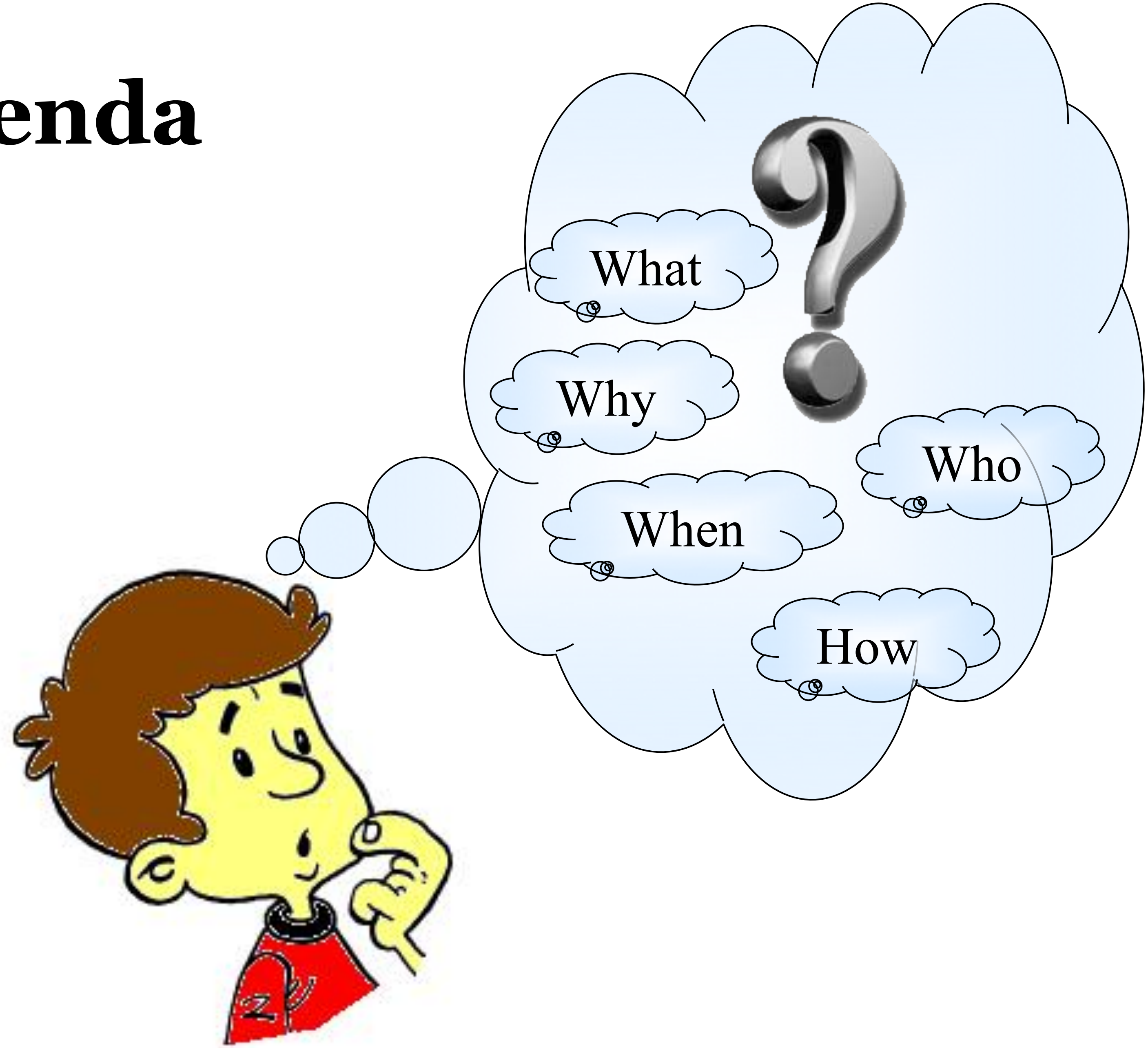
Hands-on
Camera Calibration

By
Shireen Y. Elhabian

February 2008

Agenda



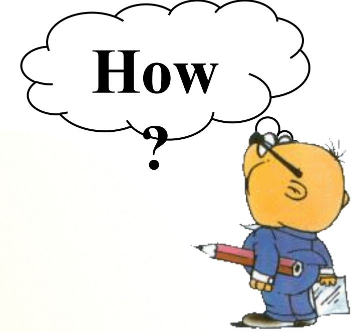
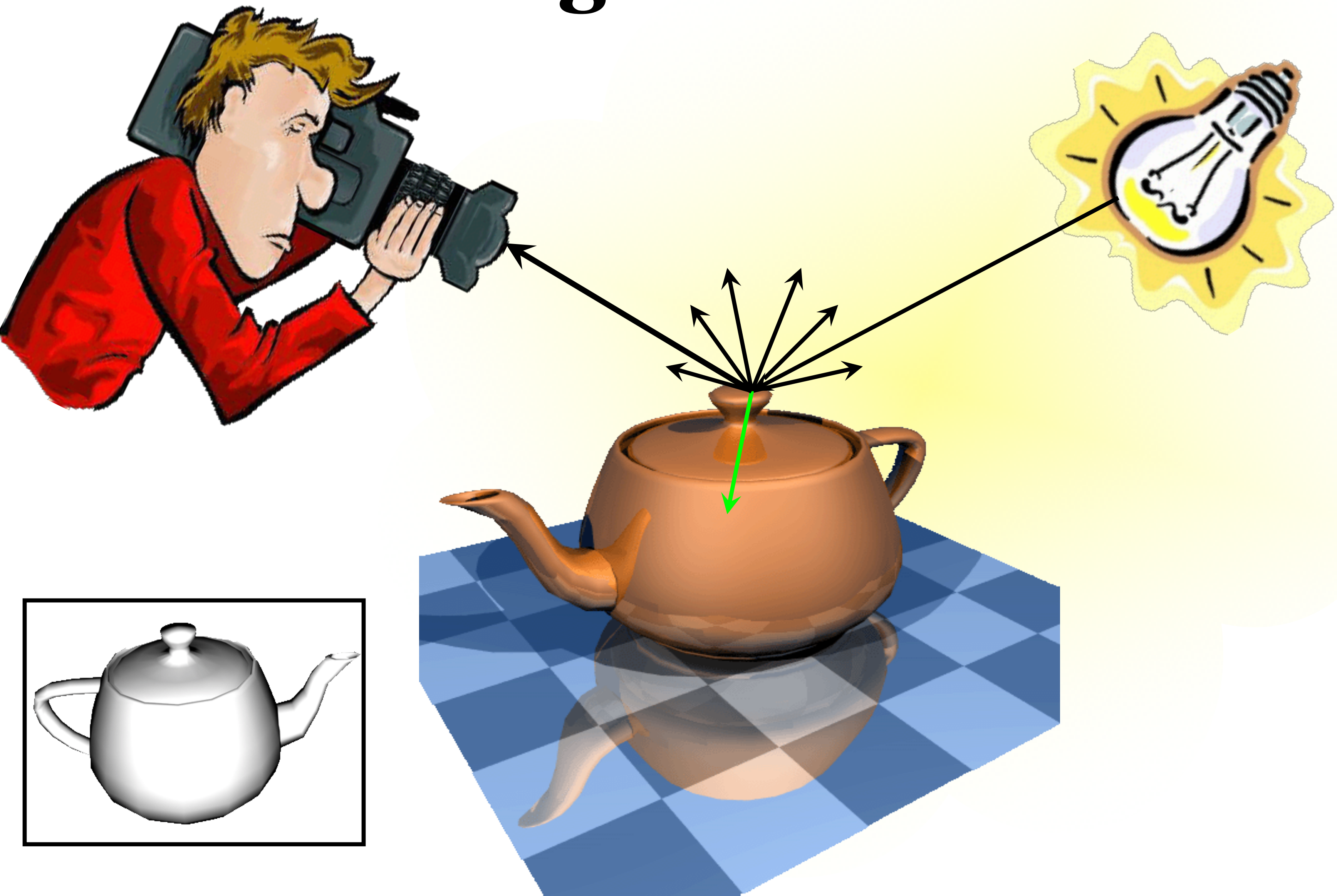
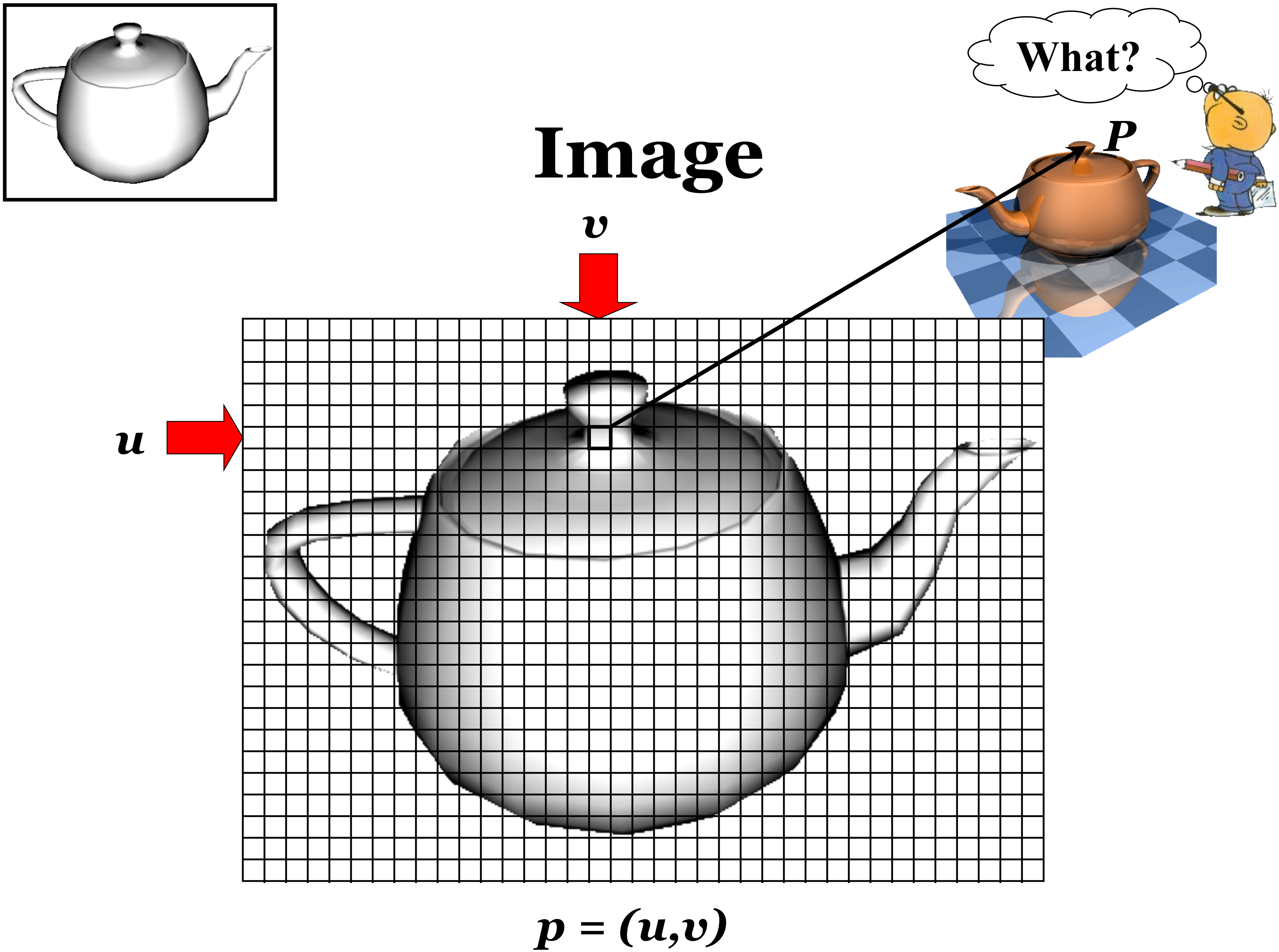
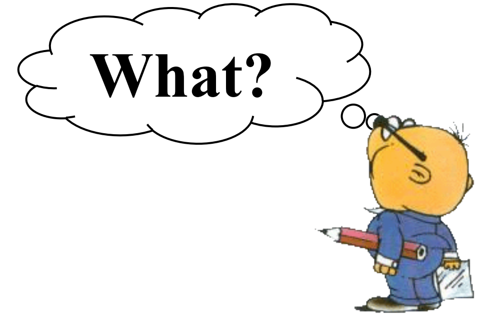


Image Formation





What?
P
Image
v
u
p = (u,v)

