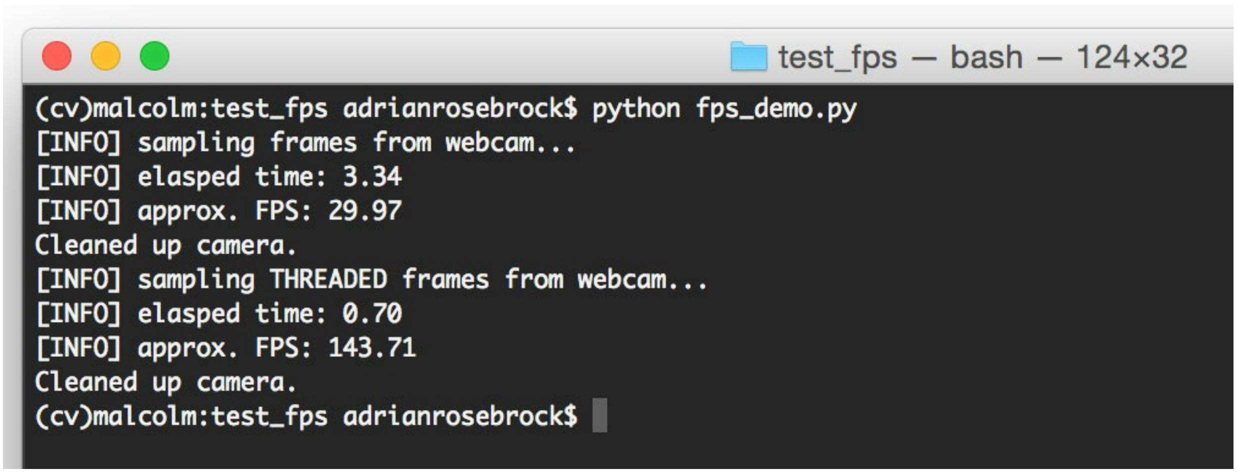
# Increasing webcam FPS with Python and OpenCV

pyimagesearch.com/2015/12/21/increasing-webcam-fps-with-python-and-opencv

Adrian Rosebrock December 21, 2015



```
(cv)malcolm:test_fps adrianrosebrock$ python fps_demo.py
[INFO] sampling frames from webcam...
[INFO] elasped time: 3.34
[INFO] approx. FPS: 29.97
Cleaned up camera.
[INFO] sampling THREADED frames from webcam...
[INFO] elasped time: 0.70
[INFO] approx. FPS: 143.71
Cleaned up camera.
(cv)malcolm:test_fps adrianrosebrock$
```

Over the next few weeks, I'll be doing a series of blog posts on how to ***improve your frames per second (FPS) from your webcam*** using Python, OpenCV, and threading.

Using threading to handle I/O-heavy tasks (such as reading frames from a camera sensor) is a programming model that has existed for decades.

For example, if we were to build a web crawler to spider a series of webpages (a task that is, by definition, I/O bound), our main program would spawn *multiple threads* to handle downloading the set of pages *in parallel* instead of relying on only a *single thread* (our "main thread") to download the pages in *sequential order*. Doing this allows us to spider the webpages substantially faster.

The same notion applies to computer vision and reading frames from a camera — **we can improve our FPS simply by creating a new thread that does nothing but poll the camera for new frames while our main thread handles processing the current frame.**

This is a simple concept, but it's one that's rarely seen in OpenCV examples since it does add a few extra lines of code (or sometimes *a lot* of lines, depending on your threading library) to the project. Multithreading can also make your program harder to debug, but once you get it right, you can dramatically improve your FPS.

We'll start off this series of posts by writing a threaded Python class to access your webcam or USB camera using OpenCV.

Next week we'll use threads to improve the FPS of your Raspberry Pi and the picamera module.

Finally, we'll conclude this series of posts by creating a class that unifies *both* the threaded webcam/USB camera code and the threaded

picamera
 code into a ***single class***, making all webcam/video processing examples on PyImageSearch *not only run faster*, *but run on either your laptop/desktop or the Raspberry Pi* ***without changing a single line of code!***

Looking for the source code to this post?

Jump Right To The Downloads Section →

## Use threading to obtain higher FPS

The "secret" to obtaining higher FPS when processing video streams with OpenCV is to move the I/O (i.e., the reading of frames from the camera sensor) to a separate thread.

