Advanced Topics in 3D Computer Vision

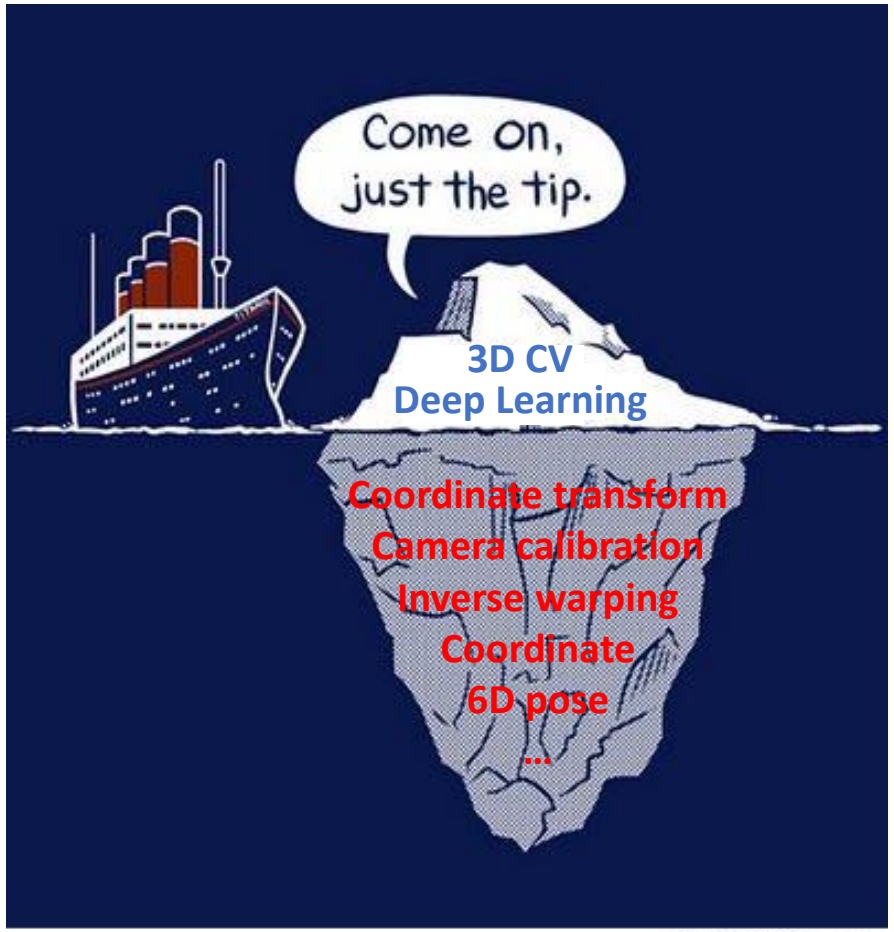
Lecture 1. Basic Tips and Knowledges for Deep 3D Vision

14.04.2021

HyunJun Jung

Motivation

In Deep 3D Vision, the deep learning part is just the tip of the iceberg!



Contents :

2. Practical uses (for assignments)

1. Rendering 3D object for rendering based 6D Pose tracking pipeline

   Intro - Rendering based 6D Pose tracking

   1.1. Pyrender : easy rendering library for python

   1.2. How to deal with unknown coordinate orientation

   1.3. Conversion between different coordinate system

2. Augmented Reality for Outside-In tracking setup

   Intro - What is Outside-In tracking? And useful application

   2.1. Camera calibration

   2.2. Coordinate conversion

3. Self-supervised learning loss via warping

   Intro - Why do we do need self-supervised learning?

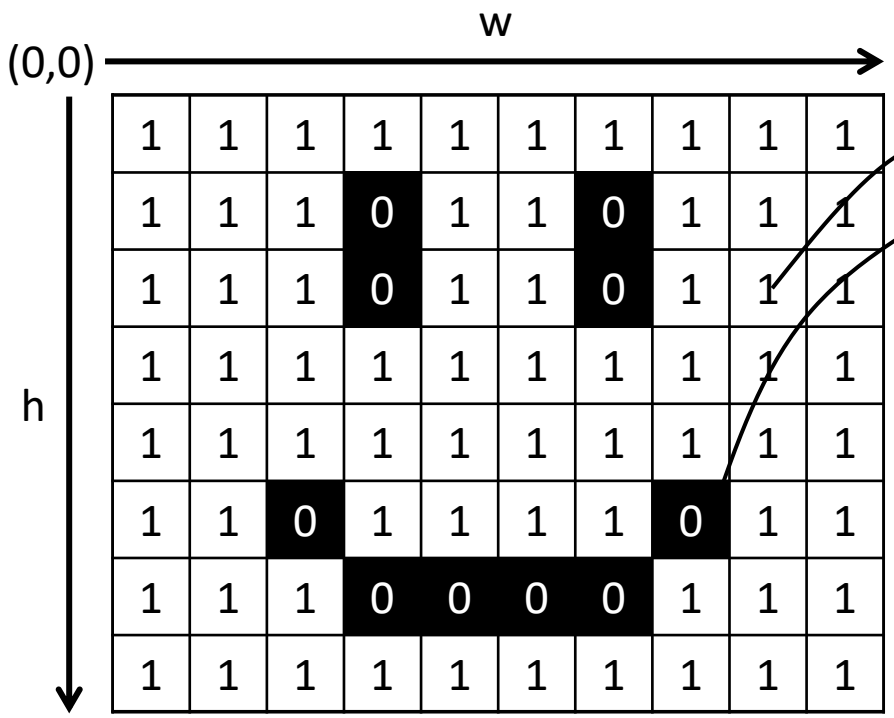
   3.1. Warping the image using Depth and Relative pose

   3.2. Differentiable bilinear interpolation

1. Basic theories

1 - Homogeneous coordinate system

Image is just a 2D array of pixels filled with either scalars (grey scale image) or 3D vectors (RGB image)



We use 2D vector (coordinate) to access the pixel value

Smiley[(8,2)] -> 1

Smiley[(7,5)] -> 0

Adv

- Very intuitive
- Easy manipulation via matrix multiplication

Disadv

- **Matrix multiplication is not capable with the origin (0,0)**

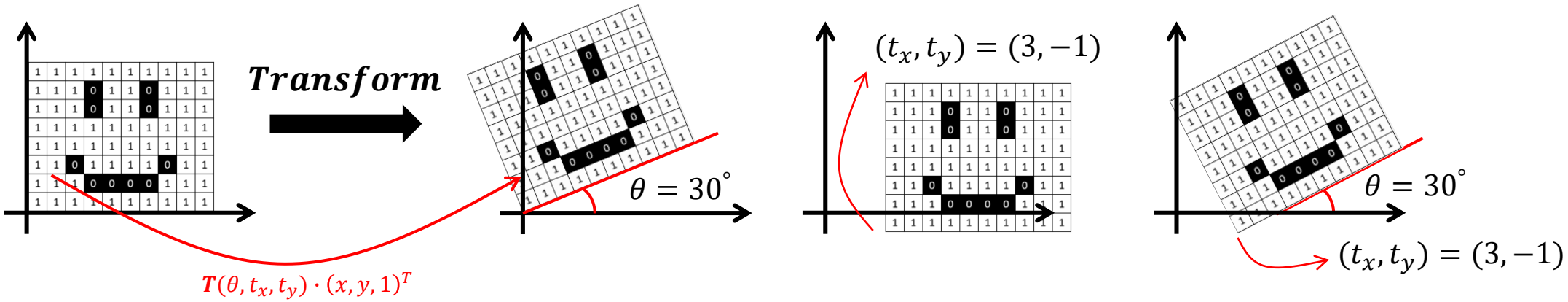
$$A \cdot (0,0)^T = (0,0)^T \quad (A \in R^{2\times 2})$$

1 - Homogeneous coordinate system

Homogeneous coordinate system is the solution!

$$\begin{pmatrix} x \\ y \\ \boxed{1} \end{pmatrix} \quad \text{Always keep 1 here!} \quad \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{14} \\ \boxed{0} & \boxed{0} & \boxed{1} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} \cdot x + a_{12} \cdot y + a_{13} \\ a_{21} \cdot x + a_{22} \cdot y + a_{23} \\ \boxed{0 \cdot x + 0 \cdot y + 1} \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ \boxed{1} \end{pmatrix}$$

Rotation and Translation of the image can be expressed with matrix multiplication!



$\boldsymbol{Transform}$: *for all combination* (x, y):

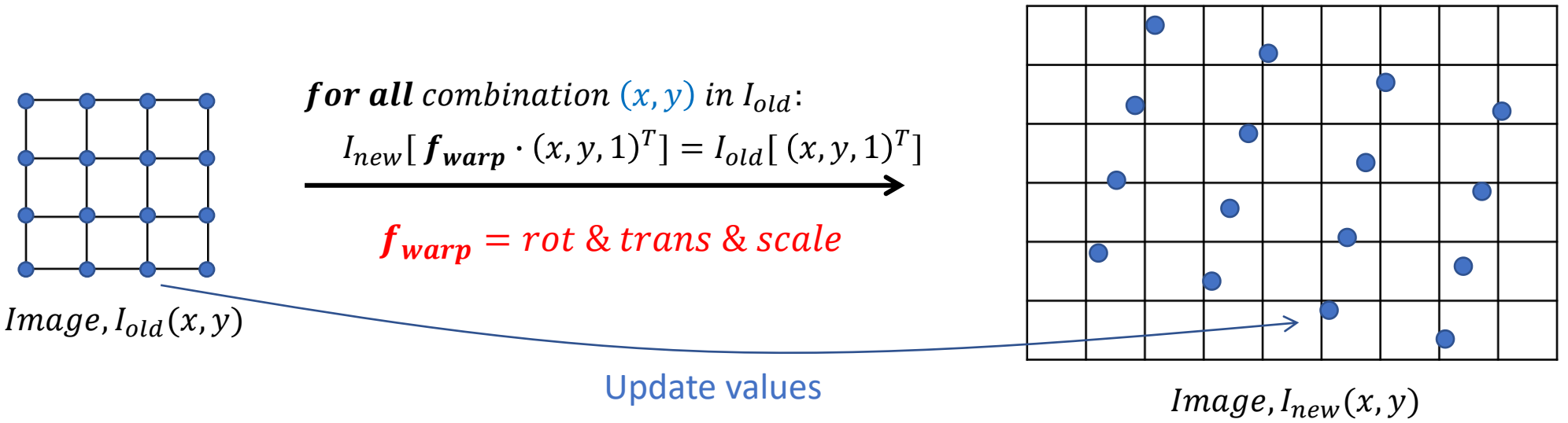
$$Image_{new}[\,\boldsymbol{T}(\theta, t_x, t_y) \cdot (x, y, 1)^T] = Image_{old}[\,(x, y, 1)^T]$$

$$\boldsymbol{T}(\theta, t_x, t_y) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & t_x \\ \sin(\theta) & \cos(\theta) & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

2 - (inverse) Warping and interpolation

Warping : The process of digitally manipulating an image **(Wikipedia)**
(i.e. Rotation / translation of the image, stretching / shrinking image … etc)

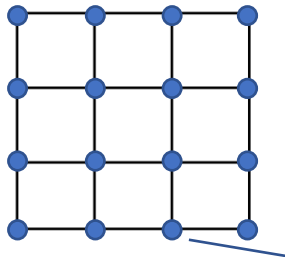
Example : Combination of image stretching and rotation / translation



Oh wait! 1) We have so many holes in the new image..!! → Inverse warping
2) Coordinates are in fractional value ..!! → Interpolation

2 - (inverse) Warping and interpolation

Forward warpping :



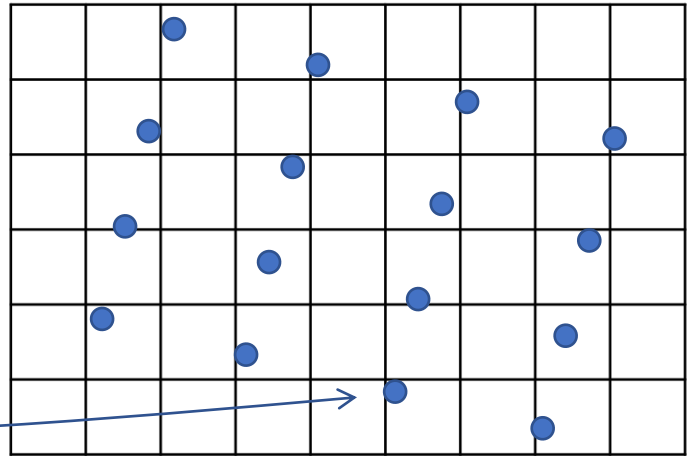
$Image, I_{old}(x, y)$

for all *combination* (x, y) *in* I_{old}:

$$I_{new}[\, \boldsymbol{f_{warp}} \cdot (x, y, 1)^T] = I_{old}[\, (x, y, 1)^T]$$

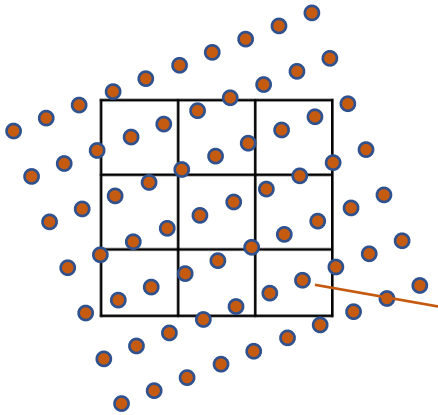
$$\boldsymbol{f_{warp}} = rot \;\&\; trans \;\&\; scale$$

Update values



$Image, I_{new}(x', y')$

Inverse warpping :



