The **OpenVX™** Specification

Version 1.0

Document Revision: r28647
Generated on Fri Oct 17 2014 10:15:55

Khronos Vision Working Group

Editor: Susheel Gautam
Editor: Erik Rainey

Contents

Chapter 1

Introduction

1.1 Abstract

OpenVX is a low-level programming framework domain to enable software developers to efficiently access computer vision hardware acceleration with both functional and performance portability. OpenVX has been designed to support modern hardware architectures, such as mobile and embedded SoCs as well as desktop systems. Many of these systems are parallel and heterogeneous: containing multiple processor types including multi-core CPUs, DSP subsystems, GPUs, dedicated vision computing fabrics as well as hardwired functionality. Additionally, vision system memory hierarchies can often be complex, distributed, and not fully coherent. OpenVX is designed to maximize functional and performance portability across these diverse hardware platforms, providing a computer vision framework that efficiently addresses current and future hardware architectures with minimal impact on applications.

OpenVX contains:

- a library of predefined and customizable vision functions,
- a graph-based execution model to combine function enabling both task and data-independent execution, and;
- a set of memory objects that abstract the physical memory.

OpenVX defines a C Application Programming Interface (API) for building, verifying, and coordinating graph execution, as well as for accessing memory objects. The graph abstraction enables OpenVX implementers to optimize the execution of the graph for the underlying acceleration architecture.

OpenVX also defines the `vxu` utility library, which exposes each OpenVX predefined function as a directly callable C function, without the need for first creating a graph. Applications built using the `vxu` library do not benefit from the optimizations enabled by graphs; however, the vxu library can be useful as the simplest way to use OpenVX and as first step in porting existing vision applications.

As the computer vision domain is still rapidly evolving, OpenVX provides an extensibility mechanism to enable developer-defined functions to be added to the application graph.

1.2 Purpose

The purpose of this document is to detail the Application Programming Interface (API) for OpenVX.

1.3 Scope of Specification

The document contains the definition of the OpenVX API. Tthe conformance tests that are used to determine whether an implementation is consistent to this specification are defined separately.

1.4 Normative References

The section “Module Documentation” forms the normative part of the specification. Each API definition provided in that chapter has certain preconditions and post conditions specified that are normative. If these normative conditions are not met, the behavior of the function is undefined.

1.5 Version/Change History

- OpenVX 1.0 Provisional - November, 2013
- OpenVX 1.0 Provisional V2 - June, 2014
- OpenVX 1.0 - September 2014

1.6 Requirements Language

In this specification, the words *shall* or *must* express a requirement that is binding, *should* expresses design goals or recommended actions, and *may* expresses an allowed behavior.

1.7 Typographical Conventions

The following typographical conventions are use used in this specification.

- **Bold** words indicate warnings or strongly communicated concepts that are intended to draw attention to the text.
- `Monospace` words signify an API element (i.e., class, function, structure) or a filename.
- *Italics* denote an emphasis on a particular concept, an abstraction of a concept, or signify an argument, parameter, or member.
- Throughout this specification, code examples given to highlight a particular issue use the format as shown below:
- ```
  /* Example Code Section */
  int main(int argc, char *argv[])
  {
     return 0;
  }
  ```
- Some “mscgen” message diagrams are included in this specification. The graphical conventions for this tool can be found on its website.

  See also

  http://www.mcternan.me.uk/mscgen/

1.7.1 Naming Conventions

The following naming conventions are use used in this specification.

- Opaque objects and atomics are named as `vx_object`, e.g., `vx_image` or `vx_uint8`, with an underscore separating the object name from the “vx” prefix.
- Defined Structures are named as `vx_struct_t`, e.g., `vx_imagepatch_addressing_t`, with underscores separating the structure from the “vx” prefix and a “t” to denote that it is a structure.
- Defined Enumerations are named as `vx_enum_e`, e.g., `vx_type_e`, with underscores separating the enumeration from the “vx” prefix and an “e” to denote that it is an enumerated value.
- Application Programming Interfaces are named `vxsomeFunction()` using camel case, starting with lowercase, and no underscores, e.g., `vxCreateContext()`.
- Vision functions also have a naming convention that follows a lower-case, inverse dotted hierarchy similar to Java Packages, e.g.,

  `"org.khronos.openvx.color_convert".`
  ```

This minimizes the possibility of name collisions and promotes sorting and readability when querying the namespace of available vision functions. Each vision function should have a unique dotted name of the style: *tld.vendor.library.function*. The hierarchy of such vision function namespaces is undefined outside the subdomain "org.khronos", but they do follow existing international standards. For OpenVX-specified vision functions, the "function" section of the unique name does not use camel case and uses underscores to separate words.

1.8 Glossary and Acronyms

- Atomic: The specification mentions *atomics*, which means a C primitive data type. Usages that have additional wording, such as *atomic operations* do not carry this meaning.
- API: Application Programming Interface that specifies how a software component interacts with another.
- Framework: A generic software abstraction in which users can override behaviors to produce application-specific functionality.
- Engine: A purpose-specific software abstraction that is tunable by users.
- Run-time: The execution phase of a program.
- Kernel: OpenVX uses the term *kernel* to mean an abstract *computer vision function*, not an Operating System kernel. Kernel may also refer to a set of convolution coefficients in some computer vision literature (e.g., the Sobel "kernel"). OpenVX does not use this meaning. OpenCL uses kernel (specifically `cl_kernel`) to qualify a function written in "CL" which the OpenCL may invoke directly. This is close to the meaning OpenVX uses; however, OpenVX does not define a language.

1.9 Acknowledgements

This specification would not be possible without the contributions from this partial list of the following individuals from the Khronos Working Group and the companies that they represented at the time:

- Erik Rainey - Amazon
- Susheel Gautam - QUALCOMM
- Victor Erukhimov - Itseez
- Doug Knisely - QUALCOMM
- Frank Brill - Samsung
- Kari Pulli - NVIDIA
- Thierry Lepley - NVIDIA
- Neil Trevett - NVIDIA
- Tomer Schwartz - Broadcom Corporation
- Shervin Emami - NVIDIA
- Olivier Pothier - STMicroelectronics International NV
- Andy Kuzma - Intel
- Mostafa Hagog - Intel
- Shorin Kyo - Huawei
- Renato Grottesi - ARM Limited
- Dave Schreiner - ARM Limited
- Chris Tseng - Texas Instruments, Inc.

- Daniel Laroche - CogniVue Corporation
- Andrew Garrard - Samsung Electronics
- Tomer Yanir - Samsung Electronics
- Erez Natan - Samsung Electronics
- Chang-Hyo Yu - Samsung Electronics
- Hans-Peter Nilsson - Axis Communications
- Stephen Neuendorffer - Xilinx, Inc.
- Amit Shoham - BDTi
- Paul Buxton - Imagination Technologies
- Yuki Kobayashi - Renesas Electronics
- Cormac Brick - Movidius Ltd
- Mikael Bourges-Sevenier - Aptina Imaging Corporation
- Tao Zhang - QUALCOMM
- Jesse Villareal - Texas Instruments, Inc.
- Vadim Pisarevsky - Itseez
- Andrey Kamaev - Itseez
- Vlad Vinogradov - Itseez
- Roman Donchenko - Itseez
- Alexander Alekhin - Itseez
- Radha Giduthuri - AMD
- Xin Wang - Vivante Corporation
- Anshu Arya - MulticoreWare

Chapter 2

Design Overview

2.1 Software Landscape

OpenVX is intended to be used either directly by applications or as the acceleration layer for higher-level vision frameworks, engines or platform APIs.

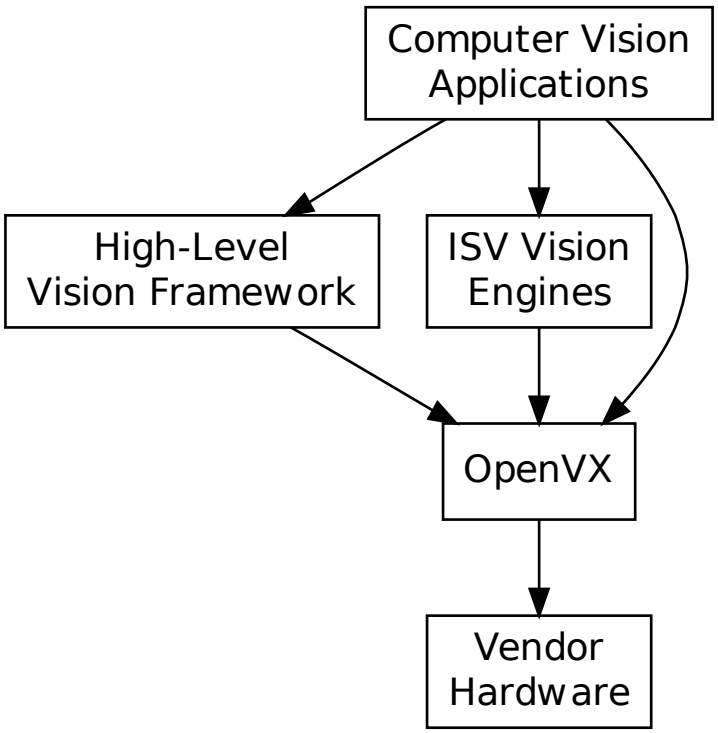


Figure 2.1: OpenVX Usage Overview

2.2 Design Objectives

OpenVX is designed as a framework of standardized computer vision functions able to run on a wide variety of platforms and potentially to be accelerated by a vendor's implementation on that platform. OpenVX can improve the

performance and efficiency of vision applications by providing an abstraction for commonly-used vision functions and an abstraction for aggregations of functions (a “graph”), thereby providing the implementer the opportunity to minimize the run-time overhead.

The functions in OpenVX 1.0 are intended to cover common functionality required by many vision applications.

2.2.1 Hardware Optimizations

This specification makes no statements as to which acceleration methodology or techniques may be used in its implementation. Vendors may choose any number of implementation methods such as parallelism and/or specialized hardware offload techniques.

This specification also makes no statement or requirements on a “level of performance” as this may vary significantly across platforms and use cases.

2.2.2 Hardware Limitations

The OpenVX 1.0 focuses on vision functions that can be significantly accelerated by diverse hardware. Future versions of this specification may adopt additional vision functions into the core standard when hardware acceleration for those functions becomes practical.

2.3 Assumptions

2.3.1 Portability

OpenVX 1.0 has been designed to maximize functional and performance portability wherever possible, while recognizing that the API is intended to be used on a wide diversity of devices with specific constraints and properties. Tradeoffs are made for portability where possible: for example, portable Graphs constructed using this API should work on any OpenVX implementation and return similar results within the precision bounds defined by the OpenVX conformance tests.

2.3.2 Opaqueness

To avoid forcing hardware-specific requirements onto any particular implementation, the API is designed to be *opaque*.

OpenVX is intended to address a very broad range of devices and platforms - from deeply embedded systems to desktop machines, and even distributed computing architectures.

The range of implementations is quite discreet, and as such, the API shall only address all these spaces through opaqueness.

All data, except client-facing structures, are opaque and hidden behind a reference that may be as thin or thick as an implementation needs. Each implementation provides the standardized interfaces for accessing data that takes care of specialized hardware, platform, or allocation requirements. Memory that is *imported* or *shared* from other APIs is not subsumed by OpenVX and is still maintained and accessible by the originator.

OpenVX does not dictate any requirements on memory allocation methods or the layout of opaque memory objects and it does not dictate byte packing or alignment for structures on architectures.

2.4 Object-Oriented Behaviors

OpenVX objects are both strongly typed at compile-time for safety critical applications and are strongly typed at run-time for dynamic applications. Each object has its typedef’d type and its associated enumerated value in the `vx_type_e` list. Any object may be down-cast to a `vx_reference` safely to be used in functions that require this, specifically `vxQueryReference`, which can be used to get the `vx_type_e` value using an `vx_enum`.

2.5 OpenVX Framework Objects

This specification defines the following OpenVX framework objects.

- Object: Context - The OpenVX context is the object domain for all OpenVX objects. All data objects *live* in the context as well as all framework objects. The OpenVX context keeps reference counts on all objects and must do garbage collection during its deconstruction to free lost references. While multiple clients may connect to the OpenVX context, all data are private in that the references that refer to data objects are given only to the creating party. The results of calling an OpenVX function on data objects created in different contexts are undefined.
- Object: Kernel - A Kernel in OpenVX is the abstract representation of a computer vision function, such as a "Sobel Gradient" or "Lucas Kanade Feature Tracking". A vision function may implement many similar or identical features from other functions, but it is still considered a single, unique kernel as long as it is named by the same string and enumeration and conforms to the results specified by OpenVX. Kernels are similar to function signatures in this regard.
- Object: Parameter - An abstract input, output, or bidirectional data object passed to a computer vision function. This object contains the signature of that parameter's usage from the kernel description. This information includes:
  - *Signature Index* - The numbered index of the parameter in the signature.
  - *Object Type* - e.g. `VX_TYPE_IMAGE`, or `VX_TYPE_ARRAY`, or some other object type from `vx_type_e`.
  - *Usage Model* - e.g. `VX_INPUT`, `VX_OUTPUT`, or `VX_BIDIRECTIONAL`.
  - *Presence State* - e.g. `VX_PARAMETER_STATE_REQUIRED`, or `VX_PARAMETER_STATE_OPTIONAL`.
- Object: Node - A node is an instance of a kernel that will be paired with a specific set of references (the parameters). Nodes are created from and associated with a single graph only. When a `vx_parameter` is extracted from a Node, an additional attribute can be accessed:
  - *Reference* - The `vx_reference` assigned to this parameter index from the Node creation function (e.g., `vxSobel3x3Node`).
- Object: Graph - A set of nodes connected in a directed (only goes one-way) acyclic (does not loop back) fashion. A Graph may have sets of Nodes that are unconnected to other sets of Nodes within the same Graph. See Graph Formalisms.

2.6 OpenVX Data Objects

Data objects are object that are processed by graphs in nodes.

- Object: Array An opaque array object that could be an array of primitive data types or an array of structures.
- Object: Convolution An opaque object that contains MxN matrix of `vx_int16` values. Also contains a scaling factor for normalization. Used specifically with `vxuConvolve` and `vxConvolveNode`.
- Object: Delay An opaque object that contains a manually controlled, temporally-delayed list of objects.
- Object: Distribution An opaque object that contains a frequency distribution (e.g., a histogram).
- Object: Image An opaque image object that may be some format in `vx_df_image_e`.
- Object: LUT An opaque lookup table object used with `vxTableLookupNode` and `vxuTableLookup`.
- Object: Matrix An opaque object that contains MxN matrix of some scalar values.
- Object: Pyramid An opaque object that contains multiple levels of scaled `vx_image` objects.
- Object: Remap An opaque object that contains the map of source points to destination points used to transform images.
- Object: Scalar An opaque object that contains a single primitive data type.
- Object: Threshold An opaque object that contains the thresholding configuration.

2.7 Error Objects

Error objects are specialized objects that may be returned from other object creator functions when serious platform issue occur (i.e., out of memory or out of handles). These can be checked at the time of creation of these objects, but checking also may be put-off until usage in other APIs or verification time, in which case, the implementation must return appropriate errors to indicate that an invalid object type was used.

```
vx_<object> obj = vxCreate<Object><Method>(context, ...);
vx_status status = vxGetStatus((vx_refernce)obj);
if (obj && status == VX_SUCCESS) {
    // object is good
}
```

2.8 Graphs Concepts

The *graph* is the central computation concept of OpenVX. The purpose of using graphs to express the Computer Vision problem is to allow for the possibility of any implementation to maximize its optimization potential because all the operations of the graph and its dependencies are known ahead of time, before the graph is processed.

Graphs are composed of one or more *nodes* that are added to the graph through node creation functions. Graphs in OpenVX must be created ahead of processing time and verified by the implementation, after which they can be processed as many times as needed.

2.8.1 Linking Nodes

Graph Nodes are linked together via data dependencies with *no explicitly-stated ordering*. The same reference may be linked to other nodes. Linking has a limitation, however, in that only one node in a graph may output to any specific data object reference. That is, only a single writer of an object may exist in a given graph. This prevents indeterminate ordering from data dependencies. All writers in a graph shall produce output data before any reader of that data accesses it.

2.8.2 Virtual Data Objects

Graphs in OpenVX depend on data objects to link together nodes. When clients of OpenVX know that they do not need access to these *intermediate* data objects, they may be created as `virtual`. Virtual data objects can be used in the same manner as non-virtual data objects to link nodes of a graph together; however, virtual data objects are different in the following respects.

- Inaccessible - No calls to an Access/Commit API shall succeed given a reference to an object created through a `virtual` create function from a Graph *external* perspective. Calls to Access/Commit from within client-defined functions may succeed as they are Graph *internal*.
- Dimensionless or Formatless - Virtual data objects may be declared to have no dimensions or format and they may return zeros or generic values for formats when queried.
- Scoped - Virtual data objects are scoped within the Graph in which they are created; they cannot be shared outside their scope.
- Intermediates - Virtual data objects should be used only for intermediate operations within Graphs, because they are fundamentally inaccessible to clients of the API.
- Optimizations - Virtual data objects do not have to be created during Graph validation and execution and therefore may be of zero *size*.

These restrictions enable vendors the ability to optimize some aspects of the data object or its usage. Some vendors may not allocate such objects, some may create intermediate sub-objects of the object, and some may allocate the object on remote, inaccessible memories. OpenVX does not proscribe *which* optimization the vendor does, merely that it *may* happen.

2.8.3 Node Parameters

Parameters to node creation functions are defined as either atomic types, such as vx_int32, vx_enum, or as objects, such as vx_scalar, vx_image. The atomic variables of the Node creation functions shall be converted by the framework into vx_scalar references for use by the Nodes. A node parameter of type vx_scalar can be changed during the graph execution; whereas, a node parameter of an atomic type (vx_int32 etc.) require at least a graph revalidation if changed. All node parameter objects may be modified by retrieving the reference to the vx_parameter via vxGetParameterByIndex, and then passing that to vxQueryParameter to retrieve the reference to the object.

```
vx_parameter param = vxGetParameterByIndex(node, p);
vx_reference ref;
vxQueryParameter(param, VX_PARAMETER_ATTRIBUTE_REF, &ref,
sizeof(ref));
```

If the type of the parameter is unknown, it may be retrieved with the same function.

```
    vx_enum type;
    vxQueryParameter(param, VX_PARAMETER_ATTRIBUTE_TYPE,
&type, sizeof(type));
    /* cast the ref to the correct vx_<type>. Atomics are now vx_scalar */
```

2.8.4 Graph Parameters

Parameters may exist on Graphs, as well. These parameters are defined by the author of the Graph and each Graph parameter is defined as a specific parameter from a Node within the Graph using vxAddParameterToGraph. Graph parameters communicate to the implementation that there are specific Node parameters that may be modified by the client between Graph executions. Additionally, they are parameters that the client may set without the reference to the Node but with the reference to the Graph using vxSetGraphParameterByIndex. This allows for the Graph authors to construct *Graph Factories*. How these factories work falls outside the scope of this document.

See also

Framework: Graph Parameters

2.8.5 Execution Model

Graphs must execute in both:

- *Synchronous blocking mode* (in that vxProcessGraph will block until the graph has completed), and in
- *Asychronous single-issue-per-reference mode* (via vxScheduleGraph and vxWaitGraph).

Asychronous Mode

In asynchronous mode, Graphs must be single-issue-per-reference. This means that given a constructed graph reference G, it may be scheduled multiple times but only executes sequentially with respect to itself. Multiple graphs references given to the asynchronous graph interface do not have a defined behavior and may execute in parallel or in series based on the behavior or the vendor's implementation.

2.8.6 Graph Formalisms

To use graphs several rules must be put in place to allow deterministic execution of Graphs. The behavior of a processGraph(G) call is determined by the structure of the Processing Graph G. The Processing Graph is a bipartite graph consisting of a set of Nodes $N_1 \ldots N_n$ and a set of data objects $d_1 \ldots d_i$. Each edge (N_x, D_y) in the graph represents a data object D_y that is written by Node N_x and each edge (D_x, N_y) represents a data object D_x that is read by Node N_y. Each edge e has a name Name(e), which gives the parameter name of the node that references the corresponding data object. Each Node Parameter also as a type Type(node, name) in {INPUT, OUTPUT, INOUT}. Some data objects are *Virtual*, and some data objects are *Delay*. Delay data objects are just collections of data objects with indexing (like an image list) and known linking points in a graph. A node may be classified as a *head node*, which has no backward dependency. Alternatively, a node may be a *dependent node*, which has a backward dependency to the head node. In addition, the Processing Graph has several restrictions:

1. *Output typing* - Every output edge (N_x, D_y) requires `Type(` N_x`, Name(` N_x`,` D_y`))` in `{OUTPUT, INOUT}`
2. *Input typing* - Every input edge (N_x, D_y) requires `Type(` N_y`, Name(` D_x`,` N_y`))` in `{INPUT}` or `{INOUT}`
3. *Single Writer* - Every data object is the target of at most one output edge.
4. *Broken Cycles* - Every cycle in G must contain at least input edge (D_x, N_y) where D_x is Delay.
5. *Virtual images must have a source* - If D_y is Virtual, then there is at least one output edge that writes D_y (N_x, D_y)
6. *Bidirectional data objects shall not be virtual* - If `Type(` N_x`, Name(` N_x`,` D_y`))` is INOUT implies D_y is non-Virtual.
7. *Delay data objects shall not be virtual* - If D_x is Delay then it shall not be Virtual.

The execution of each node in a graph consists of an atomic operation (sometimes referred to as *firing*) that consumes data representing each input data object, processes it, and produces data representing each output data object. A node may execute when all of its input edges are marked *present*. Before the graph executes, the following initial marking is used:

- All input edges (D_x, N_y) from non-Virtual objects Dx are marked (parameters must be set).
- All input edges (D_x, N_y) with an output edge (N_z, D_x) are unmarked.
- All input edges (D_x, N_y) where D_x is a Delay data object are marked.

Processing a node results in unmarking all the corresponding input edges and marking all its output edges; marking an output edge (N_x, D_y) where D_y is not a Delay results in marking all of the input edges (D_y, N_z). Following these rules, it is possible to statically schedule the nodes in a graph as follows: Construct a precedence graph P, including all the nodes $N_1 \ldots N_x$, and an edge (N_x, N_z) for every pair of edges (N_x, D_y) and (D_y, N_z) where D_y is not a Delay. Then unconditionally fire each node according to any topological sort of P.

The following assertions should be verified:

- P is a Directed Acyclic Graph (DAG), implied by 4 and the way it is constructed.
- Every data object has a value when it is executed, implied by 5, 6, 7, and the marking.
- Execution is deterministic if the nodes are deterministic, implied by 3, 4, and the marking.
- Every node completes its execution exactly once.

The execution model described here just acts as a formalism. For example, independent processing is allowed across multiple depended and depending nodes and edges, provided that the result is invariant with the execution model described here.

2.8.7 Node Execution Independence

In the following example a client computes the gradient magnitude and gradient phase from a blurred input image. The `vxMagnitudeNode` and `vxPhaseNode` are *independently* computed, in that each does not depend on the output of the other. OpenVX does not mandate that they are run simultaneously or in parallel, but it could be implemented this way by the OpenVX vendor.

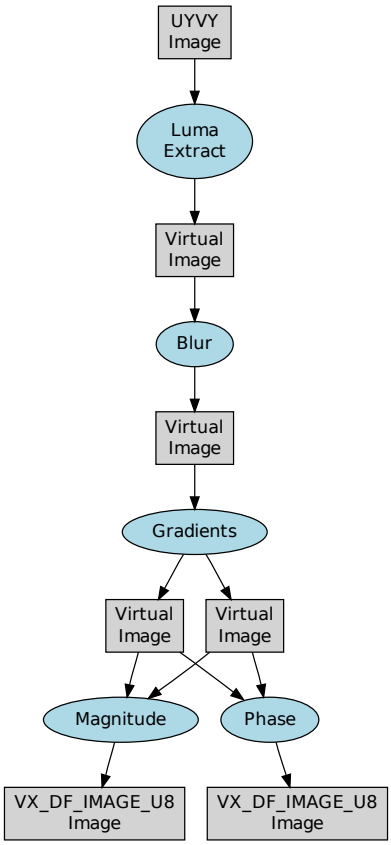


Figure 2.2: A simple graph with some independent nodes.

The code to construct such a graph can be seen below.

```
vx_context context = vxCreateContext();
vx_image images[] = {
        vxCreateImage(context, 640, 480, VX_DF_IMAGE_UYVY),
        vxCreateImage(context, 640, 480, VX_DF_IMAGE_U8),
        vxCreateImage(context, 640, 480, VX_DF_IMAGE_U8),
};
vx_graph graph = vxCreateGraph(context);
vx_image virts[] = {
        vxCreateVirtualImage(graph, 0, 0,
  VX_DF_IMAGE_VIRT),
        vxCreateVirtualImage(graph, 0, 0,
  VX_DF_IMAGE_VIRT),
        vxCreateVirtualImage(graph, 0, 0,
  VX_DF_IMAGE_VIRT),
        vxCreateVirtualImage(graph, 0, 0,
  VX_DF_IMAGE_VIRT),
};

vxChannelExtractNode(graph, images[0], VX_CHANNEL_Y, virts[0]),
vxGaussian3x3Node(graph, virts[0], virts[1]),
vxSobel3x3Node(graph, virts[1], virts[2], virts[3]),
vxMagnitudeNode(graph, virts[2], virts[3], images[1]),
vxPhaseNode(graph, virts[2], virts[3], images[2]),

status = vxVerifyGraph(graph);
if (status == VX_SUCCESS)
{
    status = vxProcessGraph(graph);
}
vxReleaseContext(&context); /* this will release everything */
```

2.8.8 Verification

Graphs within OpenVX must go through a rigorous validation process before execution to satisfy the design concept of eliminating run-time overhead (parameter checking) that guarantees safe execution of the graph. OpenVX must check for (but is not limited to) these conditions:

- Parameters To Nodes:
  - Each required parameter is given to the node (vx_parameter_state_e). Optional parameters may not be present and therefore are not checked when absent. If present, they are checked.
  - Each parameter given to a node must be of the right *direction* (a value from `vx_direction_e`).
  - Each parameter given to a node must be of the right *object type* (from the object range of `vx_type_e`).
  - Each parameter attribute or value that has algorithmic significance must be verified. In the case of a scalar value, it may need to be range checked (e.g., $0.5 <= k <= 1.0$). The implementation is not required to do run-time range checking of scalar values. If the value of the scalar changes at run time to go outside the range, the results are undefined. The rationale is that the potential performance hit for run-time range checking is too large to be enforced. It will still be checked at graph verification time as a time-zero sanity check. If the scalar is an output parameter of another node, it must be initialized to a legal value. In the case of `vxScaleImageNode`, the relation of the input image dimensions to the output image dimensions determines the scaling factor. These values or attributes of data objects must be checked for compatibility on each platform.
  - Graph Connectivity - the `vx_graph` must be a Directed Acyclic Graph (DAG). No cycles or feedback is allowed. The `vx_delay` object has been designed to explicitly address feedback between Graph executions.
  - Resolution of Virtual Data Objects - Any changes to *Virtual* data objects from unspecified to specific format or dimensions, as well as the related creation of objects of specific type that are observable at processing time, takes place at Verification time.

2.9 Callbacks

Callbacks are a method to control graph flow and to make decisions based on completed work. The `vxAssignNodeCallback` call takes as a parameter a callback function. This function will be called after the execution of the particular node, but prior to the completion of the graph. If nodes are arranged into independent sets, the order of the callbacks is unspecified. Nodes that are arranged in a serial fashion due to data dependencies perform callbacks in order. The callback function may use the node reference first to extract parameters from the node, and then extract the data references. Data outputs of Nodes with callbacks shall be available (via Access/Commit methods) when the callback is called.

2.10 User Kernels

OpenVX supports the concept of *client-defined functions* that shall be executed as *Nodes* from inside the Graph or are Graph *internal*. The purpose of this paradigm is to:

- Further exploit independent operation of nodes within the OpenVX platform.
- Allow componentized functions to be reused elsewhere in OpenVX.
- Formalize strict verification requirements (i.e., Contract Programming).

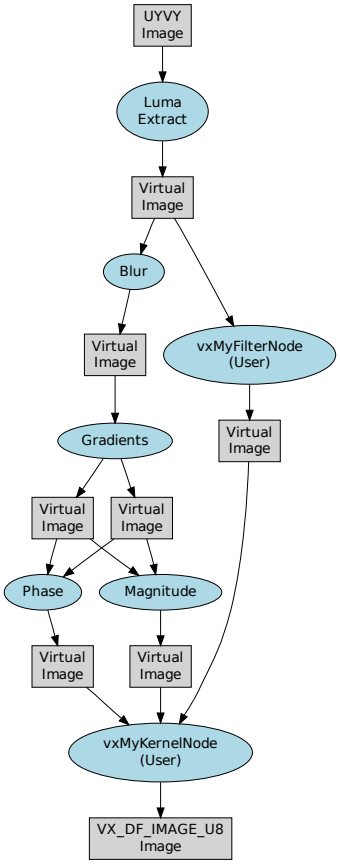


Figure 2.3: A graph with User Kernel nodes which are independent of the “base” nodes.

In this example, to execute client-supplied functions, the graph does not have to be halted and then resumed. These nodes shall be executed in an independent fashion with respect to independent base nodes within OpenVX. This allows implementations to further minimize execution time if hardware to exploit this property exists.

2.10.1 Parameter Validation

User Kernels must aid in the Graph Verification effort by providing explicit validation functions for each vision function they implement. Each parameter passed to the instanced Node of a User Kernel are validated using the client-supplied validation functions. The client must check these attributes and/or values of each parameter:

- If an attribute or value of the parameter has algorithmic significance, it must be checked. For example, the size of array, or the value of a scalar to be within a range, or a dimensionality constraint of an image such as width divisibility. (Some implementations may have restrictions, such as an image width be evenly divisible by some fixed number).
- If the output parameters depend on attributes or values from input parameters, those relationships must be checked (within the output validator).

Input validators execute before output validators. This allows any or all inputs to be used as dependents of output parameter validation.

The Meta Format Object

The Meta Format Object is an opaque object used to collect requirements about the output parameter, which then the OpenVX implementation will check. The Client must manually set relevant object attributes to be checked against output parameters, such as dimensionality, format, scaling, etc.

Delta Rectangles

There is a special case with vx_image output parameters where the User Kernel output validation function can specify a positional and/or size-related change of the valid region of the output image relative to the input image during verification time. This is intended to give the optimizer more information about memory usage, and could lead to better outcomes or different strategies. Delta rectangles (specified using the vx_delta_rectangle_t parameter) are used to update a valid region for the user kernels with a call to vxSetMetaFormatAttribute from the output validator.

For example, for a 5x5 box filter where 2 border pixels of the output are lost (invalid), and with no center shift, use:

```
vx_delta_rectangle_t delta = {2, 2, -2, -2};
```

For the same 5x5 box filter, except with a center-shift into the upper-left corner:

```
vx_delta_rectangle_t delta = {0, 0, -4, -4};
```

If this attribute has not been set prior to graph verification, the graph manager must determine the new valid region based on vxCommitImagePatch calls during the execution time.

```
// User Kernel Function is a 5x5 filter with no explicit border handling code.
// So, 2 pixels on each edge of the image will be lost.
vx_delta_rectangle_t delta = {2,2,-2,-2};
vxSetMetaFormatAttribute(meta,
      VX_META_FORMAT_ATTRIBUTE_DELTA_RECTANGLE, &delta, sizeof(delta));
```

2.10.2 User Kernels Naming Conventions

User Kernels must be exported with a unique name (see Naming Conventions for information on OpenVX conventions) and a unique enumeration. Clients of OpenVX may use either the name or enumeration to retrieve a kernel, so collisions due to non-unique names will cause problems. The kernel enumerations may be extended by following this example:

```
#define VX_KERNEL_NAME_KHR_XYZ "org.khronos.example.xyz"

#define VX_LIBRARY_XYZ (0x3) // assigned from Khronos, vendors control their own

enum vx_kernel_xyz_ext_e {
    VX_KERNEL_KHR_XYZ = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_XYZ) + 0x0,
    // up to 0xFFF kernel enums can be created.
};
```

Each vendor of a vision function or an implementation must apply to Khronos to get a unique identifier (up to a limit of $2^{12} - 1$ vendors). Until they obtain a unique ID vendors must use VX_ID_DEFAULT.

To construct a kernel enumeration, a vendor must have both their ID and a *library* ID. The library ID's are completely *vendor* defined (however when using the VX_ID_DEFAULT ID, many libraries may collide in namespace).

Once both are defined, a kernel enumeration may be constructed using the VX_KERNEL_BASE macro and an offset. (The offset is optional, but very helpful for long enumerations.)

2.11 Immediate Mode Functions

OpenVX also contains an interface defined within <VX/vxu.h> that allows for immediate execution of vision functions. These interfaces are prefixed with vxu to distinguish them from the Node interfaces, which are of the form vx<Name>Node. Each of these interfaces replicates a Node interface with some exceptions. Immediate mode functions are defined to *behave* as *Single Node Graphs*, which have no leaking side-effects (e.g., no Log entries) within the Graph Framework after the function returns. The following tables refer to both the Immediate Mode and Graph Mode vision functions. The Module documentation for each vision function draws a distinction on each API by noting that it is either an immediate mode function with the tag [Immediate] or it is a Graph mode function by the tag [Graph].

2.12 Base Vision Functions

OpenVX comes with a standard or *base* set of vision functions. The following table lists the supported set of vision functions, their input types (first table) and output types (second table), and the version of OpenVX in which they are supported.

2.12.1 Inputs

Vision Function	U8	U16	S16	S32	U32	F32	4CC
AbsDiff	1.0						
Accumulate	1.0						
AccumulateSquared	1.0						
AccumulateWeighted	1.0						
Add	1.0		1.0				
And	1.0						
Box3x3	1.0						
CannyEdgeDetector	1.0						
ChannelCombine	1.0						
ChannelExtract							1.0
ColorConvert							1.0
ConvertDepth	1.0		1.0				
Convolve	1.0						
Dilate3x3	1.0						
EqualizeHistogram	1.0						
Erode3x3	1.0						
FastCorners	1.0						
Gaussian3x3	1.0						
HarrisCorners	1.0						
HalfScaleGaussian	1.0						
Histogram	1.0						
IntegralImage	1.0						
TableLookup	1.0						
Magnitude			1.0				
MeanStdDev	1.0						
Median3x3	1.0						
MinMaxLoc	1.0		1.0				
Multiply	1.0		1.0				
Not	1.0						

Optical↩ FlowLK	1.0						
Or	1.0						
Phase			1.0				
Gaussian↩ Pyramid	1.0						
Remap	1.0						
Scale↩ Image	1.0						
Sobel3x3	1.0						
Subtract	1.0		1.0				
Threshold	1.0						
WarpAffine	1.0						
Warp↩ Perspective	1.0						
Xor	1.0						

2.12.2 Outputs

Vision Function	U8	U16	S16	U32	S32	F32	4CC
AbsDiff	1.0						
Accumu- late			1.0				
Accumulate↩ Squared			1.0				
Accumulate↩ Weighted	1.0						
Add	1.0		1.0				
And	1.0						
Box3x3	1.0						
Canny↩ Edge↩ Detector	1.0						
Channel↩ Combine							1.0
Channel↩ Extract	1.0						
Color↩ Convert							1.0
Convert↩ Depth	1.0		1.0				
Convolve	1.0		1.0				
Dilate3x3	1.0						
Equalize↩ Histogram	1.0						
Erode3x3	1.0						
Fast↩ Corners	1.0						

Gaus-sian3x3	1.0						
Harris↩ Corners	1.0						
Half↩ Scale↩ Gaussian	1.0						
Histogram					1.0		
Integral↩ Image				1.0			
Table↩ Lookup	1.0						
Magnitude			1.0				
MeanStd↩ Dev						1.0	
Median3x3	1.0						
MinMax↩ Loc	1.0		1.0		1.0		
Multiply	1.0		1.0				
Not	1.0						
Optical↩ FlowLK				1.0			
Or	1.0						
Phase	1.0						
Gaussian↩ Pyramid	1.0						
Remap	1.0						
Scale↩ Image	1.0						
Sobel3x3			1.0				
Subtract	1.0		1.0				
Threshold	1.0						
WarpAffine	1.0						
Warp↩ Perspective	1.0						
Xor	1.0						

2.13 Lifecycles

2.13.1 OpenVX Context Lifecycle

The lifecycle of the context is very simple.

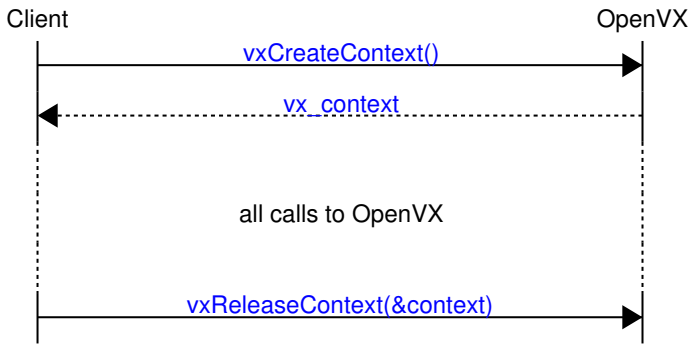


Figure 2.4: The lifecycle model for an OpenVX Context.

2.13.2 Graph Lifecycle

OpenVX has four main phases of graph lifecycle:

- Construction - Graphs are created via `vxCreateGraph`, and Nodes are connected together by data objects.
- Verification - The graphs are checked for consistency, correctness, and other conditions. Memory allocation may occur.
- Execution - The graphs are executed via `vxProcessGraph` or `vxScheduleGraph`. Between executions data may be updated by the client or some other external mechanism. The client of OpenVX may change reference of input data to a graph, but this may require the graph to be validated again by checking `vxIsGraphVerified`.
- Deconstruction - Graphs are released via `vxReleaseGraph`. All Nodes in the Graph are released.

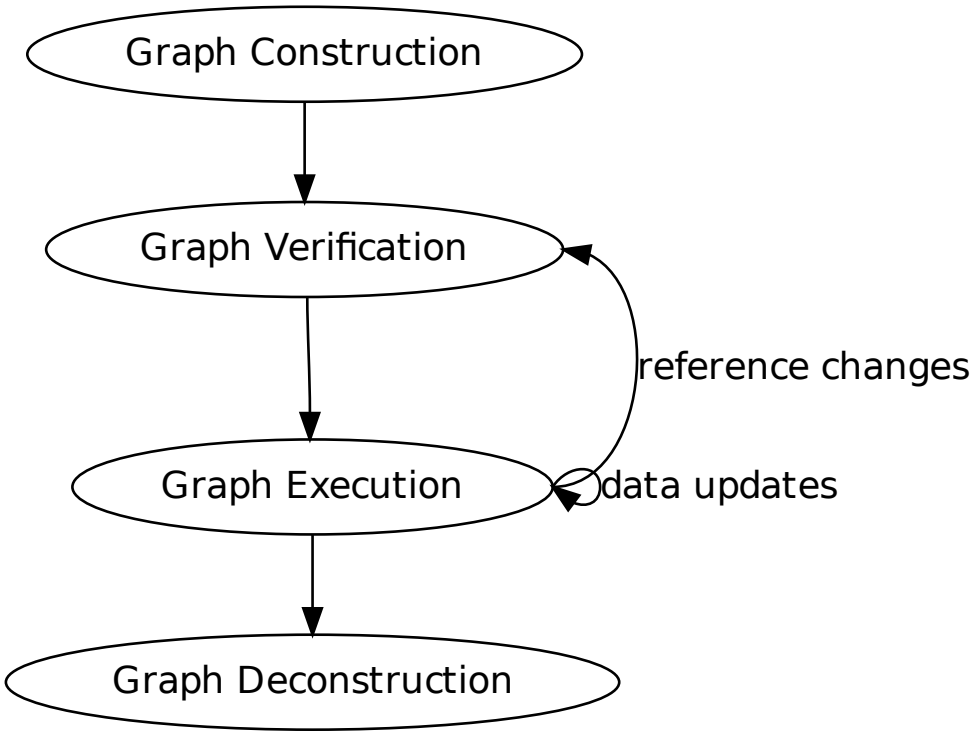


Figure 2.5: Graph Lifecycle

2.13.3 Data Object Lifecycle

All objects in OpenVX follow a similar lifecycle model. All objects are

- Created via `vxCreate<Object><Method>` or retreived via `vxGet<Object><Method>` from the parent object if they are internally created.
- Used within Graphs or immediate functions as needed.
- Then objects must be released via `vxRelease<Object>` or via `vxReleaseContext` when all objects are released.

OpenVX Image Lifecycle

This is an example of the Image Lifecycle using the OpenVX Framework API. This would also apply to other data types with changes to the types and function names.

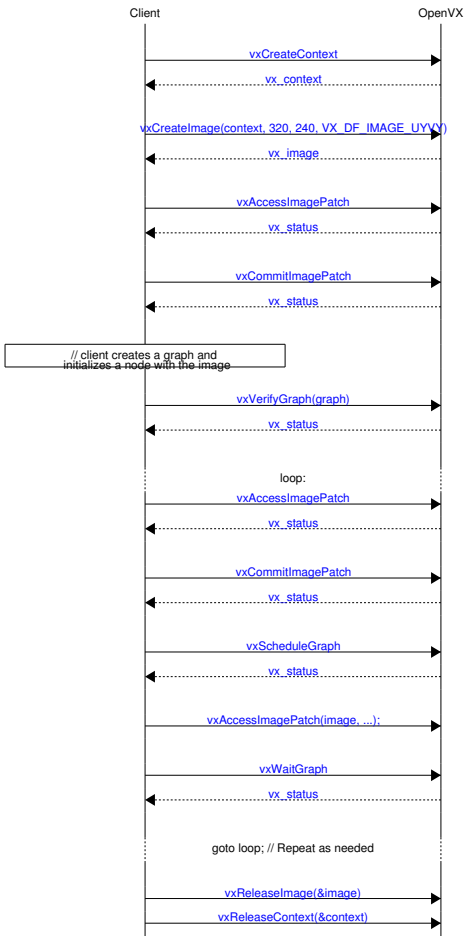


Figure 2.6: Image Object Lifecycle

2.14 Host Memory Data Object Access Patterns

For objects retrieved from OpenVX that are 2D in nature, such as `vx_image`, `vx_matrix`, and `vx_convolution`, the manner in which the host-side has access to these memory regions is well-defined. OpenVX uses a row-major storage (that is each unit in a column is memory-adjacent to its row adjacent unit). Two-dimensional objects are always created (using `vxCreateImage` or `vxCreateMatrix`) in width (columns) by height (rows) notation, with the arguments in that order. When accessing these structures in "C" with two-dimensional arrays of declared size, the user must therefore provide the array dimensions in the reverse of the order of the arguments to the Create function. This layout ensures *row-wise* storage in C on the host. A pointer could also be allocated for the matrix data and would have to be indexed in this row-major method.

2.14.1 Matrix Access Example

```
    const vx_size columns = 3;
    const vx_size rows = 4;
    vx_matrix matrix = vxCreateMatrix(context,
      VX_TYPE_FLOAT32, columns, rows);
    if (matrix)
    {
        vx_int32 j, i;
#if defined(OPENVX_USE_C99)
        vx_float32 mat[rows][columns]; /* note: row major */
#else
```

```
        vx_float32 *mat = (vx_float32 *)malloc(rows*columns*sizeof(
      vx_float32));
#endif
        if (vxAccessMatrix(matrix, mat) == VX_SUCCESS) {
            for (j = 0; j < rows; j++)
                for (i = 0; i < columns; i++)
#if defined(OPENVX_USE_C99)
                    mat[j][i] = (vx_float32)rand()/(vx_float32)RAND_MAX;
#else
                    mat[j*columns + i] = (vx_float32)rand()/(
      vx_float32)RAND_MAX;
#endif
            vxCommitMatrix(matrix, mat);
        }
#if !defined(OPENVX_USE_C99)
        free(mat);
#endif
    }
```

2.14.2 Image Access Example

Images and Array differ slightly in how they are accessed due to more complex memory layout requirements.