You see, accessing your webcam/USB camera using the

cv2.VideoCapture
 function and the
.read()
 method is a *blocking operation*. The main thread of our Python script is completely blocked (i.e., "stalled") until the frame is read from the camera device and returned to our script.
I/O tasks, as opposed to CPU bound operations, tend to be quite slow. While computer vision and video processing applications are certainly quite CPU heavy (especially if they are intended to run in real-time), it turns out that camera I/O can be a huge bottleneck as well.

As we'll see later in this post, just by adjusting the the camera I/O process, ***we can increase our FPS by as much as 379%!***

Of course, this isn't a *true increase* of FPS as it is a *dramatic reduction in latency* (i.e., a frame is always available for processing; we don't need to poll the camera device and wait for the I/O to complete). Throughout the rest of this post, I will refer to our metrics as an "FPS increase" for brevity, but also keep in mind that it's a *combination* of both a decrease in latency and an increase in FPS.

In order to accomplish this FPS increase/latency decrease, our goal is to move the reading of frames from a webcam or USB device to an *entirely different thread, totally separate from our main Python script.*

This will allow frames to be read *continuously* from the I/O thread, all while our root thread processes the current frame. Once the root thread has finished processing its frame, it simply needs to grab the current frame from the I/O thread. This is accomplished *without* having to wait for blocking I/O operations.

The first step in implementing our threaded video stream functionality is to define a

FPS
class that we can use to measure our frames per second. This class will help us obtain quantitative evidence that threading does indeed increase FPS.
We'll then define a

WebcamVideoStream
class that will access our webcam or USB camera in a threaded fashion.
Finally, we'll define our driver script,

fps_demo.py
, that will compare single threaded FPS to multi-threaded FPS.
***Note:*** *Thanks to Ross Milligan and his blog who inspired me to do this blog post.*

I've actually already implemented webcam/USB camera and

picamera
threading inside the imutils library. However, I think a discussion of the implementation can greatly improve our knowledge of *how* and *why* threading increases FPS.
To start, if you don't already have

imutils
installed, you can install it using
pip
:
Increasing webcam FPS with Python and OpenCV

```
$ pip install imutils
```

Otherwise, you can upgrade to the latest version via:

Increasing webcam FPS with Python and OpenCV

```
$ pip install --upgrade imutils
```

As I mentioned above, the first step is to define a

FPS
class that we can use to approximate the frames per second of a given camera + computer vision processing pipeline:

```
# import the necessary packages
import datetime
class FPS:
def __init__(self):
# store the start time, end time, and total number of frames
# that were examined between the start and end intervals
self._start = None
self._end = None
self._numFrames = 0
def start(self):
# start the timer
self._start = datetime.datetime.now()
return self
def stop(self):
# stop the timer
self._end = datetime.datetime.now()
def update(self):
# increment the total number of frames examined during the
# start and end intervals
self._numFrames += 1
def elapsed(self):
# return the total number of seconds between the start and
# end interval
return (self._end - self._start).total_seconds()
def fps(self):
# compute the (approximate) frames per second
return self._numFrames / self.elapsed()
```

On **Line 5-10** we define the constructor to our

FPS
class. We don’t require any arguments, but we do initialize three important variables:

- _start
  : The starting timestamp of when we commenced measuring the frame read.
- _end
  : The ending timestamp of when we stopped measuring the frame read.
- _numFrames
  : The total number of frames that were read during the
  _start
  and
  _end
  interval.

**Lines 12-15** define the

start
method, which as the name suggests, kicks-off the timer.
Similarly, **Lines 17-19** define the

stop
method which grabs the ending timestamp.
The

update
method on **Lines 21-24** simply increments the number of frames that have been read during the starting and ending interval.
We can grab the total number of seconds that have elapsed between the starting and ending interval on **Lines 26-29** by using the

elapsed
method.
And finally, we can approximate the FPS of our camera + computer vision pipeline by using the

fps
method on **Lines 31-33**. By taking the total number of frames read during the interval and dividing by the number of elapsed seconds, we can obtain our estimated FPS.
Now that we have our

FPS
class defined (so we can empirically compare results), let’s define the
WebcamVideoStream
class which encompasses the actual threaded camera read:
Increasing webcam FPS with Python and OpenCV

```
# import the necessary packages
from threading import Thread
import cv2
class WebcamVideoStream:
def __init__(self, src=0):
# initialize the video camera stream and read the first frame
# from the stream
self.stream = cv2.VideoCapture(src)
(self.grabbed, self.frame) = self.stream.read()
# initialize the variable used to indicate if the thread should
# be stopped
self.stopped = False
```