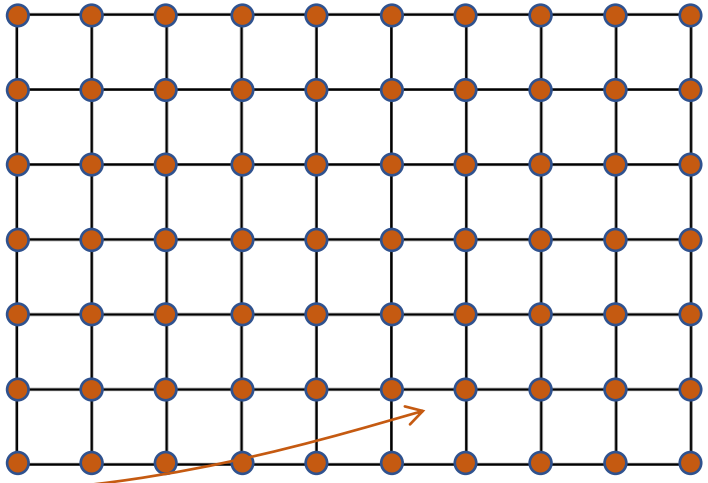
$Image, I_{old}(x, y)$

for all *combination* (x', y') *in* I_{new}:

$$I_{new}[(x', y', 1)^T] = I_{old}[\, \boldsymbol{f_{warp}^{-1}}(x', y', 1)^T]$$

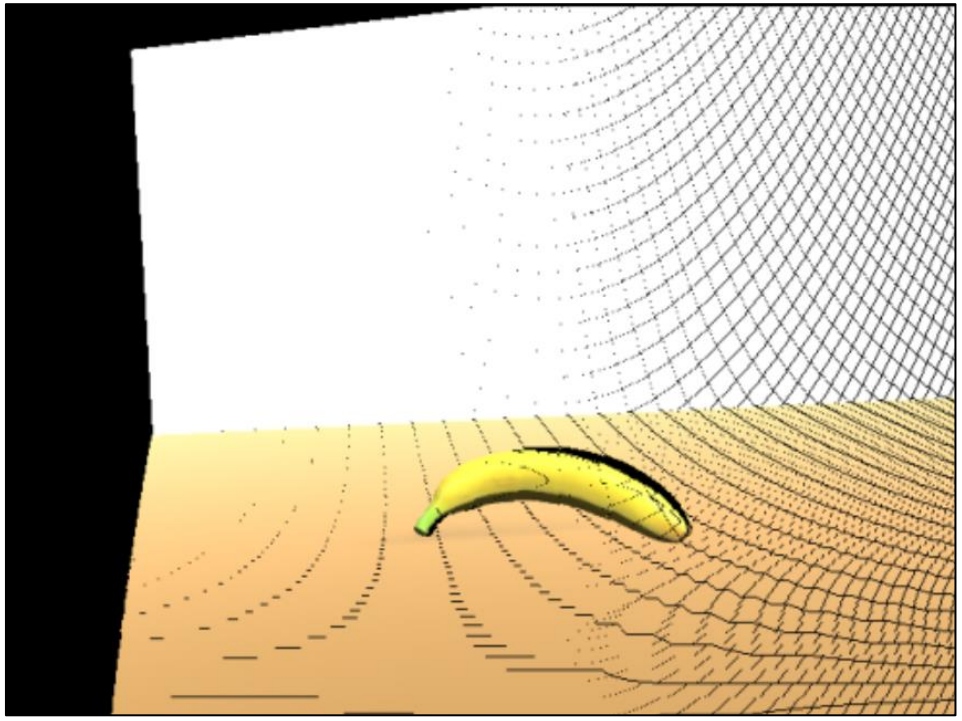
$$\boldsymbol{f_{warp}^{-1}} = rot^{-1} \;\&\; trans^{-1} \;\&\; scale^{-1}$$

Update values

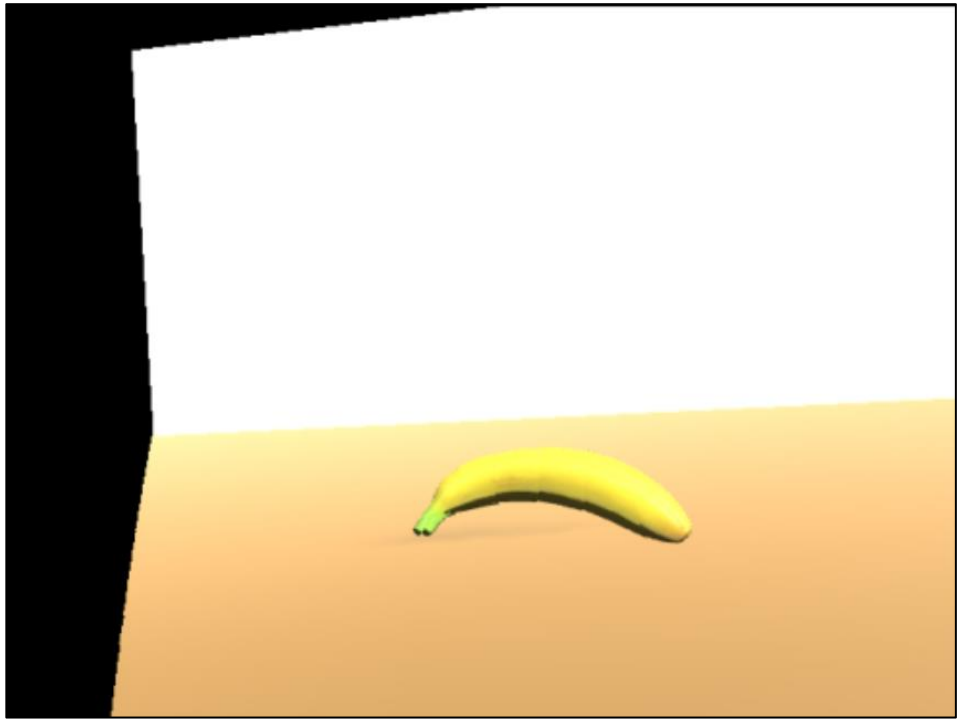


$Image, I_{new}(x', y')$

2 - (inverse) Warping and interpolation



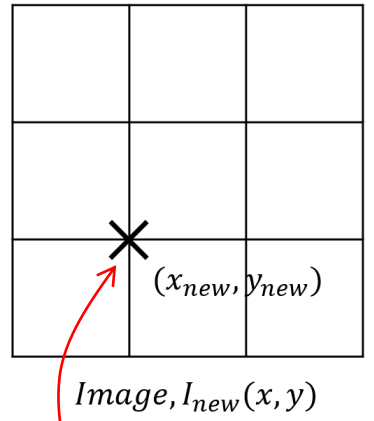
Example after forward warping



Example after inverse warping

2 - (inverse) Warping and interpolation

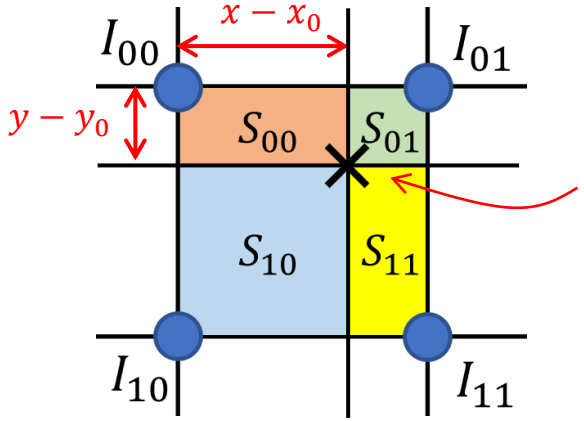
Interpolation : Interpolation is a type of estimation, a method of constructing new data points within the range of a discrete set of know data points. **(Wikipedia)**



Nearest neighbor interpolation :

$$I_{new}\,[\,(x_{new}, y_{new}, 1)^T] = I_{old}[\,(round(\boldsymbol{x}), round(\boldsymbol{y}), 1)^T]$$
$$= I_{old}[\,(\ x_1,\ y_0, 1\ )^T]$$

Bilinear interpolation :

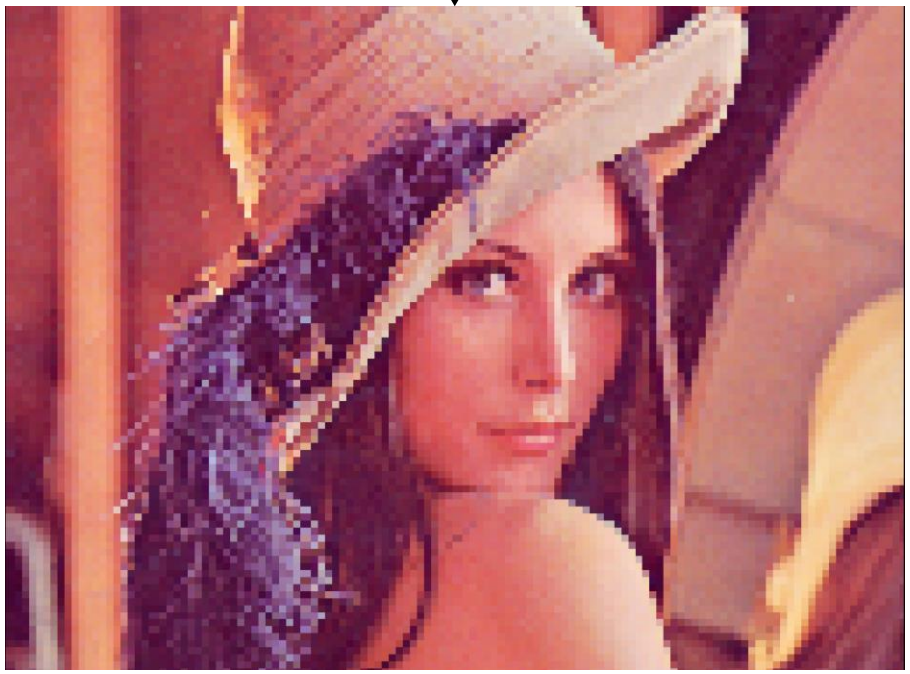


$$I_{new}\,[\,(x_{new}, y_{new}, 1)^T] = S_{00} \cdot I_{11} + S_{01} \cdot I_{10}$$
$$+ S_{10} \cdot I_{01} + S_{11} \cdot I_{00}$$

2 - (inverse) Warping and interpolation

Original image

Upscale x 2 with
Nearest Neighbor interpolation



Upscale x 2 with
Bilinear interpolation

3 – 6D pose for 3D world

Same homogeneous coordinate rule can be applied to the 3D world!

$$\begin{pmatrix} x \\ y \\ z \\ \boxed{1} \end{pmatrix} \quad \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} \cdot x + a_{12} \cdot y + a_{13} \cdot z + a_{14} \\ a_{21} \cdot x + a_{22} \cdot y + a_{23} \cdot z + a_{24} \\ a_{31} \cdot x + a_{32} \cdot y + a_{33} \cdot z + a_{34} \\ \boxed{0 \cdot x + 0 \cdot y + 0 \cdot z + 1} \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ \boxed{1} \end{pmatrix}$$

Always keep 1 here!

Rotation and Translation of 3D points can be expressed with matrix multiplication!

$$\begin{pmatrix} R & t \\ 0 \; 0 \; 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

$T(R \mid t) \cdot (x, y, z, 1)^T = (x', y', z', 1)^T$

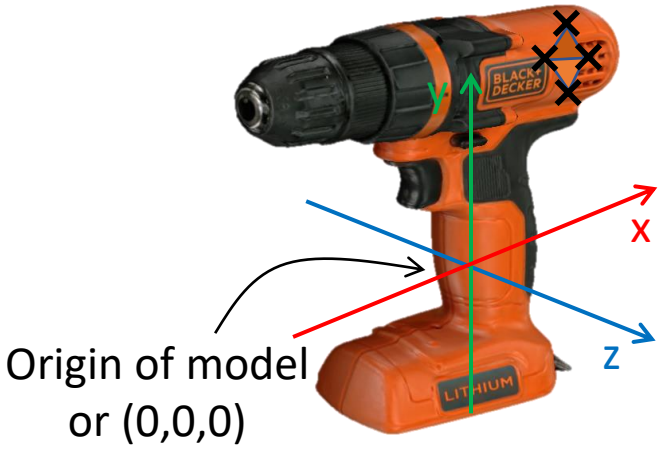
$\boldsymbol{R}$: (3x3) Rotational matrix. Often parameterized as combination of three rotational matrix (i.e. $R = rot_x(\theta_x) \cdot rot_y(\theta_y) \cdot rot_x(\theta_z)$)

$\boldsymbol{t}$: (3x1) Translational vector. Parameterized as $[t_x, t_y, t_z]^T$

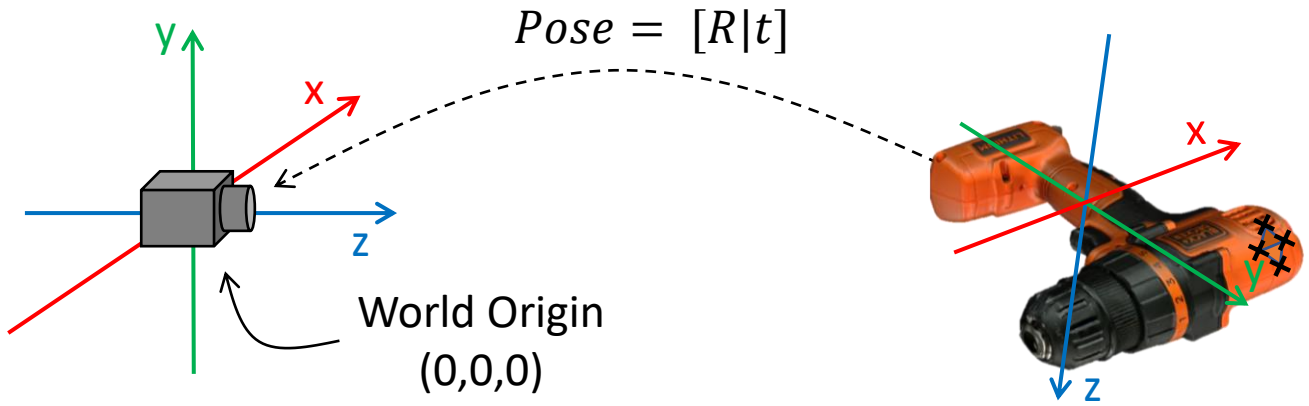
$\boldsymbol{T(R \mid t)}$: function of $(\theta_x, \theta_y, \theta_z, t_x, t_y, t_z)$ → **6 DoF**

3 – 6D pose for 3D world

Pose of 3D model in the world can be expressed as this Single 4x4 matrix (6D pose)



3D model of the object
is defined as list of 3D points
and meshes from its origin



