Computer Vision Homework #1 Report

Homework Settings

> Given linearized render of an object from one view with multiple light direction and light intensity, recover the normal and albedo image of the object using a) a simple linear Lambertian model and b) a simple linear Lambertian model with highlight and shadow removal.

Data specification

- 96 images of an object with a resolution of 612 * 512, stored as 16-bit `png`
- A mask file for valid object pixels
- Light directions and light intensities used for all rendered objects in `txt`

Code structure

This code is implemented in `python`, powered by `pytorch`, other required packages are listed in `requirements.txt`

Assume you've got a valid `python` installation, run:

```
pip install -r requirements.txt # to install dependencies
```

We actually recommend installing `pytorch` with `cuda` support through `conda` (possibly with the help of `mamba`), but that hustle is beyond the scope of this small project... (just like the hustle of installing `matlab`, luckily we don't need `matlab`)

- `main.py`: for the main logic of the algorithm implemented in this project
- `utils.py`: for utility functions used by the main program
- `run_exps.py`: a script for running all experiment settings on all provided datasets (given a `--data_root` directory)

Both `main.py` and `run_exps.py` provides command line interfaces, run them with `--help` to see help to the command line arguments. Optionally, see `run_exps.py` for example use cases of `main.py`.

Method

Method Description

We implemented our algorithms in `pytorch`, considering that the noisy data and abundant input as a perfect candidate for optimization based algorithms.

Another benefit of `pytorch` is we are able to easily implement parallel algorithms running on GPU to make these largely parallel optimization finish in time. (performance-aware)

One could easily choose to use some closed form least square implementation given the simplicity of the Lambertian model. However, due to the noisy nature of the rendering process (We assume the provided images were rendered with a path tracer?), closed form solution could easily fall into implementation pitfalls (like non-singular matrices etc.)

We tested three versions of the simple photometric algorithm:

- A vanilla version that uses linear Lambertian model to reconstruct the images.
    - We use mask to compute indices for valid pixels to optimize, we set this number to `P`, note that this is shared across lights.
    - We set `normal` and `albedo` as optimizable to perform gradient descent on them (both 3D tensors).
    - We activate raw `normal` values with an `normalize` operation to make it represent a 3D direction (2 DoF). We chose this 3D Cartesian representation for easier implementation.
    - We activate raw `albedo` values with a `softplus` activation to make them non-negative (no limit on upper bound). This achieves better results than a simple `relu` since `relu` cuts gradients to negative values.
    - We perform forward pass on all pixels to be optimized all at once (~1G VRAM).
    - We use an `Adam` optimizer with a learning rate of `1e-1` and optimize for 5k iterations (takes 15 seconds on a 3090).
    - We do not clip rendered `rgb` values during training, since the initialized normal maybe incorrect, producing incorrect unshaded area. Although this violates the Lambert shading model, it's better for optimization. (And simpler to implement).
- A simple global pixel value sorting version to remove effects of shadows and highlights.
    - We sort all pixel values (masked) and get indices of valid pixels within masked pixels, note that this is **not** shared across lights.

  - The reason for computing indices instead of using the original `mask` is that CPU code and GPU code are executed asynchronously as long as no explicit sync signals or syncing required operation is performed (a rule of thumb: as long as we won't need GPU' s output for launching the next `cuda` kernel, we are good. `cuda` kernel launches require sizes to be predetermined.). And taking indices from a mask is such **syncing required operation** since future `cuda` kernel launches will need the size of the indices (computed from the `mask` tensor).
  - We use an `--rtol_hi` and `--rtol_lo` parameter to control ratio to discard the pixel values. We set this value to 0.05 (5%) and 0.01 (1%) respectively in all experiments.
  - We convert the `rgb` values to grayscale using:

