# ECE661: Homework 6

Ahmed Mohamed (akaseb@purdue.edu)

October 28, 2014

## 1 Otsu Segmentation Algorithm

Given a grayscale image, my implementation of the Otsu algorithm follows these steps:

1. Construct a 256-level histogram $h$ of the image, such that $h[i] = n_i$ is the number of pixels whose grayscale value equal to $i$.

2. Calculate the average grayscale value of the image.

   $$\mu_T = \sum_{1}^{L} i p_i$$

   where

   $$p_i = n_i / N$$

   and $L$ is the total number of levels, and $N$ is the total number of pixels in the image.

3. For each level in the histogram, calculate:

   (a) The zeroth-order cumulative moment

   $$\omega(k) = \sum_{1}^{k} p_i$$

   (b) The first-order cumulative moment

   $$\mu(k) = \sum_{1}^{k} i p_i$$

   (c) The between-class variance

   $$\sigma_B^2(k) = [\mu_T \omega(k) - \mu(k)]^2 / [\omega(k)(1 - \omega(k))]$$

4. Choose $threshold = k^*$ such that $\sigma_B^2(k^*)$ is maximum.

5. Construct a mask whose pixels is 1 if the corresponding pixels in the original image is greater than the threshold, and 0 otherwise. This mask represents the foreground of the image.

6. Repeat this process for an arbitrary number of iterations. In each iteration, the new histogram will not contain the pixels below the threshold. The number of iterations is chosen manually to increase the quality of the results.

## 2 RGB Image Segmentation Using the Otsu algorithm

Given a color image, my implementation follows the following steps to extract the foreground of the image.

1. Separate the RGB color channels of the input image into three grayscale images.
2. Get the foreground mask for each channel using the Otsu algorithm as described earlier.
3. To merge the three masks together into a single foreground mask, we need to manually know about the colors of the foreground and the background of the image.
   (a) For the lake image, the lake is mostly blue, while the background has other colors. Hence, we should treat the blue mask as the foreground mask, while the green and red masks as the background masks. Hence, the overall foreground mask is:

   $$mask = mask_b\ AND\ (NOT\ mask_g)\ AND\ (NOT\ mask_r)$$

   where $mask, mask_b, mask_g, mask_r$ are the overall, blue, green, and red masks respectively.

   (b) For the tiger image, there is no dominant color for the background or the foreground. Hence, we treat all the masks as foreground masks. Hence, the overall foreground mask is:

   $$mask = mask_b\ AND\ mask_g\ AND\ mask_r$$

## 3 Texture-based Segmentation Using the Otsu algorithm

Given a color image, my implementation follows the following steps to extract the foreground of the image.

1. Convert the image into a grayscale image.
2. Create a grayscale image whose pixels represent the variance of the grayscale values of the $N * N$ window around the corresponding pixels in the original grayscale image.
3. Do the previous steps for $N = 3$, $N = 5$, and $N = 7$ to get three grayscale images that represent the texture-based features of the original image. These three grayscale images are considered three channels of an image.
4. Get the foreground mask for each channel using the Otsu algorithm as described earlier.
5. To merge the three masks together into a single foreground mask, we need to manually know about the structure of the foreground and the background of the image.
   (a) For the lake image, the pixels of the lake have almost no variance. Hence, we treat the three masks as background masks. Hence, the overall foreground mask is:

   $$mask = (NOT\ mask_1)\ AND\ (NOT\ mask_2)\ AND\ (NOT\ mask_3)$$

   where $mask, mask_1, mask_2, mask_3$ are the overall mask and the masks of the individual channels respectively.

   (b) For the tiger image, the pixels of the tiger have larger variance than others. Hence, we treat the three masks as foreground masks. Hence, the overall foreground mask is:

   $$mask = mask_1\ AND\ mask_2\ AND\ mask_3$$

   where $mask, mask_1, mask_2, mask_3$ are the overall mask and the masks of the individual channels respectively.

## 4 Noise Elimination

The foreground masks may be noisy. To eliminate the noise we use combinations of dilation and erosion as follows:

1. Erosion then Dilation with a square window to remove the noise in the background.
2. Dilation then Erosion with a square window to remove the noise in the foreground.

For the lake image, we needed both approaches to eliminate the noise in both the foreground and the background. However, for the tiger image, we used only the second approach because there were many holes in the foreground, and applying the first approach would have removed most of the mask.

## 5 Contour Extraction

Given a binary mask, the contour is the foreground pixels that touch the background. Thus, we follow the following steps:

1. For each pixel in the binary mask:
   (a) If the pixel value is 0, it doesn't belong to the contour.
   (b) If the pixel value is 1, and all adjacent pixels in a 3x3 window (8-connectivity) are 1, it doesn't belong to the contour.
   (c) Only if the pixel is 1, and one or more of its adjacent pixels are zero, it is considered on the contour.

