PS3 Review Session

Krishnan Srinivasan
CS231A
02/18/2022

Overview

1. Space carving
2. Representation Learning
3. (EC) Monocular Depth Estimation
4. Unsupervised Monocular Depth Estimation
5. Tracking

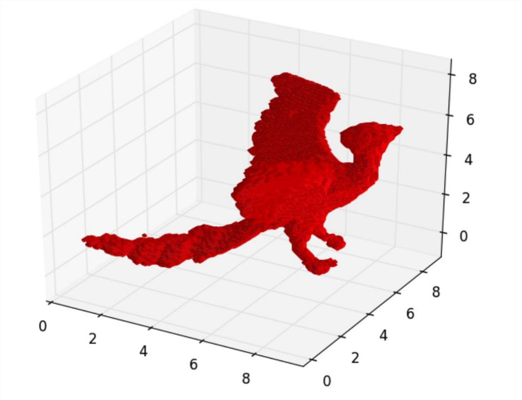
Space Carving

Objective:

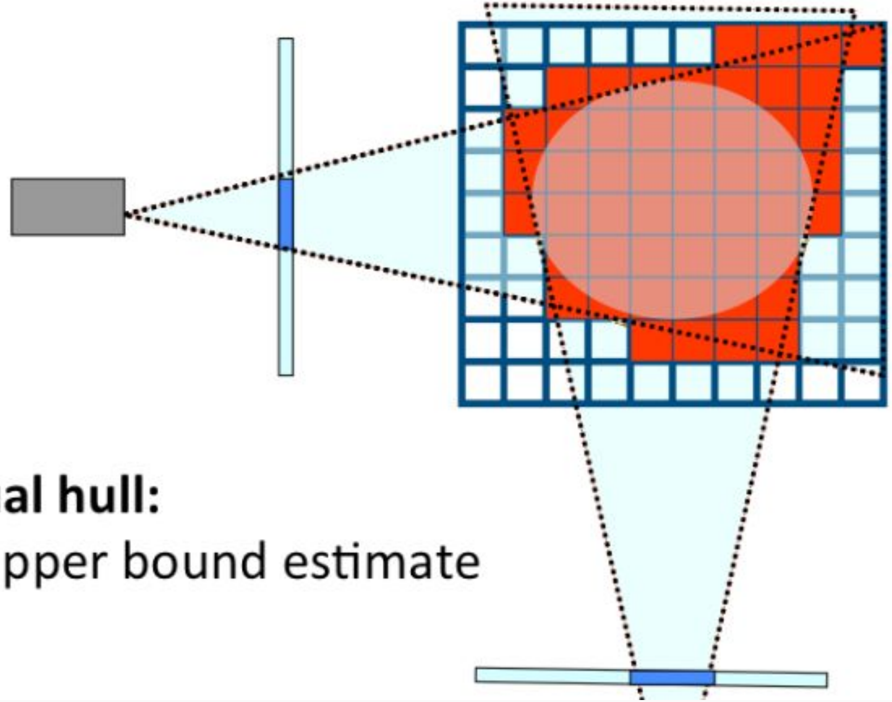
- Implement the process of space carving.

Lectures:

- Active Stereo & Volumetric Stereo



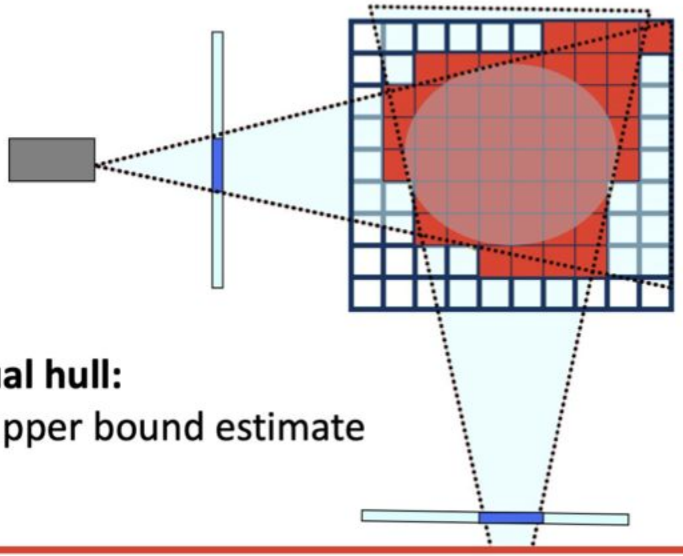
Review: Space Carving



Visual hull:
an upper bound estimate

Space Carving: Overview

Computing Visual Hull in 2D

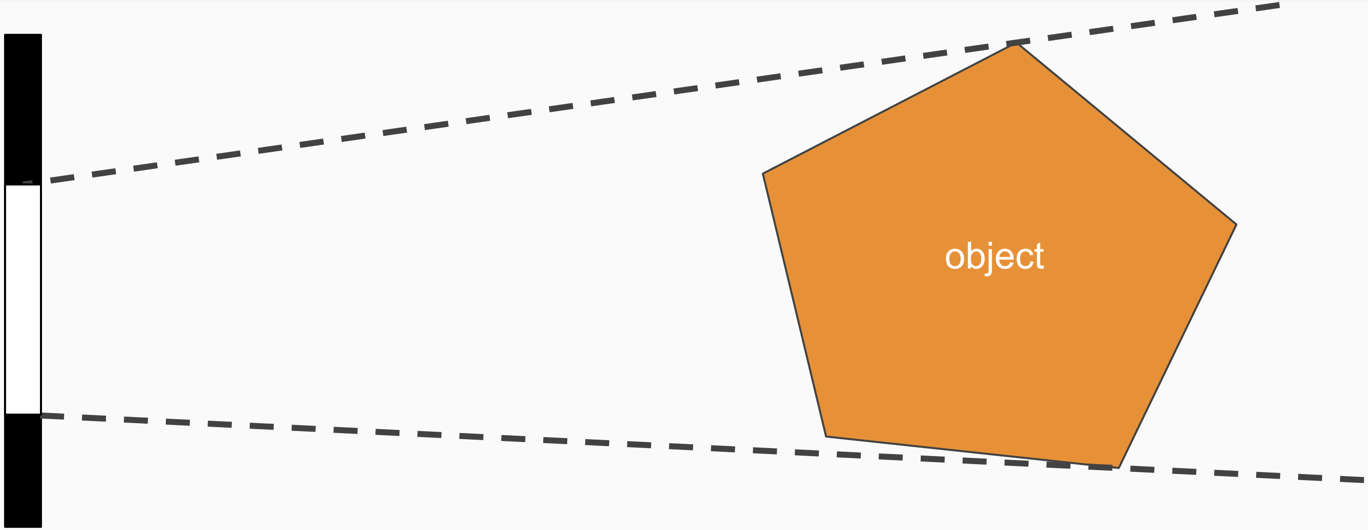


Visual hull:
an upper bound estimate

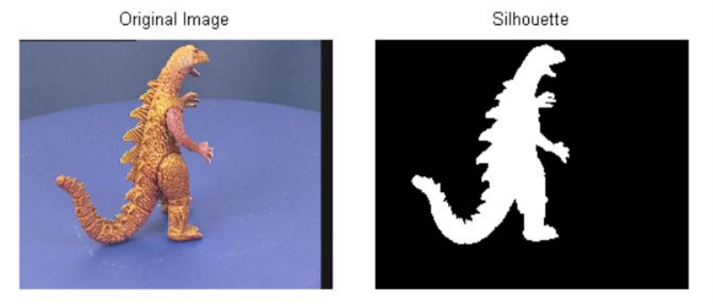
Consistency:

