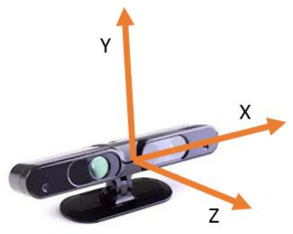
Dpto. Ingeniería de Sistemas y Automática
Universidad de Málaga

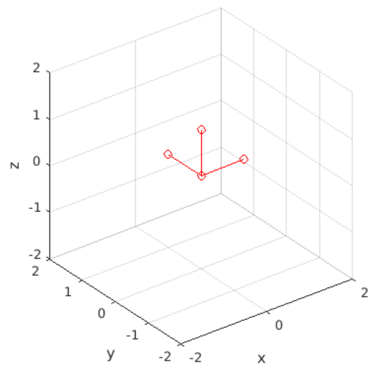
VISIÓN POR COMPUTADOR

Exercise 9: Camera model.

Concepts: Perspective projection matrix, spatial transformation.

1. **Homogeneous transformations:** Given a camera located at the origin of coordinates, which the following reference frame:





Obtain the transformation matrices (4 x 4) to:

Transformation	Equivalent in videogame
Move the camera 0.5 meters forward.	The player moves 0.5 meters forward, i.e. he/she is walking in straight line.
Rotate the camera 35º to look to the left (yaw rotation). *Note: the angles must be introduced in radians. You can use the helper function **deg2rad(deg).***	The player rotates 35º to the left in place.
Move the camera 0.5 meters upwards and rotate it downwards.	The player jumps and looks down, for example, to observe the same object from a higher perspective, to jump on a box, etc.

Recall that a transformation matrix has the following form:

$$T = \begin{bmatrix} R_{3x3} & t_{3x1} \\ 0_{1x3} & 1 \end{bmatrix}$$

Show the original and the transformed reference frames with ***showTransformation(T)***, where *T* is the transformation matrix of each exercise part.

```
% Initial translations
    tx = 0; ty = 0; tz = 0;

% Initial reference frame
    T = [1  0  0  tx
         0  1  0  ty
         0  0  1  tz
         0  0  0   1];
```

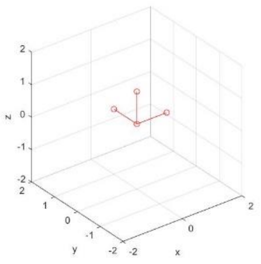


Image 1

```
% Move the camera 0.5 meters forward
     tx = 0.5; ty = 0; tz = 0;
     T = [1  0  0  tx
          0  1  0  ty
          0  0  1  tz
          0  0  0   1];
```

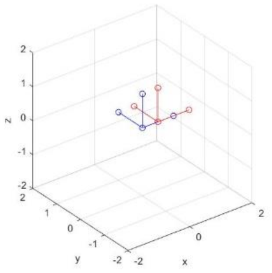


Image 2

```
% Rotate the camera 35degrees to look to the left (yaw rotation)
     tx = 0; ty = 0; tz = 0;
     yaw = deg2rad(35);
     cy=cos(yaw);
     sy=sin(yaw);
     I = [1  0  0  tx
          0  1  0  ty
          0  0  1  tz
          0  0  0  1];
      R =[cy -sy  0  0
          sy   cy  0  0
          0    0   1  0
          0    0   0  1];
      T=I*R;
   % Move the camera 0.5 meters upwards and rotate it downwards
     tx = 0; ty = 0; tz = 0.5;
     pitch = deg2rad(60);
     cp=cos(pitch);
     sp=sin(pitch);
     T = [cp  0    sp  tx
          0   1    0   ty
         -sp  0    cp  tz
          0   0    0   1];
```

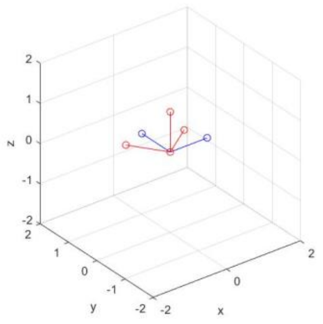


Image 3

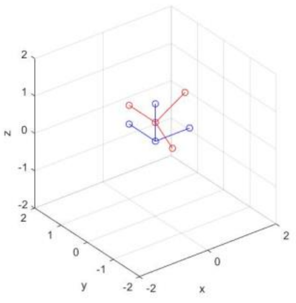
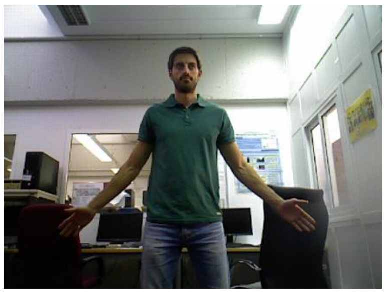


Image 4

2. **Camera-to-world transformation**: In this exercise we will work with images provided with an RGB-D camera. These cameras provide standard RGB images together with depth images which measure how far is the point observed by each pixel.



