Chapter 5

Implementation Aspects

The application, called OmniTracking, was written using the C++ programming language and runs on a PC-based system with Microsoft's *Windows 2000* as the operating system. An Intel Pentium processor-based machine is used, to benefit from its *MMX technology*. The application also makes use of Intel's *OpenCV* computer vision library.

5.1 Requirements of Computer Vision Applications

Generally, computer vision applications put great computational demands on the processor and also have high memory requirements. Processing rates must also be sustained over long time periods of time, to be able to process video streams in real-time. Another requirement is for fast I/O when acquiring data from a video camera[1]

In the early days of computer vision, applications were traditionally implemented on dedicated DSP[2]-based machines. But nowadays general-purpose PCs are becoming more powerful, are widely available, relatively inexpensive and are supported by a large variety of software tools and hardware peripherals (such as cameras, frame grabbers, etc.). In addition, the processors of modern-day PCs are starting to contain DSP-like functionality in them (for example, Intel's MMX technology). So, general-purpose PCs are being used more frequently for computer vision applications.

[1] In the case of this application, the video streams are not acquired in real-time, but offline video streams are used. This shifts the I/O demand on to the disk unit, which needs to be able to achieve the necessary sustained data throughput. For example, in the PETS-ICVS dataset, the average size of each (compressed) frame is 46Kb. At 24 fps, this means a disk throughput of ~1Mb/sec. – well within the reach of modern PCs.

[2] Digital Signal Processors

The system used for this thesis (for development and testing), consists of a PC with an Intel Pentium III-E 550MHz processor and 384Mb of main memory.

5.2 MMX Technology

Intel developed the *MMX technology* under the Native Signal Processing initiative for its Pentium line of processors [INTEL99]. The word MMX stands for "*m*ulti-*m*edia e*x*tensions", and as the name suggests, it was designed to improve the performance of media-based (audio, video) applications, which have a lot in common with computer vision applications. A lot of algorithms in computer vision are pixel-based algorithms, where the same operation is done on all of the pixels independently in a pixel-by-pixel fashion – it is these *inner loops* that do the pixel-by-pixel processing that cause the performance bottleneck.

MMX uses a *single-instruction multiple-data* (*SIMD*) approach to exploit data parallelism. It packs several pixel values into one register and then performs operations on these numbers in parallel. Assuming 8-bit pixels, 8 pixels could fit into one of the 64-bit MMX registers and these could be processed in one cycle – a performance increase of 8 times [INTEL99]. In reality, the performance increase is less, because the processor has to spend some time packing and unpacking the registers. [TALL99] reports increases ranging from ×1 to ×4.7 over a different range of signal processing algorithms.

Other functionality offered by Intel's MMX technology that can benefit computer vision algorithms is saturation arithmetic (useful for thresholding operations) and a multiply-and-accumulate operation (useful for convolution and morphology) [LECK98].

But one limitation of the current MMX implementation is that mixing MMX and floating-point operations is very costly. The reason is that no new functional unit (register set) has been added to the Pentium processor for MMX. Instead MMX shares registers with the floating-point unit ([INTEL99] cites backward compatibility reasons). The operation of 'switching' the registers from floating-point to MMX and vice-versa can take up to 50 clock cycles [LECK98]. Trying to keep integer-based calculations

separate from floating-point ones in a program, helps to reduce the number of switchovers.

For the OmniTracking application, most of the repetitive floating-point calculations have been implemented using fixed-point integer arithmetic to avoid this problem and because in general integer math is faster than floating-point math.

Starting from the Pentium III processor, Intel extended the MMX technology from integer multi-media data to cover also floating-point multi-media data. This was marketed under the separate name of *Streaming SIMD* [INTEL99]. And it was implemented properly by using a separate functional unit. Apart from Intel, other microprocessor manufacturers, like AMD and Cyrix, were quick to implement similar MMX technologies in their processor families.

Unfortunately, it's not possible for the processor or a compiler to automate how pixels are packed and unpacked from the MMX registers, as context information about the algorithm is needed. To benefit from this technology, one has to hand-code the algorithm, normally by using assembly instructions (as done by [LECK98] in his tests). And existing C++ compilers do not have the necessary language constructs that map to MMX operations (except for Intel's C++ compiler, with its *intrinsics* feature [INTEL99]). But libraries of MMX-optimised functions are starting to become available, such as Intel's *Performance Library Suite* (*PLS*) and *OpenCV*.

