OpenCV for Secret Agents

OpenCV is a grand collection of image processing functions and computer vision algorithms. It is open source, it supports many programming languages and platforms, and it is fast enough for many real-time applications. What a lot of gadgets we can build with such a handy library!

Taking inspiration from the world of James Bond, this book adds a spark of adventure and computer vision to your daily routine. Protect your home and car with intelligent camera systems that analyze people, cats, and obstacles. Let your search engine praise or criticize the images that it finds. Hear a voice from your phone that responds to your body language. Attune yourself to another person’s rhythm by glancing at a display that magnifies a heartbeat or a breath. Learn OpenCV and see your world as never before.

Who this book is written for

This book is for programmers who want to expand their skills by building fun, smart, and useful systems with OpenCV. The projects are ideal in helping you to think creatively about the uses of computer vision, natural user interfaces, and ubiquitous computers (in your home, car, and hand).

What you will learn from this book

- Install OpenCV, a Python development environment, and an Android development environment on Windows, Mac, or Linux and install a Unity development environment on Windows or Mac
- Get to grips with motion detection and gesture recognition as a means of controlling a guessing game on a smartphone
- Detect car headlights, estimate distances to them, and provide feedback to the driver
- Spot and recognize human faces and cat faces as a means of controlling an alarm
- Amplify motion in real-time video so that a person’s heartbeat and breathing become clearly visible
- Draw a ball-in-a-maze puzzle on paper and see it come to life as a physics simulation on a smartphone
- Integrate OpenCV with other libraries, as well as popular frameworks for GUI apps and games

Use OpenCV in six secret projects to augment your home, car, phone, eyesight, and any photo or drawing.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 'Seeing a Heartbeat with a Motion Amplifying Camera'
- A synopsis of the book’s content
- More information on OpenCV for Secret Agents

About the Author

Joseph Howse has four first-rate cats; yet, if his books sell well, he could build a menagerie fit for a pharaoh.

OpenCV for Secret Agents is Joseph’s third book, following OpenCV Computer Vision with Python and Android Application Programming with OpenCV. When not writing books or grooming cats, Joseph is working to grow the augmented reality industry by providing software development and training services through his company, Nummist Media (http://nummist.com).
OpenCV for Secret Agents

Computer vision systems are deployed in the Arctic Ocean to spot icebergs at night. They are flown over the Amazon rainforest to create aerial maps of fires, blights, and illegal logging. They are set up in ports and airports worldwide to scan for suspects and contraband. They are sent to the depths of the Marianas Trench to guide autonomous submarines. They are used in operating rooms to help surgeons visualize the planned procedure and the patient's current condition. They are launched from battlefields as the steering systems of heat-seeking, anti-aircraft rockets.

We might seldom—or never—visit these places. However, stories often encourage us to imagine extreme environments and a person's dependence on tools in these unforgiving conditions. Perhaps fittingly, one of contemporary fiction's most popular characters is an almost ordinary man (handsome but not too handsome, clever but not too clever) who wears a suit, works for the British Government, always chooses the same drink, the same kind of woman, the same tone for delivering a pun, and is sent to do dangerous jobs with a peculiar collection of gadgets.

Bond. James Bond.

This book teaches seriously useful technologies and techniques with a healthy dose of inspiration from spy fiction. The Bond franchise is rich in ideas about detection, disguise, smart devices, image capture, and sometimes even computer vision specifically. With imagination, plus dedication of learning new skills, we can become the next generation of gadget makers to rival Bond's engineer, Q!

What This Book Covers

Chapter 1, Preparing for the Mission helps us to install OpenCV, a Python development environment, and an Android development environment on Windows, Mac, or Linux systems. In this chapter, we also install a Unity development environment on Windows or Mac.

Chapter 2, Searching for Luxury Accommodations Worldwide helps us to classify images of real estate based on color schemes. Are we outside a luxury dwelling or inside a Stalinist apartment? In this chapter, we use the classifier in a search engine that labels its image results.

Chapter 3, Training a Smart Alarm to Recognize the Villain and His Cat helps us to detect and recognize human faces and cat faces as a means of controlling an alarm. Has Ernst Stavro Blofeld returned with his blue-eyed Angora cat?

Chapter 4, Controlling a Phone App with Your Suave Gestures helps us to detect motion and recognize gestures as a means of controlling a guessing game on a smartphone. The phone knows why Bond is nodding even if no one else does.
Chapter 5, *Equipping Your Car with a Rearview Camera and Hazard Detection* helps us to detect car headlights, classify their color, estimate distances to them, and provide feedback to a driver. Is that car tailing us?

Chapter 6, *Seeing a Heartbeat with a Motion Amplifying Camera* helps us to amplify motion in live video, in real time, so that a person's heartbeat and breathing become clearly visible. See the passion!

Chapter 7, *Creating a Physics Simulation Based on a Pen and Paper Sketch* helps us to draw a ball-in-a-maze puzzle on paper and see it come to life as a physics simulation on a smartphone. Physics and timing are everything!
Seeing a Heartbeat with a Motion Amplifying Camera

Remove everything that has no relevance to the story. If you say in the first chapter that there is a rifle hanging on the wall, in the second or third chapter it absolutely must go off. If it's not going to be fired, it shouldn't be hanging there.

— Anton Chekhov

King Julian: I don't know why the sacrifice didn't work. The science seemed so solid.


Despite their strange design and mysterious engineering, Q's gadgets always prove useful and reliable. Bond has such faith in the technology that he never even asks how to charge the batteries.

One of the inventive ideas in the Bond franchise is that even a lightly equipped spy should be able to see and photograph concealed objects, anywhere, anytime. Let's consider a timeline of a few relevant gadgets in the movies:

- 1979 (Moonraker): A cigarette case contains an X-ray imaging system that is used to reveal the tumblers of a safe's combination lock.
- 1989 (License to Kill): A Polaroid camera takes X-ray photos. Oddly enough, its flash is a visible, red laser.
• 1995 (*GoldenEye*): A tea tray contains an X-ray scanner that can photograph documents beneath the tray.

