Modern C++ for Computer Vision and Image Processing

Tutorial : OpenCV4

Ignacio Vizzo and Cyrill Stachniss
OpenCV

- Popular library for **Image Processing**
- We will be using **version 4** of OpenCV
- We will be using just a small part of it
- `#include <opencv2/opencv.hpp>` to use all functionality available in OpenCV
- Namespace `cv::`
Data types

- OpenCV uses **own types**
- OpenCV trusts you to pick the correct type
- Names of types follow pattern
  `CV_<bit_count><identifier><num_of_channels>`

**Example**: RGB image is `CV_8UC3`:
8-bit unsigned char with 3 channels for RGB

**Example**: Grayscale image is `CV_8UC1`:
single 8-bit unsigned char for intensity

- Better to use `DataType`

**Example**: `DataType<uint>::type == CV_8UC1`
Basic Matrix Type

- Every image is a `cv::Mat`, for “Matrix”
- `Mat image(rows, cols, DataType, Value);`
- `Mat_<T> image(rows, cols, Value);`
- Initialize with `zeros`:
  1. `cv::Mat image = cv::Mat::zeros(10, 10, CV_8UC3);`
  2. `using Matf = cv::Mat_<float>;
  3. `Matf image_float = Matf::zeros(10, 10);`

- Get type identifier with `image.type()`;
- Get size with `image.rows, image.cols`
- **I/O:**
  - Read image with `imread`
  - Write image with `imwrite`
  - Show image with `imshow`
  - Detects I/O method from extension
cv::Mat is sort of shared pointer

It does not use std::shared_ptr but follows the same principle of reference counting

```cpp
#include <opencv2/opencv.hpp>
#include <iostream>

int main() {
    using Matf = cv::Mat_<float>;
    Matf image = Matf::zeros(10, 10);
    Matf image_no_copy = image; // Does not copy!
    image_no_copy.at<float>(5, 5) = 42.42f;
    std::cout << image.at<float>(5, 5) << std::endl;
    Matf image_copy = image.clone(); // Copies image.
    image_copy.at<float>(1, 1) = 42.42f;
    std::cout << image.at<float>(1, 1) << std::endl;
}
```

```
c++ -std=c++11 -o copy copy.cpp \ `pkg-config --libs --cflags opencv`
```
**imread**

- Read image from file
- ```Mat imread(const string& file, int mode=1)```  
- **Different modes:**
  - unchanged: `cv::IMREAD_UNCHANGED < 0`
  - 1 channel: `cv::IMREAD_GREYSCALE == 0`
  - 3 channels: `cv::IMREAD_COLOR > 0`

```cpp
#include <opencv2/imgcodecs.hpp>
#include <opencv2/opencv.hpp>
using namespace cv;

int main() {
    Mat i1 = imread("logo_opencv.png", cv::IMREAD_GRAYSCALE);
    Mat_<uint8_t> i2 = imread("logo_opencv.png", cv::IMREAD_GRAYSCALE);
    std::cout << (i1.type() == i2.type()) << std::endl;
    return 0;
}
```
imwrite

- Write the image to file
- Format is guessed from extension
- `bool imwrite(const string& file, const Mat& img);`

```cpp
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
int main() {
    cv::Mat image = cv::imread("logo_opencv.png", cv::IMREAD_GRAYSCALE);
    cv::imwrite("copy.jpg", image);
    return 0;
}
```
Write float images to *.exr files

- When storing floating point images OpenCV expects the values to be in $[0, 1]$ range
- When storing arbitrary values the values might be cut off
- Save to *.exr files to avoid this
- These files will store and read values as is without losing precision
# include <iostream>
#include <string>

#include <opencv2/opencv.hpp>

int main () {
    using Matf = cv::Mat_<float>;
    Matf image = Matf::zeros(10, 10);
    image.at<float>(5, 5) = 42.42f;
    std::string f = "test.exr";
    cv::imwrite(f, image);
    Matf copy = cv::imread(f, cv::IMREAD_UNCHANGED);
    std::cout << copy.at<float>(5, 5) << std::endl;
    return 0;
}

Hint: try what happens when using png images instead
### imshow

- Display the image on screen
- Needs a window to display the image

