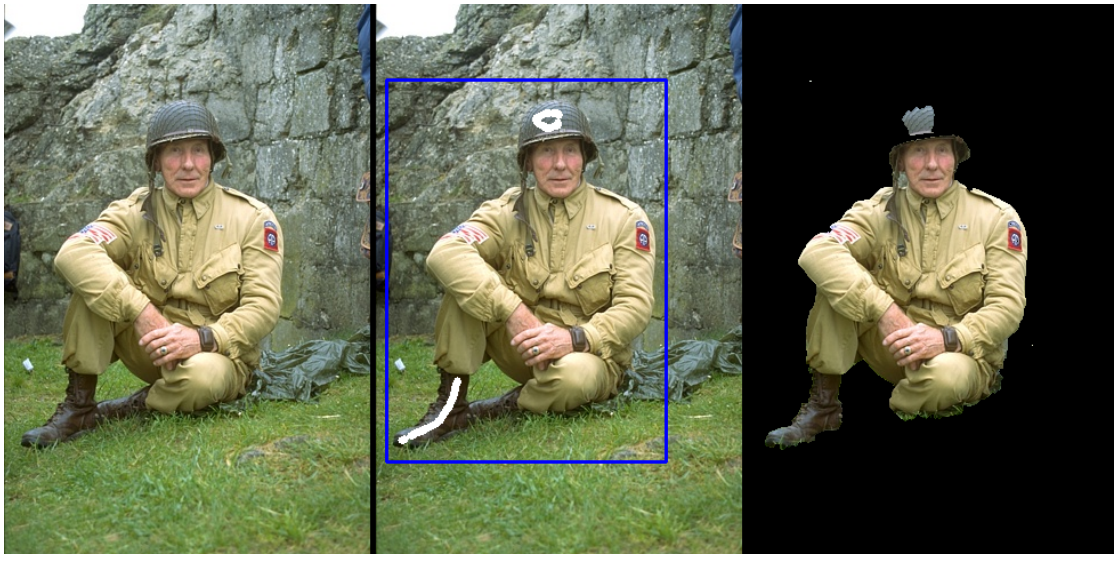


FIGURE 3.7: Harder extraction example with more user imput

There are even harder cases where simple user inputs are not enough. As in the case in Figure 3.6 we might assume its just like the case in Figure 3.4, where we might be able to add the soldier's boots and cap into the selection with a little more labelling from the user. In Figure 3.7 we see that the boot works as we might

have though, but it does not work as well for the cap. This is most likely due to the grid-like texture pattern on the cap. When the model is updating, the newly labelled parts extend to contours within the texture rather than encompassing the entire cap. So to compensate for cases such as these we propose to use superpixels to easily add in those regions.

Chapter 4

Other Approaches Investigated

Before finding a solution for easier refinement, other avenues of trying to get a better extraction was explored. Though they were not successful, they lead to finding the current solution.

4.1 Segmentation

The goal in making an improvement over GrabCut is to take regions that are difficult to easily add or remove and have them grouped together. Whether they be noisy regions or difficult textures, if they are consolidated into one or more segments, it would be a simpler task on the user's part to manipulate them. The first step in getting these segments is to find a segmentation method that has the grouping properties we want.

4.1.1 Bag of Words Clustering

To look for something that would give segments that would keep relavent portions together and make sure to get edges of objects we first turned to a feature normally used for image classification and object detection, SIFT descriptors [10]. SIFT stands for Scale-Invariant Feature transform. This feature looks for keypoints, or points of interest, in the image that gave distinctive information about pieces of the image. SIFT features are commonly used in image recognition tasks, so it was believed that the features would be able to group similar regions together.

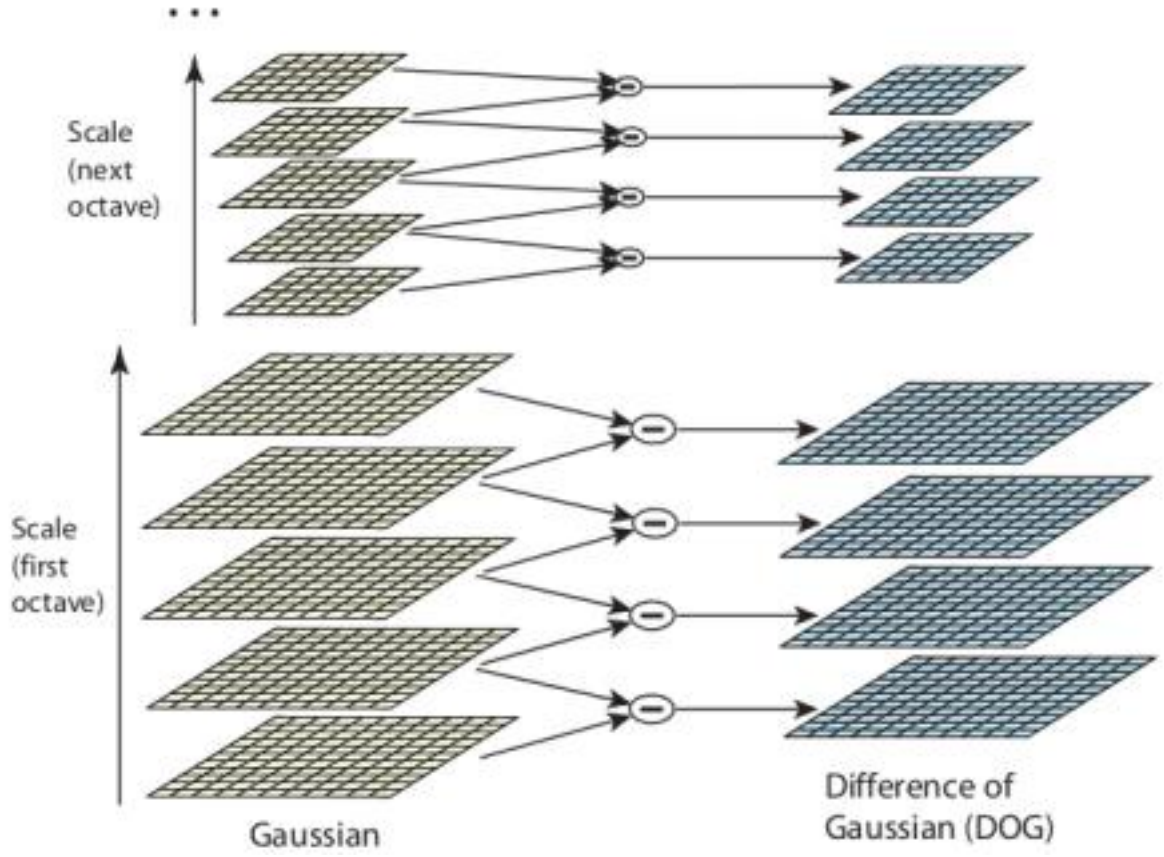


FIGURE 4.1: Difference of Gaussians[4]

To find potential keypoints, the image is blurred at 4 different octaves, or sizes, with a Gaussian window at 5 different scales of σ, $(\sigma, k\sigma, \ldots, 4k\sigma)$. This makes each pixel a weighed average of its neighbors. The weights are normally distributed around center of a 2D matrix. The values of σ and k are set as 1.6 and $\sqrt{2}$ respectively which were found to be optimal in [10]. At each octave, after application of the Gaussian filters, differences of paired filtered images of consecutive σ values are taken to produce 4 Difference of Gaussians (DoG) at each scale, as shown in Figure 4.1. The potential keypoints from these DoG come from finding local extrema in space and scale. A point is determined as a local extrema when compared to its 8 neighboring pixels, and 9 pixels from each the scale before or after it.

