EE106A: Lab 6 - Computer Vision *

Fall 2020

Goals

By the end of this lab you should be able to:

- Explain the concept behind pointclouds and what they represent
- Implement some basic techniques in image segmentation and explain your results
- Explain the concept of pinhole cameras and describe how they work
- Combine information from different sensors to isolate an object of interest. (Build a point cloud segmentation pipeline)

Relevant Tutorials and Documentation:

- OpenCV Python Tutorial

If you get stuck at any point in the lab you may submit a help request during your lab section at `https://tinyurl.com/106alabs20`. You can check you position on the queue at `https://tinyurl.com/106Fall20LabQueue`.

Note: For all labs this semester you may collaborate with a lab partner but we expect everyone to do every part of the labs themselves. You should work closely with your partner to overcome obstacles in the labs but each member of the team must do the lab themselves.

Contents

*Converted to remote by Amay Saxena and Tiffany Cappellari, Fall 2020 (the year of the plague). Developed by Amay Saxena and Grant Wang, Fall 2019.

1 Introduction

This lab will introduce you to various techniques used in processing data from advanced sensors like the Intel RealSense. In particular, you will learn about image segmentation, point cloud processing, and how we can combine various data modalities to leverage the strengths and weaknesses of each data form.

You will write your own *point-cloud segmentation pipeline*. This means that by the end of this lab, you will be able to isolate an object of interest in a point-cloud, by combining information you get from RGB image sensors.

2 Starter Code

Create a new workspace in your `~/ros_workspaces` directory called `lab6` and clone the starter code into the `src` subdirectory. After creating your workspace and copying over the contents of `lab6_starter` into `lab6/src`, use `catkin_make` to build your workspace.

```
git clone https://github.com/ucb-ee106/lab6_starter.git
```

After creating your workspace and copying over the contents of `lab6_starter` into `lab6/src`, use `catkin_make` to build your workspace. Then, go into `segmentation/src` and make all python files executable using

```
chmod +x *.py
```

The starter code includes the following packages:

1. `ros_numpy`: A very useful ROS package that can be used to convert between ROS message datatypes and numpy datatypes. Check out the documentation at http://wiki.ros.org/ros_numpy.
2. `segmentation`: This is the package that contains all the skeleton code. All files that you will need to edit are in this package.
3. `ddynamic_reconfigure`: Allows us to update parameters without restarting nodes. You will not need to interact with this package in this lab, but know that it is there.

3 Intel RealSense

In this lab, you will be using data collected from an Intel RealSense camera (see Figure 1), which is a powerful sensor equipped with an RGB camera, a depth camera, an Inertial Measurement Unit, and an IR sensor which is capable of producing a rich colored point-cloud. We will concern ourselves with the RGB camera and the point-cloud. You can get similar data from other RGBD sensors like the Kinect. You can also simulate many such sensors in Gazebo or other simulators, and we encourage you to look into this for your final projects so you can process data collected in real time. For this lab, we wanted you to be able to use real world data, so we will be providing you with data that we collected offline in the form of a ROS "bag" file.

Figure 1: Intel RealSense Camera

3.1 Bag files

Bag files are the canonical "ROS" way of recording data (recall using a bag file back in Lab 3 to "simulate" a Baxter robot). The `rosbag` utility allows us to record messages and TF data being published and then play it back later in the form of a "bag" file. In this lab, we have provided you with a bag file which you can download from here. Unfortunately, bag files are often quite large and we were unable to store it in the GitHub with the rest of the starter code. You should place this file in the `bagfiles` subdirectory of `lab6_starter`. This bag file was recorded while a RealSense camera was pointed at a green MegaBlocks block being moved around on a table. We will be using this bag file as a drop-in replacement for the sensor. Recall that when you implement algorithms using ROS, your nodes only care about what topics and TF frames are being published, so your code can be completely agnostic to if the publishing is being done by a specific sensor or by a datastream like `rosbag`.

Use the following command to play the bagfile in an infinite loop. First, you will have to start-up a master node using `roscore`. This node will be our stand-in for a real sensor.

```
rosbag play -l bagfiles/realsense.bag
```

Use `rostopic list` to look at the various topics being published.

4 Point-Clouds

A point-cloud is a kind of data type used to represent 3D scenes. A point-cloud is simply an unordered set of points. Point-clouds are a very popular 3D data modality. It is the kind of data returned by LiDAR sensors, which are staple as the primary kind of sensor used by self-driving cars.

