Image, Video, and Data Files

* HighGUI: Portable Graphics Toolkit
  - → "high-level graphical user interface"
  - → Interact with OS, the file system and hardware such as Camera.
  - → Read and write graphics-related files (Images & Video)
  - → Open and manage windows, to display images;
    - → Handle simple mouse pointer and Keyboard event.

⇒ HighGUI library in OpenCV can be divided into three parts:
- → Hardware part (Operation of Cameras)
- → file system part
- → GUI part:

* Working with Image file

⇒ Functions here deals, with the complexities associated with compressing and decompressing the image data.

* Loading and Saving Images

⇒ The easiest way to do this is with the high-level functions cv::imread() and cv::imwrite().

* Reading a file with cv::imread()

```
cv::Mat cv::imread(
    const string& filename,
    int flags = cv::IMREAD_COLOR
);
```

{declaration of cv::imread}

⇒ When opening an image, cv::imread() doesn't look at the file extension;
  ↳ Instead, it analyzes the first few bytes of the file (a.k.a signature) and determines the appropriate codec using it.

Table 8.1 → All the flags

cv::IMREAD_COLOR (Default)

⇒ cv::imread() does not give a runtime error when it fails to load an image.
  ↳ It simply returns empty cv::Mat.

* Writing a file with cv::imwrite()

```
bool cv::imwrite(
    const String& filename,
    cv::InputArray image,
    const vector<int>& params = vector<int>()
);
```

⇒ The first argument gives the filename, whose extension is used to determine the format in which the file will be stored.

Example: of some popular extensions supported by OpenCV.

.jpg , .jpeg , .png etc

⇒ The second argument is the image to be stored.

Codecs → {Compression and Decompression Libraries}

★ Reading Video with the cv::VideoCapture object

⇒ cv::VideoCapture object contains the information needed for reading frame from a camera or video file.

⇒ Depending on the source, we use one of three different calls to create a cv::VideoCapture object.

ⓐ cv::VideoCapture::VideoCapture(
const string& filename,
);

ⓑ cv::VideoCapture::VideoCapture(
int device
);

ⓒ cv::VideoCapture::VideoCapture();

⇒ If the open is successful and we are able to start reading frames, the member function cv::VideoCapture::isOpened() will return true.

⇒ As with the image codecs, you will need to have the appropriate libraries already residing on your computer in order to successfully read the video file.

⇒ If you use 3rd construction then you need to call cv::VideoCapture::Open().

Example

cap.open("my_video.avi");

* Reading frames with cv::VideoCapture::read()

```
bool cv::VideoCapture::read(
    cv::OutputArray image
);
```

↳ {this will get the next frame}

⇒ This action will automatically "advance" the video capture object such that a subsequent call to cv::VideoCapture::read() will return the next frame and so on.

⇒ If the read was not successful, then this function will return false (otherwise it will return true).

* Reading frames with cv::VideoCapture::operator >> ()

⇒ Behaves exactly the same as cv::VideoCapture::read(), except that, it returns reference to the original cv::VideoCapture object regardless of whether it succeeded.

↳ In this case, you must check that the return array is not empty.

★ Reading frames with cv::VideoCapture::grab and cv::VideoCapture::retrieve()

⇒ There are many reasons to grab and retrieve separately rather than together as would be the case in calling cv::VideoCapture::read()

Example: When there are multiple cameras (Stereo Imaging)

★ Camera properties: cv::VideoCapture::get() and cv::VideoCapture::set()

⇒ Video file contains not only video frames themselves, but also important meta-data, which can be essential for handling the files correctly.

Table 8.4 → {List of VideoCapture Properties}

★ Writing Video with the cv::VideoWriter object

⇒ We need to create cv::VideoWriter object before we can write out our video.

```
cv::VideoWriter::VideoWriter(
    const string& filename,
    int fourcc,
    double fps,
    cv::Size frame_size,
    bool is_color = true
);
```

Example

```
cv::VideoWriter out;
out.open(
    "my_video.mpg",
    CV_FOURCC('D','I','V','X'),
    30.0,
    cv::Size(640, 480),
    true
);
```

⇒ cv::VideoWriter::isOpened() method, which will return true if you are good to go.

* Writing frames with cv::VideoWriter::write()

```
cv::VideoWriter::write(
    const Mat& image
);
```

★ Writing frames with cv::VideoWriter ::operator<<()

```
my_video_writer << my_frame
```

★ Data Persistence

⇒ The basic mechanism for reading and writing files is the cv::FileStorage object.

★ Writing to a cv::FileStorage

```
FileStorage::FileStorage(String fileName, int flag);
```

⇒ The cv::FileStorage object is a representation of an XML or YAML data file.

```
FileStorage::FileStorage();
cv::FileStorage::open();
```

flag → cv::FileStorage::WRITE
flag → cv::FileStorage::APPEND

⇒ Data inside cv::FileStorage is stored in one of two forms, either as a "mapping" or a "sequence".

↳ At the top level, the data you write to the file storage is all mapping and inside of that you can place other mappings or sequence.

⇒ Once you are completely done writing, you close the file with the cv::FileStorage::release() member function.

* Reading from a cv::FileStorage

⇒ The cv::FileStorage object can be opened for reading the same way it is opened for writing except that the flag argument should be set to cv::FileStorage::READ

⇒ The data can be read using overloaded array operator cv::FileStorage::operator[]().

- → We need to pass the string key
- → Returned object is of type cv::FileNode.

* cv::FileNode

⇒ If it represents an object, you can just load it into a variable of the appropriate type with the overloaded extraction operator cv::FileNode::operator>>().

Example

```
int num;
myFileStorage["SumInteger"] >> num;
```