

---

## MultiNeRF: A Code Release for Mip-NeRF 360, Ref-NeRF, and RawNeRF

*This is not an officially supported Google product.*

This repository contains the code release for three CVPR 2022 papers: Mip-NeRF 360, Ref-NeRF, and RawNeRF. This codebase was written by integrating our internal implementations of Ref-NeRF and RawNeRF into our mip-NeRF 360 implementation. As such, this codebase should exactly reproduce the results shown in mip-NeRF 360, but may differ slightly when reproducing Ref-NeRF or RawNeRF results.

This implementation is written in JAX, and is a fork of mip-NeRF. This is research code, and should be treated accordingly.

### Setup

```
1 # Clone the repo.
2 git clone https://github.com/google-research/multinerf.git
3 cd multinerf
4
5 # Make a conda environment.
6 conda create --name multinerf python=3.9
7 conda activate multinerf
8
9 # Prepare pip.
10 conda install pip
11 pip install --upgrade pip
12
13 # Install requirements.
14 pip install -r requirements.txt
15
16 # Manually install rmbrua's `pycolmap` (don't use pip's! It's
   different).
17 git clone https://github.com/rmbrua/pycolmap.git ./internal/pycolmap
18
19 # Confirm that all the unit tests pass.
20 ./scripts/run_all_unit_tests.sh
```

You'll probably also need to update your JAX installation to support GPUs or TPUs.

### Running

Example scripts for training, evaluating, and rendering can be found in [scripts/](#). You'll need to change the paths to point to wherever the datasets are located. Gin configuration files for our model and some ablations can be found in [configs/](#). After evaluating on the test set of each scene in one of

---

the datasets, you can use `scripts/generate_tables.ipynb` to produce error metrics across all scenes in the same format as was used in tables in the paper.

## OOM errors

You may need to reduce the batch size (`Config.batch_size`) to avoid out of memory errors. If you do this, but want to preserve quality, be sure to increase the number of training iterations and decrease the learning rate by whatever scale factor you decrease batch size by.

## Using your own data

Summary: first, calculate poses. Second, train MultiNeRF. Third, render a result video from the trained NeRF model.

1. Calculating poses (using COLMAP):

```
1 DATA_DIR=my_dataset_dir
2 bash scripts/local_colmap_and_resize.sh ${DATA_DIR}
```

2. Training MultiNeRF:

```
1 python -m train \
2   --gin_configs=configs/360.gin \
3   --gin_bindings="Config.data_dir = '${DATA_DIR}'" \
4   --gin_bindings="Config.checkpoint_dir = '${DATA_DIR}/checkpoints'" \
5   --logtostderr
```

3. Rendering MultiNeRF:

```
1 python -m render \
2   --gin_configs=configs/360.gin \
3   --gin_bindings="Config.data_dir = '${DATA_DIR}'" \
4   --gin_bindings="Config.checkpoint_dir = '${DATA_DIR}/checkpoints'" \
5   --gin_bindings="Config.render_dir = '${DATA_DIR}/render'" \
6   --gin_bindings="Config.render_path = True" \
7   --gin_bindings="Config.render_path_frames = 480" \
8   --gin_bindings="Config.render_video_fps = 60" \
9   --logtostderr
```

Your output video should now exist in the directory `my_dataset_dir/render/`.

See below for more detailed instructions on either using COLMAP to calculate poses or writing your own dataset loader (if you already have pose data from another source, like SLAM or RealityCapture).

---

## Running COLMAP to get camera poses

In order to run MultiNeRF on your own captured images of a scene, you must first run COLMAP to calculate camera poses. You can do this using our provided script `scripts/local_colmap_and_resize.sh`. Just make a directory `my_dataset_dir/` and copy your input images into a folder `my_dataset_dir/images/`, then run:

```
1 bash scripts/local_colmap_and_resize.sh my_dataset_dir
```

This will run COLMAP and create 2x, 4x, and 8x downsampled versions of your images. These lower resolution images can be used in NeRF by setting, e.g., the `Config.factor = 4` gin flag.

By default, `local_colmap_and_resize.sh` uses the OPENCV camera model, which is a perspective pinhole camera with  $k_1$ ,  $k_2$  radial and  $t_1$ ,  $t_2$  tangential distortion coefficients. To switch to another COLMAP camera model, for example OPENCV\_FISHEYE, you can run

```
1 bash scripts/local_colmap_and_resize.sh my_dataset_dir OPENCV_FISHEYE
```

If you have a very large capture of more than around 500 images, we recommend switching from the exhaustive matcher to the vocabulary tree matcher in COLMAP (see the script for a commented-out example).

Our script is simply a thin wrapper for COLMAP—if you have run COLMAP yourself, all you need to do to load your scene in NeRF is ensure it has the following format:

```
1 my_dataset_dir/images/    <--- all input images
2 my_dataset_dir/sparse/0/  <--- COLMAP sparse reconstruction files (
    cameras, images, points)
```

## Writing a custom dataloader

If you already have poses for your own data, you may prefer to write your own custom dataloader.

MultiNeRF includes a variety of dataloaders, all of which inherit from the base Dataset class.

The job of this class is to load all image and pose information from disk, then create batches of ray and color data for training or rendering a NeRF model.

Any inherited subclass is responsible for loading images and camera poses from disk by implementing the `_load_renderings` method (which is marked as abstract by the decorator `@abc.abstractmethod`). This data is then used to generate train and test batches of ray + color data for feeding through the NeRF model. The ray parameters are calculated in `_make_ray_batch`.