Typically, these points are given by just their (x, y, z) co-ordinates in the camera's reference frame, but they may include additional dimensions for additional data that the sensor captures for each point, like color. Indeed, the pointclouds published by the RealSense will include 7 dimensions for each point, (x, y, z, r, g, b, a), where (x, y, z) are the co-ordinates of the point, (r, g, b) is the color of the point in RGB format, and a is the intensity registered by the sensor.

We can visualize the point-cloud being published in Rviz. First, open up Rviz by running

```
rosrun rviz rviz
```

Change the fixed frame to be `camera_depth_optical_frame`. Next, add a new Display of type `PointCloud2`. Set the topic for this display to `/camera/depth/color/points`. You may have to wait a bit for Rviz to begin registering pointcloud messages (as you can imagine, point-cloud messages tend to be pretty heavy). To get a better look at the pointcloud, you may have to check the box labelled `Invert Z Axis` in the right hand panel.

You should begin to see the point-cloud displayed in RViz in real time as a big cloud of points. A typical scan from the RealSense will include $\sim 100,000$ points. Note that the refresh rate on point-cloud topics tends to be much lower than that of Image topics (which can be visualized with minimal-to-no lag).

Now, also add two `Image` displays. Set the topic for one to `/camera/color/image_raw`. Set the topic for the other one to `/camera/depth/image_rect_raw`. These two displays will display the RGB and the Depth images respectively.

Our goal for this lab will be to take a pointcloud from the RealSense, and then isolate an object of interest in the scene. We will then publish a new pointcloud to a new topic, that will only contain points that correspond to the object of interest (a process called "segmentation").

Unfortunately, due to the unordered nature of point-clouds, it is usually very difficult to extract useful inferences from point-clouds alone, without first processing them into a different kind of data-structure (like a proximity graph, mesh, or voxel grid). This is because a regular point-cloud has no real structural arrangement to go off of - each point stands alone, and no order is guaranteed.

On the other hand, it is comparatively much easier to extract such features from an RGB image. The inherent structure of a pixel grid means that we have geometric cues to go off of when trying to locate an object of interest. Pixels corresponding to the same object are also close together in the pixel grid. This is also reflected in the state of the art of machine learning methods for object detection. Deep learning methods on images are leagues ahead of the state of the art in object detection directly on point-clouds, owing in large part to the unstructured nature of point-clouds.

4.1 An approach to point-cloud segmentation

Let's say we want to filter out all points from a pointcloud that do not correspond to some object of interest. We'll use a green MegaBloks block. We already claimed that it will be difficult to locate the block in the pointcloud directly. But as it turns out, we will be able to detect it easily the RGB image. So, how can we use this to our advantage to detect the block in the pointcloud?

Well, we can take advantage of the fact that both the RGB image and the pointcloud are scanning the same scene. So our strategy will be this: First, we will locate the block in the RGB image. Then, we will figure out how points in the point-cloud correspond to pixels in the image. After all, each point is just a 3D co-ordinate for some point in the scene that we took the RGB image of, so we should be able **to map each point to some pixel in the image.** Next, we simply keep those points which map to a pixel that was detected as belonging to the green block.

So then, the first step will be to detect the block in the RGB image. Our aim will be to assign to each pixel in the image either a 1 or a 0. A pixel gets a 1 if it belongs to the block, and a 0 otherwise. Such a grid is called a "segmentation" or "segmentation map" of the original image, and the process of producing assigning such a class label to each pixel in an image is called "image segmentation".

Next, we will map each point in the point-cloud to some pixel of the image. Then we will keep any point that lands on a pixel with a 1 in the segmentation map, and discard any point that lands on a 0. Finally, we simply create a new point-cloud that comprises of only the points that we kept, and publish it to some new topic for visualization.

5 Image Segmentation

Image segmentation will allow us to "segment" or partition out our specific object of interest in the image, which in this case will be our MegaBloks block. Our first step will be to create some functions that we can use for image segmentation and then apply it to our block to detect it. There are a number of different ways that image segmentation can be performed, but the ones that we will particularly look at are segmentation via thresholding, edge-detection, and clustering. These are all methods widely used in computer vision and robotics. Note though, since this is not a computer vision course, we will only touch on these methods at a surface-level and the results will be far from perfect. They will, however, be enough to accomplish our task and get you introduced into the rich field of computer vision.

