Using K-Means clustering to export a NeRF for faster rendering in CG applications while preserving view-dependent appearance

Jurre Jilles Karim Groenendijk[1]

Supervisors: Elmar Eisemann[1]**, Michael Weinmann, Petr Kellnhofer**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Jurre Jilles Karim Groenendijk
Final project course: CSE3000 Research Project
Thesis committee: Elmar Eisemann, Petr Kellnhofer, Michael Weinmann, Jan van Gemert

Abstract

With the current state-of-the-art research, exporting a NeRF to a mesh has the side effect of having to evaluate a Multi Layer Perceptron at render-time, causing a significant decrease in performance. We have found a way to use K-Means clustering to pre-compute values for this MLP, storing them in multiple octahedron maps for the GPU to fetch when it's time to render the object. This improves render times by a factor of 3-4x.

1 Introduction

Neural Radiance Fields (also called NeRF) [5] are a fascinating way to bridge the gap between the real and virtual world, allowing anyone to take video media of the real world and use it to train a Multi Layer Perceptron to synthesize novel views of the scene with remarkable realism and detail. The barrier to entry of this technology is as low as using images taken from a phone using specialized applications, and can be as high as specialized cameras that capture RGBD information and store high-precision offsets. The next step is to enable its use in Computer Graphics applications, which could massively impact fields such as industrial design, architecture and video game creation.

A lot of research at this point still focuses on implementing different types of NeRFs or improving processing time[7][4], output quality[1], number of inputs[6], etc. However, there are a few research papers on exporting NeRFs to meshes. Within the current bounds of research, it is possible to train and export a NeRF while preserving view-dependant qualities using a method called nerf2mesh[8], however, this still depends on evaluating a Multi Layer Perceptron for view-dependent effects (specifically specularities). Evaluating an MLP is computationally expensive, and especially when multiple NeRF-exported objects are placed within one scene, these costs can add up quickly.

This research aims at removing the MLP from the equation entirely, pre-computing its output colors, and storing those colors in additional textures used for the final rendering process. This is achieved by using K-Means clustering to group colors together that would have similar specularities, and then creating octahedron spectral maps for every cluster. This significantly increases performance, while resulting in an acceptable compromise on accuracy compared to the original model. Using these techniques, a 3x rendering time decrease was achieved.

2 Method

2.1 Nerf2mesh's method

Nerf2mesh's current approach generates a few files to be used in the final rendering process.

- An object file, containing the geometry of the final scene.
- A diffuse texture, for fetching the diffuse color.
- A specular texture, for fetching one of the input values of the multilayer perceptron (MLP).
- A JSON file containing the weights of the MLP.

Then, during the rendering process, for every ray in the shader, the diffuse color is fetched from the diffuse texture (c_d). Then, the specular input color is fetched from the specular texture (f_s). This is then combined with the view direction (d) as input to the MLP. The output of this MLP is the final specular color (c_s).

$$c_s = MLP(f_s, d)$$

Then, these two colors are summed together to get the final color.

$$c = c_d + c_s$$

2.2 Our method

For our method, we pre-compute the output colors of this MLP. For a specular input color, we generate an octahedron map [2] that stores the result of the MLP by placing the output colors for all the possible view directions into a texture.

However, doing this for all the specular input colors would create an octahedron atlas of many terabytes, and therefore we have to be conservative in the number of octahedron maps we create.

To that end, we perform K-means clustering [3] on the input specular colors (f_s) of the MLP. However, since two specular colors being similar does not directly mean that the output (c_s) will be similar. As such, we have to use something else as our criteria to cluster on.

For our approach, we create N evenly spaced points on a sphere and calculate the output specular color for all those points. These colors act as an identifier, an approximation if you will, for the domain of the MLP for particular input color.

$$ident_{f_s} = \{MLP(f_s, d) | d \in D\}$$

Where D is a number of evenly spaced points on a sphere.
This is the data that we cluster on, with the general idea being that if those identifying vectors are close together, the final specular domain of the MLP for those input colors will be as well.

Since clustering and generating identifying vectors for the same input colors is unnecessary, we first group the unique input colors together while taking note of the number of occurrences, so that we only have to generate one identifying vector per color used in the specular input map. Then, we perform our clustering algorithm on those vectors, taking the number of occurrences into account.

