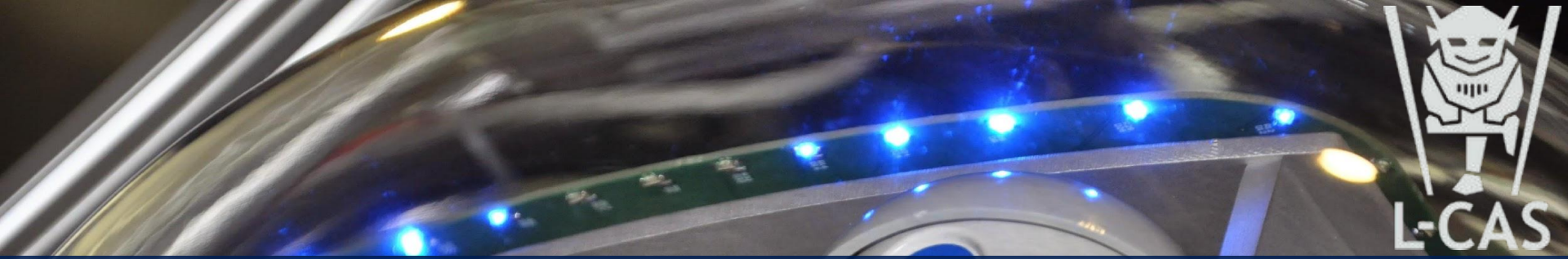


CMP9767M - ROBOT PROGRAMMING

Dr Grzegorz Cielniak

https://staff.lincoln.ac.uk/gcielniak

Synopsis

1. Module introduction and assessment presentation
2. The ROS & Python primer
3. Kinematic chains, tf, sensor and actuator frames
4. **Robot vision**
5. ROS systems, the big picture
6. Localisation
7. Navigation and obstacle avoidance
8. - Reading Week -
9. Mapping
10. Topological navigation
11. Software engineering reloaded
12. ROS advanced and to the future (or trip to Riseholme)
13. Manipulation (moveit)

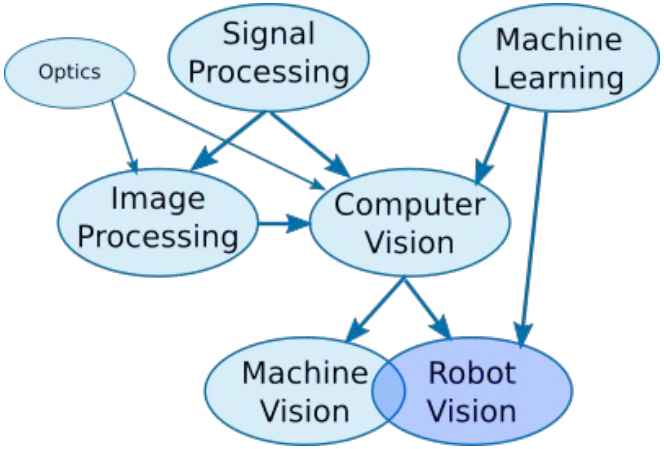
Robot Vision

A combination of hardware (cameras/sensors) and software (algorithms) which enable robots to “see”

The richest source of information about the robot’s environment

Today:

- Part I - Images
- Part II - Image Geometry
- Part III - Point Clouds



Owen-Hill (2016) Robot Vision vs Computer Vision: What's the Difference?

Part I - Images

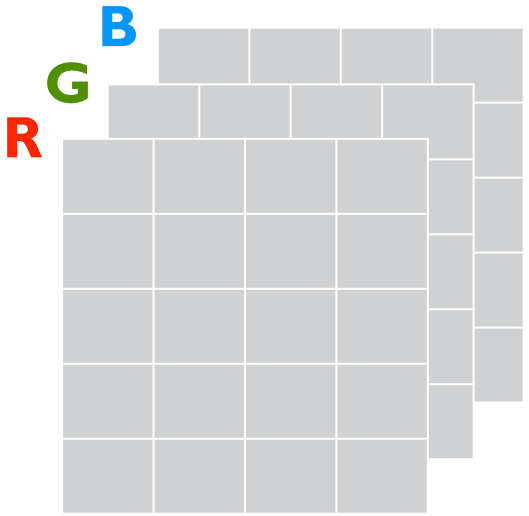
A set of pixels arranged into a 2D matrix (matrices)

Each pixel corresponds to light intensity

Discretised intensity levels (e.g. 8b, 16b)

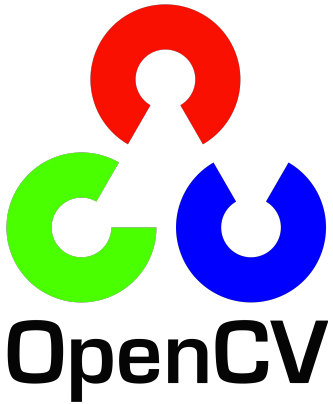
Mono-chrome, colour and multi-spectral sensors

A source of information for image processing and computer vision



OpenCV (Open Source Computer Vision Library)

- initially launched by Intel in 1999
- +2500 optimized CV&ML algorithms - “gold standard”
- C++ with multi-language bindings (**Python**, Java and MATLAB), supports **ROS**
- multi-os, parallel hardware
- commercial, research and gov.; +47k users, 18m downloads
- applications: object/face/action recognition, camera/object/eye tracking, 3D reconstruction, image stitching & indexing, etc.
- Intro to OpenCV with Python https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html



OpenCV - Basic Operations

```
from cv2 import namedWindow, imread, imshow
from cv2 import waitKey, destroyAllWindows, startWindowThread
from cv2 import blur, Canny, circle

# declare windows you want to display
namedWindow("original")
namedWindow("blur")
namedWindow("canny")

# this is always needed to run the GUI thread
startWindowThread()

# load the input image
img1 = imread('../blofeld.jpg')
# display the image
imshow("original", img1)

# create a new blurred image
img2 = blur(img1, (7, 7))
# draw on the image
circle(img2, (100, 100), 30, (255, 0, 255), 5)
imshow("blur", img2)

# Canny is an algorith for edge detection
img3 = Canny(img1, 10, 200)
imshow("canny", img3)
```

GUI operations
- reading and showing images
- handling keyboard, mouse
- drawing

Image processing
- applying existing algorithms
- per pixel access read/write

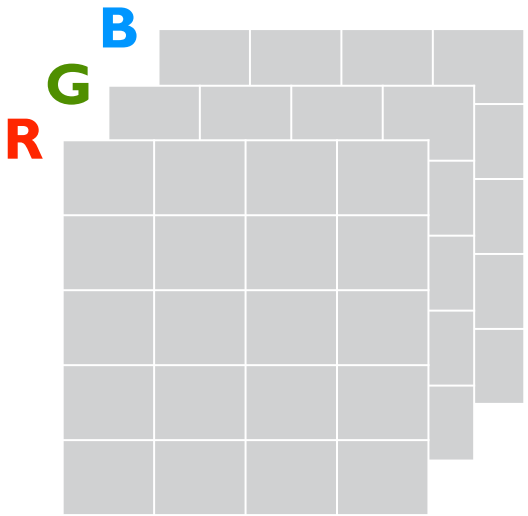
https://github.com/LCAS/teaching/blob/kinetic/cmp3103m-code-fragments/scripts/opencv_intro.py

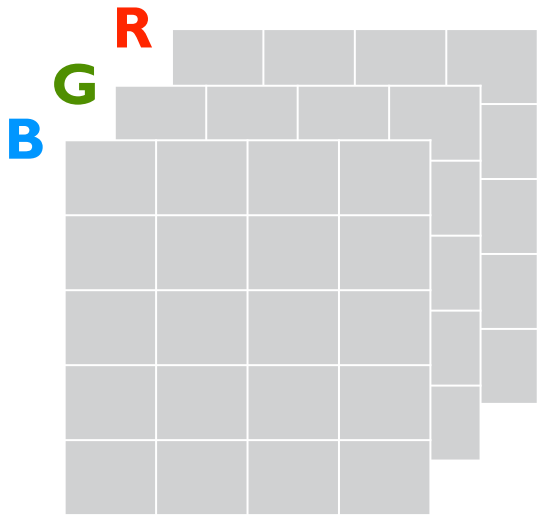
OpenCV - Image Representation

- Matrices like in Matlab!
  - based on NumPy

- not RGB, but BGR!

