Image Segmentation and Matting in Realtime on a Mobile Device

Elad Richardson, supervised by Anastasia Dubrovina & Aaron Wetzler

October 29, 2014
Contents

1 List of Figures 3

2 Abstract 4

3 Introduction 5
  3.1 General Project Progress .................................................. 5

4 Theoretical Background & Algorithm 6
  4.1 Segmentation Algorithm .................................................... 6
  4.2 Alpha Matting ................................................................. 7

5 Software Implementation & Further Algorithm Development 8
  5.1 Implementing the algorithm .................................................. 8
    5.1.1 openCV Libraries ....................................................... 8
    5.1.2 Dijkstra Algorithm ................................................. 8
    5.1.3 Bresenham’s line algorithm ........................................ 8
    5.1.4 Weight Calculation .................................................. 9
    5.1.5 Combining the algorithms .......................................... 9
  5.2 Android First Port .......................................................... 10
    5.2.1 Simple User Interface ............................................... 10
    5.2.2 Porting the Algorithm .............................................. 11
  5.3 Alpha Matting Implementation ........................................... 12
    5.3.1 Choosing Alpha Intensity .......................................... 13
  5.4 Multiple Scribbles Implementation ..................................... 14
  5.5 Scaling the Image .......................................................... 14
    5.5.1 Mask Smoothing ...................................................... 14
    5.5.2 Alpha Calculation .................................................... 16
  5.6 Updated Android Port ..................................................... 17
  5.7 Additional Features ........................................................ 18
    5.7.1 Blurring ............................................................... 18
    5.7.2 Porting the additional feature .................................... 19
  5.8 Android Design Makeover ................................................ 21

6 Results 22

7 Further Work 26
  7.1 Content Aware Fill ....................................................... 26
  7.2 Poisson Blending ........................................................... 26
  7.3 FIGTree Alternatives ....................................................... 27

8 Summary 30

9 References 30
1 List of Figures

List of Figures

1 First Results :) ...................................................... 10
2 First Interface ...................................................... 11
3 Some Android Results ........................................... 12
4 Alpha Band .......................................................... 12
5 Original Mask Vs Alpha Mask .................................. 13
6 Different r values ................................................... 13
7 Multiple Scribbles ................................................... 14
8 Scaled Up Result ..................................................... 15
9 Different smoothing intensities ................................ 15
10 Improvement steps ............................................... 16
11 Updated Port UI .................................................. 18
12 Different smoothing intensities ................................ 19
13 Android Blurring UI .............................................. 20
14 Android Makeover ................................................ 21
15 Hand Segmentation ............................................... 22
16 Rat Segmentation ................................................. 23
17 Giraffe Segmentation ............................................. 23
18 Ballon Segmentation .............................................. 23
19 Deer Segmentation ............................................... 24
20 Android Results .................................................. 24
21 Flower Blurring .................................................... 25
22 Monkey Blurring .................................................. 25
23 Inpainting .......................................................... 26
24 Poisson Blending ................................................... 27
25 Rat 8x8x8 Histogram Result .................................. 27
26 Hand 8x8x8 Histogram Result ................................. 28
27 Hand 16x16x16 Histogram With Smoothing ................ 28
28 FIGTree VS pdf Smoothing ..................................... 29
Image Segmentation is a known problem in the world of Image Processing, where we try to partition a given image into several different segments that compose the original image. One use of the segmentation process is the separation of an object of interest from the rest of the image. Many algorithms use some form of user input in order to locate a specific object of interest and segment it out. One optional form of user inputs are scribbles, simple lines drawn by the user. The user is usually requested to draw at least one scribble inside the object of the interest and one on the outside, those two scribbles give the algorithm the basic information it needs in order to segment the object of interest from the rest of the image. In this project our aim was to implement a scribble-based segmentation algorithm for extracting object from natural photos and pasting them seamlessly into a different background, while doing the process fast enough so it can be ran on an Android device in real time. The algorithm was first developed and tested on a personal computer with the help of openCV’s C++ libraries and was then ported to Android using Android’s Native Development Kit, while the application interface itself was built using Android’s standard Software Development Kit and was written in Java.
3 Introduction

Nowadays, we all carry a high-resolution camera in our pocket in the form of a smartphone, but our smartphone isn’t only a camera, it’s actually a quite capable computer with hardware that would have been more than sufficient for a personal computer a few years back. That power, with the combination of new, more efficient, Computer Vision algorithms allows our mobile phones to run CV tasks that used to require a powerful computer or server.

