NeRF

Implementation detail

1. In `encoder.py`

Adding positional encoding to the input.

Create frequency band with `torch.linspace`

```
# log_sampling = True
freq_bands = 2.0 ** torch.linspace(0.0, max_freq, steps = N_freqs) # log-spaced
# log_sampling = False
freq_bands = torch.linspace(2.0**0.0, 2.0 ** max_freq, steps = N_freqs) # evenly-spaced
```

Then apply the sin/cos function to input * frequency_band

```
for freq in freq_bands:
    for p_fn in periodic_fns:
        # apply the periodic function to the input
        embed_fn = lambda x, p_fn = p_fn, freq = freq: p_fn(x * freq)
        embed_fns.append(embed_fn)
        out_dim += d
```

2. In `nerf.py`

Create Module List (including skip connection)

```
self.pts_linears = nn.ModuleList(
        [nn.Linear(input_ch, W)] +
        [nn.Linear(W, W) if i not in self.skips else nn.Linear(W + input_ch, W) for i in range(D-1)])
```

If the layer is in the skip list, simply extend the dimension by `input_ch` . Concatenation with input is handled in forward pass.

Create MLP for feature, alpha and rgb

```
if use_viewdirs:
    self.feature_linear = nn.Linear(W, W)
    self.alpha_linear = nn.Linear(W, 1)
    self.rgb_linear = nn.Linear(W//2, 3)
else:
    self.output_linear = nn.Linear(W, output_ch)
```

If `use_viewdirs` is True, the view direction information will be encoded and pass to nueral network. There will be two branches of MLP:

(a) Spatial MLP (pts_linears):

Input: Positionally encoded 3D coordinates

Output: W-dimensional features + density (alpha)

(b) View-dependent branch MLP:

concatenate spatial features with encoded view directions

3. In `render.py`

Convert Raw Density to Alpha (Opacity)

```
def raw2alpha(raw, dists, act_fn = F.relu):
    alpha = 1.0 - torch.exp(-act_fn(raw) * dists)
    return alpha
```

By volume rendering equation, $\alpha = 1 - \exp(-\sigma \cdot \delta)$.

`raw` is the raw density from NeRF model

Compute Weights for each sample

```
weights = alpha * torch.cumprod(torch.cat([torch.ones((alpha.shape[0], 1)),
            1.0-alpha + 1e-10], -1), -1)[:, :-1]
```

Compute the contribution of each sample to the final color based on its opacity and the transmittance of previous samples (`cumprod`). Use `t` `orch.cat` to pad one to the tensor before cumulative product and drop the last item of the tensor in the end.

Formula:

$$w_i = \alpha_i \cdot \prod_{j=1}^{i-1} (1 - \alpha_j)$$

Compute RGB Map

```
rgb_map = torch.sum(weights[..., None] * rgb, -2)
```

Integrate RGB value along the ray using the computed weights.

Compute Depth Map

```
depth_map = torch.sum(weights * z_vals, -1)
```

Similar to RGB map, compute the expected depth along the ray.

Computer Disparity Map

```
disp_map = 1.0 / torch.max(1e-10 * torch.ones_like(depth_map), depth_map / torch.sum(weights, -1))
```

Disparity is the inverse depth, adding `1e-10` to avoid division by zero.

Compute Accumulation Map

```
acc_map = torch.sum(weights, -1)
```

Sum of weights along the ray

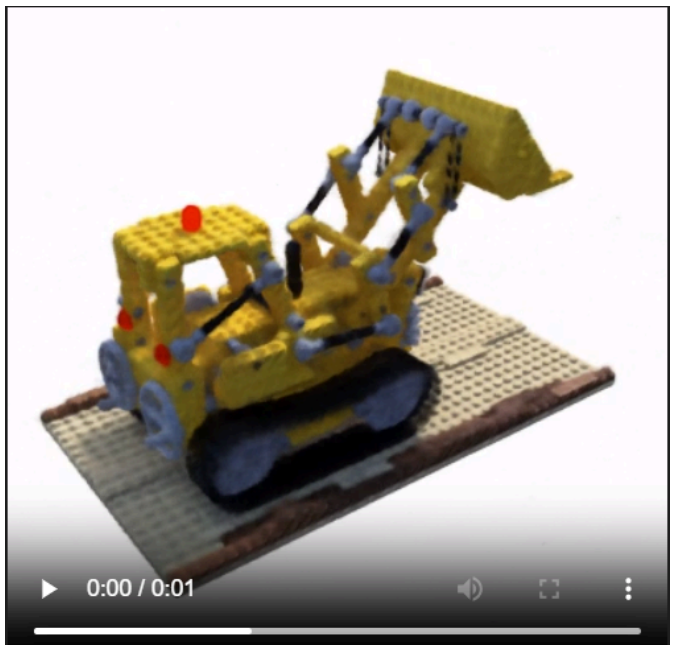
Compute 3D Points in `render_rays()`

```
pts = rays_o[..., None, :] + rays_d[..., None, :] * z_vals[..., :, None]
```