A voxel must be projected into a silhouette in each image

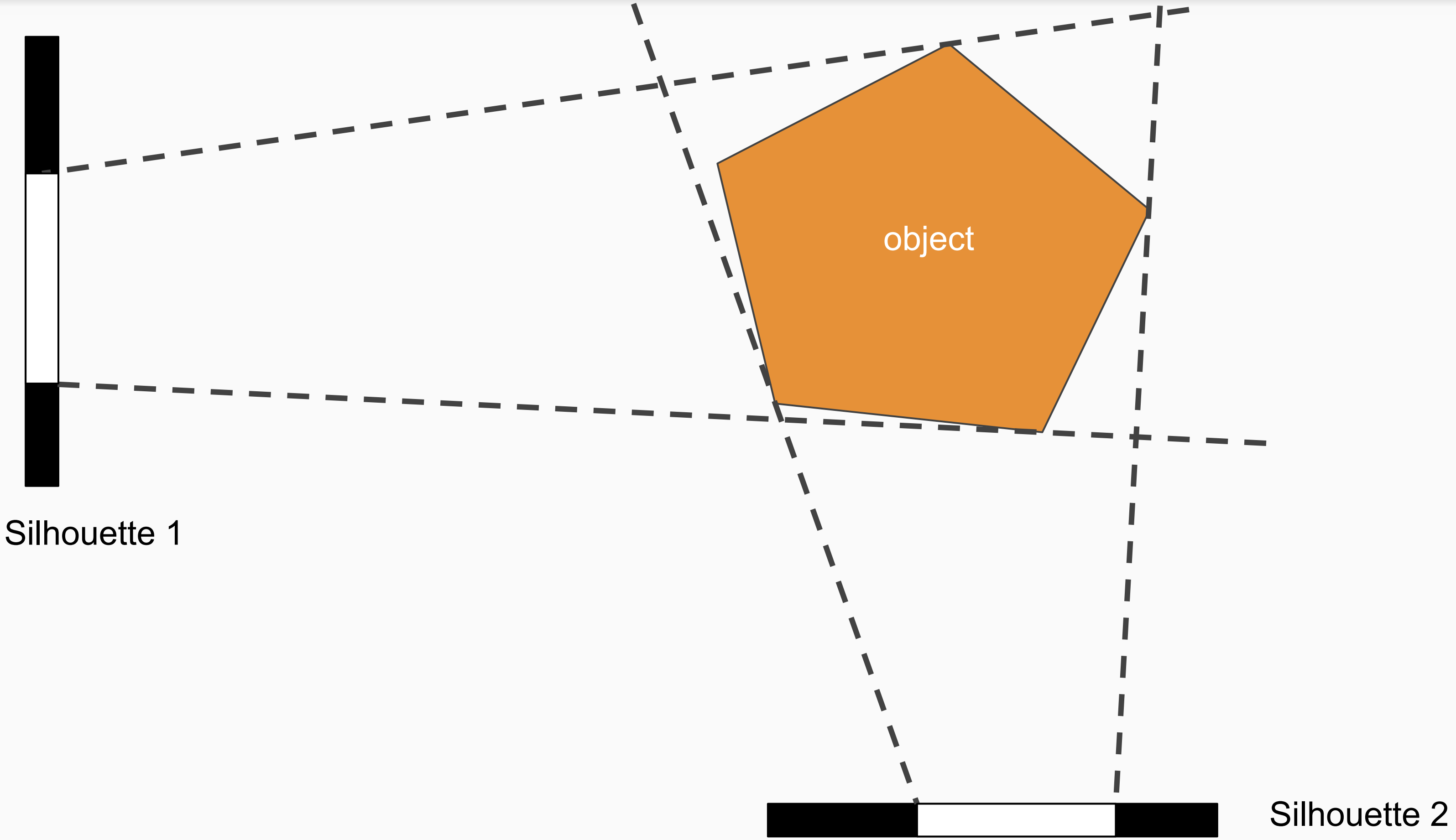
Review: Space Carving



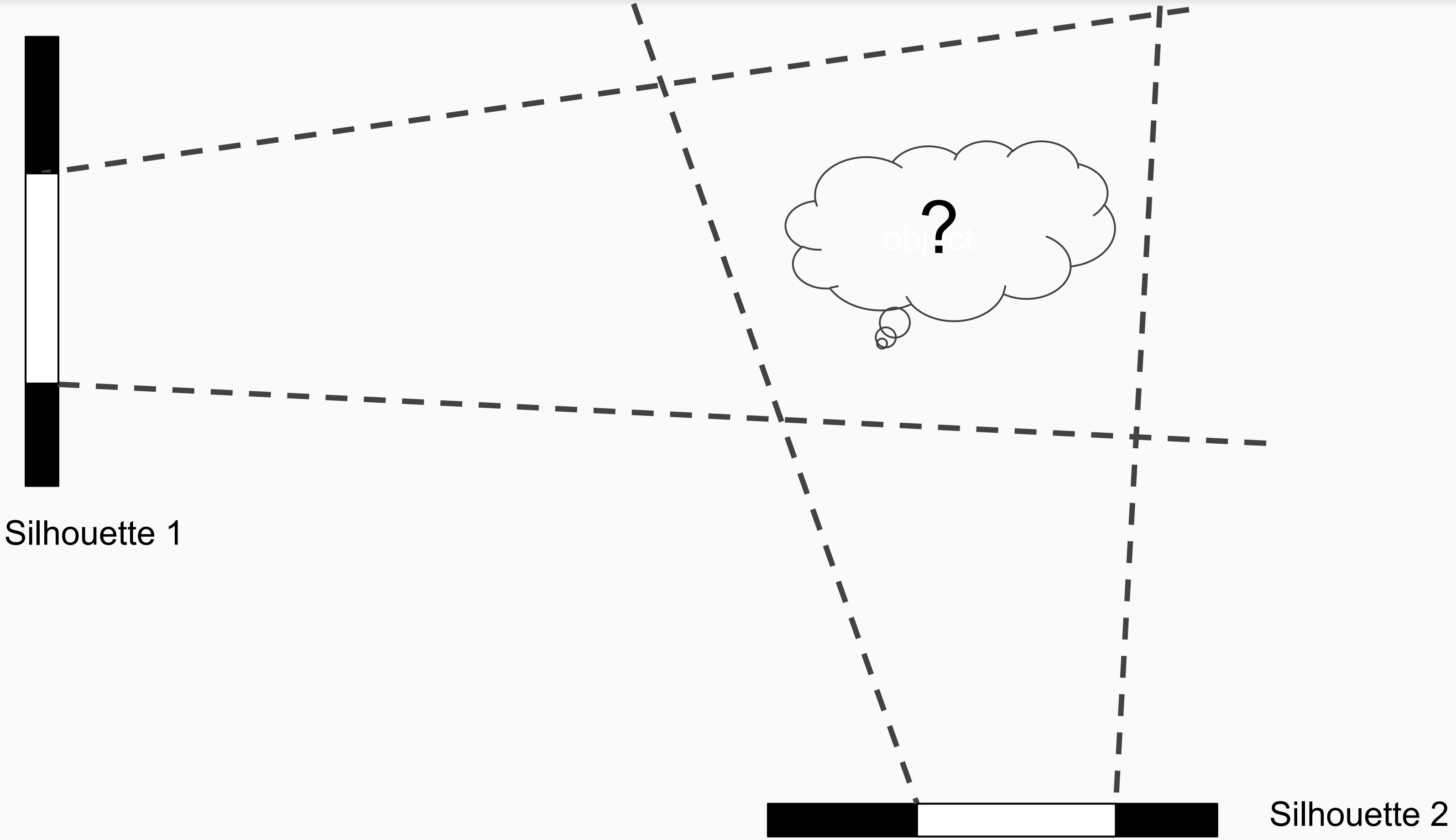
Silhouette 1



