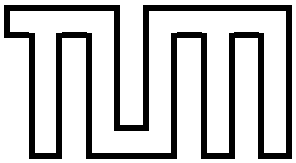


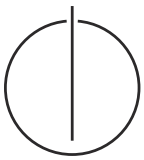
DEPARTMENT OF INFORMATICS

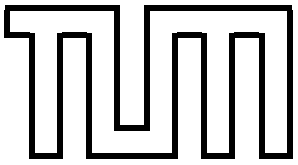
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Opacity-Based Rendering of Lagrangian Particle Trajectories in Met.3D

Maximilian Bandle





DEPARTMENT OF INFORMATICS

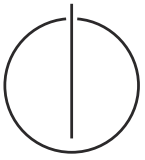
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Opacity-Based Rendering of Lagrangian Particle Trajectories in Met.3D

# Opazitätsbasiertes Rendering von Lagrangeschen Partikeltrajektorien in Met.3D

| | |
|---|---|
| Author: | Maximilian Bandle |
| Supervisor: | Prof. Dr. Rüdiger Westermann |
| Advisor: | Dr. rer. nat. Marc Rautenhaus |
| Submission Date: | September 15, 2016 |



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, September 14, 2016 Maximilian Bandle

# Abstract

This bachelor's thesis presents a feature extension to the trajectory actor of the meteorological visualization software Met.3D. It first explains the difficulties of handling multiple layers of transparency and presents a highly optimized Order Independent Transparency approach which is capable of displaying the scene in real-time. Upon this foundation, several parameters such as trajectory ascent or curvature are mapped to the opacity in combination with a technique to reduce the occlusions.

# Zusammenfassung

Diese Bachelor-Arbeit präsentiert eine Funktionserweiterung der Trajektorien-Komponente von Met.3D. Dabei werden zunächst die Schwierigkeiten beim Rendern von Szenen mit transparenten Ebenen aufgezeigt. Anschließend wird ein optimierter *Order Independent Transparency* Ansatz präsentiert, der in der Lage ist diese Szenen in Echtzeit darzustellen. Darauf aufbauend werden verschiedene Faktoren, wie Aufstieg oder Krümmung einer Trajektorie, mithilfe von Transparenz dargestellt. Um die Sichtbarkeit von relevanten Komponenten zu erhöhen, wird eine Technik zur Reduktion von Überdeckung vorgestellt.

# Contents

# 1. Introduction

Visualization tools are needed by meteorologists to explore their massive datasets gathered throughout campaigns or computed by various numerical models. Until today most visualization tools use only a two-dimensional (2D) data view which might hide some important features. Meteorologists often need to use their visual imagination capabilities, like physicians when examining X-Ray images, to create a 3D picture of a meteorological situation from various cut-planes. [Rau15]

Even though the globe is a three-dimensional sphere, 3D visualization methods are not widely accepted by meteorologists because they are, like physicians at the hospital, used to 2D-visualizations [Szo+03].

Furthermore, additional effort in meteorological visualization is required through the implementation of ensemble forecasts. An ensemble forecast incorporates multiple runs of weather models with different numerical prediction models or slightly varied initial conditions. By that, forecast uncertainty is represented better than in a single run. [GR05] Therefore new visualization and post-processing methods have to be developed, which is challenging since 3D ensemble visualization software is a new area. [Rau+15b]

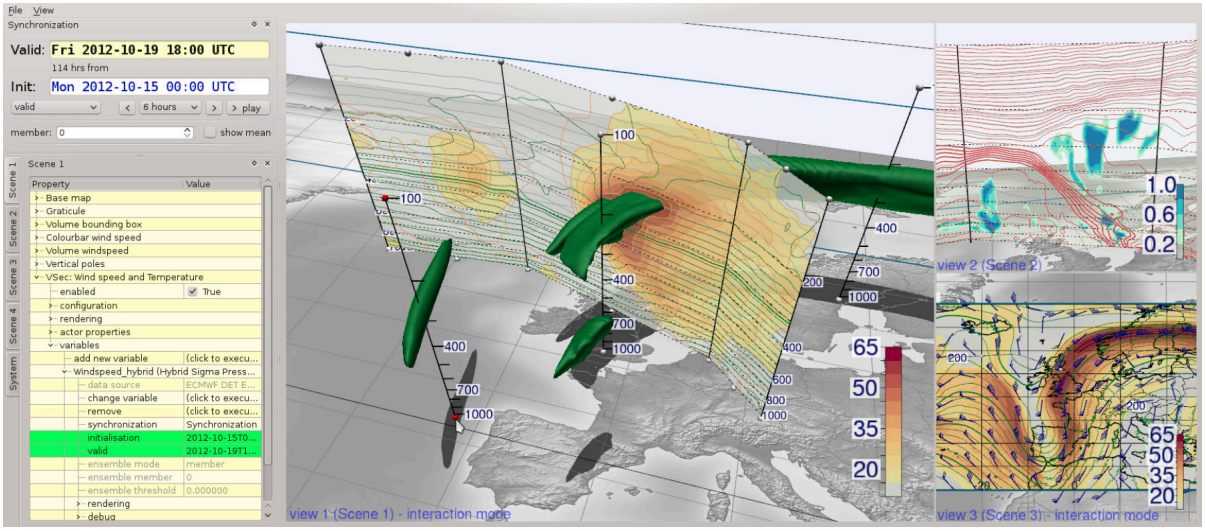



**Figure 1.1.:** Combining 2D and 3D features with multiple scenes in Met.3D (Figure adapted from [Rau15])

*Met.3D* is an open-source tool developed at TU München. [Rau+15b; Rau+15a] The tool embeds the 2D cut-planes and weather maps into a three-dimensional space and also implements

