ECE432: Homework 1
Implementing Lucas & Kanade

Sven Olsen

Compiled at 4:52 AM on June 29, 2005

1 Equations

The equations that I'm using are equivalent to those given in the notes, but in order to make my implementation faster I've defined my terms slightly differently. Most significantly, I never explicitly define a weight matrix.

The problem we are trying to solve is:

$$\min_v \; E(v) = \sum_{i \in \Omega} (w_i \nabla I \cdot v + w_i \frac{\partial I}{\partial t})^2.$$

Which is of the form

$$\min_x \; (Ax - b)^T (Ax - b) \tag{1}$$

where

$$A = \begin{bmatrix} w_1 \frac{\partial I_1}{\partial x} & w_1 \frac{\partial I_1}{\partial y} \\ \vdots & \\ w_N \frac{\partial I_N}{\partial x} & w_N \frac{\partial I_N}{\partial y} \end{bmatrix}_{N \times 2}, \qquad b = \begin{bmatrix} \frac{-\partial I_1}{\partial t} \\ \vdots \\ \frac{-\partial I_N}{\partial t} \end{bmatrix}_{N \times 1}.$$

Equation (1) goes to:

$$\min_x \; (x^T A^T A x - 2(A^T b)^T x + b^T b)$$

This is a quadratic with a symmetric positive semi-definite Hessian[1], therefore minimizing it is equivalent to satisfying the linear equation:

$$A^T A x = A^T b.$$

Doing out the multiplications, we see that:

$$A^T A = \begin{bmatrix} \sum_{i \in \Omega} (w_i^2 \frac{\partial I_i}{\partial x}^2) & \sum_{i \in \Omega} (w_i^2 \frac{\partial I_i}{\partial x} \frac{\partial I_i}{\partial y}) \\ \sum_{i \in \Omega} (w_i^2 \frac{\partial I_i}{\partial x} \frac{\partial I_i}{\partial y}) & \sum_{i \in \Omega} (w_i^2 \frac{\partial I_i}{\partial y}^2) \end{bmatrix}_{2 \times 2}, \qquad A^T b = \begin{bmatrix} -\sum_{i \in \Omega} (w_i \frac{\partial I_i}{\partial x} \frac{\partial I_i}{\partial t}) \\ -\sum_{i \in \Omega} (w_i \frac{\partial I_i}{\partial y} \frac{\partial I_i}{\partial t}) \end{bmatrix}_{2 \times 1}.$$

[1] $A^T A$ is always a symmetric positive semi-definite matrix, regardless of the form of A.

Thus,

$$v_x = \frac{\frac{-XT \times Y^2}{XY} + YT}{\frac{Y^2X^2}{XY} - XY}, \qquad v_y = \frac{-XT - XY \times v_x}{Y^2} \tag{2}$$

where

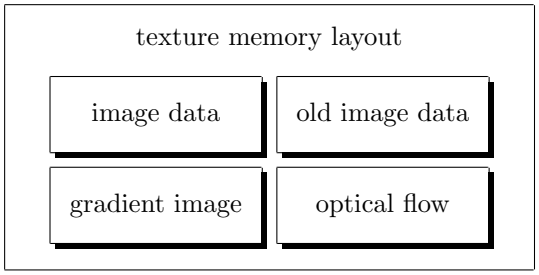
$$X^2 = \sum_{i \in \Omega} (w_i \frac{\partial I_i}{\partial x})^2, \qquad Y^2 = \sum_{i \in \Omega} (w_i \frac{\partial I_i}{\partial y})^2, \qquad XY = \sum_{i \in \Omega} (w_i^2 \frac{\partial I_i}{\partial x} \frac{\partial I_i}{\partial y}),$$

$$XT = \sum_{i \in \Omega} (w_i \frac{\partial I_i}{\partial x} \frac{\partial I_i}{\partial t}), \qquad YT = \sum_{i \in \Omega} (w_i \frac{\partial I_i}{\partial y} \frac{\partial I_i}{\partial t}).$$

2 Implementation

The algorithms are implemented on an Nvidia nv40 GPU, and nearly all operations are performed in parallel.

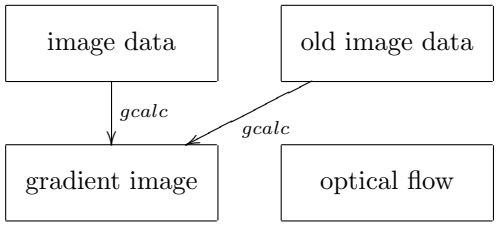
The texture memory is arranged on the card as follows:



First a grayscale version of each frame is created using the formula

$$I_i = .30r_i + .59g_i + .11b_i$$

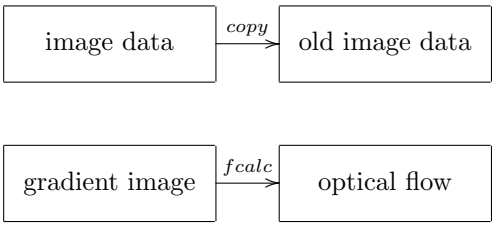
Then a "gradient image" is created, for which the values of each pixel are $(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}, \frac{\partial I}{\partial t})$.



$\frac{\partial I}{\partial x}$ and $\frac{\partial I}{\partial y}$ are approximated using the Sobel matrices, while $\frac{\partial I}{\partial t}$ is calculated using the difference with the previous image.

Once the gradient data has been found, A^TA and A^Tb are calculated for each pixel, and the implied linear system solved using (2). A final operation

copies the current image data over the old image data.



3 Results

For each dataset I've printed out the optical flow field viewed using vectors, as well as bitmap captures of the gradient image and optical flow image. The bitmap captures only represent a truncated potion of the floating point images that actually exist on the card, but they're interesting all the same. The format of the data is:

`<dataset_name>-s<sigma>-g<radius>.<image number>.bmp`

Gradient and optical flow color bitmap images are prefixed with 'c' and 'g' respectively. A sigma of 0 implies that the image used a constant weighted neighborhood, otherwise, sigma characterizes a Gaussian weighting scheme. Ω is defined to be a square with the given radius, centered on the current pixel.

As is apparent from the results, constant weighted neighborhoods perform very poorly, and hardly ever produce noticeable optical flow fields (this performance is actually worse than I expected, I almost suspect that there is a bug in my implementation of the constant case). Larger neighborhoods with broader Gaussians lead to smoother, more regular data, exactly as one would expect.

I have also hooked the algorithms up to a webcam, and generated fields in realtime[2] from the captured video, but unfortunately such results cannot be easily saved, because the cost of outputting the data ruins the framerate.

I have yet to get around to either multiresolution experiments or Horn-Schunck.

3.1 Performance

The total time required to process the 20 images in the 'office' dataset using $r = 10$ and $\sigma = 6$ is 2.91 seconds (though when the cost of creating the debugging images is included, the time increases to 3.63 seconds). The processing times for $r = 1$, 2, 5, 10, and 20 are 0.43, 0.53, 1.05, 2.91, and 9.22 seconds respectively. While I expect that my current GPU implementation will scale better with the size of the input images than a cpu implementation, it will not scale better with r (because the accumulations of $A^T A$ and $A^T b$ are performed using a for loop). However, assuming sufficient texture memory (sizeof(Image) $\times$ sizeof(Ω)), the

[2]Right now, the system only runs at about 10 fps, but it's a long way from optimized.

accumulations could also be parallelized via sum-reduction techniques. Unfortunately, such a highly parallel implementation would be quite memory intensive, and thus difficult to implement given the current Nvidia drivers and OpenGL extensions.