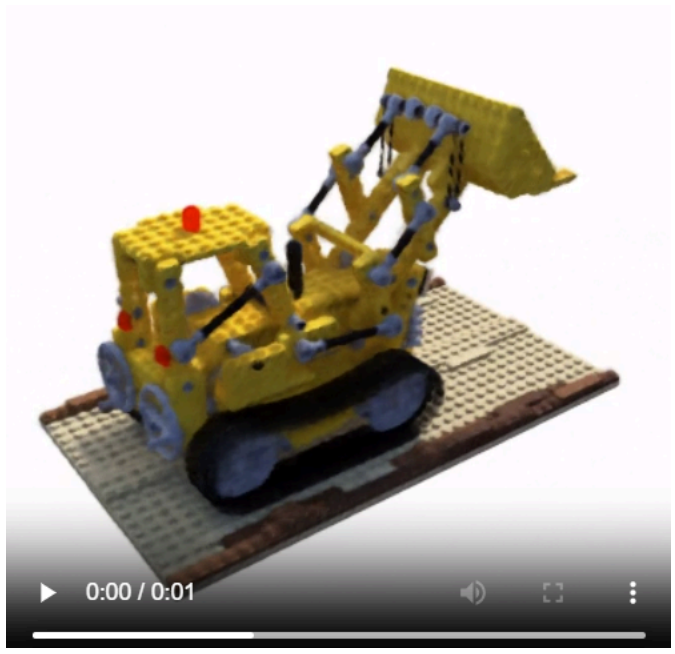
$\text{pts} = \text{rays\_o} + \text{rays\_d} \cdot \text{z\_vals}$

