Applications of Pyramid Structures to Multiscale Optical Flow

Ted Camus

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-90-09
August 16, 1990

Applications of Pyramid Structures to Multiscale Optical Flow

Ted Camus
Brown University

August 16, 1990

An important problem in machine vision is the description of motion of a 3-D world as projected on a 2-D retina (optical flow). Most current optical flow algorithms are not suited for practical implementations such as tracking because they either require massively parallel supercomputers, highly specialized hardware, or up to an hour with a scientific workstation. This paper solves this problem in three ways. First, we provide a fast serial implementation for a recent optical flow algorithm that can run up to 10 frames per second for practical image sizes on a conventional workstation. Second, an efficient method for abstracting a high-level description of the motion in an image is described, along with an application for tracking. Finally, we provide a heuristic that speeds up the original optical flow algorithm by a factor of 3 using a pyramid architecture.

Applications of Pyramid Structures to Multiscale Optical Flow

Ted Camus
Brown University *

August 28, 1990

1 Introduction

An important problem in machine vision is the interpretation of a 3-D world via a 2-D sensor input. One particular instance of this problem is that of optical flow, the description of motion in images projected on a 2-D retina. An object moving in 3-D space has a three dimensional velocity vector field $W(x, y, z)$. This 3-D field is projected onto the retina of an observer as the 2-D field $W_p(x, y)$. What is observed by the retina is the change in a point's intensity value $\Delta E(x, y)$ for some discrete time interval Δt. The optical flow is a vector field describing this change. In general it is not possible to determine the correct optical flow field given a pair of successive image frames. If certain assumptions are enforced, however, then the problem becomes well-posed, and can be satisfactorily solved in most cases. This paper reviews a recent optical flow algorithm [BLP89a], which previously could only be run on a massively parallel machine (such as a Connection Machine) in anything close to real time, along with a fast serial implementation with real-time potential. We describe a simple but powerful abstraction mechanism, based on a pyramid architecture, that can quickly extract the information from an optical flow field which is necessary for real-time tracking. Finally we provide a heuristic for providing a multiple time speedup by utilizing similar pyramid structures.

We begin with the ideal case for optical flow computation. Here we have only ambient illumination, i.e. lighting is uniform across the object being viewed. All irradiance (image brightness) is due to the reflectance of the object being viewed, and is not dependent on the position of the camera or light source. We also assume that our objects are Lambertian surfaces, so for example a uniform sphere rotating in a constant lighting would not show any apparent motion. We need these assumptions to justify the statement that all

*Box 1910, Brown University, Providence, R.I. 02912. tac@cs.brown.edu

changes in intensity values are due to the motion of the object under view. In practice, they are generally adhered to as long as the motion of any objects is relatively small compared to the size of the objects.

With these assumptions we can state that the brightness of a given point is constant as it moves, $dE/dt = 0$, where E is the point's irradiance. From [HS81] we have

$$\frac{\partial E}{\partial x}\frac{dx}{dt} + \frac{\partial E}{\partial y}\frac{dy}{dt} + \frac{\partial E}{\partial t} = 0,$$

i.e. all changes in intensity values are due to motion. Since we have a single linear equation with two unknowns (∂E/∂t in the image plane can be measured), this gives us an ill-posed problem, thus we do not have sufficient information to give the desired optical flow. (This phenomenon is also responsible for the aperture problem; see [NS88].) For example, consider a field whose intensity increases linearly with x but exponentially with y. Given just ∂E/∂t, we would not be able to discriminate between linear motion in x or logarithmic motion in y; further constraints are needed. The most common constraint added is that of smoothness. Since we are generally concerned with the flow of rigid-bodied objects, it is usually the case that any given pixel has the same velocity as those of its neighbors. The addition of this constraint can be manifested either locally or globally as a form of regularization [LV89]. Of course, this additional constraint does not hold at discontinuities, such as edges. For algorithms that implement the smoothness constraint globally, special measures must be taken to avoid blending occluding objects together. Other optical flow algorithms, such as the one discussed here, implement the smoothness constraint implicitly. This is described in the next section.

2 A Robust Optical Flow Algorithm

In this section we will review a recent optical flow algorithm, that has been shown to be largely consistent with human psychophysics [BLP89b]. The basic optical flow techniques used later in this paper are derived from this algorithm, and are applied towards machine vision oriented problems.

Each pixel [x,y] in the image is given two degrees of freedom of motion in the image plane, one for x displacement and another for y displacement. Each of these possible displacements shall be limited to a given range for any problem we discuss. We will assume that the maximum possible displacements for both x and y will be the same, indicated by k, although in general they can be different. Thus, the size of the window

over which our pixel is allowed to move is comprised of (2k+1)(2k+1) pixels. To simplify our notation, we will represent (2k+1) by 'δ'. As an example, consider figure 2. Assume the pixel in question is at [2,2]. If k equaled 1, then the pixel would be allowed to move anywhere within the box delimited by the corners at [1,1],[1,3],[3,3],[3,1], for example to [3,3], representing a (Δx,Δy) of (+1,+1). There are $\delta * \delta$ = 3*3 possible displacements allowed for the pixel. If k equaled 2, there would be 5*5 possible displacements, for example (-1,+2). The actual value of δ depends on the expected velocities of the pixels in the image plane, which itself is a function of the objects' velocities, distance from the camera, the focal length of the camera, and the elapsed time between the two frames.

Given our assumption of locally homogeneous motion (which generally holds except for non-rigid objects and object boundaries) and fronto-parallel motion (a close approximation for limited motions) it should be the case that if a given pixel moves by (Δx,Δy) then the surrounding pixels also move by (Δx,Δy). These surrounding pixels are assumed to be in a square neighborhood, or window, of size ν centered around the given pixel. For example, if our neighborhood patch consisted of the pixel in question $\pm$3 pixels along each axis (x and y) then we would have $\nu = 7$ and a square patch of 49 pixels total. So we want the correct motion of the $\nu * \nu$ patch of pixels that is centered at [x,y], rather than considering just the pixel itself. The motion for the pixel at [x,y] is defined to be the determined motion of the patch centered at [x,y], out of $\delta * \delta$ possible displacements. It is this definition that gives the algorithm its smoothness properties and its noise-resistance [LBP88].

We determine the correct motion of the patch of pixels by simulating the motion of the patch for each possible displacement of [x,y] and considering a match strength for each displacement. If ϕ represents a matching function which returns a value proportional to the correlation of two given pixels, then the match strength M(x,y;u,w) for a point [x,y] and displacement (u,w) is calculated by taking the sum of the match values between each pixel in the displaced patch P_ν in the first image and the corresponding pixel in the actual patch in the second image :

$$\forall u, w : M(x, y; u, w) = \sum \phi(E_1(i, j) - E_2(i + u, j + w)), (i, j) \epsilon P_\nu$$

An appropriate function for ϕ is the the absolute difference between the two pixels' intensity values. Another possibility for ϕ is the squared difference of their respective intensity values. In these cases a lower value

indicates a better match. It is also possible to perform matches using primitives other than intensity values. For example, one could match features such as edges. We could additionally discriminate based on the intensity gradient of the edge, i.e. the slope of the zero crossing. Given two edge diagrams, where each pixel had a sign indicating a positive gradient, negative gradient, or no edge, then ϕ would equal one iff the two pixels had the same sign.

In an ideal situation, where we have only fronto-parallel motion, no noise, no lighting changes, and a translation of an integer number of pixels (within our given range) we would expect the pixels of the two patches to match perfectly. The actual motion of the pixel is taken to be that of the particular displacement, out of $\delta * \delta$ possible displacements, with the maximum neighborhood match strength (equivalently minimum patch difference). With a large neighborhood, ties are relatively rare and are decided arbitrarily. Take for example figure 3. We have our pixel at [2,2] in the first image which is considering a (+1,+1) displacement to position [3,3] in the second image. We consider a neighborhood size of $\nu = 3$ for our example. The lower left hand box, labeled A, represents the patch of pixels (centered at [2,2]) which translates to the position of the box centered at [3,3], labeled B, if it were to undergo a displacement of (+1,+1). We now must calculate the match strength of the displacement. To do this, we take the sum of the absolute differences between each of the pixels in box A in image 1 and each of the pixels in box B in image 2. Notice again that the position of box B is actually the position of box A added to the simulated motion of (+1,+1). This only gives the match strength of one of the $\delta * \delta$ displacements. If the chosen displacement is the correct one all pixel values will match and the difference between the two patches will be minimal. By default the match strength will be at its maximum for exactly this displacement. (Note also that since we can easily record the top few match strengths, or those that are above a given threshold, this system can support the notion of motion transparency.) Clearly, we expect there to be little difference between the patch for the correct motion and the patch for its discretized approximation. This is repeated for each pixel in the image, independently of the other pixels, with the exception that motion is not computed along a $\delta + \nu$ border (since the matching operator would have to access points that are not available in the image itself). This independence of one pixel's chosen displacement from all other pixels' displacements motivates massive parallel implementations such as the one implemented on the connection machine [BPL89b].

The exact value of the neighborhood window size ν depends on the size of the expected areas of rigid homogeneous motion in the image plane. This is equivalent to the expected distance between discontinuities of moving objects in the image, for example the width of a moving person. The larger the size of the neighborhood, the smoother (and generally more accurate) the result, but it cannot be too large, or the motion will be taken to be that of the static background. It turns out that this neighborhood size can be set arbitrarily, as large as is practical without significantly affecting the execution time on a serial machine. This is a surprising result, as intuitively it would seem that larger window sizes would take more time to compute. This result is detailed in the next section.

3 A Fast Serial Implementation

We present a very fast serial implementation that can run in real-time, on a sun workstation, for practical tracking problems. In particular, the neighborhood windows over which the match strengths are computed can be calculated in constant time.

