OpenVX tutorial
CVPR 2015

Editor : Thierry Lepley

Introduction

This document contains necessary information for the OpenVX tutorial at CVPR 2015. The objective of this tutorial is to allow attendees to tackle the most important aspects of OpenVX in a practical way, by developing a simple feature tracker implementation in 3 steps.

OpenVX is a computer vision (CV) standard created by the Khronos consortium with the participation of leading companies in the mobile/embedded computer vision area. OpenVX defines a set of CV functions and a framework for managing CV data objects and processing or producing them with CV functions. OpenVX version 1.0 was released in October 2014, and in June 2015, version 1.0.1 was released, replacing the earlier 1.0 version. OpenVX has been designed with performance and portability in mind. Target hardware architectures can be as complex as heterogeneous System-on-chips (SoCs) containing CPU, GPU, DSP, and dedicated accelerators with a complex memory architecture. While the primary target of OpenVX is for use in mobile and embedded fields, it is suited and can further be extended for Desktop and larger-scale compute platforms.

This tutorial has been prepared with contributions from Elif Albuz (NVIDIA), Victor Eruhimov (Itseez), Radha Giduthuri (AMD), Thierry Lepley (NVIDIA), Kari Pulli (Light), Colin Tracey (NVIDIA), and Vlad Vinogradov (Itseez).

Material required for the tutorial

To enable everybody to participate in the tutorial, we will use the same Linux Ubuntu environment that can be hosted by an x86-based 64-bit OS. A VirtualBox image has been created for this purpose.

We will also use the Khronos 'Sample' implementation that is a prototype open-source OpenVX implementation provided by the Khronos consortium. This Khronos 'Sample' implementation is a non-optimized functional OpenVX implementation that is not as robust as a production implementation, but it allows everybody to learn the use of the API. Participants who already have access to a production OpenVX implementation may prefer to use it instead.

Here is the material needed for this tutorial:

- *Ubuntu_64_Virtualbox.7z*
  This is a Linux Ubuntu 14.04 64-bit VirtualBox image properly configured for the tutorial (e.g., OpenVX sample implementation, OpenCV, and tutorial files are pre-installed).
- VirtualBox can be downloaded here:
  http://www.oracle.com/technetwork/server-storage/virtualbox/downloads/index.html
- *The OpenVX 1.0.1 specification*
  The tutorial is based on OpenVX 1.0.1, that differs from OpenVX 1.0 for some aspects. It can be found here: https://www.khronos.org/openvx.

Note: We strongly advise you to install VirtualBox, download the Ubuntu image (> 1 GB), and check that everything works well, prior to the tutorial session at CVPR.

Copyright ©2015 The Khronos Group Inc.

Setup

Booting the virtual Ubuntu image

First uncompress *Ubuntu_64_Virtualbox.7z* image and launch VirtualBox. The Ubuntu virtual image is configured with 2GB of memory, 2 cores, and 128 MB of video memory. Depending on your system, you may want to adjust this configuration. Note that 3D acceleration has been disabled as it is still an experimental feature in VirtualBox and may create issues (especially when applications are not in full-screen mode).

Once the Virtual machine has booted, it should log automatically as *openvx* user

- user: *openvx*
- password: *openvx*

File system structure

In the user 'home' directory (`/home/openvx`), you will find directories related to the tutorial:

★ `tutorial_CVPR`
It contains the source files of the tutorial.
  - For each step of the tutorial, there is a '`step<n>_start`' directory that contains the source files that can be started from, and a '`step<n>`' directory that contains an example of what is expected as final result for this step. Each step can be compiled from `make` or `cmake`.
  - There is also an additional 'other_example' directory that will not be used during the tutorial, but that shows another example of OpenVX feature tracker written in a more compact way. It also imports input images differently by copying them from OpenCV to OpenVX.

★ `tutorial_videos`
This directory contains a set of videos to be used in the tutorial.

★ `tutorial_slides`
This directory contains the set of slides presenting the tutorial.