After getting potential keypoints, a Taylor series expansion of scale gives a more accurate location of the extrema, and if the intensity value of the extrema is less than the threshold (found to be optimally 0.03 [10]), then it is rejected for being too low contrast. DoG is known to give a very strong response at edges, so keypoints at the edges are discarded as deteremined by a 2x2 Hessian matrix that takes a second derivative as a measure of relative contrast. For the keypoints

that remain, they are made rotation invariant by assigning an orientation to each keypoint. These are calculated by looking at a gradient magnitude and a Gaussian-weighted circular window in an area around the keypoint's location. From this an orientation histogram is made and the strongest responses determine the direction of the keypoint.

For the actual keypoint descriptor, a 16x16 neighborhood around the keypoint is taken and split into 16 4x4 sub-blocks. Each sub-block has an 8 bin orientation histogram giving a vector of 128 bin values. This vector is what comprises the SIFT keypoint descriptor.

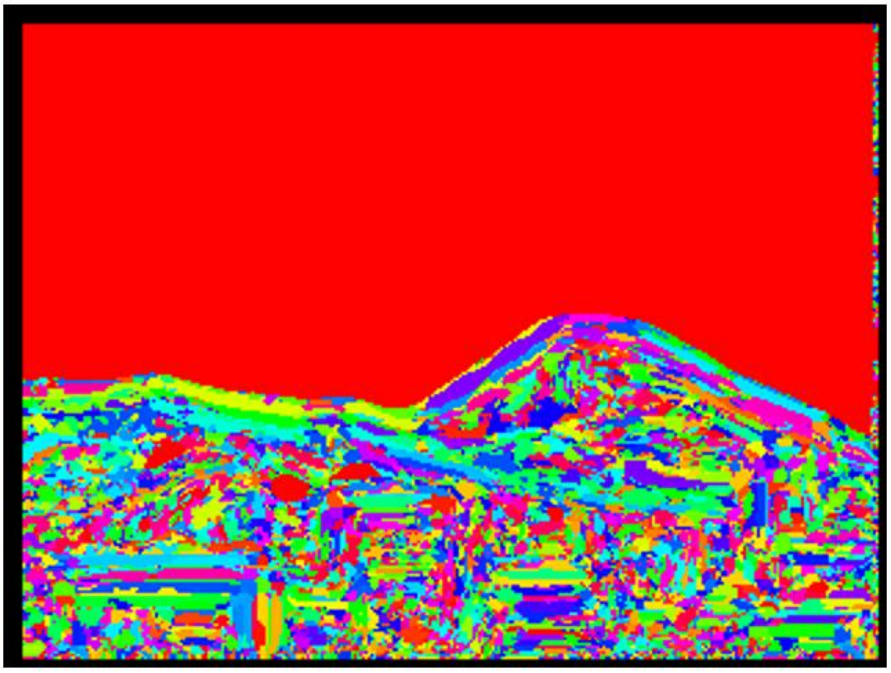
To use the descriptor to try and segment the image, we used a Bag of Words approach to grouping descriptors. K-means clustering was used, with $K = 1000$, to group descriptors into "bags". Each is mapped to a color so that the bag of each point in the image can be displayed. This approach is shown in Figure 4.2. The only portion from this segmentation is the sky. Though the contours of some of the objects are visible, it is very evident that using the sift descriptor is not the best approach to segmenting the image. The next features to be explored for clustering were color and position which had already been done as SLIC superpixels.

4.1.2 Simple Linear Iterative Clustering

After trying out SIFT descriptors for clustering, we moved on to try to use color as the clustering feature instead and found that it was already implemented as Simple Linear Iterative Clustering (SLIC) [5]. SLIC is superpixel algorithm that is meant to group pixels meaningfully so that it can replace a pixel grid structure. They are often used in preprocessing steps to reduce the complexity of further image precessing steps and used to extract features, both of which we hope to use the superpixels for.

The SLIC method is built upon a K-means clustering approach, much like we were trying to accomplish using SIFT descriptors. More details on K-means clustering can be found in Chapter 2. To get superpixels by using K-means, the features used for the clustering is color information and the position of the pixel in the image. The user needs to provide the number of desired superpixels. This number of cluster centers are equally distributed spacially throughout the image. Then in a 3x3 pixel patch in the neighborhood around the cluster centers the lowest

(a) Original Image



(b) Clustered Image

FIGURE 4.2: SIFT Bag Of Words Clustering

gradient position is found and chosen as the new cluster center. This is to prevent the superpixel from being centered on an edge or seeding with a noisy pixel.

Then pixels are labelled part of the cluster center it is closest to in its feature space. Unlike a standard K-means clustering, the search for the nearest cluster center is limited to four times the area of the average superpixel around the point (twice the length and with of the average superpixel). Limiting the search space significantly improves the speed of the algorithm. The distance measure to determine how close a point is to a cluster needs to be a combination of the color and position information of the pixels. The color is is represented in CIELAB color space. In CIELAB representation RGB becomes Lab, where L is lightness, a is the red/green attribute, and b is the blue/yellow attribute. The CIELAB space is often used because the Euclidean distance between colors approximates a perceptually uniform measure for color distances. Simply using the raw x and y pixel position would inconsistently weigh the position against color information based on the size of the superpixels and image. The algorithm is summarized as:

Algorithm 1 Efficient superpixel segmentation

1: Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid steps S.
2: Perturb cluster centers in an $n \times n$ neighborhood, to the lowest gradient position.
3: **repeat**
4: **for** each cluster center C_k **do**
5: Assign the best matching pixels from a $2S \times 2S$ square neighborhood around the cluster center according to the distance measure (Eq. 1).
6: **end for**
7: Compute new cluster centers and residual error E {$L1$ distance between previous centers and recomputed centers}
8: **until** $E \leq$ threshold
9: Enforce connectivity.

where l,a, and b are the colors in CIELAB color space and x and y is the position within the image.

The distance measure in SLIC normalizes the color and spatial by their maximum within a cluster. The maximum spatial distance corresponds to the superpixel size. The maximum color distance is left as a constant that can be set by the user to control the weights between the color and spatial information. The higher this normalization value is set, the more grid-like the superpixels become. The lower the value is, the closer it adheres to edges within the image. With this

distance measure, the algorithm iterates by adjusting the cluster centers based on the means of the color and spatial features within the cluster. A new cluster is then form based on the new cluster centers. Normally the algorithm would iterate until change in the cluster centers went below a certain threshold, but in most cases 10 iterations suffice.