```
    vx_status status = VX_SUCCESS;
    void *base_ptr = NULL;
    vx_uint32 width = 640, height = 480, plane = 0;
    vx_image image = vxCreateImage(context, width, height,
      VX_DF_IMAGE_U8);
    vx_rectangle_t rect;
    vx_imagepatch_addressing_t addr;

    rect.start_x = rect.start_y = 0;
    rect.end_x = rect.end_y = PATCH_DIM;

    status = vxAccessImagePatch(image, &rect, plane,
                                &addr, &base_ptr,
                                VX_READ_AND_WRITE);
    if (status == VX_SUCCESS)
    {
        vx_uint32 x,y,i,j;
        vx_uint8 pixel = 0;

        /* a couple addressing options */

        /* use linear addressing function/macro */
        for (i = 0; i < addr.dim_x*addr.dim_y; i++) {
            vx_uint8 *ptr2 = vxFormatImagePatchAddress1d(base_ptr,
                                                         i, &addr);
            *ptr2 = pixel;
        }

        /* 2d addressing option */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *ptr2 = vxFormatImagePatchAddress2d(base_ptr,
                                                             x, y, &addr);
                *ptr2 = pixel;
            }
        }

        /* direct addressing by client
         * for subsampled planes, scale will change
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = ((addr.stride_y*y*addr.scale_y) /
                      VX_SCALE_UNITY) +
                    ((addr.stride_x*x*addr.scale_x) /
                      VX_SCALE_UNITY);
                tmp[i] = pixel;
            }
        }

        /* more efficient direct addressing by client.
         * for subsampled planes, scale will change.
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            j = (addr.stride_y*y*addr.scale_y)/VX_SCALE_UNITY;
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = j + (addr.stride_x*x*addr.scale_x) /
                    VX_SCALE_UNITY;
                tmp[i] = pixel;
            }
```

```
        }

        /* this commits the data back to the image. If rect were 0 or empty, it
         * would just decrement the reference (used when reading an image only)
         */
        status = vxCommitImagePatch(image, &rect, plane, &addr, base_ptr);
    }
    vxReleaseImage(&image);
```

2.14.3 Array Access Example

Arrays only require a single value, the stride, instead of the entire addressing structure that images need.

```
        vx_size i, stride = 0ul;
        void *base = NULL;
        /* access entire array at once */
        vxAccessArrayRange(array, 0, num_items, &stride, &base,
      VX_READ_AND_WRITE);
        for (i = 0; i < num_items; i++)
        {
            vxArrayItem(mystruct, base, i, stride).some_uint += i;
            vxArrayItem(mystruct, base, i, stride).some_double = 3.14f;
        }
        vxCommitArrayRange(array, 0, num_items, base);
```

Access/Commit pairs can also be called on individual elements of array using a method similar to this:

```
        /* access each array item individually */
        for (i = 0; i < num_items; i++)
        {
            mystruct *myptr = NULL;
            vxAccessArrayRange(array, i, i+1, &stride, (void **)&myptr,
      VX_READ_AND_WRITE);
            myptr->some_uint += 1;
            myptr->some_double = 3.14f;
            vxCommitArrayRange(array, i, i+1, (void *)myptr);
        }
```

2.15 Extending OpenVX

Beyond User Kernels there are other mechanisms for vendors to extend features in OpenVX. These mechanisms are not available to User Kernels.

2.15.1 Extending Attributes

When extending attributes, vendors *must* use their assigned ID from `vx_vendor_id_e` in conjunction with the appropriate macros for creating new attributes with `VX_ATTRIBUTE_BASE`. The typical mechanism to extend a new attribute for some object type (for example a `vx_node` attribute from `VX_ID_TI`) would look like this:

```
enum {
   VX_NODE_ATTRIBUTE_TI_NEWTHING = VX_ATTRIBUTE_BASE(VX_ID_TI,
      VX_TYPE_NODE) + 0x0,
}
```

2.15.2 Vendor Custom Kernels

Vendors wanting to add more kernels to the base set supplied to OpenVX should provide a header of the form

```
#include <VX/vx_ext_<vendor>.h>
```

that contains definitions of each of the following.

- New Node Creation Function Prototype per function.

```
    vx_node vxXYZNode(vx_graph graph, vx_image input,
          vx_uint32 value, vx_image output, vx_array temp);
```

- A new Kernel Enumeration(s) and Kernel String per function.

```
#define VX_KERNEL_NAME_KHR_XYZ "org.khronos.example.xyz"

#define VX_LIBRARY_XYZ (0x3) // assigned from Khronos, vendors control their own

enum vx_kernel_xyz_ext_e {
    VX_KERNEL_KHR_XYZ = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_XYZ) + 0x0,
    // up to 0xFFF kernel enums can be created.
};
```

- A new VXU Function per function.

```
vx_status vxuXYZ(vx_context context, vx_image input,
      vx_uint32 value, vx_image output, vx_array temp);
```

This should come with good documentation for each new part of the extension. Ideally, these sorts of extensions should not require linking to new objects to facilitate usage.

2.15.3 Vendor Custom Extensions

Some extensions affect *base* vision functions and thus may be invisible to most users. In these circumstances, the vendor must report the supported extensions to the base nodes through the VX_CONTEXT_ATTRIBUTE_EXTENSIONS attribute on the context.

```
vx_char *tmp, *extensions = NULL;
vx_size size = 0ul;
vxQueryContext(context,VX_CONTEXT_ATTRIBUTE_EXTENSIONS_SIZE
  ,&size,sizeof(size));
extensions = malloc(size);
vxQueryContext(context,VX_CONTEXT_ATTRIBUTE_EXTENSIONS,
               extensions, size);
```

Extensions in this list are dependent on the extension itself; they may or may not have a header and new kernels or framework feature or data objects. The common feature is that they are implemented and supported by the implementation vendor.

2.15.4 Hinting

The specification defines a Hinting API that allows Clients to feed information to the implementation for *optional* behavior changes. See Framework: Hints. It is assumed that most of the hints will be vendor- or implementation-specific. Check with the OpenVX implementation vendor for information on vendor-specific extensions.

2.15.5 Directives

The specification defines a Directive API to control implementation behavior. See Framework: Directives. This *may* allow things like disabling parallelism for debugging, enabling cache writing-through for some buffers, or any implementation-specific optimization.

2.16 Known Extensions to OpenVX

2.16.1 User Kernel Tiling

The User Kernel Tiling facility enables optimizations of the user kernels (e.g., locality of execution or parallelism) when performing computation on the image data. Modern processors have a diverse memory hierarchy that varies from relatively small but fast and expensive memory to relatively large but slow and inexpensive memory. Image data are typically too large to fit into the fast but small memory. The ability to break the image data into smaller sized units allows for optimized computation on these smaller units with fast memory access or parallel execution of a user kernel on multiple image tiles simultaneously. The OpenVX Graph Manager possesses the knowledge about the memory hierarchy of the platform and is hence in a position to break the image data into smaller units for memory optimization. Knowledge of the memory access pattern of an algorithm is key for the graph manager to enable optimizations.

The Khronos OpenVX Working Group will include this extension as part of the OpenVX 1.1 specification, contingent on community feedback.

Chapter 3

Module Documentation

3.1 Vision Functions

3.1.1 Detailed Description

These are the base vision functions supported in OpenVX 1.0.

These functions were chosen as a subset of a larger pool of possible functions that fall under the following criteria:

- Applicable to Acceleration Hardware
- Very Common Usage
- Encumbrance Free

Modules

- Absolute Difference

  Computes the absolute difference between two images.
- Accumulate

  Accumulates an input image into output image.
- Accumulate Squared

  Accumulates a squared value from an input image to an output image.
- Accumulate Weighted

  Accumulates a weighted value from an input image to an output image.
- Arithmetic Addition

  Performs addition between two images.
- Arithmetic Subtraction

  Performs subtraction between two images.
- Bitwise AND

  Performs a bitwise AND operation between two `VX_DF_IMAGE_U8` images.
- Bitwise EXCLUSIVE OR

  Performs a bitwise EXCLUSIVE OR (XOR) operation between two `VX_DF_IMAGE_U8` images.
- Bitwise INCLUSIVE OR

  Performs a bitwise INCLUSIVE OR operation between two `VX_DF_IMAGE_U8` images.
- Bitwise NOT

  Performs a bitwise NOT operation on a `VX_DF_IMAGE_U8` input image.
- Box Filter

  Computes a Box filter over a window of the input image.
- Canny Edge Detector

  Provides a Canny edge detector kernel.
- Channel Combine

Implements the Channel Combine Kernel.

- Channel Extract

  Implements the Channel Extraction Kernel.
- Color Convert

  Implementes the Color Conversion Kernel.
- Convert Bit depth

  Converts image bit depth.
- Custom Convolution

  Convolves the input with the client supplied convolution matrix.
- Dilate Image

  Implements Dilation, which grows the white space in a `VX_DF_IMAGE_U8` Boolean image.
- Equalize Histogram

  Equalizes the histogram of a grayscale image.
- Erode Image

  Implements Erosion, which shrinks the white space in a `VX_DF_IMAGE_U8` Boolean image.
- Fast Corners

  Computes the corners in an image using a method based upon FAST9 algorithm suggested in [3] and with some updates from [4] with modifications described below.
- Gaussian Filter

  Computes a Gaussian filter over a window of the input image.
- Harris Corners

  Computes the Harris Corners of an image.
- Histogram

  Generates a distribution from an image.
- Gaussian Image Pyramid

  Computes a Gaussian Image Pyramid from an input image.
- Integral Image

  Computes the integral image of the input.
- Magnitude

  Implements the Gradient Magnitude Computation Kernel.
- Mean and Standard Deviation

  Computes the mean pixel value and the standard deviation of the pixels in the input image (which has a dimension width and height).
- Median Filter

  Computes a median pixel value over a window of the input image.
- Min, Max Location

  Finds the minimum and maximum values in an image and a location for each.
- Optical Flow Pyramid (LK)

  Computes the optical flow using the Lucas-Kanade method between two pyramid images.
- Phase

  Implements the Gradient Phase Computation Kernel.
- Pixel-wise Multiplication

  Performs element-wise multiplication between two images and a scalar value.
- Remap

  Maps output pixels in an image from input pixels in an image.
- Scale Image

  Implements the Image Resizing Kernel.
- Sobel 3x3

  Implements the Sobel Image Filter Kernel.
- TableLookup

  Implements the Table Lookup Image Kernel.

- Thresholding

  Thresholds an input image and produces an output Boolean image.
- Warp Affine

  Performs an affine transform on an image.
- Warp Perspective

  Performs a perspective transform on an image.

3.2 Absolute Difference

3.2.1 Detailed Description

Computes the absolute difference between two images.

Absolute Difference is computed by:

$$out(x,y) = |in_1(x,y) - in_2(x,y)|$$

Functions

- vx_node vxAbsDiffNode (vx_graph graph, vx_image in1, vx_image in2, vx_image out)

  [Graph] Creates an AbsDiff node.
- vx_status vxuAbsDiff (vx_context context, vx_image in1, vx_image in2, vx_image out)

  [Immediate] Computes the absolute difference between two images.

3.2.2 Function Documentation

vx_node vxAbsDiffNode (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*)

[Graph] Creates an AbsDiff node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	An input image in VX_DF_IMAGE_U8
in	*in2*	An input image in VX_DF_IMAGE_U8
out	*out*	The output image in VX_DF_IMAGE_U8

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuAbsDiff (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*)

[Immediate] Computes the absolute difference between two images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	An input image
in	*in2*	An input image
out	*out*	The output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.3 Accumulate

3.3.1 Detailed Description

Accumulates an input image into output image.

Accumulation is computed by:

$$accum(x,y) = accum(x,y) + input(x,y)$$

The overflow policy used is VX_CONVERT_POLICY_SATURATE.

Functions

- vx_node vxAccumulateImageNode (vx_graph graph, vx_image input, vx_image accum)

  [Graph] Creates an accumulate node.
- vx_status vxuAccumulateImage (vx_context context, vx_image input, vx_image accum)

  [Immediate] Computes an accumulation.

3.3.2 Function Documentation

vx_node vxAccumulateImageNode (vx_graph *graph*, vx_image *input*, vx_image *accum*)

[Graph] Creates an accumulate node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in,out	*accum*	The accumulation image in VX_DF_IMAGE_S16.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuAccumulateImage (vx_context *context*, vx_image *input*, vx_image *accum*)

[Immediate] Computes an accumulation.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in,out	*accum*	The accumulation image in VX_DF_IMAGE_S16

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.4 Accumulate Squared

3.4.1 Detailed Description

Accumulates a squared value from an input image to an output image.

Accumulate squares is computed by:

$$accum(x,y) = saturate_{int16}((uint16)accum(x,y) + (((uint16)input(x,y)^2) >> (shift)))$$

Where $0 \leq shift \leq 15$

The overflow policy used is VX_CONVERT_POLICY_SATURATE.

Functions

- vx_node vxAccumulateSquareImageNode (vx_graph graph, vx_image input, vx_scalar shift, vx_image accum)

  [Graph] Creates an accumulate square node.
- vx_status vxuAccumulateSquareImage (vx_context context, vx_image input, vx_scalar shift, vx_image accum)

  [Immediate] Computes a squared accumulation.

3.4.2 Function Documentation

vx_node vxAccumulateSquareImageNode (vx_graph *graph,* vx_image *input,* vx_scalar *shift,* vx_image *accum*)

[Graph] Creates an accumulate square node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*shift*	The input VX_TYPE_UINT32 with a value in the range of $0 \leq shift \leq 15$.
in,out	*accum*	The accumulation image in VX_DF_IMAGE_S16.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuAccumulateSquareImage (vx_context *context,* vx_image *input,* vx_scalar *shift,* vx_image *accum*)

[Immediate] Computes a squared accumulation.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*shift*	A VX_TYPE_UINT32 type, the input value with the range $0 \leq shift \leq 15$.
in,out	*accum*	The accumulation image in VX_DF_IMAGE_S16

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.5 Accumulate Weighted

3.5.1 Detailed Description

Accumulates a weighted value from an input image to an output image.

Weighted accumulation is computed by:

$$accum(x,y) = (1-\alpha) * accum(x,y) + \alpha * input(x,y)$$

Where $0 \leq \alpha \leq 1$ Conceptually, the rounding for this is defined as:

$$output(x,y) = uint8((1-\alpha) * float32(int32(output(x,y))) + \alpha * float32(int32(input(x,y))))$$

Functions

- vx_node vxAccumulateWeightedImageNode (vx_graph graph, vx_image input, vx_scalar alpha, vx_image accum)

  [Graph] Creates a weighted accumulate node.
- vx_status vxuAccumulateWeightedImage (vx_context context, vx_image input, vx_scalar scale, vx_image accum)

  [Immediate] Computes a weighted accumulation.

3.5.2 Function Documentation

vx_node vxAccumulateWeightedImageNode (vx_graph *graph*, vx_image *input*, vx_scalar *alpha*, vx_image *accum*)

[Graph] Creates a weighted accumulate node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*alpha*	The input VX_TYPE_FLOAT32 scalar value with a value in the range of $0.0 \leq \alpha \leq 1.0$.
in,out	*accum*	The VX_DF_IMAGE_U8 accumulation image.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuAccumulateWeightedImage (vx_context *context*, vx_image *input*, vx_scalar *scale*, vx_image *accum*)

[Immediate] Computes a weighted accumulation.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*scale*	A VX_TYPE_FLOAT32 type, the input value with the range $0.0 \leq \alpha \leq 1.0$.
in,out	*accum*	The VX_DF_IMAGE_U8 accumulation image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.6 Arithmetic Addition

3.6.1 Detailed Description

Performs addition between two images.

Arithmetic addition is performed between the pixel values in two VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 images. The output image can be VX_DF_IMAGE_U8 only if both source images are VX_DF_IMAGE_U8 and the output image is explicitly set to VX_DF_IMAGE_U8. It is otherwise VX_DF_IMAGE_S16. If one of the input images is of type VX_DF_IMAGE_S16, all values are converted to VX_DF_IMAGE_S16. The overflow handling is controlled by an overflow-policy parameter. For each pixel value in the two input images:

$$out(x,y) = in_1(x,y) + in_2(x,y)$$

Functions

- vx_node vxAddNode (vx_graph graph, vx_image in1, vx_image in2, vx_enum policy, vx_image out)

  [Graph] Creates an arithmetic addition node.
- vx_status vxuAdd (vx_context context, vx_image in1, vx_image in2, vx_enum policy, vx_image out)

  [Immediate] Performs arithmetic addition on pixel values in the input images.

3.6.2 Function Documentation

vx_node vxAddNode (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*)

[Graph] Creates an arithmetic addition node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16.
in	*in2*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16.
in	*policy*	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
out	*out*	The output image, a VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 image.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuAdd (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_enum *policy*, vx_image *out*)

[Immediate] Performs arithmetic addition on pixel values in the input images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image.
in	*in2*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image.
in	*policy*	A vx_convert_policy_e enumeration.
out	*out*	The output image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.7 Arithmetic Subtraction

3.7.1 Detailed Description

Performs subtraction between two images.

Arithmetic subtraction is performed between the pixel values in two VX_DF_IMAGE_U8 or two VX_DF_IMAGE_S16 images. The output image can be VX_DF_IMAGE_U8 only if both source images are VX_DF_IMAGE_U8 and the output image is explicitly set to VX_DF_IMAGE_U8. It is otherwise VX_DF_IMAGE_S16. If one of the input images is of type VX_DF_IMAGE_S16, all values are converted to VX_DF_IMAGE_S16. The overflow handling is controlled by an overflow-policy parameter. For each pixel value in the two input images:

$$out(x,y) = in_1(x,y) - in_2(x,y)$$

Functions

- vx_node vxSubtractNode (vx_graph graph, vx_image in1, vx_image in2, vx_enum policy, vx_image out)

  [Graph] Creates an arithmetic subtraction node.
- vx_status vxuSubtract (vx_context context, vx_image in1, vx_image in2, vx_enum policy, vx_image out)

  [Immediate] Performs arithmetic subtraction on pixel values in the input images.

3.7.2 Function Documentation

vx_node vxSubtractNode (vx_graph *graph,* vx_image *in1,* vx_image *in2,* vx_enum *policy,* vx_image *out*)

[Graph] Creates an arithmetic subtraction node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16, the minuend.
in	*in2*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16, the subtrahend.
in	*policy*	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
out	*out*	The output image, a VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 image.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuSubtract (vx_context *context,* vx_image *in1,* vx_image *in2,* vx_enum *policy,* vx_image *out*)

[Immediate] Performs arithmetic subtraction on pixel values in the input images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image, the minuend.
in	*in2*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image, the subtrahend.
in	*policy*	A vx_convert_policy_e enumeration.

out	*out*	The output image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.8 Bitwise AND

3.8.1 Detailed Description

Performs a *bitwise AND* operation between two VX_DF_IMAGE_U8 images.

Bitwise AND is computed by the following, for each bit in each pixel in the input images:

$$out(x,y) = in_1(x,y) \wedge in_2(x,y)$$

Or expressed as C code:

```
out(x,y) = in_1(x,y) & in_2(x,y)
```

Functions

- vx_node vxAndNode (vx_graph graph, vx_image in1, vx_image in2, vx_image out)

  [Graph] Creates a bitwise AND node.
- vx_status vxuAnd (vx_context context, vx_image in1, vx_image in2, vx_image out)

  [Immediate] Computes the bitwise and between two images.

3.8.2 Function Documentation

vx_node vxAndNode (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*)

[Graph] Creates a bitwise AND node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	A VX_DF_IMAGE_U8 input image.
in	*in2*	A VX_DF_IMAGE_U8 input image.
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuAnd (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*)

[Immediate] Computes the bitwise and between two images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 input image
in	*in2*	A VX_DF_IMAGE_U8 input image
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.9 Bitwise EXCLUSIVE OR

3.9.1 Detailed Description

Performs a *bitwise EXCLUSIVE OR* (XOR) operation between two VX_DF_IMAGE_U8 images.

Bitwise XOR is computed by the following, for each bit in each pixel in the input images:

$$out(x,y) = in_1(x,y) \oplus in_2(x,y)$$

Or expressed as C code:

```
out(x,y) = in_1(x,y) ^ in_2(x,y)
```

Functions

- vx_status vxuXor (vx_context context, vx_image in1, vx_image in2, vx_image out)

  [Immediate] Computes the bitwise exclusive-or between two images.
- vx_node vxXorNode (vx_graph graph, vx_image in1, vx_image in2, vx_image out)

  [Graph] Creates a bitwise EXCLUSIVE OR node.

3.9.2 Function Documentation

vx_node vxXorNode (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*)

[Graph] Creates a bitwise EXCLUSIVE OR node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	A VX_DF_IMAGE_U8 input image.
in	*in2*	A VX_DF_IMAGE_U8 input image.
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuXor (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*)

[Immediate] Computes the bitwise exclusive-or between two images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 input image
in	*in2*	A VX_DF_IMAGE_U8 input image
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.10 Bitwise INCLUSIVE OR

3.10.1 Detailed Description

Performs a *bitwise INCLUSIVE OR* operation between two VX_DF_IMAGE_U8 images.

Bitwise INCLUSIVE OR is computed by the following, for each bit in each pixel in the input images:

$$out(x,y) = in_1(x,y) \vee in_2(x,y)$$

Or expressed as C code:

```
out(x,y) = in_1(x,y) | in_2(x,y)
```

Functions

- vx_node vxOrNode (vx_graph graph, vx_image in1, vx_image in2, vx_image out)

  [Graph] Creates a bitwise INCLUSIVE OR node.
- vx_status vxuOr (vx_context context, vx_image in1, vx_image in2, vx_image out)

  [Immediate] Computes the bitwise inclusive-or between two images.

3.10.2 Function Documentation

vx_node vxOrNode (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_image *out*)

[Graph] Creates a bitwise INCLUSIVE OR node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	A VX_DF_IMAGE_U8 input image.
in	*in2*	A VX_DF_IMAGE_U8 input image.
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuOr (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_image *out*)

[Immediate] Computes the bitwise inclusive-or between two images.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 input image
in	*in2*	A VX_DF_IMAGE_U8 input image
out	*out*	The VX_DF_IMAGE_U8 output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.11 Bitwise NOT

3.11.1 Detailed Description

Performs a *bitwise NOT* operation on a VX_DF_IMAGE_U8 input image.

Bitwise NOT is computed by the following, for each bit in each pixel in the input image:

$$out(x,y) = \overline{in(x,y)}$$

Or expressed as C code:

```
out(x,y) = ~in_1(x,y)
```

Functions

- vx_node vxNotNode (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a bitwise NOT node.
- vx_status vxuNot (vx_context context, vx_image input, vx_image output)

  [Immediate] Computes the bitwise not of an image.

3.11.2 Function Documentation

vx_node vxNotNode (vx_graph *graph*, vx_image *input*, vx_image *output*)

[Graph] Creates a bitwise NOT node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	A VX_DF_IMAGE_U8 input image.
out	*output*	The VX_DF_IMAGE_U8 output image.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuNot (vx_context *context*, vx_image *input*, vx_image *output*)

[Immediate] Computes the bitwise not of an image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The VX_DF_IMAGE_U8 input image
out	*output*	The VX_DF_IMAGE_U8 output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.12 Box Filter

3.12.1 Detailed Description

Computes a Box filter over a window of the input image.

This filter uses the following convolution matrix:

$$\mathbf{K}_{box} = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{vmatrix} * \frac{1}{9}$$

Functions

- vx_node vxBox3x3Node (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a Box Filter Node.
- vx_status vxuBox3x3 (vx_context context, vx_image input, vx_image output)

  [Immediate] Computes a box filter on the image by a 3x3 window.

3.12.2 Function Documentation

vx_node vxBox3x3Node (vx_graph *graph*, vx_image *input*, vx_image *output*)

[Graph] Creates a Box Filter Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuBox3x3 (vx_context *context*, vx_image *input*, vx_image *output*)

[Immediate] Computes a box filter on the image by a 3x3 window.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.13 Canny Edge Detector

3.13.1 Detailed Description

Provides a Canny edge detector kernel.

This function implements an edge detection algorithm similar to that described in [2]. The main components of the algorithm are:

- Gradient magnitude and orientation computation using a noise resistant operator (Sobel).
- Non-maximum suppression of the gradient magnitude, using the gradient orientation information.
- Tracing edges in the modified gradient image using hysteresis thresholding to produce a binary result.

The details of each of these steps are described below.

- **Gradient Computation:** Conceptually, the input image is convolved with vertical and horizontal Sobel kernels of the size indicated by the *gradient_size* parameter. The Sobel kernels used for the gradient computation shall be as shown below. The two resulting directional gradient images (dx and dy) are then used to compute a gradient magnitude image and a gradient orientation image. The norm used to compute the gradient magnitude is indicated by the *norm_type* parameter, so the magnitude may be $|dx|+|dy|$ for VX_NORM_L1 or $\sqrt{dx^2+dy^2}$ for VX_NORM_L2. The gradient orientation image is quantized into 4 values: 0, 45, 90, and 135 degrees.
- For gradient size 3:

$$\mathbf{sobel}_x = \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}$$

$$\mathbf{sobel}_y = transpose(sobel_x) = \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix}$$

- For gradient size 5:

$$\mathbf{sobel}_x = \begin{vmatrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 4 \\ -1 & -2 & 0 & 2 & 1 \end{vmatrix}$$

$$\mathbf{sobel}_y = transpose(sobel_x)$$

- For gradient size 7:

$$\mathbf{sobel}_x = \begin{vmatrix} -1 & -4 & -5 & 0 & 5 & 4 & 1 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -15 & -60 & -75 & 0 & 75 & 60 & 15 \\ -20 & -80 & -100 & 0 & 100 & 80 & 20 \\ -15 & -60 & -75 & 0 & 75 & 60 & 15 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -1 & -4 & -5 & 0 & 5 & 4 & 1 \end{vmatrix}$$

$$\mathbf{sobel}_y = transpose(sobel_x)$$

- **Non-Maximum Suppression:** This is then applied such that a pixel is retained as a potential edge pixel if and only if its magnitude is greater than or equal to the pixels in the direction perpendicular to its edge orientation. For example, if the pixel's orientation is 0 degrees, it is only retained if its gradient magnitude is larger than that of the pixels at 90 and 270 degrees to it. If a pixel is suppressed via this condition, it must not appear as an edge pixel in the final output, i.e., its value must be 0 in the final output.
- **Edge Tracing:** The final edge pixels in the output are identified via a double thresholded hysteresis procedure. All retained pixels with magnitude above the *high* threshold are marked as known edge pixels (valued 255) in the final output image. All pixels with magnitudes less than or equal to the *low* threshold must not be marked as edge pixels in the final output. For the pixels in between the thresholds, edges are traced and marked as edges (255) in the output. This can be done by starting at the known edge pixels and moving in all eight directions recursively until the gradient magnitude is less than or equal to the low threshold.

- **Caveats:** The intermediate results described above are conceptual only; so for example, the implementation may not actually construct the gradient images and non-maximum-suppressed images. Only the final binary (0 or 255 valued) output image must be computed so that it matches the result of a final image constructed as described above.

Enumerations

- enum vx_norm_type_e {
  VX_NORM_L1 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NORM_TYPE << 12)) + 0x0,
  VX_NORM_L2 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_NORM_TYPE << 12)) + 0x1 }

  A normalization type.

Functions

- vx_node vxCannyEdgeDetectorNode (vx_graph graph, vx_image input, vx_threshold hyst, vx_int32 gradient_size, vx_enum norm_type, vx_image output)

  [Graph] Creates a Canny Edge Detection Node.
- vx_status vxuCannyEdgeDetector (vx_context context, vx_image input, vx_threshold hyst, vx_int32 gradient_size, vx_enum norm_type, vx_image output)

  [Immediate] Computes Canny Edges on the input image into the output image.

3.13.2 Enumeration Type Documentation

enum vx_norm_type_e

A normalization type.

See also

Canny Edge Detector

Enumerator

VX_NORM_L1 The L1 normalization.

VX_NORM_L2 The L2 normalization.

Definition at line 1100 of file vx_types.h.

3.13.3 Function Documentation

vx_node vxCannyEdgeDetectorNode (vx_graph *graph,* vx_image *input,* vx_threshold *hyst,* vx_int32 *gradient_size,* vx_enum *norm_type,* vx_image *output*)

[Graph] Creates a Canny Edge Detection Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*hyst*	The double threshold for hysteresis.
in	*gradient_size*	The size of the Sobel filter window, must support at least 3, 5, and 7.
in	*norm_type*	A flag indicating the norm used to compute the gradient, VX_NORM_L1 or VX_NORM_L2.
out	*output*	The output image in VX_DF_IMAGE_U8 format with values either 0 or 255.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuCannyEdgeDetector (vx_context *context*, vx_image *input*, vx_threshold *hyst*, vx_int32 *gradient_size*, vx_enum *norm_type*, vx_image *output*)

[Immediate] Computes Canny Edges on the input image into the output image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*hyst*	The double threshold for hysteresis.
in	*gradient_size*	The size of the Sobel filter window, must support at least 3, 5 and 7.
in	*norm_type*	A flag indicating the norm used to compute the gradient, VX_NORM_L1 or VX_NORM_L2.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.14 Channel Combine

3.14.1 Detailed Description

Implements the Channel Combine Kernel.

This kernel takes multiple VX_DF_IMAGE_U8 planes to recombine them into a multi-planar or interleaved format from vx_df_image_e. The user must specify only the number of channels that are appropriate for the combining operation. If a user specifies more channels than necessary, the operation results in an error. For the case where the destination image is a format with subsampling, the input channels are expected to have been subsampled before combining (by stretching and resizing).

Functions

- vx_node vxChannelCombineNode (vx_graph graph, vx_image plane0, vx_image plane1, vx_image plane2, vx_image plane3, vx_image output)

  [Graph] Creates a channel combine node.
- vx_status vxuChannelCombine (vx_context context, vx_image plane0, vx_image plane1, vx_image plane2, vx_image plane3, vx_image output)

  [Immediate] Invokes an immediate Channel Combine.

3.14.2 Function Documentation

vx_node vxChannelCombineNode (vx_graph *graph*, vx_image *plane0*, vx_image *plane1*, vx_image *plane2*, vx_image *plane3*, vx_image *output*)

[Graph] Creates a channel combine node.

Parameters

in	*graph*	The graph reference.
in	*plane0*	The plane that forms channel 0. Must be VX_DF_IMAGE_U8.
in	*plane1*	The plane that forms channel 1. Must be VX_DF_IMAGE_U8.
in	*plane2*	[optional] The plane that forms channel 2. Must be VX_DF_IMAGE_U8.
in	*plane3*	[optional] The plane that forms channel 3. Must be VX_DF_IMAGE_U8.
out	*output*	The output image. The format of the image must be defined, even if the image is virtual.

See also

VX_KERNEL_CHANNEL_COMBINE

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuChannelCombine (vx_context *context*, vx_image *plane0*, vx_image *plane1*, vx_image *plane2*, vx_image *plane3*, vx_image *output*)

[Immediate] Invokes an immediate Channel Combine.

Parameters

in	*context*	The reference to the overall context.
in	*plane0*	The plane that forms channel 0. Must be VX_DF_IMAGE_U8.
in	*plane1*	The plane that forms channel 1. Must be VX_DF_IMAGE_U8.
in	*plane2*	[optional] The plane that forms channel 2. Must be VX_DF_IMAGE_U8.
in	*plane3*	[optional] The plane that forms channel 3. Must be VX_DF_IMAGE_U8.
out	*output*	The output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.15 Channel Extract

3.15.1 Detailed Description

Implements the Channel Extraction Kernel.

This kernel removes a single VX_DF_IMAGE_U8 channel (plane) from a multi-planar or interleaved image format from vx_df_image_e.

Functions

- vx_node vxChannelExtractNode (vx_graph graph, vx_image input, vx_enum channel, vx_image output)

  [Graph] Creates a channel extract node.
- vx_status vxuChannelExtract (vx_context context, vx_image input, vx_enum channel, vx_image output)

  [Immediate] Invokes an immediate Channel Extract.

3.15.2 Function Documentation

vx_node vxChannelExtractNode (vx_graph *graph,* vx_image *input,* vx_enum *channel,* vx_image *output*)

[Graph] Creates a channel extract node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image. Must be one of the defined vx_df_image_e multi-planar formats.
in	*channel*	The vx_channel_e channel to extract.
out	*output*	The output image. Must be VX_DF_IMAGE_U8.

See also

VX_KERNEL_CHANNEL_EXTRACT

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuChannelExtract (vx_context *context,* vx_image *input,* vx_enum *channel,* vx_image *output*)

[Immediate] Invokes an immediate Channel Extract.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image. Must be one of the defined vx_df_image_e multiplanar formats.
in	*channel*	The vx_channel_e enumeration to extract.
out	*output*	The output image. Must be VX_DF_IMAGE_U8.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.16 Color Convert

3.16.1 Detailed Description

Implementes the Color Conversion Kernel.

This kernel converts an image of a designated vx_df_image_e format to another vx_df_image_e format for those combinations listed in the below table, where the columns are output types and the rows are input types. The API version first supporting the conversion is also listed.

I/O	RGB	RGBX	NV12	NV21	UYVY	YUYV	IYUV	YUV4
RGB		1.0	1.0				1.0	1.0
RGBX	1.0		1.0				1.0	1.0
NV12	1.0	1.0					1.0	1.0
NV21	1.0	1.0					1.0	1.0
UYVY	1.0	1.0	1.0				1.0	
YUYV	1.0	1.0	1.0				1.0	
IYUV	1.0	1.0	1.0					1.0
YUV4								

The vx_df_image_e encoding, held in the VX_IMAGE_ATTRIBUTE_FORMAT attribute, describes the data layout. The interpretation of the colors is determined by the VX_IMAGE_ATTRIBUTE_SPACE (see vx_color_space_e) and VX_IMAGE_ATTRIBUTE_RANGE (see vx_channel_range_e) attributes of the image. OpenVX 1.0 implementations are required only to support images of VX_COLOR_SPACE_BT709 and VX_CHANNEL_RANGE_FULL.

If the channel range is defined as VX_CHANNEL_RANGE_FULL, the conversion between the real number and integer quantizations of color channels is defined for red, green, blue, and Y as:

$$value_{real} = \frac{value_{integer}}{256.0}$$

$$value_{integer} = max(0, min(255, floor(value_{real} * 256.0)))$$

For the U and V channels, the conversion between real number and integer quantizations is:

$$value_{real} = \frac{(value_{integer} - 128.0)}{256.0}$$

$$value_{integer} = max(0, min(255, floor((value_{real} * 256.0) + 128)))$$

If the channel range is defined as VX_CHANNEL_RANGE_RESTRICTED, the conversion between the integer quantizations of color channels and the continuous representations is defined for red, green, blue, and Y as:

$$value_{real} = \frac{(value_{integer} - 16.0)}{219.0}$$

$$value_{integer} = max(0, min(255, floor((value_{real} * 219.0) + 16.5)))$$

For the U and V channels, the conversion between real number and integer quantizations is:

$$value_{real} = \frac{(value_{integer} - 128.0)}{224.0}$$

$$value_{integer} = max(0, min(255, floor((value_{real} * 224.0) + 128.5)))$$

The conversions between nonlinear-intensity Y'PbPr and R'G'B' real numbers are:

$$R' = Y' + 2(1 - K_r)Pr$$

$$B' = Y' + 2(1 - K_b)Pb$$

$$G' = Y' - \frac{2(K_r(1 - K_r)Pr + K_b(1 - K_b)Pb)}{1 - K_r - K_b}$$

$$Y' = (K_r * R') + (K_b * B') + (1 - K_r - K_b)G'$$

$$Pb = \frac{B'}{2} - \frac{(R' * K_r) + G'(1 - K_r - K_b)}{2(1 - K_b)}$$

$$Pr = \frac{R'}{2} - \frac{(B' * K_b) + G'(1 - K_r - K_b)}{2(1 - K_r)}$$

The means of reconstructing Pb and Pr values from chroma-downsampled formats is implementation-defined. In VX_COLOR_SPACE_BT601_525 or VX_COLOR_SPACE_BT601_625:

$$K_r = 0.299$$
$$K_b = 0.114$$

In VX_COLOR_SPACE_BT709:

$$K_r = 0.2126$$
$$K_b = 0.0722$$

In all cases, for the purposes of conversion, these colour representations are interpreted as nonlinear in intensity, as defined by the BT.601, BT.709, and sRGB specifications. That is, the encoded colour channels are nonlinear R', G' and B', Y', Pb, and Pr.

Each channel of the R'G'B' representation can be converted to and from a linear-intensity RGB channel by these formulae:

$$value_{nonlinear} = 1.099 * value_{linear}{}^{0.45} - 0.099 \quad for \quad 1 \geq value_{linear} \geq 0.018$$
$$value_{nonlinear} = 4.500 * value_{linear} \quad for \quad 0.018 > value_{linear} \geq 0$$

$$value_{linear} = \left(\frac{value_{nonlinear} + 0.099}{1.099} \right)^{\frac{1}{0.45}} \quad for \quad 1 \geq value_{nonlinear} > 0.081$$
$$value_{linear} = \frac{value_{nonlinear}}{4.5} \quad for \quad 0.081 \geq value_{nonlinear} \geq 0$$

As the different color spaces have different RGB primaries, a conversion between them must transform the color coordinates into the new RGB space. Working with linear RGB values, the conversion formulae are:

$$R_{BT601\_525} = R_{BT601\_625} * 1.112302 + G_{BT601\_625} * -0.102441 + B_{BT601\_625} * -0.009860$$
$$G_{BT601\_525} = R_{BT601\_625} * -0.020497 + G_{BT601\_625} * 1.037030 + B_{BT601\_625} * -0.016533$$
$$B_{BT601\_525} = R_{BT601\_625} * 0.001704 + G_{BT601\_625} * 0.016063 + B_{BT601\_625} * 0.982233$$

$$R_{BT601\_525} = R_{BT709} * 1.065379 + G_{BT709} * -0.055401 + B_{BT709} * -0.009978$$
$$G_{BT601\_525} = R_{BT709} * -0.019633 + G_{BT709} * 1.036363 + B_{BT709} * -0.016731$$
$$B_{BT601\_525} = R_{BT709} * 0.001632 + G_{BT709} * 0.004412 + B_{BT709} * 0.993956$$

$$R_{BT601\_625} = R_{BT601\_525} * 0.900657 + G_{BT601\_525} * 0.088807 + B_{BT601\_525} * 0.010536$$
$$G_{BT601\_625} = R_{BT601\_525} * 0.017772 + G_{BT601\_525} * 0.965793 + B_{BT601\_525} * 0.016435$$
$$B_{BT601\_625} = R_{BT601\_525} * -0.001853 + G_{BT601\_525} * -0.015948 + B_{BT601\_525} * 1.017801$$

$$R_{BT601\_625} = R_{BT709} * 0.957815 + G_{BT709} * 0.042185$$
$$G_{BT601\_625} = G_{BT709}$$
$$B_{BT601\_625} = G_{BT709} * -0.011934 + B_{BT709} * 1.011934$$

$$R_{BT709} = R_{BT601\_525} * 0.939542 + G_{BT601\_525} * 0.050181 + B_{BT601\_525} * 0.010277$$
$$G_{BT709} = R_{BT601\_525} * 0.017772 + G_{BT601\_525} * 0.965793 + B_{BT601\_525} * 0.016435$$
$$B_{BT709} = R_{BT601\_525} * -0.001622 + G_{BT601\_525} * -0.004370 + B_{BT601\_525} * 1.005991$$

$$R_{BT709} = R_{BT601\_625} * 1.044043 + G_{BT601\_625} * -0.044043$$
$$G_{BT709} = G_{BT601\_625}$$
$$B_{BT709} = G_{BT601\_625} * 0.011793 + B_{BT601\_625} * 0.988207$$

A conversion between one YUV color space and another may therefore consist of the following transformations:

1. Convert quantized Y'CbCr ("YUV") to continuous, nonlinear Y'PbPr.
2. Convert continuous Y'PbPr to continuous, nonlinear R'G'B'.
3. Convert nonlinear R'G'B' to linear-intensity RGB (gamma-correction).
4. Convert linear RGB from the first color space to linear RGB in the second color space.
5. Convert linear RGB to nonlinear R'G'B' (gamma-conversion).
6. Convert nonlinear R'G'B' to Y'PbPr.
7. Convert continuous Y'PbPr to quantized Y'CbCr ("YUV").

   The above formulae and constants are defined in the ITU BT.601 and BT.709 specifications. The formulae for converting between RGB primaries can be derived from the specified primary chromaticity values and the specified white point by solving for the relative intensity of the primaries.

Functions

- vx_node vxColorConvertNode (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a color conversion node.
- vx_status vxuColorConvert (vx_context context, vx_image input, vx_image output)

  [Immediate] Invokes an immediate Color Conversion.

3.16.2 Function Documentation

vx_node vxColorConvertNode (vx_graph *graph,* vx_image *input,* vx_image *output*)

[Graph] Creates a color conversion node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image from which to convert.
out	*output*	The output image to which to convert.

See also

VX_KERNEL_COLOR_CONVERT

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle..

vx_status vxuColorConvert (vx_context *context,* vx_image *input,* vx_image *output*)

[Immediate] Invokes an immediate Color Conversion.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image.
out	*output*	The output image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.17 Convert Bit depth

3.17.1 Detailed Description

Converts image bit depth.

This kernel converts an image from some source bit-depth to another bit-depth as described by the table below. If the input value is unsigned the shift must be in zeros. If the input value is signed, the shift used must be an arithmetic shift. The columns in the table below are the output types and the rows are the input types. The API version on which conversion is supported is also listed. (An *X* denotes an invalid operation.)

I/O	U8	U16	S16	U32	S32
U8	X		1.0		
U16		X	X		
S16	1.0	X	X		
U32				X	X
S32				X	X

Down-conversions with VX_CONVERT_POLICY_WRAP follow this equation:

```
output(x,y) = ((uint8)(input(x,y) >> shift));
```

Down-conversions with VX_CONVERT_POLICY_SATURATE follow this equation:

```
int16 value = input(x,y) >> shift;
value = value < 0 ? 0 : value;
value = value > 255 ? 255 : value;
output(x,y) = (uint8)value;
```

Up-conversions ignore the policy and perform this operation:

```
output(x,y) = ((int16)input(x,y)) << shift;
```

The valid values for 'shift' are as specified below, all other values produce undefined behavior.

