Homework 1

Computer Vision 2016 Spring

March 13, 2016

OpenCV

- BSD 3-clause license
- De facto standard for computer vision and image processing
- Highly optimized C++ implementation of many CV algorithms
- Cross-platform
- Many bindings (Python, Matlab, Java . . .)

OpenCV Installation (in Windows)

1. Download and install *OpenCV* [1]
2. Suppose we extract *OpenCV* to `C:\hw1\` at step 1
   - Add a new environment variable `OpenCV_DIR` with value `C:\hw1\opencv\build`
   - Add `%OpenCV_DIR%\x64\vc14\bin` [2] to the environment variable `PATH`

Note

The source code can be found on GitHub—Itseez/opencv and GitHub—Itseez/opencv_contrib

[1] link to *OpenCV* 3.1 Windows Installer (For *VS2013*, *VS2015*)

[2] `vc12` for *VS2013*, `vc14` for *VS2015*

Create a Visual Studio Project

With OpenCV

1. Download and install *CMake* [3]
2. Create an empty directory, for example `C:\hw1\test_cv\`
3. In the directory create a text file named `main.cpp`
4. In the directory create a text file named `CMakeLists.txt` with the following contents:

CMakeLists.txt

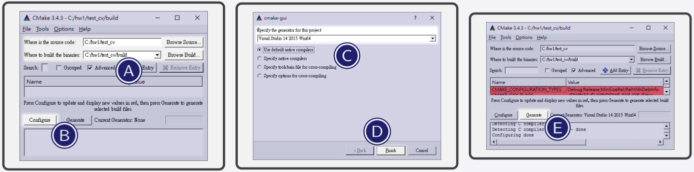
```
cmake_minimum_required(VERSION 2.8.11)
find_package(OpenCV REQUIRED)
set(vs_solution_name "test_cv_vs_solution")
set(vs_project_name "test_cv_vs_project")
project(${vs_solution_name})
add_executable(${vs_project_name} main.cpp)
target_link_libraries(${vs_project_name} ${OpenCV_LIBS})
```

[3] link to *CMake* 3.4.3 Win32 Installer

Create a Visual Studio Project

With OpenCV

5. Open `cmake-gui`. Follow structions below:



- A Set `Where is the source code` to `C:/hw1/test_cv` and `Where to build the binaries` to `C:/hw1/test_cv/build`
- B Click `Configure`
- C Select `Visual Studio 14 2015 Win64` as the generator, then Click `Finish`.
- D Click `Finish`
- E Click `Generate`

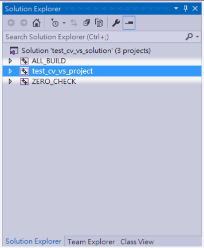
Create a Visual Studio Project

With OpenCV

6. Open `C:\hw1\test_cv\build\test_cv_vs_solution.sln`
7. Follow the instructions below:

   A In `Solution Explorer` select `test_cv_vs_project`

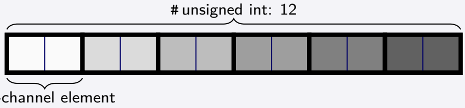
   B On the `Project menu` choose `Set as StartUp Project`

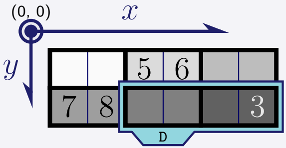


Mat

- Dense multi-dimensional array
- Handle the memory automatically

cv::Mat(2, 3, CV_8UC(2))





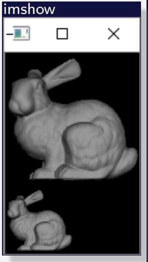

```
Mat A(Mat::zeros(2, 3, CV_8UC(2)));
A.at<Vec<uchar, 2>>(0, 1) = Vec<uchar, 2>(5, 6);
A.at<Vec2b>(Point(0, 1)) = Vec2b(7, 8);
Mat B(A), C;
C = A;
Mat D(A(Rect(1, 1, 2, 1))); // Rect(x, y, w, h)
A.release(); B.release(); C.release();
// the whole array is still referenced by D
D.at<Vec2b>(0, 1)[1] = 3;
```

