

OpenVL: Towards A Novel Software Architecture for Computer Vision

Changsong Shen and Sidney S. Fels
Department of Electrical and Computer Engineering
University of British Columbia
{csshen, ssfels}@ece.ubc.ca

James J. Little
Department of Computer Science
University of British Columbia
little@cs.ubc.ca

Abstract

This paper presents our progress on OpenVL - a novel software architecture to address efficiency through facilitating hardware acceleration, reusability and scalability for computer vision. A logical image understanding pipeline is introduced to allow parallel processing. As well, we discuss our middleware - 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 across various software environments and hardware platforms. We implement two different human tracking systems to validate the critical underlying concepts of OpenVL.

1. Introduction

Computer vision technology is profoundly changing a number of areas, such as human-computer interaction, robotics. However, building computer vision systems remains difficult because of software engineering issues such as efficiency, reusability and scalability. 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 of a field, just like the success of OpenGL in promoting the development of hardware acceleration coupled with significant 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 27MB/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 a mechanisms to maintain correspondence among separate but related video streams at the architectural level.

The Open Source Vision Library (OpenVL) and its Utility Toolkits (VLUT) are designed to address efficiency, reusability and scalability to facilitate progress in computer vision. By providing a hardware development middleware that supports different hardware architectures for acceleration, OpenVL allows code reuse without compromising performance. There are some other important implementation issues, such as timeliness. These issues are out of the scope of this paper and in the plan of future work.

2. Related Work

In this section, we discuss related work that addresses the issues we mentioned in the above section. They are organized as follows. Section 2.1 discusses a widely used im-

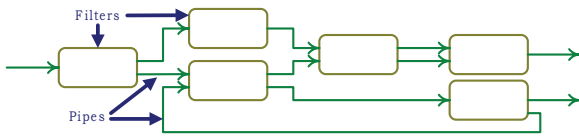


Figure 1. Pipes and Filters architecture. In the Pipes and Filters architecture, *filters* have a set of inputs and outputs. Each *pipe* implements the data flow between adjacent filters.

age processing library: OpenCV. In section 2.2, we review the Pipes and Filters software architecture. OpenGL is also discussed in 2.3 as it provides part of the motivation behind our approach. Section 2.4 outlines related hardware architectures for parallel processing that are useful structures for implementing components of OpenVL.

2.1. OpenCV

The introduction of the OpenCV [1] 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, 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 explored to improve performance. For example, many independent operations can run in parallel. The independencies 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.

2.2. Pipes and Filters and Data-Flow Approaches

Compared to sequential software architecture, a Pipes and Filters architecture [12], which naturally supports parallel and distributed processing, is more appropriate for a system processing a stream of data. In the Pipes and Fil-

ters 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. Figure 1 illustrates this architecture.

Jitter [2] 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 [3] 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 hooked together, provided they agree on the data that are being transmitted between them. Systems based on Pipes and Filters are also 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 re-configure the data flow routine in run time. This means that a Pipes and Filters architecture is typically not good at handling interactive applications.

Third, each filter's output data must be copied to its

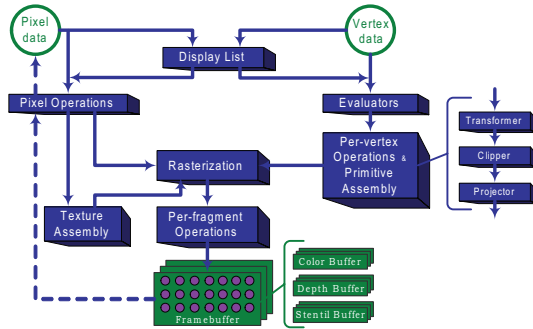


Figure 2. Image rendering pipeline in OpenGL. The pipeline enables hardware designers to accelerate the common operations in each portion of the pipeline to optimize performance.

downstream filter(s)' input, which can lead to massive and expensive data copying. 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.

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, depicted in Figure 2. 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 pre-multiplied 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 can not 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. However, since multi-camera systems have generated significantly growing interest recently, we can not ignore this issue.

2.4. Hardware Architecture for Parallel Processing

A variety of architectures have been developed for representing parallel processing. Flynn [5] 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 because there are enormous amount and variety of potential parallelism [15]. 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 bi-dimensional regular data structure. Control parallelism involves processes that can be executed at the same time. The using 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 are 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), GPUs and various co-processor platforms.

GPUs, which are using a SIMD architecture, have evolved into an extremely flexible and powerful processor. 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 [16] 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 implementation of OpenVL.

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.

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 an architecture to base OpenVL to be a tractable description of image processing tasks that can be hardware accelerated.

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.

3.1.2 OpenVL Logical Pipeline

The development of OpenVL is such 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 paper, 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. The reason we choose human tracking is that 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. 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.

Based on the reviews of human tracking algorithms in [4], we propose an image understanding pipeline for human tracking, depicted in Figure 3.