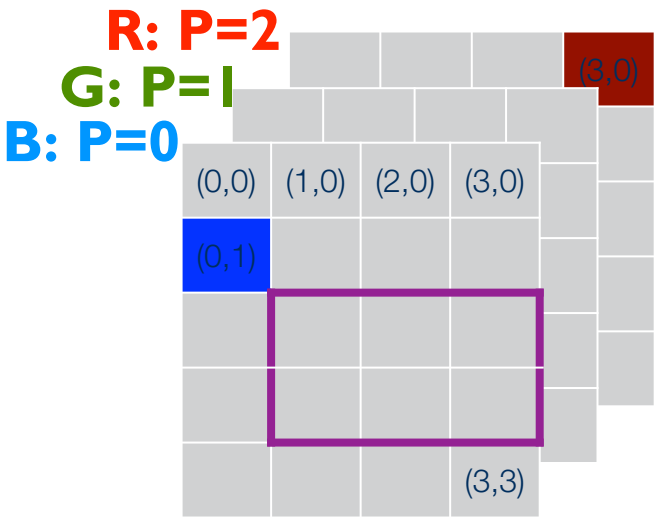
https://www.learnopencv.com/why-does-opencv-use-bgr-color-format/





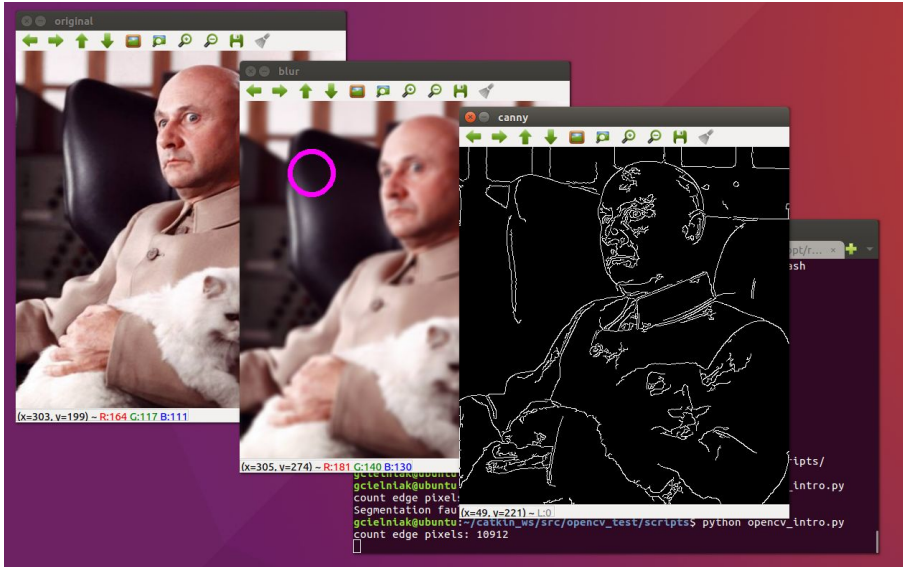
OpenCV - Image Representation

- Pixel access:
  - img[Y, X, P]
  - img[3, 0, 2]
  - img[0, 1, 0]
- Ranges:
  - img[1:3, 2:3, 0]



OpenCV - Pixel Access

```
# the shape gives you the dimensions
h = img3.shape[0]
w = img3.shape[1]

# loop over the image, pixel by pixel
count = 0

# a slow way to iterate over the pixels
for y in range(0, h):
    for x in range(0, w):
        # threshold the pixel
        if img3[y, x] > 0:
            count += 1
print('count edge pixels: %d' % count)

# wait key is also always needed to sync the GUI threads
waitKey(0)

# good practice to tidy up at the end
destroyAllWindows()
```



Getting Image Streams from a ROS topic

- OpenCV and ROS play nicely
- A dedicated CvBridge to help you getting and processing image from the robot
  - `CvBridge::imgmsg_to_cv2()` and `CvBridge::cv2_to_imgmsg()`

- subscribe to Image topic
- convert to OpenCV in callback
- process the image using OpenCV

```
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
```

http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython

OpenCvBridge

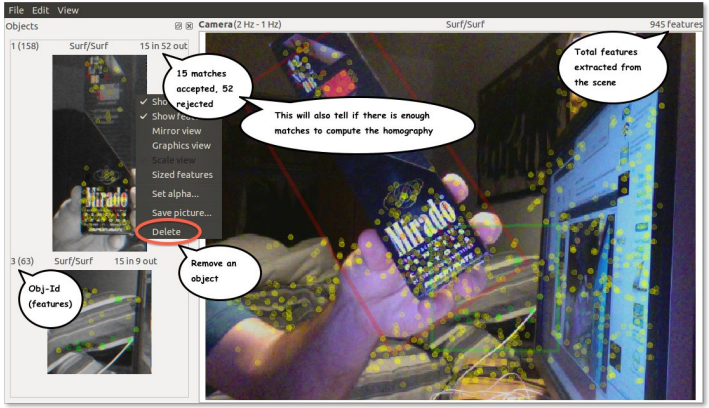
```python
#!/usr/bin/env python

import rospy
from cv2 import namedWindow, cvtColor, imshow
from cv2 import destroyAllWindows, startWindowThread
from cv2 import COLOR_BGR2GRAY, waitKey
from cv2 import blur, Canny
from numpy import mean
from sensor_msgs.msg import Image
from cv_bridge import CvBridge


class image_converter:

    def __init__(self):

        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("/cam/image_raw",
                                          Image, self.callback)
        #self.image_sub = rospy.Subscriber("/turtlebot_1/camera/rgb/image_raw",Image,self.callback)


    def callback(self, data):
        namedWindow("Image window")
        namedWindow("blur")
        namedWindow("canny")
        cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")

        gray_img = cvtColor(cv_image, COLOR_BGR2GRAY)
        print mean(gray_img)
        img2 = blur(gray_img, (3, 3))
        imshow("blur", img2)
        img3 = Canny(gray_img, 10, 200)
        imshow("canny", img3)

        imshow("Image window", cv_image)
        waitKey(1)


startWindowThread()
rospy.init_node('image_converter')
ic = image_converter()
rospy.spin()

destroyAllWindows()
```

https://github.com/LCAS/teaching/blob/kinetic/cmp3103m-code-fragments/scripts/opencv_bridge.py

Useful Vision Commands

- `roslaunch video_stream_opencv camera.launch video_stream_provider:=/dev/video0 camera_name:=camera visualize:=true`
- `roslaunch video_stream_opencv camera.launch video_stream_provider:=video.mp4 camera_name:=camera visualize:=true`
- `rqt_image_view`
- `rviz`
- `rostopic hz`

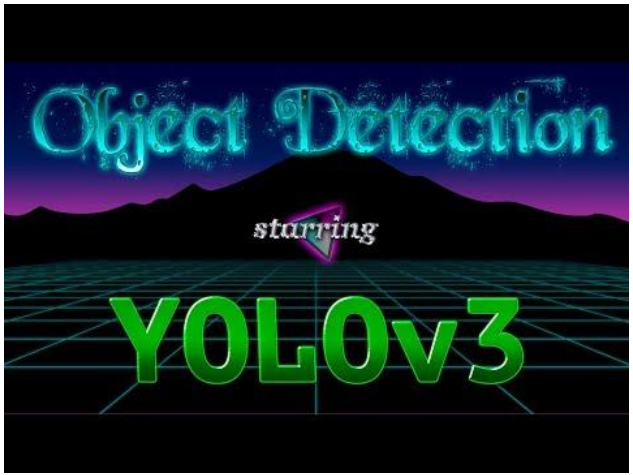
- check http://wiki.ros.org/video_stream_opencv
- check http://wiki.ros.org/opencv_apps
- check http://wiki.ros.org/find_object_2d