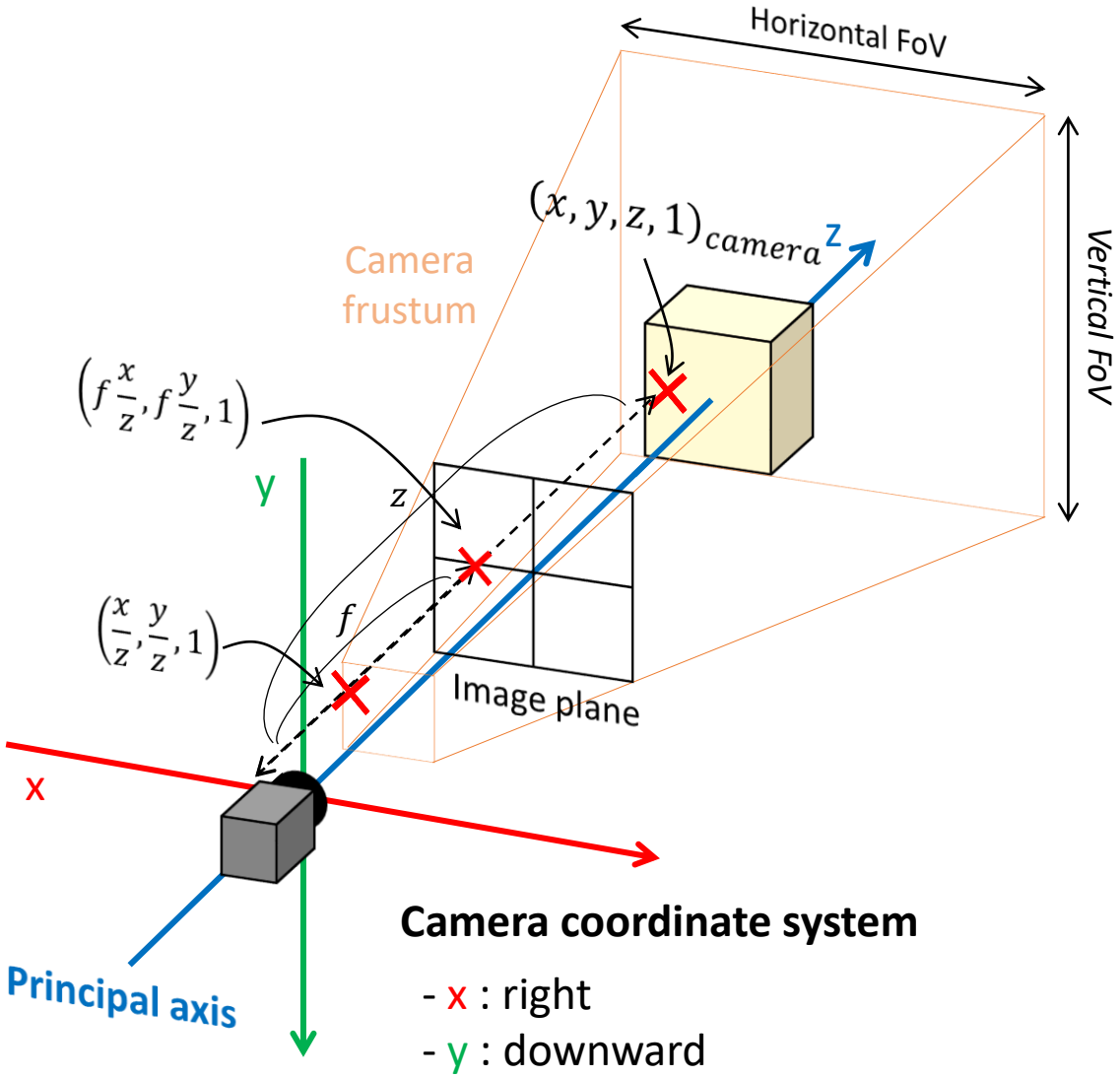
Pose of 3D model is defined as coordinate system transformation $[R|t]$ between **object's origin to world or camera's origin ($T_{obj \to world}$)**

$[R|t]$: **6 DoF → 6D Pose**

3D Points of the model (point cloud) can be expressed from world's origin by multiplying this (4x4) matrix on each point

$$[R|t]_{obj \to world} \cdot (x, y, z, 1)_{obj}{}^{T} = (x, y, z, 1)_{world}{}^{T}$$

4 - Camera model



Camera coordinate system

- x : right
- y : downward
- z : forward **(principal axis)**

How the image is formed ?

→ By projecting 3D points of the objetcs onto the image plane

What is the image plane?

→ Image plane is the plane within the camera frustum, at distance f (focal length) from the camera center

How does the projection work?

1. Remove homogeneous point and divide by z

$$X'_{camera} = \frac{1}{z} \cdot (x, y, z)^T = \left(\frac{x}{z}, \frac{y}{z}, 1\right)^T$$

Use as 2D Homogeneous coordinate

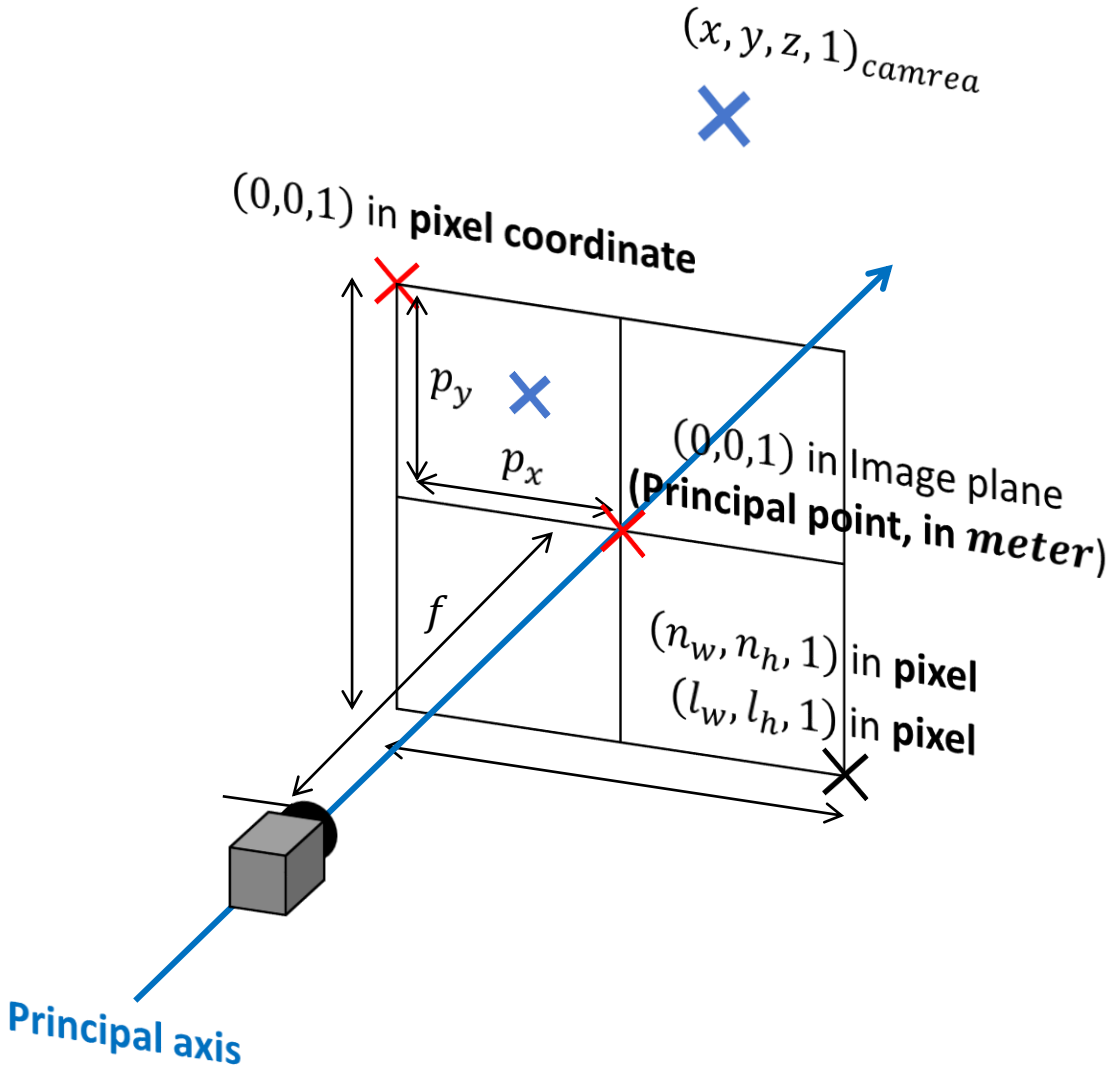
2. Scale up to the image plane, except for homogeneous point

$$X_{proj} = \left(f \cdot \frac{x}{z}, f \cdot \frac{y}{z}, 1\right)^T$$

How can we convert coordinates image plane's (meter) into a pixel coordinate of image array?

→ By using camera Intrinsic!

5.1 - Camera intrinsic



Camera intrinsic can convert coordinate of image plane (meter) into pixel coordinate of the image !

1) Shift by adding (p_x, p_y) for each point

2) Scale $\left(\frac{n_w}{l_w}, \frac{n_h}{l_h}\right)$ on each points in the object

All the processes, from projection to scaling, can be expressed as series of matrix multiplication

$$X_{image} = \begin{pmatrix} \frac{n_w}{l_w} & 0 & 0 \\ 0 & \frac{n_h}{l_h} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \frac{1}{z} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} (x, y, z, 1)^T$$

$$X_{image} = \begin{pmatrix} \alpha_x & 0 & u \\ 0 & \alpha_y & v \\ 0 & 0 & 1 \end{pmatrix} \cdot \frac{1}{z} \cdot [I|0] \cdot (x, y, z, 1)^T$$

Intrinsic matrix K

5.2 Camera calibration

Intrinsic matrix is unique for each camera that needs to be obtained before we perform any 3D vision task.

→ This process is called camera calibration

How the corresponding pixels the image are obtained from 3D points in the object :

$$(x', y', 1)^T_{image} = \underbrace{K \cdot \frac{1}{z_{cam}} \cdot [I|0] \cdot [R|t]_{obj \to cam}}_{\text{3x4 matrix } P \text{ with multiple unknowns}} \cdot \overbrace{(x, y, z, 1)^T_{obj}}^{(x, y, z, 1)^T_{camera}}$$

Each element of 3x4 matrix P is function of unknown elements from $[R|t], K$
(total 10 unknown – 4 from K, 6 from $[R|t]$)

$$(x', y', 1)^T_{image} = P \cdot (x, y, z, 1)^T_{obj}$$

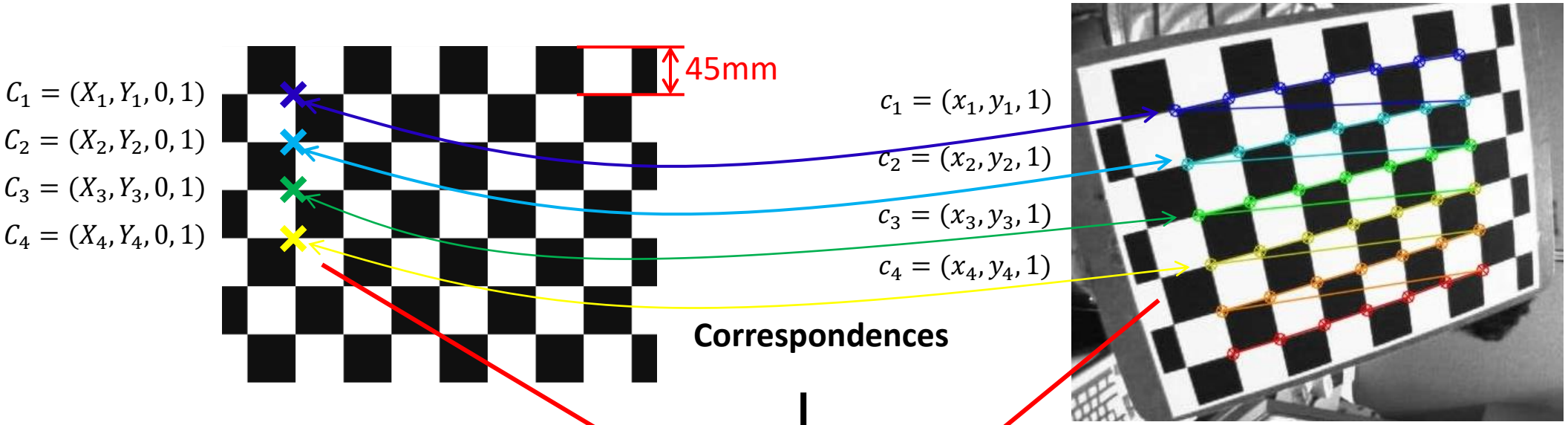
By obtaining point correspondence between the obj and the image, we will have multiple equations which can be used to optimize the unknowns in the functions in P.

→ Using known 3D object, find the multiple correspondence to solve unknowns in the functions in P!

5.2 Camera calibration

We use checkerboard for the calibration.

- Dimension of checkerboard can be predefined (known)
- Each conner makes it easier to find the accurate point between the image and object



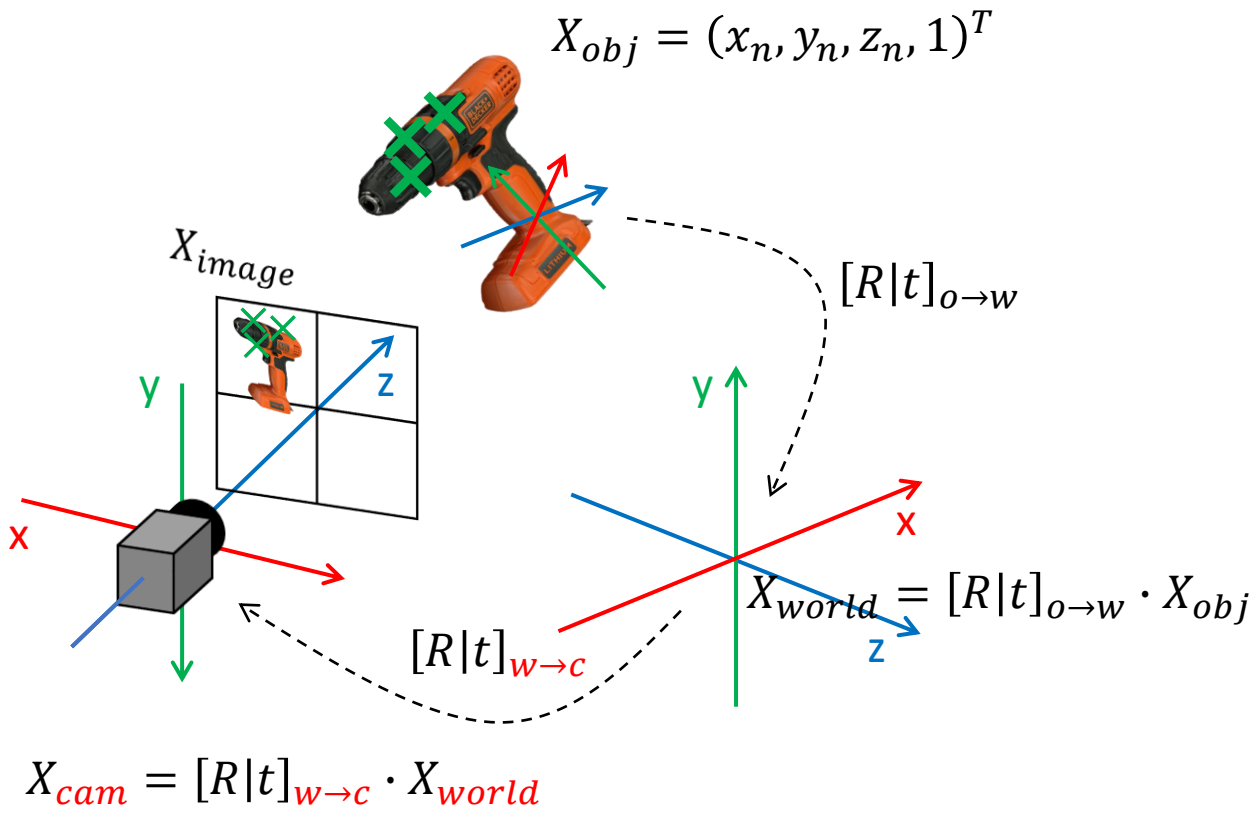
$$P \cdot (x, y, z, 1)^T_{obj} = (x', y', 1)^T_{image}$$

$$Optimize\big((C_1, c_1), (C_2, c_2), (C_3, c_3) \ldots\big) = [R|t], K$$

5.3 - Camera extrinsic

Sometimes camera's coordinate system is not aligned to the world's coordinate system !