5.3 OpenCV Library

The OmniTracking application makes use of the *OpenCV* library (version Beta 3.1). This is an open-source library from Intel, containing many image processing and computer vision algorithms, written to take advantage of the MMX technology available in the Pentium family of processors [OPENCV].

Some of the different types of functions in the OpenCV library include: low-level data structures and image handling utilities, matrix operations, camera calibration functions, object recognition functions, etc. There are also separate support modules in OpenCV that provide very basic GUI and video acquisition functionality (called

HighGUI and *cvCam* respectively), a scripting language (called *Hawk*), and several miscellaneous tools and utilities.

If available, OpenCV makes use of libraries from Intel's *Performance Library Suite* (*PLS*) for some of its internal functions – if not, it defaults to its own (less efficient) implementations. The OmniTracking application also calls some of these libraries directly (for example for reading JPEG images from disk). Unlike OpenCV, the PLS libraries are not open source and are free only for educational and non-commercial use. The PLS libraries used are:

- Intel *Image Processing Library* (*IPL*) – provides the low-level image functions,
- Intel *Image Processing Primitives* (*IPP*) – provides low-overhead versions of *IPL* functions,
- *Intel JPEG Library* (*IJL*) – provides JPEG de-compression and file reading/writing.

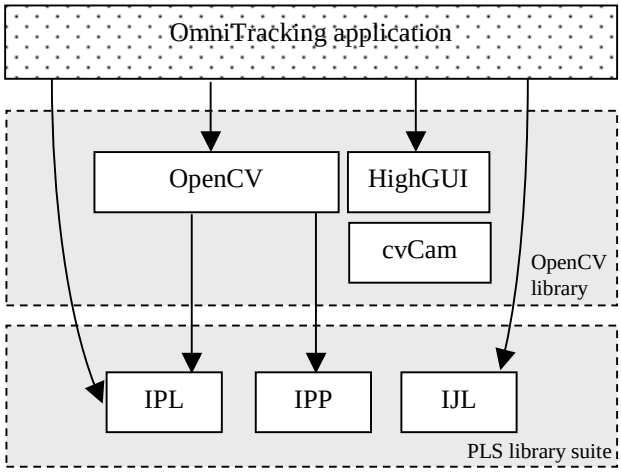


Figure 5.1: Libraries

As with all open source projects, OpenCV is always in a state of flux, some of the functionality it provides is of an experimental nature, and the library can be viewed as a loose collection of modules, tools and functions.

In general, the main advantages of using OpenCV are:

- the code makes use of Intel's MMX technology,
- the library auto-detects the type of processor and executes code customised to that processor,

- it provides a rapid environment for prototyping computer vision algorithms (especially with the addition of the support modules and the scripting language),
- the library is open source,
- OpenCV has an extensive and active community of users that interact through a dedicated mail group.

For this thesis, the main benefit of using OpenCV (and its related libraries) was in not having to re-create the basic structures for image handling and the availability of several low-level operations like edge detection, thresholding, etc. Wherever these functions are used in the application, a mention is made in the relevant parts of this document. In some other cases, although OpenCV contains the required functions, they were not used and the algorithms were 're'-implemented for this application – for reasons such as to use masks to skip pixels, to use fixed-point integer math rather than floating point, etc.

5.4 Windows and Threads

Most computer vision applications can be classified as *soft real-time systems*, meaning that the system must produce results within defined time constraints with as little latency as possible [CUTL99][3]. This depends mostly on the operating system being used.

The OmniTracking application runs on a system that uses the *Windows 2000* operating system. This is a general-purpose operating system with the aim to maximise the aggregate throughput of a system and achieve a fair sharing of resources [MSDN99]. It was not designed with the intention to be a real-time operating system, and so is not able to:

- ensure low-latency and bounded response times,
- provide predictable time-based scheduling (deadline-based), and
- provide fine granularity clock and timer services.

But Windows contains mechanisms that allow some form of real-time behaviour:

[3] In comparison, hard real-time systems must have zero latency, since a late response can be catastrophic.

- mainly through the use of threads,
- thread synchronisation mechanisms, and
- by setting process and thread priorities.

A Windows process can be assigned to one of a set of priority classes, and within each class, threads can be further given different levels of priority. The OmniTracking application makes extensive use of threads and thread synchronisation mechanisms, and to prevent other "less important" processes from pre-empting the application, the process class of the program has been assigned a high priority level.