YOLO - You Only Look Once

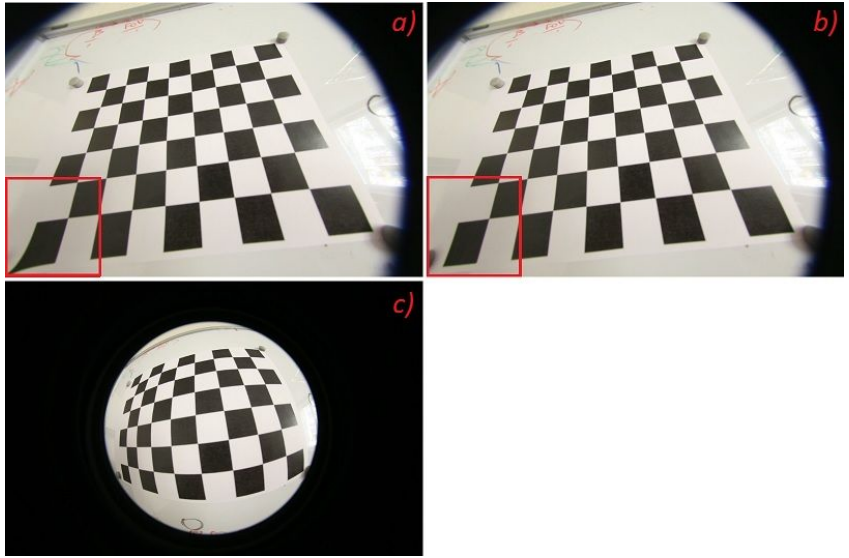
- Project website: https://pjreddie.com/darknet/yolo/
- Original paper: https://pjreddie.com/media/files/papers/yolo_1.pdf

- `roslaunch darknet_ros darknet_ros.launch`



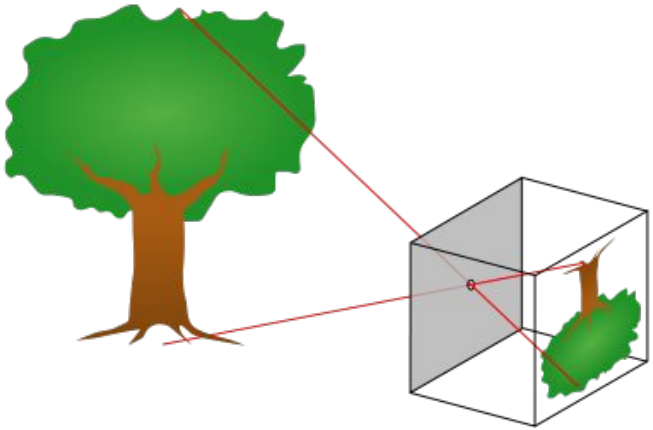
Part II - Image Geometry

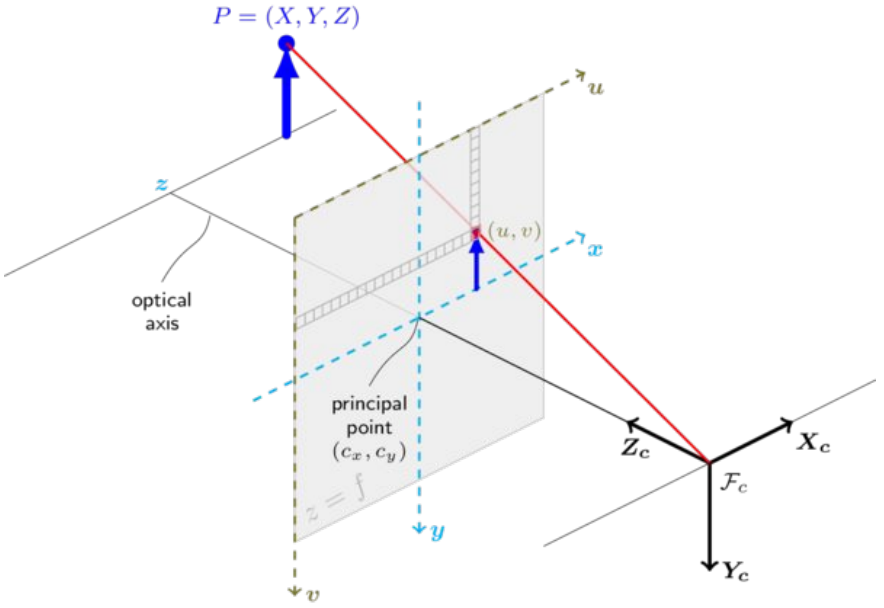
- Base material: OpenCV documentation: “Camera Calibration and 3D Reconstruction”
- “Multiple View Geometry in Computer Vision”, Hartley & Zisserman, 2003



Camera Model - Pinhole Camera

- Perspective 3D to 2D projection
- No lens modelling

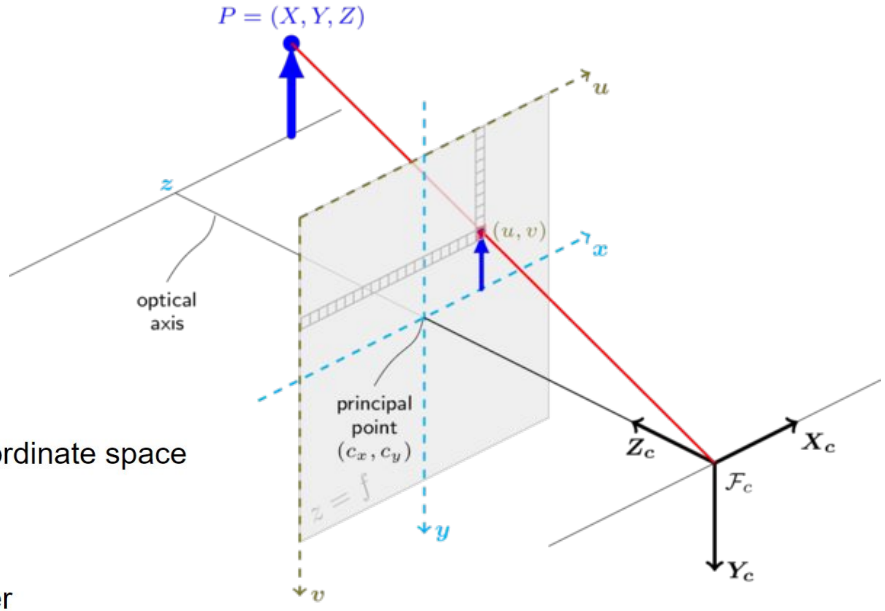




Camera Model - Pinhole Camera

$$s\, m' = A[R|t]M'$$

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{intrinsics}} \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}}_{\text{extrinsics}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$



- (X, Y, Z) are the coordinates of a 3D point in the world coordinate space
- (u, v) are the coordinates of the projection point in pixels
- A is a camera matrix, or a matrix of intrinsic parameters
- (cx, cy) is a principal point that is usually at the image center
- fx, fy are the focal lengths expressed in pixel units.