Thus, if the intrinsic parameters are known (matrix K), we can compute the 3D coordinates of the observed points. To do that, you have to:

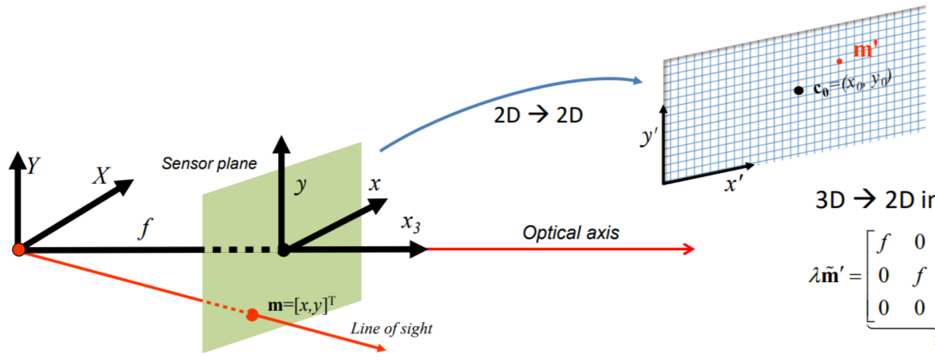
- Read the RGB and depth images provided and show them (you will have to re-scale the depth image to visualize it properly). To read the depth image use ***imread( )***.

```
% Read the rgb and depth images images
im_rgb = imread('person_rgb.png');
im_depth = imread('person_depth.png');
```

- Obtain the 3D coordinates of the observed points (relative to the reference frame of the camera), assuming that $f = 525, k_x = k_y = 1, \theta = 90°, x_0 = 319.5$ and $y_0 = 239.5$.

```
% Camera intrinsic parameters
  f = 525;
 x0 = 319.5;
 y0 = 239.5;
```

Recall the following slide for obtaining the 3D coordinates of the line passing through a point in the image. Notice that in this formulation the reference system of the image is placed at the bottom left part of the image, and in Matlab it is placed at the top left part.



3D → 2D in homogeneous

$$\lambda\tilde{\mathbf{m}}' = \underbrace{\begin{bmatrix} f & 0 & x_0 & 0 \\ 0 & f & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{P}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \mathbf{P}\tilde{\mathbf{M}}$$

2D → 3D in homogeneous

$$\tilde{\mathbf{M}} = \lambda \underbrace{\mathbf{P}^T(\mathbf{P}\mathbf{P}^T)^{-1}}_{\mathbf{P}+}{}'\tilde{\mathbf{m}}' = \lambda \underbrace{\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ x_0 & y_0 & 1 \\ 0 & 0 & 0 \end{bmatrix}}_{\mathbf{P}^T} \underbrace{1/f^2 \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ -x_0 & -y_0 & f^2+x_0^2+y_0^2 \end{bmatrix}}_{(\mathbf{P}\mathbf{P}^T)^{-1}} \tilde{\mathbf{m}}' = \lambda/f \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & f \\ 0 & 0 & 0 \end{bmatrix} \tilde{\mathbf{m}}' = \lambda \begin{bmatrix} \frac{x'-x_0}{f} \\ \frac{y'-y_0}{f} \\ 1 \\ 0 \end{bmatrix}$$

- λ = Z
- Fourth element is 0 → it's a vector = line in 3D!

Note: $\mathbf{P}^+ = \mathbf{P}^T(\mathbf{P}\mathbf{P}^T)^{-1}$ *is a right pseudoinverse since* $\mathbf{P}\mathbf{P}^+ = \mathbf{I}$

```
% Depth scale and max depth considered
scale = 0.0002;
max_depth = 5;

% Obtain the 3D coordinates of each pixel
for y=1:rows
    for x=1:cols
        points_z(y,x) = double(im_depth(y,x))*scale;
        if points_z(y,x) > max_depth % check max depth
            points_z(y,x) = 0;
        end
        points_x(y,x) = -1*((x-x0)/f)*points_z(y,x);
        points_y(y,x) = -1*((y-y0)/f)*points_z(y,x);
    end
end
```

- Plot the 3D points with their corresponding color with ***plot3DScene(****X,Y,Z,image_RGB, downsample****)***.

```
plot3DScene(points_x,points_y,points_z,im_rgb,downsample);
```

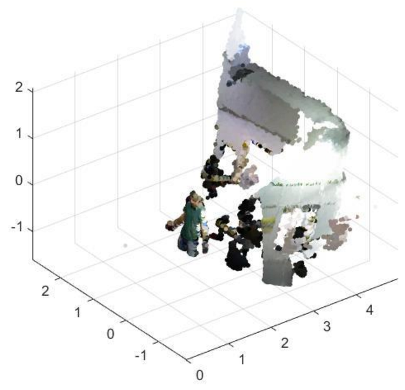


Image 5. Result of the algorithm

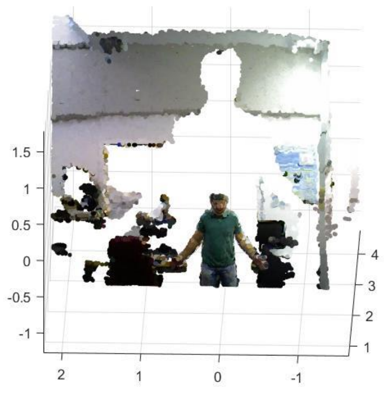


Image 6. One part of the result

As we can see in Image 5, the result looks like if we project the image from a camera.

3. **World-to-camera transformation:** Now we will simulate views of the same scene from different perspectives. To that end, you have to define the homogeneous transformation [R t] associated to these new perspectives and use it, together with K, to create the new images. From the last lesson:

$$\lambda \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} fk_x & 0 & x_0' \\ 0 & fk_y & y_0' \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K}\ \text{Scale + translation}} [\mathbf{R}\ \mathbf{t}] \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

$$\boxed{\lambda \tilde{\mathbf{m}}' = \mathbf{K}[\mathbf{R}\ \mathbf{t}]\tilde{\mathbf{M}}_W}$$

Generate at least two images from different perspectives moving the camera around the observed points. Take into account that these images will only look “reasonable” if the camera does not get very far from its original location.

```
for x=1:cols
        for y=1:rows
            […]
            kx=1;ky=1;
            K=[f*kx  0     x0
                0     f*ky  y0];
            M=[-x_t/z_t(y,x);-y_t/z_t(y,x);1];
            m=K*M;
            %M=[-x_t;-y_t;z_t(y,x)] (1)
            %We divided the Vector 1 because z_t has his own scale and we
            %need to elimate that becuase te other valor has that scale and
            %mess with the result
            proj_x(y,x) = m(1);
            proj_y(y,x) = m(2);
        end
    end
```

Use the function ***renderNewImage(****x_proj, y_proj, depth_transformed, image_RGB****)***.

```
renderNewImage(proj_x,proj_y,z_t,im_rgb);
```

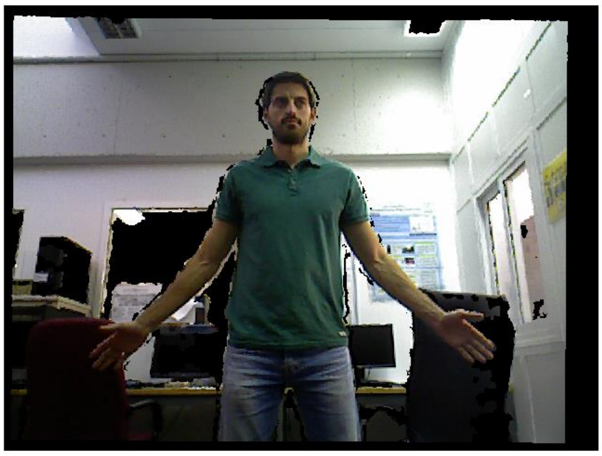


Image 7. Original image

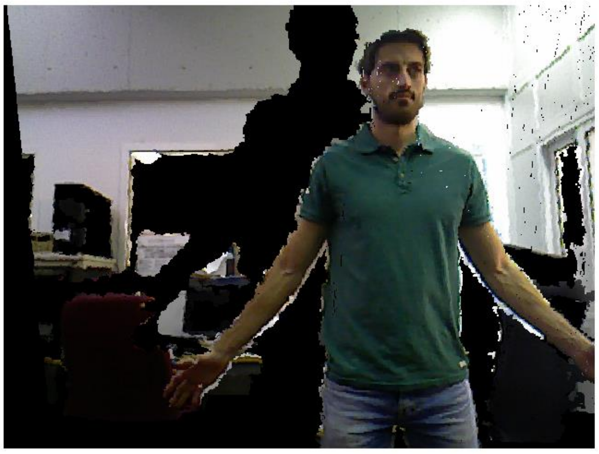


Image 8. Image moved and pitched

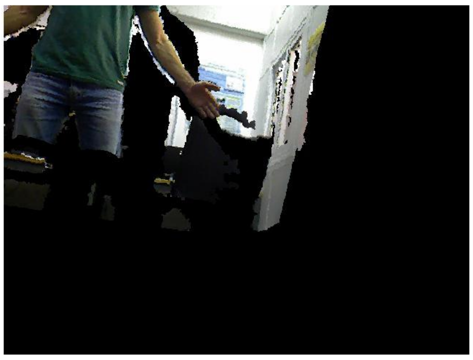


Image 9. Image moved and rotated

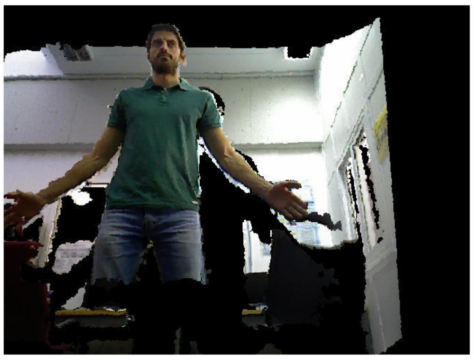


Image 10. Image moved and pitched

As we see in the images, we can see when we move the image, there are no things behind. That is because we only have a image, the algorithm cannot fill these parts with any information.