Community Experience Distilled

OpenCV 3 Blueprints

Expand your knowledge of computer vision by building amazing projects with OpenCV 3

Joseph Howse Steven Puttemans
Quan Hua Utkarsh Sinha

[PACKT] PUBLISHING open source* community experience distilled

OpenCV 3 Blueprints

Expand your knowledge of computer vision by building amazing projects with OpenCV 3

Joseph Howse

Steven Puttemans

Quan Hua

Utkarsh Sinha

[PACKT] open source
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

OpenCV 3 Blueprints

First published: November 2015

Production reference: 1281015

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-975-7

`www.packtpub.com`

Credits

Authors
Joseph Howse
Steven Puttemans
Quan Hua
Utkarsh Sinha

Reviewers
Demetris Gerogiannis
Li Jing
Walter Lucetti
Luca Del Tongo
Theodore Tsesmelis

Commissioning Editor
Julian Ursell

Acquisition Editors
Harsha Bharwani
Divya Poojari

Content Development Editor
Merwyn D'souza

Technical Editor
Pramod Kumavat

Copy Editor
Janbal Dharmaraj

Project Coordinator
Neha Bhatnagar

Proofreader
Safis Editing

Indexer
Priya Sane

Graphics
Abhinash Sahu

Production Coordinator
Komal Ramchandani

Cover Work
Komal Ramchandani

About the Authors

Joseph Howse lives in Canada. During the cold winters, he grows a beard and his four cats grow thick coats of fur. He combs the cats every day. Sometimes, the cats pull his beard.

Joseph has been writing for Packt Publishing since 2012. His books include *OpenCV for Secret Agents*, *OpenCV 3 Blueprints*, *Android Application Programming with OpenCV 3*, *Learning OpenCV 3 Computer Vision with Python*, and *Python Game Programming by Example*.

When he is not writing books or grooming cats, Joseph provides consulting, training, and software development services through his company, Nummist Media (`http://nummist.com`).

I dedicate my work to Sam, Jan, Bob, Bunny, and the cats, who have been my lifelong guides and companions.

To my coauthors, I extend my sincere thanks for the opportunity to research and write *OpenCV 3 Blueprints* together. Authoring a book is a lot of work, and your dedication has made this project possible!

I am also indebted to the many editors and technical reviewers who have contributed to planning, polishing, and marketing this book. These people have guided me with their experience and have saved me from sundry errors and omissions. From the project's beginnings, Harsha Bharwani and Merwyn D'souza have been instrumental in assembling and managing the team of authors, editors, and reviewers. Harsha has advised me through multiple book projects, and I am grateful for her continued support.

Finally, I want to thank my readers and everybody else at Packt and in the OpenCV community. We have done so much together, and our journey continues!

Steven Puttemans is a PhD research candidate at the KU Leuven, Department of Industrial Engineering Sciences. At this university, he is working for the EAVISE research group, which focuses on combining computer vision and artificial intelligence. He obtained a master of science degree in Electronics-ICT and further expanded his interest in computer vision by obtaining a master of science in artificial intelligence.

As an enthusiastic researcher, his goal is to combine state-of-the-art computer vision algorithms with real-life industrial problems to provide robust and complete object detection solutions for the industry. His previous projects include TOBCAT, an open source object detection framework for industrial object detection problems, and a variety of smaller computer vision-based industrial projects. During these projects, Steven worked closely with the industry.

Steven is also an active participant in the OpenCV community. He is a moderator of the OpenCV Q&A forum, and has submitted or reviewed many bug fixes and improvements for OpenCV 3. He also focuses on putting as much of his research as possible back into the framework to further support the open source spirit.

More info about Steven's research, projects, and interests can be found at `https://stevenputtemans.github.io`.

Firstly, I would like to thank my girlfriend and family for supporting me through the long journey of writing my first computer vision-based book. They stood by me at every step of the way. Secondly, I would like to thank the OpenCV Q&A community who have been supporting me with all the technical problems that arose while writing this book.

Quan Hua is a software engineer at Autonomous, a start-up company in robotics, where he focuses on developing computer vision and machine learning applications for personal robots. He earned a bachelor of science degree from the University of Science, Vietnam, specializing in computer vision, and published a research paper in CISIM 2014. As the owner of `Quan404.com`, he also blogs about various computer vision techniques to share his experience with the community.

I wish to thank everyone who has encouraged me on the way while writing this book.

I want to express my sincere gratitude to my coauthors, editors, and reviewers for their advice and assistance. Thank you, Neha Bhatnagar, Harsha Bharwani, and Merwyn D'souza for giving me a golden opportunity to join this project.

I would like to thank the members of my family and my girlfriend, Kim Ngoc, who supported and encouraged me in spite of all the time it took me away from them. They all kept me going, and this book would not have been possible without them.

I would also like to thank my teachers who gave me the knowledge in computer vision and machine learning.

Utkarsh Sinha lives in Pittsburgh, Pennsylvania, where he is pursuing a master's in computer vision at Carnegie Mellon University. He intends to learn the state of the art of computer vision at the university and work on real-life industrial scale computer vision challenges.

Before joining CMU, he worked as a Technical Director at Dreamworks Animation on movies such as *Home*, *How to Train your Dragon 2*, and *Madagascar 3*. His work spans multiple movies and multiple generations of graphics technology.

He earned his bachelor of engineering degree in computer science and his master of science degree in mathematics from BITS-Pilani, Goa. He has been working in the field of computer vision for about 6 years as a consultant and as a software engineer at start-ups.

He blogs at `http://utkarshsinha.com/` about various topics in technology—most of which revolve around computer vision. He also publishes computer vision tutorials on the Internet through his website, AI Shack (`http://aishack.in/`). His articles help people understand concepts in computer vision every day.

I would like to thank my parents for supporting me through the long journey of writing my first book on computer vision. Without them, this book wouldn't have been possible.