```
0 <= shift < 8;
```

Functions

- vx_node vxConvertDepthNode (vx_graph graph, vx_image input, vx_image output, vx_enum policy, vx_scalar shift)

  [Graph] Creates a bit-depth conversion node.
- vx_status vxuConvertDepth (vx_context context, vx_image input, vx_image output, vx_enum policy, vx_int32 shift)

  [Immediate] Converts the input images bit-depth into the output image.

3.17.2 Function Documentation

vx_node vxConvertDepthNode (vx_graph *graph*, vx_image *input*, vx_image *output*, vx_enum *policy*, vx_scalar *shift*)

[Graph] Creates a bit-depth conversion node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image.
out	*output*	The output image.
in	*policy*	A scalar containing a VX_TYPE_ENUM of the vx_convert_policy_e enumeration.

in	*shift*	A scalar containing a VX_TYPE_INT32 of the shift value.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuConvertDepth (vx_context *context*, vx_image *input*, vx_image *output*, vx_enum *policy*, vx_int32 *shift*)

[Immediate] Converts the input images bit-depth into the output image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image.
out	*output*	The output image.
in	*policy*	A vx_convert_policy_e enumeration.
in	*shift*	The shift value.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e..

3.18 Custom Convolution

3.18.1 Detailed Description

Convolves the input with the client supplied convolution matrix.

The client can supply a vx_int16 typed convolution matrix $C_{m,n}$. Outputs will be in the VX_DF_IMAGE_S16 format unless a VX_DF_IMAGE_U8 image is explicitly provided. If values would have been out of range of U8 for VX_DF_IMAGE_U8, the values are clamped to 0 or 255.

$$k_0 = \frac{m}{2} \tag{3.1}$$

$$l_0 = \frac{n}{2} \tag{3.2}$$

$$sum = \sum_{k=0,l=0}^{k=m-1,l=n-1} input(x+k_0-k, y+l_0-l)C_{k,l} \tag{3.3}$$

Note

The above equation for this function is different than an equivalent operation suggested by the OpenCV Filter2D function.

This translates into the C declaration:

```
// A horizontal Scharr gradient operator with different scale.
vx_int16 gx[3][3] = {
    {  3, 0, -3},
    { 10, 0,-10},
    {  3, 0, -3},
};
vx_uint32 scale = 9;
vx_convolution scharr_x = vxCreateConvolution(context, 3, 3);
vxAccessConvolutionCoefficients(scharr_x, NULL);
vxCommitConvolutionCoefficients(scharr_x, (
  vx_int16*)gx);
vxSetConvolutionAttribute(scharr_x,
  VX_CONVOLUTION_ATTRIBUTE_SCALE, &scale, sizeof(scale));
```

For VX_DF_IMAGE_U8 output, an additional step is taken:

$$output(x,y) = \begin{cases} 0 & \text{if } sum < 0 \\ 255 & \text{if } sum/scale > 255 \\ sum/scale & \text{otherwise} \end{cases}$$

For VX_DF_IMAGE_S16 output, the summation is simply set to the output

$$output(x,y) = sum/scale$$

The overflow policy used is VX_CONVERT_POLICY_SATURATE.

Functions

- vx_node vxConvolveNode (vx_graph graph, vx_image input, vx_convolution conv, vx_image output)

  [Graph] Creates a custom convolution node.
- vx_status vxuConvolve (vx_context context, vx_image input, vx_convolution matrix, vx_image output)

  [Immediate] Computes a convolution on the input image with the supplied matrix.

3.18.2 Function Documentation

vx_node vxConvolveNode (vx_graph *graph,* vx_image *input,* vx_convolution *conv,* vx_image *output*)

[Graph] Creates a custom convolution node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
in	*conv*	The vx_int16 convolution matrix.
out	*output*	The output image in VX_DF_IMAGE_S16 format.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuConvolve (vx_context *context,* vx_image *input,* vx_convolution *matrix,* vx_image *output*)

[Immediate] Computes a convolution on the input image with the supplied matrix.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
in	*matrix*	The convolution matrix.
out	*output*	The output image in VX_DF_IMAGE_S16 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.19 Dilate Image

3.19.1 Detailed Description

Implements Dilation, which *grows* the white space in a VX_DF_IMAGE_U8 Boolean image.

This kernel uses a 3x3 box around the output pixel used to determine value.

$$dst(x,y) = \max_{\substack{x-1 \le x' \le x+1 \\ y-1 \le y' \le y+1}} src(x',y')$$

Functions

- vx_node vxDilate3x3Node (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a Dilation Image Node.
- vx_status vxuDilate3x3 (vx_context context, vx_image input, vx_image output)

  [Immediate] Dilates an image by a 3x3 window.

3.19.2 Function Documentation

vx_node vxDilate3x3Node (vx_graph *graph*, vx_image *input*, vx_image *output*)

[Graph] Creates a Dilation Image Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuDilate3x3 (vx_context *context*, vx_image *input*, vx_image *output*)

[Immediate] Dilates an image by a 3x3 window.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.20 Equalize Histogram

3.20.1 Detailed Description

Equalizes the histogram of a grayscale image.

This kernel uses Histogram Equalization to modify the values of a grayscale image so that it will automatically have a standardized brightness and contrast.

Functions

- vx_node vxEqualizeHistNode (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a Histogram Equalization node.
- vx_status vxuEqualizeHist (vx_context context, vx_image input, vx_image output)

  [Immediate] Equalizes the Histogram of a grayscale image.

3.20.2 Function Documentation

vx_node vxEqualizeHistNode (vx_graph *graph*, vx_image *input*, vx_image *output*)

[Graph] Creates a Histogram Equalization node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The grayscale input image in VX_DF_IMAGE_U8.
out	*output*	The grayscale output image of type VX_DF_IMAGE_U8 with equalized brightness and contrast.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuEqualizeHist (vx_context *context*, vx_image *input*, vx_image *output*)

[Immediate] Equalizes the Histogram of a grayscale image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The grayscale input image in VX_DF_IMAGE_U8
out	*output*	The grayscale output image of type VX_DF_IMAGE_U8 with equalized brightness and contrast.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.21 Erode Image

3.21.1 Detailed Description

Implements Erosion, which *shrinks* the white space in a VX_DF_IMAGE_U8 Boolean image.

This kernel uses a 3x3 box around the output pixel used to determine value.

$$dst(x,y) = \min_{\substack{x-1 \le x' \le x+1 \\ y-1 \le y' \le y+1}} src(x',y')$$

Functions

- vx_node vxErode3x3Node (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates an Erosion Image Node.
- vx_status vxuErode3x3 (vx_context context, vx_image input, vx_image output)

  [Immediate] Erodes an image by a 3x3 window.

3.21.2 Function Documentation

vx_node vxErode3x3Node (vx_graph *graph*, vx_image *input*, vx_image *output*)

[Graph] Creates an Erosion Image Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuErode3x3 (vx_context *context*, vx_image *input*, vx_image *output*)

[Immediate] Erodes an image by a 3x3 window.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.22 Fast Corners

3.22.1 Detailed Description

Computes the corners in an image using a method based upon FAST9 algorithm suggested in [3] and with some updates from [4] with modifications described below.

It extracts corners by evaluating pixels on the Bresenham circle around a candidate point. If N contiguous pixels are brighter than the candidate point by at least a threshold value t or darker by at least t , then the candidate point is considered to be a corner. For each detected corner, its strength is computed. Optionally, a non-maxima suppression step is applied on all detected corners to remove multiple or spurious responses.

3.22.2 Segment Test Detector

The FAST corner detector uses the pixels on a Bresenham circle of radius 3 (16 pixels) to classify whether a candidate point p is actually a corner, given the following variables.

$$I = \text{input image} \tag{3.4}$$
$$p = \text{candidate point position for a corner} \tag{3.5}$$
$$I_p = \text{image intensity of the candidate point in image } I \tag{3.6}$$
$$x = \text{pixel on the Bresenham circle around the candidate point } p \tag{3.7}$$
$$I_x = \text{image intensity of the candidate point} \tag{3.8}$$
$$t = \text{intensity difference threshold for a corner} \tag{3.9}$$
$$N = \text{minimum number of contiguous pixel to detect a corner} \tag{3.10}$$
$$S = \text{set of contiguous pixel on the Bresenham circle around the candidate point} \tag{3.11}$$
$$C_p = \text{corner response at corner location } p \tag{3.12}$$
$$\tag{3.13}$$

The two conditions for FAST corner detection can be expressed as:

- C1: A set of N contiguous pixels S, $\forall x$ in S, $I_x > I_p + t$
- C2: A set of N contiguous pixels S, $\forall x$ in S, $I_x < I_p - t$

So when either of these two conditions is met, the candidate p is classified as a corner.

In this version of the FAST algorithm, the minimum number of contiguous pixels N is 9 (FAST9).

The value of the intensity difference threshold *strength_thresh*. of type VX_TYPE_FLOAT32 must be within:

$$UINT8_{MIN} < t < UINT8_{MAX}$$

These limits are established due to the input data type VX_DF_IMAGE_U8.

Corner Strength Computation Once a corner has been detected, its strength (response, saliency, or score) shall be computed if nonmax_suppression is set to true, otherwise the value of strength is undefined. The corner response C_p function is defined as the largest threshold t for which the pixel p remains a corner.

Non-maximum suppression If the `nonmax_suppression` flag is true, a non-maxima suppression step is applied on the detected corners. The corner with coordinates (x,y) is kept if and only if

$$C_p(x,y) \geq C_p(x-1,y-1) \text{ and } C_p(x,y) \geq C_p(x,y-1) \text{ and}$$
$$C_p(x,y) \geq C_p(x+1,y-1) \text{ and } C_p(x,y) \geq C_p(x-1,y) \text{ and}$$
$$C_p(x,y) > C_p(x+1,y) \text{ and } C_p(x,y) > C_p(x-1,y+1) \text{ and}$$
$$C_p(x,y) > C_p(x,y+1) \text{ and } C_p(x,y) > C_p(x+1,y+1)$$

See also

http://www.edwardrosten.com/work/fast.html
http://en.wikipedia.org/wiki/Features_from_accelerated_segment_test

Functions

- vx_node vxFastCornersNode (vx_graph graph, vx_image input, vx_scalar strength_thresh, vx_bool nonmax_suppression, vx_array corners, vx_scalar num_corners)

  [Graph] Creates a FAST Corners Node.
- vx_status vxuFastCorners (vx_context context, vx_image input, vx_scalar strength_thresh, vx_bool nonmax↩ _suppression, vx_array corners, vx_scalar num_corners)

  [Immediate] Computes corners on an image using FAST algorithm and produces the array of feature points.

3.22.3 Function Documentation

vx_node vxFastCornersNode (vx_graph *graph,* vx_image *input,* vx_scalar *strength_thresh,* vx_bool *nonmax_suppression,* vx_array *corners,* vx_scalar *num_corners*)

[Graph] Creates a FAST Corners Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*strength_thresh*	Threshold on difference between intensity of the central pixel and pixels on Bresenham's circle of radius 3 (VX_TYPE_FLOAT32 scalar).
in	*nonmax_↩ suppression*	If true, non-maximum suppression is applied to detected corners before being placed in the vx_array of VX_TYPE_KEYPOINT objects.
out	*corners*	Output corner vx_array of VX_TYPE_KEYPOINT.
out	*num_corners*	The total number of detected corners in image (optional).

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuFastCorners (vx_context *context,* vx_image *input,* vx_scalar *strength_thresh,* vx_bool *nonmax_suppression,* vx_array *corners,* vx_scalar *num_corners*)

[Immediate] Computes corners on an image using FAST algorithm and produces the array of feature points.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*strength_thresh*	Threshold on difference between intensity of the central pixel and pixels on Bresenham's circle of radius 3 (VX_TYPE_FLOAT32 scalar)
in	*nonmax_↩ suppression*	If true, non-maximum suppression is applied to detected corners before being places in the vx_array of VX_TYPE_KEYPOINT structs.
out	*corners*	Output corner vx_array of VX_TYPE_KEYPOINT.
out	*num_corners*	The total number of detected corners in image (optional).

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.23 Gaussian Filter

3.23.1 Detailed Description

Computes a Gaussian filter over a window of the input image.

This filter uses the following convolution matrix:

$$\mathbf{K}_{gaussian} = \begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix} * \frac{1}{16}$$

Functions

- vx_node vxGaussian3x3Node (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a Gaussian Filter Node.
- vx_status vxuGaussian3x3 (vx_context context, vx_image input, vx_image output)

  [Immediate] Computes a gaussian filter on the image by a 3x3 window.

3.23.2 Function Documentation

vx_node vxGaussian3x3Node (vx_graph *graph*, vx_image *input*, vx_image *output*)

[Graph] Creates a Gaussian Filter Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuGaussian3x3 (vx_context *context*, vx_image *input*, vx_image *output*)

[Immediate] Computes a gaussian filter on the image by a 3x3 window.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.24 Harris Corners

3.24.1 Detailed Description

Computes the Harris Corners of an image.

The Harris Corners are computed with several parameters

$$I = \text{input image} \tag{3.14}$$
$$T_c = \text{corner strength threshold} \tag{3.15}$$
$$r = \text{euclidean radius} \tag{3.16}$$
$$k = \text{sensitivity threshold} \tag{3.17}$$
$$w = \text{window size} \tag{3.18}$$
$$b = \text{block size} \tag{3.19}$$
$$\tag{3.20}$$

The computation to find the corner values or scores can be summarized as:

$$G_x = Sobel_x(w, I) \tag{3.21}$$
$$G_y = Sobel_y(w, I) \tag{3.22}$$
$$A = window_{G_{x,y}}(x-b/2, y-b/2, x+b/2, y+b/2) \tag{3.23}$$
$$trace(A) = \sum^{A} G_x^2 + \sum^{A} G_y^2 \tag{3.24}$$
$$det(A) = \sum^{A} G_x^2 \sum^{A} G_y^2 - \left(\sum^{A} (G_x G_y) \right)^2 \tag{3.25}$$
$$M_c(x,y) = det(A) - k * trace(A)^2 \tag{3.26}$$
$$V_c(x,y) = \begin{cases} M_c(x,y) \text{ if } M_c(x,y) > T_c \\ 0 \text{ otherwise} \end{cases} \tag{3.27}$$

where V_c is the thresholded corner value.

The normalized Sobel kernels used for the gradient computation shall be as shown below:

- For gradient size 3:

$$\mathbf{Sobel}_x(Normalized) = \frac{1}{4*255*b} * \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}$$

$$\mathbf{Sobel}_y(Normalized) = \frac{1}{4*255*b} * transpose(sobel_x) = \frac{1}{4*255*b} * \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix}$$

- For gradient size 5:

$$\mathbf{Sobel}_x(Normalized) = \frac{1}{16*255*b} * \begin{vmatrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 4 \\ -1 & -2 & 0 & 2 & 1 \end{vmatrix}$$

$$\mathbf{Sobel}_y(Normalized) = \frac{1}{16*255*b} * transpose(sobel_x)$$

- For gradient size 7:

$$\mathbf{Sobel}_x(Normalized) = \frac{1}{64*255*b} * \begin{vmatrix} -1 & -4 & -5 & 0 & 5 & 4 & 1 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -15 & -60 & -75 & 0 & 75 & 60 & 15 \\ -20 & -80 & -100 & 0 & 100 & 80 & 20 \\ -15 & -60 & -75 & 0 & 75 & 60 & 15 \\ -6 & -24 & -30 & 0 & 30 & 24 & 6 \\ -1 & -4 & -5 & 0 & 5 & 4 & 1 \end{vmatrix}$$

$$\mathbf{Sobel}_y(Normalized) = \frac{1}{64 * 255 * b} * transpose(sobel_x)$$

V_c is then non-maximally suppressed using the following algorithm:

- Filter the features using the non-maximum suppression algorithm defined for vxFastCornersNode.
- Create an array of features sorted by V_c in descending order: $V_c(j) > V_c(j+1)$.
- Initialize an empty feature set $F = \{\}$
- For each feature j in the sorted array, while $V_c(j) > T_c$:
  - If there is no feature i in F such that the Euclidean distance between pixels i and j is less than r, add the feature j to the feature set F.

An implementation shall support all values of Euclidean distance r that satisfy:

```
0 <= max_dist <= 30
```

The feature set F is returned as a `vx_array` of `vx_keypoint_t` structs.

Functions

- vx_node vxHarrisCornersNode (vx_graph graph, vx_image input, vx_scalar strength_thresh, vx_scalar min↩ _distance, vx_scalar sensitivity, vx_int32 gradient_size, vx_int32 block_size, vx_array corners, vx_scalar num_corners)

  [Graph] Creates a Harris Corners Node.
- vx_status vxuHarrisCorners (vx_context context, vx_image input, vx_scalar strength_thresh, vx_scalar min↩ _distance, vx_scalar sensitivity, vx_int32 gradient_size, vx_int32 block_size, vx_array corners, vx_scalar num_corners)

  [Immediate] Computes the Harris Corners over an image and produces the array of scored points.

3.24.2 Function Documentation

vx_node vxHarrisCornersNode (vx_graph *graph,* vx_image *input,* vx_scalar *strength_thresh,* vx_scalar *min_distance,* vx_scalar *sensitivity,* vx_int32 *gradient_size,* vx_int32 *block_size,* vx_array *corners,* vx_scalar *num_corners*)

[Graph] Creates a Harris Corners Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*strength_thresh*	The VX_TYPE_FLOAT32 minimum threshold with which to eliminate Harris Corner scores (computed using the normalized Sobel kernel).
in	*min_distance*	The VX_TYPE_FLOAT32 radial Euclidean distance for non-maximum suppression.
in	*sensitivity*	The VX_TYPE_FLOAT32 scalar sensitivity threshold k from the Harris-↩ Stephens equation.
in	*gradient_size*	The gradient window size to use on the input. The implementation must support at least 3, 5, and 7.
in	*block_size*	The block window size used to compute the Harris Corner score. The implementation must support at least 3, 5, and 7.
out	*corners*	The array of VX_TYPE_KEYPOINT objects.
out	*num_corners*	The total number of detected corners in image (optional).

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuHarrisCorners (vx_context *context*, vx_image *input*, vx_scalar *strength_thresh*, vx_scalar *min_distance*, vx_scalar *sensitivity*, vx_int32 *gradient_size*, vx_int32 *block_size*, vx_array *corners*, vx_scalar *num_corners*)

[Immediate] Computes the Harris Corners over an image and produces the array of scored points.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*strength_thresh*	The VX_TYPE_FLOAT32 minimum threshold which to eliminate Harris Corner scores (computed using the normalized Sobel kernel).
in	*min_distance*	The VX_TYPE_FLOAT32 radial Euclidean distance for non-maximum suppression.
in	*sensitivity*	The VX_TYPE_FLOAT32 scalar sensitivity threshold k from the Harris-Stephens equation.
in	*gradient_size*	The gradient window size to use on the input. The implementation must support at least 3, 5, and 7.
in	*block_size*	The block window size used to compute the harris corner score. The implementation must support at least 3, 5, and 7.
out	*corners*	The array of VX_TYPE_KEYPOINT structs.
out	*num_corners*	The total number of detected corners in image (optional).

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.25 Histogram

3.25.1 Detailed Description

Generates a distribution from an image.

This kernel counts the number of occurrences of each pixel value within the window size of a pre-calculated number of bins.

Functions

- vx_node vxHistogramNode (vx_graph graph, vx_image input, vx_distribution distribution)

  [Graph] Creates a Histogram node.
- vx_status vxuHistogram (vx_context context, vx_image input, vx_distribution distribution)

  [Immediate] Generates a distribution from an image.

3.25.2 Function Documentation

vx_node vxHistogramNode (vx_graph *graph,* vx_image *input,* vx_distribution *distribution*)

[Graph] Creates a Histogram node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8.
out	*distribution*	The output distribution.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuHistogram (vx_context *context,* vx_image *input,* vx_distribution *distribution*)

[Immediate] Generates a distribution from an image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8
out	*distribution*	The output distribution.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.26 Gaussian Image Pyramid

3.26.1 Detailed Description

Computes a Gaussian Image Pyramid from an input image.

This vision function creates the Gaussian image pyramid from the input image using the particular 5x5 Gaussian Kernel:

$$\mathbf{G} = \frac{1}{256} * \begin{vmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{vmatrix}$$

on each level of the pyramid then scales the image to the next level using VX_INTERPOLATION_TYPE_NEAREST_NEIGHBOR. Level 0 shall always have the same resolution as the input image. For the Gaussian pyramid, level 0 shall be the same as the input image. The pyramids must be configured with one of the following level scaling:

- VX_SCALE_PYRAMID_HALF
- VX_SCALE_PYRAMID_ORB

Functions

- vx_node vxGaussianPyramidNode (vx_graph graph, vx_image input, vx_pyramid gaussian)

  [Graph] Creates a node for a Gaussian Image Pyramid.
- vx_status vxuGaussianPyramid (vx_context context, vx_image input, vx_pyramid gaussian)

  [Immediate] Computes a Gaussian pyramid from an input image.

3.26.2 Function Documentation

vx_node vxGaussianPyramidNode (vx_graph *graph*, vx_image *input*, vx_pyramid *gaussian*)

[Graph] Creates a node for a Gaussian Image Pyramid.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*gaussian*	The Gaussian pyramid with VX_DF_IMAGE_U8 to construct.

See also

Object: Pyramid

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuGaussianPyramid (vx_context *context*, vx_image *input*, vx_pyramid *gaussian*)

[Immediate] Computes a Gaussian pyramid from an input image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8
out	*gaussian*	The Gaussian pyramid with VX_DF_IMAGE_U8 to construct.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.27 Integral Image

3.27.1 Detailed Description

Computes the integral image of the input.

Each output pixel is the sum of the corresponding input pixel and all other pixels above and to the left of it.

$$dst(x,y) = sum(x,y)$$

where, for x>=0 and y>=0

$$sum(x,y) = src(x,y) + sum(x-1,y) + sum(x,y-1) - sum(x-1,y-1)$$

otherwise,

$$sum(x,y) = 0$$

The overflow policy used is VX_CONVERT_POLICY_WRAP.

Functions

- vx_node vxIntegralImageNode (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates an Integral Image Node.
- vx_status vxuIntegralImage (vx_context context, vx_image input, vx_image output)

  [Immediate] Computes the integral image of the input.

3.27.2 Function Documentation

vx_node vxIntegralImageNode (vx_graph *graph*, vx_image *input*, vx_image *output*)

[Graph] Creates an Integral Image Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U32 format.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuIntegralImage (vx_context *context*, vx_image *input*, vx_image *output*)

[Immediate] Computes the integral image of the input.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U32 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.28 Magnitude

3.28.1 Detailed Description

Implements the Gradient Magnitude Computation Kernel.

This kernel takes two gradients in VX_DF_IMAGE_S16 format and computes the VX_DF_IMAGE_S16 normalized magnitude. Magnitude is computed as:

$$mag(x,y) = \sqrt{grad_x(x,y)^2 + grad_y(x,y)^2}$$

The conceptual definition describing the overflow is given as: uint16 z = uint16(sqrt(double(uint32(int32(x) $*$ int32(x)) + uint32(int32(y) $*$ int32(y))))); int16 r = z $>$ 32767 ? 32767 : z;

Functions

- vx_node vxMagnitudeNode (vx_graph graph, vx_image grad_x, vx_image grad_y, vx_image mag)

  [Graph] Create a Magnitude node.
- vx_status vxuMagnitude (vx_context context, vx_image grad_x, vx_image grad_y, vx_image output)

  [Immediate] Invokes an immediate Magnitude.

3.28.2 Function Documentation

vx_node vxMagnitudeNode (vx_graph *graph*, vx_image *grad_x*, vx_image *grad_y*, vx_image *mag*)

[Graph] Create a Magnitude node.

Parameters

in	*graph*	The reference to the graph.
in	*grad_x*	The input x image. This must be in VX_DF_IMAGE_S16 format.
in	*grad_y*	The input y image. This must be in VX_DF_IMAGE_S16 format.
out	*mag*	The magnitude image. This is in VX_DF_IMAGE_S16 format.

See also

VX_KERNEL_MAGNITUDE

Returns

vx_node.

Return values

0	Node could not be created.
$*$	Node handle.

vx_status vxuMagnitude (vx_context *context*, vx_image *grad_x*, vx_image *grad_y*, vx_image *output*)

[Immediate] Invokes an immediate Magnitude.

Parameters

in	*context*	The reference to the overall context.
in	*grad_x*	The input x image. This must be in VX_DF_IMAGE_S16 format.
in	*grad_y*	The input y image. This must be in VX_DF_IMAGE_S16 format.
out	*output*	The magnitude image. This will be in VX_DF_IMAGE_S16 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.29 Mean and Standard Deviation

3.29.1 Detailed Description

Computes the mean pixel value and the standard deviation of the pixels in the input image (which has a dimension width and height).

The mean value is computed as:

$$\mu = \frac{\left(\sum_{y=0}^{h}\sum_{x=0}^{w} src(x,y)\right)}{(width * height)}$$

The standard deviation is computed as:

$$\sigma = \sqrt{\frac{\left(\sum_{y=0}^{h}\sum_{x=0}^{w} (\mu - src(x,y))^2\right)}{(width * height)}}$$

Functions

- vx_node vxMeanStdDevNode (vx_graph graph, vx_image input, vx_scalar mean, vx_scalar stddev)

  [Graph] Creates a mean value and standard deviation node.
- vx_status vxuMeanStdDev (vx_context context, vx_image input, vx_float32 ∗mean, vx_float32 ∗stddev)

  [Immediate] Computes the mean value and standard deviation.

3.29.2 Function Documentation

vx_node vxMeanStdDevNode (vx_graph *graph,* vx_image *input,* vx_scalar *mean,* vx_scalar *stddev*)

[Graph] Creates a mean value and standard deviation node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image. VX_DF_IMAGE_U8 is supported.
out	*mean*	The VX_TYPE_FLOAT32 average pixel value.
out	*stddev*	The VX_TYPE_FLOAT32 standard deviation of the pixel values.

Returns

vx_node.

Return values

0	Node could not be created.
∗	Node handle.

vx_status vxuMeanStdDev (vx_context *context,* vx_image *input,* vx_float32 ∗ *mean,* vx_float32 ∗ *stddev*)

[Immediate] Computes the mean value and standard deviation.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image. VX_DF_IMAGE_U8 is supported.
out	*mean*	The average pixel value.
out	*stddev*	The standard deviation of the pixel values.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.30 Median Filter

3.30.1 Detailed Description

Computes a median pixel value over a window of the input image.
The median is the middle value over an odd-numbered, sorted range of values.

Functions

- vx_node vxMedian3x3Node (vx_graph graph, vx_image input, vx_image output)

  [Graph] Creates a Median Image Node.
- vx_status vxuMedian3x3 (vx_context context, vx_image input, vx_image output)

  [Immediate] Computes a median filter on the image by a 3x3 window.

3.30.2 Function Documentation

vx_node vxMedian3x3Node (vx_graph *graph*, vx_image *input*, vx_image *output*)

[Graph] Creates a Median Image Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuMedian3x3 (vx_context *context*, vx_image *input*, vx_image *output*)

[Immediate] Computes a median filter on the image by a 3x3 window.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output*	The output image in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.31 Min, Max Location

3.31.1 Detailed Description

Finds the minimum and maximum values in an image and a location for each.

If the input image has several minimums/maximums, the kernel returns all of them.

$$minVal = \min_{\substack{0 \le x' \le width \\ 0 \le y' \le height}} src(x', y')$$

$$maxVal = \max_{\substack{0 \le x' \le width \\ 0 \le y' \le height}} src(x', y')$$

Functions

- vx_node vxMinMaxLocNode (vx_graph graph, vx_image input, vx_scalar minVal, vx_scalar maxVal, vx_array minLoc, vx_array maxLoc, vx_scalar minCount, vx_scalar maxCount)

  [Graph] Creates a min,max,loc node.
- vx_status vxuMinMaxLoc (vx_context context, vx_image input, vx_scalar minVal, vx_scalar maxVal, vx_array minLoc, vx_array maxLoc, vx_scalar minCount, vx_scalar maxCount)

  [Immediate] Computes the minimum and maximum values of the image.

3.31.2 Function Documentation

vx_node vxMinMaxLocNode (vx_graph *graph,* vx_image *input,* vx_scalar *minVal,* vx_scalar *maxVal,* vx_array *minLoc,* vx_array *maxLoc,* vx_scalar *minCount,* vx_scalar *maxCount*)

[Graph] Creates a min,max,loc node.

Parameters

in	*graph*	The reference to create the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.
out	*minVal*	The minimum value in the image, which corresponds to the type of the input.
out	*maxVal*	The maximum value in the image, which corresponds to the type of the input.
out	*minLoc*	The minimum VX_TYPE_COORDINATES2D locations (optional). If the input image has several minimums, the kernel will return up to the capacity of the array.
out	*maxLoc*	The maximum VX_TYPE_COORDINATES2D locations (optional). If the input image has several maximums, the kernel will return up to the capacity of the array.
out	*minCount*	The total number of detected minimums in image (optional). Use a VX_TYPE_UINT32 scalar.
out	*maxCount*	The total number of detected maximums in image (optional). Use a VX_TYPE_UINT32 scalar.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuMinMaxLoc (vx_context *context,* vx_image *input,* vx_scalar *minVal,* vx_scalar *maxVal,* vx_array *minLoc,* vx_array *maxLoc,* vx_scalar *minCount,* vx_scalar *maxCount*)

[Immediate] Computes the minimum and maximum values of the image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.
out	*minVal*	The minimum value in the image.
out	*maxVal*	The maximum value in the image.
out	*minLoc*	The minimum locations (optional). If the input image has several minimums, the kernel will return all of them).
out	*maxLoc*	The maximum locations (optional). If the input image has several maximums, the kernel will return all of them).
out	*minCount*	The total number of detected minimums in image (optional).
out	*maxCount*	The total number of detected maximums in image (optional).

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.32 Optical Flow Pyramid (LK)

3.32.1 Detailed Description

Computes the optical flow using the Lucas-Kanade method between two pyramid images.

The function is an implementation of the algorithm described in [1]. The function inputs are two vx_pyramid objects, old and new, along with a vx_array of vx_keypoint_t structs to track from the old vx_pyramid. The function outputs a vx_array of vx_keypoint_t structs that were tracked from the old vx_pyramid to the new vx_pyramid. Each element in the vx_array of vx_keypoint_t structs in the new array may be valid or not. The implementation shall return the same number of vx_keypoint_t structs in the new vx_array that were in the older vx_array.

In more detail: The Lucas-Kanade method finds the affine motion vector V for each point in the old image tracking points array, using the following equation:

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i {I_x}^2 & \sum_i I_x * I_y \\ \sum_i I_x * I_y & \sum_i {I_y}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x * I_t \\ -\sum_i I_y * I_t \end{bmatrix}$$

Where I_x and I_y are obtained using the Scharr gradients on the input image:

$$G_x = \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix}$$

$$G_y = \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

I_t is obtained by a simple difference between the same pixel in both images. i is defined as the adjacent pixels to the point $p(x,y)$ under consideration. With a given window size of M, i is M^2 points. The pixel $p(x,y)$ is centered in the window. In practice, to get an accurate solution, it is necessary to iterate multiple times on this scheme (in a Newton-Raphson fashion) until:

- the residual of the affine motion vector is smaller than a threshold
- And/or maximum number of iteration achieved. Each iteration, the estimation of the previous iteration is used by changing I_t to be the difference between the old image and the pixel with the estimated coordinates in the new image. Each iteration the function checks if the pixel to track was lost. The criteria for lost tracking is that the matrix above is invertible. (The determinant of the matrix is less than a threshold : 10^{-7} .) Or the minimum eigenvalue of the matrix is smaller then a threshold (10^{-4}). Also lost tracking happens when the point tracked coordinate is outside the image coordinates. When vx_true_e is given as the input to use_initial_estimates, the algorithm starts by calculating I_t as the difference between the old image and the pixel with the initial estimated coordinates in the new image. The input vx_array of vx_keypoint_t structs with tracking_status set to zero (lost) are copied to the new vx_array.

Clients are responsible for editing the output vx_array of vx_keypoint_t structs array before applying it as the input vx_array of vx_keypoint_t structs for the next frame. For example, vx_keypoint_t structs with tracking_status set to zero may be removed by a client for efficiency.

This function changes just the *x*, *y*, and *tracking_status* members of the vx_keypoint_t structure and behaves as if it copied the rest from the old tracking vx_keypoint_t to new image vx_keypoint_t.

Functions

- vx_node vxOpticalFlowPyrLKNode (vx_graph graph, vx_pyramid old_images, vx_pyramid new_images, vx_array old_points, vx_array new_points_estimates, vx_array new_points, vx_enum termination, vx_scalar epsilon, vx_scalar num_iterations, vx_scalar use_initial_estimate, vx_size window_dimension)

  [Graph] Creates a Lucas Kanade Tracking Node.
- vx_status vxuOpticalFlowPyrLK (vx_context context, vx_pyramid old_images, vx_pyramid new_images, vx_array old_points, vx_array new_points_estimates, vx_array new_points, vx_enum termination, vx_scalar epsilon, vx_scalar num_iterations, vx_scalar use_initial_estimate, vx_size window_dimension)

  [Immediate] Computes an optical flow on two images.

3.32.2 Function Documentation

vx_node vxOpticalFlowPyrLKNode (vx_graph *graph,* vx_pyramid *old_images,* vx_pyramid *new_images,* vx_array *old_points,* vx_array *new_points_estimates,* vx_array *new_points,* vx_enum *termination,* vx_scalar *epsilon,* vx_scalar *num_iterations,* vx_scalar *use_initial_estimate,* vx_size *window_dimension*)

[Graph] Creates a Lucas Kanade Tracking Node.

Parameters

in	*graph*	The reference to the graph.
in	*old_images*	Input of first (old) image pyramid in VX_DF_IMAGE_U8.
in	*new_images*	Input of destination (new) image pyramid VX_DF_IMAGE_U8.
in	*old_points*	An array of key points in a vx_array of VX_TYPE_KEYPOINT; those key points are defined at the *old_images* high resolution pyramid.
in	*new_points_estimates*	An array of estimation on what is the output key points in a vx_array of VX_TYPE_KEYPOINT; those keypoints are defined at the *new_images* high resolution pyramid.
out	*new_points*	An output array of key points in a vx_array of VX_TYPE_KEYPOINT; those key points are defined at the *new_images* high resolution pyramid.
in	*termination*	The termination can be VX_TERM_CRITERIA_ITERATIONS or VX_TERM_CRITERIA_EPSILON or VX_TERM_CRITERIA_BOTH.
in	*epsilon*	The vx_float32 error for terminating the algorithm.
in	*num_iterations*	The number of iterations. Use a VX_TYPE_UINT32 scalar.
in	*use_initial_estimate*	Use a VX_TYPE_BOOL scalar.
in	*window_dimension*	The size of the window on which to perform the algorithm. See VX_CONTEXT_ATTRIBUTE_OPTICAL_FLOW_WINDOW_MAXIMUM_DIMENSION

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuOpticalFlowPyrLK (vx_context *context*, vx_pyramid *old_images*, vx_pyramid *new_images*, vx_array *old_points*, vx_array *new_points_estimates*, vx_array *new_points*, vx_enum *termination*, vx_scalar *epsilon*, vx_scalar *num_iterations*, vx_scalar *use_initial_estimate*, vx_size *window_dimension*)

[Immediate] Computes an optical flow on two images.

Parameters

in	*context*	The reference to the overall context.
in	*old_images*	Input of first (old) image pyramid
in	*new_images*	Input of destination (new) image pyramid
in	*old_points*	an array of key points in a vx_array of VX_TYPE_KEYPOINT those key points are defined at the old_images high resolution pyramid
in	*new_points_estimates*	an array of estimation on what is the output key points in a vx_array of VX_TYPE_KEYPOINT those keypoints are defined at the new_images high resolution pyramid
out	*new_points*	an output array of key points in a vx_array of VX_TYPE_KEYPOINT those key points are defined at the new_images high resolution pyramid
in	*termination*	termination can be VX_TERM_CRITERIA_ITERATIONS or VX_TERM_CRITERIA_EPSILON or VX_TERM_CRITERIA_BOTH
in	*epsilon*	is the vx_float32 error for terminating the algorithm
in	*num_iterations*	is the number of iterations
in	*use_initial_estimate*	Can be set to either vx_false_e or vx_true_e.

in	*window_↩ dimension*	The size of the window on which to perform the algorithm. See VX_CONTE↩ XT_ATTRIBUTE_OPTICAL_FLOW_WINDOW_MAXIMUM_DIMENSION

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.33 Phase

3.33.1 Detailed Description

Implements the Gradient Phase Computation Kernel.

This kernel takes two gradients in VX_DF_IMAGE_S16 format and computes the angles for each pixel and stores this in a VX_DF_IMAGE_U8 image.

$$\phi = \tan^{-1} \frac{grad_y(x,y)}{grad_x(x,y)}$$

Where ϕ is then translated to $0 \leq \phi < 2\pi$. Each ϕ value is then mapped to the range 0 to 255 inclusive.

Functions

- vx_node vxPhaseNode (vx_graph graph, vx_image grad_x, vx_image grad_y, vx_image orientation)

  [Graph] Creates a Phase node.
- vx_status vxuPhase (vx_context context, vx_image grad_x, vx_image grad_y, vx_image output)

  [Immediate] Invokes an immediate Phase.

3.33.2 Function Documentation

vx_node vxPhaseNode (vx_graph *graph*, vx_image *grad_x*, vx_image *grad_y*, vx_image *orientation*)

[Graph] Creates a Phase node.

Parameters

in	*graph*	The reference to the graph.
in	*grad_x*	The input x image. This must be in VX_DF_IMAGE_S16 format.
in	*grad_y*	The input y image. This must be in VX_DF_IMAGE_S16 format.
out	*orientation*	The phase image. This is in VX_DF_IMAGE_U8 format.

See also

VX_KERNEL_PHASE

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuPhase (vx_context *context*, vx_image *grad_x*, vx_image *grad_y*, vx_image *output*)

[Immediate] Invokes an immediate Phase.

Parameters

in	*context*	The reference to the overall context.
in	*grad_x*	The input x image. This must be in VX_DF_IMAGE_S16 format.
in	*grad_y*	The input y image. This must be in VX_DF_IMAGE_S16 format.
out	*output*	The phase image. This will be in VX_DF_IMAGE_U8 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.34 Pixel-wise Multiplication

3.34.1 Detailed Description

Performs element-wise multiplication between two images and a scalar value.

Pixel-wise multiplication is performed between the pixel values in two VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 images and a scalar floating-point number *scale*. The output image can be VX_DF_IMAGE_U8 only if both source images are VX_DF_IMAGE_U8 and the output image is explicitly set to VX_DF_IMAGE_U8. It is otherwise VX_DF_IMAGE_S16. If one of the input images is of type VX_DF_IMAGE_S16, all values are converted to VX_DF_IMAGE_S16.

The scale with a value of $1/2^n$, where n is an integer and $0 \leq n \leq 15$, and 1/255 (0x1.010102p-8 C99 float hex) must be supported. The support for other values of scale is not prohibited. Furthermore, for scale with a value of 1/255 the rounding policy of VX_ROUND_POLICY_TO_NEAREST_EVEN must be supported whereas for the scale with value of $1/2^n$ the rounding policy of VX_ROUND_POLICY_TO_ZERO must be supported. The support of other rounding modes for any values of scale is not prohibited.

The rounding policy VX_ROUND_POLICY_TO_ZERO for this function is defined as:

$$reference(x,y,scale) = truncate(((int32_t)in1(x,y)) * ((int32_t)in2(x,y)) * (double)scale)$$

The rounding policy VX_ROUND_POLICY_TO_NEAREST_EVEN for this function is defined as:

$$reference(x,y,scale) = round_to_nearest_even(((int32_t)in1(x,y)) * ((int32_t)in2(x,y)) * (double)scale)$$

The overflow handling is controlled by an overflow-policy parameter. For each pixel value in the two input images:

$$out(x,y) = in_1(x,y)in_2(x,y)scale$$

Functions

- vx_node vxMultiplyNode (vx_graph graph, vx_image in1, vx_image in2, vx_scalar scale, vx_enum overflow_policy, vx_enum rounding_policy, vx_image out)

  [Graph] Creates an pixelwise-multiplication node.

- vx_status vxuMultiply (vx_context context, vx_image in1, vx_image in2, vx_float32 scale, vx_enum overflow_policy, vx_enum rounding_policy, vx_image out)

  [Immediate] Performs elementwise multiplications on pixel values in the input images and a scale.

3.34.2 Function Documentation

vx_node vxMultiplyNode (vx_graph *graph*, vx_image *in1*, vx_image *in2*, vx_scalar *scale*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_image *out*)

[Graph] Creates an pixelwise-multiplication node.

Parameters

in	*graph*	The reference to the graph.
in	*in1*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16.
in	*in2*	An input image, VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16.
in	*scale*	A non-negative VX_TYPE_FLOAT32 multiplied to each product before overflow handling.
in	*overflow_policy*	A VX_TYPE_ENUM of the vx_convert_policy_e enumeration.
in	*rounding_policy*	A VX_TYPE_ENUM of the vx_round_policy_e enumeration.
out	*out*	The output image, a VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 image.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuMultiply (vx_context *context*, vx_image *in1*, vx_image *in2*, vx_float32 *scale*, vx_enum *overflow_policy*, vx_enum *rounding_policy*, vx_image *out*)

[Immediate] Performs elementwise multiplications on pixel values in the input images and a scale.

Parameters

in	*context*	The reference to the overall context.
in	*in1*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image.
in	*in2*	A VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 input image.
in	*scale*	The scale value.
in	*overflow_policy*	A vx_convert_policy_e enumeration.
in	*rounding_policy*	A vx_round_policy_e enumeration.
out	*out*	The output image in VX_DF_IMAGE_U8 or VX_DF_IMAGE_S16 format.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.35 Remap

3.35.1 Detailed Description

Maps output pixels in an image from input pixels in an image.

Remap takes a remap table object vx_remap to map a set of output pixels back to source input pixels. A remap is typically defined as:

$$output(x1, y1) = input(map_x(x0, y0), map_y(x0, y0))$$

However, the mapping functions are contained in the vx_remap object.

Functions

- vx_node vxRemapNode (vx_graph graph, vx_image input, vx_remap table, vx_enum policy, vx_image output)

  [Graph] Creates a Remap Node.
- vx_status vxuRemap (vx_context context, vx_image input, vx_remap table, vx_enum policy, vx_image output)

  [Immediate] Remaps an output image from an input image.

3.35.2 Function Documentation

vx_node vxRemapNode (vx_graph *graph,* vx_image *input,* vx_remap *table,* vx_enum *policy,* vx_image *output*)

[Graph] Creates a Remap Node.

Parameters

in	*graph*	The reference to the graph that will contain the node.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*table*	The remap table object.
in	*policy*	An interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_TYPE_AREA is not supported.
out	*output*	The output VX_DF_IMAGE_U8 image.

Note

Only VX_NODE_ATTRIBUTE_BORDER_MODE value VX_BORDER_MODE_UNDEFINED or VX_BORDER_MODE_CONSTANT is supported.

Returns

A vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuRemap (vx_context *context,* vx_image *input,* vx_remap *table,* vx_enum *policy,* vx_image *output*)

[Immediate] Remaps an output image from an input image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.

in	*table*	The remap table object.
in	*policy*	The interpolation policy from vx_interpolation_type_e. VX_INTERPOLATIO↩ N_TYPE_AREA is not supported.
out	*output*	The output VX_DF_IMAGE_U8 image.

Returns

A vx_status_e enumeration.

3.36 Scale Image

3.36.1 Detailed Description

Implements the Image Resizing Kernel.

Performs a Gaussian Blur on an image then half-scales it.

This kernel resizes an image from the source to the destination dimensions. The only format supported is VX_DF_IMAGE_U8. The supported interpolation types are currently:

- VX_INTERPOLATION_TYPE_NEAREST_NEIGHBOR
- VX_INTERPOLATION_TYPE_AREA
- VX_INTERPOLATION_TYPE_BILINEAR

The sample positions used to determine output pixel values are generated by scaling the outside edges of the source image pixels to the outside edges of the destination image pixels. As described in the documentation for vx_interpolation_type_e, samples are taken at pixel centers. This means that, unless the scale is 1:1, the sample position for the top left destination pixel typically does not fall exactly on the top left source pixel but will be generated by interpolation.

That is, the sample positions corresponding in source and destination are defined by the following equations:

$$x_{input} = \left((x_{output} + 0.5) * \frac{width_{input}}{width_{output}} \right) - 0.5$$

$$y_{input} = \left((y_{output} + 0.5) * \frac{height_{input}}{height_{output}} \right) - 0.5$$

$$x_{output} = \left((x_{input} + 0.5) * \frac{width_{output}}{width_{input}} \right) - 0.5$$

$$y_{output} = \left((y_{input} + 0.5) * \frac{height_{output}}{height_{input}} \right) - 0.5$$

- For VX_INTERPOLATION_TYPE_NEAREST_NEIGHBOR, the output value is that of the pixel whose centre is closest to the sample point.
- For VX_INTERPOLATION_TYPE_BILINEAR, the output value is formed by a weighted average of the nearest source pixels to the sample point. That is:

$$x_{lower} = \lfloor x_{input} \rfloor$$

$$y_{lower} = \lfloor y_{input} \rfloor$$

$$s = x_{input} - x_{lower}$$

$$t = y_{input} - y_{lower}$$

$$output(x_{input}, y_{input}) = (1-s)(1-t) * input(x_{lower}, y_{lower}) + s(1-t) * input(x_{lower}+1, y_{lower})$$
$$+(1-s)t * input(x_{lower}, y_{lower}+1) + s * t * input(x_{lower}+1, y_{lower}+1)$$

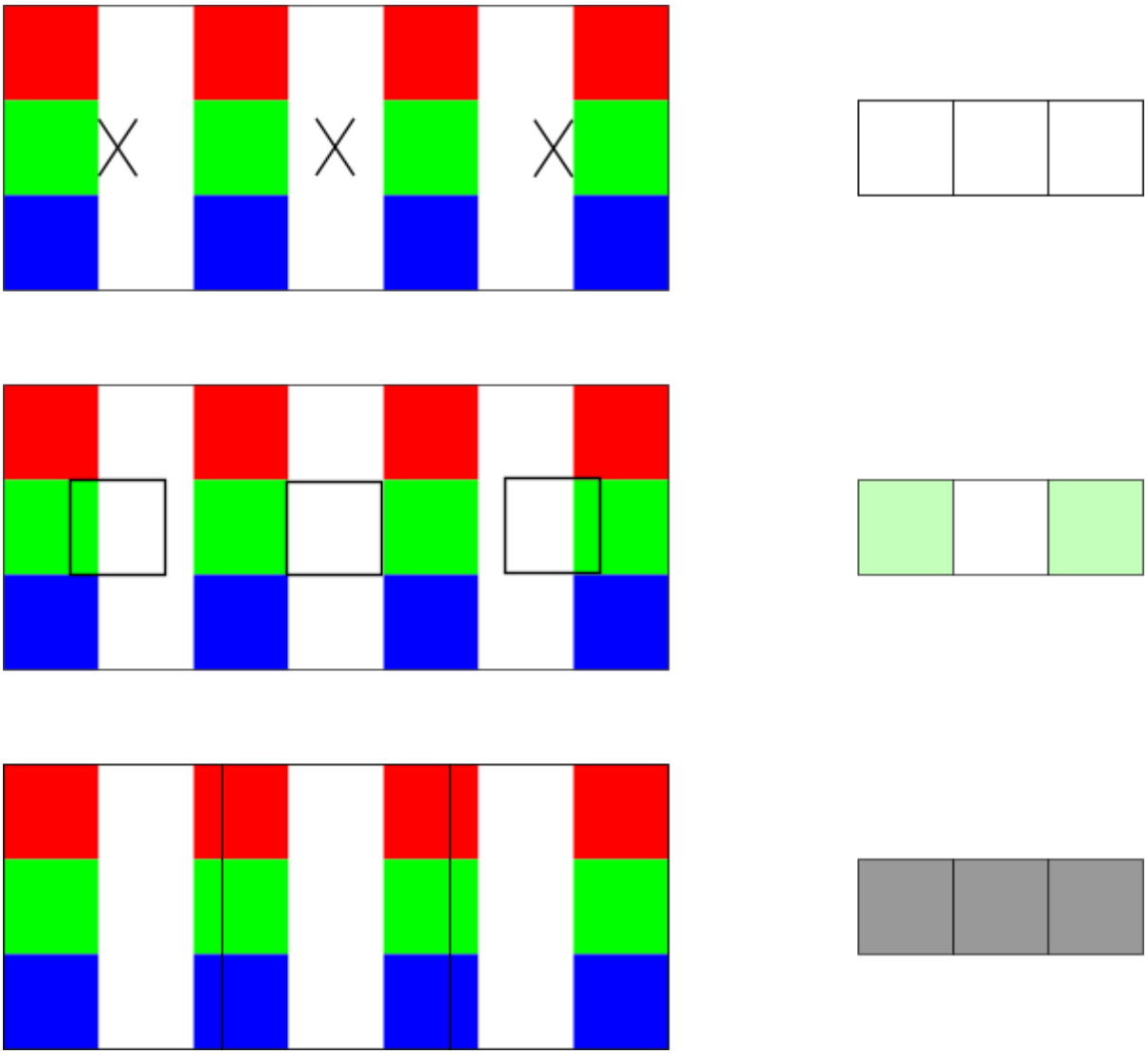
- For VX_INTERPOLATION_TYPE_AREA, the implementation is expected to generate each output pixel by sampling all the source pixels that are at least partly covered by the area bounded by:

$$\left(x_{output} * \frac{width_{input}}{width_{output}} \right) - 0.5, \left(y_{output} * \frac{height_{input}}{height_{output}} \right) - 0.5$$

and

$$\left((x_{output} + 1) * \frac{width_{input}}{width_{output}} \right) - 0.5, \left((y_{output} + 1) * \frac{height_{input}}{height_{output}} \right) - 0.5$$

The details of this sampling method are implementation-defined. The implementation should perform enough sampling to avoid aliasing, but there is no requirement that the sample areas for adjacent output pixels be disjoint, nor that the pixels be weighted evenly.



The above diagram shows three sampling methods used to shrink a 7x3 image to 3x1.

The topmost image pair shows nearest-neighbor sampling, with crosses on the left image marking the sample positions in the source that are used to generate the output image on the right. As the pixel centre closest to the sample position is white in all cases, the resulting 3x1 image is white.

The middle image pair shows bilinear sampling, with black squares on the left image showing the region in the source being sampled to generate each pixel on the destination image on the right. This sample area is always the size of an input pixel. The outer destination pixels partly sample from the outermost green pixels, so their resulting value is a weighted average of white and green.

The bottom image pair shows area sampling. The black rectangles in the source image on the left show the bounds of the projection of the destination pixels onto the source. The destination pixels on the right are formed by averaging at least those source pixels whose areas are wholly or partly contained within those rectangles. The manner of this averaging is implementation-defined; the example shown here weights the contribution of each source pixel by the amount of that pixel's area contained within the black rectangle.

Functions

- vx_node vxHalfScaleGaussianNode (vx_graph graph, vx_image input, vx_image output, vx_int32 kernel_↩ size)

  [Graph] Performs a Gaussian Blur on an image then half-scales it.

- vx_node vxScaleImageNode (vx_graph graph, vx_image src, vx_image dst, vx_enum type)

  [Graph] Creates a Scale Image Node.

- vx_status vxuHalfScaleGaussian (vx_context context, vx_image input, vx_image output, vx_int32 kernel_↩ size)

  [Immediate] Performs a Gaussian Blur on an image then half-scales it.
- vx_status vxuScaleImage (vx_context context, vx_image src, vx_image dst, vx_enum type)

  [Immediate] Scales an input image to an output image.

3.36.2 Function Documentation

vx_node vxScaleImageNode (vx_graph *graph*, vx_image *src*, vx_image *dst*, vx_enum *type*)

[Graph] Creates a Scale Image Node.

Parameters

in	*graph*	The reference to the graph.
in	*src*	The source image.
out	*dst*	The destination image.
in	*type*	The interpolation type to use.

See also

vx_interpolation_type_e.

Note

The destination image must have a defined size and format. Only VX_NODE_ATTRIBUTE_BORDER_MODE value VX_BORDER_MODE_UNDEFINED, VX_BORDER_MODE_REPLICATE or VX_BORDER_MODE_C↩ ONSTANT is supported.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_node vxHalfScaleGaussianNode (vx_graph *graph*, vx_image *input*, vx_image *output*, vx_int32 *kernel_size*)

[Graph] Performs a Gaussian Blur on an image then half-scales it.

The output image size is determined by:

$$W_{output} = \frac{W_{input} + 1}{2}, H_{output} = \frac{H_{input} + 1}{2}$$

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
out	*output*	The output VX_DF_IMAGE_U8 image.
in	*kernel_size*	The input size of the Gaussian filter. Supported values are 3 and 5.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuScaleImage (vx_context *context*, vx_image *src*, vx_image *dst*, vx_enum *type*)

[Immediate] Scales an input image to an output image.

Parameters

in	*context*	The reference to the overall context.
in	*src*	The source image.
out	*dst*	The destintation image.
in	*type*	The interpolation type.

See also

vx_interpolation_type_e.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

vx_status vxuHalfScaleGaussian (vx_context *context*, vx_image *input*, vx_image *output*, vx_int32 *kernel_size*)

[Immediate] Performs a Gaussian Blur on an image then half-scales it.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
out	*output*	The output VX_DF_IMAGE_U8 image.
in	*kernel_size*	The input size of the Gaussian filter. Supported values are 3 and 5.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.37 Sobel 3x3

3.37.1 Detailed Description

Implements the Sobel Image Filter Kernel.

This kernel produces two output planes (one can be omitted) in the x and y plane. The Sobel Operators G_x, G_y are defined as:

$$\mathbf{G}_x = \begin{vmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{vmatrix}, \mathbf{G}_y = \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{vmatrix}$$

Functions

- vx_node vxSobel3x3Node (vx_graph graph, vx_image input, vx_image output_x, vx_image output_y)

  [Graph] Creates a Sobel3x3 node.
- vx_status vxuSobel3x3 (vx_context context, vx_image input, vx_image output_x, vx_image output_y)

  [Immediate] Invokes an immediate Sobel 3x3.

3.37.2 Function Documentation

vx_node vxSobel3x3Node (vx_graph *graph*, vx_image *input*, vx_image *output_x*, vx_image *output_y*)

[Graph] Creates a Sobel3x3 node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output_x*	[optional] The output gradient in the x direction in VX_DF_IMAGE_S16.
out	*output_y*	[optional] The output gradient in the y direction in VX_DF_IMAGE_S16.

See also

VX_KERNEL_SOBEL_3x3

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuSobel3x3 (vx_context *context*, vx_image *input*, vx_image *output_x*, vx_image *output_y*)

[Immediate] Invokes an immediate Sobel 3x3.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8 format.
out	*output_x*	[optional] The output gradient in the x direction in VX_DF_IMAGE_S16.
out	*output_y*	[optional] The output gradient in the y direction in VX_DF_IMAGE_S16.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.38 TableLookup

3.38.1 Detailed Description

Implements the Table Lookup Image Kernel.

This kernel uses each pixel in an image to index into a LUT and put the indexed LUT value into the output image. The format supported is VX_DF_IMAGE_U8.

Functions

- vx_node vxTableLookupNode (vx_graph graph, vx_image input, vx_lut lut, vx_image output)

  [Graph] Creates a Table Lookup node.
- vx_status vxuTableLookup (vx_context context, vx_image input, vx_lut lut, vx_image output)

  [Immediate] Processes the image through the LUT.

3.38.2 Function Documentation

vx_node vxTableLookupNode (vx_graph *graph*, vx_image *input*, vx_lut *lut*, vx_image *output*)

[Graph] Creates a Table Lookup node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image in VX_DF_IMAGE_U8.
in	*lut*	The LUT which is of type VX_TYPE_UINT8.
out	*output*	The output image of type VX_DF_IMAGE_U8.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuTableLookup (vx_context *context*, vx_image *input*, vx_lut *lut*, vx_image *output*)

[Immediate] Processes the image through the LUT.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image in VX_DF_IMAGE_U8
in	*lut*	The LUT which is of type VX_TYPE_UINT8
out	*output*	The output image of type VX_DF_IMAGE_U8

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.39 Thresholding

3.39.1 Detailed Description

Thresholds an input image and produces an output Boolean image.

In VX_THRESHOLD_TYPE_BINARY, the output is determined by:

$$dst(x,y) = \begin{cases} 255 & \text{if } src(x,y) > threshold \\ 0 & \text{otherwise} \end{cases}$$

In VX_THRESHOLD_TYPE_RANGE, the output is determined by:

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > upper \\ 0 & \text{if } src(x,y) < lower \\ 255 & \text{otherwise} \end{cases}$$

Functions

- vx_node vxThresholdNode (vx_graph graph, vx_image input, vx_threshold thresh, vx_image output)

  [Graph] Creates a Threshold node.
- vx_status vxuThreshold (vx_context context, vx_image input, vx_threshold thresh, vx_image output)

  *[Immediate] Threshold's an input image and produces a VX_DF_IMAGE_U8 * boolean image.*

3.39.2 Function Documentation

vx_node vxThresholdNode (vx_graph *graph,* vx_image *input,* vx_threshold *thresh,* vx_image *output*)

[Graph] Creates a Threshold node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input image. VX_DF_IMAGE_U8 is supported.
in	*thresh*	The thresholding object that defines the parameters of the operation.
out	*output*	The output Boolean image. Values are either 0 or 255.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuThreshold (vx_context *context,* vx_image *input,* vx_threshold *thresh,* vx_image *output*)

[Immediate] Threshold's an input image and produces a VX_DF_IMAGE_U8 * boolean image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input image. VX_DF_IMAGE_U8 is supported.
in	*thresh*	The thresholding object that defines the parameters of the operation.
out	*output*	The output Boolean image. Values are either 0 or 255.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.40 Warp Affine

3.40.1 Detailed Description

Performs an affine transform on an image.

This kernel performs an affine transform with a 2x3 Matrix M with this method of pixel coordinate translation:

$$x0 = M_{1,1} * x + M_{1,2} * y + M_{1,3} \quad (3.28)$$

$$y0 = M_{2,1} * x + M_{2,2} * y + M_{2,3} \quad (3.29)$$

$$output(x,y) = input(x0,y0) \quad (3.30)$$

This translates into the C declaration:

```
// x0 = a x + b y + c;
// y0 = d x + e y + f;
vx_float32 mat[3][2] = {
    {a, d}, // 'x' coefficients
    {b, e}, // 'y' coefficients
    {c, f}, // 'offsets'
};
vx_matrix matrix = vxCreateMatrix(context,
  VX_TYPE_FLOAT32, 2, 3);
vxAccessMatrix(matrix, NULL);
vxCommitMatrix(matrix, mat);
```

Functions

- vx_status vxuWarpAffine (vx_context context, vx_image input, vx_matrix matrix, vx_enum type, vx_image output)

  [Immediate] Performs an Affine warp on an image.
- vx_node vxWarpAffineNode (vx_graph graph, vx_image input, vx_matrix matrix, vx_enum type, vx_image output)

  [Graph] Creates an Affine Warp Node.

3.40.2 Function Documentation

vx_node vxWarpAffineNode (vx_graph *graph*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*)

[Graph] Creates an Affine Warp Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*matrix*	The affine matrix. Must be 2x3 of type VX_TYPE_FLOAT32.
in	*type*	The interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_TYPE_AREA is not supported.
out	*output*	The output VX_DF_IMAGE_U8 image.

Note

Only VX_NODE_ATTRIBUTE_BORDER_MODE value VX_BORDER_MODE_UNDEFINED or VX_BORDER_MODE_CONSTANT is supported.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuWarpAffine (vx_context *context*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*)

[Immediate] Performs an Affine warp on an image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*matrix*	The affine matrix. Must be 2x3 of type VX_TYPE_FLOAT32.
in	*type*	The interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_TYPE_AREA is not supported.
out	*output*	The output VX_DF_IMAGE_U8 image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.41 Warp Perspective

3.41.1 Detailed Description

Performs a perspective transform on an image.

This kernel performs an perspective transform with a 3x3 Matrix M with this method of pixel coordinate translation:

$$x0 = M_{1,1} * x + M_{1,2} * y + M_{1,3} \quad (3.31)$$

$$y0 = M_{2,1} * x + M_{2,2} * y + M_{2,3} \quad (3.32)$$

$$z0 = M_{3,1} * x + M_{3,2} * y + M_{3,3} \quad (3.33)$$

$$output(x,y) = input(\frac{x0}{z0}, \frac{y0}{z0}) \quad (3.34)$$

This translates into the C declaration:

```
// x0 = a x + b y + c;
// y0 = d x + e y + f;
// z0 = g x + h y + i;
vx_float32 mat[3][3] = {
    {a, d, g}, // 'x' coefficients
    {b, e, h}, // 'y' coefficients
    {c, f, i}, // 'offsets'
};
vx_matrix matrix = vxCreateMatrix(context,
  VX_TYPE_FLOAT32, 3, 3);
vxAccessMatrix(matrix, NULL);
vxCommitMatrix(matrix, mat);
```

Functions

- vx_status vxuWarpPerspective (vx_context context, vx_image input, vx_matrix matrix, vx_enum type, vx_image output)

  [Immediate] Performs an Perspective warp on an image.
- vx_node vxWarpPerspectiveNode (vx_graph graph, vx_image input, vx_matrix matrix, vx_enum type, vx_image output)

  [Graph] Creates a Perspective Warp Node.

3.41.2 Function Documentation

vx_node vxWarpPerspectiveNode (vx_graph *graph,* vx_image *input,* vx_matrix *matrix,* vx_enum *type,* vx_image *output*)

[Graph] Creates a Perspective Warp Node.

Parameters

in	*graph*	The reference to the graph.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*matrix*	The perspective matrix. Must be 3x3 of type VX_TYPE_FLOAT32.
in	*type*	The interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_TYPE_AREA is not supported.
out	*output*	The output VX_DF_IMAGE_U8 image.

Note

Only VX_NODE_ATTRIBUTE_BORDER_MODE value VX_BORDER_MODE_UNDEFINED or VX_BORDER_MODE_CONSTANT is supported.

Returns

vx_node.

Return values

0	Node could not be created.
*	Node handle.

vx_status vxuWarpPerspective (vx_context *context*, vx_image *input*, vx_matrix *matrix*, vx_enum *type*, vx_image *output*)

[Immediate] Performs an Perspective warp on an image.

Parameters

in	*context*	The reference to the overall context.
in	*input*	The input VX_DF_IMAGE_U8 image.
in	*matrix*	The perspective matrix. Must be 3x3 of type VX_TYPE_FLOAT32.
in	*type*	The interpolation type from vx_interpolation_type_e. VX_INTERPOLATION_TYPE_AREA is not supported.
out	*output*	The output VX_DF_IMAGE_U8 image.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Success
*	An error occurred. See vx_status_e.

3.42 Basic Features

3.42.1 Detailed Description

The basic parts of OpenVX needed for computation.

Types in OpenVX intended to be derived from the C99 Section 7.18 standard definition of fixed width types.

Modules

- Objects

  Defines the basic objects within OpenVX.

Data Structures

- struct vx_coordinates2d_t

  The 2D Coordinates structure. More...
- struct vx_coordinates3d_t

  The 3D Coordinates structure. More...
- struct vx_delta_rectangle_t

  The changes in dimensions of the rectangle between input and output images in an output parameter validator. Used in conjunction with `VX_META_FORMAT_ATTRIBUTE_DELTA_RECTANGLE` and `vxSetMetaFormatAttribute`. More...
- struct vx_keypoint_t

  The keypoint data structure. More...
- struct vx_rectangle_t

  The rectangle data structure that is shared with the users. More...

Macros

- #define VX_ATTRIBUTE_BASE(vendor, object) (((vendor) $<<$ 20) | (object $<<$ 8))

  Defines the manner in which to combine the Vendor and Object IDs to get the base value of the enumeration.
- #define VX_ATTRIBUTE_ID_MASK (0x000000FF)

  An object's attribute ID is within the range of $[0, 2^8 - 1]$ (inclusive).
- #define VX_DF_IMAGE(a, b, c, d) ((a) | (b $<<$ 8) | (c $<<$ 16) | (d $<<$ 24))

  Converts a set of four chars into a `uint32_t` container of a VX_DF_IMAGE code.
- #define VX_ENUM_BASE(vendor, id) (((vendor) $<<$ 20) | (id $<<$ 12))

  Defines the manner in which to combine the Vendor and Object IDs to get the base value of the enumeration.
- #define VX_ENUM_MASK (0x00000FFF)

  A generic enumeration list can have values between $[0, 2^{12} - 1]$ (inclusive).
- #define VX_ENUM_TYPE(e) (((vx_uint32)e & VX_ENUM_TYPE_MASK) $>>$ 12)

  A macro to extract the enum type from an enumerated value.
- #define VX_ENUM_TYPE_MASK (0x000FF000)

  A type of enumeration. The valid range is between $[0, 2^8 - 1]$ (inclusive).
- #define VX_FMT_REF "%p"
- #define VX_FMT_SIZE "%zu"
- #define VX_KERNEL_BASE(vendor, lib) (((vendor) $<<$ 20) | (lib $<<$ 12))

  Defines the manner in which to combine the Vendor and Library IDs to get the base value of the enumeration.
- #define VX_KERNEL_MASK (0x00000FFF)

  An individual kernel in a library has its own unique ID within $[0, 2^{12} - 1]$ (inclusive).
- #define VX_LIBRARY(e) (((vx_uint32)e & VX_LIBRARY_MASK) $>>$ 12)

  A macro to extract the kernel library enumeration from a enumerated kernel value.
- #define VX_LIBRARY_MASK (0x000FF000)

  A library is a set of vision kernels with its own ID supplied by a vendor. The vendor defines the library ID. The range is $[0, 2^8 - 1]$ inclusive.

- #define VX_MAX_LOG_MESSAGE_LEN (1024)

  Defines the maximum length of a message buffer to copy from the log.
- #define VX_SCALE_UNITY (1024u)
- #define VX_TYPE(e) (((vx_uint32)e & VX_TYPE_MASK) >> 8)

  A macro to extract the type from an enumerated attribute value.
- #define VX_TYPE_MASK (0x000FFF00)

  A type mask removes the scalar/object type from the attribute. It is 3 nibbles in size and is contained between the third and second byte.
- #define VX_VENDOR(e) (((vx_uint32)e & VX_VENDOR_MASK) >> 20)

  A macro to extract the vendor ID from the enumerated value.
- #define VX_VENDOR_MASK (0xFFF00000)

  Vendor IDs are 2 nibbles in size and are located in the upper byte of the 4 bytes of an enumeration.
- #define VX_VERSION VX_VERSION_1_0
- #define VX_VERSION_1_0 (VX_VERSION_MAJOR(1) | VX_VERSION_MINOR(0))

  Defines the predefined version number for 1.0.
- #define VX_VERSION_MAJOR(x) ((x & 0xFF) << 8)
- #define VX_VERSION_MINOR(x) ((x & 0xFF) << 0)

Typedefs

- typedef char vx_char

  An 8 bit ASCII character.
- typedef uint32_t vx_df_image

  Used to hold a VX_DF_IMAGE code to describe the pixel format and color space.
- typedef int32_t vx_enum

  Sets the standard enumeration type size to be a fixed quantity.
- typedef float vx_float32

  A 32-bit float value.
- typedef double vx_float64

  A 64-bit float value (aka double).
- typedef int16_t vx_int16

  A 16-bit signed value.
- typedef int32_t vx_int32

  A 32-bit signed value.
- typedef int64_t vx_int64

  A 64-bit signed value.
- typedef int8_t vx_int8

  An 8-bit signed value.
- typedef size_t vx_size

  A wrapper of `size_t` to keep the naming convention uniform.
- typedef vx_enum vx_status

  A formal status type with known fixed size.
- typedef uint16_t vx_uint16

  A 16-bit unsigned value.
- typedef uint32_t vx_uint32

  A 32-bit unsigned value.
- typedef uint64_t vx_uint64

  A 64-bit unsigned value.
- typedef uint8_t vx_uint8

  An 8-bit unsigned value.

Enumerations

- enum vx_bool {
  vx_false_e = 0,
  vx_true_e }

  A Boolean value. This allows 0 to be FALSE, as it is in C, and any non-zero to be TRUE.
- enum vx_channel_e {
  VX_CHANNEL_0 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x0,
  VX_CHANNEL_1 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x1,
  VX_CHANNEL_2 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x2,
  VX_CHANNEL_3 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CHANNEL << 12)) + 0x3,
  VX_CHANNEL_R = VX_CHANNEL_0,
  VX_CHANNEL_G = VX_CHANNEL_1,
  VX_CHANNEL_B = VX_CHANNEL_2,
  VX_CHANNEL_A = VX_CHANNEL_3,
  VX_CHANNEL_Y = VX_CHANNEL_0,
  VX_CHANNEL_U = VX_CHANNEL_1,
  VX_CHANNEL_V = VX_CHANNEL_2 }

  The channel enumerations for channel extractions.
- enum vx_convert_policy_e {
  VX_CONVERT_POLICY_WRAP = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVERT_POLICY << 12)) + 0x0,
  VX_CONVERT_POLICY_SATURATE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_CONVERT_POLICY << 12)) + 0x1 }

  The Conversion Policy Enumeration.
- enum vx_df_image_e {
  VX_DF_IMAGE_VIRT = (('V') | ('I' << 8) | ('R' << 16) | ('T' << 24)),
  VX_DF_IMAGE_RGB = (('R') | ('G' << 8) | ('B' << 16) | ('2' << 24)),
  VX_DF_IMAGE_RGBX = (('R') | ('G' << 8) | ('B' << 16) | ('A' << 24)),
  VX_DF_IMAGE_NV12 = (('N') | ('V' << 8) | ('1' << 16) | ('2' << 24)),
  VX_DF_IMAGE_NV21 = (('N') | ('V' << 8) | ('2' << 16) | ('1' << 24)),
  VX_DF_IMAGE_UYVY = (('U') | ('Y' << 8) | ('V' << 16) | ('Y' << 24)),
  VX_DF_IMAGE_YUYV = (('Y') | ('U' << 8) | ('Y' << 16) | ('V' << 24)),
  VX_DF_IMAGE_IYUV = (('I') | ('Y' << 8) | ('U' << 16) | ('V' << 24)),
  VX_DF_IMAGE_YUV4 = (('Y') | ('U' << 8) | ('V' << 16) | ('4' << 24)),
  VX_DF_IMAGE_U8 = (('U') | ('0' << 8) | ('0' << 16) | ('8' << 24)),
  VX_DF_IMAGE_U16 = (('U') | ('0' << 8) | ('1' << 16) | ('6' << 24)),
  VX_DF_IMAGE_S16 = (('S') | ('0' << 8) | ('1' << 16) | ('6' << 24)),
  VX_DF_IMAGE_U32 = (('U') | ('0' << 8) | ('3' << 16) | ('2' << 24)),
  VX_DF_IMAGE_S32 = (('S') | ('0' << 8) | ('3' << 16) | ('2' << 24)) }

  Based on the VX_DF_IMAGE definition.
- enum vx_enum_e {

VX_ENUM_DIRECTION = 0x00,
VX_ENUM_ACTION = 0x01,
VX_ENUM_HINT = 0x02,
VX_ENUM_DIRECTIVE = 0x03,
VX_ENUM_INTERPOLATION = 0x04,
VX_ENUM_OVERFLOW = 0x05,
VX_ENUM_COLOR_SPACE = 0x06,
VX_ENUM_COLOR_RANGE = 0x07,
VX_ENUM_PARAMETER_STATE = 0x08,
VX_ENUM_CHANNEL = 0x09,
VX_ENUM_CONVERT_POLICY = 0x0A,
VX_ENUM_THRESHOLD_TYPE = 0x0B,
VX_ENUM_BORDER_MODE = 0x0C,
VX_ENUM_COMPARISON = 0x0D,
VX_ENUM_IMPORT_MEM = 0x0E,
VX_ENUM_TERM_CRITERIA = 0x0F,
VX_ENUM_NORM_TYPE = 0x10,
VX_ENUM_ACCESSOR = 0x11,
VX_ENUM_ROUND_POLICY = 0x12 }

The set of supported enumerations in OpenVX.

- enum vx_interpolation_type_e {
VX_INTERPOLATION_TYPE_NEAREST_NEIGHBOR = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_I↩
NTERPOLATION << 12)) + 0x0,
VX_INTERPOLATION_TYPE_BILINEAR = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_INTERPOLATI↩
ON << 12)) + 0x1,
VX_INTERPOLATION_TYPE_AREA = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_INTERPOLATION
<< 12)) + 0x2 }

The image reconstruction filters supported by image resampling operations.

- enum vx_status_e {
VX_STATUS_MIN = -25,
VX_ERROR_REFERENCE_NONZERO = -24,
VX_ERROR_MULTIPLE_WRITERS = -23,
VX_ERROR_GRAPH_ABANDONED = -22,
VX_ERROR_GRAPH_SCHEDULED = -21,
VX_ERROR_INVALID_SCOPE = -20,
VX_ERROR_INVALID_NODE = -19,
VX_ERROR_INVALID_GRAPH = -18,
VX_ERROR_INVALID_TYPE = -17,
VX_ERROR_INVALID_VALUE = -16,
VX_ERROR_INVALID_DIMENSION = -15,
VX_ERROR_INVALID_FORMAT = -14,
VX_ERROR_INVALID_LINK = -13,
VX_ERROR_INVALID_REFERENCE = -12,
VX_ERROR_INVALID_MODULE = -11,
VX_ERROR_INVALID_PARAMETERS = -10,
VX_ERROR_OPTIMIZED_AWAY = -9,
VX_ERROR_NO_MEMORY = -8,
VX_ERROR_NO_RESOURCES = -7,
VX_ERROR_NOT_COMPATIBLE = -6,
VX_ERROR_NOT_ALLOCATED = -5,
VX_ERROR_NOT_SUFFICIENT = -4,
VX_ERROR_NOT_SUPPORTED = -3,
VX_ERROR_NOT_IMPLEMENTED = -2,
VX_FAILURE = -1,
VX_SUCCESS = 0 }

The enumeration of all status codes.

- enum vx_type_e {

VX_TYPE_INVALID = 0x000,
VX_TYPE_CHAR = 0x001,
VX_TYPE_INT8 = 0x002,
VX_TYPE_UINT8 = 0x003,
VX_TYPE_INT16 = 0x004,
VX_TYPE_UINT16 = 0x005,
VX_TYPE_INT32 = 0x006,
VX_TYPE_UINT32 = 0x007,
VX_TYPE_INT64 = 0x008,
VX_TYPE_UINT64 = 0x009,
VX_TYPE_FLOAT32 = 0x00A,
VX_TYPE_FLOAT64 = 0x00B,
VX_TYPE_ENUM = 0x00C,
VX_TYPE_SIZE = 0x00D,
VX_TYPE_DF_IMAGE = 0x00E,
VX_TYPE_BOOL = 0x010,
VX_TYPE_SCALAR_MAX,
VX_TYPE_RECTANGLE = 0x020,
VX_TYPE_KEYPOINT = 0x021,
VX_TYPE_COORDINATES2D = 0x022,
VX_TYPE_COORDINATES3D = 0x023,
VX_TYPE_STRUCT_MAX,
VX_TYPE_USER_STRUCT_START = 0x100,
VX_TYPE_REFERENCE = 0x800,
VX_TYPE_CONTEXT = 0x801,
VX_TYPE_GRAPH = 0x802,
VX_TYPE_NODE = 0x803,
VX_TYPE_KERNEL = 0x804,
VX_TYPE_PARAMETER = 0x805,
VX_TYPE_DELAY = 0x806,
VX_TYPE_LUT = 0x807,
VX_TYPE_DISTRIBUTION = 0x808,
VX_TYPE_PYRAMID = 0x809,
VX_TYPE_THRESHOLD = 0x80A,
VX_TYPE_MATRIX = 0x80B,
VX_TYPE_CONVOLUTION = 0x80C,
VX_TYPE_SCALAR = 0x80D,
VX_TYPE_ARRAY = 0x80E,
VX_TYPE_IMAGE = 0x80F,
VX_TYPE_REMAP = 0x810,
VX_TYPE_ERROR = 0x811,
VX_TYPE_META_FORMAT = 0x812,
VX_TYPE_OBJECT_MAX }

The type enumeration lists all the known types in OpenVX.

- enum vx_vendor_id_e {

VX_ID_KHRONOS = 0x000,
VX_ID_TI = 0x001,
VX_ID_QUALCOMM = 0x002,
VX_ID_NVIDIA = 0x003,
VX_ID_ARM = 0x004,
VX_ID_BDTI = 0x005,
VX_ID_RENESAS = 0x006,
VX_ID_VIVANTE = 0x007,
VX_ID_XILINX = 0x008,
VX_ID_AXIS = 0x009,
VX_ID_MOVIDIUS = 0x00A,
VX_ID_SAMSUNG = 0x00B,
VX_ID_FREESCALE = 0x00C,
VX_ID_AMD = 0x00D,
VX_ID_BROADCOM = 0x00E,
VX_ID_INTEL = 0x00F,
VX_ID_MARVELL = 0x010,
VX_ID_MEDIATEK = 0x011,
VX_ID_ST = 0x012,
VX_ID_CEVA = 0x013,
VX_ID_ITSEEZ = 0x014,
VX_ID_IMAGINATION =0x015,
VX_ID_COGNIVUE = 0x016,
VX_ID_VIDEANTIS = 0x017,
VX_ID_MAX = 0xFFF,
VX_ID_DEFAULT = VX_ID_MAX }

The Vendor ID of the Implementation. As new vendors submit their implementations, this enumeration will grow.

Functions

- vx_status vxGetStatus (vx_reference reference)

  Provides a generic API to return status values from Object constructors if they fail.

3.42.2 Data Structure Documentation

struct vx_coordinates2d_t

The 2D Coordinates structure.

Definition at line 1390 of file vx_types.h.

Data Fields

vx_uint32	x	The X coordinate.
vx_uint32	y	The Y coordinate.

struct vx_coordinates3d_t

The 3D Coordinates structure.

Definition at line 1398 of file vx_types.h.

Data Fields

vx_uint32	x	The X coordinate.
vx_uint32	y	The Y coordinate.
vx_uint32	z	The Z coordinate.

struct vx_delta_rectangle_t

The changes in dimensions of the rectangle between input and output images in an output parameter validator. Used in conjunction with VX_META_FORMAT_ATTRIBUTE_DELTA_RECTANGLE and vxSetMetaFormat↩

Attribute.

See also

vx_kernel_output_validate_f
vx_meta_format

Definition at line 1380 of file vx_types.h.

Data Fields

vx_int32	delta_start_x	The change in the start x.
vx_int32	delta_start_y	The change in the start y.
vx_int32	delta_end_x	The change in the end x.
vx_int32	delta_end_y	The change in the end y.

struct vx_keypoint_t

The keypoint data structure.
Definition at line 1352 of file vx_types.h.

Data Fields

vx_int32	x	The x coordinate.
vx_int32	y	The y coordinate.
vx_float32	strength	The strength of the keypoint. Its definition is specific to the corner detector.
vx_float32	scale	Initialized to 0 by corner detectors.
vx_float32	orientation	Initialized to 0 by corner detectors.
vx_int32	tracking_status	A zero indicates a lost point. Initialized to 1 by corner detectors.
vx_float32	error	A tracking method specific error. Initialized to 0 by corner detectors.

struct vx_rectangle_t

The rectangle data structure that is shared with the users.
Definition at line 1365 of file vx_types.h.

Data Fields

vx_uint32	start_x	The Start X coordinate.
vx_uint32	start_y	The Start Y coordinate.
vx_uint32	end_x	The End X coordinate.
vx_uint32	end_y	The End Y coordinate.

3.42.3 Macro Definition Documentation

#define VX_VERSION_MAJOR(*x*) ((x & 0xFF) << 8)

Defines the major version number macro.
Definition at line 57 of file vx.h.

#define VX_VERSION_MINOR(*x*) ((x & 0xFF) << 0)

Defines the minor version number macro.
Definition at line 62 of file vx.h.

#define VX_VERSION VX_VERSION_1_0

Defines the OpenVX Version Number.
Definition at line 72 of file vx.h.

#define VX_TYPE_MASK (0x000FFF00)

A type mask removes the scalar/object type from the attribute. It is 3 nibbles in size and is contained between the third and second byte.

See also

vx_type_e

Definition at line 393 of file vx_types.h.

#define VX_DF_IMAGE(*a, b, c, d*) ((a) | (b << 8) | (c << 16) | (d << 24))

Converts a set of four chars into a `uint32_t` container of a VX_DF_IMAGE code.

Note

Use a `vx_df_image` variable to hold the value.

#define VX_ENUM_BASE(*vendor, id*) (((vendor) << 20) | (id << 12))

Defines the manner in which to combine the Vendor and Object IDs to get the base value of the enumeration.

From any enumerated value (with exceptions), the vendor, and enumeration type should be extractable. Those types that are exceptions are `vx_vendor_id_e`, `vx_type_e`, `vx_enum_e`, `vx_df_image_e`, and `vx_bool`.

#define VX_FMT_REF "%p"

Use to aid in debugging values in OpenVX.

Definition at line 1242 of file vx_types.h.

#define VX_FMT_SIZE "%zu"

Use to aid in debugging values in OpenVX.

Definition at line 1246 of file vx_types.h.

#define VX_SCALE_UNITY (1024u)

Use to indicate the 1:1 ratio in Q22.10 format.

Definition at line 1251 of file vx_types.h.

3.42.4 Typedef Documentation

typedef int32_t vx_enum

Sets the standard enumeration type size to be a fixed quantity.

All enumerable fields must use this type as the container to enforce enumeration ranges and sizeof() operations.

Definition at line 119 of file vx_types.h.

typedef vx_enum vx_status

A formal status type with known fixed size.

See also

vx_status_e

Definition at line 365 of file vx_types.h.

3.42.5 Enumeration Type Documentation

enum vx_bool

A Boolean value. This allows 0 to be FALSE, as it is in C, and any non-zero to be TRUE.

```
vx_bool ret = vx_true_e;
if (ret) printf("true!\n");
ret = vx_false_e;
if (!ret) printf("false!\n");
```

This would print both strings.

Enumerator

vx_false_e The "false" value.

vx_true_e The "true" value.

Definition at line 250 of file vx_types.h.

enum vx_type_e

The type enumeration lists all the known types in OpenVX.

Enumerator

VX_TYPE_INVALID An invalid type value. When passed an error must be returned.

VX_TYPE_CHAR A vx_char.

VX_TYPE_INT8 A vx_int8.

VX_TYPE_UINT8 A vx_uint8.

VX_TYPE_INT16 A vx_int16.

VX_TYPE_UINT16 A vx_uint16.

VX_TYPE_INT32 A vx_int32.

VX_TYPE_UINT32 A vx_uint32.

VX_TYPE_INT64 A vx_int64.

VX_TYPE_UINT64 A vx_uint64.

VX_TYPE_FLOAT32 A vx_float32.

VX_TYPE_FLOAT64 A vx_float64.

VX_TYPE_ENUM A vx_enum. Equivalent in size to a vx_int32.

VX_TYPE_SIZE A vx_size.

VX_TYPE_DF_IMAGE A vx_df_image.

VX_TYPE_BOOL A vx_bool.

VX_TYPE_SCALAR_MAX A floating value for comparison between scalars and structs.

VX_TYPE_RECTANGLE A vx_rectangle_t.

VX_TYPE_KEYPOINT A vx_keypoint_t.

VX_TYPE_COORDINATES2D A vx_coordinates2d_t.

VX_TYPE_COORDINATES3D A vx_coordinates3d_t.

VX_TYPE_STRUCT_MAX A floating value for comparison between structs and objects.

VX_TYPE_REFERENCE A vx_reference.

VX_TYPE_CONTEXT A vx_context.

VX_TYPE_GRAPH A vx_graph.

VX_TYPE_NODE A vx_node.

VX_TYPE_KERNEL A vx_kernel.

VX_TYPE_PARAMETER A vx_parameter.

VX_TYPE_DELAY A vx_delay.

VX_TYPE_LUT A vx_lut.

VX_TYPE_DISTRIBUTION A vx_distribution.

VX_TYPE_PYRAMID A vx_pyramid.

VX_TYPE_THRESHOLD A vx_threshold.

VX_TYPE_MATRIX A vx_matrix.

VX_TYPE_CONVOLUTION A vx_convolution.

VX_TYPE_SCALAR A vx_scalar. when needed to be completely generic for kernel validation.

VX_TYPE_ARRAY A vx_array.

VX_TYPE_IMAGE A vx_image.

VX_TYPE_REMAP A vx_remap.

VX_TYPE_ERROR An error object which has no type.

VX_TYPE_META_FORMAT A vx_meta_format.

VX_TYPE_OBJECT_MAX A value used for bound checking the object types.

Definition at line 268 of file vx_types.h.

enum vx_status_e

The enumeration of all status codes.

See also

vx_status.

Enumerator

VX_STATUS_MIN Indicates the lower bound of status codes in VX. Used for bounds checks only.

VX_ERROR_REFERENCE_NONZERO Indicates that an operation did not complete due to a reference count being non-zero.

VX_ERROR_MULTIPLE_WRITERS Indicates that the graph has more than one node outputting to the same data object. This is an invalid graph structure.

VX_ERROR_GRAPH_ABANDONED Indicates that the graph is stopped due to an error or a callback that abandoned execution.

VX_ERROR_GRAPH_SCHEDULED Indicates that the supplied graph already has been scheduled and may be currently executing.

VX_ERROR_INVALID_SCOPE Indicates that the supplied parameter is from another scope and cannot be used in the current scope.

VX_ERROR_INVALID_NODE Indicates that the supplied node could not be created.

VX_ERROR_INVALID_GRAPH Indicates that the supplied graph has invalid connections (cycles).

VX_ERROR_INVALID_TYPE Indicates that the supplied type parameter is incorrect.

VX_ERROR_INVALID_VALUE Indicates that the supplied parameter has an incorrect value.

VX_ERROR_INVALID_DIMENSION Indicates that the supplied parameter is too big or too small in dimension.

VX_ERROR_INVALID_FORMAT Indicates that the supplied parameter is in an invalid format.

VX_ERROR_INVALID_LINK Indicates that the link is not possible as specified. The parameters are incompatible.

VX_ERROR_INVALID_REFERENCE Indicates that the reference provided is not valid.

VX_ERROR_INVALID_MODULE This is returned from vxLoadKernels when the module does not contain the entry point.

VX_ERROR_INVALID_PARAMETERS Indicates that the supplied parameter information does not match the kernel contract.

VX_ERROR_OPTIMIZED_AWAY Indicates that the object refered to has been optimized out of existence.

VX_ERROR_NO_MEMORY Indicates that an internal or implicit allocation failed. Typically catastrophic. After detection, deconstruct the context.

See also

vxVerifyGraph.

VX_ERROR_NO_RESOURCES Indicates that an internal or implicit resource can not be acquired (not memory). This is typically catastrophic. After detection, deconstruct the context.

See also

vxVerifyGraph.

VX_ERROR_NOT_COMPATIBLE Indicates that the attempt to link two parameters together failed due to type incompatibilty.

VX_ERROR_NOT_ALLOCATED Indicates to the system that the parameter must be allocated by the system.

VX_ERROR_NOT_SUFFICIENT Indicates that the given graph has failed verification due to an insufficient number of required parameters, which cannot be automatically created. Typically this indicates required atomic parameters.

See also

vxVerifyGraph.

VX_ERROR_NOT_SUPPORTED Indicates that the requested set of parameters produce a configuration that cannot be supported. Refer to the supplied documentation on the configured kernels.

See also

vx_kernel_e.

VX_ERROR_NOT_IMPLEMENTED Indicates that the requested kernel is missing.

See also

vx_kernel_e vxGetKernelByName.

VX_FAILURE Indicates a generic error code, used when no other describes the error.

VX_SUCCESS No error.

Definition at line 331 of file vx_types.h.

enum vx_enum_e

The set of supported enumerations in OpenVX.
These can be extracted from enumerated values using VX_ENUM_TYPE.

Enumerator

VX_ENUM_DIRECTION Parameter Direction.

VX_ENUM_ACTION Action Codes.

VX_ENUM_HINT Hint Values.

VX_ENUM_DIRECTIVE Directive Values.

VX_ENUM_INTERPOLATION Interpolation Types.

VX_ENUM_OVERFLOW Overflow Policies.

VX_ENUM_COLOR_SPACE Color Space.

VX_ENUM_COLOR_RANGE Color Space Range.

VX_ENUM_PARAMETER_STATE Parameter State.

VX_ENUM_CHANNEL Channel Name.

VX_ENUM_CONVERT_POLICY Convert Policy.

VX_ENUM_THRESHOLD_TYPE Threshold Type List.

VX_ENUM_BORDER_MODE Border Mode List.

VX_ENUM_COMPARISON Comparison Values.

VX_ENUM_IMPORT_MEM The memory import enumeration.

VX_ENUM_TERM_CRITERIA A termination criteria.

VX_ENUM_NORM_TYPE A norm type.

VX_ENUM_ACCESSOR An accessor flag type.

VX_ENUM_ROUND_POLICY Rounding Policy.

Definition at line 486 of file vx_types.h.

enum vx_convert_policy_e

The Conversion Policy Enumeration.

Enumerator

VX_CONVERT_POLICY_WRAP Results are the least significant bits of the output operand, as if stored in two's complement binary format in the size of its bit-depth.

VX_CONVERT_POLICY_SATURATE Results are saturated to the bit depth of the output operand.

Definition at line 563 of file vx_types.h.

enum vx_df_image_e

Based on the VX_DF_IMAGE definition.

Note

Use `vx_df_image` to contain these values.

Enumerator

VX_DF_IMAGE_VIRT A virtual image of no defined type.

VX_DF_IMAGE_RGB A single plane of 24-bit pixel as 3 interleaved 8-bit units of R then G then B data. This uses the BT709 full range by default.

VX_DF_IMAGE_RGBX A single plane of 32-bit pixel as 4 interleaved 8-bit units of R then G then B data, then a *don't care* byte. This uses the BT709 full range by default.

VX_DF_IMAGE_NV12 A 2-plane YUV format of Luma (Y) and interleaved UV data at 4:2:0 sampling. This uses the BT709 full range by default.

VX_DF_IMAGE_NV21 A 2-lane YUV format of Luma (Y) and interleaved VU data at 4:2:0 sampling. This uses the BT709 full range by default.

VX_DF_IMAGE_UYVY A single plane of 32-bit macro pixel of U0, Y0, V0, Y1 bytes. This uses the BT709 full range by default.

VX_DF_IMAGE_YUYV A single plane of 32-bit macro pixel of Y0, U0, Y1, V0 bytes. This uses the BT709 full range by default.

VX_DF_IMAGE_IYUV A 3 plane of 8-bit 4:2:0 sampled Y, U, V planes. This uses the BT709 full range by default.

VX_DF_IMAGE_YUV4 A 3 plane of 8 bit 4:4:4 sampled Y, U, V planes. This uses the BT709 full range by default.

VX_DF_IMAGE_U8 A single plane of unsigned 8-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

VX_DF_IMAGE_U16 A single plane of unsigned 16-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

VX_DF_IMAGE_S16 A single plane of signed 16-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

VX_DF_IMAGE_U32 A single plane of unsigned 32-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

VX_DF_IMAGE_S32 A single plane of unsigned 32-bit data. The range of data is not specified, as it may be extracted from a YUV or generated.

Definition at line 576 of file vx_types.h.

enum vx_channel_e

The channel enumerations for channel extractions.

See also

vxChannelExtractNode
vxuChannelExtract
VX_KERNEL_CHANNEL_EXTRACT

Enumerator

VX_CHANNEL_0 Used by formats with unknown channel types.

VX_CHANNEL_1 Used by formats with unknown channel types.

VX_CHANNEL_2 Used by formats with unknown channel types.

VX_CHANNEL_3 Used by formats with unknown channel types.

VX_CHANNEL_R Use to extract the RED channel, no matter the byte or packing order.

VX_CHANNEL_G Use to extract the GREEN channel, no matter the byte or packing order.

VX_CHANNEL_B Use to extract the BLUE channel, no matter the byte or packing order.

VX_CHANNEL_A Use to extract the ALPHA channel, no matter the byte or packing order.

VX_CHANNEL_Y Use to extract the LUMA channel, no matter the byte or packing order.

VX_CHANNEL_U Use to extract the Cb/U channel, no matter the byte or packing order.

VX_CHANNEL_V Use to extract the Cr/V/Value channel, no matter the byte or packing order.

Definition at line 953 of file vx_types.h.

enum vx_interpolation_type_e

The image reconstruction filters supported by image resampling operations.

The edge of a pixel is interpreted as being aligned to the edge of the image. The value for an output pixel is evaluated at the center of that pixel.

This means, for example, that an even enlargement of a factor of two in nearest-neighbor interpolation will replicate every source pixel into a 2x2 quad in the destination, and that an even shrink by a factor of two in bilinear interpolation will create each destination pixel by average a 2x2 quad of source pixels.

Samples that cross the boundary of the source image have values determined by the border mode - see `vx_border_mode_e` and `VX_NODE_ATTRIBUTE_BORDER_MODE`.

See also

vxuScaleImage
vxScaleImageNode
VX_KERNEL_SCALE_IMAGE
vxuWarpAffine
vxWarpAffineNode
VX_KERNEL_WARP_AFFINE
vxuWarpPerspective
vxWarpPerspectiveNode
VX_KERNEL_WARP_PERSPECTIVE

Enumerator

VX_INTERPOLATION_TYPE_NEAREST_NEIGHBOR Output values are defined to match the source pixel whose center is nearest to the sample position.

VX_INTERPOLATION_TYPE_BILINEAR Output values are defined by bilinear interpolation between the pixels whose centers are closest to the sample position, weighted linearly by the distance of the sample from the pixel centers.

VX_INTERPOLATION_TYPE_AREA Output values are determined by averaging the source pixels whose areas fall under the area of the destination pixel, projected onto the source image.

Definition at line 1013 of file vx_types.h.

enum vx_vendor_id_e

The Vendor ID of the Implementation. As new vendors submit their implementations, this enumeration will grow.

Enumerator

VX_ID_KHRONOS The Khronos Group.

VX_ID_TI Texas Instruments, Inc.

VX_ID_QUALCOMM Qualcomm, Inc.

VX_ID_NVIDIA NVIDIA Corporation.

VX_ID_ARM ARM Ltd.

VX_ID_BDTI Berkley Design Technology, Inc.

VX_ID_RENESAS Renasas Electronics.

VX_ID_VIVANTE Vivante Corporation.

VX_ID_XILINX Xilinx Inc.

VX_ID_AXIS Axis Communications.

VX_ID_MOVIDIUS Movidius Ltd.

VX_ID_SAMSUNG Samsung Electronics.

VX_ID_FREESCALE Freescale Semiconductor.

VX_ID_AMD Advanced Micro Devices.

VX_ID_BROADCOM Broadcom Corporation.

VX_ID_INTEL Intel Corporation.

VX_ID_MARVELL Marvell Technology Group Ltd.

VX_ID_MEDIATEK MediaTek, Inc.

VX_ID_ST STMicroelectronics.

VX_ID_CEVA CEVA DSP.

VX_ID_ITSEEZ Itseez, Inc.

VX_ID_IMAGINATION Imagination Technologies.

VX_ID_COGNIVUE CogniVue Corporation.

VX_ID_VIDEANTIS Videantis.

VX_ID_DEFAULT For use by all Kernel authors until they can obtain an assigned ID.

Definition at line 37 of file vx_vendors.h.

3.42.6 Function Documentation

vx_status vxGetStatus (vx_reference *reference*)

Provides a generic API to return status values from Object constructors if they fail.

Note

Users do not need to strictly check every object creator as the errors should properly propogate and be detected during verification time or run-time.