---

**Existing data loaders** To work from an example, you can see how this function is overloaded for the different dataloaders we have already implemented:

- Blender
- DTU dataset
- Tanks and Temples, as processed by the NeRF++ paper
- Tanks and Temples, as processed by the Free View Synthesis paper

The main data loader we rely on is LLFF (named for historical reasons), which is the loader for a dataset that has been posed by COLMAP.

**Making your own loader by implementing `_load_renderings`** To make a new dataset, make a class inheriting from `Dataset` and overload the `_load_renderings` method:

```
1 class MyNewDataset(Dataset):
2     def _load_renderings(self, config):
3         ...
```

In this function, you **must** set the following public attributes:

- `images`
- `camtoworlds`
- `pixtocams`
- `height, width`

Many of our dataset loaders also set other useful attributes, but these are the critical ones for generating rays. You can see how they are used (along with a batch of pixel coordinates) to create rays in `camera_utils.pixels_to_rays`.

### Images

`images` = [N, height, width, 3] numpy array of RGB images. Currently we require all images to have the same resolution.

### Extrinsic camera poses

`camtoworlds` = [N, 3, 4] numpy array of extrinsic pose matrices. `camtoworlds[i]` should be in **camera-to-world** format, such that we can run

```
1 pose = camtoworlds[i]
2 x_world = pose[:3, :3] @ x_camera + pose[:3, 3:4]
```

to convert a 3D camera space point `x_camera` into a world space point `x_world`.

These matrices must be stored in the **OpenGL** coordinate system convention for camera rotation: x-axis to the right, y-axis upward, and z-axis backward along the camera's focal axis.

---

The most common conventions are

- `[right, up, backwards]`: OpenGL, NeRF, most graphics code.
- `[right, down, forwards]`: OpenCV, COLMAP, most computer vision code.

Fortunately switching from OpenCV/COLMAP to NeRF is simple: you just need to right-multiply the OpenCV pose matrices by `np.diag([1, -1, -1, 1])`, which will flip the sign of the y-axis (from down to up) and z-axis (from forwards to backwards):

```
1 camtoworlds_opengl = camtoworlds_opencv @ np.diag([1, -1, -1, 1])
```

You may also want to **scale** your camera pose translations such that they all lie within the  $[-1, 1]^3$  cube for best performance with the default mipnerf360 config files.

We provide a useful helper function `camera_utils.transform_poses_pca` that computes a translation/rotation/scaling transform for the input poses that aligns the world space x-y plane with the ground (based on PCA) and scales the scene so that all input pose positions lie within  $[-1, 1]^3$ . (This function is applied by default when loading mip-NeRF 360 scenes with the LLFF data loader.) For a scene where this transformation has been applied, `camera_utils.generate_ellipse_path` can be used to generate a nice elliptical camera path for rendering videos.

### Intrinsic camera poses

`pixtocams`= [N, 3, 4] numpy array of inverse intrinsic matrices, OR [3, 4] numpy array of a single shared inverse intrinsic matrix. These should be in **OpenCV** format, e.g.

```
1 camtopix = np.array([
2     [focal,    0, width/2],
3     [    0, focal, height/2],
4     [    0,    0,    1],
5 ])
6 pixtocam = np.linalg.inv(camtopix)
```

Given a focal length and image size (and assuming a centered principal point, this matrix can be created using `camera_utils.get_pixtocam`.

Alternatively, it can be created by using `camera_utils.intrinsic_matrix` and inverting the resulting matrix.

### Resolution

`height` = int, height of images.

`width` = int, width of images.

### Distortion parameters (optional)

---

`distortion_params` = dict, camera lens distortion model parameters. This dictionary must map from strings -> floats, and the allowed keys are ['k1', 'k2', 'k3', 'k4', 'p1', 'p2'] (up to four radial coefficients and up to two tangential coefficients). By default, this is set to the empty dictionary {}, in which case undistortion is not run.

### Details of the inner workings of Dataset

The public interface mimics the behavior of a standard machine learning pipeline dataset provider that can provide infinite batches of data to the training/testing pipelines without exposing any details of how the batches are loaded/created or how this is parallelized. Therefore, the initializer runs all setup, including data loading from disk using `_load_renderings`, and begins the thread using its parent `start()` method. After the initializer returns, the caller can request batches of data straight away.

The internal `self._queue` is initialized as `queue.Queue(3)`, so the infinite loop in `run()` will block on the call `self._queue.put(self._next_fn())` once there are 3 elements. The main thread training job runs in a loop that pops 1 element at a time off the front of the queue. The Dataset thread's `run()` loop will populate the queue with 3 elements, then wait until a batch has been removed and push one more onto the end.

This repeats indefinitely until the main thread's training loop completes (typically hundreds of thousands of iterations), then the main thread will exit and the Dataset thread will automatically be killed since it is a daemon.

### Citation

If you use this software package, please cite whichever constituent paper(s) you build upon, or feel free to cite this entire codebase as:

```
1 @misc{multinerf2022,  
2     title={{MultiNeRF}: {A} {Code} {Release} for {Mip-NeRF} 360, {Ref  
3     -NeRF}, and {RawNeRF}},  
4     author={Ben Mildenhall and Dor Verbin and Pratul P. Srinivasan  
5     and Peter Hedman and Ricardo Martin-Brualla and Jonathan T.  
6     Barron},  
7     year={2022},  
8     url={https://github.com/google-research/multinerf},  
9 }
```