Modern C++ for Computer Vision and Image Processing

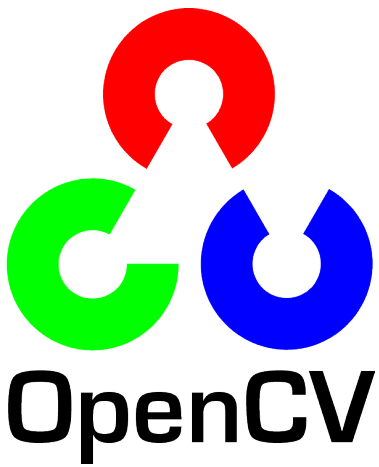
Tutorial : OpenCV4

Ignacio Vizzo and Cyrill Stachniss

UNIVERSITÄT BONN

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::`
- More here: http://opencv.org/



Data types

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

```
#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`

```
1  #include <opencv2/imgcodecs.hpp>
2  #include <opencv2/opencv.hpp>
3  using namespace cv;
4  int main() {
5    Mat i1 = imread("logo_opencv.png",
6                    cv::IMREAD_GRAYSCALE);
7    Mat_<uint8_t> i2 = imread("logo_opencv.png",
8                              cv::IMREAD_GRAYSCALE);
9    std::cout << (i1.type() == i2.type()) << std::endl;
10   return 0;
11 }
```

imwrite

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

```
1 #include <opencv2/core.hpp>
2 #include <opencv2/highgui.hpp>
3 int main() {
4   cv::Mat image = cv::imread("logo_opencv.png",
5                              cv::IMREAD_GRAYSCALE);
6   cv::imwrite("copy.jpg", image);
7   return 0;
8 }
```

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

Float images I/O example

```cpp
#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
- `void imshow(const string& window_name, const Mat& mat)`

```
1  // clang-format off
2  #include <opencv2/opencv.hpp>
3  int main() {
4    cv::Mat image = cv::imread("logo_opencv.png",
5                               cv::IMREAD_COLOR);
6    std::string window_name = "Window name";
7    // Create a window.
8    cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);
9    cv::imshow(window_name, image);  // Show image.
10   cv::waitKey();  // Don't close window instantly.
11   return 0;
12 }
```

OpenCV vector type

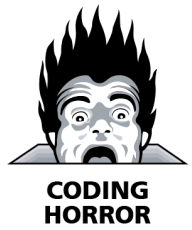
- OpenCV vector type: `cv::Vec<Type, SIZE>`
- Many typedefs available: `Vec3f`, `Vec3b`, etc.
- Used for pixels in multidimensional images: `mat.at<Vec3b>(row, col);`

```
1 #include <opencv2/opencv.hpp>
2 #include <iostream>
3 using namespace cv;
4 int main() {
5     Mat mat = Mat::zeros(10, 10, CV_8UC3);
6     std::cout << mat.at<Vec3b>(5, 5) << std::endl;
7     Mat_<Vec3f> matf3 = Mat_<Vec3f>::zeros(10, 10);
8     std::cout << matf3.at<Vec3f>(5, 5) << std::endl;
9 }
```

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!




```
#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**: **S**cale **I**nvariant **F**eature **T**ransform
- **Popular features**: illumination, rotation and translation invariant (to some degree)

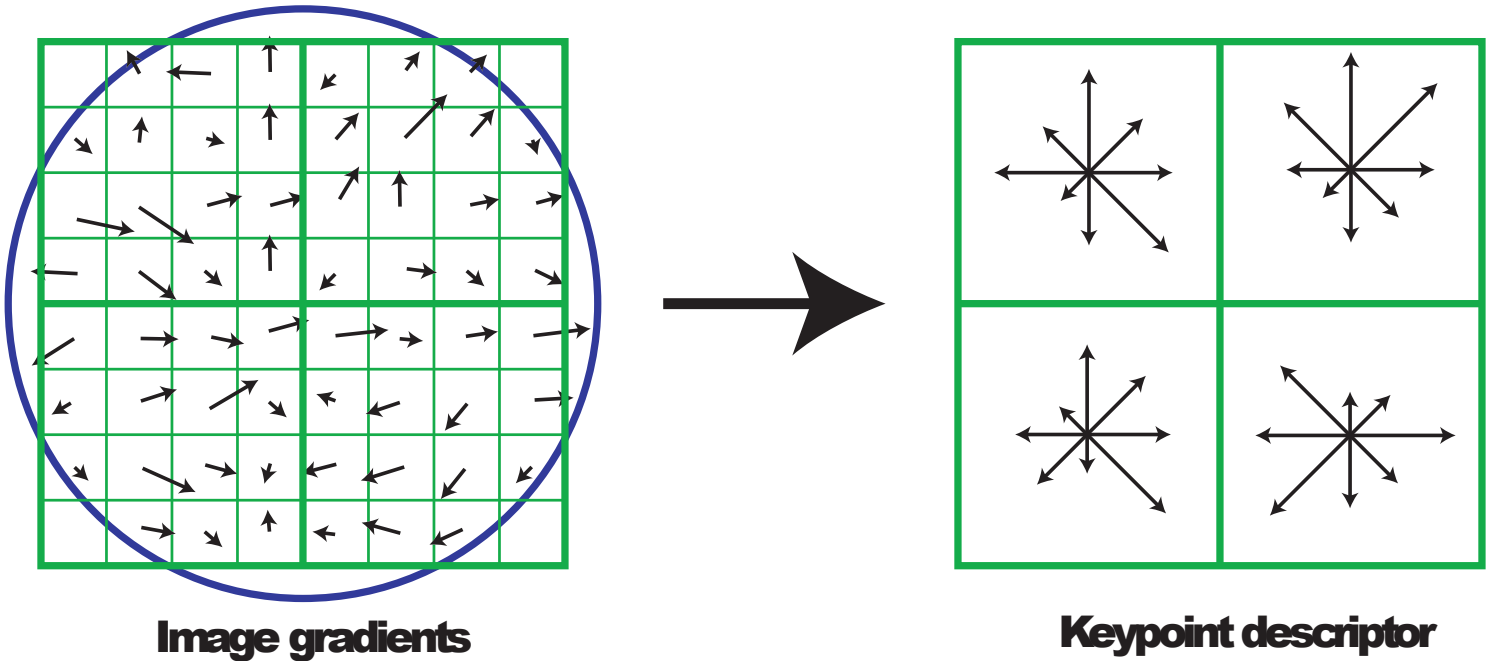


image courtesy of David G. Lowe

SIFT Extraction With OpenCV

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

```
1  // Detect key points.
2  auto detector = SiftFeatureDetector::create();
3  vector<cv::KeyPoint> keypoints;
4  detector->detect(input, keypoints);
5
6  // Show the keypoints on the image.
7  Mat image_with_keypoints;
8  drawKeypoints(input, keypoints, image_with_keypoints);
9
10 // extract the SIFT descriptors
11 auto extractor = SiftDescriptorExtractor::create();
12 extractor->compute(input, keypoints, descriptors);
```

FLANN in OpenCV

- **FLANN**: **F**ast **L**ibrary for **A**pproximate **N**earest **N**eighbors
- build K-d tree, search for neighbors there

```
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)`

```
1 find_package(OpenCV 3 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