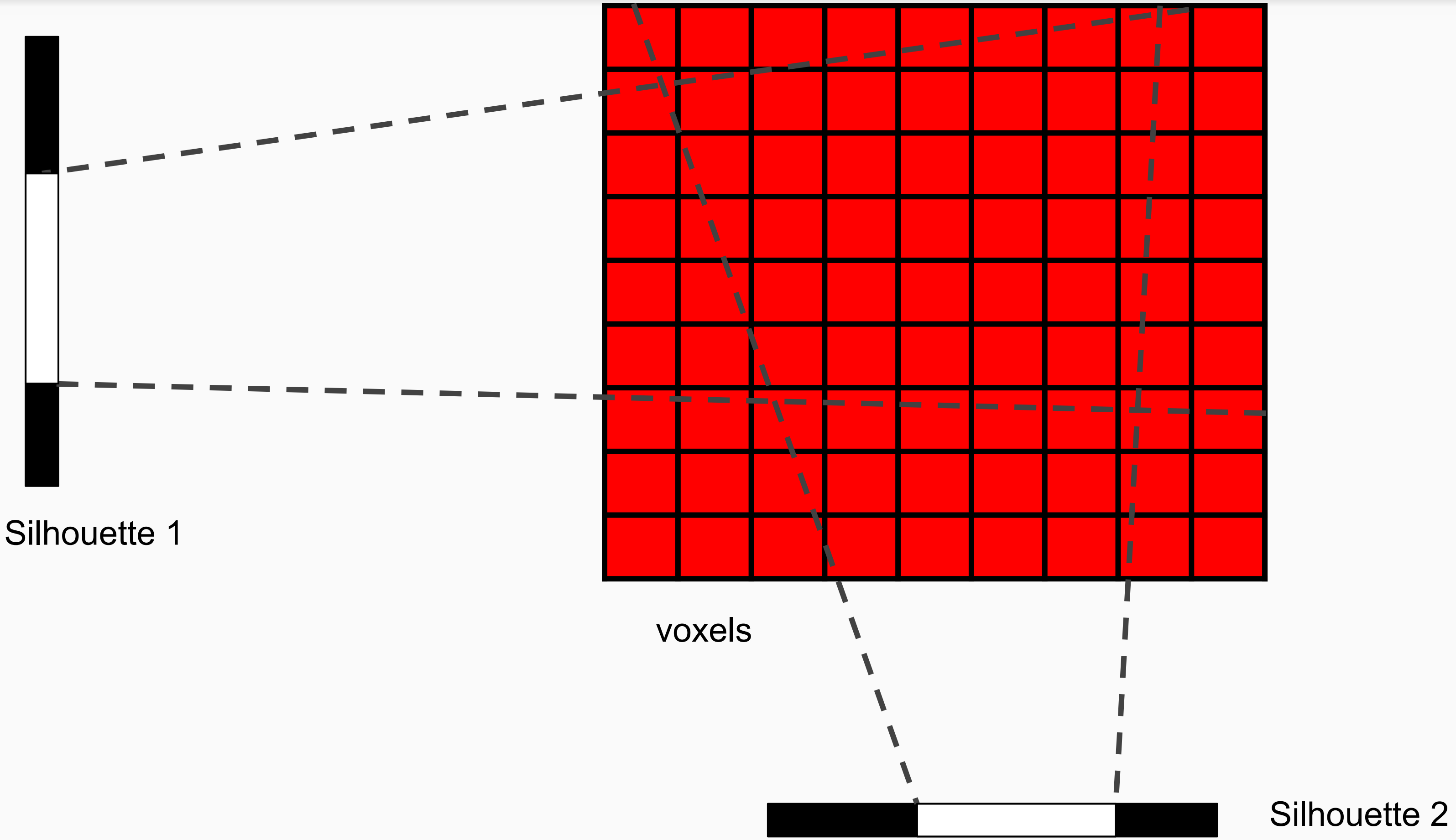
Review: Space Carving



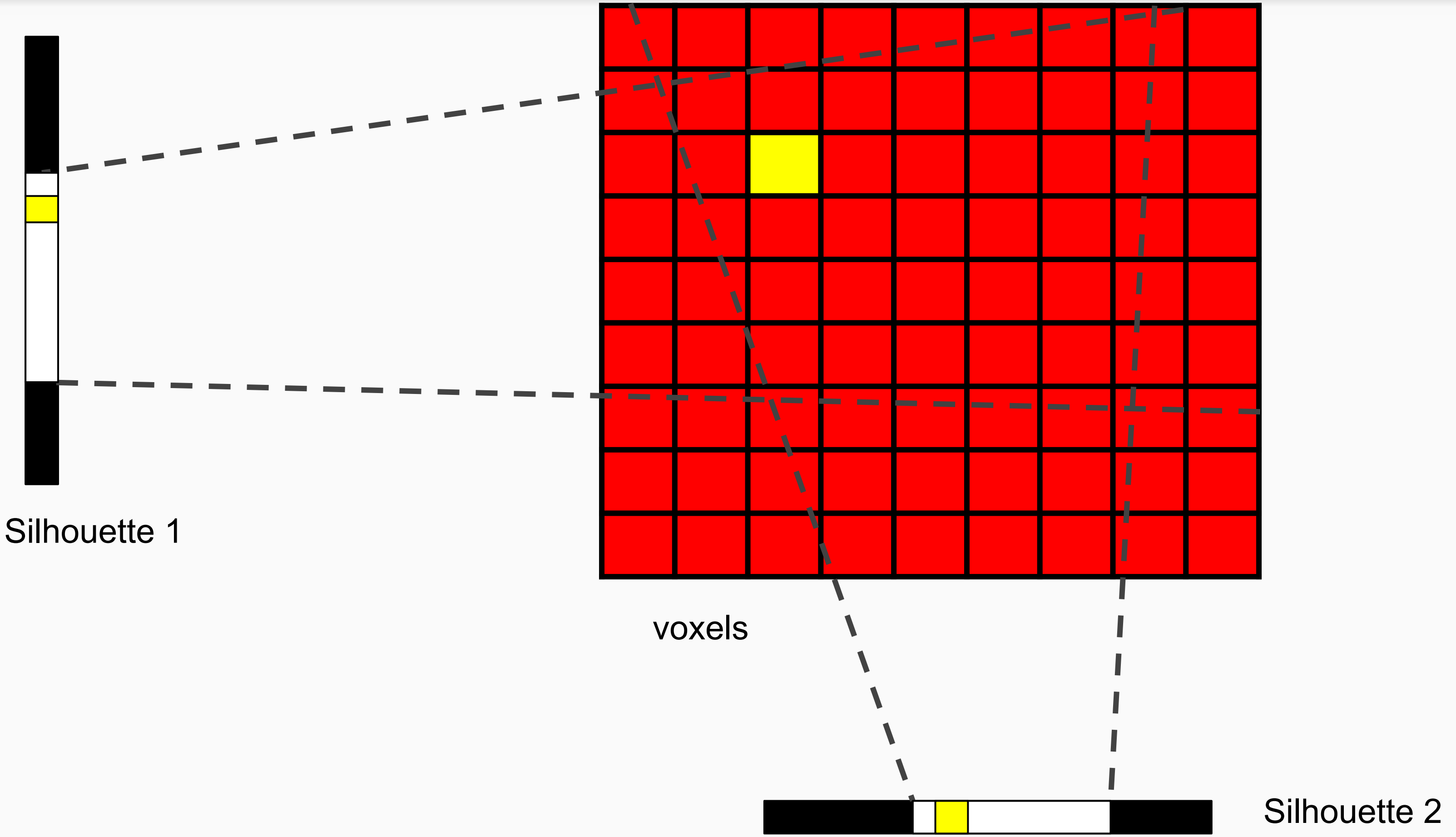
Goal of Space Carving



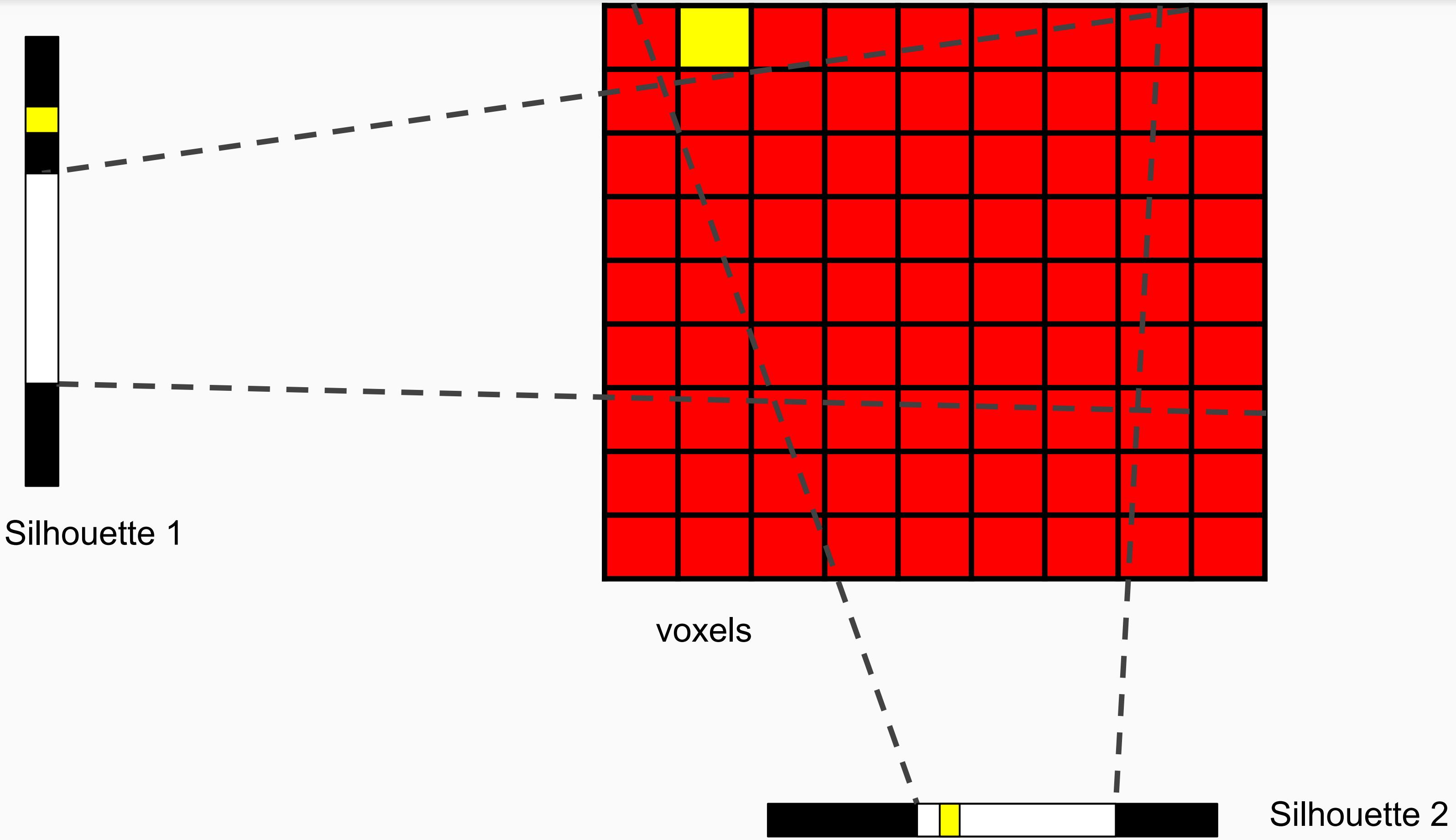
Review: Space Carving



