CHAPTER 6

Basic image transformations

Contents

At the time of writing this book, computer vision is developing with a very fast pace, so building an API to account for rapid changes in the field is an extremely challenging task. However, there is a pretty well-defined set of methods that many computer vision algorithms use, and being able to execute these methods efficiently is a core requirement for building a real-time embedded product. It is no surprise that basic image processing functions are a significant part of OpenVX. This chapter will go over the `vx_image` object, which encapsulates images, discuss image properties such as color space and region of interest, and talk about linear filtering border modes. We will then look at Hough transform, remapping and its application to fast undistorted transformation, as well as perspective transformations in OpenVX.

6.1 OpenVX image object

OpenVX is a computer vision API, and it is no surprise that an image object is a first-class citizen. As we observed in Chapter 2 (see Example 1), we need fairly limited information to create an image: a context, width, height, and color space. For example:

https://doi.org/10.1016/B978-0-12-816425-9.00012-7

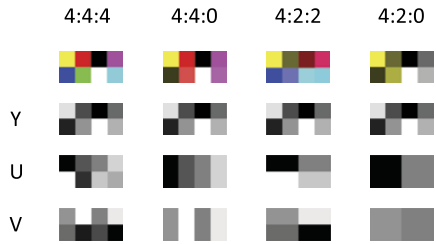


Figure 6.1 An illustration of chroma subsampling.

```
vx_context context = vxCreateContext();
vx_image image = vxCreateImage(context, 640, 480, VX_DF_IMAGE_U8);
```

Whereas the first three arguments of `vxCreateImage` are straightforward, the fourth one that encodes color space is less trivial. There is a range of color space options supported by OpenVX, both grayscale and color. Most functions in OpenVX support grayscale 8-bit images, but a few functions will have 16- and 32-bit images as input and/or output. Here is a list of grayscale image types:

- `VX_DF_IMAGE_U8`: a single plane of unsigned 8-bit pixels (pixel intensity varies from 0 to 255)
- `VX_DF_IMAGE_U16`: a single plane of unsigned 16-bit pixels (pixel intensity varies from 0 to 65535)
- `VX_DF_IMAGE_S16`: a single plane of signed 16-bit pixels (pixel intensity varies from −32768 to 32767)
- `VX_DF_IMAGE_U32`: a single plane of unsigned 32-bit pixels
- `VX_DF_IMAGE_S32`: a single plane of signed 32-bit pixels

There are a few OpenVX functions that can work with color images, but this is not the only reason color images are supported by the standard. Cameras will often produce images in various formats that OpenVX has to support to avoid expensive copying or transforming pixel data for each frame. Many of these images come in the `YUV` format, which uses a different number of bits to encode the intensity `Y` and the chroma channels `U` and `V`. Usually this subsampling is described by three numbers `A:B:C`, for example, `4:2:2`. Here the first number means the block of `Ax2` pixels (`A` columns and 2 rows). `B` describes the number of pixels used to encode chroma channels for the first row, and `C` for the second row. This is illustrated by Fig. 6.1. `4:4:4` encodes chroma for a 4×2 block of pixels using 8 values for U and V channels, `4:4:0` is using 4 values, so that pixels in each column have the

same color, and `4:2:0` is using only two values. OpenVX supports a range of color formats:

- `VX_DF_IMAGE_RGB`: standard RGB color space in 3 separate planes
- `VX_DF_IMAGE_RGBX`: RGB color space with alpha channel in 4 separate planes
- `VX_DF_IMAGE_NV12`: a YUV color space with 2 planes: a Y plane and an interleaved UV plane at 4:2:0 sampling
- `VX_DF_IMAGE_NV21`: a YUV color space with 2 planes: a Y plane and an interleaved VU plane at 4:2:0 sampling
- `VX_DF_IMAGE_UYVY`: a YUV color space with 4:2:2 sampling, organized into a single interleaved plane of 32-bit macropixels of U0, Y0, V0, Y1 bytes
- `VX_DF_IMAGE_YUYV`: a YUV color space with 4:2:2 sampling, organized into a single interleaved plane of 32-bit macropixels of Y0, U0, Y1, V1 bytes
- `VX_DF_IMAGE_IYUV`: a YUV color space with 4:2:0 sampling in 3 separate planes
- `VX_DF_IMAGE_YUV4`: a YUV color space with 4:4:4 sampling in 3 separate planes

6.2 Image filtering

6.2.1 Simple image filtering example

One of the most primitive image transformations in computer vision is Gaussian filtering. A 3×3 linear filter is independently applied to each color channel. We will create an OpenVX program that takes a color image and smoothes it with the Gaussian filter using the function `vxGaussian3x3Node`. We will use the graph API, and the resulting code will be similar to "changeImage.c," which was discussed in Chapter 2. `vxGaussian3x3Node` works only with greyscale images, so we will need to split a color input image into channels, run the filter on each of the channels, and then combine the channels into an output color image. To make the result more noticeable, we run filtering several times. The corresponding graph is shown in Fig. 6.2.

You can find the source code in "filter/filterGaussImage.c". The graph is constructed in the `makeFilterGraph` function. After creating the graph and allocating images, we add the nodes that extract individual channels from an input image:

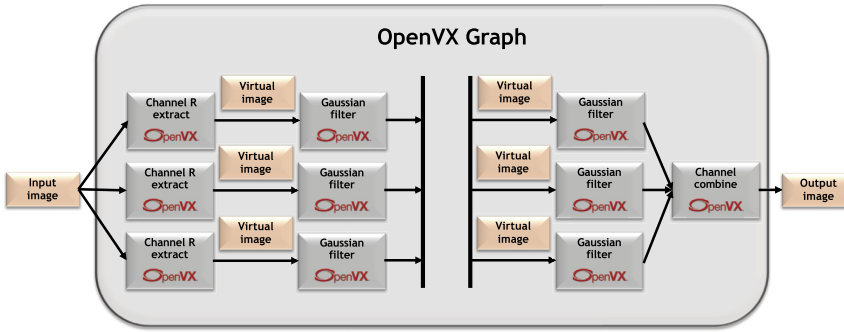


Figure 6.2 OpenVX graph for the Gaussian filter example.

```
vxChannelExtractNode(graph, input, VX_CHANNEL_R, virtu8[0]);
vxChannelExtractNode(graph, input, VX_CHANNEL_G, virtu8[1]);
vxChannelExtractNode(graph, input, VX_CHANNEL_B, virtu8[2]);
```

Then we add the Gaussian filter nodes for each of the channels:

```
for(i = 0; i < numv8 - 3; i++)
      vxGaussian3x3Node(graph, virtu8[i], virtu8[i + 3]);
```

With `numv8=18`, this adds five Gaussian filters for each channel. For example, the images corresponding to the red color channel are transformed into the following sequence: *virtu*8[0] → *virtu*8[3] → *virtu*8[6] → *virtu*8[9] → *virtu*8[15].

Finally, we combine the filtered images into an output color image:

```
vxChannelCombineNode(graph,
virtu8[numv8 - 3], virtu8[numv8 - 2],
    virtu8[numv8 - 1], NULL, output)
```

The rest of the code, including `main()` with image read/write, looks very much like "changeImage.c," which was already discussed. Note that all `virtu8[i]` images are virtual. This means that an OpenVX implementation that has a Gaussian filter that works on color images could just run it, skipping channel extract/combine operations. Also, if an implementation supports Gaussian filter stacking, then it can use it instead of executing five filters one by one, resulting in a code generating much less memory traffic.

To compile "filterGaussImage.c," you can use either CMake or the following command:

```
$ gcc ../book_samples/filter/filterGaussImage.c
    ../book_samples/ppm-io/readImage.c
    ../book_samples/ppm-io/writeImage.c -I ~/OpenVX_sample/include -I
    ../book_samples/ppm-io -l OpenVX -o filterGaussImage
```

You can run the sample on "cup.ppm":

```
$ ./filterGaussImage ../../book_samples_data/cup.ppm output.ppm
```

Fig. 6.3 shows the input and the output images.

Input image Output image

Figure 6.3 Gaussian image filtering: input and output.

