Course of

Robot Programming with ROS 2

Day 2

2. Perception

ikerlan

Universidad Rey Juan Carlos

Intelligent Robotics Lab

Laser

sensor_msgs/LaserScan Message

File: sensor_msgs/LaserScan.msg

Raw Message Definition

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Intelligent Robotics Lab

Computer Vision

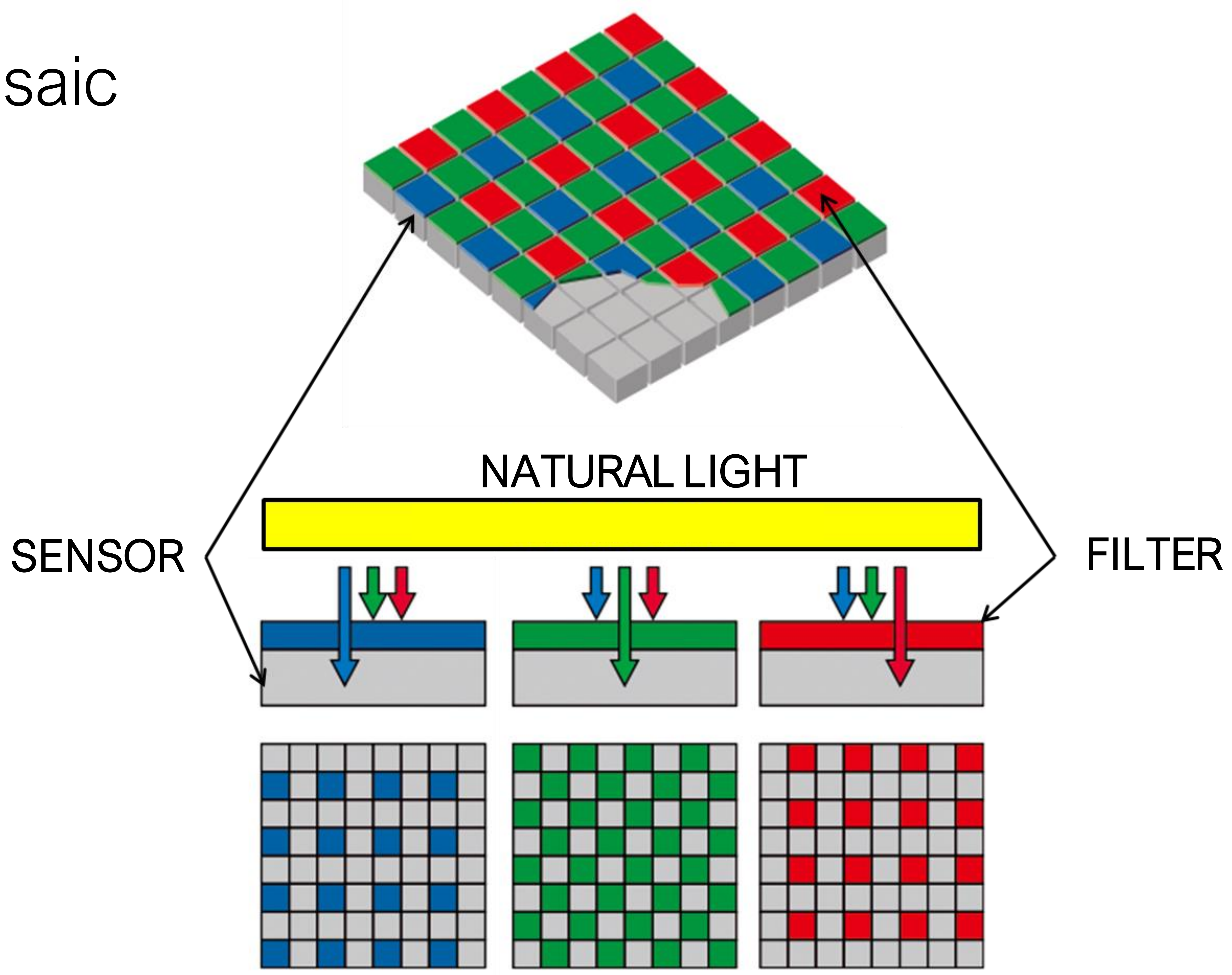
0	3	2	5	4	7	6	9	8
3	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
5	2	3	0	1	2	3	4	5
4	3	2	1	0	3	2	5	4
7	4	5	2	3	0	1	2	3
6	5	4	3	2	1	0	3	2
9	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