But the threads within the program were kept at the same priority level as this gives better response times to user input and most of the threads are synchronised on each other – so eliminating the possibility of one thread excluding the others from running for long periods of time.

The reason why Windows 2000 cannot ensure predictable time-based scheduling of threads has to do with hardware interrupts. Interrupts always have higher priority than applications (even mouse driver interrupts) and when an interrupt occurs, the interrupt handler puts the request on the DPC (Deferred Procedure Call) queue. When the Windows scheduler is about to switch from executing one thread to another, before starting the second thread, it executes all the requests that have accumulated on the DPC queue [JONE99]. This can give rise to unbounded time delays since some activities, like network I/O, can cause many DPCs to accumulate. In addition, most drivers do their (potentially lengthy) background (housekeeping) work through DPCs.

Another factor that causes this unpredictability is the fact that Windows does not have a priority inheritance mechanism. A thread that makes a call to a service offered by a lower-priority thread can become blocked until the lower-priority thread runs [RAMA98]. Some operating system function calls are executed by normal-priority threads within Window's kernel – a high priority user thread that calls such functions is slowed down by these system calls.

Despite these problems, in general Windows is able to get near good scheduling predictability within deadlines of 10 to 100 milliseconds [RAMA98]. But it may suffer from the occasional unbounded responses ('freezing' for a few seconds) if large DPC queues or system activities kick-in.

The other issue that is relevant to the OmniTracking program is that Windows' timers are not of high enough accuracy. The so-called *multimedia timers* of Windows have an accuracy of up to 1 millisecond (ms), but in practice these timers are not constant and their beat can vary from 1 to 15 ms (this has to do with the accuracy of the clock chipset and the handling by the Windows kernel of clock interrupts) [JONE99].

To achieve constant processing rates (in frames per second), the OmniTracking application makes use of the Windows timers for thread scheduling (because of the advantage that the thread is blocked until triggered by the timer, utilising no processor time). But it also uses the high performance clock available on the Intel processor to periodically adjust the timer's period. This high frequency clock runs at the processor's internal speed with nanosecond accuracy. Every 30 firings of the Windows timer, the processor clock is queried and if there is any time discrepancy, an allowance factor is added to the Windows timer. From actual test runs, this method was found to give accuracies near the 1 millisecond range.

5.5 Application Structure

The main design requirements for the OmniTracking application were to:

- keep the computer vision algorithms as separate from the GUI code as possible, and to
- adopt a flexible and modular approach that allows different computer vision algorithms to be 'plugged in' and experimented with.

The program structure adopted is the *pipeline model* – generally, input data (a video stream) is fed into one end of the pipeline, and at the other end, output is produced. The pipeline is made up of a set of modular algorithms communicating with each other by passing images and data to each other. Another way to view the pipeline is as a directed graph, where the pipeline is the graph and the individual modules of the pipeline are the *nodes* of the graph.

The use of a graph-based approach is a popular one in image processing applications (and computer graphics) because of its flexibility in allowing different arrangements, modularity, prototyping ability ("plugging in" different nodes), and allows for the creation of development environments where the specification of the graph structure

is done graphically. Examples include *Khoros*, *MIRTIS* [NICH98], and *DirectShow* [MSDN99].