## 6 More Observations

1. The results quality of the texture-based segmentation is much better than the image-based segmentation.
2. Texture-based segmentation consumes more time than the image-based segmentation.
3. Image-based segmentation require human knowledge of the different colors of foregrounds and backgrounds. For example, my implementation for the lake image will not produce good results for other images, because we had to use the information that the foreground is blue. This might not be the case with other images.
4. Texture-based segmentation requires less knowledge about the foreground and the background. It will divide the image into areas with high variances, and areas with low variances. We still need to take a human decision about which is the foreground and which is the background.
5. In brief, The Otsu algorithm is fast, but very limited. It doesn't produce very good results. Using texture-based segmentation instead of image-based segmentation increases the quality of the results, especially with challenging images such as the tiger image.
6. Erosion and Dilation are very effective removing the noise in both the background and the foreground. However, one should be careful choosing the window size, because this process could remove the entire mask.

# 7 Results

## 7.1 Image 1: Lake

Figure 1: The input image

### 7.1.1 RGB Segmentation

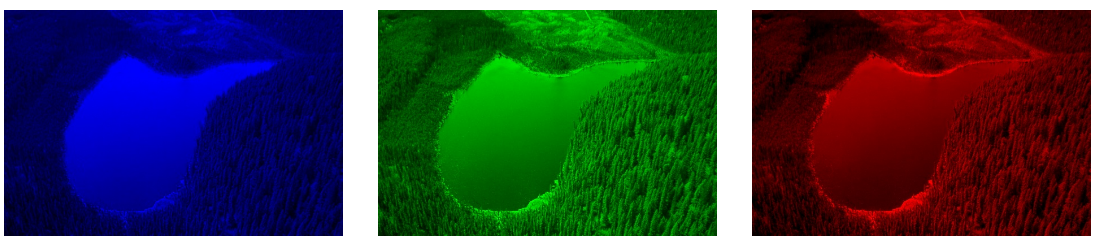

Figure 2: The three color channels (BGR) of the image

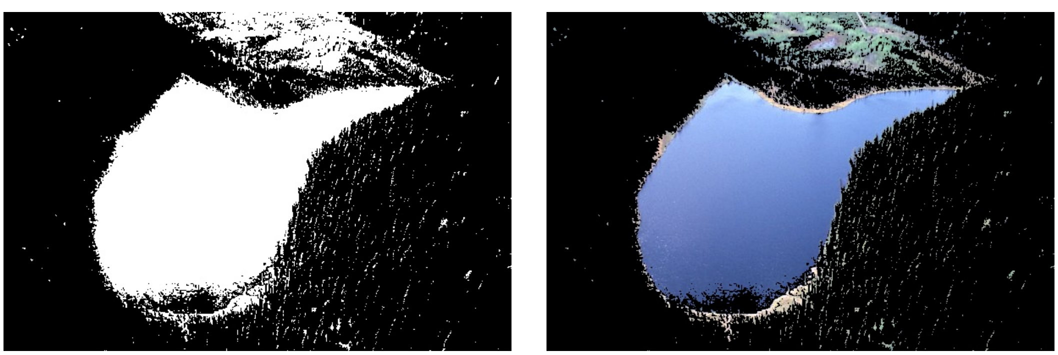

Figure 3: The foreground mask using the blue channel of the image, and the corresponding foreground

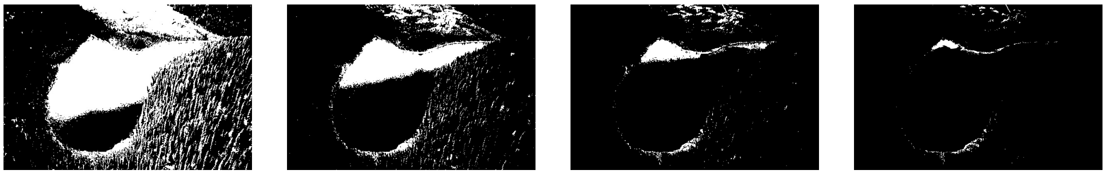

Figure 4: The foreground masks using the green channel of the image (4 iterations of Otsu's algorithm are used.)

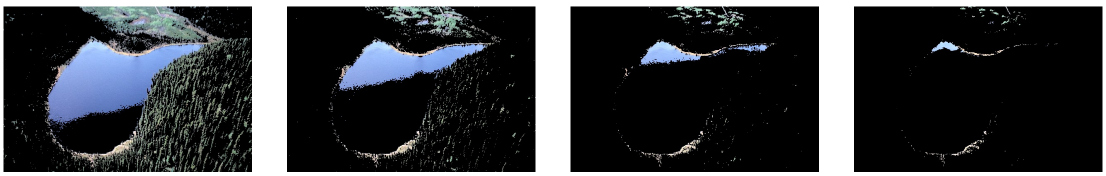

Figure 5: The foreground of the image corresponding to the masks of the green channel (4 iterations of Otsu's algorithm are used.)

