0.1 Mathematical description of a camera — cameragen

The main purpose of this function is to convert natural, computer graphics oriented, camera parameters to the parameters used in computer vision, mainly the 3×4 camera projection matrix P.

function [camera] = cameragen(pars)

Inputs:

pars	a structure containing various description of a camera
.angle	[radians] view angle of the camera, measured horizontally
.position	$[3 \times 1]$ vector containing position of the camera center expressed in the world coordinate frame
.look_at	$[3 \times 1]$ point where the camera (its optical axis) is aiming
.sky	$= [0, -1, 0]^\top$ where the sky is
.width	$= 640$ image width
.height	$= 480$ image height
.aspect_ratio	$= 1$ pix_width/pix_height
.foclen	$= 1$ focal length not really necessary, it does not change the image it changes the metric size of the image plane, consequently the pixel dimensions see below camera.width_metric
.skew	$= 0$, skew factor, if non-zero it means that rows and columns are no longer perpendicular. In fact, most of the modern cameras have no skew.

Outputs:

camera	structure containing a complete description of the camera the input structure is copied into it for completeness and beside this the computer vision parameters are computed. Note that some of the parameters are redundant and are provided for sake of completeness
.P	$[3 \times 4]$ camera projection matrix
.K	$[3 \times 3]$ upper triangular matrix containing the intrinsic camera parameters
.R	$[3 \times 3]$ rotation matrix
.t	$[3 \times 1]$ translation vector
.C	$[3 \times 1]$ position of the camera center expressed in the world coordinate frame note that P = K[Rt]
.width_metric	width of the image in metric units
.pix_width	pixel width in metric units
.pix_height	pixel height in metric units

---

orientation of the camera plane (normal vector)

```
camera.dir = -(camera.C - pars.look_at)./norm(camera.C - pars.look_at);
```

azimuth and elevation of the camera center

```
[camera.az,camera.el,camera.dist] = ...
       cart2sph(-camera.dir(1),-camera.dir(2),-camera.dir(3));
```

The composition of the camera rotation is perhaps the most complicated part of the conversion. It is important to keep in mind that the positive $z-$ axis goes from the camera center towards the `camera.look_at` position. The image plane is perpendicular to the axis and the intersection is the principal point First assume the orientation of the camera coordinate system the same as the orientation of the world system.

1. Rotate around the z-axis (vertical axis) until it fixes the `look_at` point. This is called *panning*.

   ```
   R_pan = nfi2r([0,0,1],camera.az);
   ```

2. Then rotate around the y-axis to fix the horizontal elevation. This is called *tilting*.

   ```
   R_tilt = nfi2r([0,1,0],-camera.el);
   ```

3. Rotation of the camera system itself. Remember that the z-axis of the camera coordinate system directs towards the scene.

   ```
   R_cam = nfi2r([1,0,0],-pi/2)*nfi2r([0,0,1],pi/2);
   ```

4. By default the camera is horizontally aligned with the xy plane of the world. In other words the world horizon projects as perfectly horizontal line in the image. In camera coordinate system it correponds to the (normalized) position of the sky at $[0, -1, 0]$. This default setting can be overriden by setting different `pars.sky`.

   ```
   if all(camera.sky==[0,-1,0]')
     R_sky = eye(3);
   else
     if all(-camera.sky==[0,-1,0]');
          R_sky = nfi2r([0,0,1],pi);
     else
          rotaxis = cross(camera.sky,[0,-1,0]');
          rotangle = acos(camera.sky'*[0,-1,0]' );
          R_sky = nfi2r(rotaxis,rotangle);
     end
   end
   camera.R = R_sky*R_cam*R_tilt*R_pan;
   ```