5.1 Color and Grayscale Images

Before we start implementing our segmentation, we need to get you acquainted with some of the common models used to represent images. The two particular models we will focus on are RGB and grayscale. Let's begin by talking about RGB images. The idea behind RGB images is that with the 3 base colors red, blue, and green (hence RGB), we can mix them to create pretty much any color that most humans can distinctly recognize. Thus, we can represent an image as a 3 color channel system where each pixel contains 3-channels of red, green, and blue intensity values combined, and we can adjust these three values to change the particular color at a pixel. This representation allows

us to more formally define a color image as a 3-dimensional matrix, where each dimension is of size height x width of the original image and each element constitutes a pixel that can take on any intensity value between 0-255 (each color channel is 8 bits). What color do you think we get when we combine equal intensities of red, green, and blue?

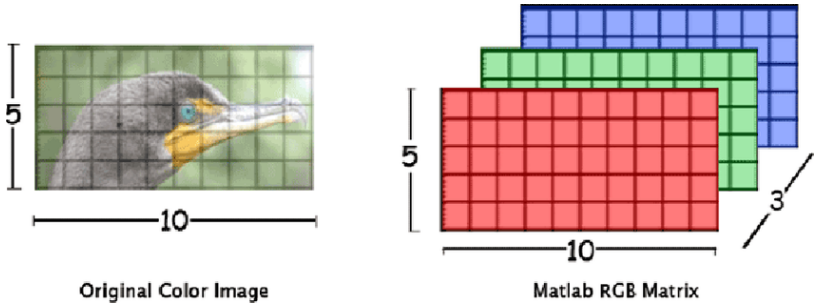

Figure 2: Matrix RGB representation of Color Image.

Sometimes, though, we may be only interested in a more compact representation - each pixel is just the particular amount of light. This is where grayscale images come in to play. Think of a grayscale image as a simpler version of an RGB image. Each pixel of a grayscale image represents just intensity (amount of light) and is composed entirely of shades of gray (mixtures of black and white). Black is of weakest intensity and white is of strongest. Using the grayscale model, any image is now in the form of a single channel 2-D matrix, where each element corresponds to a pixel and takes on an intensity value between 0 (pure black) and 255 (pure white). See Figure 3 for a visualization.

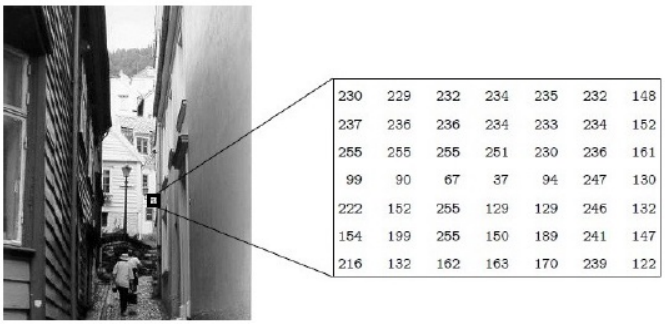

Figure 3: Matrix grayscale image.

A good amount of image processing and computer vision deals with extrapolating information from and manipulating the values in these image matrices. We will be exploring the properties of and manipulating grayscale and RGB images in image segmentation.

5.2 Thresholding

The simplest method of image segmentation is through the thresholding method. We will particularly explore grayscale thresholding, although this method can be generalized to color-based thresholding as well. The thresholding method makes use of the fact that if we have an object of interest and a background of a different grayscale intensity, we can "threshold" or clip the image pixel values by setting values above the threshold to be white and those below the threshold to be black (or you can do the other way around as well). More specifically in this lab, we will keep things simpler by working with objects on our table (which has a mostly uniform background) and we will define a threshold range that represents the intensity of the table and attempt to subtract it out. We will set a pixel to be low if in this range (is part of the background), else high (is foreground e.g. our object).

The result of this threholded image is a binary image (not grayscale) with 1's (white) representing our foreground or object of interest, and 0's (black) representing everything else.

Let's implement our thresholding strategy in code. Open up `image_segmentation.py` and implement the function `threshold_segment_naive`.

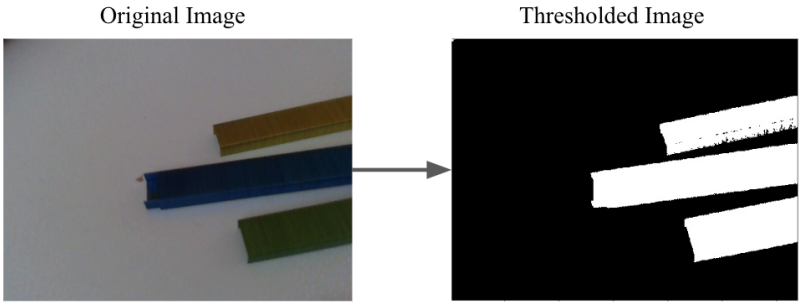

Figure 4: Thresholding an image.