Human

Computer

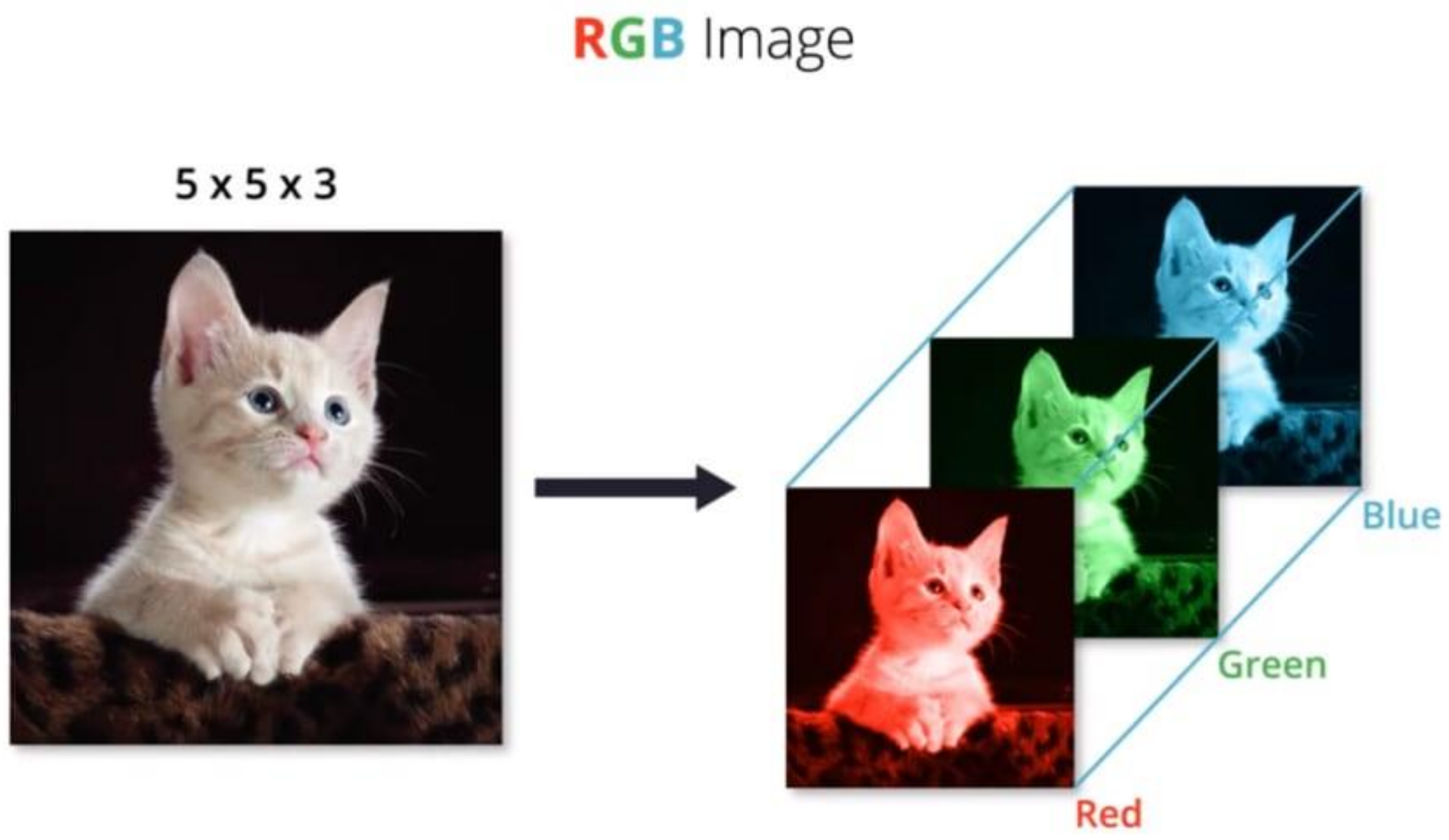
Computer Vision

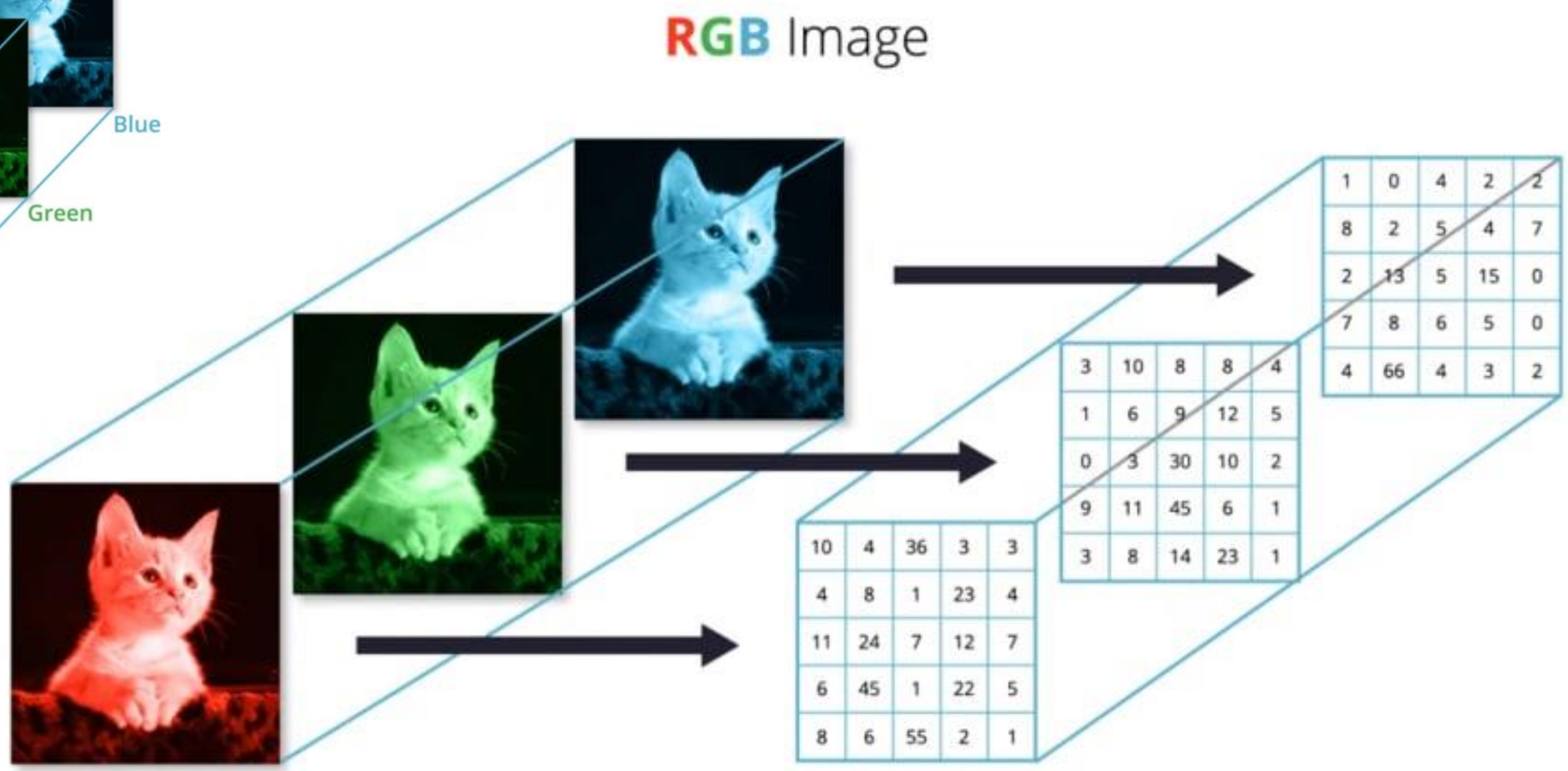
Bayer mosaic



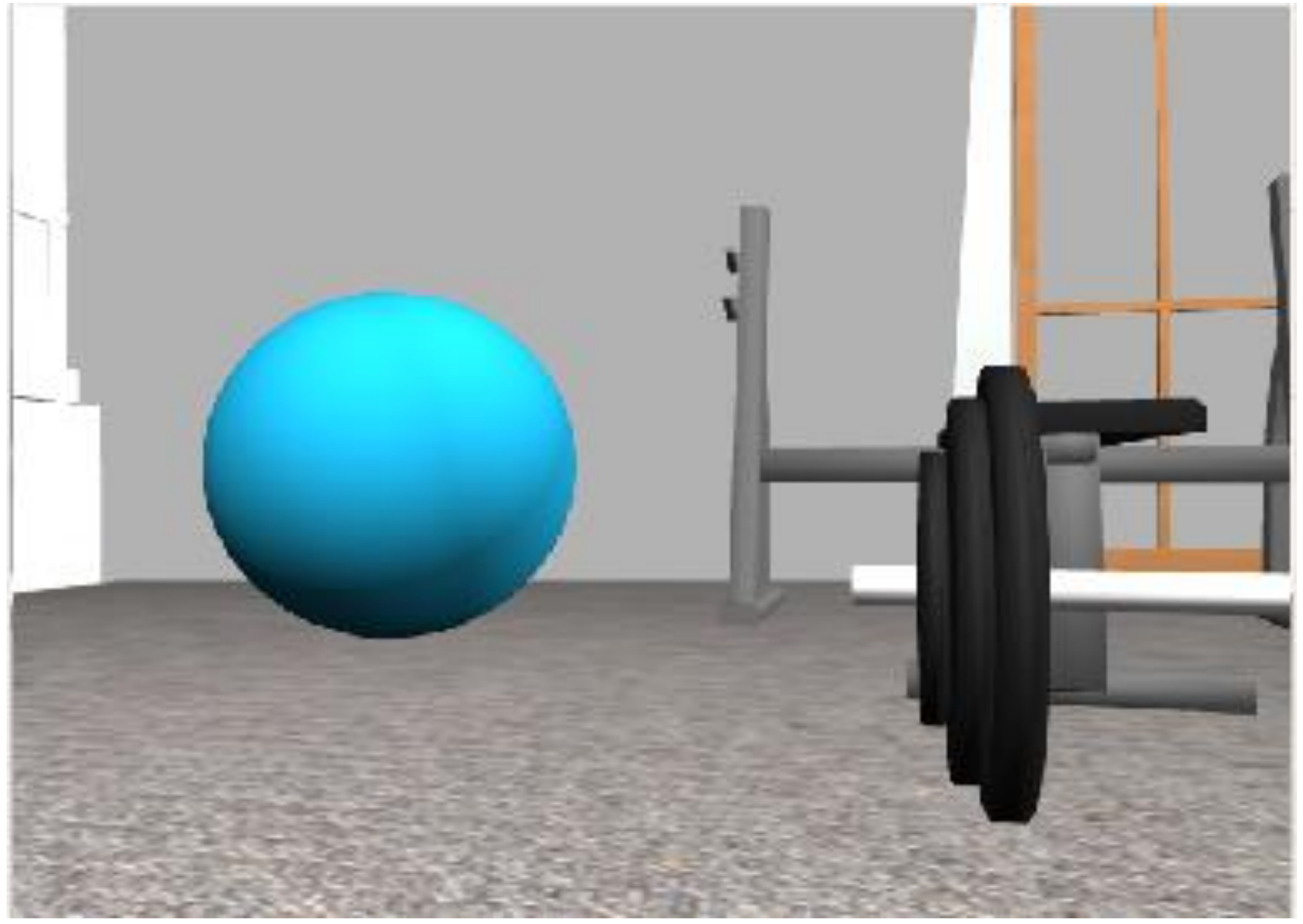
Computer Vision

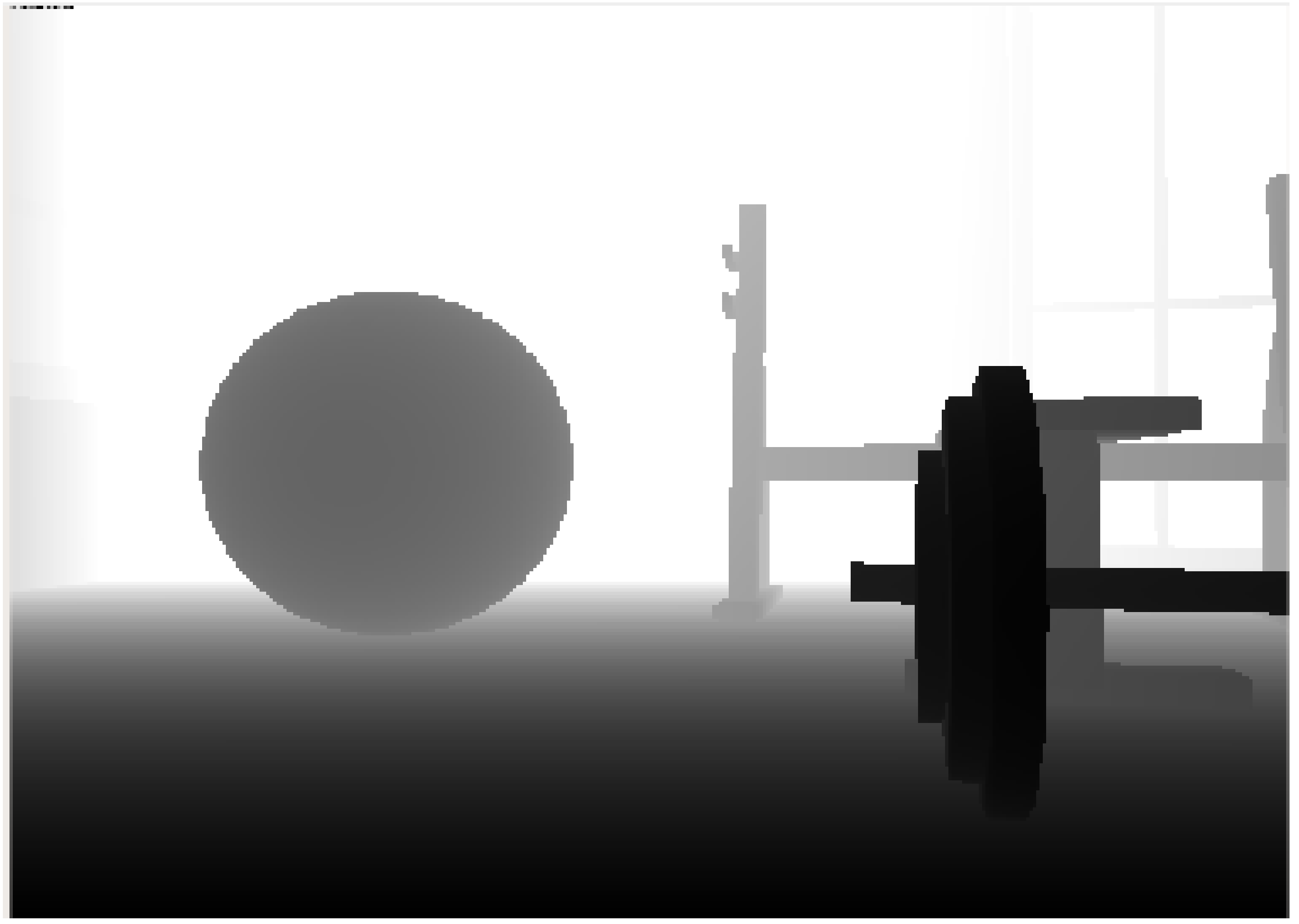
RGB image
8 bits



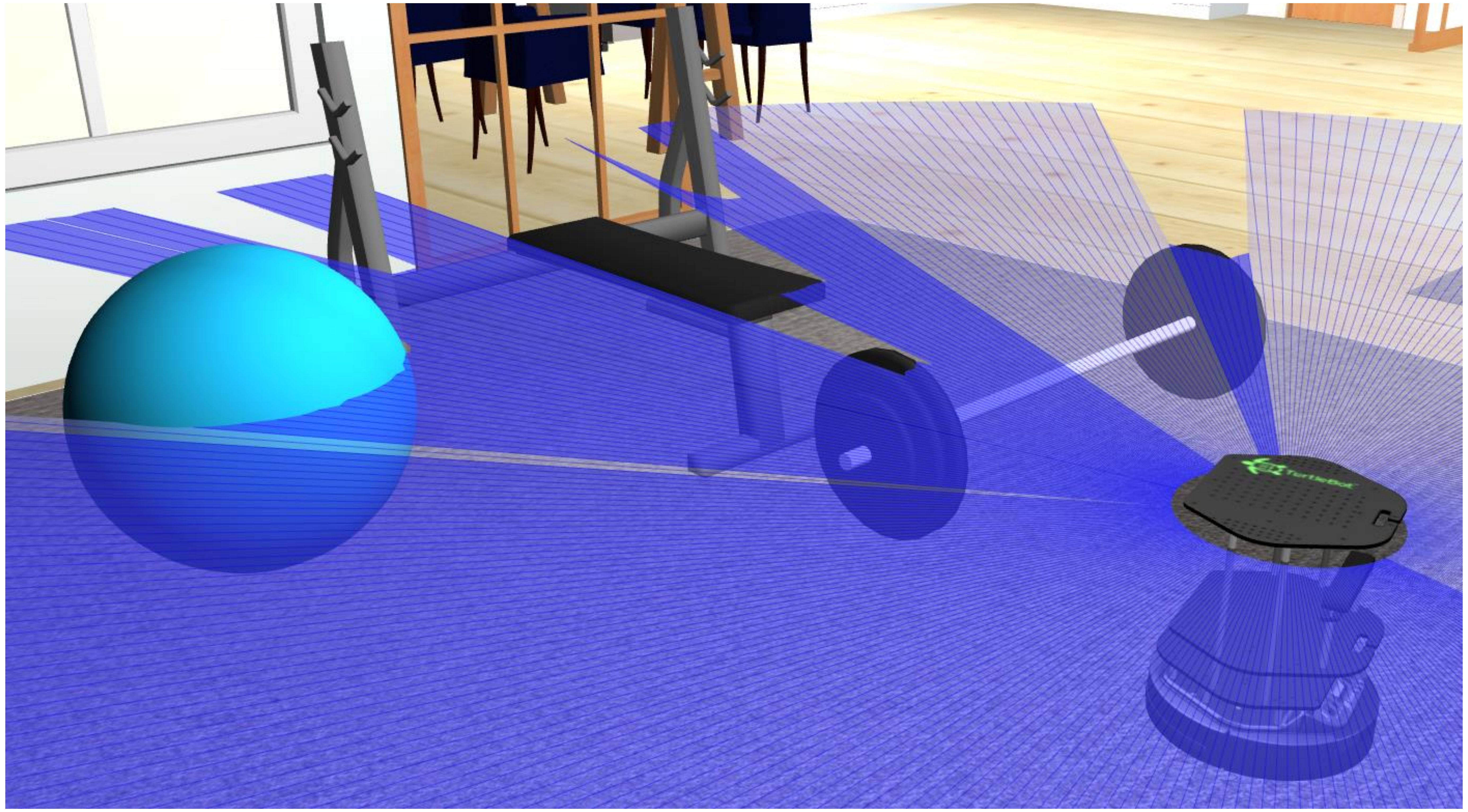


Computer Vision





Depth image
16/32 bits

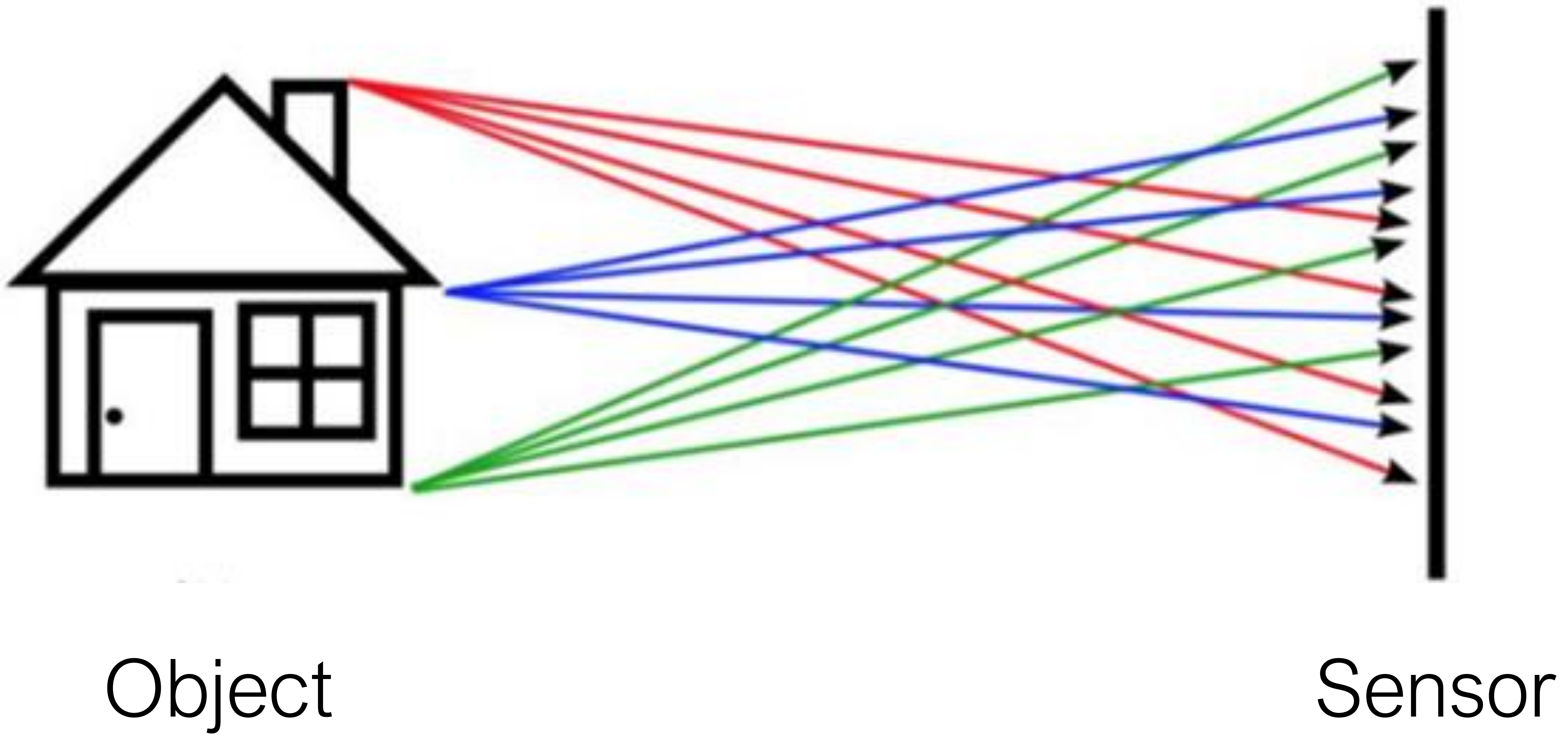


Computer Vision

PointCloud
[XYZ] [XYZRGB]

PinHole model

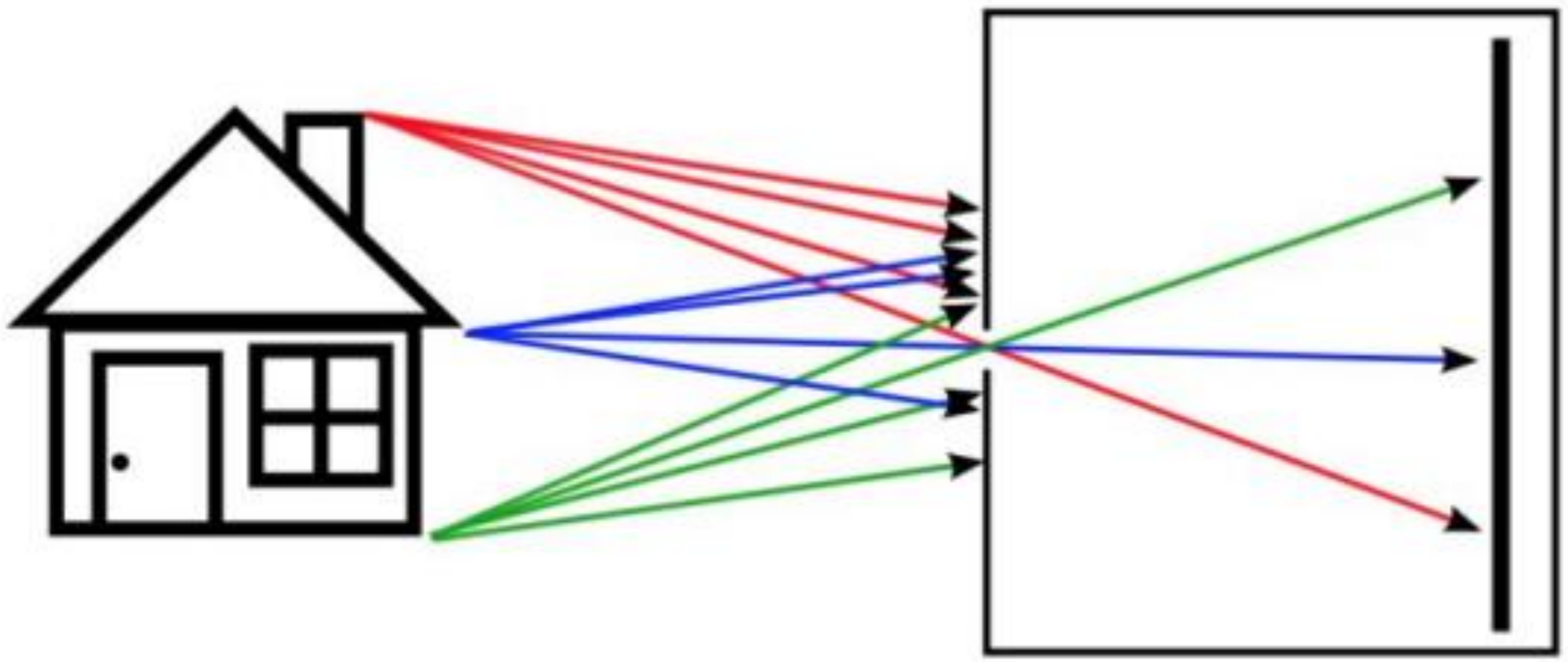
- Simple imaging device



- Unable to get a reasonable image

PinHole model

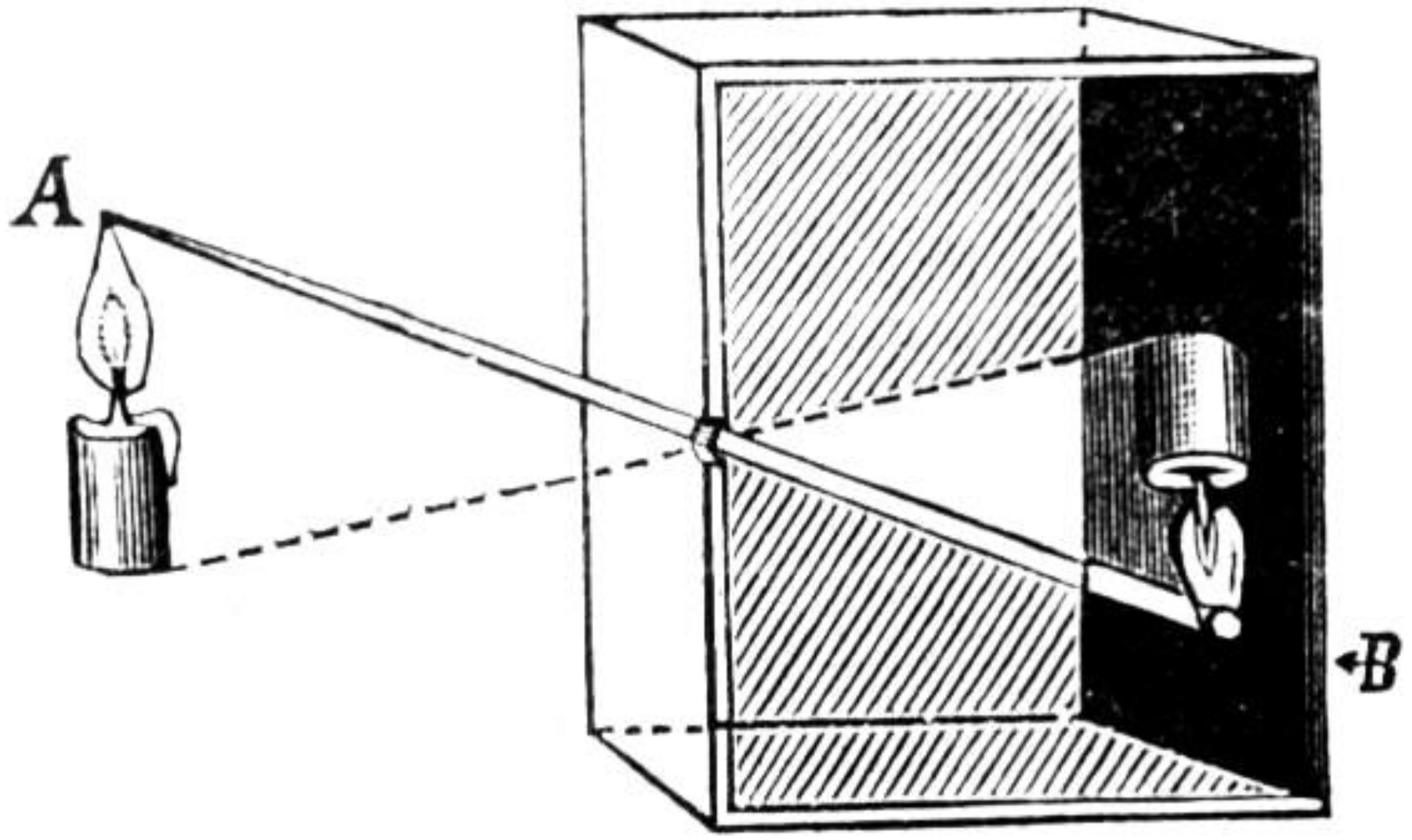
- Key idea: put a barrier with a small hole between the object and the sensor



