Bachelorproject - Optimizing Rendering of Graph Based Tree Models

Jonathan W. Hale*
Computer Science, University of Konstanz

Figure 1: *Screenshots of nature rendered during the project.*

Abstract

In this paper, we discuss optimization strategies for improving render time of an existing implementation of real-time shaders for rendering polygonal trees meshes from point clouds for leaves and graph structures for the tree trunks.

An implementation is used to show the impact of the discussed strategies on performance with the aid of frame time measurements.

As part of these optimization strategies we compare how making use of the next-generation graphics API Vulkan as opposed to the proven OpenGL API affect performance for our use case.

Keywords: optimization, trees, performance, vulkan

Concepts: •**General and reference** → **Performance;**

*e-mail:jonathan.hale@uni-konstanz.de

1 Introduction

Trees and nature in general make frequent appearance in animated movies and visual effects, video games and real time aswell as static architectural visualizations. Especially in times of mobile games and virtual reality, being able to render nature efficiently yet still detailed enough to be convincing is more important than ever.

With many thousands of leaves and branches, tree models are very complex and require not only alot of memory for storage, but on top of that need to be animated according to wind and weather conditions, as trees are usually never entirely static.

Graph based representations of trees not only require less GPU memory, but allow for simpler animation and simulation. Since rendering them on their own is not sufficient at close distances, geometry needs to be generated for every edge of the graph and point representing a leaf.

As the graph may change after every simulation step, the mesh should be regenerated every frame on-the-fly, also relieving the requirement of having to store the entire mesh for all trees in a scene in memory at the same time. This has another added benefit of allowing for dynamically adapting mesh generation parameters, e.g. number of vertices per loop in the tree trunk, to be adequate for current viewing distance, providing a simple form of level of detail.

Previously researched approches to tree animations which make use of the trees branching structure, e.g. [Habel et al. 2009; Zioma 2007], can be performed on a per branch basis rather than per-vertex making use of tree graphs, slightly reducing the problems complexity.

Regenerating the mesh every frame even allows adding or removing branches up to animating the trees growth over time, which was utilized in research already [Pirk et al. 2012; Pirk et al. 2014].

2 Related Work

The games industry has been a great force behind nature in real time graphics. Recent games feature vast nature scenes and middlewares such as SpeedTree specialize solely on rendering and modelling trees.

A common method for optimizing large nature scenes is the use of billboard clouds [Candussi et al. 2005] and impostors [Truelsen and Bonding 2008]. This works well especially when rendering large amounts of trees in the distance, but when closer up, a higher level of detail is required.

[Deussen et al. 2002] presents rendering of vast amounts of plants and vegetation using pointclouds and line representations, which are rendered directly as point and line primitives for distant plants, reducing detail based on importance factors assigned by the author of the plant model.

SpeedTree uses elaborate shaders to create realistic bark displacements and light models for leaves [Kharlamov et al. 2007] similar to methods also described in [Boudon et al. 2006].

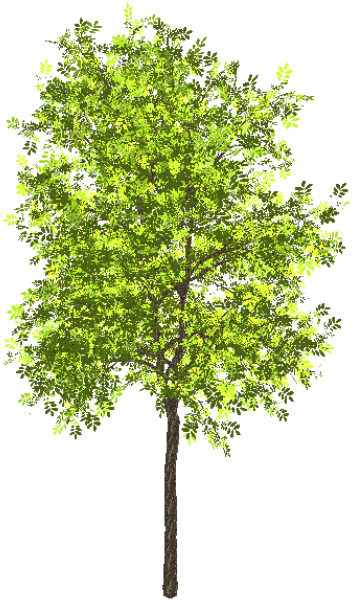

Figure 2: *A tree rendered using our pipeline.*

[Habel et al. 2009] discusses an efficient realtime animation method using 2D motion textures and hierarchical vertex displacement, improving upon the method described in [Zioma 2007]. Both of these could potentially make use of on-the-fly mesh generation from a graph based tree structure.