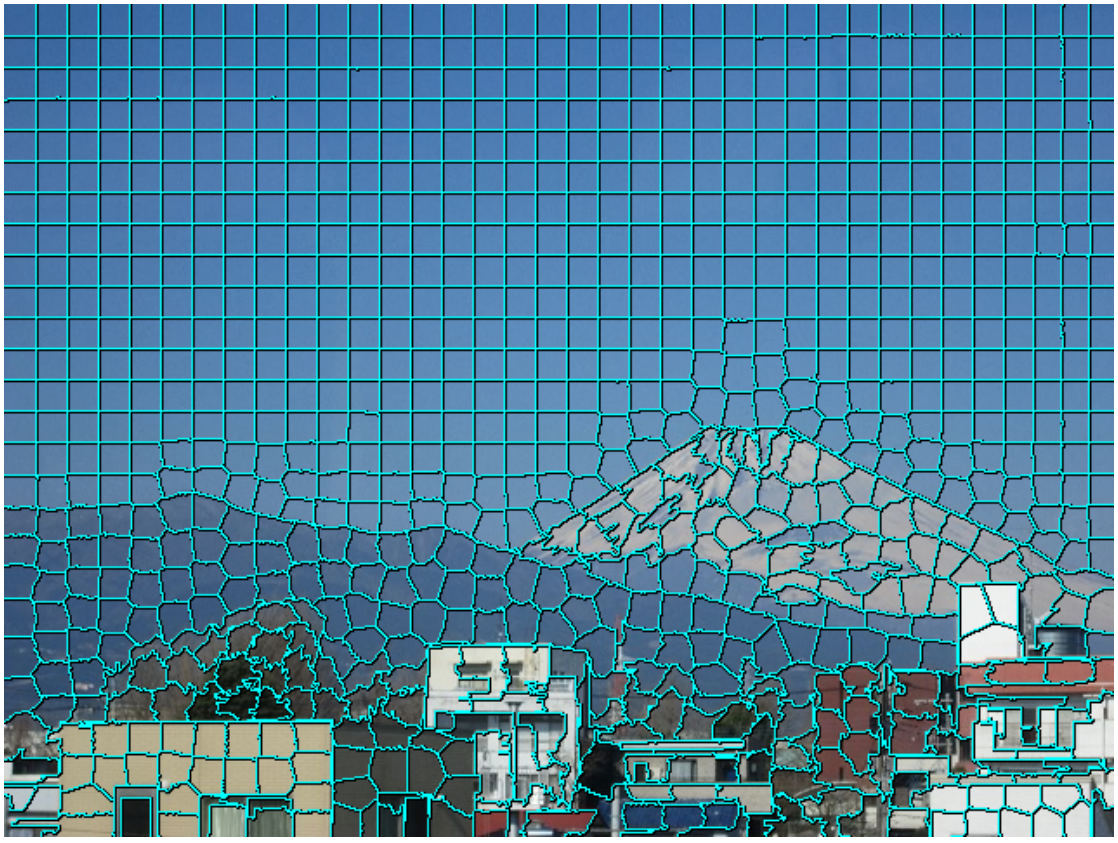


FIGURE 4.3: SLIC Superpixels

Performing superpixel segmentation on the image from the previous section gives us the result in Figure 4.3. The the relatively monochromatic sky, the color information does not provide much distinction, so the spatial component of the features force the superpixels into a grid shape. In the busier portions of the image the superpixels generally do a good job of adhering to the contours of objects. Other methods for creating superpixels exist, but the SLIC superpixels were used because they were the fastest at runtime and had the best adherence to boundaries. We went on to use these superpixels to perform classification on the images to label superpixels as either foreground or background.

4.2 Classification of Superpixels

There are a few challenges in trying to successfully label a foreground region. One is that forming superpixels decimates the number of data points that can be used to learn a model for the image. Using pixels alone gives in the neighborhood of hundreds of thousands of points, but after superpixel segmentation, we are down to hundreds when using enough superpixels to . Another is that when given a bounding box for an object, there is a good set of background data from outside the bounding box, but within the bounding box there are both background and foreground segments.

A second difficulty is that the training sets are contaminated. We are trying to perform binary classification when the samples labelled as foreground for training are both foreground and background samples. Superpixels outside the bounding box are labelled as background and the ones inside labelled as foreground. For the superpixels that were on both sides of the bounding box, it was only considered to be within the bounding box if more than two-thirds of the pixels comprising the superpixels were within the bounding box. Two-thirds was the ratio chosen because it was unlikely that a superpixel that was more than halfway outside the bounding box was part of the foreground. The features extracted from the superpixels was the RGB mean and standard deviation for each color channel. A secondary feature set was a 10 bin color histogram for each color channel.

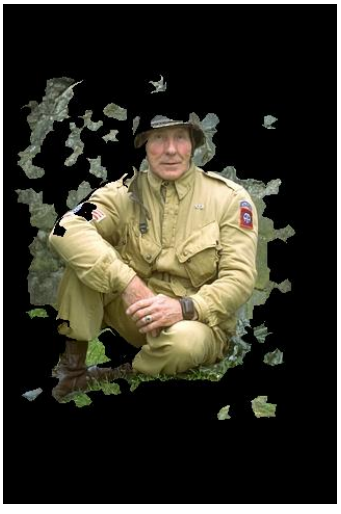
It was not possible to create Gaussian Mixture Models like the ones used in the GrabCut algorithm because there were too few observations to build more than a two component GMM, and in some image cases there was not enough observations even for that. Unable to use a GMM for classification, we turned to common binary classifiers such as Fishers Linear Discrminant (FLD) and Support Vector Machines (SVM). These classifiers were trained with the superpixels outside and inside the bounding box as described and then a binary classification was performed on the superpixels within the bounding box. The outcomes of these classifications are compiled with the results of the current solution in Chapter 6. Generally the accuracy was not terrible, it was close to that of GrabCut. The issue that made it unfeasible was the near 50% false positive rate. This meant half the background pixels within the bounding box remained falsely labelled as foreground.

The output of some of the images from the classification is shown in Figure 4.4. The same images were used as examples for GrabCut in Chapter 3 so a direct

comparison can be made. One of the clearly evident issues is that there is nothing preventing a stray segment not connected to the rest of the extraction from being included in the foreground. In each of the images in Figure 4.4 there is at least one background segment extracted that is not adjacent to and of the segments within the majority of the extraction. Some logical checks could possibly address these segments. There is also the issue of some of the true foreground segments not being included in the extraction despite being surrounded by foreground segments. Furthermore, the high false positive rate would only increase the amount of refinement users would have to do to get a good segmentation rather than simplifying work for them.

(a) Image

(b) Bounding Box



(c) Ground Truth

FIGURE 4.4: Classification on Superpixels

We looked into a few more avenues of classification, but it was evident that they behaved similarly and creating the models for that background and forground from just the superpixels was not enough. To deal with the issues of having too few samples to learn good models, the logical solution would be to just get more samples. This could be done by using the pixels composing the superpixel as samples for learning the foreground and background models. After that a minimum energy cut could be used to try and have connectivity with the segments. With these changes, the algorithm essentially becomes GrabCut. So instead of implementing our own GrabCut like algorithm, the actual GrabCut algorithm is used to the learning and initial extraction and superpixels overlays the algorithm to aid in performing refinements.