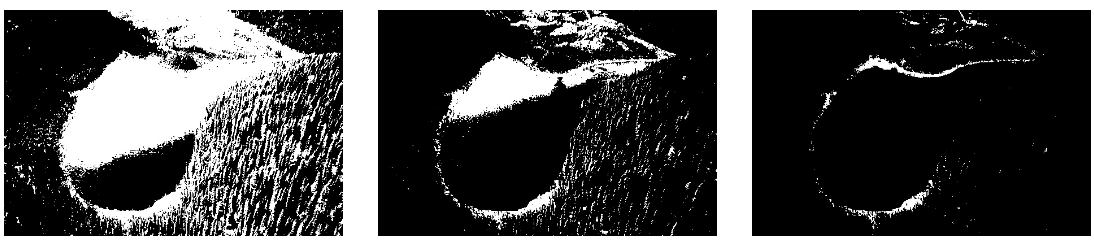

Figure 6: The foreground masks using the red channel of the image (3 iterations of Otsu's algorithm are used.)

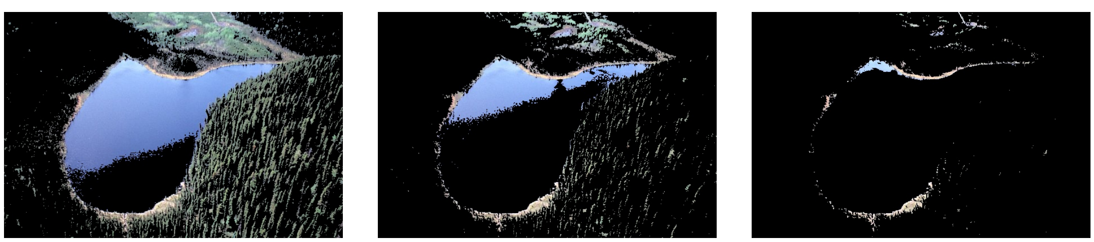

Figure 7: The foreground of the image corresponding to the masks of the red channel (3 iterations of Otsu's algorithm are used.)

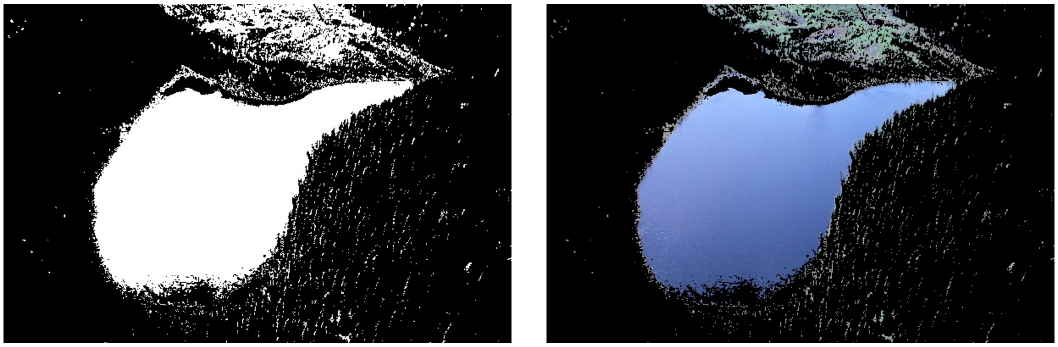

Figure 8: The overall mask using the masks of the three channels, and the corresponding foreground

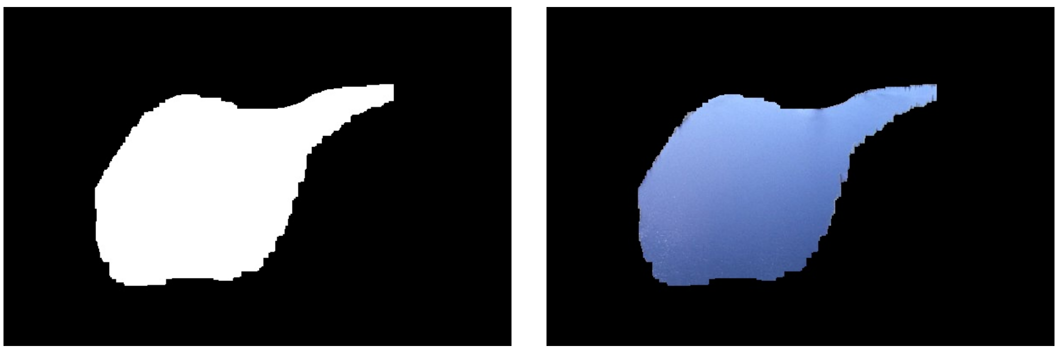

Figure 9: The final overall mask after removing the noise, and the corresponding foreground