6.2.2 Custom convolution

Now let us see how we can apply an arbitrary linear filter to an image. OpenVX has a special object for linear image filters called `vx_convolution`. We will need to create this object, set filter coefficients, and then apply the convolution to each of the image channels. You can find the source code in "filter/filterImage.c," which is very similar to "filterGaussImage," which we discussed in the previous section. We begin by defining the filter coefficients. In this example, we will use a Scharr 3 × 3 filter:

```
vx_int16 scharr_coeffs[3][3] = {
      {3, 0, -3},
      {10, 0, -10},
      {3, 0, -3}
};
```

Now we create a `vx_convolution` object and assign our convolution coefficients to it:

```
vx_convolution scharr = vxCreateConvolution(context, 3, 3);
vxCopyConvolutionCoefficients(scharr, (vx_int16*)scharr_coeffs,
    VX_WRITE_ONLY,
      VX_MEMORY_TYPE_HOST);
```

Apart from coefficients, there is one more attribute of our convolution object that we need to take care of. Convolutions used on a `VX_DF_IMAGE_U8` image can result in pixel values outside of the `0-255` range. Because OpenVX needs to be efficient on embedded platforms that may not have full support for floating point arithmetics, the output of a convolution is either a `VX_DF_IMAGE_U8` or a `VX_DF_IMAGE_S16` image. The policy used for convolution is `VX_CONVERT_POLICY_SATURATE`, which means that if the output pixel value is above the maximum, then it is clamped to the maximum value. To make it possible to use `VX_DF_IMAGE_U8` in a reasonable number of cases, `vx_convolution` has a parameter called "scale," which, together with the saturation policy, gives the output intensity $output(x, y)$ in the pixel with coordinates (x, y) for 8-bit unsigned images defined as follows:

$$output(x, y) = \begin{cases} 0 \text{ if } sum(x, y) < 0, \\ 255 \text{ if } sum(x, y)/scale > 255, \\ sum(x, y)/scale \text{ otherwise,} \end{cases} \tag{6.1}$$

where $sum(x, y)$ is the result of a convolution applied to an image patch of the same size as the convolution with a center in coordinates (x, y). To set the convolution scale, we will use the attribute set function:

```
vx_uint32 scale = 2;
vxSetConvolutionAttribute(scharr, VX_CONVOLUTION_SCALE, &scale,
    sizeof(scale));
```

Note that for the reasons of efficiency, the scale can be only a power of two (up to 2^{31}) for OpenVX 1.x. Now that we have the `vx_convolution` object set up, we construct a graph. First, we add nodes for splitting the color input image into three grayscale images:

```
vxChannelExtractNode(graph, input, VX_CHANNEL_R, virtu8[0]);
vxChannelExtractNode(graph, input, VX_CHANNEL_G, virtu8[1]);
vxChannelExtractNode(graph, input, VX_CHANNEL_B, virtu8[2]);
```

Then we add a convolution node for each channel:

```
for(i = 0; i < 3; i++)
      vxConvolveNode(graph, virtu8[i], scharr, virtu8[i + 3]);
```

Note that you can use different convolutions on different channels here. However, if you change the convolution coefficients in between graph executions, then the graph will need to be reverified each time. This allows OpenVX to check the filter coefficients and the sequence of filters in a graph and see if any optimizations can be applied for a specific hardware platform. Finally, we combine the grayscale images into a color one:

```
vxChannelCombineNode(graph, virtu8[3], virtu8[4], virtu8[5], NULL,
    output);
```

"filterImage.c" can be compiled similarly to the "filterGaussImage.c" in the previous section. Fig. 6.4 shows the input and the output images for the "filterImage.c" executed on "cup.ppm".

Input image Output image

Figure 6.4 Scharr image filtering: input and output.

6.3 Regions of interest

6.3.1 Reading from regions of interest

A standard operation in computer vision is preprocessing an image and selecting a rectangular area (region of interest, ROI) containing an object of interest for further processing. An ROI in OpenVX is an image that can be created using the `vxCreateImageFromROI` function. The ROI image is a part of a parent image, so, for instance, if a pixel value in an ROI image is updated, then this change will be reflected in a parent image. We will demonstrate the use of ROI with a slightly modified version of the Scharr filtering graph discussed in the previous section. You can find the full source code in "filter/filterImageROI.c". The first change is that we

need to define an output image in the "main" function with the size of the ROI rather than the size of the input image:

```
vx_rectangle_t rect;
rect.start_x = 48;
rect.start_y = 98;
rect.end_x = 258;
rect.end_y = 202;
int width = rect.end_x - rect.start_x;
int height = rect.end_y - rect.start_y;

vx_image output = vxCreateImage(context, width, height, VX_DF_IMAGE_RGB);
```

Then we pass the "rect" structure to the "makeFilterGraph" function, where we make the following change to the "vxChannelExtractNode" calls:

```
/* create ROI image */
vx_image roi = vxCreateImageFromROI(input, rect);

/* Do scharr filtering on the input image */
/* First, extract R, G, and B channels to individual virtual images */
vxChannelExtractNode(graph, roi, VX_CHANNEL_R, virtu8[0]);
vxChannelExtractNode(graph, roi, VX_CHANNEL_G, virtu8[1]);
vxChannelExtractNode(graph, roi, VX_CHANNEL_B, virtu8[2]);
```

Now the convolution node will take the channels of the ROI image as an input. The sample can be compiled similarly to "filterGaussImage.c" in Section 6.2.1. Fig. 6.5 shows the input and output images for the "filterImageROI.c" executed on "cup.ppm".

6.3.2 Writing to regions of interest

A less frequent use of ROI is modifying a part of an image. We will take an object on a cup and enhance its edges. We will use a Canny edge detector to find edges and binary operations to change the corresponding part of an input image. However, it will be challenging to do with the graph API. We will need to write both to an output image (to copy the pixels outside of the ROI) and to an ROI image within it. This means that two nodes will write to the same data object. Such a topology will cause a graph verification failure "VX_ERROR_MULTIPLE_WRITERS". So, to write to a region of interest, we will use the immediate mode API. You can find the source code in "filter/filterImageROIvxu.c". First, we will create an ROI image:

Input image Output image

Figure 6.5 Scharr ROI image filtering: input and output.

```
vx_rectangle_t rect;
rect.start_x = 48;
rect.start_y = 98;
rect.end_x = 258;
rect.end_y = 202;
vx_image roi = vxCreateImageFromROI(input, &rect);
```

Then we create temporary images for running the Canny edge detector and working with edge images:

```
int width = rect.end_x - rect.start_x;
int height = rect.end_y - rect.start_y;

/* create temporary images for working with edge images */
vx_image copy_channel[3], roi_channel[3], edges, edges_inv;
edges = vxCreateImage(context, width, height, VX_DF_IMAGE_U8);
edges_inv = vxCreateImage(context, width, height, VX_DF_IMAGE_U8);
```

Since we will run a Canny edge detector, we need to create a threshold object:

```
/* set the threshold value */
vx_pixel_value_t lower, higher;
lower.U32 = 50;
higher.U32 = 100;

/* create a threshold object */
vx_threshold threshold = vxCreateThresholdForImage(context,
      VX_THRESHOLD_TYPE_RANGE, VX_DF_IMAGE_U8, VX_DF_IMAGE_U8);
```

```
if(vxGetStatus(threshold) != VX_SUCCESS)
{
      printf("Threshold creation failed\n");
}

/* set threshold values */
vxCopyThresholdRange(threshold, &lower, &higher, VX_WRITE_ONLY,
      VX_MEMORY_TYPE_HOST);
```

Finally, we iterate through channels of our color image, compute a canny edge detector for each of them (stored in `edges`), invert it, and then use a bitwise "and" to make the corresponding pixels in the input image black:

```
enum vx_channel_e channels[] = {VX_CHANNEL_R, VX_CHANNEL_G,
     VX_CHANNEL_B};
for(int i = 0; i < 3; i++)
{
            roi_channel[i] = vxCreateImage(context, width, height,
                 VX_DF_IMAGE_U8);
            copy_channel[i] = vxCreateImage(context, width, height,
                 VX_DF_IMAGE_U8);
            vxuChannelExtract(context, roi, channels[i], roi_channel[i]);

            vxuCannyEdgeDetector(context, roi_channel[i], threshold, 3,
                 VX_NORM_L2, edges);

            vxuNot(context, edges, edges_inv);
            vxuAnd(context, roi_channel[i], edges_inv, copy_channel[i]);
}
```

Finally, we combine all the modified channels in the input image and save it to disk. Note that we combine channels in the `roi` image and then save the `input` image: our changes in the former have a direct impact on the latter:

```
vxuChannelCombine(context, copy\_channel[0], copy\_channel[1],
     copy\_channel[2], NULL, roi);
if(writeImage(input, \"cup\_roi.ppm\"))
            printf("Problem writing the output image\\n\");
```

The sample code can be compiled similarly to the "filterGaussImage.c" in Section 6.2.1. Fig. 6.6 shows the input and output images for the "filterImageROIvxu.c" code executed on "cup.ppm."