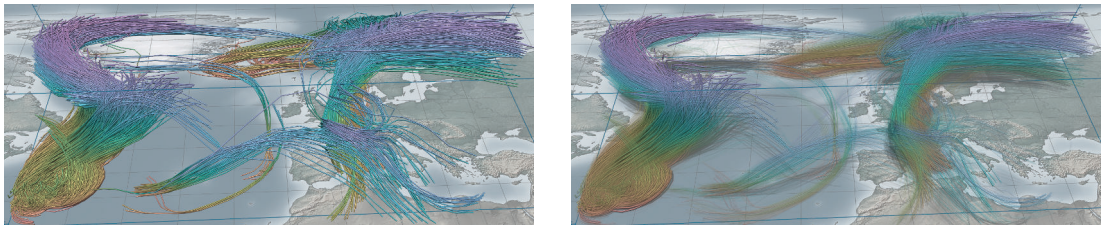
3D features, like iso-level surfaces or trajectories. As a result, composite scenes as shown in Figure 1.1 can be produced. The scene can be changed by the user to view it from different angles and zoom levels.

These features are achieved by the tools modular architecture. Each specific visualization task, like iso-surfaces, cut planes or the base map, is encapsulated in an so called actor. With this modular actors, Met.3D is capable of combining different visualization methods into multiple 3D scenes. For example, a 2D cut-plane can be combined with the iso-surface rendering of the same data set, as shown in a visualization of wind-speed in Figure 1.1 [Rau15]. Having this combination of 2D and 3D renderings, the software combines the established 2D renderings, along with 3D renderings in a three-dimensional space.

Moreover, Met.3D is not only modular but also fast, which means ensemble data can be viewed and explored in real-time from different points of view. This is achieved by using GPU acceleration for rendering the actors and sometimes also for processing the data. Because of its vector operation optimized structure, the GPU executes these tasks faster than the CPU. [Shr+13].

## 1.1. Motivation: Improving Met.3D's capability in trajectory visualization

The objective of this thesis is to improve the trajectory[1] actor. At the beginning of the thesis' work, the actor was capable of displaying trajectories and coloring them according to their pressure elevation. It was also possible to set a threshold which was used to filter the trajectories as shown left in Figure 1.2.



Line Filtering (> 500 hPa) | Min 300 hPa | Max 800 hPa | Exp 3

**Figure 1.2.:** Comparison of Met.3D before and after this thesis. The right image shows the enhanced filtering using opacity.

Solely filtering cannot provide context information because all displayed trajectories are rendered coequally not respective to their relevance[2]. There is no possibility to highlight the trajectories with the highest ascent, while not discarding any other [VKG04].

Thus this thesis will evaluate various method for assigning relevance values based on single trajectory features, to make the selection more fine-graded than it was solely using the total

[1] Trajectories are path-lines of tracked particles in a wind field.
[2] Each trajectory is either opaque or discarded.

pressure difference. These factors should be user-controllable to get an entirely customizable rendering outcome.

To be able to display this new kind of parameters, an method to handle transparent tubes, the rendering primitive of path-lines, must be introduced. Therefore several methods are compared and optimized to preserve the interactivity when viewing data in Met.3D.

Furthermore, when reducing the overall amount of opacity throughout a scene, the perception of spatial relationships between trajectories is affected because an increased transparency results in a color desaturation. This might be perceived as an increased distance to the camera. [Kan16].

To encounter this problem, the shadow rendering of the trajectory actor is improved. Now non-flickering partially transparent shadows are rendered. They reduce the noise in the out-coming image and allow a better spatial perception by letting the base-map shine through.

A scene rendered using this methods is shown for comparison in Figure 1.2 along with the used parameters.

Summarized the goal of this thesis is to improve the trajectory actor in a way that is inspired by the state of the art methods used to display relevant features in path-line data and evaluate the results in a meteorological context.

## 1.2. Use Case: Lagrangian Particle Trajectories

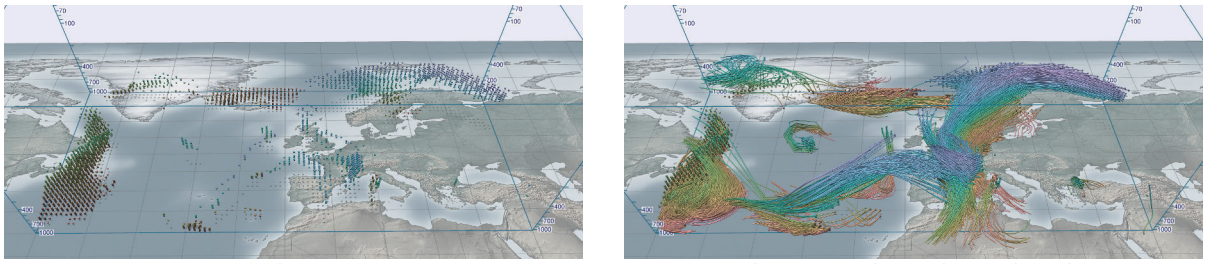
Met.3D's trajectory actor was designed to visualize Lagrangian particle trajectories, often called path-lines in visualization literature. Each path-line tracks a seeded particle in a wind field over a defined time span.

