Addressing System-Level Optimization with OpenVX Graphs

Erik Rainey, Jesse Villarreal
Texas Instruments, Inc.
erik.rainey@ti.com, jesse.villarreal@ti.com

Goksel Dedeoglu
PercepTonic, LLC
goksel@perceptonic.com

Kari Pulli, Thierry Lepley, Frank Brill
NVIDIA
karip@nvidia.com, tlepley@nvidia.com, fbrill@nvidia.com

Abstract

During the performance optimization of a computer vision system, developers frequently run into platform-level inefficiencies and bottlenecks that can not be addressed by traditional methods. OpenVX is designed to address such system-level issues by means of a graph-based computation model. This approach differs from the traditional acceleration of one-off functions, and exposes optimization possibilities that might not be available or obvious with traditional computer vision libraries such as OpenCV.

1. Introduction

Accelerating a computer vision application on an embedded processor can be a non-trivial task. On modern System-on-Chips, the optimization efforts fall under two categories: *system*-level optimization and *kernel*-level optimization. System-level optimizations pay close attention to the overall power consumption, memory bandwidth loading, low-latency functional computing, *SoC IP* entitlement, and *Inter-Processor Communication* overhead. These issues are typically addressed via *frameworks*, as the parameters of interest cannot be tuned with *compilers* or *operating systems*. To date, *kernel*-level optimizations have traditionally revolved around *one-off* or *single function acceleration*. Typically this means that an implementer will re-write a computer vision function with a more efficient algorithm, with better *compiler* optimization switches ($-O3$), using more efficient *SIMD/MIMD* instructions [1], or moving the execution to accelerators such as a GPU using languages such as OpenCL or CUDA [4]. However, this improves only the execution of a single function, it does not necessarily improve the entire *use-case*.

2. Solution

OpenVX [2] attempts to solve both of these issues on behalf of the developer by containing the system-level optimization issue within a graph model, and the kernel-level optimization via accelerators with graph-level optimizations. This moves (part of) the onus of optimization back to the implementor of OpenVX, which is frequently where the most platform knowledge exists. OpenVX's approach to solving these system-level problems is not new [3] or unique. What is new and unique is addressing them using a *standardized* interface, enabling portable, high-performance computer-vision applications. The model is similar to the standardization of 3D graphics APIs to OpenGL [5] over 20 years ago; standardization allowed software developers to separate innovations on the application level from the hardware innovations of the companies providing graphics accelerators.

To facilitate incremental adoption of OpenVX, the API supports a single-function model, called *immediate* mode. This mode of operation is specified to be equivalent to a *single-node graph*. However, using the immediate mode precludes many of the graph-level optimization possibilities discussed below.

3. Architecture

OpenVX specifies a method of computation using *graphs*. Graphs in OpenVX are directed, ordered by *data-dependencies*, and acyclic. These graphs are constructed, then verified (for correctness, consistency, connectedness, and other attributes), and they can be processed (executed) in the future, repeatedly if needed. Essentially, a developer is declaring *future* work by aggregating processing stages into a connected graph. By giving the future work as an aggregation to the framework, many optimizations, which would normally be hidden, can be identified and addressed by the OpenVX framework.

3.1. Nomenclature, Hierarchy, and API

All objects within OpenVX exist within a *context*. *Data* and *graphs* exist within this context. Graphs are composed of *nodes*. Each of these nodes is an instance of a computer vision function (a *kernel*) with its associated data references, return value, and performance information (i.e., its future call state). OpenVX further allows kernel and target introspection, if needed. *Targets* in OpenVX are not necessarily cores or devices (e.g., a CPU, GPU, DSP, or a specialized accelerator), but may also be any logically constructed, functional-execution mechanisms (e.g., an OpenCL function, or an OpenMP-compiled set of functions). The specification does not provide an exhaustive definition of possible targets, so that implementors of OpenVX would have the freedom to introduce even new kind of accelerators that do not currently exist. Anything that can execute a conformant kernel could be exposed as a target. The supported kernels are summarize in Table 1.