Figure 6.6 Canny edge effect on an ROI: input and output.

6.4 Feature extraction

6.4.1 Hough transform

Now let us see how we can use OpenVX to extract some meaningful information about the scene. One of the important functions in computer vision is extracting lines from an image. OpenVX has the function `vxHoughLinesPNode`, an implementation of the probabilistic Hough transform algorithm [35]. This function takes a binary image as an input and returns a set of lines. A single line in OpenVX is represented by a data structure `vx_line2d_t`:

```
typedef struct _vx_line2d_t {
  vx_float32
  vx_float32
  vx_float32
  vx_float32
} vx_line2d_t;
```

A collection of lines is represented by an array object `vx_array`. Being a graph API object, `vx_array` is opaque, just like `vx_image`. The `vxCopyArrayRange` function can be used to map the contents of the array into host memory for both reading and writing. Let us start with a simple example of the probabilistic hough transform, applied to an image of a road taken from a car. Our goal is to find the road lines. To do that, we will binarize an input image by computing a Sobel filter magnitude and applying a threshold and then running a hough transform on the resulting binary image. The scheme of the OpenVX graph that we will use is shown in Fig. 6.7. You can find the source code in the file "hough/houghLines.c."

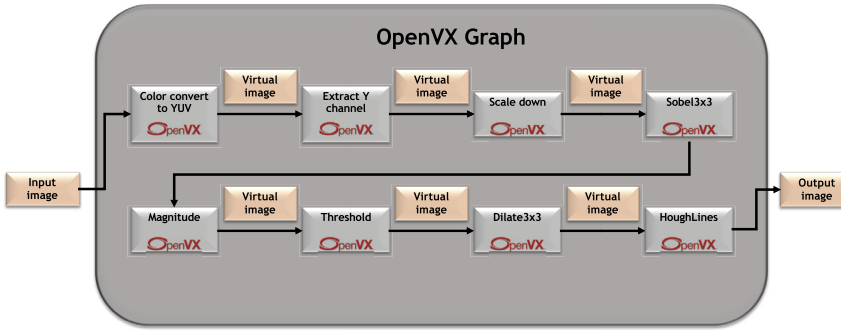


Figure 6.7 OpenVX graph for the Hough transform example.

We start by reading the input image with the function `vxa_read_image` from the `vxa` library, and finding its dimensions in the `main` function:

```
const char* input_filename = argv[1];
const char* binary_filename = argv[2];
const char* lines_filename = argv[3];

vx_context context = vxCreateContext();
vx_image image, binary;
vxa_read_image((const char *)input_filename, context, &image);

vx_uint32 width, height;
vxQueryImage(image, VX_IMAGE_WIDTH, &width, sizeof(vx_uint32));
vxQueryImage(image, VX_IMAGE_HEIGHT, &height, sizeof(vx_uint32));
```

Then we create the `lines` array, create a graph (we will review this function in detail further), register log callback function, and initiate graph processing:

```
/* create an array for storing hough lines output */
const vx_size max_num_lines = 2000;
lines = vxCreateArray(context, VX_TYPE_LINE_2D, max_num_lines);
vx_graph graph = makeHoughLinesGraph(context, image, &binary, &lines);

vxRegisterLogCallback(context, log_callback, vx_true_e);

vxProcessGraph(graph);
```

Note that the `binary` image is passed to the graph creation function as a pointer; it will be created inside the graph processing function as we do not know its dimensions here. Finally, we save the binary image, draw the detected lines on top of it, and also save the result:

```
vxa_write_image(binary, binary_filename);

// draw the lines
vx_pixel_value_t color;
color.RGB[0] = 0;
color.RGB[1] = 255;
color.RGB[2] = 0;
vx_image image_lines;
vx_size _num_lines;
// query the number of lines in the lines array
vxQueryArray(lines, VX_ARRAY_NUMITEMS, &_num_lines, sizeof(_num_lines));
draw_lines(context, binary, lines, _num_lines,
      &color, 2, &image_lines);
vxa_write_image(image_lines, lines_filename);
```

Now let us review the function that creates the graph. It starts by querying the dimensions of the image and defining the dimensions of the binary image that we will use for line detection, since the input image resolution is too high:

```
vx_graph makeHoughLinesGraph(vx_context context, vx_image input,
  vx_image* binary, vx_array lines)
{
   vx_uint32 width, height;
   vxQueryImage(input, VX_IMAGE_WIDTH, &width, sizeof(vx_uint32));
   vxQueryImage(input, VX_IMAGE_HEIGHT, &height, sizeof(vx_uint32));

   int widthr = width/4;
   int heightr = height/4;
```

Then we create a graph and allocate all images that we need:

```
vx_graph graph = vxCreateGraph(context);

#define nums16 (3)
vx_image virt_s16[nums16];

/* create virtual images */
```

```
vx_image virt_nv12 = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_NV12);
vx_image virt_y = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_U8);
vx_image virt_yr = vxCreateVirtualImage(graph, widthr, heightr,
      VX_DF_IMAGE_U8);
vx_image binary_thresh = vxCreateVirtualImage(graph, 0, 0,
     VX_DF_IMAGE_U8);

for(int i = 0; i < nums16; i++)
{
      virt_s16[i] = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_S16);
}

*binary = vxCreateImage(context, widthr, heightr, VX_DF_IMAGE_U8);
```

Now we proceed with adding the graph nodes. First, we convert the input RGB image into the YUV color format, extract the Y channel, and resize it down:

```
/* extract grayscale channel */
vxColorConvertNode(graph, input, virt_nv12);
vxChannelExtractNode(graph, virt_nv12, VX_CHANNEL_Y, virt_y);

/* resize down */
vxScaleImageNode(graph, virt_y, virt_yr, VX_INTERPOLATION_BILINEAR);
```

Then we compute image gradient magnitude with a Sobel filter:

```
vxSobel3x3Node(graph, virt_yr, virt_s16[0], virt_s16[1]);
vxMagnitudeNode(graph, virt_s16[0], virt_s16[1], virt_s16[2]);
```

Now we apply a threshold to the resulting greyscale image. Choosing a threshold value can be a complex task, but for simplicity, here we will use a constant value. As we discussed earlier, we need to create a threshold object, set its value, and only then add a thresholding graph node:

```
vx_threshold thresh = vxCreateThresholdForImage(context,
      VX_THRESHOLD_TYPE_BINARY, VX_DF_IMAGE_S16, VX_DF_IMAGE_U8);
vx_pixel_value_t pixel_value;
pixel_value.S16 = 256;
vxCopyThresholdValue(thresh, &pixel_value, VX_WRITE_ONLY,
      VX_MEMORY_TYPE_HOST);
vx_node thresh_node = vxThresholdNode(graph, virt_s16[2], thresh,
            binary_thresh);
```

There are some filters you can run on the resulting binary image to improve line detection. We will use a 3 × 3 dilate filter:

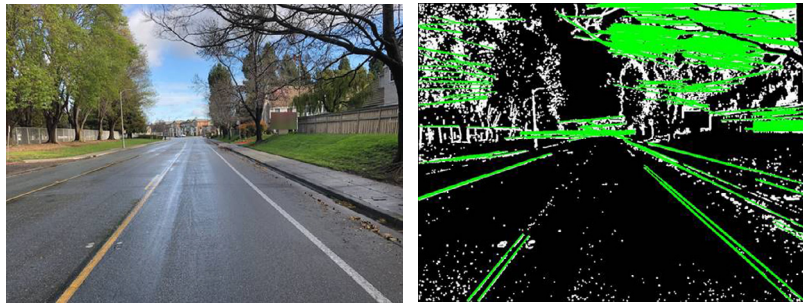
```
vxDilate3x3Node(graph, binary_thresh, *binary);
```

Finally, we are ready to run a Hough transform function on the binary image. It is important to choose right parameters for the Hough transform that are stored in the `vx_hough_lines_p_t` structure. The most important parameters are: `rho` is the size of a histogram bin for the distance from a line to the coordinate center, `theta` is the size of a histogram bin for the line orientation angle, and `threshold` is the minimum number of white pixels from the binary image that will lie on the detected line:

```
vx_hough_lines_p_t hough_params;
hough_params.rho = 1.0f;
hough_params.theta = 3.14f/180;
hough_params.threshold = 100;
hough_params.line_length = 100;
hough_params.line_gap = 10;
hough_params.theta_max = 3.14;
hough_params.theta_min = 0.0;

vx_scalar num_lines = vxCreateScalar(context, VX_TYPE_SIZE, NULL);
vxHoughLinesPNode(graph, *binary, &hough_params, lines, num_lines);
```

The results of executing the code on "IMG-7875.JPG" are given in Fig. 6.8. We can see that there are quite a few false alarms that should be filtered out. We will consider such a filtering in the next section.



