The Pipe-Group Architecture for Real-Time Active Vision

Philip F McLauchlan, Ian D Reid, Stuart M Fairley and David W Murray

Department of Engineering Science, University of Oxford, Parks Road, Oxford OX1 3PJ, UK

Abstract

This paper describes a vision processing architecture which together with a robotic camera platform and control subsystem comprise a multi-processor real-time system for research into active vision. Experimentation in time-critical aspects of the active vision paradigm over several years has demonstrated the flexibility of “pipe-groups” used as the building block in the system. The design of the entire system is overviewed, and the design of pipe-groups and their software is described in more detail. An active method of determining the exact processing latency, required for stable control, is explained. The paper illustrates the use of the pipe-group processing architecture and its real-time processing capabilities with an example of active stereo tracking and structure recovery using the method of affine tranfer.

1 Introduction

In active visual processing, visual data are fed back to control not only the physical parameters of the camera system — principally the direction of gaze — but also how future images are processed. The primary aim is to move away from the idea that vision fills a bran-tub or lucky-dip of percepts that some other cognitive process may or may not find useful, and to move towards coupling visual sensing and perception to robotic actions. The resulting set of visual behaviours is then intimately related to specific world tasks [1, 2, 3].

In recent papers, we have described the design and realization of a highly agile electromechanical camera platform and control sub-system for active vision research [4, 5], and have described a number of reactive and purposive visual behaviours running on the system. The reactive behaviours have included rapid saccadic redirections of the cameras in response to unexpected motion in the scene — both “capture” saccades when the resultant projected image motion is largely translational, and “panic” saccades when the image motion is predominantly divergent, indicating looming [6]. The more purposive behaviours include tracking for surveillance using both feature clusters [7, 8] and optical flow [9]; line following [10]; and fixation for navigation [11].

This paper describes the vision system designed to support our research into active processes. The need to implement a variety of visual behaviours whose exact nature was unknown a priori precluded the design of a fixed architecture, and instead we used MIMD processors in a reconfigurable network. However because the algorithms have the common element of images as input and angular demands as output, we designed a general communications protocol based around “pipe-
groups”. The system has been implemented on several platforms using Inmos T805 transputers and, more recently, on a system using Texas Instruments C40 DSPs. Here we describe the former.

Real-time processing is *de rigueur* for active visuo-control systems. The sampling rate determines the maximum frequency of scene movement that can be reacted to without aliasing. But a high processing rate is insufficient in itself. In addition, the process latency must be both low and known frame by frame to allow prediction over the delay. Although the gaze controller in our system is real-time in the strongest sense of the term, in that a new demand *must* be generated every 2 ms, the visual processing is real-time in the weaker sense of being “sufficiently high to effect practical visually guided control or decision making” [12]. The reason for not adopting fixed time visual processing is as follows. The gaze controller can select its input from one of several visual processes which run concurrently. Each process has its own characteristic latency lying somewhere between 100 and 200 ms. If the system has to cope with large changes of characteristic latency *between* processes, it makes little sense not to allow it to cope with the much smaller variations that arise from frame to frame *within* a process because of varying volumes of data.

Later in the paper we describe the use of the pipe-group in more detail, and give an illustrative example from our research on active vision, namely stereo tracking using affine transfer. First however we sketch the overall architecture of the active vision system and give a visual method of determining latency. Measuring and managing process latency is key to the timely functioning of the visuo-control loop.

## 2 System architecture

The overall architecture (figure 1) has four principal elements — the camera platform, the vision system, the high-level gaze controller and the low-level servo-controller. All computation for vision and control is performed on a network of some 30 T805 transputers with link switches providing reconfigurability. An alternative of using a fixed topology with routers providing virtual links between any processors on the network (such as in [13]) was rejected because it would have introduced an undesirable extra latency in communication between non-adjacent processors.

Although the T805 transputer remains a valuable processor for many embedded control applications, it has always provided a rather limited communication bandwidth between processors, and now suffers from a relatively modest processing power. However, the active paradigm encourages selective processing, and indeed the processor and communications restrictions make this virtue a necessity. One aspect of the visual selection is achieved by controlling the direction of gaze of the camera and another by selectively windowing and subsampling images. For the reactive processes mentioned in the introduction, we use a wide field of view and subsampled imagery, while for more purposive operations such as tracking we use a small, full resolution, central window cut from the imagery. This scheme is attractive because the sensor is under control: without such control, less committed schemes, such as a multi-level pyramid architecture [14], suggest themselves.
Figure 1: The overall architecture of the visuo-control loop developed for our work. The vision system provides parallel feedback loops, grouped into peripheral and foveal channels. The different delays in the different processes require platform encoder data to be stored in a ring buffer, so that they can be used in prediction.

