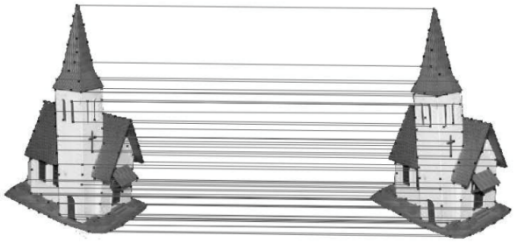
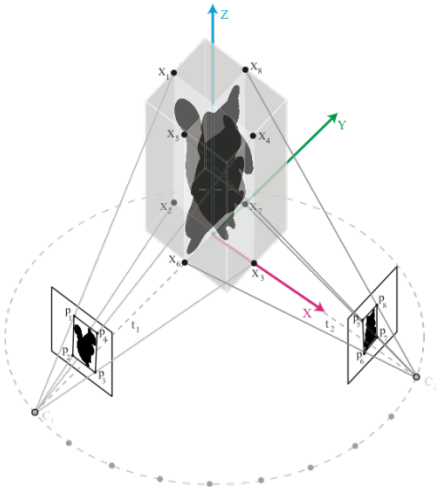
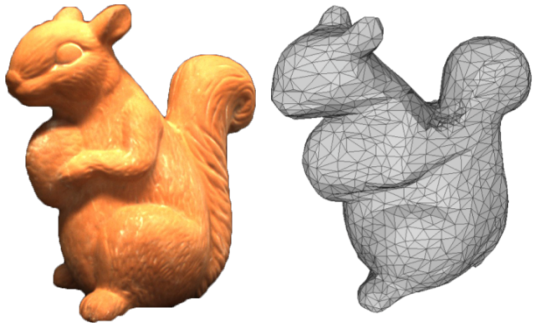
A short introduction to OpenCV

Open Source Computer Vision Library



OpenCV





What is OpenCV?

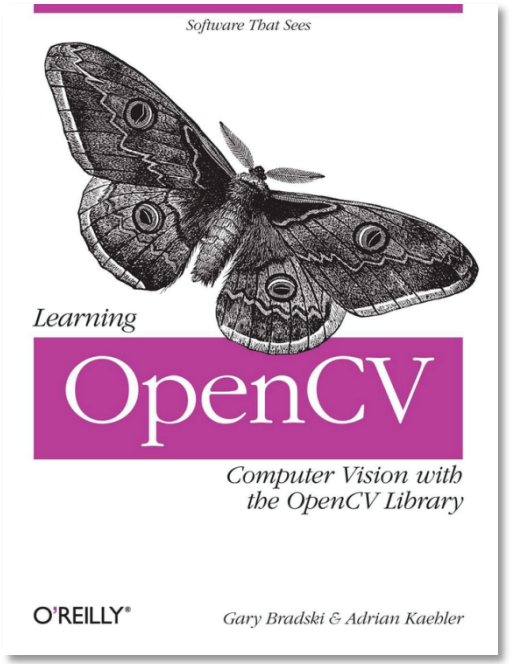
- Intel© Open Source Computer Vision Library
  - Based on Intel© IPL (Image Processing Library)
    - IPL, SPL (Signal Processing Library), IJL (Intel JPEG Library) meanwhile integrated in Intel© IPP (Integrated Performance Primitives)
- Image processing library for Windows, Mac OSX, Linux
- Available for C, C++, Python, Java, Android, iOS
- Fast! Optimized for PCs with Intel© architecture
- BSD-like license (free for academic/commercial use)

What is OpenCV?

- Over 500 functions
  - Many „basic“ functions in the areas of image processing, computer vision and general numerical algorithms
  - Many „state-of-the-art“ research algorithms
    - Changes quite frequently, so in doubt rather read the code than the documentation

Sources of information for OpenCV

- Official OpenCV (Sourceforge) Site:
  https://github.com/Itseez/opencv
- Online documentation:
  http://docs.opencv.org/
- News-Group:
  http://answers.opencv.org/questions/
- Literature:
  Gary Bradski and Adrian Kaehler,
  Learning OpenCV: Computer Vision with the OpenCV Library,
  O'Reilly Media, 2008



OpenCV users and areas of applications

- OpenCV is used by big companies such as
  - Intel, IBM, Microsoft, SONY, Siemens, Google, ...
  - Stanford, MIT, Cambridge, ...
- OpenCV is used in
  - Image processing (filtering, segmentation, ...)
  - Human-Computer-Interaction
  - Object identification
  - Face recognition
  - Gesture recognition
  - Motion tracking
  - Structure-from-Motion
  - Robotics

Face recognition example

1. Reading training data

```
std::vector<cv::Mat> images;
std::vector<int> labels;
readFacesData(images, labels); // 0 -> Damon, 1 -> Aniston
cv::Ptr<cv::FaceRecognizer> model = cv::createFisherFaceRecognizer();
model->train(images, labels);
```

2. Specifying classifier

```
cv::CascadeClassifier haarCascade;
haarCascade.load("haarcascade_frontalface_default.xml");
```

