6.4400 Final Project: Automatic Mesh LODs

Kailas B. Kahler

December 13, 2023

1 Introduction

In typical video game rendering pipelines, there is a desire for both high visual fidelity and real-time, responsive rendering. Triangle meshes, both those created by artists in modeling software and those created though other methods like photogrammetry, can deliver a high standard of visual quality by including large numbers of small triangles to represent fine, intricate details. However, such details are not always visible on the screen after rendering, and processing and storing a large number of triangles is costly from a compute and memory perspective. There is thus a motivation to have a minimally yet adequately detailed mesh loaded into the scene, and switch between versions of the mesh depending on the camera's view of it.

It has been common practice in video games to manually generate these levels-of-detail (LODs), and then manually define the points where to load a given LOD. However, this is a time sink for developers, and it often results in effects like "pop-in", where a change in mesh LOD is too abrupt and makes it appear like the object had sprung out of thin air. This project seeks to automate this process by generating the different LODs of the mesh and switching out the active LOD automatically and smoothly.

2 Background and Previous Work

This same problem has been tackled recently by Epic Games for its Nanite virtual geometry system for Unreal Engine 5. A member of Epic's team, Brian Karis, delivered a 2021 SIGGRAPH presentation on the inner workings of Nanite, and this serves as much of the inspiration from the project. The Nanite system uses edge collapsing to simplify clusters of 128 triangles, and iteratively merges clusters until there is a single cluster remaining. Edge collapsing can be performed with various algorithms, but the most popular variety uses the quadric error metric proposed by Garland and Heckbert. During the Nanite rendering process, the renderer traverses the cluster level-of-detail graph and chooses the simplest cluster which provides error that is less than a single pixel, and thus will not be visible after rendering.

Some aspects of the Nanite system are not necessary for purely LOD automation, including

the GPU-driven pipeline, and the triangle clustering and directed acyclic graph. Instead, in this project the mesh is simplified as a whole, and thus level-of-detail is selected for the entire mesh. This simplifies the design to make it achievable in the time frame of the project.

3 Implementation

The project was implemented using the GLOO framework as it existed after completion of lab 5. The implementation of the project can be divided between its two main components: (1) the mesh simplifier, and (2) the LOD selector.

The mesh simplifier implements Garland and Heckbert's mesh simplification algorithm with quadric error metrics. The mesh is imported into a MeshSimplifier class and converted from the position and index buffer format to a graph structure consisting of Triangle, Edge, and Vertex structs. Each Vertex has knowledge of all Triangles and Edges which include it, and all edges are bi-directional. Each Vertex has a quadric error metric that is defined as the sum of the distance squared between its position and each of planes it laid on in the original mesh. The plane equations are calculated for each Triangle that includes the Vertex and is stored in a symmetrical 4x4 matrix K. Given linearity, the sum of the K matrices for each plane can be stored in a single matrix Q, which reduces memory overhead.

When two Vertices are merged together, their Q matrices are summed, which is the equivalent of combining the sets of planes the matrices represent, with potential for double or triple counting. The new position p is calculated to be the point which minimizes the quadric error, defined as $p^T Q p$, for error quadric matrix Q and $p = [x\ y\ z\ 1]^T$. In the case that is minimization is not possible, the midpoint between the positions, $(p_0 + p_1)/2$, is chosen instead.

Possible merges between Vertices are stored in a set of Contraction structs, which are sorted by the quadric error of the resulting vertex, also called the contraction "cost". Each iteration, the lowest cost Contraction is selected, the Vertices are merged, and the related Triangle, Edge, and Contraction (with the cost) structs are updated with the new merged Vertex. Triangles which include both of the merged Vertices are degenerate, now only having two edges, and are thus removed from the mesh. This process repeats until the mesh has been reduced to the desired number of triangles.

The LOD selector takes the average quadric error of all of the Vertices in a given level-of-detail, and adds it as a displacement from the position of the mesh in the world coordinates. Each frame, given the view and projection matrices of the active camera in the scene, the simplest LOD level with an acceptable projected error is used for rendering. This threshold was hand-tuned to find a balance between too low of a threshold, where only the base LOD would be acceptable and result in lower performance, and too high of a threshold, where the approx. 100 triangle LODs would be used even when the camera is close, causing poor visual quality.

4 Results

Mesh simplification speed was measured for two sample models, the Stanford Bunny in Figure 1 and the Stanford Dragon in Figure 2. The time listed for LOD 0, which has the same amount of triangles as the original mesh, is the time to load the mesh into the MeshSimplifier data structures and export it back out.

LOD Level	Triangles	Time, incremental (ms)	Time, total (ms)
0	69451	293	293
1	34724	349	642
2	17361	189	831
3	8679	96	927
4	4339	40	967
5	2168	20	987
6	1084	10	997
7	541	6	1003
8	269	4	1007
9	134	3	1010
10	66	2	1012

Figure 1: Stanford Bunny mesh simplification times

LOD Level	Triangles	Time, incremental (s)	Time, total (s)
0	871414	4.324	4.324
1	435707	8.032	12.356
2	217853	4.151	16.507
3	108924	2.223	18.730
4	54461	1.124	19.854
5	27229	0.519	20.373
6	13613	0.267	20.640
7	6806	0.128	20.768
8	3403	0.066	20.834
9	1701	0.036	20.870
10	850	0.012	20.892
11	425	0.007	20.899
12	212	0.004	20.903
13	106	0.003	20.906
14	50	0.003	20.909

Figure 2: Stanford Dragon mesh simplification times

The performance listed in the figures used the default settings which has two quality issues. First, in the simplification process, it is possible for a "new" triangle to be created that is identical to a previous triangle. Depending on the renderer used, this maybe not be an issue, but for the GLOO renderer, which computes color by blending over multiple passes, this

causes abnormally bright spots in the mesh where lighting is double counted. This can be solved by adding a set during export of the mesh to the vertex and index buffers that checks that an equivalent triangle has not already been exported. Second, it is also possible for edge contractions to flip the normal vector of a triangle, as the cost metric only takes position into account. As Garland and Heckbert acknowledge, this can be alleviated by adding a high cost penalty to any edge contraction that would cause a triangle's normal vector to invert compared to its original normal vector, which can be measured by checking the sign of the dot product between the two. Adding these checks come with about a 30% performance penalty, but does result is significantly better simplified meshes in some cases, as seen in Figure 3.

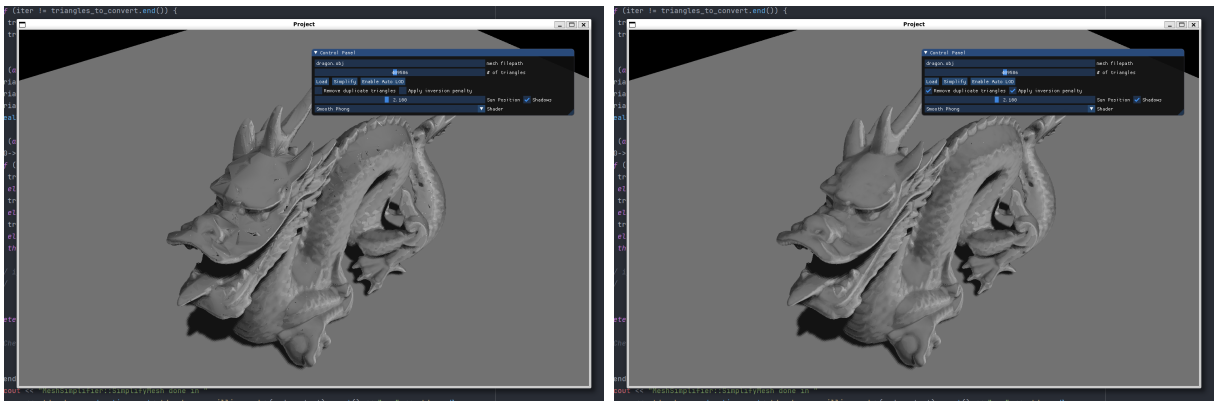

Figure 3: Left: w/o checks and penalties. Right: w/ checks and penalties

The final simplifications were of, at least to the naked eye, reasonable quality for an assortment of triangle counts in Figure 4.

The LOD selector functioned, but hand tuning is difficult, and the transitions can be noticed when the Phong shader is used. A sample video of the transitions occurring automatically can be seen here. The random triangle color shader, as desired, illustrates LOD changes clearly, while in the Phong shader the changes in LOD are less noticeable. Ideally, the changes would be imperceptible in the Phong shader, but this was not achievable in time.

5 Conclusion

The automatic mesh level-of-detail system was able to handle meshes with original triangle counts in the multiple hundred thousands fairly efficiently. However, there was a limit to the size of mesh that it could handle. The Stanford Lucy model, which has over 28 million triangles, could be loaded into the simplifier, but simplification was prohibitively slow. This seemed to be attributable to two issues. First, the memory usage of the process rose to over 10GB of RAM used. Second, as the size of the contraction set, which is sorted by cost, grows, not only does memory usage increase, but so does insertion, removal and search time. This is likely where usage of clusters becomes mandatory, as the simplifier can work only smaller subsets of the mesh at a time, so less of the model can be present in memory at once and the contraction set is much smaller.

6 References

- Brian Karis. Nanite: A Deep Dive. https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf.
- Michael Garland and Paul S. Heckbert. Surface Simplification Using Quadric Error Metrics. https://www.cs.cmu.edu/~./garland/Papers/quadrics.pdf.
- Stanford Computer Graphics Laboratory. http://graphics.stanford.edu/data/3Dscanrep/.

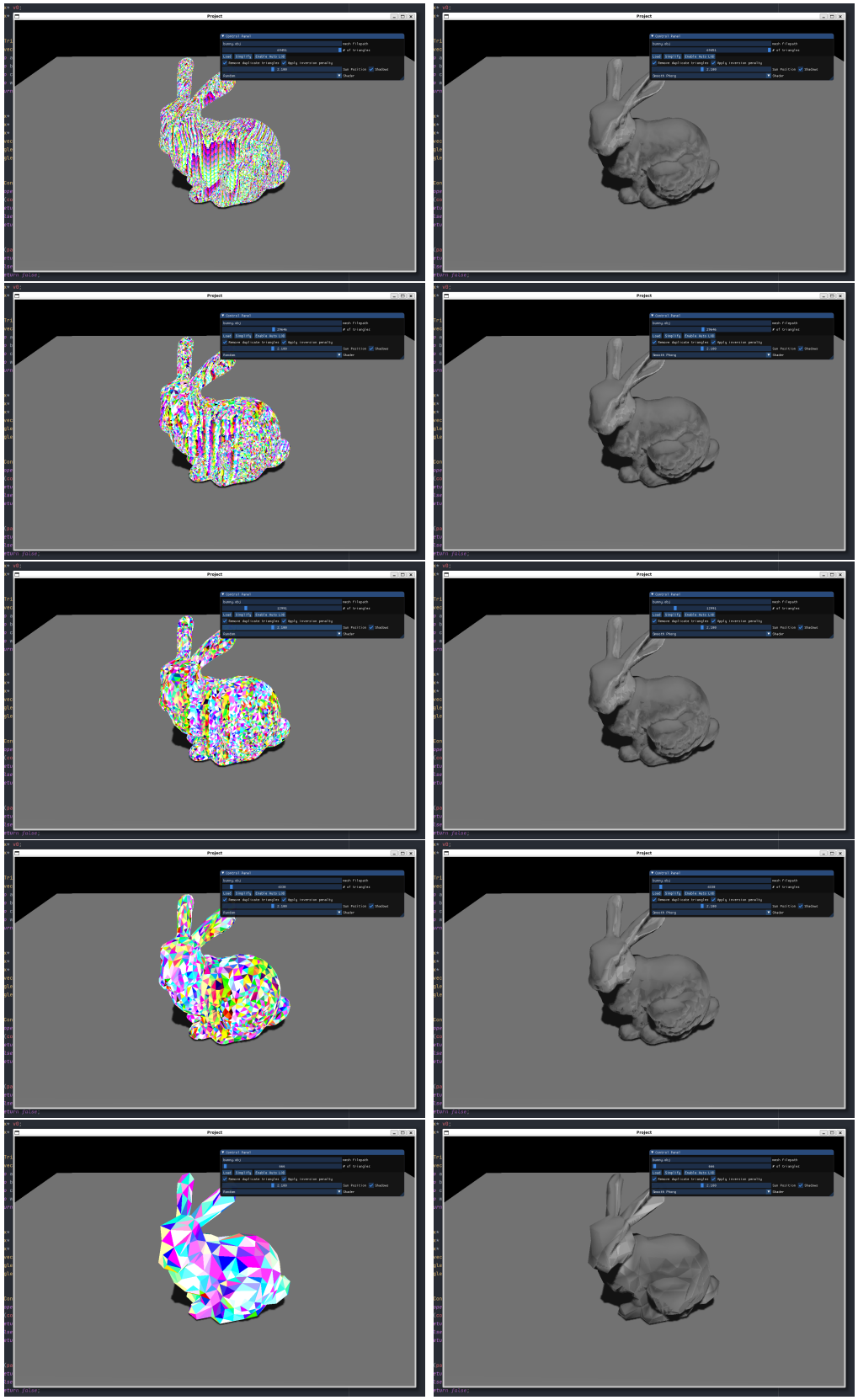

Figure 4: Left: rendered with random triangle colors. Right: rendered with smooth Phong shader.