A Differentiable Monte Carlo path tracer

Team 3

20183477 이현우

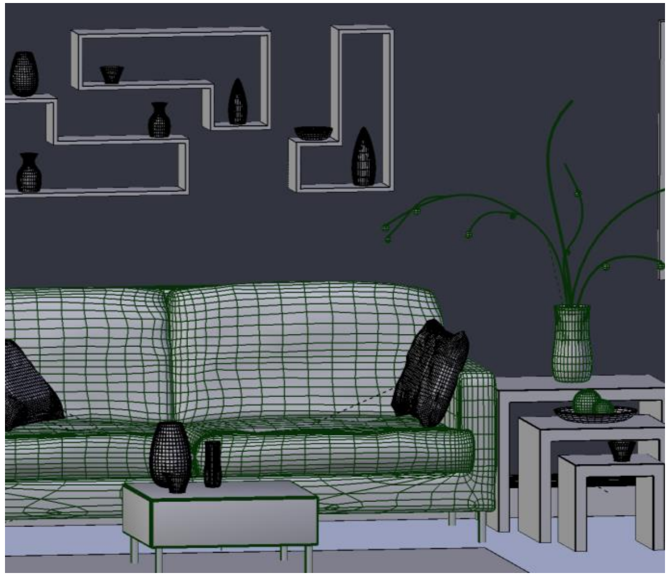
20184364 유한결

20187050 박주호

2019. 5. 30.

Review

Modern renderer produce realistic images



render

3D scene

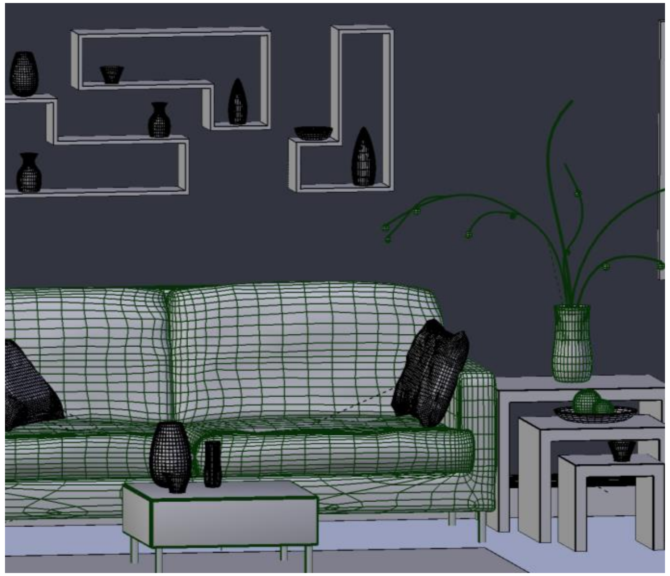
light simulation
(games, movies)