• 1999 (*The World is Not Enough*): Bond wears a stylish pair of blue-lensed glasses that can see through one layer of clothing to reveal concealed weapons. According to the *James Bond Encyclopedia* (2007), which is an official guide to the movies, the glasses display infrared video after applying special processing to it. Despite using infrared, they are commonly called X-ray specs, a misnomer.

These gadgets deal with unseen wavelengths of light (or radiation) and are broadly comparable to real-world devices such as airport security scanners and night vision goggles. However, it remains difficult to explain how Bond's equipment is so compact and how it takes such clear pictures in diverse lighting conditions and through diverse materials. Moreover, if Bond's devices are active scanners (meaning they emit X-ray radiation or infrared light), they will be clearly visible to other spies using similar hardware.

To take another approach, what if we avoid unseen wavelengths of light but instead focus on unseen frequencies of motion? Many things move in a pattern that is too fast or too slow for us to easily notice. Suppose that a man is standing in one place. If he shifts one leg more than the other, perhaps he is concealing a heavy object, such as a gun, on the side that he shifts more. We also might fail to notice deviations from a pattern. Suppose the same man has been looking straight ahead but suddenly, when he believes no one is looking, his eyes dart to one side. Is he watching someone?

We can make motions of a certain frequency more visible by repeating them, like a delayed afterimage or a ghost, with each repetition being more faint (less opaque) than the last. The effect is analogous to an echo or a ripple, and it is achieved using an algorithm called *Eulerian video magnification*.

By applying this technique, we will build a desktop app that allows us to simultaneously see the present and selected slices of the past. The idea of experiencing multiple images simultaneously is, to me, quite natural because for the first 26 years of my life, I had strabismus—commonly called a *lazy eye*—that caused double vision. A surgeon corrected my eyesight and gave me depth perception but, in memory of strabismus, I would like to name this application Lazy Eyes.

Let’s take a closer look—or two or more closer looks—at the fast-paced, moving world that we share with all the other secret agents.

The completed project for this chapter can be downloaded from my website at [http://nummist.com/opencv/7376_06.zip](http://nummist.com/opencv/7376_06.zip).
Planning the Lazy Eyes app

Of all our apps, Lazy Eyes has the simplest user interface. It just shows a live video feed with a special effect that highlights motion. The parameters of the effect are quite complex and, moreover, modifying them at runtime would have a big effect on performance. Thus, we do not provide a user interface to reconfigure the effect, but we do provide many parameters in code to allow a programmer to create many variants of the effect and the app.

The following is a screenshot illustrating one configuration of the app. This image shows me eating cake. My hands and face are moving often and we see an effect that looks like light and dark waves rippling around the places where moving edges have been. (The effect is more graceful in a live video than in a screenshot.)

For more screenshots and an in-depth discussion of the parameters, refer to the section Configuring and testing the app for various motions, later in this chapter.
Regardless of how it is configured, the app loops through the following actions:

1. Capturing an image.
2. Copying and downsampling the image while applying a blur filter and optionally an edge finding filter. We will downsample using so-called image pyramids, which will be discussed in Compositing two images using image pyramids, later in this chapter. The purpose of downsampling is to achieve a higher frame rate by reducing the amount of image data used in subsequent operations. The purpose of applying a blur filter and optionally an edge finding filter is to create haloes that are useful in amplifying motion.
3. Storing the downsampled copy in a history of frames, with a timestamp. The history has a fixed capacity and once it is full, the oldest frame is overwritten to make room for the new one.
4. If the history is not yet full, we continue to the next iteration of the loop.
5. Decomposing the history into a list of frequencies describing fluctuations (motion) at each pixel. The decomposition function is called a Fast Fourier Transform. We will discuss this in the Extracting repeating signals from video using the Fast Fourier Transform section, later in this chapter.
6. Setting all frequencies to zero except a certain chosen range that interests us. In other words, filter out the data on motions that are faster or slower than certain thresholds.
7. Recomposing the filtered frequencies into a series of images that are motion maps. Areas that are still (with respect to our chosen range of frequencies) become dark, and areas that are moving might remain bright. The recomposition function is called an Inverse Fast Fourier Transform (IFFT), and we will discuss it later alongside the FFT.
8. Upsampling the latest motion map (again using image pyramids), intensifying it, and overlaying it additively atop the original camera image.
9. Showing the resulting composite image.

There it is—a simple plan that requires a rather nuanced implementation and configuration. Let's prepare ourselves by doing a little background research.
Understanding what Eulerian video magnification can do

Eulerian video magnification is inspired by a model in fluid mechanics called Eulerian specification of the flow field. Let's consider a moving, fluid body, such as a river. The Eulerian specification describes the river's velocity at a given position and time. The velocity would be fast in the mountains in springtime and slow at the river's mouth in winter. Also, the velocity would be slower at a silt-saturated point at the river's bottom, compared to a point where the river's surface hits a rock and sprays. An alternative to the Eulerian specification is the Lagrangian specification, which describes the position of a given particle at a given time. A given bit of silt might make its way down from the mountains to the river's mouth over a period of many years and then spend eons drifting around a tidal basin.


The Lagrangian specification is analogous to many computer vision tasks, in which we model the motion of a particular object or feature over time. However, the Eulerian specification is analogous to our current task, in which we model any motion occurring in a particular position and a particular window of time. Having modeled a motion from an Eulerian perspective, we can visually exaggerate the motion by overlaying the model's results for a blend of positions and times.

Let's set a baseline for our expectations of Eulerian video magnification by studying other people's projects:


- Bryce Drennan's Eulerian-magnification library ([https://github.com/brycedrennan/eulerian-magnification](https://github.com/brycedrennan/eulerian-magnification)) implements the algorithm using NumPy, SciPy, and OpenCV. This implementation is good inspiration for us, but it is designed to process prerecorded videos and is not sufficiently optimized for real-time input.

Now, let's discuss the functions that are building blocks of these projects and ours.
Extracting repeating signals from video using the Fast Fourier Transform (FFT)

An audio signal is typically visualized as a bar chart or wave. The bar chart or wave is high when the sound is loud and low when it is soft. We recognize that a repetitive sound, such as a metronome's beat, makes repetitive peaks and valleys in the visualization. When audio has multiple channels (being a stereo or surround sound recording), each channel can be considered as a separate signal and can be visualized as a separate bar chart or wave.