Absolute Difference	Integral Image
Image Pyramid	Optical Flow Pyramid (LK)
Remap	Scale Image
Histogram (generate, equalize)	Thresholding
Accumulate (also squared, weighted)	Image Arithmetics (+, -, *)
Filters (box, custom, Gaussian, median, Sobel)	Corners (Fast, Harris)
Bitwise And, Xor, Or, Not	Edges (Canny)
Channel combine, extract	Phase, Magnitude
Convert (bit depth, color, table lookup)	Dilate, Erode
Stats (mean, std. dev., min/max location)	Warp (affine, perspective)

Table 1: OpenVX Base Kernels Version 1.0 (Provisional).

3.2. Data Opaqueness

Data objects in OpenVX are opaquely defined. When direct memory access is required, the user must call the Access and Commit API pair to get to the data. These APIs are present in order to permit the framework control over the memory location and layout of data.

3.3. User-defined Nodes

OpenVX provides for a feature where users of the API can create their own nodes and insert them in a graph, as in Figure 1. In the first version of the specification this is purely host (CPU) code (allowing such nodes to run another programmable unit such as a DSP or GPU would require a vendor extension). The purpose in doing this is to

- leverage the independent nature of node execution to provide parallelism in the work load while the framework potentially executes other nodes on other targets, and to
- clearly define a run-time verified functional boundary for the user-defined node so that it can be componentized for later (re-)use elsewhere.

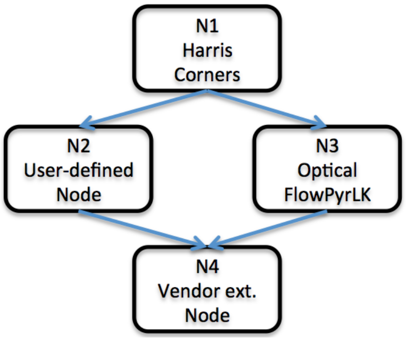


Figure 1: User-defined Nodes provide useful computation not provided by the API, yet allow the framework to determine an efficient order of processing.

Authors of user-defined nodes should be aware that their code can be invoked at any time, potentially in multiple threads or processes, and should follow guidelines in the OpenVX specification [2]. One such guideline is to not access unmanaged resources (with respect to OpenVX) without some external protection mechanism as OpenVX may not serialize the User-defined Node invocations.

3.4. Callbacks

Users of OpenVX may attach a host *callback* to any node so that the results of that node's computation is available to the host callback after execution of the node. The state of the rest of the graph (which was not a predicate) is undetermined. This mechanism allows a user to add flexible conditional logic to a graph to determine *actions*. For example, a user may determine that the maximum value from an image indicates that the image is too dark to continue processing and halt graph execution.

4. Example Graph

To facilitate discussion of the API and optimization possibilities, we introduce an example schematically (Figure 2) and the corresponding code listing (Figure 3). The format of the graph diagram only expresses nodes, and not data (arcs only order operations). We declare that the input data (e.g., an RGB camera image) is **(N1):** color-converted (RGB to YUV), **(N2):** channel-extracted (Luma), **(N3):** blurred, **(N4):** gradient operator (such as Sobel) is applied, producing components S_x and S_y, which are used to compute **(N5):** phase and **(N6):** magnitude. These are fed into a **(N7):** non-maximum suppression node, and finally **(N8):** thresholded to produce a "pseudo-Canny-edge-detection" result. We shall discuss optimizing this graph for a device of three heterogeneous targets. The actual specifics or details of the targets (e.g., a *CPU*, *GPU*, and *DSP*) are irrelevant.