→ We can use camera extrinsic to align it!



Camera extrinsic $\boldsymbol{E}$ is just 6D pose of the camera.
(It is Identity if camera is aligned to the world)

$$\boldsymbol{E} = [R|t]_{world\to camera}$$

$\boldsymbol{E}$ has to be multiplied before the projection

$$X_{cam} = \boldsymbol{E} \cdot X_{world}$$

Then we repeat the same process again :)

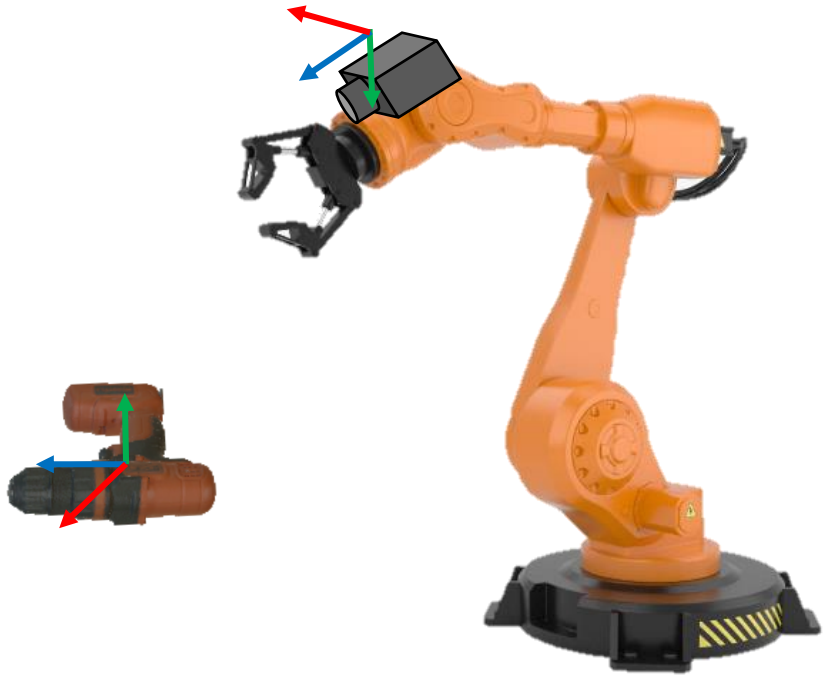
$$X_{image} = K \cdot \frac{1}{z} \cdot [I|0] \cdot \boldsymbol{E} \cdot X_{world}$$

Note, if the pose of camera is recorded as $[R|t]_{camera\to world}$, it has to be inverted ! (assignment 2)

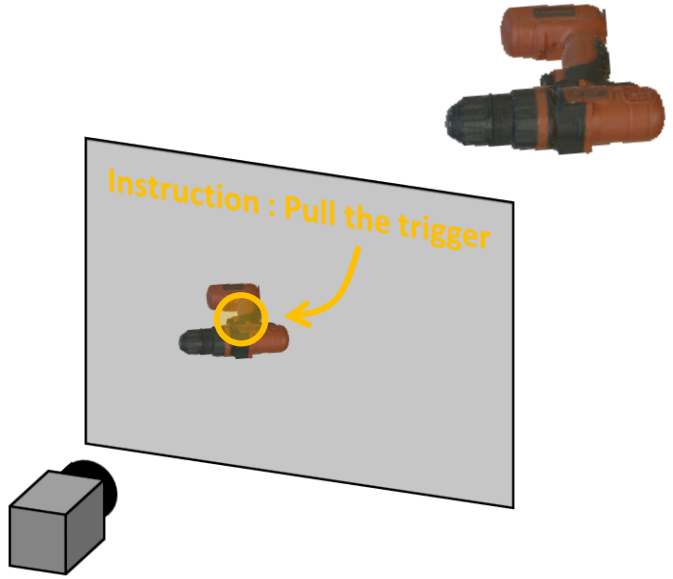
2. Practical uses

1. Rendering 3D object for rendering based 6D Pose tracking pipeline

Motivation :



< Robot operation >

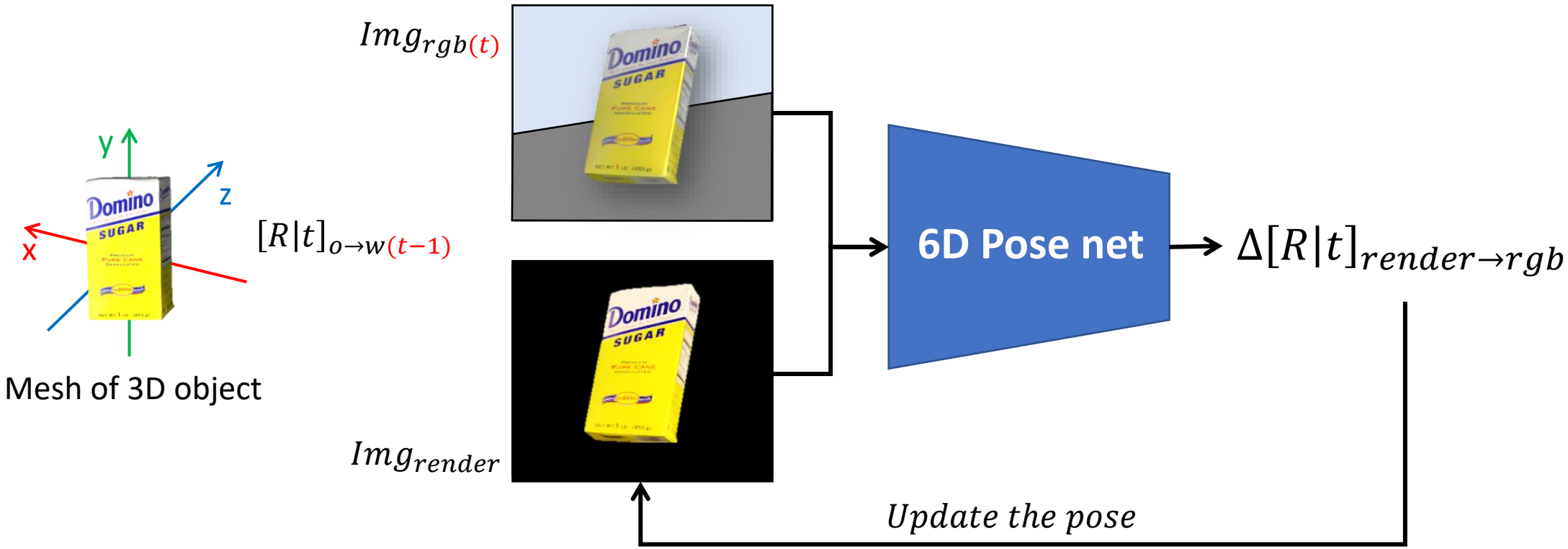


< Augmented reality >

1. Initial frame : Estimate initial 6D pose of given object via some pipeline (6D pose detection : Heavy)
2. From second frame : Reuse previous frame's pose to obtain current frame's pose (6D pose tracking : Light)

1. Rendering 3D object for rendering based 6D Pose tracking pipeline

Application : DeepIM – Deep Iterative Matching for 6D Pose Estimation



Your task for assignment 1 : Rendering the object correctly with given pose!

DeepIM : Deep Iterative Matching for 6D Pose Estimation (Y.Li, G.Wang, X,Ju, Y.Xiang, D.Fox, ECCV 2018)

1.1 Pyrender : Easy rendering library for python

Pyrender Documentation

Pyrender is a pure Python (2.7, 3.4, 3.5, 3.6) library for physically-based rendering and visualization. It is designed to meet the glTF 2.0 specification from Khronos

Pyrender is lightweight, easy to install, and simple to use. It comes packaged with both an intuitive scene viewer and a headache-free offscreen renderer with support for GPU-accelerated rendering on headless servers, which makes it perfect for machine learning applications. Check out the User Guide for a full tutorial, or fork me on Github.

```
scene = pyrender.Scene()
r = pyrender.OffscreenRenderer(400, 400)
```
Setup scene (world)

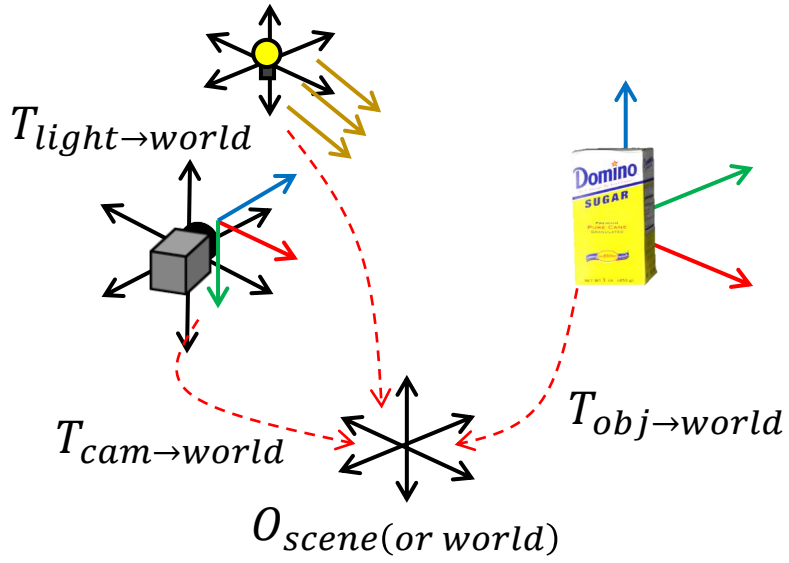
Read mesh and set pose $T_{obj \to world}$

```
obj_mesh_trimesh = trimesh.load('examples/models/fuze.obj')
obj_mesh_pyrender = pyrender.Mesh.from_trimesh(obj_mesh_pyrender)
scene.add(obj_mesh_pyrender, pose = obj_pose)
```

```
camera = pyrender.IntrinsicsCamera(fx=K[0,0],fy=K[1,1],cx=K[0,2],cy=K[1,2],
                                   znear=0.001,zfar=3)

scene.add(camera, pose = camera_pose)
```
Set camera intrinsic and pose $T_{cam \to world}$

```
light = pyrender.Spotlight(color=np.ones(3), intensity=3,
                           innerConeAngle=np.pi/16.0,innerConeAngle=np.pi/16.0)
scene.add(light, pose = camera_pose)
```
Set light and pose $T_{light \to world}$

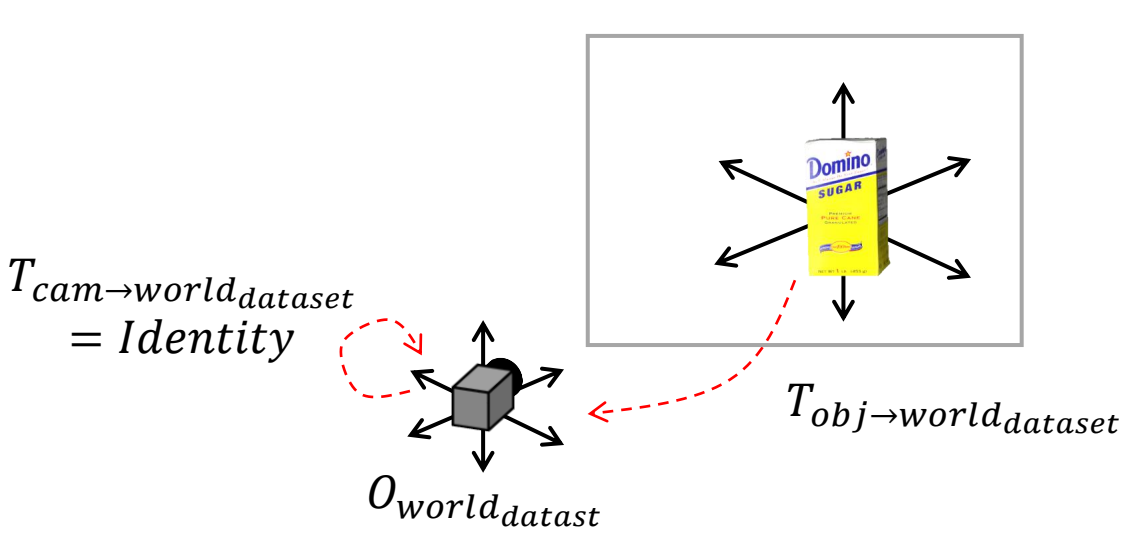
```
color, depth = r.render(scene)
```
Render the world seen from camera



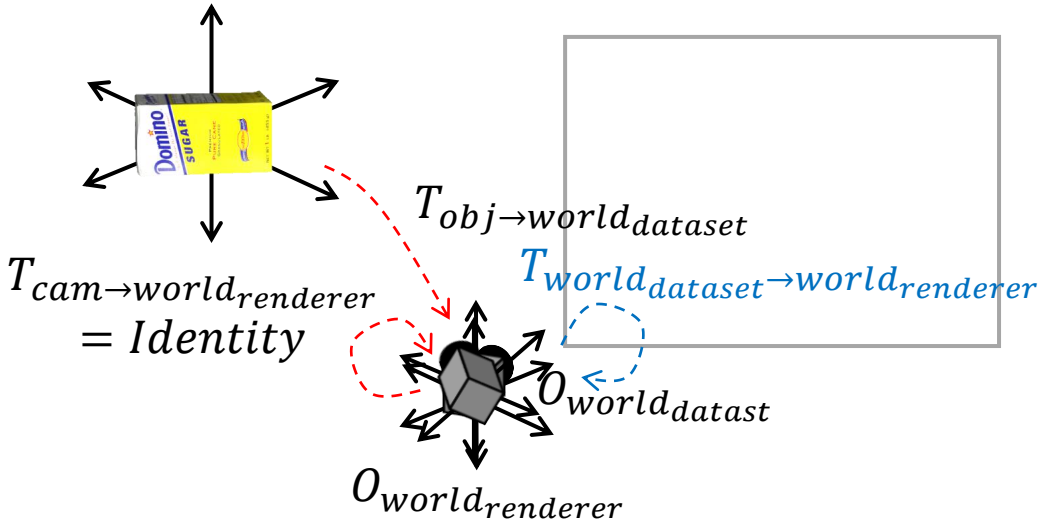
https://pyrender.readthedocs.io/en/latest/index.html

1.2 Dealing with unknown coordinate orientation

Issue : We have no idea about orientation of world for both Dataset and specific renderer
And it is possible that they are not aligned



< In 6D pose dataset >



< Renderer >

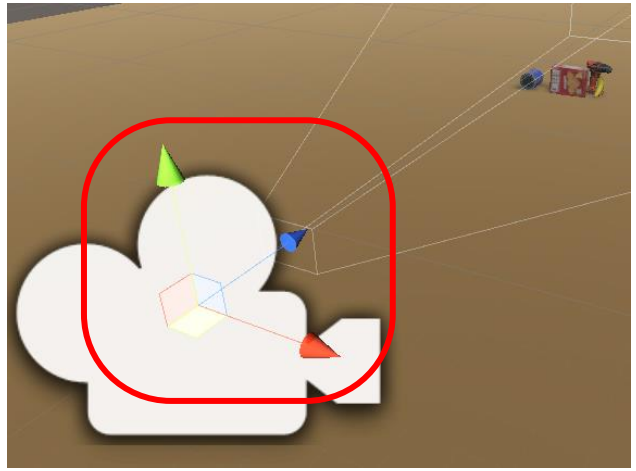
1. Find orientation of both dataset and renderer.
2. Find and multiply the conversion matrix $T_{world_{datast} \to world_{renderer}}$ on $T_{obj \to world_{datast}}$ to obtain $T_{obj \to world_{renderer}}$

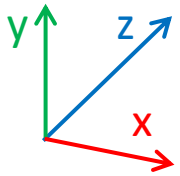
1.2 Dealing with unknown coordinate orientation

How to find the orientation of the world in the rendering API?