Chapter 5

Superpixel Selection Tool Design

Performing foreground extraction with superpixels is divided into two tasks with different priorities. The first task is to use superpixels to perform foreground extraction of an object only using a bounding box as input. These bounding boxes are the same for each extraction method so that extraction results can be directly compared. This is the basic extraction made before any further user input to make sure overlaying superpixels on GrabCut does not significantly degrade the results. The second task is to perform the extraction task and refinements with superpixels and demonstrate their relative ease of use compared to GrabCut.

5.1 Base Comparison

The basic foreground procedure, the code of which can be found in Appendix A, first requires a set of images with objects to extract and corresponding bounding boxes. The GrabCut algorithm is performed to create an initial mask which is then fed back into the GrabCut algorithm to iterate 10 times to converge the segmentation. This is extraction result that is the baseline comparison for our own methods.

The image is then segmented into superpixels with the SLIC algorithm with the desired number of superpixels set at 1000. Around a thousand superpixels ensures for most images that for the most part the superpixels will keep separate the regions meant for different labels without having to enforce connectivity. The output of the superpixel algorithm labels each pixel with a value denoting which

superpixel it is assigned to. Applying the GrabCut mask onto the superpixel representation of the image informs us as to which superpixels contain pixels labelled as foreground by GrabCut. If more than 75% of a superpixel's pixels are within the GrabCut mask, then it is included in the foreground segmentation. The set of superpixels that pass this criteria is then used as the new mask for foreground and is used to extract the object for evaluation.

5.2 User Selection Tool

The selection tool builds off of the OpenCV implementation of the GrabCut algorithm [11]. The GrabCut tool implemention code is shown in Appendix A. With the GrabCut extraction tool, the procedure is to put a bounding box around the object, let the algorithm iterate a few times and then make any refinements that are needed. Examples of this done can be found in Chapter 3.

The selection tool operates very much like the GrabCut tool. The user defines a bounding box around the desired object to extract and lets the algorithm iterate. In the displayed extration, the superpixels overlay the GrabCut mask and the superpixels with over 75% of its pixels in the mask is extracted. To refine the extraction there is a separate window that shows the superpixel representation of the image, so the user can pick and choose which segments to keep or remove. The interactions made are kept on the original image to be able to compare the effort needed to get the desired extraction compared to GrabCut.

Chapter 6

Results and Analysis

To test the automatic segmentation and user selection tools we used a subset of 50 images from the Berkeley Image database [12]. These images have a bounding box defined for desired object to extract and a ground truth mask. An example from the dataset is shown in Figure 6.1.

(a) Image (b) Bounding Box (c) Ground Truth

FIGURE 6.1: Dataset

6.1 Automatic Segmentation

To evaluate how well a method extracts the foreground, two scores are given to the extraction. The first is accuracy which indicates what percent of ground truth mask is part of the automatic extraction. The second score is the false positive what percent of the background within the bounding box was extracted as foreground. The whole background is not used because nothing outside the bounding box would be extracted as foreground, so including all of background in the calculation does not make sense. If all background was included, it would be impossible to get a 100% false positive rate.

Method	Number of Superpixels	Features	Accuracy	False Positive
GrabCut	N/A	Color GMM	0.963	0.188
Cls S.pix	500	Mean Color and STD	0.933	0.496
Cls S.pix	1000	Mean Color and STD	0.956	0.518
Cls S.pix	1000	Color Histogram	0.828	0.740
GC and SP	1000	Color GMM (C True)	0.957	0.206
GC and SP	1000	Color GMM (C False)	0.967	0.224

TABLE 6.1: Automatic Segmentation Results: Cls S.pix = Classify Superpixels, GC and SP = GrabCut and Superpixels

For the accuracy on a single image, the number of pixels in the intersection between the ground truth foreground mask and the foreground mask generated by the automatic segmentation is divided by the total number of pixels in the ground truth mask. For the false positive rate, the number of pixels extracted as foreground that are outside the ground truth mask is divided by the number of background pixels within the bounding box. The accuracy and false positive rates are averaged over the images to get an overall accuray and false positive scores for an extraction method.

From the dataset of 50 images, one image was removed because the bounding box encompassed the entire image so there were no background samples for certain methods to learn from. The automatic segmentation results are shown in Table 6.1.

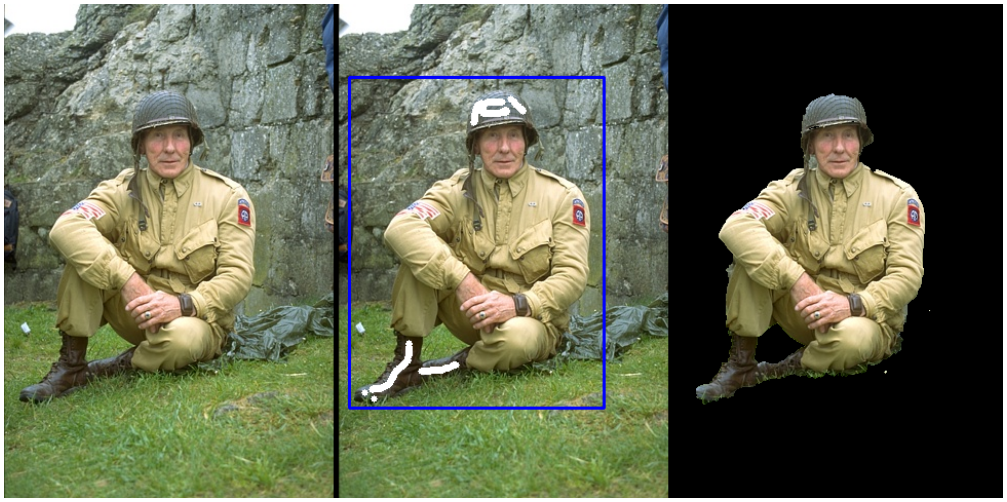
As mentioned in Chapter 4, using just the features from the superpixels was not enough to separate out the data. In Table 6.1 the only classifier result displayed is for the Fisher Linear Descriminant, but the other classifiers performed comparably to it. In the GrabCut and Superpixel cases whether C is true of false indicates whether connectivity was enforced or not. With connectivity, all points within a superpixel should be a contiguous group. Without connectiviy enforced, a superpixel is allowed to be disjoint regions.