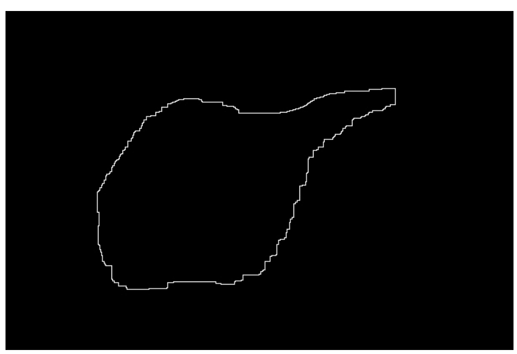

Figure 10: The final contour of the image

#### 7.1.2 Texture-based Segmentation

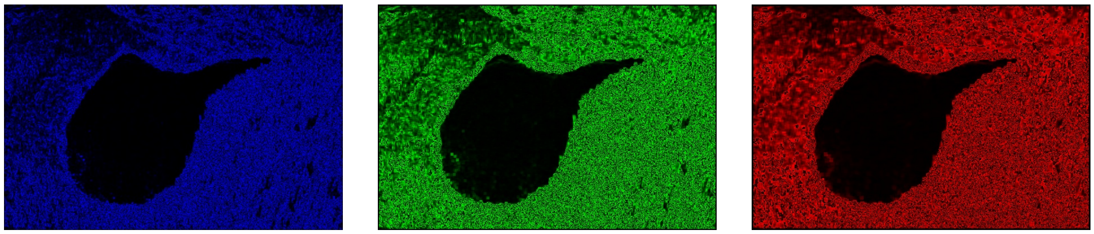

Figure 11: The three channels of the image representing the texture-based features (corresponding to the windows 3x3, 5x5, and 7x7 respectively)

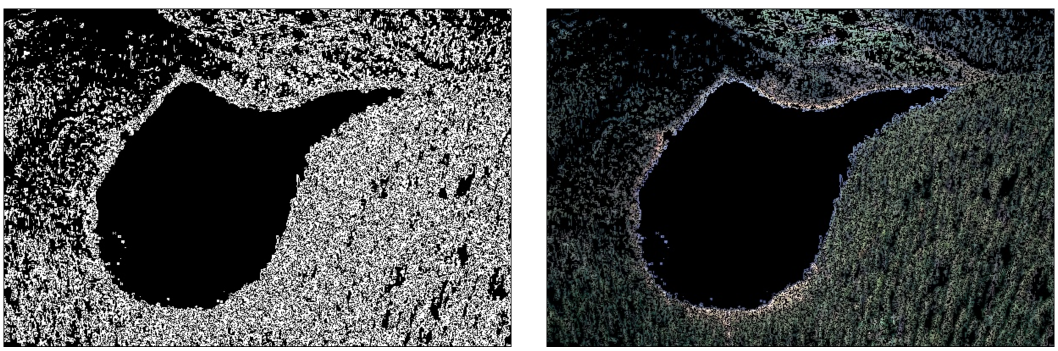

Figure 12: The foreground mask using the first channel of the image, and the corresponding foreground

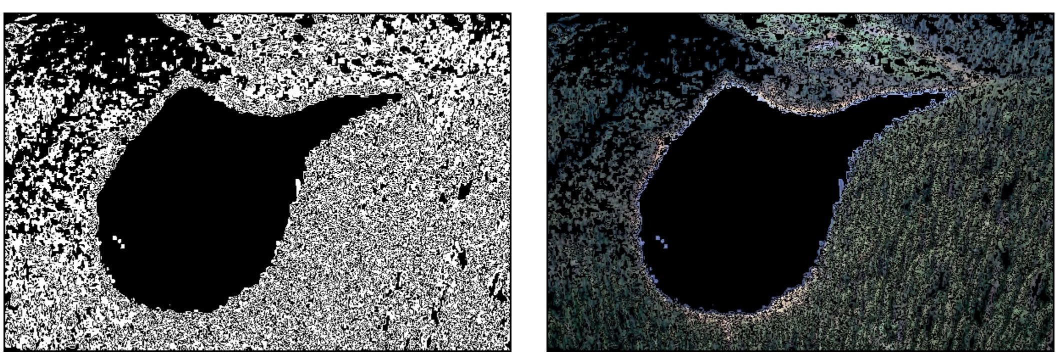

Figure 13: The foreground mask using the second channel of the image, and the corresponding foreground

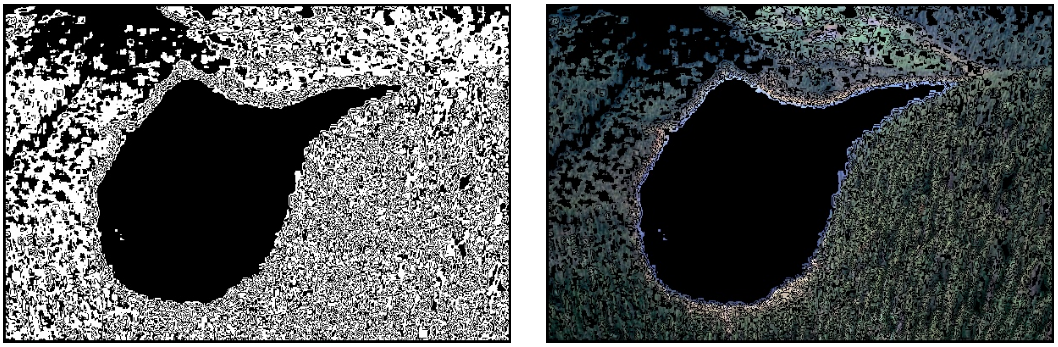

Figure 14: The foreground mask using the third channel of the image, and the corresponding foreground