For multiple video sources, multiple OpenVL pipelines would be created, as shown in Figure 3(A). Operations in one pipeline can access data buffers of the same stage in other pipelines directly, implying that synchronization and communication mechanisms are implicit. Actual implementations to support this logical communication could use shared memory or other bus structures to provide differing price/performance levels as required.

Figure 3(B) 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 sub-stages. 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. 9 pixels).

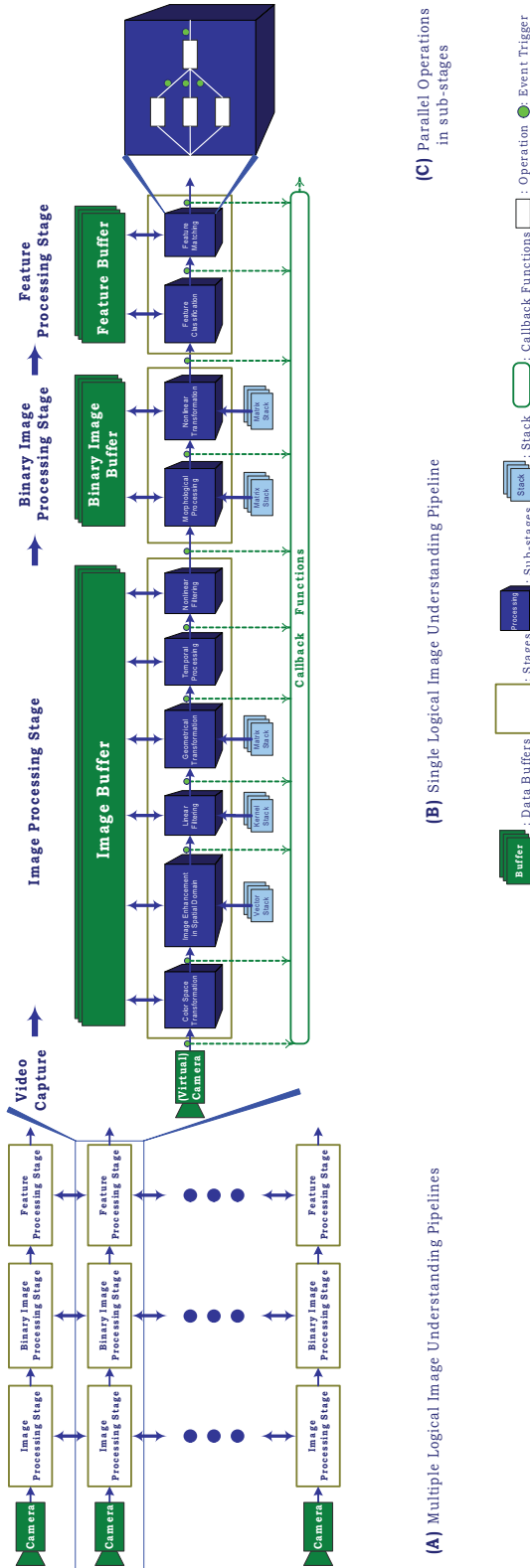


Figure 3. OpenVL logical image understanding pipeline. The pipeline is broken into several stages based on the type of data being processed. Each stage contains a set of data buffers and common operations, allowing for pipeline-based acceleration.

The input video data can be either from a physical camera using various ports, such as IEEE 1394 or USB, or even 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 sub-stages: 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 sub-stages: 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 previous stages in the pipeline. This stage has two sub-stages: feature classification and feature matching.

Figure 3(C) gives an example collection of processes in a sub-stage. Each sub-stage contains a set of operations which can run in a parallel style when hardware implementation is supported, further improving optimized performance.

The logical architecture provides a method for programmers to express their algorithms so that they can be accelerated. The logical pipeline can then be implemented by hardware designers to optimize performance appropriate to different levels of price and performance. The next subsections provide some examples of the structures that address the link between the logical and physical layers.

3.2. Stack

The lightblue boxes in Figure 3(B) are a stack buffer that we propose to insert into OpenVL to further optimize hardware acceleration by allowing preprocessing of operations. This approach is seen in OpenGL, for example, with the matrix stack for transformation. 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[6]. For example, both Gaussian filtering and edge detection use this operation. Because the convolution operation is associative, i.e. $((f * g) * h) * \dots = f * (g * h * \dots)$, where f is image data, g, h, \dots 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.

3.3. Event Driven Mechanism

Since the Pipes and Filters architecture does not provide a mechanism for users to re-configure 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. 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 by associating a procedure (i.e. callback function) with the event. The callback function is not invoked explicitly by the programmer. When the event is broadcast, the VLUT invokes all of the procedures that have been registered for the event. Thus an event announcement “implicitly” invokes the callback procedure.

In Figure 3(B), green circle represents a particular event happening. When an event happens, such as arrival of a new image, convolution completion or erosion completion, the registered callback command will be triggered, giving the user control over data flow. A simple example to use this may be to 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.