1) GUI based (i.e. unity)

→ Its there in the interface!





2) Script based (i.e. pyrender)

→ From its documentation or by educated guess!

1. Set $Pose_{cam} = [I|0]$

2. Find the forward direction (usually z is either forward/backward)

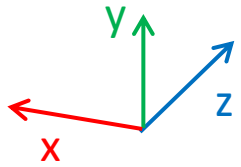
$t_{obj} = (0,0,z)$ 

$t_{obj} = (0,0,-z)$ 

3. Find the horizontal / vertical direction

$t_{obj} = (x,0,z)$ 

$t_{obj} = (0,y,z)$ 



1.2 Dealing with unknown coordinate orientation

How about 6D pose dataset (YCB-dataset)? What do we have?

RGB Image

Depth map (optional)

3D model

$$\begin{pmatrix} -3.03 \cdot 10^{-1} & 9.04 \cdot 10^{-2} & -7.03 \cdot 10^{-2} & -1.13 \cdot 10^{-1} \\ -3.61 \cdot 10^{-1} & -5.41 \cdot 10^{-2} & 9.30 \cdot 10^{-1} & 5.73 \cdot 10^{-2} \\ 8.72 \cdot 10^{-1} & -3.34 \cdot 10^{-1} & 3.57 \cdot 10^{-1} & 9.63 \cdot 10^{-1} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$Pose = [\, R \mid t \,]$

$$\begin{pmatrix} 1066.77 & 0 & 312.98 \\ 0 & 1067.48 & 241.31 \\ 0 & 0 & 1 \end{pmatrix}$$

$intrinsic$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$Extrinsic$

(Identity if not given)

1.2 Dealing with unknown coordinate orientation

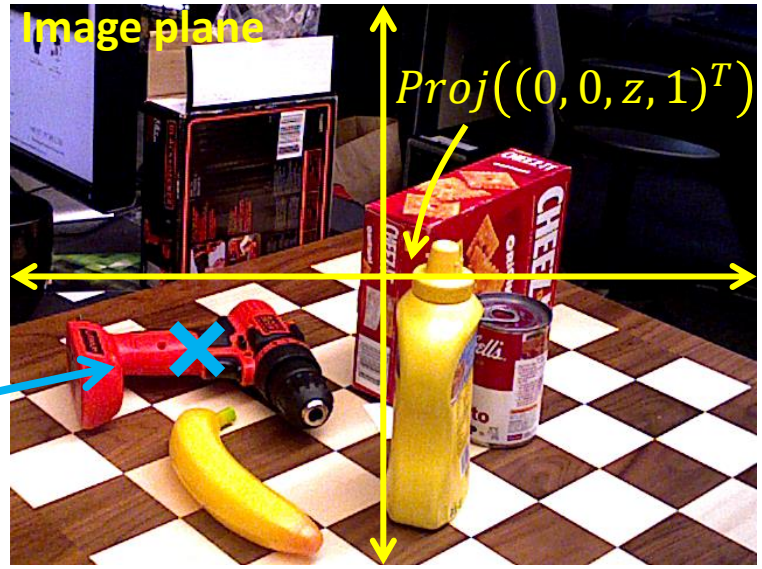
Hmm.. Again, orientation seems to be missing .. Time for another educated guess !

$$Pose_{driller} = [\, R \mid t \,]$$

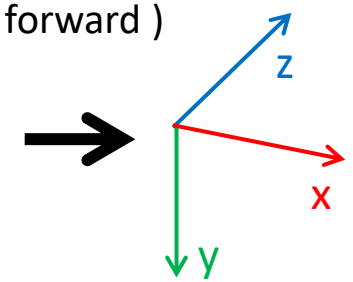
$$\begin{pmatrix} -3.03 \cdot 10^{-1} & 9.04 \cdot 10^{-2} & -7.03 \cdot 10^{-2} & -1.13 \cdot 10^{-1} \\ -3.61 \cdot 10^{-1} & -5.41 \cdot 10^{-2} & 9.30 \cdot 10^{-1} & 5.73 \cdot 10^{-2} \\ 8.72 \cdot 10^{-1} & -3.34 \cdot 10^{-1} & 3.57 \cdot 10^{-1} & 9.63 \cdot 10^{-1} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Center_{driller} = [-0.113 \quad 0.057 \quad 0.96 \quad 1]^T$$

(usually t_z is forward / backward direction)

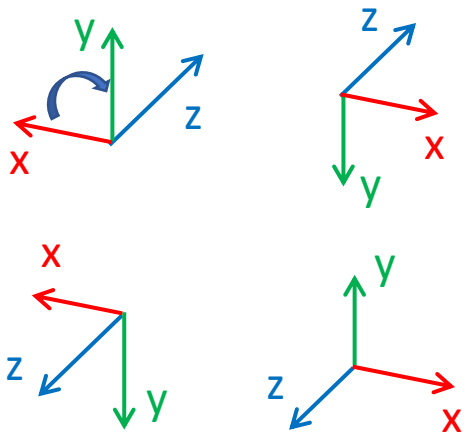


1. Positive z means in front of origin/camera (z = forward)
2. x has to be twice further from center than y
   (x = horizontal, y = vertical)
3. Negative x is left while positive y is down
   (x = right, y = down)

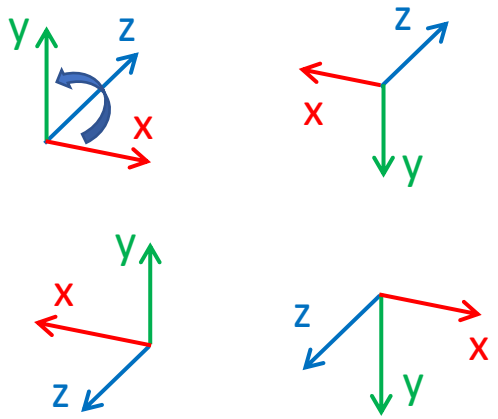


1.3 Conversion between different orientation

There are two classes of coordinate system : right-handed and left-handed



Possible right-handed coordinate system



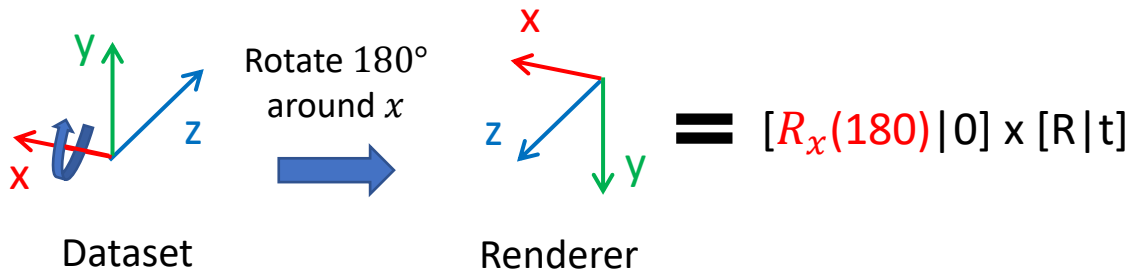
Possible left-handed coordinate system

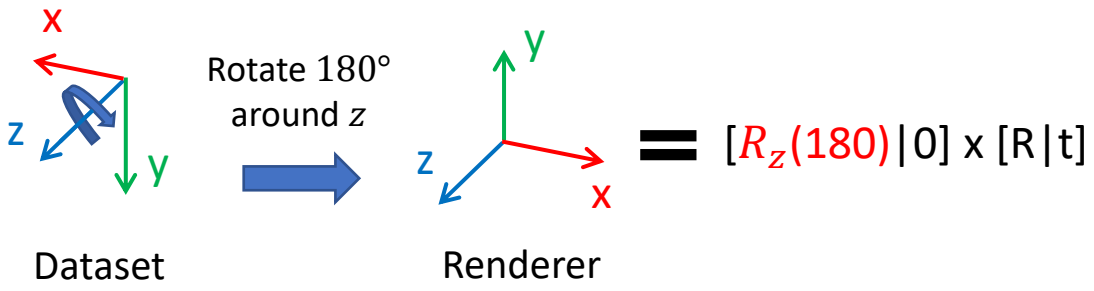
Conversion is different depending on whether the conversion is within the same handed system or between different handed system.

1.3 Conversion between different orientation

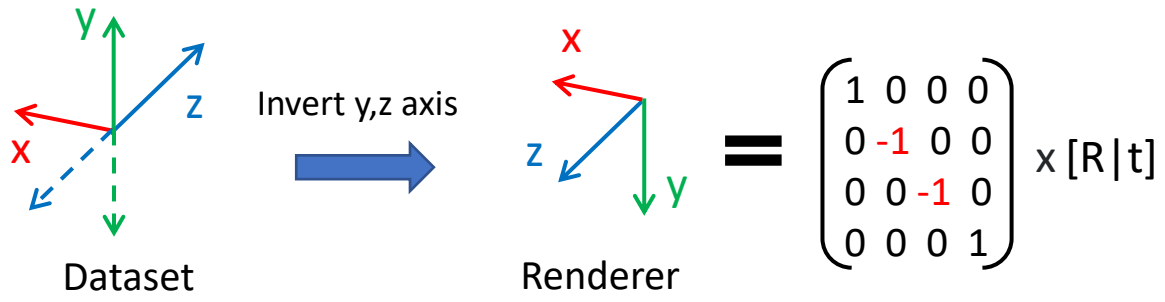
Conversion within same handed system (i.e. Right -> Right)

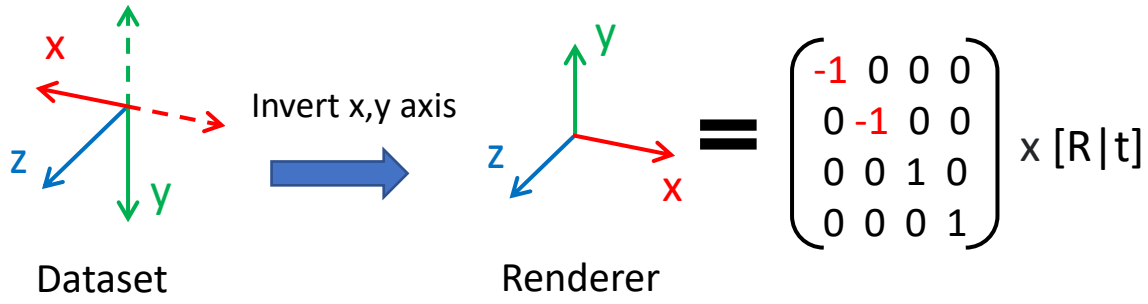
1) Rotational approach





2) Axis inverting approach



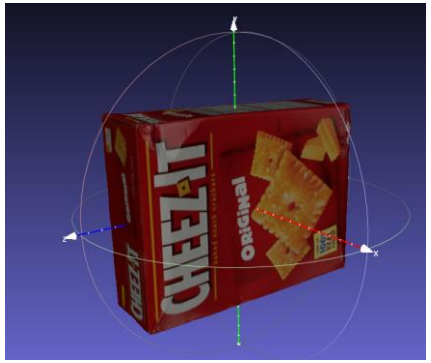


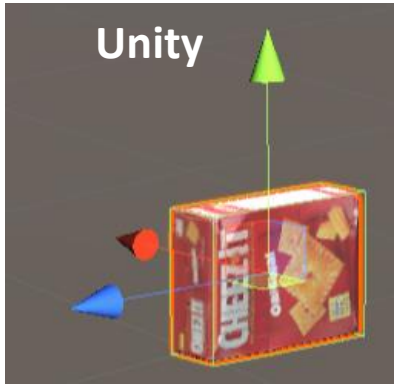
Negating corresponding rows is what we need !

1.3 Conversion between different orientation

Conversion between different handed system (i.e. Left <-> Right)

1) Certain axis of mesh is flipped between left-handed and right-handed system (i.e. Unity : x axis)





-> If the pose is recorded from flipped mesh (i.e. from left-handed system), the pose will work in properly in different coordinate system only if the mesh is flipped in the same way -> we need to flip mesh's coordinate

2) Conversion

Left-handed Dataset (i.e. from Unity)

Right-handed Renderer

Flipping orientation axis

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * [R|t] *$$

Flipping mesh axis

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \text{Mesh} =$$

Element wise multiplication

$$\begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{pmatrix} \circ [R|t] * \text{Mesh}$$

Assignment 1

Task 1. Find orientation of Pyrender and Augment YCB object properly.

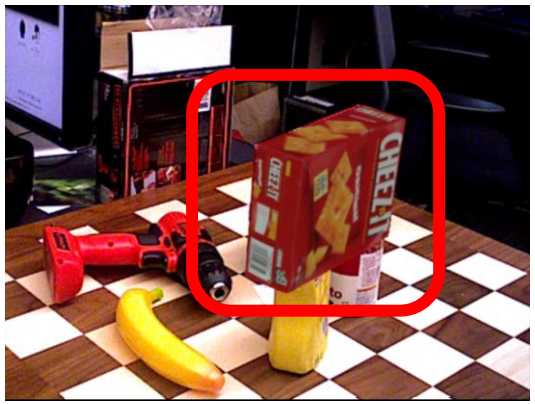
1. Check slide 25 to see how to find the orientation of the Pyrender (right)
2. Check slide 26 for YCB dataset orientation (right)
3. Check slide 29 for orientation conversion between right and right

Task 2. Find orientation of Laval dataset (right) and augment the object properly.