- Less blur: the aperture should be small as possible
- Also know as camera obscura

PinHole model

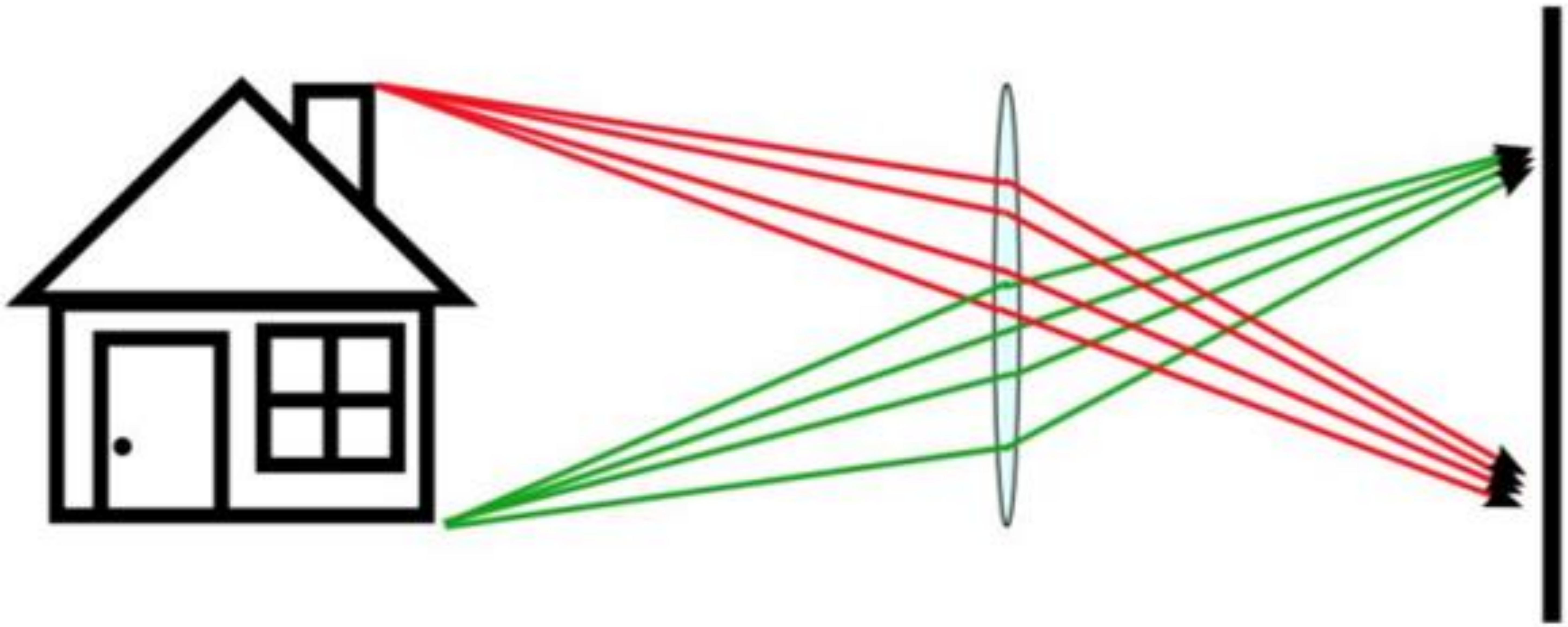
- When light from an image passes through this hole then an inverted image is formed on the opposite side



- Problems:
  Big hole: blur image
  Small hole: better focus but too dark

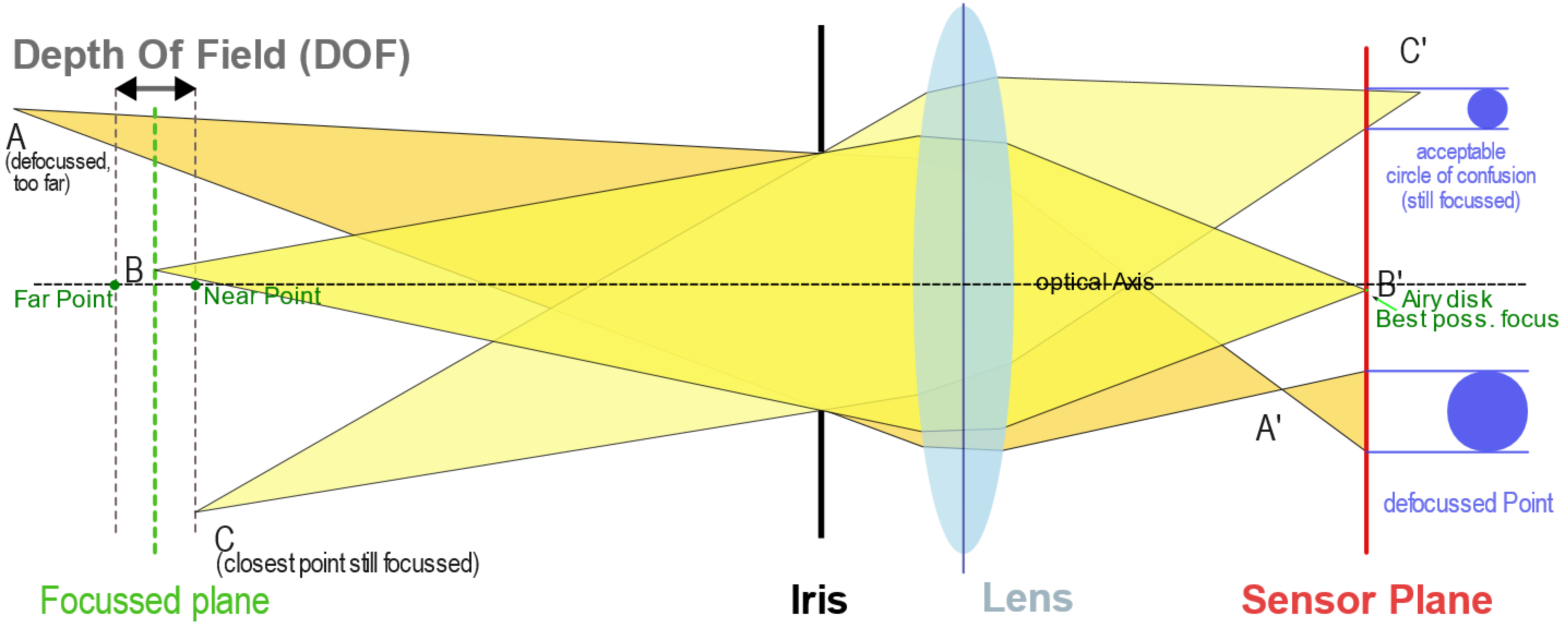
Thin lens model

- Solution: use lens to focus the object



Thin lens model

- Unlike the ideal pinhole camera, there is a specific distance at which the objects are in focus

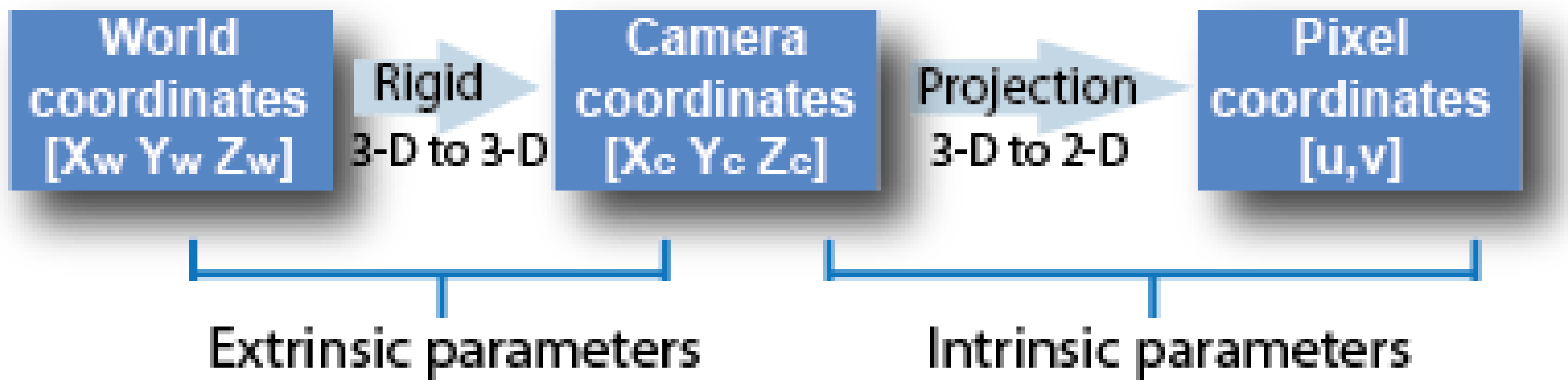


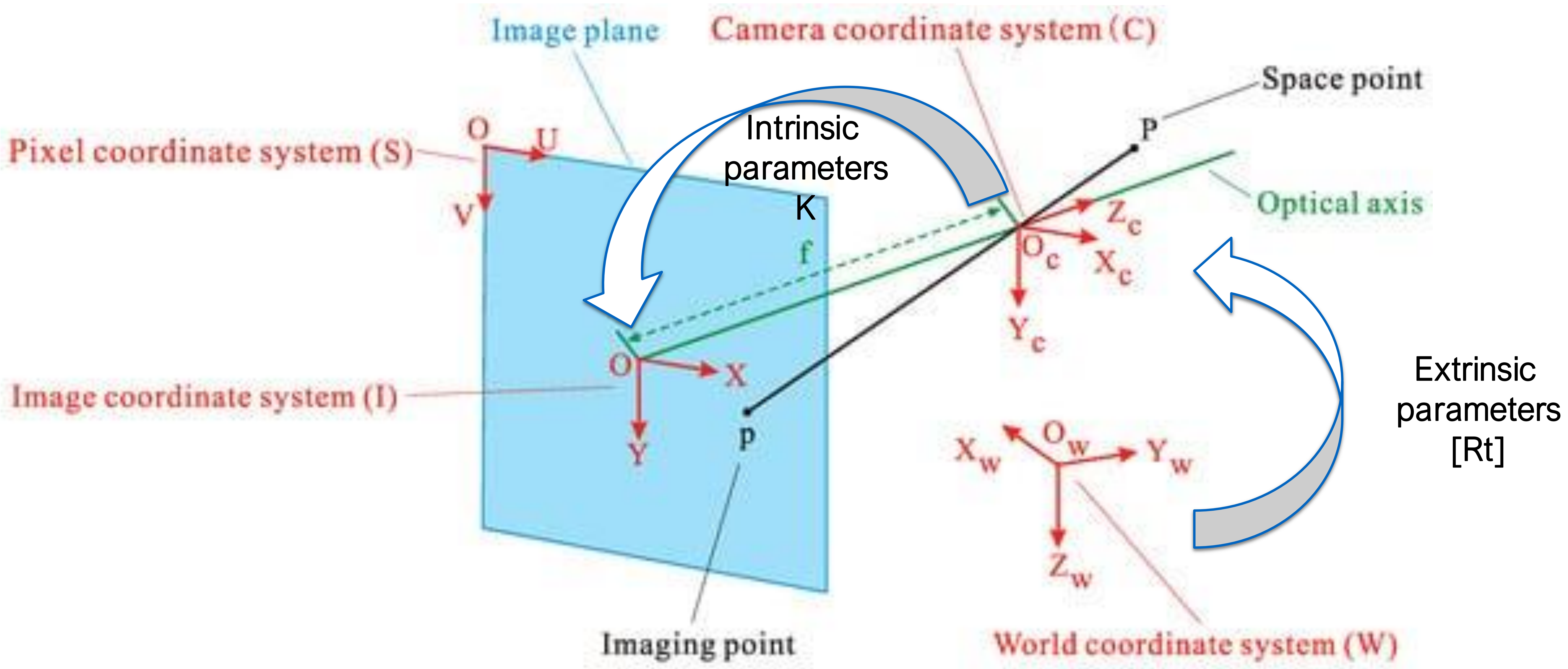
- Problem:

  Distorsion

Coordinate systems

- Commonly used coordinate systems in CV





Coordinate systems

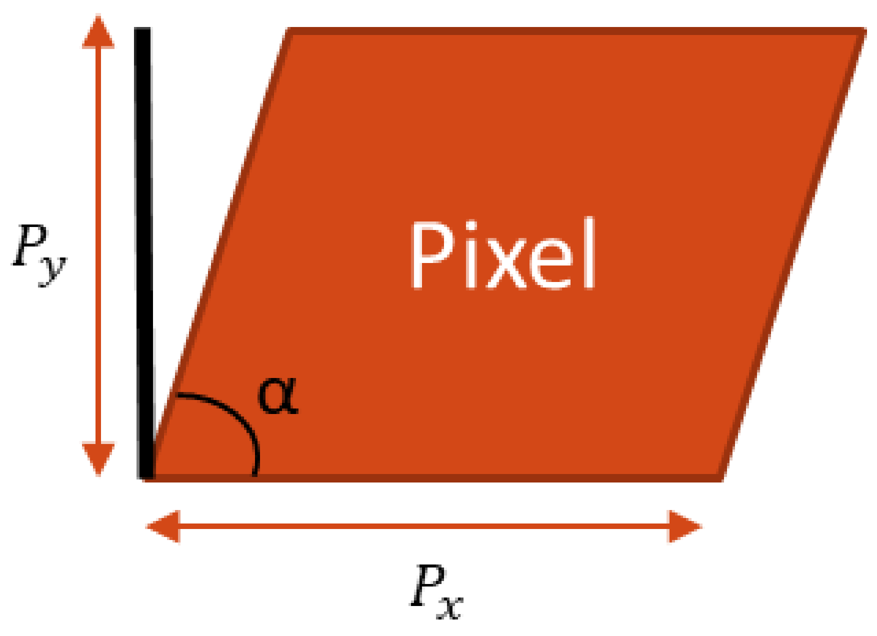
- Intrinsic parameters:

- In the Pinhole model, we find that the intrinsic parameters that allow us to project 3D points in the real world to 2D points in the image plane are as follows:
  - Focal length F
  - ($f_x = F \cdot k$, $f_y = F \cdot l$; k and l are the width and height of a pixel)
  - Center of the image c_x and c_y
- Formally, these values are represented by a matrix K

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Coordinate systems

- In the matrix K there is a value s that refers to the “bias” factor (skew). This value is related to the angle that the Y axis takes from the image plane coordinate system when the axes are not perpendicular
- In the standard models, the axes are always perpendicular, so we will (almost) always find this value set to 0



$$s = -f_x \cdot \cot(\alpha)$$

Coordinate systems

- The relationship between the $\boldsymbol{XYZ}$ 3D points of the world and the $\boldsymbol{xy}$ 2D points of the image is the following:

$$x = f_x \frac{X}{Z} - f_x \cot\alpha \frac{Y}{Z} + c_x \qquad y = \frac{f_y}{\sin\alpha} \frac{Y}{Z} + c_y$$