I am indebted to the technical reviewers and editors who have helped polish the book to be what it is now. Their efforts have really brought the book together—thinking of aspects such as planning and marketing. Harsha Bharwani and Merwyn D'souza have been linchpins in this project—managing the team of authors, editors, and reviewers. I am grateful to them for providing me with such an excellent opportunity.

My friends, teachers, and the computer vision community have helped me understand computer vision and have supported me by answering questions, and having good debates and discussions. This has been vital, and I would like to acknowledge and thank everyone for the efforts they put in.

I would also like to express my gratitude to the Packt community and OpenCV users. Computer vision is a very exciting place to be right now—and we're just getting started!

About the Reviewers

Walter Lucetti, known on internet as Myzhar, is an Italian computer engineer with a specialization in Robotics and Robotics Perception. He received the *laurea* degree in 2005 studying at Research Center E. Piaggio in Pisa (Italy), with a thesis on 3D mapping of the real world using a 2D laser tilted using a servo motor, fusing 3D with RGB data. During the writing of the thesis, he was introduced to OpenCV for the first time; it was early 2004 and OpenCV was at its larval stage.

After the *laurea*, he started working as software developer for a low-level embedded system and high-level desktop system. He deeply improved his knowledge about computer vision and machine learning as a researcher, for a little lapse of time, at Gustavo Stefanini Advanced Robotics Center in La Spezia (Italy), a spinoff of PERCRO Laboratory of the Scuola Superiore Sant'Anna of Pisa (Italy).

Now, he works in the industry, writing firmware for embedded ARM systems and software for desktop systems based on Qt framework and intelligent algorithms for video surveillance systems based on OpenCV and CUDA.

He is also working on a personal robotics project, MyzharBot. MyzharBot is a tracked ground mobile robot that uses computer vision to detect obstacles and analyze and explore the environment. The robot is guided by algorithms based on ROS, CUDA, and OpenCV. You can follow the project on its website: `http://myzharbot.robot-home.it`.

He has reviewed several books on OpenCV with Packt Publishing, including *OpenCV Computer Vision Application Programming Cookbook*, *Second Edition*.

Luca Del Tongo is a computer engineer with a strong passion for algorithms, computer vision, and image processing techniques. He's the coauthor of a free eBook called *Data Structures and Algorithms (DSA)* with over 100k downloads to date and has published several image processing tutorials on his YouTube channel using Emgu CV. While working on his master's thesis, he developed an image forensic algorithm published in a scientific paper called *Copy Move forgery detection and localization by means of robust clustering with J-Linkage*. Currently, Luca works as a software engineer in the field of ophthalmology, developing corneal topography, processing algorithms, IOL calculation, and computerized chart projectors. He loves to practice sport and follow MOOC courses in his spare time. You can contact him through his blog at `http://blogs.ugidotnet.org/wetblog`.

Theodore Tsesmelis is an engineer working in the fields of computer vision and image processing. He holds a master of science degree with specialization in computer vision and image processing from Aalborg University in the study programme of Vision Graphics and Interactive Systems (VGIS).

His main interests lie in everything that deals with computer science and especially with computer vision and image processing. In his free time, he likes to contribute to his favorite OpenCV library as well as to consult and help others to get familiar with it through the official OpenCV forum.

With the chance that this part of the book provides me, I would like to thank Steven Puttemans and Neha Bhatnagar for the nice collaboration and opportunity that they provided me to participate in this project.

Table of Contents

Preface

Open source computer vision projects, such as OpenCV 3, enable all kinds of users to harness the forces of machine vision, machine learning, and artificial intelligence. By mastering these powerful libraries of code and knowledge, professionals and hobbyists can create smarter, better applications wherever they are needed.

This is exactly where this book is focused, guiding you through a set of hands-on projects and templates, which will teach you to combine fantastic techniques in order to solve your specific problem.

As we study computer vision, let's take inspiration from these words:

"I saw that wisdom is better than folly, just as light is better than darkness."

– Ecclesiastes, 2:13

Let's build applications that see clearly, and create knowledge.

What this book covers

Chapter 1, *Getting the Most out of Your Camera System*, discusses how to select and configure camera systems to see invisible light, fast motion, and distant objects.

Chapter 2, *Photographing Nature and Wildlife with an Automated Camera*, shows how to build a "camera trap", as used by nature photographers, and process photos to create beautiful effects.

Chapter 3, *Recognizing Facial Expressions with Machine Learning*, explores ways to build a facial expression recognition system with various feature extraction techniques and machine learning methods.

Chapter 4, *Panoramic Image Stitching Application Using Android Studio and NDK*, focuses on the project of building a panoramic camera app for Android with the help of OpenCV 3's stitching module. We will use C++ with Android NDK.

Chapter 5, *Generic Object Detection for Industrial Applications*, investigates ways to optimize your object detection model, make it rotation invariant, and apply scene-specific constraints to make it faster and more robust.

Chapter 6, *Efficient Person Identification Using Biometric Properties*, is about building a person identification and registration system based on biometric properties of that person, such as their fingerprint, iris, and face.

Chapter 7, *Gyroscopic Video Stabilization*, demonstrates techniques for fusing data from videos and gyroscopes, how to stabilize videos shot on your mobile phone, and how to create hyperlapse videos.

What you need for this book

As a basic setup, the complete book is based on the OpenCV 3 software. If a chapter does not have a specific OS requirement, then it will run on Windows, Linux, and Mac. As authors, we encourage you to take the latest master branch from the official GitHub repository (`https://github.com/Itseez/opencv/`) for setting up your OpenCV installation, rather then using the downloadable packages at the official OpenCV website (`http://opencv.org/downloads.html`), since the latest master branch contains a huge number of fixes compared to the latest stable release.

For hardware, the authors expect that you have a basic computer system setup, either a desktop or a laptop, with at least 4 GB of RAM memory available. Other hardware requirements are mentioned below.

