Chapter 10
Towards OpenVL: Improving Real-Time Performance of Computer Vision Applications

Changsong Shen, James J. Little, and Sidney Fels

Abstract Meeting constraints for real-time performance is a main issue for computer vision, especially for embedded computer vision systems. This chapter presents our progress on our open vision library (OpenVL), a novel software architecture to address efficiency through facilitating hardware acceleration, reusability, and scalability for computer vision systems. A logical image understanding pipeline is introduced to allow parallel processing. We also discuss progress on our middleware—vision library utility toolkit (VLUT)—that enables applications to operate transparently over a heterogeneous collection of hardware implementations. OpenVL works as a state machine, with an event-driven mechanism to provide users with application-level interaction. Various explicit or implicit synchronization and communication methods are supported among distributed processes in the logical pipelines. The intent of OpenVL is to allow users to quickly and easily recover useful information from multiple scenes, in a cross-platform, cross-language manner across various software environments and hardware platforms. To validate the critical underlying concepts of OpenVL, a human tracking system and a local positioning system are implemented and described. The novel architecture separates the specification of algorithmic details from the underlying implementation, allowing for different components to be implemented on an embedded system without recompiling code.

10.1 Introduction

Computer vision technology is profoundly changing a number of areas, such as human-computer interaction and robotics, through its interpretation of real-world scenes from two-dimensional projections. However, building computer vision

Changsong Shen, James J. Little, Sidney S. Fels
University of British Columbia, Vancouver, BC, Canada
e-mail: csshen@ece.ubc.ca, little@cs.ubc.ca, ssfels@ece.ubc.ca

systems remains difficult because of software engineering issues such as efficiency, reusability, and scalability. Especially when computer vision technology is applied in embedded systems, in which real-time performance is emphasized, these issues become critical. In a field with as rich a theoretical history as computer vision, software engineering issues, like system implementation, are often regarded as outside the mainstream and secondary to the pure theoretical research. Nevertheless, system implementations can dramatically promote the progress and mainstream applicability of a field, just like the success of OpenGL promoted the development of hardware acceleration coupled with significant theoretical progress in computer graphics.

In current computer vision, there are three main system implementation issues. The first issue is *efficiency*. Most video operations are computationally intensive tasks that are difficult to accomplish using traditional processors. For example, for a single camera with a sequence of 24-bit RGB color images at a typical resolution (640×480 pixels) and frame rate (30 *fps*), the overall data volume to be processed is 27 MB/s. Moreover, even for a very low-level process such as edge detection, hundreds or even thousands of elementary operations per pixel are needed [7]. However, many computer vision applications, such as nearly all surveillance systems, require real-time performance, which means that the systems must interact with their environments under response-time constraints. Improving efficiency of the algorithms helps to meet these constraints.

The second issue is *reusability*. Hardware designers have developed various dedicated computer vision processing platforms [7, 9] to overcome the problem of intensive computation. However, these solutions have created another problem: heterogeneous hardware platforms have made it time-consuming and difficult (sometimes even impossible) for software developers to port their applications from one hardware platform to another.

The third issue is *scalability*. Recently, multi-camera systems have generated growing interest, especially because systems relying on a single video camera tend to restrict visual coverage. Moreover, significant decreases in camera prices have made multi-camera systems possible in practical applications. Thus, we need to provide mechanisms to maintain correspondence among separate but related video streams at the architectural level.

The open vision library (OpenVL) and its utility toolkits (VLUT) are designed to address efficiency, reusability and scalability to facilitate progress in computer vision. OpenVL, discussed in Section 10.3, provides an abstraction layer for applications developers to specify the image processing they want performed rather than how they want it performed. VLUT, discussed in Section 10.3.7, is created as a middleware layer to separate camera details, events management, and operating details from the specification of the image processing. By providing a hardware development middleware that supports different hardware architectures for acceleration, OpenVL allows code reuse without compromising performance. The novel software architecture separates the specification of algorithmic details from the underlying implementation, allowing for different components to be implemented on an embedded system without recompiling code. Further, when the embedded system's functionality changes, it is possible to change the drivers without changing

the code, allowing application programmers to match the amount of embedded processing with the needs of their image processing application without rewriting any of their application code.

The next sections are organized as follows. Section 10.2 provides an overview of related work addressing the issues we mentioned above. In Section 10.3, we discuss our implementation of OpenVL and VLUT. Two example application designs are introduced in Section 10.4 as a proof of concept including how to implement them using OpenVL and VLUT. Conclusions and future works are briefly discussed in Section 10.5.

10.2 Related Work

In this section, we discuss previously published work that addresses the efficiency, reusability, and scalability issues. They are organized as follows. Section 10.2.1 discusses a widely used image processing library: OpenCV. In Section 10.2.2, we review the pipes and filters software architecture. OpenGL is also discussed in Section 10.2.3 as it provides part of the motivation behind our approach. Section 10.2.4 outlines related hardware architectures for parallel processing that are useful structures for implementing components of OpenVL.

10.2.1 OpenCV

The introduction of the OpenCV [5] is an important milestone addressing system implementation issues in computer vision. Currently it is probably the most widely used vision library for real-time extraction and processing of meaningful data from images.

The OpenCV library provides more than 500 functions whose performance can be enhanced on the Intel architecture. If available, the Intel integrated performance primitives (IPP) is used for lower-level operations for OpenCV. IPP provides a cross-platform interface to highly optimized low-level functions that perform image processing and computer vision primitive operations. IPP exists on multiple platforms including IA32, IA64, and StrongARM, and OpenCV can automatically benefit from using IPP on all of these platforms. When running applications using OpenCV, a built-in DLL switcher is called at run time to automatically detect the processor type and load the appropriate optimized DLL for that processor. If the processor type cannot be determined (or if the appropriate DLL is not available), an optimized C code DLL is used.

However, because OpenCV assumes essentially a sequential software architecture, the potential acceleration resources in computer vision are not fully exploited to improve performance. For example, many independent operations can run in parallel. The dependencies of operations are not explicitly specified in OpenCV,

limiting hardware designers in fully utilizing possible speedup resources. Moreover, the OpenCV library does not provide an explicit capacity to support multi-camera streams, which limits the system scalability and puts the complexity for managing these solutions on the shoulders of application developers.