We've included a couple sample images (`lego.jpg`, `legos.jpg`, and `staples.jpg`) for you to test your implementation. Go down to the `main` function and uncomment the line `test_thresh_naive` if it is not already. Pass in the image of interest into `read_img`.

Then run the script:

```
rosrun segmentation image_segmentation.py
```

You will need to tune the `lower_thresh` and `upper_thresh` values depending on the image.

5.3 Edge Detection

Thresholding is a good choice when we know that the background of the image and the objects of interest in the image have quite different grayscale intensities, allowing us to threshold easily and partition the two. However, such scenarios are not always the case, and if the background and object have quite similar grayscale properties, all bets are off. Luckily though, objects have a boundary that separate them from the background: the edges. If we can detect the edges of different objects, then we have knowledge of where the objects are in the scene.

An edge can be defined as any region in the image where there is a sharp change in intensity. A way of expressing this change in intensity is using derivatives or gradients (the multi-variable equivalent of the derivative). Imagine if we were to flatten out our grayscale image matrix into a 1-D signal of pixel intensities and a part of the image looked something like this when the pixel values are plotted:

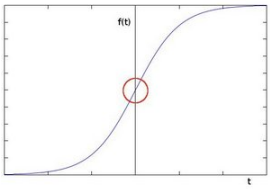

Figure 5: The point highlighted by the red circle represents a sharp change in intensity.

A sharp change in the first derivative reflects a sharp change in intensity of the image, or where there is an edge. So if we can compute the first derivatives of the pixels of our image in both the horizontal and vertical directions, we can combine them to get the gradient. And pixels where the gradient is high are where edges occur.

A way to compute the gradients at each pixel is by using Sobel Filters, which are widely used in edge detection. To use Sobel filters to detect edges, we perform what is known as a convolution between our filters represented by the matrices in Equation 2 and the image of interest I to output G_x and G_y, which represent the horizontal and vertical first derivatives of our image. We define K_x and K_y to be:

$$K_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \tag{1}$$

We define G_x and G_y to be:

$$G_x = K_x * I \qquad\qquad G_y = K_y * I \tag{2}$$

The result is a new image matrix G_x which represents the first derivative in the horizontal direction at each pixel of the original image I and G_y the vertical direction. The details of convolution are beyond the scope of this course, and all you have to know is that the 3x3 Sobel filter slides across each 3x3 region of interest in the image, a dot product is being computed between this filter (matrix) and the region of interest, and the outputted value represents the derivative at that pixel. For a good visualization of how convolution in imaging works, we refer you to this, and if you want to learn more about the math this is a start or take EE 120/123. (the link shows convolution of images for RGB images, since we're only working with grayscale, we only care about performing convolutions on a single dimension). One nuance is that before we perform convolution, we need to pad our original image with 0's on all sides, why do you think that's the case?

Finally, we approximate the gradient G at each pixel by combing G_x and G_y using the following equation:

$$G = \sqrt{G_x^2 + G_y^2} \tag{3}$$

G will give us all the detected edges (locations where the gradient is high) of the original image. Let's now implement this in code. We will take advantage of some existing functions in OpenCV (a popular computer vision package). One subtle thing we will do in our implementation is to blur the image first using Gaussian blurring (see starter code for more details). This is done commonly to remove unwanted noise in the image. Fill in the function `edge_detect_naive` in `image_segmentation.py`. To test your implementation, uncomment the line `test_edge_naive` in `main` and try it on some sample images provided.

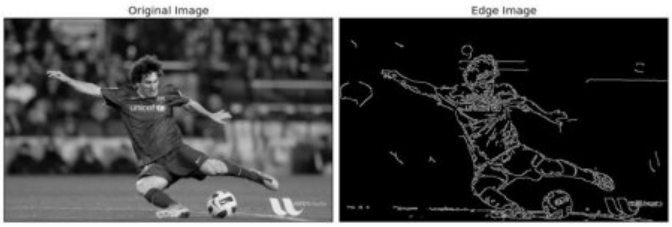

Figure 6: Edge detection using the Canny Edge Detector.

What we've just implemented is actually the first two steps of the famous Canny Edge Detector. To test how your edge detector does in comparison, uncomment out `test_edge_canny` and compare your images. What differences do you see?