World to image projection

camera coordinates

world coordinates

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$

$$y' = y/z$$

projection

$$u = f_x * x' + c_x$$

$$v = f_y * y' + c_y$$

pixel coordinates

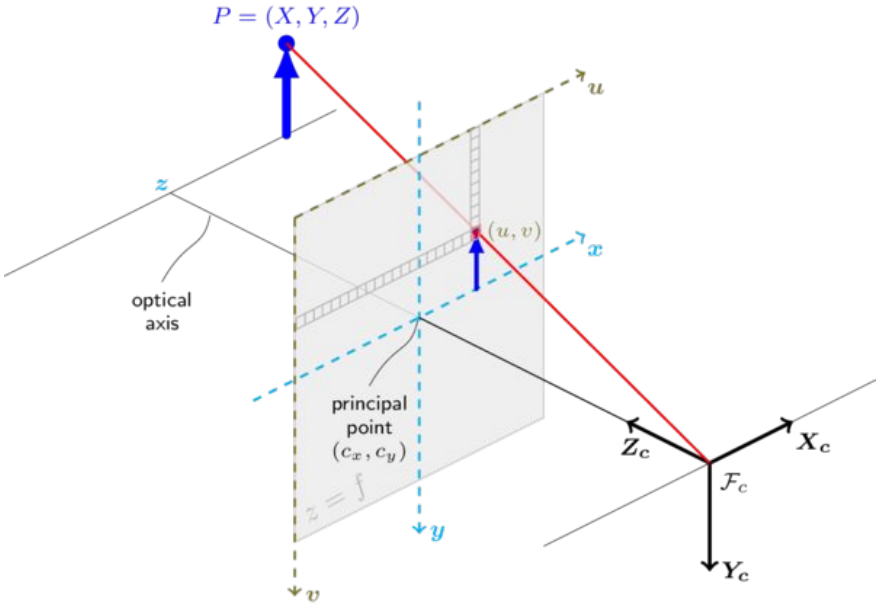
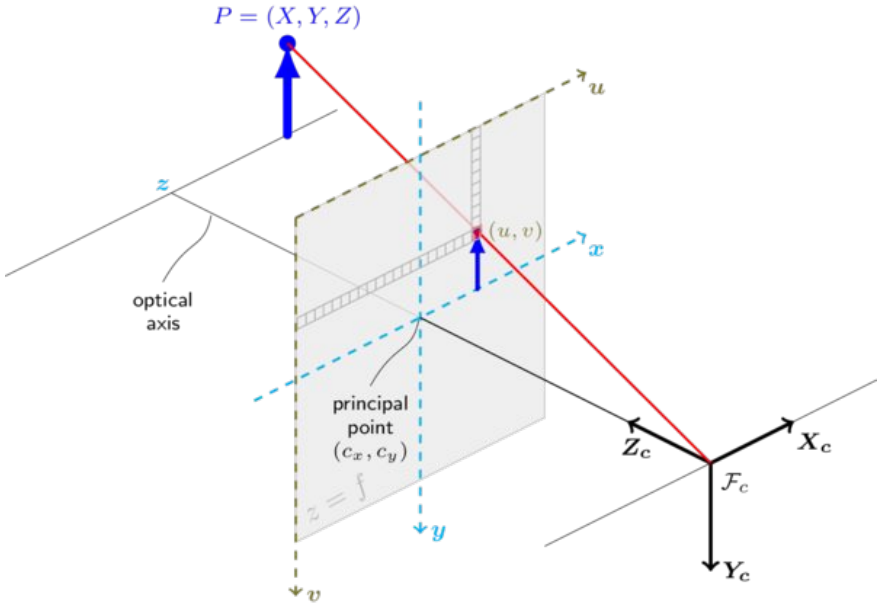


Image to world re-projection

- re-projection results in a line since the depth (z) is unknown



camera coordinates

pixel coordinates

$$x = (u - c_x)/f_x$$
$$y = (v - c_y)/f_y$$
$$z = 1$$

camera_info

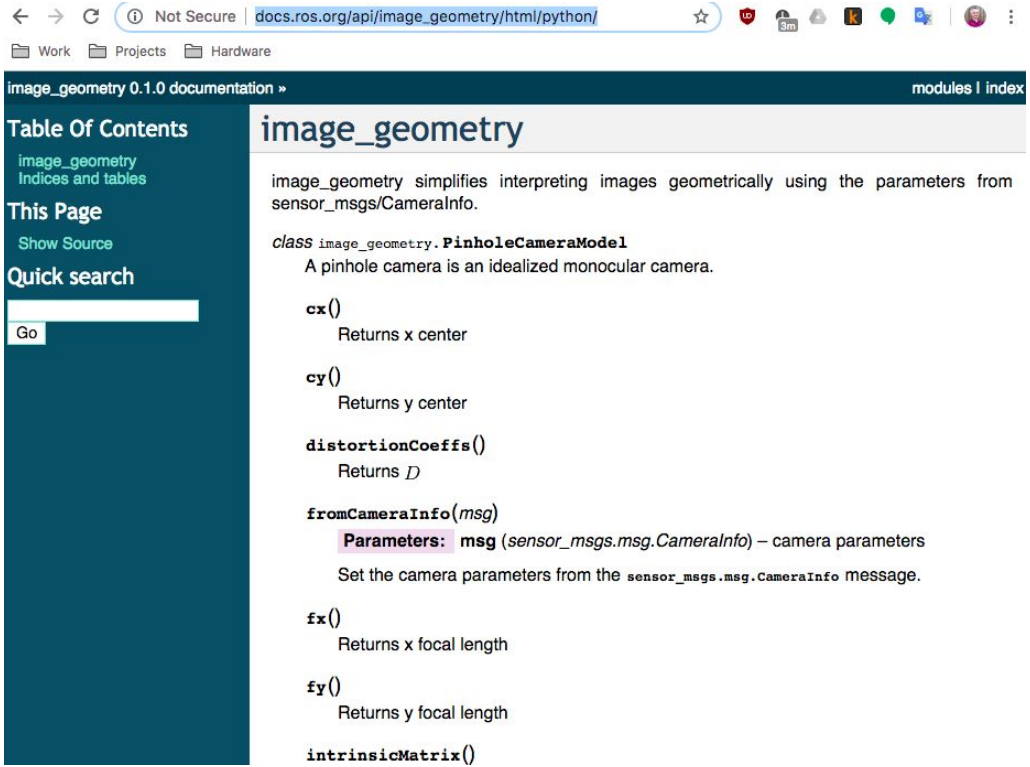
```
---
header:
  seq: 199
  stamp:
    secs: 51
    nsecs: 809000000
  frame_id: "thorvald_001/kinect2_depth_optical_frame"
height: 424
width: 512
distortion_model: "plumb_bob"
D: [0.0, 0.0, 0.0, 0.0, 0.0]
K: [365.8474052823454, 0.0, 256.5, 0.0, 365.8474052823454, 212.5, 0.0, 0.0, 1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [365.8474052823454, 0.0, 256.5, -0.0, 0.0, 365.8474052823454, 212.5, 0.0, 0.0
, 0.0, 1.0, 0.0]
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
  do_rectify: False
---
```

http://wiki.ros.org/image_pipeline/CameraInfo


```
# The distortion model used. Supported models are listed in
# sensor_msgs/distortion_models.h. For most cameras, "plumb_bob" - a
# simple model of radial and tangential distortion - is sufficient.
string distortion_model

# The distortion parameters, size depending on the distortion model.
# For "plumb_bob", the 5 parameters are: (k1, k2, t1, t2, k3).
float64[] D

# Intrinsic camera matrix for the raw (distorted) images.
#     [fx  0 cx]
# K = [ 0 fy cy]
#     [ 0  0  1]
# Projects 3D points in the camera coordinate frame to 2D pixel
# coordinates using the focal lengths (fx, fy) and principal point
# (cx, cy).
float64[9]  K # 3x3 row-major matrix

# Rectification matrix (stereo cameras only)
# A rotation matrix aligning the camera coordinate system to the ideal
# stereo image plane so that epipolar lines in both stereo images are
# parallel.
float64[9]  R # 3x3 row-major matrix

# Projection/camera matrix
#     [fx'  0  cx' Tx]
# P = [ 0  fy' cy' Ty]
#     [ 0   0   1   0]
# By convention, this matrix specifies the intrinsic (camera) matrix
#  of the processed (rectified) image. That is, the left 3x3 portion
#  is the normal camera intrinsic matrix for the rectified image.
# It projects 3D points in the camera coordinate frame to 2D pixel
#  coordinates using the focal lengths (fx', fy') and principal point
#  (cx', cy') - these may differ from the values in K.
# For monocular cameras, Tx = Ty = 0. Normally, monocular cameras will
#  also have R = the identity and P[1:3,1:3] = K.
# For a stereo pair, the fourth column [Tx Ty 0]' is related to the
#  position of the optical center of the second camera in the first
#  camera's frame. We assume Tz = 0 so both cameras are in the same
#  stereo image plane. The first camera always has Tx = Ty = 0. For
#  the right (second) camera of a horizontal stereo pair, Ty = 0 and
#  Tx = -fx' * B, where B is the baseline between the cameras.
# Given a 3D point [X Y Z]', the projection (x, y) of the point onto
#  the rectified image is given by:
#  [u v w]' = P * [X Y Z 1]'
#         x = u / w
#         y = v / w
#  This holds for both images of a stereo pair.
float64[12] P # 3x4 row-major matrix
```

image_geometry

- Helper functions to streamline image geometry operations
- http://wiki.ros.org/image_geometry
- http://docs.ros.org/api/image_geometry/html/python/



