Lethe
Patched Level-of-Detail Renderer

CSE 260

Devon McKee

March 13th, 2023

Contents

0.1 Abstract

Rendering meshes with high vertex count is an optimization problem with many solutions. In the case of meshes that are rendered far from the camera, it is a trivial process to swap the high resolution model with a pregenerated series of lower resolution versions in a technique known as Level of Detail (LOD). This project seeks to examine a technique for utilizing LOD rendering techniques on models that require closer distances to the camera by rendering patches closer to the camera in higher resolutions.

0.2 Background

0.2.1 Mesh Segmentation and LOD

Underlying this project are two important techniques in computer graphics — mesh segmentation and level of detail (LOD).

Mesh segmentation describes the process by which portions of a mesh are segmented out into different patches. The term for these patches changes depending on the literature, but most often used are segments, patches, or clusters. Mesh segmentation techniques are often separated into two distinct categories — part type segmentation and patch type segmentation. Part type segmentation is most often used for separating logical parts of a mesh (i.e., the legs from a model of a horse), where patch type segmentation is used to create (often equally sized) patches from the surface of a mesh. Patch type segmentation is more appropriate for this project, and a simple method is described in the Progress section.

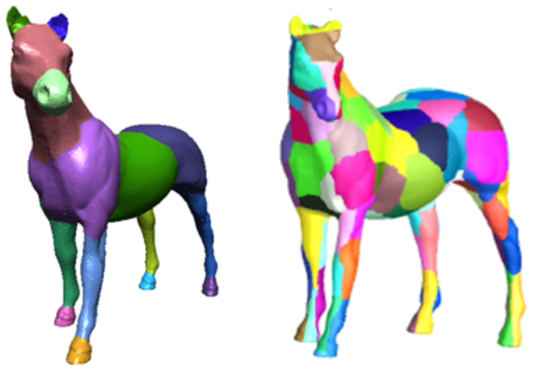

Figure 1: A visualization of part type segmentation vs. patch type segmentation. Notice that part type segmentation separates the logical components of the mesh, where patch type segmentation takes similar areas from the surface of the mesh.

Level of Detail is a technique for reducing the number of vertices rendered in a scene by swapping out higher resolution meshes for pregenerated lower resolution versions on the fly as they travel further and further from the camera. This can significantly lower the number of vertex shader invocations, and greatly optimize even dynamic scenes. A similar technique is applied to textures in a process known as mipmapping, which involves sampling from scaled down versions of textures, though this is more for sampling accuracy.

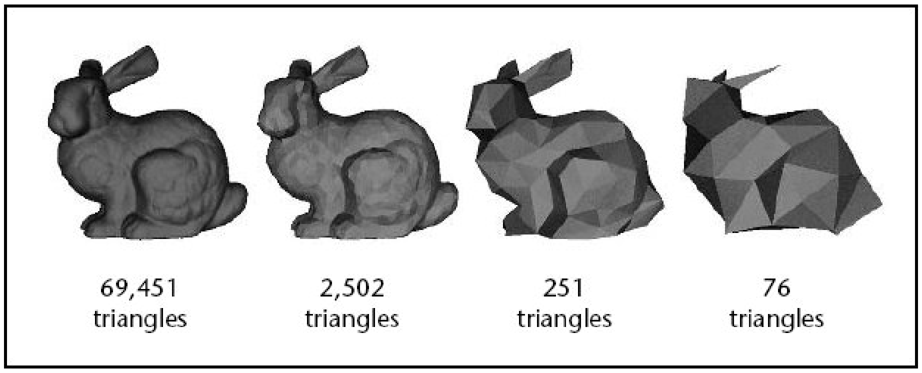

Figure 2: Level of detail being displayed on a mesh of a rabbit. The lower quality versions of the mesh can be swapped out at varying distances to continue to render the rabbit without any loss of quality, and significantly reduces the number of triangles needed to be rendered.

However, the basic concept for LOD rendering can only be applied to meshes that are likely to be far from the camera. This does not hold, for example, for a high resolution mesh of a large building where the camera is near the building. Areas of the building closer to the camera will require the higher resolution, but areas much further away will be needlessly rendered in high resolution and thus involve extra vertex shader invocations often in the scale of tens to hundreds of millions.

0.2.2 Mesh Segment Contiguity

The combination of these techniques brings up a few issues, however, and the main issue is around contiguity of patch surfaces at boundaries of regions rendered in differing levels of detail. For example, if a patch is rendered at LOD1 next to a patch rendered at LOD2, then the edges of these patches is not guaranteed to match up. A possible solution to this may be resolution of the patch boundaries at run-time, using a GPU compute pass run before the main rendering pass.

It is unclear as of yet how real applications utilizing this technique (such as Nanite, mentioned below) accomplish this, but further exploratory research may need to be done.

