1
Fun with Filters

The goal of this chapter is to develop a number of image processing filters and apply them to the video stream of a webcam in real time. These filters will rely on various OpenCV functions to manipulate matrices through splitting, merging, arithmetic operations, and applying lookup tables for complex functions.

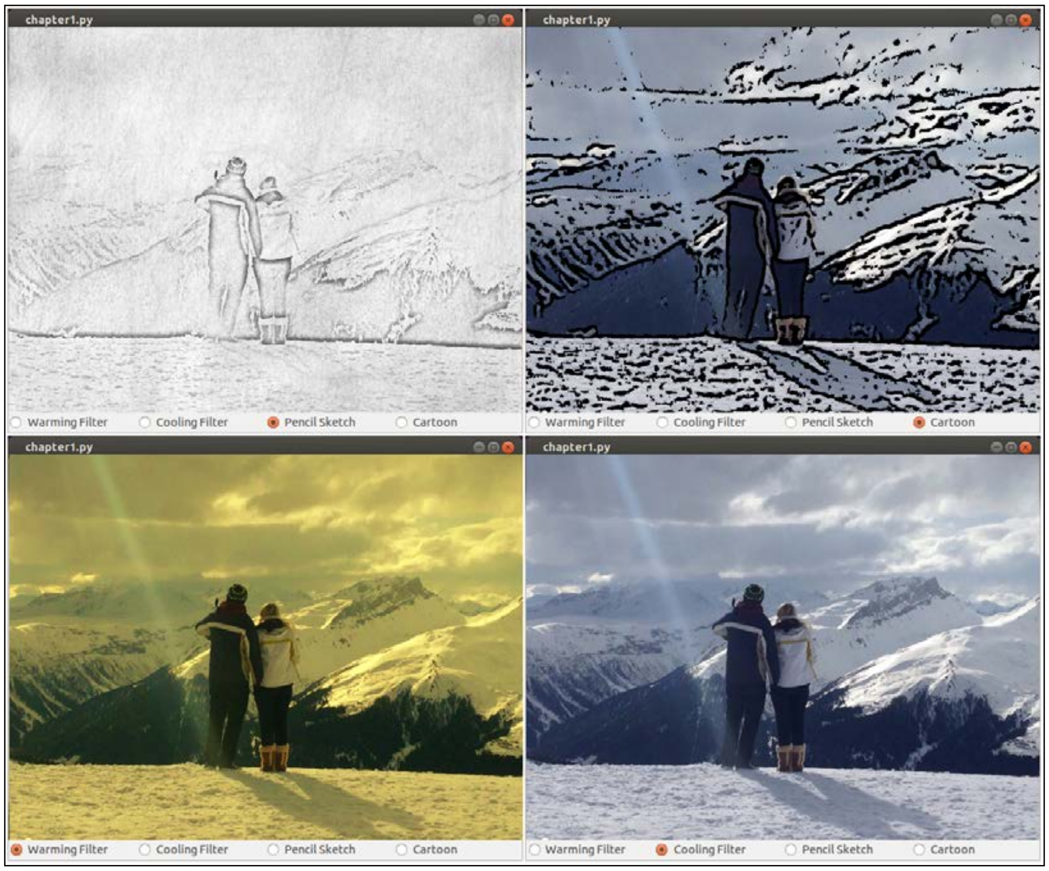
The three effects are as follows:

- **Black-and-white pencil sketch**: To create this effect, we will make use of two image blending techniques, known as **dodging** and **burning**
- **Warming/cooling filters**: To create these effects, we will implement our own **curve filters** using a lookup table
- **Cartoonizer**: To create this effect, we will combine a **bilateral filter**, a **median filter**, and **adaptive thresholding**

OpenCV is such an advanced toolchain that often the question is not how to implement something from scratch, but rather which pre-canned implementation to choose for your needs. Generating complex effects is not hard if you have a lot of computing resources to spare. The challenge usually lies in finding an approach that not only gets the job done, but also gets it done in time.

Instead of teaching the basic concepts of image manipulation through theoretical lessons, we will take a practical approach and develop a single end-to-end app that integrates a number of image filtering techniques. We will apply our theoretical knowledge to arrive at a solution that not only works but also speeds up seemingly complex effects so that a laptop can produce them in real time.

The following screenshot shows the final outcome of the three effects running on a laptop:



All of the code in this book is targeted for OpenCV 2.4.9 and has been tested on Ubuntu 14.04. Throughout this book, we will make extensive use of the NumPy package (`http://www.numpy.org`). In addition, this chapter requires the `UnivariateSpline` module of the SciPy package (`http://www.scipy.org`) as well as the wxPython 2.8 graphical user interface (`http://www.wxpython.org/download.php`) for cross-platform GUI applications. We will try to avoid further dependencies wherever possible.

Planning the app

The final app will consist of the following modules and scripts:

- `filters`: A module comprising different classes for the three different image effects. The modular approach will allow us to use the filters independently of any **graphical user interface** (**GUI**).
- `filters.PencilSketch`: A class for applying the pencil sketch effect to an RGB color image.
- `filters.WarmingFilter`: A class for applying the warming filter to an RGB color image.
- `filters.CoolingFilter`: A class for applying the cooling filter to an RGB color image.
- `filters.Cartoonizer`: A method for applying the cartoonizer effect to an RGB color image.
- `gui`: A module that provides a wxPython GUI application to access the webcam and display the camera feed, which we will make extensive use of throughout the book.
- `gui.BaseLayout`: A generic layout from which more complicated layouts can be built.
- `chapter1`: The main script for this chapter.
- `chapter1.FilterLayout`: A custom layout based on `gui.BaseLayout` that displays the camera feed and a row of radio buttons that allows the user to select from the available image filters to be applied to each frame of the camera feed.
- `chapter1.main`: The main function routine for starting the GUI application and accessing the webcam.

Creating a black-and-white pencil sketch

In order to obtain a pencil sketch (that is, a black-and-white drawing) of the camera frame, we will make use of two image blending techniques, known as **dodging** and **burning**. These terms refer to techniques employed during the printing process in traditional photography; photographers would manipulate the exposure time of a certain area of a darkroom print in order to lighten or darken it. Dodging lightens an image, whereas burning darkens it.