The rest of the computation is rather straightforward

```
camera.t = -camera.R*camera.C;
camera.width_metric = 2*camera.foclen*tan(camera.angle/2);
camera.pix_width = camera.width_metric/camera.width;
camera.pix_height = (1/camera.aspect_ratio) * camera.pix_width;
% composition of the K matrix
camera.K = [camera.foclen/camera.pix_width, camera.skew, camera.width/2; ...
              0, camera.foclen/camera.pix_height, camera.height/2; ...
              0,0,1];
% and finally the 3x4 camera projection matrix
camera.P = camera.K *[camera.R, camera.t];
```

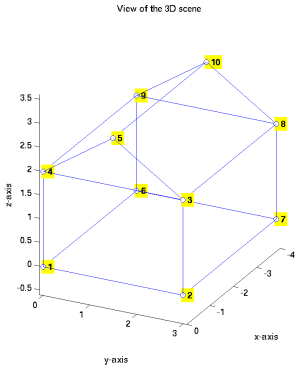

Figure 1: A generated 3D scene

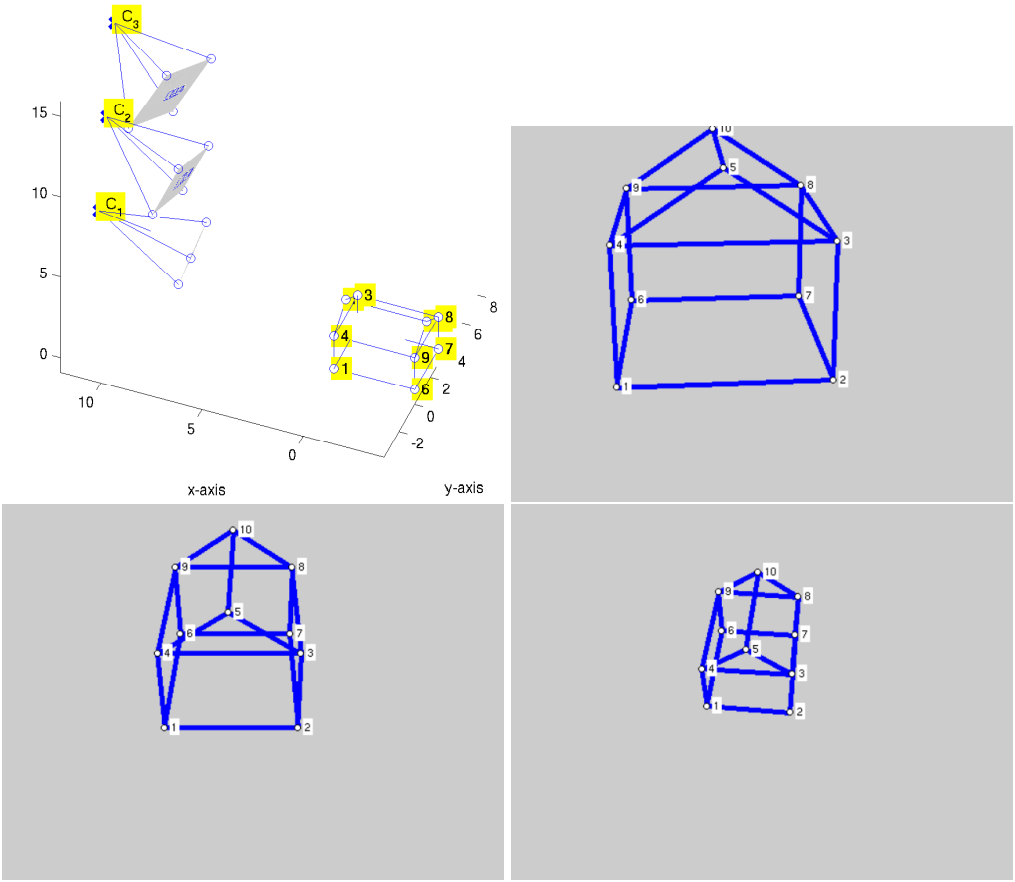

Figure 2: 3D scene with cameras. The camera rises up and slightly rotates whilst keeping the `look_at` fixed. You can observe how is the house becoming smaller with the increasing hight of the camera position. For inserting cameras into a 3D plot, see `showcams` Section 0.1.1