```cpp
#include <opencv2/opencv.hpp>

int main() {
    cv::Mat image = cv::imread("logo_opencv.png", cv::IMREAD_COLOR);
    std::string window_name = "Window name";
    // Create a window.
    cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);
    cv::imshow(window_name, image); // Show image.
    cv::waitKey(); // Don't close window instantly.
    return 0;
}
```
OpenCV vector type

- OpenCV vector type: `cv::Vec<Type, SIZE>`
- Many typedefs available: `Vec3f`, `Vec3b`, etc.
- Used for pixels in multidimensional images: `mat.at<Vec3b>(row, col)`;

```cpp
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace cv;

int main() {
    Mat mat = Mat::zeros(10, 10, CV_8UC3);
    std::cout << mat.at<Vec3b>(5, 5) << std::endl;
    Mat_<Vec3f> matf3 = Mat_<Vec3f>::zeros(10, 10);
    std::cout << matf3.at<Vec3f>(5, 5) << std::endl;
}
```
Mixing up types is painful!

- OpenCV trusts **you** to pick the type
- This can cause errors
- OpenCV interprets bytes stored in `cv::Mat` according to the type the user asks (similar to `reinterpret_cast`)
- **Make sure you are using correct types!**
Mixing up types is painful!

```cpp
#include <opencv2/opencv.hpp>

int main() {
    cv::Mat image = cv::Mat::zeros(800, 600, CV_8UC3);
    std::string window_name = "Window name";
    cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);
    cv::imshow(window_name, image);
    cv::waitKey();
    for (int r = 0; r < image.rows; ++r) {
        for (int c = 0; c < image.cols; ++c) {
            // WARNING! WRONG TYPE USED!
            image.at<float>(r, c) = 1.0f;
        }
    }
    cv::imshow(window_name, image);
    cv::waitKey();
    return 0;
}
```
SIFT Descriptors

- **SIFT**: Scale Invariant Feature Transform
- **Popular features**: illumination, rotation and translation invariant (to some degree)

Image courtesy of David G. Lowe
SIFT Extraction With OpenCV

- **SiftFeatureDetector** to detect the keypoints
- **SiftDescriptorExtractor** to compute descriptors in keypoints

```cpp
// Detect keypoints.
auto detector = SiftFeatureDetector::create();
vector<cv::KeyPoint> keypoints;
detector->detect(input, keypoints);

// Show the keypoints on the image.
Mat image_with_keypoints;
drawKeypoints(input, keypoints, image_with_keypoints);

// Extract the SIFT descriptors
auto extractor = SiftDescriptorExtractor::create();
eextractor->compute(input, keypoints, descriptors);
```
**FLANN in OpenCV**

- **FLANN**: Fast Library for Approximate Nearest Neighbors
- build K-d tree, search for neighbors there

```cpp
1 // Create a kdtree for searching the data.
2 cv::flann::KDTreeIndexParams index_params;
3 cv::flann::Index kdtree(data, index_params);
4 ...
5 // Search the nearest vector to some query
6 int k = 1;
7 Mat nearest_vector_idx(1, k, DataType<int>::type);
8 Mat nearest_vector_dist(1, k, DataType<float>::type);
9 kdtree.knnSearch(query, nearest_vector_idx,
10     nearest_vector_dist, k);
```
OpenCV 4 with CMake

- Install OpenCV 4 in the system see:
  https://gitlab.igg.uni-bonn.de/teaching/example_opencv

- Find using `find_package(OpenCV 4 REQUIRED)`
  
- Include `${OpenCV_INCLUDE_DIRS}`

- Link against `${OpenCV_LIBS}`
  
  ```
  1 add_library(some_lib some_lib_file.cpp)
  2 target_link_libraries(some_lib ${OpenCV_LIBS})
  3 add_executable(some_program some_file.cpp)
  4 target_link_libraries(some_program ${OpenCV_LIBS})
  ```
Additional OpenCV information

- We are using **OpenCV version 4**
- Example project with additional information about using SIFT and FLANN can be found here:
  
  https://gitlab.igg.uni-bonn.de/teaching/example_opencv