★ `khronos_sample`
This directory contains a specific build of the Khronos 'Sample' implementation for the tutorial with OpenVX headers. Note that this is not a formal release from Khronos.

A set of useful tools for this tutorial have also been pre-installed:

★ Build: gcc, make, and cmake
★ IDE: QtCreator is an IDE that is compatible with both make and cmake builds
★ OpenCV : it is used for image capture and display

Copyright ©2015 The Khronos Group Inc.

Setup check

To check that everything works well, please go to the `step1_start` example, compile, and execute it with the build system you prefer.

The input video file should simply be displayed.

Option 1: `make` build

```
cd ~/tutorial_CVPR/step1_start
make
./step1_start ~/tutorial_videos/Megamind.avi
# (type ESC on the keyboard to stop the video)
make clean
```

Option 2:`cmake` build

```
cd ~/tutorial_CVPR
mkdir step1_start_build && cd step1_start_build
cmake ../step1_start
make
./step1_start ~/tutorial_videos/Megamind.avi
# (press ESC on the keyboard to stop the video)
make clean
```

Option 3: build with qtcreator

Note: This build option is recommended only for people already familiar with the qtcreator IDE

```
qtcreator &

# 1/ File-> 'Open File or Project'
# 2/ choose tutorial_CVPR/step1_start/CMakelists.txt, then click 'Open'
# 3/ choose the default build directory: tutorial_CVPR/step1_start-build
#    by clicking directly 'Next'
# 4/ click 'Run CMake', then "Finish"
# 5/ click 'Projects' in the left tab, move to 'build & run', then
#   'desktop run' tabs. Enter ~/tutorial_videos/Megamind.avi in the
#    'Arguments:' box
# 6/ click on the green arrow (Run) in the left tab to execute the test
```

Copyright ©2015 The Khronos Group Inc.

Step 1: Detect and display Harris corners

The goals of this first step are:

a) Read a video with OpenCV.
b) Create an OpenVX context and declare a log callback.
c) Import the RGB OpenCV images into OpenVX.
d) Detect Harris corners with OpenVX.
   As a transition from OpenCV to OpenVX, we will use the OpenVX *immediate execution mode* (functions with `vxu`-prefix):
   - Convert RGB into YUV : `vxuColorConvert`
   - Extract the Y channel : `vxuChannelExtract`
   - Compute keypoints : `vxuHarrisCorners`
e) Display keypoints with OpenCV.

Here is a set of parameters for Harris corner proposed for this tutorial:

- *strength_thresh* : 0.0005f
- *min_distance* : 5.0f
- *sensitivity* : 0.04f
- *gradient_size* : 3
- *block_size* : 3

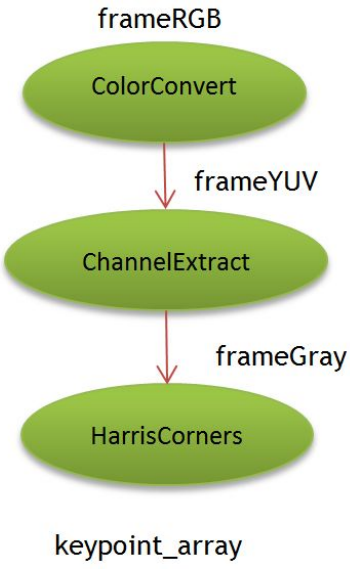

Figure 1 : Processing of each frame for step 1 (immediate execution mode).

Copyright ©2015 The Khronos Group Inc.

Here is a list of useful notes for this tutorial step:

- **The OpenVX world : the vx_context object**
  Prior to any OpenVX operation, an OpenVX *context* must be created. This context is the OpenVX 'world' in which all OpenVX objects for this tutorial will live.
- **vx_<object> and vx_reference**
  Functions taking `vx_reference` as parameter can actually take any object as parameter, if properly casted to `vx_reference`.
  Example : `vx_status status = vxGetStatus((vx_reference)context);`
- **OpenVX error management**
  Errors may occur when executing functions of the OpenVX API for various reasons (incorrect parameter, lack of resource, etc.). There are 2 complementary types of error handling in OpenVX:
  - Most of functions return a `vx_status` status. If this status is `VX_SUCCESS`, then no error occurred. Any other status code signals an error.
  - Object creation functions (for example `vxCreateImage`) return an object reference. This object may be valid or not. The way to determine it is to call `vxGetStatus` that will return a status code. If it returns `VX_SUCCESS`, then the object if valid. Any other status code signals an error in object creation.