### 1.2.1. Data

Trajectory data used in this thesis was generated during the "T-NAWDEX Falcon campaign". This aircraft-based research campaign took place in October 2012 at Deutsches Zentrum für Luft- und Raumfahrt (DLR) Oberpfaffenhofen. To plan their flights, numerical weather models of the European Centre for Medium-Range Weather Forecasts (ECMWF) were used. [Sch+14]

The trajectories were computed with the LAGRANTO Lagrangian analysis tool by Sprenger and Wernli. It calculates air parcel trajectories based on ECMWF data. [SW15]

For every available forecast time-step, 3D Lagrangian particle trajectories are integrated, starting at a fixed set of seeding points forward in time for 48 hours. This is described in detail by Rautenhaus et al. who uses it for warm conveyor belt (WCB) detection. These are structures of air flow, which transport warm moist air from the near-surface zone upward to near-tropopause levels. Hence the trajectories with a high pressure difference throughout the time are relevant. [Rau+15a]. These seeding points are shown in Figure 1.3, where each sphere stands for a particle's current position.



Spheres | Spheres and Backward Tubes

**Figure 1.3.:** Spheres showing the particle position in a regular grid at seeding time with and without backward tubes showing the previous particle path. Only trajectories and their seeding points with a total ascent of more than 500hPa are shown.

### 1.2.2. Format and Size

The total file size for a time-step of trajectory data is about 2.4 GB. Each of these files is stored in NetCDF (Network Common Data Format) file format, which is specially designed for storing scientific data and is often used for meteorological purposes [RD90]. Each file contains several arrays holding all trajectory information. This is in detail the structural part consisting of latitude, longitude, and height, hence this data is 3D spatial data.

Met.3D includes a NetCDF parser, which reads the trajectory data and stores it as an array in the main memory. After processing the file, calculating normals and the total pressure difference, the data is uploaded to the GPU memory along with the derived data for processing and rendering the data in the actor. [Rau15]

## 1.3. State of the Art Methods

A variety of different techniques exists to display or highlight the most important lines in trajectory data. Trajectory data usually fills the area that is it portrayed densely. Hence it is necessary to emphasize the important trajectories. An example of no filtering method applied is depicted as a reference in Figure 1.4.

You cannot see any relevant inner structure because everything is shadowed by the outer trajectory lines. These lines should not be in the foreground as they provide no useful information. They should mainly provide background and subtle context information. This aim is aspired to by all the presented approaches.

### 1.3.1. Line Selection

*Line selection* is the most basic technique to filter the lines since it discards the irrelevant lines. This technique is already implemented in Met.3D and helps the user to avoid the biggest visual clutter as depicted in Figure 1.4.

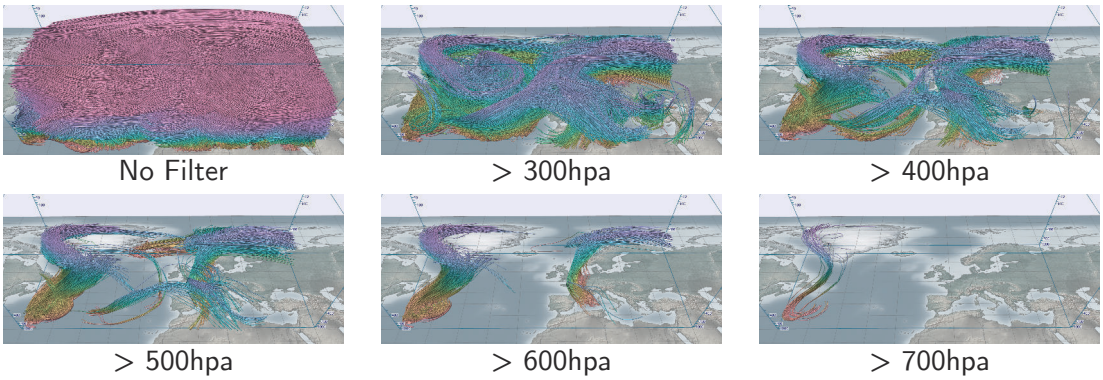



**Figure 1.4.:** Comparison of different already implemented line selection parameters in Met.3D. The higher the threshold value is chosen, the lower is the number of shown trajectories. Hence the relevant feature, here ascent, is more in focus.

The method sets the minimum total trajectory pressure difference as relevance criterium. The user can select which trajectories should be displayed by setting the value of the criterium, which is preset to 500 hPa. This discards every trajectory with a difference less than 500 hPa over all time steps.

As this method either displays the line as a whole or not at all, it does not use transparency. Hence it does not provide further context information by displaying more trajectories, solid or faded, in the background. When only filtering for the most important lines so only showing trajectories with a difference of over 700 hPa, this leads to an almost empty canvas.

### 1.3.2. Direct Mapping

*Direct Mapping* is comparable to assigning, also called mapping, color using a transfer function. When mapping transparency values directly to the trajectory, you can define a parameter to represent the relevance to control the level of transparency.