However, since we have clustered on the identifying vectors, our cluster centers will also be in the format of those vectors, we need to get back to an input color. To do this, for every cluster, we find the input color that creates the

identifying vector that is closest to the cluster center, and use that as the final input color for generating the maps for that cluster. As an additional step, we then de-duplicate the list of input colors to only get the unique inputs for the clustering, potentially saving space on the final octahedron atlas.

Then, for each of the de-duplicated clustered input color, we generate an octahedron map [2] that stores the result of the MLP for the specular input value that we assigned to the cluster, storing the output values for all the possible view directions into a square map. These maps then tile together to form the octahedron atlas, which is saved as one of the files to be used for the final rendering process.

Afterwards, we take all the input values, and assign them to their closest cluster center based on their identifier. We export this label for every value in the input specular texture, which will create a texture referencing which octahedron map to look at for the rendering of that ray.

During rendering, the process to get the final specular color given a view direction and UV in the texture consists of two steps:

1. Fetch the label from the label texture. This references which octahedron map to use.
2. Use the view direction to calculate the corresponding location in that octahedron map.

The color at the location of that octahedron map is the final specular color c_s we want to use.
Getting the final specular value

```
int cluster_sqrt = sqrt(clusters);
int label = texture(tLabelMap, uv);
int labelX = label % clusters_sqrt;
int labelY = label / clusters_sqrt;

vec2 octaUV = octahedron_mapping(view_direction);

int map_size = textureSize / clusters_sqrt;

vec2 atlas_UV = vec2(labelX + octaUV.x,
                     labelY + octaUV.y);
atlas_UV = atlas_UV * map_size / texture_size;

vec3 finalTexel = texture(tOctahedronAtlas,
                          atlas_UV);

return finalTexel;
```

Where octahedron_mapping is a function that returns the UV coordinate in an octahedron map for a given view direction, clusters is the number of clusters in the final atlas, and texture_size is the final size of the octahedron atlas.

3 Experimental Setup and Results

3.1 Frametime measurements

We generated the input texture using Nerf2Mesh's original repository [8], while setting super-sampling to 1 and using a 2048x2048 texture size. This was to circumvent the need for OpenGL, as the testing environment only supported CUDA. We believe this has no impact on our testing results, as textures are sampled sparingly, and texture sampling does not significantly slow down with a higher texture size.

For our experiments, there were two setups: One setup with the vertex shader Nerf2Mesh provided, and one setup with our new vertex shader using the pre-processed textures. Furthermore, we have tested these setups in two environments: one WebGL environment using the Three.JS library, and one environment as a native shader using Unity.

For the Unity environment, we encoded the weights of the MLP into textures in the format that the WebGL reader expects it. Then, we ported the GLSL code to HSLS code so that it works in Unity shaders. We also did this for our pre-computed method and then created a camera rendering a screen of size 8192 x 5461, which is the largest unity would allow. This is to get the highest measurable effect to compare the two methods.

For the WebGL environment, we edited our Firefox settings to allow for unlimited FPS, rather than VSync's cap of 60, and used the `renderer.html` file as Nerf2Mesh's method, and our `renderer_packed.html` file as our method.

Then, in both environments, we positioned the camera such that the entire screen is taken up by the object, so all pixels are tasked with the computation. We recorded our results in Table 1.

method	Unity FPS	WebGL FPS
nerf2mesh	47	280
ours	200	800

Table 1: Results. Higher FPS is better

Within the specs used for this experiment, as seen in appendix Table 3, this means a 3-4x FPS increase.

3.2 Pre-processing optimization

Our solution supports a few parameters:

- the output resolution of each octahedron map
- the number of clusters
- the number of points that create the identifying vectors

Increasing the output resolution means the step of generating the final octahedron maps takes longer, but there are more accurate results when sampling. Furthermore, the output size of the final octahedron atlas will be bigger, more RAM will be consumed when generating the map, and more VRAM will be used when rendering the object.
Increasing the number of clusters should reduce the mean absolute error and mean squared error of every pixel, but will

also mean clustering takes longer and will make the final octahedron atlas bigger as well.
Increasing the number of points used to create the identifying clusters should mean that the clusters are more representative of the output space they are trying to map to, and therefore, the final octahedron maps should be closer to the points we are looking for when clustering.