10.2.2 Pipes and Filters and Data-Flow Approaches

Compared to sequential software architecture, a pipes and filters architecture [14], which naturally supports parallel and distributed processing, is more appropriate for a system processing a stream of data. In the pipes and filters architecture, each component has a set of inputs and outputs. The components, termed *filters*, read streams of data as inputs and produce streams of data as outputs. The connectors, called *pipes*, serve as conduits for the streams, transmitting the output of one filter to the inputs of another. Fig. 10.1 illustrates this architecture.

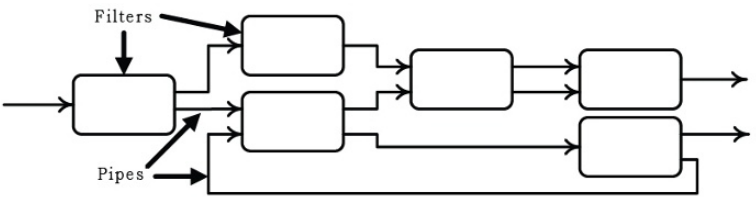

Fig. 10.1 Pipes and filters architecture. In the pipes and filters architecture, *filters* have a set of inputs and outputs. Each *pipe* implements the data flow between adjacent filters.

Jitter [6] is one example of an image library using a pipes and filters architecture. It abstracts all data as multidimensional matrices that behave as streams, so objects that process images can also process audio, volumetric data, 3D vertices, or any numerical information. Jitter's common representation simplifies the reinterpretation and transformation of media. DirectShow [12] and Khoros [10] also use pipes and filters as their underlying architecture model. The former is a library for streaming-media operations on the Microsoft Windows platform. The latter is an integrated software development environment with a collection of tools for image and digital signal processing.

The pipes and filters architecture has a number of features that make it attractive for some applications. First, this architecture allows the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of individual filters. Therefore, it is quite intuitive and relatively simple to describe, understand, and implement. It allows users to graphically create a block diagram of their applications and interactively control input, output, and system variables. Second, this architecture supports reuse: any two filters can be connected together, provided they agree on the data format being transmitted. Systems based on pipes

and filters are easy to maintain and update: new filters can be added to existing systems and old filters can be replaced by improved ones. Third, the pipes and filters architecture provides an easy synchronization mechanism, because the filters do not share data with other filters. Fourth, because data-processing objects, i.e., filters, are independent, this architecture naturally supports parallel and distributed processing.

However, the general pipes and filters architecture has its own disadvantages. First, the pipes and filters architecture does not allow instructions from multiple loop iterations (or multiple calls to the same routine) to be issued simultaneously, as the simple data dependence model prevents it from differentiating between the different loop iterations (or each invocation of the routine).

Second, because filters are intended to be strictly independent entities (they do not share state information with other filters, and the only communication between filters occurs through the pipes), the pipes and filters architecture does not provide a mechanism for users to reconfigure the data flow routine in run time. This means that a pipes and filters architecture is typically not good at handling highly interactive applications that may have many branches in the data flow.

Third, each filter's output data must be copied to its downstream filter(s)' input, which can lead to massive and expensive data copying if care is not taken. Without modification, this architecture cannot efficiently broadcast data tokens or dispatch instruction tokens in a massively parallel system because of arbitrary filters' independence.

Our approach is a variation on the pipes and filters model with adjustments made to match some of the common structures found in computer vision algorithms.

10.2.3 OpenGL

The current situation in computer vision is very similar to the state of computer graphics over a decade ago. In 1992, SGI led a consortium to create OpenGL [8], an open source graphics library geared toward hardware acceleration. GLUT [8] was also successfully designed as its middleware to standardize applications' access to operating systems and hardware platforms.

In OpenGL, one of the foundations of real-time graphics is the graphics rendering pipeline. Graphics commands and data are distributed in a graphics rendering pipeline, which enables hardware designers to accelerate these common operations in each portion of the OpenGL pipeline to optimize performance. For example, all transformations of an object in OpenGL are performed using 4×4 matrices that describe translation, rotation, shear, and scaling. Multiple matrix operations use a matrix stack. Combinations of individual rotations and translations are accomplished by multiplying two or more matrices together. If an accelerated physical architecture is used to support 4×4 matrix operations, the throughput of the system is increased. Further, by supporting a logical pipeline representation of a chain of transformation operators that are based on these multiply operations, the application programmer has different perspectives upon which to program typical graphics algorithms that

match concepts from the field. However, in the actual implementation, these operations can be premultiplied using a matrix stack, allowing significant increases in speed without impacting the logical structure that application coders are comfortable with.

Inspired by the success of OpenGL in promoting the development of hardware acceleration for computer graphics, we define and develop OpenVL for computer vision systems, bearing hardware acceleration, reusability, and scalability in mind. The intent of OpenVL is to allow users to *quickly* and *easily* recover useful information from *multiple* real dynamic scenes, and in a *portable* manner across various software environments and hardware platforms.

However, we cannot simply migrate the OpenGL architecture into computer vision, because the latter's processing is not exactly an inverse of computer graphics rendering. Moreover, OpenGL does not provide a mechanism for synchronization of multiple pipelines. Since multi-camera systems have generated significantly growing interest recently, we cannot ignore this issue.

10.2.4 Hardware Architecture for Parallel Processing

A variety of architectures have been developed for representing parallel processing. Flynn [3] classified them into three categories: (1) single instruction stream–multiple data stream (SIMD) (2) multiple instruction stream-single data stream (MISD) (3) multiple instruction stream–multiple data stream (MIMD). SIMD is well suited to low-level vision computing because many image processing operations in low-level are intrinsically parallel in the sense that the same rule must be applied to each of many data and the order in which the data are processed does not matter. Little et al. [11] implemented several computer vision algorithms using a set of primitive parallel operations on a SIMD parallel computer. SIMD is used in the graphics processing unit (GPU) on commodity video cards. However, SIMD is not particularly suitable for higher level processing where each operation involves lists and symbols rather than a small neighborhood and where we may wish to apply different operations to different part of the image. The flexibility of running different programs on each processing unit is provided by MIMD architecture. MISD, i.e., pipeline, can be employed to match the serial data inputs from camera to decrease the latency.