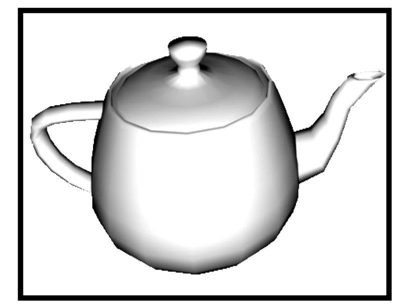


Camera

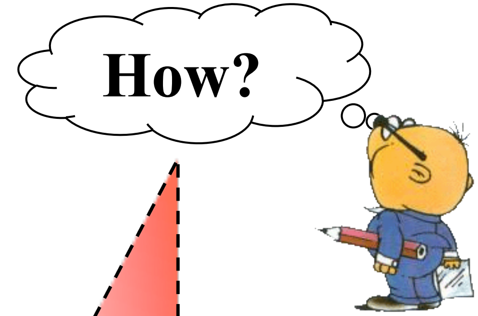
$P = (X_w, Y_w, Z_w)$

Meters, inches …

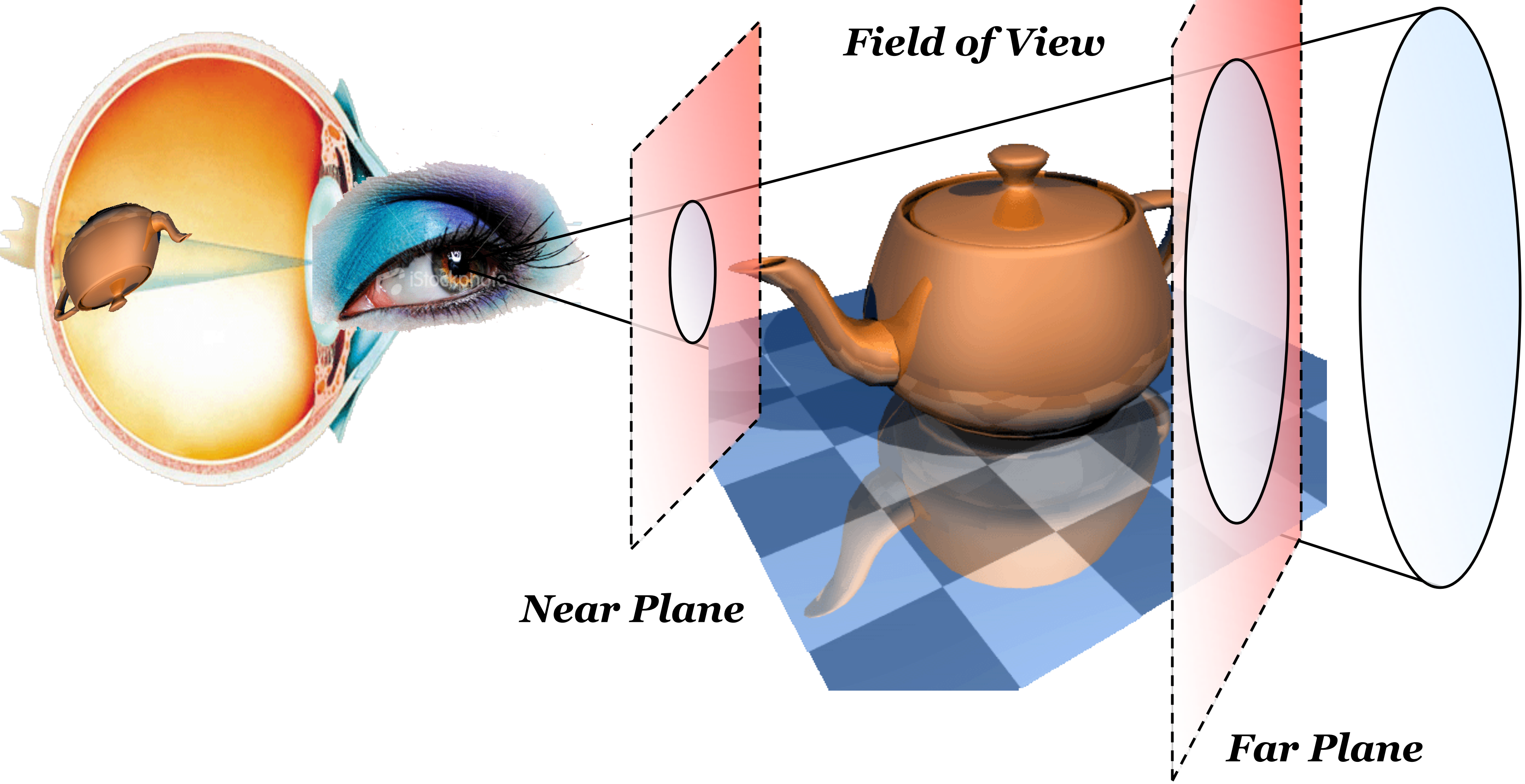
$p = (u,v)$



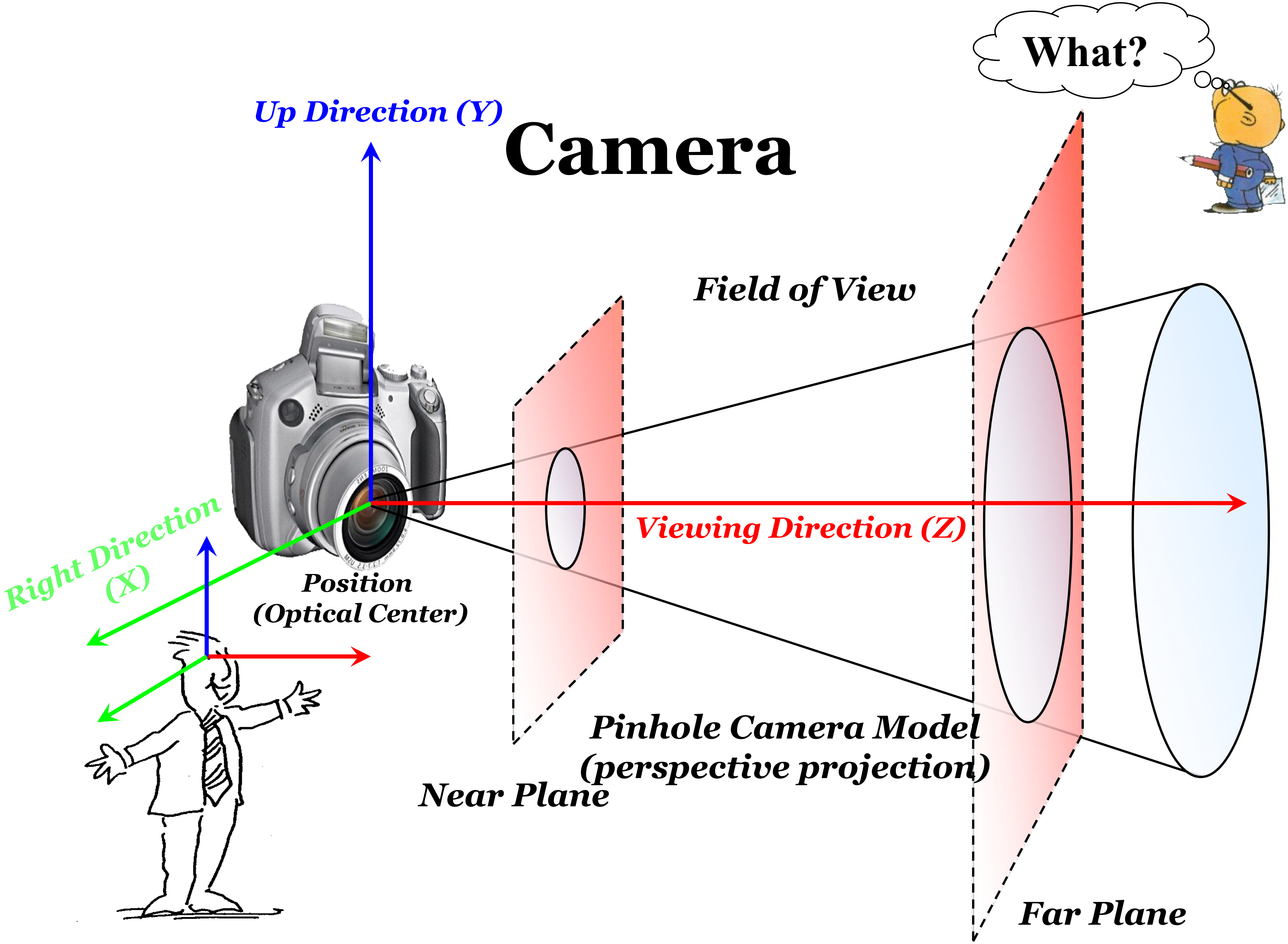
Pixels



Human See

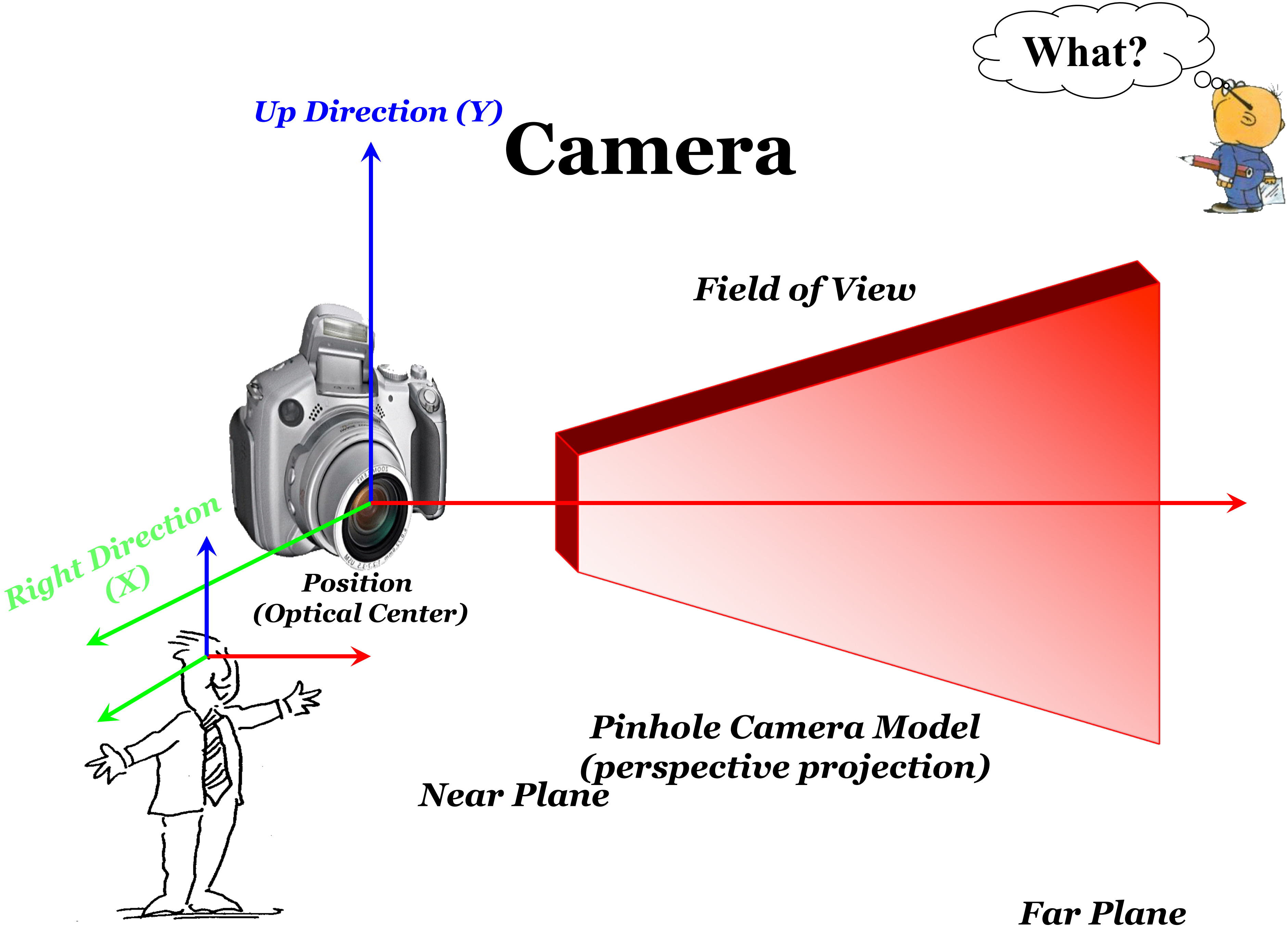


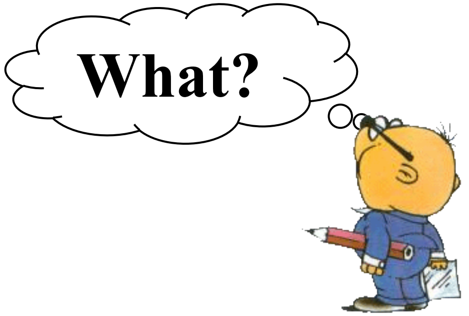
Camera



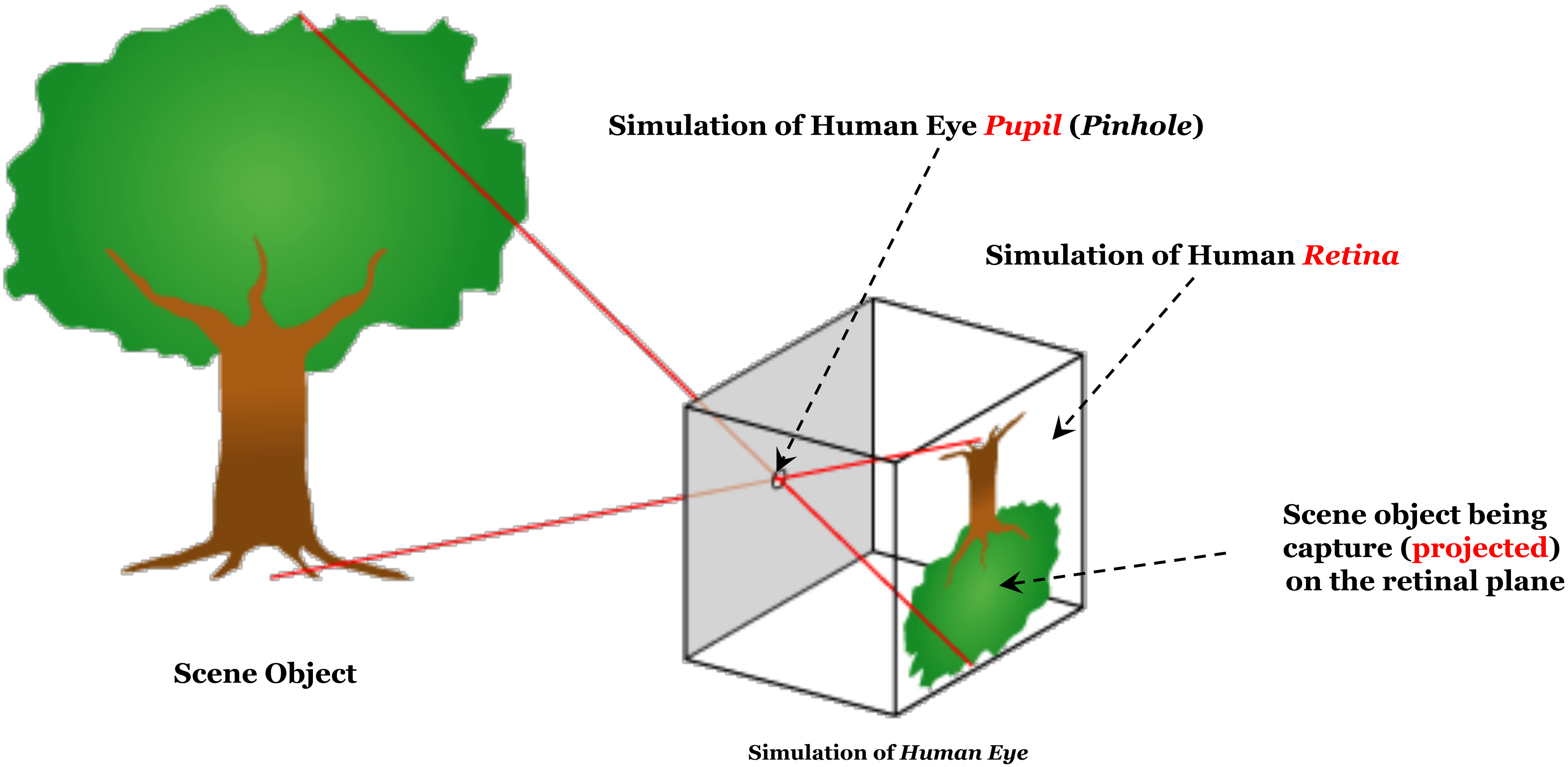
What?
Up Direction (Y)
Field of View
Viewing Direction (Z)
Right Direction (X)
Position
(Optical Center)
Pinhole Camera Model
(perspective projection)
Near Plane
Far Plane

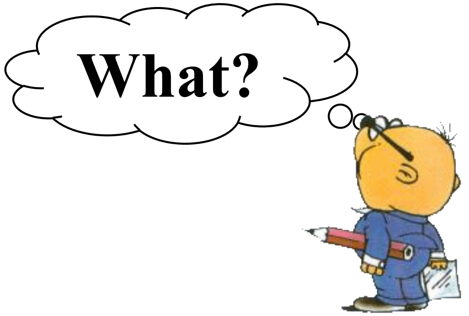
Camera


Up Direction (Y)
Field of View
Right Direction (X)
Position (Optical Center)
Pinhole Camera Model (perspective projection)
Near Plane
Far Plane
What?



