universität innsbruck

Lecture 2. Image formation. OpenCV

703142. Computer Vision

Assoz.Prof. Antonio Rodríguez-Sánchez, PhD.

Image formation

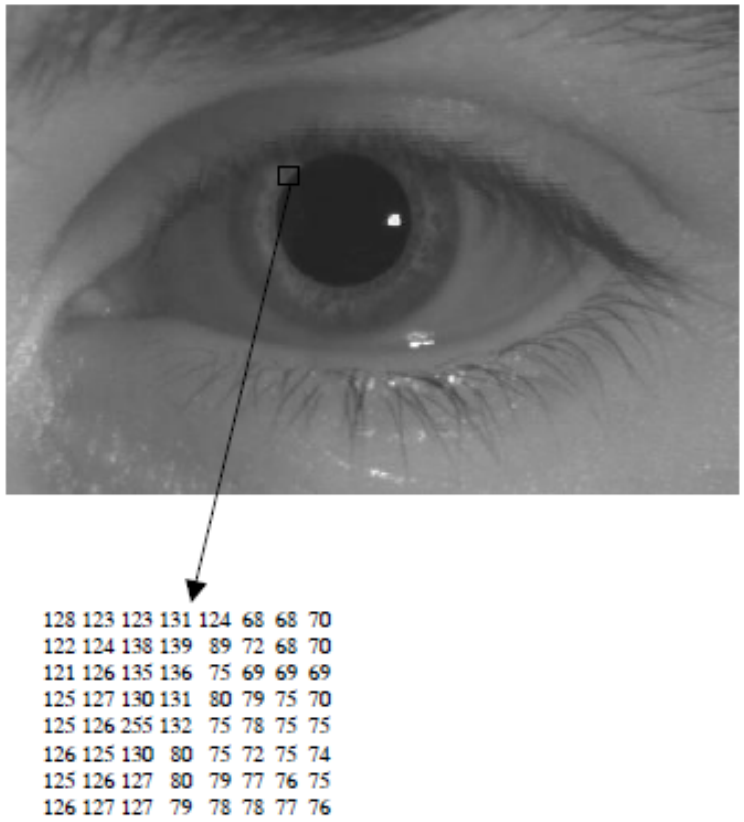
- Introduction
- Optics
- Radiometry
- Geometric image formation
- Image acquisition

Image formation

- Introduction
- Optics
- Radiometry
- Geometric image formation
- Image acquisition

Introduction

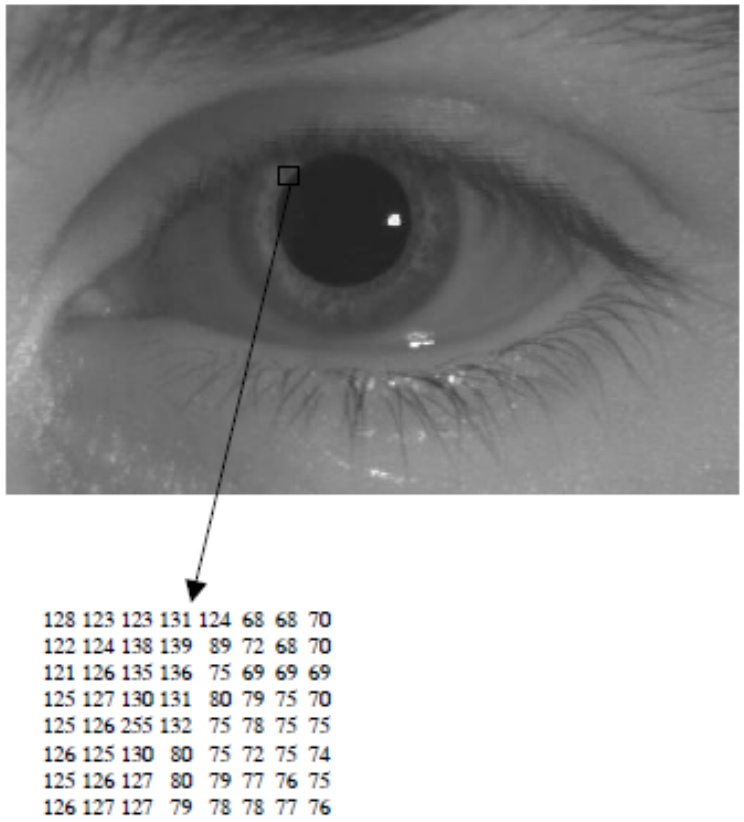
- Motivation
  - To fully understand information recovery from images
  - It is necessary to understand how images are formed
- Key questions
  - What determines where a 3D scene point will appear in an image?
  - With what intensity will the point be imaged?



Source: Richard Wildes slides

Introduction

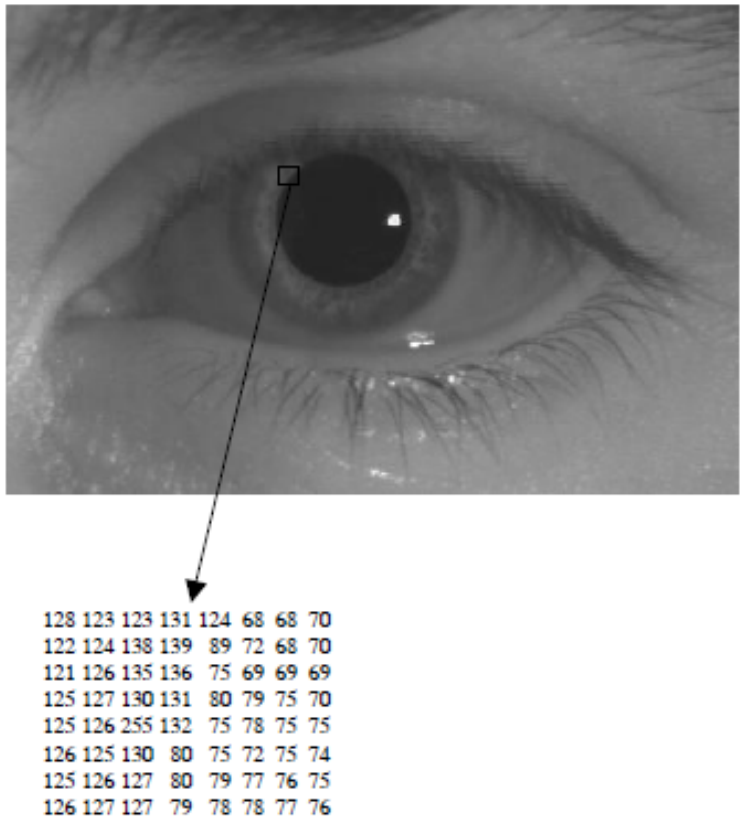
- Motivation
  - To fully understand information recovery from images
  - It is necessary to understand how images are formed
- Key questions
  - What determines where a 3D scene point will appear in an image?
  - With what intensity will the point be imaged?



Source: Richard Wildes slides

Introduction

- Major image types used in computer vision
  - Intensity images
  - Range images
- Any digital image is just a numerical array
  - Exact relationship of image of physical world depends on the image formation process.
  - Information in the images is implicit and must be recovered through processing.



Source: Richard Wildes slides

Intensity images

- **Optical parameters of lens:** Characterize sensor optics
  - lens type
  - focal length
  - field of view
  - angular apertures

Source: Richard Wildes slides

Intensity images

- **Photometric parameters:** Models of light energy reaching sensor following reflection from surfaces in scene
  - type, intensity and direction of illumination
  - reflectance properties of visible surfaces
  - effects of sensor structure on light reaching photoreceptors

Source: Richard Wildes slides

Intensity images

- **Geometric parameters:** Determine image position at which 3D points are imaged
  - type of projection
  - position and orientation of camera in space
  - geometric distortions from imaging process

Source: Richard Wildes slides

Intensity images

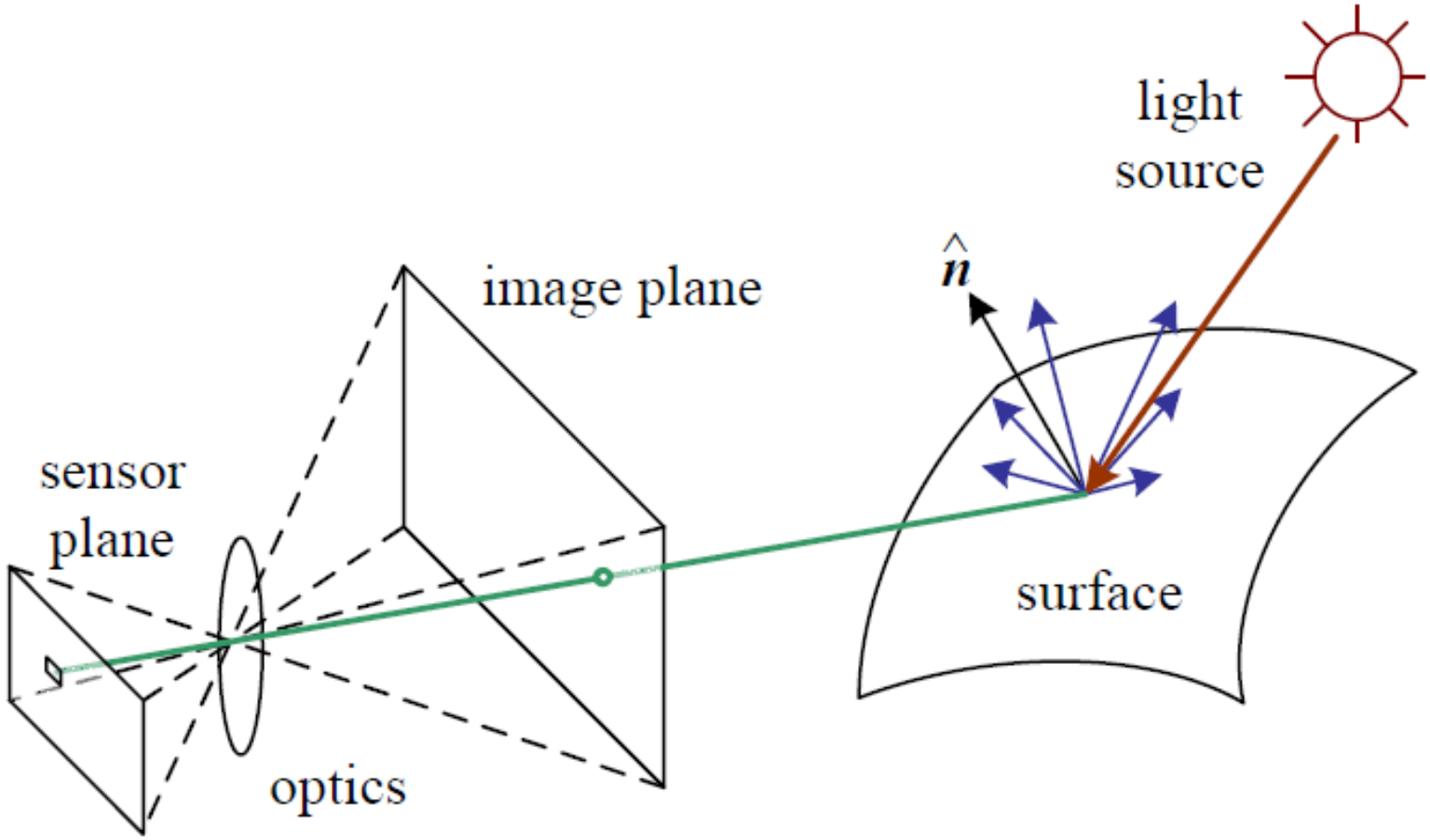
- **Parameters specific to digital imaging**: Photoreceptors of viewing camera
  - physical properties of photosensitive matrix
  - discrete nature of photoreceptors
  - quantization of intensity

Source: Richard Wildes slides

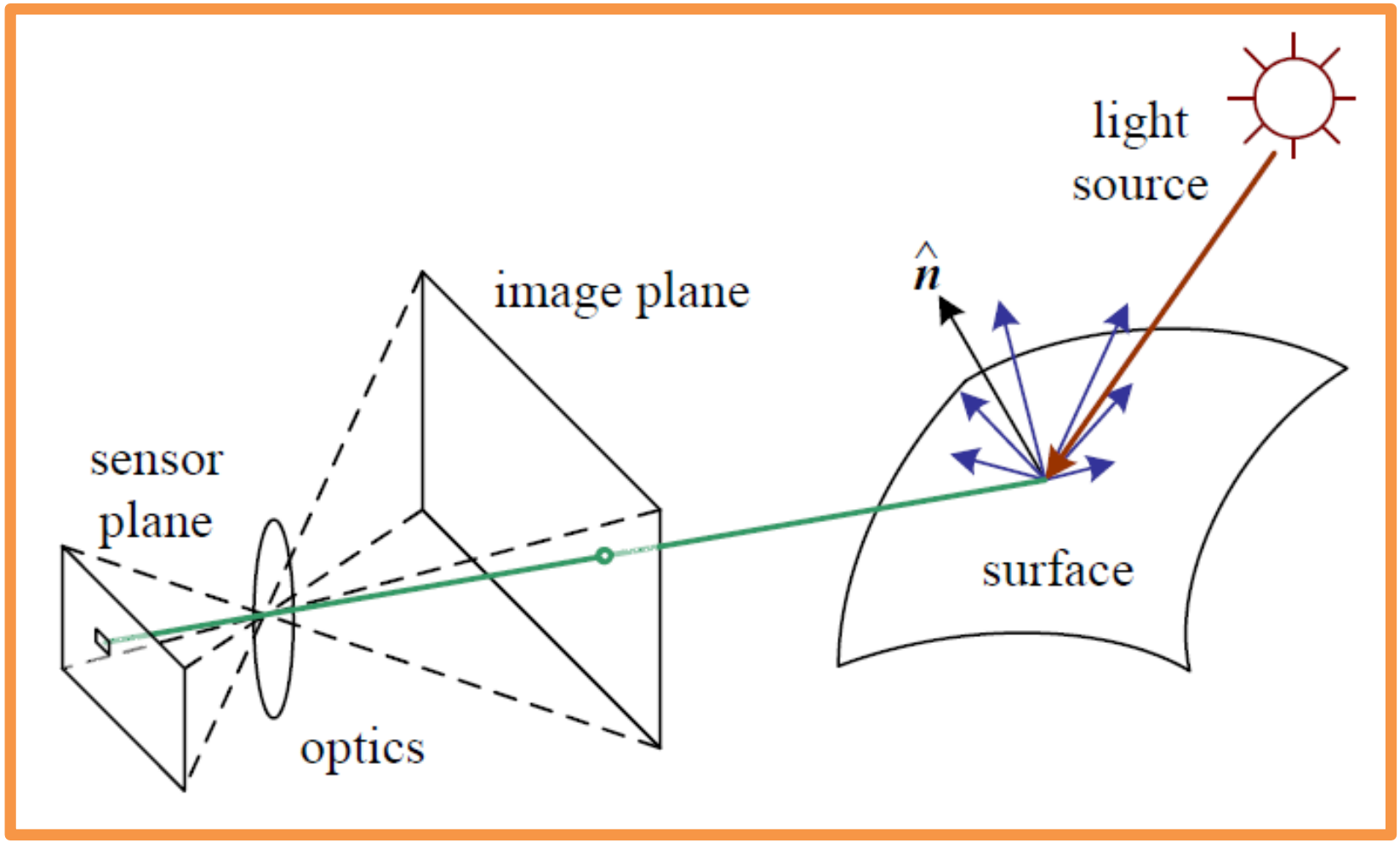
Image formation

- Introduction
- Optics
- Radiometry
- Geometric image formation
- Image acquisition

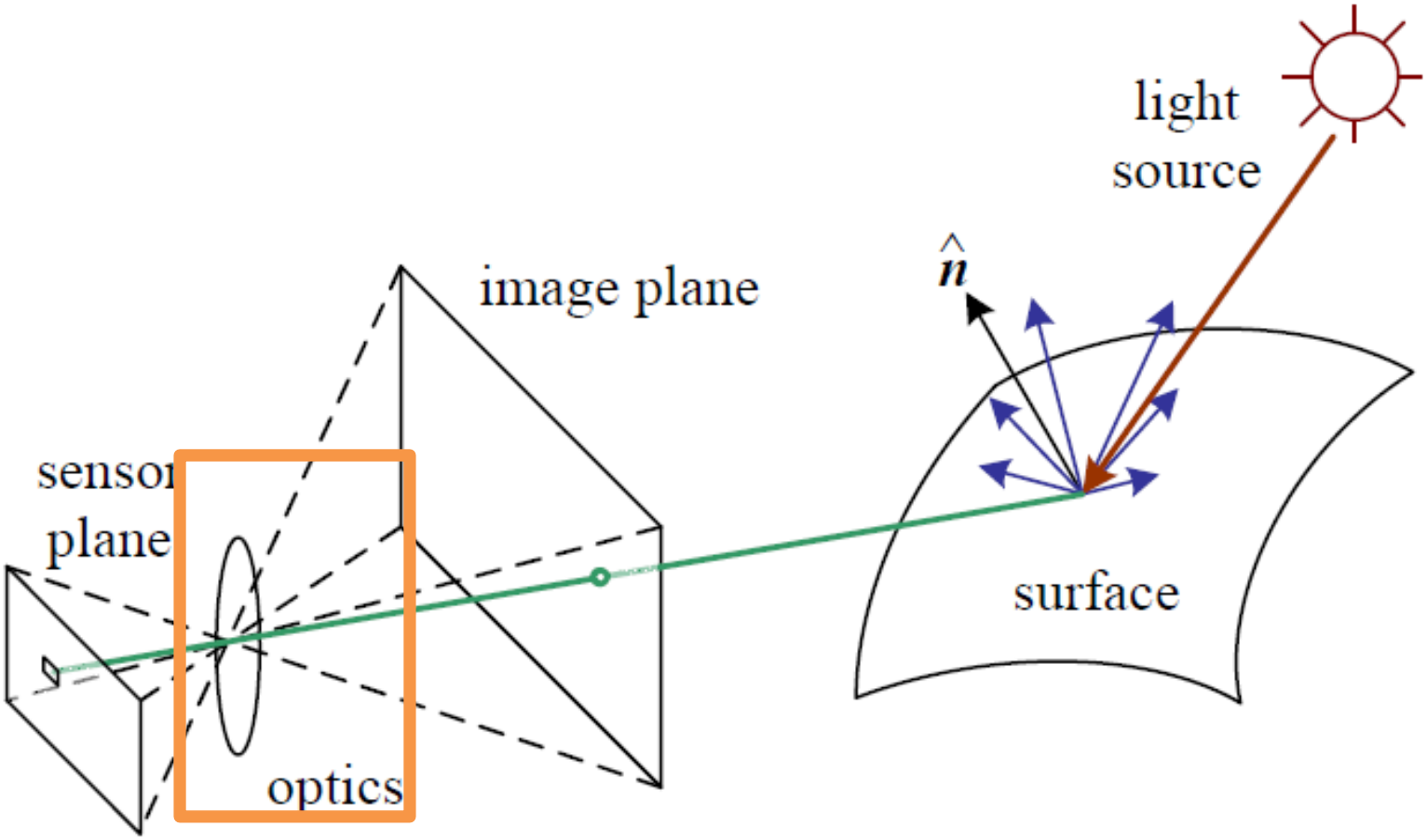
Optics and radiometry



Optics and **radiometry**

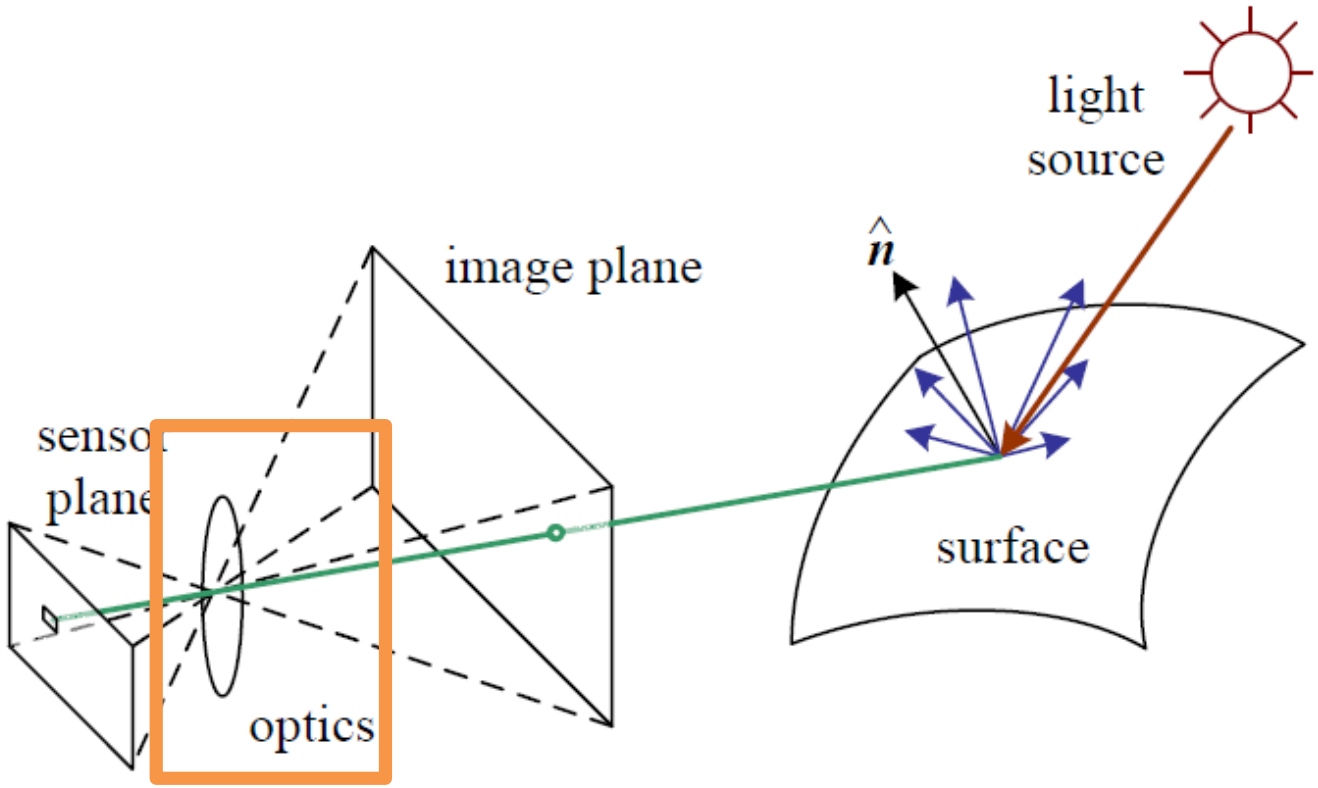


Optics and radiometry



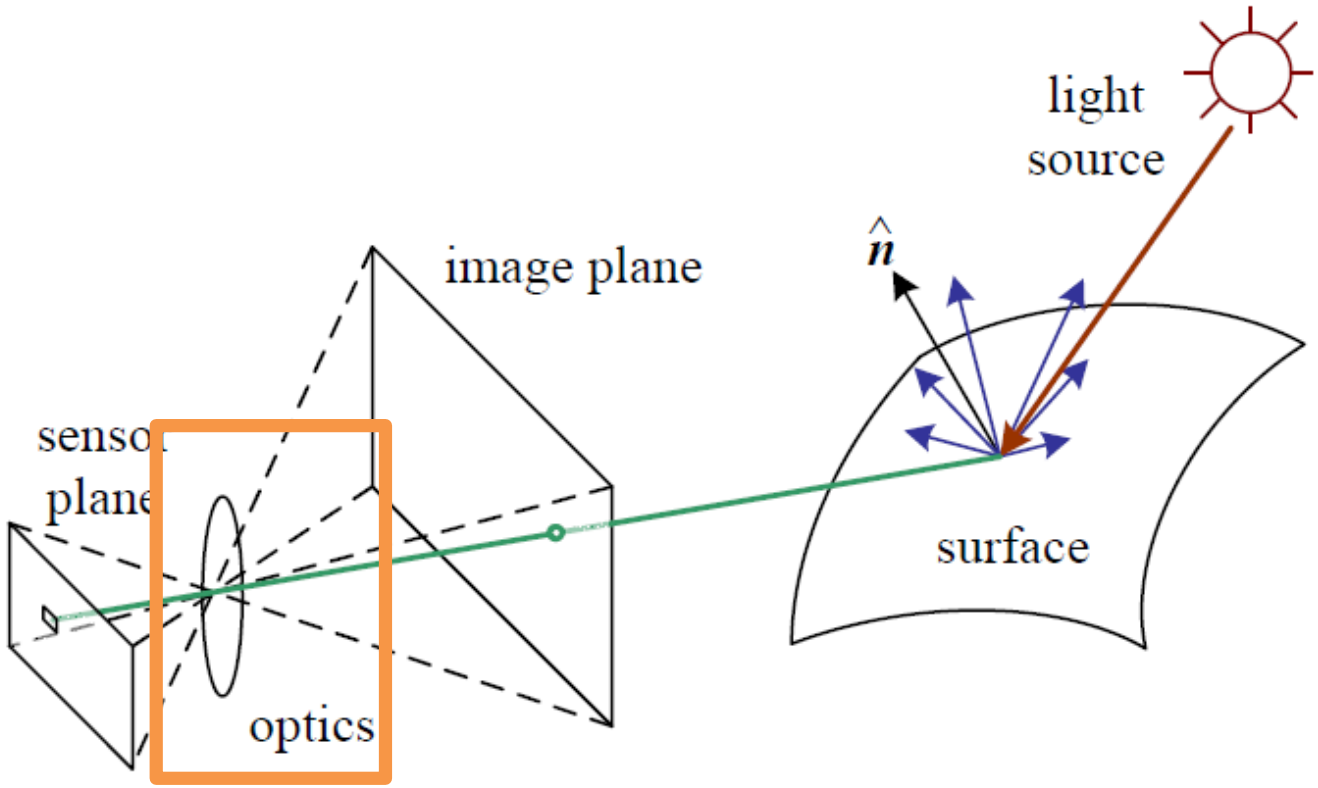
Optics

- Image formation begins when light rays enter an aperture to impinge on an image surface.