The following chapters have specific requirements that come on top of the OpenCV 3 installation:

Chapter 1, *Getting the Most out of Your Camera System*:

- **Software**: OpenNI2 and FlyCapture 2.
- **Hardware**: PS3 Eye camera or any other USB webcam, an Asus Xtion PRO live or any other OpenNI-compatible depth camera, and a Point Grey Research (PGR) camera with one or more lenses.
- **Remarks**: The PGR camera setup (with FlyCapture 2) will not run on Mac. Even if you do not have all the required hardware, you can still benefit from some sections of this chapter.

Chapter 2, Photographing Nature and Wildlife with an Automated Camera:

- **Software**: Linux or Mac operating system.
- **Hardware**: A portable laptop or a single-board computer (SBC) with battery, combined with a photo camera.

Chapter 4, Panoramic Image Stitching Application Using Android Studio and NDK:

- **Software**: Android 4.4 or later, Android NDK.
- **Hardware**: Any mobile device that supports Android 4.4 or later.

Chapter 7, Gyroscopic Video Stabilization:

- **Software**: NumPy, SciPy, Python, and Android 5.0 or later, and the Android NDK.
- **Hardware**: A mobile phone that supports Android 5.0 or later for capturing video and gyroscope signals.

Basic installation guides

As authors, we acknowledge that installing OpenCV 3 on your system can sometimes be quite cumbersome. Therefore, we have added a series of basic installation guides for installing OpenCV 3, based on the latest OpenCV 3 master branch on your system, and getting the necessary modules for the different chapters to work. For more information, take a look at `https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/installation_tutorials`.

Keep in mind that the book also uses modules from the OpenCV "contrib" (contributed) repository. The installation manual will have directions on how to install these. However, we encourage you to only install those modules that we need, because we know that they are stable. For other modules, this might not be the case.

Who this book is for

This book is ideal for you if you aspire to build computer vision systems that are smarter, faster, more complex, and more practical than the competition. This is an advanced book, intended for those who already have some experience in setting up an OpenCV development environment and building applications with OpenCV. You should be comfortable with computer vision concepts, object-oriented programming, graphics programming, IDEs, and the command line.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You can find the OpenCV software by going to `http://opencv.org` and clicking on the download link."

A block of code is set as follows:

```
Mat input = imread("/data/image.png", LOAD_IMAGE_GRAYSCALE);
GaussianBlur(input, input, Size(7,7), 0, 0);
imshow("image", input);
waitKey(0);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Mat input = imread("/data/image.png", LOAD_IMAGE_GRAYSCALE);
GaussianBlur(input, input, Size(7,7), 0, 0);
imshow("image", input);
waitKey(0);
```

Any command-line input or output is written as follows:

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

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 code is also maintained on a GitHub repository by the authors of this book. This code repository can be found at `https://github.com/OpenCVBlueprints/OpenCVBlueprints`.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/B04028_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Since this book also has a GitHub repository assigned to it, you can also report content errata by creating an issue at the following page: `https://github.com/OpenCVBlueprints/OpenCVBlueprints/issues`.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem. Or as mentioned before, you could open up an issue on the GitHub repository and one of the authors will help you as soon as possible.

1
Getting the Most out of Your Camera System

Claude Monet, one of the founders of French Impressionist painting, taught his students to paint only what they *saw*, not what they *knew*. He even went as far as to say:

> "*I wish I had been born blind and then suddenly gained my sight so that I could begin to paint without knowing what the objects were that I could see before me.*"

Monet rejected traditional artistic subjects, which tended to be mystical, heroic, militaristic, or revolutionary. Instead, he relied on his own observations of middle-class life: of social excursions; of sunny gardens, lily ponds, rivers, and the seaside; of foggy boulevards and train stations; and of private loss. With deep sadness, he told his friend, Georges Clemenceau (the future President of France):

> "*I one day found myself looking at my beloved wife's dead face and just systematically noting the colors according to an automatic reflex!*"

Monet painted everything according to his personal impressions. Late in life, he even painted the symptoms of his own deteriorating eyesight. He adopted a reddish palette while he suffered from cataracts and a brilliant bluish palette after cataract surgery left his eyes more sensitive, possibly to the near ultraviolet range.

Like Monet's students, we as scholars of computer vision must confront a distinction between *seeing* and *knowing* and likewise between input and processing. Light, a lens, a camera, and a digital imaging pipeline can grant a computer a sense of *sight*. From the resulting image data, **machine-learning (ML)** algorithms can extract *knowledge* or at least a set of meta-senses such as detection, recognition, and reconstruction (scanning). Without proper senses or data, a system's learning potential is limited, perhaps even nil. Thus, when designing any computer vision system, we must consider the foundational requirements in terms of lighting conditions, lenses, cameras, and imaging pipelines.

What do we require in order to clearly see a given subject? This is the central question of our first chapter. Along the way, we will address five subquestions:

- What do we require to see fast motion or fast changes in light?
- What do we require to see distant objects?
- What do we require to see with depth perception?
- What do we require to see in the dark?
- How do we obtain good value-for-money when purchasing lenses and cameras?

For many practical applications of computer vision, the environment is not a well-lit, white room, and the subject is not a human face at a distance of 0.6m (2')!

The choice of hardware is crucial to these problems. Different cameras and lenses are optimized for different imaging scenarios. However, software can also make or break a solution. On the software side, we will focus on the efficient use of OpenCV. Fortunately, OpenCV's **videoio** module supports many classes of camera systems, including the following:

- Webcams in Windows, Mac, and Linux via the following frameworks, which come standard with most versions of the operating system:
    - **Windows**: Microsoft Media Foundation (MSMF), DirectShow, or Video for Windows (VfW)
    - **Mac**: QuickTime
    - **Linux**: Video4Linux (V4L), Video4Linux2 (V4L2), or libv4l
- Built-in cameras in iOS and Android devices
- OpenNI-compliant depth cameras via OpenNI or OpenNI2, which are open-source under the Apache license

- Other depth cameras via the proprietary Intel Perceptual Computing SDK
- Photo cameras via libgphoto2, which is open source under the GPL license. For a list of libgphoto2's supported cameras, see `http://gphoto.org/proj/libgphoto2/support.php`.