  Note : In order to simplify error checking, `cv_vx_utility.hpp` provides 2 macros
  - `CHECK_VX_STATUS` to check a returned status.
  - `CHECK_VX_OBJECT` to check object creation.
- **Log callback**
  - A log callback is the way for the OpenVX implementation to provide more information to the developer than the status returned by functions, for example about errors, but not only. A log callback can be registered in the context via the `vxRegisterLogCallback` function.
  - Note that the log messages are not standardized and may differ across implementations.
- **`vx_array`, `vx_image`, and `vx_scalar` objects**
  - The `vxuHarrisCorners` function takes as parameters a set of `vx_image`, `vx_array`, and `vx_scalar` objects that must be created and correctly initialized before calling this function.
  - Note: a `vx_array` object has a variable number of elements, but a fixed capacity (maximum number of elements) that is given at creation time. The capacity of the output array of keypoint (*corners* parameter) informs the Harris corner function about the maximum number of keypoints that can be detected.
- **Object life cycle and release**
  An OpenVX object must be released explicitly by the application when a reference to the object is no more needed. Releasing an object doesn't necessarily mean that the object will be destroyed, since it could be still used by another object. The exception is a `vx_context` that will be destroyed (together with all the objects it contains) when released.

Copyright ©2015 The Khronos Group Inc.

- **Image data access**
  Pixel data of `vx_image` objects can be accessed by the application by using `vxAccessImagePatch` and `vxCommitImagePatch` functions.
  - `vxAccessImagePatch` is the way to request access to the pixels of an image. It can be done in 2 modes: *map* (OpenVX returns a pointer and the memory layout to access pixel data) and *copy* (the application provides a pointer and the memory layout to inform where OpenVX must copy the pixel data). In this tutorial, we will only use the 'map' mode.
  - `vxCommitImagePatch` must be called after the image pixels were accessed and prior to calling an OpenVX CV function that uses this image. `vxCommitImagePatch` commits the potential changes done by the application and gives back the control of the image to the OpenVX implementation.
  - `vx_rectangle_t` is a structure that describes the image patch that is accessed. Note that `start_x` and `start_y` give the coordinate of the top-left pixel included in the patch, while `end_x` and `end_y` give the coordinate of the bottom-right pixel excluded from the patch.
  - `vx_imagepatch_addressing_t` is a structure that describes the memory layout of the pixels. The most important fields of this structure are `stride_x` and `stride_y` that give the distance in bytes between the pixels across columns and rows in the image.
    - Note that `scale_x`, `scale_y`, `step_x`, and `step_y` are only relevant for images with sub-sampled image planes. They will not be used in this tutorial since an RGB image has a single interleaved plane where each pixel contains RGB components.
    - Note that *step* doesn't have the same meaning in OpenVX and OpenCV. The *step* notion in OpenCV is similar to the *stride_y* notion in OpenVX.
- **Image created from a handle**
  - OpenCV video provides an RGB image that must be imported into OpenVX by creating a `vx_image` object. Creating an image from handle (`vxCreateImageFromHandle`) is the best way to import an external image into OpenVX since it doesn't force a copy.
  - The handle is provided at image creation time and the usage of the memory area behind the pointer is transferred to OpenVX. The application should not access the image memory directly after the image creation. To access the image data created from a handle, the application must request the access explicitly using `vxAccessImagePatch` and `vxCommitImagePatch`, like for standard images. Note that when accessing an image created from handle in map mode, OpenVX will map the pixels at the same address and with the same memory layout as it was provided by the application at the image creation time.
- **Immediate execution mode**
  'vxu' prefixed functions are the immediate execution version of CV functions supported by

Copyright ©2015 The Khronos Group Inc.