Input image Binary image with lines

Figure 6.8 Lines detected with Hough transform.

6.4.2 Postprocessing hough transform results

Usually, it is not enough to just find the lines in an image; there is information about scene or 3D geometry that can be extracted from the lines. Specifically, for the lane detection problem we considered in the previous section, it is often useful to find the vanishing point, the crossing of the parallel lines representing road markings. We will implement the OpenVX graph shown in Fig. 6.9. The code that solves this problem is located in the "hough/houghLinesEx.c" example. First, judging from Fig. 6.8, we have too many false positives that we need to filter out. To this end, we will create a user node. We already covered the concept of user nodes in Section 4.6, so here we will just briefly review the code for the user node that implements line filtering. The filtering algorithm is based on two assumptions: the lines should be oriented almost vertically, and they should be located in the lower part of the image. First, let us review the callback function implementing the user kernel. This function (which will be called during graph execution) takes an array of lines as an input and also returns an array of lines.

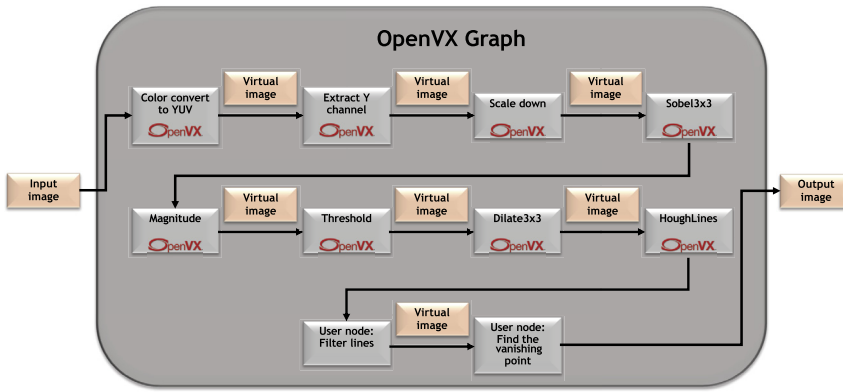


Figure 6.9 OpenVX graph for the Hough transform postprocessing example.

```
vx_status VX_CALLBACK filter_lines_calc_function( vx_node node, const
    vx_reference * refs, vx_uint32 num )
{
  vx_array lines = (vx_array) refs[0];
  vx_array lines_output = (vx_array) refs[1];
```

We start by extracting data from the input array of lines `lines`. First, we find the number of elements in the input array:

```
    vx_size num_lines = -1;
    ERROR_CHECK_STATUS(vxQueryArray(lines, VX_ARRAY_NUMITEMS,
        &num_lines, sizeof(num_lines)));
```

Note that we are using the macro `ERROR_CHECK_STATUS` and later `ERROR_CHECK_OBJECT`, which are handy for locating return value errors, as they will print both the error code, which you can look up in the OpenVX specification, and the line number of the function call generating this error. The macros are defined as follows:

```
#define ERROR_CHECK_STATUS( status ) { \
   vx_status status_ = (status); \
   if(status_ != VX_SUCCESS) { \
      printf("ERROR: failed with status = (%d) at " __FILE__ "#%d\n", \
          status_, __LINE__); \
      exit(1); \
   } \
}

#define ERROR_CHECK_OBJECT( obj ) { \
   vx_status status_ = vxGetStatus((vx_reference)(obj)); \
   if(status_ != VX_SUCCESS) { \
      printf("ERROR: failed with status = (%d) at " __FILE__ "#%d\n", \
          status_, __LINE__); \
      exit(1); \
   } \
}
```

We continue the filtering function by mapping the input array data into host memory:

```
    char* __lines = NULL;
    vx_map_id map_id;
    vx_size stride = sizeof(vx_line2d_t);
    vxMapArrayRange(lines, 0, num_lines, &map_id, &stride,
        (void**)&__lines,
          VX_READ_ONLY, VX_MEMORY_TYPE_HOST, 0);
```

Then we go through each line, and if it satisfies the filtering conditions, we copy it into specially allocated host memory `_lines_filtered`:

```
    vx_line2d_t _lines_filtered[max_num_lines];
    vx_size _num_lines_filtered = 0;
```

```
        const float max_ratio = 0.1;
        for(int i = 0; i < num_lines; i++, __lines += stride)
        {
              vx_line2d_t* _line = (vx_line2d_t*)__lines;
              int dx = _line->end_x - _line->start_x;
              int dy = _line->end_y - _line->start_y;

              if(_line->start_y < heightr/2 || _line->end_y < heightr/2)
              {
                    continue;
              }

              memcpy(&_lines_filtered[_num_lines_filtered++], _line,
                    sizeof(vx_line2d_t));
        }
```

Last, we unmap the input array and add all the copied data into the output array:

```
        vxUnmapArrayRange(lines, map_id);

        vxAddArrayItems(lines_output, _num_lines_filtered, _lines_filtered,
              sizeof(vx_line2d_t));

        return(VX_SUCCESS);
}
```

To create a user node, we need a validator function, which will be called during the graph verification to check the correctness of input and output parameter types. In our case the validator enforces the input and output arrays to be of type `VX_TYPE_LINE_2D`:

```
vx_status VX_CALLBACK filter_lines_validator( vx_node node, const
     vx_reference parameters[], vx_uint32 num, vx_meta_format metas[] )
{
   // parameter #0 -- check array type
   vx_enum param_type;
   ERROR_CHECK_STATUS( vxQueryArray( ( vx_array )parameters[0],
        VX_ARRAY_ITEMTYPE, &param_type, sizeof( param_type ) ) );
   if(param_type != VX_TYPE_LINE_2D) // check that the array contains
        lines
   {
      return VX_ERROR_INVALID_TYPE;
```

```
    }

    // parameter #1 -- check array type
    ERROR_CHECK_STATUS( vxQueryArray( ( vx_array )parameters[1],
         VX_ARRAY_ITEMTYPE, &param_type, sizeof( param_type ) ) );
    if(param_type != VX_TYPE_LINE_2D)
    {
       return VX_ERROR_INVALID_TYPE;
    }

    // set output metadata
    ERROR_CHECK_STATUS( vxSetMetaFormatAttribute( metas[1],
         VX_ARRAY_ITEMTYPE, &param_type, sizeof( param_type ) ) );

    return VX_SUCCESS;
}
```

We also need a function that registers the user kernel with OpenVX:

```
vx_status registerUserFilterLinesKernel( vx_context context )
{
    vx_kernel kernel = vxAddUserKernel( context,
                            "app.userkernels.filter_lines",
                            USER_KERNEL_FILTER_LINES,
                            filter_lines_calc_function,
                            2,  // numParams
                            filter_lines_validator,
                            NULL,
                            NULL );
    ERROR_CHECK_OBJECT( kernel );

    ERROR_CHECK_STATUS( vxAddParameterToKernel( kernel, 0, VX_INPUT,
         VX_TYPE_ARRAY, VX_PARAMETER_STATE_REQUIRED ) ); // input
    ERROR_CHECK_STATUS( vxAddParameterToKernel( kernel, 1, VX_OUTPUT,
         VX_TYPE_ARRAY, VX_PARAMETER_STATE_REQUIRED ) ); // output
    ERROR_CHECK_STATUS( vxFinalizeKernel( kernel ) );
    ERROR_CHECK_STATUS( vxReleaseKernel( &kernel ) );

    vxAddLogEntry( ( vx_reference ) context, VX_SUCCESS, "OK: registered
         user kernel app.userkernels.filter_lines\n" );
    return VX_SUCCESS;
}
```