The controller is divided into two, with the high level gaze controller handling visual processes and behaviours, and the low level servo-controller handling head kinematics, joint angles, motors and encoders. Part of the high level gaze controller operates asynchronously at a rate determined by the vision processing and selects and predicts visual output to drive gaze constructs such as pursuit, saccade, and so on. Selection of feedback loops and behaviours is achieved using a finite state machine. For prediction we use a constant velocity Kalman filter which takes in delayed position and velocity information, along with a value for the latency for a particular frame of data, and generates a filtered estimate of the prompt position and velocity. The other part of the high level controller runs synchronously at 500 Hz. Every 2 ms it extrapolates the estimate of the prompt gaze direction and velocity in head coordinates and passes these to the servo-controller. The servo in turn performs all synchronous controls, as well as the forward and inverse kinematics, trajectory limiting, and receiving feedback from encoders on the motor shafts. The servo-controller also has an important rôle as system clock. The need to combine prompt head data with delayed vision results for prediction makes timing an important issue, the more so as visual processes may have different rates and will almost certainly have different latencies. As part of its 500 Hz control loop, the servo maintains a ring buffer of mount status data, such as position and velocity and control mode (saccade, smooth pursuit, etc) at the time of image capture, data which can be requested by the vision processes and prediction stage.

Each of the three stereo camera platforms or “heads” in our laboratory (figure 2) has four axes powered using geared DC drives. To allow the axes to move without visual feedback and at much higher gains and speeds, the servo-controller [4] also derives feedback from encoders on the motor shafts. Via the forward kinematics, these measure an absolute gaze direction, as opposed to a gaze
direction relative to the visual scene. Such proprioceptive feedback is essential for rapid motions, where images are blurred, and visual feedback quite useless.

Figure 2: The large, medium-sized and small “Yorick” platforms built and used in our laboratory.

2.1 Determining latency for control

To obtain an estimate of prompt position from delayed position and velocity requires knowledge of the total latency $\Delta t$, the sum of the latency $\Delta t_c$ between the camera and the image server caused by capture and preprocessing hardware, and the latency $\Delta t_p$ within a transputer pipe-group itself.

$\Delta t_p$ is found using the servo-controller’s processor as a master clock. When a frame or field is ready for output from an image server, a request for timing data is made to the master clock, and the visual data time-stamped. A similar request is made by the high-level gaze controller at the point when it makes the prediction, and the difference used as the latency of the pipe.

The more difficult problem is determining the latency $\Delta t_c$. This is required not only to compute the total latency, but also to allow the correct platform odometry, ie axis angles and velocities, to be stamped on the imagery. The odometry is often used in the visual processing, for example to subtract image motion arising from known movement of the platform. As noted earlier in this section, the servo-controller maintains a ring buffer of “prompt” odometric data, and so the problem becomes one of determining how many entries to look back for the required data.

Our solution exploits the controllability of the camera platform. Suppose we record from the odometry the axis angle $\theta_{apparent}$ at the instant that imagery becomes available to the image server indicating that some fixed feature in the scene is centred in the image. As figure 3(a) illustrates, if the head’s angular velocity $\dot{\theta}$ is near zero, this apparent angle will be close to the actual angle $\theta_{actual}$. However, as the angular velocity increases, the latency will cause the apparent angle to be greater than the actual angle by $\Delta t_c \dot{\theta}$. Because the latency is fixed,

$$\theta_{apparent} = \theta_{actual} + \dot{\theta} \Delta t_c$$

and so the gradient of a linear fit to $\theta_{apparent}$ vs $\dot{\theta}$ gives the capture and preprocessing latency for all processes. Figure 3(b) shows such a linear fit, giving $\Delta t_c = 48 \pm 4$ ms.
Figure 3: Using controlled motions to recover visual latency using the prompt odometric feedback. (a) The apparent angle increases linearly with camera axis velocity over the actual angle. (b) The slope of the linear fit to this excess error angle versus velocity gives the capture latency.