OpenVX. This mode is similar to OpenCV since the CV functions are executed immediately and results are available at function execution completion.

Copyright ©2015 The Khronos Group Inc.

Step 2: keypoint tracking

The goal of this second step is to start from the outcome of step1, by computing keypoints for the first frame and to track the image feature across the other frames (OpticalFlowPyrLK). Feature tracking will be done by creating an image processing graph by combining nodes that correspond to CV functions. This approach is particularly useful when the same processing is repeated for each successive frame.

Tracking keypoints requires accessing data related to the current and the previous image. For this purpose, we need to add delay slots for both the *image pyramid* and the *array of keypoints*.

Here are the parameters for PyrLK optical flow proposed for this tutorial:

- *termination* : VX_TERM_CRITERIA_BOTH
- *epsilon* : 0.01f
- *num_iterations* : 5
- *use_initial_estimate* : vx_false_e
- *window_dimension* : 6

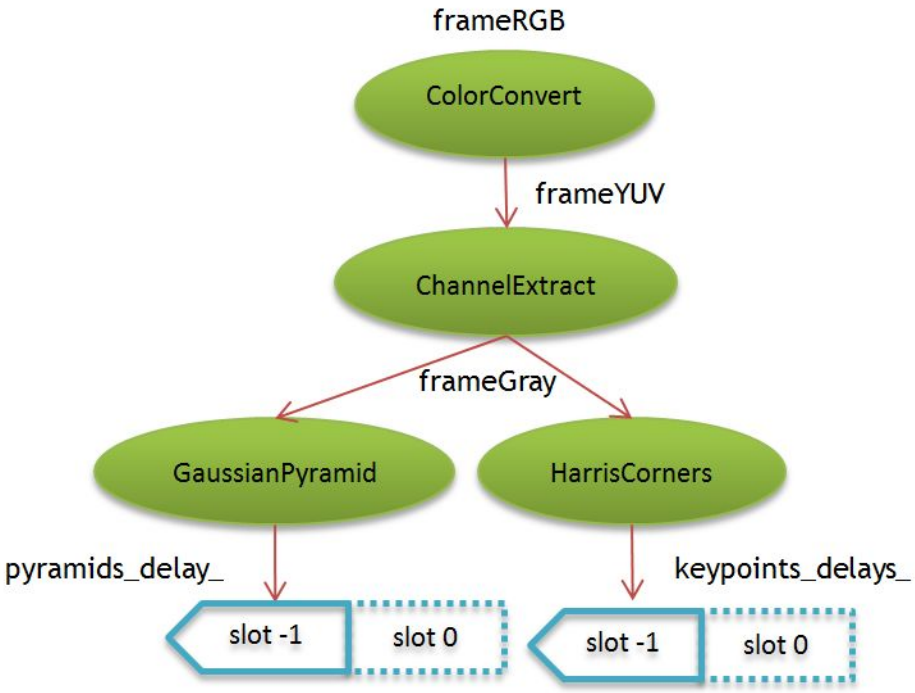

Figure 2 : Processing of the first frame for step 2 (immediate execution mode).

Copyright ©2015 The Khronos Group Inc.