image

From Tzu-Mao’s SIGGRAPH slides

Review

Inverse rendering



inverse
render

3D scene

image

From Tzu-Mao's SIGGRAPH slides

Review

Render and compare approach

3D scene:
triangle positions
camera pose
materials
…

image

∇ loss

target

From Tzu-Mao's SIGGRAPH slides

Review

Gradients update the 3D scene

3D scene:
triangle positions
camera pose
materials
…

image

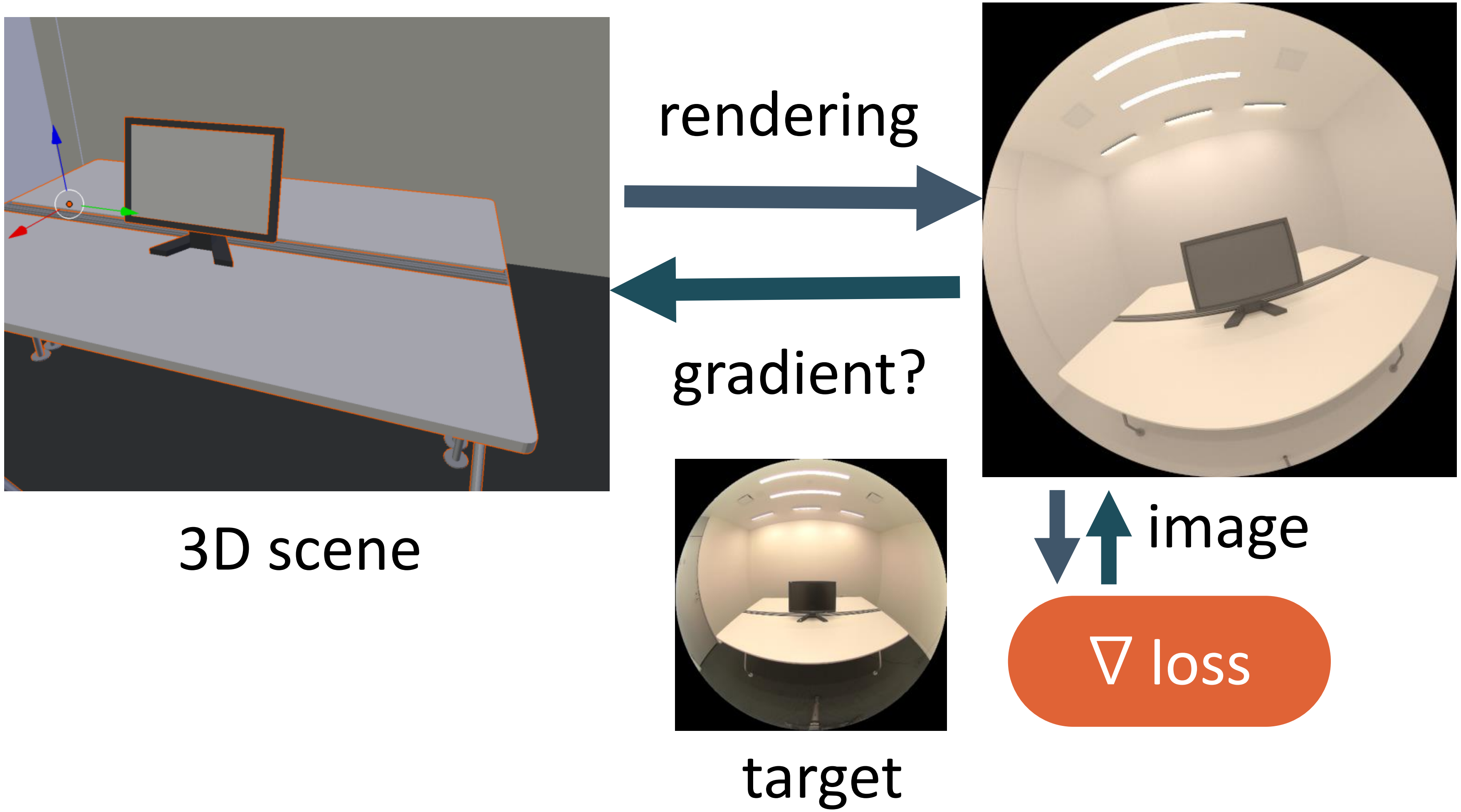
target

∇ loss

From Tzu-Mao's SIGGRAPH slides

Review

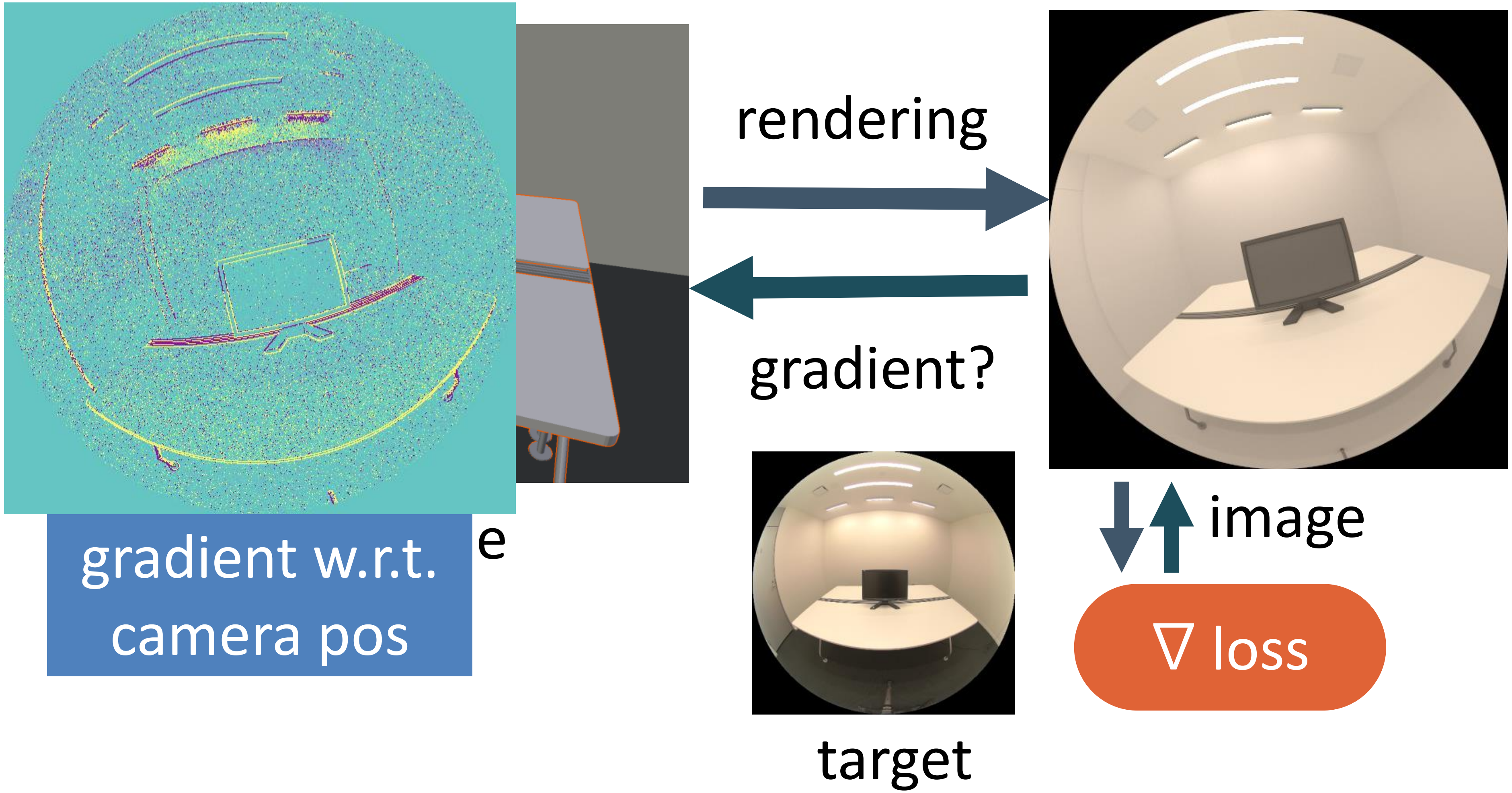
Goal: compute the rendering gradient



From Tzu-Mao’s SIGGRAPH slides

Review

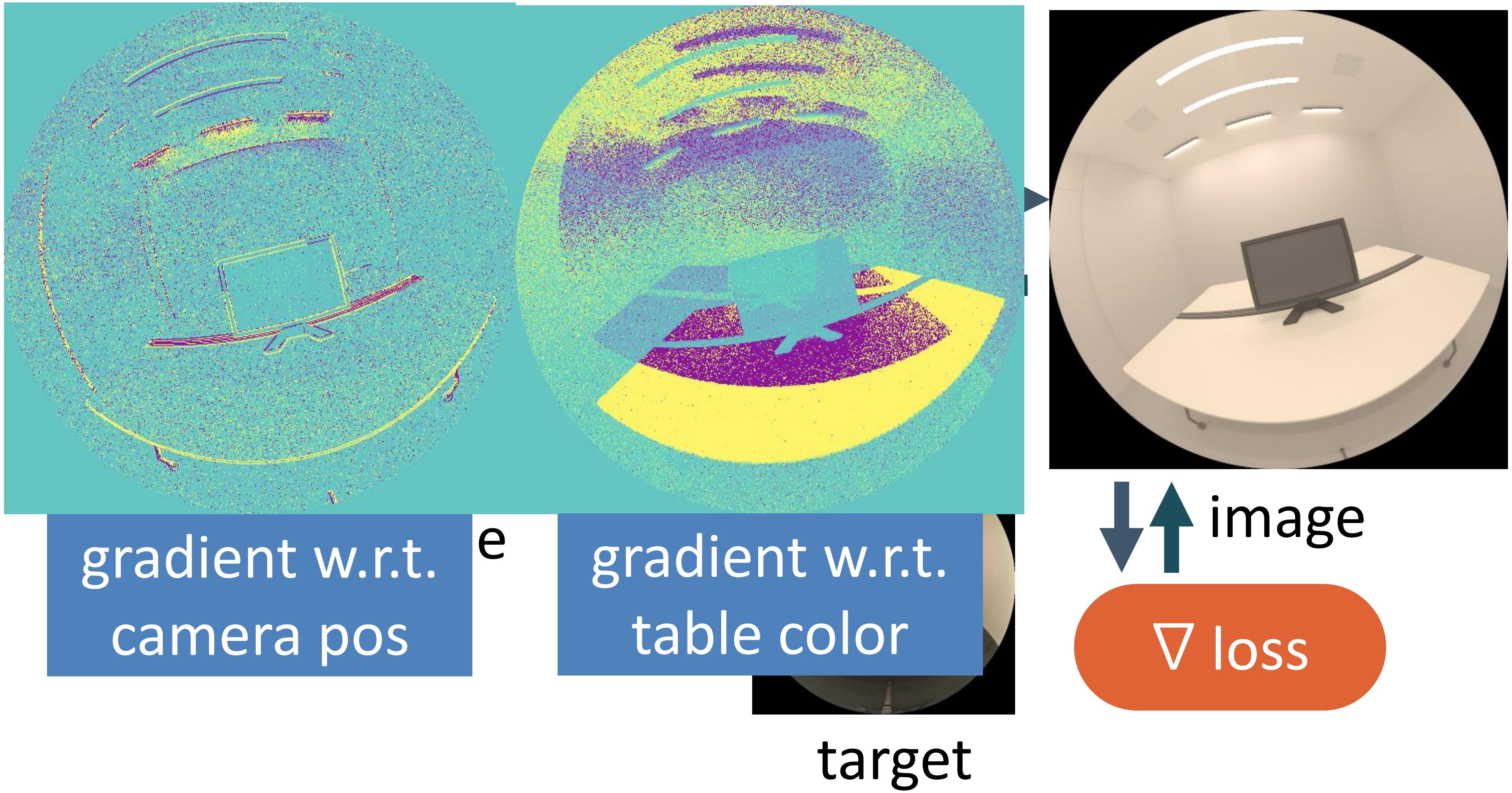
Goal: compute the rendering gradient



From Tzu-Mao's SIGGRAPH slides

Review

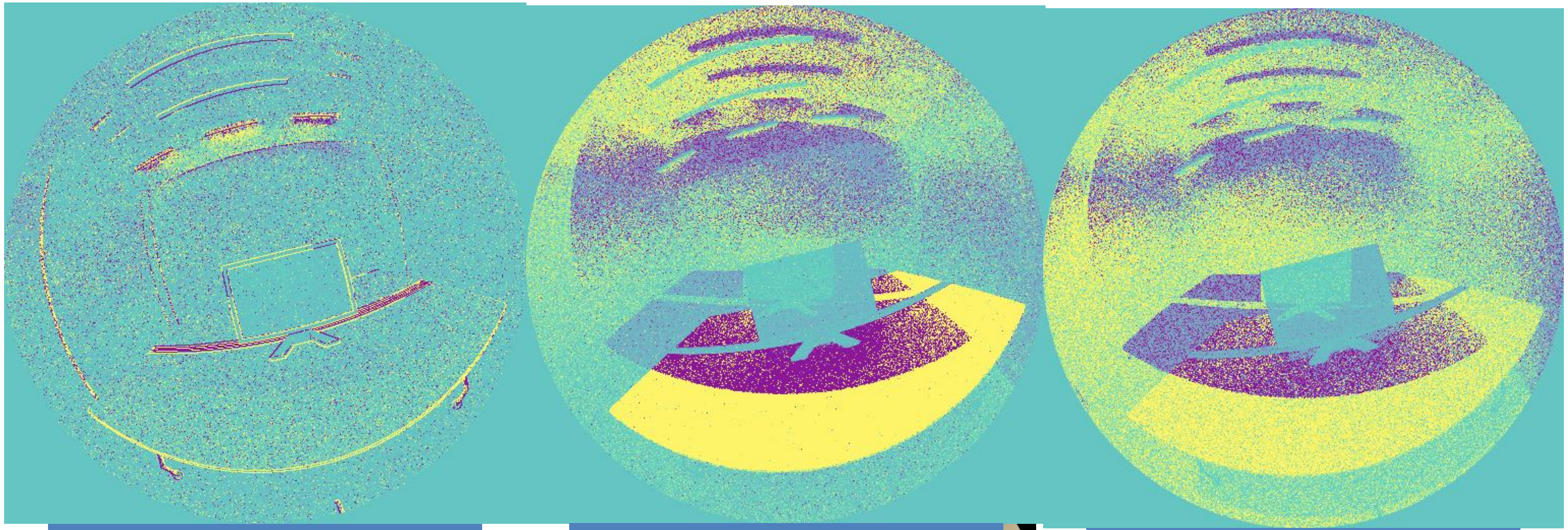
Goal: compute the rendering gradient



From Tzu-Mao’s SIGGRAPH slides

Review

Goal: compute the rendering gradient

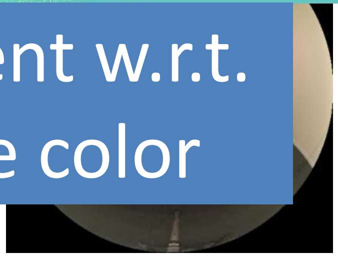


gradient w.r.t. camera pos

e

gradient w.r.t. table color

gradient w.r.t. light brightness

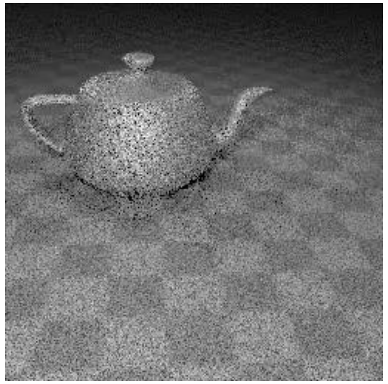


target

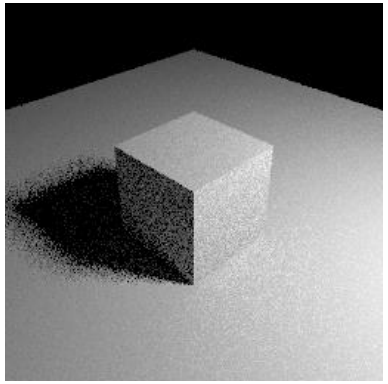
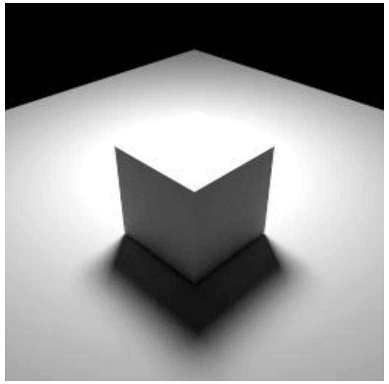
From Tzu-Mao's SIGGRAPH slides