0.1.1 Visualise a camera in a 3d plot — showcams

The main purpose is to draw a camera into a 3D sketch in a pleasant way. Camera center is plot, as well as image plane with the actual image, corner points of the image plane and lines that connect the corner points with the camera center The 3D sketch may then require a manual adjustment of the viewpoint to get the most satisfactory results. The rotation may be enabled by `rotate3d` or interactively, by clicking to the 3D rotate icon on the figure window

function `fig = showcams(fh,foclen,P1,im1,P2,im2, ...)`

function `fig = showcams(fh,foclen,P,im)`

Inputs: Variable number of input parameters

`fh`	figure handle of the 3D plot where the camera is to be drawn
`foclen`	a focal lenght. However, it needs to be understood as not the true one. It rather just control the visual appearance of the camera(s). The true metric focal lenght cannot be estimated from the `P` matrix only unless the metric size of the pixels is also provided. If a negative value is specified, `foclen=5` is set.
`P1`	$[3 \times 4]$ camera projection matrix
`im1`	image of the `P1` camera, it can be both grayscale and RGB
`P2, im2 ...`	more cameras, more images

The cameras and images may be also provided as cell arrays. `P={P1,P2,P3,...}` and `im={im1,im2,im3,...}`.

See also: `cameragen, P2KRtC`

For all the specified cameras and images do the following:

- Decompose the `P` matrix into intrinsic and extrinsic parameters

  ```
  [K,R,t,C] = P2KRtC(P);
  ```

- Back project the corner pixels into the scene

  ```
  [r c d] = size(img);
  U = [1,1; 1,r; c,r; c,1]';
  U(3,:) = 1;
  X = pinv(P)*U; % back projection
  X = X./repmat(X(end,:),4,1); X = X(1:3,:); % normalization
  ```

  direction vectors from the camera center to the backprojected image corners.

  ```
  dirvec = (X - repmat(C,1,length(X)));
  dirvec = dirvec./repmat(sqrt(sum(dirvec.^2)),3,1);
  ```

- compute the coordinate of the image plane corners in the world coordinate system

  ```
  X = dirvec*foclen + repmat(C,1,length(X));
  ```

- prepare the image for warping. Create the image plane from the backprojected corner points. And finally, do the warp by employing the Matlab `surface` command.

```
if d==3
  if i>1 % re-use the colormap
        [imind,cmap] = rgb2ind(uint8(img),cmap);
  else
        [imind,cmap] = rgb2ind(uint8(img),256);
  end
else
  imind = uint8(img);
  cmap{i} = colormap(gray(256));
end
for j=1:3,
  implane(:,:,j) = [X(j,[1,4]);X(j,[2,3])];
end
surface(implane(:,:,1),implane(:,:,2),implane(:,:,3),imind, ...
               'FaceColor','texturemap','EdgeColor','none', ...
               'CDataMapping','scaled')
```

- At the end just plot the lines conecting the camera center with the corners of the image plane and attach a label to the camera center.

```
for j=1:4,
  line([C(1),X(1,j)],[C(2),X(2,j)],[C(3),X(3,j)])
end
plot3(C(1),C(2),C(3),'x','MarkerSize', 10, 'LineWidth',3);
text(C(1),C(2),C(3), sprintf('␣␣C_%d',i), 'BackgroundColor','yellow');
```

0.1.2 Conversion of rotation parameters — nfi2r

For a human is much more natural to define a 3D rotation in terms of axis rotation and amount of rotation around it. Typical example may be: "rotate a camera around its vertical axis for 90 degrees". From the computational point of view is more convenient to represent the rotation as 3×3 matrix R. Then $\mathbf{x}_{rotated} = \mathtt{R}\mathbf{x}$.

function `R = nfi2r(n,fi)`

Inputs:

`n`	$[3 \times 1]$ axis of rotation (vector of direction)
`fi`	[rad] angle of rotation (counter clockwise)

Output: R $[3 \times 3]$ rotation matrix

See also: The algorithm can be found in [**?**]

0.1.3 RQ matrix decomposition

function `[R,Q] = rq(X)`

Input: `X` input matrix
Outputs:
`Q` unitary matrix
`R` upper triangular matrix
See also: `qr`.