In this project we’ve chosen to bring a segmentation algorithm to the mobile world. Implementing a scribble based segmentation algorithm on a mobile device is appealing because of various different reasons.

- Easier to generate input - just snap a picture and trace the scribbles using your fingers, more natural than using a mouse.
- Interesting to user - From the user’s point of view the app allows him to “Magically” create stickers and put them on different backgrounds, no need to understand the app in order to enjoy it.
- Can be optimized - There are many different ways to improve the running time of the segmentation algorithm, in order to get a fast enough algorithm for a mobile device. That way, the right balance between speed and precision can be found.

3.1 General Project Progress

The project progress can be segmented into several stages:

The first stage was to complete the theoretical and practical background needed for implementing the project. I learned about the theoretical background of Image Segmentation and the method presented in our article of interest. As well as getting familiar with openCV’s libraries, from the basic data structures that are commonly used, to the different functions provided by the openCV library.

After completing the needed background I was able to start implementing the basic segmentation algorithm on my personal computer and get first segmentation results.

The next stage was to port the algorithm to an Android Device. At first, a basic app layout was designed that allows the user to take a picture and draw scribbles on it, and then the segmentation algorithm itself was ported using openCV’s Android port and the Native Development Kit.

The final stage consisted of a few iterations of optimizations and fixes to the algorithm: we’ve allowed scribble addition without full recalculation, improved the object’s matting by calculating the alpha values for the object’s edges and improved running time by doing the calculation itself on a scaled-down image.
4 Theoretical Background & Algorithm

As noted earlier our task was to implement an algorithm that segments an object of interest out of an image by using user generated scribbles as input.

4.1 Segmentation Algorithm

The idea that stands behind scribbles-guided segmentation is quite simple, so we'll try to show the intuition behind it. Let’s assume the user has drawn two scribbles, one inside our object of interest, the $\mathcal{F}$ scribble, and one outside of the object in the background, the $\mathcal{B}$ scribble. We will call the group of pixels that were marked using the inner scribble $\Omega_{\mathcal{F}}$ and the group of pixels marked using the outer scribble $\Omega_{\mathcal{B}}$. We assume that all the pixels in $\Omega_{\mathcal{F}}$ are part of our object, while the pixels in $\Omega_{\mathcal{B}}$ aren’t, the question is how can we propagate the information to the other pixels in our image.

A logical way to propagate the information is to start reaching out from each group to the neighboring pixels and ask ourselves how “close” they are to the original one. Given the right definition of proximity, the “closer” the pixels are to the original group the more likely they are to be part of it. So, we can try and calculate the proximity of each pixel to either group and then pair the pixel to the closer group. Eventually we will get a blob of pixels that are closer to $\Omega_{\mathcal{F}}$, and we can denote them as the object.

Stated formally, we want to calculate the geodesic distance from our scribbles to each pixel using a specific weight function and attach every pixel to the closer scribble. One should see that our weight function is actually our definition of proximity, it describes how “close” two neighboring pixels are to each other.

Assuming we’ve got a definition of weight, $W$, the geodesic distance between two scribbles can be mathematically defined as follows:

Let $C_{s_1,s_2}(p)$ be a path that connects the pixels $s_1$ and $s_2$, from $p = 0$ to $p = 1$.

$$\int_0^1 |W \cdot \dot{C}_{s_1,s_2}(p)| dp$$

would be the total weight of the path according to our weight function, and it follows that the geodesic distance would simply be the path with the shortest weight:

$$d(s_1, s_2) := \min_{C_{s_1,s_2}} \int_0^1 |W \cdot \dot{C}_{s_1,s_2}(p)| dp.$$

The geodesic distance from a scribble can easily be defined as the minimal distance from the pixels that belong to the scribble:

$$D_l(x) = \min_{s \in \Omega_l} (s, x), \quad l \in \{\mathcal{F}, \mathcal{B}\}$$

And the segmented group, $O$, can be defined as follows:

$$x \in O \iff D_{\mathcal{F}}(x) < D_{\mathcal{B}}(x)$$

We now have the basic layout of a scribble-based segmentation algorithm, but in order for our algorithm to be meaningful what we need to find is a good proximity definition that would be used as our weight function. This decision is a critical one in our Segmentation Algorithm. It’s quite clear that spatial distance between pixels isn’t good for our need and that we must take into account the actual colors of the pixels in one form or another.