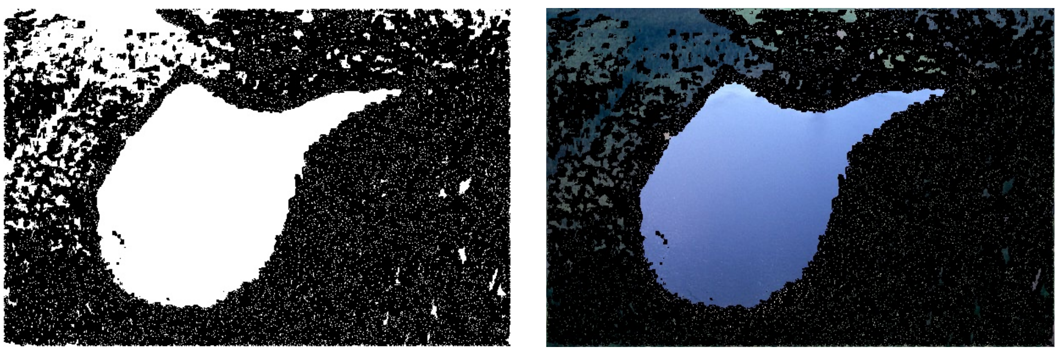

Figure 15: The overall mask using the masks of the three channels, and the corresponding foreground

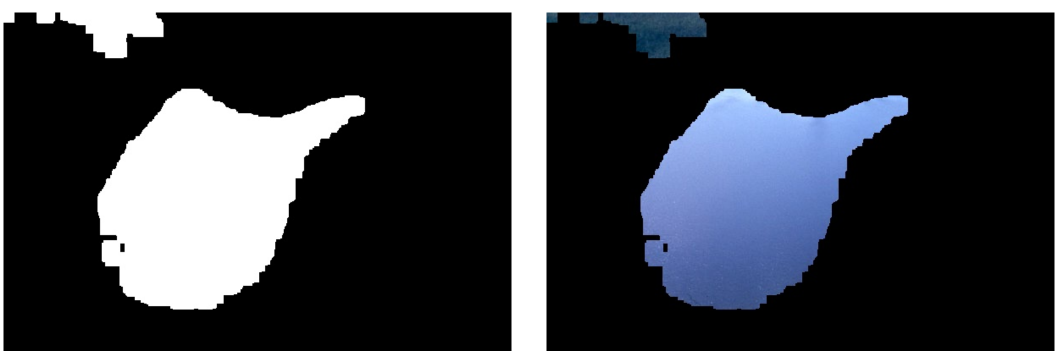

Figure 16: The final overall mask after removing the noise, and the corresponding foreground

### 7.2 Image 2: Tiger

Figure 17: The input image

#### 7.2.1 RGB Segmentation

Figure 18: The three color channels (BGR) of the image

Figure 19: The foreground mask using the blue channel of the image, and the corresponding foreground

Figure 20: The foreground mask using the green channel of the image, and the corresponding foreground

Figure 21: The foreground mask using the red channel of the image, and the corresponding foreground

Figure 22: The overall mask using the masks of the three channels, and the corresponding foreground

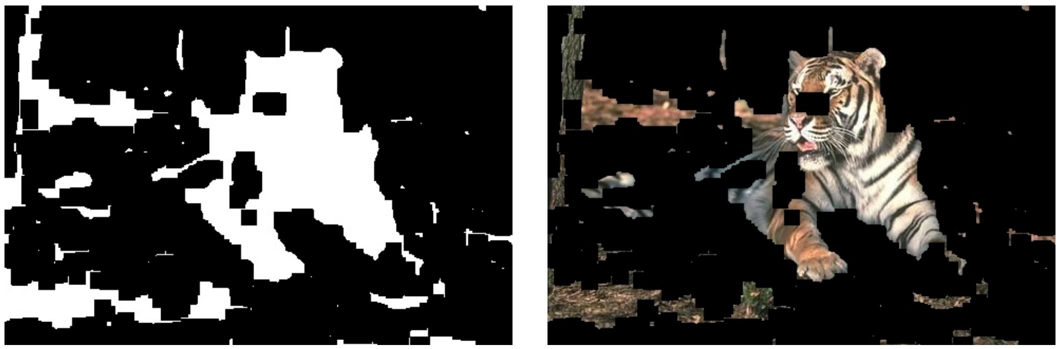

Figure 23: The final overall mask after removing the noise, and the corresponding foreground