We define the constructor to our

WebcamVideoStream
class on **Line 6**, passing in an (optional) argument: the

src
 of the stream.
If the

src
 is an *integer*, then it is presumed to be the *index* of the webcam/USB camera on your system. For example, a value of
src=0
 indicates the *first* camera and a value of
src=1
 indicates the *second* camera hooked up to your system (provided you have a second one, of course).
If

src
 is a *string*, then it assumed to be the *path* to a video file (such as .mp4 or .avi) residing on disk.
**Line 9** takes our

src
 value and makes a call to
cv2.VideoCapture
 which returns a pointer to the camera/video file.
Now that we have our

stream
 pointer, we can call the
.read()
 method to poll the stream and grab the next available frame ( **Line 10**). This is done strictly for initialization purposes so that we have an *initial* frame stored in the class.
We'll also initialize

stopped
 , a boolean indicating whether the threaded frame reading should be stopped or not.
Now, let's move on to actually utilizing threads to read frames from our video stream using OpenCV:

Increasing webcam FPS with Python and OpenCV
```
def start(self):
# start the thread to read frames from the video stream
Thread(target=self.update, args=()).start()
return self
def update(self):
# keep looping infinitely until the thread is stopped
while True:
# if the thread indicator variable is set, stop the thread
```

```
if self.stopped:
return
# otherwise, read the next frame from the stream
(self.grabbed, self.frame) = self.stream.read()
def read(self):
# return the frame most recently read
return self.frame
def stop(self):
# indicate that the thread should be stopped
self.stopped = True
```

**Lines 16-19** define our

start
method, which as the name suggests, starts the thread to read frames from our video stream. We accomplish this by constructing a
Thread
object using the
update
method as the callable object invoked by the
run()
method of the thread.
Once our driver script calls the

start
method of the
WebcamVideoStream
class, the
update
method (**Lines 21-29**) will be called.
As you can see from the code above, we start an infinite loop on **Line 23** that continuously reads the next available frame from the video

stream
via the
.read()
method (**Line 29**). If the
stopped
indicator variable is ever set, we break from the infinite loop (**Lines 25 and 26**).
Again, keep in mind that once the

start
method has been called, the
update
method is placed in a *separate thread from our main Python script* — **this separate thread is how we obtain our increased FPS performance.**

In order to access the most recently polled

frame
 from the
stream
, we'll use the
read
 method on **Lines 31-33**.
Finally, the

stop
 method (**Lines 35-37**) simply sets the
stopped
 indicator variable and signifies that the thread should be terminated.
Now that we have defined both our

FPS
 and
WebcamVideoStream
 classes, we can put all the pieces together inside
fps_demo.py
 :
Increasing webcam FPS with Python and OpenCV

```
# import the necessary packages
from __future__ import print_function
from imutils.video import WebcamVideoStream
from imutils.video import FPS
import argparse
import imutils
import cv2
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-n", "--num-frames", type=int, default=100,
help="# of frames to loop over for FPS test")
ap.add_argument("-d", "--display", type=int, default=-1,
help="Whether or not frames should be displayed")
args = vars(ap.parse_args())
```

We start off by importing our necessary packages on **Lines 2-7**. Notice how we are importing the

FPS
 and
WebcamVideoStream
 classes from the imutils library. If you do not have
imutils

installed or you need to upgrade to the latest version, please see the note at the top of this section.
**Lines 10-15** handle parsing our command line arguments. We'll require two switches here:

--num-frames
, which is the number of frames to loop over to obtain our FPS estimate, and
--display
, an indicator variable used to specify if we should use the
cv2.imshow
function to display the frames to our monitor or not.
The

--display
argument is actually really important when approximating the FPS of your video processing pipeline. Just like reading frames from a video stream is a form of I/O, *so is displaying the frame to your monitor!* We'll discuss this in more detail inside the **Threading results** section of this post.
Let's move on to the next code block which does *no threading* and uses *blocking I/O* when reading frames from the camera stream. This block of code will help us obtain a *baseline* for our FPS:

Increasing webcam FPS with Python and OpenCV

```
# grab a pointer to the video stream and initialize the FPS counter
print("[INFO] sampling frames from webcam...")
stream = cv2.VideoCapture(0)
fps = FPS().start()
# loop over some frames
while fps._numFrames < args["num_frames"]:
# grab the frame from the stream and resize it to have a maximum
# width of 400 pixels
(grabbed, frame) = stream.read()
frame = imutils.resize(frame, width=400)
# check to see if the frame should be displayed to our screen
if args["display"] > 0:
cv2.imshow("Frame", frame)
key = cv2.waitKey(1) & 0xFF
# update the FPS counter
fps.update()
# stop the timer and display FPS information
fps.stop()
print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
# do a bit of cleanup
stream.release()
```

```
cv2.destroyAllWindows()
```
**Lines 19 and 20** grab a pointer to our video stream and then start the FPS counter.

We then loop over the number of desired frames on **Line 23**, read the frame from camera (**Line 26**), update our FPS counter (**Line 35**), and optionally display the frame to our monitor (**Lines 30-32**).

After we have read

--num-frames
from the stream, we stop the FPS counter and display the elapsed time along with approximate FPS on **Lines 38-40**.
Now, let's look at our *threaded* code to read frames from our video stream:

Increasing webcam FPS with Python and OpenCV
```
# created a *threaded* video stream, allow the camera sensor to warmup,
# and start the FPS counter
print("[INFO] sampling THREADED frames from webcam...")
vs = WebcamVideoStream(src=0).start()
fps = FPS().start()
# loop over some frames...this time using the threaded stream
while fps._numFrames < args["num_frames"]:
# grab the frame from the threaded video stream and resize it
# to have a maximum width of 400 pixels
frame = vs.read()
frame = imutils.resize(frame, width=400)
# check to see if the frame should be displayed to our screen
if args["display"] > 0:
cv2.imshow("Frame", frame)
key = cv2.waitKey(1) & 0xFF
# update the FPS counter
fps.update()
# stop the timer and display FPS information
fps.stop()
print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
# do a bit of cleanup
cv2.destroyAllWindows()
vs.stop()
```
Overall, this code looks near identical to the code block above, only this time we are leveraging the

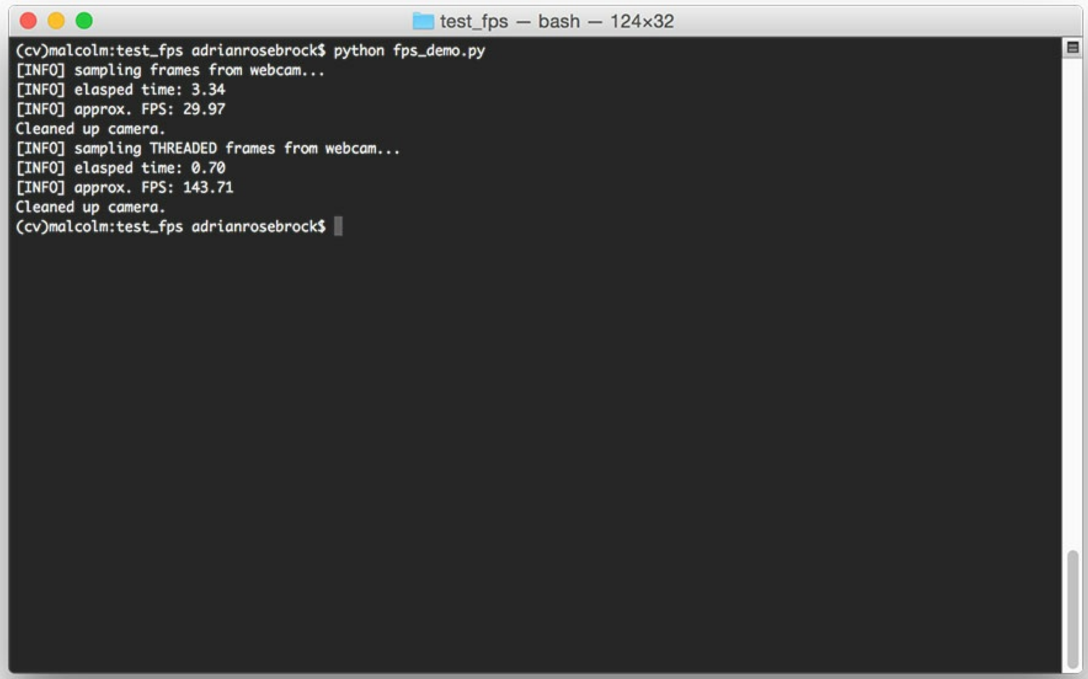
WebcamVideoStream
class.