```
vx_image img = vxCreateImage(context, 639, 480,
      VX_DF_IMAGE_UYVY);
vx_status status = vxGetStatus((vx_reference)img);
// status == VX_ERROR_INVALID_DIMENSIONS
vxReleaseImage(&img);
```

Precondition

Appropriate Object Creator function.

Postcondition

Appropriate Object Release function.

Parameters

in	*reference*	The reference to check for construction errors.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No error.
*	Some error occurred, please check enumeration list and constructor.

3.43 Objects

3.43.1 Detailed Description

Defines the basic objects within OpenVX.

All objects in OpenVX derive from a `vx_reference` and contain a reference to the `vx_context` from which they were made, except the `vx_context` itself.

Modules

- Object: Reference

  Defines the Reference Object interface.
- Object: Context

  Defines the Context Object Interface.
- Object: Graph

  Defines the Graph Object interface.
- Object: Node

  Defines the Node Object interface.
- Object: Array

  Defines the Array Object Interface.
- Object: Convolution

  Defines the Image Convolution Object interface.
- Object: Distribution

  Defines the Distribution Object Interface.
- Object: Image

  Defines the Image Object interface.
- Object: LUT

  Defines the Look-Up Table Interface.
- Object: Matrix

  Defines the Matrix Object Interface.
- Object: Pyramid

  Defines the Image Pyramid Object Interface.
- Object: Remap

  Defines the Remap Object Interface.
- Object: Scalar

  Defines the Scalar Object interface.
- Object: Threshold

  Defines the Threshold Object Interface.

3.44 Object: Reference

3.44.1 Detailed Description

Defines the Reference Object interface.

All objects in OpenVX are derived (in the object-oriented sense) from `vx_reference`. All objects shall be able to be cast back to this type safely.

Typedefs

- typedef struct _vx_reference * vx_reference

  A generic opaque reference to any object within OpenVX.

Enumerations

- enum vx_reference_attribute_e {
  VX_REF_ATTRIBUTE_COUNT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REFERENCE << 8)) + 0x0,
  VX_REF_ATTRIBUTE_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REFERENCE << 8)) + 0x1 }

  The reference attributes list.

Functions

- vx_status vxQueryReference (vx_reference ref, vx_enum attribute, void *ptr, vx_size size)

  Queries any reference type for some basic information (count, type).

3.44.2 Typedef Documentation

typedef struct _vx_reference* vx_reference

A generic opaque reference to any object within OpenVX.

A user of OpenVX should not assume that this can be cast directly to anything; however, any object in OpenVX can be cast back to this for the purposes of querying attributes of the object or for passing the object as a parameter to functions that take a `vx_reference` type. If the API does not take that specific type but may take others, an error may be returned from the API.

Definition at line 112 of file vx_types.h.

3.44.3 Enumeration Type Documentation

enum vx_reference_attribute_e

The reference attributes list.

Enumerator

VX_REF_ATTRIBUTE_COUNT Returns the reference count of the object. Use a `vx_uint32` parameter.

VX_REF_ATTRIBUTE_TYPE Returns the `vx_type_e` of the reference. Use a `vx_enum` parameter.

Definition at line 642 of file vx_types.h.

3.44.4 Function Documentation

vx_status vxQueryReference (vx_reference *ref*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries any reference type for some basic information (count, type).

Parameters

in	*ref*	The reference to query.
in	*attribute*	The value for which to query. Use vx_reference_attribute_e.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

3.45 Object: Context

3.45.1 Detailed Description

Defines the Context Object Interface.

The OpenVX context is the object domain for all OpenVX objects. All data objects *live* in the context as well as all framework objects. The OpenVX context keeps reference counts on all objects and must do garbage collection during its deconstruction to free lost references. While multiple clients may connect to the OpenVX context, all data are private in that the references referring to data objects are given only to the creating party.

Macros

- #define VX_MAX_IMPLEMENTATION_NAME (64)

  Defines the maximum number of characters in a implementation string.

Typedefs

- typedef struct _vx_context * vx_context

  An opaque reference to the implementation context.

Enumerations

- enum vx_accessor_e {
  VX_READ_ONLY = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACCESSOR << 12)) + 0x1,
  VX_WRITE_ONLY = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACCESSOR << 12)) + 0x2,
  VX_READ_AND_WRITE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACCESSOR << 12)) + 0x3 }

  *The memory accessor hint flags. These enumeration values are used to indicate desired system behavior, not the **User** intent. For example: these can be interpretted as hints to the system about cache operations or marshalling operations.*
- enum vx_context_attribute_e {
  VX_CONTEXT_ATTRIBUTE_VENDOR_ID = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x0,
  VX_CONTEXT_ATTRIBUTE_VERSION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x1,
  VX_CONTEXT_ATTRIBUTE_UNIQUE_KERNELS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x2,
  VX_CONTEXT_ATTRIBUTE_MODULES = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x3,
  VX_CONTEXT_ATTRIBUTE_REFERENCES = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x4,
  VX_CONTEXT_ATTRIBUTE_IMPLEMENTATION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x5,
  VX_CONTEXT_ATTRIBUTE_EXTENSIONS_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x6,
  VX_CONTEXT_ATTRIBUTE_EXTENSIONS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x7,
  VX_CONTEXT_ATTRIBUTE_CONVOLUTION_MAXIMUM_DIMENSION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x8,
  VX_CONTEXT_ATTRIBUTE_OPTICAL_FLOW_WINDOW_MAXIMUM_DIMENSION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0x9,
  VX_CONTEXT_ATTRIBUTE_IMMEDIATE_BORDER_MODE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0xA,
  VX_CONTEXT_ATTRIBUTE_UNIQUE_KERNEL_TABLE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONTEXT << 8)) + 0xB }

  A list of context attributes.

- enum vx_import_type_e {
  VX_IMPORT_TYPE_NONE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_IMPORT_MEM << 12)) + 0x0,
  VX_IMPORT_TYPE_HOST = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_IMPORT_MEM << 12)) + 0x1
  }

  An enumeration of memory import types.
- enum vx_round_policy_e {
  VX_ROUND_POLICY_TO_ZERO = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ROUND_POLICY << 12)) + 0x1,
  VX_ROUND_POLICY_TO_NEAREST_EVEN = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ROUND_↩ POLICY << 12)) + 0x2 }

  The Round Policy Enumeration.
- enum vx_termination_criteria_e {
  VX_TERM_CRITERIA_ITERATIONS = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_TERM_CRITERIA << 12)) + 0x0,
  VX_TERM_CRITERIA_EPSILON = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_TERM_CRITERIA << 12)) + 0x1,
  VX_TERM_CRITERIA_BOTH = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_TERM_CRITERIA << 12)) + 0x2 }

  The termination criteria list.

Functions

- vx_context vxCreateContext ()

  Creates a `vx_context`.
- vx_context vxGetContext (vx_reference reference)

  Retrieves the context from any reference from within a context.
- vx_status vxQueryContext (vx_context context, vx_enum attribute, void ∗ptr, vx_size size)

  Queries the context for some specific information.
- vx_status vxReleaseContext (vx_context ∗context)

  Releases the OpenVX object context.
- vx_status vxSetContextAttribute (vx_context context, vx_enum attribute, void ∗ptr, vx_size size)

  Sets an attribute on the context.

3.45.2 Typedef Documentation

typedef struct _vx_context∗ vx_context

An opaque reference to the implementation context.

See also

vxCreateContext

Definition at line 180 of file vx_types.h.

3.45.3 Enumeration Type Documentation

enum vx_context_attribute_e

A list of context attributes.

Enumerator

VX_CONTEXT_ATTRIBUTE_VENDOR_ID Queries the unique vendor ID. Use a `vx_uint16`.

VX_CONTEXT_ATTRIBUTE_VERSION Queries the OpenVX Version Number. Use a `vx_uint16`

VX_CONTEXT_ATTRIBUTE_UNIQUE_KERNELS Queries the context for the number of *unique* kernels. Use a `vx_uint32` parameter.

VX_CONTEXT_ATTRIBUTE_MODULES Queries the context for the number of active modules. Use a vx_uint32 parameter.

VX_CONTEXT_ATTRIBUTE_REFERENCES Queries the context for the number of active references. Use a vx_uint32 parameter.

VX_CONTEXT_ATTRIBUTE_IMPLEMENTATION Queries the context for it's implementation name. Use a vx_char[VX_MAX_IMPLEMENTATION_NAME] array.

VX_CONTEXT_ATTRIBUTE_EXTENSIONS_SIZE Queries the number of bytes in the extensions string. Use a vx_size parameter.

VX_CONTEXT_ATTRIBUTE_EXTENSIONS Retrieves the extensions string. This is a space-separated string of extension names. Use a vx_char pointer allocated to the size returned from VX_CONTEXT_ATTRIBUTE_EXTENSIONS_SIZE.

VX_CONTEXT_ATTRIBUTE_CONVOLUTION_MAXIMUM_DIMENSION The maximum width or height of a convolution matrix. Use a vx_size parameter. Each vendor must support centered kernels of size w X h, where both w and h are odd numbers, 3 <= w <= n and 3 <= h <= n, where n is the value of the VX_CONTEXT_ATTRIBUTE_CONVOLUTION_MAXIMUM_DIMENSION attribute. n is an odd number that should not be smaller than 9. w and h may or may not be equal to each other. All combinations of w and h meeting the conditions above must be supported. The behavior of vxCreateConvolution is undefined for values larger than the value returned by this attribute.

VX_CONTEXT_ATTRIBUTE_OPTICAL_FLOW_WINDOW_MAXIMUM_DIMENSION The maximum window dimension of the OpticalFlowPyrLK kernel.

See also

VX_KERNEL_OPTICAL_FLOW_PYR_LK. Use a vx_size parameter.

VX_CONTEXT_ATTRIBUTE_IMMEDIATE_BORDER_MODE The border mode for immediate mode functions. Graph mode functions are unaffected by this attribute. Use a pointer to a vx_border_mode_t structure as parameter.

Note

The assumed default value for immediate mode functions is VX_BORDER_MODE_UNDEFINED.

VX_CONTEXT_ATTRIBUTE_UNIQUE_KERNEL_TABLE Returns the table of all unique the kernels that exist in the context. Use a vx_kernel_info_t array.

Precondition

You must call vxQueryContext with VX_CONTEXT_ATTRIBUTE_UNIQUE_KERNELS to compute the necessary size of the array.

Definition at line 652 of file vx_types.h.

enum vx_import_type_e

An enumeration of memory import types.

Enumerator

VX_IMPORT_TYPE_NONE For memory allocated through OpenVX, this is the import type.

VX_IMPORT_TYPE_HOST The default memory type to import from the Host.

Definition at line 982 of file vx_types.h.

enum vx_termination_criteria_e

The termination criteria list.

See also

Optical Flow Pyramid (LK)

Enumerator

VX_TERM_CRITERIA_ITERATIONS Indicates a termination after a set number of iterations.

VX_TERM_CRITERIA_EPSILON Indicates a termination after matching against the value of eplison provided to the function.

VX_TERM_CRITERIA_BOTH Indicates that both an iterations and eplison method are employed. Whichever one matches first causes the termination.

Definition at line 1085 of file vx_types.h.

enum vx_accessor_e

The memory accessor hint flags. These enumeration values are used to indicate desired *system* behavior, not the **User** intent. For example: these can be interpretted as hints to the system about cache operations or marshalling operations.

Enumerator

VX_READ_ONLY The memory shall be treated by the system as if it were read-only. If the User writes to this memory, the results are implementation defined.

VX_WRITE_ONLY The memory shall be treated by the system as if it were write-only. If the User reads from this memory, the results are implementation defined.

VX_READ_AND_WRITE The memory shall be treated by the system as if it were readable and writeable.

Definition at line 1123 of file vx_types.h.

enum vx_round_policy_e

The Round Policy Enumeration.

Enumerator

VX_ROUND_POLICY_TO_ZERO When scaling, this truncates the least significant values that are lost in operations.

VX_ROUND_POLICY_TO_NEAREST_EVEN When scaling, this rounds to nearest even output value.

Definition at line 1140 of file vx_types.h.

3.45.4 Function Documentation

vx_context vxCreateContext ( )

Creates a `vx_context`.
This creates a top-level object context for OpenVX.

Note

This is required to do anything else.

Returns

The reference to the implementation context.

Return values

0	No context was created.
*	A context reference.

Postcondition

vxReleaseContext

vx_status vxReleaseContext (vx_context * *context*)

Releases the OpenVX object context.

All reference counted objects are garbage-collected by the return of this call. No calls are possible using the parameter context after the context has been released until a new reference from vxCreateContext is returned. All outstanding references to OpenVX objects from this context are invalid after this call.

Parameters

in	*context*	The pointer to the reference to the context.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

Precondition

vxCreateContext

vx_context vxGetContext (vx_reference *reference*)

Retrieves the context from any reference from within a context.

Parameters

in	*reference*	The reference from which to extract the context.

Returns

The overall context that created the particular reference.

vx_status vxQueryContext (vx_context *context*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries the context for some specific information.

Parameters

in	*context*	The reference to the context.
in	*attribute*	The attribute to query. Use a vx_context_attribute_e.

out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the context is not a vx_context.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.
VX_ERROR_NOT_SUPPORTED	If the attribute is not supported on this implementation.

vx_status vxSetContextAttribute (vx_context *context*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Sets an attribute on the context.

Parameters

in	*context*	The handle to the overall context.
in	*attribute*	The attribute to set from vx_context_attribute_e.
in	*ptr*	The pointer to the data to which to set the attribute.
in	*size*	The size in bytes of the data to which *ptr* points.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the context is not a vx_context.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.
VX_ERROR_NOT_SUPPORTED	If the attribute is not settable.

3.46 Object: Graph

3.46.1 Detailed Description

Defines the Graph Object interface.

A set of nodes connected in a directed (only goes one-way) acyclic (does not loop back) fashion. A Graph may have sets of Nodes that are unconnected to other sets of Nodes within the same Graph. See Graph Formalisms.

Typedefs

- typedef struct _vx_graph * vx_graph

  An opaque reference to a graph.

Enumerations

- enum vx_graph_attribute_e {
  VX_GRAPH_ATTRIBUTE_NUMNODES = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_GRAPH << 8)) + 0x0,
  VX_GRAPH_ATTRIBUTE_STATUS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_GRAPH << 8)) + 0x1,
  VX_GRAPH_ATTRIBUTE_PERFORMANCE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_GRAPH << 8)) + 0x2,
  VX_GRAPH_ATTRIBUTE_NUMPARAMETERS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_GRAPH << 8)) + 0x3 }

  The graph attributes list.

Functions

- vx_graph vxCreateGraph (vx_context context)

  Creates an empty graph.
- vx_bool vxIsGraphVerified (vx_graph graph)

  Returns a Boolean to indicate the state of graph verification.
- vx_status vxProcessGraph (vx_graph graph)

  This function causes the synchronous processing of a graph. If the graph has not been verified, then the implementation verifies the graph immediately. If verification fails this function returns a status identical to what `vxVerifyGraph` would return. After the graph verfies successfully then processing occurs. If the graph was previously verified via `vxVerifyGraph` or `vxProcessGraph` then the graph is processed. This function blocks until the graph is completed.
- vx_status vxQueryGraph (vx_graph graph, vx_enum attribute, void *ptr, vx_size size)

  Allows the user to query attributes of the Graph.
- vx_status vxReleaseGraph (vx_graph *graph)

  Releases a reference to a graph. The object may not be garbage collected until its total reference count is zero. Once the reference count is zero, all node references in the graph are automatically released as well. Data referenced by those nodes may not be released as the user may have external references to the data.
- vx_status vxScheduleGraph (vx_graph graph)

  Schedules a graph for future execution.
- vx_status vxSetGraphAttribute (vx_graph graph, vx_enum attribute, void *ptr, vx_size size)

  Allows the set to attributes on the Graph.
- vx_status vxVerifyGraph (vx_graph graph)

  Verifies the state of the graph before it is executed. This is useful to catch programmer errors and contract errors. If not verified, the graph verifies before being processed.
- vx_status vxWaitGraph (vx_graph graph)

  Waits for a specific graph to complete.

3.46.2 Typedef Documentation

typedef struct _vx_graph* vx_graph

An opaque reference to a graph.

See also

vxCreateGraph

Definition at line 173 of file vx_types.h.

3.46.3 Enumeration Type Documentation

enum vx_graph_attribute_e

The graph attributes list.

Enumerator

VX_GRAPH_ATTRIBUTE_NUMNODES Returns the number of nodes in a graph. Use a vx_uint32 parameter.

VX_GRAPH_ATTRIBUTE_STATUS Returns the overall status of the graph. Use a vx_status parameter.

VX_GRAPH_ATTRIBUTE_PERFORMANCE Returns the overall performance of the graph. Use a vx_perf_t parameter.

VX_GRAPH_ATTRIBUTE_NUMPARAMETERS Returns the number of explicitly declared parameters on the graph. Use a vx_uint32 parameter.

Definition at line 794 of file vx_types.h.

3.46.4 Function Documentation

vx_graph vxCreateGraph (vx_context *context*)

Creates an empty graph.

Parameters

in	*context*	The reference to the implementation context.

Returns

A graph reference.

Return values

0	if an error occurred.

vx_status vxReleaseGraph (vx_graph * *graph*)

Releases a reference to a graph. The object may not be garbage collected until its total reference count is zero. Once the reference count is zero, all node references in the graph are automatically released as well. Data referenced by those nodes may not be released as the user may have external references to the data.

Parameters

in	*graph*	The pointer to the graph to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxVerifyGraph (vx_graph *graph*)

Verifies the state of the graph before it is executed. This is useful to catch programmer errors and contract errors. If not verified, the graph verifies before being processed.

Precondition

Memory for data objects is not guarenteed to exist before this call.

Postcondition

After this call data objects exist unless the implementation optimized them out.

Parameters

in	*graph*	The reference to the graph to verify.

Returns

A status code for graphs with more than one error; it is undefined which error will be returned. Register a log callback using vxRegisterLogCallback to receive each specific error in the graph.
A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.
VX_ERROR_MULTIPLE_WRITERS	If the graph contains more than one writer to any data object.
VX_ERROR_INVALID_NODE	If a node in the graph is invalid or failed be created.
VX_ERROR_INVALID_GRAPH	If the graph contains cycles or some other invalid topology.
VX_ERROR_INVALID_TYPE	If any parameter on a node is given the wrong type.
VX_ERROR_INVALID_VALUE	If any value of any parameter is out of bounds of specification.
VX_ERROR_INVALID_FORMAT	If the image format is not compatible.

See also

vxConvertReference
vxProcessGraph

vx_status vxProcessGraph (vx_graph *graph*)

This function causes the synchronous processing of a graph. If the graph has not been verified, then the implementation verifies the graph immediately. If verification fails this function returns a status identical to what vxVerifyGraph would return. After the graph verfies successfully then processing occurs. If the graph was previously verified via vxVerifyGraph or vxProcessGraph then the graph is processed. This function blocks until the graph is completed.

Parameters

in	*graph*	The graph to execute.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Graph has been processed.
VX_FAILURE	A catastrophic error occurred during processing.
*	See vxVerifyGraph.

Precondition

vxVerifyGraph must return VX_SUCCESS before this function will pass.

See also

vxVerifyGraph

vx_status vxScheduleGraph (vx_graph *graph*)

Schedules a graph for future execution.

Parameters

in	*graph*	The graph to schedule.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_NO_RESOU↩ RCES	The graph cannot be scheduled now.
VX_ERROR_NOT_SUFFI↩ CIENT	The graph is not verified and has failed forced verification.
VX_SUCCESS	The graph has been scheduled.

Precondition

vxVerifyGraph must return VX_SUCCESS before this function will pass.

vx_status vxWaitGraph (vx_graph *graph*)

Waits for a specific graph to complete.

Parameters

in	*graph*	The graph to wait on.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	The graph has completed.
VX_FAILURE	The graph has not completed yet.

Precondition

vxScheduleGraph

vx_status vxQueryGraph (vx_graph *graph*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Allows the user to query attributes of the Graph.

Parameters

in	*graph*	The reference to the created graph.
in	*attribute*	The vx_graph_attribute_e type needed.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_status vxSetGraphAttribute (vx_graph *graph*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Allows the set to attributes on the Graph.

Parameters

in	*graph*	The reference to the graph.
in	*attribute*	The vx_graph_attribute_e type needed.
in	*ptr*	The location from which to read the value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_bool vxIsGraphVerified (vx_graph *graph*)

Returns a Boolean to indicate the state of graph verification.

Parameters

in	*graph*	The reference to the graph to check.

Returns

A vx_bool value.

Return values

vx_true_e	The graph is verified.
vx_false_e	The graph is not verified. It must be verified before execution either through vxVerifyGraph or automatically through vxProcessGraph or vxScheduleGraph.

3.47 Object: Node

3.47.1 Detailed Description

Defines the Node Object interface.

A node is an instance of a kernel that will be paired with a specific set of references (the parameters). Nodes are created from and associated with a single graph only. When a `vx_parameter` is extracted from a Node, an additional attribute can be accessed:

- *Reference* - The `vx_reference` assigned to this parameter index from the Node creation function (e.g., `vxSobel3x3Node`).

Typedefs

- typedef struct _vx_node * vx_node

  An opaque reference to a kernel node.

Enumerations

- enum vx_node_attribute_e {
  VX_NODE_ATTRIBUTE_STATUS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x0,
  VX_NODE_ATTRIBUTE_PERFORMANCE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x1,
  VX_NODE_ATTRIBUTE_BORDER_MODE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x2,
  VX_NODE_ATTRIBUTE_LOCAL_DATA_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x3,
  VX_NODE_ATTRIBUTE_LOCAL_DATA_PTR = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_NODE << 8)) + 0x4 }

  The node attributes list.

Functions

- vx_status vxQueryNode (vx_node node, vx_enum attribute, void *ptr, vx_size size)

  Allows a user to query information out of a node.
- vx_status vxReleaseNode (vx_node *node)

  Releases a reference to a Node object. The object may not be garbage collected until its total reference count is zero.
- void vxRemoveNode (vx_node *node)

  Removes a Node from its parent Graph and releases it.
- vx_status vxSetNodeAttribute (vx_node node, vx_enum attribute, void *ptr, vx_size size)

  Allows a user to set attribute of a node before Graph Validation.

3.47.2 Typedef Documentation

typedef struct _vx_node* vx_node

An opaque reference to a kernel node.

See also

vxCreateGenericNode

Definition at line 166 of file vx_types.h.

3.47.3 Enumeration Type Documentation

enum vx_node_attribute_e

The node attributes list.

Enumerator

VX_NODE_ATTRIBUTE_STATUS Queries the status of node execution. Use a vx_status parameter.

VX_NODE_ATTRIBUTE_PERFORMANCE Queries the performance of the node execution. Use a vx_perf_t parameter.

VX_NODE_ATTRIBUTE_BORDER_MODE Gets or sets the border mode of the node. Use a vx_border_mode_t structure.

VX_NODE_ATTRIBUTE_LOCAL_DATA_SIZE Indicates the size of the kernel local memory area. Use a vx_size parameter.

VX_NODE_ATTRIBUTE_LOCAL_DATA_PTR Indicates the pointer kernel local memory area. Use a void * parameter.

Definition at line 728 of file vx_types.h.

3.47.4 Function Documentation

vx_status vxQueryNode (vx_node *node*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Allows a user to query information out of a node.

Parameters

in	*node*	The reference to the node to query.
in	*attribute*	Use vx_node_attribute_e value to query for information.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Successful
VX_ERROR_INVALID_PARAMETERS	The type or size is incorrect.

vx_status vxSetNodeAttribute (vx_node *node*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Allows a user to set attribute of a node before Graph Validation.

Parameters

in	*node*	The reference to the node to set.
in	*attribute*	Use vx_node_attribute_e value to query for information.
out	*ptr*	The output pointer to where to send the value.
in	*size*	The size of the objects to which *ptr* points.

Note

Some attributes are inherited from the vx_kernel, which was used to create the node. Some of these can be overridden using this API, notably VX_NODE_ATTRIBUTE_LOCAL_DATA_SIZE and VX_NODE_ATTRIBUTE_LOCAL_DATA_PTR.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	The attribute was set.
VX_ERROR_INVALID_REFERENCE	node is not a vx_node.
VX_ERROR_INVALID_PARAMETER	size is not correct for the type needed.

vx_status vxReleaseNode (vx_node * *node*)

Releases a reference to a Node object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*node*	The pointer to the reference of the node to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

void vxRemoveNode (vx_node * *node*)

Removes a Node from its parent Graph and releases it.

Parameters

in	*node*	The pointer to the node to remove and release.

Postcondition

After returning from this function the reference is zeroed.

3.48 Object: Array

3.48.1 Detailed Description

Defines the Array Object Interface.

Array is a strongly-typed container, which provides random access by index to its elements in constant time. It uses value semantics for its own elements and holds copies of data. This is an example `for` loop over an Array:

```
  vx_size i, stride = 0ul;
  void *base = NULL;
  /* access entire array at once */
  vxAccessArrayRange(array, 0, num_items, &stride, &base,
VX_READ_AND_WRITE);
  for (i = 0; i < num_items; i++)
  {
      vxArrayItem(mystruct, base, i, stride).some_uint += i;
      vxArrayItem(mystruct, base, i, stride).some_double = 3.14f;
  }
  vxCommitArrayRange(array, 0, num_items, base);
```

Macros

- #define vxArrayItem(type, ptr, index, stride) (*(type *)(vxFormatArrayPointer((ptr), (index), (stride))))

  Allows access to an array item as a typecast pointer deference.
- #define vxFormatArrayPointer(ptr, index, stride) (&(((vx_uint8*)(ptr))[(index) * (stride)]))

  Accesses a specific indexed element in an array.

Typedefs

- typedef struct _vx_array * vx_array

  The Array Object. Array is a strongly-typed container for other data structures.

Enumerations

- enum vx_array_attribute_e {
  VX_ARRAY_ATTRIBUTE_ITEMTYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_ARRAY << 8)) + 0x0,
  VX_ARRAY_ATTRIBUTE_NUMITEMS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_ARRAY << 8)) + 0x1,
  VX_ARRAY_ATTRIBUTE_CAPACITY = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_ARRAY << 8)) + 0x2,
  VX_ARRAY_ATTRIBUTE_ITEMSIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_ARRAY << 8)) + 0x3
  }

  The array object attributes.

Functions

- vx_status vxAccessArrayRange (vx_array arr, vx_size start, vx_size end, vx_size *stride, void **ptr, vx_enum usage)

  Grants access to a sub-range of an Array.
- vx_status vxAddArrayItems (vx_array arr, vx_size count, void *ptr, vx_size stride)

  Adds items to the Array.
- vx_status vxCommitArrayRange (vx_array arr, vx_size start, vx_size end, void *ptr)

  Commits data back to the Array object.
- vx_array vxCreateArray (vx_context context, vx_enum item_type, vx_size capacity)

  Creates a reference to an Array object.
- vx_array vxCreateVirtualArray (vx_graph graph, vx_enum item_type, vx_size capacity)

  Creates an opaque reference to a virtual Array with no direct user access.
- vx_status vxQueryArray (vx_array arr, vx_enum attribute, void *ptr, vx_size size)

  Queries the Array for some specific information.

- vx_status vxReleaseArray (vx_array *arr)

  Releases a reference of an Array object. The object may not be garbage collected until its total reference count is zero. After returning from this function the reference is zeroed.
- vx_status vxTruncateArray (vx_array arr, vx_size new_num_items)

  Truncates an Array (remove items from the end).

3.48.2 Macro Definition Documentation

#define vxFormatArrayPointer(*ptr, index, stride*) (&(((vx_uint8*)(ptr))[(index) * (stride)]))

Accesses a specific indexed element in an array.

Parameters

in	*ptr*	The base pointer for the array range.
in	*index*	The index of the element, not byte, to access.
in	*stride*	The stride of the array range given by vxAccessArrayRange.

Definition at line 1728 of file vx_api.h.

#define vxArrayItem(*type, ptr, index, stride*) (*(type *)(vxFormatArrayPointer((ptr), (index), (stride))))

Allows access to an array item as a typecast pointer deference.

Parameters

in	*type*	The type of the item to access.
in	*ptr*	The base pointer for the array range.
in	*index*	The index of the element, not byte, to access.
in	*stride*	The stride of the array range given by vxAccessArrayRange.

Definition at line 1739 of file vx_api.h.

3.48.3 Enumeration Type Documentation

enum vx_array_attribute_e

The array object attributes.

Enumerator

VX_ARRAY_ATTRIBUTE_ITEMTYPE The type of the Array items. Use a vx_enum parameter.

VX_ARRAY_ATTRIBUTE_NUMITEMS The number of items in the Array. Use a vx_size parameter.

VX_ARRAY_ATTRIBUTE_CAPACITY The maximal number of items that the Array can hold. Use a vx_size parameter.

VX_ARRAY_ATTRIBUTE_ITEMSIZE Queries an array item size. Use a vx_size parameter.

Definition at line 928 of file vx_types.h.

3.48.4 Function Documentation

vx_array vxCreateArray (vx_context *context,* vx_enum *item_type,* vx_size *capacity*)

Creates a reference to an Array object.

User must specify the Array capacity (i.e., the maximal number of items that the array can hold).

Parameters

in	*context*	The reference to the overall Context.
in	*item_type*	The type of objects to hold. Use: • VX_TYPE_RECTANGLE for vx_rectangle_t. • VX_TYPE_KEYPOINT for vx_keypoint_t. • VX_TYPE_COORDINATES2D for vx_coordinates2d_t. • VX_TYPE_COORDINATES3D for vx_coordinates3d_t. • vx_enum Returned from vxRegisterUserStruct.
in	*capacity*	The maximal number of items that the array can hold.

Returns

vx_array.

Return values

0	No Array was created.
*	An Array was created.

vx_array vxCreateVirtualArray (vx_graph *graph*, vx_enum *item_type*, vx_size *capacity*)

Creates an opaque reference to a virtual Array with no direct user access.

Virtual Arrays are useful when item type or capacity are unknown ahead of time and the Array is used as internal graph edge. Virtual arrays are scoped within the parent graph only.

All of the following constructions are allowed.

```
vx_context context = vxCreateContext();
vx_graph graph = vxCreateGraph(context);
vx_array virt[] = {
    vxCreateVirtualArray(graph, 0, 0), // totally unspecified
    vxCreateVirtualArray(graph, VX_TYPE_KEYPOINT, 0), // unspecified
       capacity
    vxCreateVirtualArray(graph, VX_TYPE_KEYPOINT, 1000), // no access
};
```

Parameters

in	*graph*	The reference to the parent graph.
in	*item_type*	The type of objects to hold. This may to set to zero to indicate an unspecified item type.
in	*capacity*	The maximal number of items that the array can hold. This may be to set to zero to indicate an unspecified capacity.

See also

vxCreateArray for a type list.

Returns

vx_array.

Return values

0	No Array was created.

*	An Array was created or an error occurred. Use vxGetStatus to determine.

vx_status vxReleaseArray (vx_array * *arr*)

Releases a reference of an Array object. The object may not be garbage collected until its total reference count is zero. After returning from this function the reference is zeroed.

Parameters

in	*arr*	The pointer to the Array to release.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxQueryArray (vx_array *arr*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries the Array for some specific information.

Parameters

in	*arr*	The reference to the Array.
in	*attribute*	The attribute to query. Use a vx_array_attribute_e.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the *arr* is not a vx_array.
VX_ERROR_NOT_SUPPORTED	If the *attribute* is not a value supported on this implementation.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

vx_status vxAddArrayItems (vx_array *arr*, vx_size *count*, void * *ptr*, vx_size *stride*)

Adds items to the Array.

This function increases the container size.

By default, the function does not reallocate memory, so if the container is already full (number of elements is equal to capacity) or it doesn't have enough space, the function returns VX_FAILURE error code.

Parameters

in	*arr*	The reference to the Array.
in	*count*	The total number of elements to insert.

in	*ptr*	The location at which to store the input values.
in	*stride*	The stride in bytes between elements. User can pass 0, which means that stride is equal to item size.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the *arr* is not a vx_array.
VX_FAILURE	If the Array is full.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

vx_status vxTruncateArray (vx_array *arr*, vx_size *new_num_items*)

Truncates an Array (remove items from the end).

Parameters

in,out	*arr*	The reference to the Array.
in	*new_num_items*	The new number of items for the Array.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the *arr* is not a vx_array.
VX_ERROR_INVALID_PARAMETERS	The *new_size* is greater than the current size.

vx_status vxAccessArrayRange (vx_array *arr*, vx_size *start*, vx_size *end*, vx_size * *stride*, void ** *ptr*, vx_enum *usage*)

Grants access to a sub-range of an Array.

Parameters

in	*arr*	The reference to the Array.
in	*start*	The start index.
in	*end*	The end index.
out	*stride*	The stride in bytes between elements.
out	*ptr*	The user-supplied pointer to a pointer, via which the requested contents are returned. If (*ptr) is non-NULL, data is copied to it, else (*ptr) is set to the address of existing internal memory, allocated, or mapped memory. (*ptr) must be given to vxCommitArrayRange. Use a vx_rectangle_t for VX_TYPE_RECTANGLE and a vx_keypoint_t for VX_TYPE_KEYPOINT.

in	*usage*	This declares the intended usage of the pointer using the vx_accessor_e enumeration.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_OPTIMIZED_AWAY	If the reference is a virtual array and cannot be accessed or committed.
VX_ERROR_INVALID_REFERENCE	If the *arr* is not a vx_array.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

Postcondition

vxCommitArrayRange

vx_status vxCommitArrayRange (vx_array *arr*, vx_size *start*, vx_size *end*, void * *ptr*)

Commits data back to the Array object.

This allows a user to commit data to a sub-range of an Array.

Parameters

in	*arr*	The reference to the Array.
in	*start*	The start index.
in	*end*	The end index.
in	*ptr*	The user supplied pointer.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_OPTIMIZED_AWAY	If the reference is a virtual array and cannot be accessed or committed.
VX_ERROR_INVALID_REFERENCE	If the *arr* is not a vx_array.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

3.49 Object: Convolution

3.49.1 Detailed Description

Defines the Image Convolution Object interface.

Typedefs

- typedef struct _vx_convolution * vx_convolution

  The Convolution Object. A user-defined convolution kernel of MxM elements.

Enumerations

- enum vx_convolution_attribute_e {
  VX_CONVOLUTION_ATTRIBUTE_ROWS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONVOLUTION << 8)) + 0x0,
  VX_CONVOLUTION_ATTRIBUTE_COLUMNS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONVOL↩ UTION << 8)) + 0x1,
  VX_CONVOLUTION_ATTRIBUTE_SCALE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONVOLUTI↩ ON << 8)) + 0x2,
  VX_CONVOLUTION_ATTRIBUTE_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_CONVOLUTION << 8)) + 0x3 }

  The convolution attributes.

Functions

- vx_status vxAccessConvolutionCoefficients (vx_convolution conv, vx_int16 *array)

  Gets the convolution data (copy).
- vx_status vxCommitConvolutionCoefficients (vx_convolution conv, vx_int16 *array)

  Sets the convolution data (copy),.
- vx_convolution vxCreateConvolution (vx_context context, vx_size columns, vx_size rows)

  Creates a reference to a convolution matrix object.
- vx_status vxQueryConvolution (vx_convolution conv, vx_enum attribute, void *ptr, vx_size size)

  Queries an attribute on the convolution matrix object.
- vx_status vxReleaseConvolution (vx_convolution *conv)

  Releases the reference to a convolution matrix. The object may not be garbage collected until its total reference count is zero.
- vx_status vxSetConvolutionAttribute (vx_convolution conv, vx_enum attribute, void *ptr, vx_size size)

  Sets attributes on the convolution object.

3.49.2 Enumeration Type Documentation

enum vx_convolution_attribute_e

The convolution attributes.

Enumerator

VX_CONVOLUTION_ATTRIBUTE_ROWS The number of rows of the convolution matrix. Use a `vx_size` parameter.

VX_CONVOLUTION_ATTRIBUTE_COLUMNS The number of columns of the convolution matrix. Use a `vx_size` parameter.

VX_CONVOLUTION_ATTRIBUTE_SCALE The scale of the convolution matrix. Use a `vx_uint32` parameter.

Note

For 1.0, only powers of 2 are supported up to 2^31.

VX_CONVOLUTION_ATTRIBUTE_SIZE The total size of the convolution matrix in bytes. Use a vx_size parameter.

Definition at line 880 of file vx_types.h.

3.49.3 Function Documentation

vx_convolution vxCreateConvolution (vx_context *context*, vx_size *columns*, vx_size *rows*)

Creates a reference to a convolution matrix object.

Parameters

in	*context*	The reference to the overall context.
in	*columns*	The columns dimension of the convolution. Must be odd and greater than or equal to 3 and less than the value returned from VX_CONTEXT_ATTRIBUTE_CONVOLUTION_MAXIMUM_DIMENSION.
in	*rows*	The rows dimension of the convolution. Must be odd and greater than or equal to 3 and less than the value returned from VX_CONTEXT_ATTRIBUTE_CONVOLUTION_MAXIMUM_DIMENSION.

Returns

vx_convolution

vx_status vxReleaseConvolution (vx_convolution * *conv*)

Releases the reference to a convolution matrix. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*conv*	The pointer to the convolution matrix to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxQueryConvolution (vx_convolution *conv*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries an attribute on the convolution matrix object.

Parameters

in	*conv*	The convolution matrix object to set.
in	*attribute*	The attribute to query. Use a vx_convolution_attribute_e enumeration.

out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_status vxSetConvolutionAttribute (vx_convolution *conv*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Sets attributes on the convolution object.

Parameters

in	*conv*	The coordinates object to set.
in	*attribute*	The attribute to modify. Use a vx_convolution_attribute_e enumeration.
in	*ptr*	The pointer to the value to which to set the attribute.
in	*size*	The size of the data pointed to by *ptr*.

Returns

A vx_status_e enumeration.

vx_status vxAccessConvolutionCoefficients (vx_convolution *conv*, vx_int16 * *array*)

Gets the convolution data (copy).

Parameters

in	*conv*	The reference to the convolution.
out	*array*	The array to place the convolution.

See also

vxQueryConvolution and VX_CONVOLUTION_ATTRIBUTE_SIZE to get the needed number of bytes of the array.

Returns

A vx_status_e enumeration.

Postcondition

vxCommitConvolutionCoefficients

vx_status vxCommitConvolutionCoefficients (vx_convolution *conv*, vx_int16 * *array*)

Sets the convolution data (copy),.

Parameters

in	*conv*	The reference to the convolution.
out	*array*	The array to read the convolution.

See also

vxQueryConvolution and VX_CONVOLUTION_ATTRIBUTE_SIZE to get the needed number of bytes of the array.

Returns

A vx_status_e enumeration.

Precondition

vxAccessConvolutionCoefficients

3.50 Object: Distribution

3.50.1 Detailed Description

Defines the Distribution Object Interface.

Typedefs

- typedef struct _vx_distribution * vx_distribution

  The Distribution object. This has a user-defined number of bins over a user-defined range (within a uint32_t range).

Enumerations

- enum vx_distribution_attribute_e {
  VX_DISTRIBUTION_ATTRIBUTE_DIMENSIONS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x0,
  VX_DISTRIBUTION_ATTRIBUTE_OFFSET = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x1,
  VX_DISTRIBUTION_ATTRIBUTE_RANGE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x2,
  VX_DISTRIBUTION_ATTRIBUTE_BINS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x3,
  VX_DISTRIBUTION_ATTRIBUTE_WINDOW = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x4,
  VX_DISTRIBUTION_ATTRIBUTE_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DISTRIBUTION << 8)) + 0x5 }

  The distribution attribute list.

Functions

- vx_status vxAccessDistribution (vx_distribution distribution, void **ptr, vx_enum usage)

  Gets direct access to a Distribution in memory.
- vx_status vxCommitDistribution (vx_distribution distribution, void *ptr)

  Sets the Distribution back to the memory. The memory must be a vx_uint32 array of a value at least as big as the value returned via VX_DISTRIBUTION_ATTRIBUTE_RANGE.
- vx_distribution vxCreateDistribution (vx_context context, vx_size numBins, vx_size offset, vx_size range)

  Creates a reference to a 1D Distribution with a start offset, valid range, and number of equally weighted bins.
- vx_status vxQueryDistribution (vx_distribution distribution, vx_enum attribute, void *ptr, vx_size size)

  Queries a Distribution object.
- vx_status vxReleaseDistribution (vx_distribution *distribution)

  Releases a reference to a distribution object. The object may not be garbage collected until its total reference count is zero.

3.50.2 Enumeration Type Documentation

enum vx_distribution_attribute_e

The distribution attribute list.

Enumerator

VX_DISTRIBUTION_ATTRIBUTE_DIMENSIONS Indicates the number of dimensions in the distribution. Use a `vx_size` parameter.

VX_DISTRIBUTION_ATTRIBUTE_OFFSET Indicates the start of the values to use (inclusive). Use a `vx_size` parameter.

VX_DISTRIBUTION_ATTRIBUTE_RANGE Indicates end value to use as the range (exclusive). Use a `vx_size` parameter.

VX_DISTRIBUTION_ATTRIBUTE_BINS Indicates the number of bins. Use a vx_size parameter.

VX_DISTRIBUTION_ATTRIBUTE_WINDOW Indicates the range of a bin. Use a vx_uint32 parameter.

VX_DISTRIBUTION_ATTRIBUTE_SIZE Indicates the total size of the distribution in bytes. Use a vx_size parameter.

Definition at line 820 of file vx_types.h.

3.50.3 Function Documentation

vx_distribution vxCreateDistribution (vx_context *context*, vx_size *numBins*, vx_size *offset*, vx_size *range*)

Creates a reference to a 1D Distribution with a start offset, valid range, and number of equally weighted bins.

Parameters

in	*context*	The reference to the overall context.
in	*numBins*	The number of bins in the distribution.
in	*offset*	The offset into the range value.
in	*range*	The total range of the values.

Returns

vx_distribution

vx_status vxReleaseDistribution (vx_distribution * *distribution*)

Releases a reference to a distribution object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*distribution*	The reference to the distribution to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxQueryDistribution (vx_distribution *distribution*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries a Distribution object.

Parameters

in	*distribution*	The reference to the distribution to query.
in	*attribute*	The attribute to query. Use a vx_distribution_attribute_e enumeration.

out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_status vxAccessDistribution (vx_distribution *distribution,* void ** *ptr,* vx_enum *usage*)

Gets direct access to a Distribution in memory.

Parameters

in	*distribution*	The reference to the distribution to access.
out	*ptr*	The address of the location to store the pointer to the Distribution memory. • If (*ptr) is not NULL, the Distribution will be copied to that address. • If (*ptr) is NULL, the pointer will be allocated, mapped, or use internal memory. In any case, vxCommitDistribution must be called with (*ptr).
in	*usage*	The vx_accessor_e value to describe the access of the object.

Returns

A vx_status_e enumeration.

Postcondition

vxCommitDistribution

vx_status vxCommitDistribution (vx_distribution *distribution,* void * *ptr*)

Sets the Distribution back to the memory. The memory must be a vx_uint32 array of a value at least as big as the value returned via VX_DISTRIBUTION_ATTRIBUTE_RANGE.

Parameters

in	*distribution*	The Distribution to modify.
in	*ptr*	The pointer returned from (or not modified by) vxAccessDistribution.

Returns

A vx_status_e enumeration.

Precondition

vxAccessDistribution.

3.51 Object: Image

3.51.1 Detailed Description

Defines the Image Object interface.

Data Structures

- struct vx_imagepatch_addressing_t

  The addressing image patch structure is used by the Host only to address pixels in an image patch. The fields of the structure are defined as: More...

Macros

- #define VX_IMAGEPATCH_ADDR_INIT {0u, 0u, 0, 0, 0u, 0u, 0u, 0u}

  Use to initialize a `vx_imagepatch_addressing_t` structure on the stack.

Typedefs

- typedef struct _vx_image * vx_image

  An opaque reference to an image.

Enumerations

- enum vx_channel_range_e {
  VX_CHANNEL_RANGE_FULL = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_RANGE << 12)) + 0x0,
  VX_CHANNEL_RANGE_RESTRICTED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_RANGE << 12)) + 0x1 }

  The image channel range list used by the `VX_IMAGE_ATTRIBUTE_RANGE` attribute of a `vx_image`.
- enum vx_color_space_e {
  VX_COLOR_SPACE_NONE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_SPACE << 12)) + 0x0,
  VX_COLOR_SPACE_BT601_525 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_SPACE << 12)) + 0x1,
  VX_COLOR_SPACE_BT601_625 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_SPACE << 12)) + 0x2,
  VX_COLOR_SPACE_BT709 = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_COLOR_SPACE << 12)) + 0x3,
  VX_COLOR_SPACE_DEFAULT = VX_COLOR_SPACE_BT709 }

  The image color space list used by the `VX_IMAGE_ATTRIBUTE_SPACE` attribute of a `vx_image`.
- enum vx_image_attribute_e {
  VX_IMAGE_ATTRIBUTE_WIDTH = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x0,
  VX_IMAGE_ATTRIBUTE_HEIGHT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x1,
  VX_IMAGE_ATTRIBUTE_FORMAT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x2,
  VX_IMAGE_ATTRIBUTE_PLANES = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x3,
  VX_IMAGE_ATTRIBUTE_SPACE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x4,
  VX_IMAGE_ATTRIBUTE_RANGE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x5,
  VX_IMAGE_ATTRIBUTE_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_IMAGE << 8)) + 0x6 }

  The image attributes list.

Functions

- vx_status vxAccessImagePatch (vx_image image, vx_rectangle_t *rect, vx_uint32 plane_index, vx_imagepatch_addressing_t *addr, void **ptr, vx_enum usage)

  Allows the User to extract a rectangular patch (subset) of an image from a single plane.

- vx_status vxCommitImagePatch (vx_image image, vx_rectangle_t *rect, vx_uint32 plane_index, vx_imagepatch_addressing_t *addr, void *ptr)

  This allows the User to commit a rectangular patch (subset) of an image from a single plane.
- vx_size vxComputeImagePatchSize (vx_image image, vx_rectangle_t *rect, vx_uint32 plane_index)

  This computes the size needed to retrieve an image patch from an image.
- vx_image vxCreateImage (vx_context context, vx_uint32 width, vx_uint32 height, vx_df_image color)

  Creates an opaque reference to an image buffer.
- vx_image vxCreateImageFromHandle (vx_context context, vx_df_image color, vx_imagepatch_addressing_t addrs[], void *ptrs[], vx_enum import_type)

  Creates a reference to an image object that was externally allocated.
- vx_image vxCreateImageFromROI (vx_image img, vx_rectangle_t *rect)

  Creates an image from another image given a rectangle. This second reference refers to the data in the original image. Updates to this image updates the parent image. The rectangle must be defined within the pixel space of the parent image.
- vx_image vxCreateUniformImage (vx_context context, vx_uint32 width, vx_uint32 height, vx_df_image color, void *value)

  Creates a reference to an image object that has a singular, uniform value in all pixels.
- vx_image vxCreateVirtualImage (vx_graph graph, vx_uint32 width, vx_uint32 height, vx_df_image color)

  Creates an opaque reference to an image buffer with no direct user access. This function allows setting the image width, height, or format.
- void * vxFormatImagePatchAddress1d (void *ptr, vx_uint32 index, vx_imagepatch_addressing_t *addr)

  Accesses a specific indexed pixel in an image patch.
- void * vxFormatImagePatchAddress2d (void *ptr, vx_uint32 x, vx_uint32 y, vx_imagepatch_addressing_t *addr)

  Accesses a specific pixel at a 2d coordinate in an image patch.
- vx_status vxGetValidRegionImage (vx_image image, vx_rectangle_t *rect)

  Retrieves the valid region of the image as a rectangle.
- vx_status vxQueryImage (vx_image image, vx_enum attribute, void *ptr, vx_size size)

  Retrieves various attributes of an image.
- vx_status vxReleaseImage (vx_image *image)

  Releases a reference to an image object. The object may not be garbage collected until its total reference count is zero.
- vx_status vxSetImageAttribute (vx_image image, vx_enum attribute, void *out, vx_size size)

  Allows setting attributes on the image.

3.51.2 Data Structure Documentation

struct vx_imagepatch_addressing_t

The addressing image patch structure is used by the Host only to address pixels in an image patch. The fields of the structure are defined as:

- dim - The dimensions of the image in logical pixel units in the x & y direction.
- stride - The physical byte distance from a logical pixel to the next logically adjacent pixel in the positive x or y direction.
- scale - The relationship of scaling from the primary plane (typically the zero indexed plane) to this plane. An integer down-scaling factor of f shall be set to a value equal to $scale = \frac{unity}{f}$ and an integer up-scaling factor of f shall be set to a value of $scale = unity * f$. $unity$ is defined as VX_SCALE_UNITY.
- step - The step is the number of logical pixel units to skip to arrive at the next physically unique pixel. For example, on a plane that is half-scaled in a dimension, the step in that dimension is 2 to indicate that every other pixel in that dimension is an alias. This is useful in situations where iteration over unique pixels is required, such as in serializing or de-serializing the image patch information.

See also

vxAccessImagePatch