3 The pipe-group vision system

An obvious method of increasing throughput in many local low-level vision algorithms is to use spatial parallelism. The transputer is reasonably well suited to coarse-grained spatial parallelism, but the complexity of the processing in a particular area must be limited because boundary effects reduce the relative speed up as the number of processors increases. When the algorithm can be divided into a number of distinct temporal steps, an alternative solution is to use pipelining, whereby each processor applies a small step of the algorithm to the entire image, and passes the results on to the next. A pipeline architecture possesses several advantages. First, the inefficiencies introduced by boundaries are removed, both in execution and coding. Secondly, modules which are useful in multiple applications can be utilized by different pipelines. Thirdly, pipelining is more economical in the communication links it uses on source and sink processors. However, throughput is limited by the slowest process in the pipe, and if 25 Hz video-rate operation is to be maintained, each processor in the pipe can take up to 40 ms only to process its step. Thus the computational load needs equalizing by the programmer. A further disadvantage, particularly in the early stages of the pipeline, is that a large volume of data needs to be communicated in and out of each processor.

To involve a large number of processors, but eliminate long pipelines, we have combined pipelining and spatial parallelism in a common structure, called a pipe-group, around which all our algorithms are built. A pipe-group consists of a group of processors arranged into one or more pipelines, each of which will typically (for low-level vision algorithms) process a part of the im-
age. In normal running (figure 4a), data enters the head processor, either from another processor or via an external interface such as a frame-grabber, and acts as an data server for the pipe-group(s) connected to it. It receives requests for (possibly subsampled) image windows from the pipelines and thereafter obtains these windows from the incoming data stream and sends them down the pipelines. The head process is thus independent of the algorithms requesting visual data from it, allowing the same process to be used for every pipe-group. The head processor splits the data between the pipelines in the group, each pipeline processor executes its part of the data processing on the input data stream and passes it on to the next processor until the tail processor merges the incoming intermediate result data, performs the final stage of the algorithm and outputs the results.

The software library supporting the implementation of pipe-groups is designed (i) to maximize the throughput of each pipeline through synchronisation of concurrent input, processing execution and output processes; (ii) to allow dynamic re-initialisation of all processors in a pipe-group with minimal interruption to normal data-flow; and (iii) to permit modification of the pipe-group structure in specific implementations without compromising the performance or ease of use of the existing structure.

Figure 4: (a) The combination of spatially parallel and pipelined architectures in the pipe-group architecture used in this work. (b) Steps for stopping data flow in a pipe-group.

3.1 Pipe-group stopping and re-initializing

Point (ii) above is worth pursuing. While testing new real-time applications it is very useful to be able to re-initialize at any time, avoiding the time-consuming process of rerunning and possibly recompiling the program. This allows the values of any variable parameters to be tuned online to optimize the performance of the algorithms. In our gaze control research we have also found this capability useful since it allows controlled switching between different modes of behaviour.

The procedures for initialization and dynamic re-initialization are identical, and may take the
form of freeing arrays and allocating new ones, and/or setting new values for parameters. Data-flow must be suspended during this time, because re-initialization may well change the size (though not generally the type) of the data packets sent down the pipelines, in which case space for the input and output data must be re-allocated. Pipe-groups may be re-initialized independently, but not individual processes within pipe-groups: stopping data flow to one processor will of necessity stop data-flow to all others in the pipe-group, since the pipe-group is synchronized at head and tail.

Stopping a pipe-group to prepare for re-initialization is accomplished by the following stages, illustrated in figure 4(b). There is no synchronisation between pipelines except at head and tail, so the numbers should be thought of as referring to the order for each pipeline independently. At step 1 the tail processor receives new parameters, usually from user input via the host machine. Signals are sent up the pipelines while normal data-flow continues down the pipelines (steps 2 to 5). The head processor receives the signal and returns stop signals down the pipelines (step 6). The head processor does not send more data until it is explicitly sent a request. Each processor recognizes the stop signal, passes it on to the next processor and enters re-initialisation mode (steps 7 to 9). The tail processor receives the stop signal from all pipelines, indicating that data-flow has stopped.