Different parameters contribute to the relevance of the trajectory segment such as the already known trajectory difference. However, in contrast to line selection, this technique is not limited to whole lines, so parameters are assigned per segment or even per vertex. These vertex parameters are interpolated over the segments to obtain a smooth appearance.

In Figure 1.5 a simple mapping is displayed. The adduced parameter for the importance is solely the trajectory pressure difference. As a result, only whole path-lines are blended. However, in contrast to the line selection approach, you can see that the trajectories have different opacity values, which indicate how significant each of the segments is. In comparison to Figure 1.4 you can also perceive all important structures, but the former empty spaces are filled with some transparent lines.

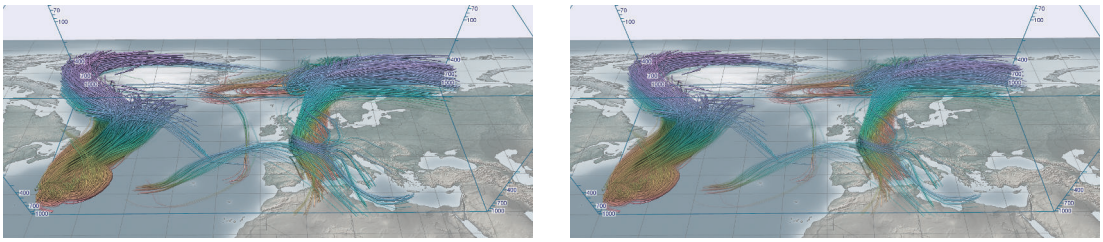



**Figure 1.5.:** Direct mapping of the total pressure difference to the trajectories transparency. The used parameters are explained in detail in Chapter 3.3

### 1.3.3. Opacity Optimization

This technique is more elaborate than the other previously described ones, as its goal is to optimize the opacity to render an image with as many important features highlighted as possible.

Günter et al. [GRT13] therefore use, in addition to the relevance, multiple viewpoint dependent parameters which weightings are controllable by the user. The outcome is shown in Figure 1.6 along with the Adaptive Line Hierarchy described in the next subsection.


**Figure 1.6.:** Comparison of different state of the art methods with an unfiltered image. The middle image shows opacity optimization by Günther [GRT13], the left adaptive line hierarchy by Kanzler [Kan16]. Curvature is used as relevance criterium. (Figure adapted from [Kan16])

Using these parameters, a linear system of equations (LSE) is set up, containing an error function for each trajectory segment. This error function uses parameters like occlusion or coverage along with the user-given weights, to penalize, for instance, segments which shadow a more important one.

This system, containing the error function, is minimized, which leads to an optimal transparency value for each segment in respect to the adduced parameters. Because these opacity values are not calculated for every pixel, they are blended over the segment to get a smooth transition.

Although the LSE is viewport dependent, it does not have to be minimized again to display an image after changing the viewport, but to display the optimal image. This is done by caching the transparency parameters for the segments. As a result, the rendering and solving steps become more independent, which avoids huge performance drops.

The authors tested their approach with up to about $8 \cdot 10^3$ segments, and it worked, solving the equation system once per second. Unfortunately, the trajectory actor handles $3.6 \cdot 10^6$ segments and minimizing an equation system scales not better than linear. To be accurate, the used algorithm minimizes it by intelligent guessing in $O(n)$ but with a too big constant time factor added [CL96]. This would lead to one finished minimization step every ten minutes, which is not sufficient for real-time data exploration.

Beyond that, the LSE has to be stored, and since the matrix has the dimensions of $n \times n$ with $n$ being the number of segments, you need about 40GB to 40TB to store the LSE[3].

Furthermore, this approach is not applicable for this kind of trajectory data because it mainly takes importance into account. When having similar lines, similar opacities are assigned, because their parameters are similar.

### 1.3.4. Line Hierarchy

*Adaptive Line Hierarchy* is a 2016 presented approach by Mathias Kanzler [Kan16], which takes a different decision than the previous approaches by using a pre-process to structure the data.

It does what is missing in all previous approaches, namely bundling similar trajectories using a pre-process to analyze similarities. This pre-process compares all trajectory segments pairwise using a mean difference metric. Therefore, all trajectories are considered as leaves of a tree. In each step, the two most similar lines are joined until a complete hierarchy tree is built.

Afterwards, representatives for each inner node have to be found, which is done by a bottom-up divide-and-conquer approach. For every two leaves, a line is found. This task is not dependent on the relevance value as it only considers the trajectories spatial parameters. As a result, this hierarchy is pre-computed in several minutes and is not reliant on the relevance.

So when applying this approach to the trajectory data, it will be converted into a hierarchical structure which represents all the lines. This structure is the foundation for thinning out the trajectories in the resulting image. Therefore, the lines are numbered according to their position in tree order when traversing it using a breadth first approach. The closer the line is to the root, the more representative is the line. So when only rendering 25% of all lines, they are taken from the top of the tree.

This space-filling approach is combined with four view-dependent attribute maps. Each is filled in one step with the data and afterward gaussian blurred to avoid sharp edges when altering the transparency based on the map.

**Maximum Importance** This parameter holds the highest importance and is used for the depth

[3]This assumption is made using floats needing 4B each to be stored. 40GB is the size when it is a sparse matrix with less than 1‰ filled. 40TB are required for storing all values uncompressed.