Pinhole Camera





Camera

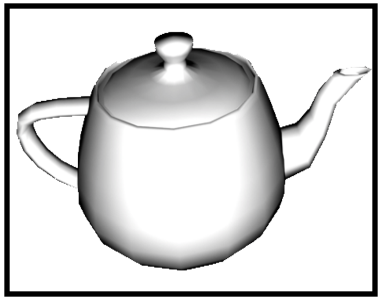
3D world

$P = (X_w, Y_w, Z_w)$

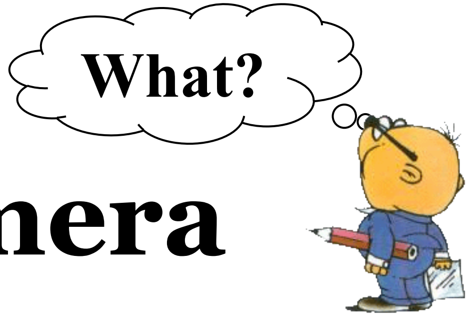
Meters, inches …

Image plane

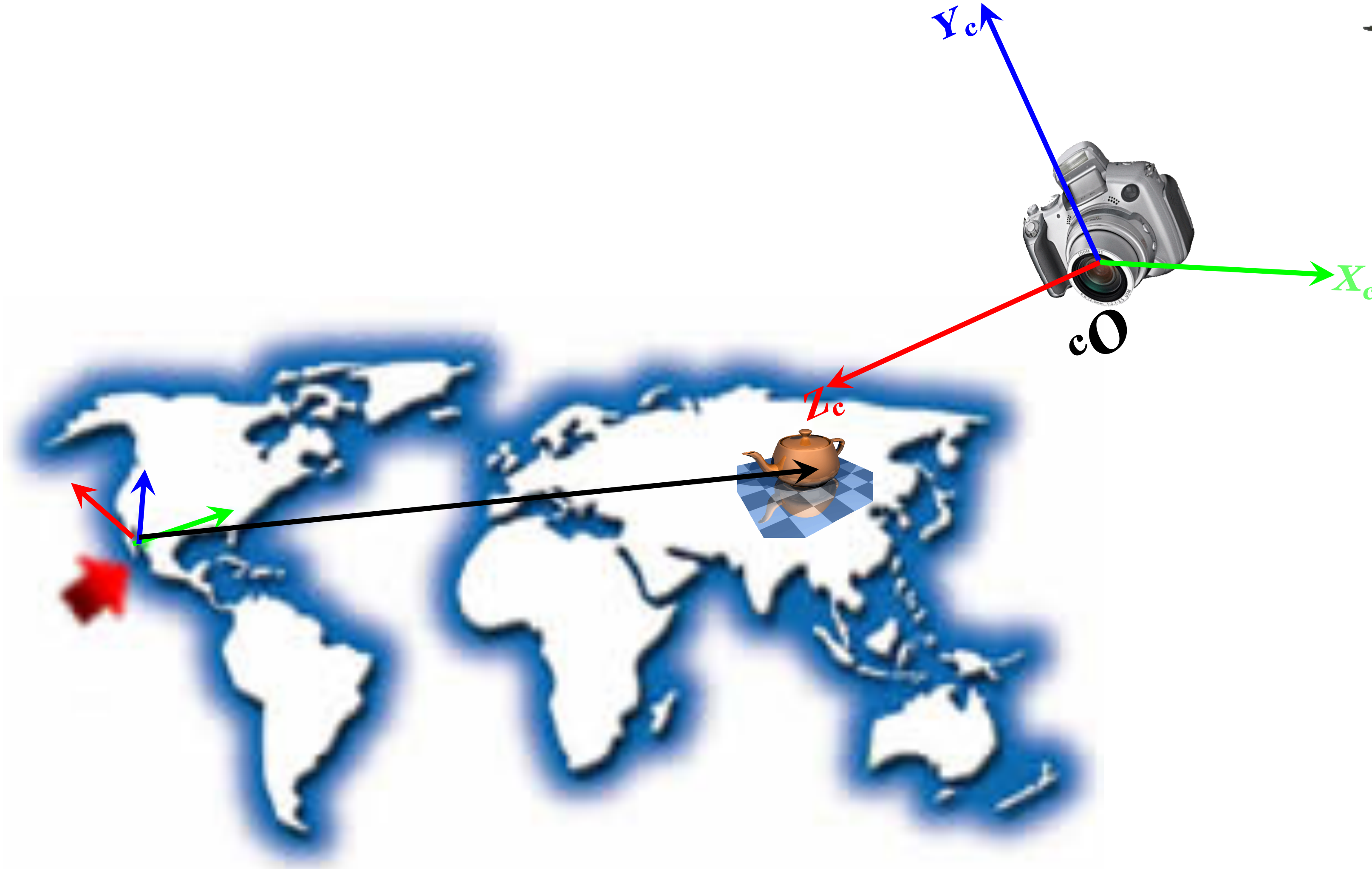
$p = (u, v)$

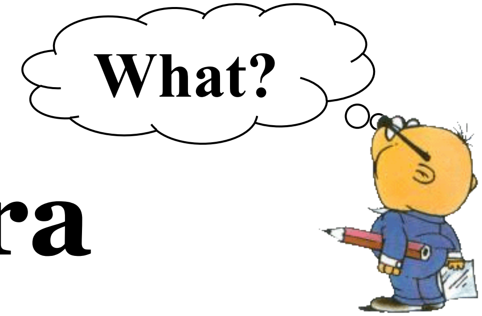


Pixels



The World to the Camera

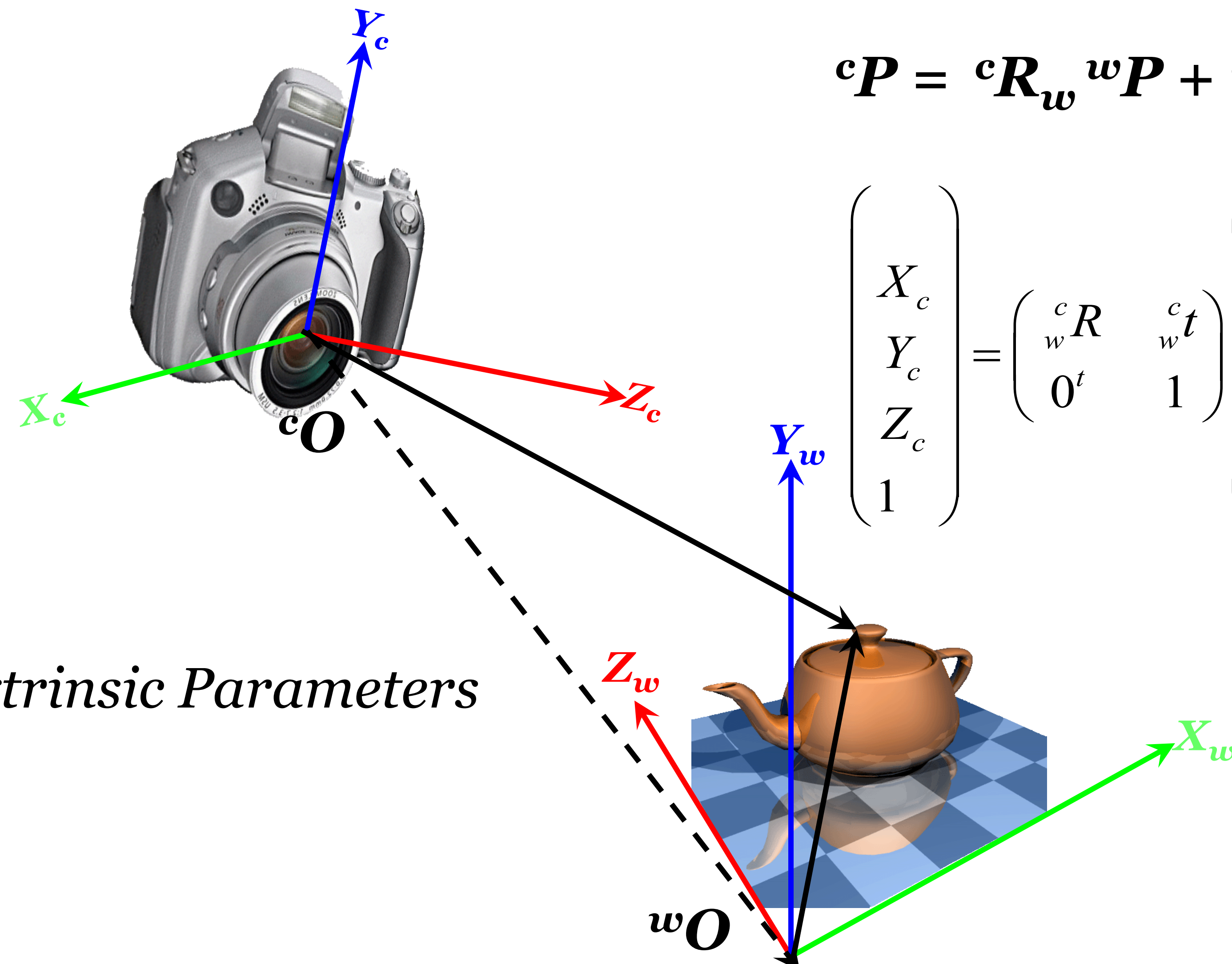




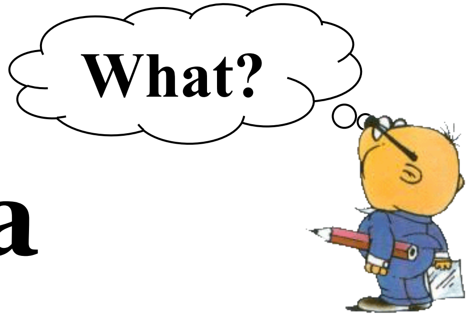
Outside the Camera

$${}^{c}P = {}^{c}R_{w}\,{}^{w}P + {}^{c}O_{w}$$

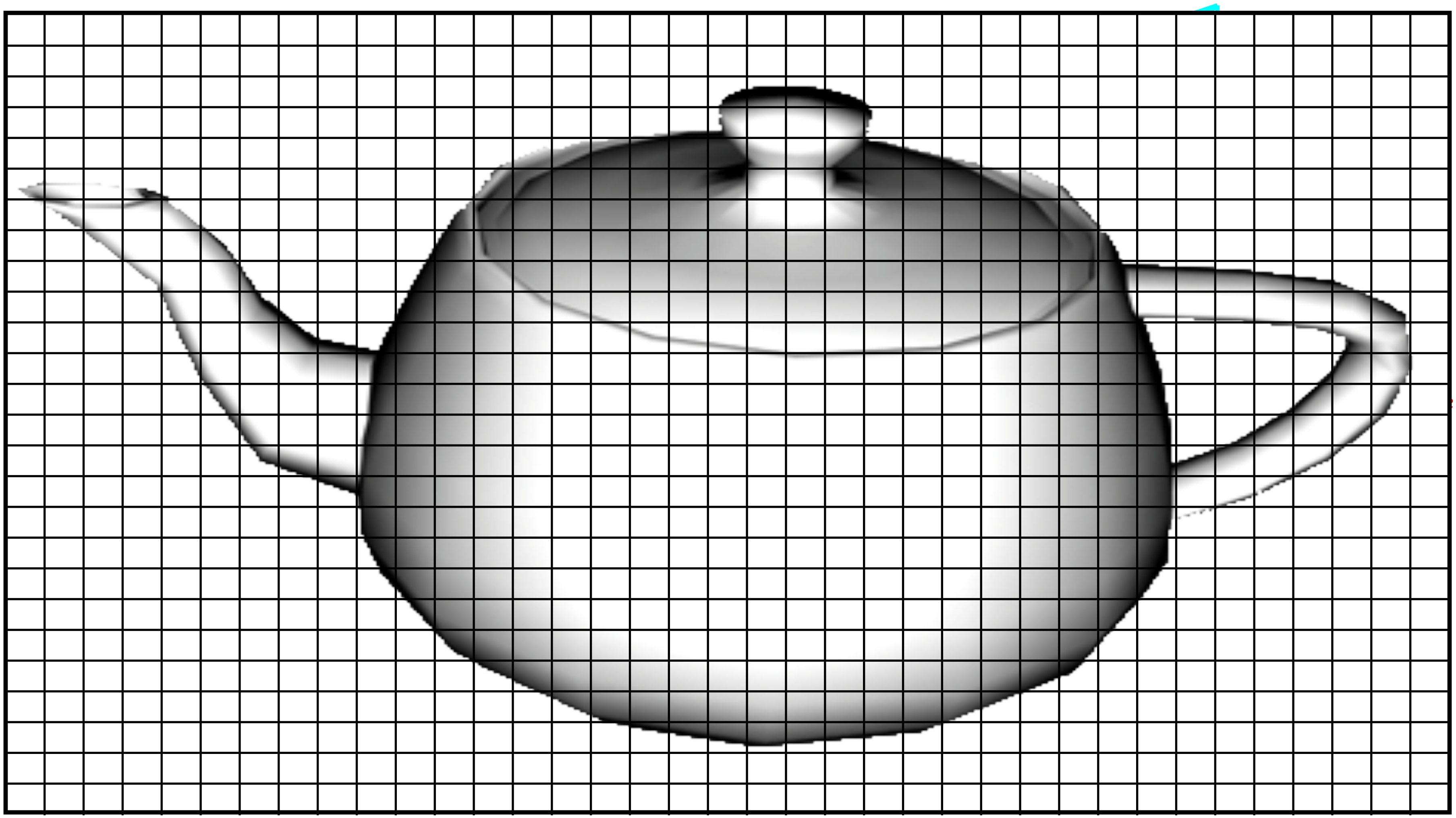
$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} = \begin{pmatrix} {}^{c}_{w}R & {}^{c}_{w}t \\ 0^t & 1 \end{pmatrix} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix}$$