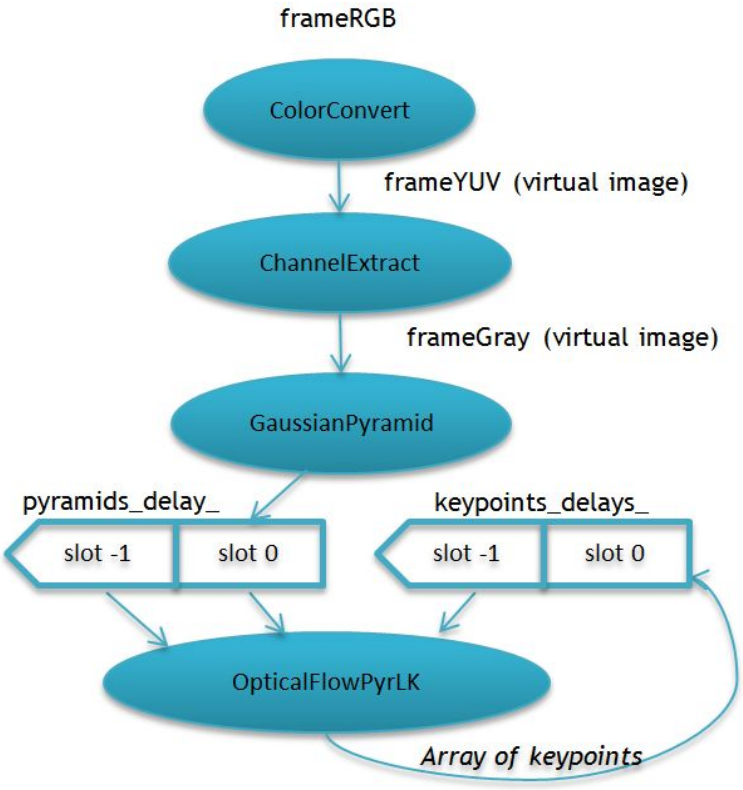

Figure 3 : Tracking graph for step 2 (for all frames except the first one).

Here is a list of useful notes for this tutorial step:

- **Graph**
  - *Introduction*: The graph construct is the best choice when the algorithm must be applied for each successive frame of a video stream, as it enables more optimizations than the immediate execution mode that was used in step 1.
  - *Building*: A graph is built by adding nodes to it. A node is a CV function. The edges of the graph are implicitly created from definition-usage relationships between node parameters. These relationships are independent from the order in which the nodes are created, and the node creation order doesn't need to follow a valid execution order.
  - *Graph verification*: The graph must be verified before executing it. At the graph verification, node parameters are checked, and potentially some optimizations on the graph execution may be performed. This verification can be explicitly requested by the application (`vxVerifyGraph`), or implicitly performed by OpenVX when needed. Changing the graph inputs and outputs (with `vxSetParameterByIndex` on nodes) without changing their meta-data (image dimensions for instance) should not trigger a re-verification (but this depends on the implementation).
  - *Graph execution*: the graph can be executed synchronously (`vxProcessGraph`) or asynchronously (`vxScheduleGraph`). We will use synchronous execution in the tutorial.

Copyright ©2015 The Khronos Group Inc.

  - *Graph and node release*: nodes of the graph are automatically released when the application releases the graph. The application then doesn't need to release nodes itself. And if it does, it must be done prior to graph release; otherwise the result is undefined, since nodes may have already been destroyed.

- **Pyramid object**
  OpticalFlowPyrLK operates on `vx_pyramid` objects. A pyramid is a collection of images (called levels) whose dimensions are determined by a scaling factor. It can be created with the `vxCreatePyramid` function. Each level of the pyramid can be accessed individually.
- **Virtual Images**
  Virtual objects are specific to graphs. They are used to store intermediate data in a graph, data that don't need to be accessed by the application. The only difference with standard objects is that their content can't be accessed by the application, which enables more optimizations. Images, arrays, and pyramids can be created as virtual objects.
- **Delay**
  - *Introduction*: A delay object is a circular buffer that keeps some history of the data. A delay contains a numbers of slots (elements of the circular buffer), and each slot is identified by an index that gives its age: slot 0 is the current, -1 is one frame old, -2 is two frames old, and so on.
  - *Delay creation*: a delay object is created in a few steps.
    - First, an exemplar of the data object requested in delay slots must be created. The exemplar is needed since OpenVX has a single function (taking `vx_reference` as parameter) that creates delays for all types of data objects as slots.
    - Then, the delay is created from this exemplar data object with the `vxCreateDelay` function. This function will extract from the exemplar data object the meta-data necessary to the creation of the actual data objects in delay slots (width, height, format for `vx_image` object).
    - Then, the exemplar data object can be released if the application does not use it any more.
    - Example :
      ```
      vx_image exemplar = vxCreateImage(context, width, height, VX_DF_IMAGE_U8);
      vxCreateDelay(context_, (vx_reference)exemplar, 2);
      vxReleaseImage(&exemplar);
      ```
    - *A note on memory consumption*: creating an exemplar data object that is only used to create a delay object should not consume much memory since an OpenVX implementation can delay most of the memory allocation until the object data is actually accessed, which in this case will never happen. For instance, if the exemplar is a 1920x1080 image and if the image is released just after the creation of the delay, then the OpenVX implementation doesn't need to allocate the memory for storing the 1920x1080 pixel array, and only needs to allocate memory for the meta data.