    ```
    0.299 * rgb[..., 0] + 0.587 * rgb[..., 1] + 0.114 * rgb[..., 2]
    ```

    before sorting them. We perform all these operations in parallel to all pixels on GPU.
  - Note that shadow is almost aways pitch black. But highlights differ in actual value. Thus we use different `rtol` for them.
- A self correction algorithm for better (sometimes) photometric-stereo.
  - The gist of the algorithm is inspired by: Self-Correction Human Parsing.
  - **We start from the previous optimization, find pixels that violates the Lambertian model, discard them and continue training for a few iterations.**
  - By repeating this process, we can filter out those points that violates the prior automatically (in a **self-supervised** way).
  - The `--rtol_hi` and `--rtol_lo` variables are reused, but this time they denote the difference in grayscale between the rendered pixels and the actual pixel color for highlights and shadows respectively (set to 0.1 and 0.2 during all experiments).
  - `--rtol_hi` and `--rtol_lo` are linearly annealed during **self-correction** for stable optimization.
  - Experiments show that our **self-correction photometric stereo** algorithm outperforms baseline methods (~~although only sometimes... and with quite a lot of noise lying around...~~ :).
  - Noise stills exists even though we have developed an annealing scheme and large number of self-correction iterations, this is possibly due to the fact that some of the shadowed or highlighted pixels does not have enough valid Lambertian observations, where they are neither occluded, facing away from the light or facing directly at the most reflective angle.
  - We reuse the code structure of the previous two implementations thus the code change is quite minimal:

    ```
    render = lambertian(dirs, ints)
    diff = rgb_to_gray(rgbs) - rgb_to_gray(render)  # N, P
    ```

    Only these two lines need to be added to facilitate one round of self-correction.
  - The reason for this different values in setup is that highlights often vary quite a bit in intensity while shadows are almost always pitch black.

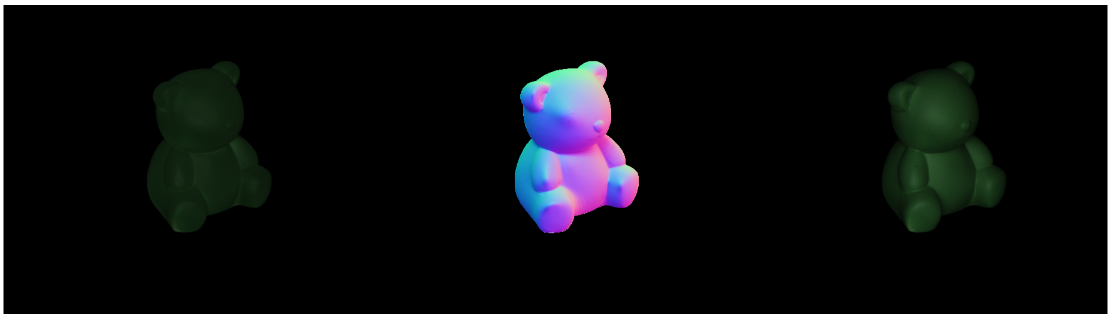
Implementation Detail

- We describe the optimizer settings in the previous Method Description section since we consider **using optimization method** as part of our method choice.
- When loading images from disk, we use the `ThreadPool` module for multi-threaded (also multi-processed) loading of the `png` files. It is not clear how much improvements one can get from multi-threaded loading from a local single disk setup, but it certainly performs much better on a network mounted drive (like a `nas` we're using).
- We perform our optimization in single precision floating points.
- We normalize the pixel values to `[0, 1]` after reading them from disk. (by dividing with 65535 (max of 16 bit `png`)).
- We clip the rendered pixels to non-negative values before saving them on disk.
- We perform 300 **self-correction** steps based on the results of the **simple highlight-shadow removal** algorithm.
- We implement all **self-correction** loops inside one python call for better GPU utilization (almost always busy).
- We introduce an annealing algorithm for the **self-correction** ratio for a stable optimization process.
- Re-rendered light direction is `[0, 0, 1]` , intensity is `[2, 2, 2]`