The use of hardware platforms with parallel processing is now generally accepted as necessary to support real-time image understanding applications [18]. Parallelism can be of several types: data, control, and flow. Data parallelism is the most common in computer vision. It arises from the nature of an image, a bidimensional regular data structure. Control parallelism involves processes that can be executed at the same time. The use of multiple cameras provides the potential source of control parallelism. Flow parallelism arises when an application can be decomposed into a set of serial operations working on a flow of similar data. The steady stream image data lends itself to pipelined parallelism.

OpenVL is intended to be cross-platform. Many hardware platforms are available that can be used to implement the OpenVL logical architecture. We anticipate that different drivers will be coupled with each implementation supplied by vendors to accelerate different components of the pipeline. Further, VLUT provides the interface to the different camera and hardware configurations that isolates applications from these details to increase reusability, much as OpenGL and GLUT work together. Some typical hardware platforms are: field-programmable gate arrays (FPGAs), digital signal processors (DSPs), digital media processors, GPUs, and various co-processor platforms.

GPUs, which are using a SIMD architecture, have evolved into extremely flexible and powerful processors. Since the GPU is built to process graphics operations that include pixel and vertex operations among others, it is particularly well suited to perform some computer vision algorithms very efficiently. For example, Yang and Pollefeys [19] implemented a stereo algorithm on an NVIDIA GeForce4 graphics card, whose performance is equivalent to the fastest commercial CPU implementations available.

The prototyping of the OpenVL hardware device on an Altera DE2 development board (using FPGA) is under development to illustrate how components of the OpenVL may be accelerated as a proof-of-concept for acceleration. We also plan to explore GPU and other co-processor architecture implementations of OpenVL.

10.3 A Novel Software Architecture for OpenVL

This section presents our novel software architecture—OpenVL to address the issues of *reusability*, *efficiency*, and *scalability* in the computer vision domain. It is a variation of the pipes and filters architecture, aiming at addressing the limitations of general pipes and filters while preserving its desirable properties by constraining it to typical image and vision processing algorithms.

10.3.1 Logical Pipeline

A general pipes and filters architecture cannot efficiently solve all of the dependencies found within an arbitrary topology of a large-scale parallel system. To address this problem, we introduce a logical pipeline to restrict the topologies of the filters into a linear sequence that are found in typical image processing tasks. This has two benefits: one, it provides a simple mental model for application developers for constructing models and, two, it provides a language model supporting a tractable description of image processing tasks that can be hardware accelerated.

10.3.1.1 Rationale for Logical Pipeline

Incorporating a logical pipeline into OpenVL makes hardware acceleration possible, because each stage can be implemented as a separate task and potentially executed in parallel with other stages. If hardware designers can provide a dedicated hardware platform to improve the performance of the common operations at each stage, the performance of the whole system can be significantly improved. Further, the structure of the pipeline itself can be used to develop dedicated hardware solutions to optimize the whole pipeline in addition to the individual components.

We differentiate between logical stages and physical stages. A logical stage has a certain task to perform, but does not specify the way that task is executed. A physical pipeline stage, on the other hand, is a specific implementation and is executed simultaneously with all the other pipeline stages. A given implementation may combine two logical stages into one physical pipeline stage, while it divides another, more time-consuming, logical stage into several physical pipeline stages, or even parallelizes it. From the perspective of an application programmer, the logical pipeline provides a clear conceptual model and language constructs to support descriptions of the processing that needs to be done. From the OpenVL implementers' perspective, the logical model provides the description needed to determine optimal ways to actually implement the operations.

10.3.1.2 OpenVL Logical Pipeline

The development of OpenVL is a large-scale project involving collaboration among researchers from various computer vision fields to assess which classes of processing tasks fit within the scope of OpenVL. In this chapter, we present human tracking as one class of algorithms that are planned to be within the scope of OpenVL and is our starting point for a proof-of-concept for OpenVL in general. We chose human tracking since it is one of the most active research areas in computer vision. Our example design implements all critical components of OpenVL to demonstrate the concepts behind it. We anticipate that if the proposed architecture can be applied to human tracking, it should be extensible to other classes of image and vision processing for other applications. We also use the class of multicapture single-image processing applications to define the OpenVL structure, however, these elements are not reported here.

Based on the reviews of human tracking algorithms in [1, 13], we propose an image understanding pipeline for human tracking.

For multiple video sources, multiple OpenVL pipelines would be created, as shown in Fig. 10.2. Operations in one pipeline can access data buffers of the same stage in other pipelines directly, suggesting that synchronization and communication mechanisms are implicit. Actual implementations to support this logical communication can use shared memory or other bus structures to provide differing price/performance levels as required.