In order to show this we consider figures 3 and 4, which show a displacement of Δx,Δy = (+1,+1), for the points [2,2] and [1,2] respectively. The neighborhood windows for both are shown. Notice that the windows overlap by a factor of 2/3. In general, there is an overlap of $(\nu - 1)/\nu$, since the two windows are coincident except that they are shifted by a single x coordinate, corresponding to one out of ν column sums in the neighborhood window. As ν increases, so does the smoothness, since the windows become more similar [LBP88]. The computation for the match strength of any given displacement can be reduced due to this overlap. The match strength for the displacement +1,+1 for point [2,2] is equal to the match strength for the same displacement for point [1,2], except that this value incorrectly includes the values at [1,i] for i=2,3,4 and omits the values at [4,i] for i=2,3,4. Thus we must also calculate these two "column sum" portions of the match strength neighborhood windows, subtracting the former and adding the latter. Note however that the column sum calculated at [4,i] for i=2,3,4 which was added in for point [2,2], displacement +1,+1 will be similarly subtracted for the same displacement at point [5,2]. Since each neighborhood window consists of ν column sums of ν matches each, by saving the last ν column sums we have reduced the calculation of $\nu * \nu$ differences (the size of the entire neighborhood window) to only ν, that of the rightmost column sum. This

is shown in figure 6. Each displacement under consideration may be processed in this manner. In doing so we have reduced a $O(n^2 * \delta^2 * \nu^2)$ algorithm to a $O(n^2 * \delta^2 * \nu)$ one. However, it does require extra memory. For each of $\delta * \delta$ displacements, we need to save the previous ν column sums, along with the neighborhood window match strengths of the previous pixel. This is $O(\delta * \delta * \nu) + O(\delta * \delta)$ memory.

We have just reduced the horizontal dimension of the window summation problem from ν to 1. We will now show how to reduce the vertical dimension from ν to 1. Consider figure 3 and 5. These show, as before, the displacement $\Delta x,\Delta y = +1,+1$ but for points [2,2] and the one below it, [2,1]. We have just stated that the neighborhood window calculation for displacement +1,+1 at [2,2] involves just the column sum at [4,i] i=2,3,4 (plus the saved previous window value, minus the leftmost column sum) but we are not satisfied with that. The very column sum needed was "almost" calculated by [2,1], the pixel below it, in the same manner that the window was almost calculated by the pixel to the left of it. In particular, the value for the column sum [4,i] i=2,3,4 for displacement +1,+1 is the column sum [4,i] i=1,2,3 for displacement +1,+1, previously calculated for point [2,1], except that this value incorrectly includes the match at [4,1] and omits the match at [4,4]. So this column sum consisting of ν calculations can be computed in only two computations using the column sum calculated one row below. This is visualized in figure 7. The key is that we only need to compute the boundaries of each window, as the bulk of it has already been done by our neighbors. In summary, we have reduced a $\nu * \nu$ calculation, where ν is the length of the neighborhood window, to only 4 calculations: (saved previous window value) - (saved leftmost column sum) + (saved right column sum of pixel below) - (bottom pixel match of right column sum of pixel below) + (missing top right pixel match).

There are four distinct areas of an image that must be processed in different ways. The very first point, [0,0], must have all of its column sums determined explicitly, as it has no pre-calculated neighbors. Every other pixel in the first row has a left neighbor, but no neighbor below it, thus it must be processed as per figure 6. The first pixel in each row (other than the first row) has neighbors below it, but no left neighbor. However, its column sums may be calculated as per figure 8. Finally, every other pixel may take full advantage of its neighbors by calculating its column sums as per figure 7. An algorithmic definition may be found in Figure 1. Here it is shown that the algorithm is

$$O((n-1)^2 * \delta^2)$$

in time and

$$O((n-1)*\delta^2)$$

in memory. For an image size of 256*256, and a displacement range of +/- 8 pixels, this memory is approximately equal to one of the two input images.

It turns out that the individual neighborhood summations implement a global low-pass box filter [for example, see BM86], which we will describe briefly. Consider figure 9, a five by five matrix of variables (generic values). We wish to replace each entry in this array with the summation of the 3x3 "box" of values centered at the original entry. Note that the 3x3 box summations are not defined at the boundaries, since they would involve values not available in the original 5x5 array. To do this, we first perform a set of partial sums for each row. The first partial sum in each row is calculated normally, i.e. we simply sum up the first three elements (the length of one row of the resulting box). All the following partial sums may be calculated by subtracting the leftmost component of the previous sum and adding the next right single element. Note that this is just two computations, independent of the ultimate size of the box sum (the key to its performance). This is done independently for each row, and is shown in figure 10. This exact same operation is then performed column-wise on the previous row-sums to produce the desired box-sums, as in figure 11.

This approach also requires extra memory, as shown in the figures. In practice these extra arrays are not constructed explicitly. A better implementation would process one row of partial sums at a time, saving the last ν rows of row-sums, then process one more box sum (one row down) for each column. This would require either $n*\delta*\delta*\nu$ memory locations to save the last ν rows, unless we reserve an array to keep the current maximums (out of $\delta*\delta$ iterations), in which case we need order $n*n+n*\nu$ locations. (In this case there is an extra factor of 2, since we will need more than the typical 8-bit values used to represent the input image itself.) We also must perform the filter in two passes through the image, rather than in one pass as in the previous approach, which may have some implications for hardware implementations. Although the previous algorithm appears superior in these respects, the box filter is markedly simpler, with much tighter inner loops that can be optimized in assembly language. It also has excellent locality of reference, even if the entire array of partial sums is maintained, which offsets the extra memory required (for most image

parameters). Thus it generally has the performance advantage.

These algorithms, as described, has potential for running in real time on workstations such as a SUN Sparcstation, as indicated in Table 1). Since the arithmetic involves nothing more than simple integer additions, subtractions and logical shifts, it is ideally suited to modern RISC processors. It has shown to be robust with respect to dozens of pairs of real world images, including many with deliberately bad (either dim or overly bright) lighting, using inexpensive cameras. This is fortunate, since an optical flow algorithm cannot make use of the special lighting techniques possible in stereo matching, such as unstructured light [N84]. Despite these encouraging results, however, algorithms for machine vision (perhaps more than for any other field) require parallel implementations, due to the large number of pixels involved. Several such implementations are discussed next.

Image size	Displacement range	Window size	Time in sec's
64x64	+/- 1 pixel	+/- 3 pixels	0.093
128x128	+/- 2 pixels	+/- 6 pixels	1.36
256x256	+/- 4 pixels	+/- 8 pixels	16.6
256x256	+/- 8 pixels	+/- 8 pixels	53.3

Table 1: timings for various image parameters on a SUN Sparcstation1

4 Parallel Implementations

There are also a number of parallel implementations possible. One suitable for a hypercube architecture is described in [LBP88], which runs in $O(\delta^2)$ time using n*n processors and hypercube connections. This version can also be adapted to a 2D mesh topology to run in $O(\delta^2 * \nu)$ time using n*n processors using only NEWS connections.

It is also possible to implement this algorithm using a highly, rather than massively, parallel architecture. One possibility would be to have an array of k processors which would process k rows in the image at a time, for $\lceil \frac{n}{k} \rceil$ iterations through the image. Each would then process the pixels using the window overlap technique of figure 6. This would take

$$O(\lceil n/k \rceil * n * \delta^2 * \nu)$$

time with k processors. It would be necessary to perform $\delta + \nu$ iterative shifts of rows of data, both above and below, to allow each processor to access only its own local memory in doing its computations.

A much better bound can be achieved if we allow each processor to exclusively read the calculated column sums of the processor below it. With a small number of processors k, where k < n, it could be possible to process k rows at a time, in a staggered fashion, with processor i working on pixel [x,y] when processor (i-1) completes the processing for pixel [x,y-1]. In this manner all of the previous column sums would be available, except those for the very first row, which must be done separately. The first row itself can be divided up among the k processors, each receiving a $\lceil \frac{n}{k} \rceil$ subarray of pixels. This would take

$$O((\delta^2 * \nu^2 + (\lceil n/k \rceil - 1) * \delta^2 * \nu) + \lceil (n-1)/k \rceil * (n + k - 1) * \delta^2)$$

time, where the first term is for the processing of the first row (consisting of the first pixel in each subarray plus the other pixels in each subarray) and the second term is for the other rows (consisting of staggered calculations each using the method of figure 7).

A highly parallel version of the box filter is much more obvious. We would simply perform the row-sums and box-sums in two separate passes with n processors. The need for parallelism in real-time computer vision is so obvious that special hardware architectures are designed specifically for the purpose. A particular architecture that is utilized in this paper (simulated in software, with hardware potential) is that of a pyramid, and is discussed in the next section.

5 Overview of Pyramid Architectures

The pyramid architecture consists of a parallel array of processors arranged in a hierarchal network, with a successively smaller number of processors at each level. Often the difference between each layer is a constant multiple such as four (two times on a side in the case of a square array at each level). Thus the layer index is the logarithm of the number of processors it contains. Each pixel in every layer (except the lowest) is considered the parent of those beneath it; in our example, there are four such children of each parent. The lowest layer often forms the retinal input of an image to the pyramid. Each higher level in the pyramid constructs a coarser representation of the layer below it. The nature of this representation is arbitrary. It could be intensity values or features such as edges. This paper uses an image's pixel intensity (gray-level) values as its the retinal input. Each pixel has the average intensity of its four children. Instead of simple averaging, it is possible for a parent to be a Gaussian weighted average of the pixels beneath it. By taking the

difference between adjacent layers in this Gaussian pyramid (with the smaller layer appropriately expanded via interpolation) one can construct a Laplacian pyramid which approximates a difference of Gaussian filter on the image which can be used for edge detection [H87].

By moving this representation up levels in the pyramid, we create coarser resolutions of the original image. This process is called downsampling the image. Clearly, a downsampled image has less detail than the original, but may well have enough detail for certain vision problems. A downsampled image would of course be much faster to process, its key advantage. In addition, when the number of processors per layer decrease logarithmically, the total number of processors is only 1/3 more than that of the lowest layer [D87].

Although the uses of a coarse image may be fairly obvious, it may not be so clear what benefits may be derived from other representations of an image. As we shall see next, an ideal candidate representation is the optical flow field itself.