In our algorithm we followed the idea suggested in “A Geodesic Framework for Fast Interactive Image and Video Segmentation and Matting” by Xue Bai and Guillermo Sapiro. As we’ve already noted we’ve got two groups of pixels $\Omega_{\mathcal{F}}$ and $\Omega_{\mathcal{B}}$ for which we already know whether they belong to the background or to the object. We can try and profile those pixels to calculate the likelihood that a new pixel belongs to either group. The paper suggests that we’ll take each group separately and profile it in order to get a Probability Density Function (pdf), which gives the probability that a pixel belongs to the group according to its color values. So, now we have in our hands the probability that a pixel belongs to the foreground and the probability that it belongs to the background, using those two pdfs we can know calculate the likelihood that a pixel will belong to the foreground rather than the
background. The likelihood function will simply be defined as the normalized probability that a pixel belongs to the foreground rather than the background.

Recall that what we are missing is a definition for “closeness” between two pixels, the paper shows that a good definition for “closeness” would be the difference of the likelihood values between the pixels. So, if two pixels are very likely to belong to the object they’ll be close to each other and the same goes for two pixels that aren’t likely to belong to the object.

Formally we can define $W$ as follows:

Let $Pr(x|F)$ be the pdf computed from $\Omega_F$ and Let $Pr(x|B)$ be the pdf profiled from $\Omega_B$.

The likelihood $P_F(x)$ is $\frac{Pr(x|F)}{Pr(x|F)+Pr(x|B)}$, the normalized probability value.

The weight function is simply defined as the gradient of the likelihood, $W = \nabla P_F(x)$.

Using $W$ we finally have a complete segmentation algorithm that we can implement.

### 4.2 Alpha Matting

In the previous subsection we defined our segmentation algorithm, which produces us our basic segmentation that divides the original image into background and object. One of various uses of the segmentation could be to matte the object into a different image. In order for the result to look more convincing a special treatment can be applied on the edges of the object.

One can see that our segmentation algorithm defines a binary mask, every pixel belongs either to the object or to the background. But as every good designer knows, matting looks much more convincing when the transparency of the edges changes progressively. Instead of using a simple alpha transition we could use a variation of our segmentation algorithm on a band around the object and calculate the matching alpha values for each pixel, so that the “further” away a pixel is from the object the more transparent it would be. Our band is defined by taking the boundaries of our mask and expanding them in both directions, thus creating a closed area that will be our area of interest. The thickness of the band is also a parameter of our algorithm that will be defined in the implementation section. The alpha value itself will be calculated according to two weight values given to each pixel, one defined by the foreground’s probability and distance functions and one defined by the background ones.

The first step in our matting algorithm is to recalculate the distance values for the pixels in the band using the information obtained from our segmentation. We define $\mathcal{F}$ to be the inner contour of our spanned band, and $\mathcal{B}$ to be the outer contour of the band. Those two scribbles define the border of our alpha calculation, the pixels in $\Omega_F$ would be opaque and the pixels in $\Omega_B$ would be transparent. Using those two scribbles we will recalculate the distance and likelihood functions $D_F, P_F$ and $D_B, P_B$ for the pixels in the band, their values are now locally adapted to the band. We can now define the weight functions as a combination between our adapted functions, $w_l(x) = \frac{P_l(x)}{D_l(x)^r}, \quad l \in \{F,B\}$. The alpha value will simply be the normalized foreground weight $\alpha(x) = \frac{w_F(x)}{w_F(x)+w_B(x)}$.

Note that the weight functions have an $r$ parameter which we haven’t talked about. The $r$ value we choose defines the power of the actual weighted distance on the final weight value. For example, if we choose $r = 0$ then the alpha values would simply be defined by the likelihood function without taking the actual distance from $\mathcal{F}$ into account, while $r \to \infty$ will give us hard segmentation. In the implementation section we will explain how we’ve determined the $r$ value we want to use.