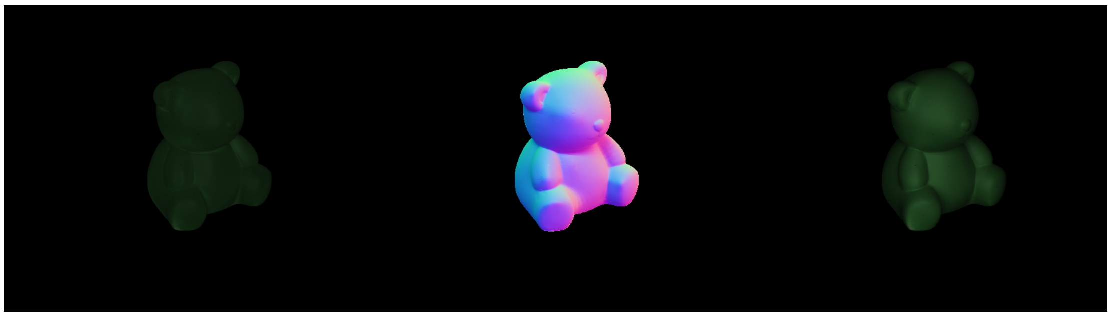
Experiment Results

`bear`

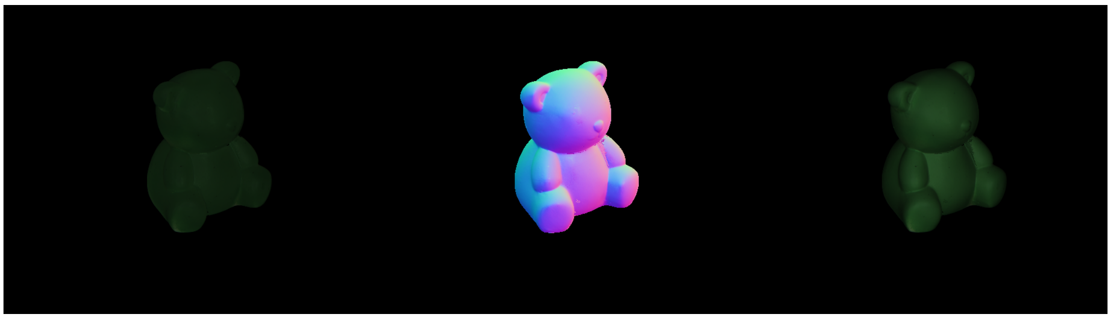
Lambertian



Global Sort

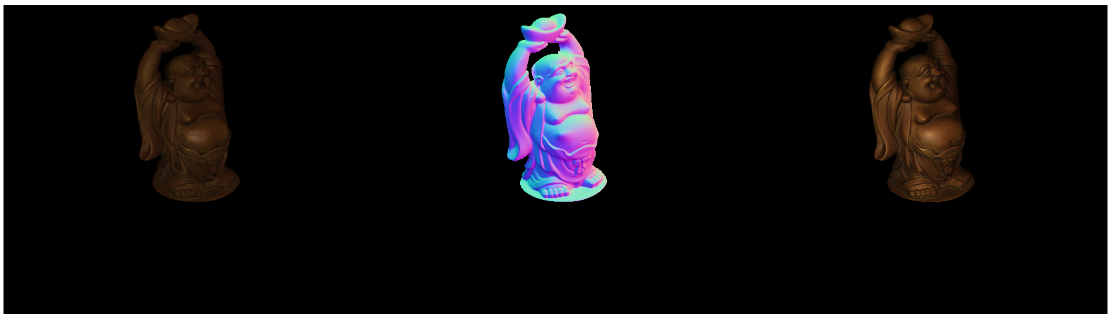


Self-Correction

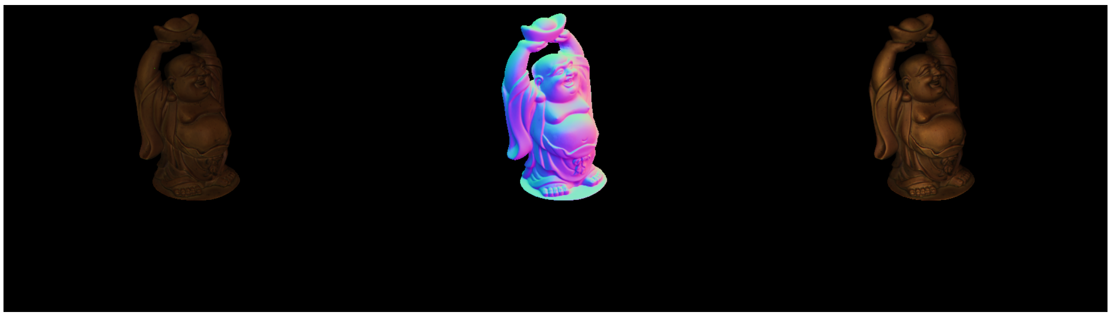


budda

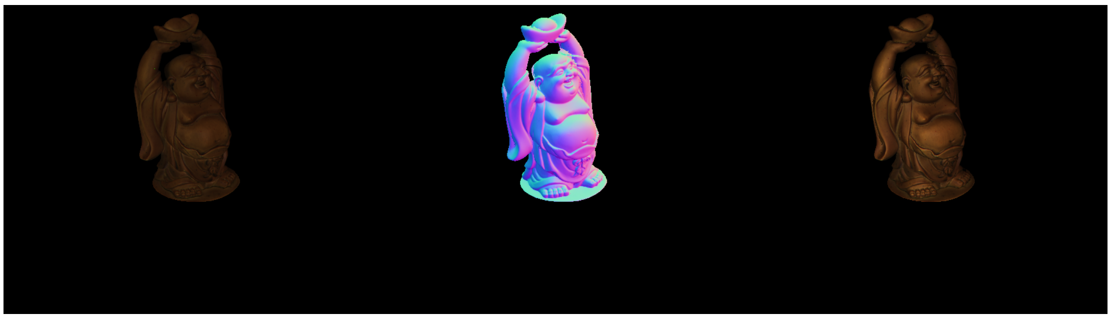
Lambertian



Global Sort

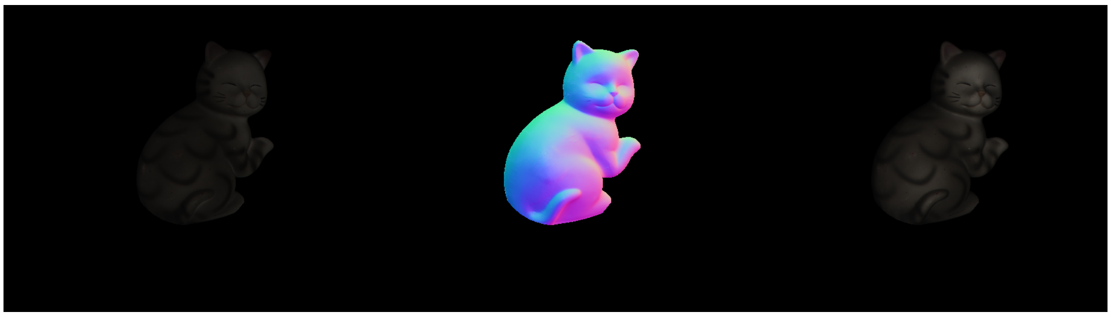


Self-Correction

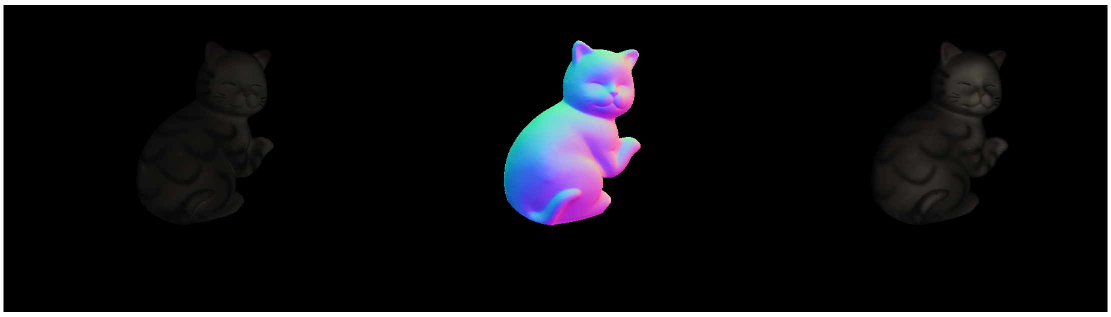


`cat`

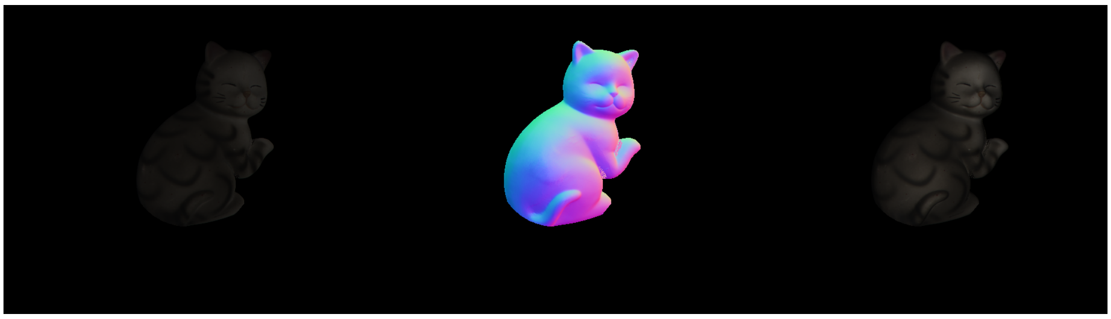
Lambertian



Global Sort

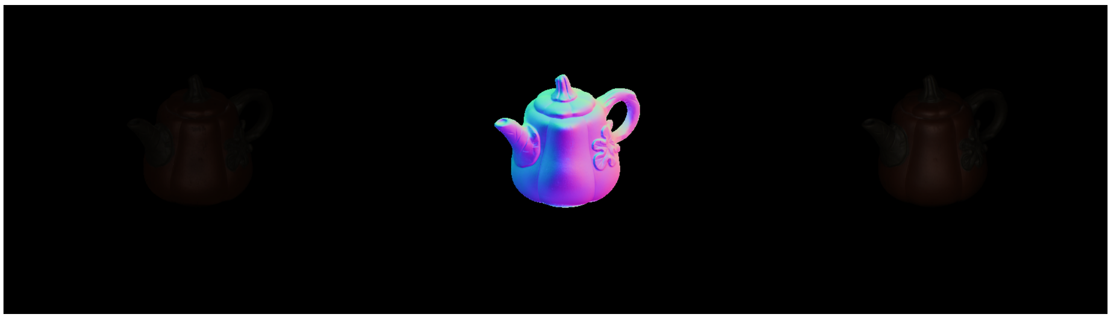


Self-Correction

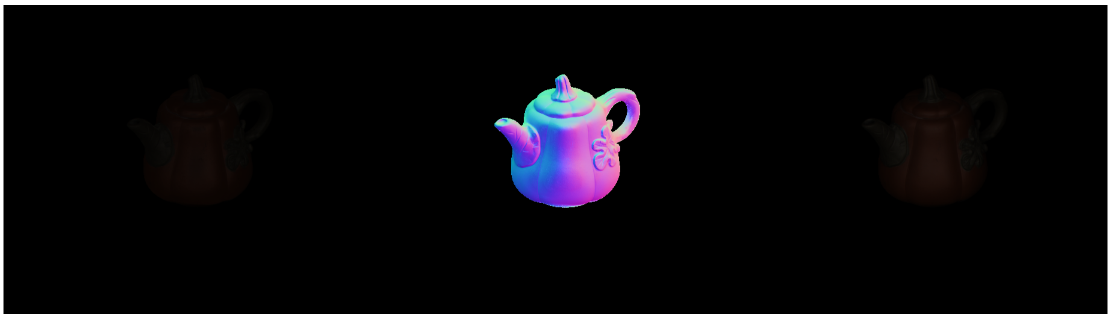


`pot`

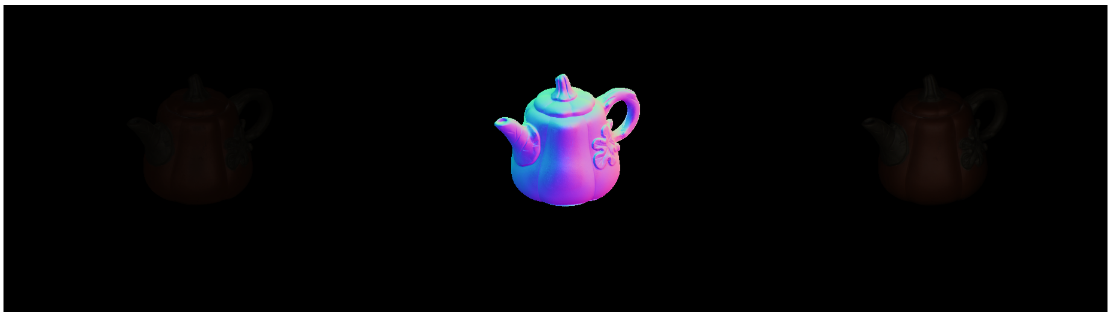
Lambertian



Global Sort



Self-Correction



Thoughts