Areas that were not supposed to undergo changes were protected with a **mask**. Today, modern image editing programs, such as Photoshop and Gimp, offer ways to mimic these effects in digital images. For example, masks are still used to mimic the effect of changing exposure time of an image, wherein areas of a mask with relatively intense values will *expose* the image more, thus lightening the image. OpenCV does not offer a native function to implement these techniques, but with a little insight and a few tricks, we will arrive at our own efficient implementation that can be used to produce a beautiful pencil sketch effect.

If you search on the Internet, you might stumble upon the following common procedure to achieve a pencil sketch from an RGB color image:

1. Convert the color image to grayscale.
2. Invert the grayscale image to get a negative.
3. Apply a Gaussian blur to the negative from step 2.
4. Blend the grayscale image from step 1 with the blurred negative from step 3 using a color dodge.

Whereas steps 1 to 3 are straightforward, step 4 can be a little tricky. Let's get that one out of the way first.

OpenCV 3 comes with a pencil sketch effect right out of the box. The `cv2.pencilSketch` function uses a domain filter introduced in the 2011 paper *Domain transform for edge-aware image and video processing*, by Eduardo Gastal and Manuel Oliveira. However, for the purpose of this book, we will develop our own filter.

Implementing dodging and burning in OpenCV

In modern image editing tools, such as Photoshop, color dodging of an image `A` with a mask `B` is implemented as the following ternary statement acting on every pixel index, called `idx`:

```
((B[idx] == 255) ? B[idx] :
    min(255, ((A[idx] << 8) / (255-B[idx]))))
```

This essentially divides the value of an `A[idx]` image pixel by the inverse of the `B[idx]` mask pixel value, while making sure that the resulting pixel value will be in the range of [0, 255] and that we do not divide by zero.

We could translate this into the following naïve Python function, which accepts two OpenCV matrices (`image` and `mask`) and returns the blended image:

```
def dodgeNaive(image, mask):
    # determine the shape of the input image
    width,height = image.shape[:2]

    # prepare output argument with same size as image
    blend = np.zeros((width,height), np.uint8)

    for col in xrange(width):
        for row in xrange(height):

            # shift image pixel value by 8 bits
            # divide by the inverse of the mask
            tmp = (image[c,r] << 8) / (255.-mask)

            # make sure resulting value stays within bounds
            if tmp > 255:
                tmp = 255
            blend[c,r] = tmp
    return blend
```

As you might have guessed, although this code might be functionally correct, it will undoubtedly be horrendously slow. Firstly, the function uses `for` loops, which are almost always a bad idea in Python. Secondly, NumPy arrays (the underlying format of OpenCV images in Python) are optimized for array calculations, so accessing and modifying each `image[c,r]` pixel separately will be really slow.

Instead, we should realize that the `<<8` operation is the same as multiplying the pixel value with the number *2^8=256*, and that pixel-wise division can be achieved with the `cv2.divide` function. Thus, an improved version of our dodge function could look like this:

```
import cv2

def dodgeV2(image, mask):
    return cv2.divide(image, 255-mask, scale=256)
```

We have reduced the dodge function to a single line! The `dodgeV2` function produces the same result as `dodgeNaive` but is orders of magnitude faster. In addition, `cv2.divide` automatically takes care of division by zero, making the result `0` where `255-mask` is zero.

Now, it is straightforward to implement an analogous burning function, which divides the inverted image by the inverted mask and inverts the result:

```
import cv2

def burnV2(image, mask):
    return 255 – cv2.divide(255-image, 255-mask, scale=256)
```

Pencil sketch transformation

With these tricks in our bag, we are now ready to take a look at the entire procedure. The final code will be in its own class in the `filters` module. After we have converted a color image to grayscale, we aim to blend this image with its blurred negative:

1. We import the OpenCV and `numpy` modules:

   ```
   import cv2
   import numpy as np
   ```

2. Instantiate the `PencilSketch` class:

   ```
   class PencilSketch:
       def __init__(self, (width, height),
           bg_gray='pencilsketch_bg.jpg'):
   ```

   The constructor of this class will accept the image dimensions as well as an optional background image, which we will make use of in just a bit. If the file exists, we will open it and scale it to the right size:

   ```
   self.width = width
   self.height = height

   # try to open background canvas (if it exists)
   self.canvas = cv2.imread(bg_gray, cv2.CV_8UC1)
   if self.canvas is not None:
       self.canvas = cv2.resize(self.canvas,
           (self.width, self.height))
   ```

3. Add a render method that will perform the pencil sketch:

   ```
   def renderV2(self, img_rgb):
   ```

4. Converting an RGB image (`imgRGB`) to grayscale is straightforward:

   ```
   img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
   ```

   Note that it does not matter whether the input image is RGB or BGR.

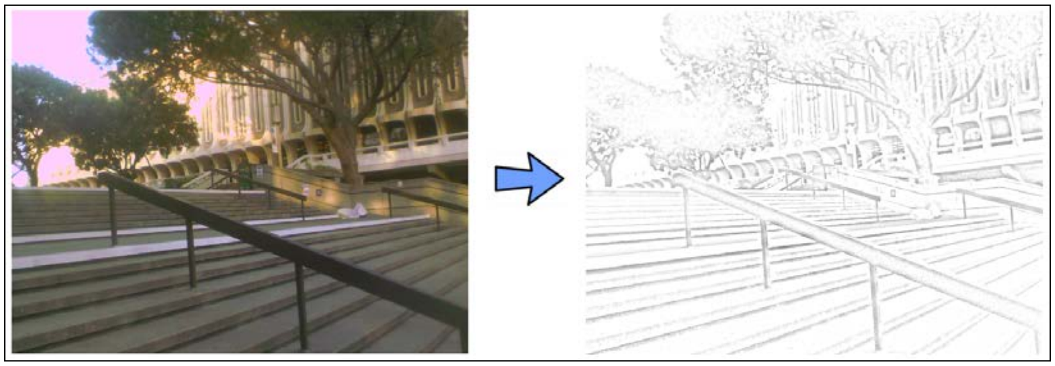
5. We then invert the image and blur it with a large Gaussian kernel of size `(21,21)`:

   ```
   img_gray_inv = 255 - img_gray
   img_blur = cv2.GaussianBlur(img_gray_inv, (21,21), 0, 0)
   ```

6. We use our `dodgeV2` dodging function from the aforementioned code to blend the original grayscale image with the blurred inverse:

   ```
   img_blend = dodgeV2(mg_gray, img_blur)
   return cv2.cvtColor(img_blend, cv2.COLOR_GRAY2RGB)
   ```

The resulting image looks like this:



Did you notice that our code can be optimized further?