5.4 Clustering

We've now looked at two methods of image segmentation that take an analytic approach. The last method of image segmentation we will explore takes a more empirical approach by treating an RGB image pixels as a set of data points with features that we are interested in using for classification. Note that we will be using RGB images instead in this part, rather than grayscale in the previous two methods. The features in this case are the Red, Green, and Blue intensity values at each pixel. Clustering is the task of partitioning the set of data points into different groups, or "clusters", such that data points in the same cluster are assigned the same label because they have more similar feature values to other data points in its cluster than those outside of the cluster. (i.e. pixel values with similar RGB intensities will be clustered with one another).

In the context of Image Segmentation, we can generate clusters for pixels of the image that share similar RGB intensities – they are part of the same thing in the image. Each cluster is a segmentation of the image that could represent the background, an object, etc. Figure 7 gives a high-level idea of how we are performing segmentation via clustering. Each data point, or pixel is plotted. Pay attention to the axes, which represent the features: RGB intensity values of each data point. Notice how pixels part of the same object in the image share the same cluster.

The algorithm we will use to generate these clusters is the K-Means algorithm, an unsupervised classification algorithm. We will not be going into the details of K-means. The steps of the algorithm, however, are actually pretty simple and we suggest you read through this step by step guide to get a feel of how the algorithm works. We will leverage the built in `kmeans` function in opencv. All you have to know is that we will feed as input into the kmeans the pixels of our image as the data points, and K-means will output a clustered image that has the same label for the

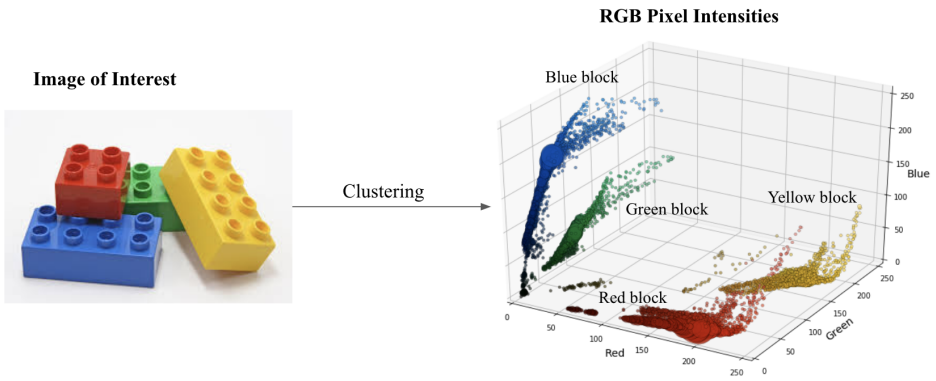

Figure 7: Clustering the pixels of our image via Red, Green, and Blue intensities as our features.

set of pixels in the same cluster but different from other clusters (this is our segmented image). We have provided the function `do_kmeans` that uses `opencv` to perform kmeans clustering for you.

Open up `image_segmentation.py` again and fill in the `cluster_segment` function. Comprehensive line by line notes have been provided in this part to get you started. Note, we first downsample the image before passing it into the K-Means algorithm to speed up clustering (why do you think that's the case?), and then when we want to return our segmented image we upsample the clustered image back to our original size. This has already been implemented for you, just make sure you're passing in the right variables to your function calls.

To test your implementation, go down to the `main` function and uncomment the line `test_cluster`. Play around with the parameter `n_clusters`. What should you set this value to be depending on the image?

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Demonstrate that your thresholding, edge-detection, and clustering implementations can segment all the images provided: `rgb_test.jpg`, `legos.jpeg`, and `lego.jpg`.
- Explain how your implementations worked for each method of image segmentation.
- Explain the performance differences and shortcomings of different segmentation methods on different images.

---

6 Projective Geometry and The Camera Matrix

Now that we have a segmentation map for the image where we can isolate which pixels belong to the green block, our next objective is to compute a correspondence between points in the point-cloud and pixels in the image. Essentially, we would like to figure out how a point in 3D space gets transformed by the camera lens and projected onto the image plane. Or more formally, we would like to compute the transform between 3D co-ordinates in the camera reference frame (X, Y, Z) to image plane co-ordinates (u, v).

6.1 Homogenous Co-ordinates

It is sometimes convenient to define points in terms of homogenous co-ordinates, wherein we append a 1 to the co-ordinate representation of points. This often allows us to get away with having only linear transformations (of the type $f(x) = Ax$) in a situation where we would otherwise need an affine transformation (of the type $f(x) = Ax + b$).