As opposed to these methods, we focus on optimizing a polygon based tree rendering pipeline which renders line and point based tree models, suited for real-time environments with high performance requirements while still providing flexibilty to alter, animate and simulate the tree easily. Resulting in a mesh, the pipeline also allows existing methods for realistic lighting and shading of leaves and bark displacement to be implemented on top of it.

3 Generating Meshes from Graph Representations

The shaders which were used as a basis for this project use geometry shaders to create the mesh for the tree graph edges, which are input as an array of line primitives first vertex representing source and the second the target node. Similarly, the leaves are billboards with an alpha masked texture, generated from point primitives.

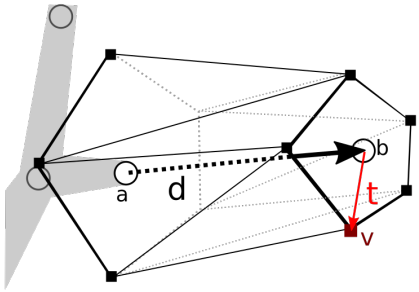

Figure 3: *Generating branch meshes*

For source and target node positions $\boldsymbol{a}, \boldsymbol{b}$, tangent $\boldsymbol{t}$, thickness of the branch at a node s, number of vertices per loop n and vertex index i in the loop, we can calculate the position of a vertex for a segment as follows:

$$\boldsymbol{d} = \boldsymbol{b} - \boldsymbol{a}$$
$$\theta = 2\pi \cdot \frac{i}{n}$$
$$\boldsymbol{v}' = \sin\theta \cdot d + \cos\theta \cdot (\boldsymbol{d} \times \boldsymbol{t})$$
$$\boldsymbol{v} = s \cdot \hat{\boldsymbol{v}}'$$

For texture coordinates, each node contains the distance from the root which is used for the v coordinate, while the u coordinate can be easily computed with $u = \theta/2\pi$.

4 Optimization Strategies

Creating the tree mesh geometry dynamically on the GPU every frame has the advantage of allowing animation through the tree graph structure and leaf particles as well as adapting the amount of geometry produced on the fly. We therefore present those optimization strategies first, which retain these properties to finally mention those which sacrifice these properties for performance.

4.1 Avoiding Normalization

During optimization of the shader, the driver cannot know whether inputs are normalized, which would allow avoiding superfluous `normalize` function calls. If we prenormalize the parts of data in our vertex buffers that is only used after normalization in the shader, we can save performance not doing so on the GPU every frame.

Further `normalize` calls can be avoided by making use of simple equations, e.g.:

$$a, b \in \mathbb{R}^3, |a| = |b| = 1 \Rightarrow |a \times b| = 1$$

$$a \in \mathbb{R}^3, |a| = 1 \Rightarrow \text{normalize}(a) = \frac{a \cdot a}{|a|} = a$$

We can manually avoid `normalize` whereever possible by keeping track of which variables are already normalized and do not require renormalization after operations which do not affect the vectors length.

4.2 Uniform Buffers

Uniform buffers (UBOs) can be used to store uniform values on the GPU to later be bound in one call as opposed to resubmitting the values to be sent to the GPU. UBOs are useful for reusing uniform values across shaders and for reusing those uniform values which do not change between frames.

Trees are static objects, their world transformation will not change between frames and can be loaded into a uniform buffer at application startup to be bound when rendered. This avoids the world transformation matrix to be resent to the GPU every frame.

In addition we can use a single global uniform buffer for all trees containing the view matrix which can be bound once per frame instead of transferring the view matrix to the GPU for every tree.

We can even share the transformation and view matrices between the leaf and tree trunk shaders.

4.3 Trigonometric Function Look Up Table

To determine the vertices for the trunk of the trees, `sin` and `cos` are required. These are calculated in fixed steps given the amount of desired vertices per ring in the pipe generated for the trunk.

Since these fixed steps are the same for every ring in the tree, the same sine and cosine values are calculated. By providing a simple look up table (LUT) in form of a constant float array (or alternatively a shader storage buffer) in the GLSL shaders, we avoid these redundant calculations.

4.4 Caching Geometry Shader Results

When using mutliple passes (e.g. pre-Z, shader pass and color pass) the geometry shader is run multiple times to generate the required tree geometry. In addition, simulation or animation may be performed on-the-fly in the vertex shader rather than changing the models data, resulting in duplicate execution aswell.