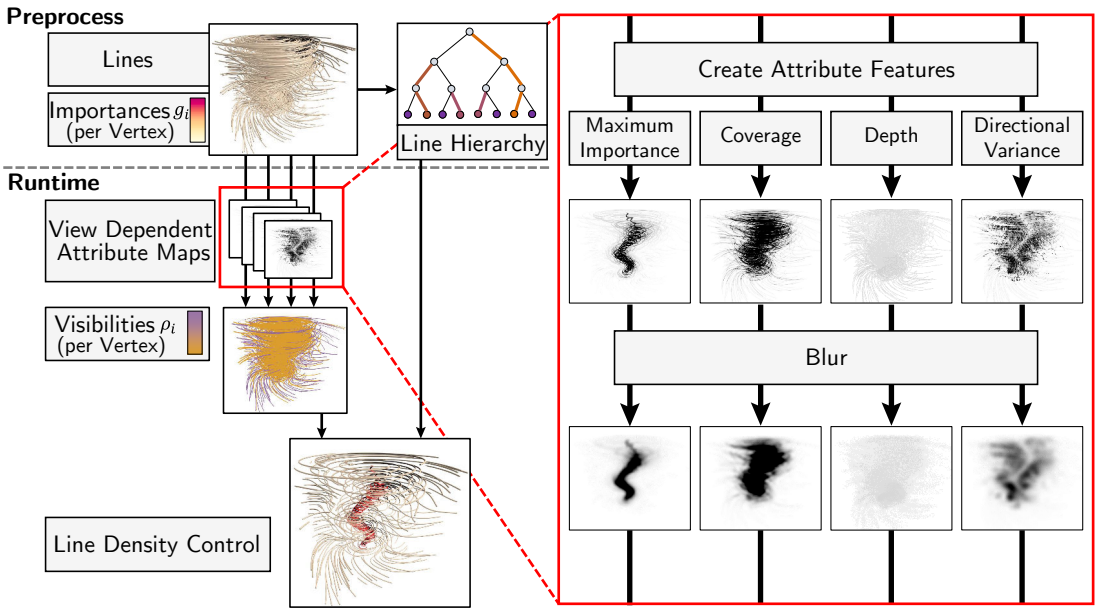



**Figure 1.7.:** Overview of the adaptive line hierarchy approach by Kanzler. The left side shows an overview of the whole process with the combination of line hierarchy and the view-dependent attribute maps. These maps and their generation are shown in detail on the right side. (Figure adapted from [Kan16])

of maximum importance. It indicates that this fragment should be visible in the resulting image.

**Depth** The depth value of maximum importance is stored to be able to filter all fragments before or behind the most important one to have a clear view.

**Coverage** It holds the total sum of overlapping layers at this pixel.

**Directional Variance** This parameter stores the extent of the overlying trajectories pointing into different directions by comparing their gradients.

Afterwards, the visibility from the four parameters and the importance is combined with the hierarchic line set. If the resulting factor is lower than the lines hierarchy value, it is faded out. This leads to a thinned out image with regulated opacity values along the line as shown in Figure 1.6.

## 1.4. Structure and Content of the Thesis

This thesis is structured roughly in the order in which I developed the parts of it. Mainly it starts with the definition of transparency and finalizes in a discussion about how the thesis improved the visualization of the trajectory data set.

Chapter 2, Blending Transparent Objects, lays the foundation for this thesis by presenting the underlying problem; rendering of multiple transparency layers in OpenGL. Multiple common approaches handling this issue are described and compared in Section 2.1. Afterwards, in Section 2.2 the base for the selected *Order Independent Transparency* is chosen because there are many different variations of this technique. Using this knowledge, the actual implementation is presented in Section 2.3. Although the algorithm produces correct scenes, it is not ready to be used as it is too slow. This is first shown using runtime measurements and afterward optimized using various tricks in Section 2.4. It totals in a more than 22 times faster algorithm which makes using transparency in Met.3D interactive.

The following Chapter 3, Opacity Mapping, builds the visualization upon the efficient OIT foundation. Therefore parameters for selecting relevant features are introduced in Section 3.1. These six non-viewpoint dependent metrics and their purpose are described (curvature, length/velocity, angle of ascent, local height difference, global and local pressure difference). To expand the possibilities of the filtering, several view-dependent parameters are introduced (coverage, directional variance, maximum importance and its depth) in Section 3.2. These are inspired by the paper of Kanzler and later used to avoid occlusion. Upon all these parameters, a user controllable importance function is defined in Section 3.3 which is used to control the transparency of the segments. Because of the enormous amount of trajectories, occlusion is a big problem. This section also introduces a penalty to apply to the segments in front of the most important ones. The chapters last section (3.4) describes how all these parameters are integrated into Met.3D and how the user can control all features.

In Chapter 4 the actual improvement of all these changes is analyzed by comparing the visible information in Met.3D before and after the thesis. Furthermore it is analyzed which problems with the algorithm still exist and how they can be solved in the future. To finish the discussion, some prospects are made regarding the used relevance factors and the applied visualization methods.

The conclusion summarizes the main points and achievements of this thesis while emphasizing the benefits and problems of the new methods.