3. Load test image

```
cv::Mat testImg = cv::imread("who_is_it.png");
cv::cvtColor(testImg, testImg, CV_BGR2GRAY);
```

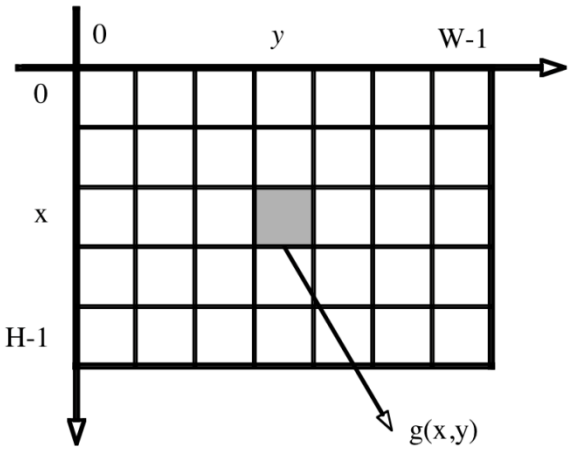
4. Find faces and recognize aniston/damon

```
std::vector<cv::Rect_<int>> faces;
haarCascade.detectMultiScale(testImg, faces);
for (int i = 0; i < faces.size(); ++i) {
    cv::Mat face = testImg(faces[i]);
    cv::resize(face, face, cv::Size(images[0].cols, images[0].rows));
    cv::rectangle(output, faces[i], CV_RGB(0, 255, 0), 1);
    std::string person = model->predict(face) == 0 ? "Matt Damon" : "Jenny Aniston";
    cv::putText(output, person, cv::Point(max(faces[i].tl().x - 10, 0), max(faces[i].tl().y - 10, 0)));
}
cv::imshow("Who is this?", output);
cv::waitKey();
```

Face recognition based on geometric features, histogram-based, ...

Mathematical model of an image

- Mathematical model of a gray-scale image
  - $W = \{0,...,n\}$ Gray value amount
  - $G = (g(x,y))$ Gray value matrix of image
  - $x = 0, ..., H - 1$ H image rows
  - $y = 0, ..., W - 1$ W image columns
  - (x,y) Local coordinates
  - $g(x,y) = g \; e \; W$ Gray value at position (x,y)



- Color images are multi-channel images with
  - $G = (g(x,y,n))$ 3D Gray value matrix of image
  - $n = 0, ..., N - 1$ N = Number of channels
  - $g(x,y) = (g_0, ..., g_{N-1})^T \; e \; W^N$ Image point at position (x,y)
- OpenCV can cope well with (multi-channel) matrices

Excerpt from the range of functions

- Linear algebra
- Histogram functions
- Thresholding
- Camera calibration
- 3D reconstruction
- Optical flow
- Motion/Object tracking
- Kalman estimation
- Drawing primitives
- Image functions
  - Creation/Deletion of images
- Data structures
  - static/dynamic types
- Image statistics
  - median, norm, moments, ...
- Contour processing
  - find/show/manipulate contours
- Geometry processing
  - convex hull, line/ellipse fitting
- Feature detection
  - edges, corners
- Morphology
  - Erosion, Dillatation, Opening, Closing

Excerpt from the range of functions

- Linear algebra
- Histogram functions
- Thresholding
- Camera calibration
- 3D reconstruction
- Optical flow
- Motion/Object tracking
- Kalman estimation
- Drawing primitives

Demo of a 3D scanner using „off-the-shelf“ components (your laptop monitor that is) later on.

- Image functions
  - Creation/Deletion of images
- Data structures
  - static/dynamic types
- Image statistics
  - median, norm, moments, ...
- Shape analysis
  - convex hull, line/ellipse fitting
- Feature detection
  - edges, corners
- Morphology
  - Erosion, Dillatation, Opening, Closing

Excerpt from the range of functions

- Linear algebra
- Histogram functions
- Thresholding (global)

C++: double **threshold**(InputArray **src**, OutputArray **dst**, double **thresh**, double **maxval**, int **type**)

Original

t = 10

t = 20

t = 30

t = 40

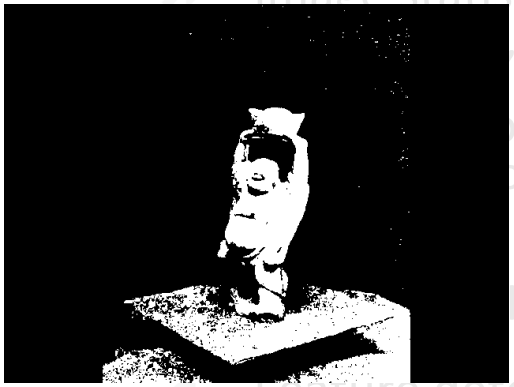
- Creation/Deletion of images
- Data structures
  - static/dynamic types
- Image statistics
- convex hull, line/ellipse fitting
- Motion/Object tracking
- Feature detection
  - edges, corners
- Kalman estimation
- Morphology
  - Erosion, Dillatation, Opening, Closing
- Drawing primitives

Excerpt from the range of functions

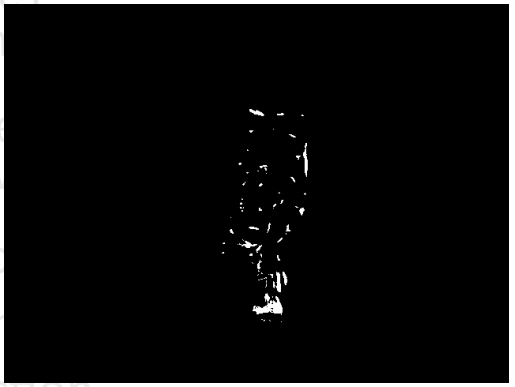
- Linear algebra
- Histogram functions
- Thresholding (global, local)