A point (X, Y) in $\mathbb{R}^2$ can be represented in homogenous coordinates by adding one additional dimension. The homogenous representation of this point would be (XT, YT, T) where T is an additional dummy variable. For any value of T, this will be a valid homogenous representation of the point (X, Y). In this way, we associate a point (X, Y) in $\mathbb{R}^2$ with a line in $\mathbb{R}^3$.

A point (X', Y', T) given in homogenous coordinates can be converted to the point (X, Y) that it represents by simply dividing through by the dummy coordinate:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} X'/T \\ Y'/T \end{bmatrix}$$

Feel free to think of this coordinate system as just a construction designed to make some of the math more convenient. We will comment on why this correspondence between points on a plane and lines in 3D arises in the problem at hand in a later section.

6.2 The Pinhole Camera

We will model our camera as a *Pinhole Camera*. The pinhole camera model defines the geometric relationship between a 3D point and its 2D corresponding projection onto the image plane. When using a pinhole camera model, this geometric mapping from 3D to 2D is called a *perspective projection*.

Let's denote the center of the perspective projection (the point in which all the rays intersect) as the optical center or camera center and the line perpendicular to the image plane passing through the optical center as the optical axis (see Figure 8). Additionally, the intersection point of the image plane with the optical axis is called the principal point. The pinhole camera that models a perspective projection of 3D points onto the image plane can be described as follows.

6.3 The Intrinsic Camera Matrix

We will model our camera as a standard pinhole model camera. Consider Figure 8.

Here, we take the co-ordinates of the point p in the camera reference frame to be (X, Y, Z), and the image plane co-ordinates of the projected point p' to be (u, v). Our final objective is find a pair of natural numbers (U, V) that are the index of the pixel onto which the point p is projected.

The standard way to define the axes of the camera reference frame is to have the z-axis point in the direction that the camera is looking in, and have the x and y axes aligned with the image plane. We can show that the equation relating the image plane coordinates to 3D space coordinates is given by:

$$\begin{bmatrix} u' \\ v' \\ w \end{bmatrix} = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{4}$$

Where (u', v', w) is a *homogenous coordinate representation* for the image plane coordinates (u, v) of the point. To recover (u, v) we simply need to divide through by the dummy coordinate:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{w} \begin{bmatrix} u' \\ v' \end{bmatrix} \tag{5}$$

In the above two equation, we have the following:

- (X, Y, Z) is the 3D space coordinates of the point p in the camera's reference frame.
- (x_0, y_0) are the coordinates of the center of the image frame in a coordinate frame affixed to the top left of the image. (x_0, y_0) is given in pixels.
- f_x and f_y are *focal lengths* of the lens, in pixels. Since our camera produces a rectangular image (which wouldn't happen with a pure pinhole camera), we model it as having two focal lengths, one corresponding to the height of the image, and one corresponding to its width.
- s is *axis skew*. This parameter will be nonzero if the given lens produces a *shear* distortion in the image; in other words, it encodes situations where the pixels are parallelograms rather than rectangles or squares. This is not the case for our camera, so for us s will be zero.

The 3x3 matrix in Equation 4 is called the *Intrinsic Camera Matrix*. *Camera Calibration* is the process of estimating this matrix, along with a host of other parameters.

Finally, we can get the indices of the pixel onto which the point is projected as the floor of the image plane coordinates:

$$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} \lfloor u \rfloor \\ \lfloor v \rfloor \end{bmatrix} \tag{6}$$

The RealSense has already been calibrated for you, and its calibration data is continuously published to the topic `camera/color/camera_info`. The typical message type used by such topics is the `sensor_msgs/CameraInfo` message type.

Task 1: Look up documentation on the `sensor_msgs/CameraInfo` message type, and then implement the function `get_camera_matrix` in the file `main.py`. This function should accept a ROS Message of the above type, and should return the 3x3 camera intrinsic matrix as a numpy array. You will need to figure out what field in this message type gives you this matrix (*Hint 1:* Typically, this matrix is represented by the letter K) (*Hint 2:* `numpy.reshape` may be useful here).