A Gaussian blur is basically a convolution with a Gaussian function. One of the beauties of convolutions is their associative property. This means that it does not matter whether we first invert the image and then blur it, or first blur the image and then invert it.

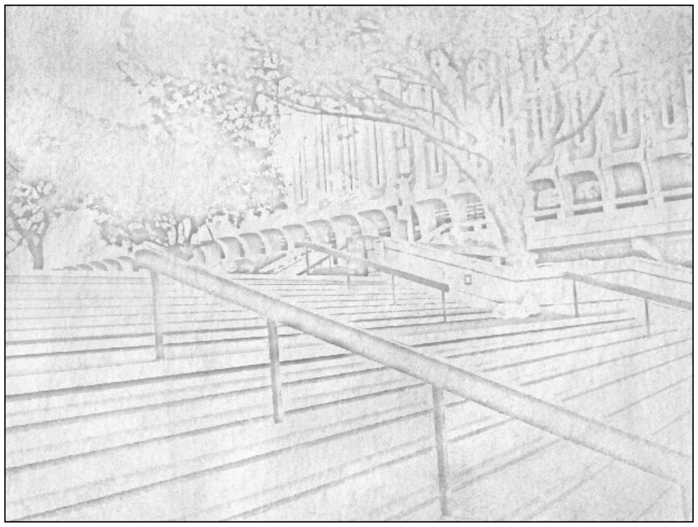
"Then what matters?" you might ask. Well, if we start with a blurred image and pass its inverse to the `dodgeV2` function, then within that function, the image will get inverted again (the `255-mask` part), essentially yielding the original image. If we get rid of these redundant operations, an optimized `render` method would look like this:

```
def render(img_rgb):
    img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2GRAY)
    img_blur = cv2.GaussianBlur(img_gray, (21,21), 0, 0)
    img_blend = cv2.divide(img_gray, img_blur, scale=256)
    return img_blend
```

For kicks and giggles, we want to lightly blend our transformed image (`img_blend`) with a background image (`self.canvas`) that makes it look as if we drew the image on a canvas:

```
if self.canvas is not None:
    img_blend = cv2.multiply(img_blend, self.canvas, scale=1./256)
return cv2.cvtColor(img_blend, cv2.COLOR_GRAY2BGR)
```

And we're done! The final output looks like what is shown here:



Generating a warming/cooling filter

When we perceive images, our brain picks up on a number of subtle clues to infer important details about the scene. For example, in broad daylight, highlights may have a slightly yellowish tint because they are in direct sunlight, whereas shadows may appear slightly bluish due to the ambient light of the blue sky. When we view an image with such color properties, we might immediately think of a *sunny day*.

This effect is no mystery to photographers, who sometimes purposely manipulate the white balance of an image to convey a certain mood. Warm colors are generally perceived as more pleasant, whereas cool colors are associated with night and drabness.

To manipulate the perceived **color temperature** of an image, we will implement a **curve filter**. These filters control how color transitions appear between different regions of an image, allowing us to subtly shift the color spectrum without adding an unnatural-looking overall tint to the image.

Color manipulation via curve shifting

A curve filter is essentially a function, *y = f(x)*, that maps an input pixel value *x* to an output pixel value *y*. The curve is parameterized by a set of *n+1* anchor points, as follows: *{(x_0,y_0), (x_1,y_1), ..., (x_n, y_n)}*.

Each anchor point is a pair of numbers that represent the input and output pixel values. For example, the pair *(30, 90)* means that an input pixel value of 30 is increased to an output value of 90. Values between anchor points are interpolated along a smooth curve (hence the name curve filter).

Such a filter can be applied to any image channel, be it a single grayscale channel or the R, G, and B channels of an RGB color image. Thus, for our purposes, all values of *x* and *y* must stay between 0 and 255.

For example, if we wanted to make a grayscale image slightly brighter, we could use a curve filter with the following set of control points: *{(0,0), (128, 192), (255,255)}*. This would mean that all input pixel values except 0 and 255 would be increased slightly, resulting in an overall brightening effect of the image.

If we want such filters to produce natural-looking images, it is important to respect the following two rules:

- Every set of anchor points should include *(0,0)* and *(255,255)*. This is important in order to prevent the image from appearing as if it has an overall tint, as black remains black and white remains white.
- The function *f(x)* should be monotonously increasing. In other words, with increasing *x*, *f(x)* either stays the same or increases (that is, it never decreases). This is important for making sure that shadows remain shadows and highlights remain highlights.

Implementing a curve filter by using lookup tables

Curve filters are computationally expensive, because the values of *f(x)* must be interpolated whenever *x* does not coincide with one of the prespecified anchor points. Performing this computation for every pixel of every image frame that we encounter would have dramatic effects on performance.

Instead, we make use of a lookup table. Since there are only 256 possible pixel values for our purposes, we need to calculate *f(x)* only for all the 256 possible values of *x*. Interpolation is handled by the `UnivariateSpline` function of the `scipy.interpolate` module, as shown in the following code snippet:

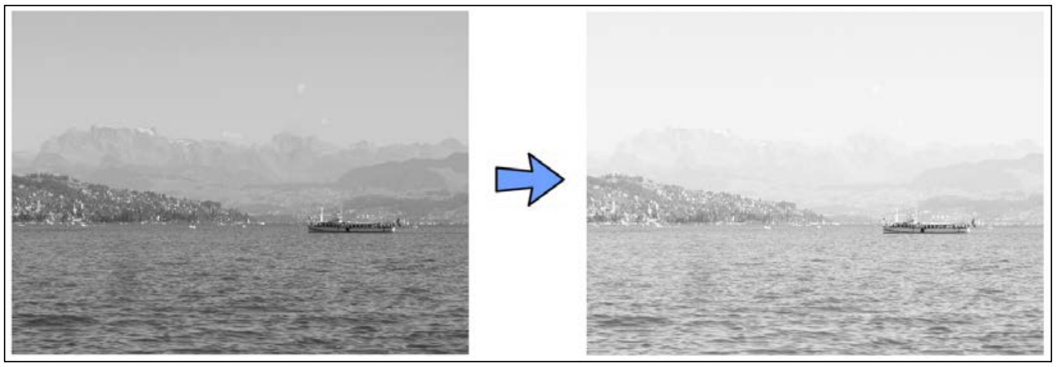
```
from scipy.interpolate import UnivariateSpline

def _create_LUT_8UC1(self, x, y):
  spl = UnivariateSpline(x, y)
  return spl(xrange(256))
```

The `return` argument of the function is a 256-element list that contains the interpolated *f(x)* values for every possible value of *x*.