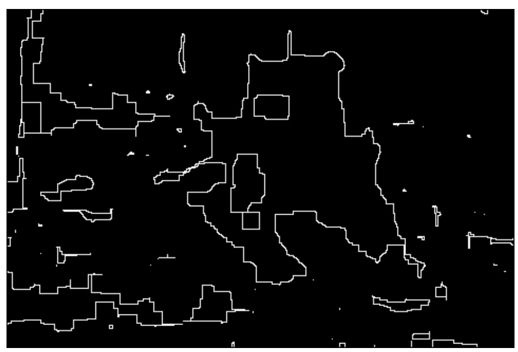

Figure 24: The final contour of the image

#### 7.2.2 Texture-based Segmentation

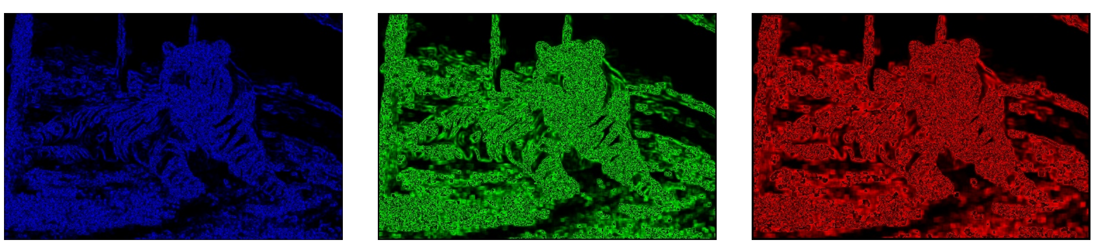

Figure 25: The three channels of the image representing the texture-based features (corresponding to the windows 3x3, 5x5, and 7x7 respectively)

Figure 26: The foreground mask using the first channel of the image, and the corresponding foreground

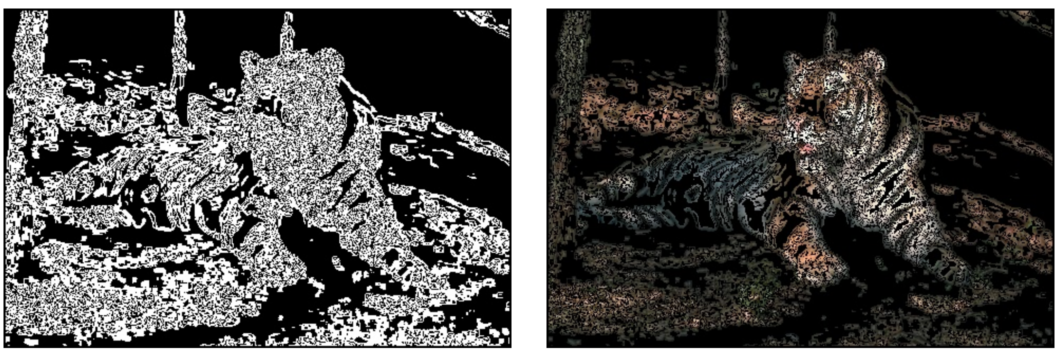

Figure 27: The foreground mask using the second channel of the image, and the corresponding foreground

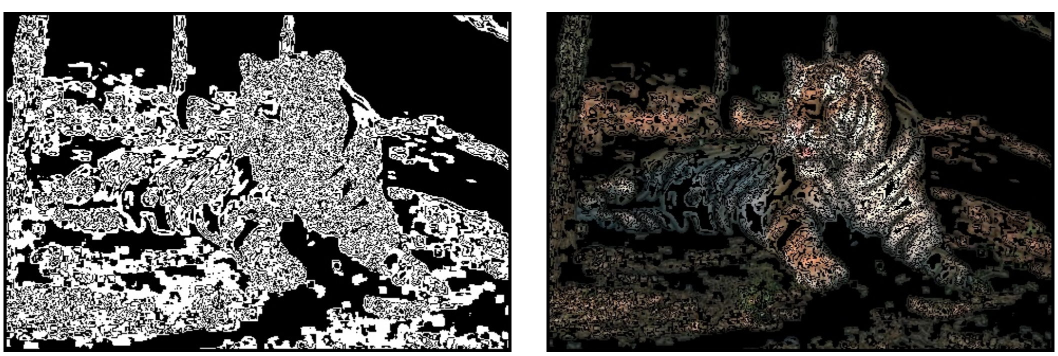

Figure 28: The foreground mask using the third channel of the image, and the corresponding foreground

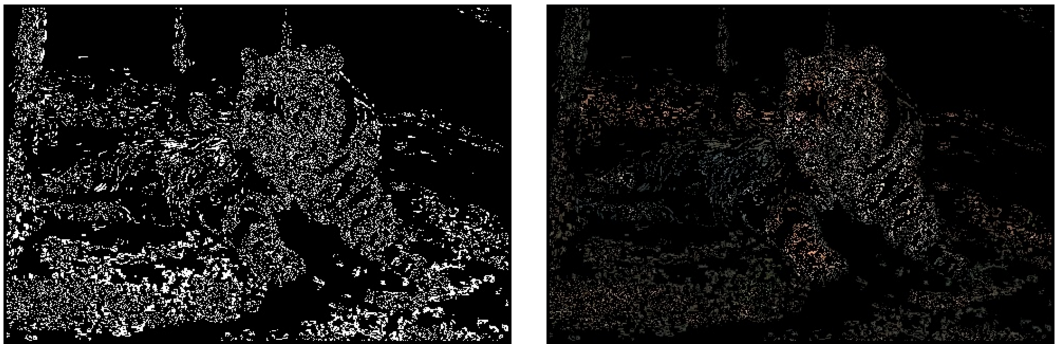

Figure 29: The overall mask using the masks of the three channels, and the corresponding foreground