Intrinsics

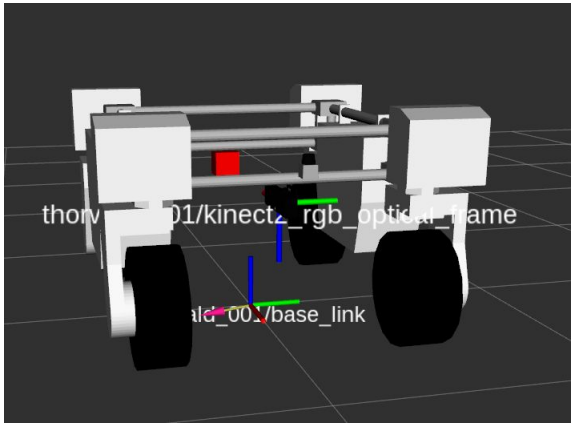
From camera
to pixel coordinates

```python
def __init__(self):

    self.bridge = CvBridge()

    self.camera_info_sub = rospy.Subscriber('/thorvald_001/kinect2_camera/hd/camera_info',
        CameraInfo, self.camera_info_callback)

    rospy.Subscriber("/thorvald_001/kinect2_camera/hd/image_color_rect",
        Image, self.image_callback)

def image_callback(self, data):
    if not self.camera_model:
        return

    #project a point in camera coordinates into the pixel coordinates
    uv = self.camera_model.project3dToPixel((0,0,0.5))

    print 'Pixel coordinates: ', uv
    print ''

    try:
        cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
    except CvBridgeError as e:
        print(e)

    cv2.circle(cv_image, (int(uv[0]),int(uv[1])), 10, 255, -1)

    #resize for visualisation
    cv_image_s = cv2.resize(cv_image, (0,0), fx=0.5, fy=0.5)

    cv2.imshow("Image window", cv_image_s)
    cv2.waitKey(1)

def camera_info_callback(self, data):
    self.camera_model = image_geometry.PinholeCameraModel()
    self.camera_model.fromCameraInfo(data)
    self.camera_info_sub.unregister() #Only subscribe once
```

UNIVERSITY OF LINCOLN

Extrinsics

From robot
to pixel coordinates



```
def __init__(self):

    self.bridge = CvBridge()

    self.camera_info_sub = rospy.Subscriber('/thorvald_001/kinect2_camera/hd/camera_info',
        CameraInfo, self.camera_info_callback)

    rospy.Subscriber("/thorvald_001/kinect2_camera/hd/image_color_rect",
        Image, self.image_callback)

    self.tf_listener = tf.TransformListener()

def image_callback(self, data):
    if not self.camera_model:
        return

    #show the camera pose with respect to the robot's pose (base_link)
    (trans, rot) = self.tf_listener.lookupTransform('thorvald_001/base_link',
        'thorvald_001/kinect2_rgb_optical_frame', rospy.Time())
    print 'Robot to camera transform:', 'T ', trans, 'R ', rot

    #define a point in robot (base_link) coordinates
    p_robot = PoseStamped()
    p_robot.header.frame_id = "thorvald_001/base_link"
    p_robot.pose.orientation.w = 1.0
    #specify a point on the ground just below the camera
    p_robot.pose.position.x = 0.45
    p_robot.pose.position.y = 0.0
    p_robot.pose.position.z = 0.0
    p_camera = self.tf_listener.transformPose('thorvald_001/kinect2_rgb_optical_frame', p_robot)
    print 'Point in the camera coordinates'
    print p_camera.pose.position

    uv = self.camera_model.project3dToPixel((p_camera.pose.position.x,p_camera.pose.position.y,
        p_camera.pose.position.z))

    print 'Pixel coordinates: ', uv
    print ''
```

UNIVERSITY OF LINCOLN

Lens Distortion - image rectification

Radial: *k*
Tangential: *p (q)*
Prismatic: *s*

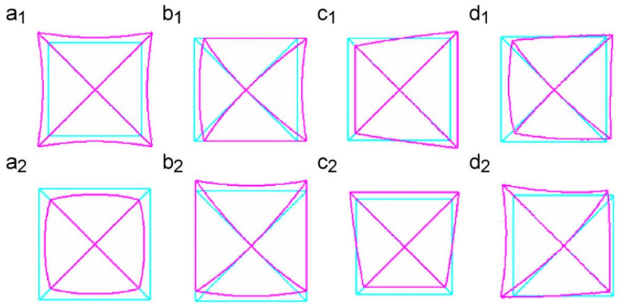


Fig. 1. Effects of the coefficients in the improved lens distortion model. The square in cyan is the distortion-free image. (a_1) and (a_2) show the effects of coefficient k_1, where $k_1 = 3.0 \times 10^{-6}$ for (a_1), and $k_1 = -3.0 \times 10^{-6}$ for (a_2); (b_1) and (b_2) show the effects of coefficients q_1 and q_2, where $q_1 = 5.0 \times 10^{-4}$, $q_2 = 0$ for (b_1), and $q_1 = 0$, $q_2 = -5.0 \times 10^{-4}$ for (b_2); (c_1) and (c_2) show the effects of coefficients p_1 and p_2, where $p_1 = 8.0 \times 10^{-4}$, $p_2 = 0$ for (c_1) and $p_1 = 0$, $p_2 = -8.0 \times 10^{-4}$ for (c_2); (d_1) and (d_2) show the effects of coefficients k_1, p_1, p_2, q_1 and q_2 together, where $k_1 = -1.0 \times 10^{-6}$, $q_1 = 3.0 \times 10^{-4}$, $q_2 = 1.0 \times 10^{-4}$, $p_1 = 3.0 \times 10^{-4}$, $p_2 = 1.0 \times 10^{-4}$ for (d_1), and $k_1 = 1.0 \times 10^{-6}$, $q_1 = -3.0 \times 10^{-4}$, $q_2 = -3.0 \times 10^{-4}$, $p_1 = -3.0 \times 10^{-4}$, $p_2 = 1.0 \times 10^{-4}$ for (d_2).

projected

rectified

$$x'' = x' \frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + 2p_1x'y' + p_2(r^2+2x'^2) + s_1r^2 + s_2r^4$$

$$y'' = y' \frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + p_1(r^2+2y'^2) + 2p_2x'y' + s_3r^2 + s_4r^4$$

$$\text{where} \quad r^2 = x'^2 + y'^2$$

$$u = f_x * x'' + c_x$$

$$v = f_y * y'' + c_y$$

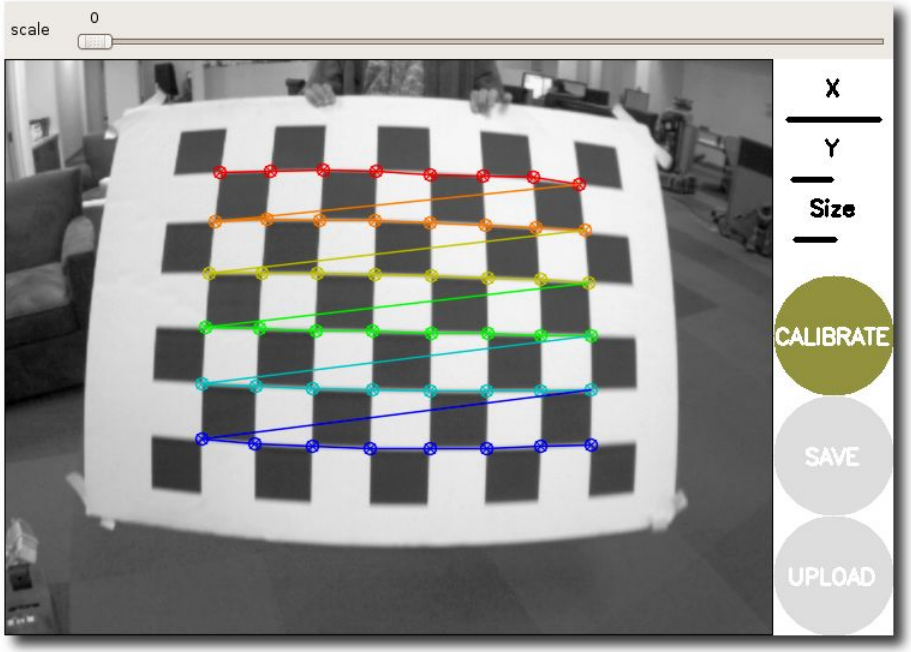
Wang et al., 2007, “A new calibration model of camera lens distortion”