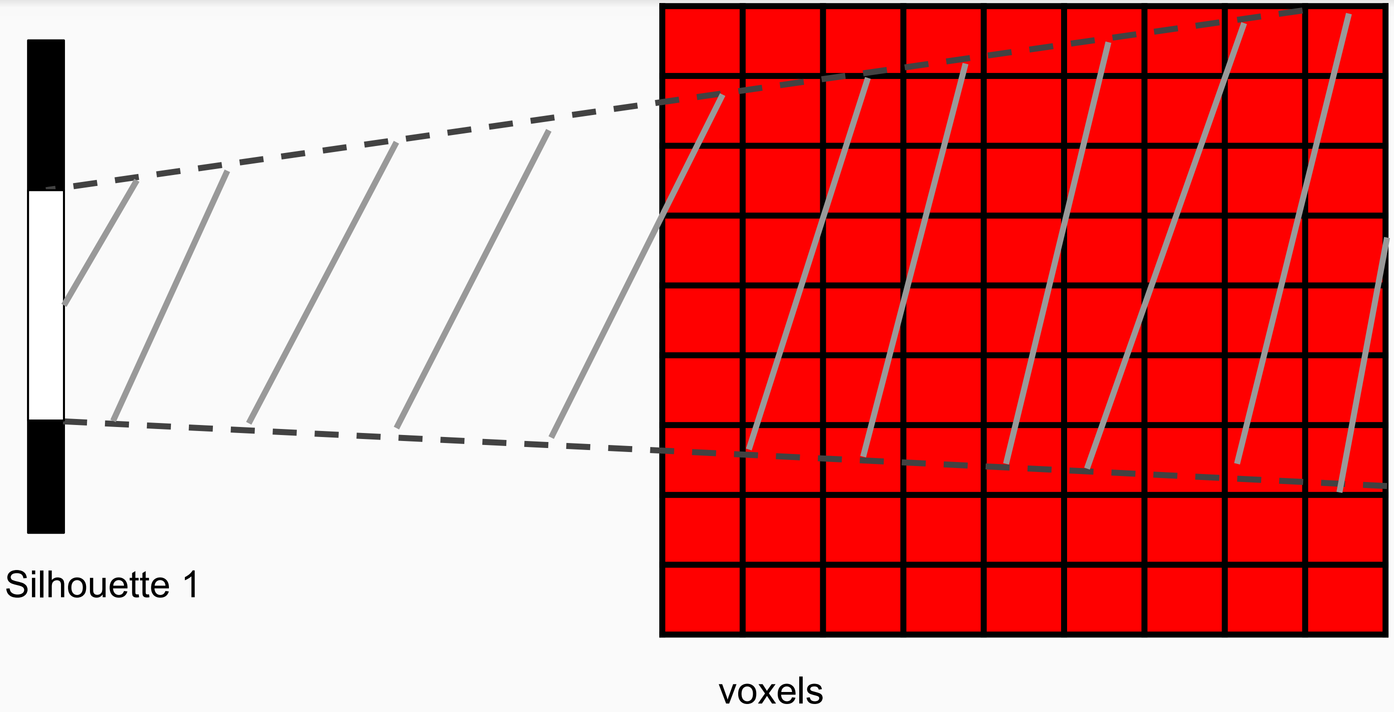
Review: Space Carving



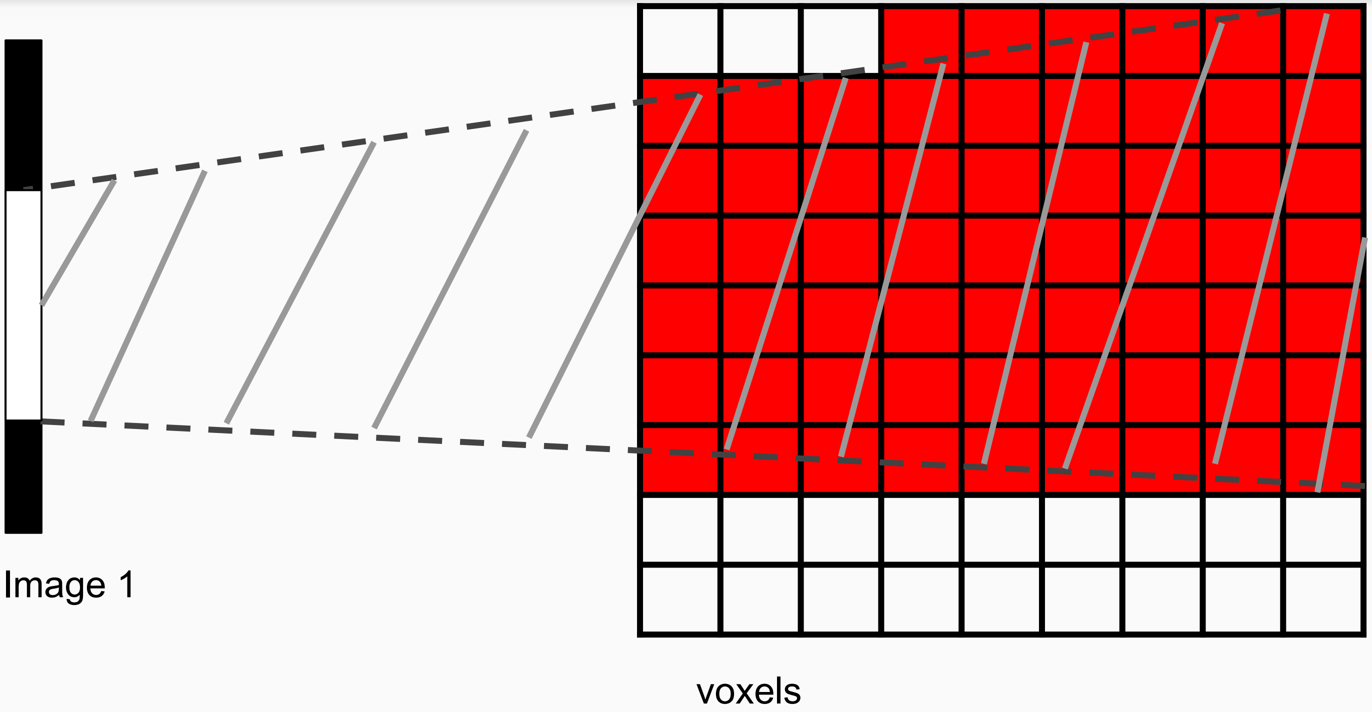
Review: Space Carving



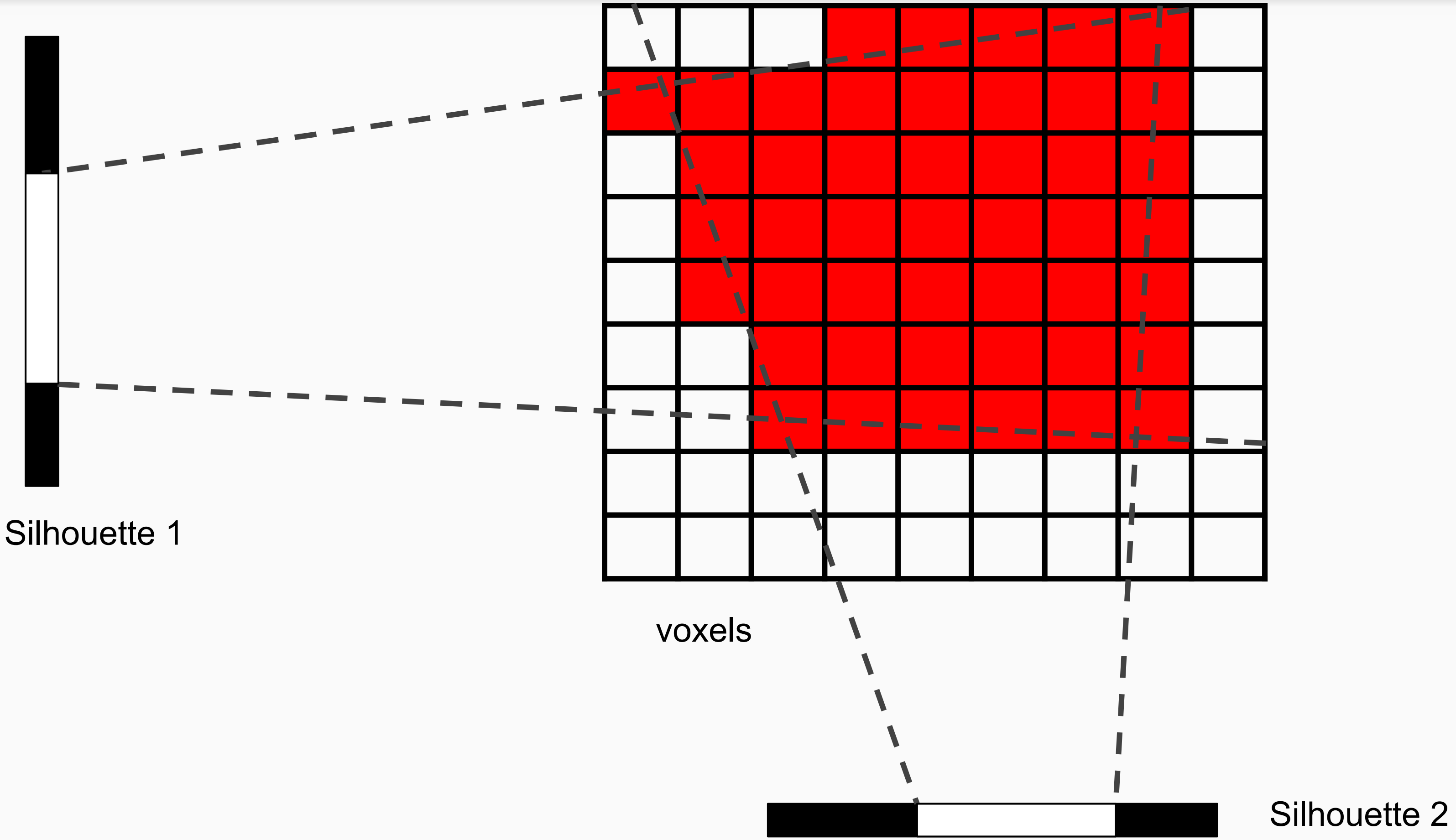
Review: Space Carving