By applying the matching alpha value to each pixel in the band we can get our improved matting :)
5 Software Implementation & Further Algorithm Development

In the previous section we’ve presented the basic idea of our segmentation algorithm. The user draws a foreground scribble and a background one, we find the weighted geodesic distance from each scribble to every pixel in the image and tag the pixels that are closer to the foreground scribble as the object of interest.

5.1 Implementing the algorithm

The next step in our project was to actually implement the algorithm. As mentioned earlier the algorithm was implemented in C++ with the help of openCV’s C++ libraries.

5.1.1 openCV Libraries

openCV is an open source library used for implementing Computer Vision algorithms, hence the name. The library provides the needed infrastructure for developing computationaly efficient, real-time computer vision applications. The libraries provided in openCV vary from basic data structures such as Matrices and Vectors to complex algorithms such as GrabCut and SIFT. openCV was introduced with a basic C interface, which was later updated to a C++ oriented one. Nowadays, openCV also has a full Python interface as well as a Java one, and was ported to various different platforms including Android and iOS. The portability of openCV made it ideal for our needs.

Though the final objective was to get the algorithm up and running on an Android device which uses the Java interface, we’ve decided to work primarly with the C++ interface. Thus retaining the portability of our code and allowing us to easily use other C++ libraries if we choose to.

Our main use of openCV was simplfing the working progess using the bulit in data structures, input and output functions and other general utilities, the actual algorithm was programmed as part of the project.

5.1.2 Dijkstra Algorithm

We’ve started the implementation from the one of the basic parts of the algorithm, the distance calculation. Since our image is actually discrete we can regard it as a graph where every pixel is a node connected by edges to it’s neighboring pixels and solve the problem by running a Shortest Path Algorithm. We’ve chosen the Dijkstra Algorithm due to it’s simplicity and reasonable complexity. We should note that from Dijkstra’s point of view the scribble can be thought of as a single source, technically we can simply start the algorithm by initializing every pixel $\Omega_F$ to distance zero and adding them to group of visited nodes.

We followed the elegant and well-known Priority Queue based Dijkstra Implementation and tested our results both numerically and graphically. The runtime of the Dijkstra algorithm is dependent on the size of the image, our implementation measured in about ~0.3s for a 400x400 image and got increased to ~4s when working with a 1000x1000 image.

5.1.3 Bresenham’s line algorithm

Due to the update time it takes to register the user’s touch the line provided by the user isn’t continuous, we’ve solved that problem by simply using Bresenham’s line algorithm. Bresenham’s algorithm suggests a way to complete a path between two given points that will be as close as can be to a straight line between the points. Running the algorithm between every two points registered from the user completes the lost points and gives us a good enough approximation of the user’s continuous scribble. The runtime of breseham’s algorithm is negligible which is one of it’s advantages.
5.1.4 Weight Calculation

As explained in the theoretical background, the weight is defined as the difference between the likelihood that two pixels belong to the object. Our first step in calculating the likelihood is calculating the pdf for both the foreground and background scribbles. There are several different ways to approximate the needed pdf, the main method we used was the Fast Improved Gauss Transform with Tree Data Structure, other solutions can be seen in the Further Work section.

Before we talk about FIGTree let us consider a simple, yet unsatisfying, solution. Since we are working with a 3-Channel image every pixel is defined by 3 color values. We can easily take the user’s scribble and build a 3D Histogram based on the pixels that it consists of. Color variations that are common in the object will belong to larger bins, while seldom ones will belong to smaller bins. Normalizing the histogram will give us a basic pdf, where for each pixel the probability value is denoted by the proportional number of times it appeared in the scribble. However, it’s clear enough that this method isn’t satisfactory. Any pixel that didn’t appear in the original scribble would have a zero probability to belong to the object whether it’s close to a common color or not.

FIGTree allows us to approximate the pdf using the Kernel Density Estimation method. Instead of simply building a 3D histogram as noted earlier by adding every pixel to a single bin, we take every pixel and place a Gaussian kernel in it’s place, the kernels are then summed to give us our estimation. By doing this operation of smoothing during the histogram creation using Gaussian kernels we get a result which makes a much more reasonable pdf.

