Assignment 2
Graph Cut Segmentation
Due 11/1

Computer Vision - Fall 2011
New York University - Computer Science Department

This time we will improve the segmentation of the last assignment by trying to classify the pixels that had an invalid depth (0). You will have to implement three things:

1. The first step is to learn a model for the color distribution of the foreground and background as a mixture of gaussians, using the data from the regions you obtained by thresholding in the previous assignment.

2. With such a model in hands, we will then classify each pixel as belonging to one of the gaussian components of the foreground or background.

3. Note that with only pointwise information, we don’t guarantee any spatial coherence in the foreground/background classification. The final step will be to incorporate a spatial coherence term to an energy function that you will be able to minimize using the graph cut algorithm. The original classification by thresholding will be maintained, you will only classify the pixels that had undefined depth.

Again, before writing any code, compile and run the template code we provide. Then add the code from your previous assignment to fill in the old blanks (if you haven’t changed anything else, you should only have to copy your old gmm_segmentation.{cpp,h}, histogram.h, kmeans_segmentation.{cpp,h}, threshold.{cpp,h} files to the src folder. If everything is running like in the previous assignment, start with this one.

1 Modeling the Foreground and Background Color Distributions with GMMs

First, for initialization purposes, you should implement the k-means algorithm, as in the previous assignment, but this time in color space instead of depth. Implement the k_means_color function in kmeans_color.cpp. Note that in this function you are given a region where to compute the k-means. Since we
have a separate model for the foreground and background, a mask will specify the region you should consider.

Then implement the `gmm_color` function in `gmm_color.cpp`. You are given a mask of the pixels defining the region where you have to estimate the model parameters (foreground or background), as well as an assignment of pixels to each component of the gaussian mixture, and you must return the mean and covariance of each gaussian, and the mixing coefficients.

Using $\alpha \in \{0, 1\}$ to denote whether a pixel belongs to the background ($\alpha = 0$) or foreground ($\alpha = 1$), the color $(z = (r, g, b))$ distribution is described by:

$$p(z|\alpha, \pi, \mu, \Sigma) = \sum_k \pi_{\alpha k} N(z|\mu_{\alpha k}, \Sigma_{\alpha k})$$

The gaussian parameters can be estimated by the sample mean and covariance of the foreground and background regions:

$$\mu_{\alpha k} = \frac{1}{N} \sum_i z_i$$

$$\Sigma_{\alpha k} = \frac{1}{N-1} \sum_i (z_i - \mu_{\alpha k})(z_i - \mu_{\alpha k})^T$$

You can approximate the mixing coefficients simply as the ratio of the size of each component, provided by the clustering given as input. If you are done with the whole assignment and have extra time, try implementing the full EM algorithm for comparison.

The code provided uses 4 components for each GMM as default, but feel free to experiment with the number of components and see what works for you.

## 2 Component Assignment

Now that we have a model for the foreground and background color distributions, we will assign each pixel to the component with highest probability. The probability that pixel $i$, with color $z_i = (r_i, g_i, b_i)$, belongs to the component $k_i$ is:

$$p(k_i|z_i, \alpha_i, \pi, \mu, \Sigma) \propto \pi_{\alpha_i k_i} N(z_i|\mu_{\alpha_i k_i}, \Sigma_{\alpha_i k_i})$$

Equivalently, we seek the component $k_i$ that minimizes

$$-\log p(k_i|z_i, \pi, \mu, \Sigma)$$

which, in turn, is equivalent to minimizing the energy:

$$D_i(\alpha_i, k_i, z_i, \pi, \mu, \Sigma) = -\log \pi_{\alpha_i k_i} + \frac{1}{2} (z_i - \mu_{\alpha_i k_i})^T \Sigma_{\alpha_i k_i}^{-1} (z_i - \mu_{\alpha_i k_i}) + \frac{1}{2} \log \det \Sigma_{\alpha_i k_i}$$

in the parameter $k_i$ (the other parameters are held fixed).
Implement the function `assign_gmm_component` in `mincut_segmentation.cpp`, where, given the region $\alpha_i$ (foreground/background) each pixel belongs to, assigns it to the GMM component with lowest energy (remember that we have two gaussian mixtures, one for the foreground, and another one for the background, so the choice of which one to use depends on the assignment of the pixel as foreground or background). Note that we are doing hard assignments here instead of the soft-weights (as in EM) in order to simplify and speed up the computations.