To re-initialize, the tail processor sends the head and each pipeline processor their new parameters in a single structure, prefixed by an identifying tag and the length of the structure. If a processor receives a tag it does not recognize, it simply reads the length and the structure data and passes it on to the next processor up the pipeline. This allows modularity to be maintained, since no processor needs to know which processes it is connected to (and hence what parameters they need), they only require their own input and output data formats. When a processor recognizes its tag, it reads the new parameter data, re-initializes itself, and replies down the pipeline indicating success or failure. These replies are passed on to the tail processor. In the event of failure, new parameters may be tried and the failed processes initialized again. Processors can be re-initialized in any order and any number of times. Upon successful re-initialization of each processor in the pipe-group (including head and tail processors), the tail processor recommences data flow by sending up the pipelines a restart signal. Every pipeline processor recognizes the restart tag as the end of the re-initialization and prepares to receive data again. When it reaches the head processor data flow restarts.

3.2 Pipe processes

The restricted topology of the pipe-group allows a fixed number of process types to be defined for each processor, removing the design burden from the vision researcher wishing to implement a new algorithm.

The head processor needs run only one process, as follows:
HEAD
1 Accept stop signals from the pipelines and return them.
2 Receive new parameters from the pipelines, attempt re-initialization and return success/failure.
3 Receive restart signal from all pipelines.
4 Start sending data down pipelines. Check whether stop signals are being sent, and if so return to step 1.

Four processes run on all pipeline processors, i.e. those processors in the body of the pipe-group. Their tasks are:

INPUT
1 Input data or stopped signals; pass them onto EXEC.
2 If stop signal was sent, wait for restart signal from EXEC before returning to step 1.

OUTPUT
1 Receive data buffer or stop signal from EXEC and output it to next processor in pipeline.
2 If stop signal was sent, wait for restart signal from EXEC before returning to step 1.

EXEC
1 Read from INPUT process.
2 If data, execute relevant stage of algorithm and pass new data to OUTPUT.
   Else if stop signal, transmit it to OUTPUT and RE-INIT.
   Wait for restart signal from RE-INIT; send restart signals to INPUT and OUTPUT, and continue at step 1.

RE-INIT
1 Wait for stop signal from a processor tail-wards down the pipeline.
2 Relay it up pipeline, and wait for stopped signal to be read and returned by EXEC.
3 Re-initialize processor using pipeline channels.
4 Wait for restart signal from tail-wards input channel, signals EXEC to restart, and continues at 1.

Their dependencies are sketched in figure 5a, and the normal running mode is illustrated in figure 5b. Data flows from INPUT to EXEC to OUTPUT, while stop signals coming from the tail are relayed on without causing any effect, until they are returned by the head and read by INPUT. Since INPUT and EXEC run concurrently, at least two sets of pipeline input data buffers must be used, one for input and one for EXEC, so that pipeline data input does not occur into an area of memory that EXEC is using. More than one input data buffer will also be required if INPUT and EXEC processes share memory, or if input data is retained by EXEC. An example of the latter is the visual tracking problem, where features are matched between the current and previous images, so two feature map buffers must be maintained. This is the method we use in our corner tracking algorithm [7]. For three-frame matching three buffers would be required, etc. At least two output
buffers are also required, since EXEC and OUTPUT run concurrently. RE-INIT waits for EXEC to send it a stopped signal. After it has received it and sent it on towards the tail, the situation is as shown in figure 5(c). RE-INIT then has exclusive use of the pipeline channels for the purpose of pipeline re-initialization.

On the tail processor, parallel processes are created to handle concurrent I/O, algorithm execution, and the re-initialization process. These are shown in figure 6(a) and their rôles are as follows:

**INPUT**
1 Read result data or stopped signals and pass them onto MERGE/EXEC.

**MERGE/EXEC**
1 Merge intermediate result data into single structure.
2 Execute final stage of algorithm on all data.
3 If pipelines have stopped, send signal to RE-INIT process.
RE-INIT
1 On request from external process, re-initialize entire pipe-group by sending stop signal and reinitialization data head-wards up all pipelines in pipe-group.

In figure 6(b) is shown the channel usage during normal execution. EXEC will typically output its results to an external process. Also the RE-INIT process, usually upon user request from the host, may request EXEC to output results (for instance) and receive an acknowledgement upon completion of the request, using the channels marked “request” and “ack” in figure 6(b). Both the EXEC and RE-INIT processes can communicate with the outside world, so care must be taken that they do not try to use the same channel simultaneously. Use of the “request” and “ack” channels ensures exclusive use of external channels. As with pipeline processors, at least two input buffers are required so that the INPUT and MERGE/EXEC processes can run concurrently.