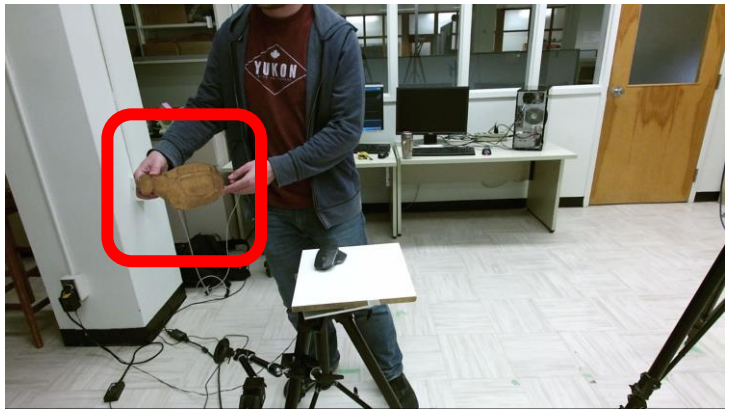
1. Check slide 27 to see how to find orientation of the unknown dataset
2. Check slide 29 for orientation conversion between right and right

Task 3. Try to convert Unity pose (left) into Pyrender pose and augment the object properly.

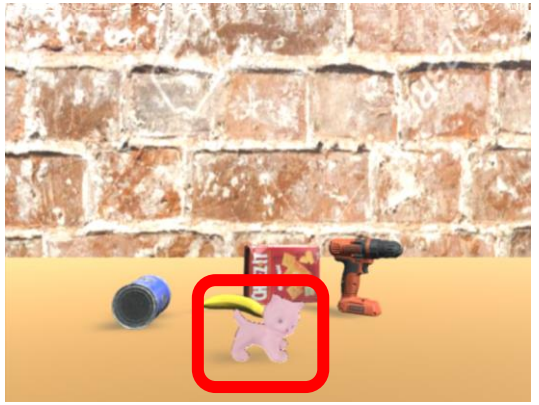
1. Unity is Left-handed system. check slide 30 to see how to deal with conversion btw left and right



Result from Task 1



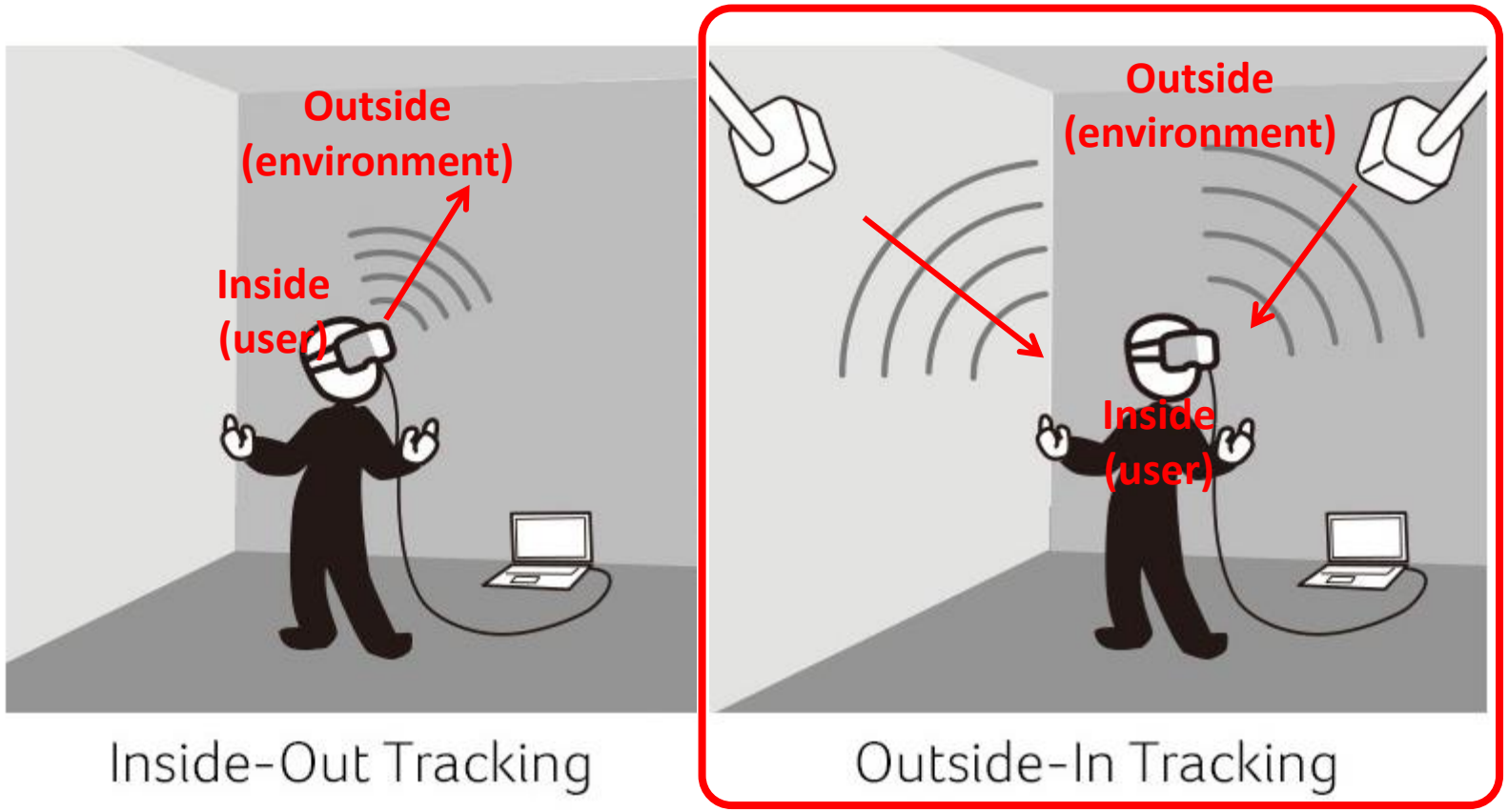
Result from Task 2



Result from Task 3

2. Augmented Reality for Outside-In tracking setup

Motivation



Advantage of Outside-In Tracking

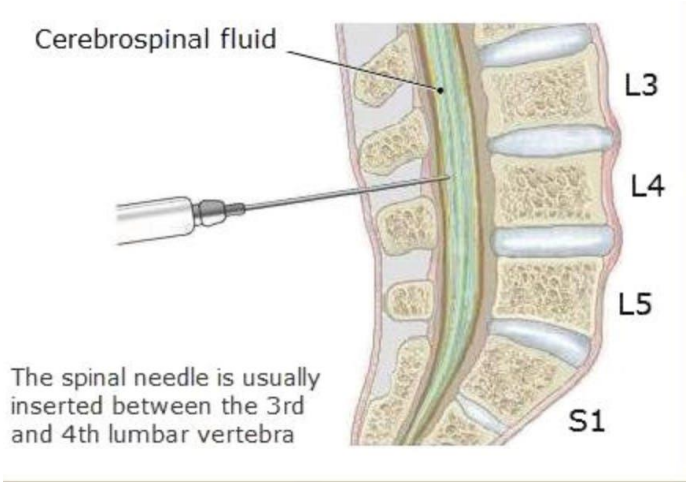
1. Better accuracy
2. Less latency
3. No line of sight problem of marker

→ **Often used in Surgical Scenario**

2. Augmented Reality for Outside-In tracking setup

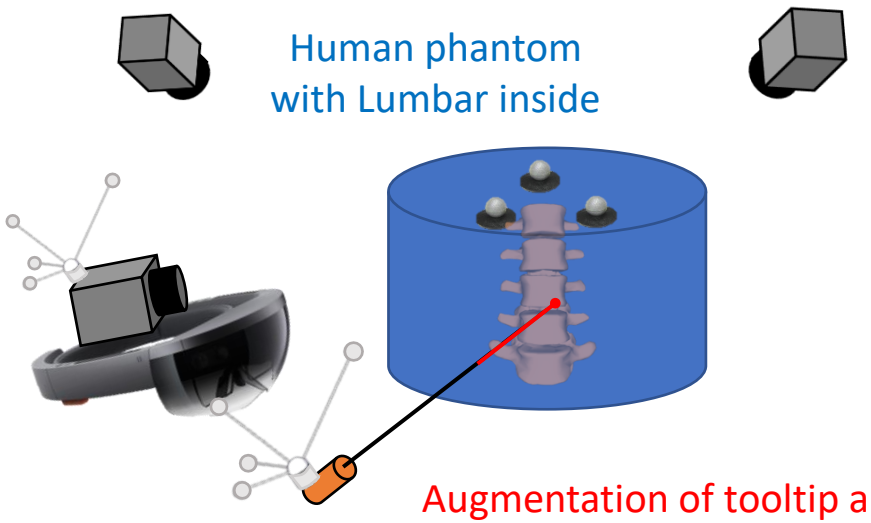
Application : AR simulator for lumbar puncture training

< Procedure >



Lumbar puncture is procedure to collect patient's Cerebrospinal fluid, by puncturing specific spot

< AR Implementation >

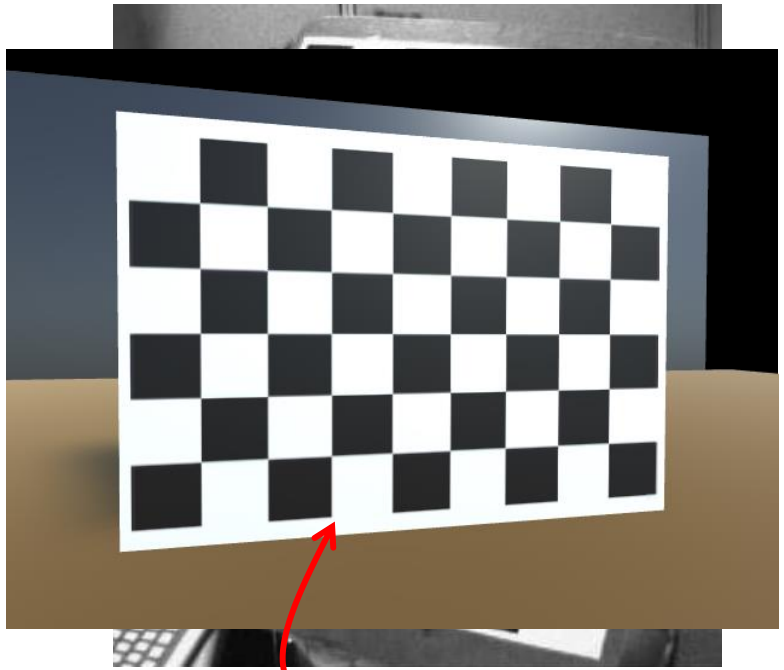


Augmentation of tooltip and Lumbar on user's camera (AR headset)

Your task for assignment 2 : Given multiple poses, augment tooltip & lumbar on user's camera

2.1 Camera calibration

```python
import numpy as np
import cv2
import glob

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*7,3), np.float32)
objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('*.jpg')

for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    # Find the chess board corners
    ret, corners = cv2.findChessboardCorners(gray, (7,6),None)

    # If found, add object points, image points (after refining them)
    if ret == True:
        objpoints.append(objp)

        corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgpoints.append(corners2)

        # Draw and display the corners
        img = cv2.drawChessboardCorners(img, (7,6), corners2,ret)
        cv2.imshow('img',img)
        cv2.waitKey(500)


ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1],None,None)
```

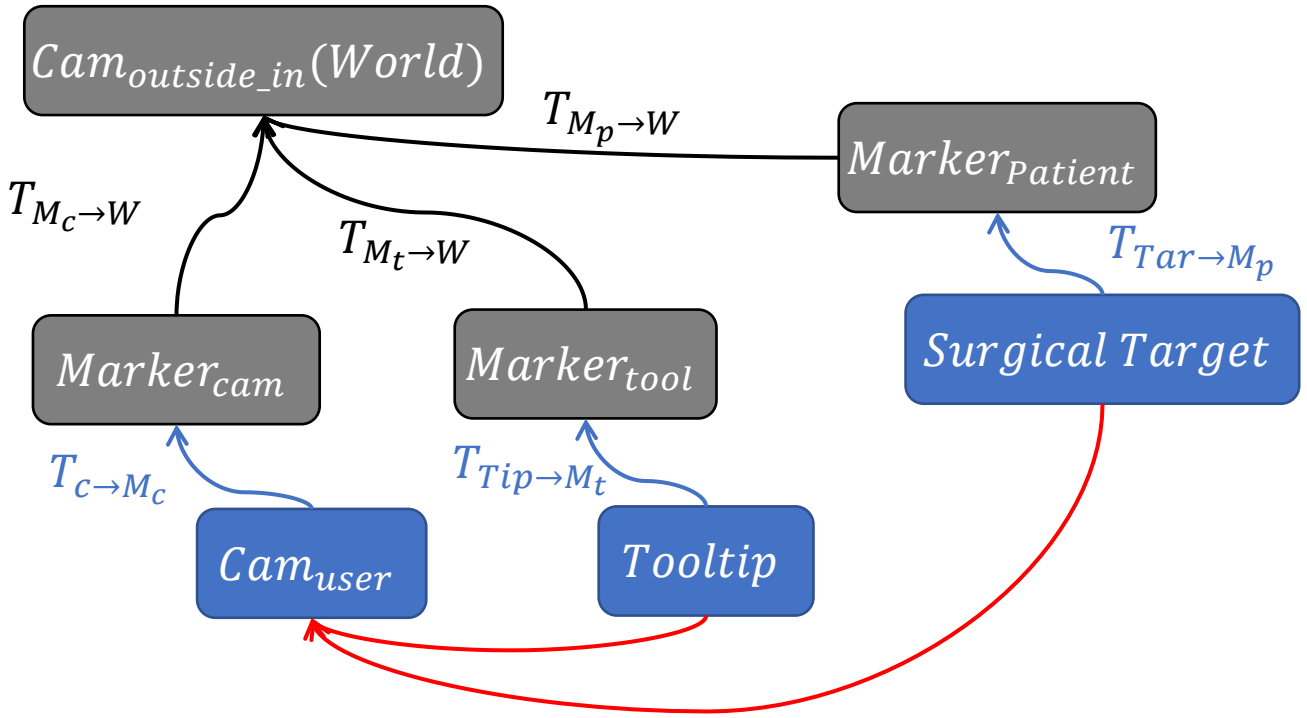


All you need to do is to count how many points should be extracted, and modify the code accordingly ;)

Image, code from https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html

2.2 Coordinate conversion

Relationship between poses of objects in given scene can be expressed as directional graph



- Each Node stands for the object in the scene
- Each edge stands for the relative transform matrix between nodes

  ($T_{a \to b}$: Transform matrix to express pose/point expressed in a's coord system into b's coord system)
- Traversing can be easily done by multiplying the transform matrix
- Reverse direction is simply inverse of the transform matrix

Example) Order: (4) (3) (2) (1)