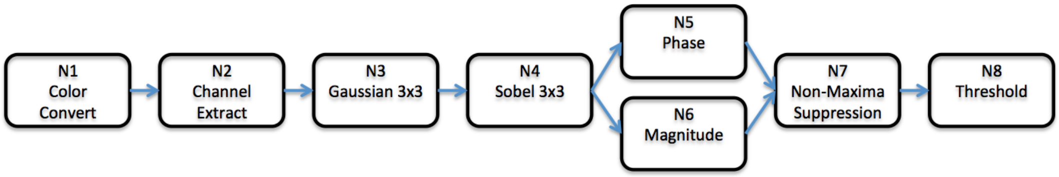


Figure 2: An example pseudo-Canny-edge-detector graph in OpenVX.

```
vx_context c     = vxCreateContext();
vx_graph g       = vxCreateGraph(c);
vx_image rgb     = vxCreateImage(c, w, h, FOURCC_RGB);
vx_image iyuv    = vxCreateVirtualImage(g, 0, 0, FOURCC_IYUV);
vx_image y       = vxCreateVirtualImage(g, 0, 0, FOURCC_VIRT);
vx_image by      = vxCreateVirtualImage(g, 0, 0, FOURCC_VIRT);
vx_image gx      = vxCreateVirtualImage(g, 0, 0, FOURCC_VIRT);
vx_image gy      = vxCreateVirtualImage(g, 0, 0, FOURCC_VIRT);
vx_image ang     = vxCreateVirtualImage(g, 0, 0, FOURCC_VIRT);
vx_image mag     = vxCreateVirtualImage(g, 0, 0, FOURCC_VIRT);
vx_image non     = vxCreateVirtualImage(g, 0, 0, FOURCC_VIRT);
vx_image out     = vxCreateImage(c, w, h, FOURCC_U8),
vx_threshold t = vxCreateThreshold(c,
                                  VX_THRESHOLD_TYPE_RANGE);
vx_uint8 upper = 240, lower = 10;
vx_node n[] = {
    vxColorConvertNode(         g, rgb,  iyuv),
    vxChannelExtractNode(       g, iyuv, VX_CHANNEL_Y, y),
    vxGaussian3x3Node(          g, y,    by),
    vxSobel3x3Node(             g, by,   gx,   gy),
    vxPhaseNode(                g, gx,   gy,   ang),
    vxMagnitudeNode(            g, gx,   gy,   mag),
    vxNonMaxSuppressionNode(    g, ang,  mag,  non),
    vxThresholdNode(            g, non,  t,    out),
};
vxSetThresholdAttribute(t, VX_THRESHOLD_ATTRIBUTE_UPPER,
                        &upper, sizeof(upper));
vxSetThresholdAttribute(t, VX_THRESHOLD_ATTRIBUTE_LOWER,
                        &lower, sizeof(lower));
if (vxVerifyGraph(g) == VX_SUCCESS)
    vxProcessGraph(g);
```

Figure 3: Code for example graph.

5. Optimization Strategies

Here we discuss some optional optimization strategies.

5.1. Remote Processing

Remote Processing is simply the practice of computing results on a non-host core such as a GPU, a DSP, or other specialized core such as an accelerator. In the following sections, we will refer to Equation 1 to discuss its impact on optimization strategies. This equation expresses the latency of remote processing for a single function call.

$$\begin{aligned} L_r(1) = C_{lf} + IPC_{send} + C_{ri} + \\ exec_r(data, params) + C_{rf} + \\ IPC_{recv} + C_{li}, \end{aligned} \tag{1}$$

where

- L_r — Latency of remote processing.
- C_{lf} — Host Core Cache Flush of impacted data.
- IPC_{send} — Total latency from Host-to-Remote-Core activation time. This accounts for line transmission, and possibly also system-thread switching and OS kernel/driver overheads.
- C_{ri} — Remote Core Cache invalidate for affected data.
- $exec_r(data, params)$ — Remote Execution time, varies on data size and other parameters.
- C_{rf} — Remote Core Cache flush for affected data.
- IPC_{recv} — Total latency from Remote-Core-to-Host activation time (with overheads as in IPC_{send}).
- C_{li} — Host Core Cache Invalidate of impacted data.

Typically all cache operations depend on the data size.

5.2. Aggregate Function Replacement

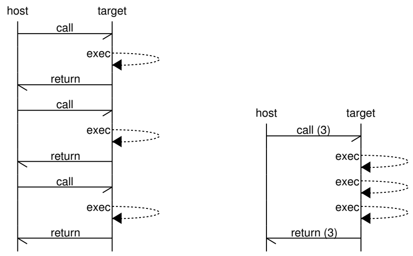
Aggregate function replacement is the practice of identifying a specific set of nodes in a graph and replacing (potentially multiple nodes) with a single node which "does-it-all" (aggregate function). In the example above, a strategy might be to replace sets of nodes with matching functionality. Reasonable examples include, but are not limited to:

- $N1$ & $N2$ — RGB to Luma (omits U, V computation).
- $N1$ & $N2$ & $N3$ — RGB to Blurred Luma (e.g., a three-channel weighted Gaussian on GPU).
- $N1$ & $N2$ & $N3$ & $N4$ — RGB to Sobel S_x, S_y.