Review

Camera pose & material




Light translation & rotation

Object translation


Camera pose

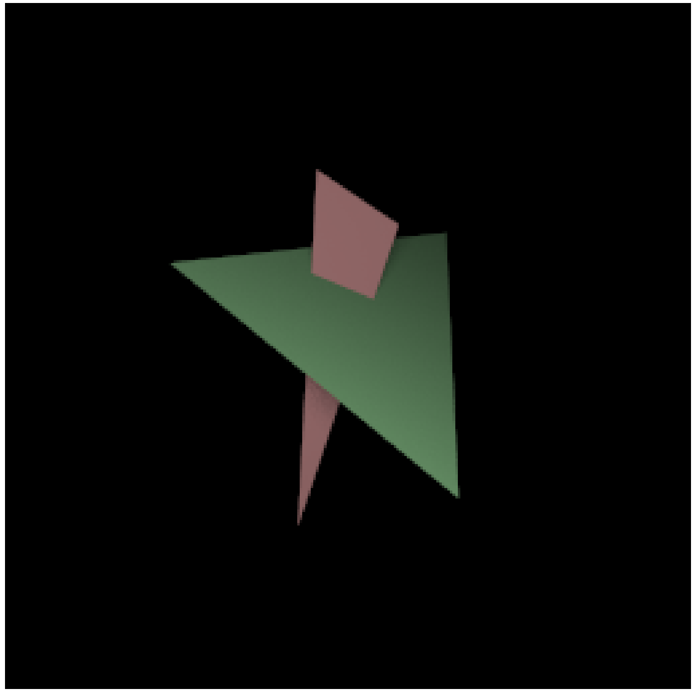

From Tzu-Mao’s SIGGRAPH slides

Our plan

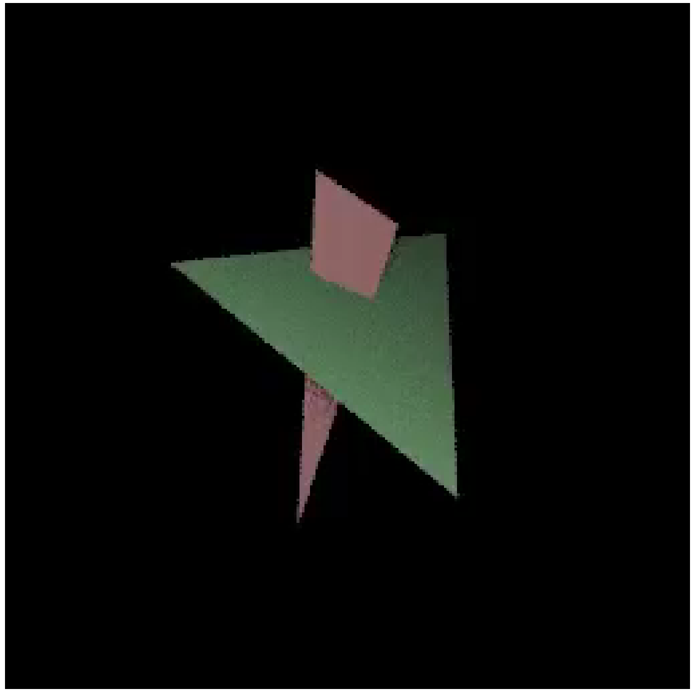
- Interpenetrating triangles
- Motion blur
- Pixel prediction
- Fast convergence by denoising

Interpenetrating two triangles

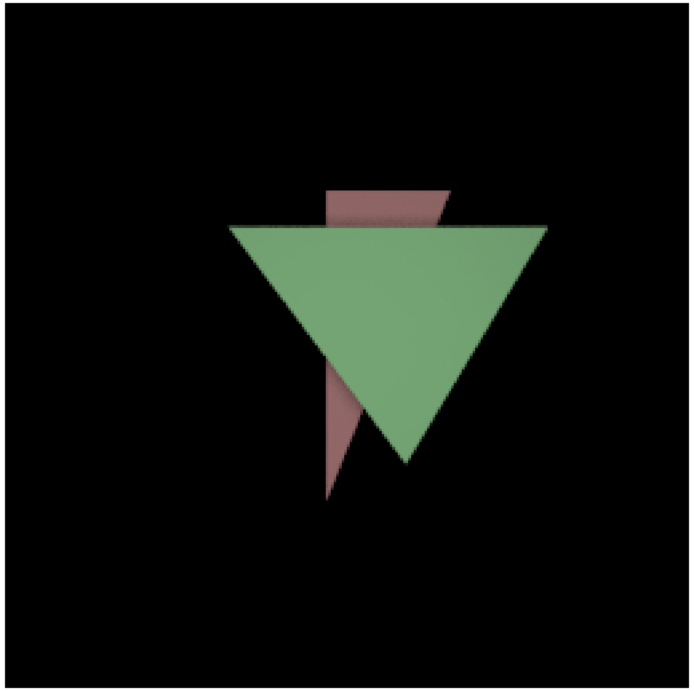
- Derivatives of interpenetrating objects require mesh splitting.
- However, strange thing happened:
  - Optimization of interpenetrating two triangles succeeded without mesh splitting