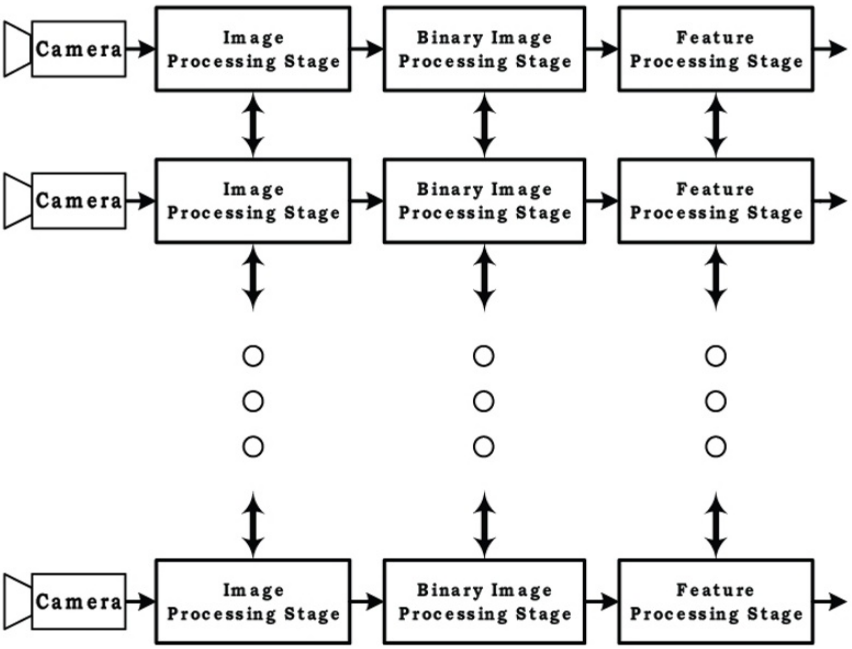

Fig. 10.2 OpenVL multiple logical image understanding pipeline. The pipeline is broken into several stages based on the type of data being processed. Each stage contains a set of data buffers and common operations, allowing for pipeline-based acceleration.

Fig. 10.3 shows a highlighted single logical pipeline. Based on the type of data representation being processed, this pipeline is broken into four primary stages: video capture, image processing, binary image processing and feature processing. Each stage can be divided further into substages. Pipelining reduces the cycle time of processing and hence increases processing throughput. As well, the granularity of the pipeline data flow may also be varied to include frames or subregions where the minimum size of the subregion is an OpenVL specification (i.e., nine pixels).

The input video data can be either from a physical camera using various ports, such as IEEE 1394 or USB, or from a "virtual camera" that loads video from files. Virtual camera concepts apply to offline analysis of video data or video editing applications, which also involve computationally intensive operations. In the image processing stage, the image buffer is used to store raw input and modified image data. This stage has several substages: color space transformation, image enhancement in the spatial domain, linear filtering, nonlinear filtering, geometrical transformation and temporal processing. In the binary image processing stage, the binary image buffer is used to store input and modified binary image data. This primary stage has two substages: morphological processing and nonlinear transformation. In the feature processing stage, a feature buffer is defined to store a feature values list. The content of the list is entirely dependent on the processing that took place at

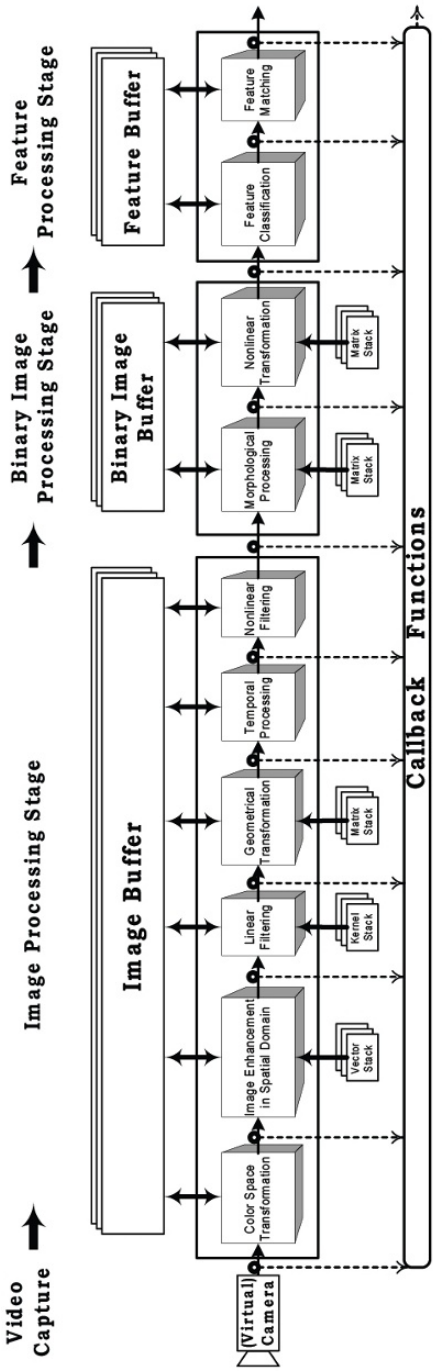

Fig. 10.3 OpenVL single logical image understanding pipeline.

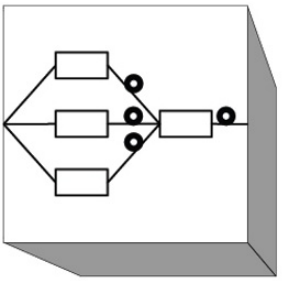

Fig. 10.4 Parallel operations in OpenVL substages.

previous stages in the pipeline. This stage has two substages: feature classification and feature matching.

Fig. 10.4 gives an example collection of processes in a substage. Each substage contains a set of operations which can run in a parallel style when hardware implementation is supported, further improving optimized performance. For example, two edge detectors with different thresholds may run in parallel, and then an edge-linking process can connect the outputs.

Different approaches are possible to provide structures to address the link between the logical and physical layers to implement the various components of the logical pipeline. The following subsections provide some examples of these.

10.3.2 Stacks

The stack boxes in Fig. 10.3 represent a stack buffer that we propose to implement part of OpenVL's processing to optimize hardware acceleration by allowing preprocessing of operations. This approach is seen in OpenGL, for example, with the matrix stack for transformation operations. For image processing, convolution serves as an example that can use this approach. Convolution is a widely used operator in computer vision because any linear, shift-invariant operation can be expressed in terms of a convolution [4]. For example, both Gaussian filtering and edge detection use this operation. Because the convolution operation is associative, i.e.,

$$((f * g) * h) * \ldots = f * (g * h * \ldots)$$

where f is image data, $g, h, \ldots$ are filter kernels, we can improve the performance of the operation through the following means. First we can stack all of the filters and convolve them together, and then convolve the result with the image data. The resulting performance is much better compared with combinations of individual convolutions. Therefore, if physical architecture supports stacked convolution, system performance can be enhanced.