Rendering Effect

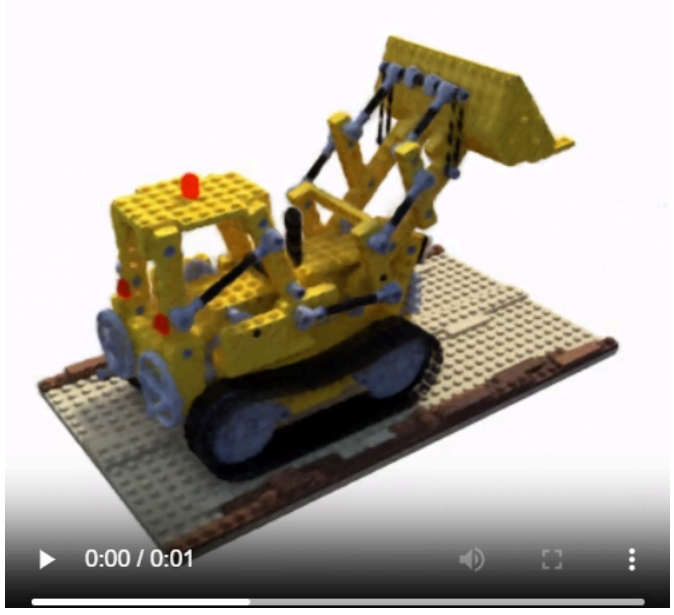
Lego



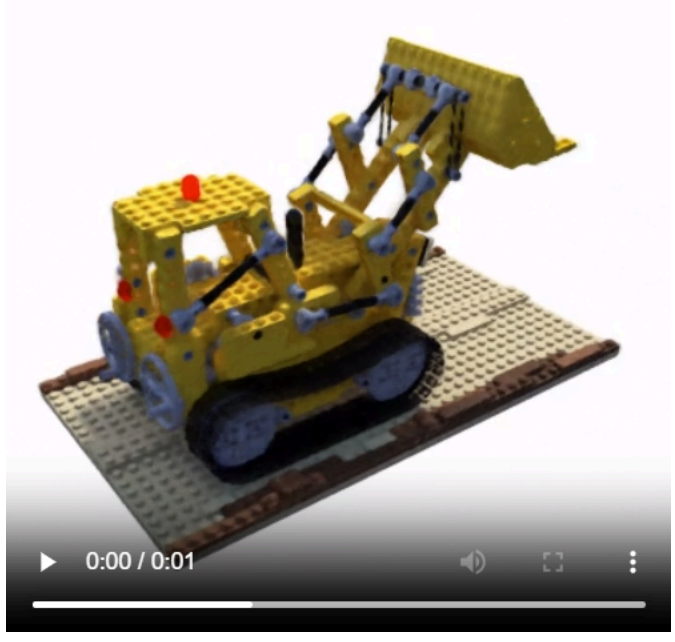
30000 Iterations



50000 Iterations



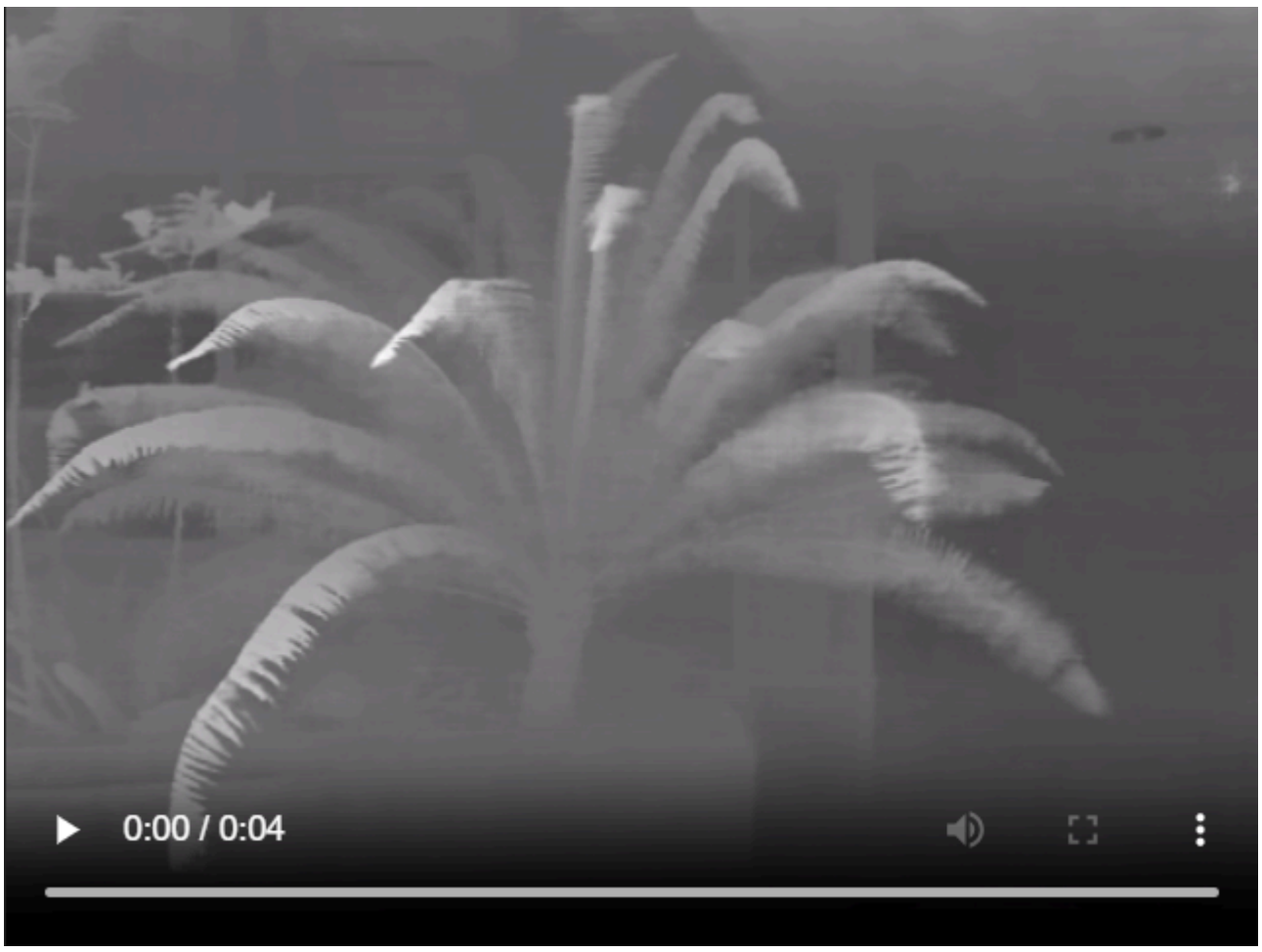
100000 Iterations



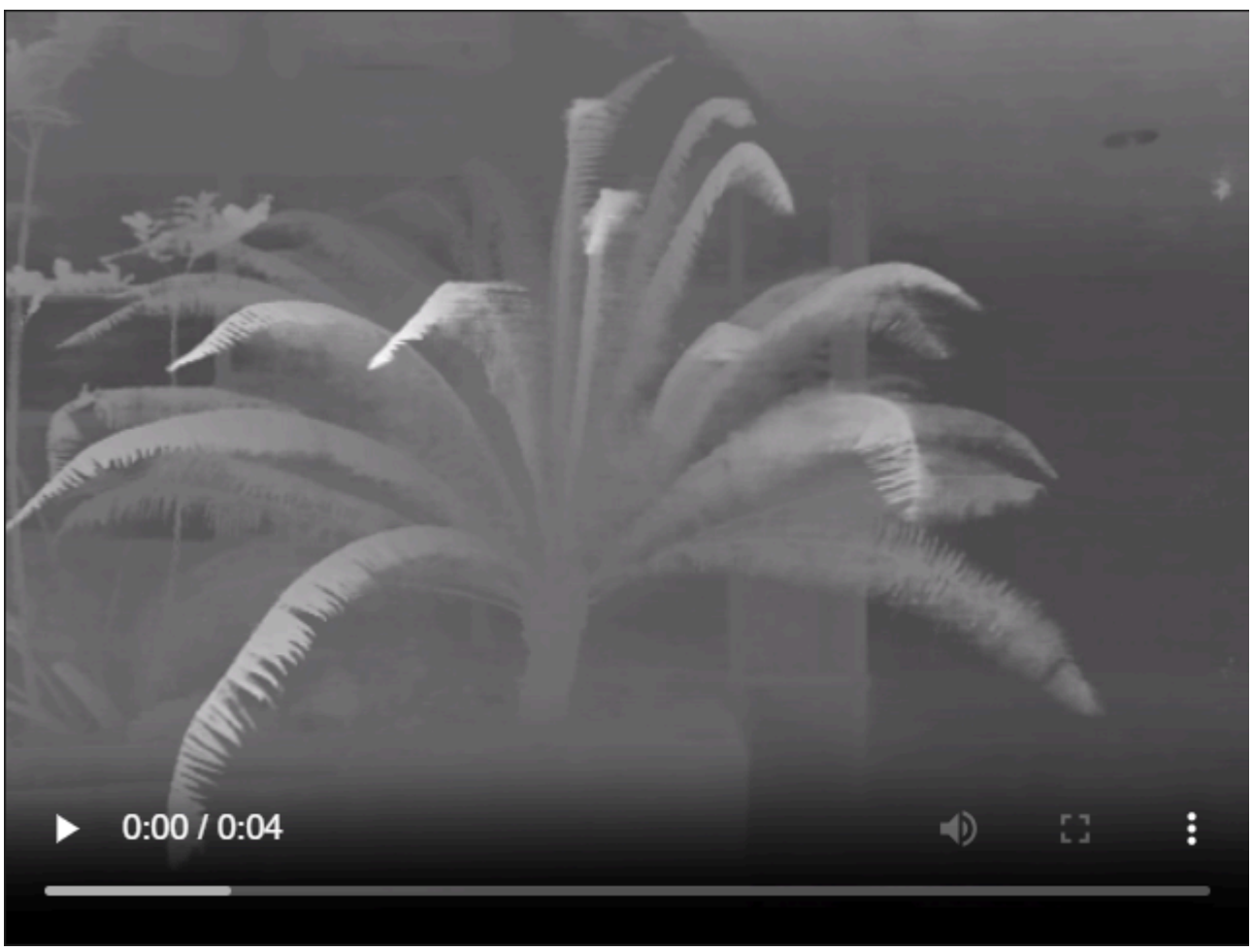
150000 Iterations

In the early stage of training, the image is blurry. From 30k to 100k iterations, we can see that the model is capturing fine details of Lego. The rendered Lego bricks are becoming sharper in edges and stubs. The model converges at around 100k iterations, after which the quality of rendering does not significantly improve.