Similarly, in a video, every channel of every pixel can be considered as a separate signal, rising and falling (becoming brighter and dimmer) over time. Imagine that we use a stationary camera to capture a video of a metronome. Then, certain pixel values rise and fall at a regular interval as they capture the passage of the metronome's needle. If the camera has an attached microphone, its signal values rise and fall at the same interval. Based on either the audio or the video, we can measure the metronome's frequency—its beats per minute (bpm) or its beats per second (Hertz or Hz). Conversely, if we change the metronome's bpm setting, the effect on both the audio and the video is predictable. From this thought experiment, we can learn that a signal—be it audio, video, or any other kind—can be expressed as a function of time and, equivalently, a function of frequency.

Consider the following pair of graphs. They express the same signal, first as a function of time and then as a function of frequency. Within the time domain, we see one wide peak and valley (in other words, a tapering effect) spanning many narrow peaks and valleys. Within the frequency domain, we see a low-frequency peak and a high-frequency peak.
The transformation from the time domain to the frequency domain is called the Fourier transform. Conversely, the transformation from the frequency domain to the time domain is called the inverse Fourier transform. Within the digital world, signals are discrete, not continuous, and we use the terms discrete Fourier transform (DFT) and inverse discrete Fourier transform (IDFT). There is a variety of efficient algorithms to compute the DFT or IDFT and such an algorithm might be described as a Fast Fourier Transform or an Inverse Fast Fourier Transform.

For algorithmic descriptions, refer to the following Wikipedia article:

The result of the Fourier transform (including its discrete variants) is a function that maps a frequency to an amplitude and phase. The amplitude represents the magnitude of the frequency's contribution to the signal. The phase represents a temporal shift; it determines whether the frequency's contribution starts on a high or a low. Typically, amplitude and phase are encoded in a complex number, $a+bi$, where $amplitude=\sqrt{a^2+b^2}$ and $phase=\text{atan2}(a,b)$.

For an explanation of complex numbers, refer to the following Wikipedia article:

The FFT and IFFT are fundamental to a field of computer science called digital signal processing. Many signal processing applications, including Lazy Eyes, involve taking the signal's FFT, modifying or removing certain frequencies in the FFT result, and then reconstructing the filtered signal in the time domain using the IFFT. For example, this approach enables us to amplify certain frequencies while leaving others unchanged.

Now, where do we find this functionality?

**Choosing and setting up an FFT library**

Several Python libraries provide FFT and IFFT implementations that can process NumPy arrays (and thus OpenCV images). Here are the five major contenders:

- **NumPy**: This provides FFT and IFFT implementations in a module called `numpy.fft` (for more information, refer to http://docs.scipy.org/doc/numpy/reference/routines.fft.html). The module also offers other signal processing functions to work with the output of the FFT.
• **SciPy**: This provides FFT and IFFT implementations in a module called `scipy.fftpack` (for more information refer to http://docs.scipy.org/doc/scipy/reference/fftpack.html). This SciPy module is closely based on the `numpy.fft` module, but adds some optional arguments and dynamic optimizations based on the input format. The SciPy module also adds more signal processing functions to work with the output of the FFT.

• **OpenCV**: This has implementations of FFT (`cv2.dft`) and IFT (`cv2.idft`). An official tutorial provides examples and a comparison to NumPy's FFT implementation at http://docs.opencv.org/doc/tutorials/core/discrete_fourier_transform/discrete_fourier_transform.html. OpenCV's FFT and IFT interfaces are not directly interoperable with the `numpy.fft` and `scipy.fftpack` modules that offer a broader range of signal processing functionality. (The data is formatted very differently.)

• **PyFFTW**: This is a Python wrapper (https://hgomersall.github.io/pyFFTW/) around a C library called the *Fastest Fourier Transform in the West* (FFTW) (for more information, refer to http://www.fftw.org/). FFTW provides multiple implementations of FFT and IFFT. At runtime, it dynamically selects implementations that are well optimized for given input formats, output formats, and system capabilities. Optionally, it takes advantage of multithreading (and its threads might run on multiple CPU cores, as the implementation releases Python's Global Interpreter Lock or GIL). PyFFTW provides optional interfaces matching NumPy's and SciPy's FFT and IFFT functions. These interfaces have a low overhead cost (thanks to good caching options that are provided by PyFFTW) and they help to ensure that PyFFTW is interoperable with a broader range of signal processing functionality as implemented in `numpy.fft` and `scipy.fftpack`.

• **Reinka**: This is a Python library for GPU-accelerated computations using either PyCUDA (http://mathema.tician.de/software/pycuda/) or PyOpenCL (http://mathema.tician.de/software/pyopencl/) as a backend. Reinka (http://reikna.publicfields.net/en/latest/) provides FFT and IFFT implementations in a module called `reikna.fft`. Reinka internally uses PyCUDA or PyOpenCL arrays (not NumPy arrays), but it provides interfaces for conversion from NumPy arrays to these GPU arrays and back. The converted NumPy output is compatible with other signal processing functionality as implemented in `numpy.fft` and `scipy.fftpack`. However, this compatibility comes at a high overhead cost due to locking, reading, and converting the contents of the GPU memory.

NumPy, SciPy, OpenCV, and PyFFTW are open-source libraries under the BSD license. Reinka is an open-source library under the MIT license.
I recommend PyFFTW because of its optimizations and its interoperability (at a low overhead cost) with all the other functionality that interests us in NumPy, SciPy, and OpenCV. For a tour of PyFFTW’s features, including its NumPy- and SciPy-compatible interfaces, refer to the official tutorial at https://hgomersall.github.io/pyFFTW/sphinx/tutorial.html.