camera_calibration

http://wiki.ros.org/camera_calibration

http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration

https://youtu.be/yAYqt3RpT6c



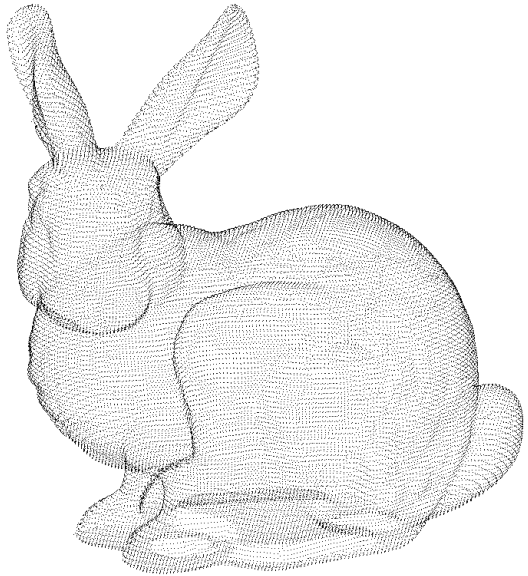
Part II - Point Clouds

Set of data points in 3D space

Each point has x,y,z coordinates expressed in the sensor's frame of reference

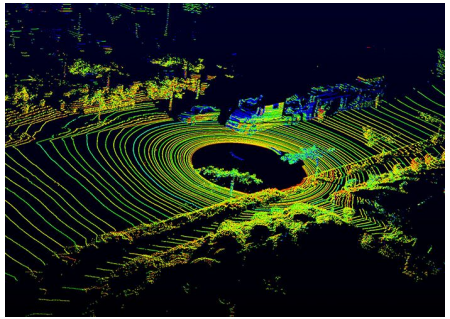
Geometry is easier to work with when compared to standard cameras

Very popular in robotics due to affordable sensors

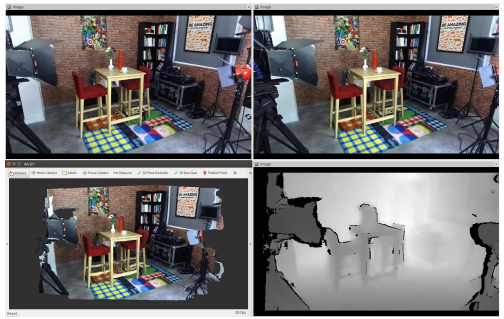


Point Cloud Sensors in Robotics

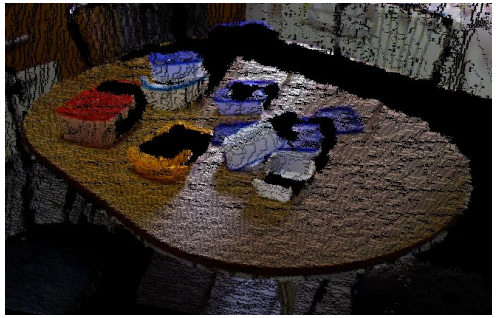
laser scanners



stereo camera



RGBD camera



A comparison of 3D sensing technology: Borghi (2019), Human Pose Understanding on Depth Maps, pages 5-18

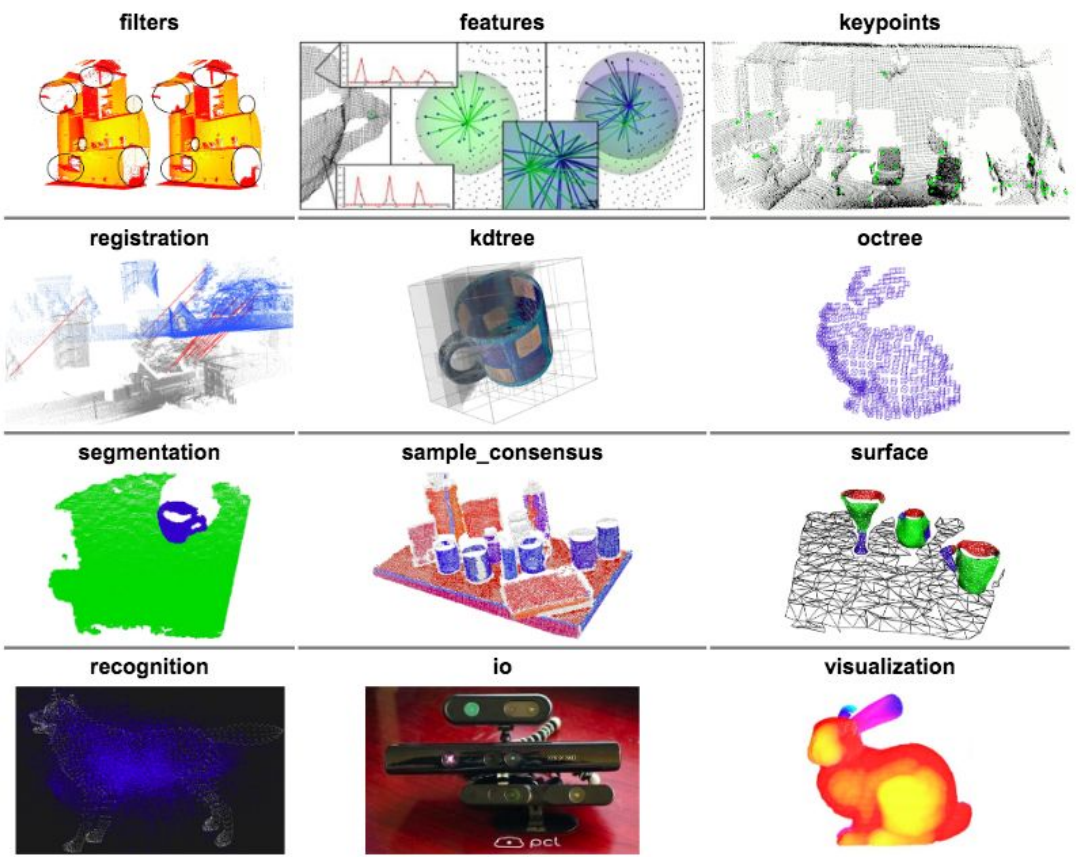
Point Cloud Library

OpenCV-equivalent for point clouds

Open source, multi-os, integrated into ROS

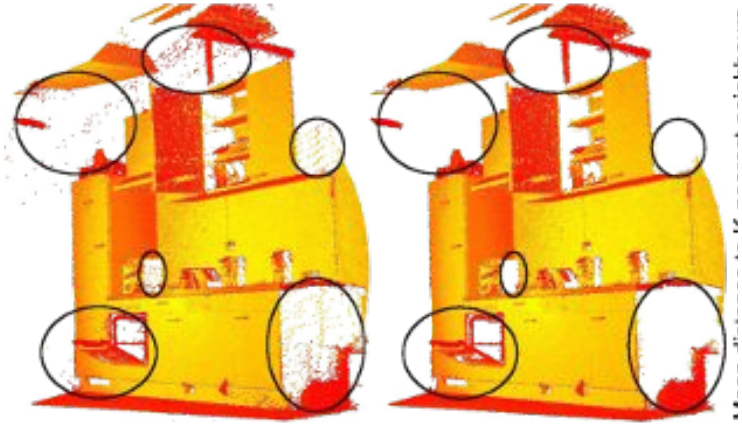
Base material for this part: PCL documentation