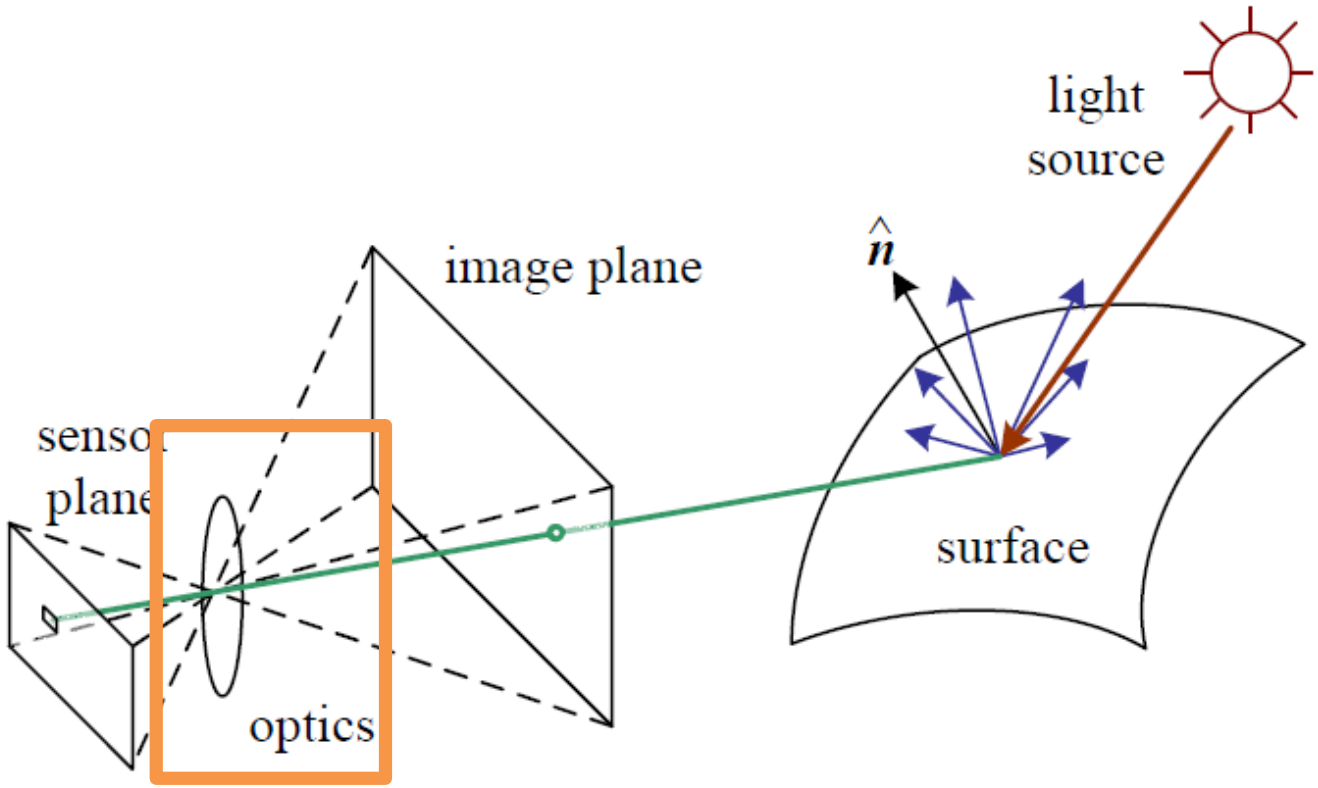
Optics

- Typically, these rays are reflections of light rays off surfaces in the scene,



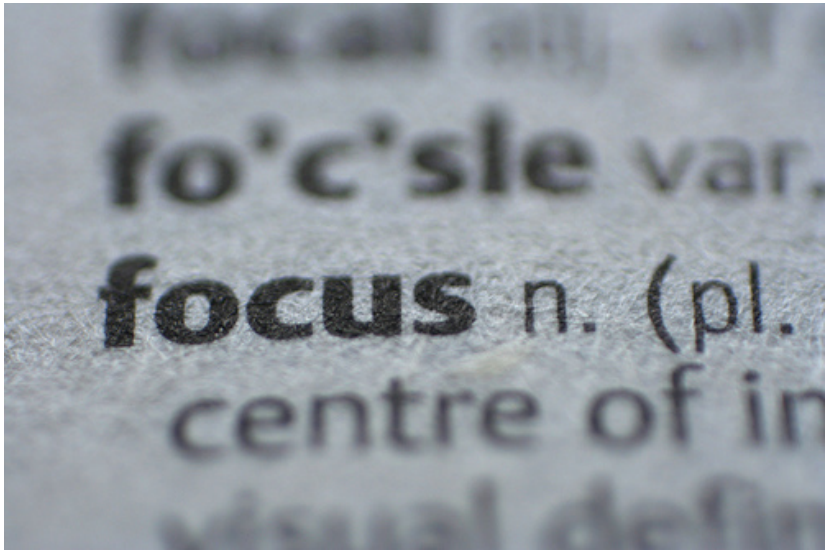
Optics

- But can also be direct images of light sources in the scene.



Optics

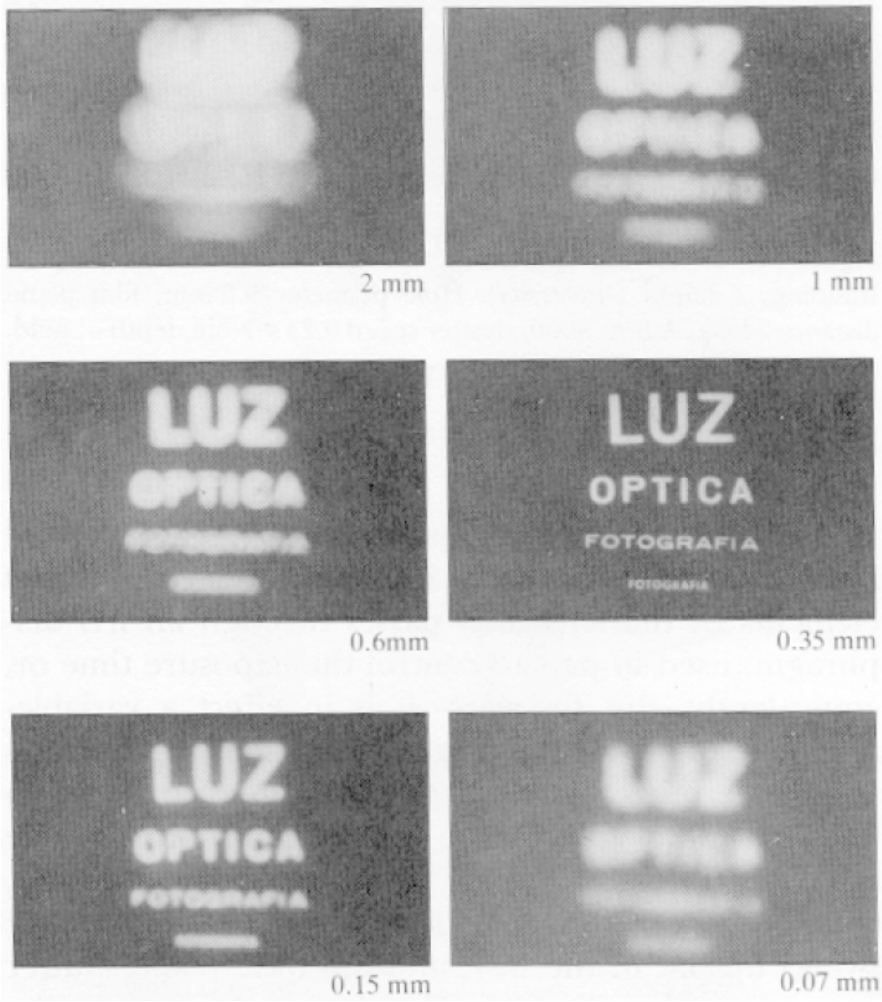
- Focus
  - Any single point in the world reflect light in many directions.
  - Many rays reflected by same point may enter the camera.
  - To obtain sharp images, we want all rays from a single scene point *P* to converge on a single image point *p*.



Modified from Richard Wildes slides

Optics

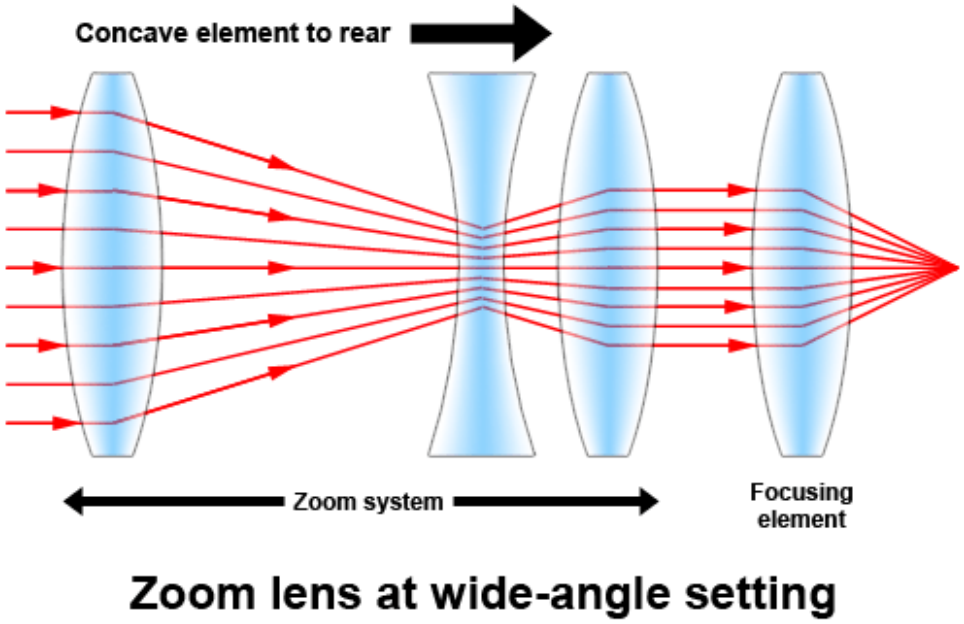
- Achieving focus
  - Pinhole camera
    - Only one ray from a given point enters the camera.
    - Sharp, undistorted images over wide range of distances.
    - Requires long exposure times.



Modified from Richard Wildes slides

Optics

- Achieving focus
  - Introduce an optical system with lenses and apertures
    - Designed to make all rays coming from the same 3D point converge to the same image point.
    - Sharp, undistorted images over a range of exposure times.



Zoom lens at wide-angle setting

Modified from Richard Wildes slides

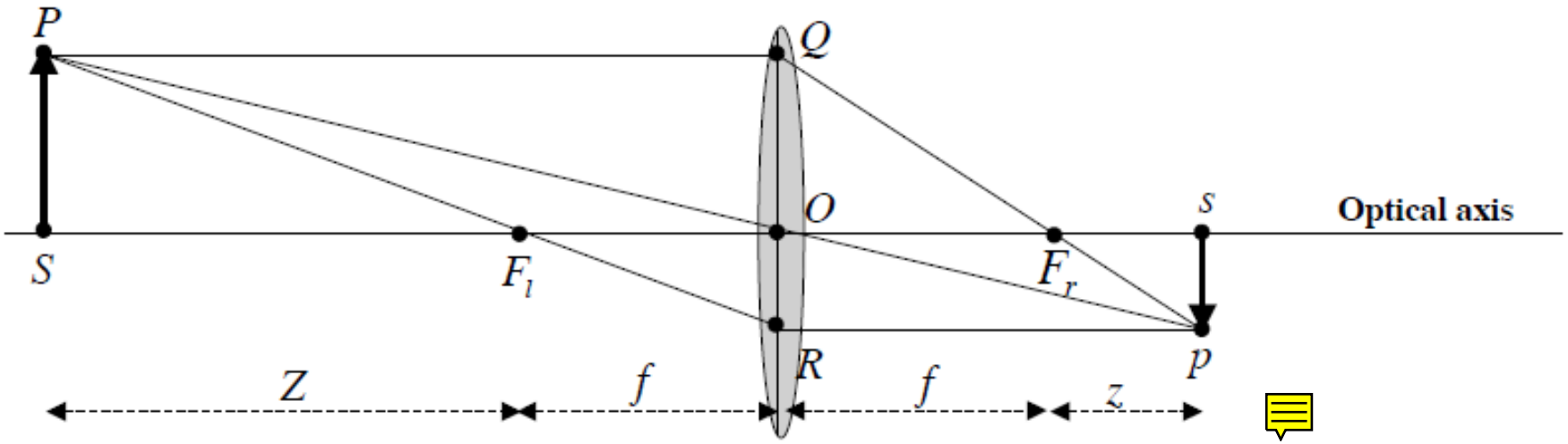
Optics

- Achieving focus
  - Introduce an optical system with lenses and apertures
    - Designed to make all rays coming from the same 3D point converge to the same image point.
    - Sharp, undistorted images over a range of exposure times.

Modified from Richard Wildes slides

Optics

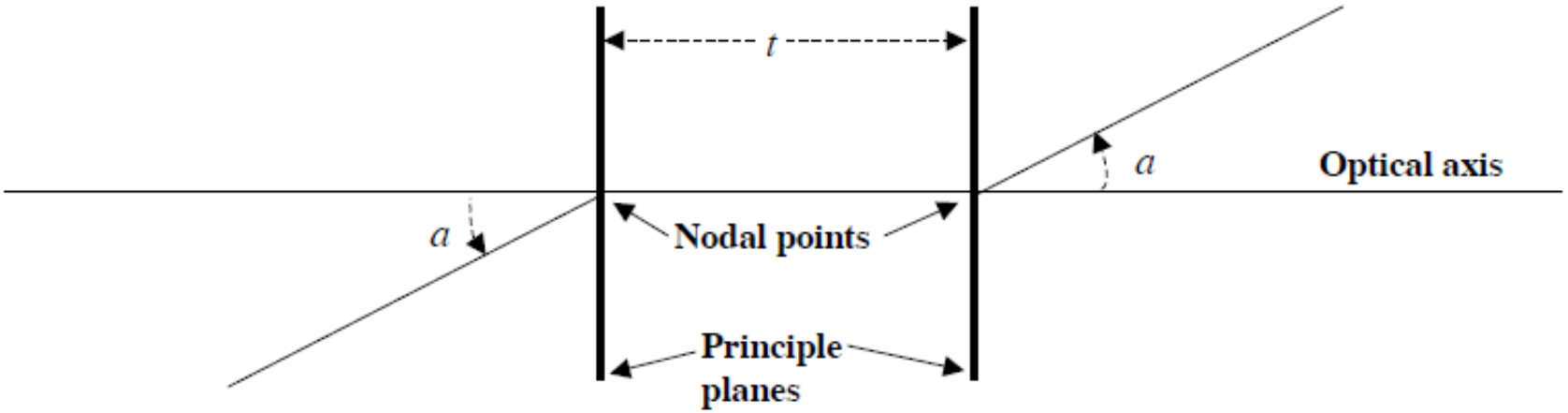
- Simplification: The Thin lens



$$\frac{1}{Z+f}+\frac{1}{z+f}=\frac{1}{f}$$

Optics

- In reality: Thick lens

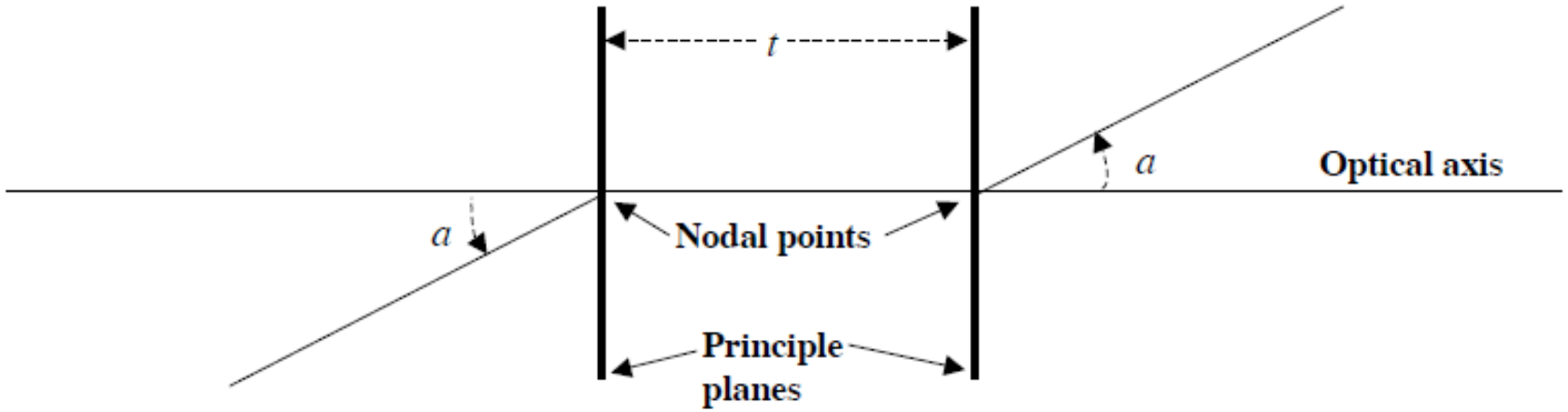


  - Any simple lens will have a number of optical defects.
  - For better imaging it is customary to combine several lenses.
  - The thick lens provides a reasonable model.

Source: Richard Wildes slides

Optics

- In reality: Thick lens

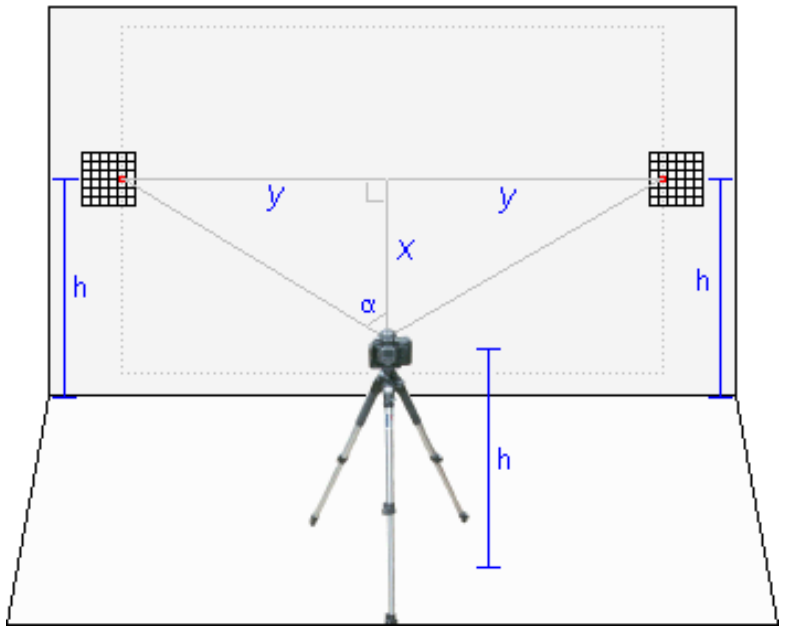


  - Two characterizing elements
    1. A pair of principle planes parallel to the common optical axis.
    2. A pair of nodal points, separated by *t* (thickness) where planes intersect the optical axis.