C++: void **adaptiveThreshold**(InputArray **src**, OutputArray **dst**, double **maxValue**, int **adaptiveMethod**, int **thresholdType**, int **blockSize**, double **C**)

Original



Global threshold



Local threshold

Excerpt from the range of functions

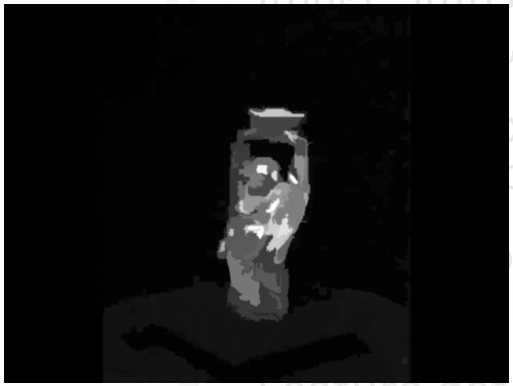
OpenCV 3.0

C++: bool StaticSaliencySpectralResidual::**computeSaliency**(const InputArray **image**, OutputArray **saliencyMap**)

Hou, Xiaodi, and Liqing Zhang. "Saliency detection: A spectral residual approach." Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on. IEEE, 2007.

- Thresholding

Original



Saliency based [1]

[1] Ming-Ming Cheng, Guo-Xin Zhang, Niloy J. Mitra, Xiaolei Huang, and Shi-Min Hu. Global Contrast based Salient Region Detection. In *IEEE Computer Vision and Pattern Recognition (CVPR)*, pages 409–416, Colorado Springs, USA, June 2011. IEEE Computer Society.

Excerpt from the range of functions

C++: bool **findChessboardCorners**(InputArray **image**, Size **patternSize**, OutputArray **corners**, int **flags**=CALIB_CB_ADAPTIVE_THRESH+CALIB_CB_NORMALIZE_IMAGE)

C++: double **calibrateCamera**(InputArrayOfArrays **objectPoints**, InputArrayOfArrays **imagePoints**, Size **imageSize**, InputOutputArray **cameraMatrix**, InputOutputArray **distCoeffs**, OutputArrayOfArrays **rvecs**, OutputArrayOfArrays **tvecs**, int **flags**=0, TermCriteria **criteria**=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON))

Chessboard pattern for intrinsic camera calibration

$$A = \begin{bmatrix} \alpha_x & \gamma & u_0 \\ 0 & \alpha_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Camera calibration (intrinsic)