To see the effects of these changes, we tested the general visual resemblance. To do this, we take the mean absolute error and mean squared error for the clustering directions for every pixel in the source texture. To clarify, for a single color:

$$MAE_{pixel}(p, cluster) = \frac{1}{|D|} \sum_{d \in D} |p - sample(cluster, d)|$$

$$MSE_{pixel}(p, cluster) = \frac{1}{|D|} \sum_{d \in D} (p - sample(cluster, d))^2$$

And for the entire texture:

$$MAE_{texture} = \frac{1}{|P|} \sum_{p \in P} MAE_{pixel}(p, get\_cluster(p))$$

$$MSE_{texture} = \frac{1}{|P|} \sum_{p \in P} MSE_{pixel}(p, get\_cluster(p))$$

where D is the list of directions we sample, P is the list of pixels in the original input specular texture, $get\_cluster$ is a function that returns the corresponding cluster to a pixel, and $sample$ is a function that samples a cluster in a certain direction. Doing this for our original 2048x2048 input texture for varying numbers of clusters, with the individual octahedron maps being 64x64 pixels and 100 points being used as identifiers, gives the results in Table 2

Clusters	MAE	MSE	Time to generate(s)
1	12.933	609.745	839
4	11.868	476.378	854
16	11.151	409.153	1176
64	10.789	379.223	1928
256	10.637	367.115	4723
1024	10.587	362.210	15289
2048	10.578	361.085	34509

Table 2: Results. Lower MAE and MSE is better

For our input texture, after even 64 clusters the extra processing time and storage space a higher-cluster setup would require seem to outweigh the benefit of higher accuracy for most use-cases.

It is worth noting that the points used for evaluation are evenly spaced, and therefore a higher texture resolution will not necessarily generate better scores for the metrics used here. A higher texture resolution does however greatly improve visual clarity of the final result, but this effect is a lot harder to measure in a reasonable timeframe.

4 Responsible Research

Under the M.I.T. License agreement provided within the Nerf2Mesh repository, we can see that we are free to: "deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so.", given that we include the license agreement in our version of the project as well. This has been done, as well as us using the same license ourselves.

Our code is shared on the github repository[1], together with instructions on how to download the dataset used and reproduce our results. The original used dataset, as well as all the commands used to generate the results and the results itself are located on this github as well.

5 Discussion

Section 3.1 focuses mainly on the worst-case scenario: the largest screen size possible, looking at only pixels that require our vertex shader to be run. We imagine in actual uses of this technology, it will be mixed with other shaders in the scene, and this will only be a small portion of what is being viewed. However, we also believe that wherever possible, it is worth making optimisations to reduce frame time, so that other effects have sufficient GPU capacity left.

6 Conclusions and Future Work

With the current state-of-the-art research, exporting a NeRF to a mesh has the side effect of having to evaluate a Multi Layer Perceptron at render-time, causing a significant decrease in performance. We have found a way to use K-Means clustering to pre-compute values for this MLP, storing them in multiple octahedron maps for the GPU to fetch when it's time to render the object. This improves render times by a factor of 3-4x.
For future work, one could look into using algorithms other than K-Means clustering to see if this can create better clusters or speed up clustering. Moreover, one could look at different ways to create the identifying vectors, rather than taking evenly spaced points on a sphere. Furthermore, one could look into estimating material properties from these maps using a material model, enabling more freedom when modifying the mesh afterwards.

[1] https://github.com/jurrejelle/nerf2mesh

A Specifications used for experiments

Slot	Component
CPU	AMD Ryzen 7 7700X
GPU	NVIDIA GeForce RTX 3070
RAM	4x16GBKingston FURY DDR5-5200 (Underclocked to 3600 MHz because of CPU limitations)
Motherboard	Gigabyte B650M DS3H

Table 3: Specifications used for experiments

References

[1] Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. *ICCV*, 2021.

[2] Thomas Engelhardt and Carsten Dachsbacher. Octahedron environment maps. 2008.

[3] Aristidis Likas, Nikos Vlassis, and Jacob J. Verbeek. The global k-means clustering algorithm. 2000.

[4] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields, 2021.

[5] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[6] Michael Niemeyer, Jonathan T. Barron, Ben Mildenhall, Mehdi S. M. Sajjadi, Andreas Geiger, and Noha Radwan. Regnerf: Regularizing neural radiance fields for view synthesis from sparse inputs. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[7] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In *International Conference on Computer Vision (ICCV)*, 2021.

[8] Jiaxiang Tang, Hang Zhou, Xiaokang Chen, Tianshu Hu, Errui Ding, Jingdong Wang, and Gang Zeng. Delicate textured mesh recovery from nerf via adaptive surface refinement. *arXiv preprint arXiv:2303.02091*, 2022.