Photometric stereo using simple Lambertian model is an intrinsically ill-posed problem since the Lambertian model does not cover all aspects of the rendering process and the material properties possibly exhibited in the rendered images. And some pixels does not contain enough valid information to recover even a simple Lambertian model.

We assume the provided data is rendered by a noisy path tracer. And we have utilized two key attributes in the data during optimizaiton:

1. Shadows are almost always black.
2. When the rendering results of a Lambertian model differs from the actual pixel color, it usually indicates highlights or shadow (things that a simple Lambertian model cannot explain)

Thus two of the most obvious improvements we could do are:

1. Improve the rendering process of the recovery algorithm: by performing **shape-from-shading** on the result of photometric stereo, we can get a rough (or highly accuracy in case we use dense lighting) shape estimation from the provided data. Then, by rendering the Lambertian model with the **visibility** term in the rendering equation, we can solve for shadows. (At least find an accurate filter for shadowed pixels)
2. Improve the material model (and possibly add to the rendering process): by modeling with a full blown BRDF/BSDF (or some simplified version like Microfacet) model, we can accurately render specular highlights (assuming accurate material parameters) instead of being limited by the simplicity of the Lambertian model.

One of the most important thing in the implementation of these algorithms is the use of a GPU. It speeds things up quite significantly and makes iteration on the algorithm (like making various improvements) a extremely satisfactory process.