initial guess



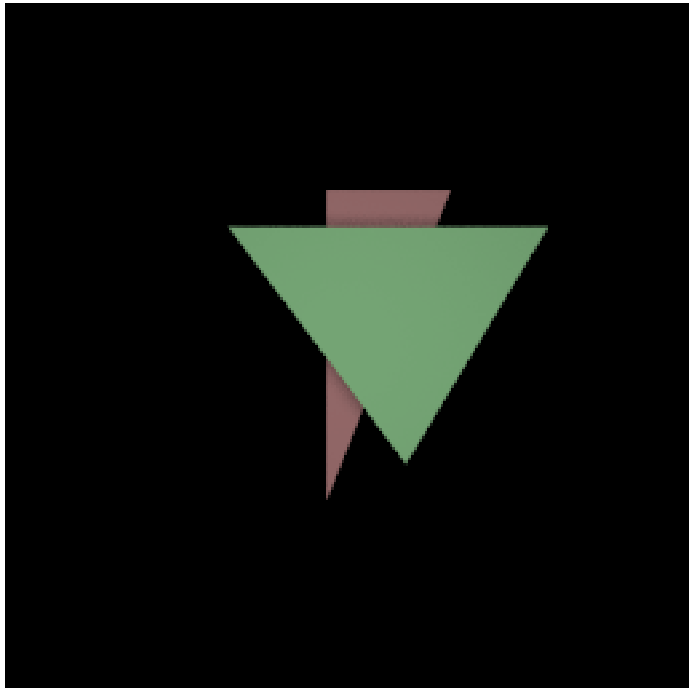
optimization



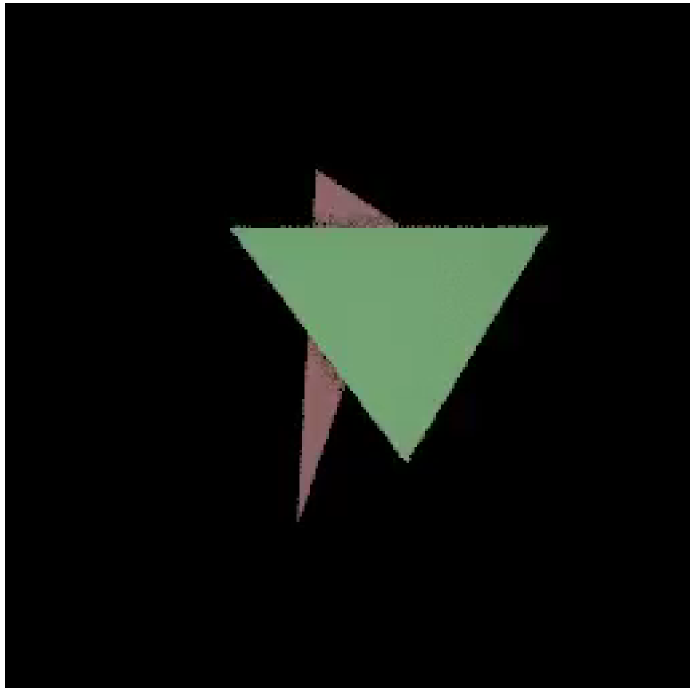
target

Interpenetrating two triangles

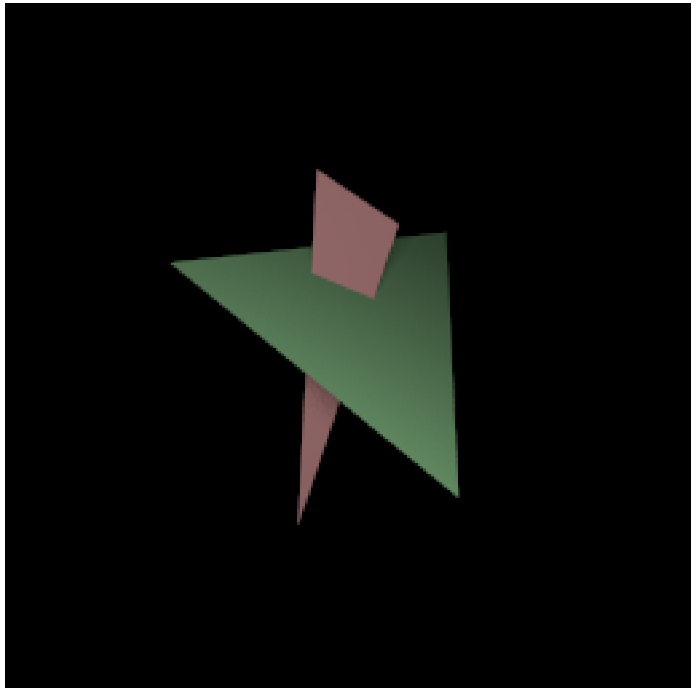
- Interestingly, reverse optimization fails!
- Unfortunately we couldn’t figure out why it works for forward optimization but fails for reverse situation.



initial guess



optimization



target

Motion blur

- Supporting motion blur effect was one of our plans

- Clarifying the goal, we arrived at two possibilities.
    - First, give motion blur effect of certain image by per-pixel gradient of moving objects.
    - Second, compute gradient of motion blurred image.
- First one is application of the technique. (similar to pixel prediction)
- Second one is way of improving technique.

Computing gradient of motion blurred image

- It was hard to formalize ..
- Python (high level, loss design and optimization) and C++ (low level, rendering and back propagation)
- C++ code was complicated..

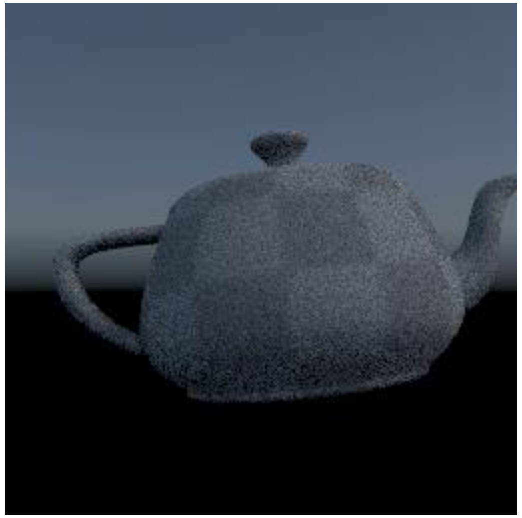
Pixel prediction by per-pixel gradient

- Two problems
  - 1. Technical Problem
    - Pytorch Autograd library does not support forward mode AD
  - 2. Intrinsic limitation
    - Gradient does not tell the future
    - Pixel prediction cannot really predict any kind of external influence.

Pixel prediction

$$I[i,j]\,(\Phi_1) \quad = \quad I[i,j](\Phi_0) \quad + \quad \nabla I[i,j](\Phi_0)\,\cdot\,d\Phi$$

? =

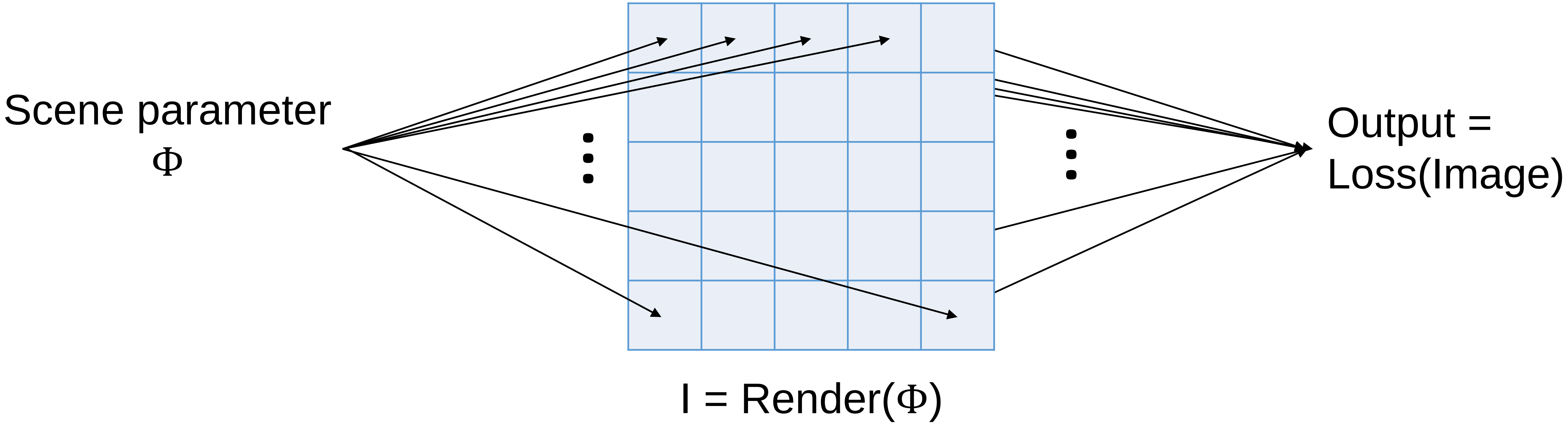


+

$\times\ d\Phi$

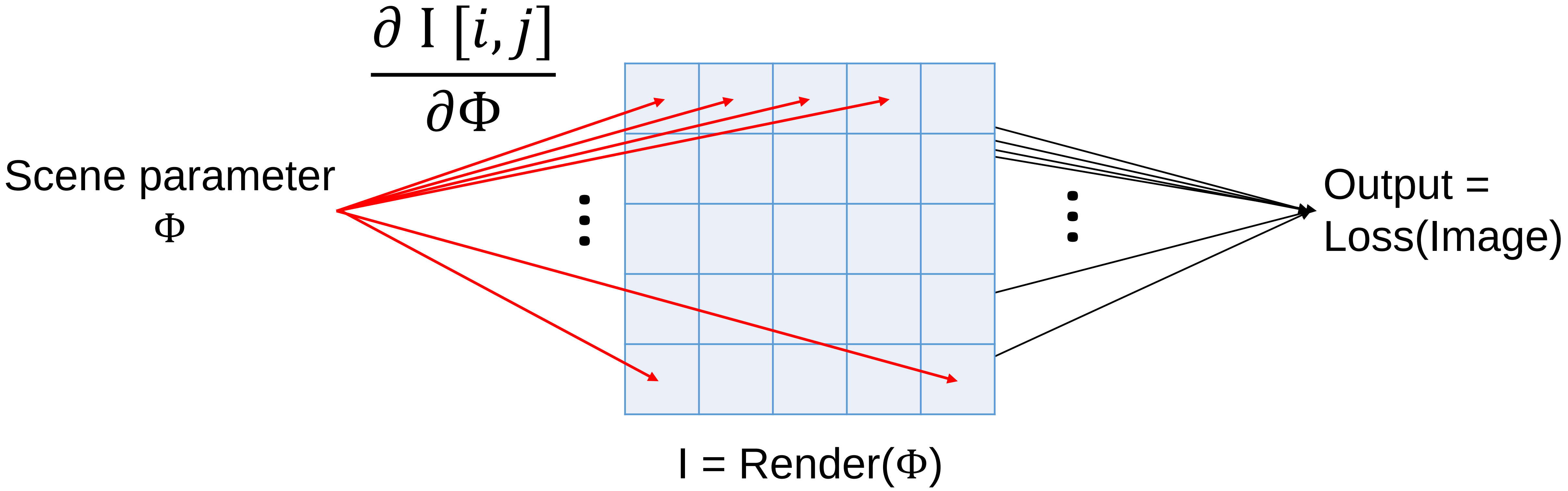
We need forward mode AD!

- Pytorch Autograd does not support forward mode.