[Note that the GPL license is not appropriate for use in closed source software.]

- IIDC/DCAM-compliant industrial cameras via libdc1394, which is open-source under the LGPLv2 license
- For Linux, unicap can be used as an alternative interface for IIDC/DCAM-compliant cameras, but unicap is GPL-licensed and thus not appropriate for use in closed-source software.
- Other industrial cameras via the following proprietary frameworks:
    - Allied Vision Technologies (AVT) PvAPI for GigE Vision cameras
    - Smartek Vision Giganetix SDK for GigE Vision cameras
    - XIMEA API

[The videoio module is new in OpenCV 3. Previously, in OpenCV 2, video capture and recording were part of the highgui module, but in OpenCV 3, the highgui module is only responsible for GUI functionality. For a complete index of OpenCV's modules, see the official documentation at `http://docs.opencv.org/3.0.0/`.]

However, we are not limited to the features of the videoio module; we can use other APIs to configure cameras and capture images. If an API can capture an array of image data, OpenCV can readily use the data, often without any copy operation or conversion. As an example, we will capture and use images from depth cameras via OpenNI2 (without the videoio module) and from industrial cameras via the FlyCapture SDK by Point Grey Research (PGR).

An industrial camera or **machine vision camera** typically has interchangeable lenses, a high-speed hardware interface (such as FireWire, Gigabit Ethernet, USB 3.0, or Camera Link), and a complete programming interface for all camera settings.

Most industrial cameras have SDKs for Windows and Linux. PGR's FlyCapture SDK supports single-camera and multi-camera setups on Windows as well as single-camera setups on Linux. Some of PGR's competitors, such as Allied Vision Technologies (AVT), offer better support for multi-camera setups on Linux.

We will learn about the differences among categories of cameras, and we will test the capabilities of several specific lenses, cameras, and configurations. By the end of the chapter, you will be better qualified to design either consumer-grade or industrial-grade vision systems for yourself, your lab, your company, or your clients. I hope to surprise you with the results that are possible at each price point!

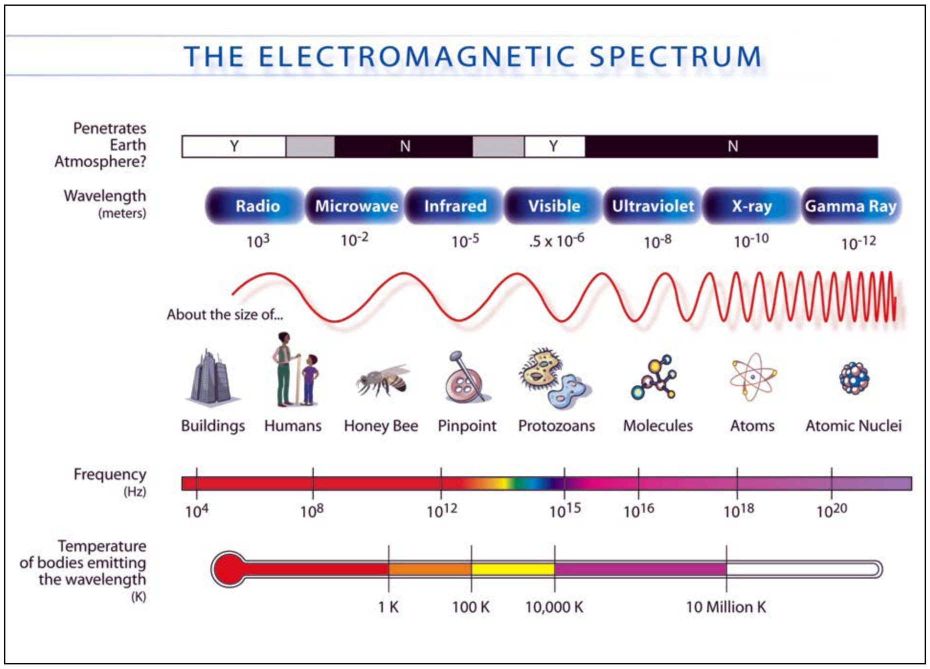
Coloring the light

The human eye is sensitive to certain wavelengths of electromagnetic radiation. We call these wavelengths "visible light", "colors", or sometimes just "light". However, our definition of "visible light" is anthropocentric as different animals see different wavelengths. For example, bees are blind to red light, but can see ultraviolet light (which is invisible to humans). Moreover, machines can assemble human-viewable images based on almost any stimulus, such as light, radiation, sound, or magnetism. To broaden our horizons, let's consider eight kinds of electromagnetic radiation and their common sources. Here is the list, in order of decreasing wavelength:

- **Radio waves** radiate from certain astronomical objects and from lightning. They are also generated by wireless electronics (radio, Wi-Fi, Bluetooth, and so on).
- **Microwaves** radiated from the Big Bang and are present throughout the Universe as background radiation. They are also generated by microwave ovens.
- **Far infrared (FIR) light** is an invisible glow from warm or hot things such as warm-blooded animals and hot-water pipes.
- **Near infrared (NIR) light** radiates brightly from our sun, from flames, and from metal that is red-hot or nearly red-hot. However, it is a relatively weak component in commonplace electric lighting. Leaves and other vegetation brightly reflect NIR light. Skin and certain fabrics are slightly transparent to NIR.