Finally, we need a function that adds a user node to the graph:

```
vx_node userFilterLinesNode(vx_graph graph,
                    vx_array input,
                    vx_array output)
{
   vx_context context = vxGetContext( ( vx_reference ) graph );
   vx_kernel kernel = vxGetKernelByEnum( context,
        USER_KERNEL_FILTER_LINES);
   ERROR_CHECK_OBJECT( kernel );
   vx_node node  = vxCreateGenericNode( graph, kernel );
   ERROR_CHECK_OBJECT( node );

   ERROR_CHECK_STATUS( vxSetParameterByIndex( node, 0, ( vx_reference )
        input ) );
   ERROR_CHECK_STATUS( vxSetParameterByIndex( node, 1, ( vx_reference )
        output ) );

   ERROR_CHECK_STATUS( vxReleaseKernel( &kernel ) );

   return node;
}
```

At this point, we have everything we need to run line filtering as a user node in an OpenVX graph. Now let us add a function that finds the crossing point of all filtered lines. We will implement it as a user kernel, which will be executed right after the line filtering kernel. We will use the uniform coordinates (e.g., see [26]) for representing lines as they are very convenient for finding line crossings. Each line is represented by a three-dimensional vector l so that the equation describing the line can be represented as $(x, y, 1)^T l = 0$, where x, y are pixel coordinates. We start with the function that computes the coordinates of the cross point for two lines, each given by a three-dimensional vector. According to projective geometry [26], the cross point in the uniform coordinates will be given by the vector product of these vectors:

```
void find_cross_point(const float* line1, const float* line2,
 float* cross_point)
{
 cross_point[0] = line1[1]*line2[2] - line1[2]*line2[1];
 cross_point[1] = line1[2]*line2[0] - line1[0]*line2[2];
 cross_point[2] = line1[0]*line2[1] - line1[1]*line2[0];
}
```

Now let us see the implementation of the kernel that computes the cross point of the road lines. As with the line filtering kernel, it starts with mapping the input array to host memory:

```
vx_status VX_CALLBACK vanishing_point_calc_function( vx_node node, const
    vx_reference * refs, vx_uint32 num )
{
  vx_array lines = (vx_array)refs[0];
  vx_array vanishing_points = (vx_array)refs[1];

  vx_size num_lines = -1;
  ERROR_CHECK_STATUS(vxQueryArray(lines, VX_ARRAY_NUMITEMS, &num_lines,
      sizeof(num_lines)));

  char* __lines = 0;
  vx_size stride = sizeof(vx_line2d_t);
  vx_map_id map_id;

  vxMapArrayRange(lines, 0, num_lines, &map_id, &stride, (void**)&__lines,
    VX_READ_ONLY, VX_MEMORY_TYPE_HOST, 0);
```

Then we convert the lines into uniform coordinates:

```
      float lines_uniform[max_num_lines][3];
      for(int i = 0; i < num_lines; i++, __lines += stride)
      {
            vx_line2d_t* _line = (vx_line2d_t*)__lines;
            float x0 = _line->start_x;
            float y0 = _line->start_y;
            float dx = _line->end_x - _line->start_x;
            float dy = _line->end_y - _line->start_y;

            lines_uniform[i][0] = dy;
            lines_uniform[i][1] = -dx;
            lines_uniform[i][2] = -x0*dy + y0*dx;
      }

      vxUnmapArrayRange(lines, map_id);
```

Now we calculate the crossing point of each pair of lines and find the average. Obviously, this algorithm is sensitive to strong outliers, and there are many ways to make it more robust, but for brevity, we will stick with the simplest version. The final result is stored in the output `vx_array` object:

```
    vx_coordinates2d_t avg_cross_point = {0.0, 0.0};
    int count = 0;
    for(int i = 0; i < num_lines; i++)
    {
        for(int j = 0; j < num_lines; j++)
        {
            float cross_point[3];
            find_cross_point(lines_uniform[i], lines_uniform[j],
                cross_point);
            if(fabs(cross_point[2]) < FLT_MIN)
            {
                // filter the vanishing point
                continue;
            }
            float cx = cross_point[0]/cross_point[2];
            float cy = cross_point[1]/cross_point[2];

            if(cx < 0 || cy < 0 || cx > widthr || cy > heightr)
            {
                // we know the cross point lies inside an image,
                    so this is an outlier
                continue;
            }

            avg_cross_point.x += (int)cx;
            avg_cross_point.y += (int)cy;
            count++;
        }
    }

    avg_cross_point.x /= count;
    avg_cross_point.y /= count;

    vxAddArrayItems(vanishing_points, 1, &avg_cross_point,
        sizeof(avg_cross_point));

    return(VX_SUCCESS);
}
```

The node validation and registration functions and the function for adding this node to a graph are similar to the line filtering node. The graph creation function is almost the same as in the previous section, except for the last

part where the Hough transform is computed. The line filtering node and the node for finding the vanishing point are added:

```
    vx_array _lines = vxCreateVirtualArray(graph, VX_TYPE_LINE_2D,
        max_num_lines);

    vx_scalar num_lines = vxCreateScalar(context, VX_TYPE_SIZE, NULL);

    /* run hough transform */
    vx_hough_lines_p_t hough_params;
    hough_params.rho = 1.0f;
    hough_params.theta = 3.14f/180;
    hough_params.threshold = 100;
    hough_params.line_length = 100;
    hough_params.line_gap = 10;
    hough_params.theta_max = 3.14;
    hough_params.theta_min = 0.0;

    vxHoughLinesPNode(graph, *binary, &hough_params, _lines,
        num_lines);

    userFilterLinesNode(graph, _lines, lines);
    userFindVanishingPoint(graph, lines, vanishing_points);

    return graph;
}
```

Note that the array `_lines`, which we use as an input to the `userFilterLinesNode`, is virtual. Nevertheless, the calls to `vxMapArrayRange` and `vxUnmapArrayRange` executed from the user node will be executed successfully, as they are called during graph execution. This is guaranteed by the OpenVX spec; see the subsection "Virtual Data Objects" in the chapter "Graph Concepts": "No calls to an Map/Unmap or Copy APIs will succeed given a reference to an object created through a virtual create function from a graph external perspective. Calls to Map/Unmap or Copy APIs from within client-defined node that belongs to the same graph as the virtual object will succeed as they are graph internal."

The `main` function is also very similar, and we add the drawing of a circle at the vanishing point:

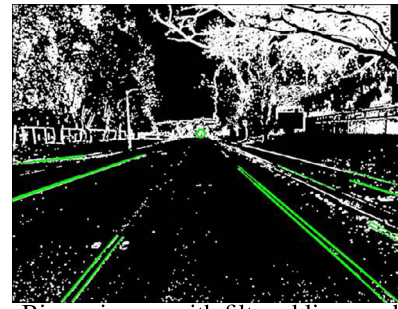
```
// read the coordinates of the vanishing point
vx_coordinates2d_t coordinates;
```

```
vxCopyArrayRange(vanishing_points, 0, 1, sizeof(coordinates),
     &coordinates, VX_READ_ONLY, VX_MEMORY_TYPE_HOST);

// draw the circle around each vanishing point coordinate
vx_image image_final;
draw_circles(context, image_lines, vanishing_points, 1, 10, &color, 3,
     &image_final);
vxa_write_image(image_final, lines_filename);
```

The results of running "houghLinesEx.c" on "IMG-7875.JPG" are shown in Fig. 6.10.

Input image



Binary image with filtered lines and the vanishing point.

Figure 6.10 Lines detected with Hough transform.

6.5 Geometric image transformations

6.5.1 Undistortion implemented with remap

One of the essential geometric transformations of an image in computer vision is correcting for lens distortion, usually called "undistort." A typical effect of lens distortion is that a straight line in a three dimensional space is not straight in an image generated with a camera. Undistort transformation maps an image into another image as if the new image is generated with a perspective transformation, and straight lines in 3D are mapped to straight lines in the image. Undistort needs information about camera intrinsic parameters and lens distortion coefficients. OpenVX, being a library focused on runtime, has no way to compute this data (a process typically referred to as "camera calibration"). Also, OpenVX contains no any specific model for lens distortion. Since the undistort transformation, given camera and

lens parameters, maps pixels only depending on their positions in images, undistort can be implemented with a remap. So, we first will use OpenCV offline to create a remap transformation.