10.3.3 Event-Driven Mechanism

Since the pipes and filters architecture does not provide a mechanism for users to reconfigure the data flow routine in run time, it is not good at handling interactive applications. However, providing user interaction is important for computer vision applications. For example, different background segmentation techniques may be used based on background environments or the results of an image processing operation may be used to actuate some other process, such as some OpenGL process for visualization.

To support run-time interaction, an event management mechanism is introduced in VLUT (see Section 10.3.7) . Users employ event-handling and callback functions to perform application specific processing at appropriate points in the pipeline. Users can register interest in an event, such as when a feature stage has completed, by associating a procedure (i.e., callback function) with the event. The callback function is not invoked explicitly by the programmer. Instead, when the event occurs during OpenVL processing, the VLUT invokes all of the procedures that have been registered for that particular event. Thus an event announcement implicitly invokes the callback procedure to allow the application programmer to retrieve the result associated with the event.

In Fig. 10.3, a black circle represents a particular event happening. When an event happens, such as the arrival of a new image, convolution completion or erosion completion, the registered callback command will be triggered, giving the user control over data flow. For example, an application programmer may set up two callbacks: one to display an image as it comes in to the pipeline and another after the image processing stage is complete to see the effects visually.

Like OpenGL, using the event handling mechanism, OpenVL works as a state machine. We put into it various states that then remain in effect until we change them based on some events.

10.3.4 Data Buffers

One limitation in the general pipes and filters model is that each filter's output data must be copied to its downstream filter(s)'s input, which can lead to expensive data copying. To solve this problem, we introduce a data buffer concept, i.e., the buffer plane layers in Fig. 10.3. We abstract representations of common data structures from computer vision algorithms and store them in the data buffer. Because all processes in a stage can access data buffers belonging to that stage, data buffers are modeled as shared memory space in the logical architecture. This allows hardware designs to use *physical* shared memory as an option to avoid data copying as would be implied by a general pipes and filters architecture. Though, these designs need to ensure proper mutual exclusion and condition synchronization to realize this potential. In OpenVL, there are currently several primary data buffers: front, back, image, binary image, and feature.

10.3.5 Synchronization and Communication

In the OpenVL architecture, there are several kinds of synchronization and communication issues that we need to consider: (1) between camera and pipeline; (2) between processes in the same pipeline; (3) between processes in different pipelines; and (4) between user callback functions and the logical pipeline. We present each of them separately in the following sections.

10.3.5.1 Camera Capture and Processes in the Pipeline

The speeds of the camera capturing (writer) and processes in the pipeline (reader) may not be exactly the same. If a reader is faster than the writer, it will have to wait for additional data to arrive. Conversely, if the writer is faster than the readers, it may have to either drop data or disrupt the execution of the pipeline to deal with the new data. To deal with this in the logical pipeline we use a front and back buffer and configuration modes to establish which data handling policy to use. This may be implemented using a double buffer mechanism with mutual exclusion to allow for captured data to be stored in one buffer while the pipeline is processing the other.

10.3.5.2 Processes in the Same Pipeline

Because processes in the same pipeline can read or write a data buffer at the same time, mutual exclusion is needed to ensure that the data is shared consistently. In OpenVL, the mutual exclusion required is managed implicitly, thus, the implementation of OpenVL must ensure that buffers are protected to allow either one writer or multiple reader access exclusively.

10.3.5.3 Processes in Different Pipelines

In OpenVL, the logical architecture supports the notion that processes within a single stage of the pipeline have shared access to data buffers associated with that particular stage. This extends to all processes at the same stage in other pipelines as well. This provides a simple semantic mechanism for application developers. However, the actual implementation of this model may introduce significant delays if done sequentially, as data must be transferred explicitly between pipelines. However, hardware designers may also introduce special busses and/or shared memory segments that can handle multiple pipelines accessing these protected data spaces. Notice that data transfers between stages are not supported in the logical architecture directly, thus access to these must be done explicitly by the application programmer. An implementation may extend OpenVL by supporting these complicated transfers. However, as we anticipate their requirement is rare, we expect that most applications will not need the extra complexity and expense to support it.

10.3.5.4 Callback Functions and Processes in the Pipeline

Based on the relationships between callback functions and processes in the pipeline, we can categorize callback functions into three basic modes: fully synchronous callback, partial synchronous callback, and asynchronous callback as described below. A different synchronization mechanism is provided for each mode of callback functions as specified by the application developer in OpenVL. Callback functions may also run in a mixed mode, which is a combination of two or even three basic modes. In these cases, different synchronization mechanisms are used together. These mechanisms provide the OpenVL programmer flexible control for dealing with different types of timing constraints appropriate to their application. These modes are also designed to take into account different choices for optimizing hardware or software implementations of OpenVL.

Fully Synchronous Callback

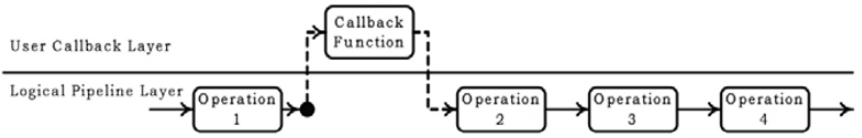

Fig. 10.5 Fully synchronous callback.

When the user needs to modify data using specific operations not provided in a given implementation of a pipeline, the callback function can be used to implement these operations. After the callback function finishes, results need to be joined back into the pipeline to gain accelerated performance. This is called fully synchronous callback mode, as shown in Fig. 10.5.

In this mode, the callback function works as a process in the pipeline. Therefore, synchronization in this case is also a multiple-writer and multiple-reader problem. Mutual exclusion, the same as that between processes in the same pipeline, should be provided by the OpenVL implementation.

Partial Synchronous Callback

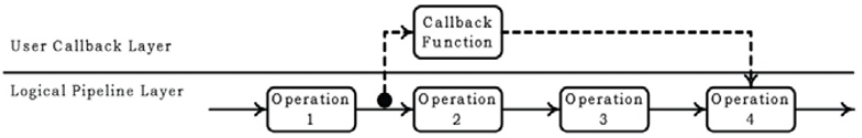

Fig. 10.6 Partial synchronous callback.