Copyright ©2015 The Khronos Group Inc.

- *Delay aging*: delays must be *aged* explicitly by the application. Aging will shift the content of delay slots, moving for instance the content of slot 0 into slot -1.

Copyright ©2015 The Khronos Group Inc.

Step 3 : Recompute keypoints when not enough are tracked

Usually, feature trackers will recompute keypoints when too many of them have been lost by the optical flow tracking (for instance 20% of the original keypoints). This is the goal of the last step of this tutorial. Keypoints will be recomputed in the graph by creating a node instantiated from a user kernel that computes the number of tracked keypoints and calls HarrisCorners in the immediate execution mode if needed.

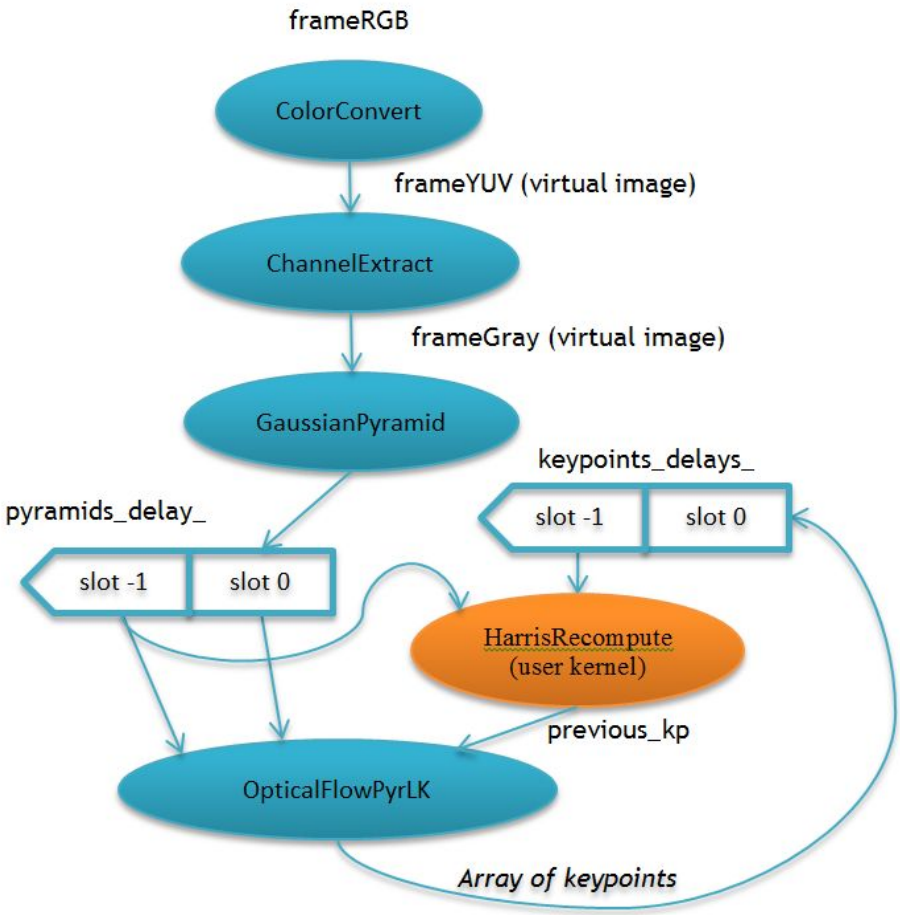

Figure 4 : Tracking graph for step 3 (for all frames except the first one).