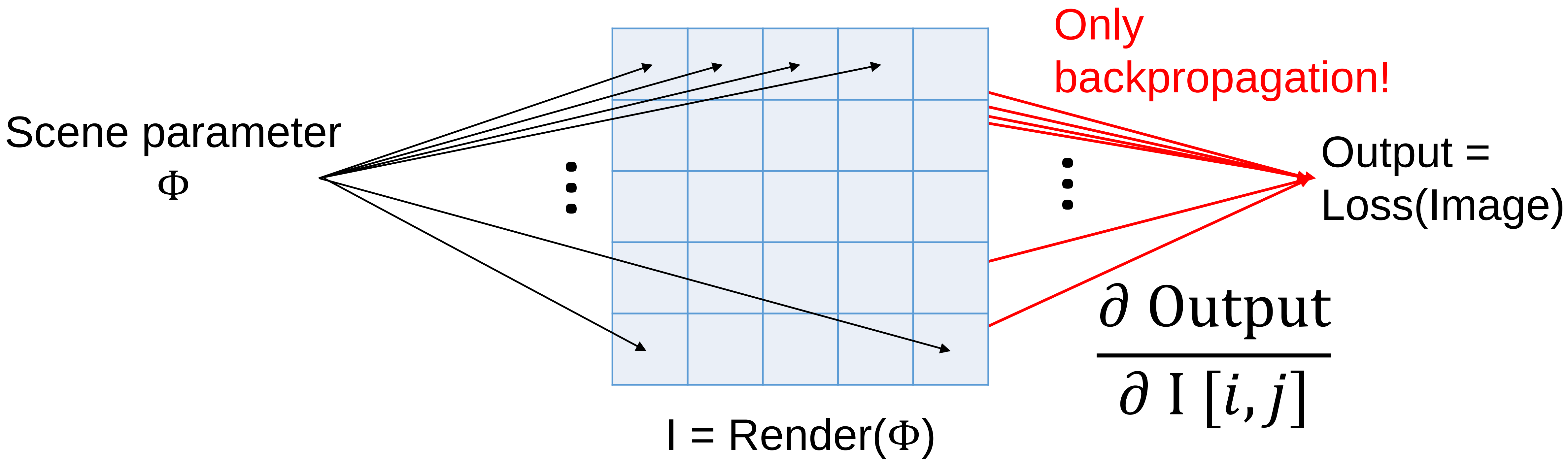
We need forward mode AD!

- Pytorch Autograd does not support forward mode.



We need forward mode AD!

- Pytorch Autograd does not support forward mode.

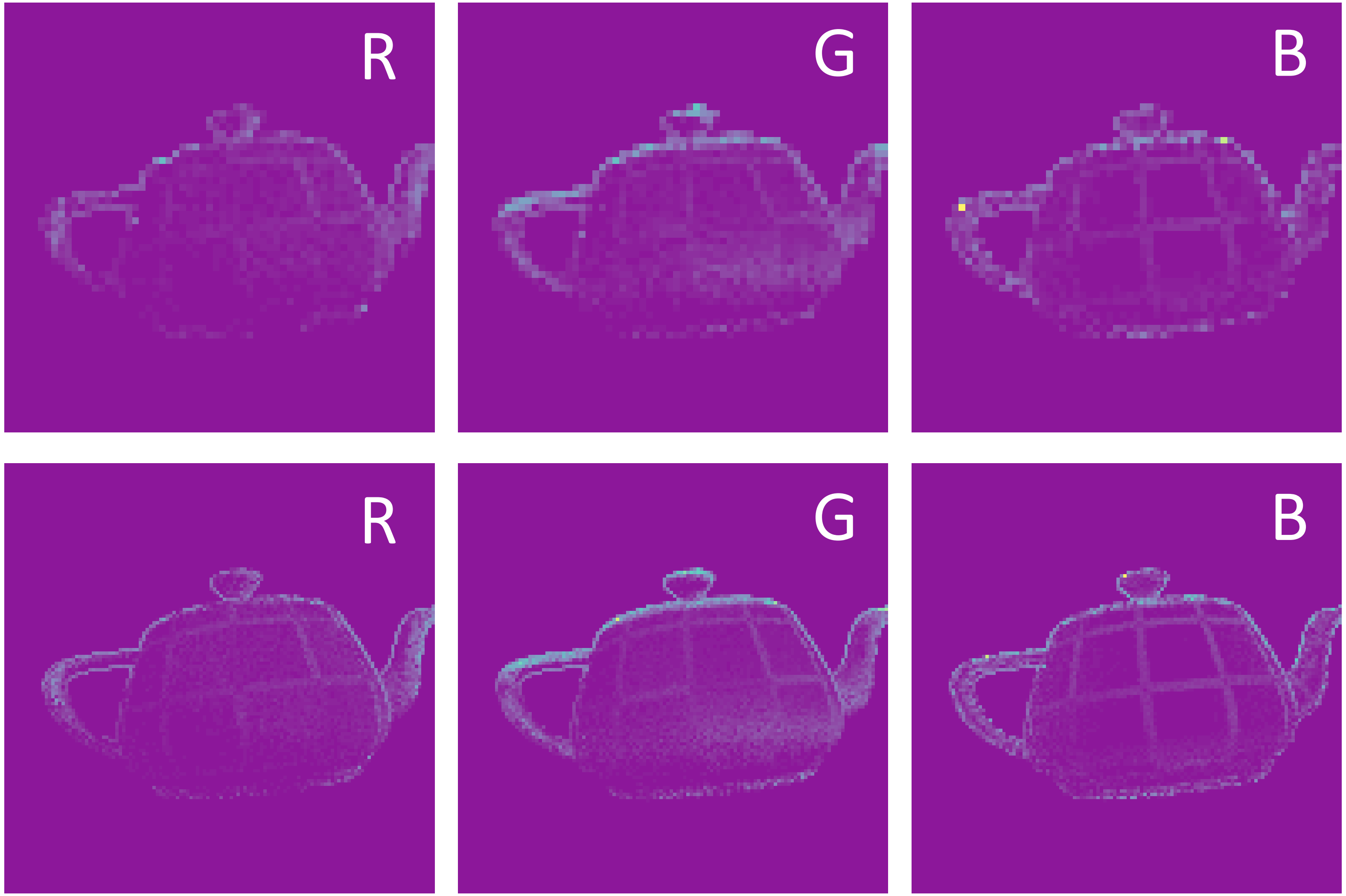


$$\frac{\partial\ \mathrm{Output}}{\partial\ \mathrm{I}\ [i, j]}$$

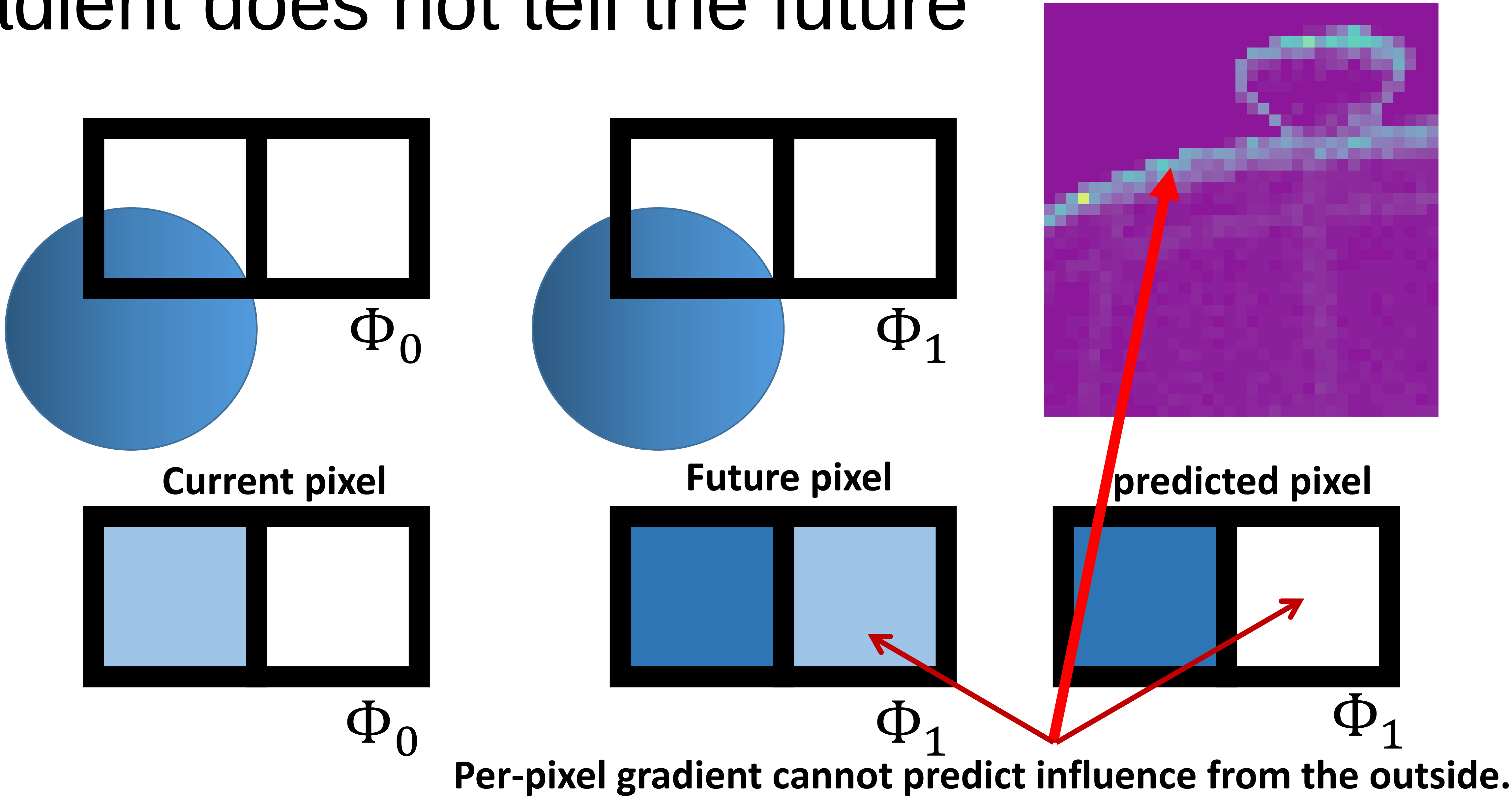
Naïve approach to per-pixel gradient

```
for i in range(img.size(0)):
    for j in range(img.size(1)):
        for k in range(img.size(2)):
            # print('({}, {}, {})  translation_params.grad: {}, euler_angles.grad: {}'.format(i, j, k, translation_param
            # mask = torch.zeros_like(img)
            # mask[i, j, k] = 1
            # img.backward(mask, retain_graph=True)
            img[i, j, k].backward(retain_graph=True)
            grad_img[i, j, k] = torch.sqrt(euler_angles.grad.pow(2).sum() + translation_params.grad.pow(2).sum())
            euler_angles.grad.data.zero_()
            translation_params.grad.data.zero_()
```