Depending on our platform, we can set up PyFFTW in one of the following ways:

- In Windows, download and run a binary installer from https://pypi.python.org/pypi/pyFFTW. Choose the installer for either a 32-bit Python 2.7 or 64-bit Python 2.7 (depending on whether your Python installation, not necessarily your system, is 64-bit).
- In Mac with MacPorts, run the following command in Terminal:
  
  ```bash
  $ sudo port install py27-pyfftw
  ```
- In Ubuntu 14.10 (Utopic) and its derivatives, including Linux Mint 14.10, run the following command in Terminal:
  
  ```bash
  $ sudo apt-get install python-fftw3
  ```
  
  In Ubuntu 14.04 and earlier versions (and derivatives thereof), do not use this package, as its version is too old. Instead, use the PyFFTW source bundle, as described in the last bullet of this list.
- In Debian Jessie, Debian Sid, and their derivatives, run the following command in Terminal:
  
  ```bash
  $ sudo apt-get install python-pyfftw
  ```
  
  In Debian Wheezy and its derivatives, including Raspbian, this package does not exist. Instead, use the PyFFTW source bundle, as described in the next bullet.
- For any other system, download the PyFFTW source bundle from https://pypi.python.org/pypi/pyFFTW. Decompress it and run the setup.py script inside the decompressed folder.

Some old versions of the library are called PyFFTW3. We do not want PyFFTW3. However, on Ubuntu 14.10 and its derivatives, the packages are misnamed such that python-fftw3 is really the most recent packaged version (whereas python-fftw is an older PyFFTW3 version).

We have our FFT and IFFT needs covered by FFTW (and if we were cowboys instead of secret agents, we could say, "Cover me!"). For additional signal processing functionality, we will use SciPy, which can be set up in the way described in Chapter 1, Preparing for the Mission, Setting up a development machine.
Signal processing is not the only new material that we must learn for Lazy Eyes, so let’s now look at other functionality that is provided by OpenCV.

**Compositing two images using image pyramids**

Running an FFT on a full-resolution video feed would be slow. Also, the resulting frequencies would reflect localized phenomena at each captured pixel, such that the motion map (the result of filtering the frequencies and then applying the IFFT) might appear noisy and overly sharpened. To address these problems, we want a cheap, blurry downsampling technique. However, we also want the option to enhance edges, which are important to our perception of motion.

Our need for a blurry downsampling technique is fulfilled by a **Gaussian image pyramid**. A Gaussian filter blurs an image by making each output pixel a weighted average of many input pixels in the neighborhood. An image pyramid is a series in which each image is half the width and height of the previous image. The halving of image dimensions is achieved by **decimation**, meaning that every other pixel is simply omitted. A Gaussian image pyramid is constructed by applying a Gaussian filter before each decimation operation.

Our need to enhance edges in downsampled images is fulfilled by a **Laplacian image pyramid**, which is constructed in the following manner. Suppose we have already constructed a Gaussian image pyramid. We take the image at level $i+1$ in the Gaussian pyramid, upsample it by duplicating the pixels, and apply a Gaussian filter to it again. We then subtract the result from the image at level $i$ in the Gaussian pyramid to produce the corresponding image at level $i$ of the Laplacian pyramid. Thus, the Laplacian image is the difference between a blurry, downsampled image and an even blurrier image that was downsampled, downsampled again, and upsampled.

You might wonder how such an algorithm is a form of edge finding. Consider that edges are areas of local contrast, while non-edges are areas of local uniformity. If we blur a uniform area, it is still uniform—zero difference. If we blur a contrasting area, it becomes more uniform—nonzero difference. Thus, the difference can be used to find edges.

Besides using image pyramids to downsample the FFT's input, we also use them to upsample the most recent frame of the IFFT's output. This upsampling step is necessary to create an overlay that matches the size of the original camera image so that we can composite the two. Like in the construction of the Laplacian pyramid, upsampling consists of duplicating pixels and applying a Gaussian filter.

OpenCV implements the relevant downsizing and upsizing functions as cv2.pyrDown and cv2.pyrUp. These functions are useful in compositing two images in general (whether or not signal processing is involved) because they enable us to soften differences while preserving edges. The OpenCV documentation includes a good tutorial on this topic at http://docs.opencv.org/trunk/doc/py_tutorials/py_imgproc/py_pyramids/py_pyramids.html.

Now, we are armed with the knowledge to implement Lazy Eyes!

Implementing the Lazy Eyes app

Let's create a new folder for Lazy Eyes and, in this folder, create copies of or links to the ResizeUtils.py and WxUtils.py files from any of our previous Python projects such as the project The Living Headlights in Chapter 5, Equipping Your Car with a Rearview Camera and Hazard Detection. Alongside the copies or links, let's create a new file, LazyEyes.py. Edit it and enter the following import statements:

```python
import collections
import numpy
import cv2
import threading
import timeit
import wx

import pyfftw.interfaces.cache
from pyfftw.interfaces.scipy_fftpack import fft
from pyfftw.interfaces.scipy_fftpack import ifft
```
from scipy.fftpack import fftfreq
import ResizeUtils
import WxUtils

Besides the modules that we have used in previous projects, we are now using the standard library's collections module for efficient collections and timeit module for precise timing. Also for the first time, we are using signal processing functionality from PyFFTW and SciPy.

Like our other Python applications, Lazy Eyes is implemented as a class that extends wx.Frame. Here are the declarations of the class and its initializer:

class LazyEyes(wx.Frame):
    def __init__(self, maxHistoryLength=360,
                 minHz=5.0/6.0, maxHz=1.0,
                 amplification=32.0, numPyramidLevels=2,
                 useLaplacianPyramid=True,
                 useGrayOverlay=True,
                 numFFTThreads = 4, numIFFTThreads=4,
                 cameraDeviceID=0, imageSize=(480, 360),
                 title='Lazy Eyes'):

The initializer's arguments affect the app's frame rate and the manner in which motion is amplified. These effects are discussed in detail in the section Configuring and testing the app for various motions, later in this chapter. The following is just a brief description of the arguments:

- maxHistoryLength is the number of frames (including the current frame and preceding frames) that are analyzed for motion.
- minHz and maxHz, respectively, define the slowest and fastest motions that are amplified.
- amplification is the scale of the visual effect. A higher value means motion is highlighted more brightly.
- numPyramidLevels is the number of pyramid levels by which frames are downsampled before signal processing is done. Remember that each level corresponds to downsampling by a factor of 2. Our implementation assumes numPyramidLevels>0.