# 2. Blending Transparent Objects

Blending is a technique to handle transparency within objects and superpositions of various opaque and transparent objects. Transparency is all about objects, or fragments as a part of them, not having a final color. Their color is determined by mixing the color of the scene behind with the objects color. [Vri15]

Objects can be opaque, which means it cannot be looked through them at all. Transparent objects are either partially transparent with the background color shining through but altered by the object, or completely transparent. Such objects do not affect the color of the object behind, and the background color remains unchanged when seeing through the object.

The transparency value must be stored to be able to represent the objects opacity. Therefore the alpha value is introduced, which is assigned like the RGB-values to represent the color. Hence the objects color is now encoded by a vector with the four components red, green, blue and alpha[1]. An alpha value of 0.0 indicates a completely transparent object, in contrast to a value of 1.0 which indicates an opaque object.

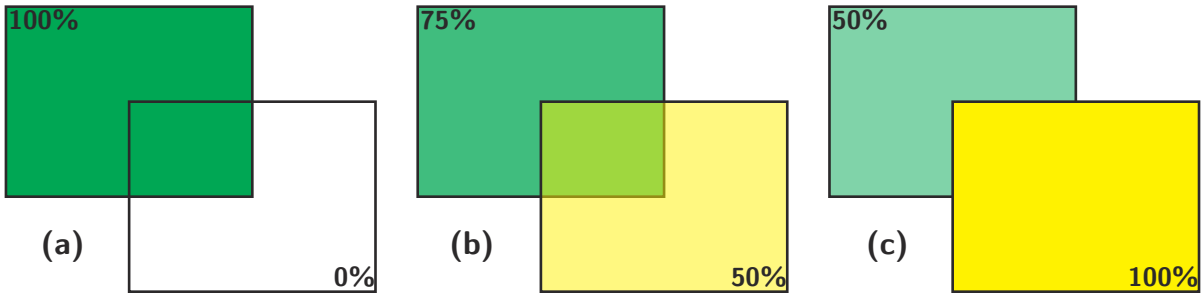



**Figure 2.1.:** Blending of fragments with different transparency values.

Figure 2.1 shows the different results when blending fragments[2] with various opacities.

**Figure (a)** An entirely transparent rectangle is blended over the green one. As stated before, the color remains unchanged.

**Figure (b)** Two partially transparent rectangles are blended over each other. Both the white background and the green fragment is seen through

[1] In this thesis all components of the vector refer to normalized floating point numbers. A blue amount of 1.0 indicates 100% blue color saturation

[2] In the rasterizing stage each object is divided into fragments.

**Figure (c)** A opaque rectangle is blended over a partially transparent one. Every structure behind the front rectangle is shadowed and not visible.

To correctly blend the fragments, they must be composed in the order of their distance to the camera, either in *Back To Front* (BTF) or *Front To Back* (FTB) order. Each order has its blending equation. Using this equation, the color of the marked pixel in Figure 2.1b is determined.

#### Back To Front Blending Equation

$$\mathrm{RGB}_{\mathrm{front}} = \alpha_{\mathrm{front}} \cdot \mathrm{RGB}_{\mathrm{front}} + (1 - \alpha_{\mathrm{front}}) \cdot \mathrm{RGB}_{\mathrm{back}} \tag{2.1}$$

$$\alpha_{\mathrm{front}} = 1.0 \tag{2.2}$$

$\mathrm{RGB}\alpha_{\mathrm{front}}$ is the resulting color vector after each blending step. Because BTF depends on an entirely opaque background to blend the fragments over, the resulting vectors $\alpha$ value is always set to 1.

The example in Table 2.1 uses the paper as a white background to start the blending.

| **Fragment** | **RGB$\alpha$** | **Depth** | **Blended Result** |
|---|---|---|---|
| White Background | $(1, 1, 1, 1)$ | $\infty$ | $(1, 1, 1, 1)$ |
| Green Rectangle | $(0, 1, 0, 0.75)$ | 2 | $(0.25, 1, 0.25, 1)$ |
| Yellow Rectangle | $(1, 1, 0, 0.5)$ | 1 | $(0.625, 1, 0.125, 1)$ |

**Table 2.1.:** BTF blending of fragment b from Figure 2.1 using the BTF blending equations (2.1 & 2.2)

#### Front to Back Blending Equation

$$\mathrm{RGB}_{\mathrm{back}} = (1 - \alpha_{\mathrm{front}}) \cdot \mathrm{RGB}_{\mathrm{back}} + \mathrm{RGB}_{\mathrm{front}} \tag{2.3}$$

$$\alpha_{\mathrm{back}} = \alpha_{\mathrm{front}} + \alpha_{\mathrm{back}} \cdot (1 - \alpha_{\mathrm{front}}) \tag{2.4}$$

$\mathrm{RGB}\alpha_{\mathrm{back}}$ is the resulting color vector after each blending step. The FTB blending equation starts with a transparent viewing ray. Hence the resulting colors alpha factor can be lower than 1 indicating a partially transparent fragment. In this case, the color values have to be corrected by dividing the RGB component by $\alpha$ because otherwise, the saturation would be lower than it should be.