```
/*
 * Copyright (c) 2013-2014 The Khronos Group Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and/or associated documentation files (the
 * "Materials"), to deal in the Materials without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Materials, and to
 * permit persons to whom the Materials are furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in all copies or substantial portions of the Materials.
 *
 * THE MATERIALS ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 * IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 * CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 * TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 * MATERIALS OR THE USE OR OTHER DEALINGS IN THE MATERIALS.
 */
#include <VX/vx.h>

#define PATCH_DIM 16

vx_status example_imagepatch(vx_context context)
{
    vx_status status = VX_SUCCESS;
    void *base_ptr = NULL;
    vx_uint32 width = 640, height = 480, plane = 0;
    vx_image image = vxCreateImage(context, width, height,
      VX_DF_IMAGE_U8);
    vx_rectangle_t rect;
    vx_imagepatch_addressing_t addr;

    rect.start_x = rect.start_y = 0;
    rect.end_x = rect.end_y = PATCH_DIM;

    status = vxAccessImagePatch(image, &rect, plane,
                                &addr, &base_ptr,
                                VX_READ_AND_WRITE);
    if (status == VX_SUCCESS)
    {
        vx_uint32 x,y,i,j;
        vx_uint8 pixel = 0;

        /* a couple addressing options */

        /* use linear addressing function/macro */
        for (i = 0; i < addr.dim_x*addr.dim_y; i++) {
            vx_uint8 *ptr2 = vxFormatImagePatchAddress1d(base_ptr,
                                                         i, &addr);
            *ptr2 = pixel;
        }

        /* 2d addressing option */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *ptr2 = vxFormatImagePatchAddress2d(base_ptr,
                                                             x, y, &addr);
                *ptr2 = pixel;
            }
        }

        /* direct addressing by client
         * for subsampled planes, scale will change
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = ((addr.stride_y*y*addr.scale_y) /
                      VX_SCALE_UNITY) +
                    ((addr.stride_x*x*addr.scale_x) /
                      VX_SCALE_UNITY);
                tmp[i] = pixel;
            }
        }

        /* more efficient direct addressing by client.
         * for subsampled planes, scale will change.
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
```

```
            j = (addr.stride_y*y*addr.scale_y)/VX_SCALE_UNITY;
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = j + (addr.stride_x*x*addr.scale_x) /
                    VX_SCALE_UNITY;
                tmp[i] = pixel;
            }
        }

        /* this commits the data back to the image. If rect were 0 or empty, it
         * would just decrement the reference (used when reading an image only)
         */
        status = vxCommitImagePatch(image, &rect, plane, &addr, base_ptr);
    }
    vxReleaseImage(&image);
    return status;
}
```

Definition at line 1273 of file vx_types.h.

Data Fields

vx_uint32	dim_x	Width of patch in X dimension in pixels.
vx_uint32	dim_y	Height of patch in Y dimension in pixels.
vx_int32	stride_x	Stride in X dimension in bytes.
vx_int32	stride_y	Stride in Y dimension in bytes.
vx_uint32	scale_x	Scale of X dimension. For sub-sampled planes this is the scaling factor of the dimension of the plane in relation to the zero plane. Use VX_SC↩ ALE_UNITY in the numerator.
vx_uint32	scale_y	Scale of Y dimension. For sub-sampled planes this is the scaling factor of the dimension of the plane in relation to the zero plane. Use VX_SC↩ ALE_UNITY in the numerator.
vx_uint32	step_x	Step of X dimension in pixels.
vx_uint32	step_y	Step of Y dimension in pixels.

3.51.3 Typedef Documentation

typedef struct _vx_image* vx_image

An opaque reference to an image.

See also

vxCreateImage

Definition at line 144 of file vx_types.h.

3.51.4 Enumeration Type Documentation

enum vx_image_attribute_e

The image attributes list.

Enumerator

VX_IMAGE_ATTRIBUTE_WIDTH Queries an image for its height. Use a vx_uint32 parameter.

VX_IMAGE_ATTRIBUTE_HEIGHT Queries an image for its width. Use a vx_uint32 parameter.

VX_IMAGE_ATTRIBUTE_FORMAT Queries an image for its format. Use a vx_df_image parameter.

VX_IMAGE_ATTRIBUTE_PLANES Queries an image for its number of planes. Use a vx_size parameter.

VX_IMAGE_ATTRIBUTE_SPACE Queries an image for its color space (see vx_color_space_e). Use a vx_enum parameter.

VX_IMAGE_ATTRIBUTE_RANGE Queries an image for its channel range (see vx_channel_range_e). Use a vx_enum parameter.

VX_IMAGE_ATTRIBUTE_SIZE Queries an image for its total number of bytes. Use a vx_size parameter.

Definition at line 766 of file vx_types.h.

enum vx_color_space_e

The image color space list used by the VX_IMAGE_ATTRIBUTE_SPACE attribute of a vx_image.

Enumerator

VX_COLOR_SPACE_NONE Use to indicate that no color space is used.

VX_COLOR_SPACE_BT601_525 Use to indicate that the BT.601 coefficients and SMPTE C primaries are used for conversions.

VX_COLOR_SPACE_BT601_625 Use to indicate that the BT.601 coefficients and BTU primaries are used for conversions.

VX_COLOR_SPACE_BT709 Use to indicate that the BT.709 coefficients are used for conversions.

VX_COLOR_SPACE_DEFAULT All images in VX are by default BT.709.

Definition at line 1027 of file vx_types.h.

enum vx_channel_range_e

The image channel range list used by the VX_IMAGE_ATTRIBUTE_RANGE attribute of a vx_image.

Enumerator

VX_CHANNEL_RANGE_FULL Full range of the unit of the channel.

VX_CHANNEL_RANGE_RESTRICTED Restricted range of the unit of the channel based on the space given.

Definition at line 1044 of file vx_types.h.

3.51.5 Function Documentation

vx_image vxCreateImage (vx_context *context*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *color*)

Creates an opaque reference to an image buffer.
Not guaranteed to exist until the vx_graph containing it has been verified.

Parameters

in	*context*	The reference to the implementation context.
in	*width*	The image width in pixels.
in	*height*	The image height in pixels.
in	*color*	The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space.

Returns

An image reference or zero when an error is encountered.

See also

vxAccessImagePatch to obtain direct memory access to the image data.

vx_image vxCreateImageFromROI (vx_image *img*, vx_rectangle_t * *rect*)

Creates an image from another image given a rectangle. This second reference refers to the data in the original image. Updates to this image updates the parent image. The rectangle must be defined within the pixel space of the parent image.

Parameters

in	*img*	The reference to the parent image.
in	*rect*	The region of interest rectangle. Must contain points within the parent image pixel space.

Returns

The reference to the sub-image or zero if the rectangle is invalid.

vx_image vxCreateUniformImage (vx_context *context,* vx_uint32 *width,* vx_uint32 *height,* vx_df_image *color,* void * *value*)

Creates a reference to an image object that has a singular, uniform value in all pixels.

The value pointer must reflect the specific format of the desired image. For example:

Color	Value Ptr
VX_DF_IMAGE_U8	vx_uint8 *
VX_DF_IMAGE_S16	vx_int16 *
VX_DF_IMAGE_U16	vx_uint16 *
VX_DF_IMAGE_S32	vx_int32 *
VX_DF_IMAGE_U32	vx_uint32 *
VX_DF_IMAGE_RGB	vx_uint8 pixel[3] in R, G, B order
VX_DF_IMAGE_RGBX	vx_uint8 pixels[4]
Any YUV	vx_uint8 pixel[3] in Y, U, V order

Parameters

in	*context*	The reference to the implementation context.
in	*width*	The image width in pixels.
in	*height*	The image height in pixels.
in	*color*	The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space.
in	*value*	The pointer to the pixel value to which to set all pixels.

Returns

An image reference or zero when an error is encountered.

See also

vxAccessImagePatch to obtain direct memory access to the image data.

Note

vxAccessImagePatch and vxCommitImagePatch may be called with a uniform image reference.

vx_image vxCreateVirtualImage (vx_graph *graph,* vx_uint32 *width,* vx_uint32 *height,* vx_df_image *color*)

Creates an opaque reference to an image buffer with no direct user access. This function allows setting the image width, height, or format.

Virtual data objects allow users to connect various nodes within a graph via data references without access to that data, but they also permit the implementation to take maximum advantage of possible optimizations. Use this API to create a data reference to link two or more nodes together when the intermediate data are not required to be accessed by outside entities. This API in particular allows the user to define the image format of the data without requiring the exact dimensions. Virtual objects are scoped within the graph they are declared a part of, and can't be shared outside of this scope. All of the following constructions of virtual images are valid.

```
vx_context context = vxCreateContext();
vx_graph graph = vxCreateGraph(context);
vx_image virt[] = {
    vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_U8), // no specified
       dimension
    vxCreateVirtualImage(graph, 320, 240, VX_DF_IMAGE_VIRT), // no
       specified format
    vxCreateVirtualImage(graph, 640, 480, VX_DF_IMAGE_U8), // no user
       access
};
```

Parameters

in	*graph*	The reference to the parent graph.
in	*width*	The width of the image in pixels. A value of zero informs the interface that the value is unspecified.
in	*height*	The height of the image in pixels. A value of zero informs the interface that the value is unspecified.
in	*color*	The VX_DF_IMAGE (vx_df_image_e) code that represents the format of the image and the color space. A value of VX_DF_IMAGE_VIRT informs the interface that the format is unspecified.

Returns

An image reference or zero when an error is encountered.

Note

Passing this reference to vxAccessImagePatch will return an error.

vx_image vxCreateImageFromHandle (vx_context *context*, vx_df_image *color*, vx_imagepatch_addressing_t *addrs[]*, void * *ptrs[]*, vx_enum *import_type*)

Creates a reference to an image object that was externally allocated.

Parameters

in	*context*	The reference to the implementation context.
in	*color*	See the vx_df_image_e codes. This mandates the number of planes needed to be valid in the *addrs* and *ptrs* arrays based on the format given.
in	*addrs[]*	The array of image patch addressing structures that define the dimension and stride of the array of pointers.
in	*ptrs[]*	The array of platform-defined references to each plane.
in	*import_type*	vx_import_type_e. When giving VX_IMPORT_TYPE_HOST the *ptrs* array is assumed to be HOST accessible pointers to memory.

Returns

vx_image.

Return values

0	Image could not be created.
*	Valid Image reference.

vx_status vxQueryImage (vx_image *image*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Retrieves various attributes of an image.

Parameters

in	*image*	The reference to the image to query.
in	*attribute*	The attribute to query. Use a vx_image_attribute_e.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the image is not a `vx_image`.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.
VX_ERROR_NOT_SUPPORTED	If the attribute is not supported on this implementation.

vx_status vxSetImageAttribute (vx_image *image*, vx_enum *attribute*, void * *out*, vx_size *size*)

Allows setting attributes on the image.

Parameters

in	*image*	The reference to the image on which to set the attribute.
in	*attribute*	The attribute to set. Use a `vx_image_attribute_e` enumeration.
in	*out*	The pointer to the location from which to read the value.
in	*size*	The size of the object pointed to by *out*.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the image is not a `vx_image`.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

vx_status vxReleaseImage (vx_image * *image*)

Releases a reference to an image object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*image*	The pointer to the image to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a `vx_graph`.

vx_size vxComputeImagePatchSize (vx_image *image*, vx_rectangle_t * *rect*, vx_uint32 *plane_index*)

This computes the size needed to retrieve an image patch from an image.

Parameters

in	*image*	The reference to the image from which to extract the patch.
in	*rect*	The coordinates. Must be 0 <= start < end <= dimension where dimension is width for x and height for y.
in	*plane_index*	The plane index from which to get the data.

Returns

vx_size

vx_status vxAccessImagePatch (vx_image *image*, vx_rectangle_t * *rect*, vx_uint32 *plane_index*, vx_imagepatch_addressing_t * *addr*, void ** *ptr*, vx_enum *usage*)

Allows the User to extract a rectangular patch (subset) of an image from a single plane.

Parameters

in	*image*	The reference to the image from which to extract the patch.
in	*rect*	The coordinates from which to get the patch. Must be 0 <= start < end.
in	*plane_index*	The plane index from which to get the data.
out	*addr*	The addressing information for the image patch to be written into the data structure.
out	*ptr*	The pointer to a pointer of a location to store the data. • If the user passes in a NULL, an error occurs. • If the user passes in a pointer to a NULL, the function returns internal memory, map, or allocates a buffer and returns it. • If the user passes in a pointer to a non-NULL pointer, the function attempts to copy to the location provided by the user. (*ptr) must be given to vxCommitImagePatch.
in	*usage*	This declares the intended usage of the pointer using the vx_accessor_e enumeration.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_OPTIMIZED_AWAY	The reference is a virtual image and cannot be accessed or committed.
VX_ERROR_INVALID_PARAMETERS	The *start*, *end*, *plane_index*, *stride_x*, or *stride_y* pointer is incorrect.
VX_ERROR_INVALID_REFERENCE	The image reference is not actually an image reference.

Note

The user may ask for data outside the bounds of the valid region, but such data has an undefined value. Users must be cautious to prevent passing in *uninitialized* pointers or addresses of uninitialized pointers to this function.

Precondition

vxComputeImagePatchSize if users wish to allocate their own memory.

Postcondition

vxCommitImagePatch with same (*ptr) value.

```
/*
 * Copyright (c) 2013-2014 The Khronos Group Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and/or associated documentation files (the
 * "Materials"), to deal in the Materials without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Materials, and to
 * permit persons to whom the Materials are furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in all copies or substantial portions of the Materials.
 *
 * THE MATERIALS ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 * IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 * CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 * TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 * MATERIALS OR THE USE OR OTHER DEALINGS IN THE MATERIALS.
 */
#include <VX/vx.h>

#define PATCH_DIM 16

vx_status example_imagepatch(vx_context context)
{
    vx_status status = VX_SUCCESS;
    void *base_ptr = NULL;
    vx_uint32 width = 640, height = 480, plane = 0;
    vx_image image = vxCreateImage(context, width, height,
      VX_DF_IMAGE_U8);
    vx_rectangle_t rect;
    vx_imagepatch_addressing_t addr;

    rect.start_x = rect.start_y = 0;
    rect.end_x = rect.end_y = PATCH_DIM;

    status = vxAccessImagePatch(image, &rect, plane,
                                &addr, &base_ptr,
                                VX_READ_AND_WRITE);
    if (status == VX_SUCCESS)
    {
        vx_uint32 x,y,i,j;
        vx_uint8 pixel = 0;

        /* a couple addressing options */

        /* use linear addressing function/macro */
        for (i = 0; i < addr.dim_x*addr.dim_y; i++) {
            vx_uint8 *ptr2 = vxFormatImagePatchAddress1d(base_ptr,
                                                         i, &addr);
            *ptr2 = pixel;
        }

        /* 2d addressing option */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *ptr2 = vxFormatImagePatchAddress2d(base_ptr,
                                                             x, y, &addr);
                *ptr2 = pixel;
            }
        }

        /* direct addressing by client
         * for subsampled planes, scale will change
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = ((addr.stride_y*y*addr.scale_y) /
                      VX_SCALE_UNITY) +
                    ((addr.stride_x*x*addr.scale_x) /
                      VX_SCALE_UNITY);
                tmp[i] = pixel;
            }
        }

        /* more efficient direct addressing by client.
         * for subsampled planes, scale will change.
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            j = (addr.stride_y*y*addr.scale_y)/VX_SCALE_UNITY;
```

```
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = j + (addr.stride_x*x*addr.scale_x) /
                    VX_SCALE_UNITY;
                tmp[i] = pixel;
            }
        }

        /* this commits the data back to the image. If rect were 0 or empty, it
         * would just decrement the reference (used when reading an image only)
         */
        status = vxCommitImagePatch(image, &rect, plane, &addr, base_ptr);
    }
    vxReleaseImage(&image);
    return status;
}
```

vx_status vxCommitImagePatch (vx_image *image*, vx_rectangle_t * *rect*, vx_uint32 *plane_index*, vx_imagepatch_addressing_t * *addr*, void * *ptr*)

This allows the User to commit a rectangular patch (subset) of an image from a single plane.

Parameters

in	*image*	The reference to the image from which to extract the patch.
in	*rect*	The coordinates to which to set the patch. Must be 0 <= start <= end. This may be 0 or a rectangle of zero area in order to indicate that the commit must only decrement the reference count.
in	*plane_index*	The plane index to which to set the data.
in	*addr*	The addressing information for the image patch.
in	*ptr*	The pointer of a location from which to read the data. If the user allocated the pointer they must free it. If the pointer was set by vxAccessImagePatch, the user may not access the pointer after this call completes.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_OPTIMIZED_AWAY	The reference is a virtual image and cannot be accessed or committed.
VX_ERROR_INVALID_PARAMETERS	The *start*, *end*, *plane_index*, *stride_x*, or *stride_y* pointer is incorrect.
VX_ERROR_INVALID_REFERENCE	The image reference is not actually an image reference.

```
/*
 * Copyright (c) 2013-2014 The Khronos Group Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and/or associated documentation files (the
 * "Materials"), to deal in the Materials without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Materials, and to
 * permit persons to whom the Materials are furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in all copies or substantial portions of the Materials.
 *
 * THE MATERIALS ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 * IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 * CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 * TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 * MATERIALS OR THE USE OR OTHER DEALINGS IN THE MATERIALS.
 */
#include <VX/vx.h>

#define PATCH_DIM 16
```

```
vx_status example_imagepatch(vx_context context)
{
    vx_status status = VX_SUCCESS;
    void *base_ptr = NULL;
    vx_uint32 width = 640, height = 480, plane = 0;
    vx_image image = vxCreateImage(context, width, height,
      VX_DF_IMAGE_U8);
    vx_rectangle_t rect;
    vx_imagepatch_addressing_t addr;

    rect.start_x = rect.start_y = 0;
    rect.end_x = rect.end_y = PATCH_DIM;

    status = vxAccessImagePatch(image, &rect, plane,
                                &addr, &base_ptr,
                                VX_READ_AND_WRITE);
    if (status == VX_SUCCESS)
    {
        vx_uint32 x,y,i,j;
        vx_uint8 pixel = 0;

        /* a couple addressing options */

        /* use linear addressing function/macro */
        for (i = 0; i < addr.dim_x*addr.dim_y; i++) {
            vx_uint8 *ptr2 = vxFormatImagePatchAddress1d(base_ptr,
                                                         i, &addr);
            *ptr2 = pixel;
        }

        /* 2d addressing option */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *ptr2 = vxFormatImagePatchAddress2d(base_ptr,
                                                             x, y, &addr);
                *ptr2 = pixel;
            }
        }

        /* direct addressing by client
         * for subsampled planes, scale will change
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = ((addr.stride_y*y*addr.scale_y) /
                      VX_SCALE_UNITY) +
                    ((addr.stride_x*x*addr.scale_x) /
                      VX_SCALE_UNITY);
                tmp[i] = pixel;
            }
        }

        /* more efficient direct addressing by client.
         * for subsampled planes, scale will change.
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            j = (addr.stride_y*y*addr.scale_y)/VX_SCALE_UNITY;
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = j + (addr.stride_x*x*addr.scale_x) /
                    VX_SCALE_UNITY;
                tmp[i] = pixel;
            }
        }

        /* this commits the data back to the image. If rect were 0 or empty, it
         * would just decrement the reference (used when reading an image only)
         */
        status = vxCommitImagePatch(image, &rect, plane, &addr, base_ptr);
    }
    vxReleaseImage(&image);
    return status;
}
```

Note

If the implementation gives the client a pointer from `vxAccessImagePatch` then implementation-specific behavior may occur. If not, then a copy occurs from the users pointer to the internal data of the object.
If the rectangle intersects bounds of the current valid region, the valid region grows to the union of the two rectangles as long as they occur within the bounds of the original image dimensions.

void* vxFormatImagePatchAddress1d (void * *ptr,* vx_uint32 *index,* vx_imagepatch_addressing_t * *addr*)

Accesses a specific indexed pixel in an image patch.

Parameters

in	*ptr*	The base pointer of the patch as returned from vxAccessImagePatch.
in	*index*	The 0 based index of the pixel count in the patch. Indexes increase horizontally by 1 then wrap around to the next row.
in	*addr*	The pointer to the addressing mode information returned from vxAccessImagePatch.

Returns

void * Returns the pointer to the specified pixel.

Precondition

vxAccessImagePatch

```
/*
 * Copyright (c) 2013-2014 The Khronos Group Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and/or associated documentation files (the
 * "Materials"), to deal in the Materials without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Materials, and to
 * permit persons to whom the Materials are furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in all copies or substantial portions of the Materials.
 *
 * THE MATERIALS ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 * IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 * CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 * TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 * MATERIALS OR THE USE OR OTHER DEALINGS IN THE MATERIALS.
 */
#include <VX/vx.h>

#define PATCH_DIM 16

vx_status example_imagepatch(vx_context context)
{
    vx_status status = VX_SUCCESS;
    void *base_ptr = NULL;
    vx_uint32 width = 640, height = 480, plane = 0;
    vx_image image = vxCreateImage(context, width, height,
      VX_DF_IMAGE_U8);
    vx_rectangle_t rect;
    vx_imagepatch_addressing_t addr;

    rect.start_x = rect.start_y = 0;
    rect.end_x = rect.end_y = PATCH_DIM;

    status = vxAccessImagePatch(image, &rect, plane,
                                &addr, &base_ptr,
                                VX_READ_AND_WRITE);
    if (status == VX_SUCCESS)
    {
        vx_uint32 x,y,i,j;
        vx_uint8 pixel = 0;

        /* a couple addressing options */

        /* use linear addressing function/macro */
        for (i = 0; i < addr.dim_x*addr.dim_y; i++) {
            vx_uint8 *ptr2 = vxFormatImagePatchAddress1d(base_ptr,
                                                         i, &addr);
            *ptr2 = pixel;
        }

        /* 2d addressing option */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *ptr2 = vxFormatImagePatchAddress2d(base_ptr,
                                                             x, y, &addr);
                *ptr2 = pixel;
            }
        }

        /* direct addressing by client
```

```
             * for subsampled planes, scale will change
             */
            for (y = 0; y < addr.dim_y; y+=addr.step_y) {
                for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                    vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                    i = ((addr.stride_y*y*addr.scale_y) /
                          VX_SCALE_UNITY) +
                        ((addr.stride_x*x*addr.scale_x) /
                          VX_SCALE_UNITY);
                    tmp[i] = pixel;
                }
            }

            /* more efficient direct addressing by client.
             * for subsampled planes, scale will change.
             */
            for (y = 0; y < addr.dim_y; y+=addr.step_y) {
                j = (addr.stride_y*y*addr.scale_y)/VX_SCALE_UNITY;
                for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                    vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                    i = j + (addr.stride_x*x*addr.scale_x) /
                        VX_SCALE_UNITY;
                    tmp[i] = pixel;
                }
            }

            /* this commits the data back to the image. If rect were 0 or empty, it
             * would just decrement the reference (used when reading an image only)
             */
            status = vxCommitImagePatch(image, &rect, plane, &addr, base_ptr);
        }
        vxReleaseImage(&image);
        return status;
    }
```

void* vxFormatImagePatchAddress2d (void * *ptr*, vx_uint32 *x*, vx_uint32 *y*, vx_imagepatch_addressing_t * *addr*)

Accesses a specific pixel at a 2d coordinate in an image patch.

Parameters

in	*ptr*	The base pointer of the patch as returned from vxAccessImagePatch.
in	*x*	The x dimension within the patch.
in	*y*	The y dimension within the patch.
in	*addr*	The pointer to the addressing mode information returned from vxAccessImagePatch.

Returns

void * Returns the pointer to the specified pixel.

Precondition

vxAccessImagePatch

```
/*
 * Copyright (c) 2013-2014 The Khronos Group Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and/or associated documentation files (the
 * "Materials"), to deal in the Materials without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Materials, and to
 * permit persons to whom the Materials are furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in all copies or substantial portions of the Materials.
 *
 * THE MATERIALS ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 * IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 * CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 * TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 * MATERIALS OR THE USE OR OTHER DEALINGS IN THE MATERIALS.
```

```
 */
#include <VX/vx.h>

#define PATCH_DIM 16

vx_status example_imagepatch(vx_context context)
{
    vx_status status = VX_SUCCESS;
    void *base_ptr = NULL;
    vx_uint32 width = 640, height = 480, plane = 0;
    vx_image image = vxCreateImage(context, width, height,
      VX_DF_IMAGE_U8);
    vx_rectangle_t rect;
    vx_imagepatch_addressing_t addr;

    rect.start_x = rect.start_y = 0;
    rect.end_x = rect.end_y = PATCH_DIM;

    status = vxAccessImagePatch(image, &rect, plane,
                                &addr, &base_ptr,
                                VX_READ_AND_WRITE);
    if (status == VX_SUCCESS)
    {
        vx_uint32 x,y,i,j;
        vx_uint8 pixel = 0;

        /* a couple addressing options */

        /* use linear addressing function/macro */
        for (i = 0; i < addr.dim_x*addr.dim_y; i++) {
            vx_uint8 *ptr2 = vxFormatImagePatchAddress1d(base_ptr,
                                                         i, &addr);
            *ptr2 = pixel;
        }

        /* 2d addressing option */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *ptr2 = vxFormatImagePatchAddress2d(base_ptr,
                                                             x, y, &addr);
                *ptr2 = pixel;
            }
        }

        /* direct addressing by client
         * for subsampled planes, scale will change
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = ((addr.stride_y*y*addr.scale_y) /
                      VX_SCALE_UNITY) +
                    ((addr.stride_x*x*addr.scale_x) /
                      VX_SCALE_UNITY);
                tmp[i] = pixel;
            }
        }

        /* more efficient direct addressing by client.
         * for subsampled planes, scale will change.
         */
        for (y = 0; y < addr.dim_y; y+=addr.step_y) {
            j = (addr.stride_y*y*addr.scale_y)/VX_SCALE_UNITY;
            for (x = 0; x < addr.dim_x; x+=addr.step_x) {
                vx_uint8 *tmp = (vx_uint8 *)base_ptr;
                i = j + (addr.stride_x*x*addr.scale_x) /
                    VX_SCALE_UNITY;
                tmp[i] = pixel;
            }
        }

        /* this commits the data back to the image. If rect were 0 or empty, it
         * would just decrement the reference (used when reading an image only)
         */
        status = vxCommitImagePatch(image, &rect, plane, &addr, base_ptr);
    }
    vxReleaseImage(&image);
    return status;
}
```

vx_status vxGetValidRegionImage (vx_image *image,* vx_rectangle_t * *rect*)

Retrieves the valid region of the image as a rectangle.

After the image is allocated but has not been written to this returns the full rectangle of the image so that functions do not have to manage a case for uninitialized data. The image still retains an uninitialized value, but once the image is written to via any means such as vxCommitImagePatch, the valid region is altered to contain the maximum bounds of the written area.

Parameters

in	*image*	The image from which to retrieve the valid region.
out	*rect*	The destination rectangle.

Returns

vx_status

Return values

VX_ERROR_INVALID_REFERENCE	Invalid image.
VX_ERROR_INVALID_PARAMETERS	Invalid rect.
VX_STATUS	Valid image.

Note

This rectangle can be passed directly to vxAccessImagePatch to get the full valid region of the image. Modifications from vxCommitImagePatch grows the valid region.

3.52 Object: LUT

3.52.1 Detailed Description

Defines the Look-Up Table Interface.

A lookup table is an array that simplifies run-time computation by replacing computation with a simpler array indexing operation.

Typedefs

- typedef struct _vx_lut * vx_lut

  The Look-Up Table (LUT) Object.

Enumerations

- enum vx_lut_attribute_e {
  VX_LUT_ATTRIBUTE_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_LUT << 8)) + 0x0,
  VX_LUT_ATTRIBUTE_COUNT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_LUT << 8)) + 0x1,
  VX_LUT_ATTRIBUTE_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_LUT << 8)) + 0x2 }

  The Look-Up Table (LUT) attribute list.

Functions

- vx_status vxAccessLUT (vx_lut lut, void **ptr, vx_enum usage)

  Gets direct access to the LUT table data.
- vx_status vxCommitLUT (vx_lut lut, void *ptr)

  Commits the Lookup Table.
- vx_lut vxCreateLUT (vx_context context, vx_enum data_type, vx_size count)

  Creates LUT object of a given type.
- vx_status vxQueryLUT (vx_lut lut, vx_enum attribute, void *ptr, vx_size size)

  Queries attributes from a LUT.
- vx_status vxReleaseLUT (vx_lut *lut)

  Releases a reference to a LUT object. The object may not be garbage collected until its total reference count is zero.

3.52.2 Enumeration Type Documentation

enum vx_lut_attribute_e

The Look-Up Table (LUT) attribute list.

Enumerator

VX_LUT_ATTRIBUTE_TYPE Indicates the value type of the LUT. Use a `vx_enum`.

VX_LUT_ATTRIBUTE_COUNT Indicates the number of elements in the LUT. Use a `vx_size`.

VX_LUT_ATTRIBUTE_SIZE Indicates the total size of the LUT in bytes. Uses a `vx_size`.

Definition at line 808 of file vx_types.h.

3.52.3 Function Documentation

vx_lut vxCreateLUT (vx_context *context*, vx_enum *data_type*, vx_size *count*)

Creates LUT object of a given type.

Parameters

in	*context*	The reference to the context.
in	*data_type*	The type of data stored in the LUT.
in	*count*	The number of entries desired.

Note

For OpenVX 1.0, count must be equal to 256 and data_type can only be VX_TYPE_UINT8.

Returns

vx_lut

vx_status vxReleaseLUT (vx_lut * *lut*)

Releases a reference to a LUT object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*lut*	The pointer to the LUT to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxQueryLUT (vx_lut *lut*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries attributes from a LUT.

Parameters

in	*lut*	The LUT to query.
in	*attribute*	The attribute to query. Use a vx_lut_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_status vxAccessLUT (vx_lut *lut*, void ** *ptr*, vx_enum *usage*)

Gets direct access to the LUT table data.

There are several variations of call methodology:

- If *ptr* is NULL (which means the current data of the LUT is not desired), the LUT reference count is incremented.
- If *ptr* is not NULL but (*ptr) is NULL, (*ptr) will contain the address of the LUT data when the function returns and the reference count will be incremented. Whether the (*ptr) address is mapped or allocated is undefined. (*ptr) must be returned to vxCommitLUT.

- If *ptr* is not NULL and (∗ptr) is not NULL, the user is signalling the implementation to copy the LUT data into the location specified by (∗ptr). Users must use `vxQueryLUT` with `VX_LUT_ATTRIBUTE_SIZE` to determine how much memory to allocate for the LUT data.

In any case, `vxCommitLUT` must be called after LUT access is complete.

Parameters

in	*lut*	The LUT from which to get the data.
in,out	*ptr*	The address of the location to store the pointer to the LUT memory.
in	*usage*	This declares the intended usage of the pointer using the ∗ vx_accessor↩ _e enumeration.

Returns

A vx_status_e enumeration.

Postcondition

vxCommitLUT

vx_status vxCommitLUT (vx_lut *lut*, void ∗ *ptr*)

Commits the Lookup Table.

Commits the data back to the LUT object and decrements the reference count. There are several variations of call methodology:

- If a user should allocated their own memory for the LUT data copy, the user is obligated to free this memory.
- If *ptr* is not NULL and the (∗ptr) for vxAccessLUT was NULL, it is undefined whether the implementation will unmap or copy and free the memory.

Parameters

in	*lut*	The LUT to modify.
in	*ptr*	The pointer used with vxAccessLUT. This cannot be NULL.

Returns

A vx_status_e enumeration.

Precondition

vxAccessLUT.

3.53 Object: Matrix

3.53.1 Detailed Description

Defines the Matrix Object Interface.

Typedefs

- typedef struct _vx_matrix * vx_matrix

  The Matrix Object. An MxN matrix of some unit type.

Enumerations

- enum vx_matrix_attribute_e {
  VX_MATRIX_ATTRIBUTE_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x0,
  VX_MATRIX_ATTRIBUTE_ROWS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x1,
  VX_MATRIX_ATTRIBUTE_COLUMNS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x2,
  VX_MATRIX_ATTRIBUTE_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_MATRIX << 8)) + 0x3 }

  The matrix attributes.

Functions

- vx_status vxAccessMatrix (vx_matrix mat, void *array)

  Gets the matrix data (copy).
- vx_status vxCommitMatrix (vx_matrix mat, void *array)

  Sets the matrix data (copy)
- vx_matrix vxCreateMatrix (vx_context c, vx_enum data_type, vx_size columns, vx_size rows)

  Creates a reference to a matrix object.
- vx_status vxQueryMatrix (vx_matrix mat, vx_enum attribute, void *ptr, vx_size size)

  Queries an attribute on the matrix object.
- vx_status vxReleaseMatrix (vx_matrix *mat)

  Releases a reference to a matrix object. The object may not be garbage collected until its total reference count is zero.

3.53.2 Enumeration Type Documentation

enum vx_matrix_attribute_e

The matrix attributes.

Enumerator

VX_MATRIX_ATTRIBUTE_TYPE The value type of the matrix. Use a `vx_enum` parameter.

VX_MATRIX_ATTRIBUTE_ROWS The M dimension of the matrix. Use a `vx_size` parameter.

VX_MATRIX_ATTRIBUTE_COLUMNS The N dimension of the matrix. Use a `vx_size` parameter.

VX_MATRIX_ATTRIBUTE_SIZE The total size of the matrix in bytes. Use a `vx_size` parameter.

Definition at line 866 of file vx_types.h.

3.53.3 Function Documentation

vx_matrix vxCreateMatrix (vx_context *c,* vx_enum *data_type,* vx_size *columns,* vx_size *rows*)

Creates a reference to a matrix object.

Parameters

in	*c*	The reference to the overall context.
in	*data_type*	The unit format of the matrix. VX_TYPE_INT32 or VX_TYPE_FLOAT32.
in	*columns*	The first dimensionality.
in	*rows*	The second dimensionality.

Returns

vx_matrix

vx_status vxReleaseMatrix (vx_matrix * *mat*)

Releases a reference to a matrix object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*mat*	The matrix reference to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxQueryMatrix (vx_matrix *mat*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries an attribute on the matrix object.

Parameters

in	*mat*	The matrix object to set.
in	*attribute*	The attribute to query. Use a vx_matrix_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_status vxAccessMatrix (vx_matrix *mat*, void * *array*)

Gets the matrix data (copy).

Parameters

in	*mat*	The reference to the matrix.
out	*array*	The array in which to place the matrix.

See also

vxQueryMatrix and VX_MATRIX_ATTRIBUTE_COLUMNS and VX_MATRIX_ATTRIBUTE_ROWS to get the needed number of elements of the array.

Returns

A vx_status_e enumeration.

Postcondition

vxCommitMatrix

vx_status vxCommitMatrix (vx_matrix *mat,* void * *array*)

Sets the matrix data (copy)

Parameters

in	*mat*	The reference to the matrix.
out	*array*	The array to read the matrix.

See also

vxQueryMatrix and VX_MATRIX_ATTRIBUTE_COLUMNS and VX_MATRIX_ATTRIBUTE_ROWS to get the needed number of elements of the array.'

Returns

A vx_status_e enumeration.

Precondition

vxAccessMatrix

3.54 Object: Pyramid

3.54.1 Detailed Description

Defines the Image Pyramid Object Interface.

A Pyramid object in OpenVX represents a collection of related images. Typically, these images are created by either downscaling or upscaling a *base image*, contained in level zero of the pyramid. Successive levels of the pyramid increase or decrease in size by a factor given by the VX_PYRAMID_ATTRIBUTE_SCALE attribute. For instance, in a pyramid with 3 levels and VX_SCALE_PYRAMID_HALF, the level one image is one-half the width and one-half the height of the level zero image, and the level two image is one-quarter the width and one quarter the height of the level zero image. When downscaling or upscaling results in a non-integral number of pixels at any level, fractional pixels always get rounded up to the nearest integer. (E.g., a 3-level image pyramid beginning with level zero having a width of 9 and a scaling of VX_SCALE_PYRAMID_HALF results in the level one image with a width of $5 = \mathbf{ceil}(9 * 0.5)$ and a level two image with a width of $3 = \mathbf{ceil}(5 * 0.5)$. Position (r_N, c_N) at level N corresponds to position $(r_{N-1}/\mathbf{scale}, c_{N-1}/\mathbf{scale})$ at level $N-1$.

Macros

- #define VX_SCALE_PYRAMID_HALF (0.5f)

  Use to indicate a half-scale pyramid.
- #define VX_SCALE_PYRAMID_ORB ((vx_float32)0.8408964f)

  Use to indicate a ORB scaled pyramid whose scaling factor is $\frac{1}{\sqrt[4]{2}}$.

Typedefs

- typedef struct _vx_pyramid * vx_pyramid

  The Image Pyramid object. A set of scaled images.

Enumerations

- enum vx_pyramid_attribute_e {
  VX_PYRAMID_ATTRIBUTE_LEVELS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x0,
  VX_PYRAMID_ATTRIBUTE_SCALE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x1,
  VX_PYRAMID_ATTRIBUTE_WIDTH = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x2,
  VX_PYRAMID_ATTRIBUTE_HEIGHT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x3,
  VX_PYRAMID_ATTRIBUTE_FORMAT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PYRAMID << 8)) + 0x4 }

  The pyramid object attributes.

Functions

- vx_pyramid vxCreatePyramid (vx_context context, vx_size levels, vx_float32 scale, vx_uint32 width, vx_uint32 height, vx_df_image format)

  Creates a reference to a pyramid object of the supplied number of levels.
- vx_pyramid vxCreateVirtualPyramid (vx_graph graph, vx_size levels, vx_float32 scale, vx_uint32 width, vx_uint32 height, vx_df_image format)

  Creates a reference to a virtual pyramid object of the supplied number of levels.
- vx_image vxGetPyramidLevel (vx_pyramid pyr, vx_uint32 index)

  Retrieves a level of the pyramid as a `vx_image`*, which can be used elsewhere in OpenVX.*
- vx_status vxQueryPyramid (vx_pyramid pyr, vx_enum attribute, void *ptr, vx_size size)

  Queries an attribute from an image pyramid.
- vx_status vxReleasePyramid (vx_pyramid *pyr)

Releases a reference to a pyramid object. The object may not be garbage collected until its total reference count is zero.

3.54.2 Enumeration Type Documentation

enum vx_pyramid_attribute_e

The pyramid object attributes.

Enumerator

VX_PYRAMID_ATTRIBUTE_LEVELS The number of levels of the pyramid. Use a `vx_size` parameter.

VX_PYRAMID_ATTRIBUTE_SCALE The scale factor between each level of the pyramid. Use a `vx_float32` parameter.

VX_PYRAMID_ATTRIBUTE_WIDTH The width of the 0th image in pixels. Use a `vx_uint32` parameter.

VX_PYRAMID_ATTRIBUTE_HEIGHT The height of the 0th image in pixels. Use a `vx_uint32` parameter.

VX_PYRAMID_ATTRIBUTE_FORMAT The `vx_df_image_e` format of the image. Use a `vx_df_image` parameter.

Definition at line 898 of file vx_types.h.

3.54.3 Function Documentation

vx_pyramid vxCreatePyramid (vx_context *context*, vx_size *levels*, vx_float32 *scale*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *format*)

Creates a reference to a pyramid object of the supplied number of levels.

Parameters

in	*context*	The reference to the overall context.
in	*levels*	The number of levels desired. This is required to be a non-zero value.
in	*scale*	Used to indicate the scale between pyramid levels. This is required to be a non-zero positive value. In OpenVX 1.0, the only permissible values are `VX_SCALE_PYRAMID_HALF` or `VX_SCALE_PYRAMID_ORB`.
in	*width*	The width of the 0th level image in pixels.
in	*height*	The height of the 0th level image in pixels.
in	*format*	The format of all images in the pyramid.

Returns

`vx_pyramid`

Return values

0	No pyramid was created.
*	A pyramid reference.

vx_pyramid vxCreateVirtualPyramid (vx_graph *graph*, vx_size *levels*, vx_float32 *scale*, vx_uint32 *width*, vx_uint32 *height*, vx_df_image *format*)

Creates a reference to a virtual pyramid object of the supplied number of levels.

Virtual Pyramids can be used to connect Nodes together when the contents of the pyramids will not be accessed by the user of the API. All of the following constructions are valid:

```
vx_context context = vxCreateContext();
vx_graph graph = vxCreateGraph(context);
vx_pyramid virt[] = {
    vxCreateVirtualPyramid(graph, 4, VX_SCALE_PYRAMID_HALF, 0, 0
      , VX_DF_IMAGE_VIRT), // no dimension and format specified for level 0
    vxCreateVirtualPyramid(graph, 4, VX_SCALE_PYRAMID_HALF, 640,
       480, VX_DF_IMAGE_VIRT), // no format specified.
    vxCreateVirtualPyramid(graph, 4, VX_SCALE_PYRAMID_HALF, 640,
       480, VX_DF_IMAGE_U8), // no access
};
```

Parameters

in	*graph*	The reference to the parent graph.
in	*levels*	The number of levels desired. This is required to be a non-zero value.
in	*scale*	Used to indicate the scale between pyramid levels. This is required to be a non-zero positive value. In OpenVX 1.0, the only permissible values are VX_SCALE_PYRAMID_HALF or VX_SCALE_PYRAMID_ORB.
in	*width*	The width of the 0th level image in pixels. This may be set to zero to indicate to the interface that the value is unspecified.
in	*height*	The height of the 0th level image in pixels. This may be set to zero to indicate to the interface that the value is unspecified.
in	*format*	The format of all images in the pyramid. This may be set to VX_DF_IMAGE_VIRT to indicate that the format is unspecified.

Returns

A vx_pyramid reference.

Note

Images extracted with vxGetPyramidLevel behave as Virtual Images and cause vxAccessImagePatch to return errors.

Return values

0	No pyramid was created.
*	A pyramid reference.

vx_status vxReleasePyramid (vx_pyramid * *pyr*)

Releases a reference to a pyramid object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*pyr*	The pointer to the pyramid to release.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

Postcondition

After returning from this function the reference is zeroed.

vx_status vxQueryPyramid (vx_pyramid *pyr,* vx_enum *attribute,* void * *ptr,* vx_size *size*)

Queries an attribute from an image pyramid.

Parameters

in	*pyr*	The pyramid to query.

in	*attribute*	The attribute for which to query. Use a vx_pyramid_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_image vxGetPyramidLevel (vx_pyramid *pyr,* vx_uint32 *index*)

Retrieves a level of the pyramid as a vx_image, which can be used elsewhere in OpenVX.

Parameters

in	*pyr*	The pyramid object.
in	*index*	The index of the level, such that index is less than levels.

Returns

A vx_image reference.

Return values

0	Indicates that the index or the object is invalid.

3.55 Object: Remap

3.55.1 Detailed Description

Defines the Remap Object Interface.

Typedefs

- typedef struct _vx_remap * vx_remap

  The remap table Object. A remap table contains per-pixel mapping of output pixels to input pixels.

Enumerations

- enum vx_remap_attribute_e {
  VX_REMAP_ATTRIBUTE_SOURCE_WIDTH = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REMAP << 8)) + 0x0,
  VX_REMAP_ATTRIBUTE_SOURCE_HEIGHT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REMAP << 8)) + 0x1,
  VX_REMAP_ATTRIBUTE_DESTINATION_WIDTH = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_REMAP << 8)) + 0x2,
  VX_REMAP_ATTRIBUTE_DESTINATION_HEIGHT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_RE↩MAP << 8)) + 0x3 }

  The remap object attributes.

Functions

- vx_remap vxCreateRemap (vx_context context, vx_uint32 src_width, vx_uint32 src_height, vx_uint32 dst_↩width, vx_uint32 dst_height)

  Creates a remap table object.
- vx_status vxGetRemapPoint (vx_remap table, vx_uint32 dst_x, vx_uint32 dst_y, vx_float32 *src_x, vx_float32 *src_y)

  Retrieves the source pixel point from a destination pixel.
- vx_status vxQueryRemap (vx_remap r, vx_enum attribute, void *ptr, vx_size size)

  Queries attributes from a Remap table.
- vx_status vxReleaseRemap (vx_remap *table)

  Releases a reference to a remap table object. The object may not be garbage collected until its total reference count is zero.
- vx_status vxSetRemapPoint (vx_remap table, vx_uint32 dst_x, vx_uint32 dst_y, vx_float32 src_x, vx_float32 src_y)

  Assigns a destination pixel mapping to the source pixel.

3.55.2 Enumeration Type Documentation

enum vx_remap_attribute_e

The remap object attributes.

Enumerator

VX_REMAP_ATTRIBUTE_SOURCE_WIDTH The source width. Use a `vx_uint32` parameter.

VX_REMAP_ATTRIBUTE_SOURCE_HEIGHT The source height. Use a `vx_uint32` parameter.

VX_REMAP_ATTRIBUTE_DESTINATION_WIDTH The destination width. Use a `vx_uint32` parameter.

VX_REMAP_ATTRIBUTE_DESTINATION_HEIGHT The destination height. Use a `vx_uint32` parameter.

Definition at line 914 of file vx_types.h.

3.55.3 Function Documentation

vx_remap vxCreateRemap (vx_context *context,* vx_uint32 *src_width,* vx_uint32 *src_height,* vx_uint32 *dst_width,* vx_uint32 *dst_height*)

Creates a remap table object.

Parameters

in	*context*	The reference to the overall context.
in	*src_width*	Width of the source image in pixel.
in	*src_height*	Height of the source image in pixels.
in	*dst_width*	Width of the destination image in pixels.
in	*dst_height*	Height of the destination image in pixels.

Returns

vx_remap

Return values

0	Object could not be created.
*	Object was created.

vx_status vxReleaseRemap (vx_remap * *table*)

Releases a reference to a remap table object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*table*	The pointer to the remap table to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxSetRemapPoint (vx_remap *table*, vx_uint32 *dst_x*, vx_uint32 *dst_y*, vx_float32 *src_x*, vx_float32 *src_y*)

Assigns a destination pixel mapping to the source pixel.

Parameters

in	*table*	The remap table reference.
in	*dst_x*	The destination x coordinate.
in	*dst_y*	The destination y coordinate.
in	*src_x*	The source x coordinate in float representation to allow interpolation.
in	*src_y*	The source y coordinate in float representation to allow interpolation.

Returns

A vx_status_e enumeration.

vx_status vxGetRemapPoint (vx_remap *table*, vx_uint32 *dst_x*, vx_uint32 *dst_y*, vx_float32 * *src_x*, vx_float32 * *src_y*)

Retrieves the source pixel point from a destination pixel.

Parameters

in	*table*	The remap table reference.
in	*dst_x*	The destination x coordinate.
in	*dst_y*	The destination y coordinate.
out	*src_x*	The pointer to the location to store the source x coordinate in float representation to allow interpolation.
out	*src_y*	The pointer to the location to store the source y coordinate in float representation to allow interpolation.

Returns

A vx_status_e enumeration.

vx_status vxQueryRemap (vx_remap *r*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries attributes from a Remap table.

Parameters

in	*r*	The remap to query.
in	*attribute*	The attribute to query. Use a vx_remap_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

3.56 Object: Scalar

3.56.1 Detailed Description

Defines the Scalar Object interface.

Typedefs

- typedef struct _vx_scalar * vx_scalar

  An opaque reference to a scalar.

Enumerations

- enum vx_scalar_attribute_e { VX_SCALAR_ATTRIBUTE_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_↩ TYPE_SCALAR << 8)) + 0x0 }

  The scalar attributes list.

Functions

- vx_status vxAccessScalarValue (vx_scalar ref, void *ptr)

  Gets the scalar value out of a reference.
- vx_status vxCommitScalarValue (vx_scalar ref, void *ptr)

  Sets the scalar value in a reference.
- vx_scalar vxCreateScalar (vx_context context, vx_enum data_type, void *ptr)

  Creates a reference to a scalar object. Also see Node Parameters.
- vx_status vxQueryScalar (vx_scalar scalar, vx_enum attribute, void *ptr, vx_size size)

  Queries attributes from a scalar.
- vx_status vxReleaseScalar (vx_scalar *scalar)

  Releases a reference to a scalar object. The object may not be garbage collected until its total reference count is zero.

3.56.2 Typedef Documentation

typedef struct _vx_scalar* vx_scalar

An opaque reference to a scalar.
A scalar can be up to 64 bits wide.

See also

vxCreateScalar

Definition at line 137 of file vx_types.h.

3.56.3 Enumeration Type Documentation

enum vx_scalar_attribute_e

The scalar attributes list.

Enumerator

VX_SCALAR_ATTRIBUTE_TYPE Queries the type of atomic that is contained in the scalar. Use a `vx_enum` parameter.

Definition at line 786 of file vx_types.h.

3.56.4 Function Documentation

vx_scalar vxCreateScalar (vx_context *context,* vx_enum *data_type,* void * *ptr*)

Creates a reference to a scalar object. Also see Node Parameters.

Parameters

in	*context*	The reference to the system context.
in	*data_type*	The vx_type_e of the scalar. Must be greater than VX_TYPE_INVALID and less than VX_TYPE_SCALAR_MAX.
in	*ptr*	The pointer to the initial value of the scalar.

Returns

A vx_scalar reference.

Return values

0	The scalar could not be created.
*	The scalar was created. Check for further errors with vxGetStatus.

vx_status vxReleaseScalar (vx_scalar * *scalar*)

Releases a reference to a scalar object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*scalar*	The pointer to the scalar to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxQueryScalar (vx_scalar *scalar*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries attributes from a scalar.

Parameters

in	*scalar*	The scalar object.
in	*attribute*	The enumeration to query. Use a vx_scalar_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_status vxAccessScalarValue (vx_scalar *ref*, void * *ptr*)

Gets the scalar value out of a reference.

Note

Use this in conjunction with Query APIs that return references which should be converted into values.

Parameters

in	*ref*	The reference from which to get the scalar value.
out	*ptr*	An appropriate typed pointer that points to a location to which to copy the scalar value.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	If the ref is not a valid reference.
VX_ERROR_INVALID_PARAMETERS	If *ptr* is NULL.
VX_ERROR_INVALID_TYPE	If the type does not match the type in the reference or is a bad value.

vx_status vxCommitScalarValue (vx_scalar *ref,* void * *ptr*)

Sets the scalar value in a reference.

Note

Use this in conjunction with Parameter APIs that return references to parameters that need to be altered.

Parameters

in	*ref*	The reference from which to get the scalar value.
in	*ptr*	An appropriately typed pointer that points to a location to which to copy the scalar value.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	If the ref is not a valid reference.
VX_ERROR_INVALID_PARAMETERS	If *ptr* is NULL.
VX_ERROR_INVALID_TYPE	If the type does not match the type in the reference or is a bad value.

3.57 Object: Threshold

3.57.1 Detailed Description

Defines the Threshold Object Interface.

Typedefs

- typedef struct _vx_threshold * vx_threshold

  The Threshold Object. A thresholding object contains the types and limit values of the thresholding required.

Enumerations

- enum vx_threshold_attribute_e {
  VX_THRESHOLD_ATTRIBUTE_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x0,
  VX_THRESHOLD_ATTRIBUTE_THRESHOLD_VALUE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x1,
  VX_THRESHOLD_ATTRIBUTE_THRESHOLD_LOWER = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x2,
  VX_THRESHOLD_ATTRIBUTE_THRESHOLD_UPPER = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x3,
  VX_THRESHOLD_ATTRIBUTE_TRUE_VALUE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x4,
  VX_THRESHOLD_ATTRIBUTE_FALSE_VALUE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_THRESHOLD << 8)) + 0x5 }

  The threshold attributes.
- enum vx_threshold_type_e {
  VX_THRESHOLD_TYPE_BINARY = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_THRESHOLD_TYPE << 12)) + 0x0,
  VX_THRESHOLD_TYPE_RANGE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_THRESHOLD_TYPE << 12)) + 0x1 }

  The Threshold types.

Functions

- vx_threshold vxCreateThreshold (vx_context c, vx_enum thresh_type, vx_enum data_type)

  Creates a reference to a threshold object of a given type.
- vx_status vxQueryThreshold (vx_threshold thresh, vx_enum attribute, void *ptr, vx_size size)

  Queries an attribute on the threshold object.
- vx_status vxReleaseThreshold (vx_threshold *thresh)

  Releases a reference to a threshold object. The object may not be garbage collected until its total reference count is zero.
- vx_status vxSetThresholdAttribute (vx_threshold thresh, vx_enum attribute, void *ptr, vx_size size)

  Sets attributes on the threshold object.

3.57.2 Enumeration Type Documentation

enum vx_threshold_type_e

The Threshold types.

Enumerator

VX_THRESHOLD_TYPE_BINARY A threshold with only 1 value.

VX_THRESHOLD_TYPE_RANGE A threshold with 2 values (upper/lower). Use with Canny Edge Detection.

Definition at line 838 of file vx_types.h.

enum vx_threshold_attribute_e

The threshold attributes.

Enumerator

VX_THRESHOLD_ATTRIBUTE_TYPE The value type of the threshold. Use a vx_enum parameter. Will contain a vx_threshold_type_e.

VX_THRESHOLD_ATTRIBUTE_THRESHOLD_VALUE The value of the single threshold. Use a vx_int32 parameter.

VX_THRESHOLD_ATTRIBUTE_THRESHOLD_LOWER The value of the lower threshold. Use a vx_int32 parameter.

VX_THRESHOLD_ATTRIBUTE_THRESHOLD_UPPER The value of the higher threshold. Use a vx_int32 parameter.

VX_THRESHOLD_ATTRIBUTE_TRUE_VALUE The value of the TRUE threshold. Use a vx_int32 parameter.

VX_THRESHOLD_ATTRIBUTE_FALSE_VALUE The value of the FALSE threshold. Use a vx_int32 parameter.

Definition at line 848 of file vx_types.h.

3.57.3 Function Documentation

vx_threshold vxCreateThreshold (vx_context *c,* vx_enum *thresh_type,* vx_enum *data_type*)

Creates a reference to a threshold object of a given type.

Parameters

in	*c*	The reference to the overall context.
in	*thresh_type*	The type of threshold to create.
in	*data_type*	The data type of the threshold's value(s).

Note

For OpenVX 1.0, data_type can only be VX_TYPE_UINT8.

Returns

vx_threshold

vx_status vxReleaseThreshold (vx_threshold * *thresh*)

Releases a reference to a threshold object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*thresh*	The pointer to the threshold to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxSetThresholdAttribute (vx_threshold *thresh*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Sets attributes on the threshold object.

Parameters

in	*thresh*	The threshold object to set.
in	*attribute*	The attribute to modify. Use a vx_threshold_attribute_e enumeration.
in	*ptr*	The pointer to the value to which to set the attribute.
in	*size*	The size of the data pointed to by *ptr*.

Returns

A vx_status_e enumeration.

vx_status vxQueryThreshold (vx_threshold *thresh*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Queries an attribute on the threshold object.

Parameters

in	*thresh*	The threshold object to set.
in	*attribute*	The attribute to query. Use a vx_threshold_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

3.58 Administrative Features

3.58.1 Detailed Description

Defines the Administrative Features of OpenVX.

These features are administrative in nature and require more understanding and are more complex to use.

Modules

- Advanced Objects
- Advanced Framework API

  Describes components that are considered to be advanced.

3.59 Advanced Objects

3.59.1 Detailed Description

Modules

- Object: Array (Advanced)

  Defines the advanced features of the Array Interface.
- Object: Node (Advanced)

  Defines the advanced features of the Node Interface.
- Object: Delay

  Defines the Delay Object interface.
- Object: Kernel

  Defines the Kernel Object and Interface.
- Object: Parameter

  Defines the Parameter Object interface.

3.60 Object: Array (Advanced)

3.60.1 Detailed Description

Defines the advanced features of the Array Interface.

Functions

- vx_enum vxRegisterUserStruct (vx_context context, vx_size size)

  Registers user-defined structures to the context.

3.60.2 Function Documentation

vx_enum vxRegisterUserStruct (vx_context *context,* vx_size *size*)

Registers user-defined structures to the context.

Parameters

in	*context*	The reference to the implementation context.
in	*size*	The size of user struct in bytes.

Returns

A vx_enum value that is a type given to the User to refer to their custom structure when declaring a vx_array of that structure.

Return values

VX_TYPE_INVALID	If the namespace of types has been exhausted.

Note

This call should only be used once within the lifetime of a context for a specific structure.

```
    typedef struct _mystruct {
        vx_uint32 some_uint;
        vx_float64 some_double;
    } mystruct;
#define MY_NUM_ITEMS (10)
    vx_enum mytype = vxRegisterUserStruct(context, sizeof(mystruct));
    vx_array array = vxCreateArray(context, mytype, MY_NUM_ITEMS);
```

3.61 Object: Node (Advanced)

3.61.1 Detailed Description

Defines the advanced features of the Node Interface.

Modules

- Node: Border Modes

  Defines the border mode behaviors.

Functions

- vx_node vxCreateGenericNode (vx_graph graph, vx_kernel kernel)

  Creates a reference to a node object for a given kernel.

3.61.2 Function Documentation

vx_node vxCreateGenericNode (vx_graph *graph,* vx_kernel *kernel*)

Creates a reference to a node object for a given kernel.

This node has no references assigned as parameters after completion. The client is then required to set these parameters manually by `vxSetParameterByIndex`. When clients supply their own node creation functions (for use with User Kernels), this is the API to use along with the parameter setting API.

Parameters

`in`	*graph*	The reference to the graph in which this node exists.
`in`	*kernel*	The kernel reference to associate with this new node.

Returns

vx_node

Return values

0	The node failed to create.
*	A node was created.

Postcondition

Call `vxSetParameterByIndex` for as many parameters as needed to be set.

3.62 Node: Border Modes

3.62.1 Detailed Description

Defines the border mode behaviors.

Border Mode behavior is set as an attribute of the node, not as a direct parameter to the kernel. This allows clients to *set-and-forget* the modes of any particular node that supports border modes. All nodes shall support VX_BORDER_MODE_UNDEFINED.

Data Structures

- struct vx_border_mode_t

  Use with the enumeration VX_NODE_ATTRIBUTE_BORDER_MODE to set the border mode behavior of a node that supports borders. More...

Enumerations

- enum vx_border_mode_e {
  VX_BORDER_MODE_UNDEFINED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_BORDER_MODE << 12)) + 0x0,
  VX_BORDER_MODE_CONSTANT = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_BORDER_MODE << 12)) + 0x1,
  VX_BORDER_MODE_REPLICATE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_BORDER_MODE << 12)) + 0x2 }

  The border mode list.

3.62.2 Data Structure Documentation

struct vx_border_mode_t

Use with the enumeration VX_NODE_ATTRIBUTE_BORDER_MODE to set the border mode behavior of a node that supports borders.

Definition at line 1339 of file vx_types.h.

Data Fields

vx_enum	mode	See vx_border_mode_e.
vx_uint32	constant_value	For the mode VX_BORDER_MODE_CONSTANT, this value is filled into each pixel. If there are sub-channels in the pixel then this value is divided up accordingly.

3.62.3 Enumeration Type Documentation

enum vx_border_mode_e

The border mode list.

Enumerator

VX_BORDER_MODE_UNDEFINED No defined border mode behavior is given.

VX_BORDER_MODE_CONSTANT For nodes that support this behavior, a constant value is *filled-in* when accessing out-of-bounds pixels.

VX_BORDER_MODE_REPLICATE For nodes that support this behavior, a replication of the nearest edge pixels value is given for out-of-bounds pixels.

Definition at line 1068 of file vx_types.h.

3.63 Object: Delay

3.63.1 Detailed Description

Defines the Delay Object interface.

A Delay is an opaque object that contains a manually-controlled, temporally-delayed list of objects.

Typedefs

- typedef struct _vx_delay * vx_delay

  The delay object. This is like a ring buffer of objects that is maintained by the OpenVX implementation.

Enumerations

- enum vx_delay_attribute_e {
  VX_DELAY_ATTRIBUTE_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DELAY << 8)) + 0x0,
  VX_DELAY_ATTRIBUTE_COUNT = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_DELAY << 8)) + 0x1 }

  The delay attribute list.

Functions

- vx_status vxAgeDelay (vx_delay delay)

  Ages the internal delay ring by one. This means that once this API is called the reference from index 0 will go to index -1 and so forth until $-count+1$ is reached. This last object will become 0. Once the delay has been aged, it updates the reference in any associated nodes.
- vx_delay vxCreateDelay (vx_context context, vx_reference exemplar, vx_size count)

  Creates a Delay object.
- vx_reference vxGetReferenceFromDelay (vx_delay delay, vx_int32 index)

  Retrieves a reference from a delay object.
- vx_status vxQueryDelay (vx_delay delay, vx_enum attribute, void *ptr, vx_size size)

  Queries a `vx_delay` object attribute.
- vx_status vxReleaseDelay (vx_delay *delay)

  Releases a reference to a delay object. The object may not be garbage collected until its total reference count is zero.

3.63.2 Typedef Documentation

typedef struct _vx_delay* vx_delay

The delay object. This is like a ring buffer of objects that is maintained by the OpenVX implementation.

See also

vxCreateDelay

Definition at line 188 of file vx_types.h.

3.63.3 Enumeration Type Documentation

enum vx_delay_attribute_e

The delay attribute list.

Enumerator

VX_DELAY_ATTRIBUTE_TYPE The type of reference contained in the delay. Use a `vx_enum` parameter.

VX_DELAY_ATTRIBUTE_COUNT The number of items in the delay. Use a `vx_uint32` parameter.

Definition at line 1110 of file vx_types.h.

3.63.4 Function Documentation

vx_status vxQueryDelay (vx_delay *delay,* vx_enum *attribute,* void * *ptr,* vx_size *size*)

Queries a `vx_delay` object attribute.

Parameters

in	*delay*	The coordinates object to set.
in	*attribute*	The attribute to query. Use a vx_delay_attribute_e enumeration.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

vx_status vxReleaseDelay (vx_delay * *delay*)

Releases a reference to a delay object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*delay*	The pointer to the delay to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_delay vxCreateDelay (vx_context *context*, vx_reference *exemplar*, vx_size *count*)

Creates a Delay object.

This function uses only the metadata from the exemplar, ignoring the object data. It does not alter the exemplar or keep or release the reference to the exemplar.

Parameters

in	*context*	The reference to the system context.
in	*exemplar*	The exemplar object.
in	*count*	The number of reference in the delay.

Returns

vx_delay

vx_reference vxGetReferenceFromDelay (vx_delay *delay*, vx_int32 *index*)

Retrieves a reference from a delay object.

Parameters

in	*delay*	The reference to the delay object.
in	*index*	An index into the delay from which to extract the reference.

Returns

vx_reference

Note

The delay index is in the range $[-count + 1, 0]$. 0 is always the *current* object.
A reference from a delay object must not be given to its associated release API (e.g. vxReleaseImage).
Use the vxReleaseDelay only.

vx_status vxAgeDelay (vx_delay *delay*)

Ages the internal delay ring by one. This means that once this API is called the reference from index 0 will go to index -1 and so forth until $-count + 1$ is reached. This last object will become 0. Once the delay has been aged, it updates the reference in any associated nodes.

Parameters

in	*delay*	

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Delay was aged.
VX_ERROR_INVALID_REFERENCE	The value passed as delay was not a vx_delay.

3.64 Object: Kernel

3.64.1 Detailed Description

Defines the Kernel Object and Interface.

A Kernel in OpenVX is the abstract representation of an computer vision function, such as a “Sobel Gradient” or “Lucas Kanade Feature Tracking”. A vision function may implement many similar or identical features from other functions, but it is still considered a single unique kernel as long as it is named by the same string and enumeration and conforms to the results specified by OpenVX. Kernels are similar to function signatures in this regard.

In each of the cases, a client of OpenVX could request the kernels in nearly the same manner. There are two main approaches, which depend on the method a client calls to get the kernel reference. The first uses enumerations.

```
  vx_kernel kernel = vxGetKernelByEnum(context,
VX_KERNEL_SOBEL_3x3);
  vx_node node = vxCreateGenericNode(graph, kernel);
```

The second method depends on using strings to get the kernel reference.

```
  vx_kernel kernel = vxGetKernelByName(context, "
org.khronos.openvx.sobel3x3");
  vx_node node = vxCreateGenericNode(graph, kernel);
```

Data Structures

- struct vx_kernel_info_t

  The Kernel Information Structure. This is returned by the Context to indicate which kernels are available in the OpenVX implementation. More...

Macros

- #define VX_MAX_KERNEL_NAME (256)

  Defines the maximum string length of a kernel name to be added to OpenVX.

Typedefs

- typedef struct _vx_kernel * vx_kernel

  An opaque reference to the descriptor of a kernel.

Enumerations

- enum vx_kernel_attribute_e {
  VX_KERNEL_ATTRIBUTE_PARAMETERS = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x0,
  VX_KERNEL_ATTRIBUTE_NAME = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x1,
  VX_KERNEL_ATTRIBUTE_ENUM = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x2,
  VX_KERNEL_ATTRIBUTE_LOCAL_DATA_SIZE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x3,
  VX_KERNEL_ATTRIBUTE_LOCAL_DATA_PTR = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_KERNEL << 8)) + 0x4 }

  The kernel attributes list.
- enum vx_kernel_e {
  VX_KERNEL_INVALID = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x0,
  VX_KERNEL_COLOR_CONVERT = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1,
  VX_KERNEL_CHANNEL_EXTRACT = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BA↩
  SE) + 0x2,
  VX_KERNEL_CHANNEL_COMBINE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BA↩

SE) + 0x3,
VX_KERNEL_SOBEL_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x4,
VX_KERNEL_MAGNITUDE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x5,
VX_KERNEL_PHASE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x6,
VX_KERNEL_SCALE_IMAGE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x7,
VX_KERNEL_TABLE_LOOKUP = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x8,
VX_KERNEL_HISTOGRAM = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x9,
VX_KERNEL_EQUALIZE_HISTOGRAM = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR↩
_BASE) + 0xA,
VX_KERNEL_ABSDIFF = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xB,
VX_KERNEL_MEAN_STDDEV = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xC,
VX_KERNEL_THRESHOLD = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xD,
VX_KERNEL_INTEGRAL_IMAGE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xE,
VX_KERNEL_DILATE_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0xF,
VX_KERNEL_ERODE_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x10,
VX_KERNEL_MEDIAN_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x11,
VX_KERNEL_BOX_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x12,
VX_KERNEL_GAUSSIAN_3x3 = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x13,
VX_KERNEL_CUSTOM_CONVOLUTION = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KH↩
R_BASE) + 0x14,
VX_KERNEL_GAUSSIAN_PYRAMID = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_B↩
ASE) + 0x15,
VX_KERNEL_ACCUMULATE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x16,
VX_KERNEL_ACCUMULATE_WEIGHTED = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_K↩
HR_BASE) + 0x17,
VX_KERNEL_ACCUMULATE_SQUARE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR↩
_BASE) + 0x18,
VX_KERNEL_MINMAXLOC = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x19,
VX_KERNEL_CONVERTDEPTH = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1A,
VX_KERNEL_CANNY_EDGE_DETECTOR = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_K↩
HR_BASE) + 0x1B,
VX_KERNEL_AND = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1C,
VX_KERNEL_OR = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1D,
VX_KERNEL_XOR = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1E,
VX_KERNEL_NOT = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x1F,
VX_KERNEL_MULTIPLY = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x20,
VX_KERNEL_ADD = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x21,
VX_KERNEL_SUBTRACT = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x22,
VX_KERNEL_WARP_AFFINE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x23,
VX_KERNEL_WARP_PERSPECTIVE = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_B↩
ASE) + 0x24,
VX_KERNEL_HARRIS_CORNERS = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x25,
VX_KERNEL_FAST_CORNERS = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x26,
VX_KERNEL_OPTICAL_FLOW_PYR_LK = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KH↩
R_BASE) + 0x27,
VX_KERNEL_REMAP = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR_BASE) + 0x28,
VX_KERNEL_HALFSCALE_GAUSSIAN = VX_KERNEL_BASE(VX_ID_KHRONOS, VX_LIBRARY_KHR↩
_BASE) + 0x29,

VX_KERNEL_MAX_1_0 }

The standard list of available vision kernels.

Functions

- vx_kernel vxGetKernelByEnum (vx_context context, vx_enum kernel)

  Obtains a reference to the kernel using the `vx_kernel_e` enumeration.
- vx_kernel vxGetKernelByName (vx_context context, vx_char *name)

  Obtains a reference to a kernel using a string to specify the name.
- vx_status vxQueryKernel (vx_kernel kernel, vx_enum attribute, void *ptr, vx_size size)

  This allows the client to query the kernel to get information about the number of parameters, enum values, etc.
- vx_status vxReleaseKernel (vx_kernel *kernel)

  Release the reference to the kernel. The object may not be garbage collected until its total reference count is zero.

3.64.2 Data Structure Documentation

struct vx_kernel_info_t

The Kernel Information Structure. This is returned by the Context to indicate which kernels are available in the OpenVX implementation.

Definition at line 1311 of file vx_types.h.

Data Fields

vx_enum	enumeration	The kernel enumeration value from `vx_kernel_e` (or an extension thereof). See also vxGetKernelByEnum
vx_char	name[VX_MAX_KERNEL_NAME]	The kernel name in dotted hierarchical format. e.g. "org.khronos.openvx.sobel3x3". See also vxGetKernelByName

3.64.3 Typedef Documentation

typedef struct _vx_kernel* vx_kernel

An opaque reference to the descriptor of a kernel.

See also

vxGetKernelByName
vxGetKernelByEnum

Definition at line 152 of file vx_types.h.

3.64.4 Enumeration Type Documentation

enum vx_kernel_e

The standard list of available vision kernels.

Each kernel listed here can be used with the `vxGetKernelByEnum` call. When programming the parameters, use

- `VX_INPUT` for [in]
- `VX_OUTPUT` for [out]

- VX_BIDIRECTIONAL for [in,out]

When programming the parameters, use

- VX_TYPE_IMAGE for a vx_image in the size field of vxGetParameterByIndex or vxSetParameterByIndex *
- VX_TYPE_ARRAY for a vx_array in the size field of vxGetParameterByIndex or vxSetParameterByIndex *
- or other appropriate types in vx_type_e.

Enumerator

VX_KERNEL_INVALID The invalid kernel is used to for conformance failure in relation to some kernel operation (Get/Release). If the kernel is executed it shall always return an error. The kernel has no parameters. To address by name use "org.khronos.openvx.invalid".

VX_KERNEL_COLOR_CONVERT The Color Space conversion kernel. The conversions are based on the vx_df_image_e code in the images.

See also

Color Convert

VX_KERNEL_CHANNEL_EXTRACT The Generic Channel Extraction Kernel. This kernel can remove individual color channels from an interleaved or semi-planar, planar, sub-sampled planar image. A client could extract a red channel from an interleaved RGB image or do a Luma extract from a YUV format.

See also

Channel Extract

VX_KERNEL_CHANNEL_COMBINE The Generic Channel Combine Kernel. This kernel combine multiple individual planes into a single multiplanar image of the type specified in the output image.

See also

Channel Combine

VX_KERNEL_SOBEL_3x3 The Sobel 3x3 Filter Kernel.

See also

Sobel 3x3

VX_KERNEL_MAGNITUDE The Magnitude Kernel. This kernel produces a magnitude plane from two input gradients.

See also

Magnitude

VX_KERNEL_PHASE The Phase Kernel. This kernel produces a phase plane from two input gradients.

See also

Phase

VX_KERNEL_SCALE_IMAGE The Scale Image Kernel. This kernel provides resizing of an input image to an output image. The scaling factor is determined but the relative sizes of the input and output.

See also

Scale Image

VX_KERNEL_TABLE_LOOKUP The Table Lookup kernel.

See also

TableLookup

VX_KERNEL_HISTOGRAM The Histogram Kernel.

See also

Histogram

VX_KERNEL_EQUALIZE_HISTOGRAM The Histogram Equalization Kernel.

See also

Equalize Histogram

VX_KERNEL_ABSDIFF The Absolute Difference Kernel.

See also

Absolute Difference

VX_KERNEL_MEAN_STDDEV The Mean and Standard Deviation Kernel.

See also

Mean and Standard Deviation

VX_KERNEL_THRESHOLD The Threshold Kernel.

See also

Thresholding

VX_KERNEL_INTEGRAL_IMAGE The Integral Image Kernel.

See also

Integral Image

VX_KERNEL_DILATE_3x3 The dilate kernel.

See also

Dilate Image

VX_KERNEL_ERODE_3x3 The erode kernel.

See also

Dilate Image

VX_KERNEL_MEDIAN_3x3 The median image filter.

See also

Median Filter

VX_KERNEL_BOX_3x3 The box filter kernel.

See also

Box Filter

VX_KERNEL_GAUSSIAN_3x3 The gaussian filter kernel.

See also

Gaussian Filter

VX_KERNEL_CUSTOM_CONVOLUTION The custom convolution kernel.

See also

Custom Convolution

VX_KERNEL_GAUSSIAN_PYRAMID The gaussian image pyramid kernel.

See also

Gaussian Image Pyramid

VX_KERNEL_ACCUMULATE The accumulation kernel.

See also

Accumulate

VX_KERNEL_ACCUMULATE_WEIGHTED The weigthed accumulation kernel.

See also

Accumulate Weighted

VX_KERNEL_ACCUMULATE_SQUARE The squared accumulation kernel.

See also

Accumulate Squared

VX_KERNEL_MINMAXLOC The min and max location kernel.

See also

Min, Max Location

VX_KERNEL_CONVERTDEPTH The bit-depth conversion kernel.

See also

Convert Bit depth

VX_KERNEL_CANNY_EDGE_DETECTOR The Canny Edge Detector.

See also

Canny Edge Detector

VX_KERNEL_AND The Bitwise And Kernel.

See also

Bitwise AND

VX_KERNEL_OR The Bitwise Inclusive Or Kernel.

See also

Bitwise INCLUSIVE OR

VX_KERNEL_XOR The Bitwise Exclusive Or Kernel.

See also

Bitwise EXCLUSIVE OR

VX_KERNEL_NOT The Bitwise Not Kernel.

See also

Bitwise NOT

VX_KERNEL_MULTIPLY The Pixelwise Multiplication Kernel.

See also

Pixel-wise Multiplication

VX_KERNEL_ADD The Addition Kernel.

See also

Arithmetic Addition

VX_KERNEL_SUBTRACT The Subtraction Kernel.

See also

Arithmetic Subtraction

VX_KERNEL_WARP_AFFINE The Warp Affine Kernel.

See also

Warp Affine

VX_KERNEL_WARP_PERSPECTIVE The Warp Perspective Kernel.

See also

Warp Perspective

VX_KERNEL_HARRIS_CORNERS The Harris Corners Kernel.

See also

Harris Corners

VX_KERNEL_FAST_CORNERS The FAST Corners Kernel.

See also

Fast Corners

VX_KERNEL_OPTICAL_FLOW_PYR_LK The Optical Flow Pyramid (LK) Kernel.

See also

Optical Flow Pyramid (LK)

VX_KERNEL_REMAP The Remap Kernel.

See also

Remap

VX_KERNEL_HALFSCALE_GAUSSIAN The Half Scale Gaussian Kernel.

See also

Scale Image

Definition at line 56 of file vx_kernels.h.

enum vx_kernel_attribute_e

The kernel attributes list.

Enumerator

VX_KERNEL_ATTRIBUTE_PARAMETERS Queries a kernel for the number of parameters the kernel supports. Use a `vx_uint32` parameter.

VX_KERNEL_ATTRIBUTE_NAME Queries the name of the kernel. Not settable. Use a `vx_char`[VX_MAX_KERNEL_NAME] array (not a `vx_array`).

VX_KERNEL_ATTRIBUTE_ENUM Queries the enum of the kernel. Not settable. Use a `vx_enum` parameter.

VX_KERNEL_ATTRIBUTE_LOCAL_DATA_SIZE The local data area allocated with each kernel when it becomes a node. Use a `vx_size` parameter.

Note

If not set it will default to zero.

VX_KERNEL_ATTRIBUTE_LOCAL_DATA_PTR The local data pointer allocated with each kernel when it becomes a node. Use a void pointer parameter. Use a `vx_size` parameter.

Definition at line 700 of file vx_types.h.

3.64.5 Function Documentation

vx_kernel vxGetKernelByName (vx_context *context*, vx_char * *name*)

Obtains a reference to a kernel using a string to specify the name.

Parameters

in	*context*	The reference to the implementation context.
in	*name*	The string of the name of the kernel to get.

Returns

A kernel reference or zero if an error occurred.

Return values

0	The kernel name is not found in the context.

Precondition

`vxLoadKernels` if the kernel is not provided by the OpenVX implementation.

Note

User Kernels should follow a "dotted" heirarchical syntax. For example: "com.company.example.xyz".

vx_kernel vxGetKernelByEnum (vx_context *context*, vx_enum *kernel*)

Obtains a reference to the kernel using the `vx_kernel_e` enumeration.
Enum values above the standard set are assumed to apply to loaded libraries.

Parameters

in	*context*	The reference to the implementation context.
in	*kernel*	A value from `vx_kernel_e` or a vendor or client-defined value.

Returns

A `vx_kernel`.

Return values

0	The kernel enumeration is not found in the context.

Precondition

`vxLoadKernels` if the kernel is not provided by the OpenVX implementation.

vx_status vxQueryKernel (vx_kernel *kernel*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

This allows the client to query the kernel to get information about the number of parameters, enum values, etc.

Parameters

in	*kernel*	The kernel reference to query.
in	*attribute*	The attribute to query. Use a `vx_kernel_attribute_e`.
out	*ptr*	The pointer to the location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the kernel is not a `vx_kernel`.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.
VX_ERROR_NOT_SUPPORTED	If the attribute value is not supported in this implementation.

vx_status vxReleaseKernel (vx_kernel * *kernel*)

Release the reference to the kernel. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*kernel*	The pointer to the kernel reference to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

3.65 Object: Parameter

3.65.1 Detailed Description

Defines the Parameter Object interface.

An abstract input, output, or bidirectional data object passed to a computer vision function. This object contains the signature of that parameter's usage from the kernel description. This information includes:

- *Signature Index* - The numbered index of the parameter in the signature.
- *Object Type* - e.g., VX_TYPE_IMAGE or VX_TYPE_ARRAY or some other object type from vx_type_e.
- *Usage Model* - e.g., VX_INPUT, VX_OUTPUT, or VX_BIDIRECTIONAL.
- *Presence State* - e.g., VX_PARAMETER_STATE_REQUIRED or VX_PARAMETER_STATE_OPTIONAL.

Typedefs

- typedef struct _vx_parameter * vx_parameter

  An opaque reference to a single parameter.

Enumerations

- enum vx_direction_e {
  VX_INPUT = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTION << 12)) + 0x0,
  VX_OUTPUT = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTION << 12)) + 0x1,
  VX_BIDIRECTIONAL = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTION << 12)) + 0x2 }

  An indication of how a kernel will treat the given parameter.
- enum vx_parameter_attribute_e {
  VX_PARAMETER_ATTRIBUTE_INDEX = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x0,
  VX_PARAMETER_ATTRIBUTE_DIRECTION = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x1,
  VX_PARAMETER_ATTRIBUTE_TYPE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x2,
  VX_PARAMETER_ATTRIBUTE_STATE = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x3,
  VX_PARAMETER_ATTRIBUTE_REF = (((VX_ID_KHRONOS) << 20) | (VX_TYPE_PARAMETER << 8)) + 0x4 }

  The parameter attributes list.
- enum vx_parameter_state_e {
  VX_PARAMETER_STATE_REQUIRED = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_PARAMETER_STATE << 12)) + 0x0,
  VX_PARAMETER_STATE_OPTIONAL = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_PARAMETER_STATE << 12)) + 0x1 }

  The parameter state type.

Functions

- vx_parameter vxGetKernelParameterByIndex (vx_kernel kernel, vx_uint32 index)

  Retrieves a vx_parameter from a vx_kernel.
- vx_parameter vxGetParameterByIndex (vx_node node, vx_uint32 index)

  Retrieves a vx_parameter from a vx_node.
- vx_status vxQueryParameter (vx_parameter param, vx_enum attribute, void *ptr, vx_size size)

  Allows the client to query a parameter to determine its meta-information.
- vx_status vxReleaseParameter (vx_parameter *param)

  Releases a reference to a parameter object. The object may not be garbage collected until its total reference count is zero.

- vx_status vxSetParameterByIndex (vx_node node, vx_uint32 index, vx_reference value)

  Sets the specified parameter data for a kernel on the node.
- vx_status vxSetParameterByReference (vx_parameter parameter, vx_reference value)

  Associates a parameter reference and a data reference with a kernel on a node.

3.65.2 Typedef Documentation

typedef struct _vx_parameter* vx_parameter

An opaque reference to a single parameter.

See also

vxGetParameterByIndex

Definition at line 159 of file vx_types.h.

3.65.3 Enumeration Type Documentation

enum vx_direction_e

An indication of how a kernel will treat the given parameter.

Enumerator

VX_INPUT The parameter is an input only.

VX_OUTPUT The parameter is an output only.

VX_BIDIRECTIONAL The parameter is both an input and output.

Definition at line 524 of file vx_types.h.

enum vx_parameter_attribute_e

The parameter attributes list.

Enumerator

VX_PARAMETER_ATTRIBUTE_INDEX Queries a parameter for its index value on the kernel with which it is associated. Use a `vx_uint32` parameter.

VX_PARAMETER_ATTRIBUTE_DIRECTION Queries a parameter for its direction value on the kernel with which it is associated. Use a `vx_enum` parameter.

VX_PARAMETER_ATTRIBUTE_TYPE Queries a parameter for its size in bytes or if it is a `vx_image` or `vx_array` its `vx_type_e` is returned. Use a `vx_enum` parameter.

VX_PARAMETER_ATTRIBUTE_STATE Queries a parameter for its state. A value in `vx_parameter_state_e` is returned. Use a `vx_enum` parameter.

VX_PARAMETER_ATTRIBUTE_REF Use to extract the reference contained in the parameter. Use a `vx_reference` parameter.

Definition at line 750 of file vx_types.h.

enum vx_parameter_state_e

The parameter state type.

Enumerator

VX_PARAMETER_STATE_REQUIRED Default. The parameter must be supplied. If not set, during Verify, an error is returned.

VX_PARAMETER_STATE_OPTIONAL The parameter may be unspecified. The kernel takes care not to deference optional parameters until it is certain they are valid.

Definition at line 1054 of file vx_types.h.

3.65.4 Function Documentation

vx_parameter vxGetKernelParameterByIndex (vx_kernel *kernel,* vx_uint32 *index*)

Retrieves a vx_parameter from a vx_kernel.

Parameters

in	*kernel*	The reference to the kernel.
in	*index*	The index of the parameter.

Returns

A vx_parameter.

Return values

0	Either the kernel or index is invalid.
*	The parameter reference.

vx_parameter vxGetParameterByIndex (vx_node *node*, vx_uint32 *index*)

Retrieves a vx_parameter from a vx_node.

Parameters

in	*node*	The node from which to extract the parameter.
in	*index*	The index of the parameter to which to get a reference.

Returns

vx_parameter

vx_status vxReleaseParameter (vx_parameter * *param*)

Releases a reference to a parameter object. The object may not be garbage collected until its total reference count is zero.

Parameters

in	*param*	The pointer to the parameter to release.

Postcondition

After returning from this function the reference is zeroed.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If graph is not a vx_graph.

vx_status vxSetParameterByIndex (vx_node *node*, vx_uint32 *index*, vx_reference *value*)

Sets the specified parameter data for a kernel on the node.

Parameters

in	*node*	The node that contains the kernel.
in	*index*	The index of the parameter desired.
in	*value*	The reference to the parameter.

Returns

A vx_status_e enumeration.

See also

vxSetParameterByReference

vx_status vxSetParameterByReference (vx_parameter *parameter,* vx_reference *value*)

Associates a parameter reference and a data reference with a kernel on a node.

Parameters

in	*parameter*	The reference to the kernel parameter.
in	*value*	The value to associate with the kernel parameter.

Returns

A vx_status_e enumeration.

See also

vxGetParameterByIndex

vx_status vxQueryParameter (vx_parameter *param*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Allows the client to query a parameter to determine its meta-information.

Parameters

in	*param*	The reference to the parameter.
in	*attribute*	The attribute to query. Use a vx_parameter_attribute_e.
out	*ptr*	The location at which to store the resulting value.
in	*size*	The size of the container to which *ptr* points.

Returns

A vx_status_e enumeration.

3.66 Advanced Framework API

3.66.1 Detailed Description

Describes components that are considered to be advanced.

Advanced topics include: extensions through User Kernels; Reflection and Introspection; Performace Tweaking through Hinting and Directives; and Debugging Callbacks.

Modules

- Framework: Node Callbacks

  Allows Clients to receive a callback after a specific node has completed execution.
- Framework: Performance Measurement

  Defines Performance measurement and reporting interfaces.
- Framework: Log

  Defines the debug logging interface.
- Framework: Hints

  Defines the Hints Interface.
- Framework: Directives

  Defines the Directives Interface.
- Framework: User Kernels

  Defines the User Kernels, which are a method to extend OpenVX with new vision functions.
- Framework: Graph Parameters

  Defines the Graph Parameter API.

3.67 Framework: Node Callbacks

3.67.1 Detailed Description

Allows Clients to receive a callback after a specific node has completed execution.

Callbacks are not guaranteed to be called *immediately* after the Node completes. Callbacks are intended to be used to create simple *early exit* conditions for Vision graphs using vx_action_e return values. An example of setting up a callback can be seen below:

```
    vx_graph graph = vxCreateGraph(context);
    if (graph) {
        vx_uint8 lmin = 0, lmax = 0;
        vx_uint32 minCount = 0, maxCount = 0;
        vx_scalar scalars[] = {
            vxCreateScalar(context, VX_TYPE_UINT8, &lmin),
            vxCreateScalar(context, VX_TYPE_UINT8, &lmax),
            vxCreateScalar(context, VX_TYPE_UINT32, &minCount),
            vxCreateScalar(context, VX_TYPE_UINT32, &maxCount),
        };
        vx_array arrays[] = {
            vxCreateArray(context, VX_TYPE_COORDINATES2D, 1),
            vxCreateArray(context, VX_TYPE_COORDINATES2D, 1)
        };
        vx_node nodes[] = {
            vxMinMaxLocNode(graph, input, scalars[0], scalars[1], arrays[0], arrays[1],
      scalars[2], scalars[3]),
        };
        status = vxAssignNodeCallback(nodes[0], &analyze_brightness);
        // do other
    }
```

Once the graph has been initialized and the callback has been installed then the callback itself will be called during graph execution.