Naïve approach to per-pixel gradient



Gradient does not tell the future



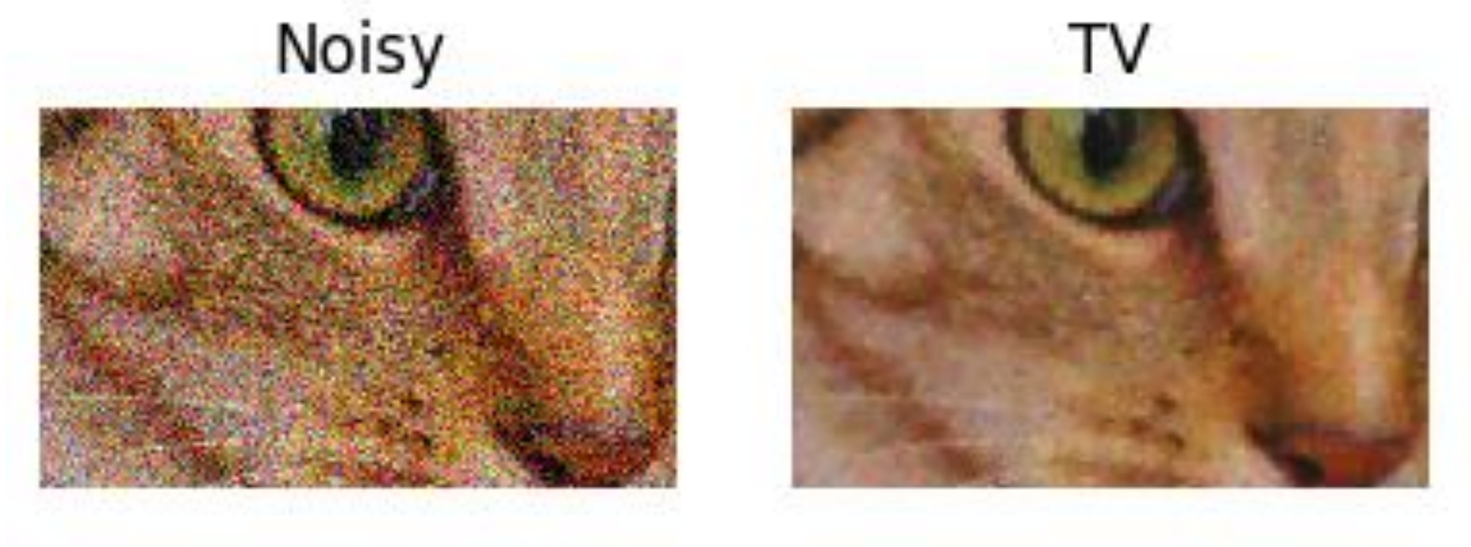
Denoising intermediate images

What if we use images with less noise?

Our hypothesis:

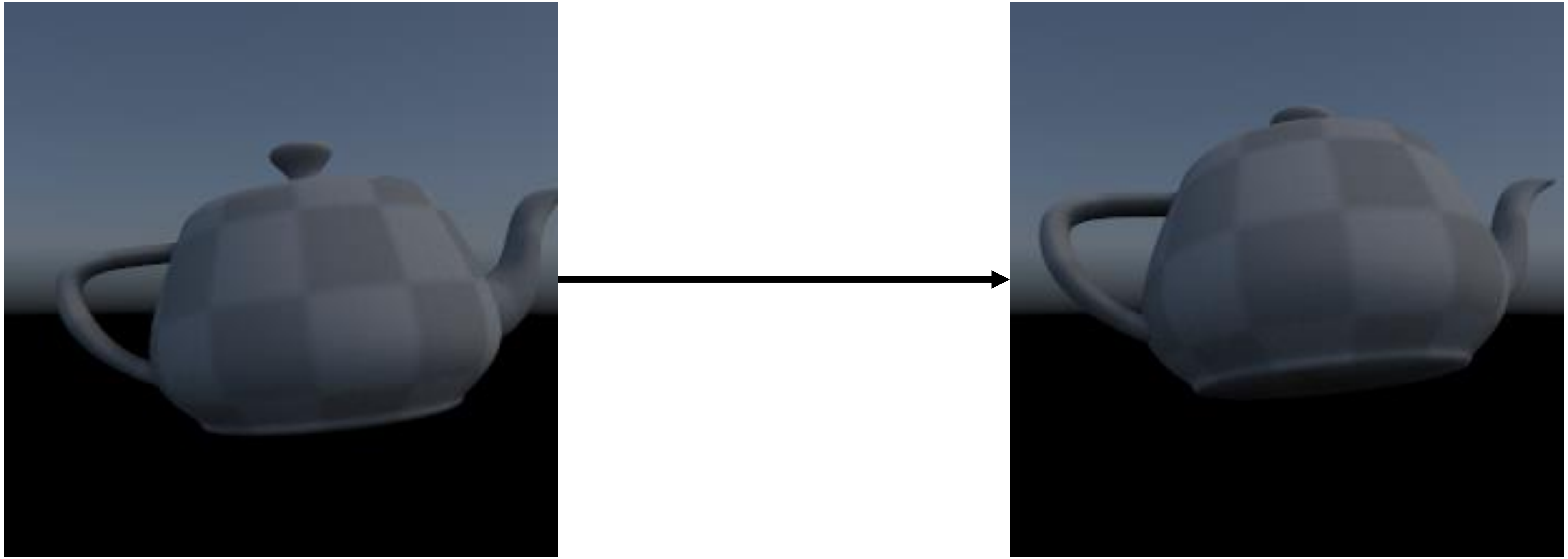
Less noise may increase speed of convergence.

Denoising Filter - tv Filter



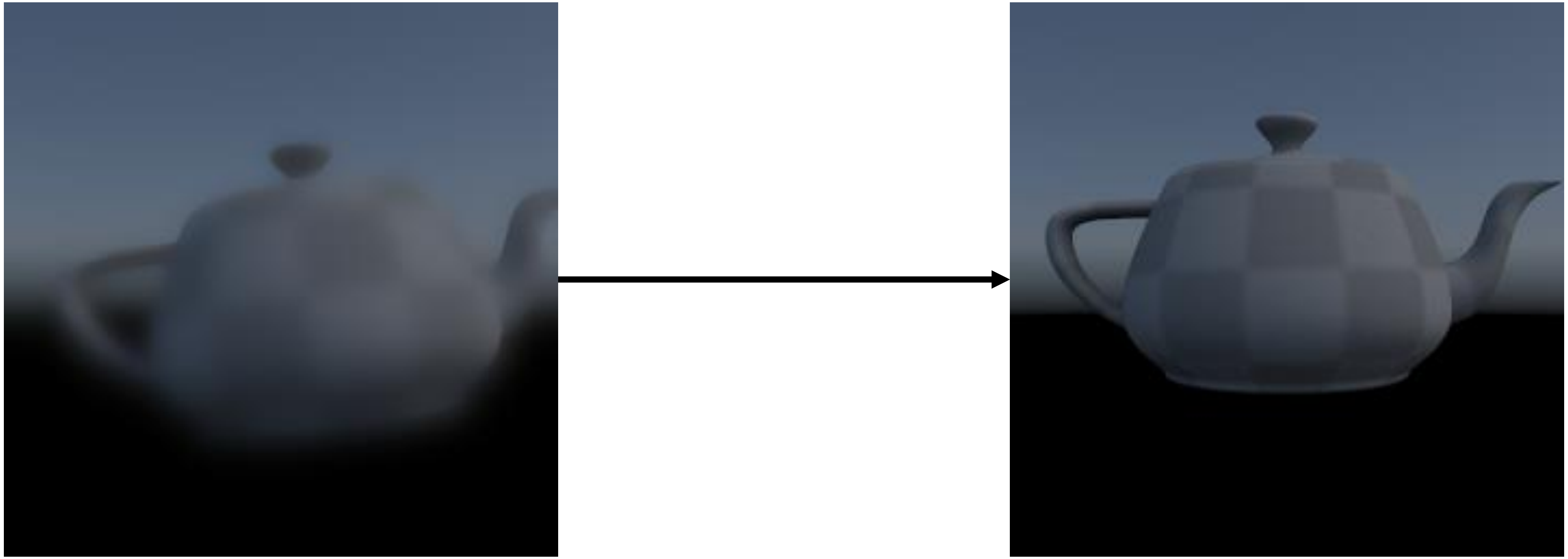

```python
if denoiseOn:
    denoisedImage=denoise_tv_chambolle(img.data.numpy(),denoiseWeight,multichannel=True)
    img.data=torch.tensor(denoisedImage)
```