- **Visible light** radiates brightly from our sun and from commonplace electric light sources. Visible light includes the colors red, orange, yellow, green, blue, and violet (in order of decreasing wavelength).
- **Ultraviolet (UV) light**, too, is abundant in sunlight. On a sunny day, UV light can burn our skin and can become slightly visible to us as a blue-gray haze in the distance. Commonplace, silicate glass is nearly opaque to UV light, so we do not suffer sunburn when we are behind windows (indoors or in a car). For the same reason, UV camera systems rely on lenses made of non-silicate materials such as quartz. Many flowers have UV markings that are visible to insects. Certain bodily fluids such as blood and urine are more opaque to UV than to visible light.
- **X-rays** radiate from certain astronomical objects such as black holes. On Earth, radon gas, and certain other radioactive elements are natural X-ray sources.
- **Gamma rays** radiate from nuclear explosions, including supernovae. To lesser extents the sources of gamma rays also include radioactive decay and lightning strikes.

NASA provides the following visualization of the wavelength and temperature associated with each kind of light or radiation:



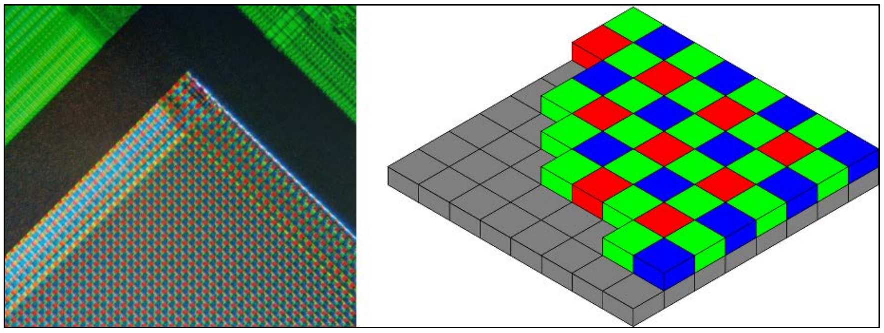
Passive imaging systems rely on **ambient** (commonplace) light or radiation sources as described in the preceding list. **Active** imaging systems include sources of their own so that the light or radiation is **structured** in more predictable ways. For example, an active night vision scope might use a NIR camera plus a NIR light.

For astronomy, passive imaging is feasible across the entire electromagnetic spectrum; the vast expanse of the Universe is flooded with all kinds of light and radiation from sources old and new. However, for terrestrial (Earth-bound) purposes, passive imaging is mostly limited to the FIR, NIR, visible, and UV ranges. Active imaging is feasible across the entire spectrum, but the practicalities of power consumption, safety, and interference (between our use case and others) limit the extent to which we can flood an environment with excess light and radiation.

Whether active or passive, an imaging system typically uses a lens to bring light or radiation into focus on the surface of the camera's sensor. The lens and its coatings transmit some wavelengths while blocking others. Additional filters may be placed in front of the lens or sensor to block more wavelengths. Finally, the sensor itself exhibits a varying **spectral response**, meaning that for some wavelengths, the sensor registers a strong (bright) signal, but for other wavelengths, it registers a weak (dim) signal or no signal. Typically, a mass-produced digital sensor responds most strongly to green, followed by red, blue, and NIR. Depending on the use case, such a sensor might be deployed with a filter to block a range of light (whether NIR or visible) and/or a filter to superimpose a pattern of varying colors. The latter filter allows for the capture of multichannel images, such as RGB images, whereas the unfiltered sensor would capture **monochrome** (gray) images.

The sensor's surface consists of many sensitive points or **photosites**. These are analogous to pixels in the captured digital image. However, photosites and pixels do not necessarily correspond one-to-one. Depending on the camera system's design and configuration, the signals from several photosites might be blended together to create a neighborhood of multichannel pixels, a brighter pixel, or a less noisy pixel.

Consider the following pair of images. They show a sensor with a Bayer filter, which is a common type of color filter with two green photosites per red or blue photosite. To compute a single RGB pixel, multiple photosite values are blended. The left-hand image is a photo of the filtered sensor under a microscope, while the right-hand image is a cut-away diagram showing the filter and underlying photosites:



The preceding images come from Wikimedia. They are contributed by the users Natural Philo, under the Creative Commons Attribution-Share Alike 3.0 Unported license (left), and Cburnett, under the GNU Free Documentation License (right).

As we see in this example, a simplistic model (an RGB pixel) might hide important details about the way data are captured and stored. To build efficient image pipelines, we need to think about not just pixels, but also channels and **macropixels**—neighborhoods of pixels that share some channels of data and are captured, stored, and processed in one block. Let's consider three categories of image formats:

- A **raw image** is a literal representation of the photosites' signals, scaled to some range such as 8, 12, or 16 bits. For photosites in a given row of the sensor, the data are contiguous but for photosites in a given column, they are not.
- A **packed image** stores each pixel or macropixel contiguously in memory. That is to say, data are ordered according to their neighborhood. This is an efficient format if most of our processing pertains to multiple color components at a time. For a typical color camera, a raw image is *not* packed because each neighborhood's data are split across multiple rows. Packed color images usually use RGB channels, but alternatively, they may use **YUV channels**, where Y is brightness (grayscale), U is blueness (versus greenness), and V is redness (also versus greenness).

- A **planar image** stores each channel contiguously in memory. That is to say, data are ordered according to the color component they represent. This is an efficient format if most of our processing pertains to a single color component at a time. Packed color images usually use YUV channels. Having a Y channel in a planar format is efficient for computer vision because many algorithms are designed to work on grayscale data alone.

An image from a monochrome camera can be efficiently stored and processed in its raw format or (if it must integrate seamlessly into a color imaging pipeline) as the Y plane in a planar YUV format. Later in this chapter, in the sections *Supercharging the PlayStation Eye* and *Supercharging the GS3-U3-23S6M-C and other Point Grey Research cameras*, we will discuss code samples that demonstrate efficient handling of various image formats.