6.4 Projecting the Point-cloud

Now, we are ready to project our pointcloud onto the frame of the RGB image. For a given point $\hat{p}$ in some world reference frame, we can compute the indices (U, V) such that $\hat{p}$ gets projected onto the pixel `image_array[V][U]` using the following steps:

1. Represent $\hat{p}$ in the reference frame of the camera. If the transform taking $\hat{p}$'s reference frame to that of the camera is (R, t), then find $p = R\hat{p} + t$. We will need to do this, since the pointcloud is generated in the reference frame of the depth camera, which has a slight offset from the RGB camera.
2. Find q' - the homogenous representation of the image plane coordinates of p - using eq (4):
$$q' = Kp$$
3. Convert the homogenous representation $q' = (u', v', w)$ into an ordinary point $p' = (u, v)$ on the plane using eq (5).
4. Finally, convert the coordinates (u, v) into pixel indices (U, V) by taking the floor, as in eq (6).

Task 2: Fill in the function `project_points` in file `pointcloud_segmentation.py`. This function takes as input a pointcloud of (x, y, z) points given as a 3xN numpy array, the camera intrinsic matrix, the (R, t) transform that converts points in the pointcloud to the reference frame of the RGB camera, and the dimensions of the image produced by the RGB camera. This function should return a new numpy array of integers, of size 2xN, where the ith pair is the pixel coordinates (U, V) of the ith point in the pointcloud.

Fill in the blanks in the code to implement steps 1-4 above.
Once you have an implementation, run the sanity test with

```
rosrun segmentation test_projection.py
```

You will see a reference image, and the result of projecting a corresponding reference pointcloud using your projection code. Make sure that the two frames match appropriately.

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Using the sanity test, show your TA that your code correctly projects the reference pointcloud.
- Explain what each entry in the camera intrinsic matrix represents.

---

7 Putting it all together

Now you have working implementations of both image segmentation and point-cloud projection. Next, we will start up a node that subscribes to topics for the RGB image, point-cloud, and camera info from the RealSense sensor, and will then publish a segmented point-cloud to a new topic. This node has been written for you, and you can look at its implementation in the file `main.py`. This node performs the following tasks, in order:

1. Get a tuple (I_t, P_t, K_t). I_t is the most recent RGB image at time t, P_t is the most recent point-cloud at time t, and K_t is the most recent camera intrinsic matrix at time t. We can get this by subscribing to three topics, one for each of those datapoints.

2. Use your image segmentation code to create a segmented image I'_t from I_t. This segmented image should have a 1 for any pixels that belong to your object of interest, and 0 elsewhere.

3. Use your pointcloud projection code to project every point of P_t onto pixels of the segmented image I'_t.

4. Create a new pointcloud P'_t by keeping any points from P_t that landed on a nonzero pixel, and discarding any point that landed on a zero pixel. P'_t should now only contain points belonging to the object of interest.

5. Publish P'_t to a new topic `segmented_points`.

The following are a few salient features of this node.

7.1 Time Synchronization

Notice that in step 1 above, we need to acquire three data points (I_t, P_t, K_t) from three different datastreams (topics). However, we also want to ensure that (I_t, P_t, K_t) come from approximately the same time. For instance, it may be the case that the topic publishing pointclouds has much greater lag than the other topics, or that one of the three topics is stalled for some reason. In this case, if we just naively use the last received message from each of the topics, then we will be projecting an old pointcloud onto a new image, which will give us an incorrect result. So we want some way to ensure that the pointcloud and image we use were collected close to each other.

The built-in ROS package `message_filters` has functionality that will give us exactly what we want. Instead of defining three naive subscribers, each with its own callback, we can instead define a single *message filter* that listens to three topics, and only lets through triplets of messages that satisfy the "approximate time" condition. The `ApproximateTimeSynchronizer` implements this functionality, and it allows us to define one single callback that takes three message arguments. The time synchronizer filter will make sure that only triplets of data with close timestamps get sent to this callback.

7.2 Queue Processing

One way to implement steps 1-5 is to perform all computations inside the subscriber callback, and then publish from within the callback itself. While this is certainly possible, it is not advisable. Instead, the callback simply pushes the triple (I_t, P_t, K_t) onto a queue, from where a separate routine pops, processes, and publishes.

Task 3: We will now put everything together and process both images and pointclouds from the sensor. Make sure the bag file from earlier is running.

In `image_segmentation.py`, fill in the function `segment_image` with an image segmentation algorithm of your choice. By default, it uses your thresholding implementation (you will need to put in the right thresholds). You can also experiment with using the clustering implementation instead.

Open up RViz. Set the fixed frame to `camera_depth_optical_frame`. In the right hand side window, check the box labelled `Invert z-axis`. Create an `Image` display and set its topic to `camera/color/image_raw`. Now create a `Pointcloud2` display, and set its topic to `segmented_points` (you may not able to do this until after you have started up `main.py`).