5.3. IPC Aggregation

Inter-Processor Communication aggregation is the practice of co-locating the computation of functions (which may themselves be aggregate functions) on remote cores without any "master" intervention. This is an IPC optimization which reduces system "chatter" or messaging overhead. An example of a set of non-aggregated IPC transactions can be seen in Figure 4.

In Figure 4, a sub-graph is passed to the remote core in a single IPC transaction. If the remote core is sufficiently efficient at computation, the data set of sufficiently large and the IPC overhead is significant, this methodology reduces the number of IPC transactions and saves time which would be otherwise wasted with this IPC overhead.



(a) No Aggregation

(b) Aggregated

Figure 4: IPC overhead is reduced with aggregation

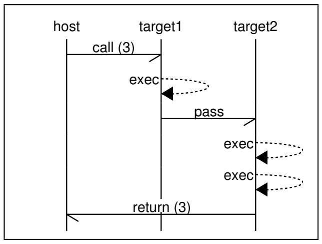


Figure 5: Peer-to-peer IPC with aggregation.

Aggregating IPC only charges the remote latency component of Equation 1 once, instead of multiplying it by the number of functions F. Instead, when F functions are aggregated together, only the execution times are largely affected, though some cache operation latency may be linearly increased due to how data is laid out. See Equation 2.

$$L_r(F) = F * (C_{lf} + C_{ri} + C_{rf} + C_{li}) + IPC_{send} + \sum^{F} exec_r(data, params) + IPC_{recv} \tag{2}$$

5.4. Peer-to-peer IPC Topologies

Use of peer-to-peer IPC topologies builds on top of the previous method by further removing the “master” that arbitrates the communication stream. This can only be fully realized in a system with more than two targets which are independently operating devices. Underlying peer-based frameworks further remove IPC overhead by directly communicating completed work to the next target (whether it is co-located or not). An example of a peer-to-peer IPC with aggregation can be seen in Figure 5.

5.5. Compilation Strategies

For implementations which utilize internally run-time-compilable kernels, many optimization strategies exist that are well known in the compiler domain. OpenVX further improves this by providing the compiler with the future call state. Strategies may include in-lining, loop unrolling, restricting pointers, replacing tail-recursive functions with their iterative counterparts, etc. These have been covered in numerous other papers and will not be improved upon here. We only note that pertinent data for those strategies is now contained in the graph and context.

5.6. Parallelism

In the example graph of Figure 2 there is an opportunity, expressed in nodes $N5$ and $N6$, for independent node execution. On platforms which support multiple targets, or if the targets support multiple execution units, these nodes can be executed in *parallel*. *Parallelism*, itself, is not directly expressed in the graph, but its prerequisite, *independence*, can be extracted from the graph structure.

5.7. Block/Tile Processing

Tilable kernels are typically ones where the output depends on only a subset of the input, not the entire data set. In Figure 6, the data to be processed is automatically broken into *tiles* which are then fed into the graph. Each node then processes only a tile’s worth of data at a time, such that the graph executes N times for a single image, where N is equal to the number of tiles in the image. This is useful for several reasons. First, if the intermediate data tiles are stored in on-chip memory, then external memory usage and memory bandwidth are reduced by a factor of how many intermediate images there otherwise would have been. This has the effect of reducing $L_r(F)$ by eliminating intermediate write/read round-trips of data to/from external memory. Second, tiling speeds up processing by keeping the graph inside a domain tied to an optimized memory interface (e.g., DMA). This will reduce C_{rf} and C_{ri} terms to zero. Finally, it is possible to use hardware which is able to *pipeline* tile processing (see below).

5.8. Pipelining

Multiple tiling kernels can be connected to form a pipeline in order to take advantage of platform resources (e.g., *DMA*, *Cache*, specialized hardware). The input data may be separated into any variety of shapes or sizes depending on system requirements, but cannot be broken down further than a basic algorithmic unit. Not all kernels share this minimum algorithmic unit and a *least common multiples* (LCM) approach may be needed to find a usable size between tiling kernels. For example, a $8 \times 3 \rightarrow 6 \times 1$ (input$\rightarrow$ output) **SIMD** algorithm tiled with a $2 \times 2 \rightarrow 2 \times 2$ algorithm will need an intermediate tile size of some multiple of 6×2. The tile’s neighborhood would then be set to 1 on each side to accommodate the additional area around the tile needed by the 8×3 input. The pipeline then becomes $8 \times 4 \rightarrow 6 \times 2$.