Original

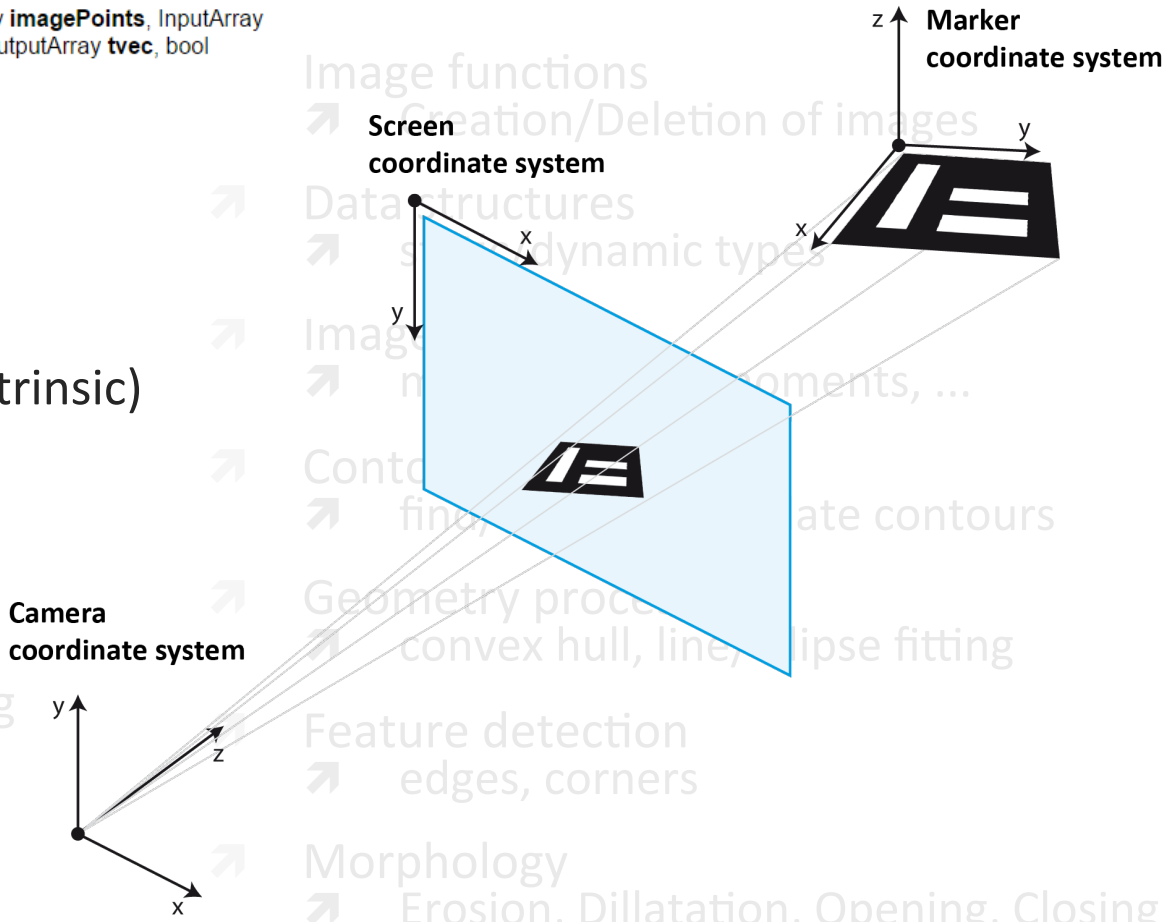
Straightened out

Extracted

Excerpt from the range of functions

C++: bool **solvePnP**(InputArray **objectPoints**, InputArray **imagePoints**, InputArray **cameraMatrix**, InputArray **distCoeffs**, OutputArray **rvec**, OutputArray **tvec**, bool **useExtrinsicGuess**=false, int **flags**=ITERATIVE)

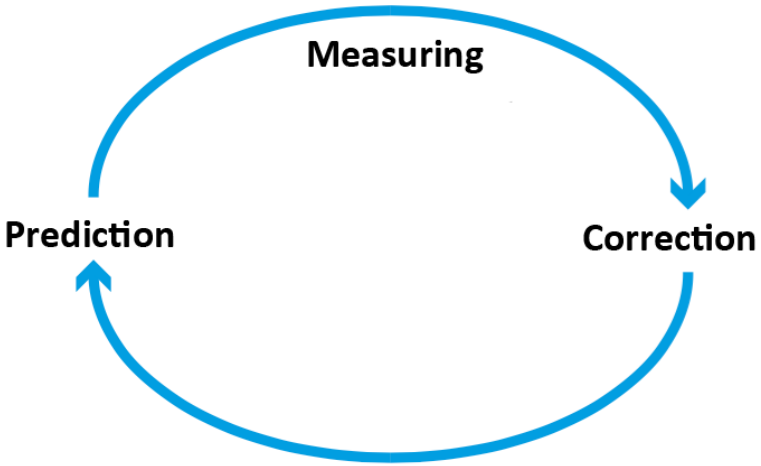
- Histogram functions
- Thresholding
- Camera calibration (extrinsic)
- 3D reconstruction
- Optical flow
- Motion/Object tracking
- Kalman estimation
- Drawing primitives



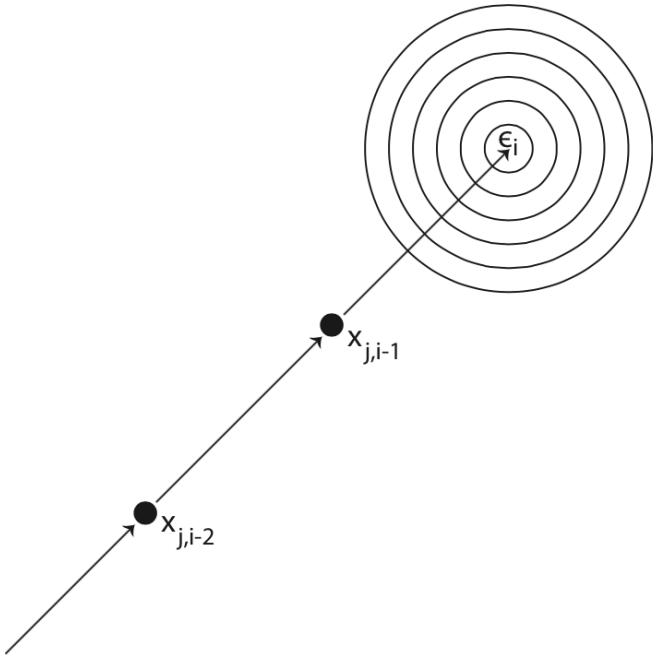
With known camera intrinsics, camera pose estimation with Perspective-n-Point Algorithm (PnP) using OpenCV

Excerpt from the range of functions

C++: `KalmanFilter::KalmanFilter(int dynamParams, int measureParams, int controlParams=0, int type=CV_32F)`