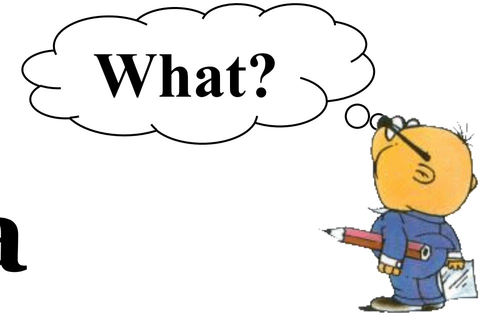


Extrinsic Parameters

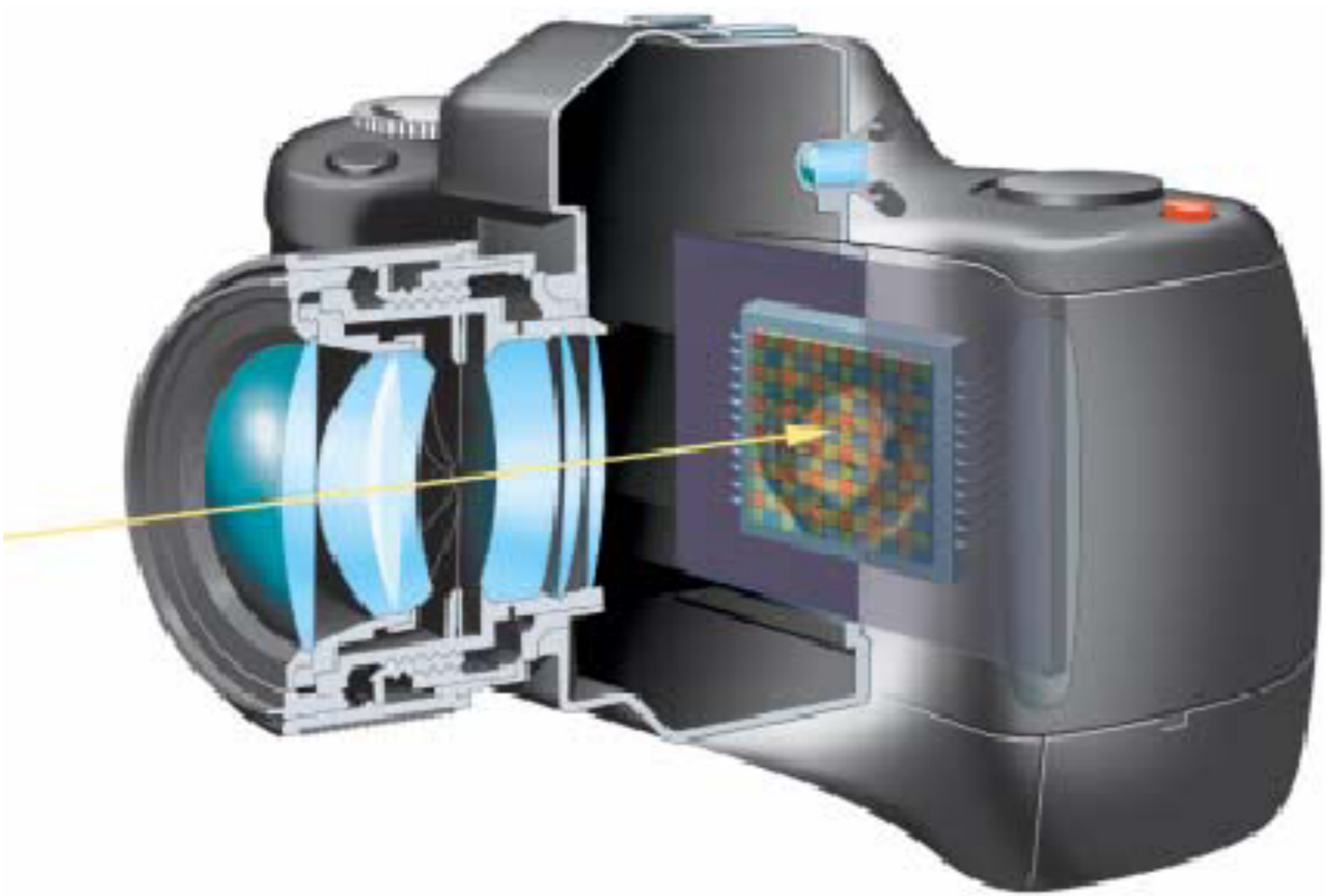


Inside the Camera

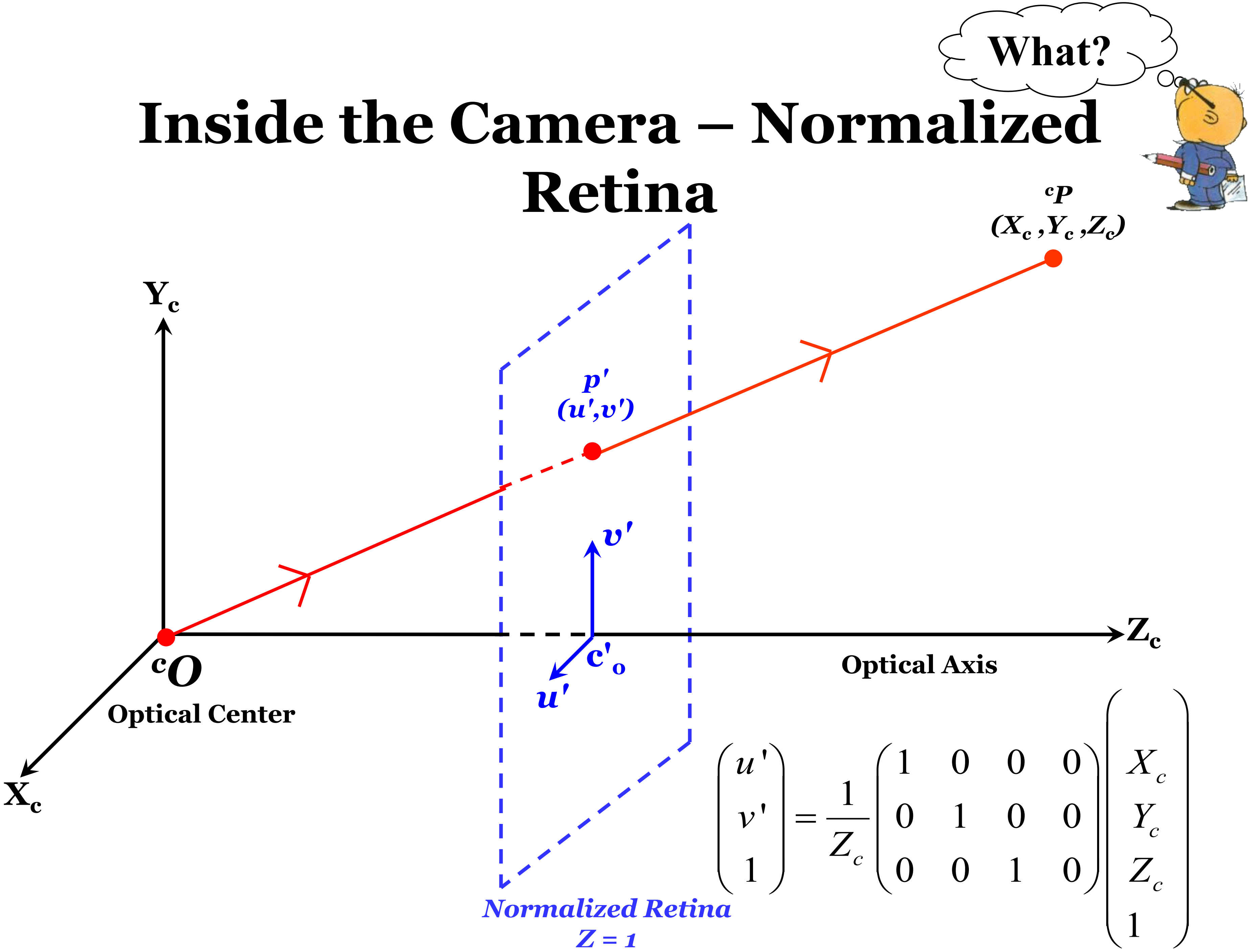




Inside the Camera



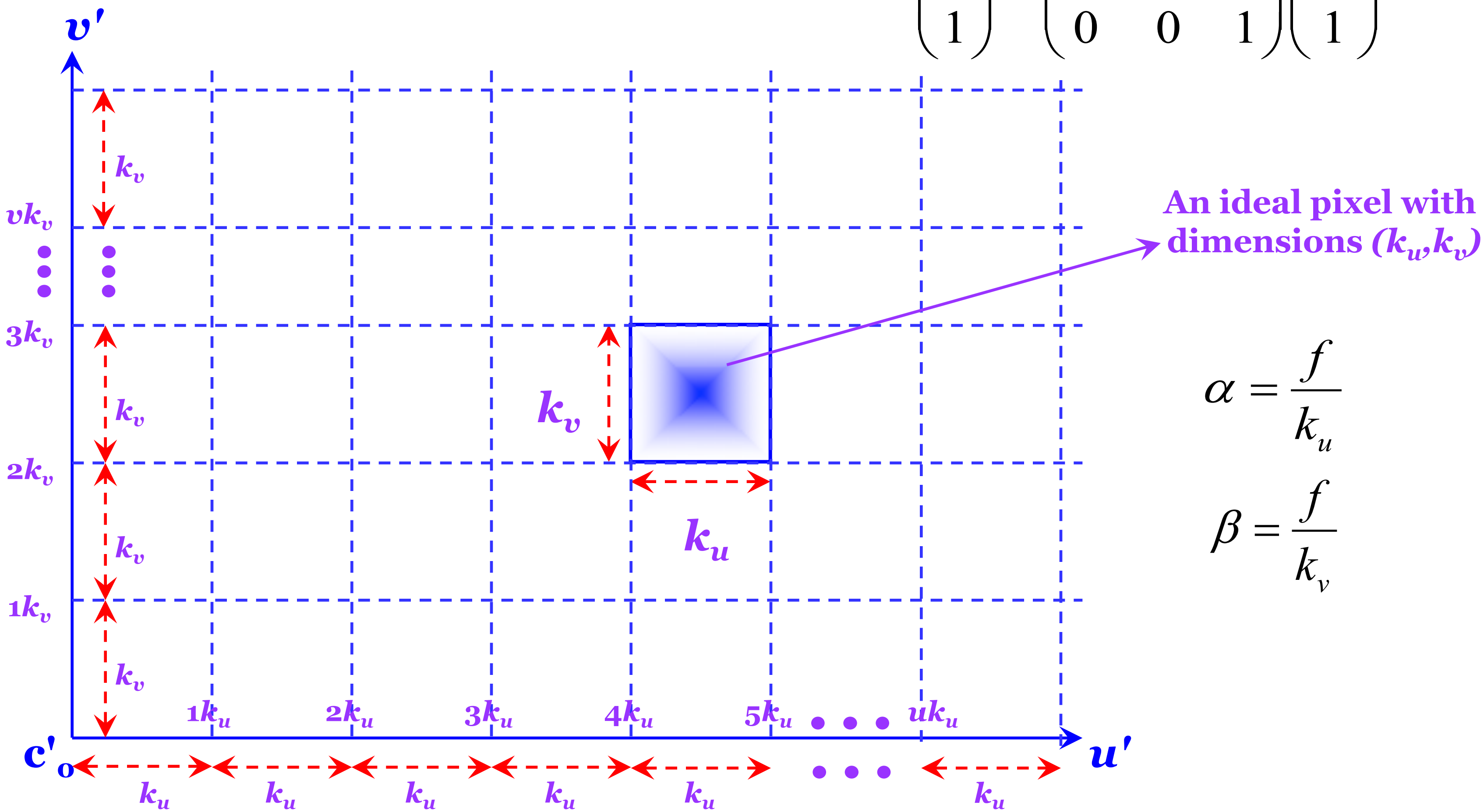
Inside the Camera – Normalized Retina



$$\begin{pmatrix} u' \\ v' \\ 1 \end{pmatrix} = \frac{1}{Z_c} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix}$$

Inside the Camera – Image Plane

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u' \\ v' \\ 1 \end{pmatrix}$$

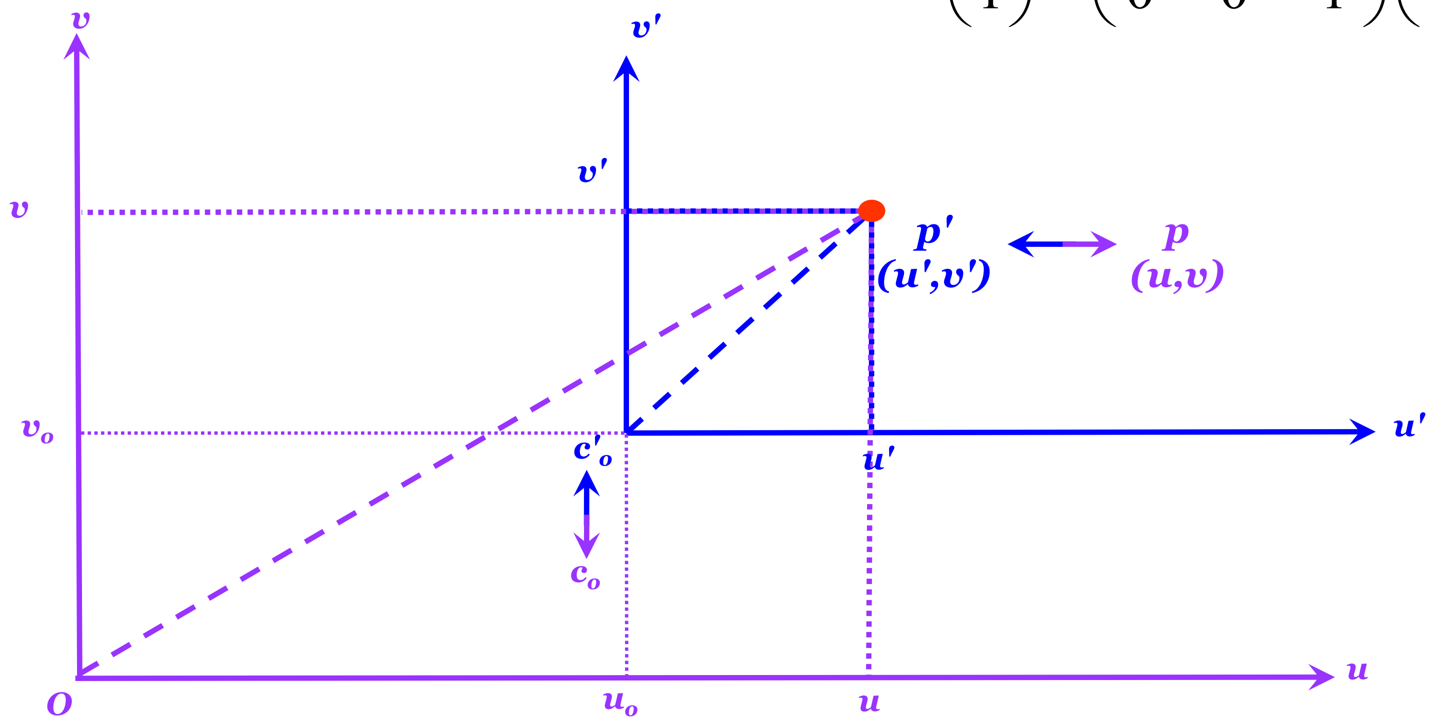


$$\alpha = \frac{f}{k_u}$$

$$\beta = \frac{f}{k_v}$$

Inside the Camera – Image Plane

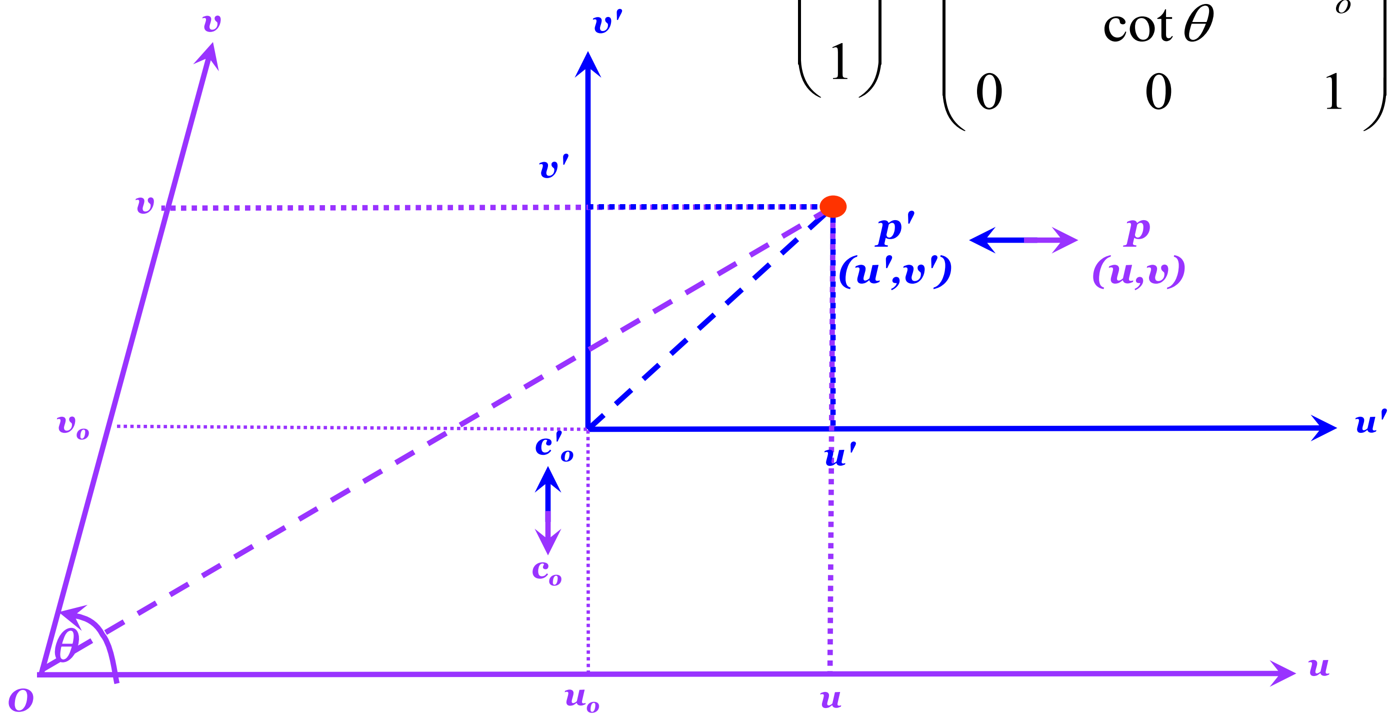
$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha & 0 & u_o \\ 0 & \beta & v_o \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u' \\ v' \\ 1 \end{pmatrix}$$

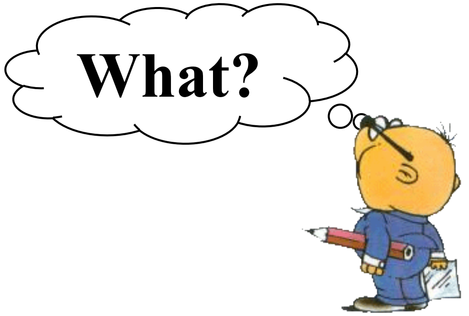


Inside the Camera – Image Plane

Intrinsic Parameters

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \overbrace{\begin{pmatrix} \alpha & -\alpha \cot\theta & u_o \\ 0 & \dfrac{\beta}{\cot\theta} & v_o \\ 0 & 0 & 1 \end{pmatrix}}^{K} \begin{pmatrix} u' \\ v' \\ 1 \end{pmatrix}$$





Camera Model

World Frame

^{w}P

Camera Frame

$$\begin{pmatrix} {}^{c}P \\ 1 \end{pmatrix} = \begin{pmatrix} {}^{c}_{w}R & {}^{c}_{w}t \\ 0^{t} & 1 \end{pmatrix} \begin{pmatrix} {}^{w}P \\ 1 \end{pmatrix}$$

Extrinsic

Normalized Retina

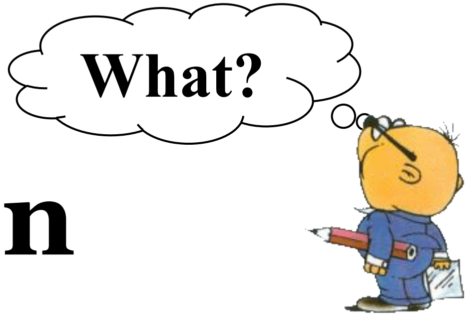
$$\begin{pmatrix} p' \\ 1 \end{pmatrix} = \frac{1}{Z_c} \begin{bmatrix} I & 0 \end{bmatrix} \begin{pmatrix} {}^{c}P \\ 1 \end{pmatrix}$$

Intrinsic

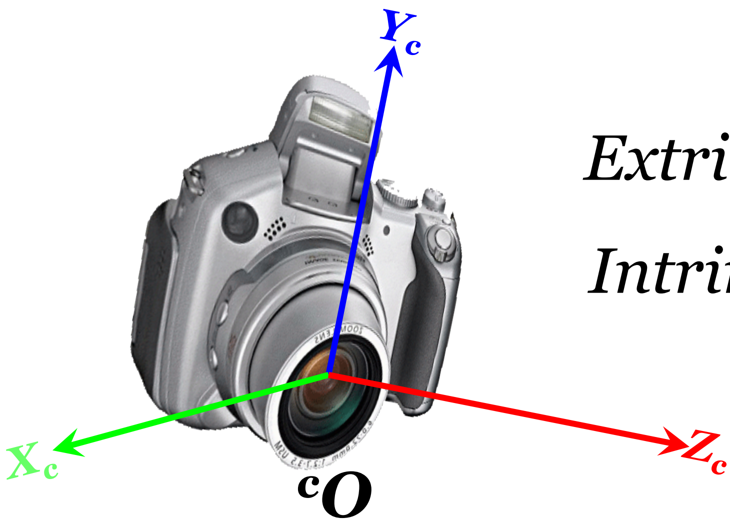
Image Plane

$$p = Kp'$$

$$p = \frac{1}{Z_c} K \begin{bmatrix} R & t \end{bmatrix} {}^{w}P = \frac{1}{Z_c} M \, {}^{w}P$$