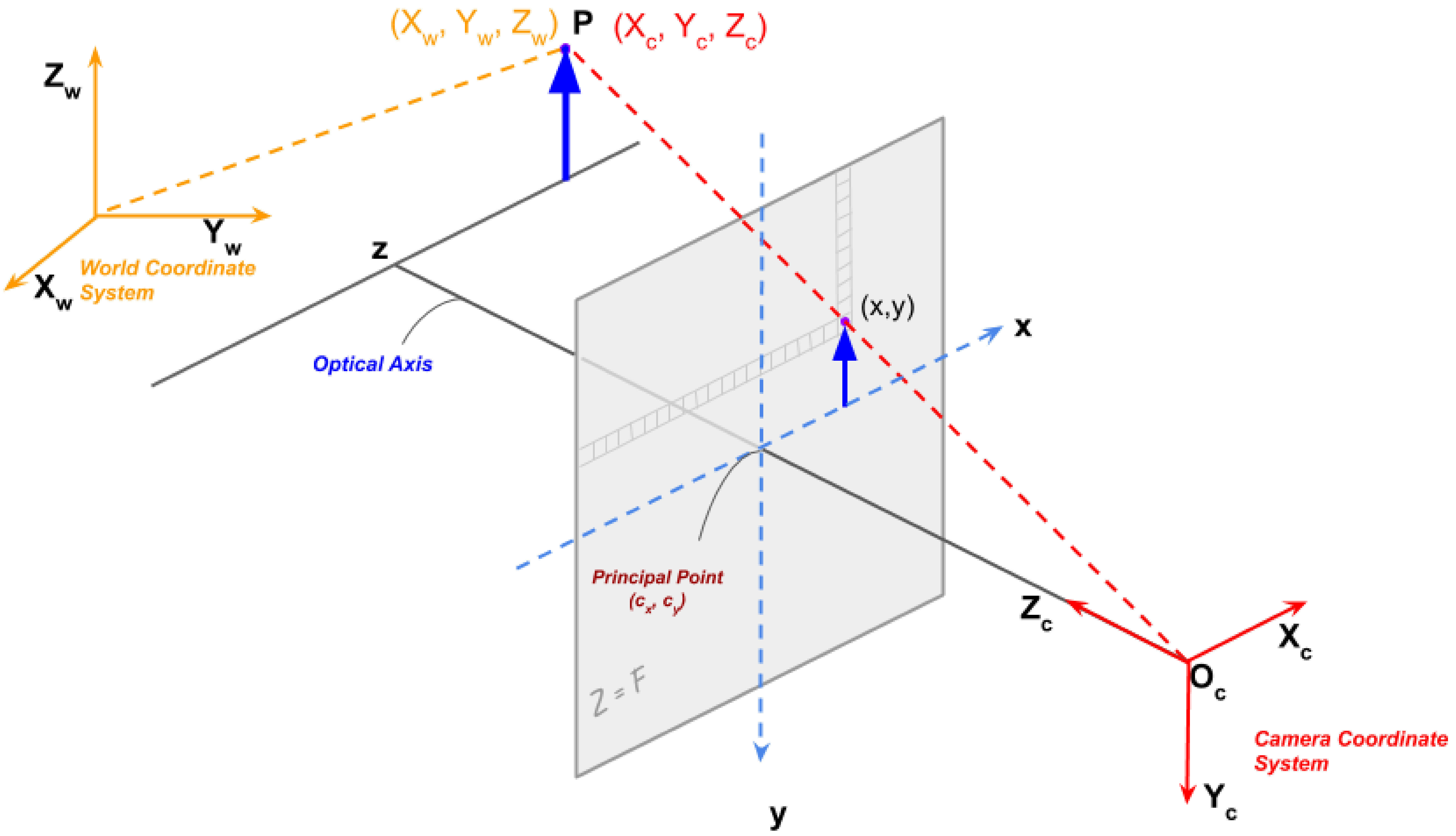
- For convenience, the parameters are summarized in the matrix $\boldsymbol{K}$ which is expressed in matrix form, so that it can be operated very easily

$$[x \quad y \quad 1]^T = K[X \quad Y \quad Z]^T$$

- In this way, we can calculate the position $\boldsymbol{xy}$ in the image for any point $\boldsymbol{XYZ}$ in the real world

Intelligent Robotics Lab

Coordinate systems



Coordinate systems

- These parameters also allow us to know the relationship between the $\boldsymbol{xy}$ 2D points of the image and the $\boldsymbol{XYZ}$ 3D points of the world

$$X = \frac{(x - c_x) * d}{f_x} \qquad Y = \frac{(y - c_y) * d}{f_y} \qquad Z = d$$

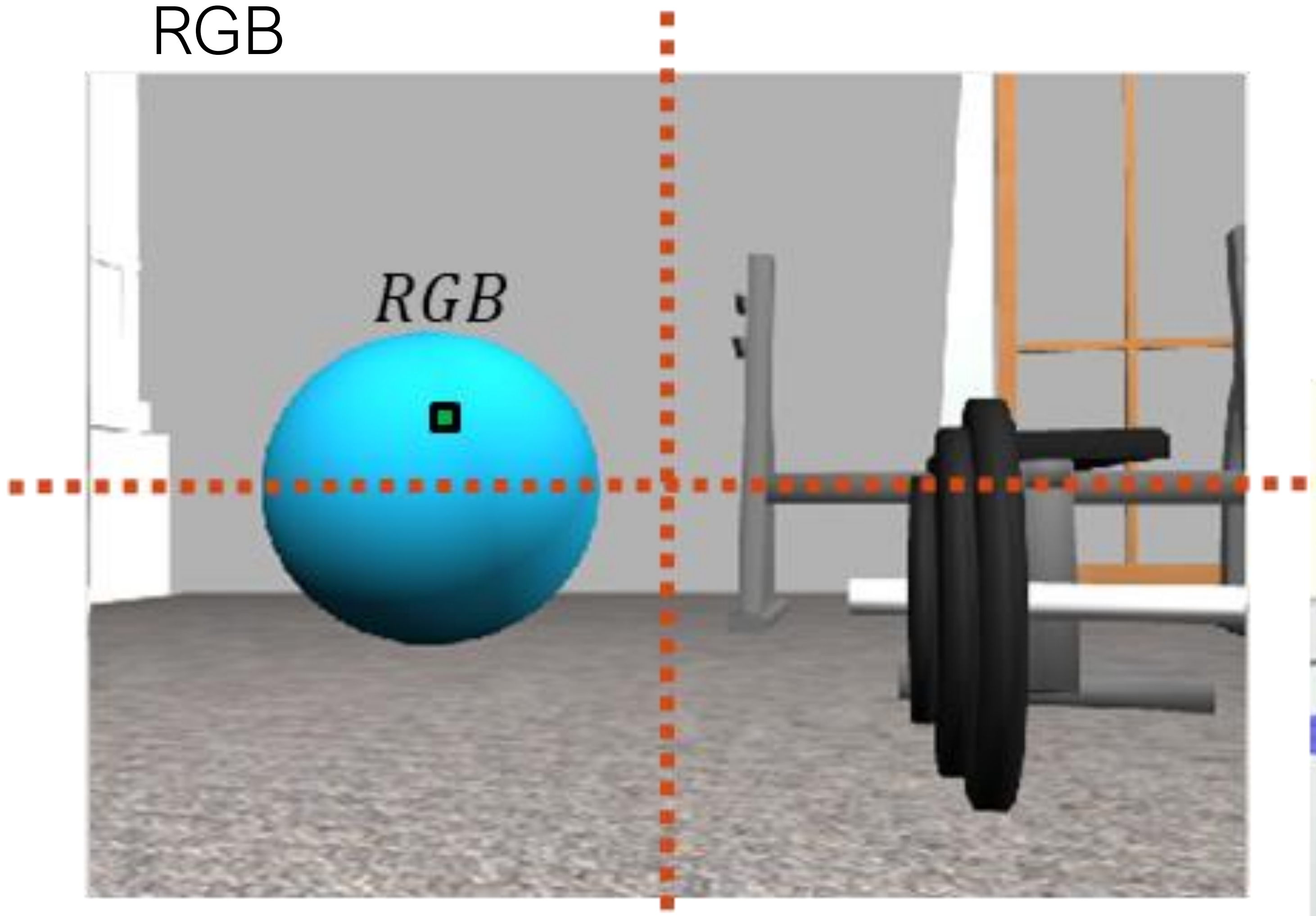
where d is the depth value that must be known a priori. Normally using the depth image, laser, or disparity image calculated by a stereo system

- In this way, we can calculate the position $\boldsymbol{XYZ}$ of the real world for any point $\boldsymbol{xy}$ of the image

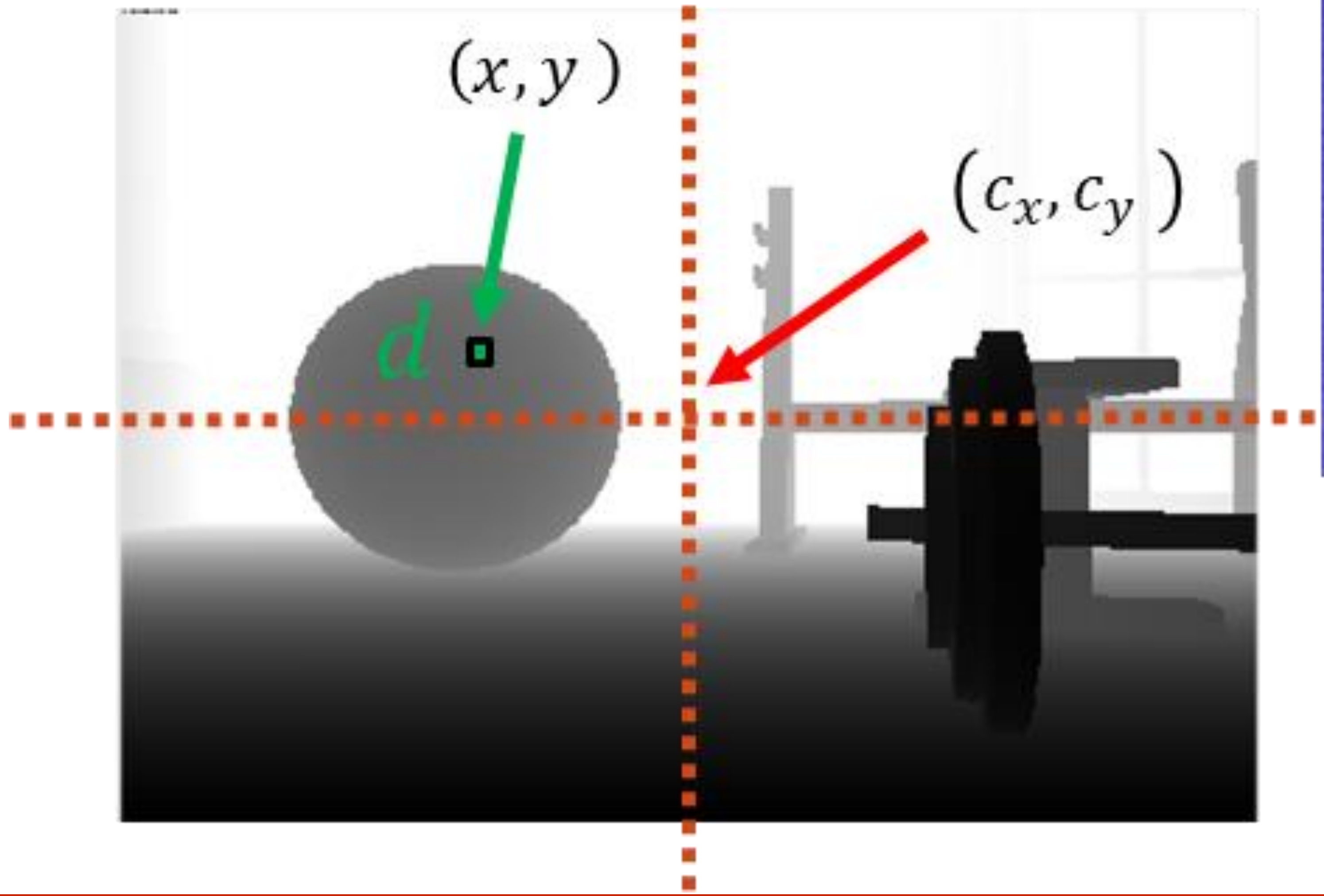
Intelligent Robotics Lab

Coordinate systems

RGB

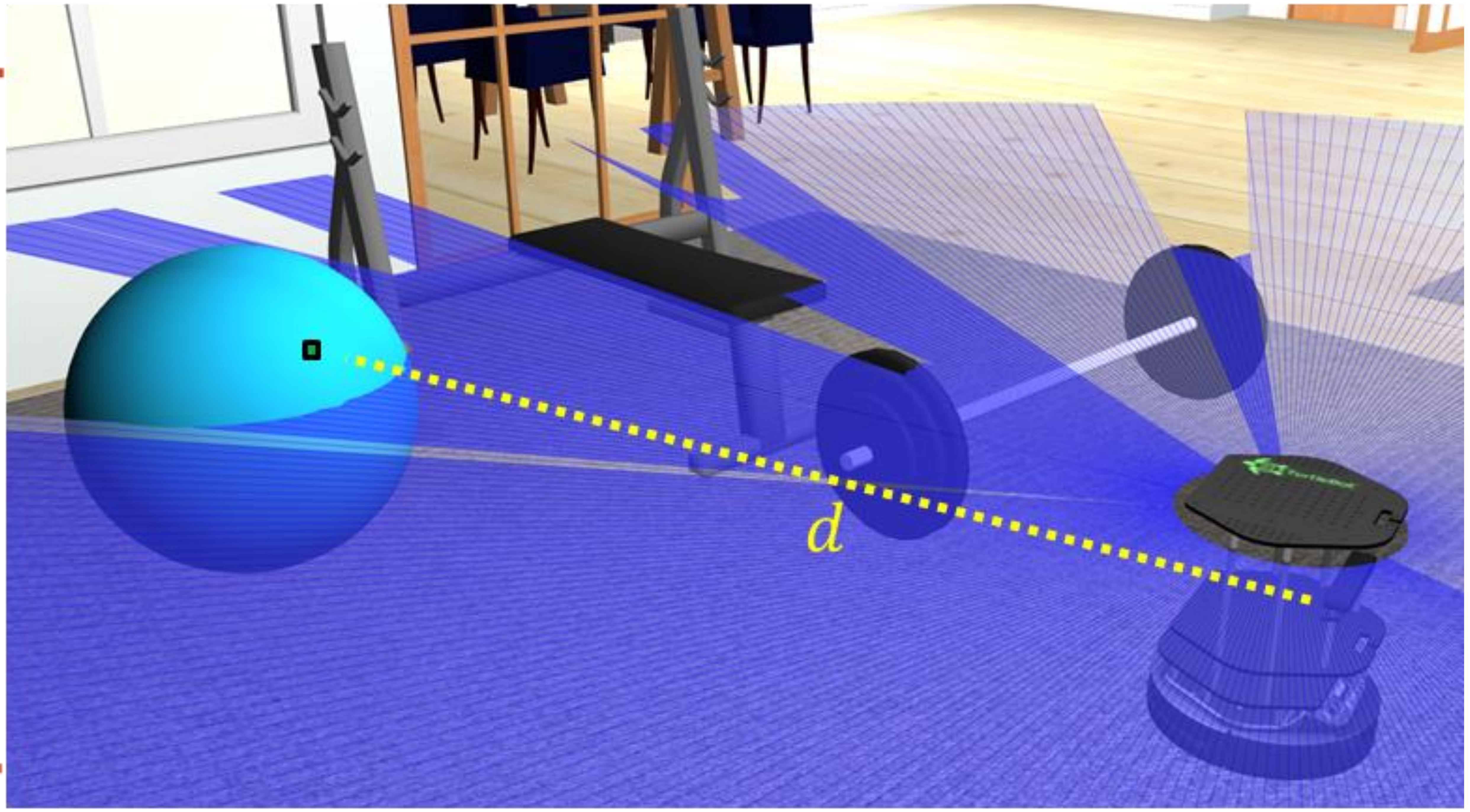


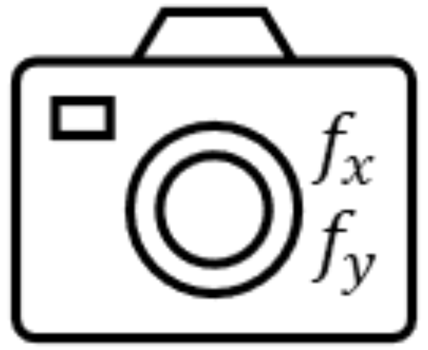
DEPTH



2D to 3D conversion

$$(X, Y, Z) = \left(\frac{(x - c_x) * d}{f_x}, \frac{(y - c_y) * d}{f_y}, d\right)$$