All we need to do now is come up with a set of anchor points, *(x_i, y_i)*, and we are ready to apply the filter to a grayscale input image (`img_gray`):

```
import cv2
import numpy as np

x = [0, 128, 255]
y = [0, 192, 255]
myLUT = _create_LUT_8UC1(x, y)
img_curved = cv2.LUT(img_gray, myLUT).astype(np.uint8)
```

The result looks like this (the original image is on the left, and the transformed image is on the right):



Designing the warming/cooling effect

With the mechanism to quickly apply a generic curve filter to any image channel in place, we now turn to the question of how to manipulate the perceived color temperature of an image. Again, the final code will have its own class in the `filters` module.

If you have a minute to spare, I advise you to play around with the different curve settings for a while. You can choose any number of anchor points and apply the curve filter to any image channel you can think of (red, green, blue, hue, saturation, brightness, lightness, and so on). You could even combine multiple channels, or decrease one and shift another to a desired region. What will the result look like?

However, if the number of possibilities dazzles you, take a more conservative approach. First, by making use of our `_create_LUT_8UC1` function developed in the preceding steps, let's define two generic curve filters, one that (by trend) increases all pixel values of a channel, and one that generally decreases them:

```
class WarmingFilter:

  def __init__(self):
    self.incr_ch_lut = _create_LUT_8UC1([0, 64, 128, 192, 256],
      [0, 70, 140, 210, 256])
    self.decr_ch_lut = _create_LUT_8UC1([0, 64, 128, 192, 256],
      [0, 30,  80, 120, 192])
```

The easiest way to make an image appear as if it was taken on a hot, sunny day (maybe close to sunset), is to increase the reds in the image and make the colors appear vivid by increasing the color saturation. We will achieve this in two steps:

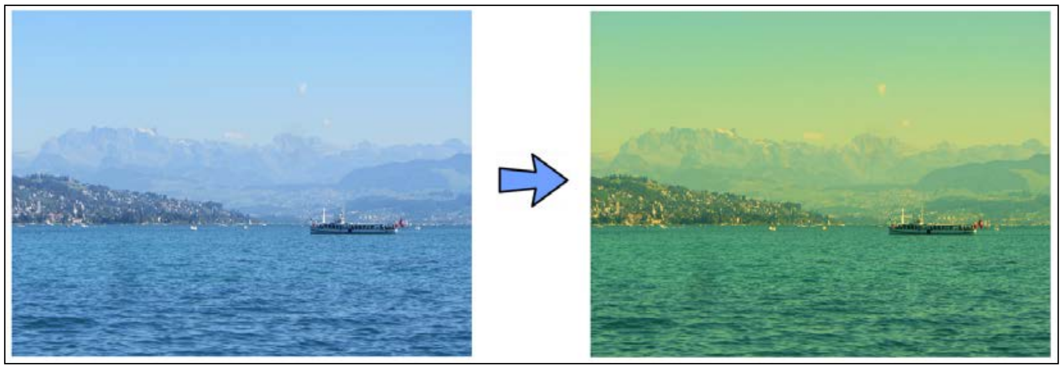
1. Increase the pixel values in the R channel and decrease the pixel values in the B channel of an RGB color image using `incr_ch_lut` and `decr_ch_lut`, respectively:

   ```
   def render(self, img_rgb):
       c_r, c_g, c_b = cv2.split(img_rgb)
       c_r = cv2.LUT(c_r, self.incr_ch_lut).astype(np.uint8)
       c_b = cv2.LUT(c_b, self.decr_ch_lut).astype(np.uint8)
       img_rgb = cv2.merge((c_r, c_g, c_b))
   ```

2. Transform the image into the **HSV** color space (**H** means hue, **S** means saturation, and **V** means value), and increase the S channel using `incr_ch_lut`. This can be achieved with the following function, which expects an RGB color image as input:

   ```
   c_b = cv2.LUT(c_b, decrChLUT).astype(np.uint8)

   # increase color saturation
   c_h, c_s, c_v = cv2.split(cv2.cvtColor(img_rgb,
       cv2.COLOR_RGB2HSV))
   c_s = cv2.LUT(c_s, self.incr_ch_lut).astype(np.uint8)
   return cv2.cvtColor(cv2.merge((c_h, c_s, c_v)),
       cv2.COLOR_HSV2RGB)
   ```

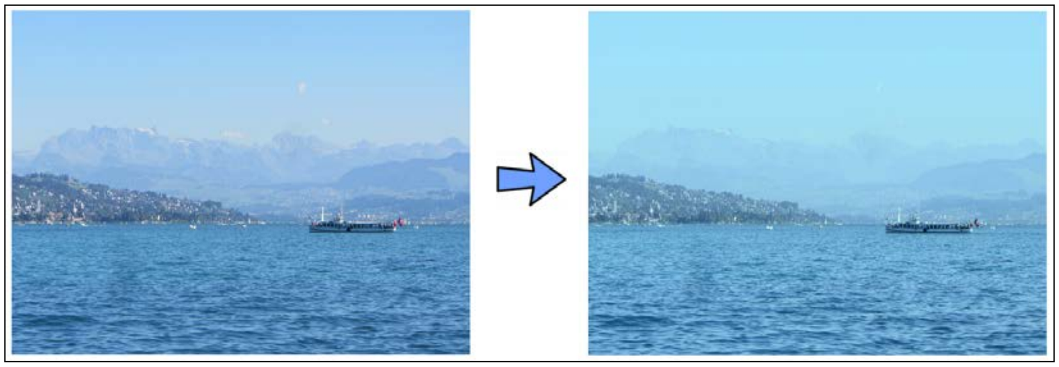
The result looks like what is shown here:



Analogously, we can define a cooling filter that increases the pixel values in the B channel, decreases the pixel values in the R channel of an RGB image, converts the image into the HSV color space, and decreases color saturation via the S channel:

```
class CoolingFilter:
    def render(self, img_rgb):
        c_r, c_g, c_b = cv2.split(img_rgb)
        c_r = cv2.LUT(c_r, self.decr_ch_lut).astype(np.uint8)
        c_b = cv2.LUT(c_b, self.incr_ch_lut).astype(np.uint8)
        img_rgb = cv2.merge((c_r, c_g, c_b))
        # decrease color saturation
        c_h, c_s, c_v = cv2.split(cv2.cvtColor(img_rgb,
            cv2.COLOR_RGB2HSV))
        c_s = cv2.LUT(c_s, self.decr_ch_lut).astype(np.uint8)
        return cv2.cvtColor(cv2.merge((c_h, c_s, c_v)),
            cv2.COLOR_HSV2RGB)
```

Now, the result looks like this:



Cartoonizing an image

Over the past few years, professional cartoonizer software has popped up all over the place. In order to achieve the basic cartoon effect, all that we need is a **bilateral filter** and some **edge detection**. The bilateral filter will reduce the color palette, or the numbers of colors that are used in the image. This mimics a cartoon drawing, wherein a cartoonist typically has few colors to work with. Then we can apply edge detection to the resulting image to generate bold silhouettes. The real challenge, however, lies in the computational cost of bilateral filters. We will thus use some tricks to produce an acceptable cartoon effect in real time.

We will adhere to the following procedure to transform an RGB color image into a cartoon:

1. Apply a bilateral filter to reduce the color palette of the image.
2. Convert the original color image into grayscale.
3. Apply a **median blur** to reduce image noise.
4. Use **adaptive thresholding** to detect and emphasize the edges in an edge mask.
5. Combine the color image from step 1 with the edge mask from step 4.

Using a bilateral filter for edge-aware smoothing

A strong bilateral filter is ideally suitable for converting an RGB image into a color painting or a cartoon, because it smoothens flat regions while keeping edges sharp. It seems that the only drawback of this filter is its computational cost, as it is orders of magnitude slower than other smoothing operations, such as a Gaussian blur.

The first measure to take when we need to reduce the computational cost is to perform an operation on an image of low resolution. In order to downscale an RGB image (`imgRGB`) to a quarter of its size (reduce the width and height to half), we could use `cv2.resize`:

```
import cv2

img_small = cv2.resize(img_rgb, (0,0), fx=0.5, fy=0.5)
```

A pixel value in the resized image will correspond to the pixel average of a small neighborhood in the original image. However, this process may produce image artifacts, which is also known as aliasing. While this is bad enough on its own, the effect might be enhanced by subsequent processing, for example, edge detection.

A better alternative might be to use the **Gaussian pyramid** for downscaling (again to a quarter of the original size). The Gaussian pyramid consists of a blur operation that is performed before the image is resampled, which reduces aliasing effects:

```
img_small = cv2.pyrDown(img_rgb)
```

However, even at this scale, the bilateral filter might still be too slow to run in real time. Another trick is to repeatedly (say, five times) apply a small bilateral filter to the image instead of applying a large bilateral filter once:

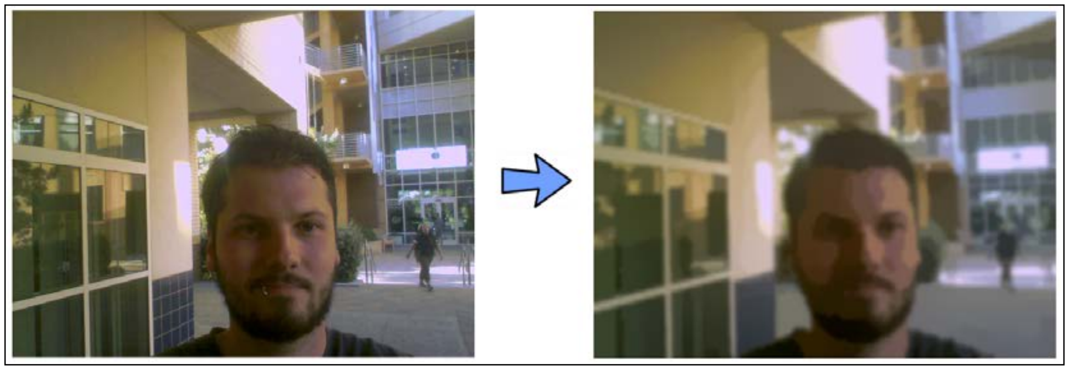
```
num_iter = 5
for _ in xrange(num_iter):
    img_small = cv2.bilateralFilter(img_small, d=9, sigmaColor=9,
        sigmaSpace=7)
```

The three parameters in `cv2.bilateralFilter` control the diameter of the pixel neighborhood (`d`) and the standard deviation of the filter in the color space (`sigmaColor`) and coordinate space (`sigmaSpace`).

Don't forget to restore the image to its original size:

```
img_rgb = cv2.pyrUp(img_small)
```

The result looks like a blurred color painting of a creepy programmer, as follows:



Detecting and emphasizing prominent edges

Again, when it comes to edge detection, the challenge often does not lie in how the underlying algorithm works, but instead which particular algorithm to choose for the task at hand. You might already be familiar with a variety of edge detectors. For example, **Canny edge detection** (`cv2.Canny`) provides a relatively simple and effective method to detect edges in an image, but it is susceptible to noise.

The **Sobel** operator (`cv2.Sobel`) can reduce such artifacts, but it is not rotationally symmetric. The **Scharr** operator (`cv2.Scharr`) was targeted at correcting this, but only looks at the first image derivative. If you are interested, there are even more operators for you, such as the **Laplacian** or **ridge** operator (which includes the second derivative), but they are far more complex. And in the end, for our specific purposes, they might not look better, maybe because they are as susceptible to lighting conditions as any other algorithm.