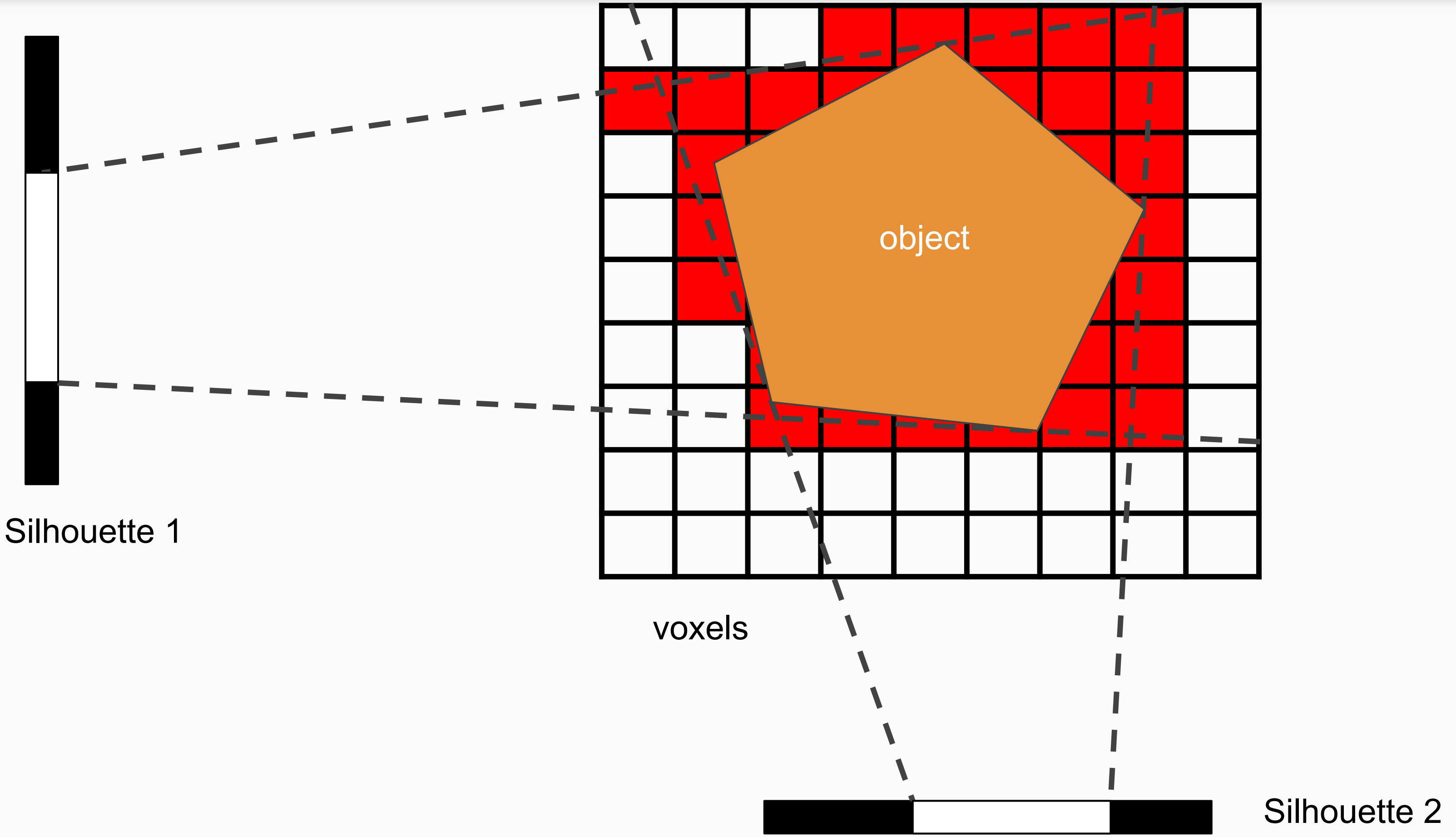
Review: Space Carving



Review: Space Carving



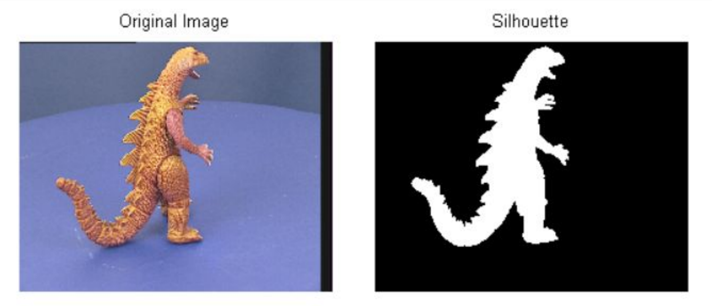
Review: Space Carving



Space carving - overview

Steps:

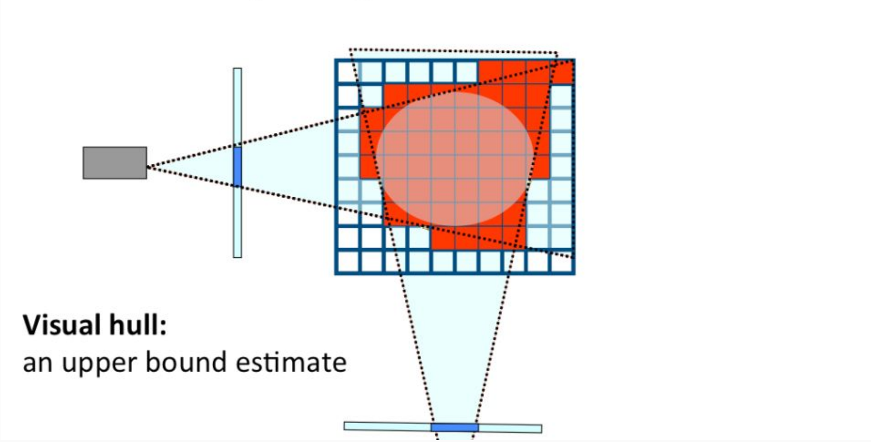
- Estimate silhouettes of images (could be based on some heuristics, e.g. color)
- Form the initial voxels as a cuboid
- Iterate over cameras and remove the voxels which project to the dark part of each silhouette



Space carving - (a) (b) (c)

Steps:

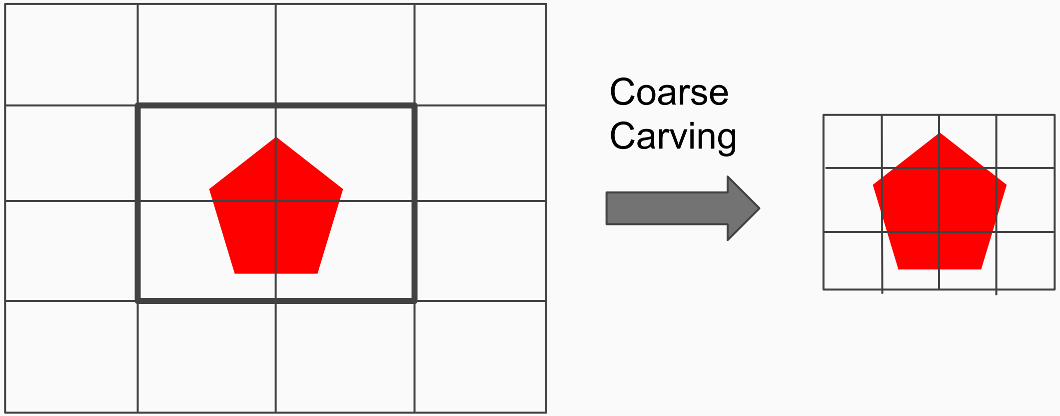
- Estimate silhouettes of images (could be based on some heuristics, e.g. color)
- Form the initial voxels as a cuboid
  - *You may find these functions useful: np.meshgrid, np.repeat, np.tile*
- Iterate over cameras and remove the voxels which project to the dark part of each silhouette
  - *Question: What will the voxels look like after the first, second, … iteration?*



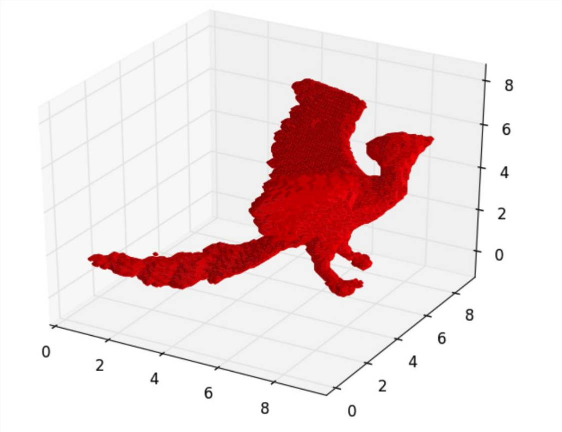
Space carving - (d)

Steps:

- Estimate silhouettes of images (could be based on some heuristics, e.g. color)
- Form the initial voxels as a cuboid
  - Question: What will the cuboid look like after each iteration?
  - Improvement: tighter bounded cuboid
  - How to do a coarser carving first? (use num_voxels=4000)
- Iterate over cameras and remove the voxels which project to the dark part of each silhouette



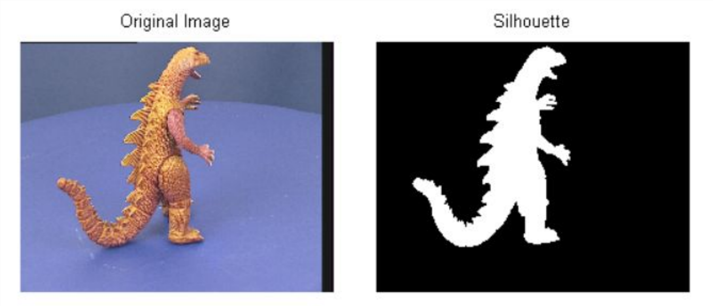
Final Output



Space carving - (e)

Steps:

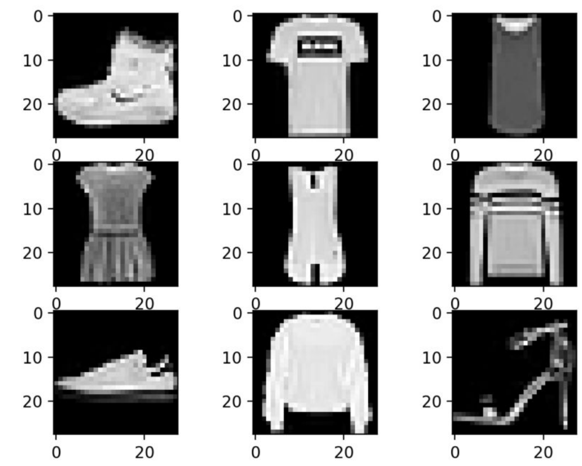
- Estimate silhouettes of images (could be based on some heuristics, e.g. color)
  - Problem: The quality of silhouettes is not perfect.
  - The silhouette from each camera is not perfect, but the result is ok. Why?
  - Experiment: Use only a few of the silhouettes.
- Form the initial voxels as a cuboid
- Iterate over cameras and remove the voxels which project to the dark part of each silhouette