Until now, we have covered a brief taxonomy of light, radiation, and color—their sources, their interaction with optics and sensors, and their representation as channels and neighborhoods. Now, let's explore some more dimensions of image capture: time and space.

Capturing the subject in the moment

Robert Capa, a photojournalist who covered five wars and shot images of the first wave of D-Day landings at Omaha Beach, gave this advice:

> "*If your pictures aren't good enough, you're not close enough.*"

Like a computer vision program, a photographer is the intelligence behind the lens. (Some would say the photographer is the soul behind the lens.) A good photographer continuously performs detection and tracking tasks—scanning the environment, choosing the subject, predicting actions and expressions that will create the right moment for the photo, and choosing the lens, settings, and viewpoint that will most effectively frame the subject.

By getting "close enough" to the subject and the action, the photographer can observe details quickly with the naked eye and can move to other viewpoints quickly because the distances are short and because the equipment is typically light (compared to a long lens on a tripod for a distant shot). Moreover, a close-up, wide-angle shot pulls the photographer, and viewer, into a first-person perspective of events, as if we become the subject or the subject's comrade for a single moment.

Photographic aesthetics concern us further in *Chapter 2, Photographing Nature and Wildlife with an Automated Camera*. For now, let's just establish two cardinal rules: don't miss the subject and don't miss the moment! Poor visibility and unfortunate timing are the worst excuses a photographer or a practitioner of computer vision can give. To hold ourselves to account, let us define some measurements that are relevant to these cardinal rules.

Resolution is the finest level of detail that the lens and camera can see. For many computer vision applications, recognizable details are the subject of the work, and if the system's resolution is poor, we might miss this subject completely. Resolution is often expressed in terms of the sensor's photosite counts or the captured image's pixel counts, but at best these measurements only tell us about one limiting factor. A better, empirical measurement, which reflects all characteristics of the lens, sensor, and setup, is called **line pairs per millimeter** (lp/mm). This means the maximum density of black-on-white lines that the lens and camera can resolve, in a given setup. At any higher density than this, the lines in the captured image blur together. Note that lp/mm varies with the subject's distance and the lens's settings, including the focal length (optical zoom) of a zoom lens. When you approach the subject, zoom in, or swap out a short lens for a long lens, the system should of course capture more detail! However, lp/mm does not vary when you crop (digitally zoom) a captured image.

Lighting conditions and the camera's **ISO speed** setting also have an effect on lp/mm. High ISO speeds are used in low light and they boost both the signal (which is weak in low light) and the noise (which is as strong as ever). Thus, at high ISO speeds, some details are blotted out by the boosted noise.

To achieve anything near its potential resolution, the lens must be properly focused. Dante Stella, a contemporary photographer, describes a problem with modern camera technology:

> "*For starters, it lacks … thought-controlled predictive autofocus.*"

That is to say, autofocus can fail miserably when its algorithm is mismatched to a particular, intelligent use or a particular pattern of evolving conditions in the scene. Long lenses are especially unforgiving with respect to improper focus. The **depth of field** (the distance between the nearest and farthest points in focus) is shallower in longer lenses. For some computer vision setups—for example, a camera hanging over an assembly line—the distance to the subject is highly predictable and in such cases manual focus is an acceptable solution.

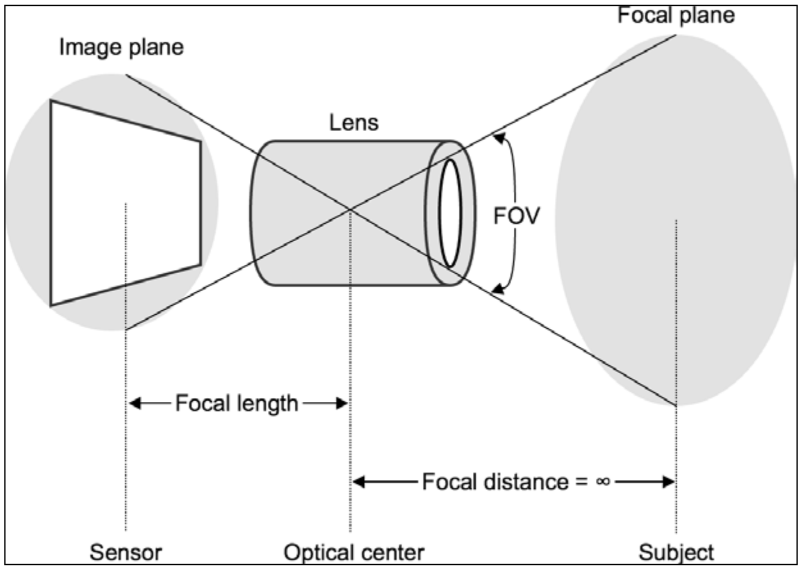
Field of view (FOV) is the extent of the lens's vision. Typically, FOV is measured as an angle, but it can be measured as the distance between two peripherally observable points at a given depth from the lens. For example, a FOV of 90 degrees may also be expressed as a FOV of 2m at a depth of 1m or a FOV of 4m at a depth of 2m. Where not otherwise specified, FOV usually means diagonal FOV (the diagonal of the lens's vision), as opposed to horizontal FOV or vertical FOV. A longer lens has a narrower FOV. Typically, a longer lens also has higher resolution and less distortion. If our subject falls outside the FOV, we miss the subject completely! Toward the edges of the FOV, resolution tends to decrease and distortion tends to increase, so preferably the FOV should be wide enough to leave a margin around the subject.