We’ve chosen to work with the Lab color space, so every pixel is passed to FIGTree in that form. The runtime of the pdf calculation is consisted mainly from the FIGTree calculation which is dependant on both the size of the image and the length of the scribble in question. When working on a 400x400 image, simple scribbles will measure in about ~0.3s, while more complex ones could measure in ~1s and more. Once we calculate the pdf values for both the foreground and background scribbles we can calculate the likelihood using the formula stated in the theoretical background section.

5.1.5 Combining the algorithms

Once we’ve implemented the basic building blocks we were ready to combine them to the major algorithm

- Get scribbles from user and fill them using Bresenham’s algorithm.
- Calculate the pdf for the front and back scribbles using FIGTree.
  - We’ve experimented with several bandwidth values for the KDE until we were happy with the results
- Calculate the likelihood function for each pixel based on both pdfs.
- Run Dijkstra for each scribble while using the likelihood function to determine the weights of the edges.
- Create a binary mask that contains the pixels that are closer to the foreground scribble.

Applying the mask obtained from the joined algorithm on the original image gives us our resulting segmentation, and thus completes the first stage of development
The total runtime on this 600x800 image measured in ~6s, about ~1s for each pdf calculation and ~2s for each Dijkstra run.

5.2 Android First Port

Before doing any other updates to the algorithm we wanted to port our existing algorithm to Android, (to be precise we've started it even earlier, when the Dijkstra Algorithm was complete).

5.2.1 Simple User Interface

For the first Android port our mission was to create a simple interface that allows the following operations:

- Loading an Image either from gallery of by using the camera.
- Allowing scribble drawing on the loaded image.
- Displaying needed information and instructions.

The resulting interface was a simple Holo-Themed interface with a single screen where all the action happens. The buttons were positioned on the Action Bar, while the instructions where positioned at the bottom of the screen with some more action buttons, the rest of the screen was used to display the loaded image. When entering the application the user was instructed to load an image, after loading an image the user was prompted to draw the foreground scribble and background scribble one after the other. Once both scribbles were drawn the application started the calculating process and the resulting object showed on the screen.
The user interface was manually designed using Android’s XML Layouts. In order to allow scribble drawing the user touches were recorded, translated to coordinates on the actual image and saved as part of the user’s scribble, every touch was also sent to the Drawing View which displayed it on top of the original image.

5.2.2 Porting the Algorithm

As mentioned earlier in the section openCV now have an Android port. That means that the functions and data structures we rely upon are still available for us when working with Android. The library itself is provided using a standalone library that the user is prompted to download. OpenCV’s Android port has it’s own Java Interface, but in order to allow portability between our different versions and use other C++ libraries we’ve decided to go with the C++ interface and use the source code we’ve already written.

The Android SDK allows development only using the Java Programming Language, but we were able to run our C++ code using Android’s Native Development Kit. The NDK is a toolset that allow developers to implement various parts of the application using native C/C++ code and call the native functions with the help of Java Native Interface.

Another advantage of using the NDK is improved running time, since we cut down the time needed for translating Java’s byte code to actual machine instructions by the Virtual Machine. We should note that the NDK is not recommended for most Android applications and is much less documented, but after some painful tries we were finally able to get the port working. Once the scribbles are ready they are passed using a JNI call to our C++ function which segments the image and returns the object of interest which is then displayed as a bitmap.
Naturally, the total runtime of the algorithm was increased after porting it to an Android Device due to the hardware limitations. For example, the runtime on a simple 400x400 image with basic scribbles could get increased from ~1s to ~3s, with a much more noticeable effect when working on big images.

5.3 Alpha Matting Implementation

In the theoretical background section we’ve presented the use of alpha matting in order to get better looking result when pasting a segmented object into another background.

The first step in Alpha Matting was to define the band around the object. This was easily done using openCV::findContours on our segmented mask, which returns a contour surrounding our object of interest. The contour was then used to span a band and get it’s outer borders which were used as our two new scribbles. One of the advantages of using findContours is it’s ability to save the hierarchy between different contours and thus allowing us to easily take care of objects with holes.
Once we had our new scribbles, the next step was to run most of our segmentation algorithm for the pixels in the band. The likelihood function was recalculated according to our region of interest as well as the distance functions from each scribble. The alpha values for each pixel was then calculated using the formula we stated above. Since our calculation is done only on the pixels in the band the runtime of the alpha matthing varies according to the size of the band.