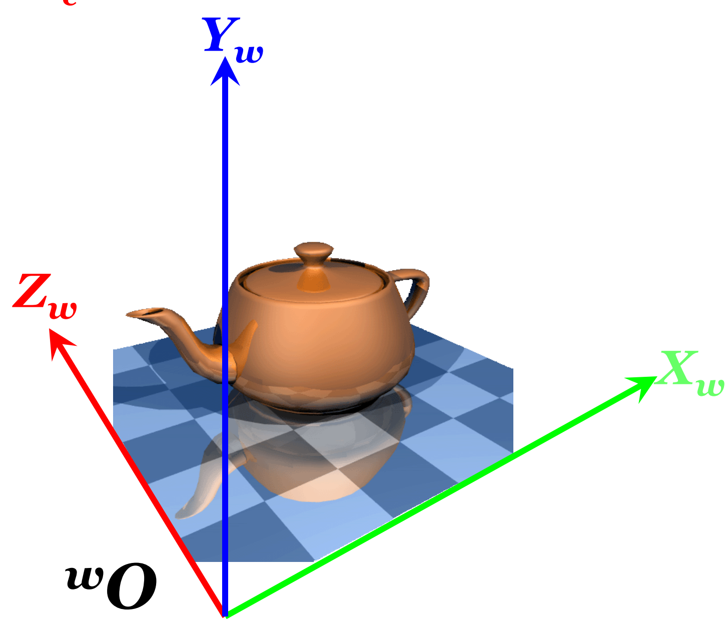


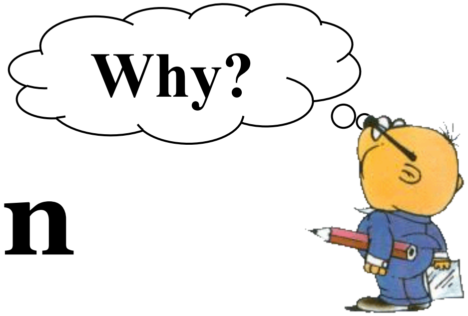
Camera Calibration



Extrinsic Parameters $(\theta_x,\theta_y,\theta_z,t_x,t_y,t_z)$

Intrinsic Parameters $(\alpha,\beta,\theta,u_o,v_o)$



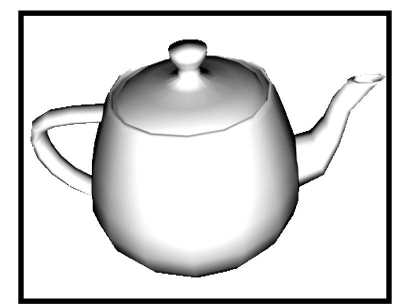


Camera Calibration

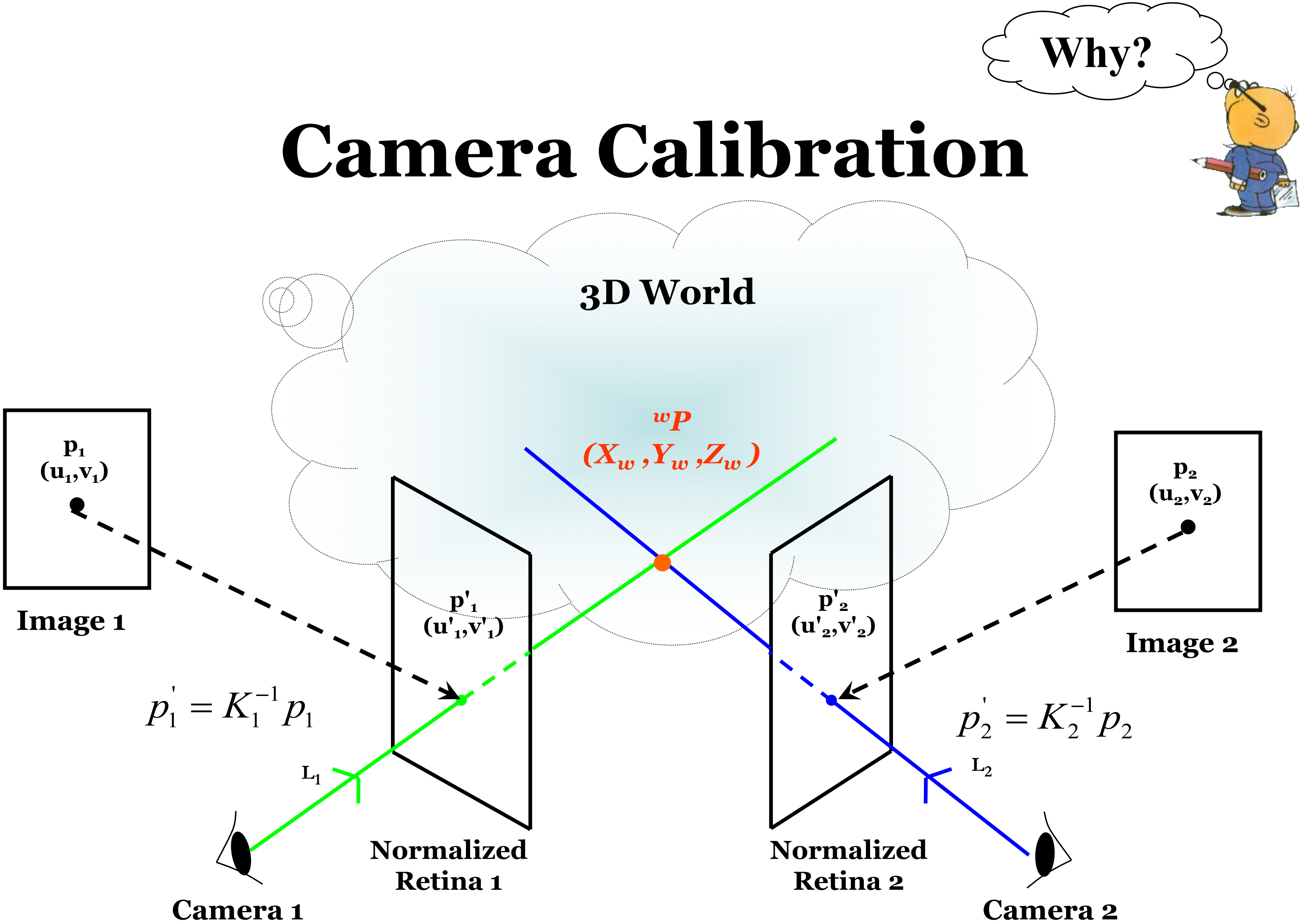
$^{w}P = (X_w, Y_w, Z_w)$

$p = (u, v)$

$$p = \frac{1}{Z_c} M\, {}^{w}P$$



Camera Calibration

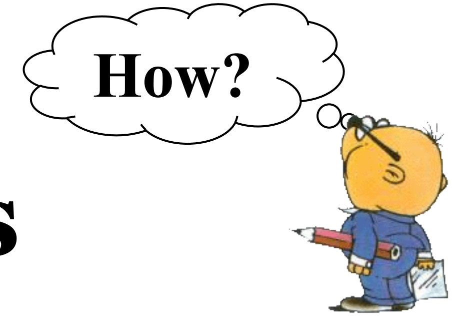


What is needed?

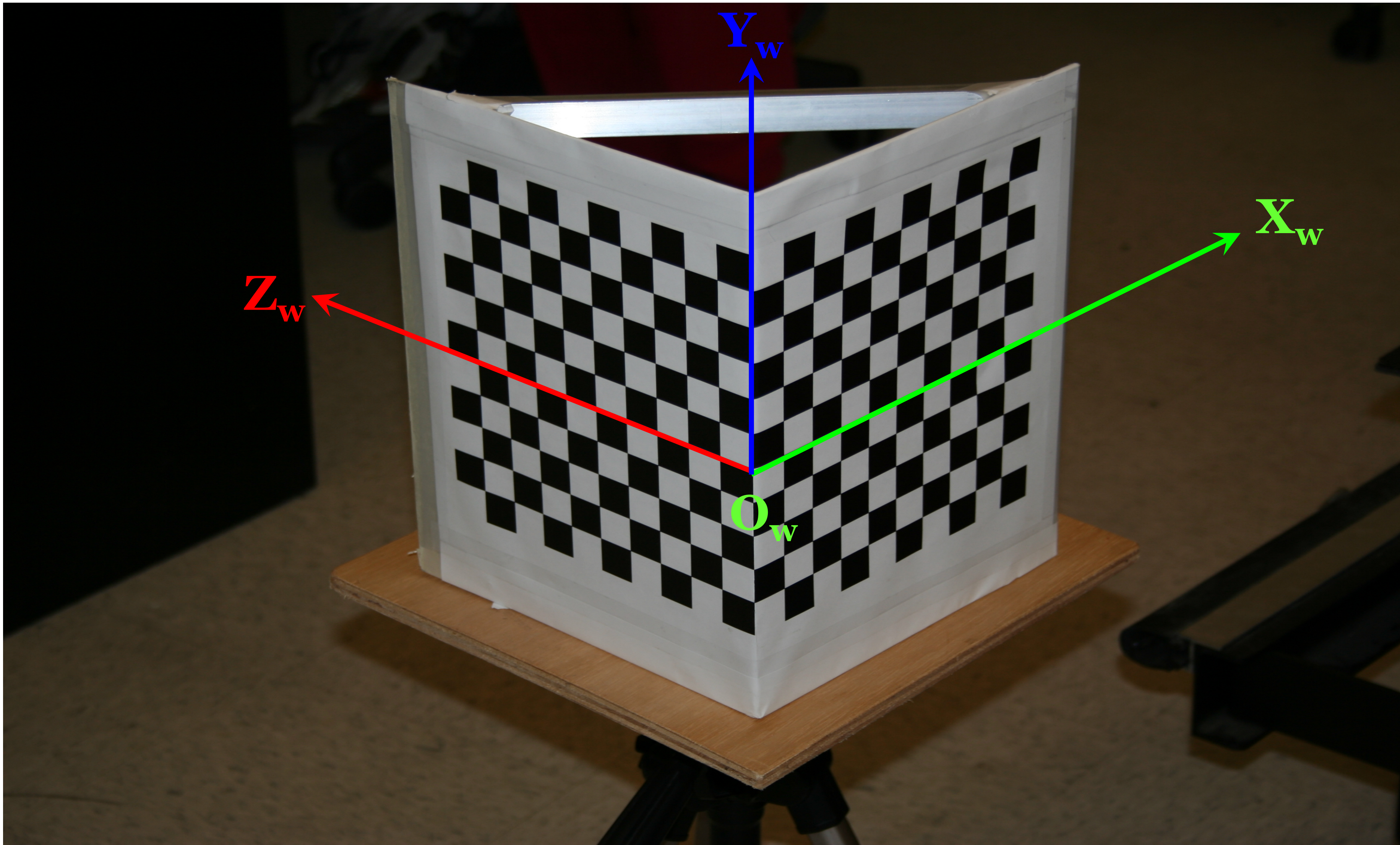
Camera Calibration

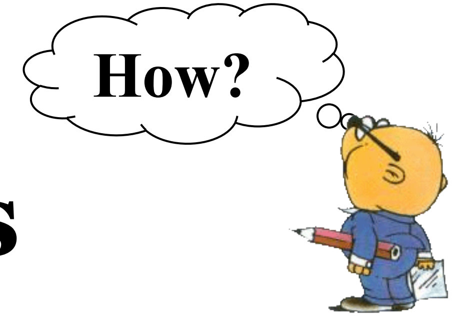
- Equations relating *known* coordinates of 3D points and 2D pixels to solve for camera parameters.

- Set of measurements (3D points and corresponding 2D pixels).

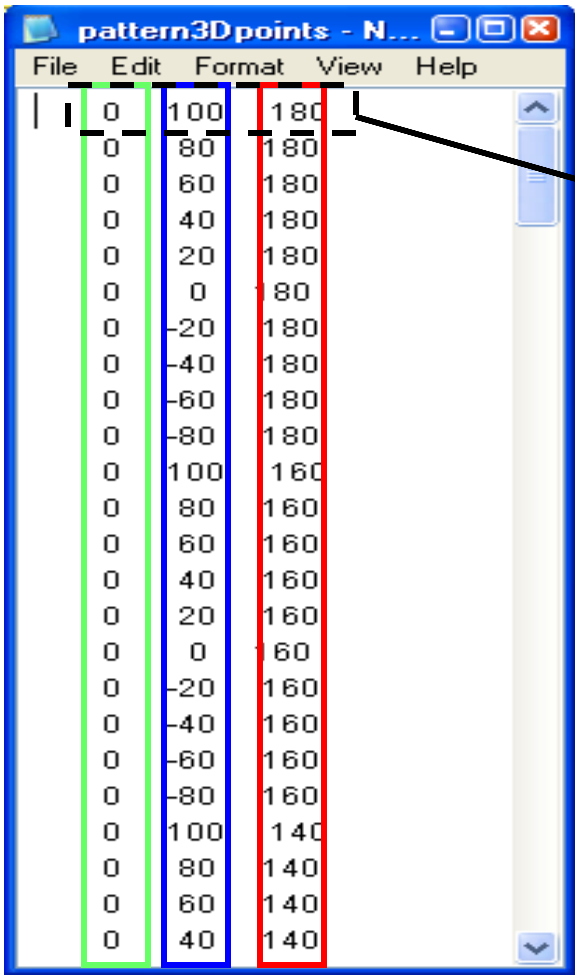


3D Measurements

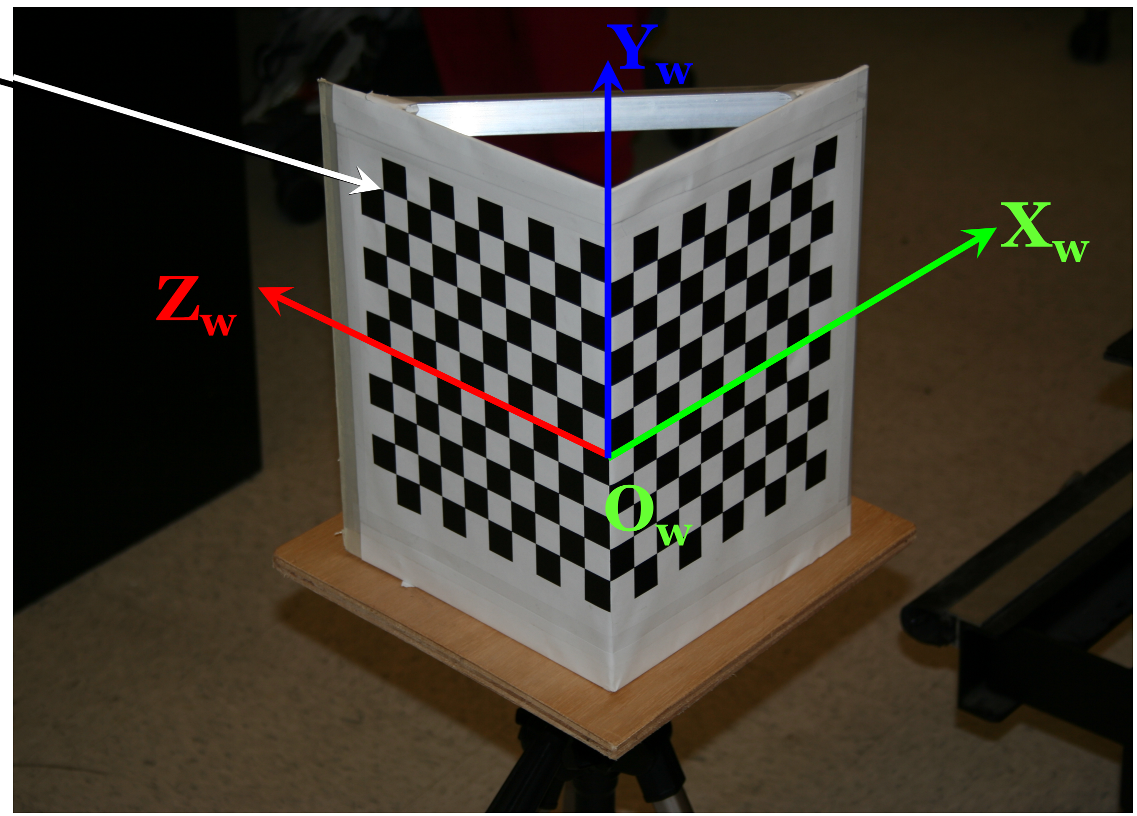


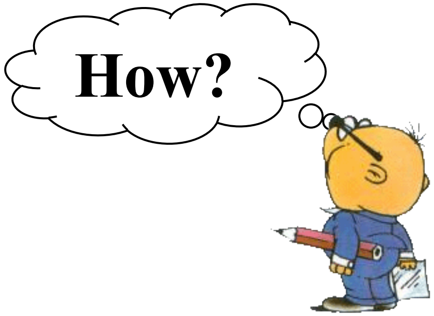


3D Measurements



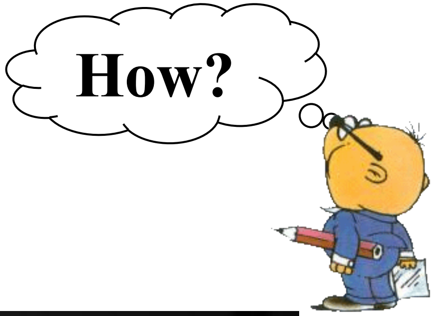
0	100	180
0	80	180
0	60	180
0	40	180
0	20	180
0	0	180
0	-20	180
0	-40	180
0	-60	180
0	-80	180
0	100	160
0	80	160
0	60	160
0	40	160
0	20	160
0	0	160
0	-20	160
0	-40	160
0	-60	160
0	-80	160
0	100	140
0	80	140
0	60	140
0	40	140