The standard way to find lens parameters is calibrating a camera by making images of a known pattern like a checkerboard. OpenCV "calibration" sample is used to obtain camera intrinsic parameters and lens distortion coefficients. We refer to the OpenCV calibration tutorial [36] for details. Then we need to calculate the remap corresponding to the undistort mapping.

Remap is a very simple image transformation: for each pixel with coordinates (x_d, y_d) in the destination image, it specifies floating point coordinates in the source image $\big(x_s(x_d, y_d), y_s(x_d, y_d)\big)$, so that the destination image intensity I_d is defined using the source image intensity I_s as

$$I_d(x_d, y_d) = I_s\left(x_s(x_d, y_d), y_s(x_d, y_d)\right). \tag{6.2}$$

Since x_s, y_s are floating point and do not necessarily fall into a pixel center, I_s is interpolated from the intensities in the neighboring pixels of the source image. OpenCV and OpenVX specify several such interpolation methods.

The method for generating a remap transformation given camera parameters is implemented in the "undistort/undistortOpenCV.cpp" sample. It reads a file of camera parameters created by an OpenCV calibration procedure and then writes a file of undistortion remapping data suitable for use by OpenVX. First, it reads the camera parameters and image dimensions saved by the OpenCV calibration sample:

```
FileStorage fs(camera_file, FileStorage::READ);
Mat intrinsic_params, dist_coeffs;
fs["camera_matrix"] >> intrinsic_params;
fs["distortion_coefficients"] >> dist_coeffs;
int width, height;
fs["image_width"] >> width;
fs["image_height"] >> height;
```

Then we generate undistort remap transformation with

```
Mat map1, map2, new_camera;
initUndistortRectifyMap(intrinsic_params, dist_coeffs, Mat(),
      intrinsic_params, Size(width, height), CV_32FC2, map1, map2);
```

The result is stored in `map1`, whereas `map2` is not used. Now we save the remap together with source and destination image dimensions required by OpenVX:

```
FileStorage fs1(map_fname, FileStorage::WRITE);
fs1 << "remap" << map1;
fs1 << "remap_src_width" << width;
fs1 << "remap_src_height" << height;
fs1 << "remap_dst_width" << width;
fs1 << "remap_dst_height" << height;
```

As a result of executing this sample, we get an xml or yml file with the remap transformation. Now let us import it into OpenVX.

A remap in OpenVX is encapsulated by a special object `vx_remap`. It is very similar to `vx_image`. Instead of width and height attributes, it has both source and destination image sizes:

```
enum vx_remap_attribute_e {
      VX_REMAP_SOURCE_WIDTH = VX_ATTRIBUTE_BASE(VX_ID_KHRONOS,
          VX_TYPE_REMAP) + 0x0,
  VX_REMAP_SOURCE_HEIGHT = VX_ATTRIBUTE_BASE(VX_ID_KHRONOS,
      VX_TYPE_REMAP) + 0x1,
  VX_REMAP_DESTINATION_WIDTH = VX_ATTRIBUTE_BASE(VX_ID_KHRONOS,
      VX_TYPE_REMAP) +
      0x2,
  VX_REMAP_DESTINATION_HEIGHT = VX_ATTRIBUTE_BASE(VX_ID_KHRONOS,
      VX_TYPE_REMAP) +
           0x3};
```

Similar to `vx_image`, data can be copied to/from a `vx_remap` object using the functions `vxMapRemapPatch`, `vxUnmapRemapPatch`, and `vxCopyRemapPatch`. As of OpenVX version 1.3, the only graph node that uses a remap is created with `vxRemapNode`. This node has one image as an input and one image as an output.

The image undistort transformation based on OpenVX is implemented in "undistort/undistort-remap.c." This sample uses the library "vxa" [31] (https://github.com/relrotciv/vxa) to read an input image from a jpeg file:

```
vx_image input_image;
if(vxa_read_image(image_filename, context, &input_image) != 1)
{
```

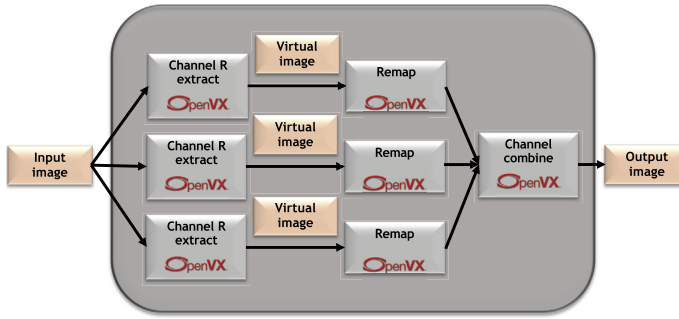


Figure 6.11 A remap transformation graph.

```
        printf("Error reading image 1\n");
        return(-1);
}
```

We use the same library to read a remap from a file created by OpenCV. The function that reads remap also returns us the width and height of the output image:

```
int width, height;
vx_remap remap;
if(vxa_import_opencv_remap(remap_filename, "remap", context, &remap,
        &width, &height) != 1)
{
        printf("Error reading remap1\n");
        return(-1);
}
```

Then we create an output color image using the dimensions provided to us by the remap:

```
vx_image output_image = vxCreateImage(context, width, height,
     VX_DF_IMAGE_RGB);
```

Now that all the data are in OpenVX, let us create a graph (see the `makeRemapGraph` function). Our graph is shown in Fig. 6.11. Since a remap node only accepts greyscale images, as usual, we will use virtual images to process remaps individually on each channel. A smart OpenVX implementation can figure this out and process all three channels together, saving

on memory data transfer. The `makeRemapGraph` function starts by creating a graph object and the helper virtual images:

```
const int numu8 = 2;
vx_image virtu8[numu8][3];
int i, j;

vx_graph graph = vxCreateGraph(context);

for(i = 0; i < numu8; i++)
      for (j = 0; j < 3; j++)
            virtu8[i][j] = vxCreateVirtualImage(graph, 0, 0,
                VX_DF_IMAGE_U8);
```

Then we extract each channel from an input image and setup a remap node with a bilinear interpolation:

```
enum vx_channel_e channels[] = {VX_CHANNEL_R, VX_CHANNEL_G,
     VX_CHANNEL_B};
for(i = 0; i < 3; i++)
{
      vxChannelExtractNode(graph, input_image, channels[i],
           virtu8[0][i]);
      vxRemapNode(graph, virtu8[0][i], remap, VX_INTERPOLATION_BILINEAR,
            virtu8[1][i]);
}
```

Now we combine all the remapped images into a single-color image and release the virtual images:

```
vxChannelCombineNode(graph, virtu8[1][0], virtu8[1][1], virtu8[1][2],
      NULL, output_image);

for (i = 0; i < numu8; i++)
      for(j = 0; j < 3; j++)
            vxReleaseImage(&virtu8[i][j]);
```

The graph is set up, and we only need to run the graph processing and save the results to a file using the "vxa" library:

```
if((status = vxVerifyGraph(graph)))
{
      printf("Graph verification failed, error code %d, %d\n",
            (int)status, (int)VX_ERROR_NOT_SUFFICIENT);
```

```
}
else if (vxProcessGraph(graph))
            printf("Error processing graph\n");
else if (vxa_write_image(output_image, output_filename) != 1)
            printf("Problem writing the output image\n");
vxReleaseContext(&context);
```

Examples input and output images for two different cameras are shown in Figs. 6.12 and 6.13. Note that the GoPro undistorted image shows a small part of the original image. This is a common effect of the undistort function on a wide angle camera with a relatively strong distortion.