This can be avoided by caching the geometry shader result either through transform feedback or by moving mesh generation into a compute shader which is dispatched before any of the passes are run. The generated geometry can then be drawn every pass using a simpler version of the fragment and vertex shaders.

Since the vertex data for the tree meshes needs to be stored throughout the entire frame this way, rather than being temporarily held between geometry and fragment shader stages in the pipline within a draw call, larger amounts of RAM are required on the GPU. This could be reduced by compressing the vertex data.

4.5 Almost Zero Driver Overhead API

Next-Gen APIs like DirectX 12 and Vulkan provide a way of graphics programming with "Almost Zero Driver Overhead" (AZDO). The above methods allow reducing workload on the GPU, allowing us to issue many draw calls. By implementing the tree shaders in an AZDO API, we can minimize the driver overhead to further reduce workload also on the CPU.

We chose Vulkan for its novelty and because it is available on multiple platforms (e.g. Windows, Linux, Android, etc.) while DirectX is only available on Windows. Furthermore, Vulkan supports loading our existing GLSL shaders using the `NV_glsl_shader` extension.

Vulkan provides a way to batch multiple draw calls into so called *Command Buffers* which are submitted to queues for graphics devices to process. Using these we can combine all draw calls into one.

4.6 Pre-Z Pass

Dense nature scenes contain alot of geometry from different meshes which overlap but do not completely occlude each other, since the treetops are porous. To avoid shading pixels which are overwritten by another mesh in a later draw call (so called overshading), we can make use of a pre-Z pass.

In this pass we render the entire scene only to the depth buffer and later set the depth function to *equal* ensuring merely the pixels of the objects closest to the camera pass the depth test and reach the fragment shading stage.

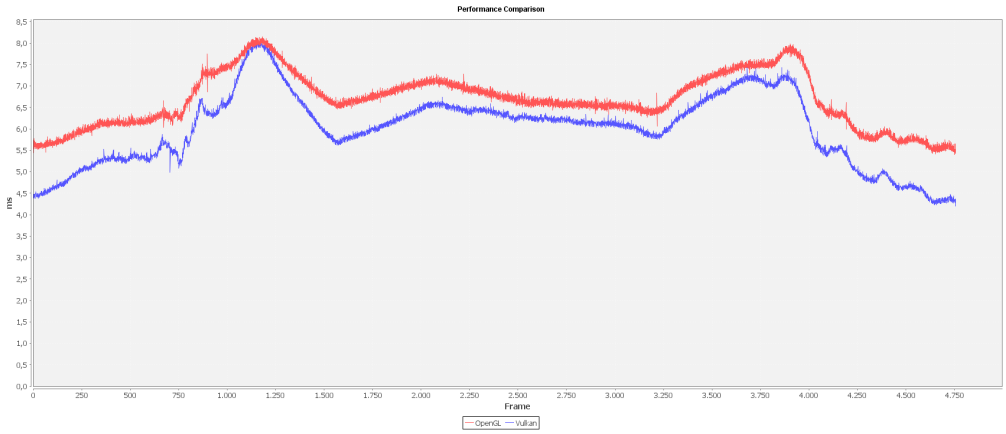

Figure 4: *Comparison of performance using OpenGL vs Vulkan.*

4.7 Depth Sorting

While a pre-Z pass can reduce overshading, it does not reduce overdraw. Overdraw happens when pixel is written and later overwritten because a later draw call resulted in a pixel at the same location, but closer to the camera. The pre-Z pass still writes the same amounts of pixels to the depth buffer as a color pass would have. The only difference is, that the fragment shader for pre-Z is simplistic and removes the need to calculate the color for the overdrawn pixels.

To minimize overdraw, ideally all triangles would be rendered in front to back order. This way all occluded pixels of triangles behind another would be discarded through depth test. But considering the order of triangles is view dependent, all triangles would need to be sorted after any update of view transformation up to every frame.

Instead of sorting all of the geometry, we can sort the trees according to their distance to the camera, which already yields improvement over unsorted draw calls, which in worst case potentially occur back to front order.