Figure 6: (a) Processes on tail processor and their inter-communication. The remaining figures show Channel communication during (b) normal data-flow. (c) pipe-group stopping procedure, and (d) re-initialization. Channels marked with dotted lines are not used.

The pipeline input channels are used by both INPUT and RE-INIT processes, but simultaneous use is prevented since RE-INIT only uses the channels during re-initialization when data-flow has been stopped. The stopping procedure uses the channels as shown in figure 6(c).
Once MERGE has received the stopped signals from all the pipelines and signalled RE-INIT, re-initialization occurs as shown in figure 6(d). Parameter structures are sent up the pipelines and replies are returned. Restart signals are then sent by RE-INIT both to the pipelines and to MERGE/EXEC, and we return to the situation shown in figure 6(b). Finally a reply is sent by RE-INIT to the external process which started the re-initialization.

### 3.2.1 Pipe-group extensions

As mentioned above, connections of the head and tail processors to the outside world are not restricted. The EXEC and RE-INIT processes on pipeline processors may also communicate in arbitrary ways with other processors, but we have found it beneficial to include certain external links as part of the pipe-group protocol. If an external processor is receiving output data from EXEC, it may require re-initialization when the pipe-group itself is re-initialized. Thus RE-INIT will have to prevent EXEC using the external I/O channels during re-initialization just as it does with the pipeline channels. The external processor may be another processor in the same pipe-group or a processor not part of any pipe-group.\(^\text{1}\) We have extended the standard pipe-group scheme described above to allow an arbitrary number of extra input and output channels. Extra processes are created to allow concurrent I/O using them. These processes are identical to the INPUT and OUTPUT processes in figure 5. The arrangements with one input and one output channel respectively are shown in figures 7(a) and (b). The extra output channels require an input channel for replies to re-initialization attempts.

![Figure 7](image.png)

Figure 7: (a) Pipeline processor with extra input channel. (b) Extra output channel. The OUTPUT process in (b) is not shown for the sake of clarity.

Alternative pipeline structures can be realized within the pipe-group framework. For instance, if it is required that a pair of pipelines feed their results into a single pipeline, the solution is to create two pipe-groups and join them end-to-end. The lower pipe-group contains a single pipeline, and its head doubles as the tail of the upper pipe-group with two pipelines. Re-initialization of the

\(^{1}\text{Pipe-groups should not in general be linked, because of the lack of synchronization.}\)
lower group will spread to the upper pipegroup via the shared processor. This is the arrangement we use for real-time feature tracking illustrated in the next section.

### 3.3 The user interface

The real-time vision machine is hosted by a Sun workstation, and parameters required for initialization and re-initialization are supplied via an X-windows graphical front-end. This also allows display and analysis of results, as illustrated in figure 8. All the vision algorithms maintain a ring-buffer of the most recent few seconds worth of results at the tail processor of each pipe-group. This can be requested by the host via the root transputer, and then viewed off-line. During real-time operation, the host is able to display the results, but this is done on a request basis, so that the host never holds-up the real-time network. When a number of algorithms are to communicate with the host in this way, intermediate multiplexer processors are required because of the limit of four links per transputer.

![Figure 8: An example of the user interface used for stereo tracking.](image)

### 4 Active vision example: tracking using stereo affine transfer

As an illustrative example of active vision research implemented using the pipe-group architecture, we describe simultaneous stereo tracking and structure recovery of an object moving in the environment. The method derives from work [15, 16, 17, 18] which showed that structure can be recovered up to a 3D global affine transformation, and sufficient to compute images from arbitrary novel viewpoints. This last process, known as *transfer*, is used to give a stable object-based position for fixation. The theory of monocular affine transfer was detailed in [7] and that for stereo transfer in [8]. For brevity and clarity here we will outline the method using the minimal number of point correspondences, namely four points in three images.
4.1 Theory review

Where scene relief is small in comparison with depth, it is valid to assume that images are formed under an affine camera projection

\[ \mathbf{x} = \mathbf{mX} + \mathbf{t} \]

where \( \mathbf{x} \) is a 2\times1 image position vector, \( \mathbf{m} \) is a 2\times3 matrix, \( \mathbf{X} \) is a 3\times1 world position vector, and \( \mathbf{t} \) is a translation vector. Consider a set of four points, \( \mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3 \) and \( \mathbf{X}_4 \), in general (non-coplanar) position on an object (see figure 9). The four points define a basis set \( \{ \mathbf{X}_{21} = \mathbf{X}_2 - \mathbf{X}_1, \mathbf{X}_{31} = \mathbf{X}_3 - \mathbf{X}_1, \mathbf{X}_{41} = \mathbf{X}_4 - \mathbf{X}_1 \} \) in which the position of any scene point \( \mathbf{X}_g \) — here particularly the scene point we wish to fixate upon — is defined by three affine coordinates, \( \alpha, \beta, \gamma \):

\[
\mathbf{X}_{g1} = \mathbf{X}_g - \mathbf{X}_1 = \alpha \mathbf{X}_{21} + \beta \mathbf{X}_{31} + \gamma \mathbf{X}_{41} .
\]