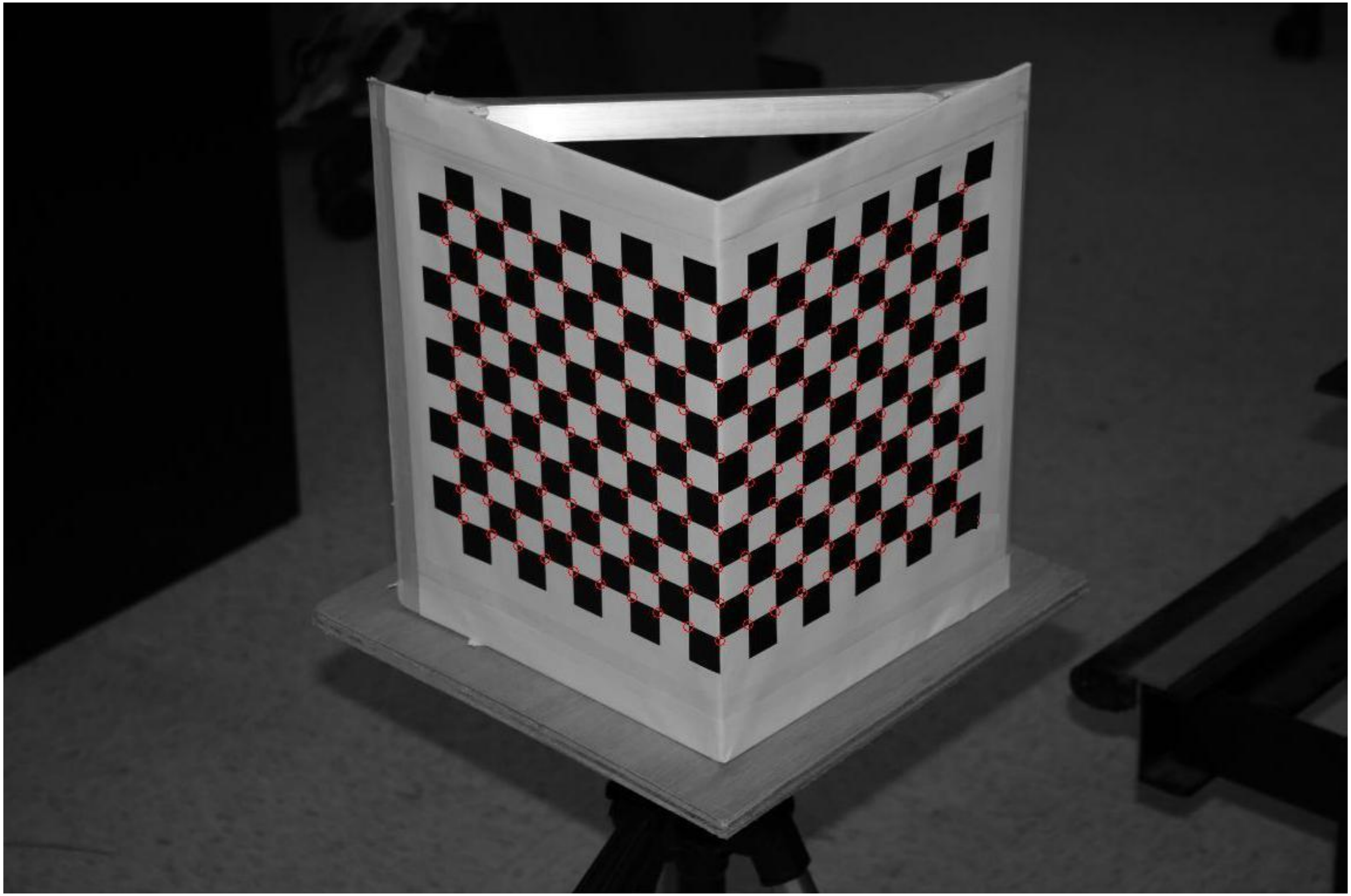


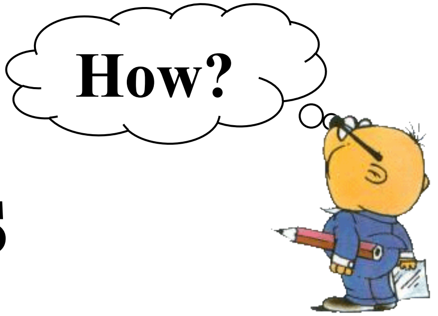
2D Measurements

- User intervention.
- Linear hough transform (to search for straight lines), corners will be points of intersection, yet how to correspond this with 3D points ?!!!
- Corner detectors …



2D Measurements





Camera Equations

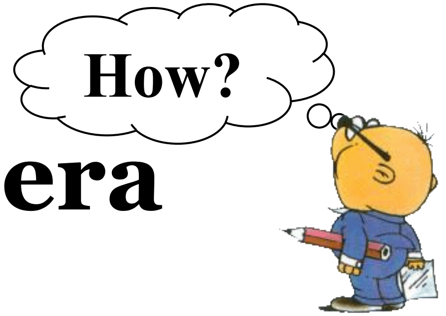
$$\begin{pmatrix} p \\ 1 \end{pmatrix} = \frac{1}{Z_c} M \begin{pmatrix} {}^{w}P \\ 1 \end{pmatrix}$$

$$= \frac{1}{Z_c} K \begin{bmatrix} R & t \end{bmatrix} \begin{pmatrix} {}^{w}P \\ 1 \end{pmatrix}$$

$$= \frac{1}{Z_c} \begin{pmatrix} \alpha & -\alpha \cot\theta & u_o \\ 0 & \dfrac{\beta}{\cot\theta} & v_o \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}_{w}^{c}R & {}_{w}^{c}t \\ 0^t & 1 \end{pmatrix} \begin{pmatrix} {}^{w}P \\ 1 \end{pmatrix}$$

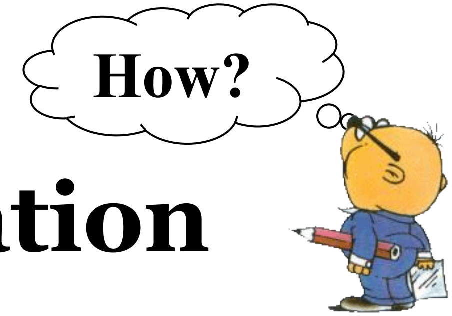
Nonlinear Equations !!!

$${}_{w}^{c}R = \begin{pmatrix} \cos\theta_y \cos\theta_z & \cos\theta_z \sin\theta_x \sin\theta_y - \cos\theta_x \sin\theta_z & \sin\theta_x \sin\theta_z + \cos\theta_x \cos\theta_z \sin\theta_y \\ \cos\theta_y \sin\theta_z & \sin\theta_x \sin\theta_y \sin\theta_z + \cos\theta_x \cos\theta_z & \cos\theta_x \sin\theta_y \sin\theta_z - \cos\theta_z \sin\theta_x \\ -\sin\theta_y & \cos\theta_y \sin\theta_x & \cos\theta_x \cos\theta_y \end{pmatrix}$$



Linear Approach to Camera Calibration

- We decompose the calibration process into:
  - Estimation of the projection matrix.
  - Extraction of camera parameters from the projection matrix.



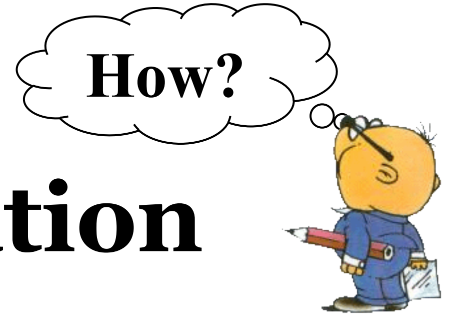
Projection Matrix Estimation

$$\begin{pmatrix} p \\ 1 \end{pmatrix} = \frac{1}{Z_c} M \begin{pmatrix} {}^{w}P \\ 1 \end{pmatrix}$$

$$Z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Given N-3D world points and their corresponding image pixels

$$u_i = \frac{m_{11}X_{wi} + m_{12}Y_{wi} + m_{13}Z_{wi} + m_{14}}{m_{31}X_{wi} + m_{32}Y_{wi} + m_{33}Z_{wi} + m_{34}} \qquad v_i = \frac{m_{21}X_{wi} + m_{22}Y_{wi} + m_{23}Z_{wi} + m_{24}}{m_{31}X_{wi} + m_{32}Y_{wi} + m_{33}Z_{wi} + m_{34}}$$

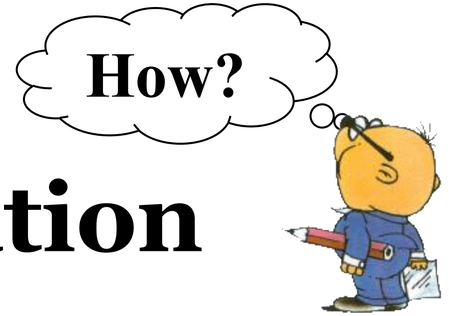


Projection Matrix Estimation

Arrange them in 2N linear equations in ***m's*** *in the form Pm = 0*

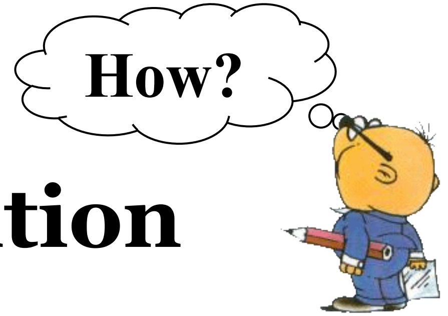
$$P = \begin{bmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1X_1 & -u_1Y_1 & -u_1Z_1 & -u_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1X_1 & -v_1Y_1 & -v_1Z_1 & -v_1 \\ X_2 & Y_2 & Z_2 & 1 & 0 & 0 & 0 & 0 & -u_2X_2 & -u_2Y_2 & -u_2Z_2 & -u_2 \\ 0 & 0 & 0 & 0 & X_2 & Y_2 & Z_2 & 1 & -v_2X_2 & -v_2Y_2 & -v_2Z_2 & -v_2 \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ X_N & Y_N & Z_N & 1 & 0 & 0 & 0 & 0 & -u_NX_N & -u_NY_N & -u_NZ_N & -u_N \\ 0 & 0 & 0 & 0 & X_N & Y_N & Z_N & 1 & -v_NX_N & -v_NY_N & -v_NZ_N & -v_N \end{bmatrix}$$

$$m = [m_{11}, m_{12}, m_{13}, \ldots\ldots, m_{33}, m_{34}]^{\mathrm{T}}$$



Projection Matrix Estimation

- Use singular value decomposition to decompose P as $P = USV^T$
- The solution is the eigenvector V corresponds to the smallest singular value (related to the smallest eigenvalue) in the main diagonal of S.



Projection Matrix Decomposition

Let

$$M = \rho\left(A\ b\right) = K\left(R\ \mathrm{t}\right)$$

a_1^T, a_2^T, a_3^T are the rows of A *and* $R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} r_1^T \\ r_2^T \\ r_3^T \end{bmatrix}$

Then

Intrinsic Parameters

$$\rho = \frac{\varepsilon}{|a_3|} \quad where\ \varepsilon = \pm 1$$

$$u_0 = \rho^2\left(a_1 \cdot a_3\right) \qquad v_0 = \rho^2\left(a_2 \cdot a_3\right)$$

$$\cos\theta = -\frac{\left(a_1 \times a_3\right)\cdot\left(a_2 \times a_3\right)}{|a_1 \times a_3||a_2 \times a_3|}$$

$$\alpha = \rho^2 |a_1 \times a_3| \sin\theta \qquad \beta = \rho^2 |a_2 \times a_3| \sin\theta$$

Extrinsic Parameters

$$r_3 = \rho a_3$$

$$r_1 = \frac{a_2 \times a_3}{|a_2 \times a_3|} \qquad r_2 = r_3 \times r_1$$

$$\theta_y = \sin^{-1} r_{13}$$

$$\theta_x = \cos^{-1}(r_{33} / \cos\theta_y)$$

$$\theta_z = \cos^{-1}(r_{11} / \cos\theta_y)$$

$$t = \rho K^{-1} b$$

Synthetic Data

Parameter	$t_x(cm)$	t_y	t_z	$\theta_x(rad)$	θ_y	θ_z	$\alpha\ (pixel)$	β	u_0	v_0	$\theta(rad)$
Ground Truth	-27	-28	701	0.09	0.8	-0.03	556	549	172	121	$\pi/2$

```
% intrinsic parameters

% optical center position when projected on the image plane (row and
% column)
Groundtruth_Camera.Uo = 172 ; % column
Groundtruth_Camera.Vo = 121 ; % row

% skew angle in radians
Groundtruth_Camera.Theta = pi/2 ; %  = 1.5708 -- 90 in degrees

% the focal length measured in pixels (alpha = kf and beta = lf)
Groundtruth_Camera.Alpha = 556 ; % in pixels
Groundtruth_Camera.Beta = 549 ; % in pixels

% extrinsic parameters

% the rotation angles in the three directions ... measure in radians

Groundtruth_Camera.theta_x = 0.09 ; % 5.15 in degrees
Groundtruth_Camera.theta_y = 0.8 ; % 45.8 in degrees
Groundtruth_Camera.theta_z = -0.03 ; % -1.7 in degrees

% the translation vector which measures how far the origin of the world
% coordinate system from the camera coordinate system ... measure in
% the worlds metric (i.e. inches , feet ...)

Groundtruth_Camera.tx = -27 ;
Groundtruth_Camera.ty = -28 ;
Groundtruth_Camera.tz = 701 ;
```

Groundtruth M

```
%% Obtain the perspective projection matrix M = K * Mproj * D

% the intrinsic parameters matrix
K = get_IntrinsicMatrix
(Groundtruth_Camera.Alpha,Groundtruth_Camera.Beta,
 Groundtruth_Camera.Theta,Groundtruth_Camera.Uo,Groundtruth_Camera.Vo);

% the perspective projection matrix
Mproj = [ 1    0    0    0 ;
          0    1    0    0 ;
          0    0    1    0 ];

% the extrinsic parameters matrix
D = get_ExtrinsicMatrix
(Groundtruth_Camera.theta_x,Groundtruth_Camera.theta_y,
Groundtruth_Camera.theta_z,Groundtruth_Camera.tx,Groundtruth_Camera.ty,
Groundtruth_Camera.tz);
```

```
Groundtruth_M = K * Mproj * D ;
```

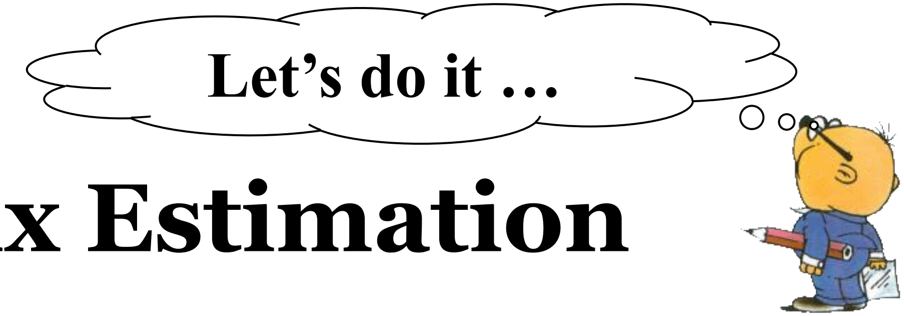
```
function K = get_IntrinsicMatrix (alpha,beta,theta,Uo,Vo)

K = [alpha    -alpha * cot(theta)     Uo ;
       0        beta/sin(theta)       Vo ;
       0             0                 1  ];
```

```
function D = get_ExtrinsicMatrix (theta_x,theta_y,theta_z,tx,ty,tz)

R = [ cos(theta_y)*cos(theta_z)     cos(theta_z) * sin(theta_x) * sin(theta_y) - cos(theta_x) * sin(theta_z)     sin(theta_x) * sin(theta_z) + cos(theta_x) * cos(theta_z) * sin(theta_y);
      cos(theta_y) * sin(theta_z)   sin(theta_x) * sin(theta_y) * sin(theta_z) + cos(theta_x) * cos(theta_z)     cos(theta_x) * sin(theta_y) * sin(theta_z) - cos(theta_z) * sin(theta_x);
          -sin(theta_y)                             cos(theta_y) * sin(theta_x)                                           cos(theta_x) * cos(theta_y)                          ];

D = [R [tx;ty;tz];
    0 0 0 1];
```