4.8 Offline Mesh Generation

Generating the mesh of the trees dynamically every frame is very GPU-heavy which results in rendering being GPU bound: the graphics hardware is working on complex draw calls while the CPU is idle and waiting for it to complete.

By simplifying the shaders we can reduce the load on the GPU. If we remove the dynamic mesh generation code all together and instead generate the mesh beforehand on the GPU once at application startup, the shaders are reduced to a specialized phong implementation.

This however also completely removes the benefits of the on-the-fly generation of tree geometry.

4.9 Skinned Offline Generated Meshes

When associating each vertex with the group of edges in the tree graph they belong to, we can combine the benefits of on-the-fly rendering with offline mesh generation.

We hereby reduce the mesh generating problem to a skinning problem for which every vertex is affected by at most two bones (the tree graph edges), possibly just one, if the vertices which at the intersections of two segments are duplicated. We can then further simplify the tree graphs data by representing the transformations of bones as dual quaternions rather than source and target nodes, saving memory, even when storing additional data like thickness which, while not needed for mesh generation anymore, may still be useful for simulations.

Since the leaves need to follow the branches, we can skin them using this same method aswell, further unifying the pipeline up to the fragment shader, which could further be unified using an ubershader.

5 Implementation and Results

We implemented the tree shaders in C++ and GLSL using the Magnum OpenGL C++11/14 graphics engine and pugixml for loading the tree graphs efficiently from an xml based Xfrog format. The following sets of optimizations were applied:

OpenGL:

- Avoiding Normalization
- Uniform Buffers
- Trigonometric Function LUT
- Pre-Z Pass
- Depth Sorting
- Generating Tree Meshes via Compute Shaders

Table 1: *Amounts of data in the benchmark scene*

	input vertices ($2 \cdot edges$)	input points (leaves)	vertices trunk	vertices leaves	total vertices
Robinia Pseudoacacia	21 242	7 594	106 210	30 376	136 586
Salix Alba	15 798	19 523	78 990	78 092	157 082
Palm	3 288	18 316	16 440	73 264	89 704
Lagerstroemia	30 740	18 316	153 700	73 264	226 964
Aesculus hippocastanum	26 716	8 193	133 580	32 772	166 352
Sorbus torminalis	7 180	10 365	35 900	41 460	77 360
Entire forest	1 049 640	823 070	5 249 200	3 292 280	8 540 480

Vulkan:

- Avoiding Normalization
- Uniform Buffers
- Trigonometric Function LUT
- AZDO API

The benchmarks consist of rendering a camera animation along a path using dual quaternion screw interpolation through a forest containing 60 trees, independently rendered as if they were individual tree graphs, but using ten copies six unique graphs. Table 1 shows the amount of input data and geometry rendered each frame. The images are rendered at HD resolution (1280x720 pixels) without antialiasing.

The benchmarks are run on a desktop computer with an Intel Core i7-4790K 4.0GHz Processor, 16GB RAM and a NVidia Geforce GTX 970 GPU. Frame times are measured starting with the first draw call (after animation interpolations) and ending after a flush/wait for GPU hardware to complete after the last draw call (before present).

5.1 Trigonometric Function LUT

While this only results in a minor performance improvement of $\sim$ 0.2 ms, the more important effect to note is the stability of the frame times compared to using trigonometric functions, to be seen in Figure 5.

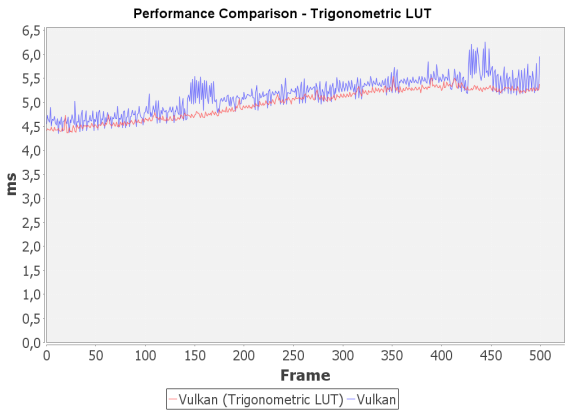

Figure 5: *Comparison of performance using a LUT.*

5.2 AZDO API

The geometry shaders which create the trunk mesh aswell as the great amount of geometry for the leaves put high load onto the GPU. The matching CPU overhead is not very significant and as a result the performance improvements achieved by using Vulkan as opposed to OpenGL are merely $\sim 0 - 1.5$ ms.