0.2.3 Unreal Engine: Nanite

A practical example of the combination of the previously mentioned techniques is Unreal Engine's Nanite. Nanite describes itself as a "virtualized geometry system" that can be used to "achieve pixel scale detail and high object counts."

Nanite is built into Unreal Engine 5, and can be applied to most static meshes in a scene.

Nanite utilizes both the segmentation and LOD techniques mentioned in previous sections, and does so with minimal cost at runtime. This is a commercial implementation with significant amounts of effort made to clean up the rough edges, and displays some of the higher level benefits of this model, especially for scenes with large, high resolution static props.

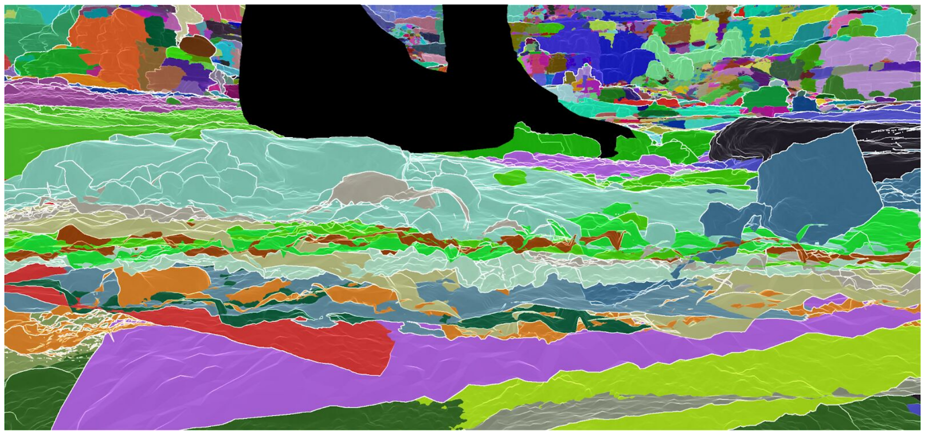

Figure 3: Nanite's segmentation for a complex scene — the black section corresponds to a dynamic mesh that cannot be rendered using Nanite, but can be combined with Nanite-rendered meshes.

Working directly with Nanite for this project was considered, but as it is very heavily built into Unreal Engine and hooking into it would involve modification of the engine source code, it was considered to be an infeasible approach.

0.2.4 WebGPU

Underlying the rendering system for this project is WebGPU, which is a new framework for modern access to GPU hardware via web interfaces. In contrast with WebGL, WebGPU follows the modern graphics/compute API layout put forward by other popular frameworks such as Vulkan/Metal/DirectX. It provides a robust graphics and compute API, and has a well constructed debug layer that provides the programmer with useful warnings when errors occur.

WebGPU itself doesn't directly interact with the GPU, as this would require a fair amount of heavy interaction with the GPU driver and state management by the browser. Instead, WebGPU translates calls to the appropriate framework on the system. On Windows machines this is DirectX, on macOS machines this is Metal, and on Linux machines this is Vulkan.

WebGPU is in active development on major browsers including Chrome, Firefox, and Safari, but the most up to date implementation is provided in

Chrome. It can be accessed behind a flag on Chrome Canary or Dev, or enabled via Origin Trials in stable versions. WebGPU is expected to enter the stable release of Chrome in Chrome 113, but this date may be pushed back if necessary.

However, future work with WebGPU in GPU synchronization may come with some issues. As of the time of writing, Metal fails to include some of the synchronization primitives required to implement fine-grained synchronization, specifically around acquire-release ordering. This necessarily weakens the WebGPU specification, and until Metal includes these primitives it may be impossible to properly implement certain types of synchronization on GPU.

0.3 Project Goals

The major goals for this project are as follows:

- Develop preprocessing tooling for preparing meshes used for this project to perform tasks such as:
  - Read in a mesh in Wavefront OBJ format
  - Segment it into patches with some reasonably quick method
  - Generate hierarchical levels of detail for patches
- Develop a rendering system capable of taking in these segmented meshes and rendering patches at different levels of detail depending on distance from camera
- If time permits, extend this rendering system to resolve contiguity between mesh patches using a real-time method in the compute pipeline

0.4 Progress

As of the time of writing, significant work has been put into both portions of the project that will be detailed below. However, certain components are missing that are integral to the display of the entire pipeline, so visual documentation is limited.

The preprocessing tools and renderer are able to read and write Wavefront OBJ files using some custom built parsing - this is required as the OBJ file format does not provide vertices and faces in a format capable of being rendered easily in a traditional graphics pipeline without some manipulation (due to indexing happening at the level of vertex attributes rather than vertices themselves).