```
#define MY_DESIRED_THRESHOLD (10)
vx_action analyze_brightness(vx_node node) {
    // extract the max value
    vx_action action = VX_ACTION_ABANDON;
    vx_parameter pmax = vxGetParameterByIndex(node, 2); // Max Value
    if (pmax) {
        vx_scalar smax = 0;
        vxQueryParameter(pmax, VX_PARAMETER_ATTRIBUTE_REF, &smax,
       sizeof(smax));
        if (smax) {
            vx_uint8 value = 0u;
            vxAccessScalarValue(smax, &value);
            if (value >= MY_DESIRED_THRESHOLD) {
                action = VX_ACTION_CONTINUE;
            }
            vxReleaseScalar(&smax);
        }
        vxReleaseParameter(&pmax);
    }
    return action;
}
```

Warning

> This should be used with ***extreme*** caution as it can *ruin* optimizations in the power/performance efficiency of a graph.

The callback must return a vx_action code indicating how the graph processing should proceed.

- If VX_ACTION_CONTINUE is returned, the graph will continue execution with no changes.
- If VX_ACTION_ABANDON is returned, execution is unspecified for all nodes for which this node is a dominator. Nodes that are dominators of this node will have executed. Execution of any other node is unspecified.
- If VX_ACTION_RESTART is returned, execution is unspecified for all nodes for which this node is a dominator. Nodes that are dominators of this node will have executed. Execution of any other node is unspecified. Once the graph halts it will restart execution.

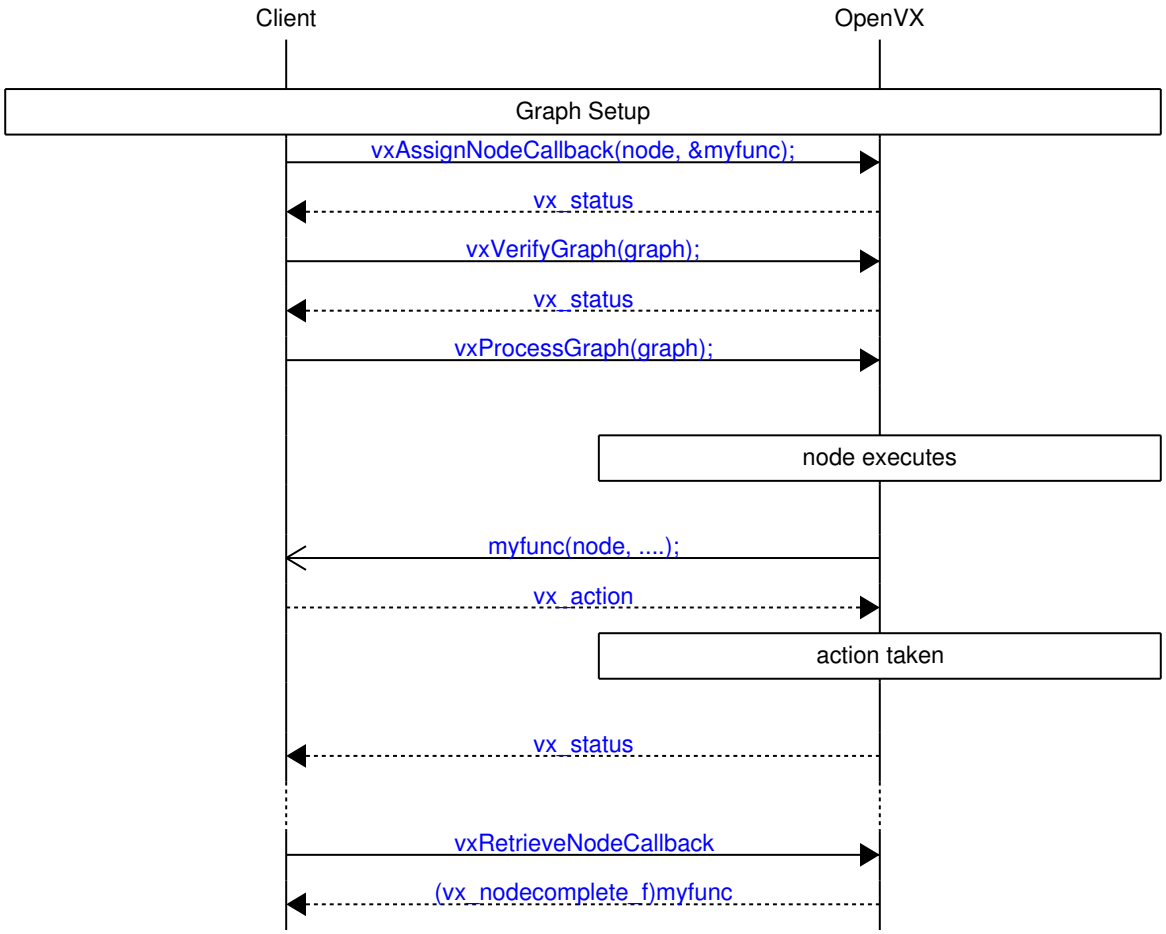


Figure 3.1: Node Callback Sequence

Typedefs

- typedef vx_enum vx_action

  The formal typedef of the response from the callback.
- typedef vx_action(∗ vx_nodecomplete_f)(vx_node node)

  A callback to the client after a particular node has completed.

Enumerations

- enum vx_action_e {
  VX_ACTION_CONTINUE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACTION << 12)) + 0x0,
  VX_ACTION_RESTART = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACTION << 12)) + 0x1,
  VX_ACTION_ABANDON = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_ACTION << 12)) + 0x2 }

  A return code enumeration from a `vx_nodecomplete_f` during execution.

Functions

- vx_status vxAssignNodeCallback (vx_node node, vx_nodecomplete_f callback)

  Assigns a callback to a node. If a callback already exists in this node, this function must return an error and the user may clear the callback by passing a NULL pointer as the callback.
- vx_nodecomplete_f vxRetrieveNodeCallback (vx_node node)

  Retrieves the current node callback function pointer set on the node.

3.67.2 Typedef Documentation

typedef vx_enum vx_action

The formal typedef of the response from the callback.

See also

vx_action_e

Definition at line 371 of file vx_types.h.

typedef vx_action(∗ vx_nodecomplete_f)(vx_node node)

A callback to the client after a particular node has completed.

See also

vx_action
vxAssignNodeCallback

Parameters

in	*node*	The node to which the callback was attached.

Returns

An action code from vx_action_e.

Definition at line 380 of file vx_types.h.

3.67.3 Enumeration Type Documentation

enum vx_action_e

A return code enumeration from a vx_nodecomplete_f during execution.

See also

vxAssignNodeCallback

Enumerator

VX_ACTION_CONTINUE Continue executing the graph with no changes.

VX_ACTION_RESTART Stop executing the graph at the current point and restart from the beginning.

VX_ACTION_ABANDON Stop executing the graph.

Definition at line 512 of file vx_types.h.

3.67.4 Function Documentation

vx_status vxAssignNodeCallback (vx_node *node,* vx_nodecomplete_f *callback*)

Assigns a callback to a node. If a callback already exists in this node, this function must return an error and the user may clear the callback by passing a NULL pointer as the callback.

Parameters

in	*node*	The reference to the node.
in	*callback*	The callback to associate with completion of this specific node.

Warning

This must be used with ***extreme*** caution as it can *ruin* optimizations in the power/performance efficiency of a graph.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Callback assigned.
VX_ERROR_INVALID_REFERENCE	The value passed as node was not a `vx_node`.

vx_nodecomplete_f vxRetrieveNodeCallback (vx_node *node*)

Retrieves the current node callback function pointer set on the node.

Parameters

`in`	*node*	The reference to the `vx_node` object.

Returns

vx_nodecomplete_f The pointer to the callback function.

Return values

NULL	No callback is set.
*	The node callback function.

3.68 Framework: Performance Measurement

3.68.1 Detailed Description

Defines Performance measurement and reporting interfaces.

In OpenVX, both vx_graph objects and vx_node objects track performance information. A client can query either object type using their respective vxQuery<Object> function with their attribute enumeration VX_<OBJECT>_ATTRIBUTE_PERFORMANCE along with a vx_perf_t structure to obtain the performance information.

```
vx_perf_t perf;
vxQueryNode(node, VX_NODE_ATTRIBUTE_PERFORMANCE, &perf, sizeof(perf
    ));
```

Data Structures

- struct vx_perf_t

  The performance measurement structure. More...

3.68.2 Data Structure Documentation

struct vx_perf_t

The performance measurement structure.

Definition at line 1292 of file vx_types.h.

Data Fields

vx_uint64	tmp	Holds the last measurement.
vx_uint64	beg	Holds the first measurement in a set.
vx_uint64	end	Holds the last measurement in a set.
vx_uint64	sum	Holds the summation of durations.
vx_uint64	avg	Holds the average of the durations.
vx_uint64	min	Holds the minimum of the durations.
vx_uint64	num	Holds the number of measurements.

3.69 Framework: Log

3.69.1 Detailed Description

Defines the debug logging interface.

The functions of the debugging interface allow clients to receive important debugging information about OpenVX.

See also

vx_status_e for the list of possible errors.

Figure 3.2: Log messages only can be received after the callback is installed.

Typedefs

- typedef void(∗ vx_log_callback_f)(vx_context context, vx_reference ref, vx_status status, vx_char string[])

  The log callback function.

Functions

- void vxAddLogEntry (vx_reference ref, vx_status status, const char ∗message,...)

  Adds a line to the log.
- void vxRegisterLogCallback (vx_context context, vx_log_callback_f callback, vx_bool reentrant)

  Registers a callback facility to the OpenVX implementation to receive error logs.

3.69.2 Function Documentation

void vxAddLogEntry (vx_reference *ref,* vx_status *status,* const char ∗ *message,* ...)

Adds a line to the log.

Parameters

in	*ref*	The reference to add the log entry against. Some valid value must be provided.
in	*status*	The status code. VX_SUCCESS status entries are ignored and not added.
in	*message*	The human readable message to add to the log.
in	...	a list of variable arguments to the message.

Note

Messages may not exceed VX_MAX_LOG_MESSAGE_LEN bytes and will be truncated in the log if they exceed this limit.

void vxRegisterLogCallback (vx_context *context,* vx_log_callback_f *callback,* vx_bool *reentrant*)

Registers a callback facility to the OpenVX implementation to receive error logs.

Parameters

in	*context*	The overall context to OpenVX.
in	*callback*	The callback function. If NULL, the previous callback is removed.
in	*reentrant*	If reentrancy flag is vx_true_e, then the callback may be entered from multiple simultaneous tasks or threads (if the host OS supports this).

3.70 Framework: Hints

3.70.1 Detailed Description

Defines the Hints Interface.

Hints are messages given to the OpenVX implementation that it may support. (These are optional.)

Enumerations

- enum vx_hint_e { VX_HINT_SERIALIZE = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_HINT << 12)) + 0x0 }

  These enumerations are given to the `vxHint` API to enable/disable platform optimizations and/or features. Hints are optional and usually are vendor-specific.

Functions

- vx_status vxHint (vx_context context, vx_reference reference, vx_enum hint)

  Provides a generic API to give platform-specific hints to the implementation.

3.70.2 Enumeration Type Documentation

enum vx_hint_e

These enumerations are given to the `vxHint` API to enable/disable platform optimizations and/or features. Hints are optional and usually are vendor-specific.

See also

`vxHint`

Enumerator

VX_HINT_SERIALIZE Indicates to the implementation that the user wants to disable any parallelization techniques. Implementations may not be parallelized, so this is a hint only.

Definition at line 538 of file vx_types.h.

3.70.3 Function Documentation

vx_status vxHint (vx_context *context,* vx_reference *reference,* vx_enum *hint*)

Provides a generic API to give platform-specific hints to the implementation.

Parameters

in	*context*	The reference to the implementation context.
in	*reference*	The reference to the object to hint at. This could be `vx_context`, `vx_graph`, `vx_node`, `vx_image`, `vx_array`, or any other reference.
in	*hint*	A `vx_hint_e` *hint* to give the OpenVX context. This is a platform-specific optimization or implementation mechanism.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No error.
VX_ERROR_INVALID_REFERENCE	If context or reference is invalid.

VX_ERROR_NOT_SUPP↩ ORTED	If the hint is not supported.

3.71 Framework: Directives

3.71.1 Detailed Description

Defines the Directives Interface.

Directives are messages given the OpenVX implementation that it must support. (These are required, i.e., non-optional.)

Enumerations

- enum vx_directive_e {
  VX_DIRECTIVE_DISABLE_LOGGING = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTIVE << 12)) + 0x0,
  VX_DIRECTIVE_ENABLE_LOGGING = (((VX_ID_KHRONOS) << 20) | (VX_ENUM_DIRECTIVE << 12)) + 0x1 }

  These enumerations are given to the `vxDirective` API to enable/disable platform optimizations and/or features. Directives are not optional and usually are vendor-specific, by defining a vendor range of directives and starting their enumeration from there.

Functions

- vx_status vxDirective (vx_context context, vx_reference reference, vx_enum directive)

  Provides a generic API to give platform-specific directives to the implementations.

3.71.2 Enumeration Type Documentation

enum vx_directive_e

These enumerations are given to the `vxDirective` API to enable/disable platform optimizations and/or features. Directives are not optional and usually are vendor-specific, by defining a vendor range of directives and starting their enumeration from there.

See also

`vxDirective`

Enumerator

VX_DIRECTIVE_DISABLE_LOGGING Disables recording information for graph debugging.

VX_DIRECTIVE_ENABLE_LOGGING Enables recording information for graph debugging.

Definition at line 553 of file vx_types.h.

3.71.3 Function Documentation

vx_status vxDirective (vx_context *context*, vx_reference *reference*, vx_enum *directive*)

Provides a generic API to give platform-specific directives to the implementations.

Parameters

in	*context*	The reference to the implementation context.
in	*reference*	The reference to the object to set the directive on. This could be `vx_context`, `vx_graph`, `vx_node`, `vx_image`, `vx_array`, or any other reference.

in	*directive*	The directive to set.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	No error.
VX_ERROR_INVALID_REFERENCE	If context or reference is invalid.
VX_ERROR_NOT_SUPPORTED	If the directive is not supported.

3.72 Framework: User Kernels

3.72.1 Detailed Description

Defines the User Kernels, which are a method to extend OpenVX with new vision functions.

User Kernels can be loaded by OpenVX and included as nodes in the graph or as immediate functions (if the Client supplies the interface). User Kernels will typically be loaded and executed on HLOS/CPU compatible targets, not on remote processors or other accelerators. This specification does not mandate what constitutes compatible platforms.

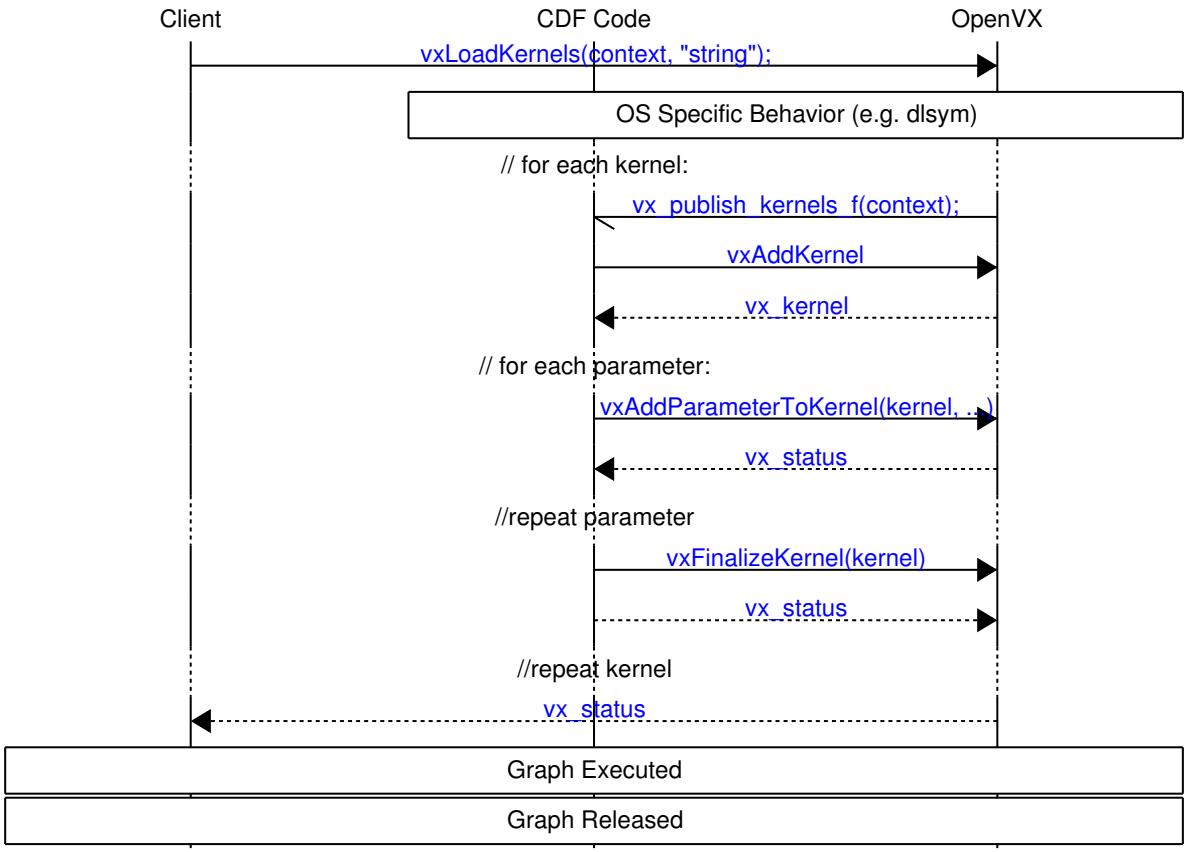


Figure 3.3: Call sequence of User Kernels Installation

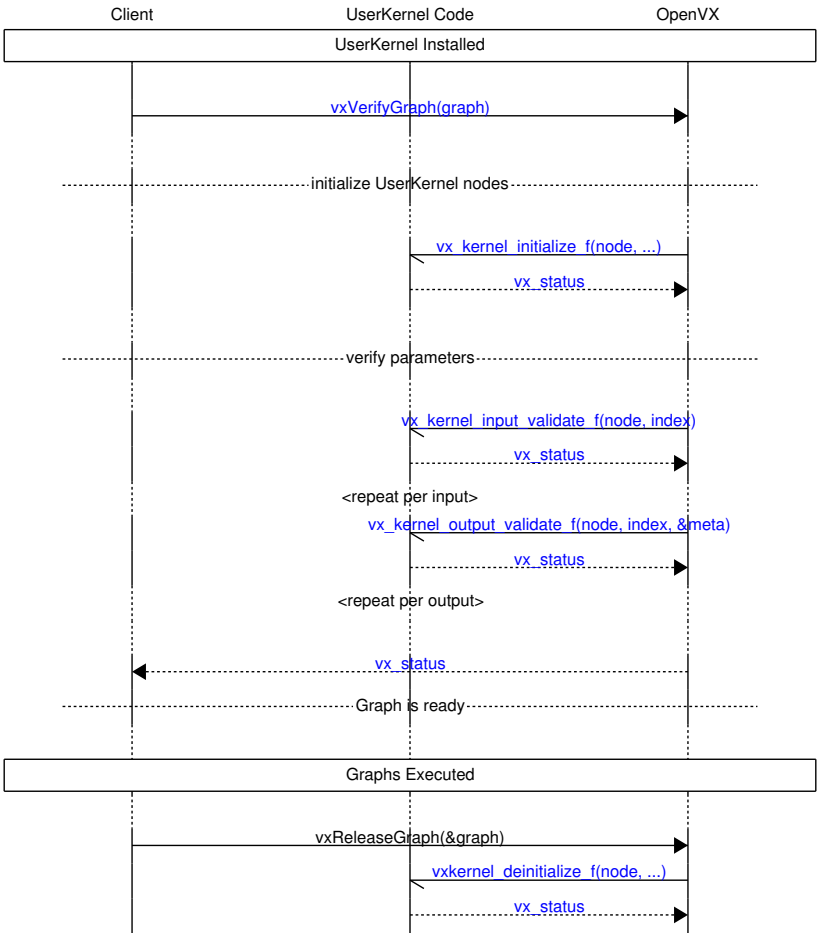


Figure 3.4: Call sequence of a Graph Verify and Release with User Kernels.

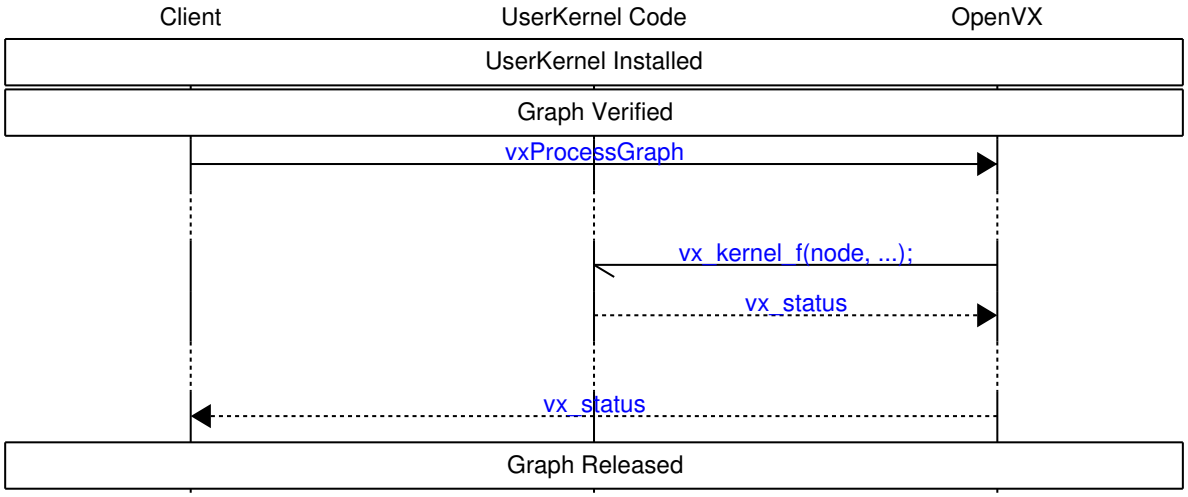


Figure 3.5: Call sequence of a Graph Execution with User Kernels

Typedefs

- typedef vx_status(∗ vx_kernel_deinitialize_f)(vx_node node, vx_reference ∗parameters, vx_uint32 num)

  The pointer to the kernel deinitializer. If the host code requires a call to deinitialize data during a node garbage collection, this function is called if not NULL.

- typedef vx_status(∗ vx_kernel_f)(vx_node node, vx_reference ∗parameters, vx_uint32 num)

  The pointer to the Host side kernel.
- typedef vx_status(∗ vx_kernel_initialize_f)(vx_node node, vx_reference ∗parameters, vx_uint32 num)

  The pointer to the kernel initializer. If the host code requires a call to initialize data once all the parameters have been validated, this function is called if not NULL.
- typedef vx_status(∗ vx_kernel_input_validate_f)(vx_node node, vx_uint32 index)

  The user-defined kernel node input parameter validation function.
- typedef vx_status(∗ vx_kernel_output_validate_f)(vx_node node, vx_uint32 index, vx_meta_format meta)

  The user-defined kernel node output parameter validation function. The function only needs to fill in the meta data structure.
- typedef struct _vx_meta_format ∗ vx_meta_format

  This structure is used to extract meta data from a validation function. If the data object between nodes is virtual, this allows the framework to automatically create the data object, if needed.
- typedef vx_status(∗ vx_publish_kernels_f)(vx_context context)

  The entry point into modules loaded by `vxLoadKernels`.

Enumerations

- enum vx_meta_format_attribute_e { VX_META_FORMAT_ATTRIBUTE_DELTA_RECTANGLE = (((VX_I↩ D_KHRONOS) << 20) | (VX_TYPE_META_FORMAT << 8)) + 0x0 }

  The meta format object attributes.

Functions

- vx_kernel vxAddKernel (vx_context context, vx_char name[VX_MAX_KERNEL_NAME], vx_enum enumeration, vx_kernel_f func_ptr, vx_uint32 numParams, vx_kernel_input_validate_f input, vx_kernel_output_↩ validate_f output, vx_kernel_initialize_f init, vx_kernel_deinitialize_f deinit)

  Allows users to add custom kernels to the known kernel database in OpenVX at run-time. This would primarily be used by the module function `vxPublishKernels`.
- vx_status vxAddParameterToKernel (vx_kernel kernel, vx_uint32 index, vx_enum dir, vx_enum data_type, vx_enum state)

  Allows users to set the signatures of the custom kernel.
- vx_status vxFinalizeKernel (vx_kernel kernel)

  This API is called after all parameters have been added to the kernel and the kernel is ready to be used.
- vx_status vxLoadKernels (vx_context context, vx_char ∗module)

  Loads one or more kernels into the OpenVX context. This is the interface by which OpenVX is extensible. Once the set of kernels is loaded new kernels and their parameters can be queried.
- vx_status vxRemoveKernel (vx_kernel kernel)

  Removes a non-finalized `vx_kernel` from the `vx_context`. Once a `vx_kernel` has been finalized it cannot be removed.
- vx_status vxSetKernelAttribute (vx_kernel kernel, vx_enum attribute, void ∗ptr, vx_size size)

  Sets kernel attributes.
- vx_status vxSetMetaFormatAttribute (vx_meta_format meta, vx_enum attribute, void ∗ptr, vx_size size)

  Allows a user to set the attributes of a `vx_meta_format` object in a kernel output validator.

3.72.2 Typedef Documentation

typedef vx_status(∗ vx_publish_kernels_f)(vx_context context)

The entry point into modules loaded by `vxLoadKernels`.

Parameters

in	*context*	The handle to the implementation context.

Note

The symbol exported from the user module must be `vxPublishKernels` in extern C format.

Definition at line 1153 of file vx_types.h.

typedef vx_status(* vx_kernel_f)(vx_node node, vx_reference *parameters, vx_uint32 num)

The pointer to the Host side kernel.

Parameters

in	*node*	The handle to the node that contains this kernel.
in	*parameters*	The array of parameter references.
in	*num*	The number of parameters.

Definition at line 1162 of file vx_types.h.

typedef vx_status(* vx_kernel_initialize_f)(vx_node node, vx_reference *parameters, vx_uint32 num)

The pointer to the kernel initializer. If the host code requires a call to initialize data once all the parameters have been validated, this function is called if not NULL.

Parameters

in	*node*	The handle to the node that contains this kernel.
in	*parameters*	The array of parameter references.
in	*num*	The number of parameters.

Definition at line 1173 of file vx_types.h.

typedef vx_status(* vx_kernel_deinitialize_f)(vx_node node, vx_reference *parameters, vx_uint32 num)

The pointer to the kernel deinitializer. If the host code requires a call to deinitialize data during a node garbage collection, this function is called if not NULL.

Parameters

in	*node*	The handle to the node that contains this kernel.
in	*parameters*	The array of parameter references.
in	*num*	The number of parameters.

Definition at line 1184 of file vx_types.h.

typedef vx_status(* vx_kernel_input_validate_f)(vx_node node, vx_uint32 index)

The user-defined kernel node input parameter validation function.

Note

This function is called once for each VX_INPUT or VI_BIDIRECTIONAL parameter index.

Parameters

in	*node*	The handle to the node that is being validated.
in	*index*	The index of the parameter being validated.

Returns

An error code describing the validation status on this parameter.

Return values

VX_ERROR_INVALID_FORMAT	The parameter format was incorrect.
VX_ERROR_INVALID_VALUE	The value of the parameter was incorrect.
VX_ERROR_INVALID_DIMENSION	The dimensionality of the parameter was incorrect.
VX_ERROR_INVALID_PARAMETERS	The index was out of bounds.

Definition at line 1200 of file vx_types.h.

typedef vx_status(∗ vx_kernel_output_validate_f)(vx_node node, vx_uint32 index, vx_meta_format meta)

The user-defined kernel node output parameter validation function. The function only needs to fill in the meta data structure.

Note

This function is called once for each VX_OUTPUT parameter index.

Parameters

in	*node*	The handle to the node that is being validated.
in	*index*	The index of the parameter being validated.
in	*ptr*	A pointer to a pre-allocated structure that the system holds. The validation function fills in the correct type, format, and dimensionality for the system to use either to create memory or to check against existing memory.

Returns

An error code describing the validation status on this parameter.

Return values

VX_ERROR_INVALID_PARAMETERS	The index is out of bounds.

Definition at line 1216 of file vx_types.h.

3.72.3 Enumeration Type Documentation

enum vx_meta_format_attribute_e

The meta format object attributes.

Enumerator

VX_META_FORMAT_ATTRIBUTE_DELTA_RECTANGLE Configures a delta rectangle during kernel output parameter validation. Use a `vx_delta_rectangle_t`.

Definition at line 942 of file vx_types.h.

3.72.4 Function Documentation

vx_status vxLoadKernels (vx_context *context,* vx_char ∗ *module*)

Loads one or more kernels into the OpenVX context. This is the interface by which OpenVX is extensible. Once the set of kernels is loaded new kernels and their parameters can be queried.

Note

When all references to loaded kernels are released, the module may be automatically unloaded.

Parameters

in	*context*	The reference to the implementation context.
in	*module*	The short name of the module to load. On systems where there are specific naming conventions for modules, the name passed should ignore such conventions. For example: `libxyz.so` should be passed as just `xyz` and the implementation will *do the right thing* that the platform requires.

Note

This API uses the system pre-defined paths for modules.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	No errors.
VX_ERROR_INVALID_REFERENCE	If the context is not a `vx_context`.
VX_ERROR_INVALID_PARAMETERS	If any of the other parameters are incorrect.

See also

vxGetKernelByName

vx_kernel vxAddKernel (vx_context *context*, vx_char *name[VX_MAX_KERNEL_NAME]*, vx_enum *enumeration*, vx_kernel_f *func_ptr*, vx_uint32 *numParams*, vx_kernel_input_validate_f *input*, vx_kernel_output_validate_f *output*, vx_kernel_initialize_f *init*, vx_kernel_deinitialize_f *deinit*)

Allows users to add custom kernels to the known kernel database in OpenVX at run-time. This would primarily be used by the module function `vxPublishKernels`.

Parameters

in	*context*	The reference to the implementation context.
in	*name*	The string to use to match the kernel.
in	*enumeration*	The enumerated value of the kernel to be used by clients.
in	*func_ptr*	The process-local function pointer to be invoked.
in	*numParams*	The number of parameters for this kernel.
in	*input*	The pointer to `vx_kernel_input_validate_f`, which validates the input parameters to this kernel.
in	*output*	The pointer to `vx_kernel_output_validate_f` , which validates the output parameters to this kernel.
in	*init*	The kernel initialization function.
in	*deinit*	The kernel de-initialization function.

Returns

`vx_kernel`

Return values

0	Indicates that an error occurred when adding the kernel.
*	Kernel added to OpenVX.

vx_status vxFinalizeKernel (vx_kernel *kernel*)

This API is called after all parameters have been added to the kernel and the kernel is *ready* to be used.

Parameters

in	*kernel*	The reference to the loaded kernel from vxAddKernel.

Returns

A vx_status_e enumeration. If an error occurs, the kernel is not available for usage by the clients of OpenVX. Typically this is due to a mismatch between the number of parameters requested and given.

Precondition

vxAddKernel and vxAddParameterToKernel

vx_status vxAddParameterToKernel (vx_kernel *kernel*, vx_uint32 *index*, vx_enum *dir*, vx_enum *data_type*, vx_enum *state*)

Allows users to set the signatures of the custom kernel.

Parameters

in	*kernel*	The reference to the kernel added with vxAddKernel.
in	*index*	The index of the parameter to add.
in	*dir*	The direction of the parameter. This must be a value from vx_direction_e.
in	*data_type*	The type of parameter. This must be a value from vx_type_e.
in	*state*	The state of the parameter (required or not). This must be a value from vx_parameter_state_e.

Returns

A vx_status_e enumerated value.

Return values

VX_SUCCESS	Parameter is successfully set on kernel.
VX_ERROR_INVALID_REFERENCE	The value passed as kernel was not a vx_kernel.

Precondition

vxAddKernel

vx_status vxRemoveKernel (vx_kernel *kernel*)

Removes a non-finalized vx_kernel from the vx_context. Once a vx_kernel has been finalized it cannot be removed.

Parameters

in	*kernel*	The reference to the kernel to remove. Returned from vxAddKernel.

Note

Any kernel enumerated in the base standard cannot be removed; only kernels added through vxAddKernel can be removed.

Returns

A vx_status_e enumeration.

Return values

VX_ERROR_INVALID_REFERENCE	If an invalid kernel is passed in.
VX_ERROR_INVALID_PARAMETER	If a base kernel is passed in.

vx_status vxSetKernelAttribute (vx_kernel *kernel*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Sets kernel attributes.

Parameters

in	*kernel*	The reference to the kernel.
in	*attribute*	The enumeration of the attributes. See vx_kernel_attribute_e.
in	*ptr*	The pointer to the location from which to read the attribute.
in	*size*	The size of the data area indicated by *ptr* in bytes.

Note

After a kernel has been passed to vxFinalizeKernel, no attributes can be altered.

Returns

A vx_status_e enumeration.

vx_status vxSetMetaFormatAttribute (vx_meta_format *meta*, vx_enum *attribute*, void * *ptr*, vx_size *size*)

Allows a user to set the attributes of a vx_meta_format object in a kernel output validator.

Parameters

in	*meta*	The reference to the vx_meta_format object to set.
in	*attribute*	Use attributes from other objects that match the parameter type or from vx_meta_format_attribute_e.
in	*ptr*	The input pointer of the value to set on the meta format object.
in	*size*	The size of the object to which *ptr* points.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	The attribute was set.
VX_ERROR_INVALID_REFERENCE	meta was not a vx_meta_format.
VX_ERROR_INVALID_PARAMETER	size was not correct for the type needed.
VX_ERROR_NOT_SUPPORTED	the object attribute was not supported on the meta format object.
VX_ERROR_INVALID_TYPE	attribute type did not match known meta format type.

3.73 Framework: Graph Parameters

3.73.1 Detailed Description

Defines the Graph Parameter API.

Graph parameters allow Clients to create graphs with Client settable parameters. Clients can then create Graph creation methods (a.k.a. *Graph Factories*). When creating these factories, the client will typically not be able to use the standard Node creator functions such as `vxSobel3x3Node` but instead will use the *manual* method via `vxCreateGenericNode`.

```
vx_graph vxCornersGraphFactory(vx_context context)
{
    vx_status  status = VX_SUCCESS;
    vx_uint32  i;
    vx_float32 strength_thresh = 10000.0f;
    vx_float32 r = 1.5f;
    vx_float32 sensitivity = 0.14f;
    vx_int32 window_size = 3;
    vx_int32 block_size = 3;
    vx_enum channel = VX_CHANNEL_Y;
    vx_graph graph = vxCreateGraph(context);
    if (graph)
    {
        vx_image virts[] = {
            vxCreateVirtualImage(graph, 0, 0,
      VX_DF_IMAGE_VIRT),
            vxCreateVirtualImage(graph, 0, 0,
      VX_DF_IMAGE_VIRT),
        };
        vx_kernel kernels[] = {
            vxGetKernelByEnum(context,
      VX_KERNEL_CHANNEL_EXTRACT),
            vxGetKernelByEnum(context, VX_KERNEL_MEDIAN_3x3),
            vxGetKernelByEnum(context, VX_KERNEL_HARRIS_CORNERS),
        };
        vx_node nodes[dimof(kernels)] = {
            vxCreateGenericNode(graph, kernels[0]),
            vxCreateGenericNode(graph, kernels[1]),
            vxCreateGenericNode(graph, kernels[2]),
        };
        vx_scalar scalars[] = {
            vxCreateScalar(context, VX_TYPE_ENUM, &channel),
            vxCreateScalar(context, VX_TYPE_FLOAT32, &strength_thresh),
            vxCreateScalar(context, VX_TYPE_FLOAT32, &r),
            vxCreateScalar(context, VX_TYPE_FLOAT32, &sensitivity),
            vxCreateScalar(context, VX_TYPE_INT32, &window_size),
            vxCreateScalar(context, VX_TYPE_INT32, &block_size),
        };
        vx_parameter parameters[] = {
            vxGetParameterByIndex(nodes[0], 0),
            vxGetParameterByIndex(nodes[2], 6)
        };
        // Channel Extract
        status |= vxAddParameterToGraph(graph, parameters[0]);
        status |= vxSetParameterByIndex(nodes[0], 1, (
      vx_reference)scalars[0]);
        status |= vxSetParameterByIndex(nodes[0], 2, (
      vx_reference)virts[0]);
        // Median Filter
        status |= vxSetParameterByIndex(nodes[1], 0, (
      vx_reference)virts[0]);
        status |= vxSetParameterByIndex(nodes[1], 1, (
      vx_reference)virts[1]);
        // Harris Corners
        status |= vxSetParameterByIndex(nodes[2], 0, (
      vx_reference)virts[1]);
        status |= vxSetParameterByIndex(nodes[2], 1, (
      vx_reference)scalars[1]);
        status |= vxSetParameterByIndex(nodes[2], 2, (
      vx_reference)scalars[2]);
        status |= vxSetParameterByIndex(nodes[2], 3, (
      vx_reference)scalars[3]);
        status |= vxSetParameterByIndex(nodes[2], 4, (
      vx_reference)scalars[4]);
        status |= vxSetParameterByIndex(nodes[2], 5, (
      vx_reference)scalars[5]);
        status |= vxAddParameterToGraph(graph, parameters[1]);

        for (i = 0; i < dimof(scalars); i++)
        {
            vxReleaseScalar(&scalars[i]);
        }
        for (i = 0; i < dimof(virts); i++)
```

```
        {
            vxReleaseImage(&virts[i]);
        }
        for (i = 0; i < dimof(kernels); i++)
        {
            vxReleaseKernel(&kernels[i]);
        }
        for (i = 0; i < dimof(nodes);i++)
        {
            vxReleaseNode(&nodes[i]);
        }
        for (i = 0; i < dimof(parameters); i++)
        {
            vxReleaseParameter(&parameters[i]);
        }
    }
    return graph;
}
```

Some data are contained in these Graphs and do not become exposed to Clients of the factory. This allows ISVs or Vendors to create custom IP or IP-sensitive factories that Clients can use but may not be able to determine what is inside the factory. As the graph contains internal references to the data, the objects will not be freed until the graph itself is released.

Functions

- vx_status vxAddParameterToGraph (vx_graph graph, vx_parameter parameter)

  Adds the given parameter extracted from a `vx_node` to the graph.
- vx_parameter vxGetGraphParameterByIndex (vx_graph graph, vx_uint32 index)

  Retrieves a `vx_parameter` from a `vx_graph`.
- vx_status vxSetGraphParameterByIndex (vx_graph graph, vx_uint32 index, vx_reference value)

  Sets a reference to the parameter on the graph. The implementation must set this parameter on the originating node as well.

3.73.2 Function Documentation

vx_status vxAddParameterToGraph (vx_graph *graph*, vx_parameter *parameter*)

Adds the given parameter extracted from a `vx_node` to the graph.

Parameters

in	*graph*	The graph reference that contains the node.
in	*parameter*	The parameter reference to add to the graph from the node.

Returns

A `vx_status_e` enumeration.

Return values

VX_SUCCESS	Parameter added to Graph.
VX_ERROR_INVALID_REFERENCE	The parameter is not a valid `vx_parameter`.
VX_ERROR_INVALID_PARAMETER	The parameter is of a node not in this graph.

vx_status vxSetGraphParameterByIndex (vx_graph *graph*, vx_uint32 *index*, vx_reference *value*)

Sets a reference to the parameter on the graph. The implementation must set this parameter on the originating node as well.

Parameters

in	*graph*	The graph reference.
in	*index*	The parameter index.
in	*value*	The reference to set to the parameter.

Returns

A vx_status_e enumeration.

Return values

VX_SUCCESS	Parameter set to Graph.
VX_ERROR_INVALID_REFERENCE	The value is not a valid vx_reference.
VX_ERROR_INVALID_PARAMETER	The parameter index is out of bounds or the dir parameter is incorrect.

vx_parameter vxGetGraphParameterByIndex (vx_graph *graph*, vx_uint32 *index*)

Retrieves a vx_parameter from a vx_graph.

Parameters

in	*graph*	The graph.
in	*index*	The index of the parameter.

Returns

vx_parameter reference.

Return values

0	if the index is out of bounds.
*	The parameter reference.

Bibliography

[1] Jean-Yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker description of the algorithm, 2000. 83

[2] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, June 1986. 46

[3] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European Conference on Computer Vision*, volume 1, pages 430–443, May 2006. 26, 64

[4] Edward Rosten, Reid Porter, and Tom Drummond. Faster and better: A machine learning approach to corner detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 32:105–119, October 2010. 26, 64

Index