To generate remap xml files, use the "undistortOpenCV.cpp" on the "*camera.xml" files in the data folder. Assuming that the full path to the data folder is in `\$BOOK_SAMPLES_DATA`, run

```
$ ./undistortOpenCV $BOOK_SAMPLES_DATA/canon-camera.xml
     $BOOK_SAMPLES_DATA/canon-test.jpg output.jpg
     $BOOK_SAMPLES_DATA/canon-remap.xml
$ ./undistortOpenCV $BOOK_SAMPLES_DATA/gopro-camera.xml
     $BOOK_SAMPLES_DATA/gopro-test.png output.jpg
     $BOOK_SAMPLES_DATA/gopro-remap.xml
```

Run the "undistort.c" sample to reproduce the results in Figs. 6.12 and 6.13:

```
$ ./undistort $BOOK_SAMPLES_DATA/canon-remap.xml
     $BOOK_SAMPLES_DATA/canon-test.jpg output.jpg
$ ./undistort $BOOK_SAMPLES_DATA/gopro-remap.xml
     $BOOK_SAMPLES_DATA/gopro-test.png output.jpg
```

Input image Output image

Figure 6.12 ***Undistort image transformation: input and output.*** Undistort image transformation: input and output, Canon EOS 100D, Sigma 18 mm.

Input image Output image

Figure 6.13 ***Undistort image transformation: input and output.*** Undistort image transformation: input and output, GoPro HERO 3+, video 1080 p.

6.5.2 Perspective transformations

OpenVX supports two most commonly used image transformations in computer vision, affine and perspective. An affine transformation is given by a 2×3 matrix, which defines a pixel coordinate mapping from the output image to the input. Specifically:

$$\begin{aligned} x_0 &= M_{1,1}x + M_{2,1}y + M_{3,1}, \\ y_0 &= M_{1,2}x + M_{2,2}y + M2, 3. \end{aligned} \tag{6.3}$$

Here (x_0, y_0) and x, y are the coordinates of a pixel in the input and output images, respectively, and M is an affine matrix. An example of affine transformation has been given in Chapter 2, where it was used to rotate an image 90 degrees. So in this chapter, we focus on the perspective transformation. The API for both functions is very similar, and everything we learn here can be applied to the affine transformation too.

The homography or perspective transformation is defined by a 3×3 matrix that defines a mapping of pixel coordinates:

$$\begin{aligned} x_u &= M_{1,1}x + M_{2,1}y + M_{3,1}, \\ y_u &= M_{1,2}x + M_{2,2}y + M_{2,3}, \\ z_u &= M_{1,3}x + M_{2,3}y + M_{3,3}. \end{aligned} \tag{6.4}$$

Here x, y are pixel coordinates in the output image, and x_u, y_u, z_u are uniform pixel coordinates in the input image. The normal input pixel coordinates are given by

$$\begin{aligned} x_0 &= x_u/z_u, \\ y_0 &= y_u/z_u. \end{aligned} \tag{6.5}$$

The OpenVX function that creates a perspective transformation graph node is specified as follows:

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

The algorithm implemented in this node computes the intensity in each output image pixel by mapping it to an input image using Eqs. (6.4)–(6.5). Since there usually is no one-to-one mapping between input and output pixels, the output pixel intensity is computed by interpolating the neighboring pixels intensity. The specific interpolation method is given by the "type" parameter. If the output pixel is mapped outside of the input image boundaries, then the border mode is used to compute the input pixel intensity. The perspective node supports `BORDER_MODE_UNDEFINED` and `BORDER_MODE_CONSTANT`. Note that the output image dimensions do not necessarily have to be equal to the input image dimensions. This puts a not too obvious restriction on the output image: its dimensions cannot be inferred from the input image dimensions, so the output image cannot be a virtual image without specified width and height. The same is true for the affine transformation. The dimensions of the output image for both `vxWarpAffineNode` and `vxWarpPerspectiveNode` must always be specified.

To illustrate the OpenVX perspective transformation, we will use the previously developed example of using the Hough transform to detect road lanes. Section 6.4.2 describes finding the vanishing point as a crossing of parallel lanes. We will extend this sample to generate a bird's eye view from a single image. The bird's eye view sample is implemented in "birds-eye/birdsEyeView.c," which is created by modifying "filter/houghLinesEx.c." The result of the algorithm is shown in Fig. 6.14. To reproduce these results, run

```
./birdsEyeView $BOOK_SAMPLES_DATA/IMG-7875.JPG output.jpg
```

Since a road is flat, a change in camera position can be simulated with a perspective transformation (see [26]). So, we need to come up with a perspective transformation that sends the vanishing point to infinity, and this will make the road lines parallel to each other. Since a perspective transformation depends on the vanishing point, it will have to be generated during graph execution time, so we will need a user node for that. We will discuss how to do this a little later; for now, let us see how we can apply the perspective transformation to an image.

Figure 6.14 Results of the bird's eye view perspective transformation.

6.5.2.1 *Applying a perspective transformation*

The perspective transformation node is added to an OpenVX graph in the graph creation function "makeBirdsEyeViewGraph," which is almost the same as "the makeHoughLinesGraph" from "houghLinesEx.c." The scheme of the graph we will discuss in this section is shown in Fig. 6.15.

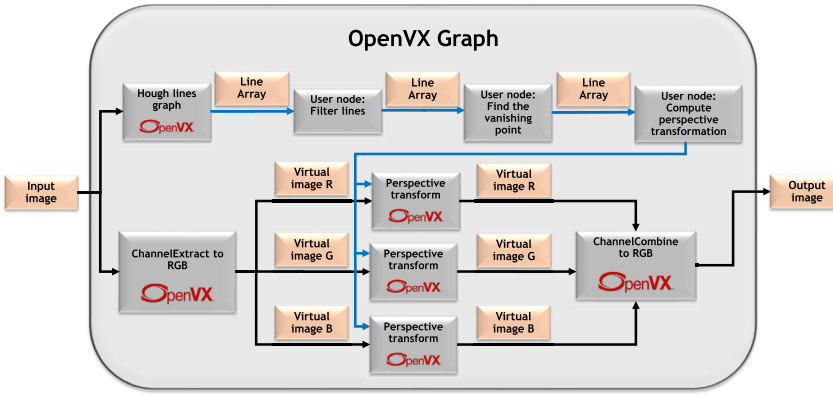


Figure 6.15 OpenVX graph for generating bird's eye view.

After adding the node that calculates a position of the vanishing point "userFindVanishingPoint," we add the user node that returns a perspective transformation:

```
userFindVanishingPoint(graph, lines, vanishing_points);
userComputeBirdsEyeTransform(graph, vanishing_points, input,
    perspective);
```

Then we apply the perspective transformation to the input image. Since "vxWarpPerspectiveNode" works with grayscale images only, we split the input image into three channels, process each of them, and then combine them back into the output image:

```
/* Create the same processing subgraph for each channel */
enum vx_channel_e channels[] = {VX_CHANNEL_R, VX_CHANNEL_G,
     VX_CHANNEL_B};

for(int i = 0; i < 3; i++)
{
      /* First, extract input and logo R, G, and B channels to individual
      virtual images */
      vxChannelExtractNode(graph, input, channels[i], virt_u8[i]);

      vx_node warp_node = vxWarpPerspectiveNode(graph, virt_u8[i],
           perspective, VX_INTERPOLATION_BILINEAR, virt_u8[i + 3]);
      ERROR_CHECK_OBJECT(warp_node);

      // set the border mode to constant with zero value
      vx_border_t border_mode;
      border_mode.mode = VX_BORDER_CONSTANT;
      border_mode.constant_value.U8 = 0;
      vxSetNodeAttribute(warp_node, VX_NODE_BORDER, &border_mode,
           sizeof(border_mode));

}
vxChannelCombineNode(graph, virt_u8[3], virt_u8[4], virt_u8[5], NULL,
     birds_eye);
```

Note that the matrix generated by the `userComputeBirdsEyeTransform` node is an input to the `vxWarpPerspectiveNode`. Since no object metadata change here, graph reverification will not be triggered for each graph execution.

There will be a substantial amount of pixels in the output image that will be mapped outside of the input image boundaries. We want them to be black, and so we set the border mode to `VX_BORDER_CONSTANT` with the pixel value equal to 0. Also, note that we use virtual images, so that an OpenVX implementation can execute this operation in a more optimal way, for example, running the perspective transformation on a color image in one pass. Since the `vxWarpPerspectiveNode` cannot figure out the size of the output image from the input image, the virtual images have to be ini-

tialized with specific values for width and height; see the beginning of the `makeBirdsEyeViewGraph` implementation:

```
vx_uint32 width, height;
vxQueryImage(input, VX_IMAGE_WIDTH, &width, sizeof(vx_uint32));
vxQueryImage(input, VX_IMAGE_HEIGHT, &height, sizeof(vx_uint32));

/* create virtual images */
const int numu8 = 6;
vx_image virt_u8[numu8];

for(int i = 0; i < numu8; i++)
{
      virt_u8[i] = vxCreateVirtualImage(graph, width, height,
          VX_DF_IMAGE_U8);
}
```

Now let us see how we can create a perspective transformation during graph execution time.