The difference between a simple mask and one with alpha values can easily be seen in this picture:

![Original Mask Vs Alpha Mask](image)

The difference is most visible when the object contains fur which doesn’t cut it through the original mask but does show on the alpha mask.

### 5.3.1 Choosing Alpha Intensity

As we’ve mentioned in our theoretical background the weight functions which are used to produce our final alpha value have an $r$ parameter which we haven’t defined yet.

$$ w_l(x) = \frac{P_l(x)}{D_l(x)^r}, \quad l \in \{F, B\} $$

The $r$ value defines the power of the actual weighted distance on the final weight value. The larger the $r$ value is, the more hard-segmented the final result will be. In order to find a good $r$ value we’ve generated our results with a range of different $r$ values, finally choosing the value that gave us the right balance for most pictures.

![Different r values](image)

Like we’ve seen earlier, the difference is most noticable on objects with fur, the final $r$ value we’ve chosen was 1.5.
5.4 Multiple Scribbles Implementation

As can be seen from the shown figures two scribbles are often enough data for the algorithm to generate the right segmentation. However, for more complex images the first scribbles the user has drawn might not be sufficient. The logical solution would be to allow the user to refine his results by adding secondary scribbles and re-segement our object. The user can add one scribble for each recalculation, once the new scribble was added the matching pdf function is recalculated (no need for recalculation for the other scribble) and the rest of the segmentation algorithm is used again to create a more precise mask.

Figure 7: Multiple Scribbles

Multiple scribbles will always be needed if the object has holes that are part of the background or is consisted of two seperated objects since each foreground scribble gives us only one continuous object.

5.5 Scaling the Image

It's quite clear that the size of the image has a major effect over the running time of our algorithm. For larger images both the pdf calculating and Dijkstra running will take considerably more time, as we've already seen. When running the algorithm on a Personal Computer the total running time is usually still reasonable. However, when running our completed algorithm on an Android Device, which is after all not as strong as a PC, the difference became too noticable, thus harming our Real Time segmentation. In order to overcome our problem we've decided to do the actual segmentation calculations on a scaled down version of the image and then blow up the given mask. Of course, the quality of the edges was harmed by the process but we overcome this problem using two methods.

5.5.1 Mask Smoothing

As mentioned in the section’s intro we’ve chosen to work with a scaled down image, the size of ~400x400 was found ideal. The image was first scaled down to the right size using openCV::resize with a bilinear interpolation. Once the calculation was done the resulting mask was scaled back using openCV::resize with a bicubic interpolation and was then rounded to a binary mask.

As can be expected the resulting mask was choppy and created an unnatural effect.
The obvious solution was to round the scaled up mask, we’ve experimented with different blurring intensities to find the balance between smoothing noise and removing actual data.

We found out that a radius of $\frac{5}{\text{scale}}$, where scale is the proportion between the original image and the scaled one, gives us good balance between smoothing and over-smoothing. However, smoothing still made us lose data in pictures with special edges, for example furred objects since the hair strands were easily lost, which leads us to our next step in overcoming the scaling noise.
5.5.2 Alpha Calculation

As we’ve already seen alpha calculation helps in getting better looking edges. Since the alpha calculation is done only once in each use, after all the scribbles were drawn and the user is happy with the result, we’ve decided that we can allow the alpha algorithm to be ran on the original image. Since the alpha calculation is done only on the band around the object the running time would still be reasonable (~10s) even for fairly large images, we’ve determined the thickness of the band to be $10 / \text{scale}$.

The combined effect of the smoothing and alpha matting can be seen here:

![Original Scaled Mask](image1)

![Smoothed Mask](image2)

![Mask With Alpha](image3)

Figure 10: Improvement steps
5.6 Updated Android Port

Once we’ve got the alpha part complete it was time for an updated Android port that allowed the user to take the resulting object (a.k.a cutout) and stick it in another picture. Thus the following changes were made to allow the sticker interface as well as other needed features

- Welcome Screen - Instead of entering the main screen immediately the user was first presented with a welcome screen with simple instructions.