If useLaplacianPyramid is True, frames are downsampled using a Laplacian pyramid before the signal processing is done. The implication is that only edge motion is highlighted. Alternatively, if useLaplacianPyramid is False, a Gaussian pyramid is used, and the motion in all areas is highlighted.
• If `useGrayOverlay` is True, frames are converted to grayscale before the signal processing is done. The implication is that motion is only highlighted in areas of grayscale contrast. Alternatively, if `useGrayOverlay` is False, motion is highlighted in areas that have contrast in any color channel.

• `numFFTThreads` and `numIFFTThreads`, respectively, are the numbers of threads used in FFT and IFFT computations.

• `cameraDeviceID` and `imageSize` are our usual capture parameters.

The initializer's implementation begins in the same way as our other Python apps. It sets flags to indicate that the app is running and (by default) should be mirrored. It creates the capture object and configures its resolution to match the requested width and height if possible. Failing that, the device's default capture resolution is used. The relevant code is as follows:

```python
self.mirrored = True
self._running = True

self._capture = cv2.VideoCapture(cameraDeviceID)
size = ResizeUtils.cvResizeCapture(
    self._capture, imageSize)
w, h = size
self._imageWidth = w
self._imageHeight = h
```

Next, we will determine the shape of the history of frames. It has at least 3 dimensions: a number of frames, a width, and height for each frame. The width and height are downsampled from the capture width and height based on the number of pyramid levels. If we are concerned about the color motion and not just the grayscale motion, the history also has a fourth dimension, consisting of 3 color channels. Here is the code to calculate the history's shape:

```python
self._useGrayOverlay = useGrayOverlay
if useGrayOverlay:
    historyShape = (maxHistoryLength, h >> numPyramidLevels, w >> numPyramidLevels)
else:
    historyShape = (maxHistoryLength, h >> numPyramidLevels, w >> numPyramidLevels, 3)
```

Note the use of `>>`, the right bit shift operator, to reduce the dimensions by a power of 2. The power is equal to the number of pyramid levels.
We will store the specified maximum history length. For the frames in the history, we will create a NumPy array of the shape that we just determined. For timestamps of the frames, we will create a deque (double-ended queue), a type of collection that allows us to cheaply add or remove elements from either end:

```python
self._maxHistoryLength = maxHistoryLength
self._history = numpy.empty(historyShape, numpy.float32)
self._historyTimestamps = collections.deque()
```

We will store the remaining arguments because later, in each frame, we will pass them to the pyramid functions and signal the processing functions:

```python
self._numPyramidLevels = numPyramidLevels
self._useLaplacianPyramid = useLaplacianPyramid
self._minHz = minHz
self._maxHz = maxHz
self._amplification = amplification
self._numFFTThreads = numFFTThreads
self._numIFFTThreads = numIFFTThreads
```

To ensure meaningful error messages and early termination in the case of invalid arguments, we could add code such as the following for each argument:

```python
assert numPyramidLevels > 0, \\n  'numPyramidLevels must be positive.'
```

For brevity, such assertions are omitted from our code samples.

We call the following two functions to tell PyFFTW to cache its data structures (notably, its NumPy arrays) for a period of at least 1.0 second from their last use. (The default is 0.1 seconds.) Caching is a critical optimization for the PyFFTW interfaces that we are using, and we will choose a period that is more than long enough to keep the cache alive from frame to frame:

```python
pyfftw.interfaces.cache.enable()
pyfftw.interfaces.cache.set_keepalive_time(1.0)
```

The initializer's implementation ends with code to set up a window, event bindings, a bitmap, layout, and background thread—all familiar tasks from our previous Python projects:

```python
style = wx.CLOSE_BOX | wx.MINIMIZE_BOX | \n      wx.CAPTION | wx.SYSTEM_MENU | \n```
wx.CLIP_CHILDREN
wx.Frame.__init__(self, None, title=title,
    style=style, size=size)
self.Bind(wx.EVT_CLOSE, self._onCloseWindow)
quitCommandID = wx.NewId()
self.Bind(wx.EVT_MENU, self._onQuitCommand,
    id=quitCommandID)
acceleratorTable = wx.AcceleratorTable([ (wx.ACCEL_NORMAL, wx.WXK_ESCAPE,
    quitCommandID)
])
self.SetAcceleratorTable(acceleratorTable)
self._staticBitmap = wx.StaticBitmap(self,
    size=size)
self._showImage(None)
rootSizer = wx.BoxSizer(wx.VERTICAL)
rootSizer.Add(self._staticBitmap)
self.SetSizerAndFit(rootSizer)
self._captureThread = threading.Thread(
    target=self._runCaptureLoop)
self._captureThread.start()

We must modify our usual _onCloseWindow callback to disable PyFFTW's cache. Disabling the cache ensures that resources are freed and that PyFFTW's threads terminate normally. The callback's implementation is given in the following code:

def _onCloseWindow(self, event):
    self._running = False
    self._captureThread.join()
    pyfftw.interfaces.cache.disable()
    self.Destroy()

The Esc key is bound to our usual _onQuitCommand callback, which just closes the app:

def _onQuitCommand(self, event):
    self.Close()
The loop running on our background thread is similar to the one in our other Python apps. In each frame, it calls a helper function, `_applyEulerianVideoMagnification`. Here is the loop's implementation:

```python
def _runCaptureLoop(self):
    while self._running:
        success, image = self._capture.read()
        if image is not None:
            self._applyEulerianVideoMagnification(image)
            if (self.mirrored):
                image[:] = numpy.fliplr(image)
            wx.CallAfter(self._showImage, image)
```

The `_applyEulerianVideoMagnification` helper function is quite long so we will consider its implementation in several chunks. First, we will create a timestamp for the frame and copy the frame to a format that is more suitable for processing. Specifically, we will use a floating point array with either one gray channel or 3 color channels, depending on the configuration:

```python
def _applyEulerianVideoMagnification(self, image):
    timestamp = timeit.default_timer()

    if self._useGrayOverlay:
        smallImage = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY).astype(numpy.float32)
    else:
        smallImage = image.astype(numpy.float32)

    Using this copy, we will calculate the appropriate level in the Gaussian or Laplacian pyramid:

    # Downsample the image using a pyramid technique.
    i = 0
    while i < self._numPyramidLevels:
        smallImage = cv2.pyrDown(smallImage)
        i += 1
    if self._useLaplacianPyramid:
        smallImage[:] -= cv2.pyrUp(cv2.pyrDown(smallImage))
```

For the purposes of the history and signal processing functions, we will refer to this pyramid level as "the image" or "the frame".

Next, we will check the number of history frames that have been filled so far. If the history has more than one unfilled frame (meaning the history will still not be full after adding this frame), we will append the new image and timestamp and then return early, such that no signal processing is done until a later frame:

```python
historyLength = len(self._historyTimestamps)
if historyLength < self._maxHistoryLength - 1:
    # Append the new image and timestamp to the history.
    self._history[historyLength] = smallImage
    self._historyTimestamps.append(timestamp)
    # The history is still not full, so wait.
    return
```

If the history is just one frame short of being full (meaning the history will be full after adding this frame), we will append the new image and timestamp using the code given as follows:

```python
if historyLength == self._maxHistoryLength - 1:
    # Append the new image and timestamp to the history.
    self._history[historyLength] = smallImage
    self._historyTimestamps.append(timestamp)
```

If the history is already full, we will drop the oldest image and timestamp and append the new image and timestamp using the code given as follows:

```python
else:
    # Drop the oldest image and timestamp from the history and append the new ones.
    self._history[::] = self._history[1::]
    self._historyTimestamps.popleft()
    self._history[-1] = smallImage
    self._historyTimestamps.append(timestamp)
    # The history is full, so process it.
```
The history of image data is a NumPy array and, as such, we are using the terms "append" and "drop" loosely. NumPy arrays are immutable, meaning they cannot grow or shrink. Moreover, we are not recreating this array because it is large and reallocating it each frame would be expensive. We are just overwriting data within the array by moving the old data leftward and copying the new data in.

Based on the timestamps, we will calculate the average time per frame in the history, as seen in the following code:

```python
# Find the average length of time per frame.
startTime = self._historyTimestamps[0]
endTime = self._historyTimestamps[-1]
timeElapsed = endTime - startTime
timePerFrame = \
    timeElapsed / self._maxHistoryLength
# print 'FPS:', 1.0 / timePerFrame
```

We will proceed with a combination of signal processing functions, collectively called a temporal bandpass filter. This filter blocks (zeros out) some frequencies and allows others to pass (remain unchanged). Our first step in implementing this filter is to run the `pyfftw.interfaces.scipy_fftpack.fft` function using the history and a number of threads as arguments. Also, with the argument `axis=0`, we will specify that the history's first axis is its time axis:

```python
# Apply the temporal bandpass filter.
fftResult = fft(self._history, axis=0,
    threads=self._numFFTThreads)
```

We will pass the FFT result and the time per frame to the `scipy.fftpack.fftfreq` function. This function returns an array of midpoint frequencies (Hz in our case) corresponding to the indices in the FFT result. (This array answers the question, "Which frequency is the midpoint of the bin of frequencies represented by index \(i\) in the FFT?\).) We will find the indices whose midpoint frequencies lie closest (minimum absolute value difference) to our initializer's `minHz` and `maxHz` parameters. Then, we will modify the FFT result by setting the data to zero in all ranges that do not represent the frequencies of interest:

```python
frequencies = fftfreq(
    self._maxHistoryLength, d=timePerFrame)
lowBound = (numpy.abs(
    frequencies - self._minHz)).argmin()
highBound = (numpy.abs(
    ``
The FFT result is symmetrical: \texttt{fftResult[i]} and \texttt{fftResult[-i]} pertain to the same bin of frequencies. Thus, we will modify the FFT result symmetrically.

Remember, the Fourier transform maps a frequency to a complex number that encodes an amplitude and phase. Thus, while the indices of the FFT result correspond to frequencies, the values contained at those indices are complex numbers. Zero as a complex number is written in Python as \texttt{0+0j} or \texttt{0j}.

Having thus filtered out the frequencies that do not interest us, we will finish applying the temporal bandpass filter by passing the data to the \texttt{pyfftw.interfaces.scipy_fftpack.ifft} function:

\begin{verbatim}
ifftResult = ifft(fftResult, axis=0,
    threads=self._numIFFTThreads)
\end{verbatim}

From the IFFT result, we will take the most recent frame. It should somewhat resemble the current camera frame, but should be black in areas that do not exhibit recent motion matching our parameters. We will multiply this filtered frame so that the non-black areas become bright. Then, we will upsample it (using a pyramid technique) and add the result to the current camera frame so that areas of motion are lit up. Here is the relevant code, which concludes the \texttt{_applyEulerianVideoMagnification} method:

\begin{verbatim}
# Amplify the result and overlay it on the
# original image.
overlay = numpy.real(ifftResult[-1]) * \self._amplification
i = 0
while i < self._numPyramidLevels:
    overlay = cv2.pyrUp(overlay)
    i += 1
if self._useGrayOverlay:
    overlay = cv2.cvtColor(overlay,
        cv2.COLOR_GRAY2BGR
    )
cv2.convertScaleAbs(image + overlay, image)
\end{verbatim}
To finish the implementation of the LazyEyes class, we will display the image in the same manner as we have done in our other Python apps. Here is the relevant method:

```python
def _showImage(self, image):
    if image is None:
        # Provide a black bitmap.
        bitmap = wx.EmptyBitmap(self._imageWidth,
                                  self._imageHeight)
    else:
        # Convert the image to bitmap format.
        bitmap = WXUtils.wxBitmapFromCvImage(image)
        # Show the bitmap.
        self._staticBitmap.SetBitmap(bitmap)
```

Our module's main function just instantiates and runs the app, as seen in the following code:

```python
def main():
    app = wx.App()
    lazyEyes = LazyEyes()
    lazyEyes.Show()
    app.MainLoop()

    if __name__ == '__main__':
        main()
```

That's all! Run the app and stay quite still while it builds up its history of frames. Until the history is full, the video feed will not show any special effect. At the history's default length of 360 frames, it fills in about 20 seconds on my machine. Once it is full, you should see ripples moving through the video feed in areas of recent motion—or perhaps all areas if the camera is moved or the lighting or exposure is changed. The ripples will gradually settle and disappear in areas of the scene that become still, while new ripples will appear in new areas of motion. Feel free to experiment on your own. Now, let's discuss a few recipes for configuring and testing the parameters of the LazyEyes class.