Source: Richard Wildes slides

Optics

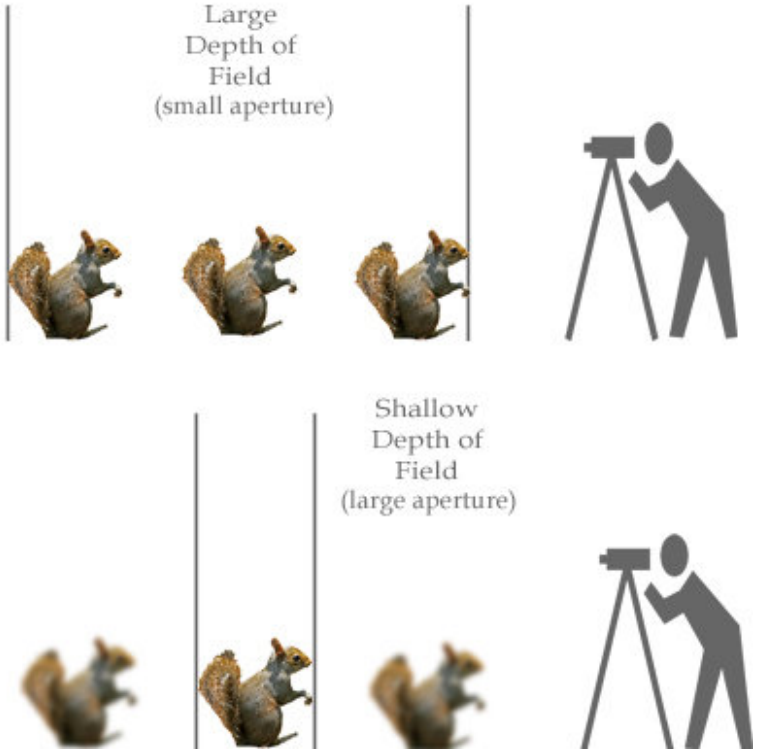
- Field of View



$$FOV = 2 * \tan^{-1} \frac{y}{x}$$

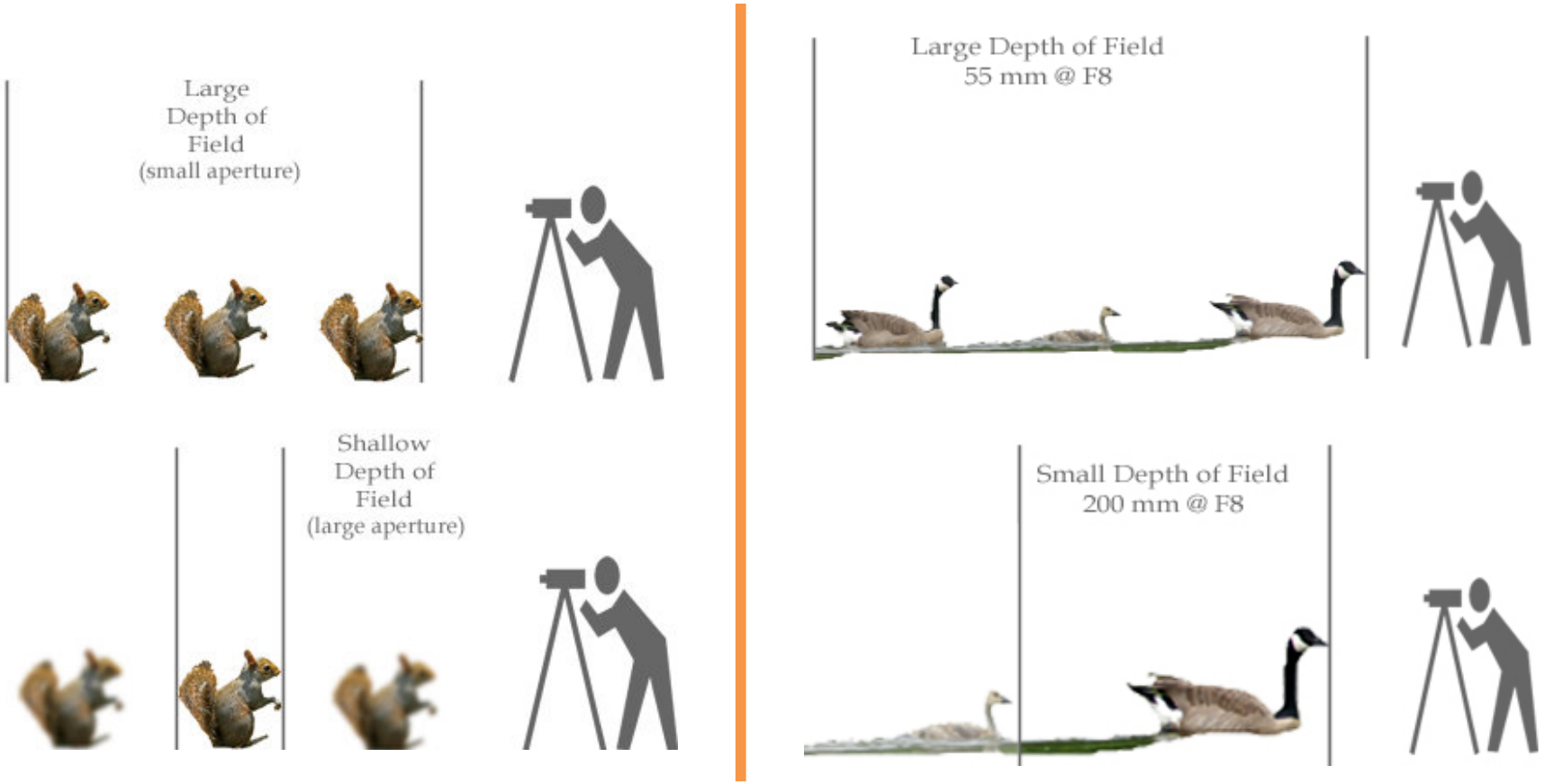
Optics

- Depth of field



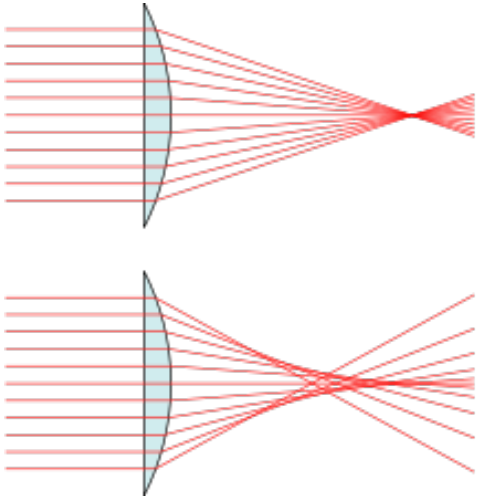
Optics

- Depth of field



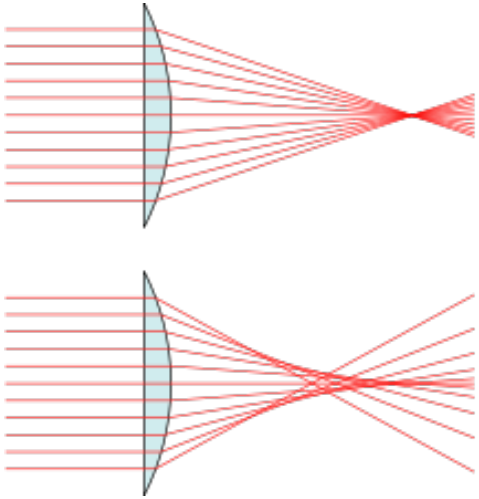
Optics

- Aberrations
  - Spherical aberration: defocusing of nonparaxial rays



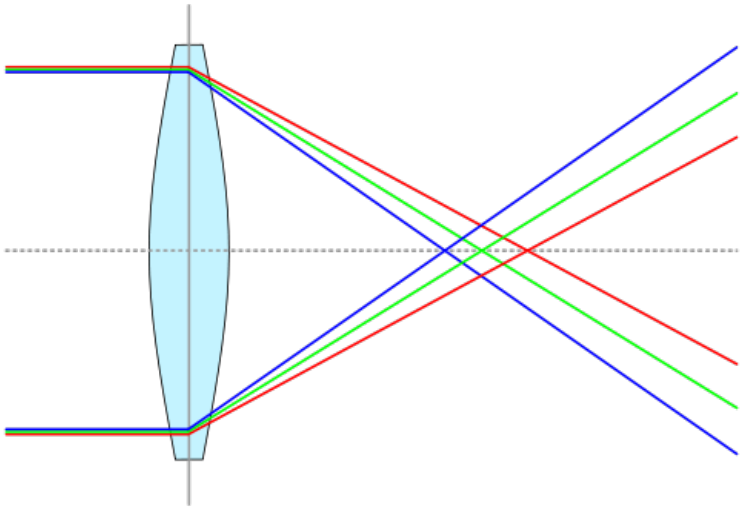
Optics

- Aberrations
  - Spherical aberration: defocusing of nonparaxial rays



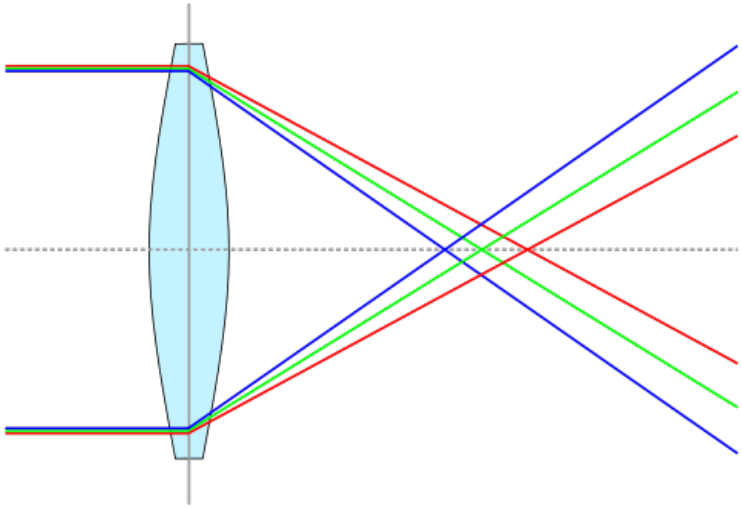
Optics

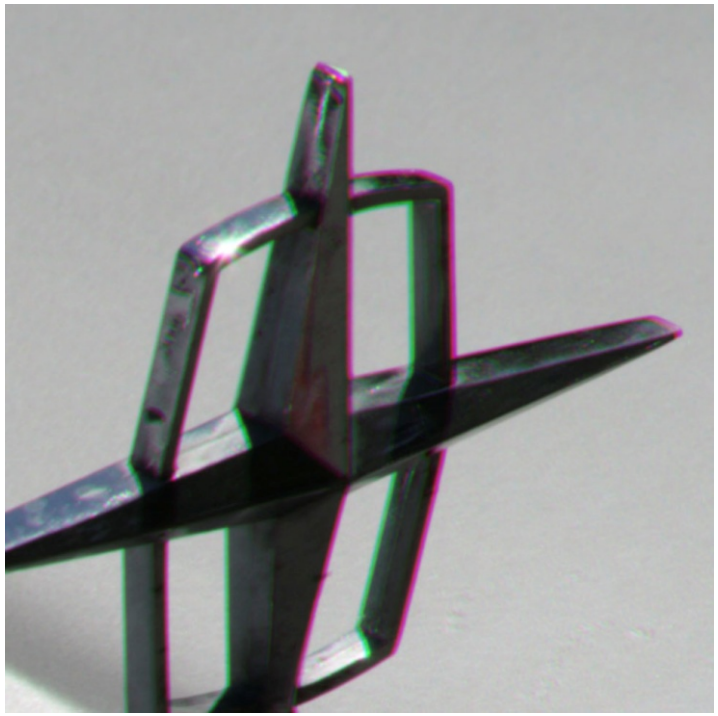
- Aberrations
  - Chromatic aberration: differential defocusing as function of wavelength of light



Optics

- Aberrations
  - Chromatic aberration: differential defocusing as function of wavelength of light





Optics

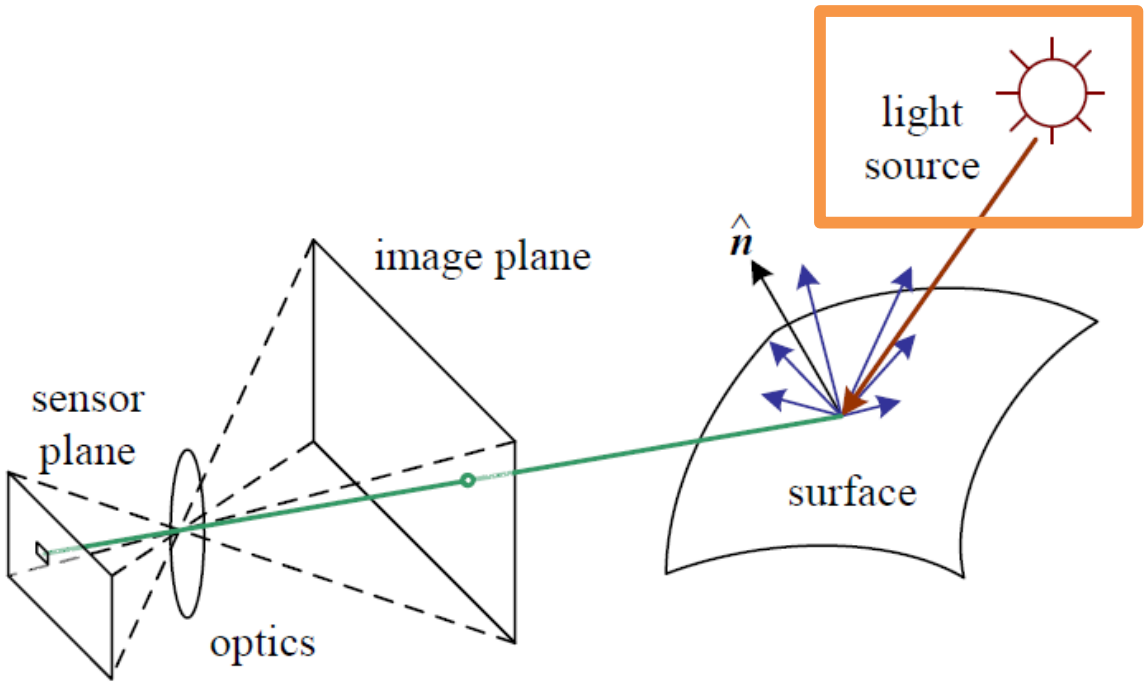
- Aberrations
  - Vigneting: loss of image intensity near periphery as complex aperture occlude light

Image formation

- Introduction
- Optics
- Radiometry
- Geometric image formation
- Image acquisition

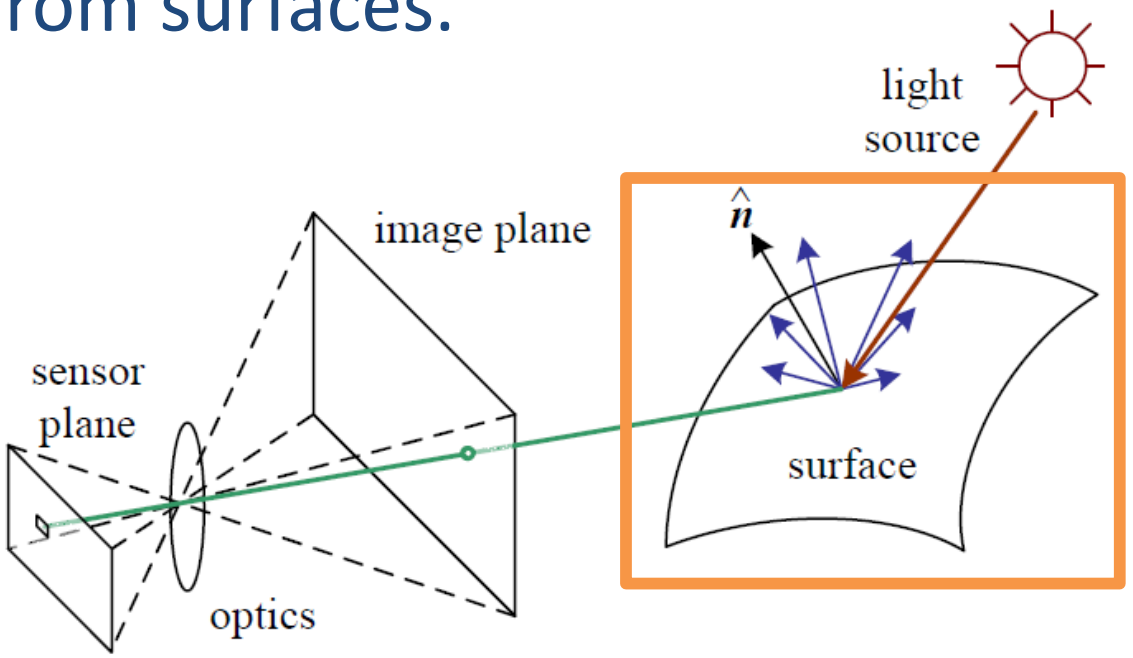
Radiometry

- **Radiometry is concerned with relations between**
  - amounts of light energy emitted from light sources.



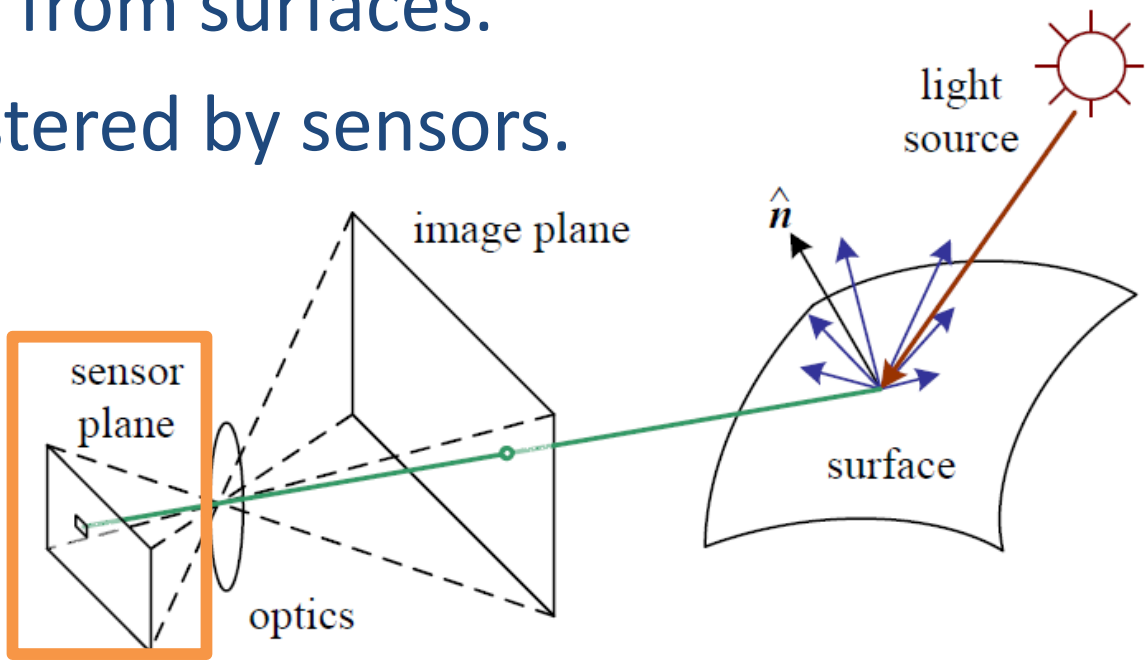
Radiometry

- **Radiometry is concerned with relations between**
  - amounts of light energy emitted from light sources.
  - reflected from surfaces.



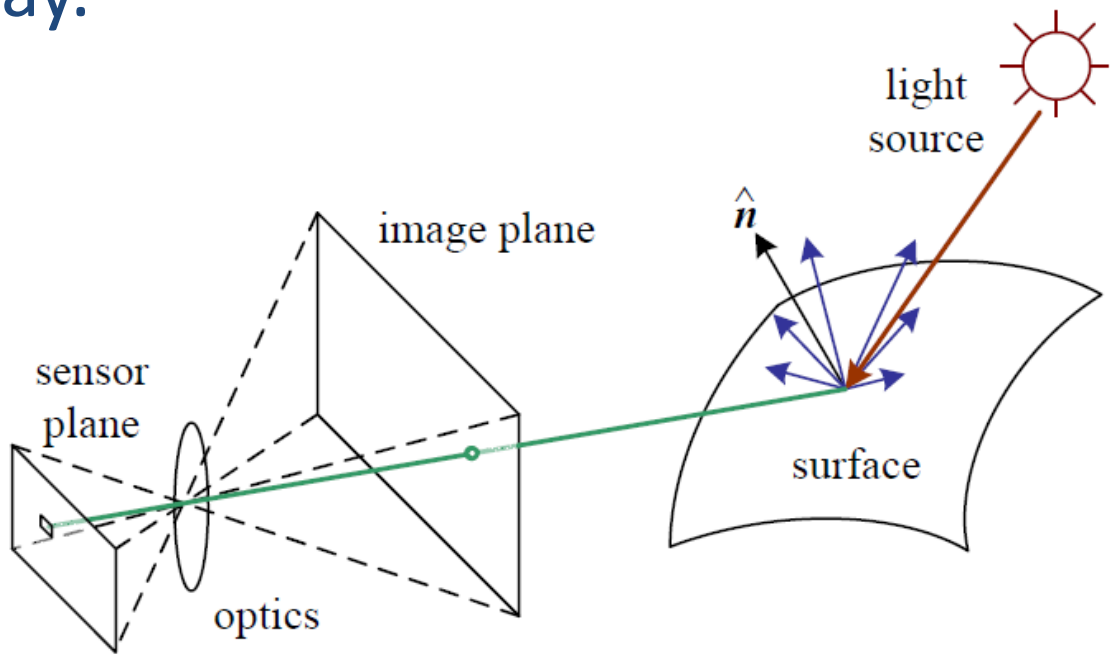
Radiometry

- **Radiometry is concerned with relations between**
  - amounts of light energy emitted from light sources.
  - reflected from surfaces.
  - and registered by sensors.



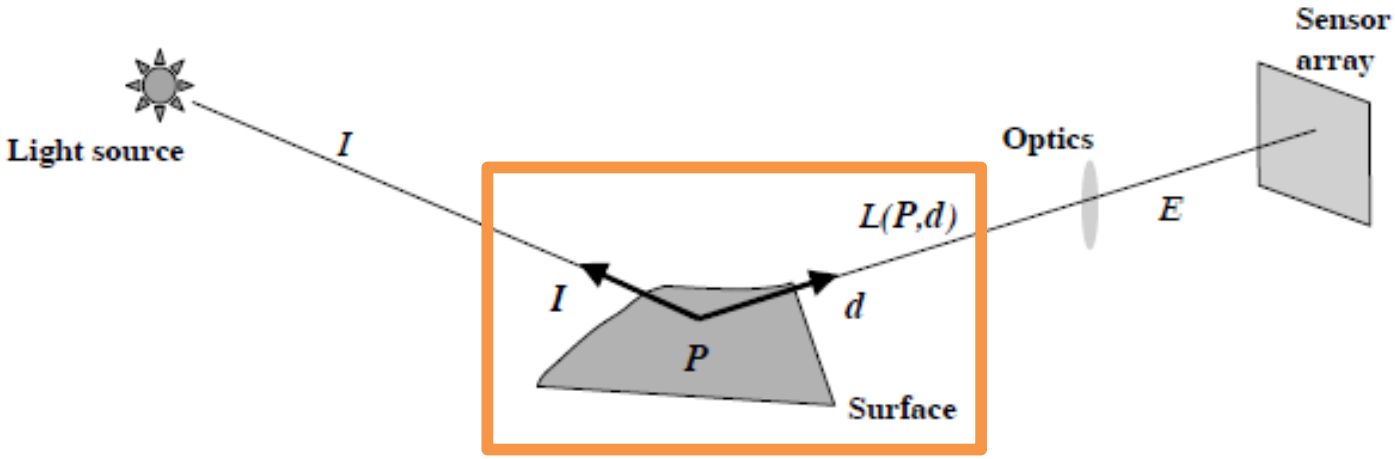
Radiometry

- Two purposes of study
  - Modeling how much of illuminating light is reflected from surfaces.
  - Modeling how much of reflected light reaches the sensor array.