6.5.2.2 Generating a perspective transformation

We want to create a perspective transformation that sends the vanishing point to infinity. This can be done by considering the mapping

$$H = KRK^{-1}, \tag{6.6}$$

where

$$K = \begin{pmatrix} f_x & 0 & c_x \\ f_y & 0 & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{6.7}$$

is the intrinsic camera matrix, and R is a rotation around x axis,

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{pmatrix}. \tag{6.8}$$

The rotation angle ϕ is chosen so that the vanishing point maps to infinity. Also, we need to keep the part of the road in front of the camera in the view; otherwise, our output will be a black image. So, we will add an additional pan and zoom transformation given by the matrix Z:

$$H_{\text{final}} = ZH. \tag{6.9}$$

Note that throughout this section, we will use the direct perspective transformation that maps an input image to an output image. OpenVX uses an inverse matrix that maps an output image to an input, and we will address this only in the end when we will generate the output `vx_matrix` object.

This algorithm is implemented in the `birdseye_transform_calc_function`. It has two input parameters: a `vx_array` with one element corresponding to the vanishing point and the input image, which is only needed to pass the required size of the output image. The output parameter is the perspective transformation in a `vx_matrix` object. First, we get the input/output parameters and image width/height:

```
vx_status VX_CALLBACK birdseye_transform_calc_function( vx_node node,
    const vx_reference * refs, vx_uint32 num )
{
  vx_array points = (vx_array)refs[0];
  vx_image image = (vx_image)refs[1];
  vx_matrix perspective = (vx_matrix)refs[2];

  // get image height
  vx_uint32 image_width, image_height;
  ERROR_CHECK_STATUS(vxQueryImage(image, VX_IMAGE_WIDTH, &image_width,
      sizeof(image_width)));
  ERROR_CHECK_STATUS(vxQueryImage(image, VX_IMAGE_HEIGHT, &image_height,
      sizeof(image_height)));
```

Then we initialize the intrinsics matrix and calculate its inverse (needed in (6.6)):

```
      // intrinsic parameters
      float _K[9] = {
        8.4026236186715255e+02*scale_factor, 0.,
            3.7724917600845038e+02*scale_factor,
        0., 8.3752885759166338e+02*scale_factor,
            4.6712164335800873e+02*scale_factor,
        0., 0., 1.
      };

      // calculate intrinsics inverse
      float _Kinv[9];
```

```
calc_inverse_3x3matrix(_K, _Kinv);
```

`scale_factor = 4` is used here and further because several operations, including camera calibration and vanishing point detection, were done on an image resized down 4 times each dimension. `calc_inverse_3x3matrix` is implemented using the LAPACK library. Then we obtain the coordinates of the vanishing point from the input `vx_array` argument:

```
// obtain the vanishing point
const int num_points = 1;
vx_coordinates2d_t* _points = 0;
vx_size stride = sizeof(vx_coordinates2d_t);
vx_map_id map_id;
vxMapArrayRange(points, 0, num_points, &map_id, &stride,
    (void**)&_points,
  VX_READ_ONLY, VX_MEMORY_TYPE_HOST, 0);
```

Now we find the corresponding uniform coordinates of the vanishing point using the inverse intrinsic matrix:

```
// generate the vanishing point in uniform coordinates
float pv[] = {_points[0].x*scale_factor,
     _points[0].y*scale_factor};
float pvu[2];
calc_homography(_Kinv, pv, pvu);
float yv = pvu[1];
```

We are ready to find the angle ϕ from (6.8). Note that we do all matrix operations with floating point arrays, and we will use `vx_matrix` only for the output:

```
// generate a homography that sends the vanishing point to infinity
float phi = atan(1/yv);
float _rotate[9] = {
  1.0f, 0.0f, 0.0f,
  0.0f, -cos(phi), -sin(phi),
  0.0f, sin(phi), -cos(phi)
};
```

Once we know the rotation matrix, we are ready to generate a perspective transformation that sends the vanishing point to infinity:

```
// generate birds eye view homography
float _temp[9], _perspective[9];
```

```
mult_3x3matrices(_K, _rotate, _temp);
mult_3x3matrices(_temp, _Kinv, _perspective);
```

We also have to make sure that the important part of the image is visible after this transformation. We will use an affine transformation that maps parallel lines to parallel lines, but we cannot make it a separate node since if the image is empty after the perspective transformation node, then the output image will be empty too. For simplicity, we will construct this mapping as a pan and zoom transformation, making sure two control points in the input image map inside the output image. First, we generate the coordinates of the control points in the input image:

```
// now map two control points using the perspective matrix,
// to adjust scale and translation
float upper_boundary_factor = 1.2f;
float control1[2] = {pv[0], pv[1]*upper_boundary_factor};
float control2[2] = {pv[0], image_height};
```

Then we map them to the output image:

```
float control1_mapped[2], control2_mapped[2];
calc_homography(_perspective, control1, control1_mapped);
calc_homography(_perspective, control2, control2_mapped);

// find y coordinates of the mapped points from the uniform
    coordinates
float y1 = control1_mapped[1];
float y2 = control2_mapped[1];
```

Now we generate a pan and zoom transformation that maps these points to the upper and lower boundaries of the output image and multiply it to the left from the perspective transformation:

```
// now define additional translation and scale to have the control
    points
// mapped to the upper and lower boundary of the output image
float scale = ((float)y2 - y1)/image_height;
float _panzoom[] = {1.0f, 0.0f, image_width*scale/2 - pv[0],
                                                  0.0f, 1.0f, -y1,
                                                  0.0f, 0.0f, scale};

// now create the final perspective transformation by multiplying
// _perspective by _panzoom from the left
```

```
    float _perspective_final[9];
    mult_3x3matrices(_panzoom, _perspective, _perspective_final);
```

We have obtained the required perspective transformation. Note that OpenVX deals with the inverse transposed homography transformation (see (6.4)), so we invert and transpose the matrix before importing it:

```
    // now we need to invert and transpose the homography for OpenVX
    float _perspective_final_inv[9];
    calc_inverse_3x3matrix(_perspective_final, _perspective_final_inv);
    transpose(_perspective_final_inv);

    vxCopyMatrix(perspective, _perspective_final_inv, VX_WRITE_ONLY,
        VX_MEMORY_TYPE_HOST);

    return VX_SUCCESS;
```

The validation of this user node is implemented in the `birdseye_transform_validator` function. We check that the output matrix is 3×3 floating point and set the corresponding metadata:

```
// parameter #2 -- check that this is a floating point 3x3 matrix
ERROR_CHECK_STATUS( vxQueryMatrix( ( vx_matrix )parameters[2],
    VX_MATRIX_TYPE, &param_type, sizeof( param_type ) ) );
if(param_type != VX_TYPE_FLOAT32)
{
            return VX_ERROR_INVALID_TYPE;
}

vx_size rows, columns;
ERROR_CHECK_STATUS( vxQueryMatrix( ( vx_matrix )parameters[2],
    VX_MATRIX_ROWS, &rows, sizeof( rows ) ) );
if(rows != 3)
{
            return VX_ERROR_INVALID_DIMENSION;
}

ERROR_CHECK_STATUS( vxQueryMatrix( ( vx_matrix )parameters[2],
    VX_MATRIX_COLUMNS, &columns, sizeof( columns ) ) );
if(columns != 3)
{
            return VX_ERROR_INVALID_DIMENSION;
}
```

```
// set output metadata
ERROR_CHECK_STATUS( vxSetMetaFormatAttribute( metas[2], VX_MATRIX_TYPE,
     &param_type, sizeof( param_type ) ) );
ERROR_CHECK_STATUS( vxSetMetaFormatAttribute( metas[2], VX_MATRIX_ROWS,
     &rows, sizeof( rows ) ) );
ERROR_CHECK_STATUS( vxSetMetaFormatAttribute( metas[2],
     VX_MATRIX_COLUMNS, &columns, sizeof( columns ) ) );
```