**Configuring and testing the app for various motions**

Currently, our main function initializes the LazyEyes object with the default parameters. By filling in the same parameter values explicitly, we would have this statement:

```python
lazyEyes = LazyEyes(maxHistoryLength=360,
                     minHz=5.0/6.0, maxHz=1.0,
```
amplification=32.0,
numPyramidLevels=2,
useLaplacianPyramid=True,
useGrayOverlay=True,
numFFTThreads = 4,
numIFFTThreads=4,
imageSize=(480, 360))

This recipe calls for a capture resolution of 480 x 360 and a signal processing resolution of 120 x 90 (as we are downsampling by 2 pyramid levels or a factor of 4). We are amplifying the motion only at frequencies of 0.833 Hz to 1.0 Hz, only at edges (as we are using the Laplacian pyramid), only in grayscale, and only over a history of 360 frames (about 20 to 40 seconds, depending on the frame rate). Motion is exaggerated by a factor of 32. These settings are suitable for many subtle upper-body movements such as a person's head swaying side to side, shoulders heaving while breathing, nostrils flaring, eyebrows rising and falling, and eyes scanning to and fro. For performance, the FFT and IFFT are each using 4 threads.

Here is a screenshot of the app that runs with these default parameters. Moments before taking the screenshot, I smiled like a comic theater mask and then I recomposed my normal expression. Note that my eyebrows and moustache are visible in multiple positions, including their current, low positions and their previous, high positions. For the sake of capturing the motion amplification effect in a still image, this gesture is quite exaggerated. However, in a moving video, we can see the amplification of more subtle movements too.
Here is a less extreme example where my eyebrows appear very tall because I raised and lowered them:

The parameters interact with each other in complex ways. Consider the following relationships between these parameters:

- Frame rate is greatly affected by the size of the input data for the FFT and IFFT functions. The size of the input data is determined by `maxHistoryLength` (where a shorter length provides less input and thus a faster frame rate), `numPyramidLevels` (where a greater level implies less input), `useGrayOverlay` (where `True` means less input), and `imageSize` (where a smaller size implies less input).

- Frame rate is also greatly affected by the level of multithreading of the FFT and IFFT functions, as determined by `numFFTThreads` and `numIFFTThreads` (a greater number of threads is faster up to some point).

- Frame rate is slightly affected by `useLaplacianPyramid` (`False` implies a faster frame rate), as the Laplacian algorithm requires extra steps beyond the Gaussian image.

- Frame rate determines the amount of time that `maxHistoryLength` represents.
Frame rate and maxHistoryLength determine how many repetitions of motion (if any) can be captured in the minHz to maxHz range. The number of captured repetitions, together with amplification, determines how greatly a motion or a deviation from the motion will be amplified.

The inclusion or exclusion of noise is affected by minHz and maxHz (depending on which frequencies of noise are characteristic of the camera), numPyramidLevels (where more implies a less noisy image), useLaplacianPyramid (where True is less noisy), useGrayOverlay (where True is less noisy), and imageSize (where a smaller size implies a less noisy image).

The inclusion or exclusion of motion is affected by numPyramidLevels (where fewer means the amplification is more inclusive of small motions), useLaplacianPyramid (False is more inclusive of motion in non-edge areas), useGrayOverlay (False is more inclusive of motion in areas of color contrast), minHz (where a lower value is more inclusive of slow motion), maxHz (where a higher value is more inclusive of fast motion), and imageSize (where a bigger size is more inclusive of small motions).

Subjectively, the visual effect is always best when the frame rate is high, noise is excluded, and small motions are included. Again subjectively, other conditions for including or excluding motion (edge versus non-edge, grayscale contrast versus color contrast, and fast versus slow) are application-dependent.

Let's try our hand at reconfiguring Lazy Eyes, starting with the numFFTThreads and numIFFTThreads parameters. We want to determine the numbers of threads that maximize Lazy Eyes' frame rate on your machine. The more CPU cores you have, the more threads you can gainfully use. However, experimentation is the best guide to pick a number. To get a log of the frame rate in Lazy Eyes, uncomment the following line in the _applyEulerianVideoMagnification method:

```python
print 'FPS:', 1.0 / timePerFrame
```

Run LazyEyes.py from the command line. Once the history fills up, the history's average FPS will be printed to the command line in every frame. Wait until this average FPS value stabilizes. It might take a minute for the average to adjust to the effect of the FFT and IFFT functions that begin running once the history is full. Take note of the FPS value, close the app, adjust the thread count parameters, and test again. Repeat until you feel that you have enough data to pick good numbers of threads to use on your hardware.
By activating additional CPU cores, multithreading can cause your system's temperature to rise. As you experiment, monitor your machine's temperature, fans, and CPU usage statistics. If you become concerned, reduce the number of FFT and IFFT threads. Having a suboptimal frame rate is better than overheating of your machine.

Now, experiment with other parameters to see how they affect FPS. The numPyramidLevels, useGrayOverlay, and imageSize parameters should all have a large effect. For subjectively good visual results, try to maintain a frame rate above 10 FPS. A frame rate above 15 FPS is excellent. When you are satisfied that you understand the parameters' effects on frame rate, comment out the following line again because the print statement is itself an expensive operation that can reduce frame rate:

```python
# print 'FPS:', 1.0 / timePerFrame
```

Let's try another recipe. Although our default recipe accentuates motion at edges that have high grayscale contrast, this next recipe accentuates motion in all areas (edge or non-edge) that have high contrast (color or grayscale). By considering 3 color channels instead of one grayscale channel, we are tripling the amount of data being processed by the FFT and IFFT. To offset this change, we will cut each dimension of the capture resolution to two third times its default value, thus reducing the amount of data to 2/3 * 2/3 = 4/9 times the default amount. As a net change, the FFT and IFFT process 3 * 4/9 = 4/3 times the default amount of data, a relatively small change. The following initialization statement shows our new recipe's parameters:

```python
lazyEyes = LazyEyes(useLaplacianPyramid=False,
                     useGrayOverlay=False,
                     imageSize=(320, 240))
```

Note that we are still using the default values for most parameters. If you have found non-default values that work well for numFFTThreads and numIFFTThreads on your machine, enter them as well.
Chapter 6

Here are the screenshots to show our new recipe's effect. This time, let's look at a non-extreme example first. I was typing on my laptop when this was taken. Note the haloes around my arms, which move a lot when I type, and a slight distortion and discoloration of my left cheek (viewer's left in this mirrored image). My left cheek twitches a little when I think. Apparently, it is a tic—already known to my friends and family but rediscovered by me with the help of computer vision!