6 Abstracting the Flow Field

There is a very simple but powerful mechanism for abstracting the optical flow present in any image using a pyramid structure. Whereas most previous applications of pyramid architectures have focused on the image itself, i.e. analyzing the image at several scales, we now look at analyzing the optical flow field at more than one resolution. Indeed, even the same hardware for a pyramid architecture could be applied to a properly represented flow field.

The sequence of images labeled figures 12-19 shows the author standing up at a 45 degree angle and the resulting flow fields for three different resolutions, 256x256, 128x128, and 64x64. As shown by the fairly accurate flow field at each of the three scales, the algorithm performs rather well. We still have the problem of interpreting a field of many thousands of optical flow needles. To do this, we downsample the resulting flow field in much the same manner as we had done for the image itself. Figure 16 shows the 64x64 flow field downsampled to a 32x32 scale. Notice that little information is actually lost in the process. In particular, as we downsample the flow field, the random noise is filtered out, either canceling with other random noise or being dominated by the correct flow needles. The method of downsampling is by simple averaging; a coarse flow needle can be viewed as the average of the finer flow needles beneath it in the pyramid. Figures 17 and

18 also display resolutions at 16x16 and 8x8 respectively. Ultimately, in figure 19 we are left with a simple 4x4 optical flow needle diagram, giving an intuitive high-level description of the motion in the scene.

Our motive in performing this downsampling of the flow field was to derive a high-level interpretation of the motion in the images. It turns out, however, that this high-level interpretation can be used directly in a tracking application. Each large pixel at the 4x4 scale represents a large area of motion. Connected areas of homogeneous motion usually represent the same object (or possibly two or more occluding objects). At the 4x4 scale there are only 16 total areas to inspect, so it is computationally trivial to determine which areas represent homogeneous regions of motion. These large coherent regions are presumably moving objects, translating proportionally to the determined (averaged) optical flow of the region. A description consisting of the locations, velocities and component areas of these regions is sufficient to track a moving object (for the 2D problem). A tracking system using a sequence of images would be further resistant to errors. An erroneous result from one pair of frames can be compensated for on the next pair, as long as the object has not left the field of view. (Ideally, with a single moving object there would only be one region of motion detected. If we had more regions to deal with, there would have to be a method for deciding which one was the object of interest, based on factors such as size and the object's previous position in the image.) These concise descriptions would be more practical than an unprocessed 4,000 pixel needle diagram, since the detail of the latter is unnecessary for typical navigation problems. In addition, the operations used in downsampling the flow field are in fact identical to those used in downsampling the image; thus any specialized hardware can be reused for this purpose. Even in software the operations are computationally trivial. If we have a ancestor whose value is the average of its descendants, then the math is just a matter of some additions (at most one per pixel) plus a lesser number of bit shifts (division by a power of 2), in a typical pyramid. (In fact, as far as our results are concerned, the time cost is negligible.) Note too that this abstraction technique is independent of the method used in determining optical flow, as long as the output is dense.

In addition to the primary coarse description of motion, finer detail can be accessed on demand merely be examining the flow field at the appropriate scale, given a typical pyramid architecture. A good example of this is found in the next section.

7 Application of a Pyramid Architecture

Although $O(n^2 * \delta^2)$ is good for a serial implementation of this type of algorithm, it is still roughly 16 times more computationally intensive to determine the optical flow at a scale of say 256x256 than for 128x128, since the displacement range is typically dependent on the image granularity. Although parallel versions are most desirable, it is impractical to devote a single Connection Machine to every robot that is to have navigational capabilities. This is certainly the case if the processing is to be performed on the host itself, as a 2600 pound Connection Machine would most likely be somewhat of a burden for most robots [TM88]. For this reason, we have developed a useful heuristic for providing detailed information at a fraction of the time cost. One way to reduce this cost would be to apply the finer level of processing only to selected areas of the image. Since this is an early stage of processing, however, it is not necessarily easy to say what areas are interesting and thus deserving of further scrutiny; we are assuming that the flow field will be used to aid segmentation, not vice versa.

To make our problem more concrete, we consider the following case. In figure 20 we have motion at two different scales, as shown in figures 21,22 and 23 at resolutions 256x256, 128x128, and 64x64 respectively. At the 128x128 scale, we see the figure on the left is rotating away from the monitor at a rate of 2 pixels per frame, but his head is moving at a slower rate, only 1 pixel per frame (this is reflected by the different needle lengths). At the 64x64 scale, the subject is moving at 1 pixel per frame, but the head's motion does not show up at all; it is motion at a scale less than the coarser resolution can discriminate. But this information need not be lost forever, if we could define the appropriate areas of interest and analyze them at a 128x128 scale. An obvious answer would be to consider the outline of objects to be significant, but this could require costly segmentation analysis, even when at the coarser scale (4,000 pixels). However, we can greatly facilitate the process by again using the concept of a pyramid.

In figure 24 we have downsampled the 64x64 flow field two levels, yielding a 16x16 result. An area of interest can simply be defined by any empty pixel (zero motion at the 16x16 scale) that has two contiguous non-empty 16-scale pixels adjacent to it; intuitively, we are doing edge detection at a very coarse scale, and are interested in the areas immediately next to these edges. Note that this primitive "edge detection" is done on a 16x16 flow field, and for practical purposes is negligible in time cost. The area of interest, a single

pixel at a 16x16 scale, corresponds to an 4x4 sub-array at the 64x64 scale, and an 8x8 sub-array at the finer 128x128 scale. For this array we can perform our finer motion detection, at the 128x128 scale, as usual. This array of optical flow needles can then be downsampled back to the 64x64 level, filling in the missing information (but only for pixels that have no motion previously detected by the normal 64x64 routine; these values are not overwritten). The corresponding 16x16 scale pixel is also updated, if necessary; it may be the case that the addition of some amount of motion at the 128x128 scale could cause some aggregate motion at the 16x16 scale where there previously was none. Note that by doing so we may create a new area of interest, using our old definition of adjacency to other 16x16 scale needles, because we have created a new non-zero motion pixel. Thus we may need to perform another check for each pixel in the 16x16 field for being interesting (by our definition). But since we are using a very coarse 16x16 flow field, it is computationally inexpensive to detect coarse (16x16 scale) edges for as many iterations as needed it until all areas that are considered interesting have been checked, and do not contain enough motion to justify filling in that area's 16x16 scale pixel. This is shown in figure 25. Note that there are needles representing the two different scales of motion, 128x128 and 64x64, although the plot is 64x64. The downsampled flow fields are shown in figures 26-29. Notice in particular the motion of the left subject's right hand, clearly shown in the full 256x256 and 128x128 scales and approximated in the pyramid scheme. As we downsample the latter flow field, this information is not lost. In fact, it is still clearly available at as coarse a scale as 8x8.

There are several advantages to this technique. First, it is completely modular. It only needs the original images and the output of the coarser processing. Because the pyramid scheme is iterative and adds more detail to the flow field with each iteration through the 16x16 downsampled field, it can be asynchronously interrupted, giving a monotonically improving (more detailed) flow field with time. This places it in the class of "anytime algorithms" [DB88], a desirable property for control techniques. The interruption can also occur on a finer scale, even in the middle of an iteration. For example, after the first four rows have been checked for interesting areas and processed, we can easily stop and return the flow field that we have at the time. The only risk in this example would be that there may be slightly more detail in those first four rows than in the rest. Further, a time-dependent planner can be used decide whether it would be cost-effective to perform the selected finer level processing or to make do with the coarser level and use the extra time to

perform some other work [BD89]. Of course, its primary advantage is that it provides a detailed flowfield at a reduced time cost; in this example, the detail of a 128x128 image with displacement window $\delta = 5$ and neighborhood patch $\nu = 13$ is provided in approximately .5 seconds, or a three times speedup.

Look again at the downsampled flow field in figure 29. The final result clearly indicates two large bodies of motion, one moving left and another moving up, the correct result. However, the final 4x4 field does not actually entail the presence of two separate objects. It may be the case that there is but a single object with two large parts, moving in some unfathomable way. To do this, we again make use of the pyramid architecture. A simple blob analysis on the 4x4 field would quickly reveal that the two large areas of homogeneous motion are connected at one coordinate. It is a simple matter to scale the coordinates by two and examine that boundary at a 8x8 level. At this granularity we see that the area of alleged connection is much smaller. We can again look at this yet smaller area at a 16x16 scale, where we discover that the objects are in fact not connected, at least not by anything of any significant size. If the respective flow fields were actually connected, we would be unable to tell whether we had two overlapping objects or a single object undergoing heterogeneous motion. In this case, we would need an additional mechanism to segment the scene. We would, however, be able to provide such a mechanism with an explicit representation of the location of a potential boundary between two objects.

There may be a problem if the desired fine scale motion does not occur near any significant body of coarser scale motion. This is because the pyramid technique searches for finer scale motion only near the coarse "edges"; this is what gives it its improved speed. Without these edge clues, the algorithm would not know where to look. There are a few possible solutions to this. One could always use feature detectors to define interesting areas. A better solution found in [C90] is to control the image sampling rate to find the fastest motion in the image. Once this is found we will always have some edges to use as our interesting areas, unless there is no motion in the image, in which case we have no further analysis to perform.

8 Conclusions and Related Work

This paper presented an efficient serial algorithm with a low time and time * processor bound. This particular implementation, for an image size 64x64 with parameters $\delta = 3$ and $\nu = 7$, which runs at over 10 frames per

second on a SUN Sparcstation1, has shown to be robust on dozens of sample image pairs and is the only known version that can be considered for a practical application.

We have seen that a pyramid architecture can be used to create a concise abstraction of an optical flow field which can be particularly useful for tracking purposes. It would be possible to use the same hardware to downsample both images and flow fields.

Finally we have utilized the abstraction present in a pyramid structure to apply detailed processing to selected areas of an image via a form of coarse edge detection. This gives a result approximating the output of the traditional algorithm using 128x128 images with $\delta = 5$ in only about .5 seconds, or a speedup factor of about three times.