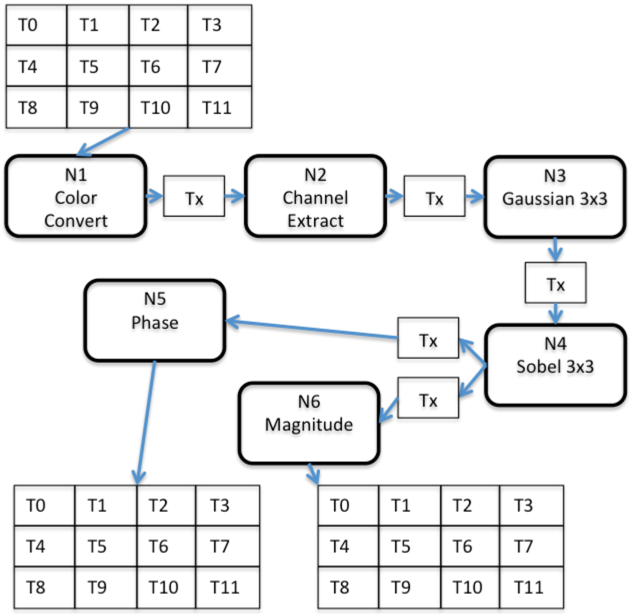


Figure 6: Memory Tiling.

5.8.1 Pipelining Targets

If there are multiple targets that are compatible with this mode of operation then a multi-target pipeline can be created. Tiles may be processed in or out of order, and concurrently or in parallel depending on system resources and graph structure.

In the example above, $N1$, $N2$, $N3$, and $N4$ may be pipelined to reduce the overall execution time of these nodes. Part of this optimization relies on the OpenVX implementation providing adequate memory bandwidth to a multi-target pipeline and the other part of the optimization requires an IPC system capable of triggering multiple targets simultaneously.

5.8.2 Multiple Invocations of Graphs

In addition to segmenting the data processing within a graph, future versions of OpenVX may be able to pipeline multiple invocations of the same graph. In this model, the graphs are scheduled asynchronously and are triggered consecutively after dependent data for each invocation has been freed for use in the next invocation.

5.9. Cache Operations

Another side-effect of having a graph model is that the system understands the data relationship between two or more nodes, and can efficiently operate on system and core caches, reducing unnecessary flushing and invalidation. Depending on the operating system latency and cache operation costs, this could result in significant savings at run time by reducing some cache operations in Equation 1 to zero.

6. Resource Management

Since the structure of the graph is known ahead of time, there can be several look-ahead resource management systems put in place. These help prevent critical resources from being over-subscribed, and divert the load to other resources. This is possible with OpenVX graphs, as the underlying kernel may be implemented for more than one target. The framework has the possibility to dynamically switch targets as needed. This opens the door to many dynamic resource management schemes. These schemes may be implemented at verification or execution time. Graphs in OpenVX may be executed synchronously (i.e., a blocking call) or asynchronously (a nonblocking call). These could possibly emanate from multiple calling threads in either case.

However, dynamic resource management is not appropriate for every user and use-case. There are use-cases where fixed-resourcing is critical which OpenVX supports by allowing the user to statically assign the target for a node (e.g., force $Gaussian_{3\times3}$ to run on the *GPU*). However, for the remainder of this section, we consider the flexible case only.

6.1. Target Execution Priority

Given that a kernel may be implemented on multiple targets available on the same platform (CPU, GPU, etc.), the framework needs to determine which one is the best (with respect to some measure, such as execution time or power usage) for a particular use-case. Typically, the primary consideration is simply performance. To that end, an implementation may institute a priority ranking scheme for searching through its kernels to find an appropriate target. In practice, this may mean that an internal table of ranked targets is searched to determine the best target for a particular kernel. Below is an example ranking based on which target the kernel runs on:

- Accelerator — fastest, but limited data type support,
- GPU — fast, but uses more power than an accelerator,
- DSP — medium, has more optimizations than CPU, but fewer functional units than a GPU,
- CPU — slowest, but permits standard C code.

Once the best target is determined, the node is assigned to that target. Other graphs with identical kernels (but separate *instantiations*) may be assigned to different targets by run-time resource manager.