Radiometry

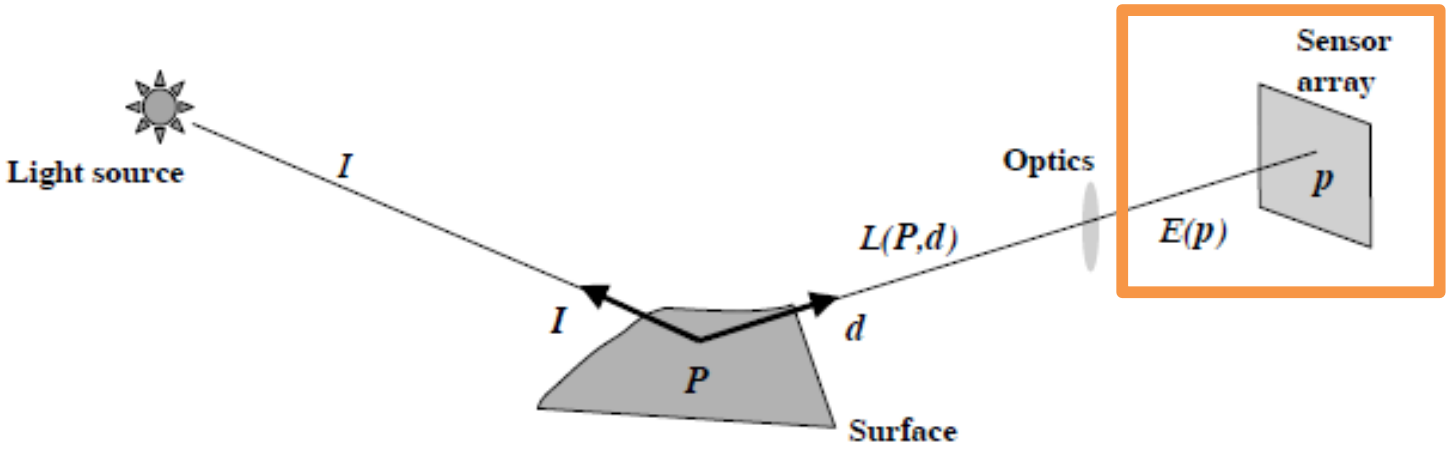
- Scene radiance
  - The power of light, per unit area, emitted at each point, *P*, of a surface in 3D space in a given direction, *d*.
  - Units of power per unit foreshortened area emitted into a unit solid angle W/m^2sr
  - Denote as *L(P,d)*.



Source: Richard Wildes slides

Radiometry

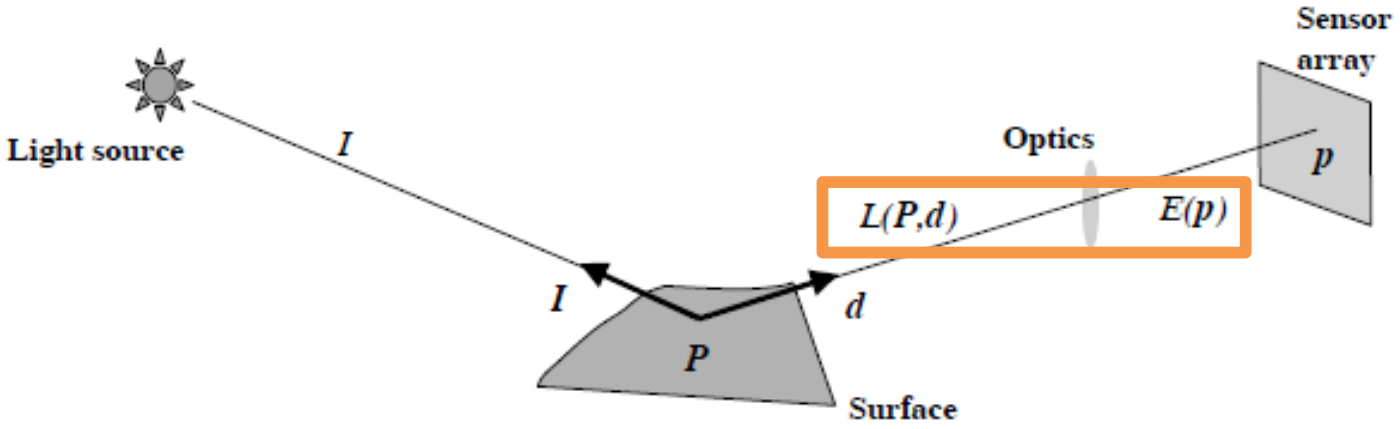
- Image irradiance
  - The power of light, per unit area, at each point, *p*, of the image plane**.**
  - Units of power per unit area W/m^2
  - Denote as *E(p)***.**



Source: Richard Wildes slides

Radiometry

- Relating radiance and irradiance
  - **Goal:** Derive the relationship between light reflected by surface and light registered by sensor.
  - Assume thin lens optical model.

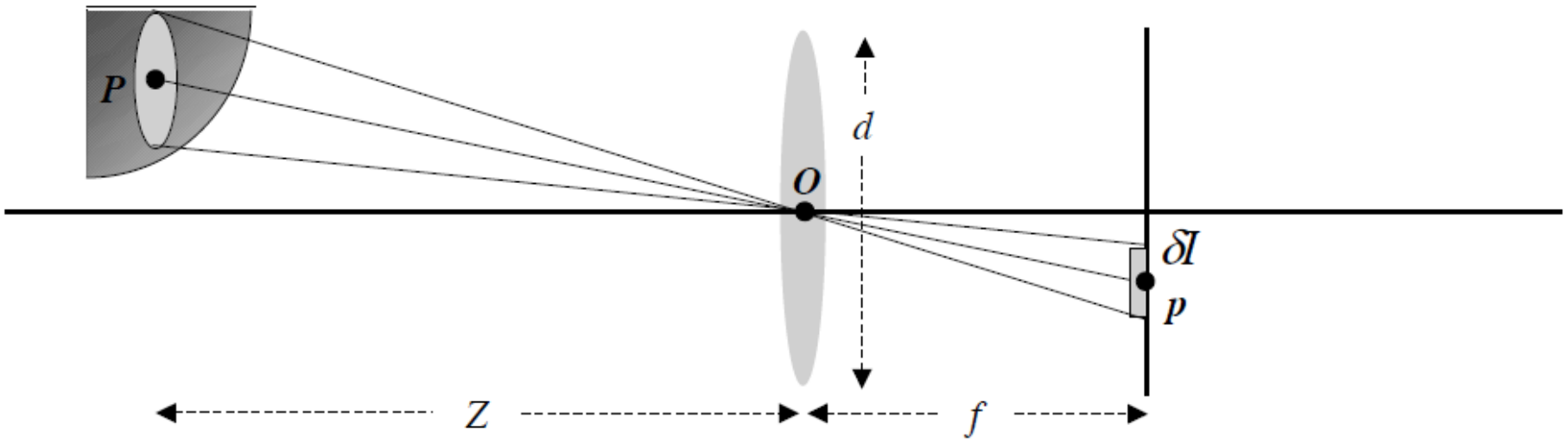


Source: Richard Wildes slides

Radiometry

- Relating radiance and irradiance
  - Image irradiance, *E*, at a point, *p*, is defined as the ratio between the power of light over a small image patch, $\delta\Pi$, and the area of the small image patch, δI.

$$E = \frac{\delta\Pi}{\delta I}$$

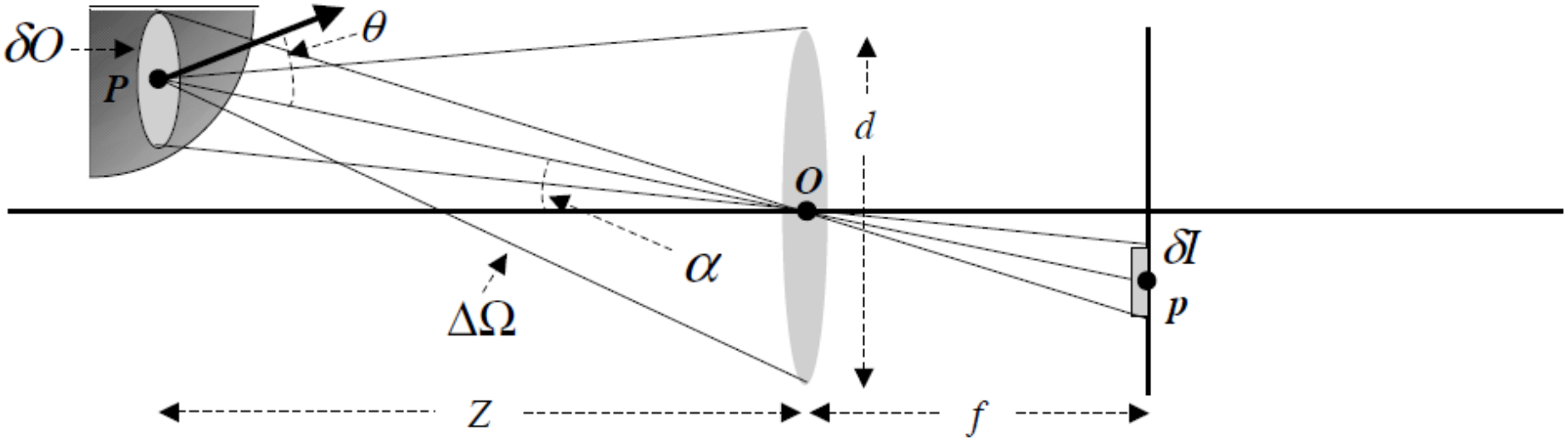


Source: Richard Wildes slides

Radiometry

- Fundamental equation of radiometric image formation
  - Image irradiance, *E*, at a point, *p*, is defined as the ratio between the power of light over a small image patch, *δΠ*, and the area of the small image patch, *δI*.

$$E = \frac{\delta\Pi}{\delta I} = L\frac{\pi}{4}\left(\frac{d}{f}\right)^2 \cos^4\alpha$$



Source: Richard Wildes slides

Radiometry

- Fundamental equation of radiometric image formation
  - Image irradiance, *E*, at a point, *p*, is defined as the ratio between the power of light over a small image patch, *δΠ*, and the area of the small image patch, *δI*.

$$E = \frac{\delta\Pi}{\delta I} = L\frac{\pi}{4}\left(\frac{d}{f}\right)^2 \cos^4\alpha$$

- What it means: The image irradiance at *p* decreases as the fourth power of the cosine of the angle between the principle ray and the optical axis.
  - For small angular aperture, this effect can be neglected

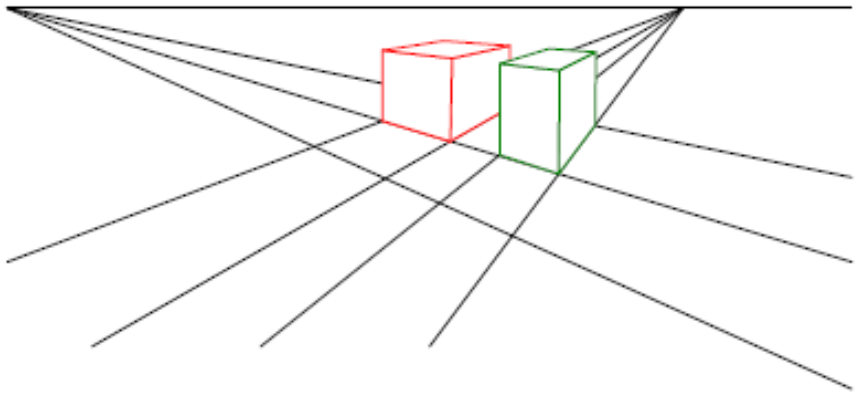
Source: Richard Wildes slides

Image formation

- Introduction
- Optics
- Radiometry
- Geometric image formation
- Image acquisition

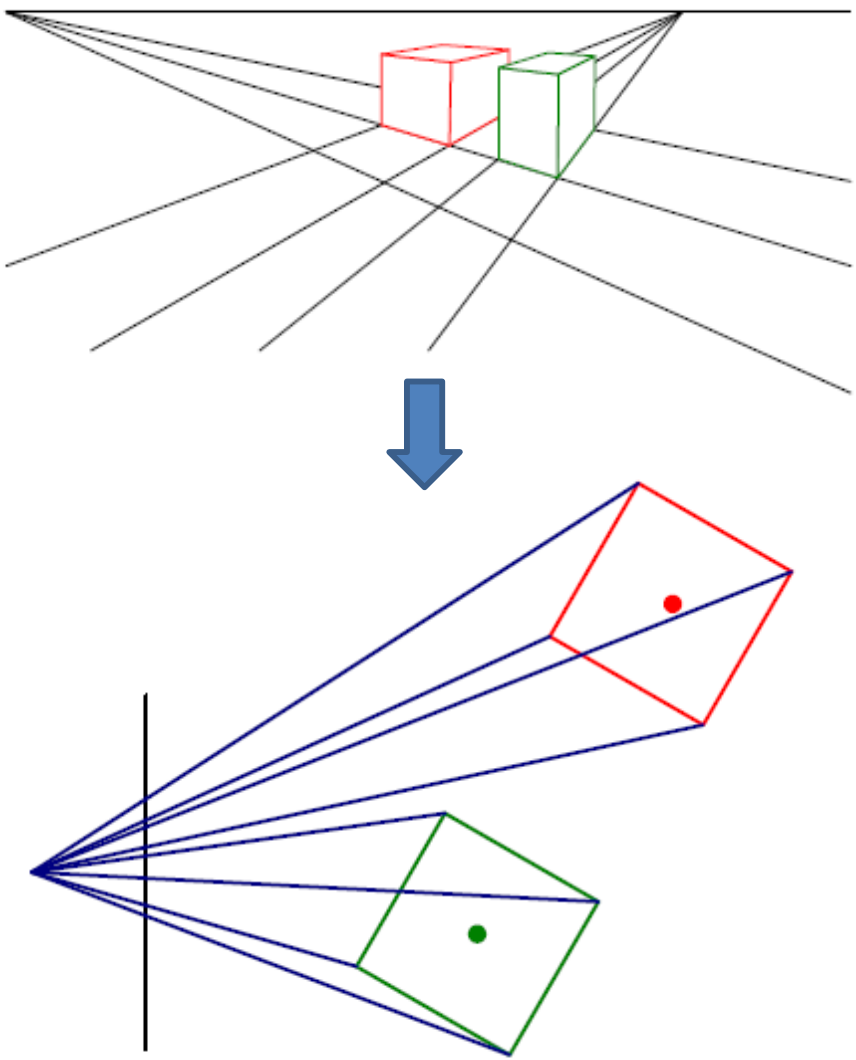
Geometry of image formation

- Specify 3D positions of scene point into the image plane (2D)
- Fundamental projections
  - Perspective
  - Affine
    - Weak-perspective
    - Ortographic
    - Para-perspective



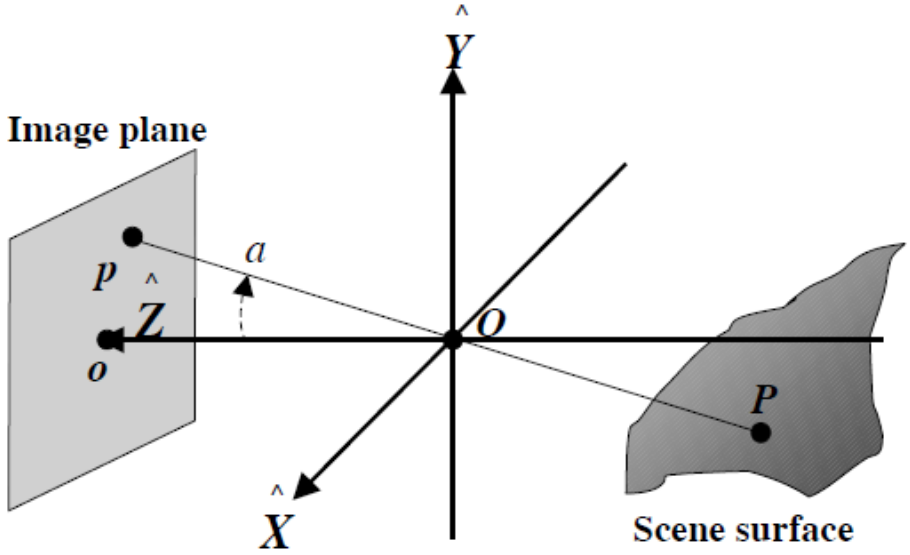
Geometry of image formation

- Fundamental projections: Perspective
  - Most commonly used projection in computer vision and computer graphics
  - Points are projected on the image plane by dividing them by their z component



Geometry of image formation

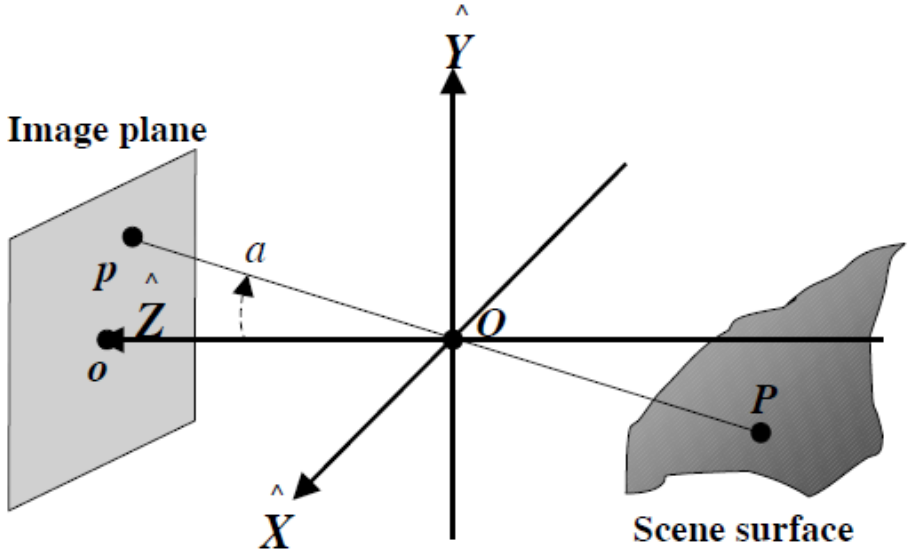
- Fundamental projections: **Perspective**
  - Define cartesian coordinate system with center O
  - Let the optical axis align at Z-axis
  - Let the image plane
    - Be parallel to XY plane
    - At a distance f, the focal length along the optical axis
    - piercing the optical axis at o, the principle point



Source: Richard Wildes slides

Geometry of image formation

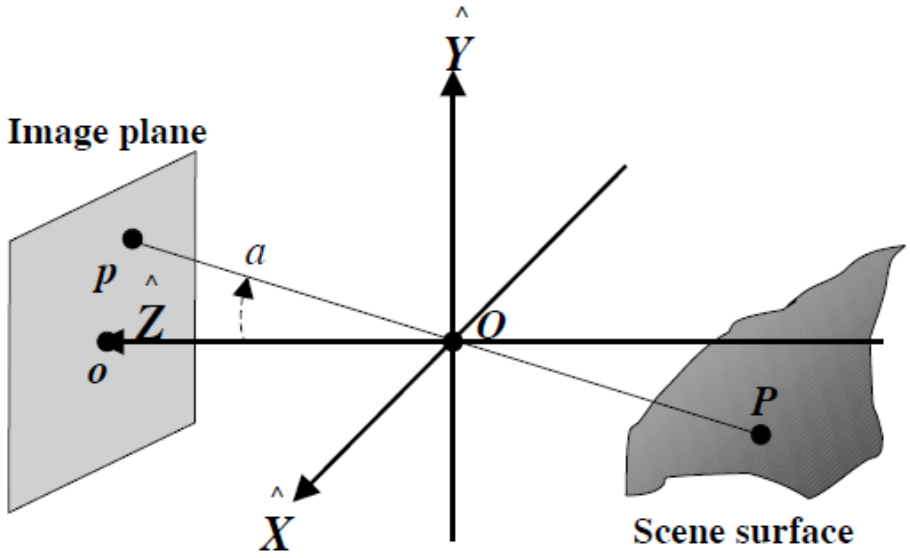
- Fundamental projections: **Perspective**
  - Define cartesian coordinate system with center O
  - Let the optical axis align at Z-axis
  - Let the image plane
    - Be parallel to XY plane
    - At a distance f, the focal length along the optical axis
    - piercing the optical axis at o, the principle point
  - Consider the projection of a scene point $P=(X,Y,Z)$ into $p=(x,y,f)$
  - Fundamental assumption: P and p are collinear (pinhole model)
  - Let the ray Pp, make an angle a with the optical axis



Source: Richard Wildes slides

Geometry of image formation

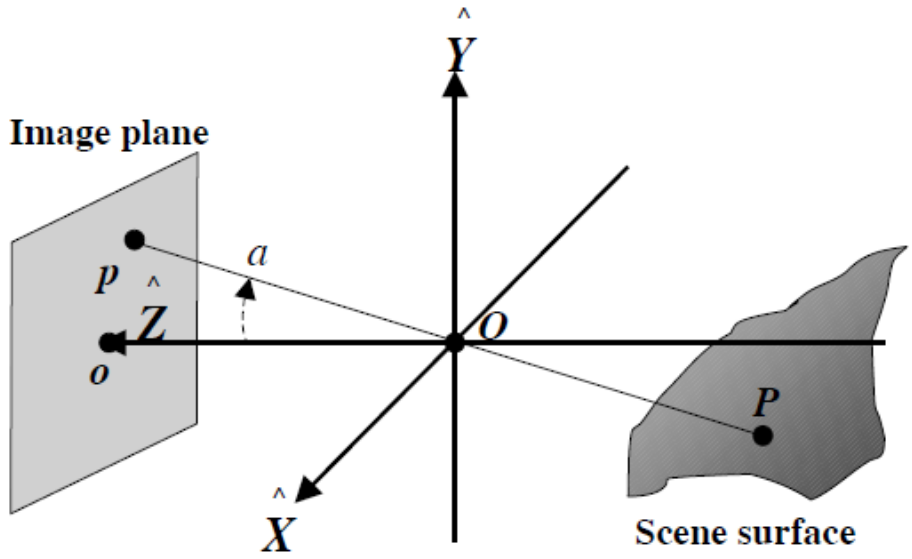
- Fundamental projections: **Perspective**
  - The length of ***P*** is $P \cdot -Z \sec a \cdot -(P \cdot \hat{Z}) \sec a$