Copyright ©2015 The Khronos Group Inc.

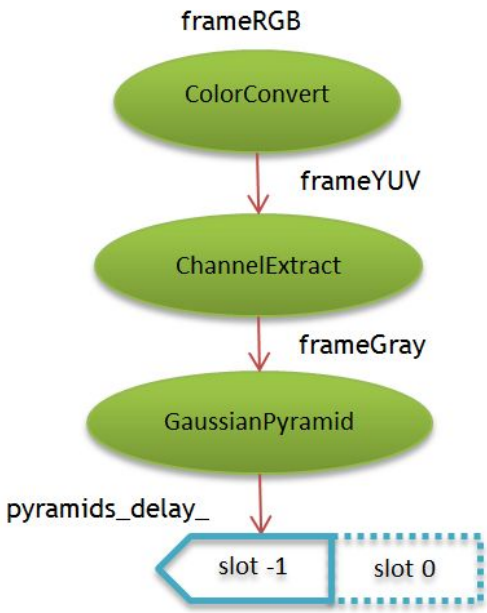

Figure 5 : Processing of the first frame for step 3 (immediate execution mode).

Here are useful notes for this tutorial step:

- **User kernel**
  - *Introduction*: user kernels are a way for the application to extend the set of CV functions that can be instantiated as a graph node. Thanks to user kernels, complex algorithms can be modeled as a single graph even if standard OpenVX CV functions do not cover all needs of the algorithm.
  - *User kernel functions*: In order to create a user kernel, the application must provide a set a functions (process function, parameter validators, optionally init/deinit functions), and proceed to the registration of the kernel.
    - The *Process function* will be called at graph execution time.
    - *Parameter validators* are called at graph verification time:
      - The *Input graph validator* verifies input parameters and returns an error status in case of error.
      - The *Output graph validator* sets a `vx_meta_format` object that describes the meta-data (e.g., image dimensions) expected for the node output. The actual check is done by the OpenVX implementation that will compare the information given into the `vx_meta_format` object with the actual output object and raise an error if needed.
    - *Initialization and de-initialization functions* are called at graph verification time and are useful to perform some setup tasks for the node. These functions are optional and we will not use them in this tutorial.
  - *User kernel registration*: the registration of a user kernel is done in 3 steps:
    - Step 1: a `vx_kernel` object is created with the `vxAddKernel` function. The name of the kernel, an enum that uniquely identifies it, the number of

Copyright ©2015 The Khronos Group Inc.

parameters, process and validation functions as well as init/deinit functions must be provided at that time. `NULL` is given if init/deinit functions are not defined for the user kernel.
    - Step 2: parameters of the kernel must be declared one-by-one with the `vxAddParameterToKernel` function. For each kernel, its direction (input or output), its type, and state (mandatory or optional) must be specified.
    - Step 3: the kernel is made available to the application by *finalizing* its creation with the `vxFinalizeKernel` function.
  - Node creation from a user kernel:
    - Step 1: a reference to the user `vx_kernel` object must be retrieved either directly from the kernel registration, or by using `vxGetKernelByEnum` or `vxGetKernelByName` functions.
    - Step 2: a node is created from the kernel with `vxCreateGenericNode`.
    - Step 3: the node parameters are set by using `vxSetParameterByIndex`.
- **Array access**
  In order to count points that are lost in the tracking, the user-defined kernel needs to access the array of keypoints. This can be done with `vxAccessArrayRange` and `vxAccessCommitRange`. The access philosophy is quite similar to the image access that is discussed in the tutorial step 1. We will use the map mode in this tutorial.
- **vx_keypoint_t**
  HarrisCorner and OpticalFlowPyrLK operate on arrays of `vx_keypoint_t`. Among the fields of this structure, the most useful in this tutorial are `x`, `y` for coordinates and `tracking_status` that is set to 1 by corner detectors and set to 0 by optical flow when tracking is lost for a point.

Copyright ©2015 The Khronos Group Inc.