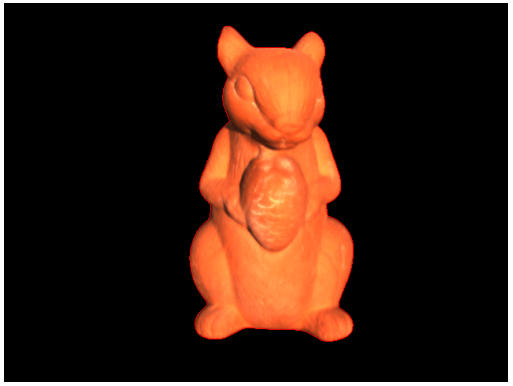
Basic approach of kalman filter

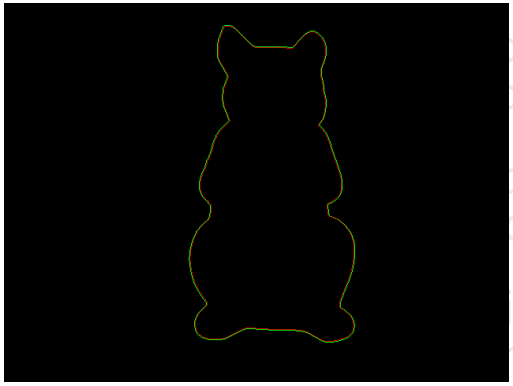


Schematic illustration of the movement model

- Kalman estimation

Excerpt from the range of functions

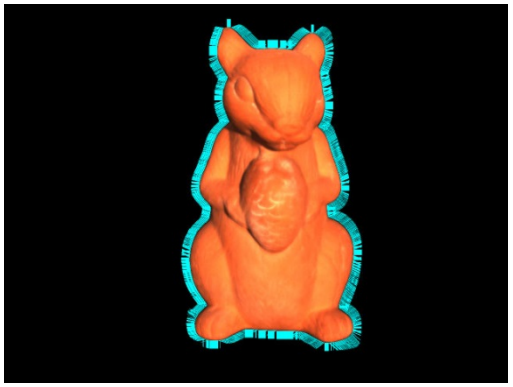


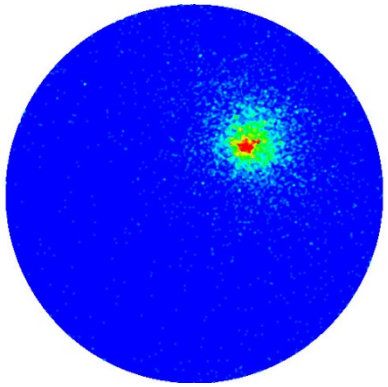


C++: void **findContours**(InputOutputArray **image**, OutputArrayOfArrays **contours**, OutputArray **hierarchy**, int **mode**, int **method**, Point **offset**=Point())

C++: void **findContours**(InputOutputArray **image**, OutputArrayOfArrays **contours**, int **mode**, int **method**, Point **offset**=Point())

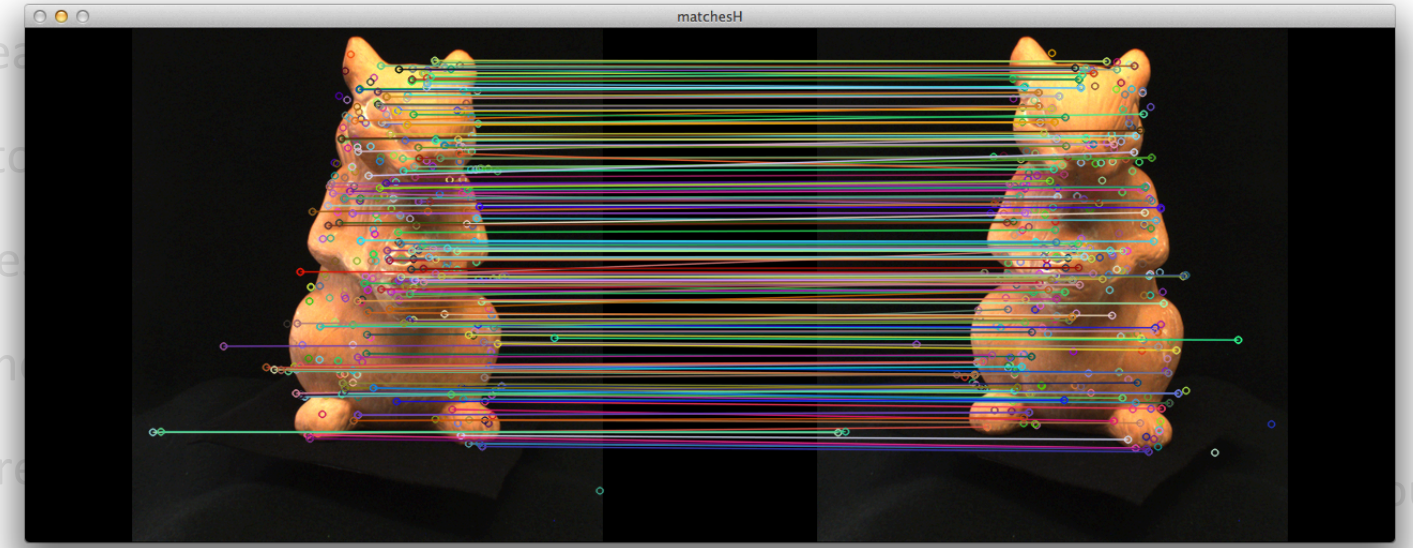
- Contour processing
  - find/show/manipulate contours





Light estimation using contours, lambert surface model and RANSAC (NOT included in OpenCV, but easy to implement)

Excerpt from the range of functions



Detects corners using the FAST algorithm

C++: void **FAST**(InputArray **image**, vector<KeyPoint>& **keypoints**, int **threshold**, bool **nonmaxSuppression**=true)

- Feature detection
  - edges, corners
- Morphology
  - Erosion, Dillatation, Opening, Closing

OpenCV Modules

1. `opencv_core` Core functionality (LA, DFT, XML, I/O)
2. `opencv_imgproc` Image processing
3. `opencv_highgui` GUI and video I/O
4. `opencv_ml` Machine Learning (SVM, NN, ...)
5. `opencv_features2d` 2D feature detectors (SURF, FAST, ...)
6. `opencv_video` Motion analysis & object tracking
7. `opencv_objdetect` Object detection (Haar, HOG)
8. `opencv_calib3d` Camera calibration, stereo correspondence
9. `opencv_flann` Fast Library Approximate Nearest Neighbours
10. `opencv_contrib` Chamfer Matching
11. `opencv_legacy` Compatibility to OpenCV verrsions < 2.2
12. `opencv_gpu` OpenCV acceleration (CUDA)
13. Many more (`nonfree`, `stitching`, `ocl`, `superres`, `viz`,...)