We start the threaded stream on **Line 49**, loop over the desired number of frames on **Lines 53-65** (again, keeping track of the total number of frames read), and then display our output on **Lines 69 and 70**.

## Threading results

To see the affects of webcam I/O threading in action, just execute the following command:

Increasing webcam FPS with Python and OpenCV
$ python fps_demo.py



```
(cv)malcolm:test_fps adrianrosebrock$ python fps_demo.py
[INFO] sampling frames from webcam...
[INFO] elasped time: 3.34
[INFO] approx. FPS: 29.97
Cleaned up camera.
[INFO] sampling THREADED frames from webcam...
[INFO] elasped time: 0.70
[INFO] approx. FPS: 143.71
Cleaned up camera.
(cv)malcolm:test_fps adrianrosebrock$
```

**Figure 1:** By using threading with Python and OpenCV,***we are able to increase our FPS by over 379%!***

As we can see, by using no threading and sequentially reading frames from our video stream in the *main thread* of our Python script, we are able to obtain a respectable 29.97 FPS.

However, once we switch over to using *threaded camera I/O*, we reach 143.71 FPS — ***an increase of over 379%!***

This is clearly a *huge decrease* in our latency and a *dramatic increase* in our FPS, obtained simply by using threading.

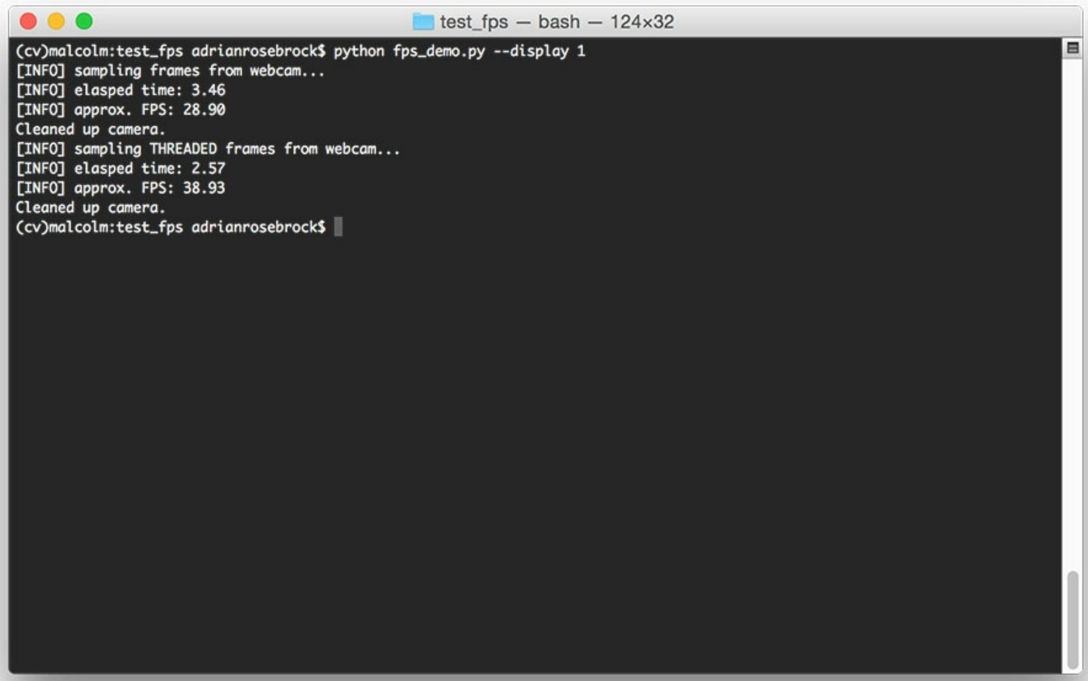
However, as we're about to find out, using the

cv2.imshow

can *substantially decrease* our FPS. This behavior makes sense if you think about it — the
cv2.show
function is just another form of I/O, only this time instead of reading a frame from a video stream, we're instead sending the frame to output on our display.
***Note:** We're also using the*