These coordinates are invariant to the affine projection in the sense that the projected coordinates \( \mathbf{x} \) of the point \( \mathbf{X} \) are the same linear combination of the projected basis vectors:

\[
\mathbf{x}_{g1} = \alpha \mathbf{x}_{21} + \beta \mathbf{x}_{31} + \gamma \mathbf{x}_{41} .
\]

Consider a temporal sequence of stereo images. At frame time \( i \), given left and right views of the four basis points \( \{ \mathbf{x}_{1,2,3,4}^L \} \) and \( \{ \mathbf{x}_{1,2,3,4}^R \} \), we can solve for the affine coordinates \( \alpha, \beta, \gamma \) of the 5th point \( \mathbf{X}_g \) by solving the over-constrained system of linear equations

\[
\begin{bmatrix}
\mathbf{x}_{g1}^L \\
\mathbf{x}_{g1}^R
\end{bmatrix} =
\begin{bmatrix}
\mathbf{x}_{21}^L & \mathbf{x}_{31}^L & \mathbf{x}_{41}^L \\
\mathbf{x}_{21}^R & \mathbf{x}_{31}^R & \mathbf{x}_{41}^R
\end{bmatrix}
\begin{bmatrix}
\alpha \\
\beta \\
\gamma
\end{bmatrix} .
\]

Having computed the affine coordinates of the gaze point \( \mathbf{X}_{g1} \), it is straightforward to determine its projection in either left or right images in the next pair in the sequence, given the projected positions of the reference basis points in that third view:

\[
\mathbf{x}_{g1}^{L,R_{i+1}} = \alpha \mathbf{x}_{21}^{L,R_{i+1}} + \beta \mathbf{x}_{31}^{L,R_{i+1}} + \gamma \mathbf{x}_{41}^{L,R_{i+1}} .
\]

At first, it appears that the same set of four points have to be matched throughout an image sequence. However, having performed transfer into the stereo pair \( i + 1 \), the fixation point can if required be described with respect to another basis set for transfer into frame \( i + 2 \). Thus the minimum that is required is that there is one set of four spatio-temporal corner correspondences between each successive stereo pair. To achieve greater immunity to noise it is in fact possible to use all the available correspondences within a least squares framework [7, 8]. Note too that the gaze point need not be, and typically will not be, a physical feature on the object — indeed it can lie on part of the object which is occluded.
4.2 Implementation

The network of eleven T805 transputers required to implement this stereo affine transfer is shown in figure 10, along with typical times for process and inter-process communication. The corner detection for the left and right images is dealt with by two separate pipe-groups with two frame-grabbers acting as heads of the groups. The pipe-groups for each image recombine at the temporal corner matching transputers which act as the tail of each pipe-group. The temporal matching transputers also act as the head of two short pipe-groups (each of one pipeline) which are combined at the stereo corner matching transputer. The stereo matching transputer acts as the head of the final pipe-group which consists of a single pipeline whose tail is the structure transputer.
The cameras supply synchronized stereo images which are captured by frame-grabbers which extract a 72 × 40 pixel central (foveal) portion, encompassing a field of view of some 8° × 6°, from each. This narrow field of view helps ensure the constraint of relatively small relief required for the affine projection approximation.

The corner detection [19] and temporal matching are described in detail in [7]. The stereo matching uses approximate epipolar lines to constrain the search for potential matches, using correlation to test the match quality. A disparity gradient check is used to reject gross outliers at this stage.