Sample

See OpenCV documentation and samples[4] for more information

main.cpp

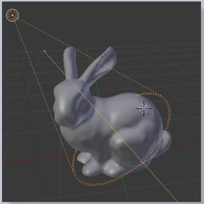
```cpp
#include <opencv2\core.hpp>
#include <opencv2\imgcodecs.hpp> // imread
#include <opencv2\highgui.hpp> // imshow, waitKey
using namespace cv;
int main() {
    // Load image as a single channel grayscale Mat.
    Mat img = imread("pic1.bmp", IMREAD_GRAYSCALE);
    // Mat is a thin template wrapper on top of the Mat class.
    // Mat_::operator()(y, x) does the same thing as Mat::at(y, x).
    Mat_<uchar> imgWrp(img);
    Mat_<uchar> smallImgWrp(img.size() / 2);
    for (int rowIndex = 0; rowIndex != smallImgWrp.rows; ++rowIndex)
        for (int colIndex = 0; colIndex != smallImgWrp.cols; ++colIndex)
            smallImgWrp(rowIndex, colIndex) =
                imgWrp(rowIndex * 2, colIndex * 2);
    Mat result(img.rows + smallImgWrp.rows, img.cols, CV_8U, Scalar(0));
    imgWrp.copyTo(result(Rect(0, 0, imgWrp.cols, imgWrp.rows)));
    smallImgWrp.copyTo(
        result(Rect(0, imgWrp.rows, smallImgWrp.cols, smallImgWrp.rows)));
    cv::imshow("Hi", result);
    cv::waitKey(); // Wait for the user to press a key.
    return 0;
}
```



[4] links to documentation and samples

Photometric Stereo

grayscale



Lambertian Reflection

$$\color{red}{i} = \mathrm{k_d}\mathrm{l}(\mathbf{s}^\intercal \mathbf{n})$$

$$\mathrm{i} = \mathrm{k_d}\mathrm{l}(\mathbf{s}^\intercal \color{red}{\boldsymbol{n}})$$

$i_{x,y}^{(m)}$	the intensity of the mth image at pixel (x, y)
$\mathrm{k_d}$	the color of the surface
l_m	the intensity of the mth incoming light
s_m	the **unit vector** pointing from the surface to the m incoming light
$\boldsymbol{n}_{x,y}$	the surface's normal vector (**unit vector**) at pixel (x, y)

For brevity, we omits the dependence of x, y and m.

Normal Estimation

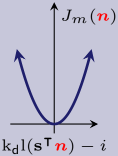
According to the reflection model, we suppose that the unknown normal vector and the intensity in the mth image at pixel (x, y) will satisfy

$$\mathrm{i}_{x,y}^{(m)} \stackrel{?}{=} \mathrm{k}_{\mathsf{d}} \mathrm{l}_m (\mathbf{s}_m^{\mathsf{T}} \boldsymbol{n})$$

To estimate how **bad** a **specific** $\boldsymbol{n}$ is, we define the *least squares* loss function

$$J(\boldsymbol{n}) = \sum_m J_m(\boldsymbol{n}) = \sum_m \|\mathrm{k}_{\mathsf{d}} \mathrm{l}_m (\mathbf{s}_m^{\mathsf{T}} \boldsymbol{n}) - \mathrm{i}^{(m)}\|^2$$

Normal Estimation



- The more loss $J(\boldsymbol{n})$ $\boldsymbol{n}$ has, the less chance $\boldsymbol{n}$ is the correct normal vector at pixel (x, y).
- Solve (`cv::Mat::inv`, `cv::Mat::t`) the following linear system to get the $\boldsymbol{n}$ with minimum loss:

$$\mathrm{S}^\intercal \mathrm{S}\boldsymbol{b} = \mathrm{S}^\intercal \mathbf{i}, \quad \text{where } \mathrm{S} = \begin{bmatrix} \mathrm{l}_1 \mathbf{s}_1^\intercal \\ \mathrm{l}_2 \mathbf{s}_2^\intercal \\ \vdots \\ \mathrm{l}_m \mathbf{s}_m^\intercal \end{bmatrix}, \mathbf{i} = \begin{bmatrix} \mathrm{i}^{(1)} \\ \mathrm{i}^{(2)} \\ \vdots \\ \mathrm{i}^{(m)} \end{bmatrix} \text{ and } \boldsymbol{b} = \mathrm{k_d}\boldsymbol{n}$$