In this mode, the callback function provides the capacity for users to reconfigure the data-flow routine in run time. The unrelated processes, i.e., operations 2 and 3

in Fig. 10.6, can be run asynchronously with the callback function, while operations after these two operations need to synchronize with the callback function.

In this mode, mutual exclusion is needed to avoid the simultaneous use of the same data: operations' states, by the callback function and operations. Because this is a single-reader and single-writer problem, a simple synchronization mechanism, such as a binary semaphore mechanism, can be used to provide mutual exclusion.

Asynchronous Callback

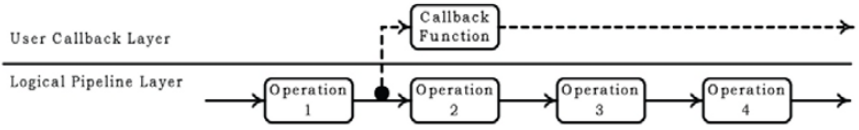

Fig. 10.7 Asynchronous callback.

In this mode, the callback function is employed by the user only to obtain intermediate results, as shown in Fig. 10.7. For example, the user needs the raw image from the camera for display. The callback function does not reconfigure the data-flow routine, and if all of the following operations do not need the results from the callback function.

Because callback functions are running asynchronously with processes in the pipeline, there is no need to let callback functions interfere with the operations. For example, this may be implemented with a bounded buffer to provide the synchronization between the callback function and the former process providing input to it. When the bounded buffer is full, there are two options for the producer, i.e., operation 1 in Fig. 10.7. The first option is to discard new input data. The second option is to store new data while the oldest data is discarded. There is only one option for reading when the bounded buffer is empty: the consumer, i.e., the callback function, has to wait.

10.3.6 Iteration

In Fig. 10.4, some operations require iteration with conditionals. Data-flow approaches have a difficult time representing semantics of iteration in a convenient way, however, this may be required given our current approach to OpenVL. Two issues need resolution: how iteration and conditional termination are expressed in the data-flow representation for OpenVL.

OpenVL provides two kinds of solutions to this issue. The first one is to provide a maximum fixed number of iterations. When the user needs more iterations, data continues being fed-back through a processing block for the given number of iterations. For the second, the application programmer specifies termination conditions

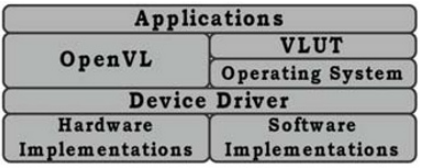

Fig. 10.8 Using device driver and VLUT to mask heterogeneity.

to stop the loop operation. For example, an erosion operation can be specified to run until there is only one pixel left. When this method is selected, some extra policies are provided to handle exceptions that may occur during the looping constructs. For example, an iteration operation may need to stop when it has exceeded some timeout parameter, or when some other conditions are met. The iteration mechanisms require language constructs such as *DoFor(n)* where *n* is the number of iterations, *DoUntil(cond, timeout)* where *cond* and *timeout* are termination conditions. However, these do not easily fit into a data-flow approach. We continue to develop convenience mechanisms to include these semantics into the logical architecture.

10.3.7 Isolating Layers to Mask Heterogeneity

Although the implementations of OpenVL may consist of a heterogeneous collection of operating systems and hardware platforms, the differences are masked by the fact that the applications use the VLUT and device driver layers to isolate the specific implementations, depicted in Fig. 10.8.

When there is hardware implementation to support processing, OpenVL calls the device driver to gain the benefit of hardware acceleration. Otherwise, a slower software implementation will be used. The VLUT layer is used to isolate the operating system, event management, hardware acceleration and cameras. Therefore, application developers cannot tell the difference between hardware and software implementations. The user will notice, however, that performance is significantly enhanced when hardware acceleration is available. Furthermore, the isolating layers make it possible for users to upgrade their applications to new OpenVL hardware, immediately taking advantage of their device's newly accelerated performance.

Moreover, individual calls can be either executed on dedicated hardware, run as software routines, or implemented as a combination of both dedicated hardware and software routines. Therefore, the isolating layers provide hardware designers with the flexibility to tailor a particular OpenVL implementation to meet their unique system cost, quality and performance objectives.

10.4 Example Application Designs

The development of real-time video analysis applications was a steering concern during development of the OpenVL software architecture. The applications presented here were used as testbeds for the implementation, testing and refinement of critical concepts in OpenVL. We first describe the procedure for implementing a computer vision application, and then two different human tracking systems are implemented to validate the critical underlying concepts of OpenVL and VLUT.

10.4.1 Procedure for Implementing Applications

The sequence of a typical OpenVL/VLUT program is illustrated in Fig. 10.9 (a):

(1) Use the VLUT function, *vlutInitializeCamera()*, to initialize the camera mode, including image resolution, color mode, geometries etc.

(2) Provide configuration of the OpenVL buffers including: which buffers are available, buffer resolutions, how many buffer planes and how many bits per pixel each buffer holds, etc., which will be used by the application. These are VLUT calls.

(3) Initialize OpenVL buffer values. The user may set initial values such as the convolution kernel values using OpenVL calls.

(4) Establish the OpenVL operation queues. The user sets the image understanding pipeline path to control data flow using OpenVL calls. These establish the state of the machine so that they are executed every cycle of the pipeline once the event manager enters its infinite main loop below (6).

(5) Register any callback functions with the event handler including the modes that the callback functions will operate in. Coupled to these are the actual callback functions that implement the desired application specific processing outside the OpenVL pipelines.

(6) The last call is to *vlutMainLoop()* to enter an infinite loop to manage events, run the pipelines and trigger callbacks.