- Main Screen - the main screen was updated to support multiple scribbles.
  - The segmentation result was updated to show as a contour around the object, thus allowing the user to add new scribbles on the background/object easily.
  - A magnifying circle was presented next to the user’s finger while drawing to show him the section he was currently hiding in greater details.
  - A dedicated zooming button that allows you to zoom into a specific region of the image before drawing the scribble.
  - A button that allows the user to finish the scribbling mode and move over to the sticker part.

- Sticker Screen - The segmented object is presented on a special view that allows to move and resize it with natural gestures.
  - The default background of the view is the camera feed, the user can decide to snap a picture and freeze the background or use an image from the gallery.
  - Once the object has been positioned the user can save the result.
5.7 Additional Features

Once we have our segmentation mask there are many different uses and effects that can be applied over either the object or the background. We’ve discussed some different effects and decided to implement blurring as a proof of concept.

5.7.1 Blurring

When applying blur to our segmented image we can decide to either apply the blur effect to the object or the background. Blurring either the object or the background can be easily done by applying a Gaussian kernel to the object of interest. Natural blurring of the object can easily be achieved by blurring a slightly bigger slice of the object.
image.
However, it turns out that when trying to blur the background it’s not as easy to get a natural looking result since
the object creates a ghosting effect around it’s edges and thus ruining the sharp look we are trying to get. The
problem was overcome using special treatment for the edges, the kernel was updated according to the mask so
that pixels that belong to the object won’t be taken into account for each calculation.

![Basic Blur](image1.png) ![Blur with Anti-Ghosting](image2.png)

Figure 12: Different smoothing intensities

Notice the ghosting effect on the edges with basic blur. The main disadvantage of our smart blur is an increase
in the total runtime of the blurring process. For example for blurring a 400x400 image, with a radius of 20 pixels,
the runtime could get increased from ~0.2s to ~1.2s.

5.7.2 Porting the additional feature

In order to allow the user to choose the action he would like to apply to the segmented image we’ve added another
mode to the main screen. Once the user finishes the segmentation process a new bar pops up from the bottom of
the screen and shows the different options.

The blurring feature was ported as a separate screen where the user can choose the intensity level using a SeekBar.
Moving the SeekBar to the right will blur the object while moving it to the left will blur the background. We’ve
restricted the blurring so that even the smart blur will still finish after a couple of seconds.
Figure 13: Android Blurring UI
5.8 Android Design Makeover

Courtesy of the GIP lab I was given the opportunity to work with a professional designer on updating the application UI and bringing it up to date with Android’s latest guidelines, after a few iterations of work we’ve reached the following design, that speaks for itself.

As can be seen we’ve also added the feature to add more than one cutout.

Figure 14: Android Makeover
6 Results

We’ve already showed the different kind of results gained by our algorithm and the process of fixing them, so this section will be used to show some of the other cool results that didn’t make it into the rest of the article.

Segmentation Results

Figure 15: Hand Segmentation

Beware of the Thing!
Figure 16: Rat Segmentation

Figure 17: Giraffe Segmentation

Figure 18: Ballon Segmentation
Figure 19: Deer Segmentation

Sticker Results

Figure 20: Android Results
Blurring Results

Figure 21: Flower Blurring

Figure 22: Monkey Blurring
7 Further Work

Our project gives us a real time system for Image Segmentation, the resulting segmentation can be used in many different ways as a starting mask for other algorithms.

7.1 Content Aware Fill

One of the features we discussed was Content Aware Fill. Content Aware Fill is a technique that allows us to remove a selected region from the image while refilling the empty area with the right patterns in order to create a natural looking result without showing any trace of the removed object. The mask provided by our algorithm could be used as a starting point for implementing a mobile version of Content Aware Fill.

As a proof of concept we've made some experiments with openCV’s inpainting that allows simple filling of small regions.

As can be seen Inpainting isn’t good enough for complex backgrounds and Content Aware Fill is required.