To perform mesh segmentation, the preprocessing tools utilize a relatively method of turning all triangles in the mesh into patches, then repeatedly merging the smallest patch into its nearest patch until the desired number of patches is achieved. The desired number of patches is currently set to the nearest power of 10 below the number of triangles in the mesh (so for a mesh with 3300 triangles, the number of desired patches would be 1000.) This provides patches with a number of triangles for each patch around 3-10, though this logic may be changed down the line.

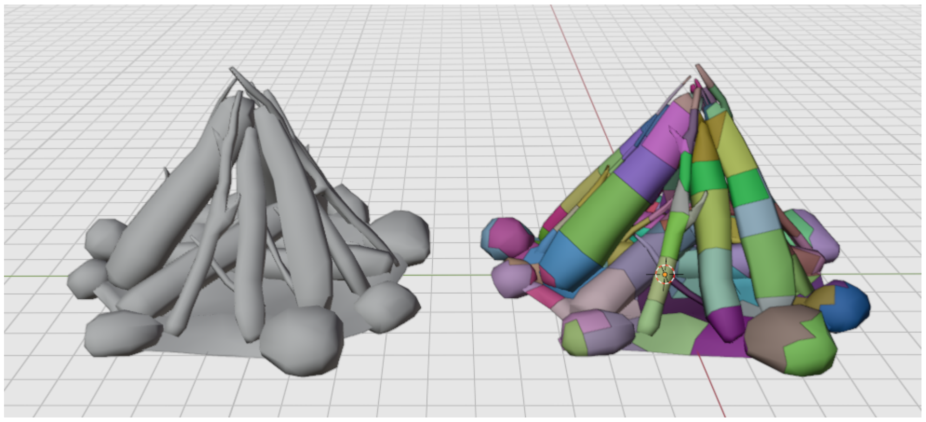

Figure 4: Segmentation of a campfire mesh used for testing, with 100 patches for 3400 triangles.

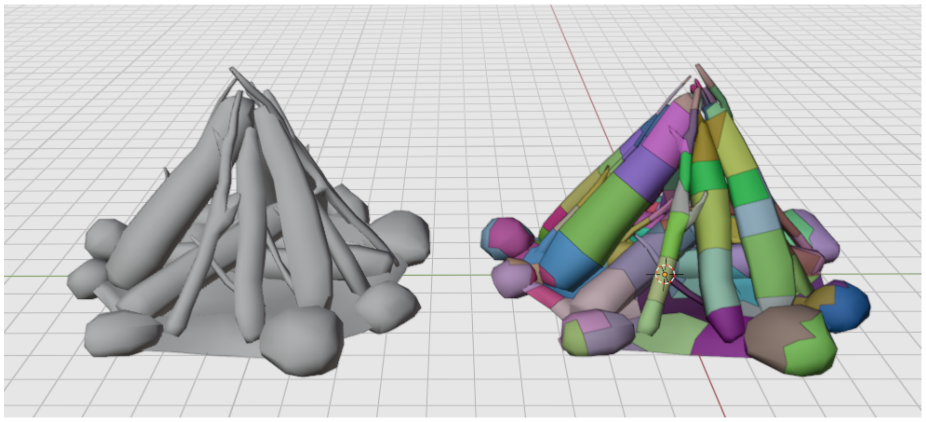

Figure 5: Segmentation of a campfire mesh used for testing, with 1000 patches for 3400 triangles.

Similar to mesh segmentation, level of detail generation is a process that can be accomplished via a number of different techniques. In search of a simple method, this project focuses on one proposed in an article published by Stan Melax, at the time an employee of Bioware and working with a group at the University of Alberta. The article describes a method of collapsing edges of a mesh in order to lower its level of detail while retaining general visual detail.

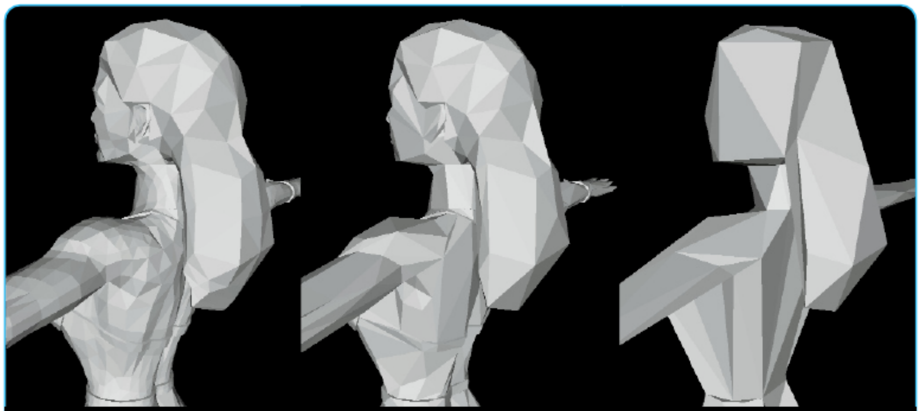

FIGURE 7. *Female human model showing 100 percent of the original polygons (left), 20 percent of the original polygons (center), and 4 percent of the original polygons (right).*

Figure 6: Figure from the Melax article, demonstrating the results of the edge collapse technique

Implementation of this method is pending, but should provide a simple way to lower detail of patches without significant processing time.