Pose estimation



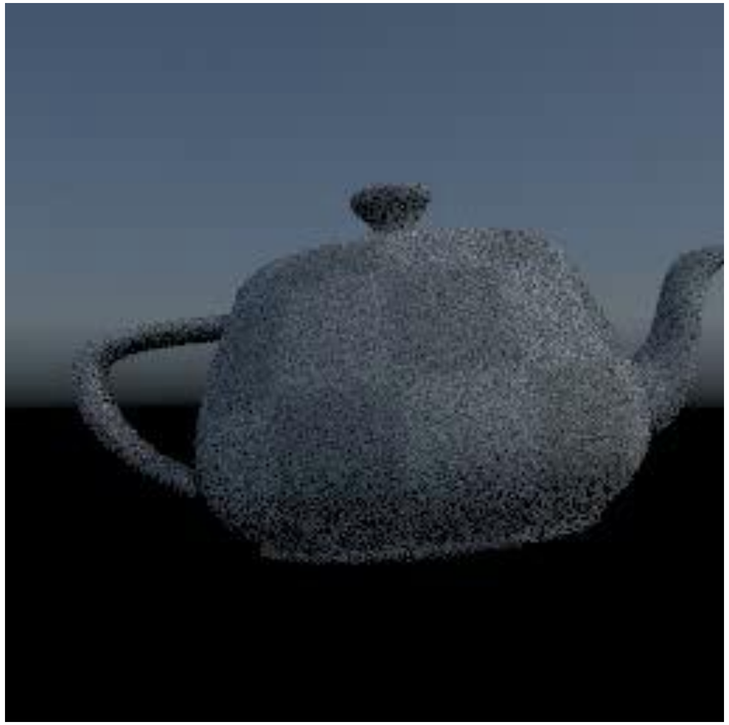
Local Minimum

Pose estimation



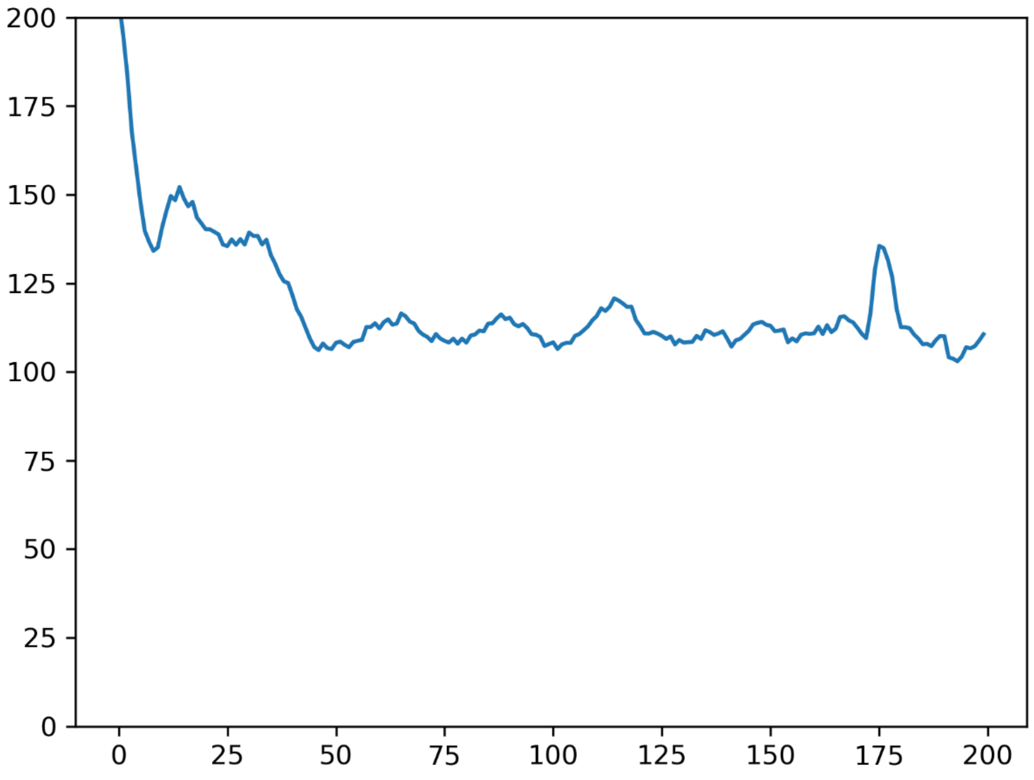
Pose estimation

Denoised(Weight=0.1)