Figure 30: The final overall mask after removing the noise, and the corresponding foreground

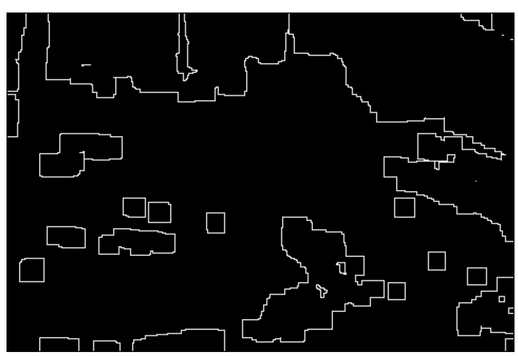

Figure 31: The final contour of the image

# 8 Source Code

The following is the entire Python source code.

```python
import cv2
import numpy as np


# The input image file name.
FILE_NAME = 'images/1.jpg'

# Whether to use texture features or not (RGB values).
```

```python
USE_TEXTURE = False


def main():

    # Read the input image.
    image = cv2.imread(FILE_NAME)
    cv2.imshow(FILE_NAME, image)

    if not USE_TEXTURE:
        s = ''

        # Segment using Otsu's algorithm on the image.
        mask = otsu_rgb(image, inverted_masks=[0, 1, 1],
                        iterations=[1, 4, 3], s=s)
    else:
        s = '_t'

        # Build an image with the texture-based features of the grayscale image.
        texture_image = get_texture_image(image)

        # Segment using Otsu's algorithm on the texture-based image.
        mask = otsu_rgb(texture_image, inverted_masks=[1, 1, 1],
                        iterations=[1, 1, 1], org_image=image, s=s)

    # Erosion then Dilation to remove the noises in the background.
    kernel = np.ones((17, 17), np.uint8)
    mask = cv2.erode(mask, kernel)
    mask = cv2.dilate(mask, kernel)

    # Dilation the Erosion to remove the noises in the foreground.
    kernel = np.ones((5, 5), np.uint8)
    mask = cv2.dilate(mask, kernel)
    mask = cv2.erode(mask, kernel)

    cv2.imshow(inject(FILE_NAME, 'filtered_mask{}'.format(s)), mask)
    cv2.imwrite(inject(FILE_NAME, 'filtered_mask{}'.format(s)), mask)
    cv2.imshow(inject(FILE_NAME, 'filtered_foreground{}'.format(s)),
               cv2.bitwise_and(image, image, mask=mask))
    cv2.imwrite(inject(FILE_NAME, 'filtered_foreground{}'.format(s)),
                cv2.bitwise_and(image, image, mask=mask))

    # Extract the contour
    contour = extract_contour(mask)
    cv2.imshow(inject(FILE_NAME, 'contour{}'.format(s)), contour)
    cv2.imwrite(inject(FILE_NAME, 'contour{}'.format(s)), contour)

    while not cv2.waitKey(50) & 0xFF == 27: pass
    cv2.destroyAllWindows()


def inject(image_name, suffix):
    return '{}_{}.{}'.format(image_name.split('.')[0], suffix,
                             image_name.split('.')[1])


def otsu_rgb(image, inverted_masks=[0, 0, 0],
             iterations=[1, 1, 1], org_image=None, s=''):

    # The original image is just used for the results.
    if org_image is None:
        org_image = image

    # Initialize the overall mask.
    overall_mask = np.ndarray((image.shape[0], image.shape[1]), np.uint8)
    overall_mask.fill(255)