Let’s do it …

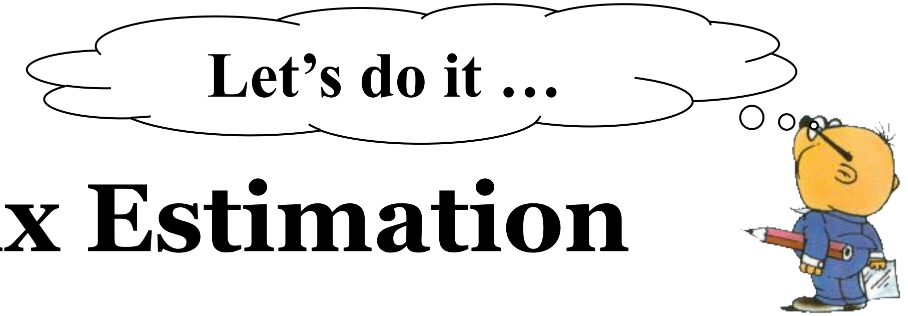
Corresponding 2D Pixels

```
%% using the 3D points in file Points3D.txt, generate the corresponding
%% projected 2D points

load Points3D.txt;
Npoints = size(Points3D,1);

% adding homogenous coordinate to the 3d points
Points3D(:,4) = ones(Npoints,1);

Xw =Points3D(:,1);
Yw =Points3D(:,2);
Zw =Points3D(:,3);

m31 = Groundtruth_M(3,1);
m32 = Groundtruth_M(3,2);
m33 = Groundtruth_M(3,3);
m34 = Groundtruth_M(3,4);

% recall p = (1/Zc) * MP;
Zc = m31 .* Xw + m32 .* Yw + m33 .* Zw + m34;

for i = 1 : Npoints
    Points2D(i,:) = (1/Zc(i)) * Groundtruth_M * Points3D(i,:)';
end

% since those 2D points are supposed to be image coordinates, therefore
% we will round them to integers
Points2D = round(Points2D);
```



Projection Matrix Estimation

```
function P = get_Pmatrix (Points3D, Points2D)

% each pair of points will generate two rows in the P matrix as follows
Npoints = size(Points3D,1);

Xw = Points3D(:,1);
Yw = Points3D(:,2);
Zw = Points3D(:,3);

u = Points2D(:,1);
v = Points2D(:,2);

n = 0;
for i = 1 : Npoints
    n = n +1;
    P(n,:) = [Xw(i) Yw(i) Zw(i) 1      0      0      0    0
                                 -u(i)*Xw(i) -u(i)*Yw(i) -u(i)*Zw(i) -u(i)];

    n = n + 1 ;
    P(n,:) = [   0      0      0    0    Xw(i) Yw(i) Zw(i) 1
                                 -v(i)*Xw(i) -v(i)*Yw(i) -v(i)*Zw(i) -v(i)];
end
```



Projection Matrix Estimation

```
function calibrate (Points2D, Points3D)

P = get_Pmatrix( Points3D, Points2D);

[U,S,V] = svd(P);

% the solution is the last column of V which corresponds to the
% smallest eignvalue (singular value) of D

m = V(:,end);

m11 = m(1);
m12 = m(2);
m13 = m(3);
m14 = m(4);

m21 = m(5);
m22 = m(6);
m23 = m(7);
m24 = m(8);

m31 = m(9);
m32 = m(10);
m33 = m(11);
m34 = m(12);

M = [m11 m12 m13 m14 ;
     m21 m22 m23 m24 ;
     m31 m32 m33 m34 ];
```

Let’s do it …

Projection Matrix Decomposition

```matlab
function Camera = DecomposeM (M,Points2D,Points3D)

% let M = [A b]
A = M(:,1:3);
b = M(:,end);

% the rows of A
a1 = A(1,:);
a2 = A(2,:);
a3 = A(3,:);

% getting the scale factor
rho = 1/norm(a3);

% the principal point (image center)
Camera.Uo = rho^2 * dot(a1,a3);
Camera.Vo = rho^2 * dot(a2,a3);
```

Projection Matrix Decomposition

```
% determining the sign of the scale factor rho
% first we choose a calibrating point far from the image center
distance = ((Points2D(:,1) - Camera.Uo).^2 + (Points2D(:,2) -
Camera.Vo).^2).^(1/2);

[maxD,index] = max(distance);

% use the estimated projection matrix to get the projection of the
% chosen calibrating point
u = M(1,1)*Points3D(i,1) + M(1,2)* Points3D(i,2)
                         + M(1,3)* Points3D(i,3)+ M(1,4);
v = M(2,1)*Points3D(i,1) + M(2,2)* Points3D(i,2)
                         + M(2,3)* Points3D(i,3)+ M(2,4);

U_hat = Points2D(i,1) - Camera.Uo;
V_hat = Points2D(i,2) - Camera.Vo;

% since sign function give zero for zero input
if(u==0) u = u + 1;end
if(v==0) v = v + 1;end
if(U_hat==0) U_hat = U_hat + 1;end
if(V_hat==0) V_hat = V_hat + 1;end

if(( sign(u) == sign(U_hat)) && (sign(v) == sign(V_hat)))
    rho = rho;
else
    rho = -rho;
end
```

Let's do it …

Projection Matrix Decomposition

```
% the last row in the rotation matrix (the extrinsic parameter)
r3 = rho .* a3 ;

% the skew angle
cosTheta = -(dot(cross(a1,a3),cross(a2,a3)))
                    /(norm(cross(a1,a3))* norm(cross(a2,a3)));
Camera.Theta = acos(cosTheta);

% focus length in pixels
Camera.Alpha = rho^2 * norm(cross(a1,a3)) * sin(Camera.Theta);
Camera.Beta  = rho^2 * norm(cross(a2,a3)) * sin(Camera.Theta);

% the first row of the rotation matrix
r1 = cross(a2,a3) ./ norm(cross(a2,a3));

% the second row of the rotation matrix
r2 = cross(r3,r1);

% rotation angles in the three directions
Camera.theta_y = asin(r1(3));
Camera.theta_x = acos(r3(3)/cos(Camera.theta_y));
Camera.theta_z = acos(r1(1)/cos(Camera.theta_y));

% the translation vector
K = get_IntrinsicMatrix
(Camera.Alpha,Camera.Beta,Camera.Theta,Camera.Uo,Camera.Vo);

t = rho .* (inv(K)*b);
Camera.tx = t(1);
Camera.ty = t(2);
```

Does it work ?!!!

Evaluation

Groundtruth_Camera =		Camera =	
Uo:	172	Uo:	172.3411
Vo:	121	Vo:	120.6886
Theta:	1.5708	Theta:	1.5717
Alpha:	556	Alpha:	556.4091
Beta:	549	Beta:	549.3892
theta_x:	0.0900	theta_x:	0.1570
theta_y:	0.8000	theta_y:	0.7910
theta_z:	-0.0300	theta_z:	0.1328
tx:	-27	tx:	-27.4131
ty:	-28	ty:	-27.6916
tz:	701	tz:	701.4125

Camera Equations

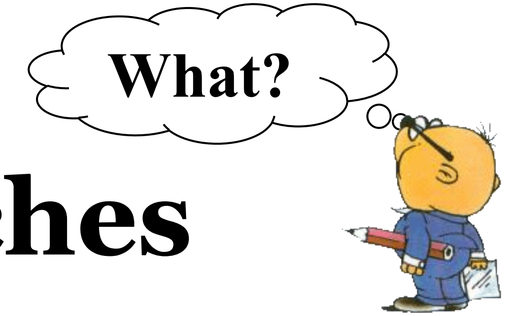
$$\begin{pmatrix} p \\ 1 \end{pmatrix} = \frac{1}{Z_c} M \begin{pmatrix} {}^{w}P \\ 1 \end{pmatrix}$$

$$= \frac{1}{Z_c} K \begin{bmatrix} R & t \end{bmatrix} \begin{pmatrix} {}^{w}P \\ 1 \end{pmatrix}$$

$$= \frac{1}{Z_c} \begin{pmatrix} \alpha & -\alpha \cot\theta & u_o \\ 0 & \dfrac{\beta}{\cot\theta} & v_o \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}_{w}^{c}R & {}_{w}^{c}t \\ 0^t & 1 \end{pmatrix} \begin{pmatrix} {}^{w}P \\ 1 \end{pmatrix}$$

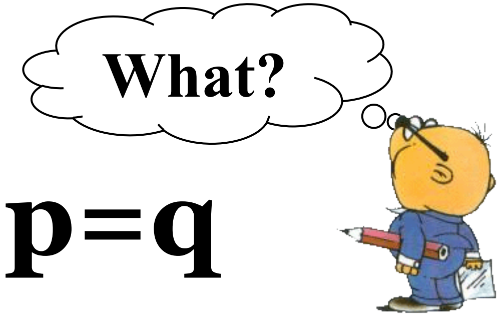
Nonlinear Equations !!!

$${}_{w}^{c}R = \begin{pmatrix} \cos\theta_y \cos\theta_z & \cos\theta_z \sin\theta_x \sin\theta_y - \cos\theta_x \sin\theta_z & \sin\theta_x \sin\theta_z + \cos\theta_x \cos\theta_z \sin\theta_y \\ \cos\theta_y \sin\theta_z & \sin\theta_x \sin\theta_y \sin\theta_z + \cos\theta_x \cos\theta_z & \cos\theta_x \sin\theta_y \sin\theta_z - \cos\theta_z \sin\theta_x \\ -\sin\theta_y & \cos\theta_y \sin\theta_x & \cos\theta_x \cos\theta_y \end{pmatrix}$$



Nonlinear Approaches

- Find an optimal solution in a least-squares sense.
- Iterative, hence require *initial* solution.
- Depend on equations derivatives.
- Rely on first-order Taylor expansion of functions in the *neighborhood* of the current estimated solution.
- Examples:
  - Newton's method.
  - Gauss-Newton method.
  - Levenberg method.
  - Levenberg-Marquardt method.



Newton's Method – p=q

$$f(x)=0$$

$$\text{where}\quad f=\begin{pmatrix} f_1 & f_2 & \cdots & f_p \end{pmatrix}^T$$

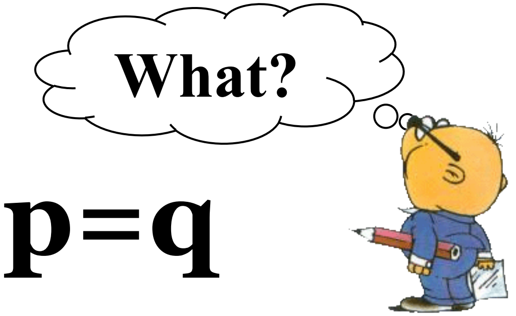
$$x=\begin{pmatrix} x_1 & x_2 & \cdots & x_q \end{pmatrix}^T$$

$$f(x+\delta x)\approx f(x)+J_f(x)\delta x$$

Objective

Find δx such that f(x+δx)=0
given the current estimate of the solution

$$\text{i.e.}\quad f(x)+J_f(x)\delta x=0\Rightarrow J_f(x)\delta x=-f(x)$$



Newton's Method– p=q

$$J_f(x)\delta x = -f(x)$$

$$\delta x = \begin{bmatrix} \delta\alpha \\ \delta\beta \\ \delta\theta \\ \delta u_o \\ \delta v_o \\ \delta\theta_x \\ \delta\theta_y \\ \delta\theta_z \\ \delta t_x \\ \delta t_y \\ \delta t_z \end{bmatrix}$$

Change of solution in the current iteration

Camera Parameters

$$x = \begin{bmatrix} \alpha & \beta & \theta & u_o & v_o & \theta_x & \theta_y & \theta_z & t_x & t_y & t_z \end{bmatrix}^{\mathrm{T}}$$

Camera equation evaluated
Using the solution in the current iteration

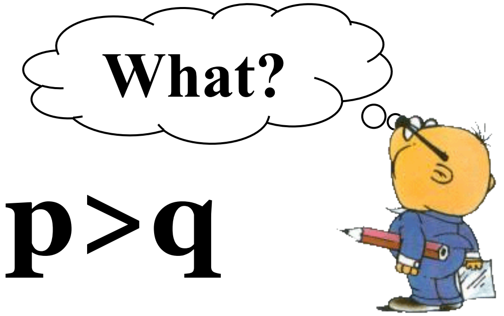
$$\begin{pmatrix} p \\ 1 \end{pmatrix} = \frac{1}{Z_c} M \begin{pmatrix} {}^{w}P \\ 1 \end{pmatrix}$$

The value of the Jacobian (first derivative) of the camera equation by substituting by the solution of the current iteration

Now it is linear again … solve for δx

It is in the form of

Ax = b



Newton's Method – p>q

$$\nabla E(x) = \nabla \left| f(x) \right|^2 = 2\nabla f(x).f(x) = 0$$

$$F(x) @ \frac{1}{2}\nabla E(x) = \nabla f(x).f(x) = J_f^T(x) f(x) = 0$$

Use Newton method for p=q to solve F(x) =0

$$J_F(x)\delta x = -F(x)$$

$$\left(J_f^T(x) J_f(x) + \nabla J_f^T(x) f(x) \right) \delta x = -J_f^T(x) f(x)$$

Hessian

Apply Newton method to solve for δx
in each iteration but with different A and b

Gauss-Newton's Method

$$E(x+\delta x)=\left|f(x+\delta x)\right|^2 \approx \left|f(x)+J_f(x)\delta x\right|^2=0$$

Avoids getting the gradient of E, hence avoid Hessian

$$f(x)+J_f(x)\delta x=0$$

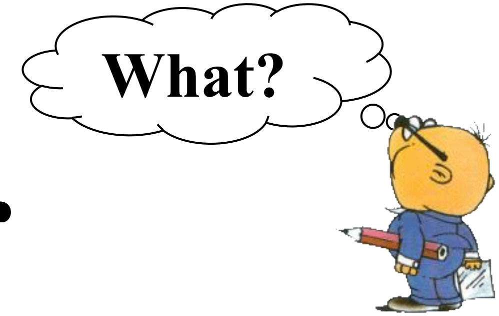
i.e. $$J_f(x)\delta x=-f(x)$$

According to the definition of pseudo-inverse

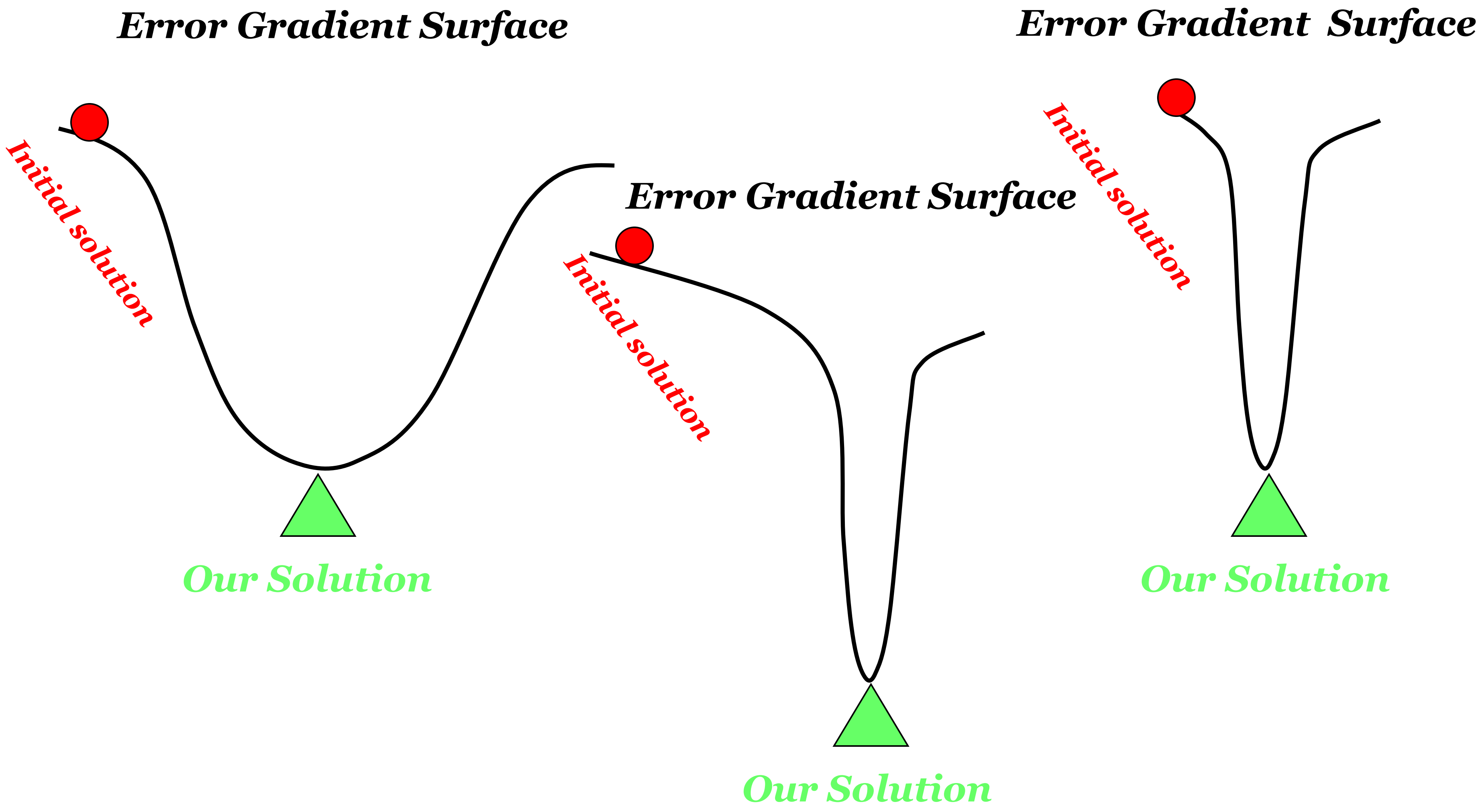
$$J_f^T(x)J_f(x)\delta x=-J_f^T(x)f(x)$$