0.2 Homography estimation from point correspondences — DLT method `u2Hdlt`

function `[H,T1,T2] = u2Hdlt(u1,u2,do_norm=1)`

Linear estimation of homography from point correspondences with (optional) point normalization

Input parameters:

`u1,u2`	$[2\|3 \times N]$ corresponding coordinates
`do_norm`	do isotropic normalization of points? If not specified, $= 1$ is assumed. $= 0$ do not normalization It may be useful when points are already normalized it speeds up.

Output parameters:

`H`	$[3 \times 3]$ homography matrix
`T1,T2`	$[3 \times 3]$ transformation matrices that did the proper normalization

See also: pointnorm, u2Fdlt

Compose the data matrix from point correspondences

```
u1 = u1';
u2 = u2';
A = [];
for i=1:size(u1,1),
  A = [A; 0 0 0 -u2(i,3)*u1(i,:) u2(i,2)*u1(i,:); ...
          u2(i,3)*u1(i,:) 0 0 0 -u2(i,1)*u1(i,:)];
end
```

Compute the solution by using SVD

```
[U,S,V] = svd(A);
H = reshape(V(:,end),3,3)';
```

Undo the point normalization if applied

```
if do_norm
  H = inv(T2)*H*T1;
end
```

0.2.1 Example of homography mapping

Generate an artifical scene. Two possible types: `house` is a 3D house and `random2D` makes randomly generated co-planar points

```
scenetype = 'house'; % 'house' 'random2D';
[X,L] = scenegen(scenetype,10);
X(4,:) = 1;
```

Define two virtual cameras and project the 3D points to them. Compute the homography mapping from selected points

Figure 3: Two projections of a planar scene. The corrsponding points used for homography computation are red encircled. Because of the planarity, all points and their homography projected counterparts exactly match. Compare to the non-planar scene scenarion, see Fig **??**.

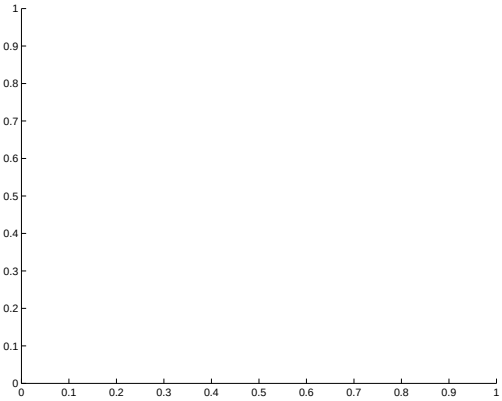

Figure 4: Two projections of the house scene. The corresponding points used for homography computation are red encircled. The selected points lie on a 3D plane. However, points that do not lie on this 3D plane where the corresponding points were taken are mapped incorrectly. This phenomena is called *out of plane parallax*. Note also, that the top most point on the frontal gable lies on the same plane as the corresponding points and this point *is* mapped correctly.

```
idxcorr = [1,2,3,4];
H = u2Hdlt(cam(1).u(:,idxcorr),cam(2).u(:,idxcorr));
```

Mapping from camera No. 1 to the camera No. 2, $\mathbf{u}_2 = \mathtt{H}\mathbf{u}_1$

```
u{2} = H*cam(1).u;
```

For a non-degenerate configuration the mapping is one to one, it means $\mathbf{u}_1 = \mathtt{H}^{-1}\mathbf{u}_2$.

```
u{1} = inv(H)*cam(2).u;
```