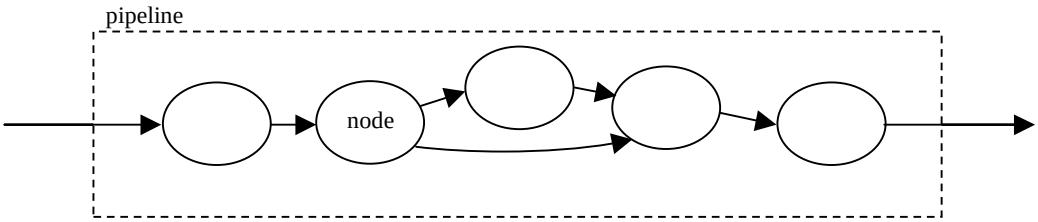


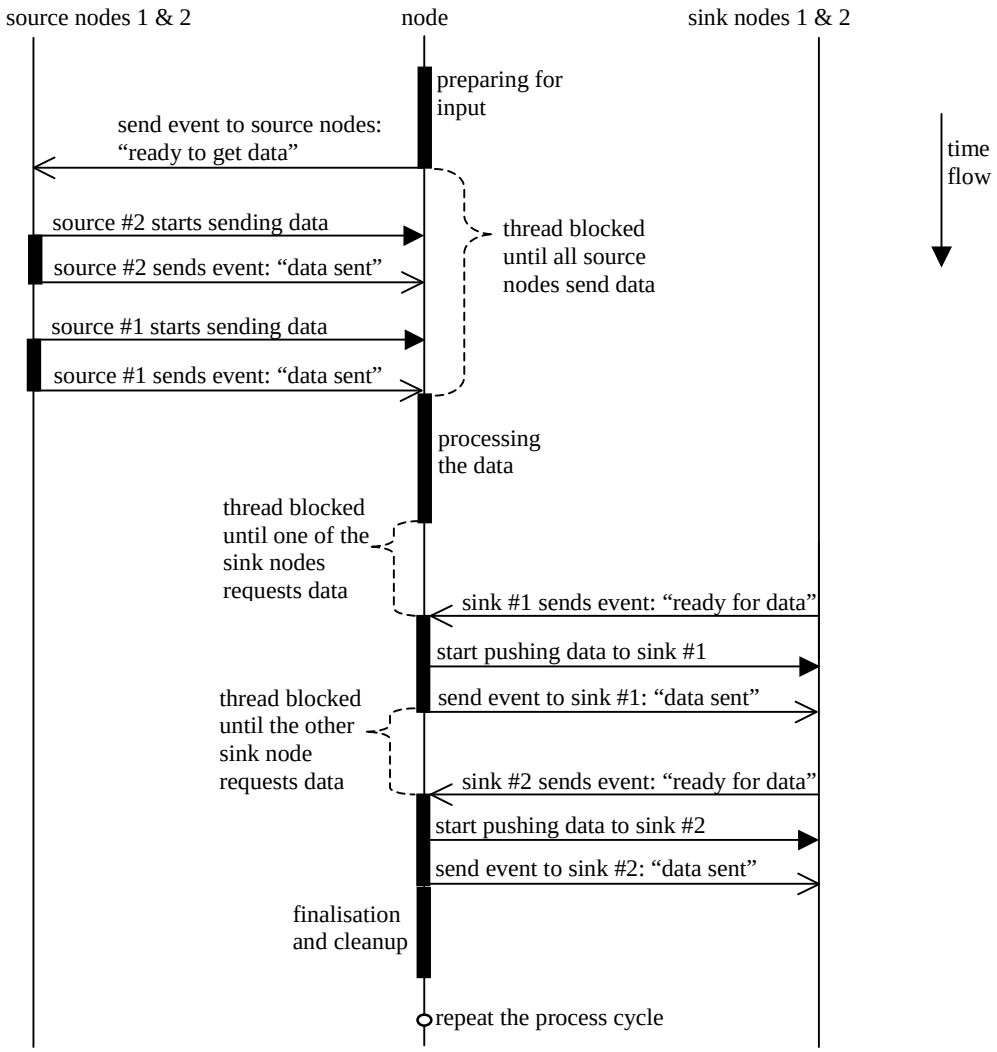
Figure 5.2: Pipeline Model

For the OmniTracking application, it was decided not to use any of the existing architectures, but instead develop a new set of classes to implement a simple (lightweight) pipeline model.

In this model, a node can accept input data from a number of *source nodes* and its output goes to a number of *sink nodes*. When a node is created, it spawns a thread and this thread does the following task repeatedly: it waits for input from the source nodes, then does some processing, and when finished, pushes the output data to the sink nodes when they are ready to accept data. This process is illustrated in Figure 5.3. When a node's thread is waiting for input or waiting for sending the output, thread synchronisation events are used – this causes the thread to block (enters a wait state) and so consumes no processor time until it is awakened again when the wait condition is satisfied. The ideal performance (and maximum processor utilisation) is achieved when the node threads don't spend any time in wait states.

The program was written in C++, and one of the great features that the C++ language offers is *class templates* (also called *generic programming*). The code definition of the pipeline classes does not specify what type of the data is passed from one node to another – it is just an abstract (generic) template parameter that will only be known at object instantiation. The use of templates does not incur any run-time overhead or suffer from loss of type safety from the compiler [STRO94 §15.11]. This makes the pipeline implementation very flexible, generic (is not tied to this particular application) and can be easily ported to use other image libraries or data types.