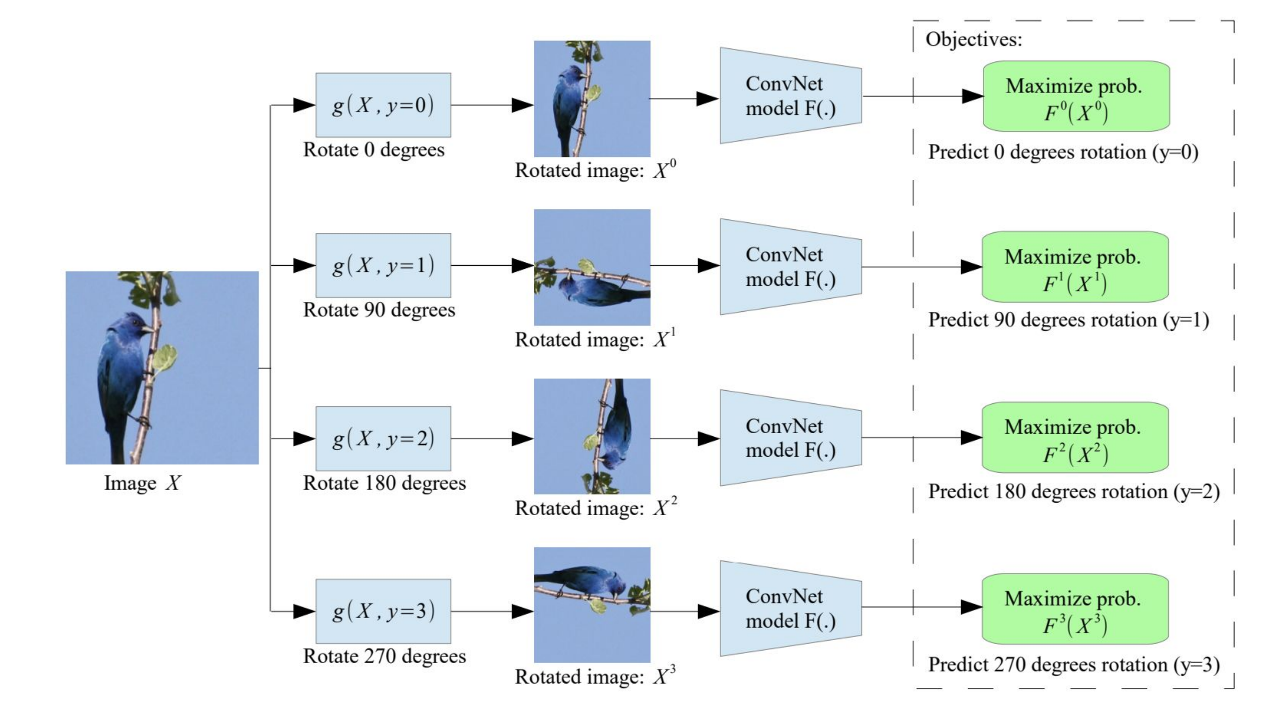
Problem 2 - Representation Learning

In this notebook, we will be using the [Fashion MNIST dataset] to showcase how self-supervised representation learning can be utilized for more efficient training in downstream tasks. We will do the following things:

1. Train a classifier from scratch on the Fashion MNIST dataset and observe how fast and well it learns
2. Train useful representations via predicting image rotations, rather than classifying images
3. Transfer our rotation pretraining features to solve the classification task with much less data than in step 1



Unsupervised Representation Learning by Predicting Image-Rotations (ICLR ‘18)



Problem 2 - Representation Learning

PyTorch Training basics (training.py):

- Use **torch.DataLoader** and **Dataset** to load datasets and make batches
- Create layers using **torch.nn** module
- Use **torch.optim** to create an **SGD** Optimizer take gradient steps
- Manipulating **torch.Tensor**:
  - use t.cpu() to move from GPU -> CPU, use t.cuda() for CPU -> GPU

Problem 2 - Representation Learning

`MNISTDatasetWrapper(Dataset)`

- __init__: load pct% of images from processed .pt file
- __getitem__: randomly rotate an image from self.imgs. **Hint:** use PIL.Image.rotate to rotate image, and then return to torch.Tensor type
- **Hint:** Use torch.tensor(rotation_idx).long() to generate rotation labels

nn.Sequential(...)

- Creates a stack of layers that pass input data through a model
- nn.Linear(...) layers form weights and biases for a single

Problem 2 - Representation Learning

Training example (from pytorch-examples repo)

- **opt.zero_grad** to zero gradients before update
- **loss.backward** to backpropagate gradients
- **opt.step** to update model params

```
# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out),
        )
loss_fn = torch.nn.MSELoss(reduction='sum')

# Use the optim package to define an Optimizer that will update the weights of
# the model for us. Here we will use Adam; the optim package contains many other
# optimization algorithms. The first argument to the Adam constructor tells the
# optimizer which Tensors it should update.
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for t in range(500):
  # Forward pass: compute predicted y by passing x to the model.
  y_pred = model(x)

  # Compute and print loss.
  loss = loss_fn(y_pred, y)
  print(t, loss.item())

  # Before the backward pass, use the optimizer object to zero all of the
  # gradients for the Tensors it will update (which are the learnable weights
  # of the model)
  optimizer.zero_grad()

  # Backward pass: compute gradient of the loss with respect to model parameters
  loss.backward()

  # Calling the step function on an Optimizer makes an update to its parameters
  optimizer.step()
```

Problem 3: Supervised Monocular Depth Estimation

High Quality Monocular Depth Estimation via Transfer Learning

Ibraheem Alhashim
KAUST
ibraheem.alhashim@kaust.edu.sa

Peter Wonka
KAUST
pwonka@gmail.com

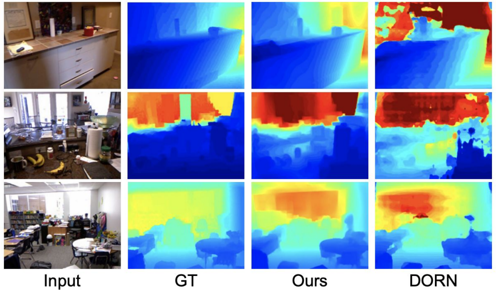
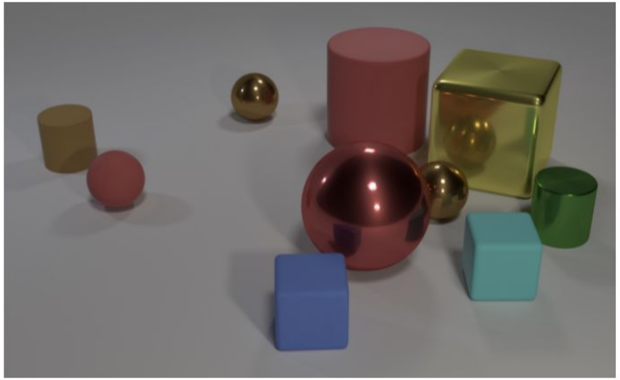


Figure 1. **Comparison of estimated depth maps:** input RGB images, ground truth depth maps, our estimated depth maps, state-of-the-art results of [9].

Problem 3: Supervised Monocular Depth Estimation

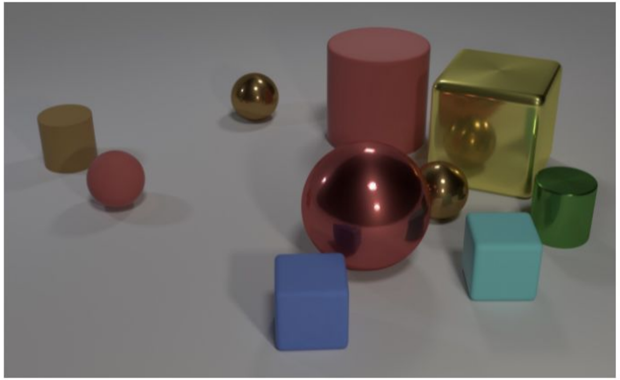
- Train DenseDepth model using labeled depth data from RGB images
- CLEVR-D dataset procedurally generated with ground-truth depth map
- Need to modify data.py, losses.py, create DenseDepth autoencoder model
- Use torch.nn module to define L1Loss



Source: Johnson et al.

Problem 3: Supervised Monocular Depth Estimation

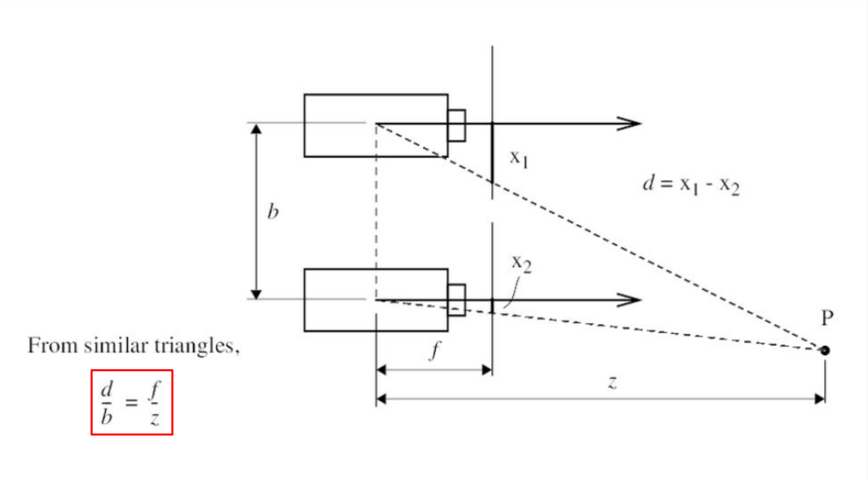
- Extra credit:
    - Modify encoder in DenseDepth to first learn RGB -> grayscale image (with bottleneck layer)
    - Remove decoder and then finetune features to go from RGB -> depth (as before)



Source: Johnson et al.