Based on the results from the results of the automatic segmentation, GrabCut+Superpixels performs close enough to GrabCut alone that it would not be a hinderance to use

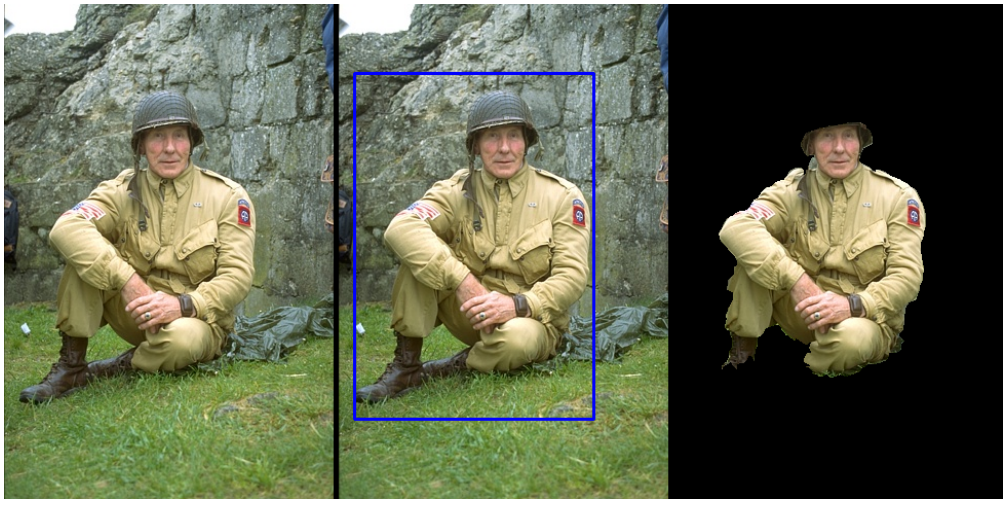
it as an initial step for the user selection tool.

6.2 User Selection Tool

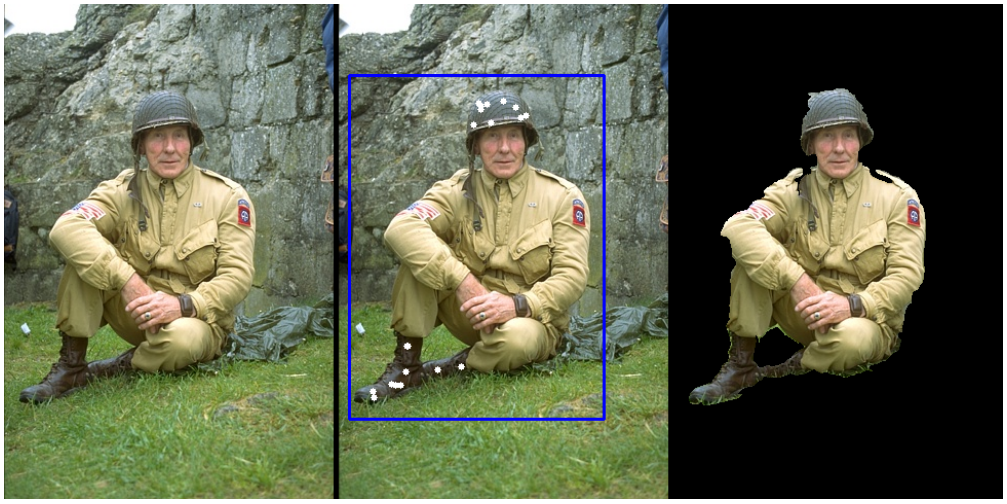
Unfortunately, unlike the automatic segmentation, evaluating the selection tool is more subjective. What can be compared is the relative number of actions needed to refine a selection in both the procedures. for this, we will be using the example previously seen in Figure 3.7. Figure 6.2 Shows what it would take a user to do to include the missing boots and cap of the soldier in the segmentation. With the GrabCut procedure, after each additional group of pixels are added to the model, the user needs to iterate through the updating process to see if the refine given was enough. While there is some guesswork involved for getting the regions by Grabcut, if using the superpixels, the user can see how much it would take to add the desired region by referencing the superpixel boundaries as in Figure 6.3.



(a) GrabCut Refinement



(b) CrabCut and Superpixel Automatic



(c) Superpixel Refinement

FIGURE 6.2: Comparing Refinement Methods

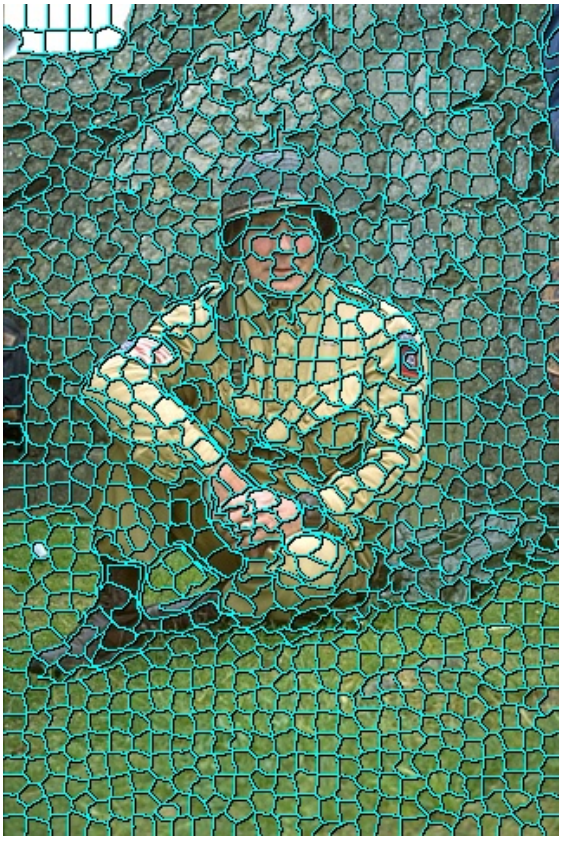


FIGURE 6.3: Reference Boundaries

Chapter 7

Conclusions and Recommendations

Adding superpixels on top of the GrabCut process did not degrade the segmentation results significantly, but did provide a convenient procedure to refine the segmentation. Having a superpixel segmentation on hand allowed a way to include portions of the image in the foreground without the previous porcedure of guessing at how much to add to the foreground model and hoping it was enough to learn the new region. There is a few downsides in speed because the superpixel segmentation takes a few seconds to perform, and it is inconvenient to edit the superpixel parameters because of the time it would take to reprocess. For a better metric for the ease of use, it would have been prudent to have multiple users to evaluate the user experience of the tool.

There were a few area of inquiry left to explore to further see the possibilities in making segmentation improvements using superpixels. One is that the pixels contained within the superpixels could be used within the iterations GrabCut uses to automatically refine the segmentation. This would also mean that when you add a superpixel to the foreground you add all the pixels within the superpixel into the foreground color model. Another avenue left unexplored is using texture features from the superpixels for classification. Though using pixels for learning models provides a lot of data points to draw from, more complex features cannot be drawn from pixels. On the other hand, it is possible to get texture information from a superpixel patch and could possibly aid in classifcation. The difficulty arises from extracting features from irregularly shaped superpixels. Usually to get

texture features a filter is used or a patch of the image is compared to a template. This might be possible to do if a rectangular portion of the superpixel was used for the feature calculation.

There is a lot more that could have possibly been done to fully explore potential for using superpixels in foreground extraction and there is still work to be done to have a more concrete evaluation of the GrabCut+Superpixel procedure. For dissemination, the GrabCut+Superpixel tool is planned to be submitted as a module for Gimp.