These assignments would be then successfully accepted if the resource model for these targets permit such operations. If the resource is oversubscribed in some respect, the next resource in the list is selected. The following sections mention possible models to account for these constraints.

6.2. Predictive Load Balancing

A predictive load-balancing system determines whether the target can execute the nodes of the graph in the future without exceeding its capacity, and if the added load saturates (exceeds capacity), then uses the next best target to dynamically execute the node. Node "load" values must be ranked per target per *data volume* to accurately gauge the predicted load value. These load values are typically expressed as $\frac{cycles}{window}$ (over a relevant time window). Frequently the chosen window size relates to the capture or processing rate of the system. A fast resource manager is typically needed in these implementations as resource needs are scheduled, used and released, quickly. Saturation of all available targets leads to a **FIFO** style of processing. Smarter implementations can use captured latency information to re-estimate target load values for a given *data volume*.

6.3. Latency Guarantees

A latency-guarantee system attempts to maximize responsiveness of all the targets. It schedules work based on a desired worst-case expected latency L_{worst} of currently processing graphs and tries to keep each target saturated with work, but not strictly the work it is best at performing. This means work is re-scheduled or re-assigned to targets where it is possible to meet L_{worst} even if there may be a faster target available. If the best-performing (i.e., least-cost) targets are saturated, then higher-cost (slower-performing) targets are chosen as long as the worst-case response can be met. If the response cannot be met, the work either cannot be scheduled (user sees a failure), or the work is done, but latency guarantee is lost (user sees graph processing time exceeding the desired worst case).

7. Trade-offs

Programming with a graph model comes with some considerations that must be weighed to determine if the situation warrants the benefits of some of the methodologies mentioned above.

7.1. Data Size vs. Remote Processing

In some instances for HLOSes (High-Level OS, such as Linux), the latency of IPC can be on the order of *milliseconds* when considering the overhead of returning from a kernel call. Further, the latency may be non-deterministic (without an RTOS), given an unknown load upon the system. In these circumstances, if the latency of processing the nodes locally is smaller than the IPC latency and remote-processing latency, the framework should decide to not offload the work. See Equation 4.

$$L_o = L_r(1) - exec_r(data, param) \tag{3}$$

$$Location = \begin{cases} \text{if } L_o >= L_{local} \text{ then local} \\ \text{else remote} \end{cases} \tag{4}$$

7.2. Framework Overhead

Every framework comes with some non-trivial overhead cost, which the implementor should attempt to minimize. The specification [2] does not mandate any minimum or maximum latencies, but in practice there are obvious benefits to streamline graph execution as much as possible. Those overheads which are unavoidable are typically moved to other areas which are not time critical, such as verification.

7.2.1 Verification Overhead

Depending on the optimization strategies chosen by the vendor, some implementations will have a relatively small verification overhead, while others may have rather large overheads. In either case, the verification step is assumed to be a heavy-weight operation and should not be done during "tight-loop" execution time, but rather during a setup phase, and then not altered again without some transition into a phase which allows for potentially long delays.

8. Conclusion

OpenVX tackles a non-trivial problem: in a fast-paced industry where vision algorithms keep evolving, how to facilitate both system- and kernel-level optimization of *future* workloads? OpenVX prescribes the aggregation of a directed set of acyclic, data-dependency-ordered graphs, which permits system-level optimizations that are simply not possible under a *single-function* paradigm. As always, an expert implementation will be necessary to achieve good performance on any given platform, and the trade-offs implicit in this model of computation must be well understood, both by the implementers and users of the API.

References

[1] G. Dedeoglu, B. Kisacanin, D. Moore, V. Sharma, and A. Miller. An optimized vision library approach for embedded systems. In *Embed. Comp. Vis. Workshop (in CVPR)*, 2011. 1

[2] S. Gautham and E. Rainey. *The Khronos OpenVX^{TM} 1.0 Specification*. The Khronos Group, 2014. 1, 2, 6

[3] T. J. Olson, R. J. Lockwood, and J. R. Taylor. Programming a pipelined image processor. *CVGIP*, 1996. 1

[4] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. Real-time computer vision with OpenCV. *CACM*, 55(6), 2012. 1

[5] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification*. Silicon Graphics Inc., 1994. 1