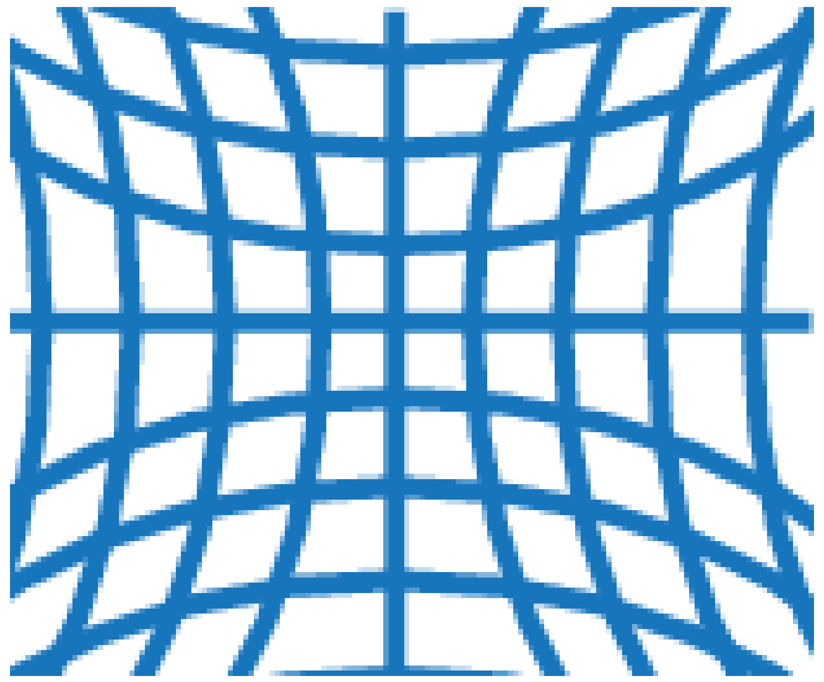


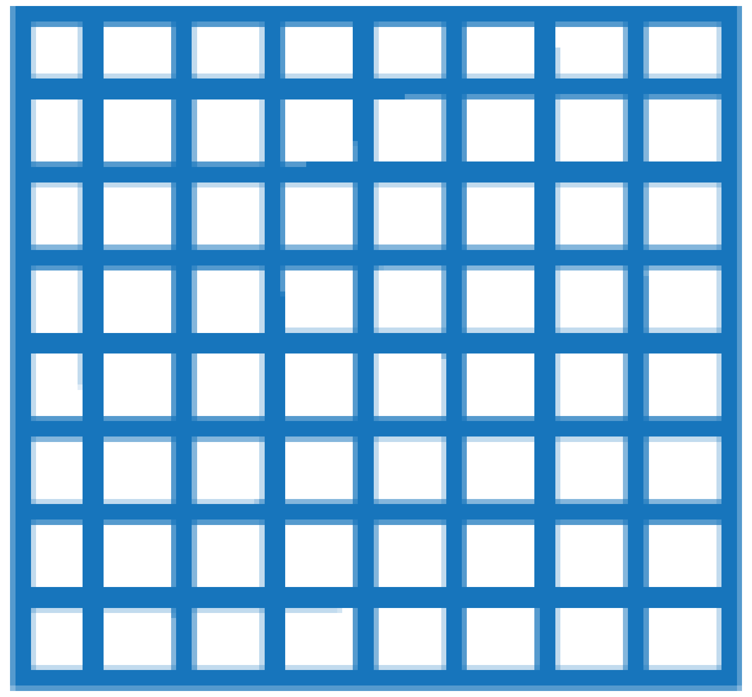
3D to 2D conversion

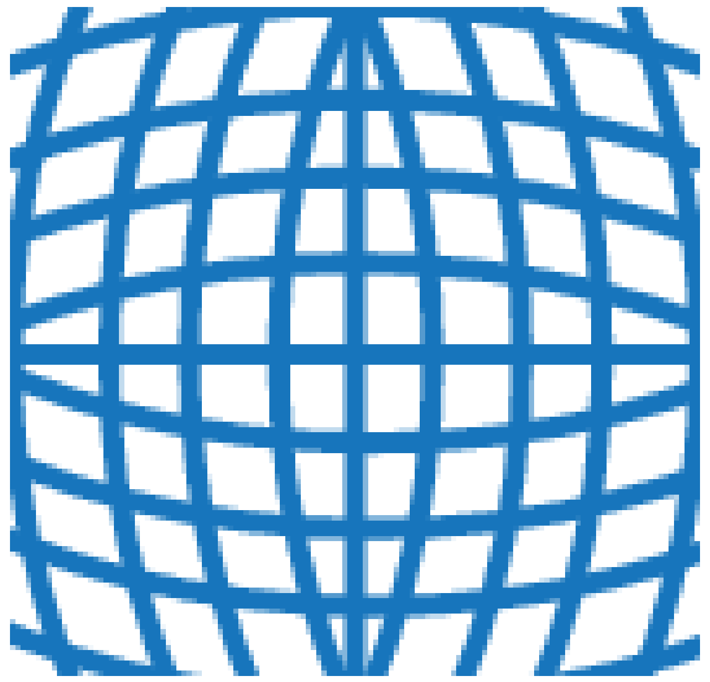
$$(x, y) = \left(f_x \frac{X}{Z}, f_y \frac{Y}{Z}\right)$$

Coordinate systems

- The lenses provide several advantages such as a defined depth of field (in contrast to the pure Pinhole model which shows infinite depth of field)
- It also introduces disadvantages such as radial distortion
- This type of distortion is not contemplated in the matrix K of intrinsic parameters of the camera, but we must also take it into account
- The matrix distortion can be obtained using computer vision techniques as calibration, or provider by the manufacturer


Negative radial distortion
"pincushion"


No distortion


Positive radial distortion
"barrel"

Coordinate systems

ROS2

- These parameters can be found in ROS, normally provided by the camera_info topic, which publishes the camera info provided by the manufacturer

```
jmguerrero@GS65:~$ ros2 topic info /head_front_camera/depth_registered/camera_info
Type: sensor_msgs/msg/CameraInfo
Publisher count: 1
Subscription count: 0
```

```
jmguerrero@GS65:~$ ros2 topic echo /head_front_camera/depth_registered/camera_info --once
header:
  stamp:
    sec: 197
    nanosec: 953000000
  frame_id: head_front_camera_rgb_optical_frame
height: 480
width: 640
distortion_model: plumb_bob
d:
- 1.0e-08
- 1.0e-08
- 1.0e-08
- 1.0e-08
- 1.0e-08
```

```
k:
- 522.1910329546544
- 0.0
- 320.5
- 0.0
- 522.1910329546544
- 240.5
- 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:
- 522.1910329546544
- 0.0
- 320.5
- -0.0
- 0.0
- 522.1910329546544
- 240.5
- 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
---
```

Intelligent Robotics Lab

Coordinate systems

ROS2

- These parameters are the ones defined in the CameraInfo message
- To get more information, we can see what the message contains

```
$ ros2 interface show sensor_msgs/msg/CameraInfo
```

```
jmguerrero@GS65:~$ ros2 interface show sensor_msgs/msg/CameraInfo
# This message defines meta information for a camera. It should be in a
# camera namespace on topic "camera_info" and accompanied by up to five
# image topics named:
#
#   image_raw - raw data from the camera driver, possibly Bayer encoded
#   image            - monochrome, distorted
#   image_color      - color, distorted
#   image_rect       - monochrome, rectified
#   image_rect_color - color, rectified
#
# The image_pipeline contains packages (image_proc, stereo_image_proc)
# for producing the four processed image topics from image_raw and
# camera_info. The meaning of the camera parameters are described in
# detail at http://www.ros.org/wiki/image_pipeline/CameraInfo.
#
# The image_geometry package provides a user-friendly interface to
# common operations using this meta information. If you want to, e.g.,
# project a 3d point into image coordinates, we strongly recommend
# using image_geometry.
#
# If the camera is uncalibrated, the matrices D, K, R, P should be left
# zeroed out. In particular, clients may assume that K[0] == 0.0
# indicates an uncalibrated camera.
```