Appendix A

Foreground Extraction Code

A.1 GrabCut

GrabCut implementation from OpenCV [11].

```
#!/usr/bin/env python
'''
==============================================================================
Interactive Image Segmentation using GrabCut algorithm.

This sample shows interactive image segmentation using grabcut algorithm.

USAGE:
    python grabcut.py <filename>

README FIRST:
    Two windows will show up, one for input and one for output.

    At first, in input window, draw a rectangle around the object using
mouse right button. Then press 'n' to segment the object (once or a few times)
For any finer touch-ups, you can press any of the keys below and draw lines on
the areas you want. Then again press 'n' for updating the output.

Key '0' - To select areas of sure background
Key '1' - To select areas of sure foreground
Key '2' - To select areas of probable background
Key '3' - To select areas of probable foreground

Key 'n' - To update the segmentation
Key 'r' - To reset the setup
Key 's' - To save the results
==============================================================================
'''
```

```
import numpy as np
import cv2
import sys

BLUE = [255,0,0]        # rectangle color
RED = [0,0,255]         # PR BG
GREEN = [0,255,0]       # PR FG
BLACK = [0,0,0]         # sure BG
WHITE = [255,255,255]   # sure FG

DRAW_BG = {'color' : BLACK, 'val' : 0}
DRAW_FG = {'color' : WHITE, 'val' : 1}
DRAW_PR_FG = {'color' : GREEN, 'val' : 3}
DRAW_PR_BG = {'color' : RED, 'val' : 2}

# setting up flags
rect = (0,0,1,1)
drawing = False         # flag for drawing curves
rectangle = False       # flag for drawing rect
rect_over = False       # flag to check if rect drawn
rect_or_mask = 100      # flag for selecting rect or mask mode
value = DRAW_FG         # drawing initialized to FG
thickness = 3           # brush thickness

def onmouse(event,x,y,flags,param):
    global img,img2,drawing,value,mask,rectangle,rect,rect_or_mask,ix,iy,rect_over

    # Draw Rectangle
    if event == cv2.EVENT_RBUTTONDOWN:
        rectangle = True
        ix,iy = x,y

    elif event == cv2.EVENT_MOUSEMOVE:
        if rectangle == True:
            img = img2.copy()
            cv2.rectangle(img,(ix,iy),(x,y),BLUE,2)
            rect = (min(ix,x),min(iy,y),abs(ix-x),abs(iy-y))
            rect_or_mask = 0

    elif event == cv2.EVENT_RBUTTONUP:
        rectangle = False
        rect_over = True
        cv2.rectangle(img,(ix,iy),(x,y),BLUE,2)
        rect = (min(ix,x),min(iy,y),abs(ix-x),abs(iy-y))
        rect_or_mask = 0
        print " Now press the key 'n' a few times until no further change \n"

    # draw touchup curves

    if event == cv2.EVENT_LBUTTONDOWN:
        if rect_over == False:
            print "first draw rectangle \n"
        else:
            drawing = True
            cv2.circle(img,(x,y),thickness,value['color'],-1)
```

```
            cv2.circle(mask,(x,y),thickness,value['val'],-1)

    elif event == cv2.EVENT_MOUSEMOVE:
        if drawing == True:
            cv2.circle(img,(x,y),thickness,value['color'],-1)
            cv2.circle(mask,(x,y),thickness,value['val'],-1)

    elif event == cv2.EVENT_LBUTTONUP:
        if drawing == True:
            drawing = False
            cv2.circle(img,(x,y),thickness,value['color'],-1)
            cv2.circle(mask,(x,y),thickness,value['val'],-1)

# print documentation
print __doc__

# Loading images
if len(sys.argv) == 2:
    filename = sys.argv[1] # for drawing purposes
else:
    print "No input image given, so loading default image, lena.jpg \n"
    print "Correct Usage: python grabcut.py <filename> \n"
    filename = 'C:/Users/arahman/Pictures/fuji.jpg'

img = cv2.imread(filename)
img2 = img.copy()                               # a copy of original image
mask = np.zeros(img.shape[:2],dtype = np.uint8) # mask initialized to PR_BG
output = np.zeros(img.shape,np.uint8)           # output image to be shown

# input and output windows
cv2.namedWindow('output')
cv2.namedWindow('input')
cv2.setMouseCallback('input',onmouse)
cv2.moveWindow('input',img.shape[1]+10,90)

print " Instructions: \n"
print " Draw a rectangle around the object using right mouse button \n"
# cv2.imwrite('M:/CIED_CDWS/segmenting/badroad.bmp',img)
while(1):

    cv2.imshow('output',output)
    cv2.imshow('input',img)
    k = 0xFF & cv2.waitKey(1)

    # key bindings
    if k == 27:         # esc to exit
        break
    elif k == ord('0'): # BG drawing
        print " mark background regions with left mouse button \n"
        value = DRAW_BG
    elif k == ord('1'): # FG drawing
        print " mark foreground regions with left mouse button \n"
        value = DRAW_FG
    elif k == ord('2'): # PR_BG drawing
        value = DRAW_PR_BG
```

```
    elif k == ord('3'): # PR_FG drawing
        value = DRAW_PR_FG
    elif k == ord('s'): # save image
        bar = np.zeros((img.shape[0],5,3),np.uint8)
        res = np.hstack((img2,bar,img,bar,output))
        cv2.imwrite('grabcut_output.png',res)
        print " Result saved as image \n"
    elif k == ord('r'): # reset everything
        print "resetting \n"
        rect = (0,0,1,1)
        drawing = False
        rectangle = False
        rect_or_mask = 100
        rect_over = False
        value = DRAW_FG
        img = img2.copy()
        mask = np.zeros(img.shape[:2],dtype = np.uint8) # mask initialized to PR_BG
        output = np.zeros(img.shape,np.uint8)           # output image to be shown
    elif k == ord('n'): # segment the image
        print """ For finer touchups, mark foreground and background after pressing keys 0-3
        and again press 'n' \n"""
        if (rect_or_mask == 0):         # grabcut with rect
            bgdmodel = np.zeros((1,65),np.float64)
            fgdmodel = np.zeros((1,65),np.float64)
            cv2.grabCut(img2,mask,rect,bgdmodel,fgdmodel,1,cv2.GC_INIT_WITH_RECT)
            rect_or_mask = 1
        elif rect_or_mask == 1:         # grabcut with mask
            bgdmodel = np.zeros((1,65),np.float64)
            fgdmodel = np.zeros((1,65),np.float64)
            cv2.grabCut(img2,mask,rect,bgdmodel,fgdmodel,1,cv2.GC_INIT_WITH_MASK)

    mask2 = np.where((mask==1) + (mask==3),255,0).astype('uint8')
    output = cv2.bitwise_and(img2,img2,mask=mask2)
#     cv2.imwrite('M:/CIED_CDWS/segmenting/badroadmask.bmp',output)


cv2.destroyAllWindows()
```