*cv2.waitKey(1)*
*function here which does add a 1ms delay to our main loop. That said, this function is necessary for keyboard interaction and to display the frame to our screen (especially once we get to the Raspberry Pi threading lessons).*
To demonstrate how the

cv2.imshow
I/O can decrease FPS, just issue this command:
Increasing webcam FPS with Python and OpenCV
$ python fps_demo.py --display 1



```
(cv)malcolm:test_fps adrianrosebrock$ python fps_demo.py --display 1
[INFO] sampling frames from webcam...
[INFO] elasped time: 3.46
[INFO] approx. FPS: 28.90
Cleaned up camera.
[INFO] sampling THREADED frames from webcam...
[INFO] elasped time: 2.57
[INFO] approx. FPS: 38.93
Cleaned up camera.
(cv)malcolm:test_fps adrianrosebrock$
```

**Figure 2:** Using the cv2.imshow function can reduce our FPS — it is another form of I/O, after all!

Using *no threading*, we reach 28.90 FPS. And *with threading* we hit 39.93 FPS. This is still a **38% increase** in FPS, but nowhere near the **379% increase** from our previous example.

Overall, I recommend using the

cv2.imshow

function to help debug your program — but if your final production code doesn't need it, *there is no reason to include it* since you'll be hurting your FPS.
A great example of such a program would be developing a home surveillance motion detector that sends you a txt message containing a photo of the person who just walked in the front door of your home. Realistically, you do not need the

cv2.imshow
function for this. By removing it, you can increase the performance of your motion detector and allow it to process more frames faster.

## Summary

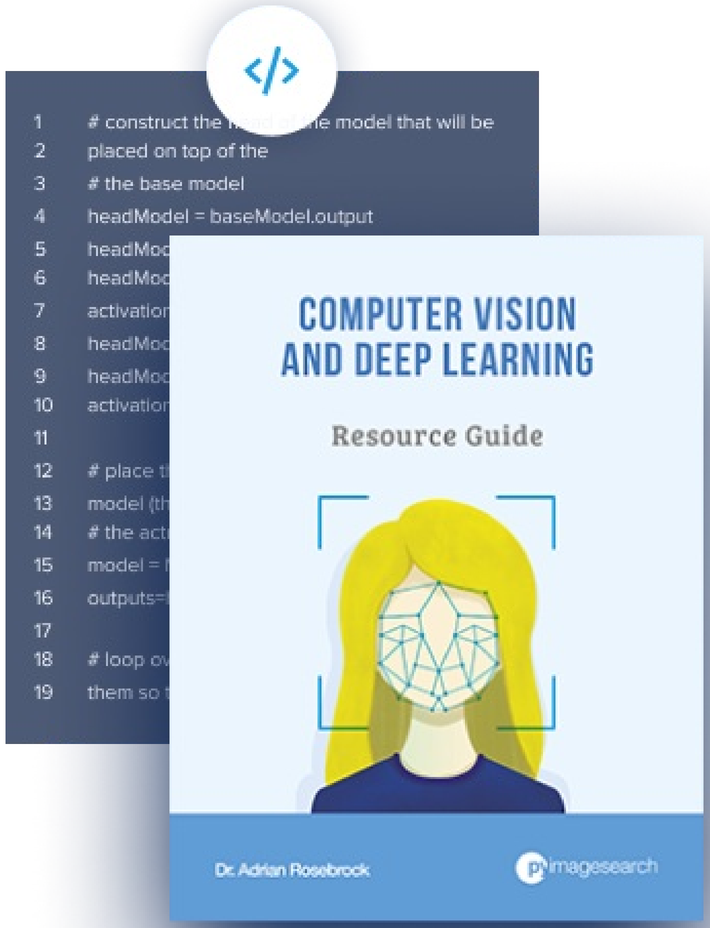
In this blog post we learned how threading can be used to increase your webcam and USB camera FPS using Python and OpenCV.

As the examples in this post demonstrated, we were able to obtain a **379% increase in FPS** simply by using threading. While this isn't necessarily a fair comparison (since we could be processing the same frame multiple times), it does demonstrate the importance of *reducing latency* and always having a frame ready for processing.

**In nearly all situations, using threaded access to your webcam can substantially improve your video processing pipeline.**

Next week we'll learn how to increase the FPS of our Raspberry Pi using the picamera module.

***Be sure to enter your email address in the form below to be notified when the next post goes live!***



## Download the Source Code and FREE 17-page Resource Guide

Enter your email address below to get a .zip of the code and a **FREE 17-page Resource Guide on Computer Vision, OpenCV, and Deep Learning.** Inside you'll find my hand-picked tutorials, books, courses, and libraries to help you master CV and DL!