For the purpose of this project, we will choose a function that might not even be associated with conventional edge detection—`cv2.adaptiveThreshold`. Like `cv2.threshold`, this function uses a threshold pixel value to convert a grayscale image into a binary image. That is, if a pixel value in the original image is above the threshold, then the pixel value in the final image will be 255. Otherwise, it will be 0. However, the beauty of adaptive thresholding is that it does not look at the overall properties of the image. Instead, it detects the most salient features in each small neighborhood independently, without regard to the global image optima. This makes the algorithm extremely robust to lighting conditions, which is exactly what we want when we seek to draw bold, black outlines around objects and people in a cartoon.

However, it also makes the algorithm susceptible to noise. To counteract this, we will preprocess the image with a median filter. A median filter does what its name suggests; it replaces each pixel value with the median value of all the pixels in a small pixel neighborhood. We first convert the RGB image (`img_rgb`) to grayscale (`img_gray`) and then apply a median blur with a seven-pixel local neighborhood:

```
# convert to grayscale and apply median blur
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
img_blur = cv2.medianBlur(img_gray, 7)
```

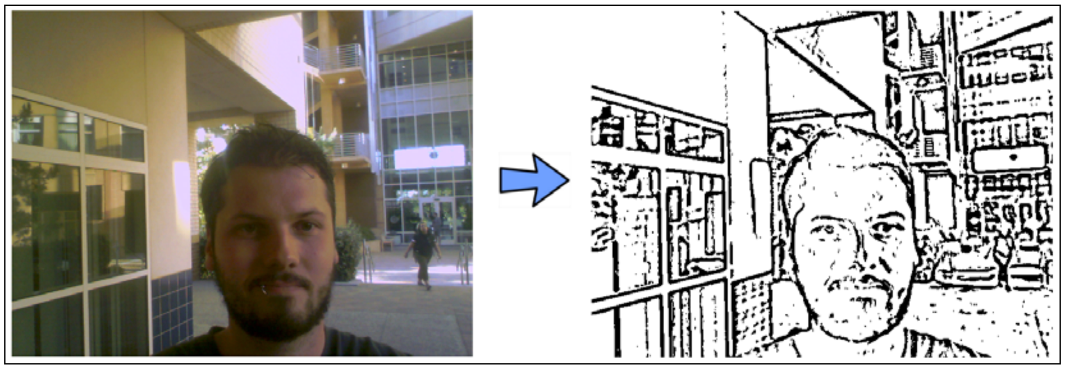
After reducing the noise, it is now safe to detect and enhance the edges using adaptive thresholding. Even if there is some image noise left, the `cv2.ADAPTIVE_THRESH_MEAN_C` algorithm with `blockSize=9` will ensure that the threshold is applied to the mean of a 9 x 9 neighborhood minus `C=2`:

```
img_edge = cv2.adaptiveThreshold(img_blur, 255,
                                 cv2.ADAPTIVE_THRESH_MEAN_C,
                                 cv2.THRESH_BINARY, 9, 2)
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

The result of the adaptive thresholding looks like this:



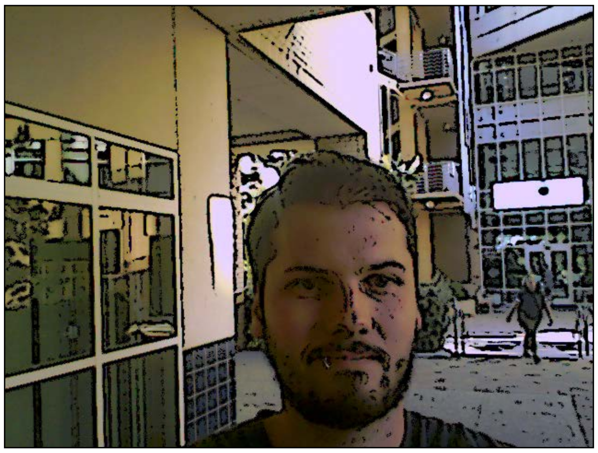
Combining colors and outlines to produce a cartoon

The last step is to combine the two. Simply fuse the two effects together into a single image using `cv2.bitwise_and`. The complete function is as follows:

```
def render(self, img_rgb):
    numDownSamples = 2 # number of downscaling steps
    numBilateralFilters = 7  # number of bilateral filtering steps

    # -- STEP 1 --
    # downsample image using Gaussian pyramid
    img_color = img_rgb
    for _ in xrange(numDownSamples):
        img_color = cv2.pyrDown(img_color)

    # repeatedly apply small bilateral filter instead of applying
    # one large filter
    for _ in xrange(numBilateralFilters):
        img_color = cv2.bilateralFilter(img_color, 9, 9, 7)

    # upsample image to original size
    for _ in xrange(numDownSamples):
        img_color = cv2.pyrUp(img_color)

    # -- STEPS 2 and 3 --
    # convert to grayscale and apply median blur
    img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    img_blur = cv2.medianBlur(img_gray, 7)
```

```
        # -- STEP 4 --
        # detect and enhance edges
        img_edge = cv2.adaptiveThreshold(img_blur, 255,
            cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 9, 2)

        # -- STEP 5 --
        # convert back to color so that it can be bit-ANDed
        # with color image
        img_edge = cv2.cvtColor(img_edge, cv2.COLOR_GRAY2RGB)
        return cv2.bitwise_and(img_color, img_edge)
```

The result looks like what is shown here:



Putting it all together

Before we can make use of the designed image filter effects in an interactive way, we need to set up the main script and design a GUI application.

Running the app

To run the application, we will turn to the `chapter1.py.` script, which we will start by importing all the necessary modules:

```
import numpy as np

import wx
import cv2
```

We will also have to import a generic GUI layout (from `gui`) and all the designed image effects (from `filters`):

```
from gui import BaseLayout
from filters import PencilSketch, WarmingFilter, CoolingFilter,
    Cartoonizer
```

OpenCV provides a straightforward way to access a computer's webcam or camera device. The following code snippet opens the default camera ID (`0`) of a computer using `cv2.VideoCapture`:

```
def main():
    capture = cv2.VideoCapture(0)
```

On some platforms, the first call to `cv2.VideoCapture` fails to open a channel. In that case, we provide a workaround by opening the channel ourselves:

```
if not(capture.isOpened()):
    capture.open()
```

In order to give our application a fair chance to run in real time, we will limit the size of the video stream to 640 x 480 pixels:

```
capture.set(cv2.cv.CV_CAP_PROP_FRAME_WIDTH, 640)
capture.set(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT, 480)
```

If you are using OpenCV 3, the constants that you are looking for might be called `cv3.CAP_PROP_FRAME_WIDTH` and `cv3.CAP_PROP_FRAME_HEIGHT`.

Then the `capture` stream can be passed to our GUI application, which is an instance of the `FilterLayout` class:

```
    # start graphical user interface
    app = wx.App()
    layout = FilterLayout(None, -1, 'Fun with Filters', capture)
    layout.Show(True)
    app.MainLoop()
```

The only thing left to do now is design the said GUI.

The GUI base class

The `FilterLayout` GUI will be based on a generic, plain layout class called `BaseLayout`, which we will be able to use in subsequent chapters as well.

The `BaseLayout` class is designed as an **abstract base class**. You can think of this class as a blueprint or recipe that will apply to all the layouts that we are yet to design—a skeleton class, if you will, that will serve as the backbone for all of our future GUI code. In order to use abstract classes, we need the following `import` statement:

```
from abc import ABCMeta, abstractmethod
```

We also include some other modules that will be helpful, especially the `wx` Python module and OpenCV (of course):

```
import time

import wx
import cv2
```

The class is designed to be derived from the blueprint or skeleton, that is, the `wx.Frame` class. We also mark the class as abstract by adding the `__metaclass__` attribute:

```
class BaseLayout(wx.Frame):
    __metaclass__ = ABCMeta
```

Later on, when we write our own custom layout (`FilterLayout`), we will use the same notation to specify that the class is based on the `BaseLayout` blueprint (or skeleton) class, for example, in `class FilterLayout(BaseLayout):`. But for now, let's focus on the `BaseLayout` class.

An abstract class has at least one abstract method. An abstract method is akin to specifying that a certain method must exist, but we are not sure at that time what it should look like. For example, suppose `BaseLayout` contains a method specified as follows:

```
@abstractmethod
def _init_custom_layout(self):
    pass
```

Then any class deriving from it, such as `FilterLayout`, must specify a fully fleshed-out implementation of a method with that exact signature. This will allow us to create custom layouts, as you will see in a moment.

But first, let's proceed to the GUI constructor.

The GUI constructor

The `BaseLayout` constructor accepts an ID (`-1`), a title string ('`Fun with Filters`'), a video capture object, and an optional argument that specifies the number of frames per second. Then, the first thing to do in the constructor is try and read a frame from the captured object in order to determine the image size:

```
def __init__(self, parent, id, title, capture, fps=10):
    self.capture = capture
    # determine window size and init wx.Frame
    _, frame = self.capture.read()
    self.imgHeight,self.imgWidth = frame.shape[:2]
```

We will use the image size to prepare a buffer that will store each video frame as a bitmap, and to set the size of the GUI. Because we want to display a bunch of control buttons below the current video frame, we set the height of the GUI to `self.imgHeight+20`:

```
self.bmp = wx.BitmapFromBuffer(self.imgWidth,
    self.imgHeight, frame)
wx.Frame.__init__(self, parent, id, title,
        size=(self.imgWidth, self.imgHeight+20))
```

We then provide two methods to initialize some more parameters and create the actual layout of the GUI:

```
self._init_base_layout()
self._create_base_layout()
```

Handling video streams

The video stream of the webcam is handled by a series of steps that begin with the `_init_base_layout` method. These steps might appear overly complicated at first, but they are necessary in order to allow the video to run smoothly, even at higher frame rates (that is, to counteract flicker).

The `wxPython` module works with events and callback methods. When a certain event is triggered, it can cause a certain class method to be executed (in other words, a method can *bind* to an event). We will use this mechanism to our advantage and display a new frame every so often using the following steps:

1. We create a timer that will generate a `wx.EVT_TIMER` event whenever 1000./fps milliseconds have passed:

   ```
   def _init_base_layout(self):
       self.timer = wx.Timer(self)
       self.timer.Start(1000./self.fps)
   ```

2. Whenever the timer is up, we want the `_on_next_frame` method to be called. It will try to acquire a new video frame:

   ```
   self.Bind(wx.EVT_TIMER, self._on_next_frame)
   ```

3. The `_on_next_frame` method will process the new video frame and store the processed frame in a bitmap. This will trigger another event, `wx.EVT_PAINT`. We want to bind this event to the `_on_paint` method, which will paint the display the new frame:

   ```
   self.Bind(wx.EVT_PAINT, self._on_paint)
   ```

The `_on_next_frame` method grabs a new frame and, once done, sends the frame to another method, `__process_frame`, for further processing:

```
def _on_next_frame(self, event):
    ret, frame = self.capture.read()
    if ret:
        frame = self._process_frame(cv2.cvtColor(frame,
            cv2.COLOR_BGR2RGB))
```

The processed frame (`frame`) is then stored in a bitmap buffer (`self.bmp`):

```
self.bmp.CopyFromBuffer(frame)
```

Calling `Refresh` triggers the aforementioned `wx.EVT_PAINT` event, which binds to `_on_paint`:

```
self.Refresh(eraseBackground=False)
```

The paint method then grabs the frame from the buffer and displays it:

```
def _on_paint(self, event):
    deviceContext = wx.BufferedPaintDC(self.pnl)
    deviceContext.DrawBitmap(self.bmp, 0, 0)
```

A basic GUI layout

The creation of the generic layout is done by a method called `_create_base_layout`. The most basic layout consists of only a large black panel that provides enough room to display the video feed:

```
def _create_base_layout(self):
    self.pnl = wx.Panel(self, -1,
                        size=(self.imgWidth, self.imgHeight))
    self.pnl.SetBackgroundColour(wx.BLACK)
```

In order for the layout to be extendable, we add it to a vertically arranged `wx.BoxSizer` object:

```
self.panels_vertical = wx.BoxSizer(wx.VERTICAL)
self.panels_vertical.Add(self.pnl, 1, flag=wx.EXPAND)
```

Next, we specify an abstract method, `_create_custom_layout`, for which we will not fill in any code. Instead, any user of our base class can make their own custom modifications to the basic layout:

```
self._create_custom_layout()
```

Then, we just need to set the minimum size of the resulting layout and center it:

```
self.SetMinSize((self.imgWidth, self.imgHeight))
self.SetSizer(self.panels_vertical)
self.Centre()
```

A custom filter layout

Now we are almost done! If we want to use the `BaseLayout` class, we need to provide code for the three methods that were left blank previously:

- `_init_custom_layout`: This is where we can initialize task-specific parameters
- `_create_custom_layout`: This is where we can make task-specific modifications to the GUI layout
- `_process_frame`: This is where we perform task-specific processing on each captured frame of the camera feed

At this point, initializing the image filters is self-explanatory, as it only requires us to instantiate the corresponding classes:

```
def _init_custom_layout(self):
    self.pencil_sketch = PencilSketch((self.imgWidth,
        self.imgHeight))
    self.warm_filter = WarmingFilter()
    self.cool_filter = CoolingFilter()
    self.cartoonizer = Cartoonizer()
```

To customize the layout, we arrange a number of radio buttons horizontally, one button per image effect mode:

```
def _create_custom_layout(self):
    # create a horizontal layout with all filter modes
    pnl = wx.Panel(self, -1 )
```

```
        self.mode_warm = wx.RadioButton(pnl, -1, 'Warming Filter',
            (10, 10), style=wx.RB_GROUP)
        self.mode_cool = wx.RadioButton(pnl, -1, 'Cooling Filter',
            (10, 10))
        self.mode_sketch = wx.RadioButton(pnl, -1, 'Pencil Sketch',
            (10, 10))
        self.mode_cartoon = wx.RadioButton(pnl, -1, 'Cartoon',
            (10, 10))
        hbox = wx.BoxSizer(wx.HORIZONTAL)
        hbox.Add(self.mode_warm, 1)
        hbox.Add(self.mode_cool, 1)
        hbox.Add(self.mode_sketch, 1)
        hbox.Add(self.mode_cartoon, 1)
        pnl.SetSizer(hbox)
```

Here, the `style=wx.RB_GROUP` option makes sure that only one of these radio buttons can be selected at a time.

To make these changes take effect, `pnl` needs to be added to list of existing panels:

```
    self.panels_vertical.Add(pnl, flag=wx.EXPAND | wx.BOTTOM | wx.TOP,
                border=1)
```

The last method to be specified is `_process_frame`. Recall that this method is triggered whenever a new camera frame is received. All that we need to do is pick the right image effect to be applied, which depends on the radio button configuration. We simply check which of the buttons is currently selected and call the corresponding render method:

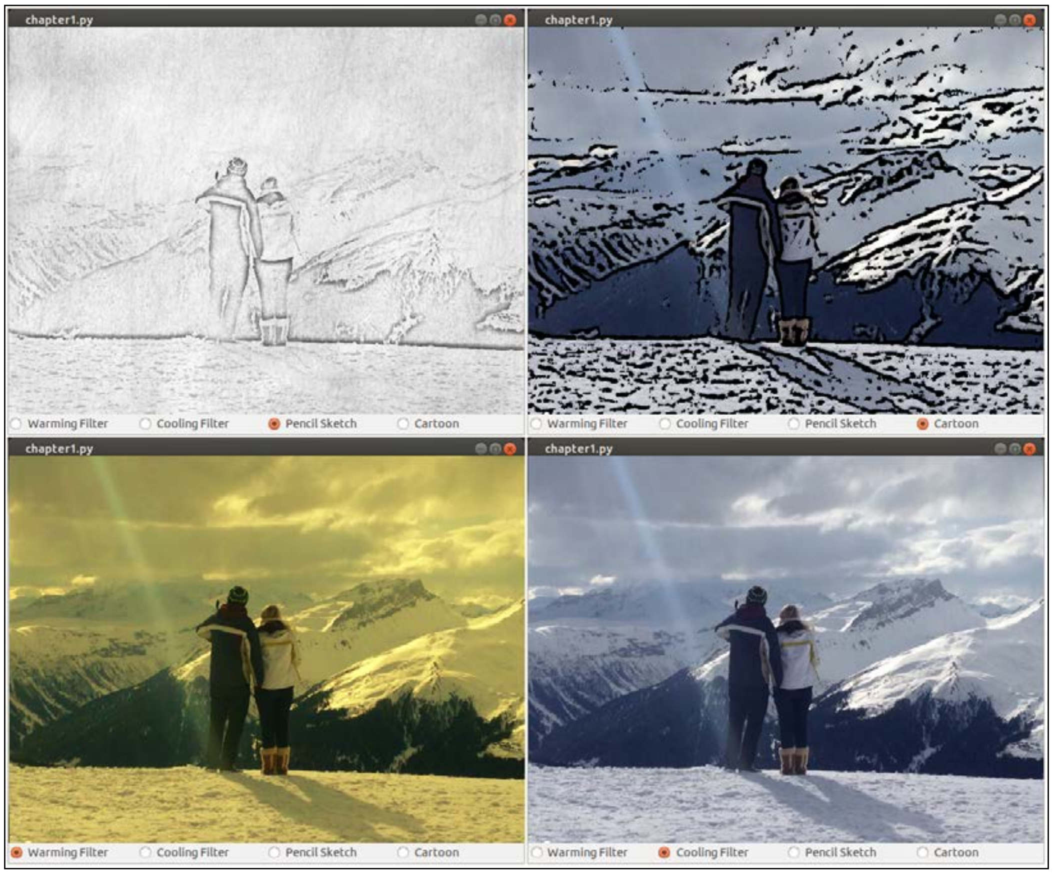
```
    def _process_frame(self, frame_rgb):
        if self.mode_warm.GetValue():
            frame = self.warm_filter.render(frame_rgb)
        elif self.mode_cool.GetValue():
            frame = self.cool_filter.render(frame_rgb)
        elif self.mode_sketch.GetValue():
            frame = self.pencil_sketch.render(frame_rgb)
        elif self.mode_cartoon.GetValue():
            frame = self.cartoonizer.render(frame_rgb)
```

Don't forget to return the processed frame:

```
    return frame
```

And we're done!

Here is the result:



Summary

In this chapter, we explored a number of interesting image processing effects. We used dodging and burning to create a black-and-white pencil sketch effect, explored lookup tables to arrive at an efficient implementation of curve filters, and got creative to produce a cartoon effect.

In the next chapter, we will shift gears a bit and explore the use of depth sensors, such as Microsoft Kinect 3D, to recognize hand gestures in real time.