```

```python
    # For each channel in the three color channels:
    for c in xrange(3):

        print 'Processing channel: ', c

        # The image representing channel c.
        channel_image = np.zeros_like(image)
        channel_image[:, :, c] = image[:, :, c]
        cv2.imshow(inject(FILE_NAME, 'channel_{}{}'.format(c, s)),
                   channel_image)
        cv2.imwrite(inject(FILE_NAME, 'channel_{}{}'.format(c, s)),
                    channel_image)

        # The mask of channel c.
        mask = None

        # For an arbitrary number of iterations: perform the segmentation
        # using Otsu's algorithm.
        for i in xrange(iterations[c]):
            # Perform the segmentation using Otsu's algorithm.
            mask = otsu(image[:, :, c], mask)

            # Save the results.
            cv2.imshow(inject(FILE_NAME, 'mask_{}_{}{}'.format(c, i, s)), mask)
            cv2.imwrite(inject(FILE_NAME, 'mask_{}_{}{}'.format(c, i, s)), mask)
            cv2.imshow(inject(FILE_NAME, 'foreground_{}_{}{}'.format(c, i, s)),
                       cv2.bitwise_and(org_image, org_image, mask=mask))
            cv2.imwrite(inject(FILE_NAME, 'foreground_{}_{}{}'.format(c, i, s)),
                       cv2.bitwise_and(org_image, org_image, mask=mask))

        # Calculate the overall mask as the logical and of masks after inverting
        # the masks that are indicated in inverted_masks.
        if inverted_masks[c] == 1:
            overall_mask = cv2.bitwise_and(overall_mask, cv2.bitwise_not(mask))
        else:
            overall_mask = cv2.bitwise_and(overall_mask, mask)

    # Save the results.
    cv2.imshow(inject(FILE_NAME, 'mask{}'.format(s)), overall_mask)
    cv2.imwrite(inject(FILE_NAME, 'mask{}'.format(s)), overall_mask)
    cv2.imshow(inject(FILE_NAME, 'foreground{}'.format(s)),
               cv2.bitwise_and(org_image, org_image, mask=overall_mask))
    cv2.imwrite(inject(FILE_NAME, 'foreground{}'.format(s)),
               cv2.bitwise_and(org_image, org_image, mask=overall_mask))

    return overall_mask


def otsu(image, mask=None):

    # The histogram of grayscale levels.
    histogram = [0] * 256

    # The total number of pixels in the mask.
    pixels_num = 0

    # The average grayscale value for the entire image (masked by the mask).
    mu_t = 0

    # Initialize the histogram based on the pixels of the image.
    for r in xrange(image.shape[0]):
        for c in xrange(image.shape[1]):
            if mask is None or mask[r][c] != 0:
                pixels_num += 1
                mu_t += image[r][c]
                histogram[image[r][c]] += 1

```

```
    # The average grayscale value for the entire image (masked by the mask).
    mu_t = float(mu_t) / pixels_num

    # The cumulative probability of pixels less than or equal level i.
    omega_i = 0

    # The cumulative average grayscale value less than or equal level i.
    mu_i = 0

    # The final chosen threshold.
    threshold = -1

    # The maximum sigma_b ^ 2 corresponding to the final chosen threshold.
    max_sigma_b = -1

    # For every grayscale level in the histogram:
    for i in xrange(256):

        # The number of pixels in the grayscale level i.
        n_i = histogram[i]

        # The probability of pixels in level i.
        p_i = n_i / float(pixels_num)

        # Update the cumulative probability of pixels less than or equal
        # level i, and the cumulative average grayscale value less than or
        # equal level i.
        omega_i += p_i
        mu_i += i * p_i

        # Ignore the very first levels and the very last levels that don't
        # contain any pixels. For these levels, sigma_b_i will cause division
        # by zero exception.
        if omega_i == 0 or omega_i == 1:
            continue

        # Update the between-class variance sigma_b ^ 2.
        sigma_b_i = (mu_t * omega_i - mu_i) ** 2 / (omega_i * (1 - omega_i))

        # Compare the between-class variance sigma_b ^ 2 to the maximum,
        # and update the best threshold.
        if sigma_b_i > max_sigma_b:
            threshold = i
            max_sigma_b = sigma_b_i

    print threshold

    # The image of the output mask.
    output_mask = np.zeros_like(image)

    if threshold == -1:
        return output_mask

    # For each pixel in the input image:
    for r in xrange(image.shape[0]):
        for c in xrange(image.shape[1]):
            # Set the corresponding output mask pixel to 1 if the pixel
            # values is greater than the threshold.
            if image[r][c] > threshold:
                output_mask[r][c] = 255

    return output_mask


def get_texture_image(color_image):

    # The grayscale version of the image.
```

```python
    image = cv2.cvtColor(color_image, cv2.COLOR_BGR2GRAY)

    # The texture-based image.
    texture_image = np.zeros_like(color_image)

    # The different window sizes used for the texture features.
    window_size = [3, 5, 7]

    # For each different window size:
    for i, w in enumerate(window_size):

        # Half of the window size.
        d = w / 2

        # For each pixel in the input image:
        for r in xrange(d, image.shape[0] - d):
            for c in xrange(d, image.shape[1] - d):
                # Calculate the variance of the pixel values in the window,
                # and store it in the texture_image.
                texture_image[r][c][i] = np.int(
                    np.var(image[r - d: r + d + 1, c - d: c + d + 1]))

    return texture_image


def extract_contour(image):

    # The contour image.
    contour = np.zeros_like(image)

    # For each pixel in the input image:
    for r in xrange(1, image.shape[0] - 1):
            for c in xrange(1, image.shape[1] - 1):

                # If the pixel is non-zero and there exist a zero pixel in
                # the surrounding 3x3 window, the pixel is considered on the
                # contour.
                if image[r][c] != 0 and \
                        np.min(image[r - 1: r + 2, c - 1: c + 2]) == 0:
                    contour[r][c] = 255

    return contour


if __name__ == "__main__":
    main()
```

Listing 1: The entire Python source code