Now start the main node with the following command. You should see a point-cloud appear in RViz with only your MegaBloks block visible in it. How accurate this is will be a function of your image segmentation implementation, so feel free to play around with the hyperparameters until you get something that looks good. Also note that if you are using clustering, there may be noticeable lag in this pointcloud. This is to be expected, and happens because your image segmentation implementation is slow. You may be able to speed it up by, for instance, downsampling the image by a greater factor before segmentation and then upsampling the result.

```
rosrun segmentation main.py
```

Checkpoint 3

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Show your segmented pointcloud in RViz to your TA.
- Explain the functioning of `main.py`.

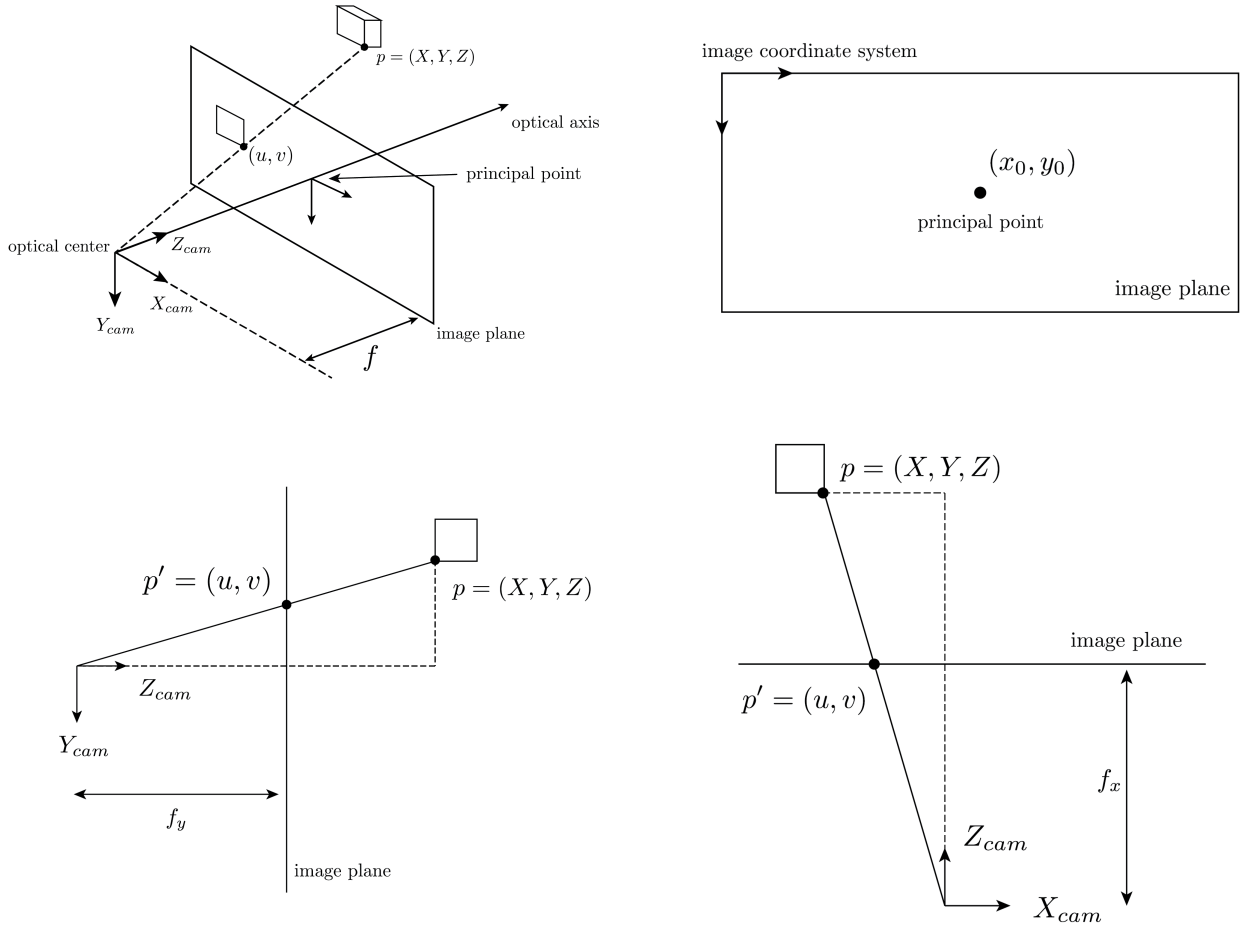

Figure 8: Geometry behind a Pinhole Camera