A.2 Superpixel Refinement

Implemention of refinement of GrabCut using superpixels.

```
# Abrar Rahman: Superpixel Refinemment

import numpy as np
import cv2
from skimage import segmentation
import os
import sys
```

```
BLUE = [255,0,0]        # rectangle color
RED = [0,0,255]         # PR BG
GREEN = [0,255,0]       # PR FG
BLACK = [0,0,0]         # sure BG
WHITE = [255,255,255]   # sure FG

DRAW_BG = {'color' : BLACK, 'val' : 0}
DRAW_FG = {'color' : WHITE, 'val' : 1}
DRAW_PR_FG = {'color' : GREEN, 'val' : 3}
DRAW_PR_BG = {'color' : RED, 'val' : 2}

# setting up flags
rect = (0,0,1,1)
drawing = False         # flag for drawing curves
rectangle = False       # flag for drawing rect
rect_over = False       # flag to check if rect drawn
rect_or_mask = 100      # flag for selecting rect or mask mode
value = DRAW_FG         # drawing initialized to FG
thickness = 3           # brush thickness

include = [] # manually added segments
exclude = [] # manually removed segments
threshold = 0.75 # overlap of segment with grabcut needed to be included

def onmouse(event,x,y,flags,param):
    global img,img2,drawing,value,mask,rectangle,rect,rect_or_mask,ix,iy,rect_over,outline,inclu

    # Draw Rectangle
    if event == cv2.EVENT_RBUTTONDOWN:
        rectangle = True
        ix,iy = x,y

    elif event == cv2.EVENT_MOUSEMOVE:
        if rectangle == True:
            img = img2.copy()
            cv2.rectangle(img,(ix,iy),(x,y),BLUE,2)
            rect = (min(ix,x),min(iy,y),abs(ix-x),abs(iy-y))
            rect_or_mask = 0

    elif event == cv2.EVENT_RBUTTONUP:
        rectangle = False
        rect_over = True
        cv2.rectangle(img,(ix,iy),(x,y),BLUE,2)
        rect = (min(ix,x),min(iy,y),abs(ix-x),abs(iy-y))
        rect_or_mask = 0
        print " Now press the key 'n' a few times until no further change \n"

    # draw touchup curves

    if event == cv2.EVENT_LBUTTONDOWN:
        if rect_over == False:
            print "first draw rectangle \n"
        else:
            drawing = True
```

```
            cv2.circle(img,(x,y),thickness,value['color'],-1)
            cv2.circle(mask,(x,y),thickness,value['val'],-1)
            modify(outline[y,x],value['val'])

    elif event == cv2.EVENT_MOUSEMOVE:
        if drawing == True:
            cv2.circle(img,(x,y),thickness,value['color'],-1)
            cv2.circle(mask,(x,y),thickness,value['val'],-1)
            modify(outline[y,x],value['val'])

    elif event == cv2.EVENT_LBUTTONUP:
        if drawing == True:
            drawing = False
            cv2.circle(img,(x,y),thickness,value['color'],-1)
            cv2.circle(mask,(x,y),thickness,value['val'],-1)
            modify(outline[y,x],value['val'])


def modify(v,mode):
    if mode:
        if v not in include:
            include.append(v)
        if v in exclude:
            exclude.remove(v)
    else:
        if v in include:
            include.remove(v)
        if v not in exclude:
            exclude.append(v)

if len(sys.argv) == 3:
    filename = sys.argv[1] # for drawing purposes
    seg = int(sys.argv[2])
else:
    print "No input image given, so loading default image, lena.jpg \n"
    print "Correct Usage: python supercut.py <filename> \n"
    filename = 'C:/Users/Rafi/Dropbox/data/images/376043.jpg'
    seg = 1000

img = cv2.imread(filename)
img2 = img.copy()                               # a copy of original image
mask = np.zeros(img.shape[:2],dtype = np.uint8) # mask initialized to PR_BG
stain = np.zeros(img.shape,dtype = np.uint8) # mask initialized to PR_BG
output = np.zeros(img.shape,np.uint8) # output image to be shown

# segmenting
outline = segmentation.slic(img2,sigma=1,n_segments=seg,enforce_connectivity=False,compactness=1
segments = np.unique(outline)

boundaries = segmentation.mark_boundaries(img2,outline)
cv2.namedWindow('output')
cv2.namedWindow('input')
cv2.namedWindow('boundaries')
cv2.setMouseCallback('input',onmouse)
cv2.setMouseCallback('boundaries',onmouse)
```

```
while(1):

    cv2.imshow('output',output)
    cv2.imshow('input',img)
    cv2.imshow('boundaries',boundaries)
    k = 0xFF & cv2.waitKey(1)

    # key bindings
    if k == 27:         # esc to exit
        break
    elif k == ord('0'): # BG drawing, not used currently
        print " mark background regions with left mouse button \n"
        value = DRAW_BG
    elif k == ord('1'): # FG drawing
        print " mark foreground regions with left mouse button \n"
        value = DRAW_FG
    elif k == ord('s'): # save image
        bar = np.zeros((img.shape[0],5,3),np.uint8)
        res = np.hstack((img2,bar,img,bar,output))
        cv2.imwrite('supercut_output.png',res)
        print " Result saved as image \n"
    elif k == ord('r'): # reset everything
        print "resetting \n"
        rect = (0,0,1,1)
        drawing = False
        rectangle = False
        rect_or_mask = 100
        rect_over = False
        value = DRAW_FG
        img = img2.copy()
        mask = np.zeros(img.shape[:2],dtype = np.uint8) # mask initialized to PR_BG
        output = np.zeros(img.shape,np.uint8)           # output image to be shown
    elif k == ord('n'): # segment the image
        print """ For finer touchups, mark foreground and background after pressing keys 0-3
        and again press 'n' \n"""
        if (rect_or_mask == 0):         # grabcut with rect
            bgdmodel = np.zeros((1,65),np.float64)
            fgdmodel = np.zeros((1,65),np.float64)
            cv2.grabCut(img2,mask,rect,bgdmodel,fgdmodel,1,cv2.GC_INIT_WITH_RECT)
            rect_or_mask = 1
        elif rect_or_mask == 1:         # grabcut with mask
            bgdmodel = np.zeros((1,65),np.float64)
            fgdmodel = np.zeros((1,65),np.float64)
            cv2.grabCut(img2,mask,rect,bgdmodel,fgdmodel,1,cv2.GC_INIT_WITH_MASK)

    # sections from grabcut
    grab_mask = np.where((mask==2)|(mask==0),0,1).astype('uint8')
    regions = outline*grab_mask
    segmented = np.unique(regions)
    segmented = segmented[1:len(segmented)]

    # manual segments
    for i in include:
        if i not in segmented:
```

```
            segmented = np.append(segmented,i)

    pxtotal = np.bincount(outline.flatten())
    pxseg = np.bincount(regions.flatten(),minlength=len(segments))

    seg_mask = np.zeros(img.shape[:2],np.uint8)
    label = (pxseg[segmented]/pxtotal[segmented].astype(float))<.75

    for i in include:
        label[segmented==i] = 0

    for i in exclude:
        label[segmented==i] = 1

    for j in range(0,len(label)):
        if label[j] == 0:
            temp = (outline == segmented[j])
            seg_mask = seg_mask+temp

    fin_mask = seg_mask>0
    mask2 = np.where((fin_mask==1),255,0).astype('uint8')
    output = cv2.bitwise_and(img2,img2,mask=mask2)

cv2.destroyAllWindows()
```