10.4.2 Local Positioning System (LPS)

Location is one of the most important context information for surveillance systems. In [15], we have designed and implemented an LPS to locate the positions of objects in an indoor space. In our approach, the main idea is to use an infrared tag as an active transmitter, sending out a unique identification number to signal its presence to the system. There are five main steps in the pipeline of this system. The first step is noise removal. A Gaussian filter is applied to remove some noisy data. The second step is to use thresholding to get binary image data. The third step is to group bright spots, since the tag's bright spot is often captured as several small ones, we need to group them into one bright spot. The fourth step is to shrink this bright spot into

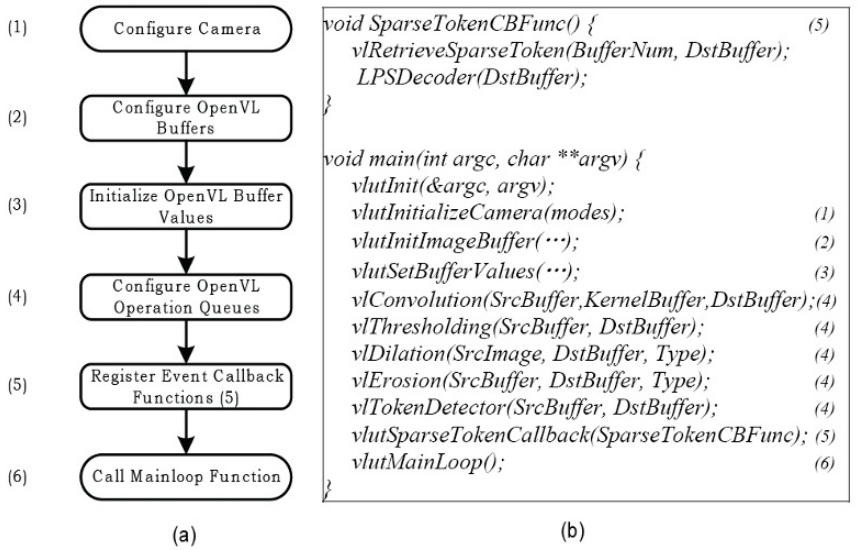

Fig. 10.9 In this figure, (a) shows the procedures to write a program using OpenVL. (b) illustrates a Local Positioning System pseudo-code using OpenVL.

one pixel so that only location information is stored. The final step is to decode the tag ID. The image processing operations in this local positioning system, such as filtering or morphological operations, are widely used in other video based surveillance systems. Moreover, these operations are some of the most time consuming ones among all image processing operations. Therefore, this local positioning system is a good choice to demonstrate OpenVL. We have reimplemented this simple local positioning system to demonstrate how OpenVL applications function and can be easily developed in a portable manner, and how they can be accelerated.

Fig. 10.9 (b) shows the concise code for implementing the local positioning system using OpenVL and VLUT. This code implements a Gaussian filter on input video data to remove noise, thresholds the result to find the bright spots in the image, dilates bright spots to group together close ones belonging to same tag, erodes each bright spot to one pixel bright spot, and finally generates a list of (x,y) locations where there is a bright spot. This list is passed into a registered callback function for the application specific code that decodes the tag IDs based on the pattern of blinking. If the OpenVL pipeline is implemented on a hardware device, this architecture not only reduces the computational load on the central CPU processor, but it also significantly reduces the amount of data that must be transferred between devices: a 99 KB (352×288) image reduces to a list of 5 to 50 points of 4 bytes (x,y) each, effectively reducing required data transfers by 500 to 5000 times.

```
void ThresholdCallback() {                                                    (5)
    vlGetBinaryImageBuf(BK_Data_Buffer);
    ……
    Communicate with other thread to display Background Subtraction results
    ……
}
void DistanceCallback() {                                                     (5)
    vlGetFeatureVectorBuf(Dis_Data_Buffer);
    ……
    Distance Calculation
    ……
}
void ViewAngleCallback() {                                                    (5)
    vlGetFeatureBuf(Angle_Data_Buffer);
    ……
    View Angle Calculation
    ……
}
void main(int argc, char **argv) {
    vlutInit(&argc, argv);
    vlutInitializeCamera(modes);                                              (1)
    vlutInitImageBuffer(…);                                                   (2)
    vlutSetBufferValues(…);                                                   (3)
    vlutSetBufferValues (VL_IMAGE_BUF_1, "/video/%d.pgm");                    (3)
    vlColorSpaceTransform(VL_RGB,VL_YUV);                                     (4)
    vlConvolution(SrcBuffer,KernelBuffer,DstBuffer);                          (4)

    // Background Subtraction
    vlGaussianModel(VL_IMAGE_BUF_1, 10, VL_UPDATE);                           (4)
    VL_THRESHOLD_VAL= 3;
    vlThreshold(VL_IMAGE_BUF_1,VL_THRESHOLD_VAL, VL_BINARY_IMAGE_BUF_1);      (4)

    // Calculate Distance
    vlFeatureMatrixMultiply(VL_FEATURE_BUF_2, FEATURE_MATRIX_1, VL_FEATURE_BUF_3);   (4)
    vlFeatureVectorSubtract(VL_FEATURE_BUF_2, VL_FEATURE_BUF_3, VL_FEATURE_BUF_4);   (4)
    vlFeatureVectorSquareroot(VL_FEATURE_BUF_4, VL_FEATURE_BUF_5);            (4)

    //Calculate ViewAngle
    vlEqualizeHist(VL_IMAGE_BUFFER_6, VL_IMAGE_BUFFER_7);                     (4)
    vlHaarDetectObjects(VL_IMAGE_BUFFER_7, cascade, storage, 1.1, 2, 0,  30, 30, VL_FEATURE_BUFFER_1);   (4)
    vlFeatureVectorScale(VL_FEATURE_BUFFER_1, scale, VL_FEATURE_BUFFER_2);    (4)

    // Register Events Callback Handles
    vlutCallback(VL_BINARY_IMAGE_BUFFER_1, ThresholdCallback);                (5)
    vlutCallback(VL_FEATURE_BUFFER_5, DistanceCallback);                      (5)
    vlutCallback(VL_FEATURE_BUFFER_2, ViewAngleCallback);                     (5)
    vlutMainLoop();                                                           (6)
}
```

Fig. 10.10 This figure illustrates OpenVL pseudo-code for a human tracking system, which calculates the quality of view (QOV) and automatically selects the camera with the best QOV.