The camera's **throughput** is the rate at which it captures image data. For many computer vision applications, a visual event might start and end in a fleeting moment and if the throughput is low, we might miss the moment completely or our image of it might suffer from motion blur. Typically, throughput is measured in frames per second (FPS), though measuring it as a bitrate can be useful, too. Throughput is limited by the following factors:

- **Shutter speed** (exposure time): For a well-exposed image, the shutter speed is limited by lighting conditions, the lens's **aperture setting**, and the camera's ISO speed setting. (Conversely, a slower shutter speed allows for a narrower aperture setting or slower ISO speed.) We will discuss aperture settings after this list.
- **The type of shutter**: A **global shutter** synchronizes the capture across all photosites. A **rolling shutter** does not; rather, the capture is sequential such that photosites at the bottom of the sensor register their signals later than photosites at the top. A rolling shutter is inferior because it can make an object appear skewed when the object or the camera moves rapidly. (This is sometimes called the "Jell-O effect" because of the video's resemblance to a wobbling mound of gelatin.) Also, under rapidly flickering lighting, a rolling shutter creates light and dark bands in the image. If the start of the capture is synchronized but the end is not, the shutter is referred to as a **rolling shutter with global reset**.
- **The camera's onboard image processing routines**, such as conversion of raw photosite signals to a given number of pixels in a given format. As the number of pixels and bytes per pixel increase, the throughput decreases.
- **The interface between the camera and host computer**: Common camera interfaces, in order of decreasing bit rates, include CoaXPress full, Camera Link full, USB 3.0, CoaXPress base, Camera Link base, Gigabit Ethernet, IEEE 1394b (FireWire full), USB 2.0, and IEEE 1394 (FireWire base).

A wide aperture setting lets in more light to allow for a faster exposure, a lower ISO speed, or a brighter image. However, a narrower aperture has the advantage of offering a greater depth of field. A lens supports a limited range of aperture settings. Depending on the lens, some aperture settings exhibit higher resolution than others. Long lenses tend to exhibit more stable resolution across aperture settings.

A lens's aperture size is expressed as an **f-number** or **f-stop**, which is the ratio of the lens's focal length to the diameter of its aperture. Roughly speaking, **focal length** is related to the length of the lens. More precisely, it is the distance between the camera's sensor and the lens system's optical center when the lens is focused at an infinitely distant target. The focal length should not be confused with the **focus distance**—the distance to objects that are in focus. The following diagram illustrates the meanings of focal length and focal distance as well as FOV:



With a higher f-number (a proportionally narrower aperture), a lens transmits a smaller proportion of incoming light. Specifically, the intensity of the transmitted light is inversely proportional to the square of the f-number. For example, when comparing the maximum apertures of two lenses, a photographer might write, "The f/2 lens is twice as fast as the f/2.8 lens." This means that the former lens can transmit twice the intensity of light, allowing an equivalent exposure to be taken in half the time.

A lens's **efficiency** or **transmittance** (the proportion of light transmitted) depends on not only the f-number but also non-ideal factors. For example, some light is reflected off the lens elements instead of being transmitted. The **T-number** or **T-stop** is an adjustment to the f-number based on empirical findings about a given lens's transmittance. For example, regardless of its f-number, a T/2.4 lens has the same transmittance as an ideal f/2.4 lens. For cinema lenses, manufacturers often provide T-number specifications but for other lenses, it is much more common to see only f-number specifications.

The sensor's **efficiency** is the proportion of the lens-transmitted light that reaches photosites and gets converted to a signal. If the efficiency is poor, the sensor misses much of the light! A more efficient sensor will tend to take well-exposed images for a broader range of camera settings, lens settings, and lighting conditions. Thus, efficiency gives the system more freedom to auto-select settings that are optimal for resolution and throughput. For the common type of sensor described in the previous section, *Coloring the light*, the choice of color filters has a big effect on efficiency. A camera designed to capture visible light in grayscale has high efficiency because it can receive all visible wavelengths at each photosite. A camera designed to capture visible light in multiple color channels typically has much lower efficiency because some wavelengths are filtered out at each photosite. A camera designed to capture NIR alone, by filtering out all visible light, typically has even lower efficiency.

Efficiency is a good indication of the system's ability to form *some kind of image* under diverse lighting (or radiation) conditions. However, depending on the subject and the real lighting, a relatively inefficient system could have higher contrast and better resolution. The advantages of selectively filtering wavelengths are not necessarily reflected in lp/mm, which measures black-on-white resolution.

By now, we have seen many quantifiable tradeoffs that complicate our efforts to capture a subject in a moment. As Robert Capa's advice implies, getting close with a short lens is a relatively robust recipe. It allows for good resolution with minimal risk of completely missing the framing or the focus. On the other hand, such a setup suffers from high distortion and, by definition, a short working distance. Moving beyond the capabilities of cameras in Capa's day, we have also considered the features and configurations that allow for high-throughput and high-efficiency video capture.

Having primed ourselves on wavelengths, image formats, cameras, lenses, capture settings, and photographers' common sense, let us now select several systems to study.

Rounding up the unusual suspects

This chapter's demo applications are tested with three cameras, which are described in the following table. The demos are also compatible with many additional cameras; we will discuss compatibility later as part of each demo's detailed description. The three chosen cameras differ greatly in terms of price and features but each one can do things that an ordinary webcam cannot!

Name	Price	Purposes	Modes	Optics
Sony PlayStation Eye	$10	Passive, color imaging in visible light	640x480 @ 60 FPS 320x240 @ 187 FPS	FOV: 75 degrees or 56 degrees (two zoom settings)
ASUS Xtion PRO Live	$230	Passive, color imaging in visible light Active, monochrome imaging in NIR light Depth estimation	Color or NIR: 1280x1024 @ 60 FPS Depth: 640x480 @ 30 FPS	FOV: 70 degrees
PGR Grasshopper 3 GS3-U3-23S6M-C	$1000	Passive, monochrome imaging in visible light	1920x1200 @ 162 FPS	C-mount lens (not included)

[For examples of lenses that we can use with the GS3-U3-23S6M-C camera, refer to the *Shopping for glass* section, later in this chapter.]