This approach to optical flow is technically considered to be based on feature matching, where the (degenerate) features are the pixels' intensities themselves. An equally common approach is based on derivatives [HS81]. A discussion of the differences between the two is found in [LV89].

The techniques described above has several advantages over other approaches. The method of homogeneous pixels described in [HB88] loses effectiveness when then object being tracked overlaps another object in the visual field. Thus, it can be lost should it move too close to furniture or some bookshelves. The methods used in this paper, however, do not suffer from these difficulties. The fast serial implementation described earlier allows greater resolution, in real-time, than homogeneous pixels.

It was shown that the order of the time bound for this optical flow algorithm was not dependent on the ν parameter. Since ν controls the smoothness of the result, we have the property that this approach does not take longer to produce smoother results, in contrast to methods such as [AW85] which require an iterative smoothing stage to regularize the motion field.

9 Future Directions

A large number of possible research directions have been created by this work. To actually use this algorithm on a mobile robot it is expected that several problems will have to be solved. These include calculating the optical flow on a moving camera head on a stable platform, or a moving platform itself [BBH+89].

One problem in machine vision is to undo the effects of perspective foreshortening in a robot undergoing

egomotion. In this case optical flow can result from both the 3D structure of a scene and from perspective effects. A moving robot with a wide angle lens camera would detect a diverging optical flow field about the focus of expansion in a looming image. A technique to invert this perspective effect and form a regularized field resulting only from the 3D structure of the scene is covered in [MBL89]. Also presented in this paper is a method for detecting obstacles, based on their distorted presence on the inverted perspective map.

A shortcoming of the algorithm is that the search area for the displacement of each pixel is proportional to $\delta * \delta$, thus the time required is quadratic in the velocities of the objects in the scene. A method to overcome this difficulty, using a controlled image sampling rate, is found in [C90].

Despite the algorithm's robustness, it has been found that it cannot easily differentiate moving objects from moving shadows. Although shadows on the ground plane can be detected via stereo processes or by an inverse perspective mapping [MBL89] moving shadows projected onto a vertical surface are currently indistinguishable from moving objects. Whether this fact is a bug or a feature is dependent on one's philosophy of optical flow; it may be inappropriate for the optical flow stage to speculate about the nature of the moving entities it detects. In any event, it would of course be necessary to include a shadow discriminating module before the downsampled flow field is presented to a robot's navigation system.

A purely brightness-based approach has a tendency to over represent the size of a large moving object. This is because of the large neighborhood size. A point just off the edge of a moving object may be motionless, but a significant fraction of its neighborhood, that which overlaps the object itself, will be moving along with the object. In some cases this may be enough to give the pixel the erroneous displacement of the nearby object rather than the correct motion of the pixel itself. The probability of this occurring is proportional to the amount of the overlap. One possible solution would be to construct the neighborhood as a two-dimensional Gaussian rather than as a square. However, this adds a non-linearity in the neighborhood summation stage which would add a v*v term to nearly every implementation described, serial or parallel. Another possibility would be to add a edge-detecting module, which could make use of the relatively close approximation made by the algorithm. Other possible fixes to this problem are under investigation.

It is worth investigating pyramid architectures that are not based on ratios of two. The current pyramid implementation used features four times the number of pixels per lower level; as a consequence, there are

considerable detail differences between adjacent layers. Custom made pyramids featuring more gentle ratios of 4:3:2 and polar downsampling are discussed in [PFH87]. Many other suggestions to augment pyramid topologies are found in [U87].

10 Acknowledgements

The author would like to thank Heinrich Bulthoff for providing guidance throughout this paper's evolution, and to Tom Dean and Solomon Shimony for reviewing drafts of this paper.

This work was supported in part by a National Science Foundation Presidential Young Investigator Award IRI-8957601 with matching funds from IBM, and by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Air Force Office of Scientific Research under Contract No. F49620-88-C-0132.

[AW85] P. Anadan, R. Weiss, Introducing a Smoothness Constraint in a Matching Approach for the Computation of Displacement Fields, DARPA IU Workshop, 186-196, 1985.

[BD89] M. Boddy, T. Dean, Solving Time-Dependent Planning Problems, IJCAI89, 979-984 1989

[C90] T. Camus, Space-Time Tradeoffs for Adaptive Real-Time Tracking, Brown AI memo 90.001, July 1990

[DB88] T. Dean, M. Boddy, An Analysis of Time-Dependent Planning, Proceedings AAAI-1988, 49-54, 1988

[BLP89a] H. Bulthoff, J. Little, T. Poggio, A Parallel Algorithm for Real-time computation of Optical Flow, Nature 337(6207):549-553, 9 Feb 1989

[BLP89b] H. Bulthoff, J. Little, T. Poggio, A Parallel Motion Algorithm Consistent with Psychophysics and Physiology, IEEE 1989 Workshop on Visual Motion, 165-172, March 1989

[BM86] R. Bates, M. McDonnell, Image Restoration and Reconstruction, Clarendon Press, Oxford, 1986

[BBH+89] P. Burt, J. Bergen, R. Hingorani, R. Kolezynski, W. Lee, A. Leung, J. Lubin, H. Shvaytser, Object Tracking With a Moving Camera, IEEE 1989 Workshop on Visual Motion, 2-12, March 1989

[D87] C. Dyer, Multiscale Image Understanding, in Uhr (ed), Parallel Computer Vision, 171-213, Academic Press, Boston, 1987

[H87] R. Hummel, Scale-space Formulation of Pyramid Data Structures, in Uhr (ed), Parallel Computer Vision, 107-123, Academic Press, Boston, 1987

[HB88] I. Horswill, R. Brooks, Situated Vision in a Dynamic World, Proceedings AAAI-1988, 796-800, 1988

[HS81] B. Horn, P. Schunck, Artificial Intelligence 17:185-203, August 1981

[LBP88] J. Little, H. Bulthoff, T. Poggio, Parallel Optical Flow Using Local Voting, Second International Conference on Computer Vision, 1988

[LV89] J. Little, A. Verri, Analysis of Differential and Matching Methods for Optical Flow, IEEE 1989 Workshop on Visual Motion, 173-180, March 1989

[MBL89] H. Mallot, H. Bulthoff, J. Little, Neural Architecture for Optical Flow Computation, AI memo 1067, March 1989

[NS88] K. Nakayama, G. Silverman, The Aperture Problem-I. Vision Research 28(6):739-746, 1988

[N84] H. K. Nishihara, Practical Real-Time Stereo Matching, Optical Engineering 23(5):536-545, Sept/Oct 1984

[PFH87] S. Peleg, O. Federbusch, R. Hummel, Custom-Made Pyramids, in Uhr (ed), Parallel Computer Vision, 125-146, Academic Press, Boston, 1987

[TM88] Thinking Machines technical report HA87-4, April 1987

[U87] L. Uhr, Highly Parallel, Heirarchal, Recognition Cone Perceptual Structures, in Uhr (ed), Parallel Computer Vision, 249-291, Academic Press, Boston, 1987

```
for y = 0 to (n-1)                                  time          memory
 for x = 0 to (n-1)                                 ----          ------
   when x=0 and y=0                                 (1)
    V displ u,w                                     (d*d)         (d*d)
     for i = 1 to v                                 (v)
      S(0,0;i) = sum(j=1,v) m(i,j)                  (v)           (v) *
      M(x,y) = M(x,y) + S(0,0;i)                    (1)           (1) **
   when x={1,n-1} and y = 0                         (n-1)         (n-1)
    V displ u,w                                     (d*d)         (d*d)
     S(x,0;v) = sum(j=1,v) m(v,j)                   (v)           (1) ***
     M(x,y) = M(x-1,y) - S(x,0;0) + S(x,0;v)        (2)           [reuse **]
   when x=0 and y={1,n-1}                           (n-1)
    V displ u,w                                     (d*d)
     for i = 1 to v                                 (v)
      S(0,y;i) = S(0,y-1;i) - m(i,0) + m(i,v)       (2)           [reuse *]
      M(x,y) = M(x,y) + S(0,y;i)                    (1)           [reuse **]
   when x={1,n-1} and y={1,n-1}                     (n-1)(n-1)
    V displ u,w                                     (d*d)
     S(x,y;v) = S(x,y-1;v) - m(v,0) + m(v,v)        (2)           [reuse ***]
     M(x,y) = M(x-1,y) - S(x,y;0) + S(x,y;v)        (2)           [reuse **]
```

In each of the following definitions, there is an implied subscript (u,w) indicating the particular displacement under consideration. Note that the column sums are dependent on their values at previous coordinates, and therefore are never re-initialized to zero.
S(x,y;i) = The ith column sum of the current window for pixel x,y.
M(x,y) = The match strength of the patch for the pixel at x,y.
m(j,k) = The match at position j,k in the current window.

Figure 1: Time and memory analysis of the serial algorithm.

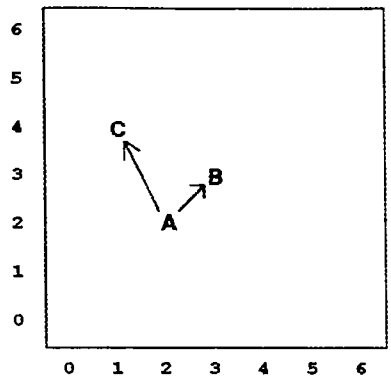

Figure 2: Possible placements for pixel A. A displacement of +1,+1 of A from image 1 would correspond to position B in image 2. Similarly, a displacement of -1,+2 would correspond to position C in image 2.

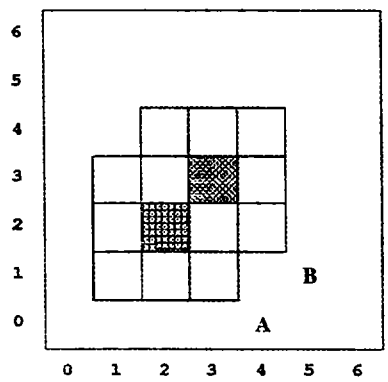

Figure 3: Translation of a patch of pixels. A displacement of +1,+1 of the 3x3 patch pf pixels centered at 2,2, labelled A, in the first image would correspond to the patch centered at 3,3, labelled B, in the second.