Source: Richard Wildes slides

Geometry of image formation

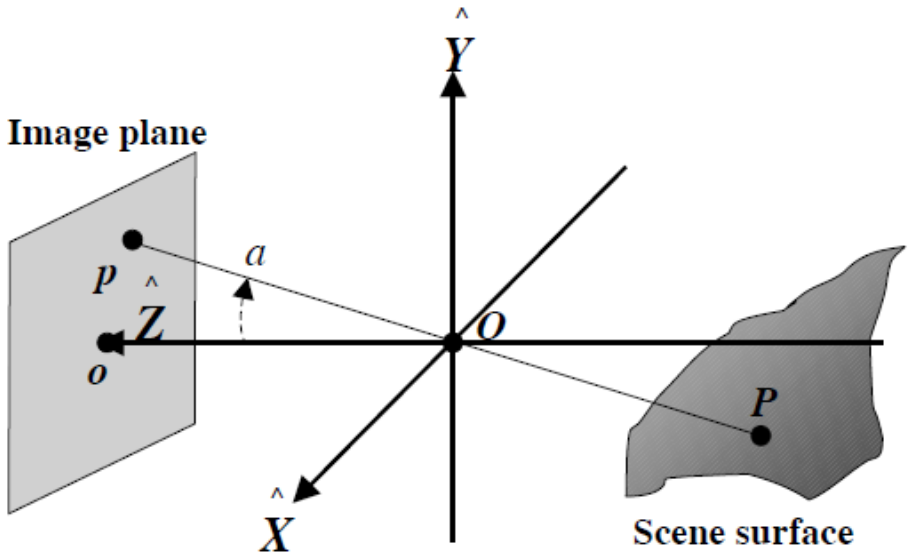
- Fundamental projections: **Perspective**
  - The length of ***P*** is $P = -Z \sec a = -(P \cdot \hat{Z}) \sec a$
  - The length of ***p*** is $p = f \sec a$
  - So $\frac{1}{f} p = \frac{1}{P \cdot \hat{Z}} P$



Source: Richard Wildes slides

Geometry of image formation

- Fundamental projections: **Perspective**
  - The length of ***P*** is $P = -Z\sec a = -(P\cdot\hat{Z})\sec a$
  - The length of ***p*** is $p = f\sec a$
  - So $\frac{1}{f}p = \frac{1}{P\cdot\hat{Z}}P$
  - In component form
    $$\frac{x}{f} = \frac{X}{Z}, \frac{y}{f} = \frac{Y}{Z}$$
  - Or
    $$x = f\frac{X}{Z}, y = f\frac{Y}{Z}$$



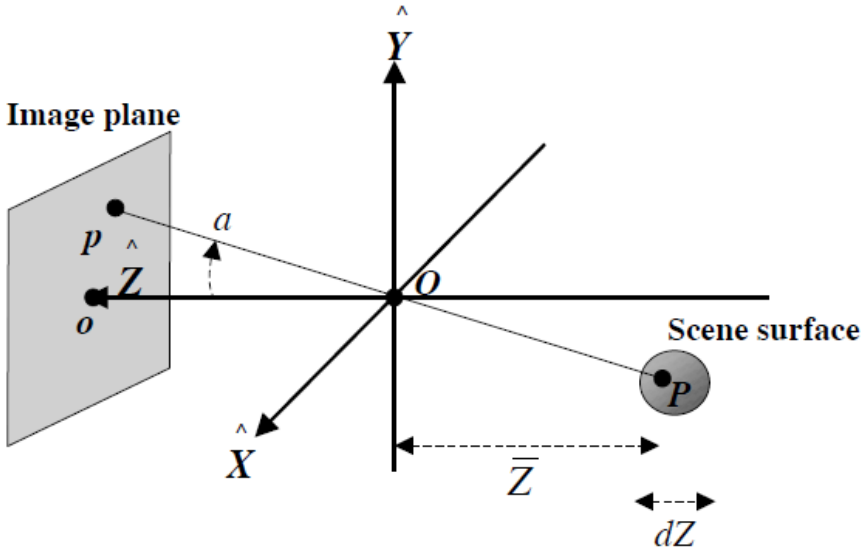
Source: Richard Wildes slides

Geometry of image formation

- Fundamental projections: Weak perspective

  – Assumes variation of distance along optical axis, dZ, is small, compared to average distance $\bar{Z}$

  – The perspective equations can be approximated as

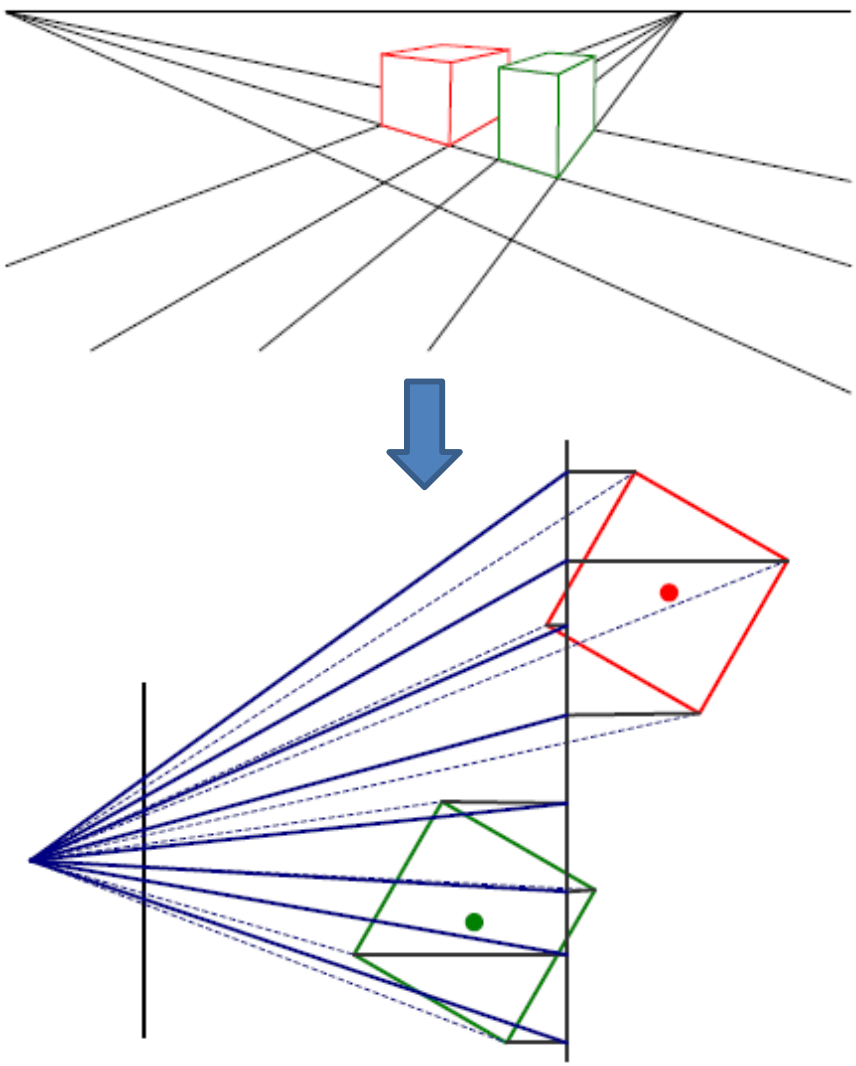


$$x = f\frac{X}{Z} \approx f\frac{X}{\bar{Z}}, y = f\frac{Y}{Z} \approx f\frac{Y}{\bar{Z}}$$

Source: Richard Wildes slides

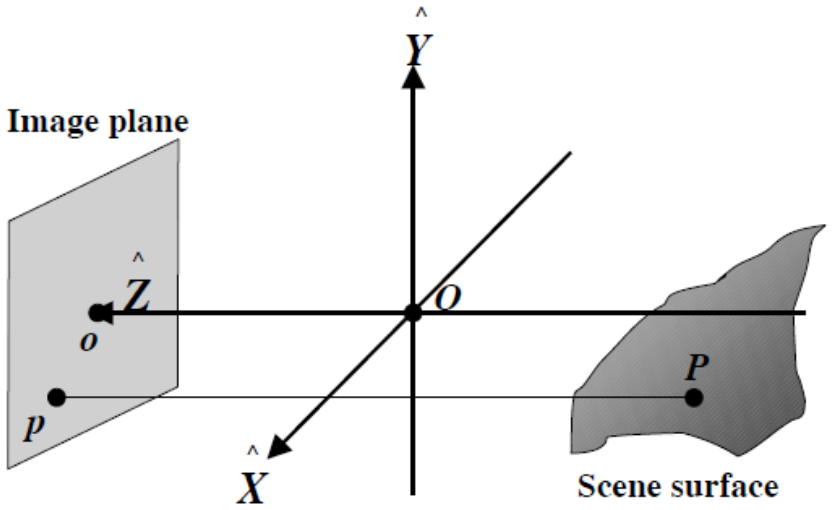
Geometry of image formation

- Fundamental projections: Ortographic
  - Drops the z component
  - Another interpretation: Perspective where
    - $f \rightarrow \infty$
    - Correspondinly, $Z \rightarrow \infty$
    - Then $f/Z \rightarrow 1$
    - And $x=X$, $y=Y$



Geometry of image formation

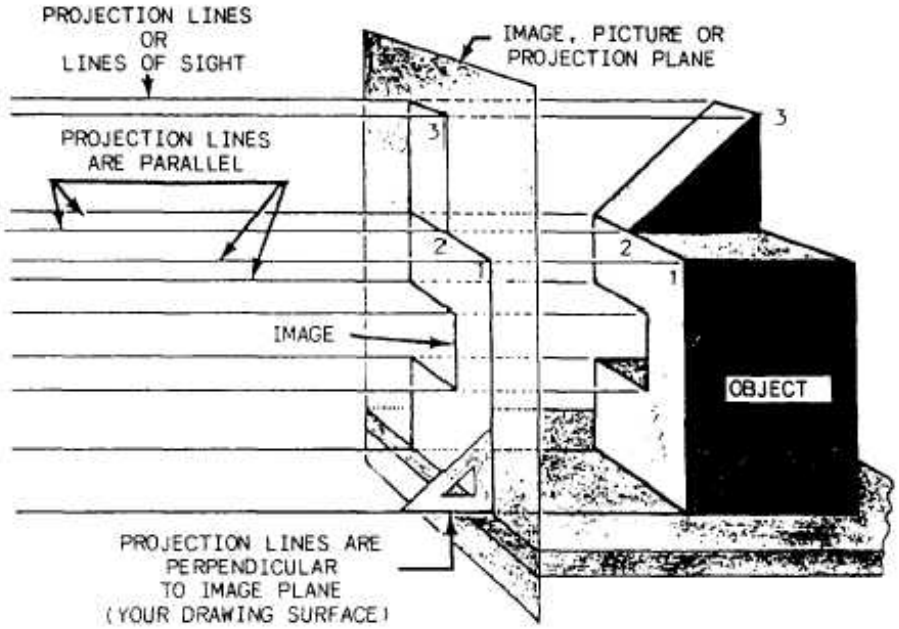
- Fundamental projections: Ortographic
  - Drops the z component
  - Another interpretation: Perspective where
    - $f \rightarrow \infty$
    - Correspondinly, $Z \rightarrow \infty$
    - Then $f/Z \rightarrow 1$
    - And $x=X$, $y=Y$



Source: Richard Wildes slides

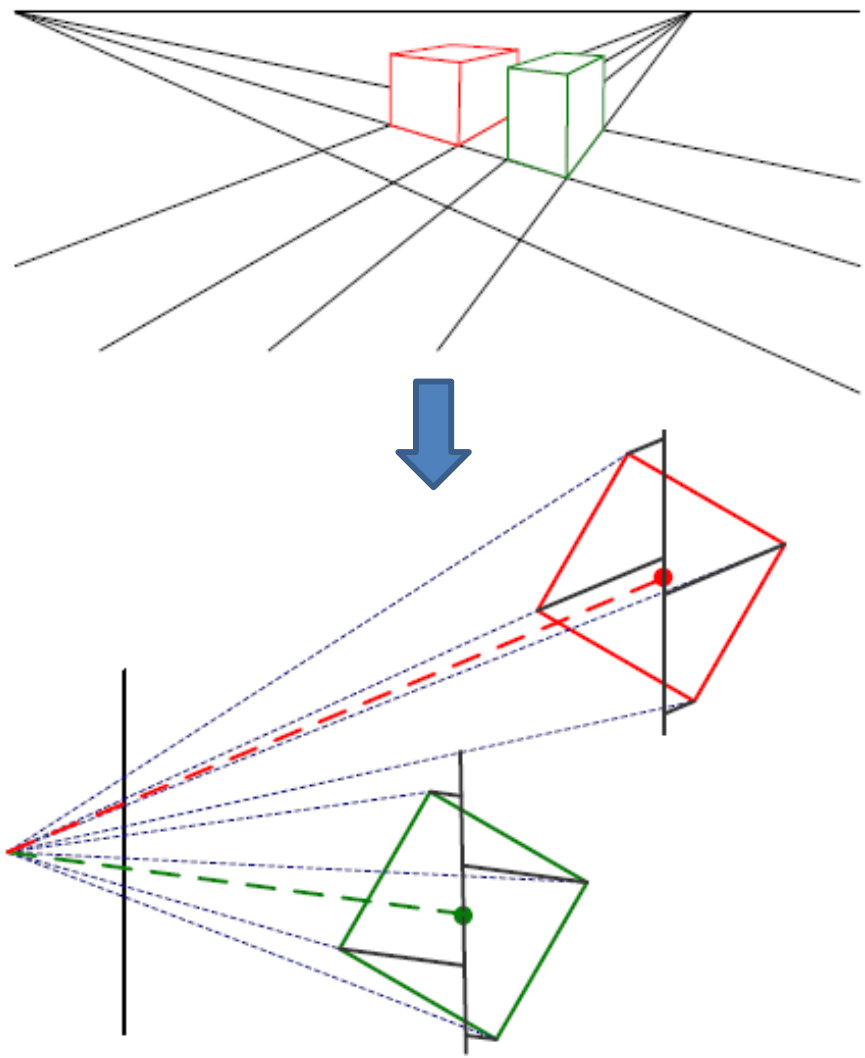
Geometry of image formation

- Fundamental projections: Ortographic
  - Drops the z component
  - Another interpretation: Perspective where
    - $f \rightarrow \infty$
    - Correspondinly, $Z \rightarrow \infty$
    - Then $f/Z \rightarrow 1$
    - And $x=X$, $y=Y$



Geometry of image formation

- Fundamental projections: Para-perspective
  - Points instead of being projected orthogonal to local reference plane, they are projected parallel to the line of sight of the object center.



Camera models

- So far, we have made a correspondence between the scene point, P and the camera point, p
  - In reality this equation is only valid when all distances are measured in the camera's reference frame
  - In practice, the world and camera coordinate systems are related by a set of physical parameters (focal length, size of pixels, position and orientation of camera, etc.)
- it is desirable to make an explicit correspondence with an external, world reference frame.

Camera models

- Two basic assumptions
  1. The camera reference frame can be located with respect to some other, known, reference frame – the world reference frame.
  2. The coordinates of the image points in the camera reference frame can be obtained from the image coordinates – the only ones directly available from the image.

Camera models

- Intrinsic parameters
  - Relate camera's coordinate system to idealized coordinate system
  - That is, link the pixel coordinates of an image point with corresponding coordinates in the camera reference frame.

Camera models

- Intrinsic parameters
  - Relate camera's coordinate system to idealized coordinate system
  - That is, link the pixel coordinates of an image point with corresponding coordinates in the camera reference frame.
- Extrinsic parameters
  - Relate the camera's coordinate system to a fixed world coordinate system and specify its position and orientation in space

Camera models

- Intrinsic parameters
  - Relate camera's coordinate system to idealized coordinate system
  - That is, link the pixel coordinates of an image point with corresponding coordinates in the camera reference frame.
- Extrinsic parameters
  - Relate the camera's coordinate system to a fixed world coordinate system and specify its position and orientation in space
- The problem of estimating the intrinsic and extrinsic parameters of a camera is known as geometric camera calibration

Geometry of image formation

- Extrinsic parameters
  - Camera to world transformation
  - Typically given via two sets of parameters
    1. A 3D translation vector, T, describing the relative positions of the two reference frames.
    2. A 3x3 rotation matrix, R, that brings the corresponding axes of the two frames into alignment.
  - Letting P_c and P_w be the camera and world coordinates of the same point, we write:

$$P_c = R(P_w - T)$$

  - By definition, R, is completely specified be 3 parameters (e.g., rotation about each of the coordinate axes); so, there are 6 extrinsic parameters in total (3 for T; 3 for R).

Source: Richard Wildes slides

Geometry of image formation

- Intrinsic parameters
  - Camera to pixel transformation
  - Typically given by 2 sets of parameters
    1. The focal length, f, serving to capture the (perspective) projection.
    2. The pixel coordinates of the principle point (image center), (o_x, o_y), and the effective pixel horizontal and vertical dimensions, (s_x, s_y), serving to capture the transformation between camera frame coordinates and pixel coordinates.
  - We already have considered how to incorporate the focal length.
  - Letting (x_i, y_i) be the pixel coordinates, we incorporate the second set of parameters via

$$x=-(x_i-o_x)s_x$$

$$y=-(y_i-o_y)s_y$$

  - There are 5 total intrinsic parameters: (f, o_x, o_y, s_x, s_y)

Source: Richard Wildes slides

Image formation

- Introduction
- Optics
- Radiometry
- Geometric image formation
- Image acquisition

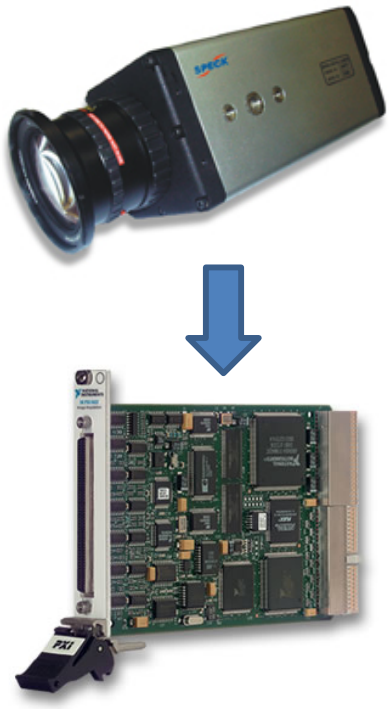
Digital image acquisition

- Three major hardware components
  - Camera

Source: Richard Wildes slides

Digital image acquisition

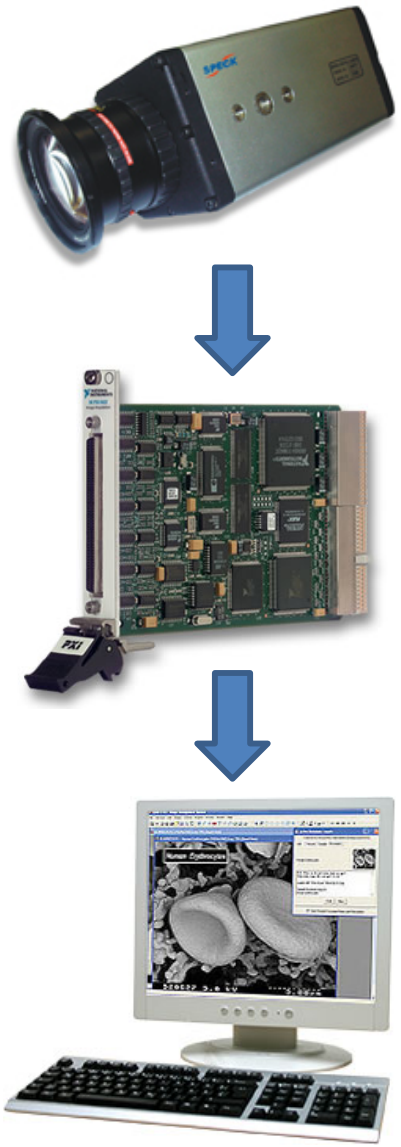
- Three major hardware components
  - Camera
  - Frame grabber



Source: Richard Wildes slides

Digital image acquisition

- Three major hardware components
  - Camera
  - Frame grabber
  - Host computer

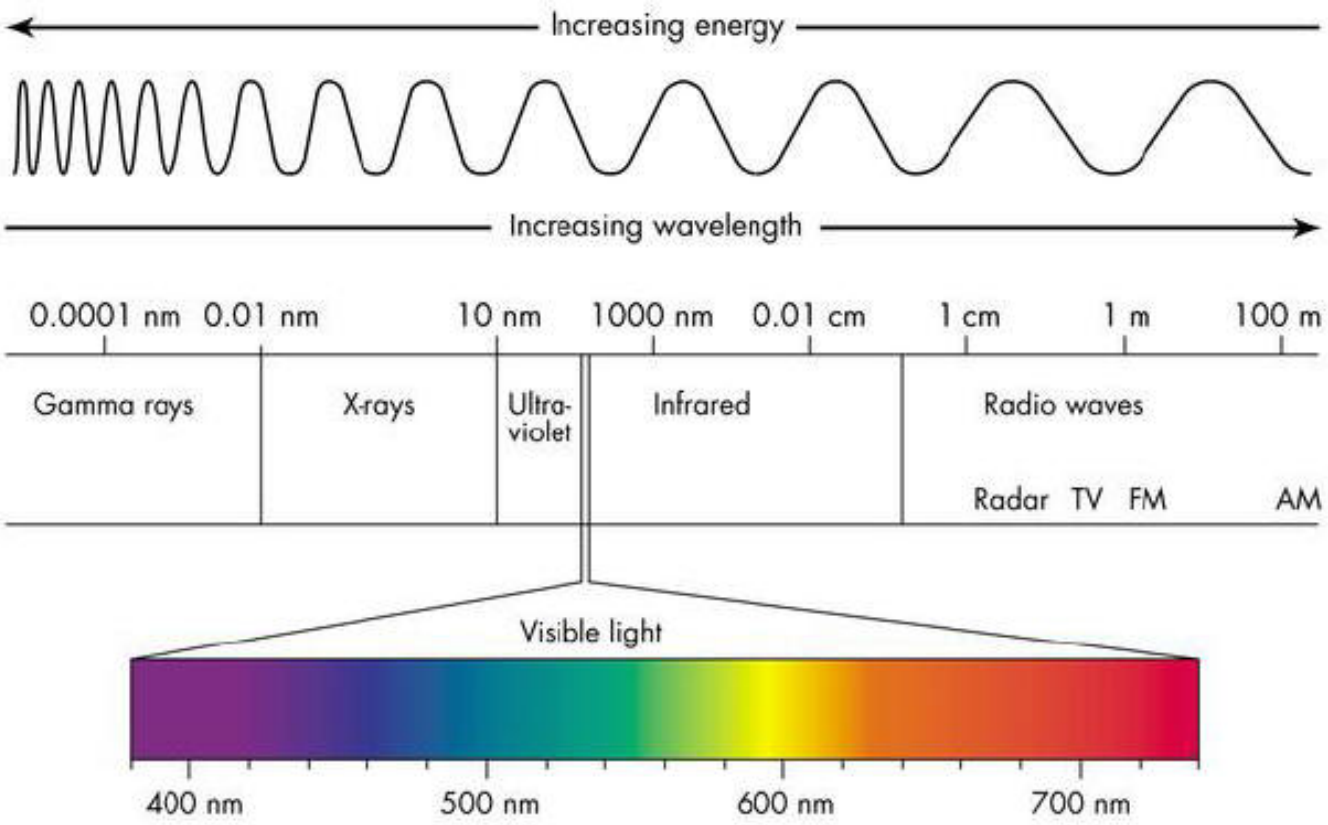


Digital image acquisition

- Digital camera
  - After starting from one or more light sources, reflecting off one or more surfaces in the world and passing through the camera's optics, light finally reaches the *imaging sensor*
  - Two types of sensors
    - CCD: Charged Coupled Device
    - CMOS: Complementary Metal Oxide on Silicon
    - In each case is an $n \times m$ rectangular grid of photosensors
    - Output is a continuous electrical signal

Digital image acquisition

- Color



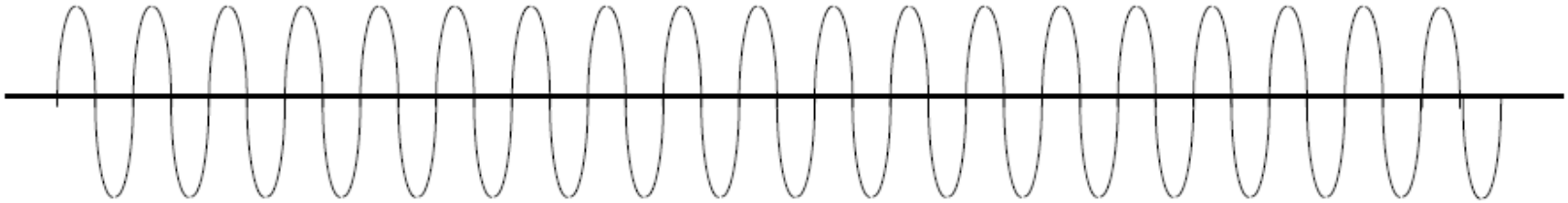
Digital image acquisition

- Noise
  - Measurements are affected by fluctuations in the signal being measured.
  - If the measurements are repeated, somewhat different results might be obtained.
  - Typically, measurements will cluster around the correct value.
  - It can be useful to consider the probability that a measurement will fall within a certain interval, roughly
    - This is the limit of the ratio of the number of measurements that fall in that interval
    - to the total number of trials,
    - as the total number or trials tends to infinity.