Significant work has been done on the rendering system for this project, but as much of the work involved in high-performance graphics applications happens before the first proper render can be done. As such, the renderer is capable of displaying a simple image right now, but the complete pipeline of patch LOD display has yet to be completed. This will likely be the first focus of future work.

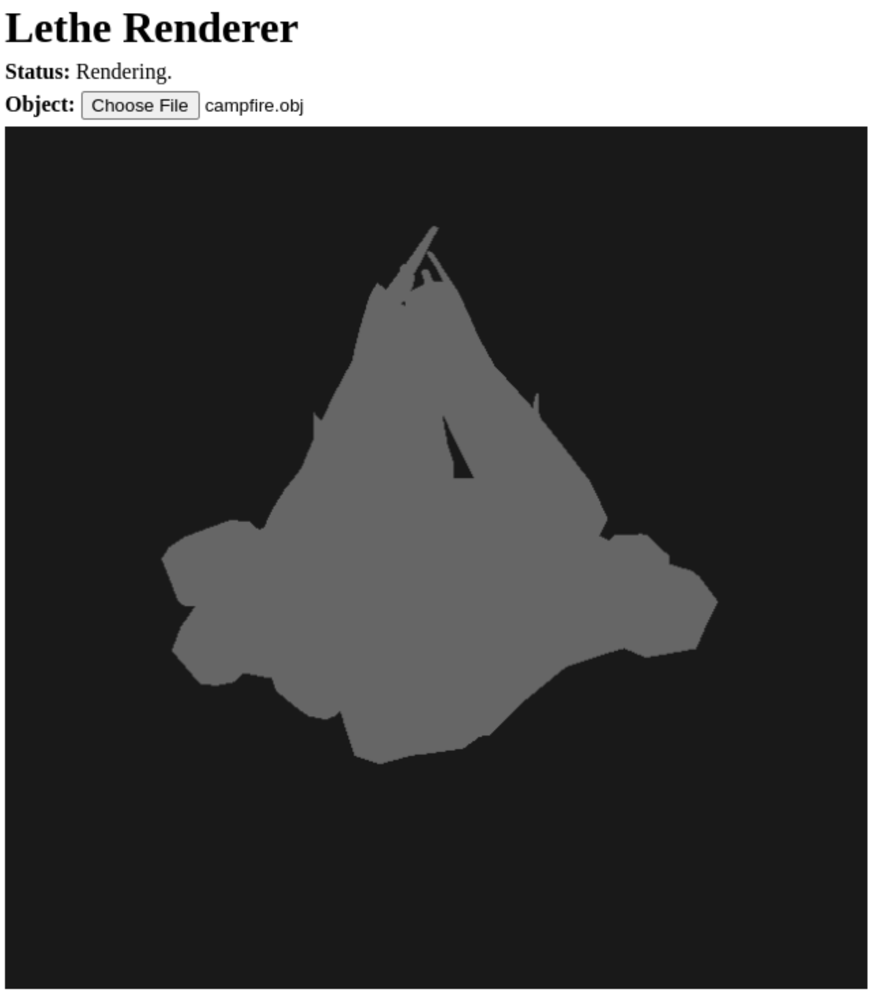

Figure 7: Initial stages of the WebGPU renderer, using the campfire test model.

0.5 Discussion & Future Work

Implementation of the goals of this project has involved a significant amount of work, and there is still more yet to be done before the model can be properly rendered with patch LOD, but many of the important components are in place and the remaining work to reach a basic rendering is likely less than a week's worth of work. After this, performance implications can be examined, and the benefit of a system of this level of complexity can be made more apparent.

Real-time patch contiguity resolution, the next major addition to this system, will likely involve more work than achieving the basic level of completion. This will involve some work to ensure a proper algorithm for doing so, and the development/debugging of a compute pass in the existing rendering system to do so.

The inspiration for this work comes from prior work done in Professor Tyler Sorensen's lab group on GPU memory models, concurrency, and synchronization. Some of this prior work has involved the examination of synchronization failures on GPUs due to store/load reorderings corresponding to a weak memory model. This behavior is heavily GPU device specific, and has been shown to cause failures on inter-workgroup locks in a recent paper submitted to ISSTA '23. Future work for this project will involve further examination of this effect on synchronization-dependent workflows, especially in graphics applications, and particularly in a segmented level-of-detail renderer such as this. Work is beginning on a GPU benchmark focused around fine-grained synchronization and dynamic work allocation, and this project will likely be included as an entry.

The code for this project can be found on GitHub at `https://github.com/boingboomtschak/lethe`.