pointcloudlibrary

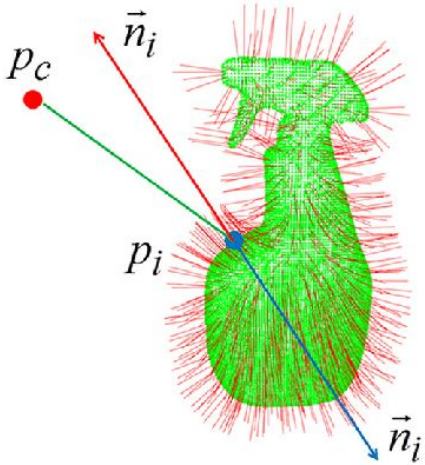


PCL - functionality

filters - sparse outlier removal



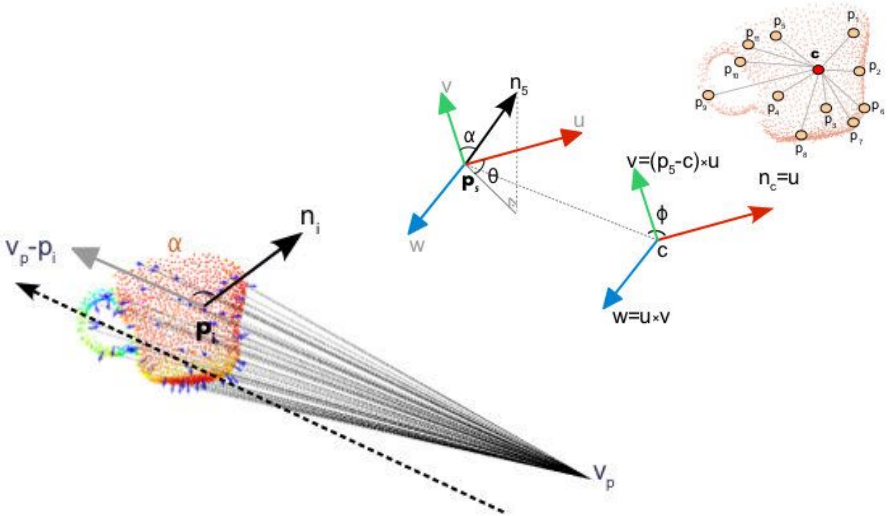
surface normal estimation

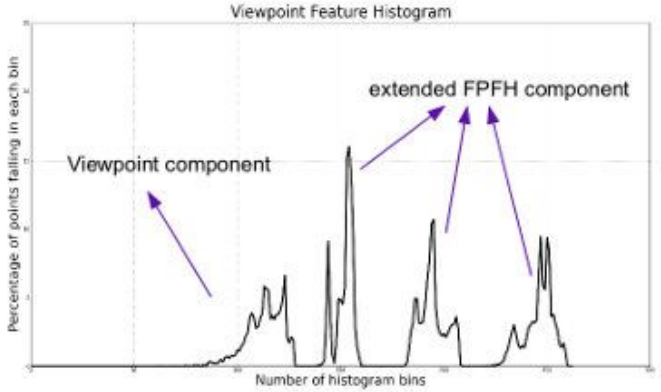


PCL - functionality

descriptors - Viewpoint Feature Histogram

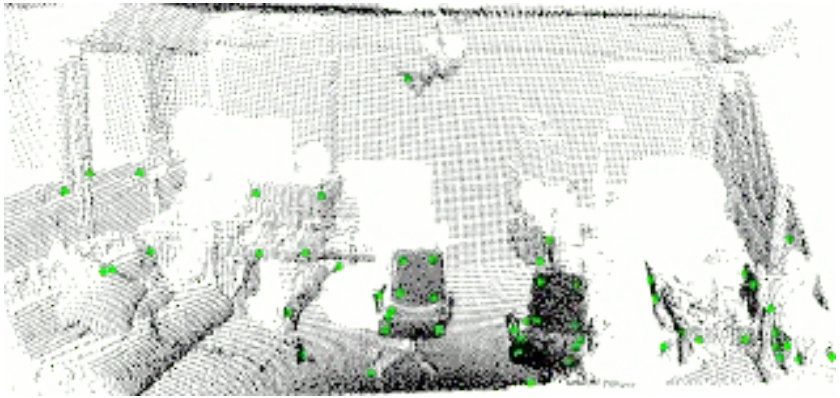
http://pointclouds.org/documentation/tutorials/vfh_estimation.php



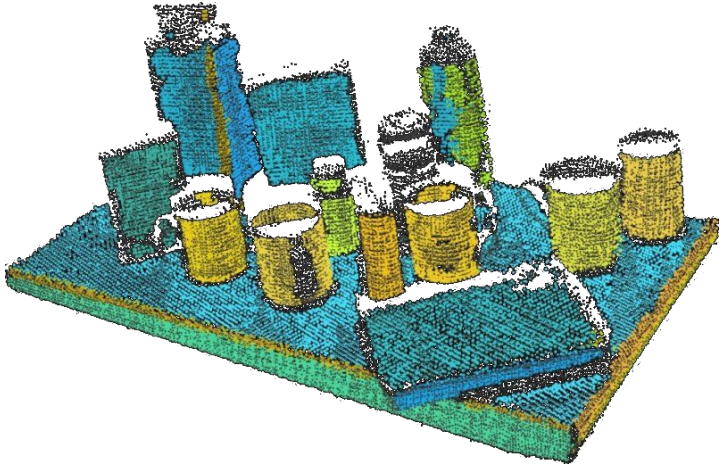


PCL - functionality

keypoint detectors



segmentation - sample consensus



PCL - functionality

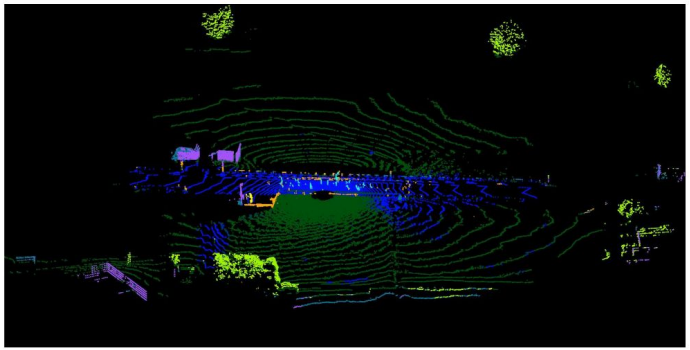
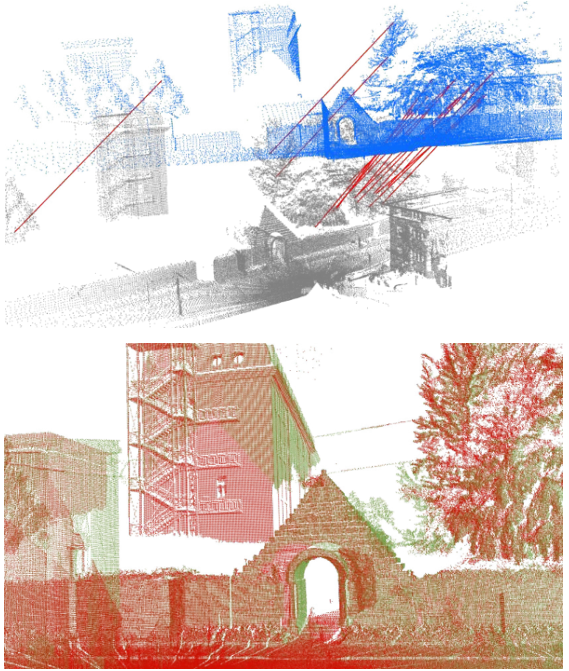


Fig. 1. An indicative point cloud from the *Semantic3d.net* dataset ('sg27 station 1') as labeled by PointNet RGB.