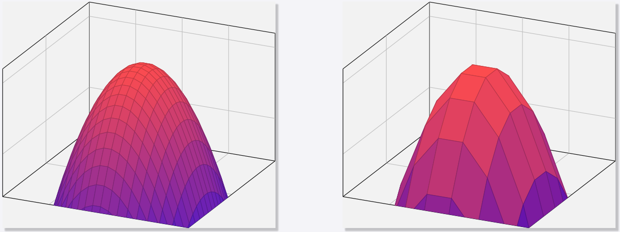
In practice we solve the problem with *QR* or *SVD*.

Surface Reconstruction

The surface $z(x, y)$ near pixel $(\mathrm{x}^*, \mathrm{y}^*)$ can be approximated by the tangent plane:

$$\mathrm{n}_1(x - \mathrm{x}^*) + \mathrm{n}_2(y - \mathrm{y}^*) + \mathrm{n}_3(z - z(\mathrm{x}^*, \mathrm{y}^*)) = 0 \qquad (1)$$

where $(\mathrm{n}_1, \mathrm{n}_2, \mathrm{n}_3)^\intercal$ is the normal vector at $(\mathrm{x}^*, \mathrm{y}^*)$.



Surface Reconstruction

The equation 1 can be rewritten as

$$z_{\text{approx}}(x, y) = \left(-\frac{\mathrm{n}_1}{\mathrm{n}_3}\right) x + \left(-\frac{\mathrm{n}_2}{\mathrm{n}_3}\right) y + \text{constant}$$

We can reconstruct the surface $\tilde{z}(x, y)$ as we know the gradient of z_{approx} at each pixel, for example, by

$$\tilde{z}(x, y) = \sum_{i=0}^{x-1} \left.\frac{\partial z_{\text{approx}}}{\partial x}\right|_{(i,0)} + \sum_{j=0}^{y-1} \left.\frac{\partial z_{\text{approx}}}{\partial y}\right|_{(x,j)}$$

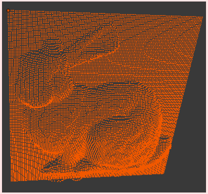
Surface Reconstruction

Note

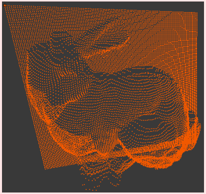
You will probably have to scale $\tilde{z}$ for visualizing the surface:

$$\tilde{z}_{\text{vis}}(x, y) = \alpha \tilde{z}(x, y)$$

$\alpha = 1$



$\alpha = 4$



Surface Reconstruction

Other Tips

- Weighted Least Squares measure loss terms with different weights

$$J_{\mathbf{W}}(\boldsymbol{n}) = \sum_m W_m J_m(\boldsymbol{n}) = \sum_m W_m \|\mathrm{k_d} \mathrm{l}_m (\mathbf{s}_\mathrm{m}{}^\intercal \boldsymbol{n}) - \mathrm{i}^{(m)}\|^2$$

$$= \|\mathbf{WS}\boldsymbol{b} - \mathbf{Wi}\|^2, \quad \text{where } \mathbf{W} = \begin{bmatrix} \mathrm{w}_1 & & \\ & \mathrm{w}_2 & \\ & & \ddots \end{bmatrix}, \mathrm{w}_m = \sqrt{\mathrm{W}_m}$$

- Sanity Check

$$\frac{\partial^2 z}{\partial x \partial y} = \frac{\partial^2 z}{\partial y \partial x}$$

Homework 1

- 70% Follow the instructions in page 8-12 to reconstruct surfaces
- 15% Experiment with tips mentioned in class and in page 14
- 15% Take pictures of real objects as input data
- 10% Reconstruct surfaces by solving optimization problems (see the Appendix in the course material)
- 10% Other cool things you find