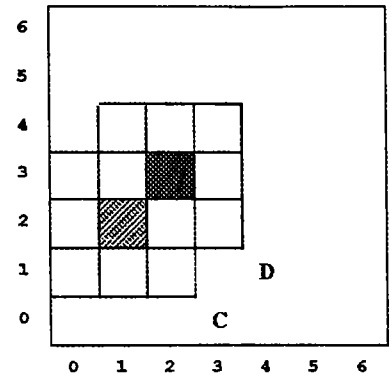

Figure 4: Translation of a patch of pixels adjacent to those of figure 3. The displacement is the same, +1,+1, but for the pixel at 1,2. Note that the windows of pixel 1,2 overlap with those of pixel 2,2.

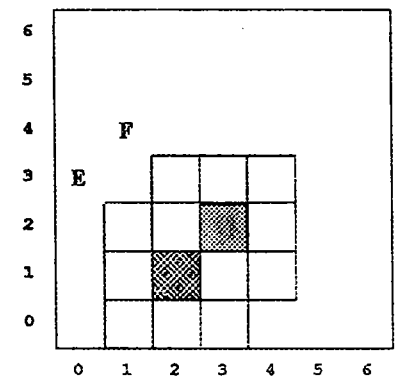

Figure 5: Translation of the patch of pixels centered at 2,1. Note that the rightmost columns of each window of pixel 2,1 overlap with the rightmost columns of each window of pixel 2,2.

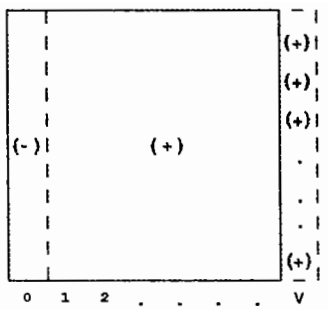

Figure 6: Summing a pixel's patch using the left neighbor's previously calculated patch. If we view the new window as the old window shifted right once, then the sum of the new window is the value of the old window, plus the rightmost column sum, minus the previously calculated leftmost column sum.

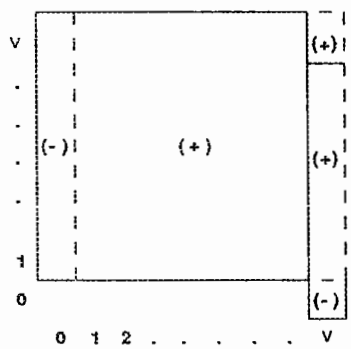

Figure 7: Summing a pixel's patch using the left neighbor's patch plus the lower neighbor's rightmost column. The value of the new window is that of the previous window minus the leftmost column sum of the previous window, plus the rightmost column sum of the lower neighbor, plus the top right match minus the bottom right match.

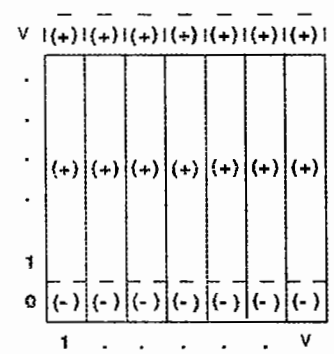

Figure 8: Summing a pixel's patch when no neighbor is present. Each column sum is derived from the column sum of its upper neighbor, plus the top match minus the bottom match.

a	b	c	d	e
f	g	h	i	j
k	l	m	n	o
p	q	r	s	t
u	v	w	x	y

Figure 9: Demonstration of a box filter. Initial setup of a 5x5 matrix of variables.

a+b+c	b+c+d	c+d+e
f+g+h	g+h+i	h+i+j
k+l+m	l+m+n	m+n+o
p+q+r	q+r+s	r+s+t
u+v+w	v+w+x	w+x+y

Figure 10: Row-wise set of partial sums (3 elements each) in a 5x3 matrix. Note that the partial sums are not defined for the left and right boundaries.

a+b+c + f+g+h + k+l+m	b+c+d + g+h+l + l+m+n	c+d+e + h+i+j + m+n+o
f+g+h + k+l+m + p+q+r	g+h+i + l+m+n + q+r+s	h+i+j + m+n+o + r+s+t
k+l+m + p+q+r + u+v+w	l+m+n + q+r+s + v+w+x	m+n+o + r+s+t + w+x+y

Figure 11: Column-wise set of sums of the previous row-wise partial sums in a 3x3 matrix. The top and bottom boundaries are not defined. Each entry is a 3x3 unweighted sum ("box") of the elements adjacent to the center element in the original array.

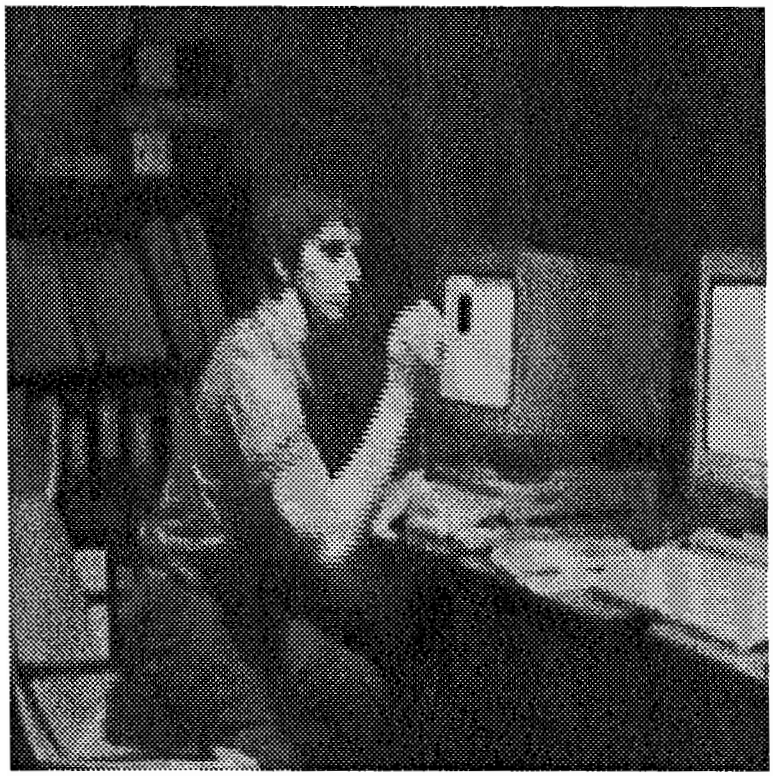

Figure 12:

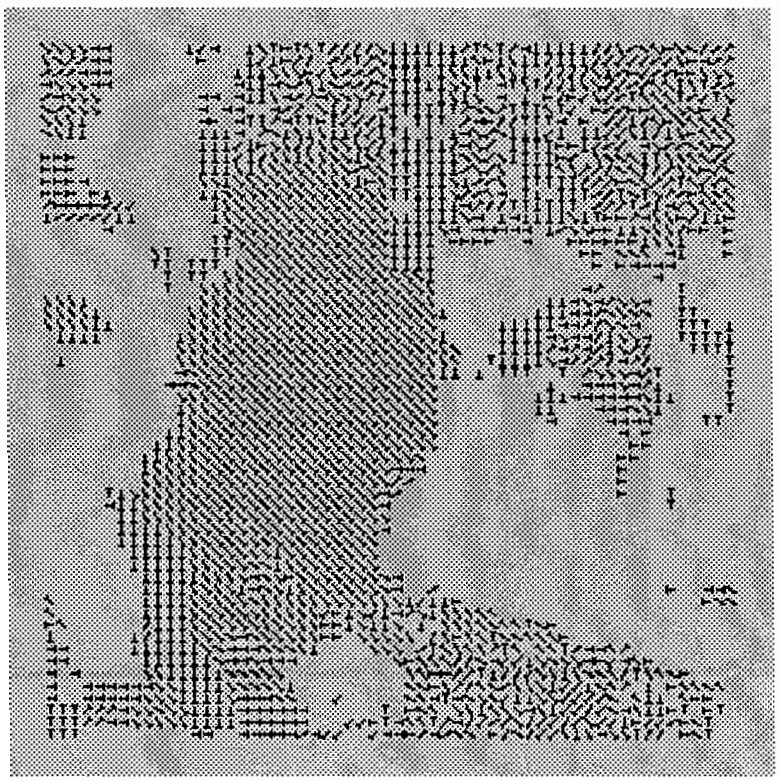

Figure 13:

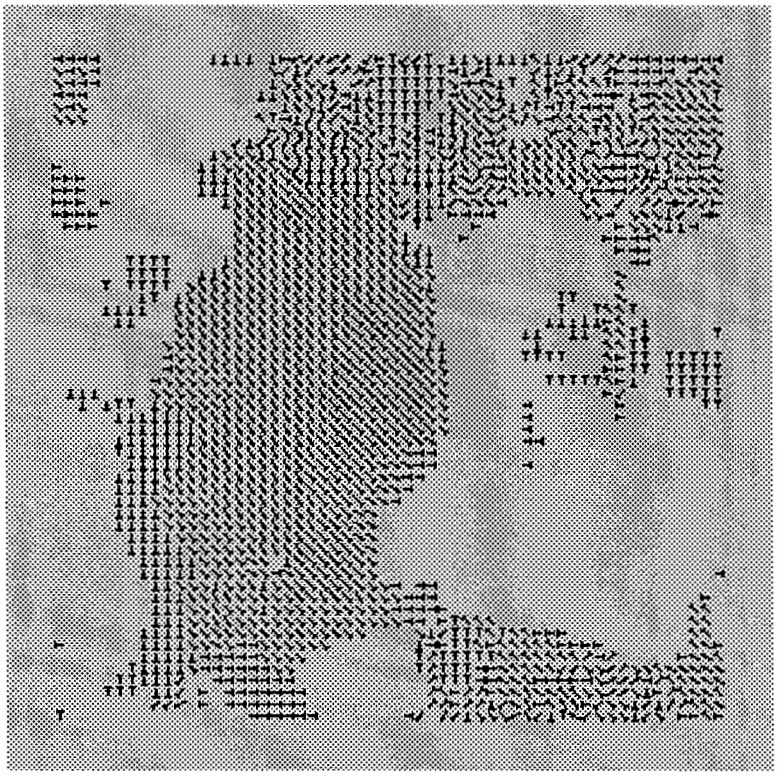

Figure 14:

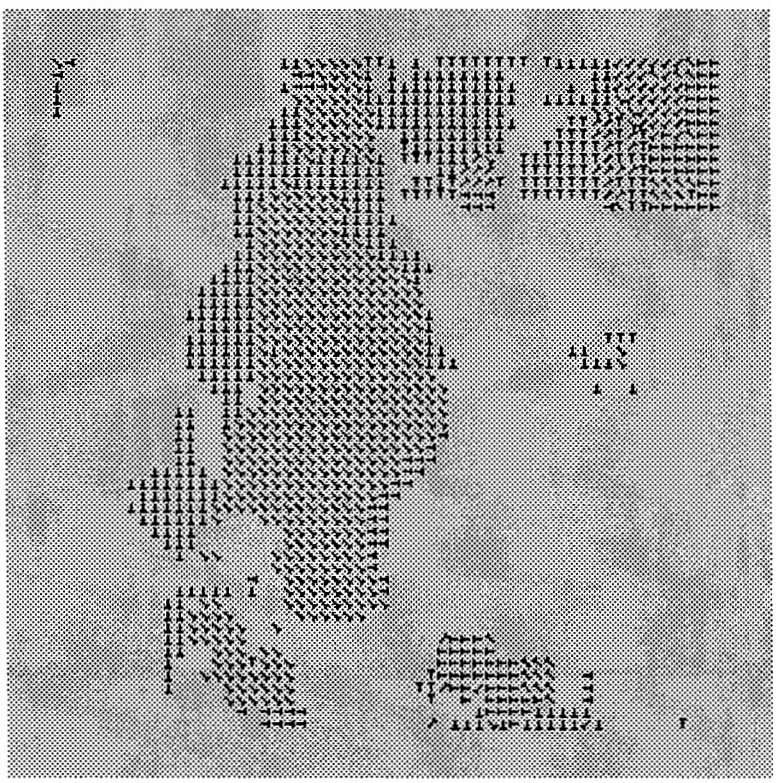

Figure 15:

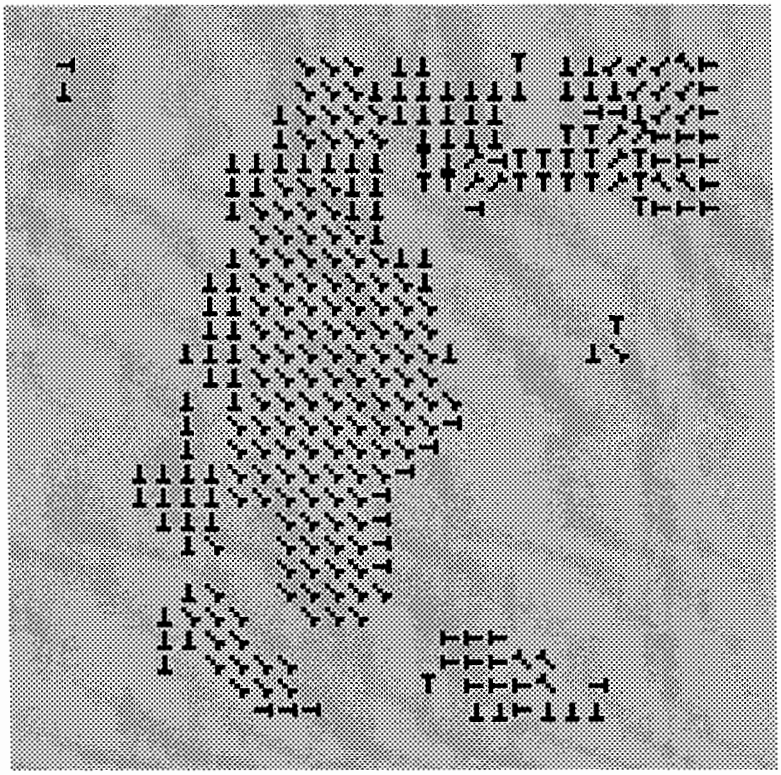

Figure 16:

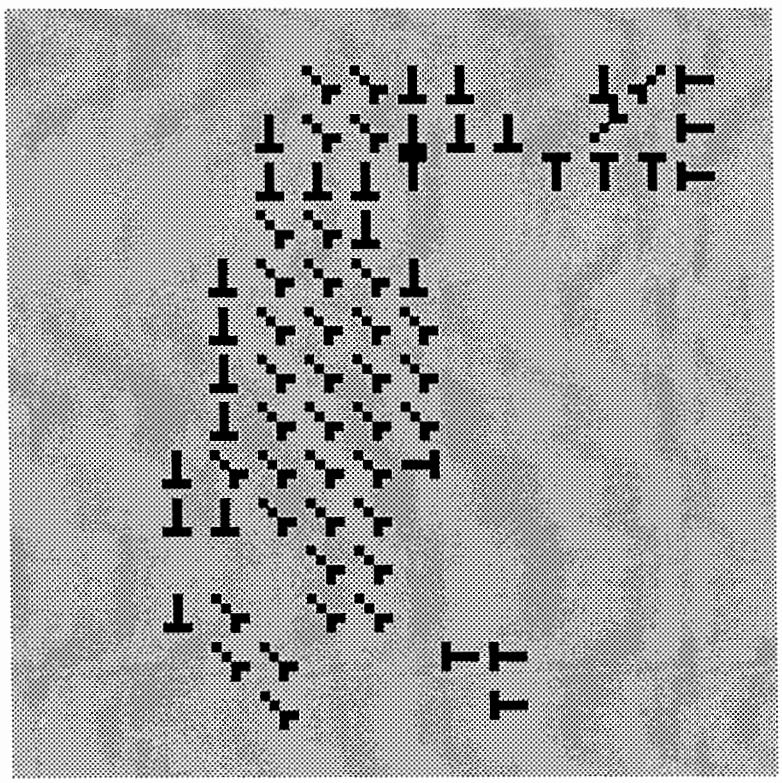

Figure 17:

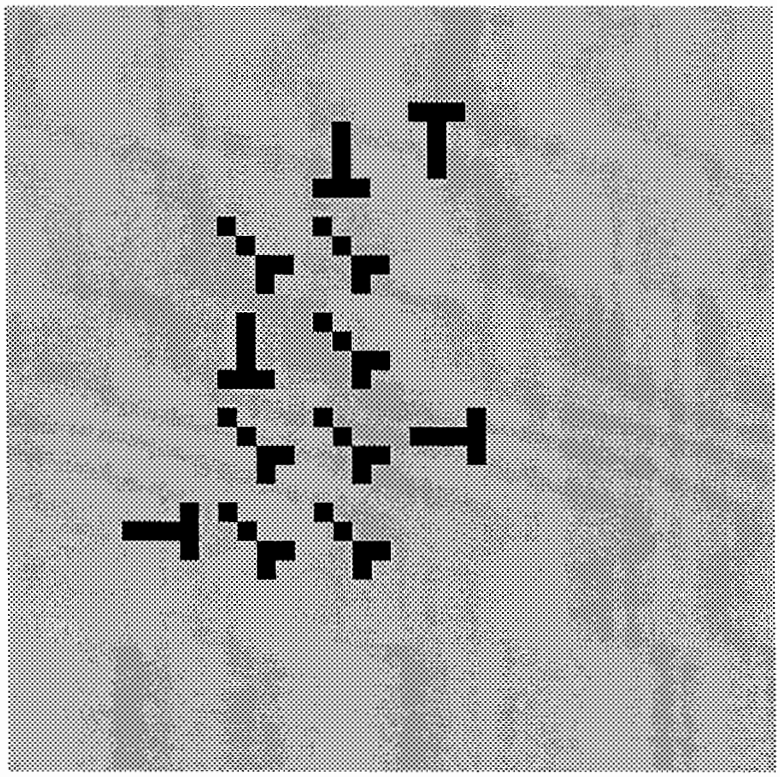

Figure 18:

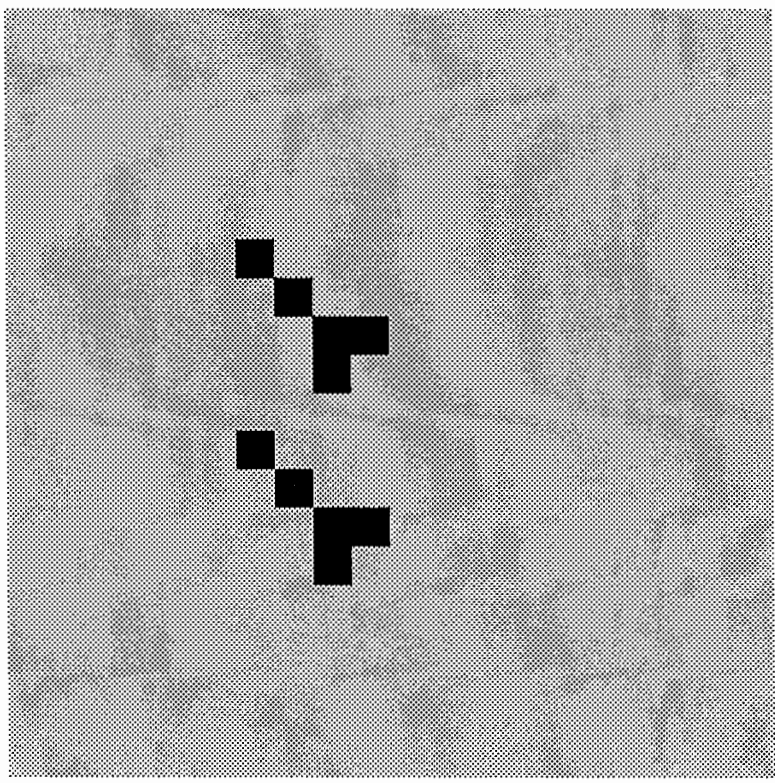

Figure 19:

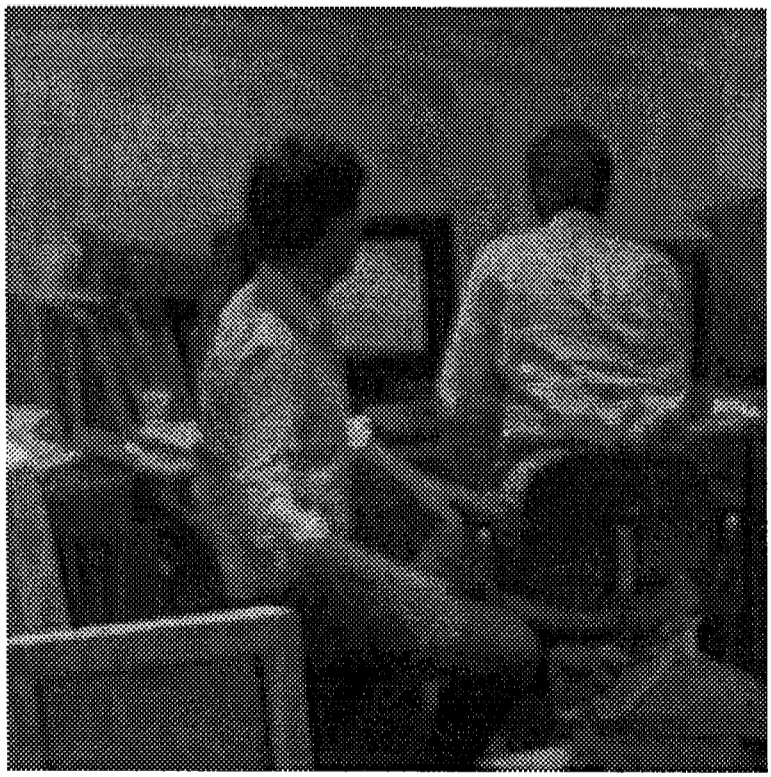

Figure 20:

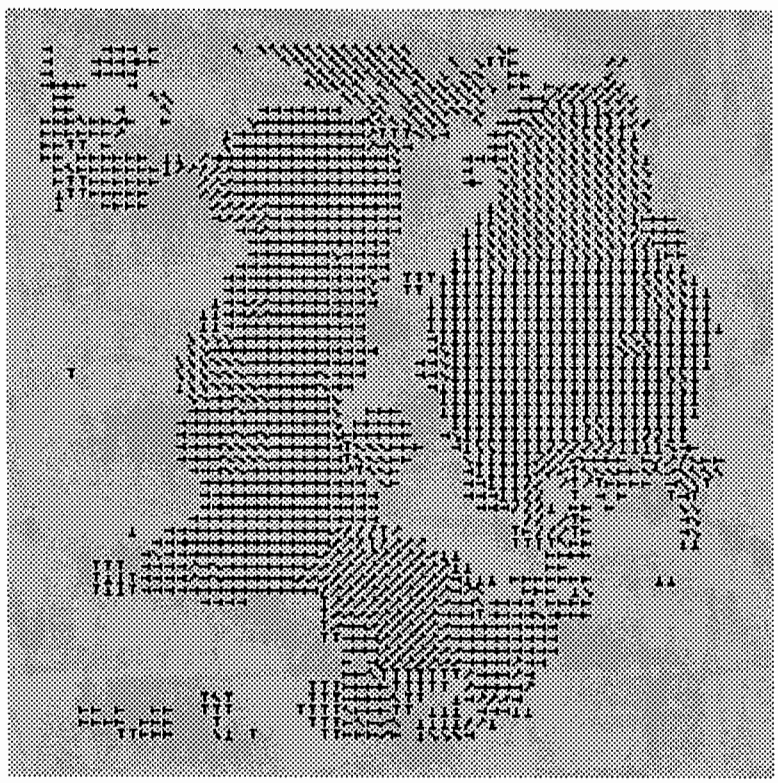

Figure 21:

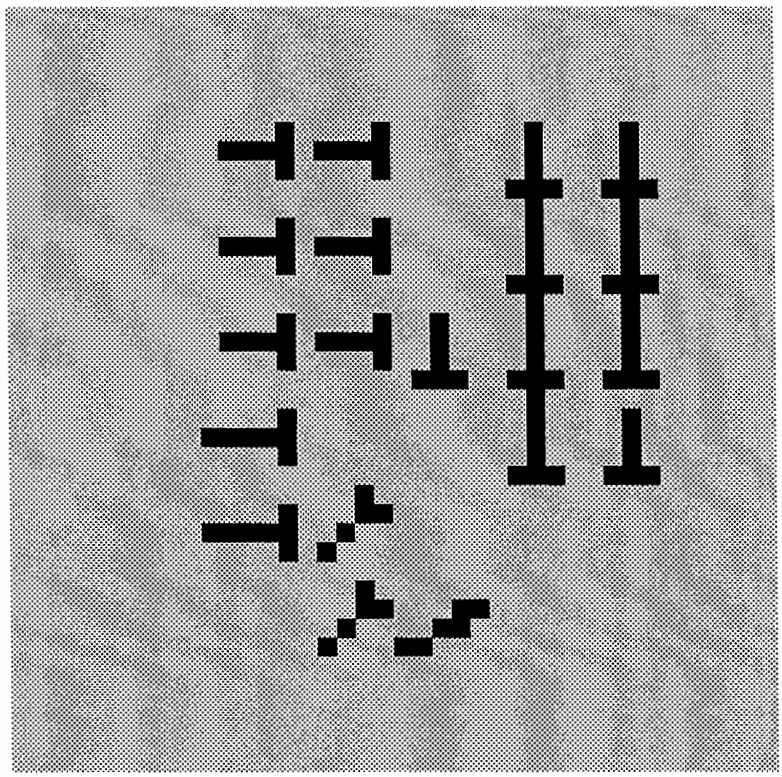

Figure 28:

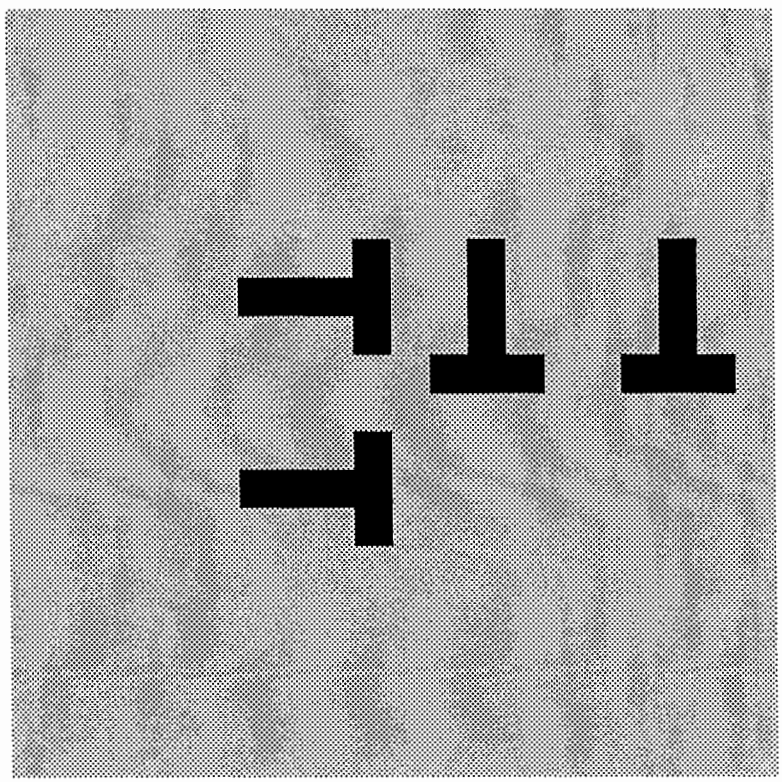

Figure 29:

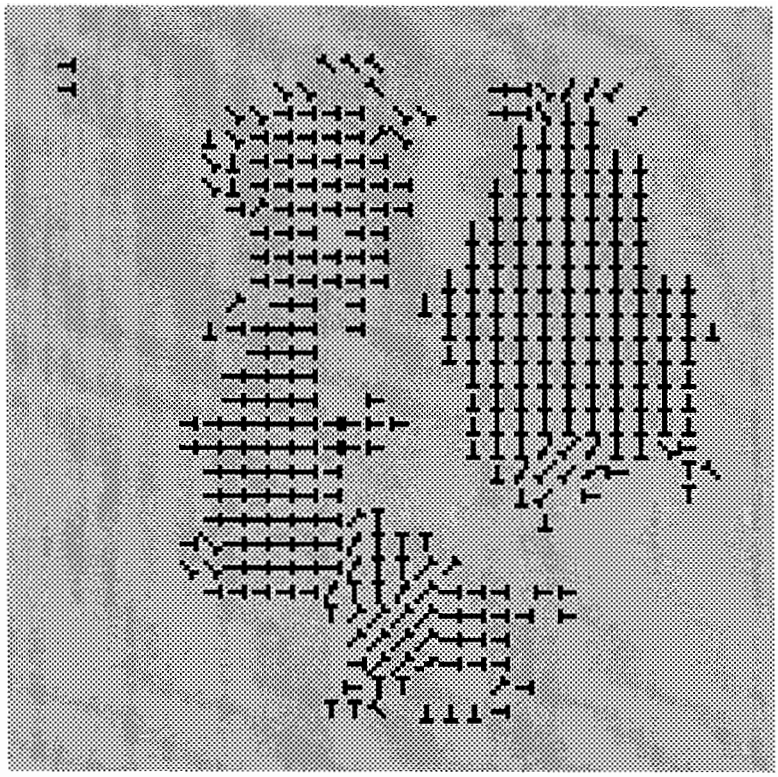

Figure 26:

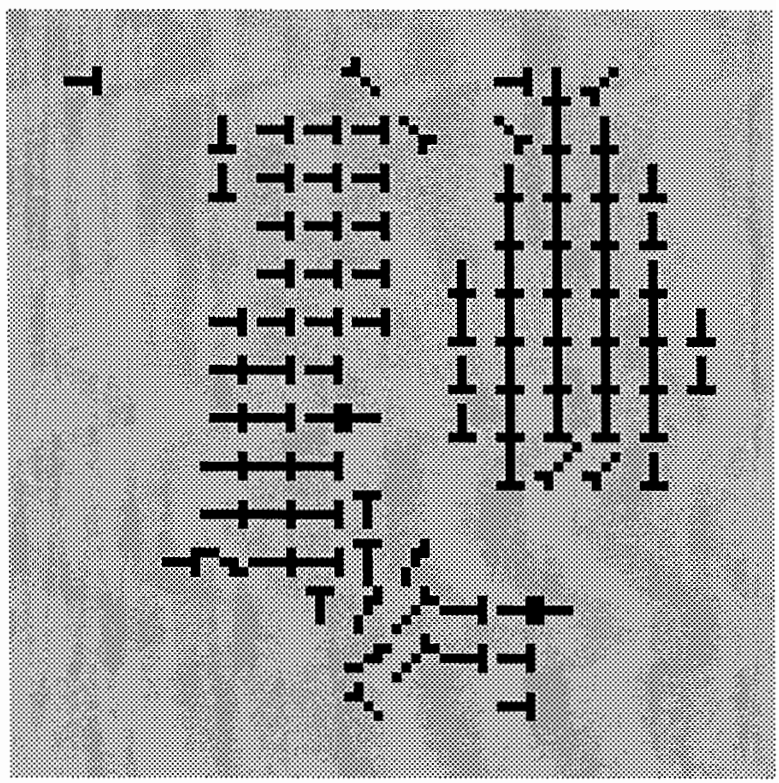

Figure 27:

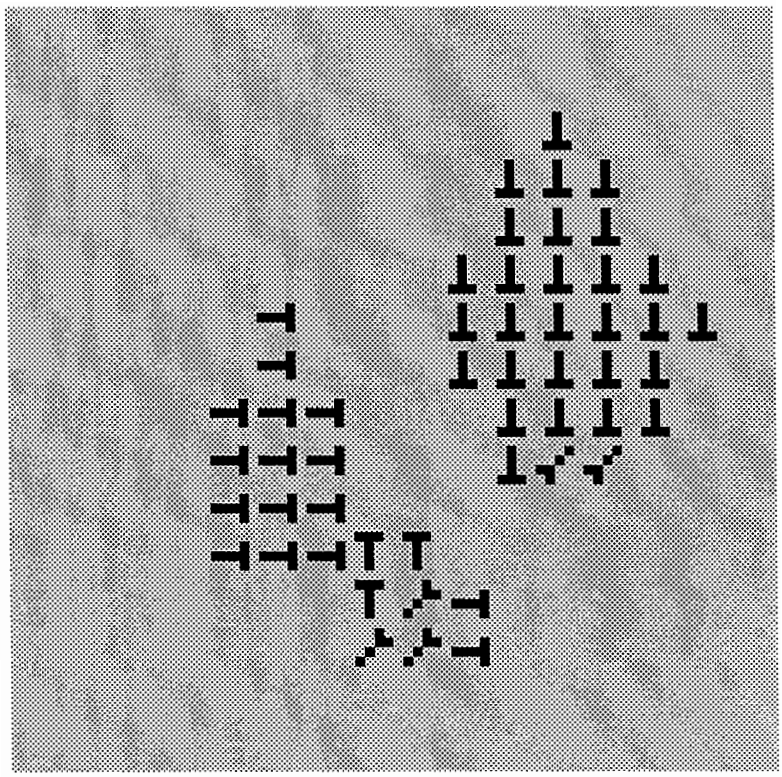

Figure 24:

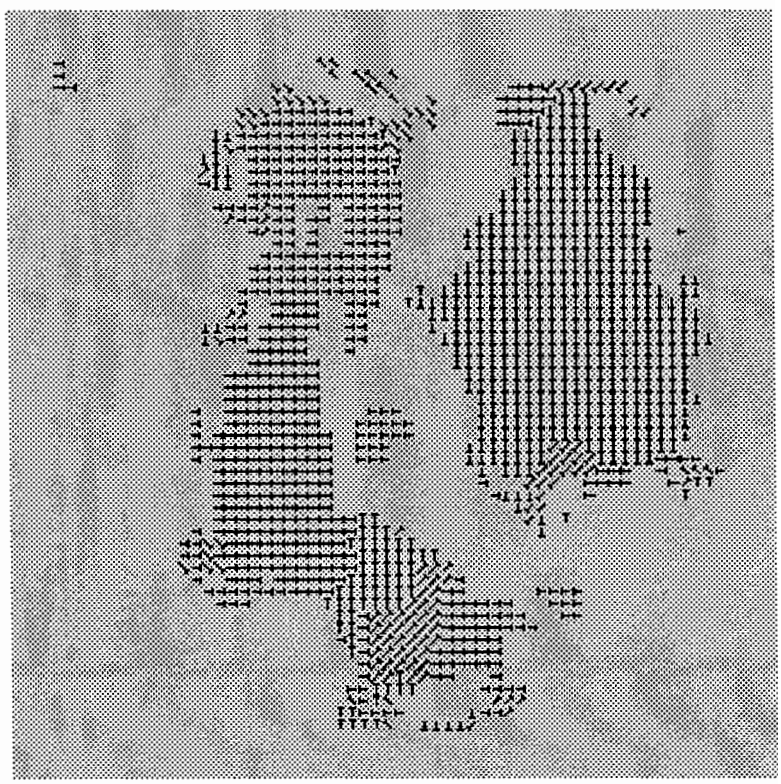

Figure 25:

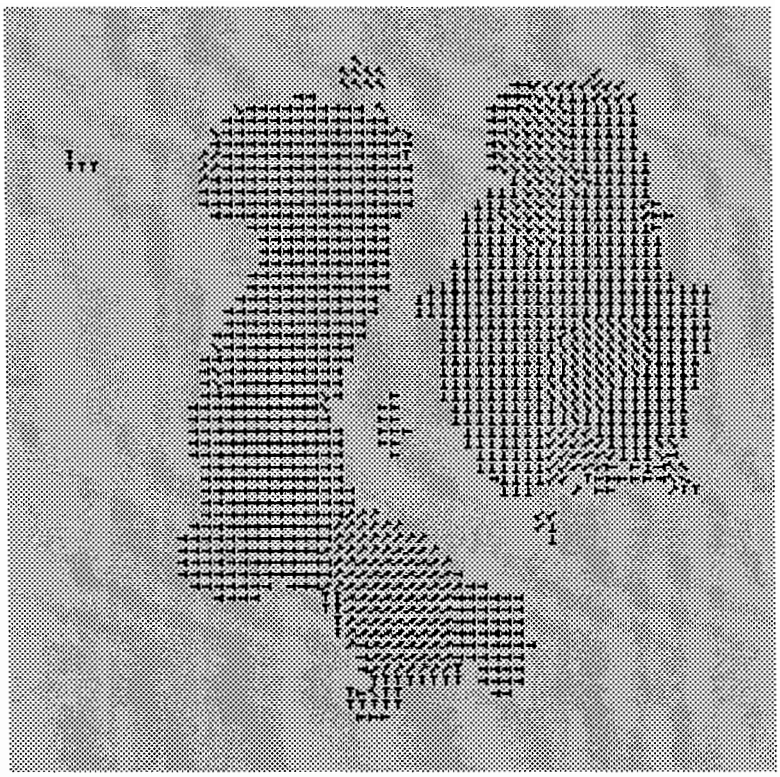

Figure 22:

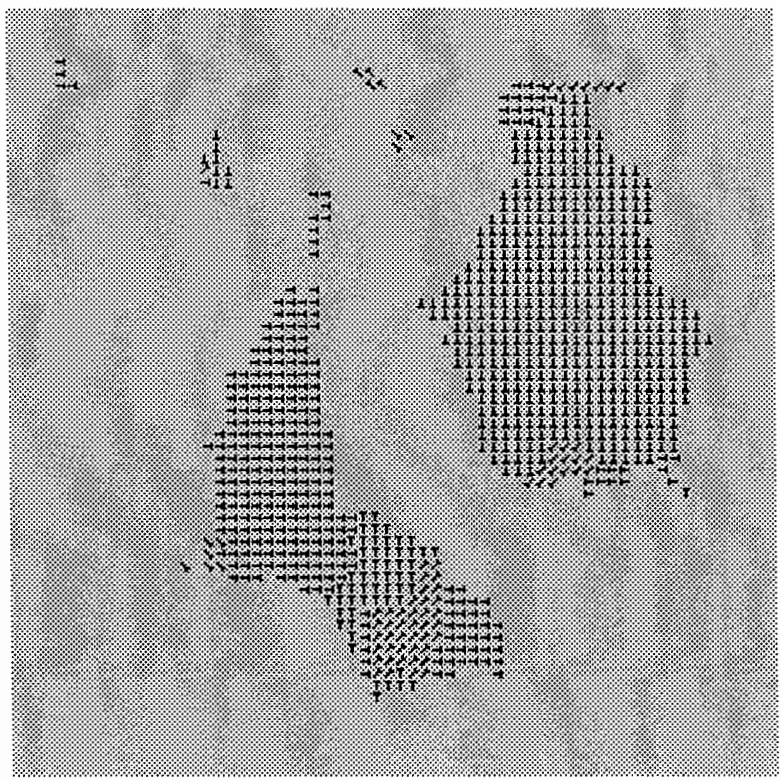

Figure 23: