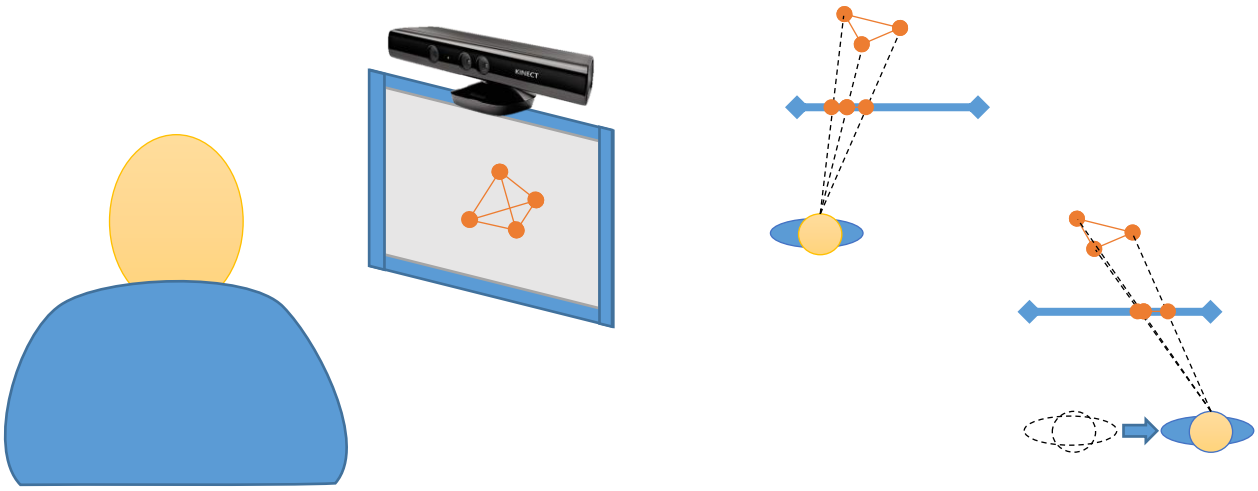
LW2: Full parallax 3D display with head tracking

The objective in this assignment is to create a head tracking 3D display using Matlab and a webcam. The sensor tracks the location of the viewer, and your task is to render the appropriate content on the computer screen according to the location. **List clearly, which steps you have completed** in the comments in the beginning of your Matlab script. Completing tasks 1-3 is required to pass the exercise with grade 1, and completing each additional task will increase the grade by 1.

To complete the whole exercise, you will need a web camera (or equivalent) on a PC with Matlab on it. Any webcam, either integrated to a laptop screen or a separate USB one should work. If you don't have Matlab on your PC, you can get a student version of it by registering with your @student.tut.fi email to Mathworks. For more information, see https://portal.tut.fi/group/pop/study-info/it-services/software-and-licenses. Note that you need administrator rights to install a) Matlab, b) the webcam adapter. If nobody in the group has access to a PC with web camera and a possibility to install the student version of Matlab on it, contact olli.j.suominen@tut.fi to find alternative solutions.

Before starting to implement the tasks, it is important to understand the desired outcome instead of routinely following directions. The resulting display should work in such a way, that when you move your head in front of the display, you see the object on screen from different perspectives. Ideally, as if the object was actually on the table instead of the computer screen. Therefore, you can peek around the corner to see the sides of the cube.

The demo code shows the result with a set of parameters for a 14" laptop. If yours is different, the demo will not be exactly accurate, but still behaves more or less correct. Note that the demo code draws a more comprehensive visualization of the setup. You only have to reproduce the part called "Main window", the rest are just to help in understanding the functionality. Since the image acquisition toolbox doesn't always play nice, *imaqreset* will come in handy.



1. Model creation (mandatory)

Create a simple wireframe 3D model of a cube consisting of N=8 vertices and M=12 polygons (triangles) by manually defining the points and the connectivity.

- Each vertex is a 3D point (x_n, y_n, z_n) in 3D space and the set of vertices is stored in a 3xN matrix of coordinates.
- Each polygon is a connection between three vertices, and is stored as triplets of indices into the array of vertices, i.e. 3xM matrix where each column represents a connection between the indexed vertices.
- The coordinates are given in meters and relative to the upper middle point of the physical display of your computer. (See Figure 1). We'll call it *sensor origin*. Define the model in such a way that if your display was a window and your model a physical object at the coordinates on the table behind the display, you would be able to see it.

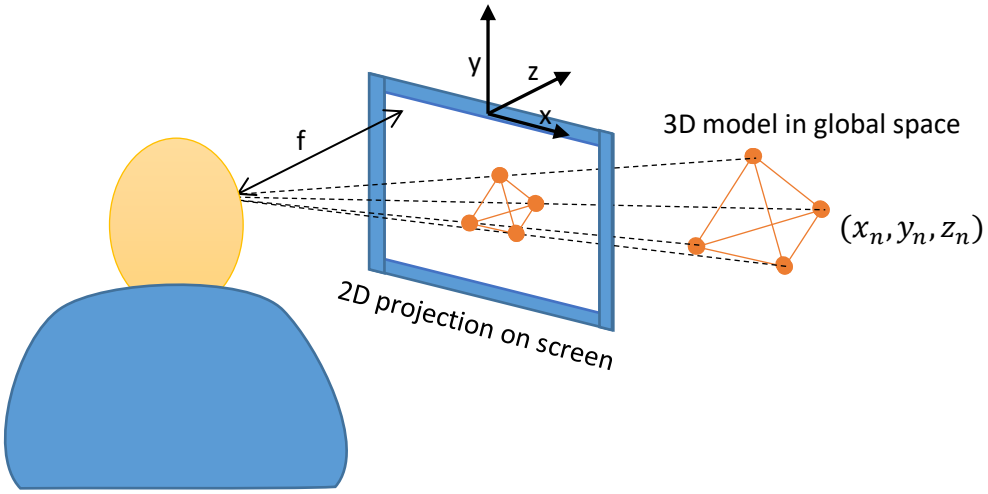


Figure 1 Coordinate space

Draw the 3D model using *fill3*. Hint: a for-loop might come in handy

2. Perspective projection (mandatory)

Apply perspective projection (Eq. 1) to project the (x_n, y_n, z_n) model to the (u, v) image plane of a pinhole camera. Make the display dependent parameters variables instead of hard coding them, so you can easily change them if the display changes. Measure an approximate position of the viewer as translation $\boldsymbol{t}$ from the sensor origin (see Figure 2). Assume that the viewer is positioned in the middle of the screen.

$$(u, v) = \left(f_x \frac{x}{z} + c_x, f_y \frac{y}{z} + c_y\right). \quad (1)$$

- The image plane is considered to be the same as your display surface
- The focal length of the pinhole camera is the approximate distance of the viewer from the screen (measure it). To express the focal length in pixels, divide the measurement with the physical pixel size of your display (measure that too).
- The principal point is in the middle of the screen (display size in pixels / 2).

Note that perspective projection assumes the viewer is located at the origin while the model was created relative to the sensor origin, so translate your model by the measured $\boldsymbol{t}$ to compensate for the shift. Draw the projected model in 2D using *fill*. If done correctly, the (u,v)-coordinates should not exceed the range from 0 to maximum resolution and the whole cube is visible on screen. Note that you should draw the sides of the cube in the correct order to avoid the familiar problem of the background occluding the foreground.

3. Changing viewpoint (mandatory)

The previous task assumed the viewer is in a static location in front of the screen. The next step is to adapt to a changing viewpoint.

Wrap your previous implementation in a function, which takes as input a 3D model (the 3xN and 3xM matrices from task 1) and viewer location (3x1 coordinate), and draws the plot of the previous task.

- The change in viewing position affects $\boldsymbol{c}$ and $\boldsymbol{t}$ (and consequently also $\boldsymbol{f}$), so change them according to the function inputs.
- $\boldsymbol{f}$ is the z-component of $\boldsymbol{t}$ (in pixels)
- Principal point $\boldsymbol{c}$ is the x and y components of $\boldsymbol{t}$ plus the middle point of the screen (display size / 2) in pixels.
- Limit the plot axis to the screen resolution using *xlim* and *ylim*.

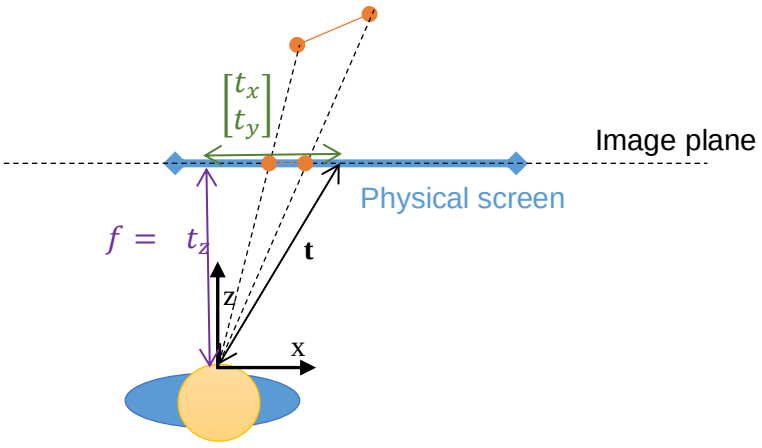
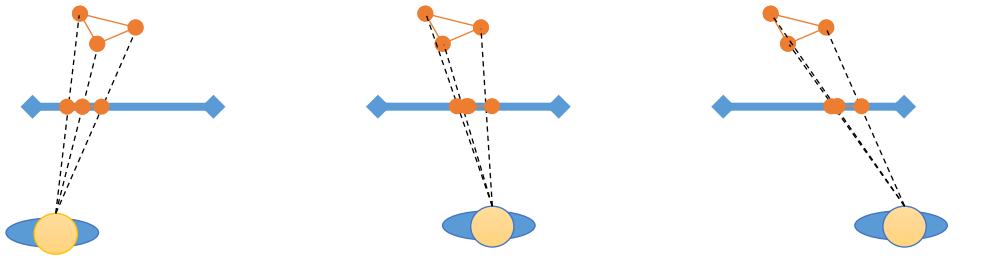


Figure 2 Relevant variables and the coordinate system of the viewer

Try different 3D viewer positions in front of the screen e.g. from side to side. Are you getting the desired effect? If not, going forward isn't going to fix anything, nor is it going to award any extra points, so review your code so far and find the mistake. If you haven't figured out what is the desired effect, going forward isn't going to fix that either, so check the demo, go back to the beginning, re-read the description and if necessary, ask.



Position 1 Position 2 Position 3…

4. Accessing a webcam (+1)

Interface with a webcam to acquire frames directly with Matlab. For this, you'll need the `winvideo` (or `linuxvideo` or `macvideo`) adapter. If it is not installed (not listed by imaqhwinfo), install it following https://se.mathworks.com/help/imaq/installing-the-support-packages-for-image-acquisition-toolbox-adaptors.html. You won't be able to do this without administrative access (e.g. on TUT student PCs).

Setting up the capture hardware goes as follows:

```
% Create the webcam object.
cam =  videoinput('winvideo',1);
% Taking a single frame when triggered
cam.FramesPerTrigger = 1;
% Allowing triggering repeatedly without the stream stopping
cam.TriggerRepeat = Inf;
% Enabling manually triggering frame capture
triggerconfig(cam, 'manual');
% Starting the stream
start(cam)
```

Capturing frames happens by triggering the stream, and retrieving the captured data

```
trigger(cam);
videoFrame = getdata(cam);
```

Figure out the focal length of your camera by taking a picture of an object of known dimensions at a known distance and the size of its projection on the camera. E.g. put a 30 cm ruler at distance of 50 cm, count how many pixels it covers in the resulting image. Compute the focal length (in pixels) using similar triangles and what you've learned of the pinhole model. Hint: since it is about the ratio between numbers, it is ok that your real world measurements are metric and the focal length and the object size in the image are in pixels. The mismatching units cancel out. Hint: *imaqreset* will recover the camera after crashes and errors

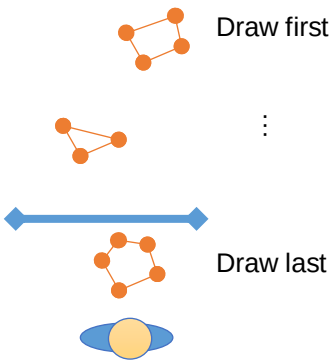
5. Face detection (+1)

Matlab has built-in functionality for detecting faces from images. Feed the head position to your implementation from previous tasks. The result will be a simple head tracking, full parallax 3D display, where you can look at the model from different sides.

1. Use the supplied `headtracking_webcam` script and add your own codes to it
2. The face detection returns a bounding box of the detected face as a four-element vector, [x y width height], that specifies in pixels the upper-left corner and size of the bounding box. Take the center of the bounding box as the viewer location and shift the coordinate space so that (0,0) is when the viewer is positioned in the middle.
3. Compute a global space 3D location for the viewer based on the previously computed f, the detected face location and the approximate viewer distance. It is done using the same kind of triangles as for computing the f, but this time the unknown is the global coordinate (in both horizontal and vertical directions).
4. Add calling your 3D projection function from task 3 to the loop after face detection using the detected global space location

6. Additional models (+1)

Duplicate the cube model and add them to the scene. Put them at different depths, one behind the box and one in front of it. The closest object should be close enough to be in front of the screen. Replace drawing with `scatter`/`line` to using 2D `patch` objects. Assign colors to the sides of the cubes. Make sure the objects are drawn on screen in the order of decreasing distance from the viewer to make objects occlude each other correctly (https://en.wikipedia.org/wiki/Painter's_algorithm).



7. Jitter stabilization (+1)

Add a filtering mechanism to the head position coordinates to reduce the effect of tracking jitter (tracked head position changing slightly even when the head is stationary) and outliers (location jumps drastically between two frames). Store a number of latest locations in a buffer and use e.g. a running average filter and/or median filtering on them before feeding the viewer location to the rendering function.