Intelligent Robotics Lab

Coordinate systems

ROS2

```
##################################################################
#                     Image acquisition info                     #
##################################################################

# Time of image acquisition, camera coordinate frame ID
std_msgs/Header header # Header timestamp should be acquisition time of image
        builtin_interfaces/Time stamp
                int32 sec
                uint32 nanosec
        string frame_id
                             # Header frame_id should be optical frame of camera
                             # origin of frame should be optical center of camera
                             # +x should point to the right in the image
                             # +y should point down in the image
                             # +z should point into the plane of the image
```

```
#######################################################################
#                      Operational Parameters                         #
#######################################################################
# These define the image region actually captured by the camera       #
# driver. Although they affect the geometry of the output image, they #
# may be changed freely without recalibrating the camera.             #
#######################################################################

# Binning refers here to any camera setting which combines rectangular
#  neighborhoods of pixels into larger "super-pixels." It reduces the
#  resolution of the output image to
#  (width / binning_x) x (height / binning_y).
# The default values binning_x = binning_y = 0 is considered the same
#  as binning_x = binning_y = 1 (no subsampling).
uint32 binning_x
uint32 binning_y

# Region of interest (subwindow of full camera resolution), given in
#  full resolution (unbinned) image coordinates. A particular ROI
#  always denotes the same window of pixels on the camera sensor,
#  regardless of binning settings.
# The default setting of roi (all values 0) is considered the same as
#  full resolution (roi.width = width, roi.height = height).
RegionOfInterest roi
        #
        uint32 x_offset  #
                         # (0 if the ROI includes the left edge of the image)
        uint32 y_offset  #
                         # (0 if the ROI includes the top edge of the image)
        uint32 height    #
        uint32 width     #
        bool do_rectify
```

```
#######################################################################
#                     Calibration Parameters                          #
#######################################################################
# These are fixed during camera calibration. Their values will be the #
# same in all messages until the camera is recalibrated. Note that    #
# self-calibrating systems may "recalibrate" frequently.              #
#                                                                     #
# The internal parameters can be used to warp a raw (distorted) image #
# to:                                                                 #
#   1. An undistorted image (requires D and K)                        #
#   2. A rectified image (requires D, K, R)                           #
# The projection matrix P projects 3D points into the rectified image.#
#######################################################################

# The image dimensions with which the camera was calibrated.
# Normally this will be the full camera resolution in pixels.
uint32 height
uint32 width

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

Intelligent Robotics Lab

Coordinate systems

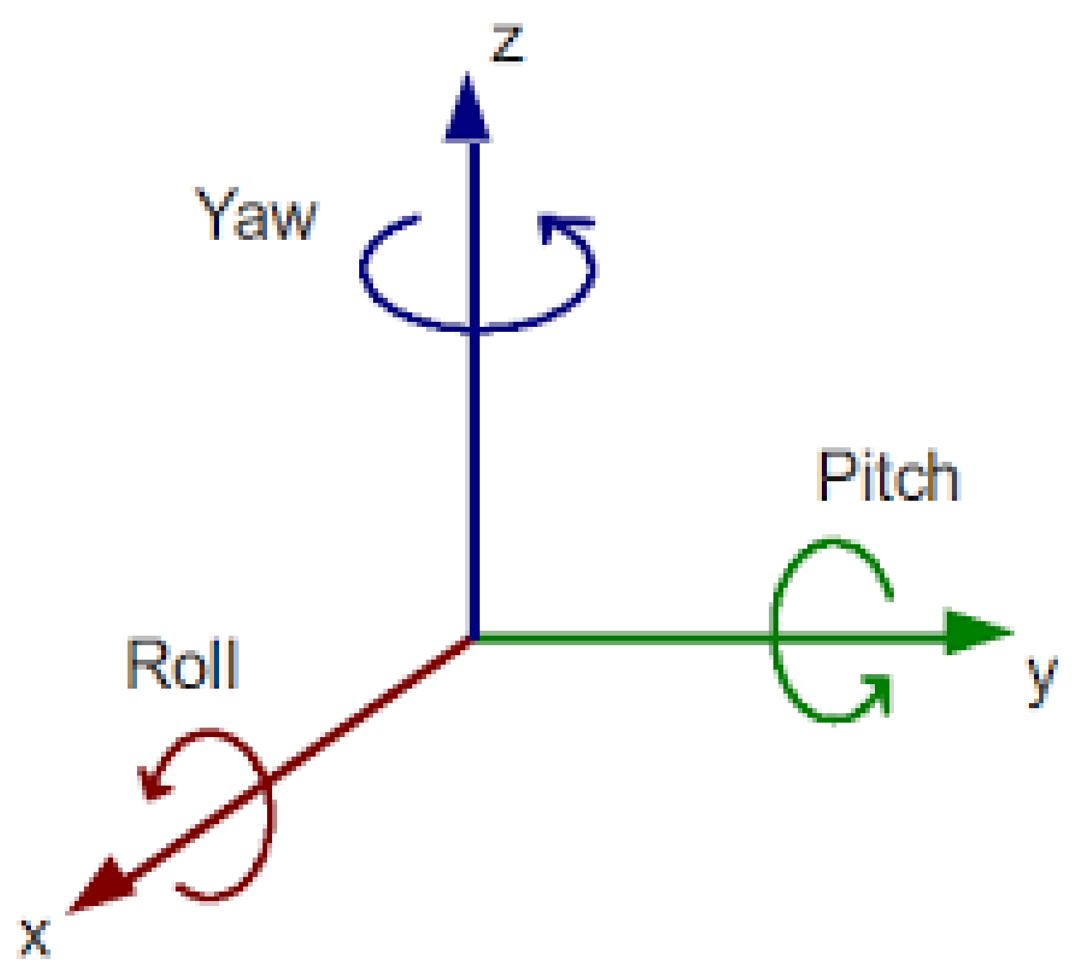
- Extrinsic parameters:

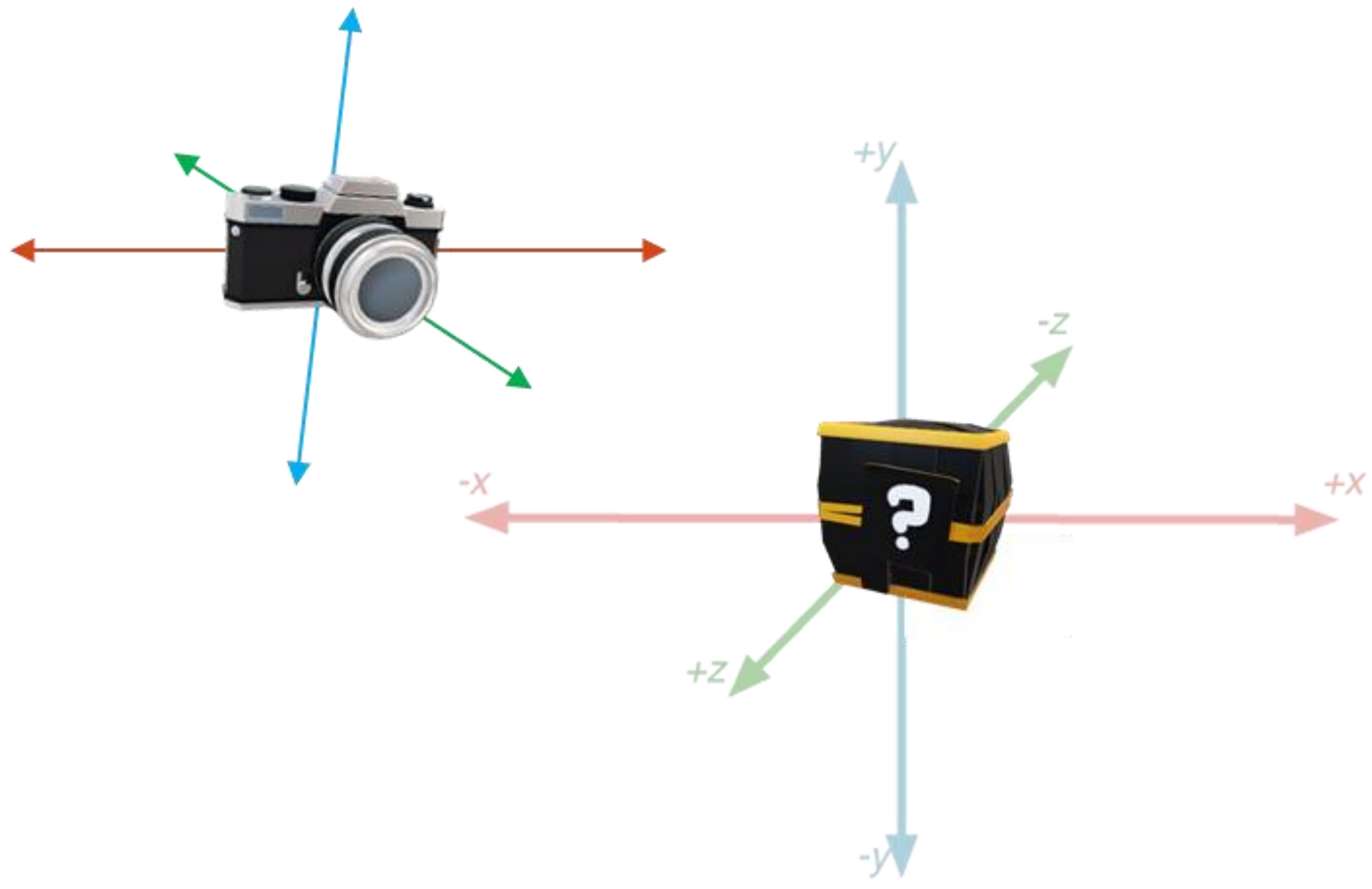
- Define the position of the camera in the real-world coordinate system
- Together with intrinsic parameters, they allow us to find out the 3D position in the world coordinate system of a certain 2D point in the image coordinate system
- These parameters are not fixed and depend on the pose of the camera with respect to the world. Because of this, they must be estimated with some method
- These parameters align the coordinate system of the three-dimensional scene points with respect to the camera coordinate system

Intelligent Robotics Lab

Coordinate systems

- The extrinsic parameters are defined by two matrices R and t:
  R: rotation of the 3 axes between the coordinate systems
  t: displacement in x,y,z between the origins of the coordinate systems





Coordinate systems

- The rotation matrix R is composed of the accumulation of the independent rotation of each axis to square both coordinate systems

$$\underset{roll}{R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}} \qquad \underset{pitch}{R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}} \qquad \underset{yaw}{R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}}$$

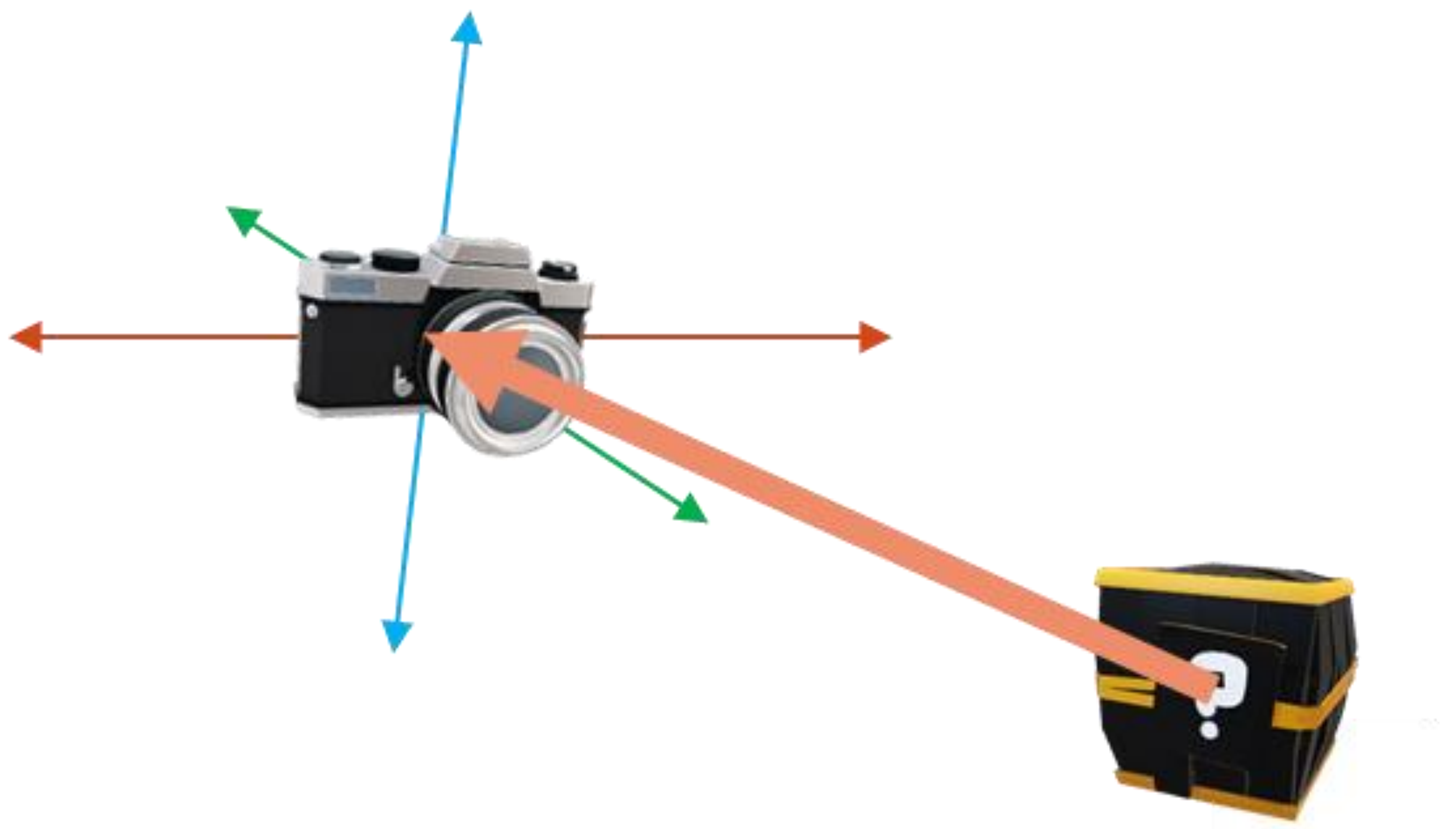
- Since $R = R_z(\Psi) \cdot R_y(\theta) \cdot R_x(\phi)$ it is common to find it directly as follows:

$$[R_0] = \begin{bmatrix} \cos(\psi)\cos(\theta) & \cos(\psi)\sin(\theta)\sin(\phi) - \sin(\psi)\cos(\phi) & \cos(\psi)\sin(\theta)\cos(\phi) + \sin(\psi)\sin(\phi) \\ \sin(\psi)\cos(\theta) & \sin(\psi)\sin(\theta)\sin(\phi) + \cos(\psi)\cos(\phi) & \sin(\psi)\sin(\theta)\cos(\phi) - \cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix}$$

Intelligent Robotics Lab

Coordinate systems

- The translation matrix t is nothing more than a vector that indicates how much the origin of one coordinate system would have to be moved to make it coincide with the other



$$t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Coordinate systems

- Joining all the matrices, the operation that must be done to finally project a 3D point in the real world to a 2D point in the image plane arises

$$x = K(RX + t)$$

- Sometimes the matrices R and t are expressed together in a matrix T of transformation

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Intelligent Robotics Lab

Coordinate systems

- Combining all the matrices:

$$w\begin{bmatrix}u\\v\\1\end{bmatrix} = \underbrace{\begin{bmatrix}f_x & [s] & c_x\\0 & f_y & c_y\\0 & 0 & 1\end{bmatrix}}_{\text{Intrinsic parameters}}\underbrace{\begin{bmatrix}r_{11} & r_{12} & r_{13} & t_x\\r_{21} & r_{22} & r_{23} & t_y\\r_{31} & r_{32} & r_{22} & t_z\end{bmatrix}}_{\text{Extrinsic parameters}}\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}$$

- (X, Y, Z) → the coordinates of a 3D point in the real world
- (u, v) → the coordinates of the projected point in pixels
- f_x and f_y → focal lengths in pixels
- c_x and c_y → center of the image
- r and t → rotation and translation matrices between the camera and the projected area
- s → is the angle a pixel has with respect to the y-axis (usually at 0)
- w → is the scale factor used to homogenize u and v

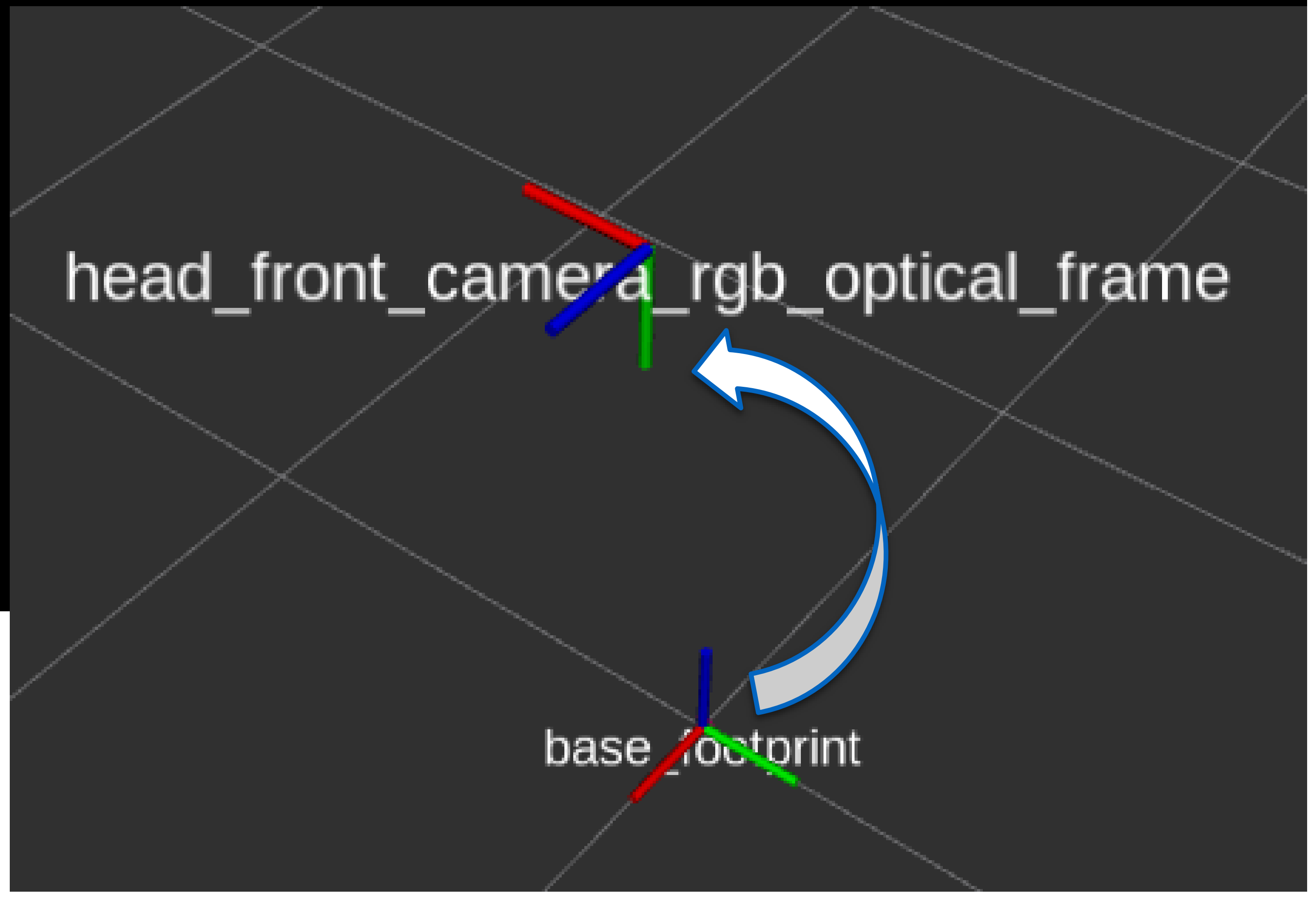
Coordinate systems

ROS2

- We can use the tf2 package to get the extrinsic parameters (rotation and translation matrices)

```
$ ros2 run tf2_ros tf2_echo <source_frame> <target_frame>
```

```
jmguerrero@GS65:~$ ros2 run tf2_ros tf2_echo base_footprint head_front_camera_rgb_optical_frame
[INFO] [1686299194.653624339] [tf2_echo]: Waiting for transform base_footprint ->  head_front_camera_rgb_op
tical_frame: Invalid frame ID "base_footprint" passed to canTransform argument target_frame - frame does no
t exist
At time 192.84000000
- Translation: [0.216, 0.022, 1.216]
- Rotation: in Quaternion [-0.500, 0.500, -0.500, 0.500]
- Rotation: in RPY (radian) [-1.571, -0.000, -1.571]
- Rotation: in RPY (degree) [-90.000, -0.000, -90.003]
- Matrix:
 -0.000 -0.000  1.000  0.216
 -1.000  0.000 -0.000  0.022
  0.000 -1.000 -0.000  1.216
  0.000  0.000  0.000  1.000
```



Intelligent Robotics Lab

OpenCV

- OpenCV (Open-Source Computer Vision) is a popular open-source computer vision and machine learning library
- It offers a wide range of functions and algorithms for image and video processing tasks
- The library includes basic image processing operations, as well as advanced functionalities like feature detection, optical flow estimation, and camera calibration
- OpenCV provides machine learning capabilities for tasks such as image classification, object detection, and face recognition. Integrated with frameworks like TensorFlow and PyTorch

OpenCV

- OpenCV supports multiple programming languages, including C++, Python, and Java
- OpenCV is portable and runs on various platforms, including Windows, Linux, macOS, Android, and iOS
- The library has an active community that contributes to its development and provides ongoing support
- OpenCV is widely used in fields such as robotics, augmented reality, surveillance, and medical imaging

OpenCV + ROS 2

- Install OpenCV: You can use package managers like apt or homebrew to install it

- Import OpenCV in your ROS 2 package:
    - Add OpenCV as a dependency in the package.xml file
    - Include OpenCV in the CMakeLists.txt file of your ROS 2 package
    - This allows the build system to link against the OpenCV libraries

OpenCV + ROS 2

- Verify that OpenCV is installed:

```
$ pkg-config --modversion opencv4
```

- You should get the version number:

```
4.7.0
```

- Otherwise, install OpenCV

```
$ sudo apt install libopencv-dev python3-opencv
```

OpenCV + ROS 2

- Create a package with dependencies:

```
$ ros2 pkg create opencv_demo --dependencies rclcpp std_msgs sensor_msgs cv_bridge image_transport OpenCV
```

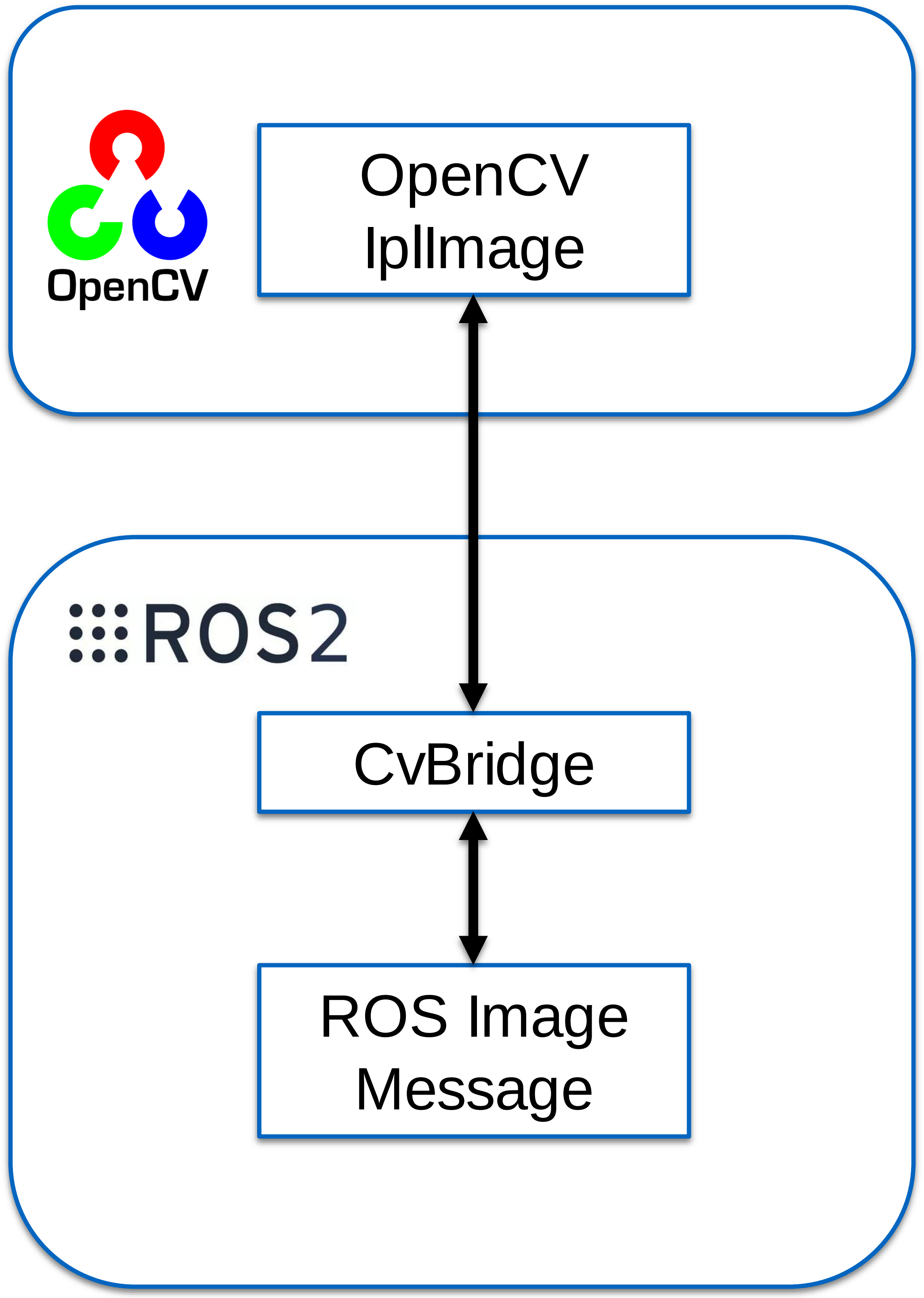
```xml
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>opencv_demo</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="josemiguel.guerrero@urjc.es">jmguerrero</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>std_msgs</depend>
  <depend>sensor_msgs</depend>
  <depend>cv_bridge</depend>
  <depend>image_transport</depend>
  <depend>OpenCV</depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

```
opencv_demo
  include / opencv_demo
  src
  CMakeLists.txt
  package.xml
```

```cmake
cmake_minimum_required(VERSION 3.8)
project(opencv_demo)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(cv_bridge REQUIRED)
find_package(image_transport REQUIRED)
find_package(OpenCV REQUIRED)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

OpenCV + ROS 2

- CvBridge converts between ROS Image messages and OpenCV images



OpenCV + ROS 2

- Publisher/Subscriber ROS Image

