Code Documentation - Advanced Rendering of Line Data

Elias Erhardt & Thomas Seirlehner

Our project involves advanced rendering of line data using ambient occlusion and transparency. The main goal of the underlying paper by Groß and Gumhold [1] is to enhance the visualization of large line data sets by adding global illumination effects to improve the perception of 3D features. This approach builds GPU-based raycasting of rounded cones (truncated cones with spherical endcaps). Additionally, it performs object-space ambient occlusion using voxel cone tracing and has a novel fragment visibility sorting strategy.

The goal of our project is to get similar results to the paper but using DirectX11 + HLSL instead of OpenGL + GLSL, as our API (both high-level APIs). We faced some difficulties in our progress to get to the final result. Our first approach was to use DirectX12 (low-level API) to have more flexibility in our resource management and to use higher shader models. Unfortunately, we had major challenges with this approach that cost us very much time to the point where we had to decide to switch to DirectX11, where these challenges would not occur, to be able to finish the project in time. An additional help for our project was an implementation of the same paper by a group of TU Vienna students, who did their approach in Vulkan + GLSL (low-level API).

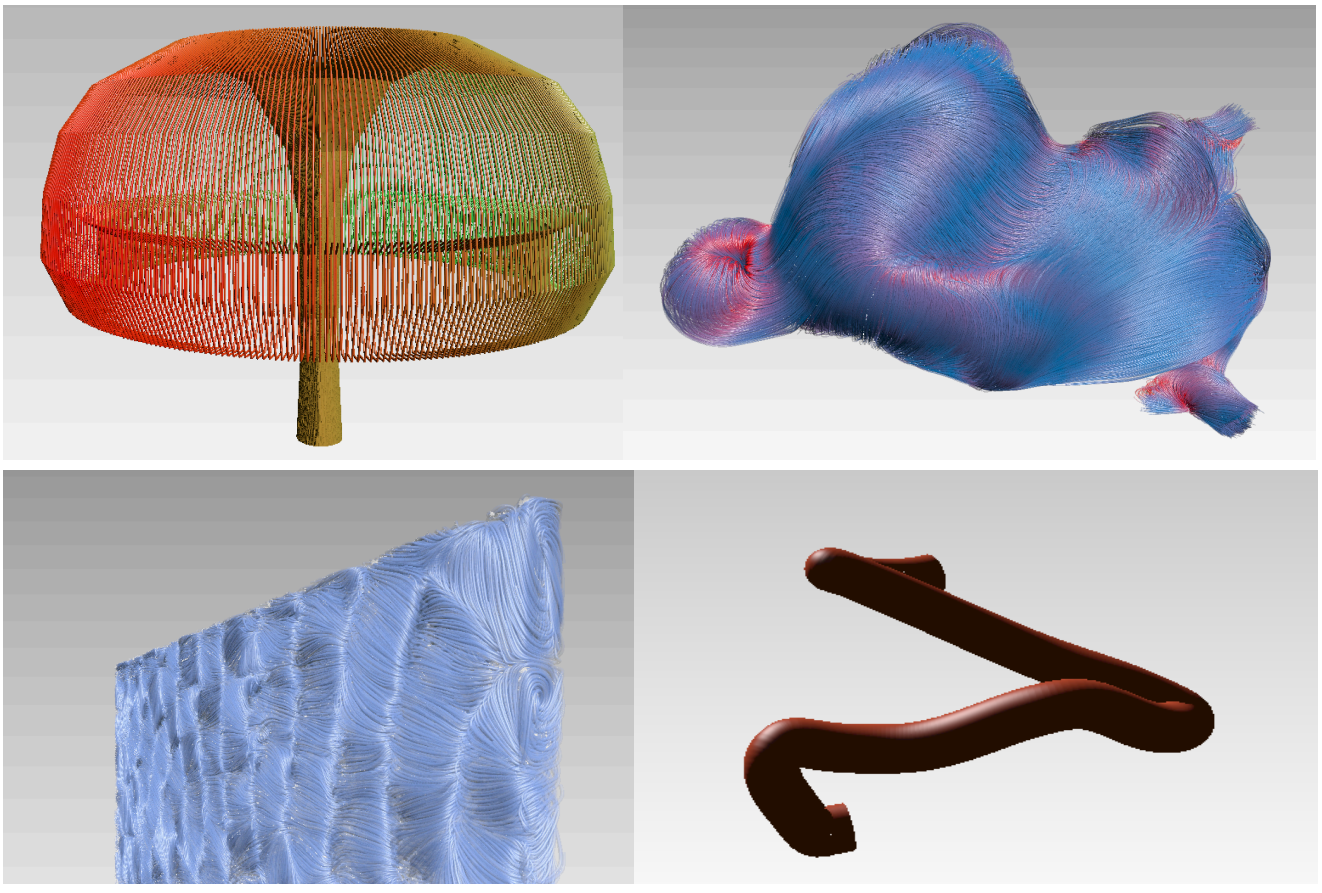
Our step-by-step approach:

1. Our first goal was to implement a compute pipeline combined with a graphics pipeline to render a simple cube.
2. Then we also added a geometry shader to the graphics pipeline to be able to render billboards out of simple lines.
3. We went through the different shaders and implemented them according to the paper and the shaders of the Vulkan-approach we found.
4. We created a very simple line data set, which consisted of just a single line.

5. The next step was to render this simple line into a tube (go through the whole pipeline) which then already looked very good.
6. To successfully implement the novel approach of the paper regarding depth visibility we then had to add another shader pipeline consisting of only a vertex and pixel shader. These shaders are based on a kbuffer which is created in our compute shader, overwritten in our first pixel shader, and read in this second pixel shader. With this second pipeline we were then able to show different lines according to their depth-layer.
7. The overall functionality was now done, so we also implemented a simple background shader pipeline, so the user is later able to change background color if needed.
8. After that, we wanted to make our program more user-friendly so we added a GUI with the “imgui”-library. With that, we added some functionality so the user can change the background color, tube color, transparency, materials & lighting, and the kbuffer layers.
9. Now that everything worked very well with a simple line, we went on to implement the loading of .obj-files and data preprocessing. This data preprocessing step splits the data inside the .obj-files into different arrays. To do this, the data has a special structure, where a ‘v1’ at the beginning of a line, for example, stands for a vertex inside a line, and a ‘l’ displays the end of a line. With these two and other letters in the dataset, we can split the data into different lines.
10. There are now two different approaches which we tried to draw these lines. Our first approach was to save draw calls (how many vertices; at which vertex does the line start) for every line and then draw the single lines in a for-loop. Unfortunately, one of our two main issues regarding DirectX11 now appeared. Since we have limited flexibility in our resource management this for-loop caused major performance issues (in some datasets we had more than 10 millions lines) and therefore we couldn’t use this approach.
11. Instead, we tried indexing to be able to draw the whole data set with a single draw call. This caused some issues regarding the quality of the result at the beginning, but after we fixed that, we had good results and the performance was much better. The only downside to this approach is that the “endcaps” of single tubes are not rendered, so in smaller datasets, this might reduce the overall quality of the results.
12. In our last step we faced our final major problem with DirectX11. When loading and rendering larger datasets (than a simple line), we realized that our depth sorting was not working correctly. This was due to an atomic operation which essentially sorts the different layers from smallest to largest according to their depth and assigns the right color to them.
    To break this problem down: In the Vulkan-approach we found, a uint64 is used to store the color+depth in a single unsigned integer. Then this single integer can be used to atomically compare it to another single integer and sort them. With this approach the entire sorting and writing into the kbuffer

happens within a single atomic operation and therefore there are no/fever artifacts.
In our approach, we use a uint2 (a 2D-vector of 32bit integers), since a uint64 is not supported with our shader model. Unfortunately, there are no atomic operations that can compare two different integers with another two integers simultaneously. So we had to compare just the depth-values atomically and then assign the color according to the sorting of the depth. But this second writing into the kbuffer is not atomic, which leads to artifacts in the final render. In order to fix this we would need to either be able to atomically compare vectors of uint, which is not possible in DirectX11/HLSL, or change our uint2 to a uint64. Unfortunately, uint64 is only supported in shader models 6.x and DirectX11 only supports shader models up to 5.x.

Results from our project:



[1]D. Gross and S. Gumhold, “Advanced Rendering of Line Data with Ambient Occlusion and Transparency,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 614–624, Oct. 2020, doi: https://doi.org/10.1109/tvcg.2020.3028954.