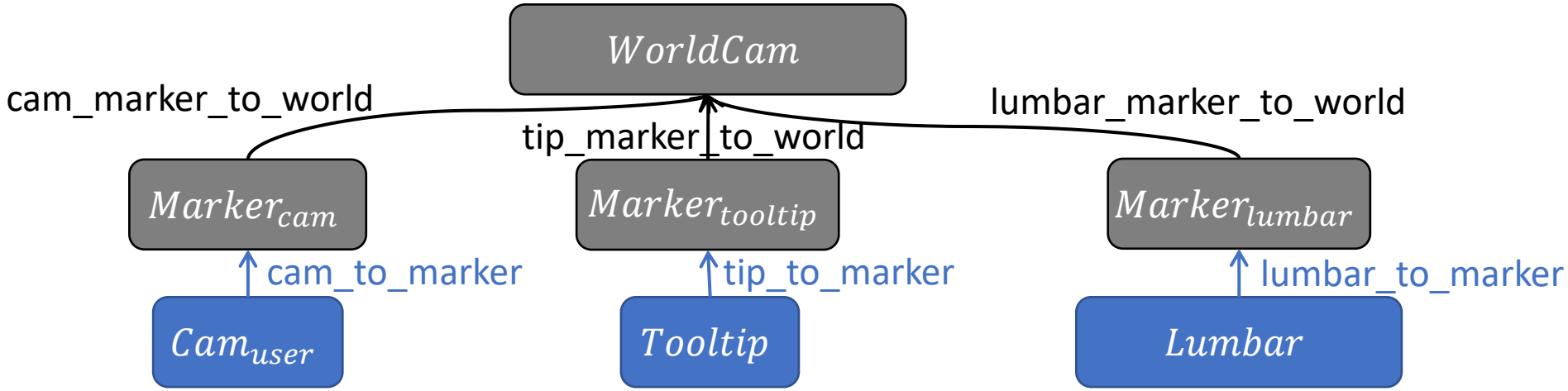
$$T_{Tip \to c} = \left(T_{c \to M_c}\right)^{-1} \cdot \left(T_{M_c \to W}\right)^{-1} \cdot T_{M_t \to W} \cdot T_{Tip \to M_t}$$: Tooltip location expressed in camera system

$$T_{Tar \to c} = \left(T_{c \to M_c}\right)^{-1} \cdot \left(T_{M_c \to W}\right)^{-1} \cdot T_{M_p \to W} \cdot T_{Tar \to M_p}$$: Pose of the target expressed in camera system

Assignment 2

Task 1. Cameras are not yet calibrated. Given checkerboard images taken from the user's camera, get camera intrinsic K (here we assume all cameras have same intrinsic)

1. Check slide 34 for details about camera calibration code.



Task 2. Given the scenario and relative pose graph, render the objects properly on world camera

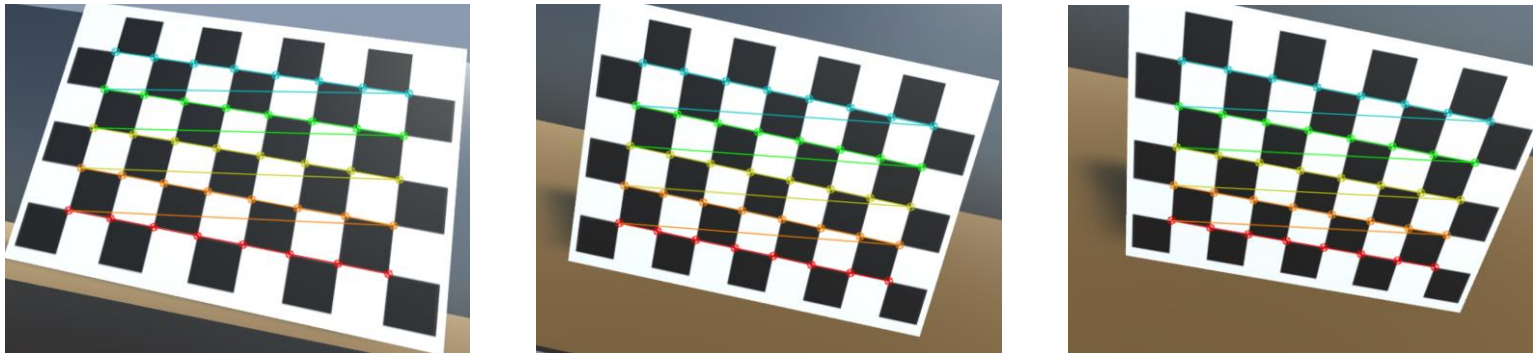
1. Try to augment tool tip and target on WorldCam using K from task1.

Task 3. With given pose graph, Now render the tool tip and target on user camera.

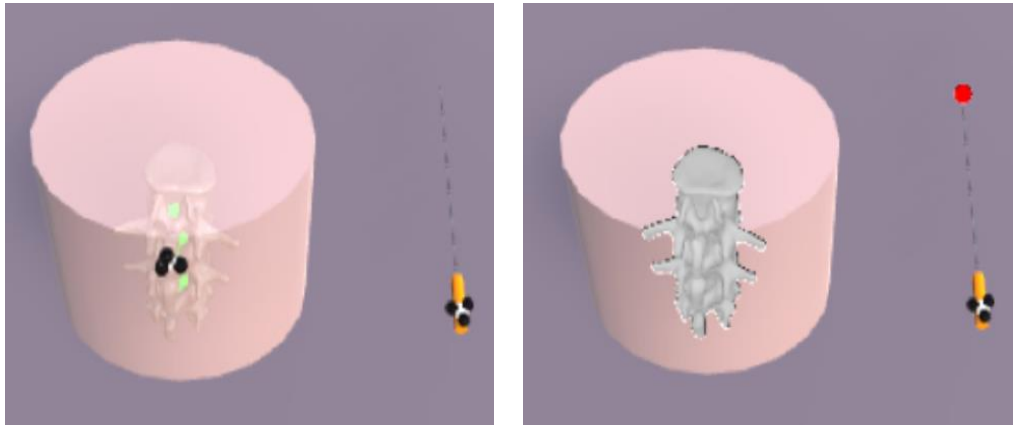
1. Details about traversing the graph is in slide 35
2. Try to augment the tip and target on Cam_{user}(use same K)

Assignment 2

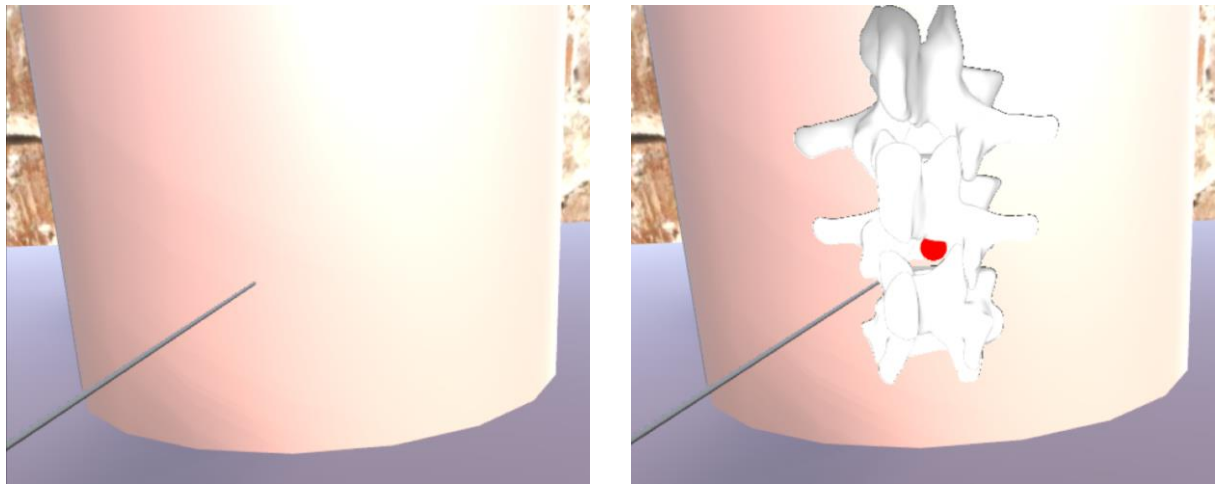
Example of correctly detected checkerboard (task1)



Augmented lumbar and tooltip



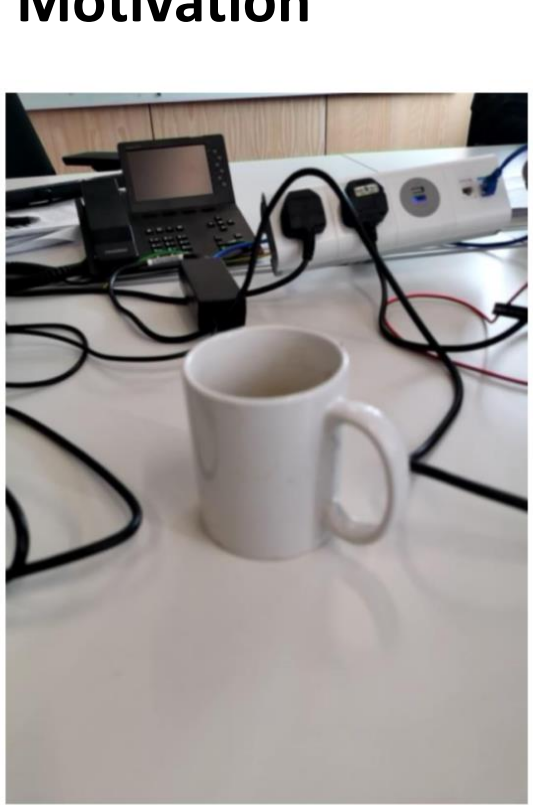
Task2 (seen from world camera)

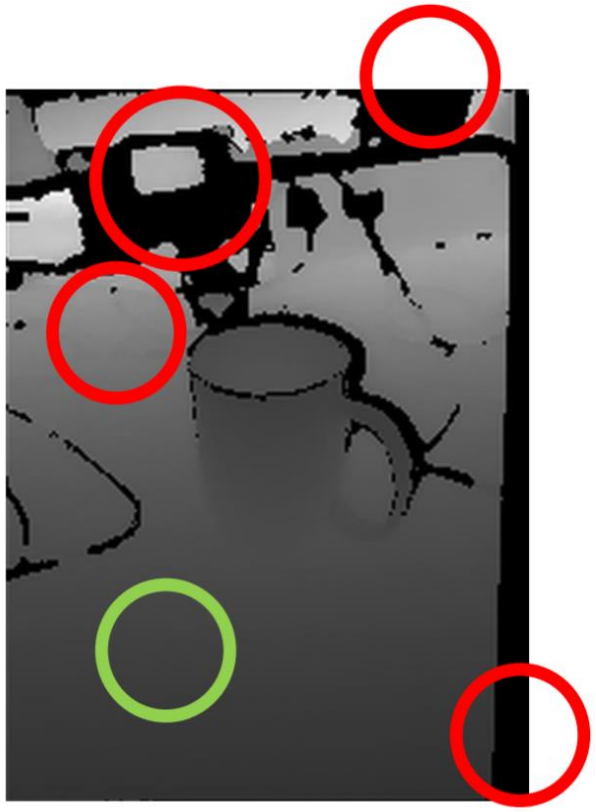


Task3 (seen from user camera)

3. Self supervised learning via warping

Motivation





Solution ?

1. Post process via human annotation
   -> Takes too much effort

2. Use synthetic data
   -> Hard to simulate real world noise

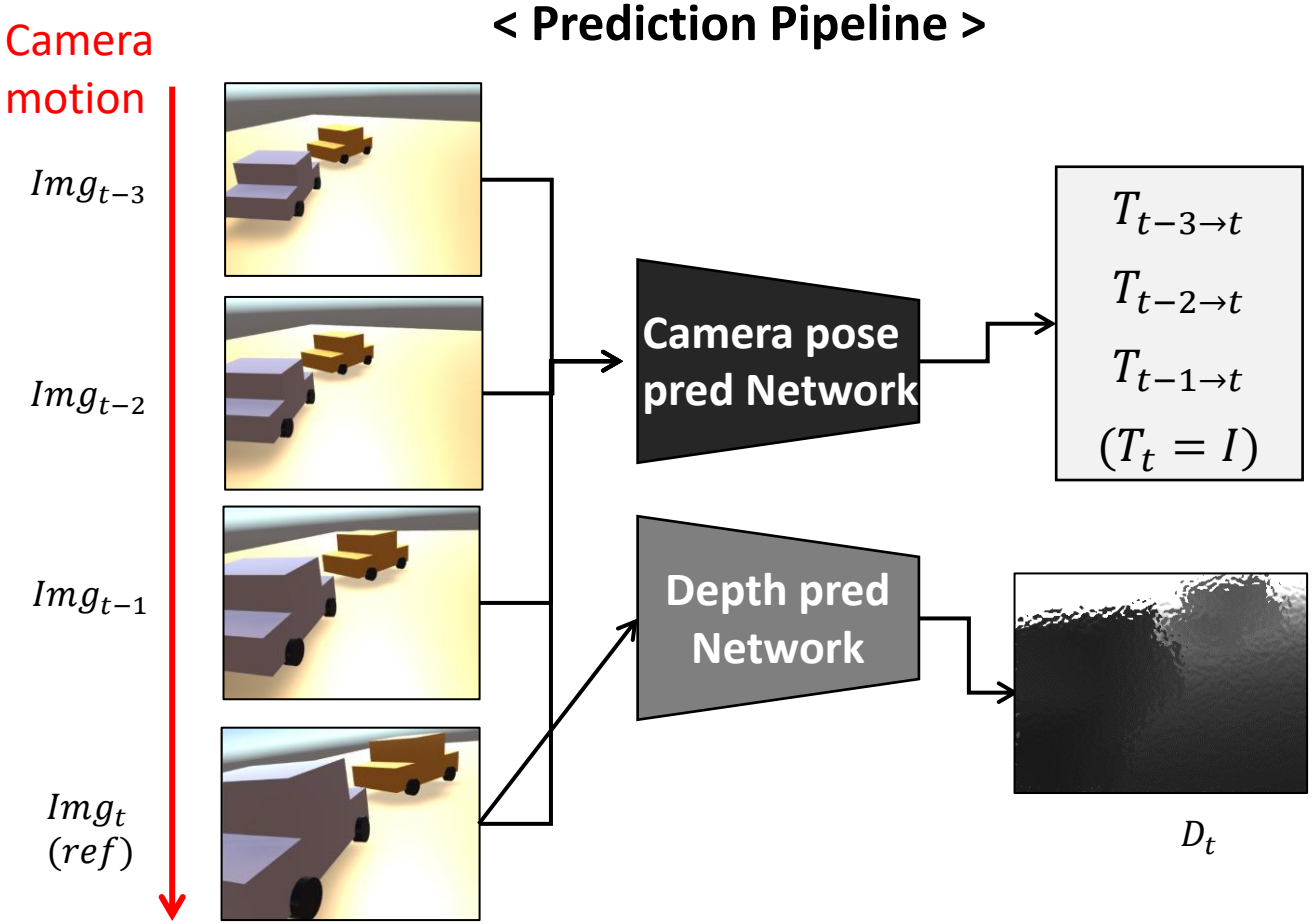
3. Use self-supervised learning
   -> Need extra information
   (but we may have them ☺!)