```
class OpenCVPubSub : public rclcpp::Node
{
public:
  OpenCVPubSub()
  : Node("opencv_pub_sub")
  {
    // Read camera info
    subscription_info_ = create_subscription<sensor_msgs::msg::CameraInfo>(
      "/camera_info", 1,
      std::bind(&OpenCVPubSub::topic_callback_info, this, std::placeholders::_1));

    // Image subscription
    subscription_image_ = create_subscription<sensor_msgs::msg::Image>(
      "/image_in", 1, std::bind(&OpenCVPubSub::topic_callback_image, this, std::placeholders::_1));

    // Image publisher
    publisher_image_ = this->create_publisher<sensor_msgs::msg::Image>(
      "image_out", rclcpp::SensorDataQoS().reliable());
  }

private:
  void topic_callback_info(sensor_msgs::msg::CameraInfo::UniquePtr msg);
  void topic_callback_image(const sensor_msgs::msg::Image::ConstSharedPtr & image_in_msg) const;

  rclcpp::Subscription<sensor_msgs::msg::CameraInfo>::SharedPtr subscription_info_;
  std::shared_ptr<image_geometry::PinholeCameraModel> camera_model_;
  rclcpp::Subscription<sensor_msgs::msg::Image>::SharedPtr subscription_image_;
  rclcpp::Publisher<sensor_msgs::msg::Image>::SharedPtr publisher_image_;

};
```

OpenCV + ROS 2

- Publisher/Subscriber ROS Image

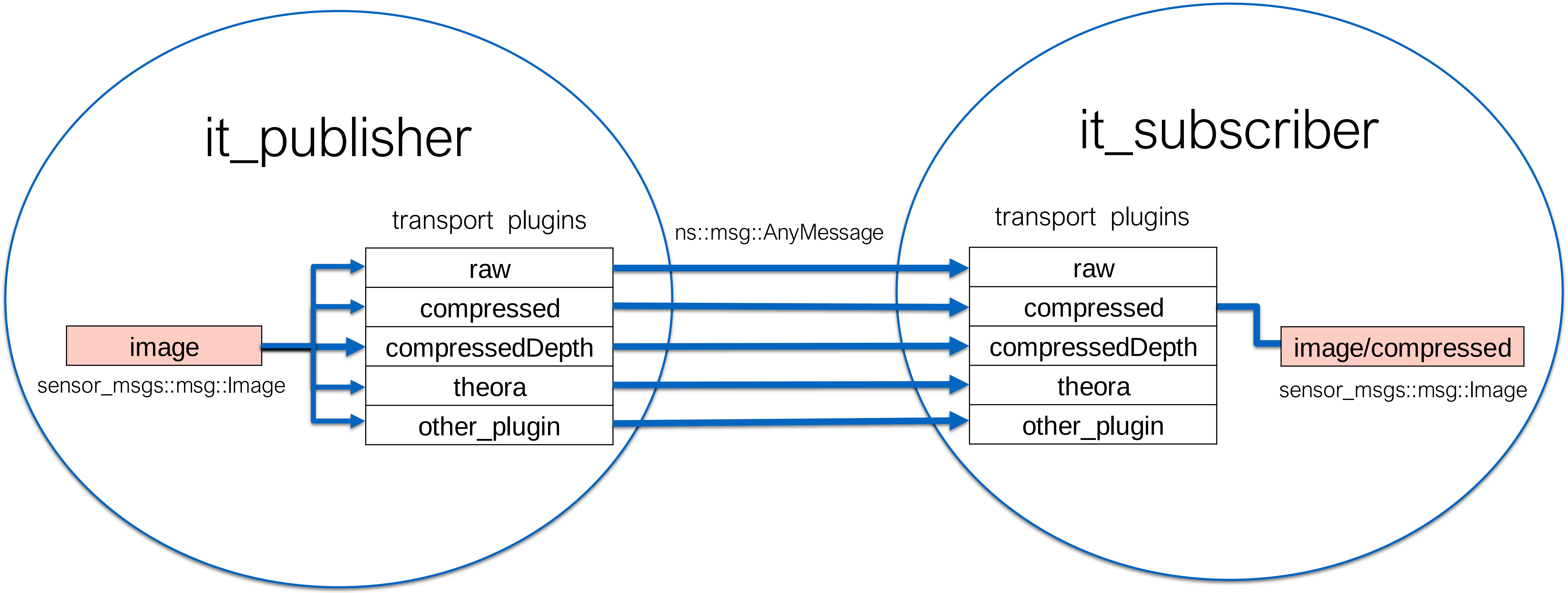
```cpp
void OpenCVPubSub::topic_callback_info(sensor_msgs::msg::CameraInfo::UniquePtr msg)
{
  RCLCPP_INFO(get_logger(), "Camera info received");
  camera_model_ = std::make_shared<image_geometry::PinholeCameraModel>();
  camera_model_->fromCameraInfo(*msg);
  subscription_info_ = nullptr;
}
```

```cpp
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);

  // create a ros2 node
  auto node = std::make_shared<perception_demo::OpenCVPubSub>();

  // process ros2 callbacks until receiving a SIGINT (ctrl-c)
  rclcpp::spin(node);
  rclcpp::shutdown();

  return 0;
}
```

```cpp
void OpenCVPubSub::topic_callback_image(
  const sensor_msgs::msg::Image::ConstSharedPtr & image_in_msg) const
{
  // Check if camera model has been received
  if (camera_model_ == nullptr) {
    RCLCPP_WARN(get_logger(), "Camera Model not yet available");
    return;
  }
  // Check if there is any subscription to the topic
  if (publisher_image_->get_subscription_count() > 0) {
    // Convert ROS Image to OpenCV Image
    cv_bridge::CvImagePtr image_in_ptr;
    try {
      image_in_ptr = cv_bridge::toCvCopy(*image_in_msg, sensor_msgs::image_encodings::BGR8);
    } catch (cv_bridge::Exception & e) {

      RCLCPP_ERROR(get_logger(), "cv_bridge exception: %s", e.what());
      return;
    }

    // Get OpenCV Image
    cv::Mat image_in = image_in_ptr->image;
    // OpenCV processing ...
    cv::Mat image_out;
    cv::cvtColor(image_in, image_out, cv::COLOR_BGR2HSV);

    // Convert OpenCV Image to ROS Image
    cv_bridge::CvImage image_out_bridge =
      cv_bridge::CvImage(image_in_msg->header, sensor_msgs::image_encodings::BGR8, image_out);
    // from cv_bridge to sensor_msgs::Image
    sensor_msgs::msg::Image image_out_msg;
    image_out_bridge.toImageMsg(image_out_msg);
    //Publish image
    publisher_image_->publish(image_out_msg);
  }
}
```

ROS 2 image_transport

- Provides transparent support for transporting images in low-bandwidth compressed formats
- Abstracts this complexity so that the developer only sees sensor_msgs/Image messages



ROS 2 image_transport

- Publisher

```cpp
int main(int argc, char ** argv)
{
  // Initialize ROS and create a node
  rclcpp::init(argc, argv);
  rclcpp::NodeOptions options;
  rclcpp::Node::SharedPtr node_ = rclcpp::Node::make_shared("image_publisher", options);

  // Create an ImageTransport instance, initializing it with our Node
  image_transport::ImageTransport it(node_);
  // Create a publisher using ImageTransport to publish on the topic
  image_transport::Publisher pub = it.advertise("image_transport", 1);

  // Publish the image
  rclcpp::WallRate loop_rate(5);
  while (rclcpp::ok()) {
    if (pub.getNumSubscribers() > 0) {
      // OpenCV Mat image
      cv::Mat image = cv::imread("path/to/file.jpg", cv::IMREAD_COLOR);
      // Convert the image to a ROS message
      std_msgs::msg::Header hdr;
      sensor_msgs::msg::Image::SharedPtr msg = cv_bridge::CvImage(
        hdr,
        sensor_msgs::image_encodings::BGR8,
        image).toImageMsg();
      pub.publish(msg);
    }
    rclcpp::spin_some(node_);
    loop_rate.sleep();
  }
}
```

ROS 2 image_transport

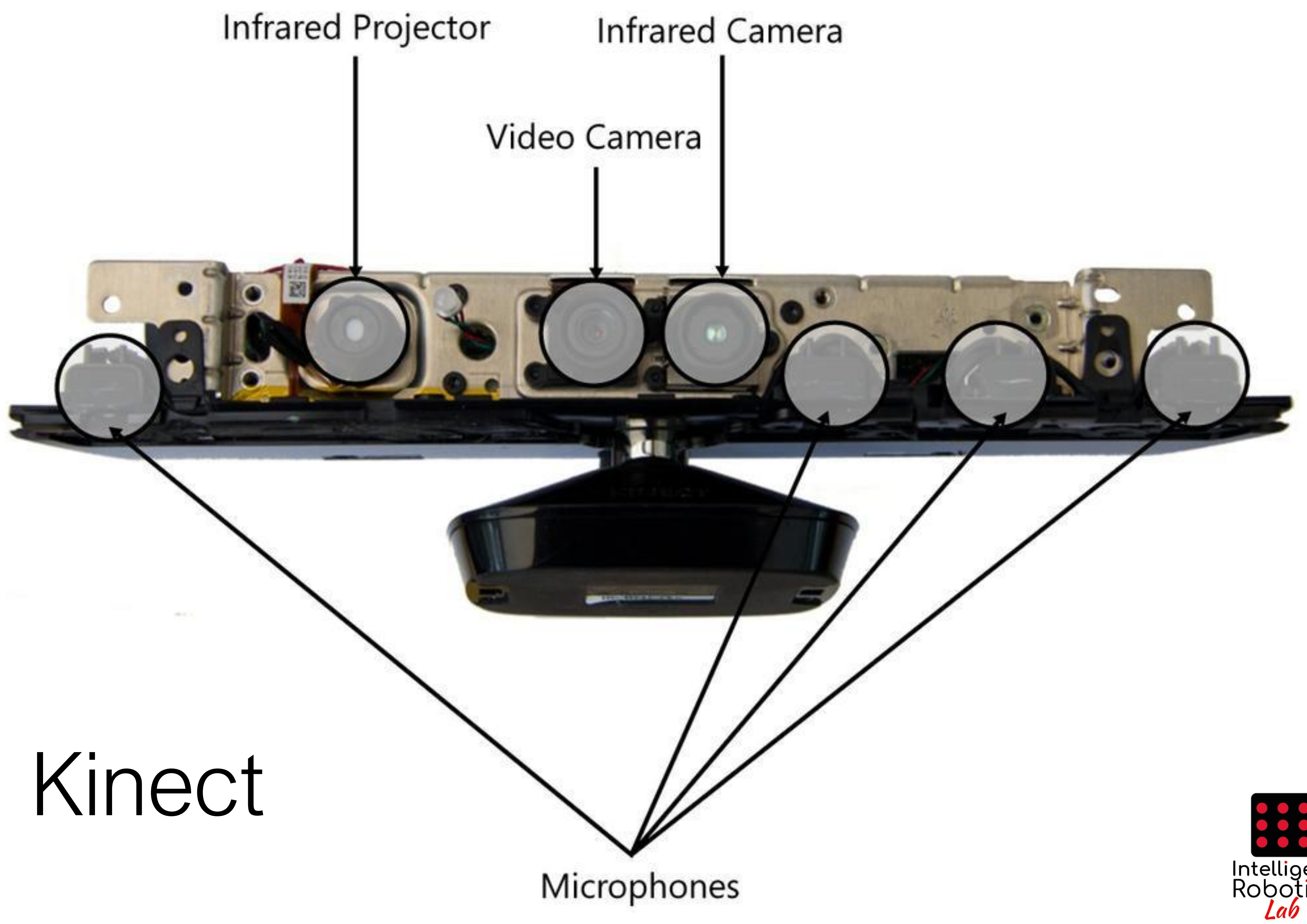
- Subscriber

```cpp
void imageCallback(const sensor_msgs::msg::Image::ConstSharedPtr & msg)
{
  try {
    // Show the image using OpenCV
    cv::imshow(transport_, cv_bridge::toCvShare(msg, msg->encoding.c_str())->image);
    cv::waitKey(10);
  } catch (const cv_bridge::Exception & e) {
    RCLCPP_ERROR(logger_, "Could not convert from '%s' to 'bgr8'.", msg->encoding.c_str());
  }
}
```

```cpp
int main(int argc, char ** argv)
{
  // Initialize ROS and create a node
  rclcpp::init(argc, argv);
  rclcpp::NodeOptions options;
  rclcpp::Node::SharedPtr node_ = rclcpp::Node::make_shared("image_subscriber", options);

  // Create a window to show the image
  cv::namedWindow(transport_);
  cv::startWindowThread();

  // Create an ImageTransport instance, initializing it with the Node
  image_transport::Subscriber subscriber_;
  // Assign the subscriber to a specific transport
  const image_transport::TransportHints hints(node_.get(), "compressed");
  try {
    auto subscription_options = rclcpp::SubscriptionOptions();
    // Create a subscription with QoS profile that will be used for the subscription.
    subscriber_ = image_transport::create_subscription(
      node_.get(),
      "image_transport",
      std::bind(&imageCallback, std::placeholders::_1),
      hints.getTransport(),
      rmw_qos_profile_sensor_data,
      subscription_options);
  } catch (image_transport::TransportLoadException & e) {
    RCLCPP_ERROR(
      logger_, "Failed to create subscriber for topic %s: %s", topic_image_.c_str(), e.what());
    return -1;
  }