In the case of the OmniTracking application, the pipeline nodes use the `IplImage` data structure of OpenCV. When passing image data from one node to another, it is



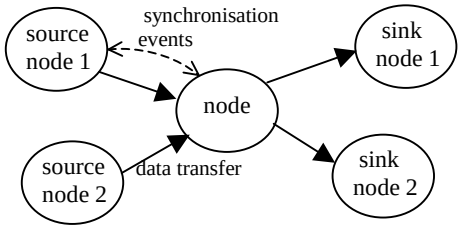


Figure 5.3: Node thread execution timeline

not possible to 'copy' the data (for obvious performance and memory reasons). Instead a single memory copy of a particular image is created, and a reference to it is passed from one node to the other across the pipeline. To know when an image is no longer being used by the nodes and can be safely deleted, a *reference counting* mechanism is used [MEYE96 pp.183-213].

When processing video streams, most of the images created and deleted by the pipeline nodes will be of the same type. Therefore to avoid the overhead of memory

allocation/deallocation, when an image is no longer in use (its reference count reaches zero), it is placed in a *memory pool*, where it can be re-used later on.

Figure 5.4 shows the pipeline used to implement the OmniTracking application. A normal issue that arises in these circumstances is the question of *granularity*: how to split the program into nodes. Ideally the choice should balance flexibility against overhead – the overhead in this case being the thread used by each node. Windows has a limited-size thread pool [MSDN99]. The program normally uses about 35 threads, which should not cause any problems.

In Figure 5.4, some nodes are marked with the symbol $^{1}/_{k}$. This means that these nodes (mostly sending output to GUI icons) run at a lower 'priority' than the other nodes. By priority, it means that they don't run every time they receive an image frame from their source nodes, but run every *k* frames, where *k* is set to 2×*fps*.

The node marked with the symbol `*`, will be instantiated either as `BlobTracker` node or as `ColourTracker` node depending on what value is configured in the initialisation file.

All nodes produce output that is *consumed* by either other nodes or GUI entities (icons, windows). In the case of GUI windows, the user may decide to close them or the windows may become hidden. It would be a waste of time, if the node in the pipeline continues producing the output data at the same rate as before. In these cases, the nodes (marked by $P\downarrow$ in the diagram) reduce their priority dynamically. These changes then cascade down the pipeline, as each node checks its sink nodes, finds which one is running at the highest rate and adjusts its priority accordingly.

Nodes can be added and removed from the pipeline dynamically while it is processing data. This had to be implemented to allow the user to create new perspective and panoramic virtual camera windows at run-time. When such a camera is created by the user, input to the pipeline is momentarily paused (to allow the pipeline to settle to a safe state), new versions of the nodes shown in block ① or block ② are created (dashed outline in Figure 5.4), and these are added to the pipeline.

Finally, Figure 5.5 shows where the pipeline fits in relation to the other modules of the program.