We will try to push these cameras to the limits of their capabilities. Using multiple libraries, we will write applications to access unusual capture modes and to process frames so rapidly that the input remains the bottleneck. To borrow a term from the automobile designers who made 1950s muscle cars, we might say that we want to "supercharge" our systems; we want to supply them with specialized or excess input to see what they can do!

Supercharging the PlayStation Eye

Sony developed the Eye camera in 2007 as an input device for PlayStation 3 games. Originally, no other system supported the Eye. Since then, third parties have created drivers and SDKs for Linux, Windows, and Mac. The following list describes the current state of some of these third-party projects:

- For Linux, the gspca_ov534 driver supports the PlayStation Eye and works out of the box with OpenCV's videoio module. This driver comes standard with most recent Linux distributions. Current releases of the driver support modes as fast as 320x240 @ 125 FPS and 640x480 @ 60 FPS. An upcoming release will add support for 320x240 @187 FPS. If you want to upgrade to this future version today, you will need to familiarize yourself with the basics of Linux kernel development, and build the driver yourself.

> See the driver's latest source code at `https://github.com/torvalds/linux/blob/master/drivers/media/usb/gspca/ov534.c`. Briefly, you would need to obtain the source code of your Linux distribution's kernel, merge the new `ov534.c` file, build the driver as part of the kernel, and finally, load the newly built gspca_ov534 driver.

- For Mac and Windows, developers can add PlayStation Eye support to their applications using an SDK called PS3EYEDriver, available from `https://github.com/inspirit/PS3EYEDriver`. Despite the name, this project is not a driver; it supports the camera at the application level, but not the OS level. The supported modes include 320x240 @ 187 FPS and 640x480 @ 60 FPS. The project comes with sample application code. Much of the code in PS3EYEDriver is derived from the GPL-licensed gspca_ov534 driver, and thus, the use of PS3EYEDriver is probably only appropriate to projects that are also GPL-licensed.
- For Windows, a commercial driver and SDK are available from Code Laboratories (CL) at `https://codelaboratories.com/products/eye/driver/`. At the time of writing, the CL-Eye Driver costs $3. However, the driver does not work with OpenCV 3's videoio module. The CL-Eye Platform SDK, which depends on the driver, costs an additional $5. The fastest supported modes are 320x240 @ 187 FPS and 640x480 @ 75 FPS.
- For recent versions of Mac, no driver is available. A driver called macam is available at `http://webcam-osx.sourceforge.net/`, but it was last updated in 2009 and does not work on Mac OS X Mountain Lion and newer versions.

Thus, OpenCV in Linux can capture data directly from an Eye camera, but OpenCV in Windows or Mac requires another SDK as an intermediary.

First, for Linux, let us consider a minimal example of a C++ application that uses OpenCV to record a slow-motion video based on high-speed input from an Eye. Also, the program should log its frame rate. Let's call this application Unblinking Eye.

Unblinking Eye's source code and build files are in this book's GitHub repository at `https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_1/UnblinkingEye`.

Note that this sample code should also work with other OpenCV-compatible cameras, albeit at a slower frame rate compared to the Eye.

Unblinking Eye can be implemented in a single file, `UnblinkingEye.cpp`, containing these few lines of code:

```
#include <stdio.h>
#include <time.h>

#include <opencv2/core.hpp>
#include <opencv2/videoio.hpp>

int main(int argc, char *argv[]) {

  const int cameraIndex = 0;
  const bool isColor = true;
  const int w = 320;
  const int h = 240;
  const double captureFPS = 187.0;
  const double writerFPS = 60.0;
  // With MJPG encoding, OpenCV requires the AVI extension.
  const char filename[] = "SlowMo.avi";
  const int fourcc = cv::VideoWriter::fourcc('M','J','P','G');
  const unsigned int numFrames = 3750;

  cv::Mat mat;

  // Initialize and configure the video capture.
  cv::VideoCapture capture(cameraIndex);
  if (!isColor) {
    capture.set(cv::CAP_PROP_MODE, cv::CAP_MODE_GRAY);
  }
  capture.set(cv::CAP_PROP_FRAME_WIDTH, w);
```

```
    capture.set(cv::CAP_PROP_FRAME_HEIGHT, h);
    capture.set(cv::CAP_PROP_FPS, captureFPS);

    // Initialize the video writer.
    cv::VideoWriter writer(
        filename, fourcc, writerFPS, cv::Size(w, h), isColor);

    // Get the start time.
    clock_t startTicks = clock();

    // Capture frames and write them to the video file.
    for (unsigned int i = 0; i < numFrames;) {
      if (capture.read(mat)) {
        writer.write(mat);
        i++;
      }
    }

    // Get the end time.
    clock_t endTicks = clock();

    // Calculate and print the actual frame rate.
    double actualFPS = numFrames * CLOCKS_PER_SEC /
        (double)(endTicks - startTicks);
    printf("FPS: %.1f\n", actualFPS);
  }
```

Note that the camera's specified mode is 320x240 @ 187 FPS. If our version of the gspca_ov534 driver does not support this mode, we can expect it to fall back to 320x240 @ 125 FPS. Meanwhile, the video file's specified mode is 320x240 @ 60 FPS, meaning that the video will play back at slower-than-real speed as a special effect. Unblinking Eye can be built using a Terminal command such as the following:

```
$ g++ UnblinkingEye.cpp -o UnblinkingEye -lopencv_core -lopencv_videoio
```

Build Unblinking Eye, run it, record a moving subject, observe the frame rate, and play back the recorded video, `SlowMo.avi`. How does your subject look in slow motion?

On a machine with a slow CPU or slow storage, Unblinking Eye might drop some of the captured frames due to a bottleneck in video encoding or file output. Do not be fooled by the low resolution! The rate of data transfer for a camera in 320x240 @ 187 FPS mode is greater than for a camera in 1280x720 @ 15 FPS mode (an HD resolution at a slightly choppy frame rate). Multiply the pixels by the frame rate to see how many pixels per second are transferred in each mode.