  // Spin until rclcpp::ok() returns false
  rclcpp::spin(node_);
  cv::destroyWindow(transport_);
  return 0;
}
```

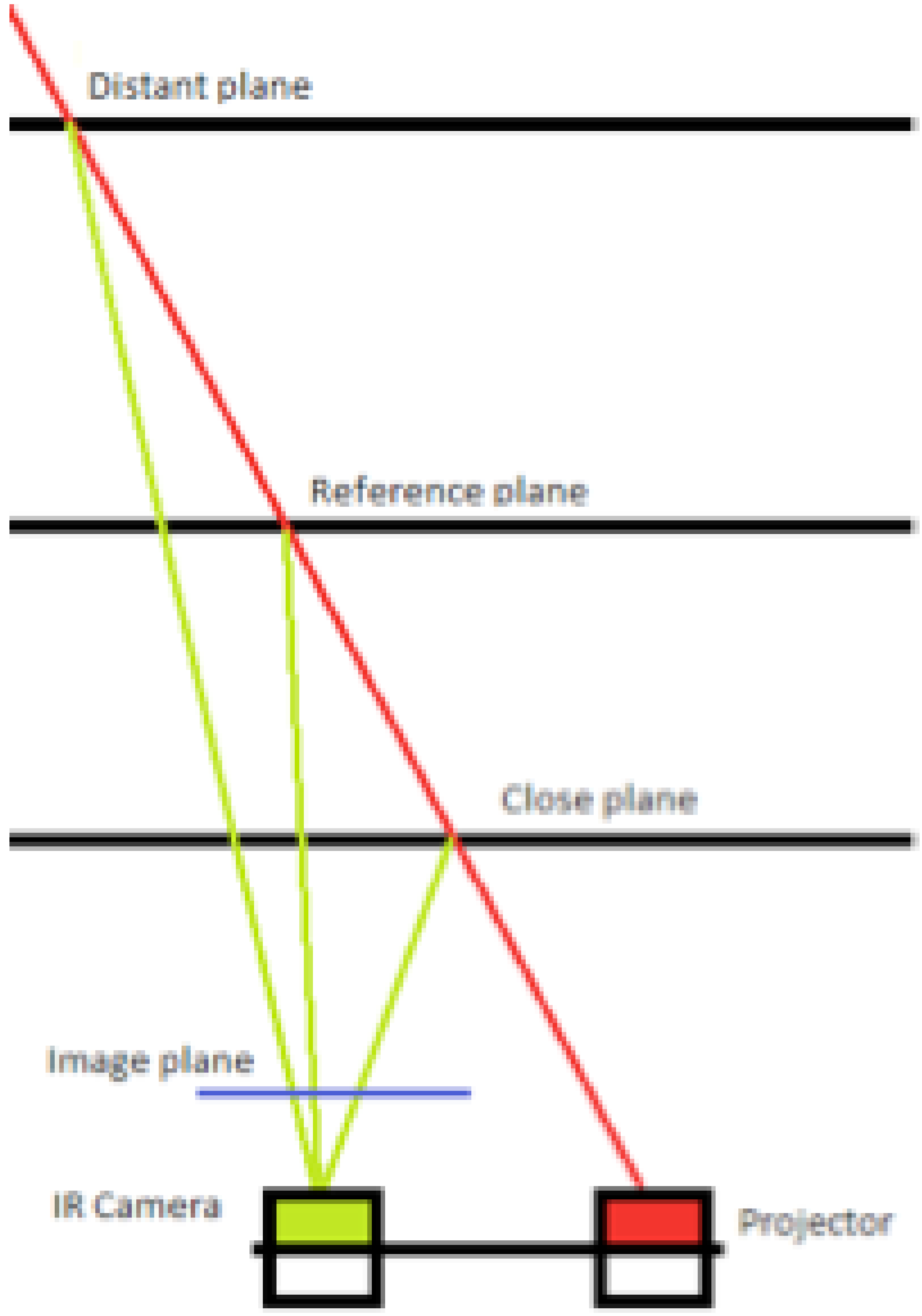
RGB-D Camera

- The devices generally contain RGB cameras, infrared projectors, and detectors that map depth through either structured light or time-of-flight calculations
- They also can contain microphones that can be used for speech recognition and voice control



RGB-D Camera

- Depth calculation:
- A pattern of dots is projected with an infrared laser
- From the infrared light that arrives bounced to the infrared camera, the depth for each pixel is obtained
- The pattern is saved for a known depth. By putting objects in front, the pattern appears distorted and thus the depth can be calculated
- Since there are fewer points in the IR pattern than in the depth map, some parts of this map are interpolated



PointCloudLibrary

- 3D Point Cloud processing Library
- Free distribution software (BSD license), developed by Willow Garage company
- Includes algorithms for:
  - Filtering
  - Feature extraction
  - Surfaces reconstruction
  - Alignment
  - Model adjustments
  - Segmentation
  - Display
- Implemented in C++
- Development platforms Linux, MacOS, Windows and Android
- Natively supports OpenNI 3D interfaces

PointCloudLibrary

- Basic structures
- **PointT**
    - There are different types of predefined points:
        - PointXYZ, PointXYZRGB, PointNormal, etc.
    - User-defined types can be added

- **PointCloud**
    - Allows you to store a cloud of points
        - int width
        - int height
        - std::vector<PointT> points
    - Arranged (height = number of rows) or unarranged (height = 1)

PointCloudLibrary

- PointCloud example:

```cpp
pcl::PointCloud<pcl::PointXYZ> cloud;
std::vector<pcl::PointXYZ> data = cloud.points;
// Arranged
cloud.width = 640; cloud.height = 480;
// Unarranged
cloud.width = 307200; cloud.height = 1;
```

PointCloudLibrary

- PCL Libraries:
- libpcl_filters

  Data filters: resolution reduction (downsampling), outlier removal, etc.
- libpcl_features

  Extraction of 3D features: surface normals, boundary points, SIFT descriptors, NARF…
- libpcl_io

  Input and Output: write and read files in PCD (Point Cloud Data) format
- libpcl_segmentation

  Segmentation: cluster extraction, model adjustments using consensus methods, etc.

Intelligent Robotics Lab

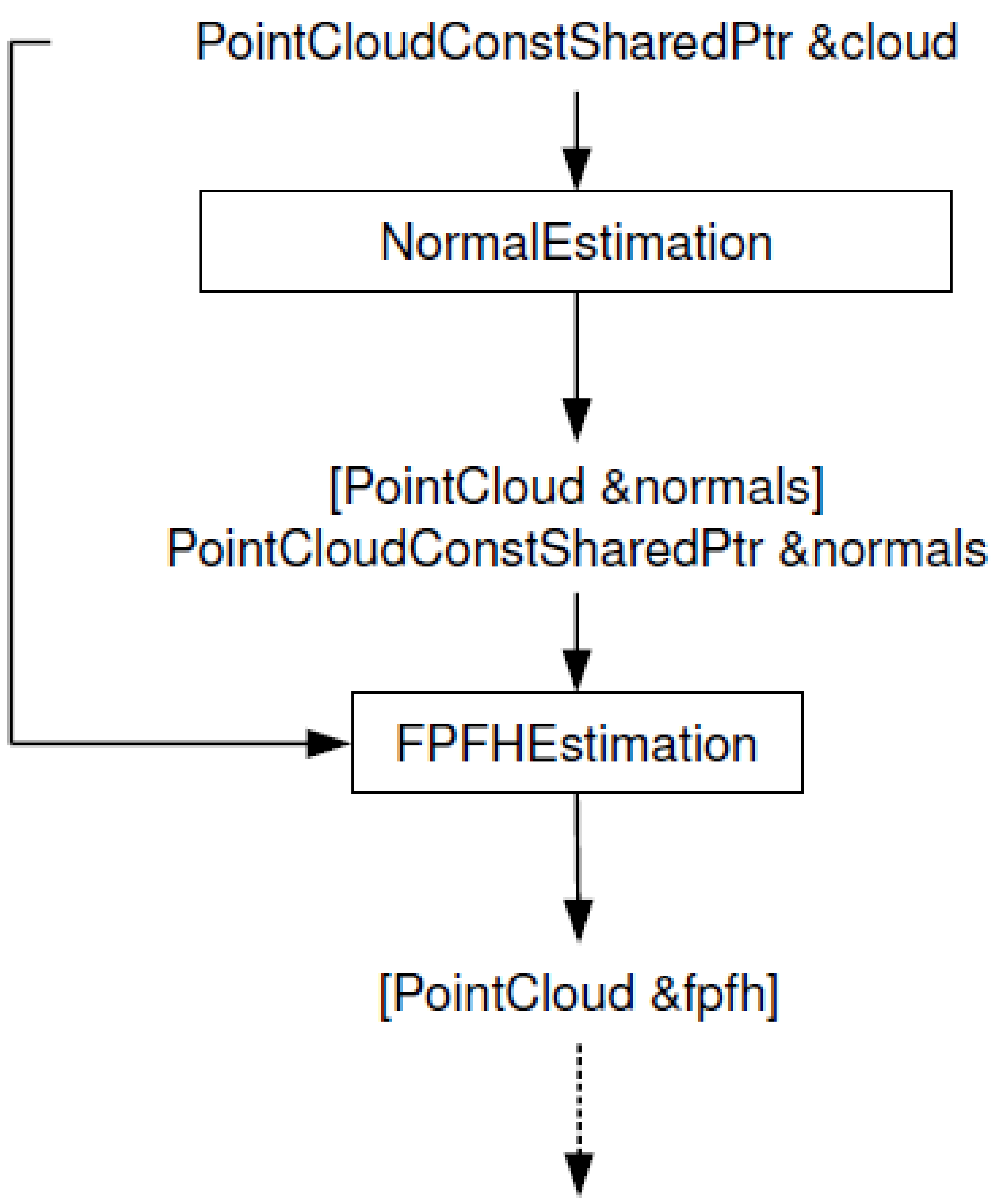
PointCloudLibrary

- PCL Libraries:
- libpcl_surface
  surface reconstruction
- libpcl_registration
  PCl
- libpcl_keypoints
  Extraction of key points that can be used to decide where to extract feature descriptors
- libpcl_range_image
  Support depth maps created from point clouds

PointCloudLibrary

- Pipeline processing in PCL:

1. Create the processing object (filter, feature estimator, segmentation, etc)
2. Use setInputCloud to pass the input point cloud to the render module
3. Update some parameters
4. Invoke the corresponding processing module (compute, filter, segment, etc) to obtain the result



ROS 2 PointCloudLibrary

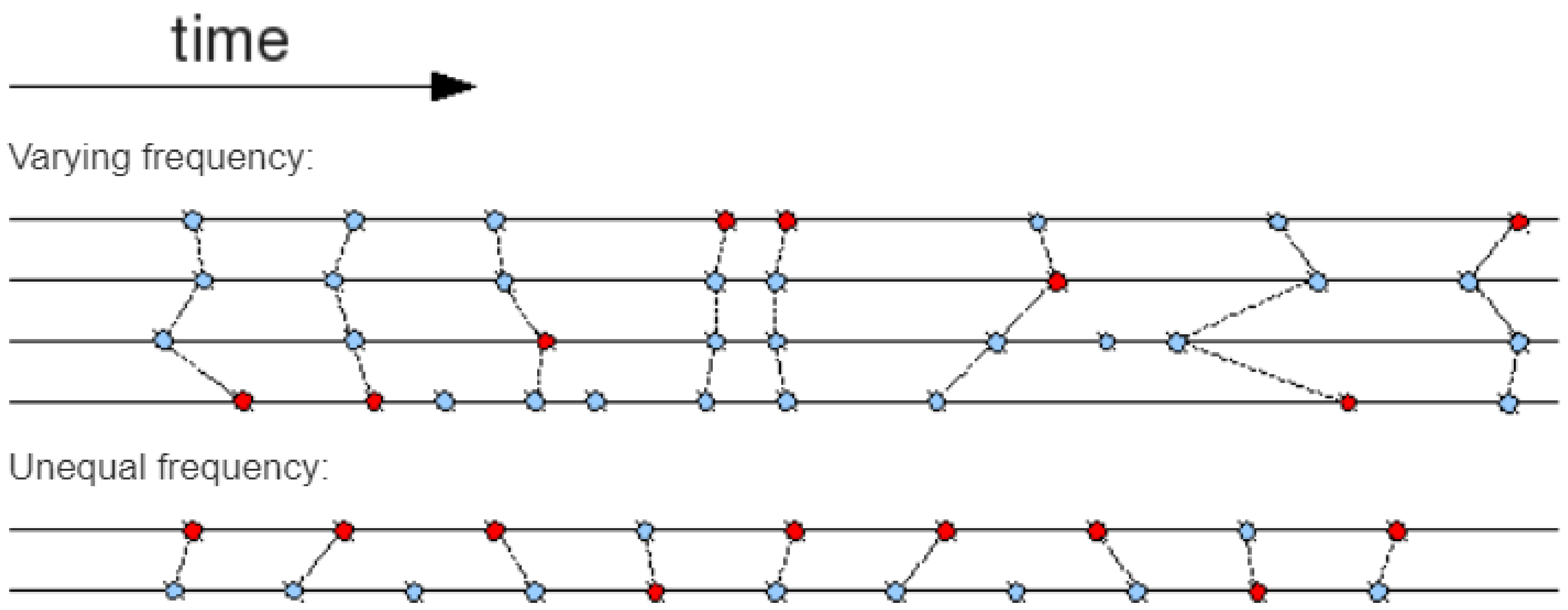
- Publisher/Subscriber ROS PointCloud

```cpp
void PCLPubSub::topic_callback_pointcloud(
  const sensor_msgs::msg::PointCloud2::ConstSharedPtr & pointcloud_msg)
{
  // Check if there is any subscription to the topic
  if (publisher_pointcloud_->get_subscription_count() > 0) {
    // Convert to PCL data type
    pcl::PointCloud<pcl::PointXYZRGB> pointcloud_in;
    pcl::fromROSMsg(*pointcloud_msg, pointcloud_in);

    // PCL processing ...
    pcl::PointCloud<pcl::PointXYZHSV> pointcloud_out;
    pcl::PointCloudXYZRGBtoXYZHSV(pointcloud_in, pointcloud_out);

    // Convert to ROS data type
    sensor_msgs::msg::PointCloud2 out_pointcloud_msg;
    pcl::toROSMsg(pointcloud_out, out_pointcloud_msg);
    out_pointcloud_msg.header = pointcloud_msg->header;

    //Publish Pointcloud
    publisher_pointcloud_->publish(out_pointcloud_msg);
  }
}
```

```cpp
class PCLPubSub : public rclcpp::Node
{

public:
  PCLPubSub()
  : Node("pcl_pub_sub")
  {
    // PointCloud subscription
    subscription_pointcloud_ = create_subscription<sensor_msgs::msg::PointCloud2>(
      "/pointcloud_in", 1,
      std::bind(&PCLPubSub::topic_callback_pointcloud, this, std::placeholders::_1));

    // PointCloud publisher
    publisher_pointcloud_ = this->create_publisher<sensor_msgs::msg::PointCloud2>(
      "pointcloud_out",
      rclcpp::SensorDataQoS().reliable());
  }

private:
  void topic_callback_pointcloud(
    const sensor_msgs::msg::PointCloud2::ConstSharedPtr & pointcloud_msg);

  rclcpp::Publisher<sensor_msgs::msg::PointCloud2>::SharedPtr publisher_pointcloud_;
  rclcpp::Subscription<sensor_msgs::msg::PointCloud2>::SharedPtr subscription_pointcloud_;

};
```

ROS 2 Synchronizer

- Sometimes is useful to synchronize two or more messages
  - Create a synchronizer policy
  - Create a synchronizer
  - Create subscriptions using message_filter
  - Register a callback to the synchronizer




```cpp
typedef message_filters::sync_policies::ApproximateTime<sensor_msgs::msg::Image,
    sensor_msgs::msg::Image, sensor_msgs::msg::PointCloud2> MySyncPolicy;
  std::shared_ptr<message_filters::Synchronizer<MySyncPolicy>> sync_;
  std::shared_ptr<message_filters::Subscriber<sensor_msgs::msg::Image>> subscription_rgb_;
  std::shared_ptr<message_filters::Subscriber<sensor_msgs::msg::Image>> subscription_depth_;
  std::shared_ptr<message_filters::Subscriber<sensor_msgs::msg::PointCloud2>>
  subscription_pointcloud_;
```

```cpp
subscription_rgb_ = std::make_shared<message_filters::Subscriber<sensor_msgs::msg::Image>>(
  this, "/image_rgb_in", rclcpp::SensorDataQoS().reliable().get_rmw_qos_profile());

subscription_depth_ = std::make_shared<message_filters::Subscriber<sensor_msgs::msg::Image>>(
  this, "/image_depth_in", rclcpp::SensorDataQoS().reliable().get_rmw_qos_profile());

subscription_pointcloud_ =
  std::make_shared<message_filters::Subscriber<sensor_msgs::msg::PointCloud2>>(
  this, "/pointcloud_in", rclcpp::SensorDataQoS().reliable().get_rmw_qos_profile());
```

```cpp
sync_ = std::make_shared<message_filters::Synchronizer<MySyncPolicy>>(
  MySyncPolicy(10), *subscription_rgb_, *subscription_depth_, *subscription_pointcloud_);
```

```cpp
sync_->registerCallback(
  std::bind(
    &CVSubscriber::topic_callback_multi, this,
    std::placeholders::_1, std::placeholders::_2, std::placeholders::_3));

void topic_callback_multi(
  const sensor_msgs::msg::Image::ConstSharedPtr & image_rgb_msg,
  const sensor_msgs::msg::Image::ConstSharedPtr & image_depth_msg,
  const sensor_msgs::msg::PointCloud2::ConstSharedPtr & pointcloud_msg) const;
```