Problem 4: Unsupervised Monocular depth estimation

- Train a network to predict *disparity* (d) between two images
- Disparity is related to ($\propto$) depth from the equation in Fig. 4
- Trained using left and right images by synthesizing left and right disparity maps after taking left image as input



Problem 4: Unsupervised Monocular depth estimation

- During training: generate left image (using output disparity from left) and left disparity using right image
    - Need to implement generate_image_left and generate_image_right functions
- Image loss: compares L1 loss of generated left and right images to actual
- Disparity loss: enforce cycle consistency comparing L1 of generated left and right disparities to actual

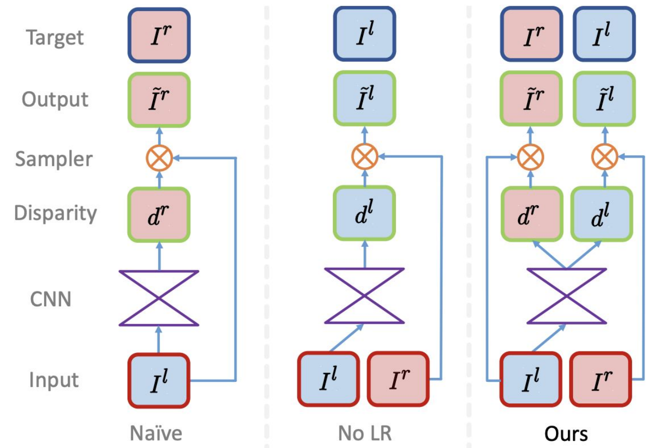


Figure 3. Sampling strategies for backward mapping. With naïve sampling the CNN produces a disparity map aligned with the target instead of the input. No LR corrects for this, but suffers from artifacts. Our approach uses the left image to produce disparities for both images, improving quality by enforcing mutual consistency.

Problem 4a. Data Augmentation

- Use torchvision.transforms.RandomHorizontalFlip to flip left and right images
    - Augmenting data allows training on 2x the amount of data (since left and right images get used as input)
- Apply self.transform to each left and right image and return using self._flip

Problem 4b. Bilinear sampler

- Given disparity, shift the image horizontally (essentially generating the left/right image, hence "sampler")
    - Do this by sampling horizontally rectified images
- Use **torch.linspace** and **torch.meshgrid** to create a grid of xy-coordinates (from 0 -> 1, using width and height of image shape)
- Add disparity x-coordinates to coordinate grid
- Combine x and y-coords using **torch.stack** to generate shifted disparity grid, and generate new sampled image using F.grid_sample()
    - Scale disparity grid between -1 and 1

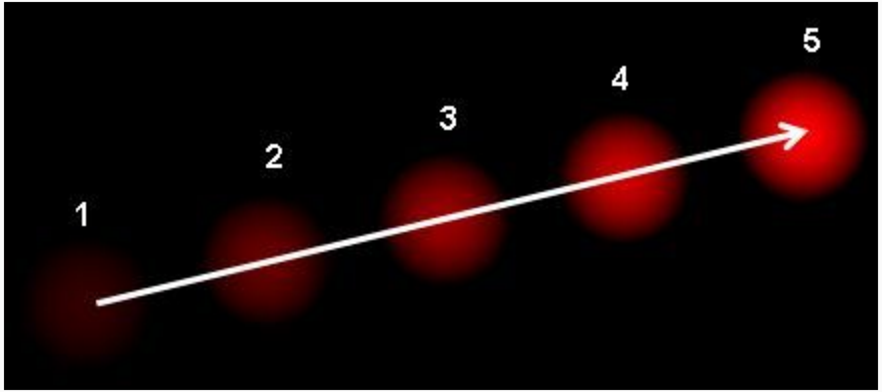
Problem 4c. Left/Right image generator

- Given image and disparity map, generate left and right images
    - (use bilinear sampler from part b)
- Given disparity map is for left -> right image mapping
- To generate left image, simply apply -disp to horizontally shift in the opposite direction

Problem 5 - Tracking and Optical Flow

Luca-Kanade point feature (sparse) optical flow:

- cv2.goodFeaturesToTrack(): finds N strongest corners to track in the image for optical flow
- For our problem, use it to find N = 200 points (i.e. features, `maxCorners` in function)
- OpenCV2 Tutorial for LK Optical Flow



Problem 5 - Tracking and Optical Flow

Track a pixel in the first image frame (at timestep t0): (x, y, t0):

- Assume that intensity does not change between frames: $I(x, y, t) = I(x + dx, y + dy, t + dt)$
- Optical flow equation (FO Taylor approx): $f_x u + f_y v + f_t = 0$ where: $f_x = \frac{\partial f}{\partial x} \; ; \; f_y = \frac{\partial f}{\partial y}$
- Lucas-Kanade is used to compute u, v (i.e., pixel movement) $u = \frac{dx}{dt} \; ; \; v = \frac{dy}{dt}$
- Steps:

  1) Detect Shi-Tomasi corners (p0) using cv2.goodFeatures

  2) iterate through frames, track points from original frame using cv2.calcOpticalFlowPyrLK

Problem 5 - Tracking and Optical Flow

```python
# params for ShiTomasi corner detection
feature_params = dict(
    maxCorners=200,
    qualityLevel=0.01,
    minDistance=7,
    blockSize=7)

# Parameters for lucas kanade optical flow
lk_params = dict(
    winSize=(75, 75),
    maxLevel=1,
    criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 100, 0.01),
    flags=(cv2.OPTFLOW_LK_GET_MIN_EIGENVALS))

# Read the frames.
frames = []
for i in range(1, 11):
    frame_path = os.path.join(folder_path, 'rgb%02d.png' % i)
    frames.append(cv2.imread(frame_path))

# Convert to gray images.
old_frame = frames[0]
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
p0 = cv2.goodFeaturesToTrack(old_gray, mask=None, **feature_params)
print("number of features to track:", len(p0))
assert len(p0) <= 200

# Create some random colors for drawing
color = np.random.randint(0, 255, (200, 3))

# Create a mask image for drawing purposes
mask = np.zeros_like(old_frame)

tracks = []

for i,frame in enumerate(frames[1:]):
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # TODO: Fill in this code
    # BEGIN YOUR CODE HERE
    pass

    #Once you compute the new feature points for this frame, comment this out
    #to save images for your PDF:
    #draw_tracks(frame_num, frame, mask, points_prev, points_curr, color, folder_path)
    # END YOUR CODE HERE
```

```python
# params for ShiTomasi corner detection
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )

# Parameters for lucas kanade optical flow
lk_params = dict( winSize  = (15,15),
                  maxLevel = 2,
                  criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 10, 0.03))

# Create some random colors
color = np.random.randint(0,255,(100,3))

# Take first frame and find corners in it
ret, old_frame = cap.read()
old_gray = cv.cvtColor(old_frame, cv.COLOR_BGR2GRAY)
p0 = cv.goodFeaturesToTrack(old_gray, mask = None, **feature_params)

# Create a mask image for drawing purposes
mask = np.zeros_like(old_frame)

while(1):
    ret,frame = cap.read()
    frame_gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    # calculate optical flow
    p1, st, err = cv.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)

    # Select good points
    if p1 is not None:
        good_new = p1[st==1]
        good_old = p0[st==1]

    # draw the tracks
```

Problem 5c-e. (Dense optical flow)

- Run dense optical flow through Flownet (NN) model and Gunnar Farneback algorithm
    - From pair of frames, generate map of pixels showing relative direction (and amount) of motion
- Running FlowNet2.0:
    - Loading ml4a pre-trained model in Colab
    - Generating flow, and reconstruction of images using **ml4a.canvas.map_image**()
- Running Farneback (using OpenCV tutorial code)

Problem 5c-e

1. Generate dense optical flow for pairs of image frames (labeled with either Farneback- or flownet-)
    a. Using globe1, globe2, and chairs images
2. Save dense flow output, and qualitatively compare prominent artifacts in both
3. Describe limitations to NN-trained model, and how it might be improved