Source: Richard Wildes slides

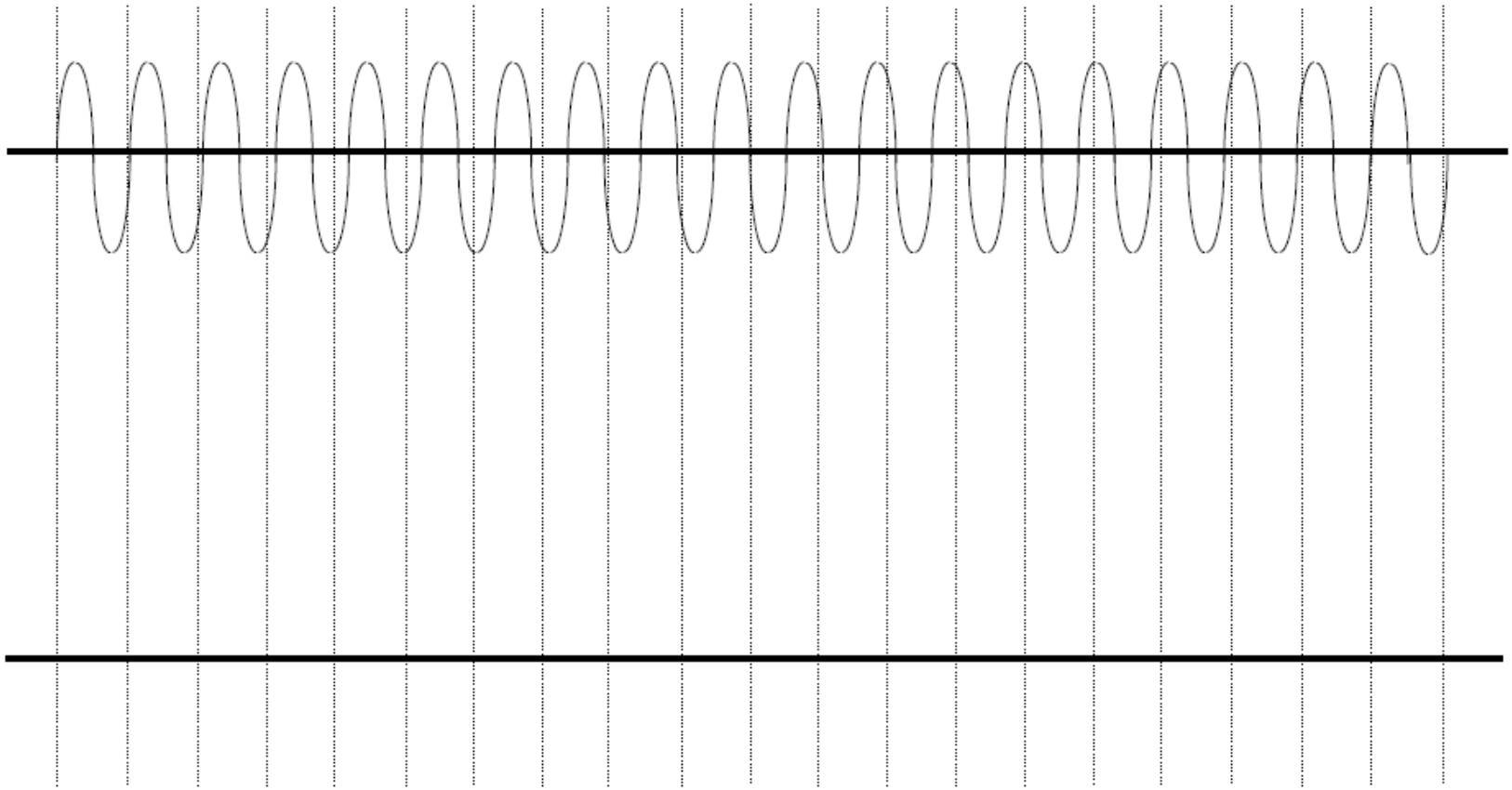
Digital image acquisition

- Sampling and aliasing



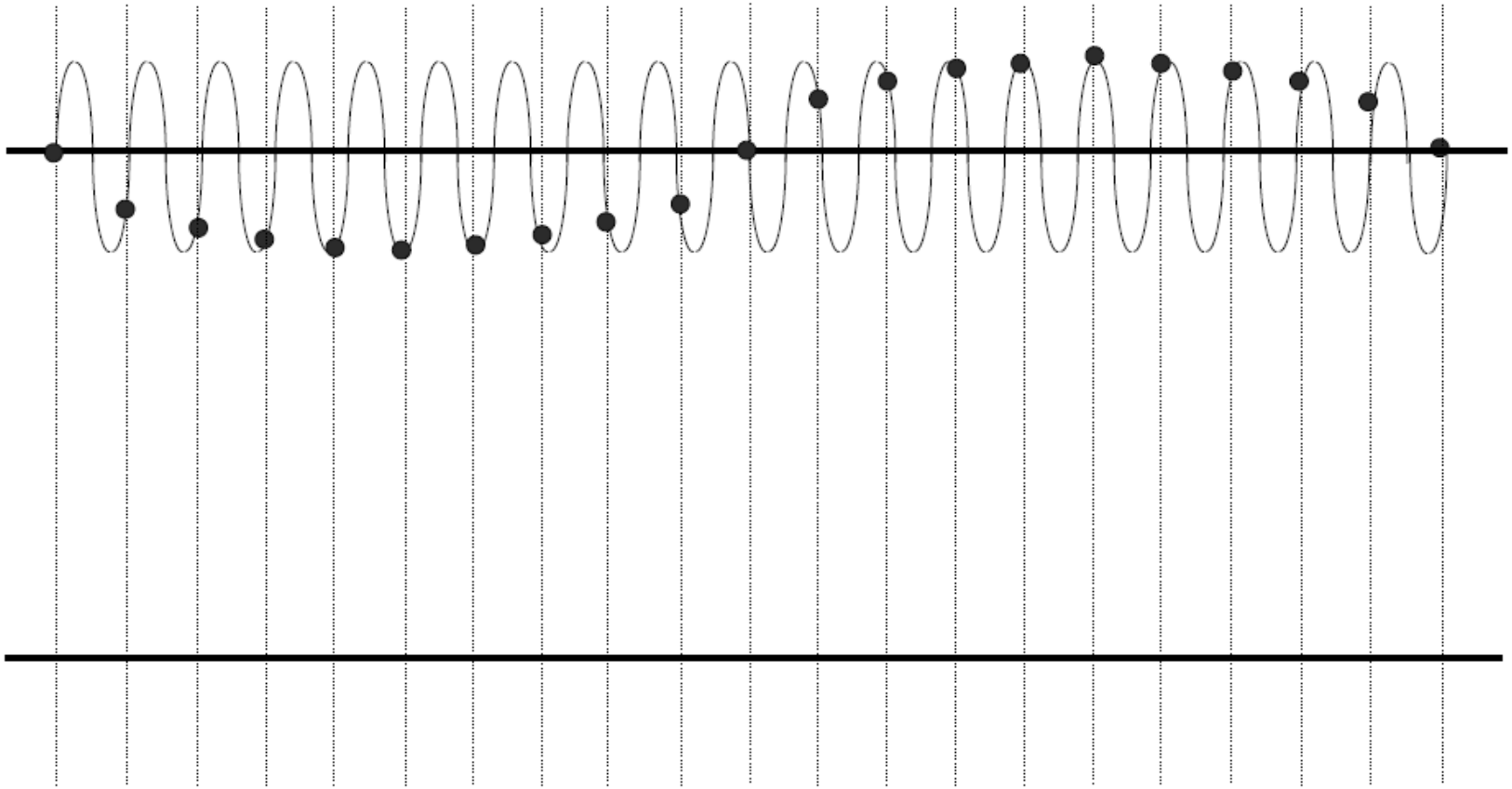
Digital image acquisition

- Sampling and aliasing



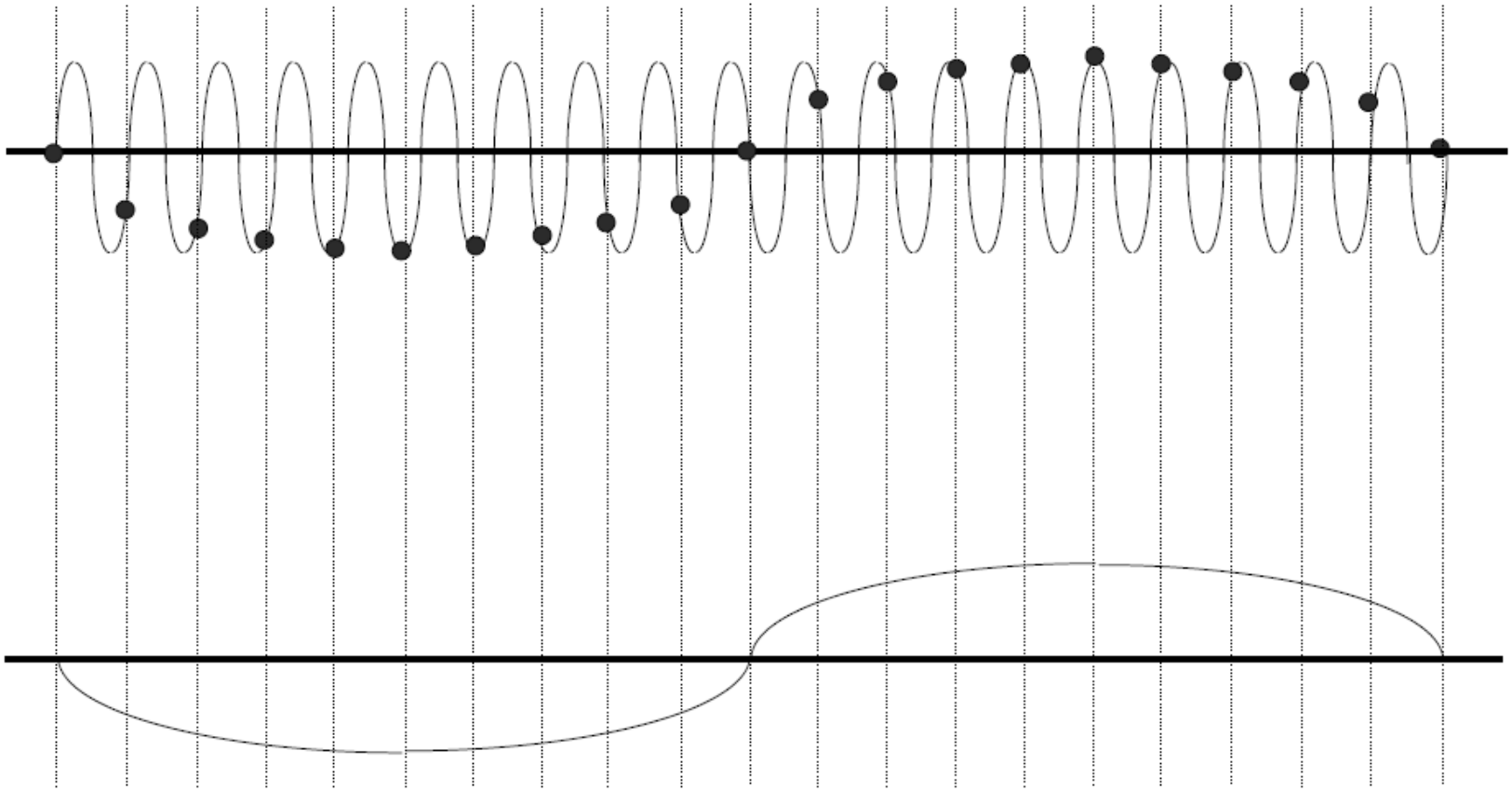
Digital image acquisition

- Sampling and aliasing



Digital image acquisition

- Sampling and aliasing



Digital image acquisition

- Sampling and aliasing
  - Shannon's sampling theorem
    - Rate required to reconstruct a signal from its instantaneous samples must be at least twice the highest frequency

$$f_s = 2f_{\max}$$

Digital image acquisition

- Spatial quantization
  - The optical component (lens, aperture, etc.) are capable of imaging spatial frequencies approximately an order of magnitude higher than what could be sampled by the sensor array
    - We are to expect undersampled images with corresponding artifacts
    - In particular, aliasing, the masquerading of high frequencies as spurious low frequencies (jaggies)

  - However,
    - The amplitude of such components as derived from common image sources contain little energy in such regions of the spectrum.
    - We do not sample at points; rather each sampling element reports the average irradiance over a finite area and thereby eliminates the highest frequency components before they can be aliased.

Source: Richard Wildes slides

Image formation

- Introduction
- Optics
- Radiometry
- Geometric image formation
- Image acquisition

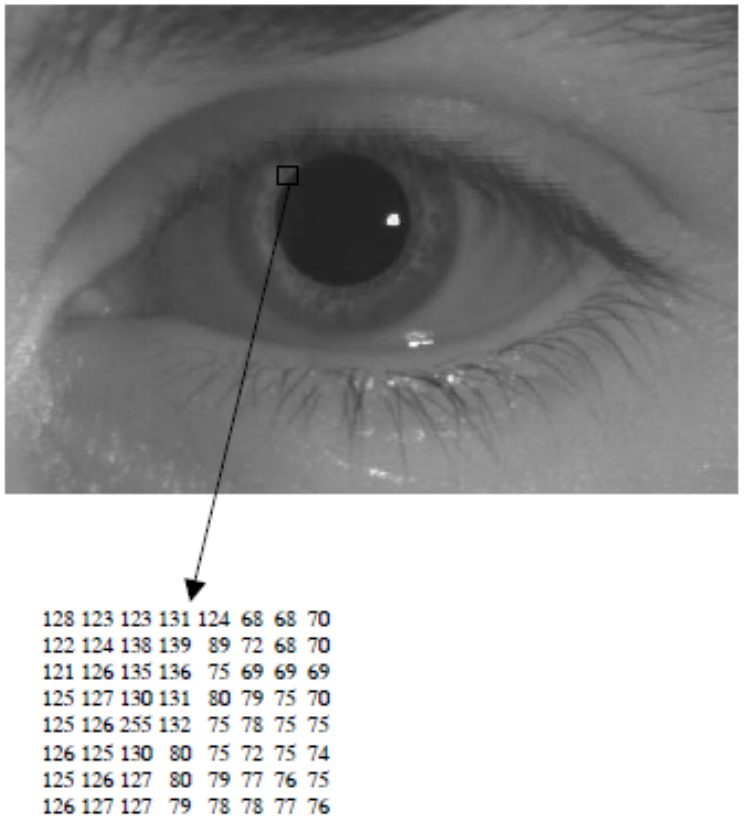
universität innsbruck

Introduction to OpenCV

Assoz.Prof. Antonio Rodríguez-Sánchez, PhD.

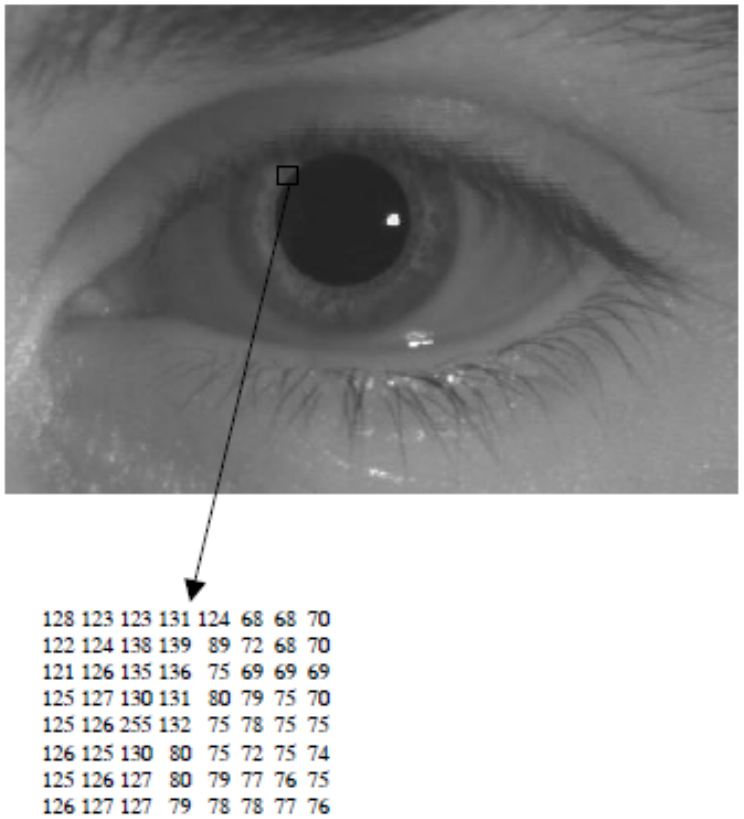
Mat: The basic image container

- No need to manually allocate and release memory



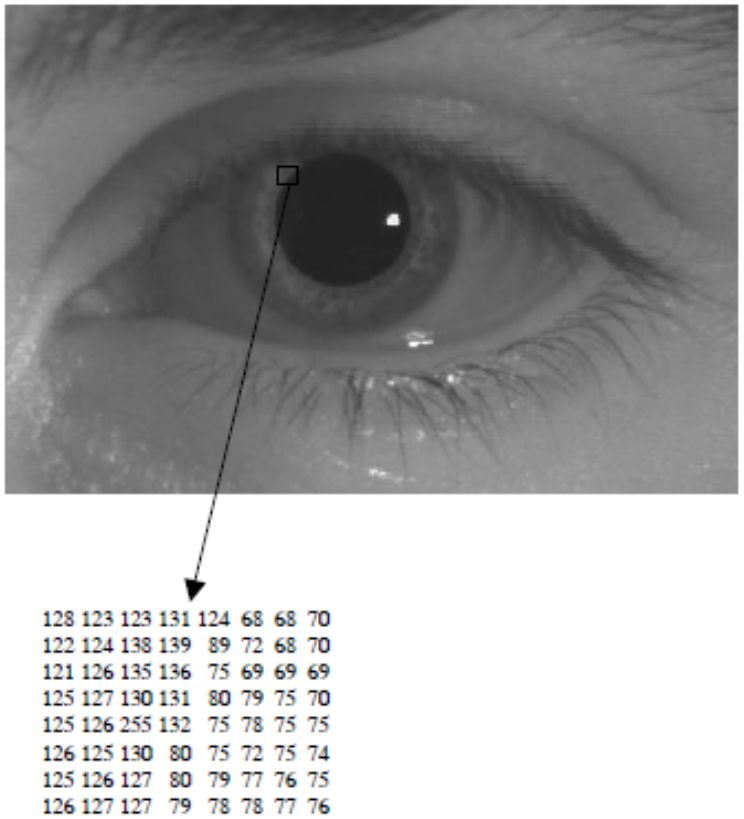
Mat: The basic image container

- No need to manually allocate and release memory
- Class with two data parts
  - Matrix header
  - Pointer to the matrix



Mat: The basic image container

- No need to manually allocate and release memory
- Class with two data parts
  - Matrix header
  - Pointer to the matrix
- OpenCV uses a reference counting system
  - Copy operators copy the headers and the pointer, not the data



Mat: The basic image container

- No need to manually allocate and release memory
- OpenCV uses a reference counting system
  - Copy operators copy the headers and the pointer

```
Mat A, C;                                 // creates just the header parts
A = imread(argv[1], CV_LOAD_IMAGE_COLOR); // here we'll know the method used (allocate matrix)

Mat B(A);                                 // Use the copy constructor

C = A;                                    // Assignment operator
```

Mat: The basic image container

- No need to manually allocate and release memory
- OpenCV uses a reference counting system
  - Copy operators copy the headers and the pointer

```
Mat A, C;                                 // creates just the header parts
A = imread(argv[1], CV_LOAD_IMAGE_COLOR); // here we'll know the method used (allocate matrix)

Mat B(A);                                 // Use the copy constructor

C = A;                                    // Assignment operator
```

```
Mat D (A, Rect(10, 10, 100, 100) ); // using a rectangle
Mat E = A(Range::all(), Range(1,3)); // using row and column boundaries
```

Mat: The basic image container

- No need to manually allocate and release memory
- OpenCV uses a reference counting system
  - Copy operators copy the headers and the pointer
  - Sometimes you need to copy the matrix itself

```
Mat F = A.clone();
Mat G;
A.copyTo(G);
```