Module `core`

- Arithmetic and logical operations on images and matrices
- Linear algebra, Vector/Matrix operations
- Discrete Fourier and Cosine Transformation (DFT, DCT)
- Data structures
- Drawing functions
- XML Input/Output

Module `highgui`

- „Smart“ windows
- Input, Output and Displaying of images
- Event processing (Keyboard events and timeouts)
- Trackbars (no further dialogs)
- Mouse callback functions
- Input/Output of video streams

(Uncalibrated) Photometric-Stereo

- Use the laptop screen and a webcam to create your own little 3D scanner

Youtube-Demo

Schindler, G. (2008). Photometric Stereo via Computer Screen Lighting for Real-time Surface Reconstruction. International Symposium on 3D Data Processing, Visualization and Transmission, 1–6.

Prerequisites

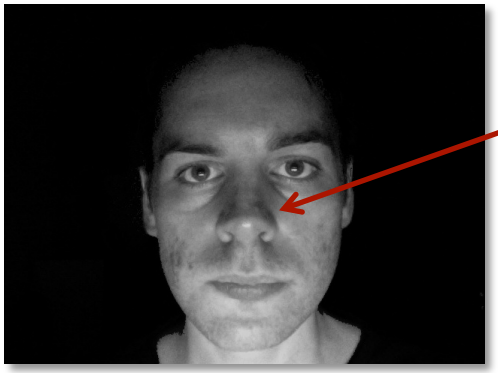
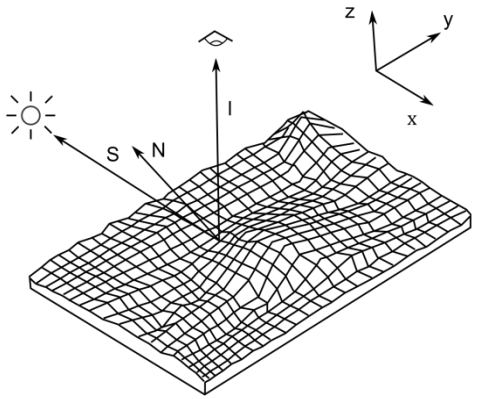


Image intensity per pixel:

$$I(x,y) = c \cdot q(x,y,z) \cdot R(n,s)$$

Lambert reflection modell (diffuse reflection):

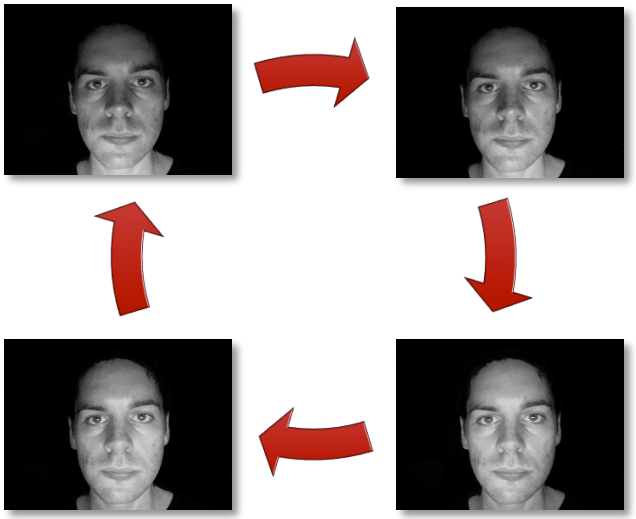
$$R(n,s) = \rho \cdot cos\varphi = \rho \cdot \langle n,s \rangle$$



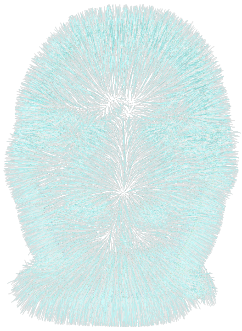
Set camera sensitivity = brightness light source = (global) albedo = 1 gives:

$$I = cos\varphi = \langle n,s \rangle$$

Prerequisites



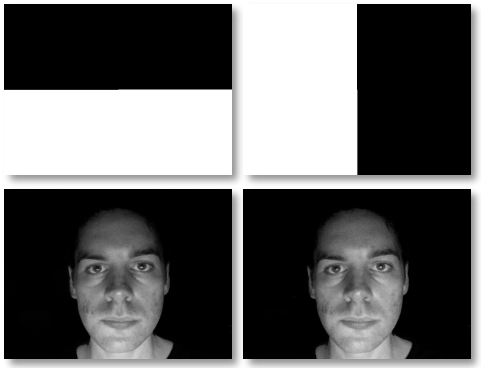
1. Acquire (at least 3) different illuminated images