10.4.3 Human Tracking and Attribute Calculation

We have also implemented a more sophisticated human tracking and attribute calculation system using OpenVL and VLUT [16]. Initially, we use the method in [17] to calibrate cameras. A set of virtual 3D points is made by waving the laser pointer through the working volume. Its projections are found with subpixel precision and verified by a robust RANSAC analysis [2]. After calibration, we record a sequence of background images without a person in the scene. For each pixel in the background, we calculate the mean and variance of pixel intensity, resulting in a Gaussian distribution. To determine whether a pixel is in the foreground or a part of the background, its intensity is fit into the Gaussian model of the corresponding pixels. If image pixels are classified as background pixels, then these pixels are used to update background models. Instead of using a RGB color space, we use a YUV color space in our system to reduce shadows. The centroid and a distance map are obtained by applying distance transforms to the binary result image derived from the background subtraction process. Each value in the distance map corresponds to be the minimum distance to the background. From these, we calculate the distance between camera and the subject's centroid. Quality of view (QOV) calculations belong to an application layer; therefore, it is defined in user defined callback functions.

In this human tracking system, more complicated algorithms are used than the LPS. Nevertheless, the procedure for implementing this system using OpenVL is same as for LPS. Fig. 10.10 shows the concise code for implementing the human tracking system using OpenVL and VLUT.

10.5 Conclusion and Future Work

In this chapter, we presented a novel software architecture for OpenVL to promote hardware acceleration, reusability, and scalability in computer vision systems. The OpenVL API defined in this work is a starting point for a standardized interface that can work seamlessly on different software/hardware platforms. This chapter focused on the description of the logical architecture and provided some insight into techniques that may be used to implement it. The OpenVL API syntax and some of our architecture's critical concepts were demonstrated with two examples so far.

There are several directions we are currently pursuing. We plan to continue developing the OpenVL image understanding pipeline to cover other classes of computer vision algorithms and applications, and hopefully lead to its widespread adoption. We plan to develop models for control structures such as *iteration* and *conditionals* that are not part of a typical pipes and filters model but necessary for many computer vision applications. The prototyping of the OpenVL hardware device on FPGA and GPU are under development to illustrate how components of the OpenVL pipeline may be accelerated as a proof-of-concept for acceleration. We believe that the enhancement and adoption of OpenVL by a community of researchers will

significantly improve the size and complexity of computer vision systems, since OpenVL enables easy and quick code development, independent of platform.

One issue in the current OpenVL design is that it is oriented towards the application programmer specifying *how* an image processing algorithm is implemented rather than *what* they want done. There are two problems with the *how* approach. One is that it is difficult to design implementations to accelerate algorithms as the details of the processes are already specified by the application leaving little freedom for optimization. Another reason is that nonimage processing experts may not know how particular algorithms work, but they do know what type of operations they want done. Thus, they are more focused on *what* needs to be done. Currently we are investigating the next generation of OpenVL to incorporate these semantics.

In summary, we have created our first version of an open vision library consisting of a logical pipeline coupled with a language model to support image processing applications that are reusable, scalable and can be accelerated by hardware by different vendors. We continue to expand and refine our pipeline to provide an open specification that can be implemented on different hardware and software platforms to support portable image processing applications.

10.6 Acknowledgements

This research is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), Bell University Labs, and the Institute for Robotics and Intelligent Systems (IRIS). We appreciate the inspiring ideas of Jim Clark, Jeremy Cooperstock, and Roel Vertegaal and the useful discussions with Steve Oldridge and Amir Afrah. Thanks to Craig Wilson for proofreading.

References

1. J. K. Aggarwal, Q. Cai (1999) Human motion analysis: A review. *Computer Vision and Image Understanding: CVIU* 73:428-440.
2. M. Fischler, R. Bolles (1987) Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM* 24(6), 381-395.
3. Michael J. Flynn (1996) Very high speed computing systems. *Proc. IEEE.*
4. David A. Forsyth, Jean Ponce (2003) *Computer Vision: A Modern Approach,* Prentice Hall.
5. Intel Inc. (2001) Open Source Computer Vision Library: Reference Manual, Intel Corporation.
6. Jitter Tutorial (2006) version 1.6, *Cycling '74.*
7. J.M. Jolion, A. Rosenfeld (1994) *A Pyramid Framework for Early Vision,* Kluwer Academic.
8. Renate Kempf, Chris Frazier (1996) *OpenGL Reference Manual. The Official Reference Document for OpenGL,* Version 1.1, Addison Wesley Longman Inc.
9. Josef Kittler, Michael J. B. Duff (1985) *Image Processing System Architecture,* Research Studies Press.

10. K. Konstantinides and J. R. Rasure (1994) The Khoros software development environment for image and signal processing, *IEEE Transactions on Image Processing* 3:243-252.
11. James J. Little, G. E. Blelloch, T. A. Cass (1989) Algorithmic techniques for computer vision on a fine-grained parallel machine, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11:244-257.
12. DirectShow Reference (2007) MSDN.
13. Thomas B. Moeslund, Erik Granum (2001) A survey of computer vision-based human motion capture, *Computer Vision and Image Understanding: CVIU* 81:231-268.
14. Mary Shaw, David Garlan (1996) *Software Architecture: Perspectives on an Emerging Discipline,* Englewood Cliffs, NJ: Prentice Hall.
15. Changsong Shen, Sidney Fels, et. al (2005) RemoteEyes: A remote low-cost position sensing infrastructure for ubiquitous computing, *Transactions of Society of Instrument and Control Engineers* E-S-1:85-90.
16. Changsong Shen, Sidney Fels (2007) A multi-camera surveillance system that estimates quality-of-view measurement, *IEEE International Conference on Image Processing.*
17. T. Svoboda, D. Martinec, T. Pajdla (2005) A convenient multi-camera self-calibration for virtual environments, *Teleoperators and Virtual Environments* pp. 407-422.
18. Charles C. Weems (1991) Architectural requirements of image understanding with respect to parallel processing, *Proc. IEEE* 79:537-547.
19. Ruigang Yang, Marc Pollefeys (2003) Multi-resolution real-time stereo on commodity graphics hardware, *Proc. IEEE CVPR 2003.*