Once you have implemented this function, there is an option in the left pane menu that lets you see the component each pixel was assigned to. The image will color each pixel with the mean of the component it was assigned to (so this also works as a color quantization algorithm).

3 Adding Spatial Coherence with Mincut

The last step is to add spatial coherence to the segmentation. For each pair of neighboring pixels $(i,j)$, we will add a spatial term to the energy of the form:

$$V(\alpha_i, \alpha_j) = [\alpha_i \neq \alpha_j] e^{-\beta||z_i - z_j||^2}$$

where $[P]$ is the indicator function for predicate $P$. This term penalizes neighboring pixels that had similar colors but were assigned to different components (background/foreground), trying to clump together similar pixels.

The full energy then becomes:

$$E(\alpha, k) = \sum_i D_i(\alpha_i, k_i, z_i, \pi, \mu, \Sigma) + \gamma \sum_{(i,j)} V(\alpha_i, \alpha_j)$$

which you can minimize in $\alpha$ (foreground/background assignment), while holding the other parameters fixed, using the mincut algorithm.

In practice, for the graph construction, this means that the edges linking each pixel $(u)$ to the source $(S)$ and sink $(T)$ will have weights:

$$w(u, S) = D_i(\alpha_i = 0, k_i, z_i, \pi, \mu)$$
$$w(u, T) = D_i(\alpha_i = 1, k_i, z_i, \pi, \mu).$$

Note, however, that some of the pixels already have a foreground/background assignment, so you must fix their $\alpha$ values somehow (remember, you are only solving for the pixels that had invalid depth, and, thus, could not be segmented by thresholding). In order to accomplish this, if you want to force pixel $u$ to belong to the foreground, you can simply set:

$$w(u, S) = 0$$
$$w(u, T) = \infty.$$
and, similarly, to force pixel $u$ to belong to the background, you can set

$$ w(u, S) = \infty $$
$$ w(u, T) = 0 $$

Edges linking neighboring pixels $i$ and $j$ will have weights:

$$ w(i, j) = \gamma e^{-\beta ||z_i - z_j||^2} $$

Experiment with the parameter $\gamma$ and see what works for you. $0 - 100$ seems to be a good range, where setting $\gamma = 0$ will take only the data term into account. See how this compares to when you add the spatial smoothness term. Also, try using 4 and 8 neighbours and see how the choice affects the result.

Alternating between the minimization in $\alpha$ using mincut, and in $k$, as described in the previous section, guarantees the energy to decrease, and converge at least to a local minima. However, don’t iterate too much for a single frame, or your program will be very slow. Try to find a balance between improvement in the result of the segmentation and real-time responsiveness, but just a couple of iterations should be fine. If you want to speed up the algorithm, one suggestion is for you to create only nodes that have a neighbour with undefined depth, since the remaining pixels won’t affect the result, only increase computation and memory requirements.

The coordinate minimization described in this section should go in the `mincut_segmentation` function in `mincut_segmentation.cpp`. Once you are done, the popup menu on the right pane will allow you to alternate between the threshold segmentation and the full mincut segmentation. Moreover, when color coding the segmentation in the left pane, you should be able to tell, from the pixels that had an invalid depth, which ones where assigned to the foreground and which ones were assigned to the background.

To actually solve the mincut problem, we’ll be using the maxflow implementation of Vladimir Kolmogorov, provided in:

http://pub.ist.ac.at/~vnk/software.html

His code is already included with the code provided for the assignment. For an example of how to build a graph and run the maxflow algorithm using Kolmogorov’s code, there is a simple example in maxflow/README.txt provided by him.

4 What to Turn In

Show up with your group ready to show your program running in your allotted time.