Figure 4 shows the performance differences for our benchmark setup.

5.3 Geometry Caching using Compute Shaders

We rewrote the mesh generation algorithm to use GLSL compute shaders on OpenGL. Using these we just achieved a very big slowdown compared to the geometry shader implementation, shown in Figure 6.

After experimenting with improvements to the compute shaders code, memory layouts and local sizes and dispatch sizes, the performance did not improve much.

It may be that this is not a task well suited for compute shaders, but we assume this can also be explained with lack of experience with compute shaders.

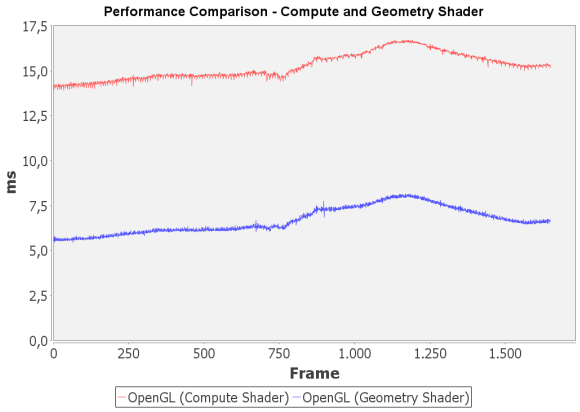

Figure 6: *Comparision of performance using compute shaders to cache generated geometry vs. previous method.*

5.4 Pre-Z Pass

Because our implementation did not make use of geometry caching (due to its bad performance shown above), the geometry shader ran twice, once for the pre-Z and once for the color pass.

The measurements in Figure 7 show a slight decrease in performance which can be explained with the incresed framebuffer writes. In addition, the fragment shaders are not complex enough to justify an entire pass to reduce their invocations.

If geometry is cached for both passes and more elaborate shaders are used which could make use of shadow mapping and an approximation for translucency for example, this technique can be more useful.

5.5 Depth Sorting

Depth sorting, similar to the trigonometric function look up table, only slightly improves overall performance for the average case. But for cases as seen in Figure 7 around frame 1100, in which a large tree in the foreground occludes most of the other trees in the background, the render times are significantly improved, stabilizing overall performance.

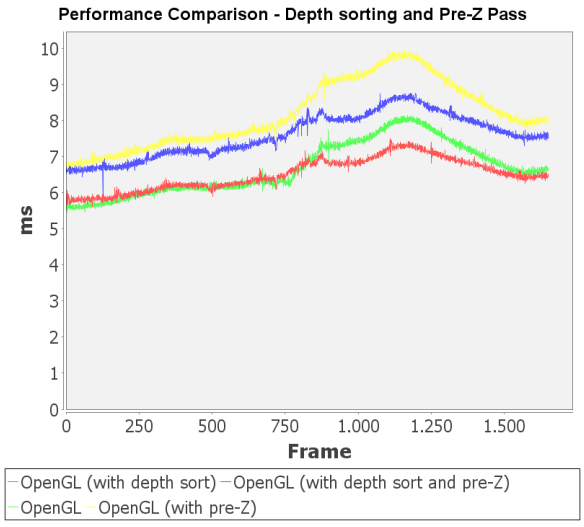

Figure 7: *Comparison of performance for pre-Z and depth sorting.*

6 Conclusion

We compared various techniques for potentially improving the performance of our tree rendering pipeline, some of which turned out more successful than others. The Vulkan implementation, even without depth sorting, performed better than the OpenGL implementation with depth sorting. By reducing driver overhead, next-gen APIs like Vulkan can help with CPU load. With depth sorting, the GPU load can be further decreased as shown with our implementation, to allow for more draw calls which in turn make the use of Vulkan more effective.