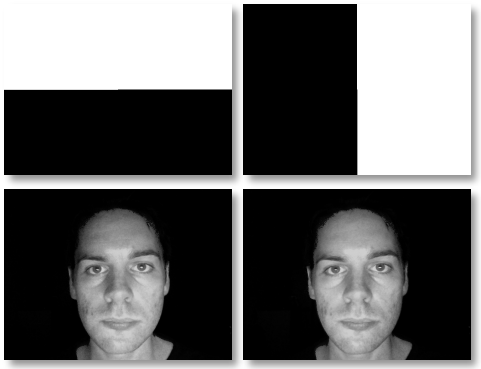
2. Estimate surface normals

3. 2D integration of gradient field

Creating images

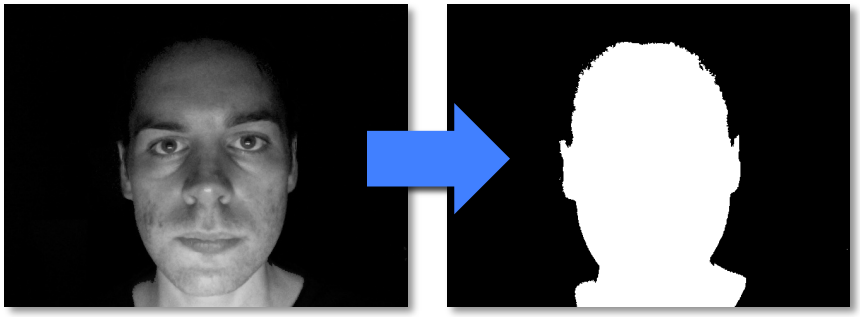


```
cv::Mat lightPattern(int width, int height, int j, int N) {
    cv::Mat img(height, width, CV_8UC1, cv::Scalar::all(0));
    for (int y = -(height/2); y < height/2; ++y) {
        for (int x = -(width/2); x < width/2; ++x) {
            if (sgn(x*cv::cos(2*CV_PI*j/N)+y*cv::sin(2*CV_PI*j/N)) == 1) {
                img.at<uchar>(y+height/2,x+width/2) = 255;
            }
        }
    }
    return img;
}
```



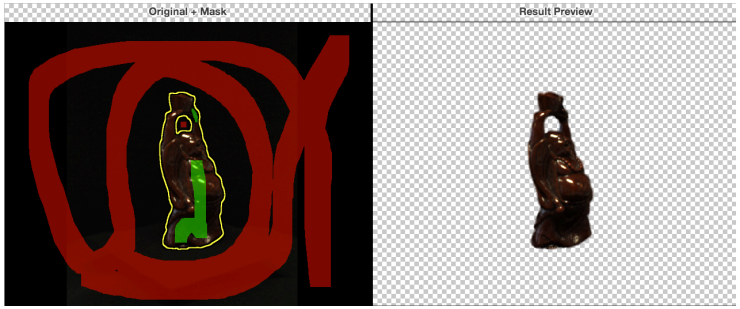
```
/* open capture device (e.g. webcam) */
std::vector<cv::Mat> camImages;
auto captureDevice = cv::VideoCapture(CV_CAP_ANY);
if (captureDevice.isOpened()) {
    /* capture images from webcam while showing pattern image */
    cv::namedWindow("camera", CV_WINDOW_FULLSCREEN);
    for (int idx = 1; idx <= numPics; ++idx) {
        cv::imshow("camera", lightPattern(width, height, idx, numPics));
        cv::Mat frame;
        captureDevice >> frame;
        cv::cvtColor(frame, frame, CV_RGB2GRAY);
        camImages.push_back(frame);
    }
}
```

Threshold using grabcut

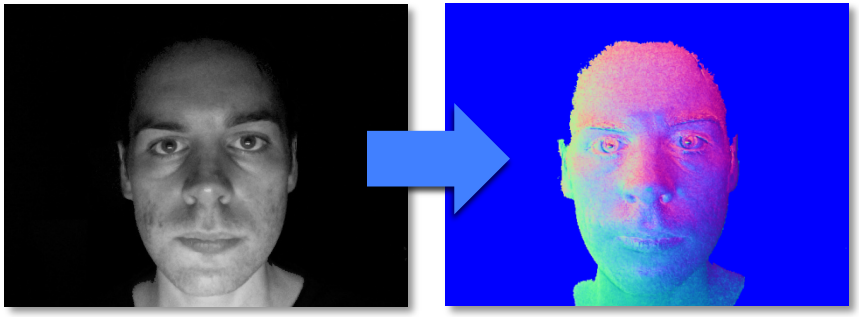


C++: void **grabCut**(InputArray **img**, InputOutputArray **mask**, Rect **rect**, InputOutputArray **bgdModel**, InputOutputArray **fgdModel**, int **iterCount**, int **mode**=GC_EVAL)

```
cv::Mat imageMask(cv::Mat image) {
    int quarter = image.cols/4.0;
    int eighth = image.rows/8.0;
    cv::Mat result, bgModel, fgModel;
    cv::Rect area(quarter, eighth, 3*quarter, 7*eighth);

    /* grabcut expects rgb images */
    cv::cvtColor(image, image, CV_GRAY2BGR);
    cv::grabCut(image, result, area, bgModel, fgModel, 1, cv::GC_INIT_WITH_RECT);
    cv::compare(result, cv::GC_PR_FGD, result, cv::CMP_EQ);
    return result;
}
```