If you are viewing the color version of this image in the e-book, you should see that the haloes around my arms take a green hue from my shirt and a red hue from the sofa. Similarly, the haloes on my cheek take a magenta hue from my skin and a brown hue from my hair.
Now, let's consider a more fanciful example. If we were Jedi instead of secret agents, we might wave a steel ruler in the air and pretend it was a lightsaber. While testing the theory that Lazy Eyes could make the ruler look like a real lightsaber, I took the following screenshot. The image shows two pairs of light and dark lines in two places where I was waving the lightsaber ruler. One of the pairs of lines passes through each of my shoulders. The Light Side (light line) and the Dark Side (dark line) show opposite ends of the ruler's path as I waved it. The lines are especially clear in the color version in the e-book.

Finally, the moment for which we have all been waiting is ... a recipe for amplifying a heartbeat! If you have a heart rate monitor, start by measuring your heart rate. Mine is approximately 87 bpm as I type these words and listen to inspiring ballads by the Canadian folk singer Stan Rogers. To convert bpm to Hz, divide the bpm value by 60 (the number of seconds per minute), giving \((87 / 60) \text{ Hz} = 1.45 \text{ Hz}\) in my case. The most visible effect of a heartbeat is that a person's skin changes color, becoming more red or purple when blood is pumped through an area. Thus, let's modify our second recipe, which is able to amplify color motions in non-edge areas. Choosing a frequency range centered on 1.45 Hz, we have the following initializer:

```
lazyEyes = LazyEyes(minHz=1.4, maxHz=1.5,  
useLaplacianPyramid=False,  
useGrayOverlay=False,  
imageSize=(320, 240))
```
Customize minHz and maxHz based on your own heart rate. Also, specify numFFTThreads and numIFFTThreads if non-default values work best for you on your machine.

Even amplified, a heartbeat is difficult to show in still images; it is much clearer in the live video while running the app. However, take a look at the following pair of screenshots. My skin in the left-hand side screenshot is more yellow (and lighter) while in the right-hand side screenshot it is more purple (and darker). For comparison, note that there is no change in the cream-colored curtains in the background.

Three recipes are a good start—certainly enough to fill an episode of a cooking show on TV. Go observe some other motions in your environment, try to estimate their frequencies, and then configure Lazy Eyes to amplify them. How do they look with grayscale amplification versus color amplification? Edge (Laplacian) versus area (Gaussian)? Different history lengths, pyramid levels, and amplification multipliers?

Check the book’s support page, http://www.nummist.com/opencv, for additional recipes and feel free to share your own by mailing me at josephhowse@nummist.com.
Seeing a Heartbeat with a Motion Amplifying Camera

Seeing things in another light

Although we began this chapter by presenting Eulerian video magnification as a useful technique for visible light, it is also applicable to other kinds of light or radiation. For example, a person's blood (in veins and bruises) is more visible when imaged in ultraviolet (UV) or in near infrared (NIR) than in visible light. (Skin is more transparent to UV and NIR). Thus, a UV or NIR video is likely to be a better input if we are trying to magnify a person's pulse.

Here are some examples of NIR and UV cameras that might provide useful results, though I have not tested them:

- The Pi NoIR camera (http://www.raspberrypi.org/products/pi-noir-camera/) is a consumer grade NIR camera with a MIPI interface. Here is a time lapse video showing the Pi NoIR renders outdoor scenes at https://www.youtube.com/watch?v=LLA9KHNvUK8. The camera is designed for Raspberry Pi, and on Raspbian it has V4L-compatible drivers that are directly compatible with OpenCV's VideoCapture class. (For instructions on setting up the drivers, refer to the section Setting up the Raspberry Pi Camera Module in Chapter 1, Preparing for the Mission.) Some Raspberry Pi clones might have drivers for it too. Unfortunately, Raspberry Pi is too slow to run Eulerian video magnification in real time. However, streaming the Pi NoIR input from Raspberry Pi to a desktop, via Ethernet, might allow for a real-time solution.

- The Agama V-1325R (http://www.agamazone.com/products_v1325r.html) is a consumer grade NIR camera with a USB interface. It is officially supported on Windows and Mac. Users report that it also works on Linux. It includes four NIR LEDs, which can be turned on and off via the vendor's proprietary software on Windows.

- Artray offers a series of industrial grade NIR cameras called InGaAs (http://www.artray.us/inggaas.html), as well as a series of industrial grade UV cameras (http://www.artray.us/usb2_uv.html). The cameras have USB interfaces. Windows drivers and an SDK are available from the vendor. A third-party project called OpenCV ARTRAY SDK (for more information refer to https://github.com/ieichromomma/CVMLAB/wiki/OpenCV-ARTRAY-SDK) aims to provide interoperability with at least OpenCV's C API.

Good luck and good hunting in the invisible light!
Summary

This chapter has introduced you to the relationship between computer vision and digital signal processing. We have considered a video feed as a collection of many signals—one for each channel value of each pixel—and we have understood that repetitive motions create wave patterns in some of these signals. We have used the Fast Fourier Transform and its inverse to create an alternative video stream that only sees certain frequencies of motion. Finally, we have superimposed this filtered video atop the original to amplify the selected frequencies of motion. There, we summarized Eulerian video magnification in 100 words!

Our implementation adapts Eulerian video magnification to real time by running the FFT repeatedly on a sliding window of recently captured frames rather than running it once on an entire prerecorded video. We have considered optimizations such as limiting our signal processing to grayscale, recycling large data structures rather than recreating them, and using several threads.

After experiencing the slow but suspenseful work of studying subtle motions, let's finish this book with panache! Our next and final project is an action-packed physics simulation that makes drawings bounce off the page!
Where to buy this book
You can buy OpenCV for Secret Agents from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.