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 green plane layers in Figure 3(B). 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 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.

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 user callback functions and the logical pipeline; and 4) between processes in different pipelines. We present each of them separately in the following sections.

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 for example.

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.

3.5.3 Processes in Different Pipelines

Processes can access data buffers in other pipelines. The synchronization and communication are the same as among processes in the same pipeline.

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

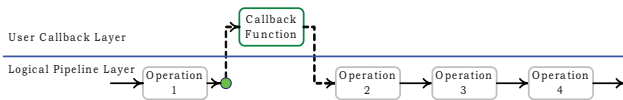


Figure 4. Fully Synchronous Callback.

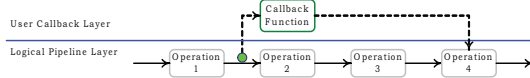


Figure 5. Partial Synchronous Callback.

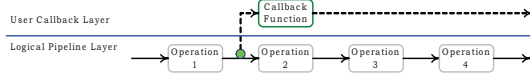


Figure 6. Asynchronous Callback.

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

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 Figure 4.

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

Partial Synchronous Callback

In this mode, the callback function provides the capacity for users to re-configure the data-flow routine in run time. The unrelated processes, i.e. operations 2 and 3 in Figure 5, 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 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

In this mode, the callback function is used only to obtain intermediate results, as shown in Figure 6. For example, the user needs the raw image from the camera for display. The callback function does not re-configure the data-flow

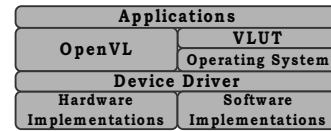


Figure 7. Using device driver and VLUT to mask heterogeneity.

routine, and if all of the 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 Figure 6. 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.

3.6. 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 Figure 7.

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. VLUT layer is used to isolate the operating systems. Therefore, application developers can not 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.

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

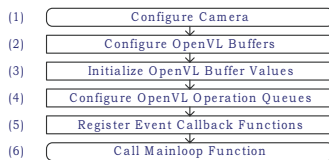


Figure 8. Procedures to write a program using OpenVL.

here were used as testbeds for the implementation, testing and refinement of critical concepts in OpenVL.

We implemented a simple active-LED tag-based tracking system we call a local positioning system (LPS) to illustrate how OpenVL and VLUT can be used [13]. The algorithm requires that small bright spots are detected which is one of the most basic operations in image processing. However, to implement this algorithm, the basic flow of an OpenVL and VLUT application can be seen and is illustrated in Figure 8. We have also implemented a more sophisticated human tracking system in OpenVL/VLUT [14]. While more complicated, the procedure for implementing this system using OpenVL is same as for LPS.

The sequence of a typical OpenVL/VLUT program are:

(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.

(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.

(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) Call `vlutMainLoop()` to enter an infinite loop to manage events, run the pipelines and trigger callbacks.

5. Conclusion and Future Work

In this paper, 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 paper focused on the description of the logical architecture and provided some insight as to 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. 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.

References

- [1] *Open Source Computer Vision Library: Reference Manual*. Intel Corporation, 2001.
- [2] Jitter tutorial, version 1.6. *Cycling '74*, 2006.
- [3] Directshow reference. *MSDN*, 2007.
- [4] J. K. Aggarwal and Q. Cai. Human motion analysis: A review. *Computer Vision and Image Understanding: CVIU*, 73(3):428–440, 1999.
- [5] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 1966.
- [6] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2003.
- [7] J. Jolion and A. Rosenfeld. *A Pyramid Framework for Early Vision*. Kluwer Academic, 1994.
- [8] R. Kempf and C. Frazier. *OpenGL Reference Manual. The Official Reference Document for OpenGL, Version 1.1*. Addison Wesley Longman Inc., 1996.
- [9] J. Kittler and M. J. B. Duff. *Image Processing System Architecture*. Research Studies Press, 1985.
- [10] K. Konstantinides and J. R. Rasure. The khoros software development environment for image & signal processing. *IEEE Transactions on Image Processing*, 3:243–252, 1994.
- [11] J. J. Little, G. E. Brelloch, and T. A. Cass. Algorithmic techniques for computer vision on a fine-grained parallel machine. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(3):244–257, 1989.
- [12] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall, 1996.
- [13] C. Shen, B. Wang, F. Vogt, S. Oldridge, and S. Fels. Remoteeyes: A remote low-cost position sensing infrastructure for ubiquitous computing. *Transactions of Society of Instrument & Control Engineers*, E-S-1:85–90, 2005.
- [14] C. Shen, C. Zhang, and S. Fels. A multi-camera surveillance system that estimates quality-of-view measurement. In *IEEE International Conference on Image Processing*, 2007.
- [15] C. C. Weems. Architectural requirements of image understanding with respect to parallel processing. *Proceedings of the IEEE*, 79(4):537–547, 1991.
- [16] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. *CVPR*, 01:211, 2003.