Recovering surface normals



C++: *static* void SVD::**compute**(InputArray **src**, OutputArray **w**, OutputArray **u**, OutputArray **vt**, int **flags**=0)

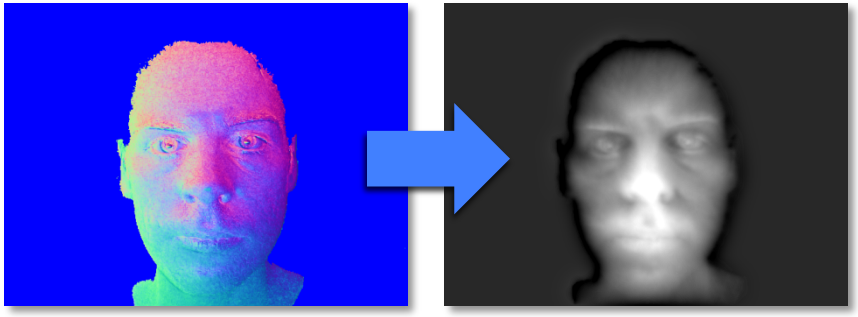
C++: *static* void SVD::**compute**(InputArray **src**, OutputArray **w**, int **flags**=0)

```
cv::Mat computeNormals(const std::vector<cv::Mat>& camImages, const cv::Mat& Mask) {
    const cv::Size imgSize = camImages[0].size();
    const int numImgs = camImages.size();
    cv::Mat A(imgSize, numImgs, CV_32FC1, cv::Scalar::all(0));
    /* populate A */
    ...

    /* speeding up computation, SVD from A^TA instead of AA^T */
    cv::Mat U,S,Vt;
    cv::SVD::compute(A.t(), S, U, Vt, cv::SVD::MODIFY_A);
    cv::Mat EV = Vt.t();

    cv::Mat Normals(imgSize, CV_8UC3, cv::Scalar::all(0));
    for (std::size_t i = 0; i < imgSize.height; ++i) {
        for (std::size_t j = 0; j < imgSize.width; ++j) {
            /* V contains the eigenvectors of A^TA, which are as well the z,x,y
             * components of the surface normals for each pixel */
            ...
        }
    }
    return Normals;
}
```

2.2 Recovering Surface Normals

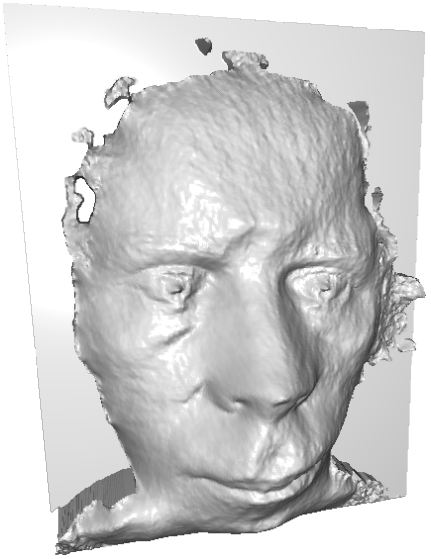
Following a standard approach [2], we treat each image I_j (of width W and height H) as a 1-dimensional column vector of intensity values of size $WH \times 1$ and place these N column vectors side by side into a matrix A of size $WH \times N$. It is well established [20] that the images live in a 3-dimensional subspace defined by the eigenvectors corresponding to the three largest eigenvalues of the matrix AA^T. These eigenvectors, which can be extracted using SVD, correspond to the z,x, and y components of the surface normal at every pixel in the image (see Figure 3).

Surface reconstruction

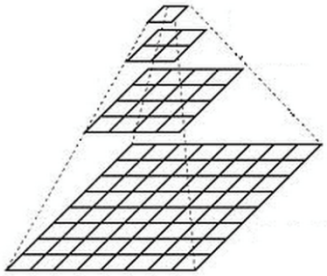


C++: void **pyrDown**(InputArray **src**, OutputArray **dst**, const Size& **dstsize**=Size(), int **borderType**=BORDER_DEFAULT)

C++: void **pyrUp**(InputArray **src**, OutputArray **dst**, const Size& **dstsize**=Size(), int **borderType**=BORDER_DEFAULT)



```
cv::Mat localHeightfield(cv::Mat Normals) {

    const int pyramidLevels = 4;
    const int iterations = 700;

    /* building image pyramid */
    std::vector<cv::Mat> pyrNormals;
    pyrNormals.push_back(Normals);
    for (int i = 0; i < pyramidLevels; ++i) {
        cv::pyrDown(Normals, Normals);
        pyrNormals.push_back(Normals.clone());
    }

    /* updating depth map along pyramid levels, starting with smallest level at top */
    cv::Mat Z(pyrNormals[pyramidLevels-1].rows, pyrNormals[pyramidLevels-1].cols, CV_32FC1, cv::Scalar::all(0));
    for (int i = pyramidLevels-1; i > 0; --i) {
        updateHeights(pyrNormals[i], Z, iterations);
        cv::pyrUp(Z, Z);
    }

    return Z;
}
```



Finish

Thank you for your time

Full source code: https://github.com/NewProggie/Uncalibrated-Photometric-Stereo