7.2 Poisson Blending

Another feature we think could be interesting is Poisson Blending. Poisson Blending allows us to seamlessly blend two pictures based on solving Poisson equations with the right boundary conditions. One could create an interesting stickers interface by allowing the user to blend the sticker into the rest of the image, thus creating a truly seamless image. We should note that the result won’t look natural due to the fact that the colors of the entire object are changed by the Poisson Blending.

We’ve actually implemented Iteration based Poisson Blending Algorithm and experimented with it. One of the interesting features we tried was using a variation of Poisson Blending and apply it only to the object’s edges with matching boundary conditions. We finally decided that our current algorithm isn’t practical for real time usage, but further development might prove fruitful.
Those are just a few examples of the ways our project can be extended and used a basis for other projects.

7.3 FIGTree Alternatives

Another direction in which the application can be taken is further improvement of the application’s running time. Using FIGTree we are usually able to get the first segmentation in ~5s with additional calculation time as a final step for the alpha matting that’s dependent on the image's dimensions, but goes on for about ~10s. But what about using alternative ways for approximating our pdf?

In our theoretical background section we’ve explained why a simple 3D Histogram won’t be useful for creating a good pdf approximation, however we could use a variation of a 3D Histogram to get a better approximation. Our first suggestion for replacing FIGTree was using a smaller 3D Histogram that uses bins, every bin will cover a range of values according to the number of bins. The scribble will then be used to fill the matching bins and the histogram will be used to determine the pdf value of each pixel (we should note that all the pixels that fall into the same bin will share the pdf value).

Surprisingly, the approximation was good enough for simple images.
However it failed for images where the contrast between the object and background wasn’t strong enough. Further than that, due to the relatively small histogram size scribble addition wasn’t always able to fix bad results since better resolution was needed.

![Hand 8x8x8 Histogram Result](image)

In order to improve those results we’ve decided to expand the histogram while smoothing the resulting pdf in order to get the pdf to cover more pixel values. That meant that there are now two parameters that define our results, the size of the histogram and the radius of the blurring, using a process of elimination we were able to deduce that using a 16x16x16 Histogram and blurring with a radius of 8 pixels yields the closest results to FIGTree, while taking considerably less time, though more scribbles were often required.

![Hand 16x16x16 Histogram With Smoothing](image)

The main advantage of pdf Smoothing is the improved running time, unlike FIGTree the length of the user scribble doesn’t have a major effect on the running time since the smoothing is done only afterwards, the smoothing itself isn’t too demanding and thus the total runtime is lower than FIGTree in a couple of seconds.
Further experimenting with pdf Smoothing shows that it is still not as good as FIGTree in creating a clean segmentation. However, the running time is a major advantage and I believe that further research in that direction can yield an algorithm that is still faster than FIGTree but also gives more accurate result with less data.
8 Summary

In this project we’ve implemented an Android Application for Real Time segmentation of objects using a simple scribble interface. The application allows the user to naturally segment the object of interest and use it as a sticker or apply other effects upon it.

For me this project was the first real meeting with the world of Computer Vision and Image Processing and I was fascinated by the algorithmic side of the task at hand. I’ve learned a lot in the theoretic side of CV during the reading of the related article and the background of the additional features we wanted to add, as well as in the practical side of things with the days spent working with openCV and the NDK. Designing a natural android interface for the app was also an integral part of the project. I think that one of the nice things about this project is the combination between the algorithmic side of developing the segmentation algorithm and the technical side of wrapping it inside a simple app that children can use without having any idea what happens in the background (At least my little sister is very excited about it).

I’d like to thank my supervisors Anastasia and Aaron for their help and endless patience during the entire project and for the pleasure of working with them. I’ve started this project with almost no knowledge in the field of Image Processing and Anastasia patiently helped me in completing both the general background and the specific knowledge and understanding needed for the project. Aaron’s help in both the algorithmic and technical side of the project was without contest. I’d also like to thank Aaron & The GIP Lab for the opportunity to work with a professional designer on the application’s look and feel, I think the results speak for themselves.

9 References

- A Geodesic Framework for Fast Interactive Image and Video Segmentation and Matting, Xue Bai and Guillermo Sapiro
- FIGTree: Fast Improved Gauss Transform with Tree Data Structure
  - http://www.umiacs.umd.edu/~morariu/figtree/
- OpenCV 2 Computer Vision Application Programming Cookbook