## 2.1. Correct Rendering of Transparent Scenes

The correct rendering of super-positioned objects is nontrivial since the *Graphics Processing Unit* (GPU) typically uses methods not suited for the task [Shr+13]. Hence assigning an alpha value to all transparent objects will not necessarily produce the desired outcome. It leads to the artifacts shown in Figure 2.2.

| **Fragment** | **RGB$\alpha$** | **Depth** | **Blended Result** |
|---|---|---|---|
| Yellow Rectangle | $(1, 1, 0, 0.5)$ | 1 | $(1, 1, 0, 0.5)$ |
| Green Rectangle | $(0, 1, 0, 0.75)$ | 2 | $(0.5, 1, 0, 0.875)$ |
| White Background | $(1, 1, 1, 1)$ | $\infty$ | $(0.625, 1, 0.125, 1)$ |

**Table 2.2.:** FTB blending of fragment b from Figure 2.1 using the FTB blending equations (2.3 & 2.4)

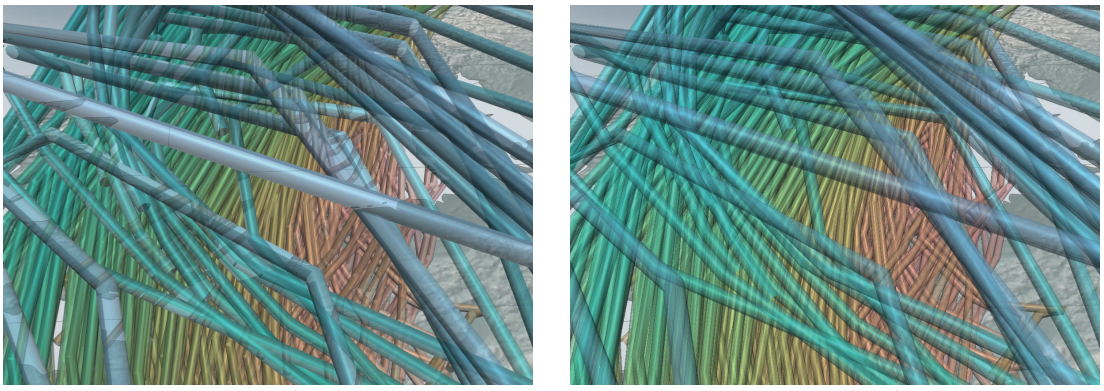



**Figure 2.2.:** Comparison of an incorrect and a correct blended scene in Met.3D. In both scenes all trajectories have an opacity of 50%. The left scene is rendered with an unaltered version of Met.3D, the right is rendered with the approach implemented in Section 2.3.

To understand this problems root, the rendering of a scene consisting solely of opaque objects has to be explained first. The GPU uses a depth buffer, also called *Z-Buffer*, additionally to the usual screen buffer which is displayed. This buffer helps to determine which fragment is nearest to the image plane as illustrated in Figure 2.3. [BGZ02]

After rasterizing a fragment into pixels, their depth is first compared with the depth buffer (comparison and merging operation symbolized as ∪ in the figure). When the depth is lower than the value stored in the buffer, it is visible as it shadows all the fragments previously rendered. Hence its color is stored in the screen buffer and the depth in the according buffer. These steps are repeated for every fragment without ordering them because the result does not depend on the order of processing.

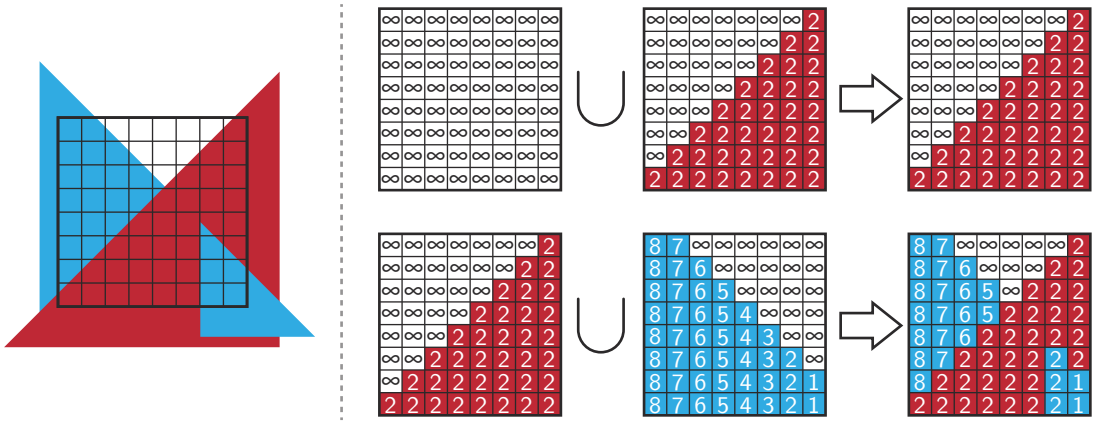


**Figure 2.3.:** Correct generation of an image scene, holding only opaque objects, using a depth buffer. The fragments are blended after each other with respect to their depth values.

When rendering scenes with transparent objects, the depth test cannot be used in the same manner anymore, since the objects behind the first fragment do matter. Their colors are needed to determine the resulting color of the transparent fragment, as they may shine through. However, they also cannot be blended in any distinct order over each other as it matters for the outcome which fragment is blended when.

Wrong order directly leads to the image artifacts as they result default GPU blending. It blends the fragments over each other in the same order they are received from the buffer. This order is consistent with the order they were written into the memory. Because this is not necessarily the next object in view direction, it can lead to artifacts. Furthermore, when rotating the scene, there cannot be any memory order which always leads to correct scenes.