Normalize both the computed coordinates to get the pixel coordinates. Then, all the points generated and computed are displayed by using the `display2Dpoints` function, see Figures **??** and **??**.

```
for i=1:2,
  u{i} = u{i}./repmat(u{i}(3,:),3,1);
end;
```

0.2.2 Isotropic point normalization — pointnorm

Normalization of point coordinates in order to achieve a better numerical stability of direct linear tranform (DLT) estimation. See [**?**] for more details

function `[u2,T] = pointnorm(u)`

Inputs: `u` $[3 \times N]$ matrix of the unnormalized coordinates of N points
Outputs:
`u2` $[3 \times N]$ normalized coordinates
`T` $[3 \times 3]$ transformation matrix, `u2 = Tu`
See also: `u2Fdlt`, `u2Hdlt`

---

centering the coordinates

```
u2 = u;
u2(1:2,:) = u(1:2,:) - repmat(centroid,1,n);
```

scale them to have average radius of $\sqrt{2}$

```
scale = sqrt(2)/mean(sqrt(sum(u2(1:2,:).^2)));
u2(1:2,:) = scale*u2(1:2,:);
```

composition of the normalization matrix

```
T = diag([scale,scale,1]);
T(1:2,3) = -scale*centroid;
```

Example of pointnorm usage

The function `pointnorm` is used as follows:
Generate and display artificial 2D data

```
u = 100*rand(2,100);
u(3,:) = 1; % make the data homegeneous

figure(1)
plot(u(1,:),u(2,:),'+')
hold on
title('original␣points')
```

Normalize points such that centroid of $\mathbf{u}_2$ will be $[0,0]^\top$. $\mathbf{u}_2 = \mathtt{T}\mathbf{u}_1$

```
[u2,T] = pointnorm(u);
figure(2)
plot(u2(1,:),u2(2,:),'k+','MarkerSize',10)
title('normalized␣points');
```

Control computation

```
u3 = inv(T)*u2;
figure(1)
plot(u(1,:),u(2,:),'ko','MarkerSize',10)
```

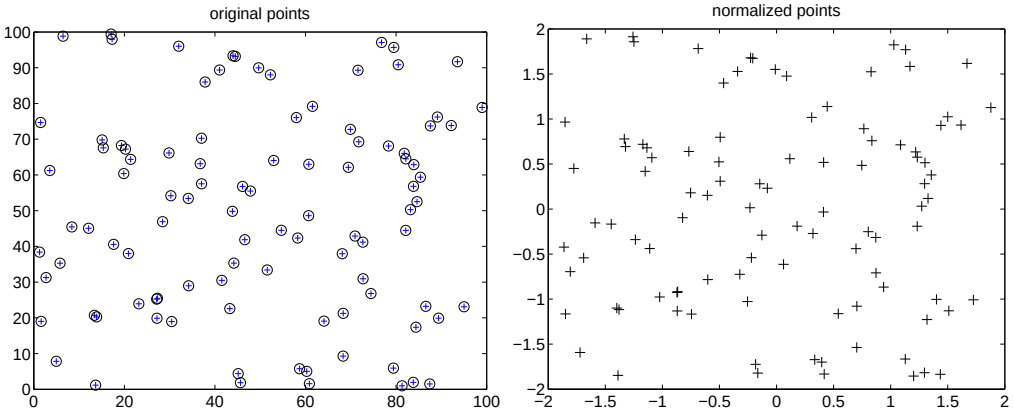

Figure 5: Left: original points (crosses) and re-computed points circles. Right: normalized coordinates.

0.3 3D point recontruction — linear method — uP2Xdlt

function `X = uP2Xdlt(P1,u1,P2,u2, ...)`

function `X = uP2Xdlt(P,u)`

Inputs:

`P1,P2,...,PN`	$[3 \times 4]$ N camera projection matrices
`u1,u2,...,uN`	$[3 \times n]$ homegeneous coordinates of n corresponding points

Outputs:

`X`	$[4 \times n]$ homogeneous coordinates of reconstructed 3D points

---