From: Robert_T_Collins@IUS5.IUS.cs.cmu.edu
Sent: Monday, October 04, 1999 8:56 AM
To: Steve Seitz
Cc: rcollins@cs.cmu.edu; Robert_T_Collins@IUS5.IUS.cs.cmu.edu
Subject: Re: vanishing points

Hi Steve. All my code was in lisp, but it should be very easy for the students to implement, depending on what exactly you want to do. Grouping an arbitrary set of lines in the image into sets that converge to a vanishing point requires a fair amount of code. If you are already given the lines that converge, and just want to compute the vanishing point, that is easier.

The basic idea (for the easy part) is the following:

1) specify each line's endpoints e1 and e2 in homogeneous coordinates
   e1 = (x1_i , y1_i, w)
   e2 = (x2_i , y2_i, w)
  where the constant w is often taken as 1, but you may want to use
  something else for better numerical conditioning (more on this later).

2) compute a homogenous coordinate vector representing the line
   as the cross product of its two endpoints
      (a_i,b_i,c_i) = e1 X e2
   note that this resulting vector is just the parameters of
   the equation a_i x + b_i y + c_i = 0 of the 2D infinite line
   passing through the two endpoints

3) if you only have two lines, l1 and l2, you can compute a homogeneous
   coordinate vector V representing their point of intersection as
   the cross product of these two line vectors
      V = l1 X l2
   scaling V so that the last coordinate is 1, i.e. (Vx, Vy, 1),
   and you have Vx and Vy as the point in the image that is
   the vanishing point. It is better to leave the vanishing
   point as a homogeneous coordinate vector, however, because the
   vanishing point could be very far off the image, or even at
   infinity (in which case the third component of V is 0, and
   you get a divide by zero when you try to scale).

4) if you have n lines l1, l2, ..., ln, you can get the "best_fit"
   vanishing point as follows:

   4a) form the 3x3 "second moment" matrix M as

$$M = \text{sum} \begin{bmatrix} a_i a_i & a_i b_i & a_i c_i \\ a_i b_i & b_i b_i & b_i c_i \\ a_i c_i & b_i c_i & c_i c_i \end{bmatrix}$$

       where the sum is taken for i = 1 to n. Note that M is
       a symmetric matrix

   4b) perform an eigendecomposition of M, using the Jacobi
       method, from numerical recipes in C, for example. (Jacobi
       method is good for finding eigenvalues of a symmetric matrix)
       [I can supply the Jacobi matrix code from NR in C if you like]

   4c) the eigenvector associated with the smallest eigenvalue is
       the vanishing point vector V

One word about numerical conditioning is in order. If you just use image pixel coordinates directly, the problem will be horribly poorly conditioned. There is a good paper by Hartley on this topic, title something like "In defense of the Eight-point algorithm." If your lines are spread out throughout the image, you can get approximately the kind of well-conditioned approach he mentions by doing the following.

  1) translate by (-imageX/2, -imageY/2) so that (0,0) is in the

```
     center of the image [I think Hartley's conditioning method would
     take the center of mass of the line endpoints as the origin].

  2) Scale coordinates so that the magnitude of all image point
     homogeneous coordinates is roughly 1 [Hartley says how to do
     this exactly, in his paper].  I used to do it approximately
     by just setting the constant w, mentioned during the first
     step of this algorithm, to be equal to half the image size
     in pixels.  So if you had a 256x256 image, then you would
     have w = 128.  IF the image width and height aren't the same,
     just take the average or something, before dividing by two.


--Bob
```