Mat: Storing

- RGB, which in OpenCV is BGR
- HSV and HLS
  - Hue, Saturation and Value/Luminance
- YCrCb
  - Used by JPEG
- CIE L*a*b*
  - Perceptually uniform color space

Scalar

- Represents a 4-element vector
  - Used for passing pixel values
  - It can represent BGR values
    - Not necessary to define the last argument

Scalar

- Represents a 4-element vector
  - Used for passing pixel values
  - It can represent BGR values
    - Not necessary to define the last argument

```
Scalar( a, b, c )
```

Mat: Storing

	Column 0	Column 1	Column ...	Column m
Row 0	0,0	0,1	...	0, m
Row 1	1,0	1,1	...	1, m
Row ...	...,0	...,1	...	..., m
Row n	n,0	n,1	n,...	n, m

	Column 0			Column 1			Column ...			Column m		
Row 0	0,0	0,0	0,0	0,1	0,1	0,1	...	...	...	0, m	0, m	0, m
Row 1	1,0	1,0	1,0	1,1	1,1	1,1	...	...	...	1, m	1, m	1, m
Row ...	...,0	...,0	...,0	...,1	...,1	...,1	...	...	...	..., m	..., m	..., m
Row n	n,0	n,0	n,0	n,1	n,1	n,1	n,...	n,...	n,...	n, m	n, m	n, m

Mat: Constructor

```
Mat M(2,2, CV_8UC3, Scalar(0,0,255));
cout << "M = " << endl << " " << M << endl << endl;
```

```
M =
 [0, 0, 255, 0, 0, 255;
  0, 0, 255, 0, 0, 255]
```

Mat: Constructor

```
Mat M(2,2, CV_8UC3, Scalar(0,0,255));
cout << "M = " << endl << " " << M << endl << endl;
```

```
M =
 [0, 0, 255, 0, 0, 255;
  0, 0, 255, 0, 0, 255]
```

```
CV_[The number of bits per item][Signed or Unsigned][Type Prefix]C[The channel number]
```

Mat: Constructor

```
Mat M(2,2, CV_8UC3, Scalar(0,0,255));
cout << "M = " << endl << " " << M << endl << endl;
```

```
M =
 [0, 0, 255, 0, 0, 255;
  0, 0, 255, 0, 0, 255]
```

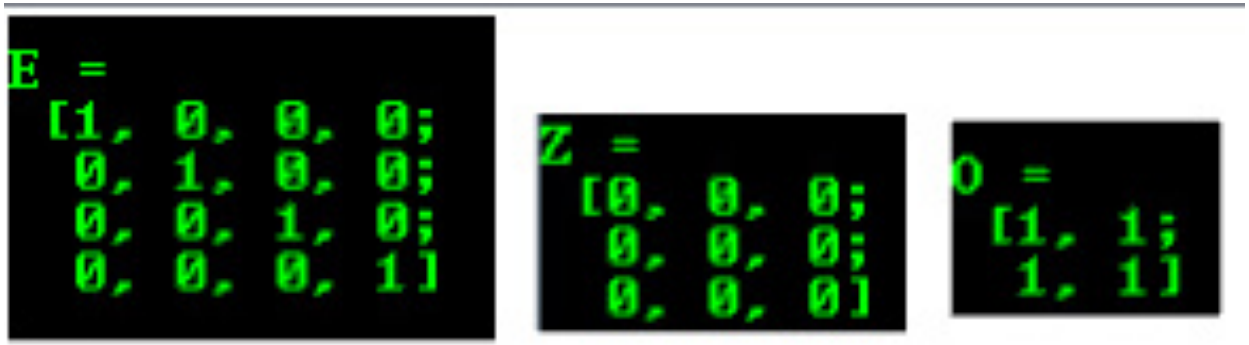
```
CV_[The number of bits per item][Signed or Unsigned][Type Prefix]C[The channel number]
```

```
IplImage* img = cvLoadImage("greatwave.png", 1);
Mat mtx(img); // convert IplImage* -> Mat
```

Mat: Constructor

```
CV_[The number of bits per item][Signed or Unsigned][Type Prefix]C[The channel number]
```

```
Mat E = Mat::eye(4, 4, CV_64F);
cout << "E = " << endl << " " << E << endl << endl;

Mat O = Mat::ones(2, 2, CV_32F);
cout << "O = " << endl << " " << O << endl << endl;

Mat Z = Mat::zeros(3,3, CV_8UC1);
cout << "Z = " << endl << " " << Z << endl << endl;
```



Mat: Filtering

```
Mat kern = (Mat_<char>(3,3) <<  0, -1,  0,
                              -1,  5, -1,
                               0, -1,  0);
```

```
filter2D(I, K, I.depth(), kern);
```

Mat: Filtering

```
Mat kern = (Mat_<char>(3,3) <<  0, -1,  0,
                               -1,  5, -1,
                                0, -1,  0);
```

```
filter2D(I, K, I.depth(), kern);
```

Example: Blending

```
#include <cv.h>
#include <highgui.h>
#include <iostream>

using namespace cv;

int main( int argc, char** argv )
{
 double alpha = 0.5; double beta; double input;

 Mat src1, src2, dst;

 /// Ask the user enter alpha
 std::cout<<" Simple Linear Blender "<<std::endl;
 std::cout<<"-----------------------"<<std::endl;
 std::cout<<"* Enter alpha [0-1]: ";
 std::cin>>input;

 /// We use the alpha provided by the user if it is between 0 and 1
 if( input >= 0.0 && input <= 1.0 )
   { alpha = input; }
```

Example: Blending

```
 /// Read image ( same size, same type )
 src1 = imread("../../images/LinuxLogo.jpg");
 src2 = imread("../../images/WindowsLogo.jpg");

 if( !src1.data ) { printf("Error loading src1 \n"); return -1; }
 if( !src2.data ) { printf("Error loading src2 \n"); return -1; }

 /// Create Windows
 namedWindow("Linear Blend", 1);

 beta = ( 1.0 - alpha );
 addWeighted( src1, alpha, src2, beta, 0.0, dst);

 imshow( "Linear Blend", dst );

 waitKey(0);
 return 0;
}
```

Example: Blending

```
 /// Read image ( same size, same type )
 src1 = imread("../../images/LinuxLogo.jpg");
 src2 = imread("../../images/WindowsLogo.jpg");

 if( !src1.data ) { printf("Error loading src1 \n"); return -1; }
 if( !src2.data ) { printf("Error loading src2 \n"); return -1; }

 /// Create Windows
 namedWindow("Linear Blend", 1);

 beta = ( 1.0 - alpha );
 addWeighted( src1, alpha, src2, beta, 0.0, dst);

 imshow( "Linear Blend", dst );

 waitKey(0);
 return 0;
}
```

Example: Brightness and contrast

```
#include <cv.h>
#include <highgui.h>
#include <iostream>

using namespace cv;
double alpha; /**< Simple contrast control */
int beta;  /**< Simple brightness control */

int main( int argc, char** argv )
{
 /// Read image given by user
 Mat image = imread( argv[1] );
 Mat new_image = Mat::zeros( image.size(), image.type() );

 /// Initialize values
 std::cout<<" Basic Linear Transforms "<<std::endl;
 std::cout<<"-------------------------"<<std::endl;
 std::cout<<"* Enter the alpha value [1.0-3.0]: ";std::cin>>alpha;
 std::cout<<"* Enter the beta value [0-100]: "; std::cin>>beta;

 /// Do the operation new_image(i,j) = alpha*image(i,j) + beta
 for( int y = 0; y < image.rows; y++ )
    { for( int x = 0; x < image.cols; x++ )
         { for( int c = 0; c < 3; c++ )
              {
      new_image.at<Vec3b>(y,x)[c] =
         saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta );
             }
    }
```

Example: Brightness and contrast

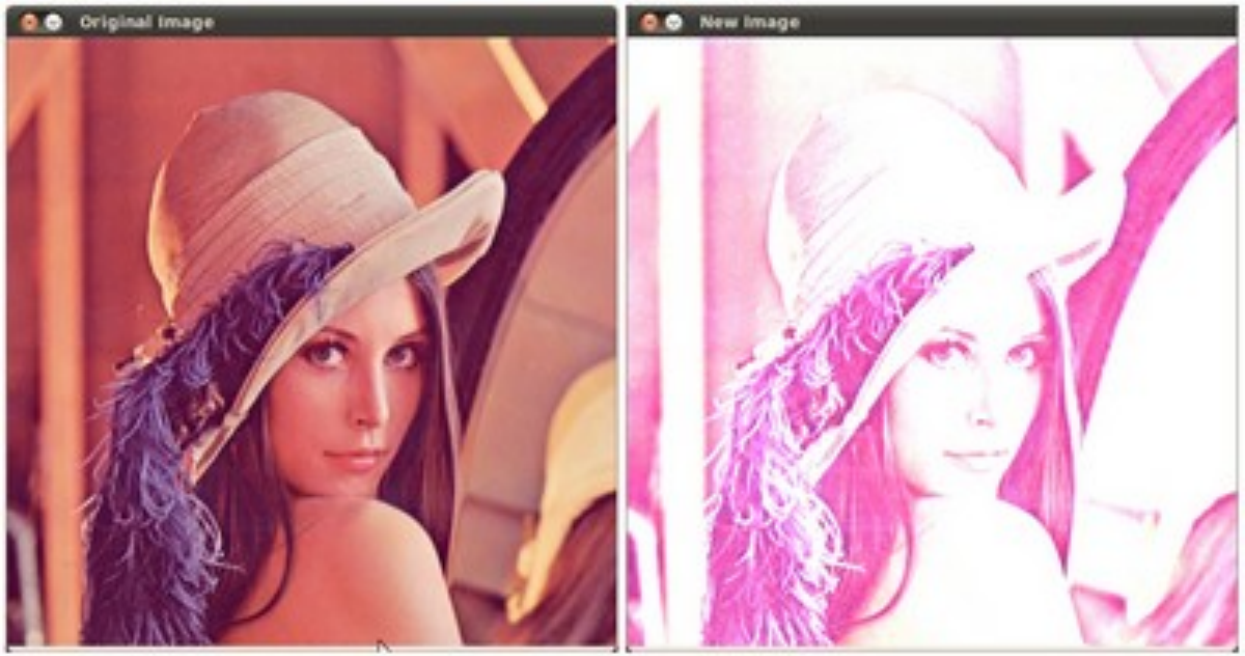
```
#include <cv.h>
#include <highgui.h>
#include <iostream>

using namespace cv;
double alpha; /**< Simple contrast control */
int beta;  /**< Simple brightness control */

int main( int argc, char** argv )
{
 /// Read image given by user
 Mat image = imread( argv[1] );
 Mat new_image = Mat::zeros( image.size(), image.type() );

 /// Initialize values
 std::cout<<" Basic Linear Transforms "<<std::endl;
 std::cout<<"-------------------------"<<std::endl;
 std::cout<<"* Enter the alpha value [1.0-3.0]: ";std::cin>>alpha;
 std::cout<<"* Enter the beta value [0-100]: "; std::cin>>beta;

 /// Do the operation new_image(i,j) = alpha*image(i,j) + beta
 for( int y = 0; y < image.rows; y++ )
    { for( int x = 0; x < image.cols; x++ )
         { for( int c = 0; c < 3; c++ )
              {
    new_image.at<Vec3b>(y,x)[c] =
       saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta );
              }
    }
```

Example: Brightness and contrast

```
 /// Do the operation new_image(i,j) = alpha*image(i,j) + beta
 for( int y = 0; y < image.rows; y++ )
    { for( int x = 0; x < image.cols; x++ )
         { for( int c = 0; c < 3; c++ )
              {
      new_image.at<Vec3b>(y,x)[c] =
         saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta );
             }
    }
    }

 /// Create Windows
 namedWindow("Original Image", 1);
 namedWindow("New Image", 1);

 /// Show stuff
 imshow("Original Image", image);
 imshow("New Image", new_image);

 /// Wait until user press some key
 waitKey();
 return 0;
}
```

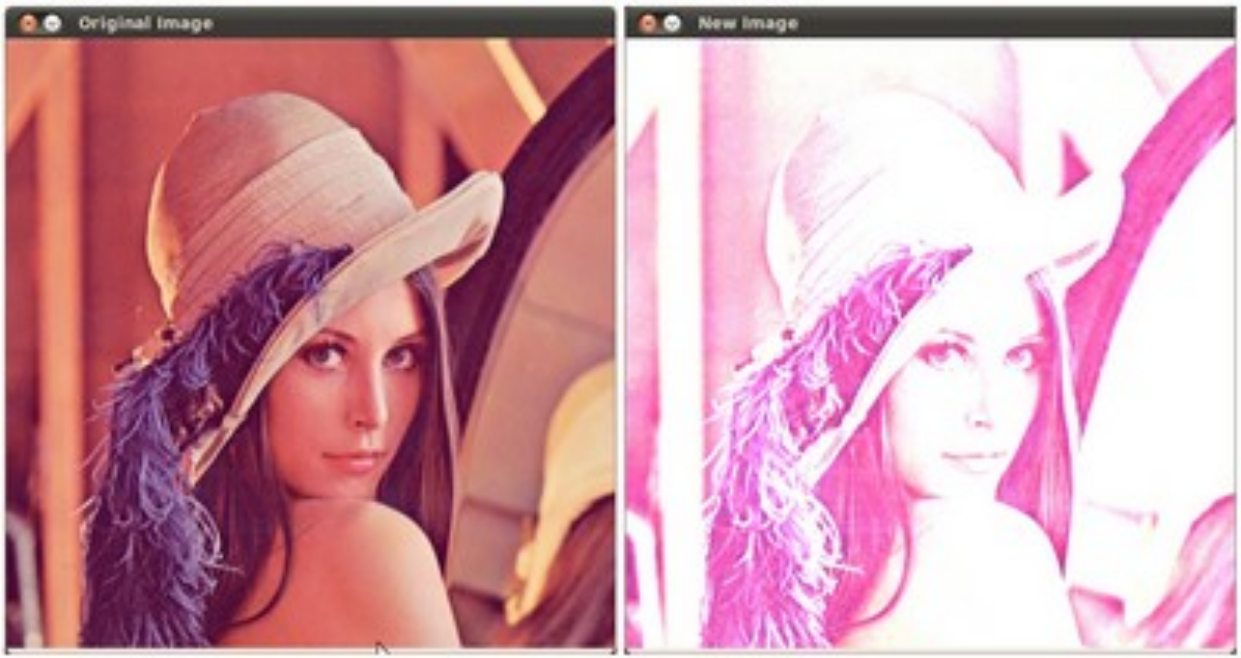
Example: Brightness and contrast

```
$ ./BasicLinearTransforms lena.jpg
Basic Linear Transforms
-------------------------
* Enter the alpha value [1.0-3.0]: 2.2
* Enter the beta value [0-100]: 50
```



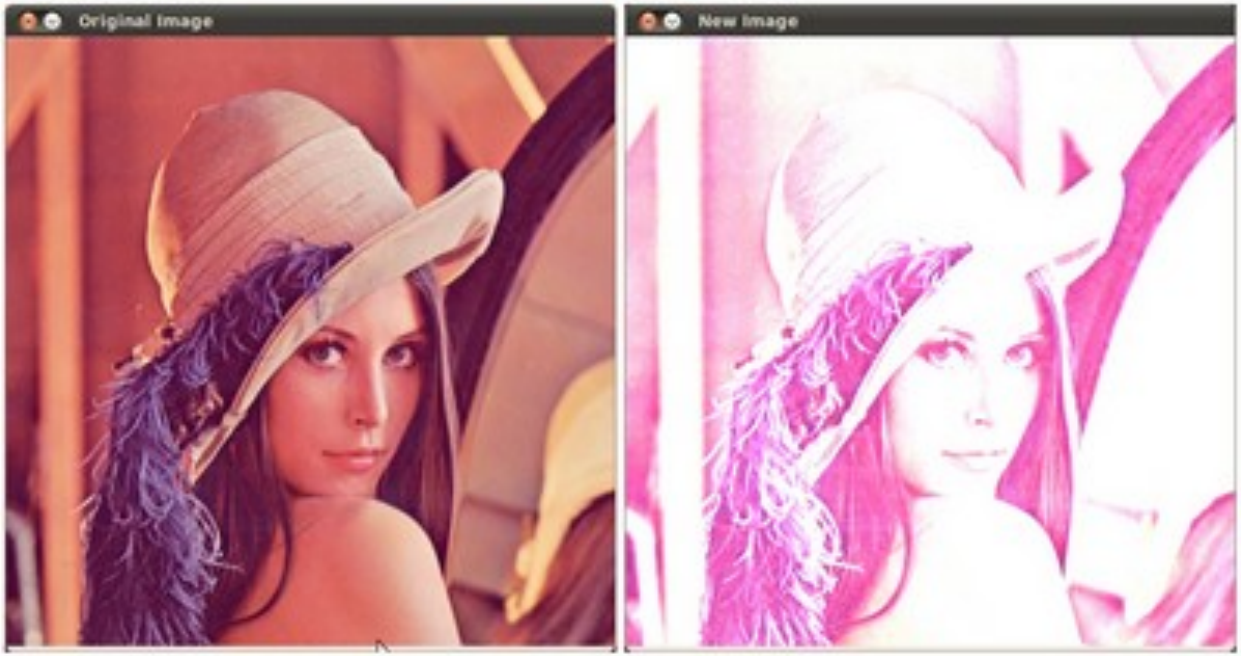
Example: Brightness and contrast

```
/// Do the operation new_image(i,j) = alpha*image(i,j) + beta
for( int y = 0; y < image.rows; y++ )
   { for( int x = 0; x < image.cols; x++ )
        { for( int c = 0; c < 3; c++ )
             {
     new_image.at<Vec3b>(y,x)[c] =
        saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta );
            }
   }
   }

/// Create Windows
namedWindow("Original Image", 1);
namedWindow("New Image", 1);

/// Show stuff
imshow("Original Image", image);
imshow("New Image", new_image);

/// Wait until user press some key
waitKey();
return 0;
}
```



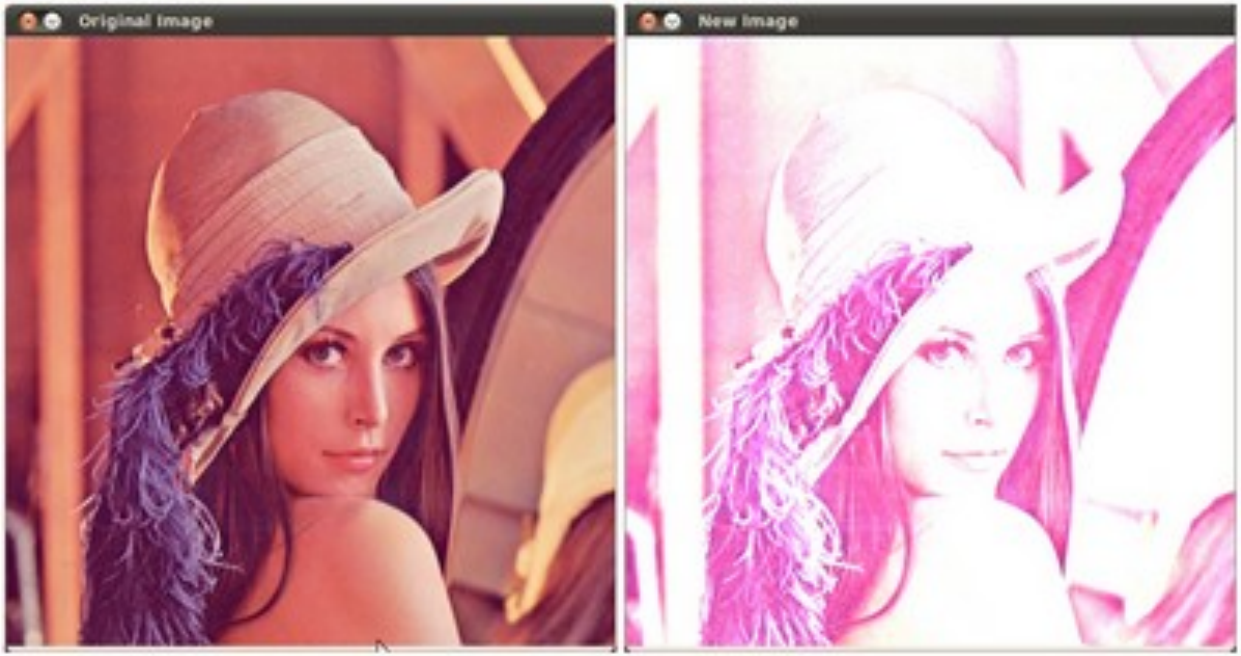
Example: Brightness and contrast

```
 /// Do the operation new_image(i,j) = alpha*image(i,j) + beta
 for( int y = 0; y < image.rows; y++ )
    { for( int x = 0; x < image.cols; x++ )
         { for( int c = 0; c < 3; c++ )
              {
      new_image.at<Vec3b>(y,x)[c] =
         saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta );
             }
    }
    }

 /// Create Windows
 namedWindow("Original Image", 1);
 namedWindow("New Image", 1);

 /// Show stuff
 imshow("Original Image", image);
 imshow("New Image", new_image);

 /// Wait until user press some key
 waitKey();
 return 0;
}
```



Example: Brightness and contrast

```
 /// Do the operation new_image(i,j) = alpha*image(i,j) + beta
   image.convertTo(new_image, -1, alpha, beta);

 /// Create Windows
 namedWindow("Original Image", 1);
 namedWindow("New Image", 1);

 /// Show stuff
 imshow("Original Image", image);
 imshow("New Image", new_image);

 /// Wait until user press some key
 waitKey();
 return 0;
}
```



Discrete Fourier transform

```
#include "opencv2/core/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
int main(int argc, char ** argv)
{
    const char* filename = argc >=2 ? argv[1] : "lena.jpg";

    Mat I = imread(filename, CV_LOAD_IMAGE_GRAYSCALE);
    if( I.empty())
        return -1;

    Mat padded;                            //expand input image to optimal size
    int m = getOptimalDFTSize( I.rows );
    int n = getOptimalDFTSize( I.cols ); // on the border add zero values
    copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));
```

Discrete Fourier transform

```
#include "opencv2/core/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
int main(int argc, char ** argv)
{
    const char* filename = argc >=2 ? argv[1] : "lena.jpg";

    Mat I = imread(filename, CV_LOAD_IMAGE_GRAYSCALE);
    if( I.empty())
        return -1;

    Mat padded;                            //expand input image to optimal size
    int m = getOptimalDFTSize( I.rows );
    int n = getOptimalDFTSize( I.cols ); // on the border add zero values
    copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));
```

Discrete Fourier transform

```
#include "opencv2/core/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
int main(int argc, char ** argv)
{
    const char* filename = argc >=2 ? argv[1] : "lena.jpg";

    Mat I = imread(filename, CV_LOAD_IMAGE_GRAYSCALE);
    if( I.empty())
        return -1;

    Mat padded;                            //expand input image to optimal size
    int m = getOptimalDFTSize( I.rows );
    int n = getOptimalDFTSize( I.cols ); // on the border add zero values
    copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));
```