Even though the compute shader implementation performed very badly in comparison to the geometry shader, the underlying idea of caching the geometry shader result could still be implemented using transform feedback. Downside of transform feedback is that Vulkan currently does not implement this feature.

While implementation of rendering using skinned offline generated meshes was out of the scope of this project, it would be interesting to see how it would compare to on-the-fly generation.

7 Acknowledgements

Many thanks go out to both Julian Kratt, my Mentor during this bachelor project, and Vladimír Vondruš, the developer of the C++11/C++14 and OpenGL graphics engine Magnum, which was used extensively throughout the project.

References

AKENINE-MOLLER, T., MOLLER, T., AND HAINES, E. 2002. *Real-Time Rendering*, 2nd ed. A. K. Peters, Ltd., Natick, MA, USA.

ANONYMOUS, 2016. Vulkan official website. https://www.khronos.org/vulkan/, Mar.

ANONYMOUS, 2017. Speedtree website. https://www.speedtree.com/, Jan.

BOUDON, F., MEYER, A., AND GODIN, C. 2006. Survey on Computer Representations of Trees for Realistic and Efficient Rendering. Tech. Rep. RR-LIRIS-2006-003, LIRIS UMR 5205 CNRS/INSA de Lyon/Universit Claude Bernard Lyon 1/Universit Lumire Lyon 2/cole Centrale de Lyon, Feb.

CANDUSSI, A., CANDUSS, N., AND HLLERER, T. 2005. Rendering Realistic Trees and Forests in Real Time. In *EG Short Presentations*, The Eurographics Association, J. Dingliana and F. Ganovelli, Eds.

COURRÈGES, A., 2016. Doom (2016) - graphics study. http://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/, Sept.

DEUSSEN, O., COLDITZ, C., STAMMINGER, M., AND DRETTAKIS, G. 2002. Interactive visualization of complex plant ecosystems. In *Proceedings of the*

Conference on Visualization '02, IEEE Computer Society, Washington, DC, USA, VIS '02, 219–226.

ELLIOTT, I., HALL, J., JONES, J., AND ET. AL., D. R., 2016. Vulkan 1.0.6 - a specification. https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf.

HABEL, R., KUSTERNIG, A., AND WIMMER, M. 2009. Physically guided animation of trees. *Computer Graphics Forum (Proceedings EUROGRAPHICS 2009) 28*, 2 (Mar.), 523–532.

HABEL, R. 2009. *Real-time Rendering and Animation of Vegetation*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria.

HARRIS, M. J., 2004. Normalization heuristics. http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/OpenGL/src/normalization_heuristics/docs/Normalization_Heuristics.pdf, July.

KAPOULKINE, A., 2017. pugixml website. https://www.pugixml.org/, Jan.

KAVAN, L., COLLINS, S., OULLIVAN, C., AND ZARA, J. 2006. Dual quaternions for rigid transformation blending. *Technical report, Trinity College Dublin*.

KHARLAMOV, A., CANTLAY, I., AND STEPANENKO, Y. 2007. Next-generation speedtree rendering. In *GPU Gems*, vol. 3. Addison Wesley Professional, Dec., ch. 4.

PIRK, S., NIESE, T., DEUSSEN, O., AND NEUBERT, B. 2012. Capturing and animating the morphogenesis of polygonal tree models. *ACM Trans. Graph. 31*, 6 (Nov.), 169:1–169:10.

PIRK, S., NIESE, T., HÄDRICH, T., BENES, B., AND DEUSSEN, O. 2014. Windy trees: Computing stress response for developmental tree models. *ACM Trans. Graph. 33*, 6 (Nov.), 204:1–204:11.

SELLERS, G., AND KESSENICH, J. 2016. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. OpenGL Series. Addison Wesley.

TRUELSEN, R., AND BONDING, M. 2008. Visualizing procedurally generated trees in real time using multiple levels of detail.

VONDRUŠ, V., 2016. Magnum github repository. https://github.org/mosra/magnum, Mar.

ZIOMA, R. 2007. Gpu-generated procedural wind animations for trees. In *GPU Gems*, vol. 3. Addison Wesley Professional, Dec., ch. 6.