The two T805s labelled “Temporal matching” in figure 10 perform the temporal corner matching and also compute the monocular fixation point transfer which is used for independent tracking in the event that stereo transfer fails. The “Stereo matching” T805 computes stereo correspondences, and the “Affine transfer” transputer transfers the fixation point in left and right cameras using the stereo affine transfer algorithm [8]. The demands required to centre the fixation point in the image are sent directly to the high-level gaze controller from the Affine transfer transputer, outside the pipeline protocol. The “Structure” transputer computes explicit structure and derives its convex hull using the Quickhull algorithm [20]. The Structure transputer also communicates with the host Sun workstation and controls all the pipelines. The Sync processor performs housekeeping tasks, ensuring that the images remain synchronized and providing time-stamping information for the frame-grabbers.

4.3 Experimental results

We present the results in three parts. First we demonstrate the spatio-temporal matching process used to obtain data for transfer and structure recovery. This information was obtained via the Structure transputer, but was produced by the first four pipe-groups described above. Then we will consider results from the affine stereo tracking algorithm, showing recovered 3D trajectories of the tracked object in an approximately Euclidean world frame. This information is produced by the Affine transfer transputer in the middle of the final pipe-group. Finally we will show results from the recovery of the affine structure of the tracked object over time, which are calculated on the tail transputer of the final pipe-group.

Spatio-temporal matching and transfer are illustrated in figure 11 using stereo data cut from a sequence where the head tracks a radio controlled buggy driven around the laboratory floor. In these images only the corners that are spatio-temporally matched and thus used in the computation of affine transfer and structure are shown, and the motion vectors are shown, not the the disparity vectors. The fixation point calculated from stereo transfer is shown as the white cross, and we note that there is no corner feature associated with it.

Another way of representing the fixation process is to recover the 3D trajectory of the fixation point in a Euclidean world frame. The buggy was driven in a circle, with radius approximately 1 metre and centre some 3 metres from the camera platform. Figure 12 shows the track obtained, viewed from the camera position, from above and from the side.
Figure 11: Spatio-temporally matched corners and the recovered fixation position through an extended sequence.

Figure 12: The trajectory recovered from stereo transfer, as viewed from the camera, from above, and from the side.
In figure 13 we show two examples of the structure recovered from the moving buggy during a sequence. The structure is shown on the left as a set of points with the wire-frame convex hull superimposed and on the right with the shaded convex hull. The sparse structure is not recognizable of itself, but the outline of the convex hull superimposed on the images allows one to match up leftmost and rightmost, and nearest and furthest points.

Figure 13: The structure recovered for the buggy at the start and some way in to a sequence.

Earlier, in figure 10, we showed the typical times for the various processes and inter-process communication. In Figure 14 we demonstrate that detailed process timing is available for each frame. Timings for three frames of the sequence passing through the pipe-group are given. The wide bars are interprocess communication, while the narrow bars labelled C, T, S, A and H denotes times for corner detection, temporal matching, spatial matching, affine transfer and hull calculation, respectively. Notice that the spatial matching process first receives raw corners from
the spatial matching process, and later received matched corners for transmission on to the affine transfer process. Notice too that processing times in frame 3, where more corners were found, were considerably longer.

Figure 14: Process times (narrow bars) and inter-process communication times (wide bars) for three successive frames of the affine transfer algorithm. C, T, S, A and H denote corner detection, temporal and spatial matching, affine transfer and hull calculation.

5 Conclusions

Our experience with parallel coding of active vision algorithms using both the tools described in this paper and more ad hoc solutions suggests to us that the former have significantly reduced the time taken to create new applications. The programmer must of course provide functions to perform data input, processing, output and the algorithm-dependent parts of the re-initialisation sequence, but a large part of the programming burden in developing algorithms for the real-time visuo-control loop is removed. The necessary synchronisation between the processes (to avoid deadlock or multiple use of a channel), concurrency of I/O and data processing, and re-initialization sequencing are performed automatically.

The design of the system has taken into account that research into active vision imposes methods of working not encountered in offline vision processing. We are more concerned with results measured in terms of visual behaviours rather than in terms of visual data per se. For example, it makes little sense to provide time-consuming access to stages of algorithms not actually generating demands in the gaze control system. These intermediate stages are therefore “hidden”. Indeed downloading of imagery and visual data for off-line analysis is rarely required.

The modularity of the pipe-group architecture has allowed the vision system to expand as new competences are added, without unduly increasing the complexity of the overall system. Nor has the imposition of structure particularly compromised the flexibility that is such a desirable feature for a research system. When required, we have found no difficulty extending the pipe-group architecture, providing, as described in the paper, extra connections within a pipe-group or connections to isolated processors outside a pipe-group.
References