→ **When it comes to training the depth prediction network, quality of GT depth is often bad for loss calculation !**

3. Self supervised learning via warping

Application : Unsupervised Learning of Depth and Ego-Motion from Video

< Prediction Pipeline >

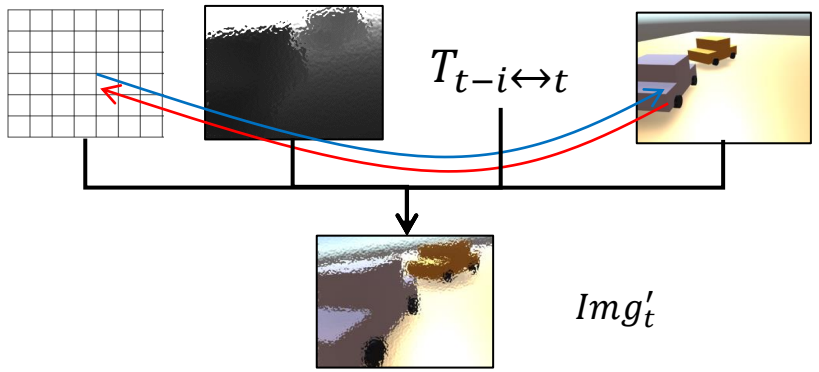


< Loss calculation Pipeline >

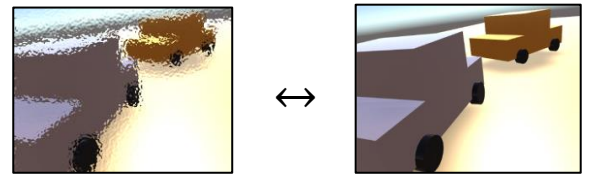
For each i in {1,2,3}

Assignment 3 !

1. $Warp^{-1}$ Img_{t-i} into Img_t, using $T_{t-i\leftrightarrow t}$ and D_t



2. Use difference btw Img'_t, Img_t as the loss

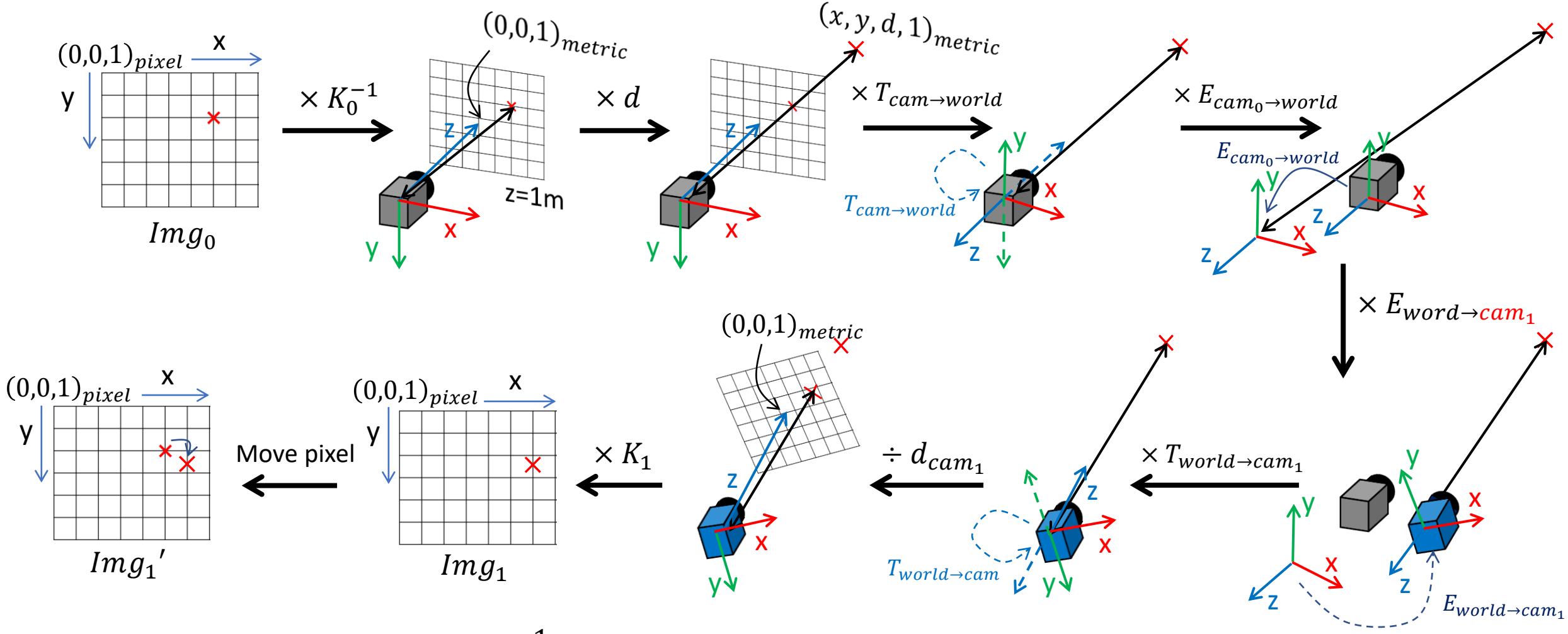


Your task for assignment 3 : Implement the Image (inverse) warping pipeline !

Unsupervised Learning of Depth and Ego-Motion from Video (T. Zhou, M. Brown, N. Snavely, D. Lowe, CVPR 2017)

3.1 Warping the image using Depth and Relative pose

How does the 3D warping between $Cam_0 \leftrightarrow Cam_1$ work?



$$[x,y,1]^T_{img1} = K_1 \cdot [I|0] \cdot (\frac{1}{d_{c1}}) \cdot T_{w\to c} \cdot E_{w\to c1} \cdot E_{c_0\to w} \cdot T_{c\to w} \cdot homo3D(d_{c0} \cdot K_0^{-1} \cdot [x,y,1]^T_{img0})$$

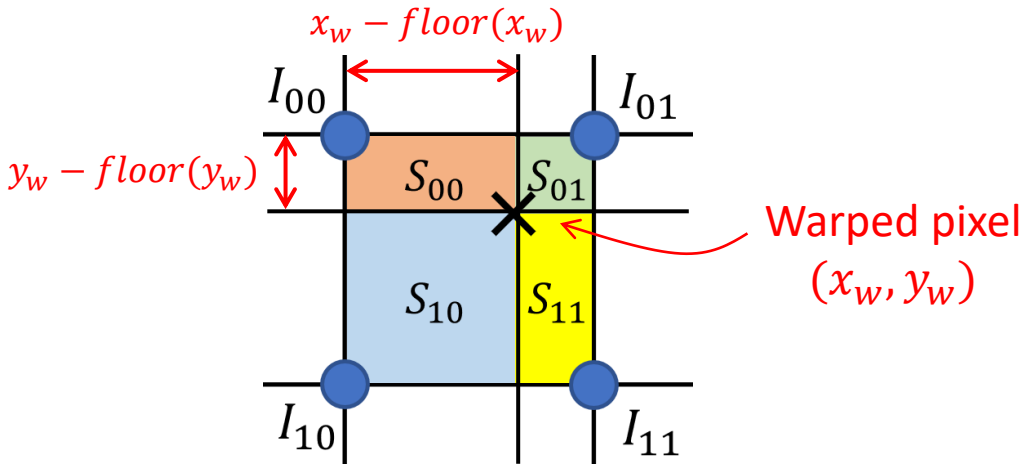
3.2 Differentiable bilinear interpolation

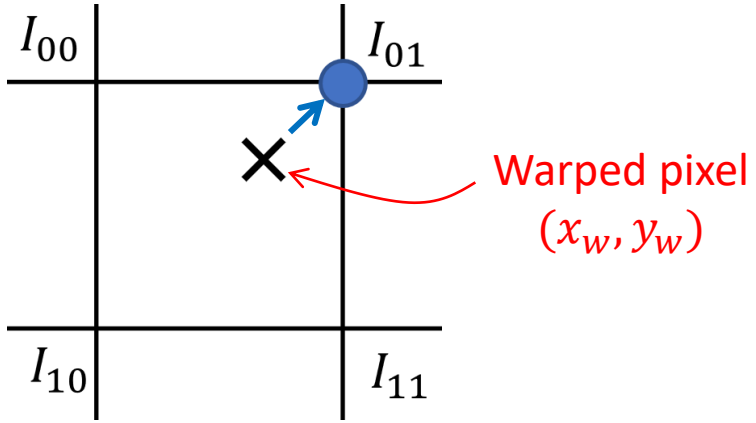
We need to use differentiable interpolation. Why?

- If the interpolation is not differentiable, the gradient from the loss will not be propagated to the network !

What is differentiable and not?

- Warped coordinates (x_w, y_w) are differentiable and so thus $S_{00,01,10,11}$, but pixel values $I_{00,01,10,11}$ aren't.





$Bilinear\ Interpolation:$

$$I_{new} = S_{00} \cdot I_{11} + S_{01} \cdot I_{10} + S_{10} \cdot I_{01} + S_{11} \cdot I_{00}$$

→ Differentiable !

$Nearest\ Neighbor\ Interpolation:$

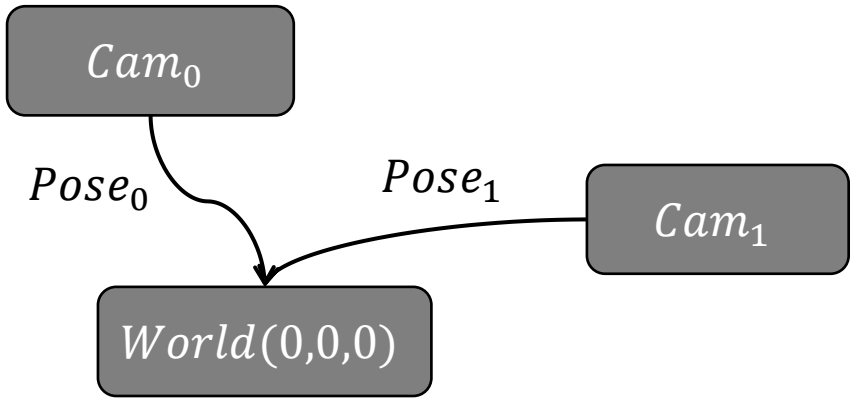
$$I_{new} = I_{01}$$

→ None-differentiable

Assignment 3

Task 1. Given the scenario and relative pose graph, implement 1) forward point warping of given pixels of camera0 (RGB_0) into camera1's viewpoint (RGB_1) using depth image of camera0 ($Depth_0$) and camera to world poses ($T_{c_0 \to World}$, $T_{c_1 \to World}$), 2) Bilinear interpolation

1. Check slide 40 for step-by step procedures of 3D warping
2. Check slide 41 for bilinear interpolation



RGB_0

Warp viewpoint

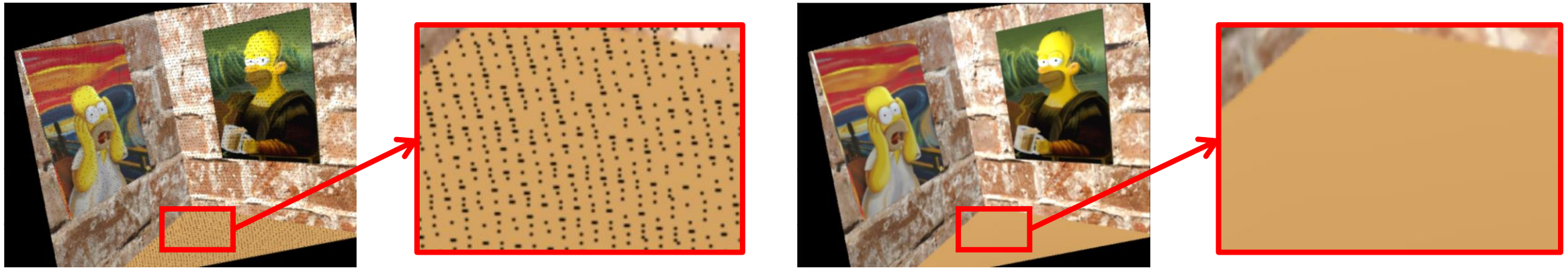
RGB_1

Task 2. Using pipeline and function from Task 1, implement both forward and inverse image warping with nested for loops and vectorized way (optional)

1. Check slide 9 for details of difference between forward and inverse warping
2. Once the pipeline works with for nested loops, try to replace them with vectorized pipeline

Assignment 3

Comparison between forward warping VS inverse warping (task2)



Comparison between nearest neighbor interpolation VS bilinear interpolation (task2)

Questions are welcome !

References

(S01) Just the tip image : https://www.writerscafe.org/writing/TheLonelySparrow/2069834/
(S01) Simpsons GIF : Simpsons S05 E16 Homer Loves Flanders
(S10) 3D model of banana : PoseCNN: A Convolutional Neural Netwprl for 6D Object Pose Estimation in Cluttered Scenes (arXiv 2017, Yu, Tanner Venkatraman, Dieter)
(S12) Lena image : https://en.wikipedia.org/wiki/Lenna
(S14,19,21,26) 3D model of driller : PoseCNN: A Convolutional Neural Netwprl for 6D Object Pose Estimation in Cluttered Scenes (arXiv 2017, Yu, Tanner Venkatraman, Dieter)
(S18) Checkerboard img : https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html
(S21) Robot arm : https://hum3d.com/3d-models/industrial-robot-arm/
(S22) DeepIM : Deep Iterative Matching for 6D Pose Estimation (Y.Li, G.Wang, X,Ju, Y.Xiang, D.Fox, ECCV 2018)
(S24,25) 3D model of sugar box : PoseCNN: A Convolutional Neural Netwprl for 6D Object Pose Estimation in Cluttered Scenes (arXiv 2017, Yu, Tanner Venkatraman, Dieter)
(S26,27,31) YCB dataset : PoseCNN: A Convolutional Neural Netwprl for 6D Object Pose Estimation in Cluttered Scenes (arXiv 2017, Yu, Tanner Venkatraman, Dieter)
(S30) 3D model of cracker box : PoseCNN: A Convolutional Neural Netwprl for 6D Object Pose Estimation in Cluttered Scenes (arXiv 2017, Yu, Tanner Venkatraman, Dieter)
(S31) Laval dataset : A framework for evaluating 6-DoF object trackers (ECCV 2018, Gardon, Laurendeau, Lalonde)
(S32) Tracking figure : https://www.acer.com/ac/en/US/content/series-design/wmr
(S33) Lumbar puncture : https://news.cancerconnect.com/treatment-care/lumbar-puncture-is-used-to-diagnose-and-treat-certain-leukemias-and-cancers-Kik_msavJk-_XAAXiIXqxw
(S33) Hololens : https://www.aitgmbh.de/blog/vr-ar-hololens/hololens-erstkontakt-mit-der-hardware/
(S39) Unsupervised Learning of Depth and Ego-Motion from Video (T. Zhou, M. Brown, N. Snavely, D. Lowe, CVPR 2017)
(S42) Homer scream image : https://www.amazon.de/GB-EYE-LTD-Simpsons-Simpson/dp/B000W9OYNS
(S43) Homer mona Lisa image : https://www.newgrounds.com/art/view/electricpants/homer-lisa