Discrete Fourier transform

```
#include "opencv2/core/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
int main(int argc, char ** argv)
{
    const char* filename = argc >=2 ? argv[1] : "lena.jpg";

    Mat I = imread(filename, CV_LOAD_IMAGE_GRAYSCALE);
    if( I.empty())
        return -1;

    Mat padded;                            //expand input image to optimal size
    int m = getOptimalDFTSize( I.rows );
    int n = getOptimalDFTSize( I.cols ); // on the border add zero values
    copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));
```

Discrete Fourier transform

```
Mat padded;                            //expand input image to optimal size
int m = getOptimalDFTSize( I.rows );
int n = getOptimalDFTSize( I.cols ); // on the border add zero values
copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));

Mat planes[] = {Mat_<float>(padded), Mat::zeros(padded.size(), CV_32F)};
Mat complexI;
merge(planes, 2, complexI);         // Add to the expanded another plane with zeros

dft(complexI, complexI);            // this way the result may fit in the source matrix

// compute the magnitude and switch to logarithmic scale
// => log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
split(complexI, planes);                   // planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
magnitude(planes[0], planes[1], planes[0]);// planes[0] = magnitude
Mat magI = planes[0];

magI += Scalar::all(1);                    // switch to logarithmic scale
log(magI, magI);

// crop the spectrum, if it has an odd number of rows or columns
magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));
```

Discrete Fourier transform

```
Mat padded;                            //expand input image to optimal size
int m = getOptimalDFTSize( I.rows );
int n = getOptimalDFTSize( I.cols ); // on the border add zero values
copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));

Mat planes[] = {Mat_<float>(padded), Mat::zeros(padded.size(), CV_32F)};
Mat complexI;
merge(planes, 2, complexI);         // Add to the expanded another plane with zeros

dft(complexI, complexI);            // this way the result may fit in the source matrix

// compute the magnitude and switch to logarithmic scale
// => log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
split(complexI, planes);                   // planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
magnitude(planes[0], planes[1], planes[0]);// planes[0] = magnitude
Mat magI = planes[0];

magI += Scalar::all(1);                    // switch to logarithmic scale
log(magI, magI);

// crop the spectrum, if it has an odd number of rows or columns
magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));
```

Discrete Fourier transform

```
Mat padded;                            //expand input image to optimal size
int m = getOptimalDFTSize( I.rows );
int n = getOptimalDFTSize( I.cols ); // on the border add zero values
copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));

Mat planes[] = {Mat_<float>(padded), Mat::zeros(padded.size(), CV_32F)};
Mat complexI;
merge(planes, 2, complexI);         // Add to the expanded another plane with zeros

dft(complexI, complexI);            // this way the result may fit in the source matrix

// compute the magnitude and switch to logarithmic scale
// => log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
split(complexI, planes);                   // planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
magnitude(planes[0], planes[1], planes[0]);// planes[0] = magnitude
Mat magI = planes[0];

magI += Scalar::all(1);                    // switch to logarithmic scale
log(magI, magI);

// crop the spectrum, if it has an odd number of rows or columns
magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));
```

Discrete Fourier transform

```
// crop the spectrum, if it has an odd number of rows or columns
magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));

// rearrange the quadrants of Fourier image  so that the origin is at the image center
int cx = magI.cols/2;
int cy = magI.rows/2;

Mat q0(magI, Rect(0, 0, cx, cy));   // Top-Left - Create a ROI per quadrant
Mat q1(magI, Rect(cx, 0, cx, cy));  // Top-Right
Mat q2(magI, Rect(0, cy, cx, cy));  // Bottom-Left
Mat q3(magI, Rect(cx, cy, cx, cy)); // Bottom-Right

Mat tmp;                            // swap quadrants (Top-Left with Bottom-Right)
q0.copyTo(tmp);
q3.copyTo(q0);
tmp.copyTo(q3);

q1.copyTo(tmp);                     // swap quadrant (Top-Right with Bottom-Left)
q2.copyTo(q1);
tmp.copyTo(q2);

normalize(magI, magI, 0, 1, CV_MINMAX); // Transform the matrix with float values into a
                                        // viewable image form (float between values 0 and 1).

imshow("Input Image"       , I   );    // Show the result
imshow("spectrum magnitude", magI);
waitKey();

return 0;
```

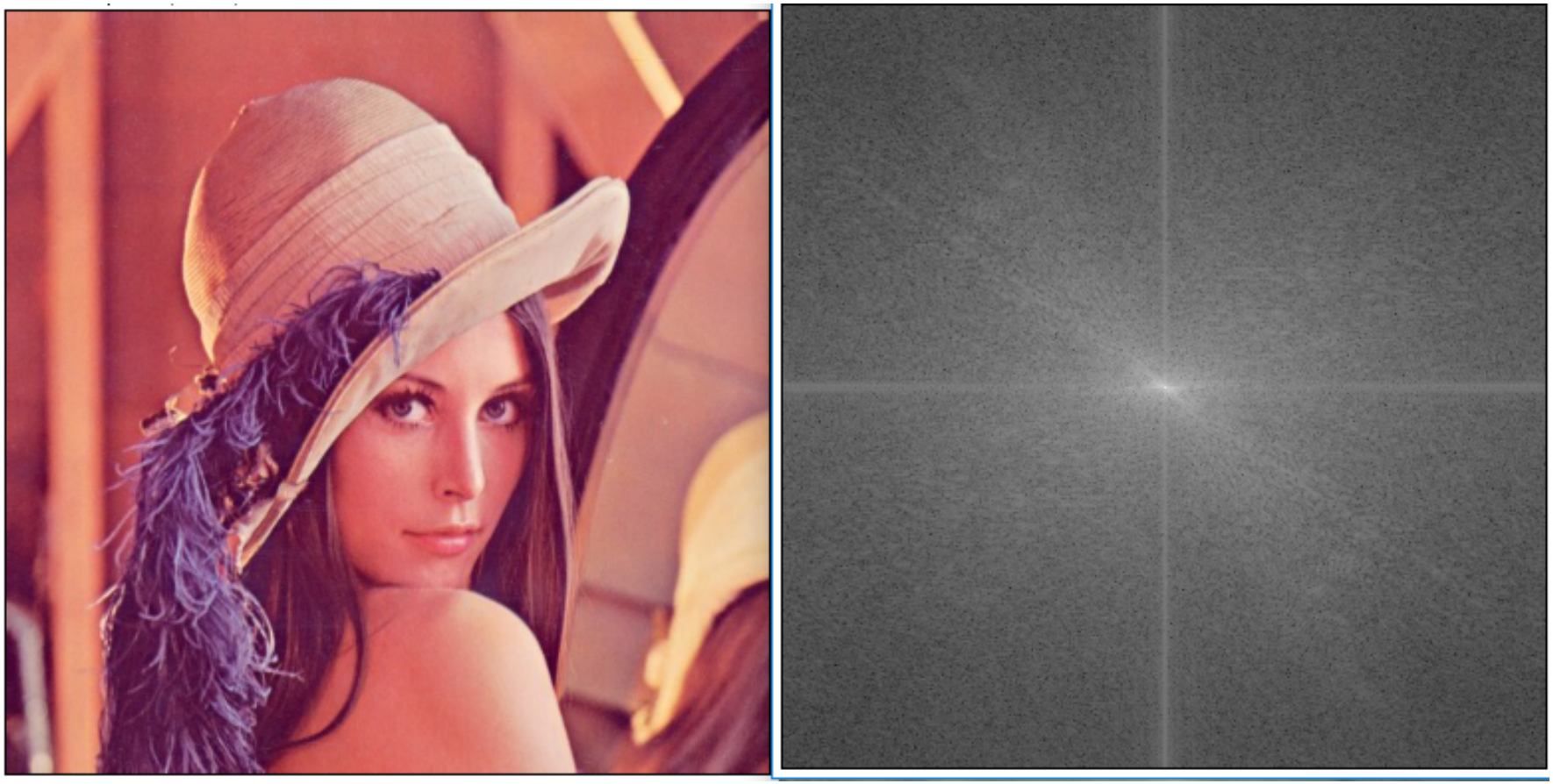
Discrete Fourier transform

```
// crop the spectrum, if it has an odd number of rows or columns
magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));

// rearrange the quadrants of Fourier image  so that the origin is at the image center
int cx = magI.cols/2;
int cy = magI.rows/2;

Mat q0(magI, Rect(0, 0, cx, cy));   // Top-Left - Create a ROI per quadrant
Mat q1(magI, Rect(cx, 0, cx, cy));  // Top-Right
Mat q2(magI, Rect(0, cy, cx, cy));  // Bottom-Left
Mat q3(magI, Rect(cx, cy, cx, cy)); // Bottom-Right

Mat tmp;                            // swap quadrants (Top-Left with Bottom-Right)
q0.copyTo(tmp);
q3.copyTo(q0);
tmp.copyTo(q3);

q1.copyTo(tmp);                     // swap quadrant (Top-Right with Bottom-Left)
q2.copyTo(q1);
tmp.copyTo(q2);

normalize(magI, magI, 0, 1, CV_MINMAX); // Transform the matrix with float values into a
                                        // viewable image form (float between values 0 and 1).

imshow("Input Image"       , I   );    // Show the result
imshow("spectrum magnitude", magI);
waitKey();

return 0;
```

Discrete Fourier transform

```
// crop the spectrum, if it has an odd number of rows or columns
magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));

// rearrange the quadrants of Fourier image  so that the origin is at the image center
int cx = magI.cols/2;
int cy = magI.rows/2;

Mat q0(magI, Rect(0, 0, cx, cy));   // Top-Left - Create a ROI per quadrant
Mat q1(magI, Rect(cx, 0, cx, cy));  // Top-Right
Mat q2(magI, Rect(0, cy, cx, cy));  // Bottom-Left
Mat q3(magI, Rect(cx, cy, cx, cy)); // Bottom-Right

Mat tmp;                            // swap quadrants (Top-Left with Bottom-Right)
q0.copyTo(tmp);
q3.copyTo(q0);
tmp.copyTo(q3);

q1.copyTo(tmp);                     // swap quadrant (Top-Right with Bottom-Left)
q2.copyTo(q1);
tmp.copyTo(q2);

normalize(magI, magI, 0, 1, CV_MINMAX); // Transform the matrix with float values into a
                                        // viewable image form (float between values 0 and 1).

imshow("Input Image"       , I   );    // Show the result
imshow("spectrum magnitude", magI);
waitKey();

return 0;
```

Discrete Fourier transform



Affine transformations

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace cv;
using namespace std;

/// Global variables
char* source_window = "Source image";
char* warp_window = "Warp";
char* warp_rotate_window = "Warp + Rotate";

/** @function main */
 int main( int argc, char** argv )
 {
   Point2f srcTri[3];
   Point2f dstTri[3];

   Mat rot_mat( 2, 3, CV_32FC1 );
   Mat warp_mat( 2, 3, CV_32FC1 );
   Mat src, warp_dst, warp_rotate_dst;

   /// Load the image
   src = imread( argv[1], 1 );
```

Affine transformations

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace cv;
using namespace std;

/// Global variables
char* source_window = "Source image";
char* warp_window = "Warp";
char* warp_rotate_window = "Warp + Rotate";

/** @function main */
 int main( int argc, char** argv )
 {
   Point2f srcTri[3];
   Point2f dstTri[3];

   Mat rot_mat( 2, 3, CV_32FC1 );
   Mat warp_mat( 2, 3, CV_32FC1 );
   Mat src, warp_dst, warp_rotate_dst;

   /// Load the image
   src = imread( argv[1], 1 );
```

Affine transformations

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace cv;
using namespace std;

/// Global variables
char* source_window = "Source image";
char* warp_window = "Warp";
char* warp_rotate_window = "Warp + Rotate";

/** @function main */
 int main( int argc, char** argv )
 {
   Point2f srcTri[3];
   Point2f dstTri[3];

   Mat rot_mat( 2, 3, CV_32FC1 );
   Mat warp_mat( 2, 3, CV_32FC1 );
   Mat src, warp_dst, warp_rotate_dst;

   /// Load the image
   src = imread( argv[1], 1 );
```

Affine transformations

```
/// Set the dst image the same type and size as src
warp_dst = Mat::zeros( src.rows, src.cols, src.type() );

/// Set your 3 points to calculate the  Affine Transform
srcTri[0] = Point2f( 0,0 );
srcTri[1] = Point2f( src.cols - 1, 0 );
srcTri[2] = Point2f( 0, src.rows - 1 );

dstTri[0] = Point2f( src.cols*0.0, src.rows*0.33 );
dstTri[1] = Point2f( src.cols*0.85, src.rows*0.25 );
dstTri[2] = Point2f( src.cols*0.15, src.rows*0.7 );

/// Get the Affine Transform
warp_mat = getAffineTransform( srcTri, dstTri );

/// Apply the Affine Transform just found to the src image
warpAffine( src, warp_dst, warp_mat, warp_dst.size() );

/** Rotating the image after Warp */

/// Compute a rotation matrix with respect to the center of the image
Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
double angle = -50.0;
double scale = 0.6;
```

Affine transformations

```
/// Set the dst image the same type and size as src
warp_dst = Mat::zeros( src.rows, src.cols, src.type() );

/// Set your 3 points to calculate the  Affine Transform
srcTri[0] = Point2f( 0,0 );
srcTri[1] = Point2f( src.cols - 1, 0 );
srcTri[2] = Point2f( 0, src.rows - 1 );

dstTri[0] = Point2f( src.cols*0.0, src.rows*0.33 );
dstTri[1] = Point2f( src.cols*0.85, src.rows*0.25 );
dstTri[2] = Point2f( src.cols*0.15, src.rows*0.7 );

/// Get the Affine Transform
warp_mat = getAffineTransform( srcTri, dstTri );

/// Apply the Affine Transform just found to the src image
warpAffine( src, warp_dst, warp_mat, warp_dst.size() );

/** Rotating the image after Warp */

/// Compute a rotation matrix with respect to the center of the image
Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
double angle = -50.0;
double scale = 0.6;
```

Affine transformations

```
/// Set the dst image the same type and size as src
warp_dst = Mat::zeros( src.rows, src.cols, src.type() );

/// Set your 3 points to calculate the  Affine Transform
srcTri[0] = Point2f( 0,0 );
srcTri[1] = Point2f( src.cols - 1, 0 );
srcTri[2] = Point2f( 0, src.rows - 1 );

dstTri[0] = Point2f( src.cols*0.0, src.rows*0.33 );
dstTri[1] = Point2f( src.cols*0.85, src.rows*0.25 );
dstTri[2] = Point2f( src.cols*0.15, src.rows*0.7 );

/// Get the Affine Transform
warp_mat = getAffineTransform( srcTri, dstTri );

/// Apply the Affine Transform just found to the src image
warpAffine( src, warp_dst, warp_mat, warp_dst.size() );

/** Rotating the image after Warp */

/// Compute a rotation matrix with respect to the center of the image
Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
double angle = -50.0;
double scale = 0.6;
```

Affine transformations

```
/// Set the dst image the same type and size as src
warp_dst = Mat::zeros( src.rows, src.cols, src.type() );

/// Set your 3 points to calculate the  Affine Transform
srcTri[0] = Point2f( 0,0 );
srcTri[1] = Point2f( src.cols - 1, 0 );
srcTri[2] = Point2f( 0, src.rows - 1 );

dstTri[0] = Point2f( src.cols*0.0, src.rows*0.33 );
dstTri[1] = Point2f( src.cols*0.85, src.rows*0.25 );
dstTri[2] = Point2f( src.cols*0.15, src.rows*0.7 );

/// Get the Affine Transform
warp_mat = getAffineTransform( srcTri, dstTri );

/// Apply the Affine Transform just found to the src image
warpAffine( src, warp_dst, warp_mat, warp_dst.size() );

/** Rotating the image after Warp */

/// Compute a rotation matrix with respect to the center of the image
Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
double angle = -50.0;
double scale = 0.6;
```

Affine transformations

```
/// Set the dst image the same type and size as src
warp_dst = Mat::zeros( src.rows, src.cols, src.type() );

/// Set your 3 points to calculate the  Affine Transform
srcTri[0] = Point2f( 0,0 );
srcTri[1] = Point2f( src.cols - 1, 0 );
srcTri[2] = Point2f( 0, src.rows - 1 );

dstTri[0] = Point2f( src.cols*0.0, src.rows*0.33 );
dstTri[1] = Point2f( src.cols*0.85, src.rows*0.25 );
dstTri[2] = Point2f( src.cols*0.15, src.rows*0.7 );

/// Get the Affine Transform
warp_mat = getAffineTransform( srcTri, dstTri );

/// Apply the Affine Transform just found to the src image
warpAffine( src, warp_dst, warp_mat, warp_dst.size() );

/** Rotating the image after Warp */

/// Compute a rotation matrix with respect to the center of the image
Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
double angle = -50.0;
double scale = 0.6;
```

Affine transformations

```
  /// Compute a rotation matrix with respect to the center of the image
  Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
  double angle = -50.0;
  double scale = 0.6;

  /// Get the rotation matrix with the specifications above
  rot_mat = getRotationMatrix2D( center, angle, scale );

  /// Rotate the warped image
  warpAffine( warp_dst, warp_rotate_dst, rot_mat, warp_dst.size() );

  /// Show what you got
  namedWindow( source_window, CV_WINDOW_AUTOSIZE );
  imshow( source_window, src );

  namedWindow( warp_window, CV_WINDOW_AUTOSIZE );
  imshow( warp_window, warp_dst );

  namedWindow( warp_rotate_window, CV_WINDOW_AUTOSIZE );
  imshow( warp_rotate_window, warp_rotate_dst );

  /// Wait until user exits the program
  waitKey(0);

  return 0;
}
```

Affine transformations

```
  /// Compute a rotation matrix with respect to the center of the image
  Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
  double angle = -50.0;
  double scale = 0.6;

  /// Get the rotation matrix with the specifications above
  rot_mat = getRotationMatrix2D( center, angle, scale );

  /// Rotate the warped image
  warpAffine( warp_dst, warp_rotate_dst, rot_mat, warp_dst.size() );

  /// Show what you got
  namedWindow( source_window, CV_WINDOW_AUTOSIZE );
  imshow( source_window, src );

  namedWindow( warp_window, CV_WINDOW_AUTOSIZE );
  imshow( warp_window, warp_dst );

  namedWindow( warp_rotate_window, CV_WINDOW_AUTOSIZE );
  imshow( warp_rotate_window, warp_rotate_dst );

  /// Wait until user exits the program
  waitKey(0);

  return 0;
}
```

Affine transformations

```
  /// Compute a rotation matrix with respect to the center of the image
  Point center = Point( warp_dst.cols/2, warp_dst.rows/2 );
  double angle = -50.0;
  double scale = 0.6;

  /// Get the rotation matrix with the specifications above
  rot_mat = getRotationMatrix2D( center, angle, scale );

  /// Rotate the warped image
  warpAffine( warp_dst, warp_rotate_dst, rot_mat, warp_dst.size() );

  /// Show what you got
  namedWindow( source_window, CV_WINDOW_AUTOSIZE );
  imshow( source_window, src );

  namedWindow( warp_window, CV_WINDOW_AUTOSIZE );
  imshow( warp_window, warp_dst );

  namedWindow( warp_rotate_window, CV_WINDOW_AUTOSIZE );
  imshow( warp_rotate_window, warp_rotate_dst );

  /// Wait until user exits the program
  waitKey(0);

  return 0;
}
```

Affine transformations

Affine transformations

Affine transformations

Histogram equalization

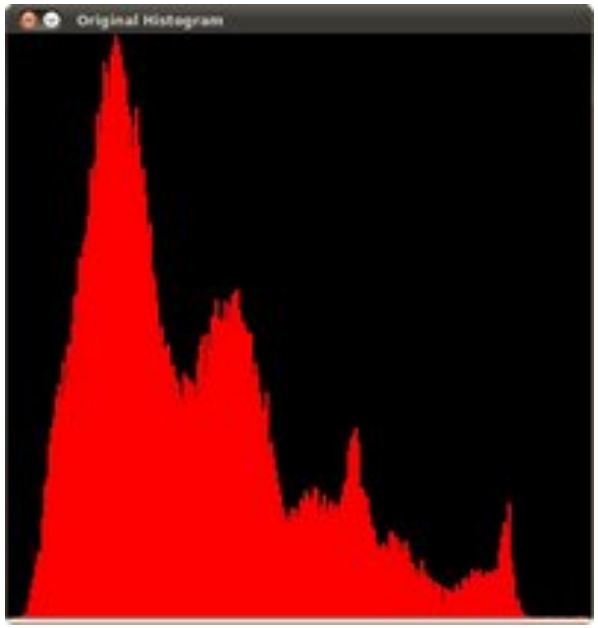
```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace cv;
using namespace std;

/** @function main */
int main( int argc, char** argv )
{
  Mat src, dst;

  char* source_window = "Source image";
  char* equalized_window = "Equalized Image";

  /// Load image
  src = imread( argv[1], 1 );

  if( !src.data )
    { cout<<"Usage: ./Histogram_Demo <path_to_image>"<<endl;
      return -1;}

  /// Convert to grayscale
  cvtColor( src, src, CV_BGR2GRAY );
```

Histogram equalization

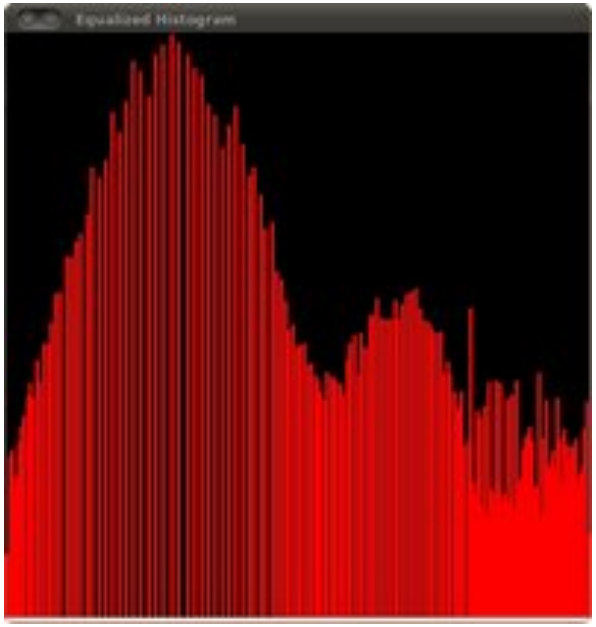
```
  /// Convert to grayscale
  cvtColor( src, src, CV_BGR2GRAY );

  /// Apply Histogram Equalization
  equalizeHist( src, dst );

  /// Display results
  namedWindow( source_window, CV_WINDOW_AUTOSIZE );
  namedWindow( equalized_window, CV_WINDOW_AUTOSIZE );

  imshow( source_window, src );
  imshow( equalized_window, dst );

  /// Wait until user exits the program
  waitKey(0);

  return 0;
}
```

Histogram equalization

Histogram equalization



Histogram equalization



Histogram equalization