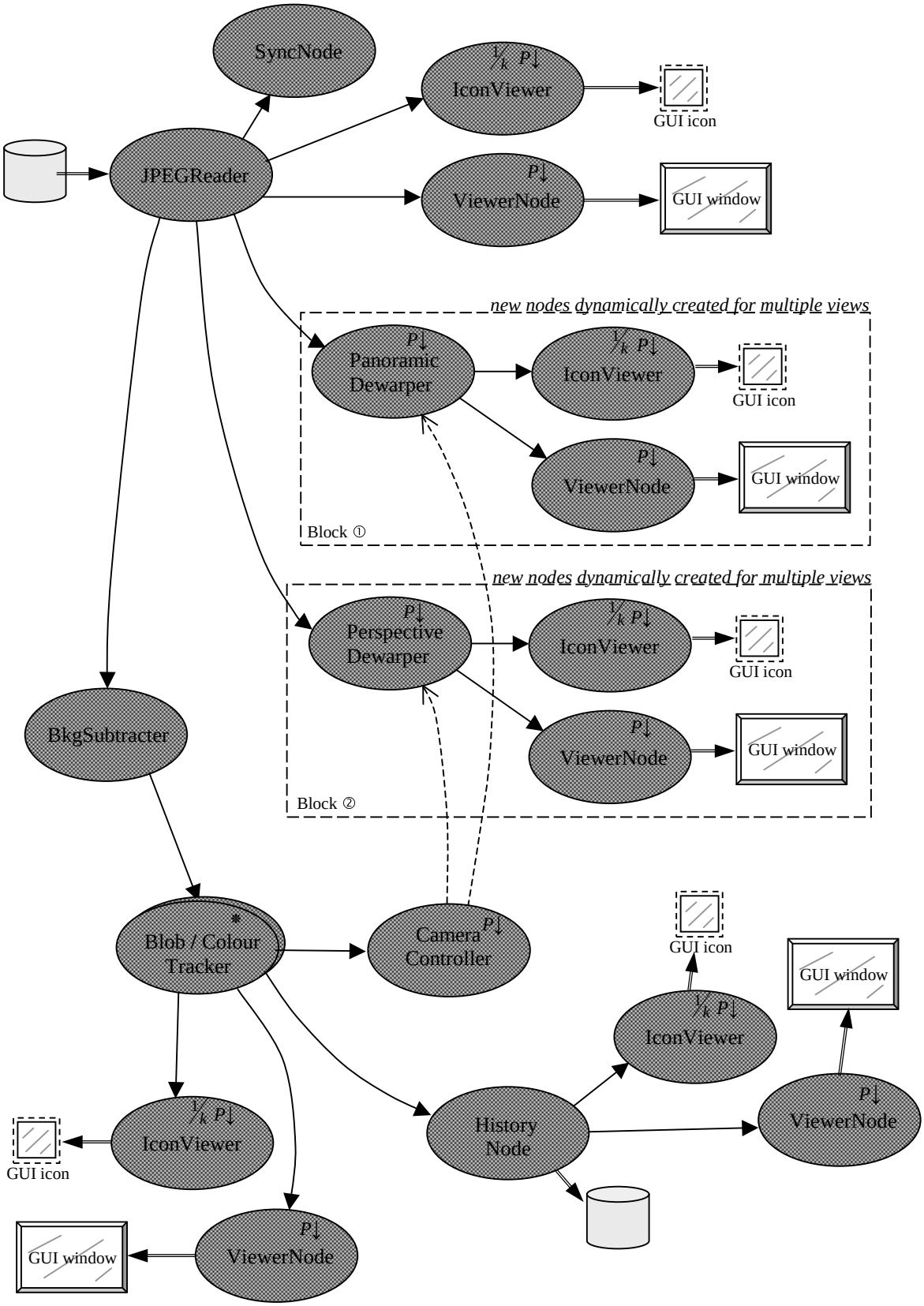


Figure 5.4: OmniTracking application pipeline
(symbols used in the diagram are explained in the accompanying text).

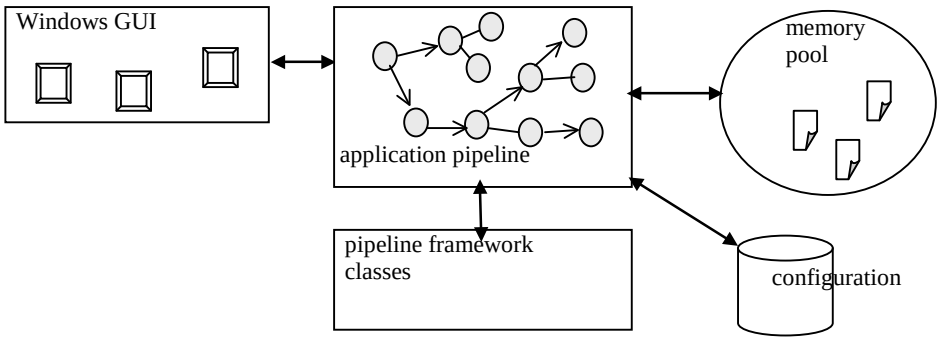


Figure 5.5: System diagram

The chosen pipeline model proved very useful for testing different modules, while the application was being developed. As regards to overhead, from some initial tests, the pipeline seems to be quite lightweight and does not cause unnecessary delays. More rigorous testing will need to be done to verify this and to measure other things such as *pipeline throughput* and *latency* (time delay between an image entering the pipeline and its results appearing at the other end) [NICH98].

There are however several limitations in the current model, including:

- no time synching between nodes (although the pipeline globally maintains the required fps, individual nodes at the same pipeline depth *k* may get out of synch),
- a node cannot select what data it wants from the source nodes (data subscription model),
- a simpler interfacing between the pipeline nodes and the GUI system (for example, combining the nodes `IconViewer` and `ViewerNode`).

5.6 Conclusion

This chapter starts with the general requirements of real-time computer vision applications. It then discusses the issues that are specific to the OmniTracking application, like Windows operating system issues and thread control. The application uses the MMX technology that comes with Pentium processors and this is discussed in this chapter together with the benefits that MMX offers to computer vision applications. The chapter finishes off by describing in detail the structure of the program and the processing pipeline model adopted.