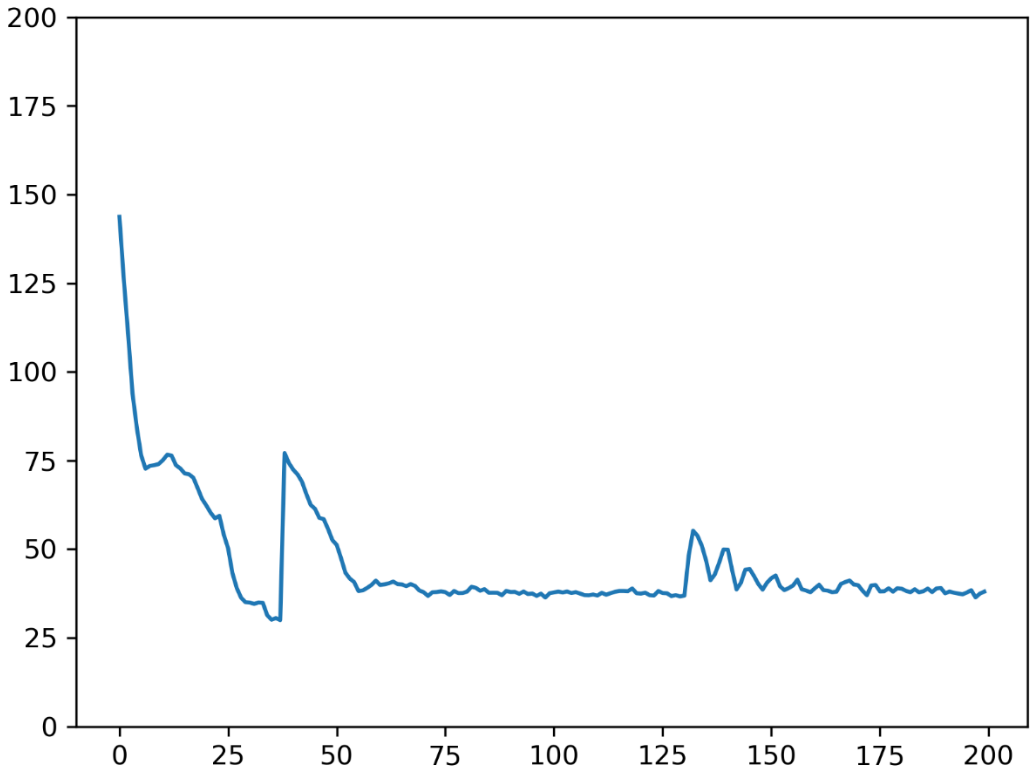


Non-Denoised

Pose estimation - Loss

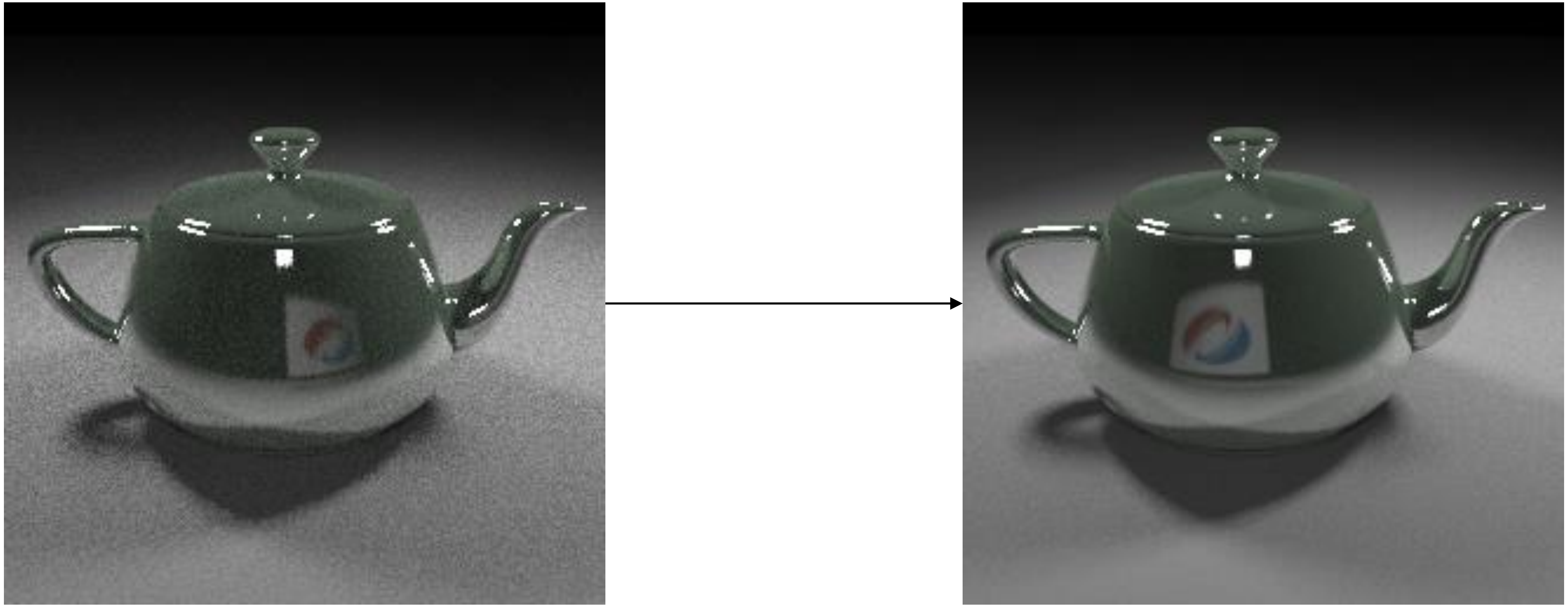


Non-denoised

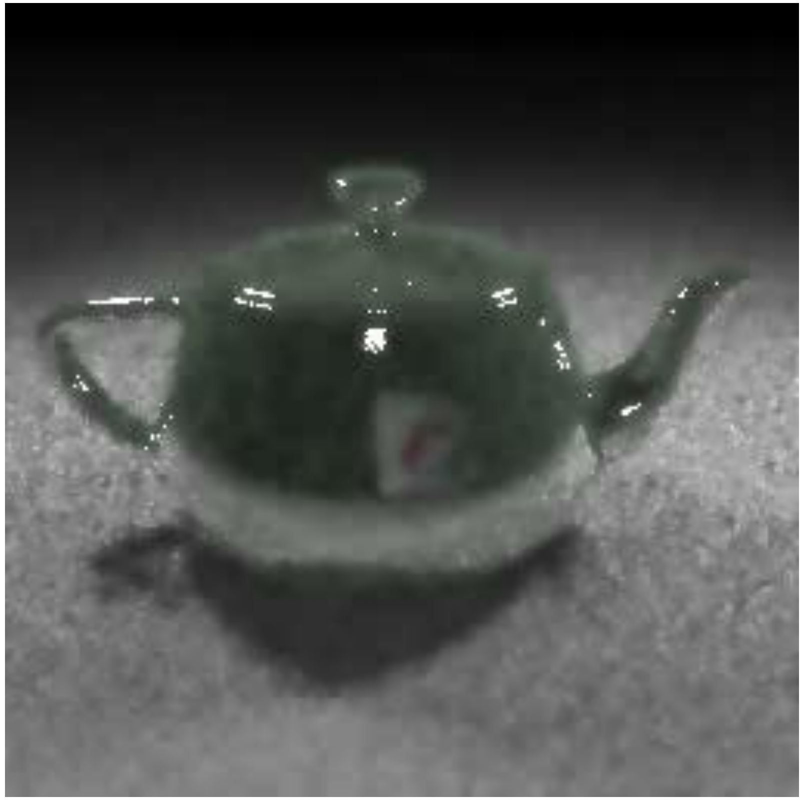


Denoised

Specular Image



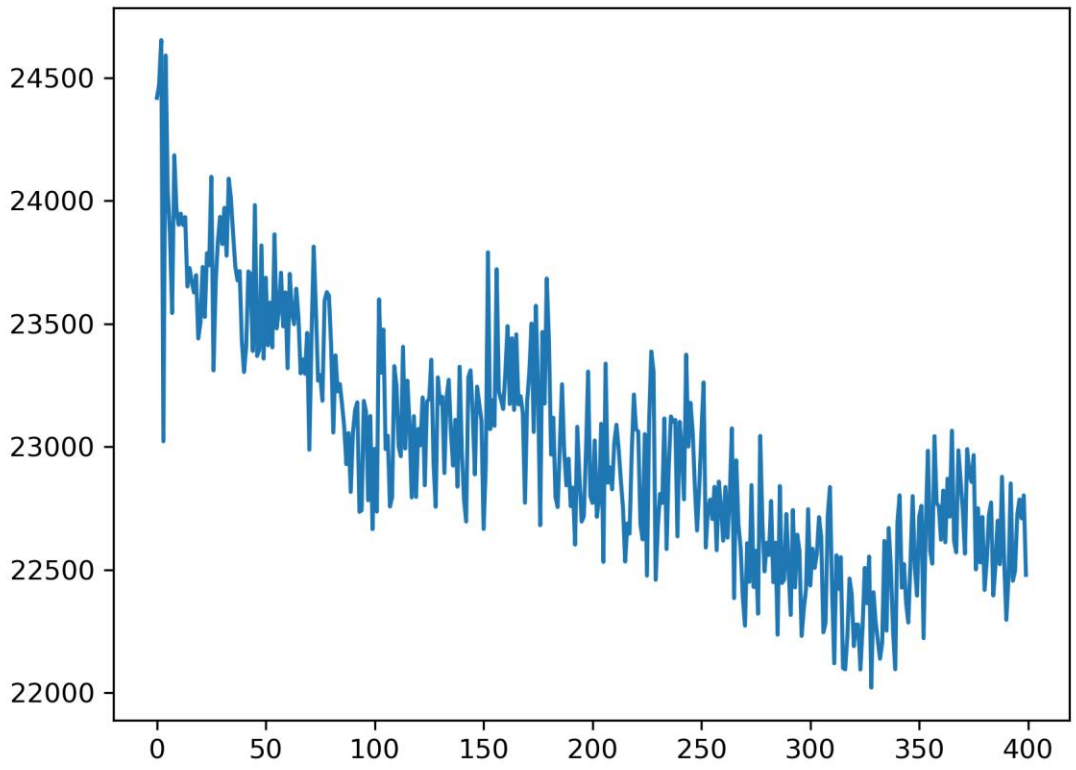
Specular Image



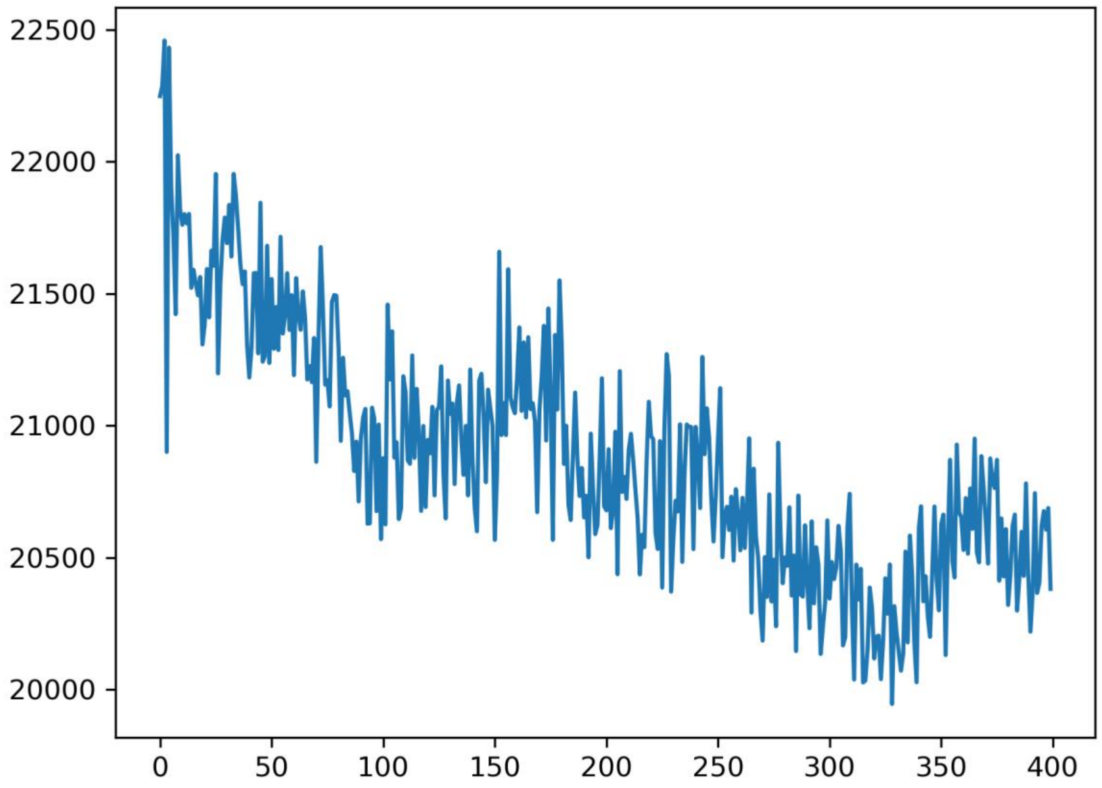
Denoised(Weight=0.1)

Non-denoised

Huge fluctuation of loss



Denoised



Non - denoised

Why?

Huge fluctuation of loss

$$Loss = Loss(\Phi) + \epsilon(t, \Phi)$$

Noise due to small sample numbers significantly contributes to loss than the actual difference.

Time Consumption

	Denoised Version	Non-denoised Version
Pose Estimation	717	572
Specular Image	1362	1150

Summary

- Interpenetrating triangles
  - We found interesting example that this paper couldn’t succeed to optimize
- Motion blur
  - We couldn’t try this due to a lack of time
- Pixel prediction
  - Technical issue: PyTorch doesn’t support forward mode AD
  - Intrinsic issue: Per pixel gradient doesn’t tell the situation outside the pixel
- Fast convergence by denoising
  - Our denoising method prevents falling into a local minimum
  - Also reduces convergence time
  - We had a discussion about fluctuating loss graph

Contribution

- Things that we did altogether
  - Presentation prepare, Discussion on the issues
- Hangyeol Yu
  - Theoretical background, (motion blur and pixel prediction), library build
- Hyunwoo Lee
  - Implementation, (denoising), experiment
- Juho Park
  - Topic suggestion, (intersecting two triangles issue), library build