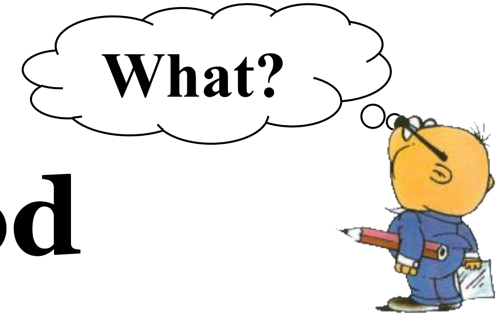
Compare to Newton's method …

$$\left(J_f^T(x)J_f(x)+\nabla J_f^T(x)f(x)\right)\delta x=-J_f^T(x)f(x)$$



Damping Factor





Levenberg Method

$$\left(J_f^T(x)J_f(x)+\mu Id\right)\delta x=-J_f^T(x)f(x)$$

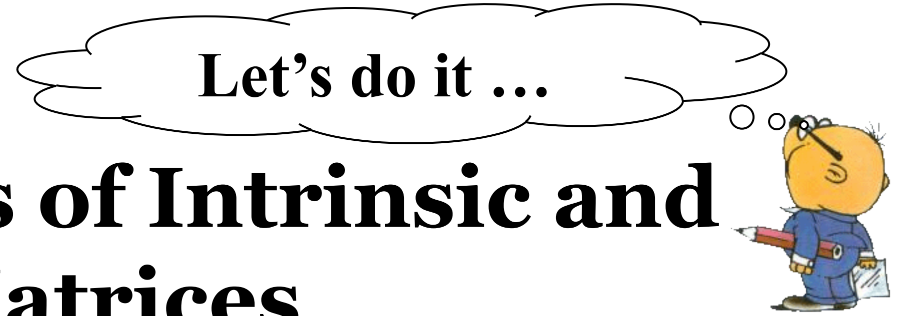
Damping factor varies according to the behavior of the error in the current iteration

Levenberg-Marquardt Method

$$\left(J_f^T(x)J_f(x)+\mu\,diag\left(J_f^T(x)J_f(x)\right)\right)\delta x=-J_f^T(x)f(x)$$

Defining Equation Variables

```
function Camera = Tune_Camera_Parameters (Camera, Points2D, Points3D , Approach)

% using the Matlab symbolic toolbox, find the analytic form of the
% Jacobians of the errors du and dv with respect to the parameters
% we want to refine (those are errors between the points 2d (image points)
% and the corresponding projected 3d points using the
% current estimated projection matrix.

% intrinsic parameters to be tuned
syms Uo Vo Alpha Beta theta

% extrinsic parameters to be tuned
syms tx ty tz theta_x theta_y theta_z real

% symbols for the 3d and their corresponding 2d points
syms Xw Yw Zw u_image v_image real
```



Defining Expressions of Intrinsic and Extrinsic Matrices

```
% expresion of the intrinsic parameter matrix
K = [Alpha  -Alpha * cot(theta)     Uo ;
       0       Beta/sin(theta)       Vo ;
       0             0               1   ];

% the extrinsic parameter matrix
% expresion of the rotation matrix
R = [ cos(theta_y)*cos(theta_z) ...
      cos(theta_z) * sin(theta_x) * sin(theta_y) - cos(theta_x) * sin(theta_z) ...
      sin(theta_x) * sin(theta_z) + cos(theta_x) * cos(theta_z) * sin(theta_y) ;

       cos(theta_y) * sin(theta_z) ...
      sin(theta_x) * sin(theta_y) * sin(theta_z) + cos(theta_x) * cos(theta_z) ...
      cos(theta_x) * sin(theta_y) * sin(theta_z) - cos(theta_z) * sin(theta_x) ;

      - sin(theta_y)  ...
      cos(theta_y) * sin(theta_x) ...
      cos(theta_x) * cos(theta_y)];


 % the expression for the translation vecotr
 t = [tx;
      ty;
      tz];
```

Defining Expressions of Camera Equation and Error Measure

```
% the expression of the perspective projection of the world points
p = K * [R t] * [Xw;Yw;Zw;1];

% making sure that the last coordinate is one (homogenous coordinates)
u = p(1)/p(3);
v = p(2)/p(3);

% the expression the geometric distance in x and y direction between the image
% points and the corresponding 3d points being projected on the image
% u_image and v_image are the 2D points extracted from the image
% u and v are their corresponding 3d points being projected on the image
% plane using the current estimated projection matrix

dx = ((u_image - u)^2)^(1/2);
dy = ((v_image - v)^2)^(1/2);

% evaluate the symbolic expression of the Jacobian w.r.t. the estimated
% parameters
Jx = jacobian(dx, [Alpha,Beta,Uo,Vo,theta,theta_x,theta_y,theta_z,tx,ty,tz]);
Jy = jacobian(dy, [Alpha,Beta,Uo,Vo,theta,theta_x,theta_y,theta_z,tx,ty,tz]);
```

Let’s do it …

Jacobian Expression (Sample)

```
Jx(1) =
     -1/((u_image-((Alpha*cos(theta_y)*cos(theta_z)-
     Beta*cot(theta)*cos(theta_y)*sin(theta_z)-
     Uo*sin(theta_y))*Xw+(Alpha*(sin(theta_x)*sin(theta_y)*cos(theta_z)-
     cos(theta_x)*sin(theta_z))-
     Beta*cot(theta)*(sin(theta_x)*sin(theta_y)*sin(theta_z)+cos(theta_x)*cos(theta_z))+Uo*
     sin(theta_x)*cos(theta_y))*Yw+(Alpha*(sin(theta_x)*sin(theta_z)+cos(theta_x)*sin(theta
     _y)*cos(theta_z))-Beta*cot(theta)*(cos(theta_x)*sin(theta_y)*sin(theta_z)-
     sin(theta_x)*cos(theta_z))+Uo*cos(theta_x)*cos(theta_y))*Zw+Alpha*tx-
     Beta*cot(theta)*ty+Uo*tz)/(-
     sin(theta_y)*Xw+sin(theta_x)*cos(theta_y)*Yw+cos(theta_x)*cos(theta_y)*Zw+tz))^2)^(1/2
     )*(u_image-((Alpha*cos(theta_y)*cos(theta_z)-
     Beta*cot(theta)*cos(theta_y)*sin(theta_z)-
     Uo*sin(theta_y))*Xw+(Alpha*(sin(theta_x)*sin(theta_y)*cos(theta_z)-
     cos(theta_x)*sin(theta_z))-
     Beta*cot(theta)*(sin(theta_x)*sin(theta_y)*sin(theta_z)+cos(theta_x)*cos(theta_z))+Uo*
     sin(theta_x)*cos(theta_y))*Yw+(Alpha*(sin(theta_x)*sin(theta_z)+cos(theta_x)*sin(theta
     _y)*cos(theta_z))-Beta*cot(theta)*(cos(theta_x)*sin(theta_y)*sin(theta_z)-
     sin(theta_x)*cos(theta_z))+Uo*cos(theta_x)*cos(theta_y))*Zw+Alpha*tx-
     Beta*cot(theta)*ty+Uo*tz)/(-
     sin(theta_y)*Xw+sin(theta_x)*cos(theta_y)*Yw+cos(theta_x)*cos(theta_y)*Zw+tz))*(cos(th
     eta_y)*cos(theta_z)*Xw+(sin(theta_x)*sin(theta_y)*cos(theta_z)-
     cos(theta_x)*sin(theta_z))*Yw+(sin(theta_x)*sin(theta_z)+cos(theta_x)*sin(theta_y)*cos
     (theta_z))*Zw+tx)/(-
     sin(theta_y)*Xw+sin(theta_x)*cos(theta_y)*Yw+cos(theta_x)*cos(theta_y)*Zw+tz);
```

Setting Initial Solution

```
% getting the initial parameters
tx = Camera.tx;
ty = Camera.ty;
tz = Camera.tz;

theta_x = Camera.theta_x;
theta_y = Camera.theta_y;
theta_z = Camera.theta_z;

Uo = Camera.Uo;
Vo = Camera.Vo;

theta = Camera.Theta;

Alpha = Camera.Alpha;
Beta = Camera.Beta;

% set the number of iterations
n_iterations = 20;

% initial value of the damping factor Mu
Mu = 0.0001;

% flag for either accept or reject the current update
update = 1;

% number of data points (twice)
Ndata = 2 * size(Points2D,1); % each point put two constaints one for u and the other for v

% number of parameters to be tuned
Nparams = 11;
```

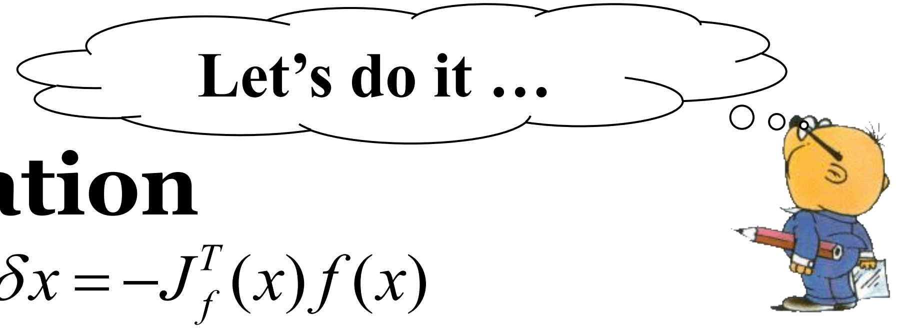
Iteration Initialization

```
for iter = 1 : n_iterations
    if (update)
       % compute the intrinsic parameter matrix using the current
       % estimated parameters
       K = get_IntrinsicMatrix (Alpha,Beta,theta,Uo,Vo);

       % compute the rotation matrix
       R = getRotationMatrix(theta_x,theta_y,theta_z);

       % compute the translation vector
       t = [tx;
             ty;
             tz];

       % evaluate the Jacobian at the current parameter values and the
       % values of geometric distance dx and dy
       J = zeros(Ndata,Nparams);
       d = zeros(Ndata,1);
```

Let’s do it …

Jacobian and Error Evaluation

```
for i = 1 : size(Points3D,1)
    Xw = Points3D(i,1);
    Yw = Points3D(i,2);
    Zw = Points3D(i,3);

    u_image = Points2D(i,1);
    v_image = Points2D(i,2);
    % computing the value of Jx and Jy evaluated at tghe current
    % world point and the current parameters.
    [jx,jy] =
              computeJ(Xw,Yw,Zw,u_image,v_image,Alpha,Beta,Uo,Vo,
                                    theta,theta_x,theta_y,theta_z,tx,ty,tz);

    J(2*(i-1)+1,:) = jx;
    J(2*(i-1)+2,:) = jy;

    % perspective projection of the current world point
    p = K * [R t] * [Xw;Yw;Zw;1];

    % making sure that the last coordinate is one (homogenous coordinates)
   u = p(1)/p(3);
   v = p(2)/p(3);

   % compute the geometric distance in x and y directions
   d(2*(i-1)+1,:) = ((Points2D(i,1) - u) ^2)^(1/2);
   d(2*(i-1)+2,:) = ((Points2D(i,2) - v)^2)^(1/2);
end
```



δx Evaluation

$$\left(J_f^T(x)J_f(x)+\mu Id\right)\delta x=-J_f^T(x)f(x)$$

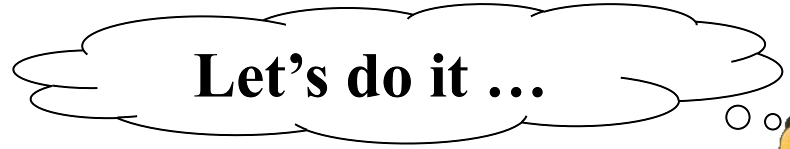
$$\left(J_f^T(x)J_f(x)+\mu diag\left(J_f^T(x)J_f(x)\right)\right)\delta x=-J_f^T(x)f(x)$$

```
% compute the approximated hessian matrix
H = J' * J;

if iter == 1 % the first iteration : compute the initial total error
    error = dot(d,d);
end

% apply the damping factor to the hessian matrix
switch Approach
    case 'Levenberg',
        H_lm = H + (Mu * eye(Nparams,Nparams));
    case 'Levenberg-Marquardt',
        H_lm = H + (Mu * diag(diag(H)));
end


% computing the change in the estimated parameters
delta_x = -(1/2).* inv(H_lm) * (J'*d(:));
```

Let’s do it …

Compute Updated Parameters

```
% compute the updated parameters
alpha_lm = Alpha + delta_x(1);
beta_lm = Beta + delta_x(2);
Uo_lm = Uo + delta_x(3);
Vo_lm = Vo + delta_x(4);
theta_lm = theta + delta_x(5);
theta_lm_x = theta_x + delta_x(6);
theta_lm_y = theta_y + delta_x(7);
theta_lm_z = theta_z + delta_x(8);
tx_lm = tx + delta_x(9);
ty_lm = ty + delta_x(10);
tz_lm = tz + delta_x(11);

% update the total geometric distance at the updated parameters
% compute the intrinsic parameter matrix using the current
% estimated parameters
K = get_IntrinsicMatrix (alpha_lm,beta_lm,theta_lm,Uo_lm,Vo_lm);

% compute the rotation matrix
R = getRotationMatrix(theta_lm_x,theta_lm_y,theta_lm_z);

% compute the translation vector
t = [tx_lm;
     ty_lm;
     tz_lm];
```



Error Resulted from Updating

```
d_lm = zeros(Ndata,1);
for i = 1 : size(Points3D,1)
     Xw = Points3D(i,1);
     Yw = Points3D(i,2);
     Zw = Points3D(i,3);

     % perspective projection of the current world point
     p = K * [R t] * [Xw;Yw;Zw;1];

     % making sure that the last coordinate is one (homogenous coordinates)
    u = p(1)/p(3);
    v = p(2)/p(3);

    % compute the geometric distance in x and y directions
    d_lm(2*(i-1)+1,:) = ((Points2D(i,1) - u)^2)^(1/2);
    d_lm(2*(i-1)+2,:) = ((Points2D(i,2) - v)^2)^(1/2);
end

% computing the error between the image coordinates and projective
% coordinates using the updated parameters
error_lm = dot(d_lm,d_lm);
```

Let’s do it …

Accept Update?!!

```
% if the total geometric distance of the updated parameters is less
% than the previous one then makes the updated parameters to be the
% current parameters and decreases the value of the damping factor.

if(error_lm < error)
    Mu = Mu/10;
     Alpha = alpha_lm;
     Beta = beta_lm;
     Uo = Uo_lm;
     Vo = Vo_lm ;
     theta = theta_lm;
     theta_x = theta_lm_x;
     theta_y = theta_lm_y ;
     theta_z = theta_lm_z ;
     tx = tx_lm;
     ty = ty_lm;
     tz = tz_lm;
     error = error_lm;

     update = 1;
else
    % otherwise increase the value of the damping factor and try
    % again
    update = 0;
    Mu = Mu * 10;
end
```

Move slower … we are approaching the solution (annotation for `Mu = Mu/10;`)

Move faster … we are away from the solution (annotation for `Mu = Mu * 10;`)

Evaluation

Groundtruth_Camera =		Camera =		Camera_LM =	
Uo:	172	Uo:	172.3411	Uo:	172.6042
Vo:	121	Vo:	120.6886	Vo:	120.3252
Theta:	1.5708	Theta:	1.5717	Theta:	1.5717
Alpha:	556	Alpha:	556.4091	Alpha:	556.3539
Beta:	549	Beta:	549.3892	Beta:	549.2924
theta_x:	0.0900	theta_x:	0.1570	theta_x:	0.0880
theta_y:	0.8000	theta_y:	0.7910	theta_y:	0.7988
theta_z:	-0.0300	theta_z:	0.1328	theta_z:	-0.0302
tx:	-27	tx:	-27.4131	tx:	-27.7481
ty:	-28	ty:	-27.6916	ty:	-27.2363
tz:	701	tz:	701.4125	tz:	701.2561

Thank you