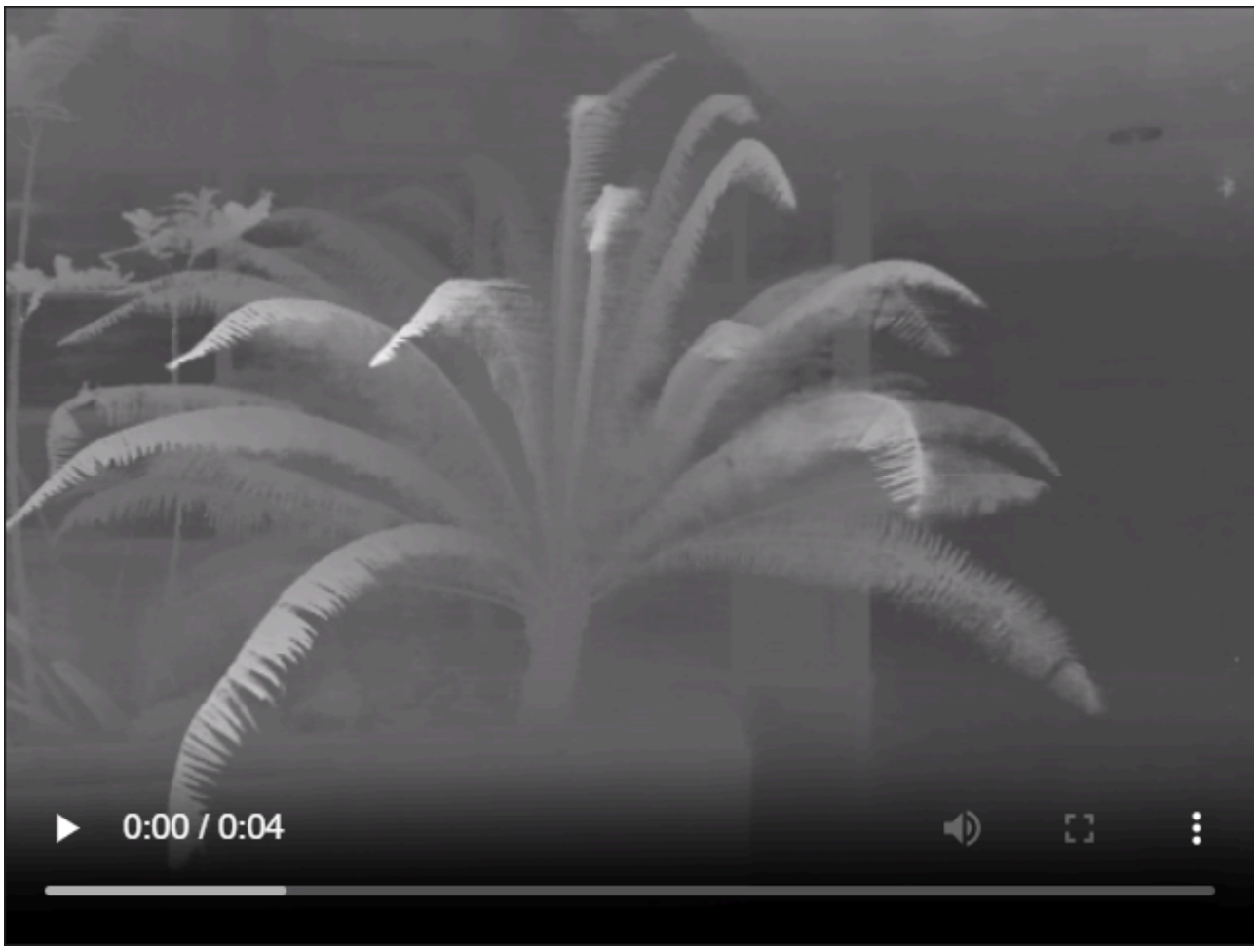
Fern



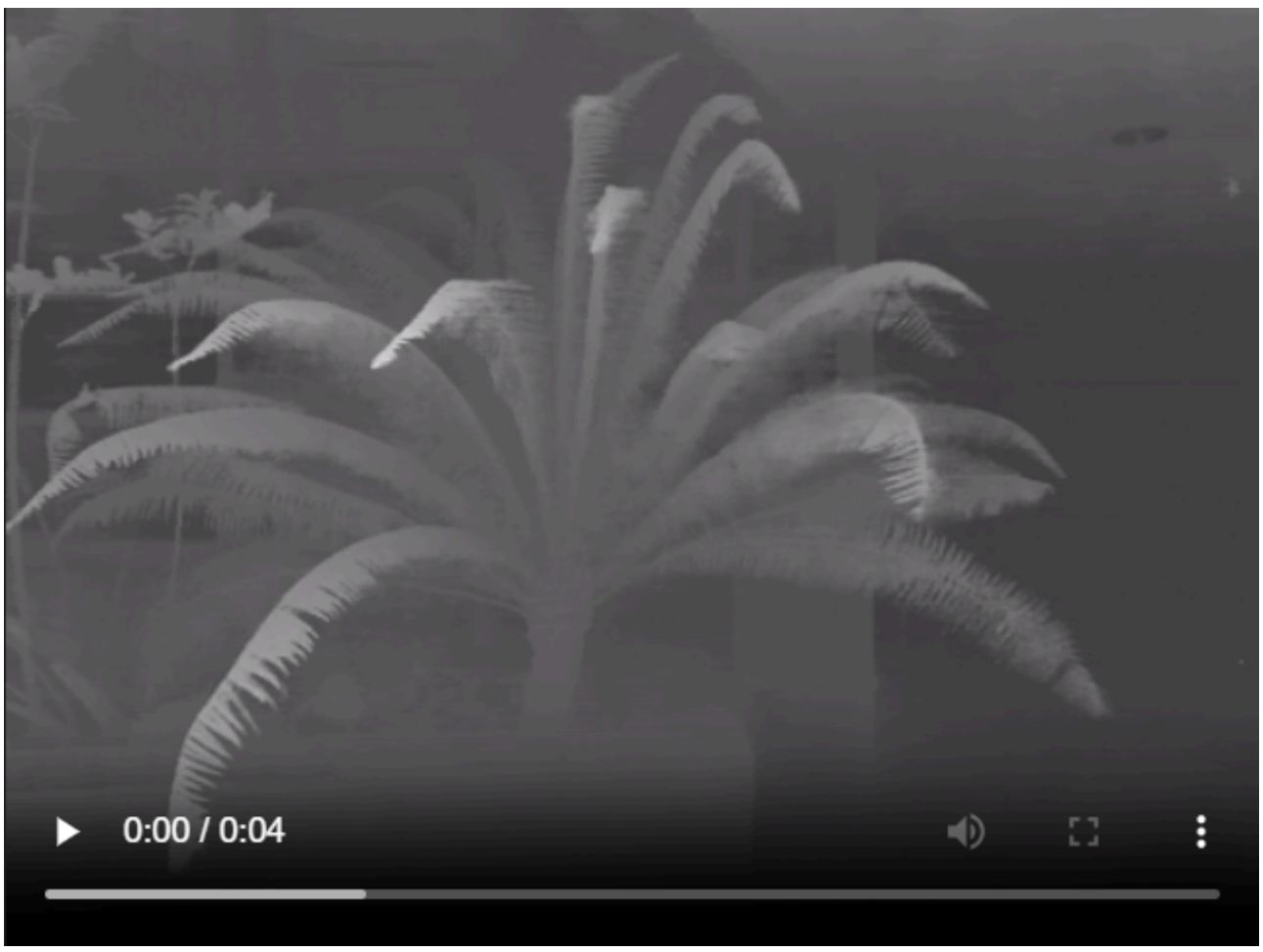
Disparity at 50k Iterations



Disparity at 100k Iterations



Disparity at 150k Iterations



Disparity at 200k Iterations

As the training continues, the model is learning more high-frequency details and building up knowledge about the depth of the scene. In the disparity map, the edges of the leaves of the fern are becoming more distinguishable from the background.

Rendered videos in `.mp4` can be found at:
https://drive.google.com/drive/folders/15CQAIGxMvpMBgk_xIKY0I0erR3LscZ6N?usp=sharing

Bonus point: synthesize novel animations with customized camera path

Changing the `render_pose` matrices in `./data_loader/load_blender.py` enables customized camera path, for example:

```
translate_val = np.linspace(-4, 4, 40+1)
render_poses = torch.stack([pose_spherical(0, 0, translate) for translate in translate_val[:-1]],0)
```

Creates a path that goes through the model, which makes it possible to see the rendered volume inside Lego.