Zaganidis et al., 2018 "*Integrating Deep Semantic Segmentation into 3D Point Cloud Registration*" http://eprints.lincoln.ac.uk/32390/

scan registration -
alignment of two views of the same scene



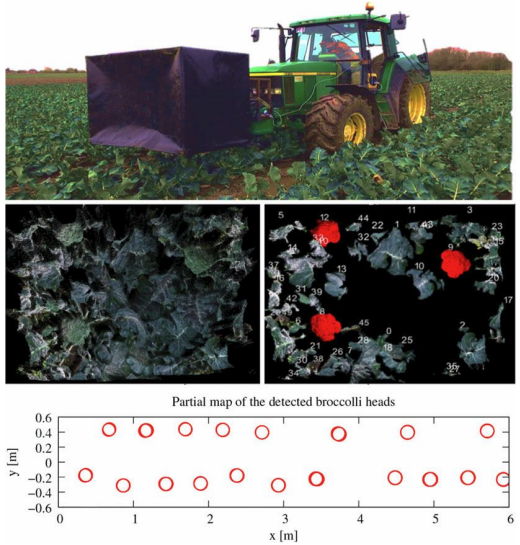
Point clouds for object detection - L-CAS research

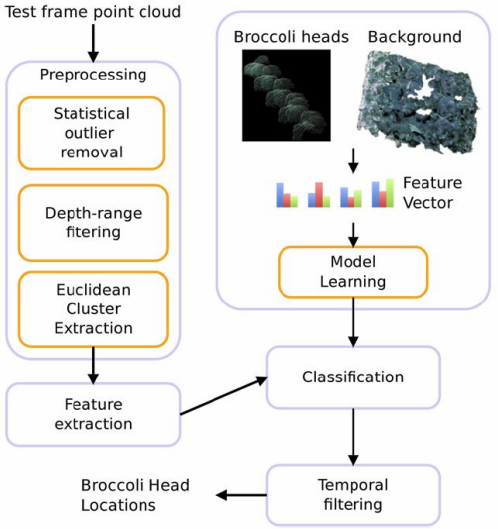
A PCL-based 3D processing pipeline

Kusumam et al. (2017) “3D-vision based detection, localization, and sizing of broccoli heads in the field”

Paper: http://eprints.lincoln.ac.uk/27782/

Video: https://youtu.be/MUTddzcWERs





Point clouds in ROS

data types

- sensor_msgs::PointCloud

  The first adopted point cloud message in ROS. Contains x, y and z points (all floats) as well as multiple channels; each channel has a string name and an array of float values. This served the initial **point_cloud_mapping** package in ROS (never released) and most of the visualization and data producers/consumers were based on this format prior to ROS 1.0.

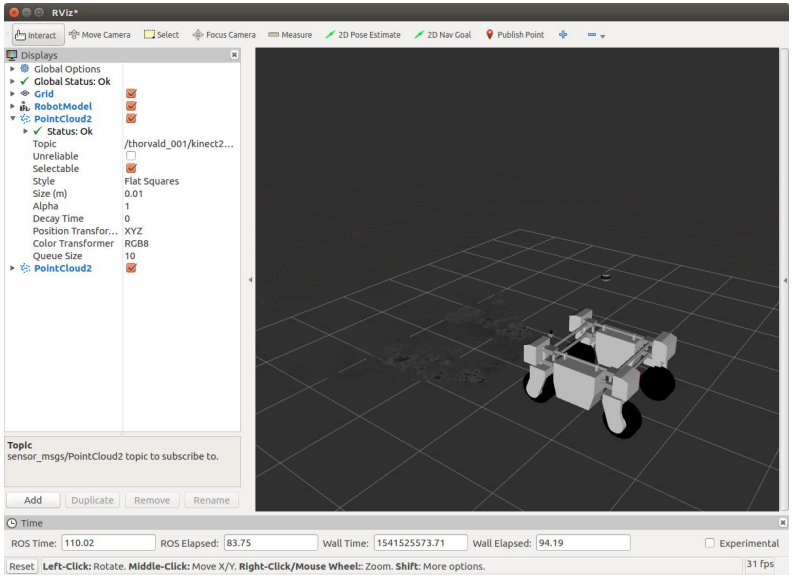
- sensor_msgs::PointCloud2

  The newly revised ROS point cloud message (and currently the *de facto* standard in **PCL**), now representing arbitrary n-D (n dimensional) data. Point values can now be of any primitive data types (int, float, double, etc), and the message can be specified as 'dense', with height and width values, giving the data a 2D structure, e.g. to correspond to an image of the same region in space. For more information on the rationale behind the new structure, see: PCL_March_2010.pdf and pcl_icra2010.pdf

- pcl::PointCloud<T>

  The core point cloud class in the PCL library; can be templated on any of the Point types listed in point_types.h or a user-defined type. This class has a similar structure to the PointCloud2 message type, including a header. Converting between the message class and the point cloud template class is straightforward (see below), and most methods in the PCL library accept objects of both types. Still, it's better to work with this template class in your point cloud processing node, rather than with the message object, among other reasons because you can work with the individual points as objects rather than having to work with their raw data.

visualisation in rviz



sensor_msgs.PointCloud2

```
# This message holds a collection of N-dimensional points, which may
# contain additional information such as normals, intensity, etc. The
# point data is stored as a binary blob, its layout described by the
# contents of the "fields" array.

# The point cloud data may be organized 2d (image-like) or 1d
# (unordered). Point clouds organized as 2d images may be produced by
# camera depth sensors such as stereo or time-of-flight.

# Time of sensor data acquisition, and the coordinate frame ID (for 3d
# points).
Header header

# 2D structure of the point cloud. If the cloud is unordered, height is
# 1 and width is the length of the point cloud.
uint32 height
uint32 width

# Describes the channels and their layout in the binary data blob.
PointField[] fields

bool    is_bigendian # Is this data bigendian?
uint32  point_step   # Length of a point in bytes
uint32  row_step     # Length of a row in bytes
uint8[] data         # Actual point data, size is (row_step*height)

bool is_dense        # True if there are no invalid points
```

sensor_msgs.PointField

```
# This message holds the description of one point entry in the
# PointCloud2 message format.
uint8 INT8    = 1
uint8 UINT8   = 2
uint8 INT16   = 3
uint8 UINT16  = 4
uint8 INT32   = 5
uint8 UINT32  = 6
uint8 FLOAT32 = 7
uint8 FLOAT64 = 8

string name      # Name of field
uint32 offset    # Offset from start of point struct
uint8  datatype  # Datatype enumeration, see above
uint32 count     # How many elements in the field
```

1.3 Common PointCloud2 field names

Because field names are generic in the new PointCloud2 message, here's the list of commonly used names within PCL:

- *x* - the **X** Cartesian coordinate of a point (float32)
- *y* - the **Y** Cartesian coordinate of a point (float32)
- *z* - the **Z** Cartesian coordinate of a point (float32)
- *rgb* - the RGB (24-bit packed) color at a point (uint32)
- *rgba* - the A-RGB (32-bit packed) color at a point (uint32), the field name is unfortunately misleading
- *normal_x* - the first component of the normal direction vector at a point (float32)
- *normal_y* - the second component of the normal direction vector at a point (float32)
- *normal_z* - the third component of the normal direction vector at a point (float32)
- *curvature* - the surface curvature change estimate at a point (float32)
- *j1* - the first moment invariant at a point (float32)
- *j2* - the second moment invariant at a point (float32)
- *j3* - the third moment invariant at a point (float32)

pcl_ros

http://wiki.ros.org/pcl_ros

conversion nodes

- bag_to_pcd
- convert_pcd_to_image
- convert_pointcloud_to_image
- pcd_to_pointcloud
- pointcloud_to_pcd

PCL filters - nodelets

- Extract Indices
- PassThrough
- ProjectInliers
- RadiusOutlierRemoval
- StatisticalOutlierRemoval
- VoxelGrid

Python bindings: python-pcl: https://github.com/strawlab/python-pcl, not integrated into ROS yet, not a full implementation