A.3 Automatic Foreground Extraction

Performing foreground extraction with a given bounding box.

```
import cv2
import os
import numpy as np
from skimage import segmentation

base_dir = "C:/Users/Rafi/Dropbox/data/"

img_dir = base_dir + "images/"

box_dir = base_dir + "boxes/"

img_files = os.listdir(img_dir)

box_files = os.listdir(box_dir)

n = 1000 # approximate number of superpixels

for i in range(0,len(img_files)):
# for i in range(0,2):
    img = cv2.imread(img_dir + img_files[i])
```

```
    mask = np.zeros(img.shape[:2],np.uint8)
    bgdModel = np.zeros((1,65),np.float64)
    fgdModel = np.zeros((1,65),np.float64)

    # read bounding box data
    f = open(box_dir + box_files[i])
    points = f.read()
    f.close()
    points = points.split()
    rect0 = (int(float(points[0])),int(float(points[1])),int(float(points[2])),int(float(points[
    rect = (rect0[0],rect0[1],rect0[2]-rect0[0],rect0[3]-rect0[1])

    output = np.zeros(img.shape,np.uint8) # output image to be shown
    outline = segmentation.slic(img,n_segments=n,enforce_connectivity=False)
#     outline = segmentation.quickshift(img)

    # perform GrabCut
    cv2.grabCut(img,mask,rect,bgdModel,fgdModel,1,cv2.GC_INIT_WITH_RECT)
    cv2.grabCut(img,mask,rect,bgdModel,fgdModel,10,cv2.GC_INIT_WITH_MASK)

    # use GrabCut mask on superpixels
    grab_mask = np.where((mask==2)|(mask==0),0,1).astype('uint8')
    regions = outline*grab_mask
    segmented = np.unique(regions)
    segmented = segmented[1:len(segmented)]

    pxtotal = np.bincount(outline.flatten())
    pxseg = np.bincount(regions.flatten())

    # determine which superpixels to include
    seg_mask = np.zeros(img.shape[:2],np.uint8)
    label = (pxseg[segmented]/pxtotal[segmented].astype(float))<.75

    for j in range(0,len(label)):
        if label[j] == 0:
            temp = (outline == segmented[j])
            seg_mask = seg_mask+temp

    mask = seg_mask>0
    mask2 = np.where((mask==1),255,0).astype('uint8')
    output = cv2.bitwise_and(img,img,mask=mask2)

    cv2.imwrite(base_dir+str(n)+"/cut2/"+img_files[i][0:-4]+'.jpg',output)
```

A.4 Evaluation Code

Get evaluation score as described in Chapter 6.

```
import cv2
```

```
import os
import numpy as np

base_dir = "C:/Users/Rafi/Dropbox/data/"

img_dir = base_dir + "images/"

segment_dir = base_dir + "segment/"

box_dir = base_dir + "boxes/"

img_files = os.listdir(img_dir)

segment_files = os.listdir(segment_dir)

box_files = os.listdir(box_dir)

back_score = 0
fore_score = 0

for i in range(0,len(img_files)):
    img = cv2.imread(img_dir + img_files[i])
    mask = np.zeros(img.shape[:2],np.uint8)

    bgdModel = np.zeros((1,65),np.float64)
    fgdModel = np.zeros((1,65),np.float64)

    f = open(box_dir + box_files[i])
    points = f.read()
    f.close()
    points = points.split()

    rect0 = (int(float(points[0])),int(float(points[1])),int(float(points[2])),int(float(points[
    rect = (rect0[0],rect0[1],rect0[2]-rect0[0],rect0[3]-rect0[1])

# #     Performing GrabCut
#     cv2.grabCut(img,mask,rect,bgdModel,fgdModel,1,cv2.GC_INIT_WITH_RECT)
#     cv2.grabCut(img,mask,rect,bgdModel,fgdModel,10,cv2.GC_INIT_WITH_MASK)
#
#     mask2 = np.where((mask==2)|(mask==0),0,1).astype('uint8')
#     img = img*mask2[:,:,np.newaxis]
#
#     pt2 = (rect[2]+rect[0],rect[3]+rect[1])
#
#     cv2.rectangle(img, (rect[0],rect[1]), pt2,BLUE,2)
#
#     cv2.imwrite(base_dir + 'grabcut_' +img_files[i],img)


# #     Extract Segmented Image
#     img = cv2.imread(base_dir + 'grab_results/grabcut_' +img_files[i])
#     img = cv2.imread(base_dir + '1000/out_base/' +img_files[i][0:-4]+'.jpg')
    img = cv2.imread(base_dir + '1000/out/' +img_files[i])
```

```
    segment = cv2.imread(segment_dir + segment_files[i])

    bg = 0
    bg0 = 0
    fg = 0
    fg0 = 0

    for j in range(rect0[1],rect0[3]):
        for k in range(rect0[0],rect0[2]):
            if (segment[j][k] == 0).all():
                bg +=1
                if (img[j][k] == 0).all():
                    bg0 += 1
            if ((segment[j][k] == 255) + (segment[j][k] == 128)).all():
                fg += 1
                if (img[j][k] == 0).all():
                    fg0 += 1

    back_score += (1-bg0/float(bg))/len(img_files)
    fore_score += (1-fg0/float(fg))/len(img_files)

print fore_score
print back_score
```

Bibliography

[1] Dingding Liu, Bilge Soran, Gregg Petrie, and Linda Shapiro. A review of computer vision segmentation algorithms, 2012. URL `https://courses.cs.washington.edu/courses/cse576/12sp/notes/remote.pdf`.

[2] Jiangyu Liu, Jian Sun, and Heung-Yeung Shum. Paint selection. *ACM Transactions on Graphics*, 28, August 2009. URL `http://research.microsoft.com/en-us/um/people/jiansun/papers/PaintSelection_SIGGRAPH09.pdf`.

[3] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. Grabcut: Interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, 23:309–314, August 2004. URL `http://research.microsoft.com/pubs/67890/siggraph04-grabcut.pdf`.

[4] Opencv 3.0.0 documentation: Introduction to sift (scale-invariant feature transform), 2014. URL `http://docs.opencv.org/trunk/doc/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html`.

[5] R. Achanta, A Shaji, K. Smith, A Lucchi, P. Fua, and S. Susstrunk. Slic superpixels compared to state-of-the-art superpixel methods. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(11):2274–2282, Nov 2012. ISSN 0162-8828. doi: 10.1109/TPAMI.2012.120.

[6] J.B MacQueen. Some methods for classification and analysis of multivariate observations. In *Processings of 5th Berkeley Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967.

[7] D. Comaniciu and P. Meer. Mean shift analysis and applications. *IEEE International Conference on Computer Vision*, pages 1197–1203, 1999.

[8] Gnu image manipulation program user manual: 2.5. fuzzy selection (magic wand), 2014. URL `http://docs.gimp.org/en/gimp-tool-fuzzy-select.html`.

[9] Gnu image manipulation program user manual: 2.7. intelligent scissors, 2014. URL `http://docs.gimp.org/en/gimp-tool-iscissors.html`.

[10] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004. ISSN 0920-5691. doi: 10.1023/B:VISI.0000029664.99615.94. URL `http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94`.

[11] G. Bradski. Opencv. *Dr. Dobb's Journal of Software Tools*, 2000.

[12] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision*, volume 2, pages 416–423, July 2001.