### 2.1.1. Geometry Sorting

A straightforward way to fix this problem is to sort the fragments in the memory in view direction, which is called *Geometry Sorting*. Therefore, the *Central Processing Unit* (CPU) has

to sort all the fragments every time the viewpoint is changed and pass the screen sorted list to the GPU. In theory, the sorted fragments are blended in memory order, and a correct rendering is generated.

Practically, two main issues will be encountered while sorting the polygons, which makes it hard to obtain a correct result. Figure 2.4 illustrates the first problem where three overlapping rectangles are shown. This case cannot be sorted in a proper order without splitting a block as shown with the dashed line because the blocks are overlapping cyclic.

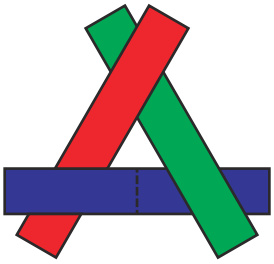


**Figure 2.4.:** Unresolvable case using depth sorting.

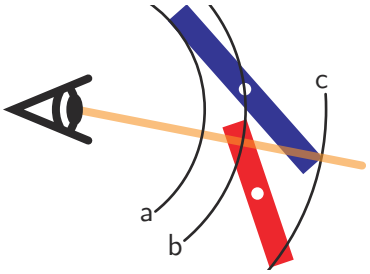



**Figure 2.5.:** Different metrics for finding the nearest object for depth sorting.

The other problem concerns the sorting of the fragments. When looking along the orange ray in Figure 2.5 the red rectangle is in front of the blue one. The three circles, whose midpoint is in the center of the pupil, illustrate three different sorting methods:

**Nearest Point (a)** The blue rectangle is longer than the red one, so its nearest point is closer than the red ones nearest. According to this metric, the blue fragment is wrongly in front.

**Midpoint (b)** The red rectangles midpoint is further away. As a conclusion, the blue fragment is in front by mistake.

**Furthest Point (c)** The furthest point of the blue rectangle is closer than the red ones. According to this metric, the blue fragment is wrongly in front.

As shown in the list above, there is no simple sorting metric ensuring an always correct order. In conclusion, the correctness of the image cannot be assured using transparency sorting. [Bak07]

Furthermore in systems like Met.3D, where the user can freely choose his viewpoint in a scene, the sorting must be performed always after a viewport change. With thousands of partially transparent lines, sorting them on the CPU, is costly.

### 2.1.2. Order Independent Transparency

*Order Independent Transparency* (OIT) is a class of blending techniques, specially designed for multiple transparent objects. The main difference between the classic depth-sorted approach is

the independence of the graphic processor's memory layout and the resulting image because the presented approaches either sort the fragments on the GPU or implicitly obtain them in the correct order.

### Depth Peeling

The phrase OIT was first used for a *Depth Peeling* approach in 2001 [Eve01]. This approach works with multiple passes utilizing depth tests to peel away one layer at a time to implicitly obtain the layers in the correct order.

Originally the *Z-Buffer* was designed to extract only the first layer of a scene which is sufficient for scenes consisting only of opaque objects. However, with a second iteration, the Z-Buffer can be used to extract the second nearest surface in a scene, as each run returns both the depth values and the color (RGB$\alpha$) information. So this technique uses $n$ passes to get $n$ layers deeper into the scene.

The resulting layers are blended with each other to obtain the image, which is a correct representation of the scene if the number $n$ of passes is greater than the maximum number of overlapping transparent layers.

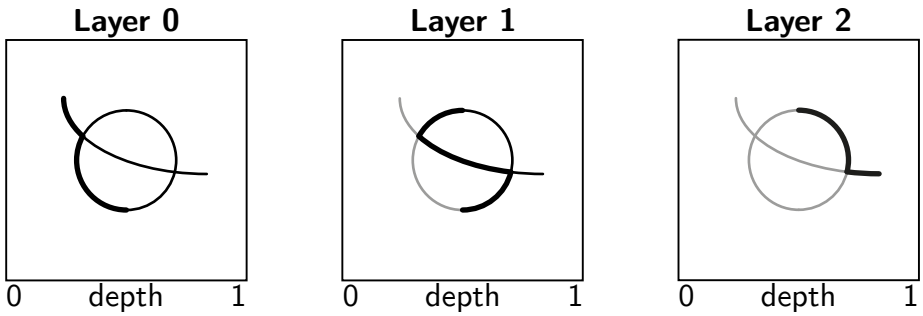



**Figure 2.6.:** Schematic function of Depth Peeling with 3 passes. Depth peeling peels away the frontmost (leftmost) layer which each path. Thick black lines show peeled away surfaces in the current path, gray lines indicate surfaces peeled away in an earlier pass.

Figure 2.6 depicts the peeling process happening after rasterization. In conclusion, the peeled away parts consist of pixels, not whole fragments or even objects.

### Buffer Based Fragment Sorting

Another order independent approach is sorting the fragments on the GPU right before they are finally blended. This type of methods needs a buffer to store the depth and color information of every rasterized fragments pixel. This idea is shown schematic in Figure 2.7. The Bucket symbolizes the buffer storing all the pixels' fragments.

After all the fragments were written to the buffer in their respective place, a second full-screen pass is needed to obtain the resulting color. All stored fragments for a pixel are fetched

and sorted. Now they are in the correct order for being blended using either an FTB or BTF approach. [Mau+11]

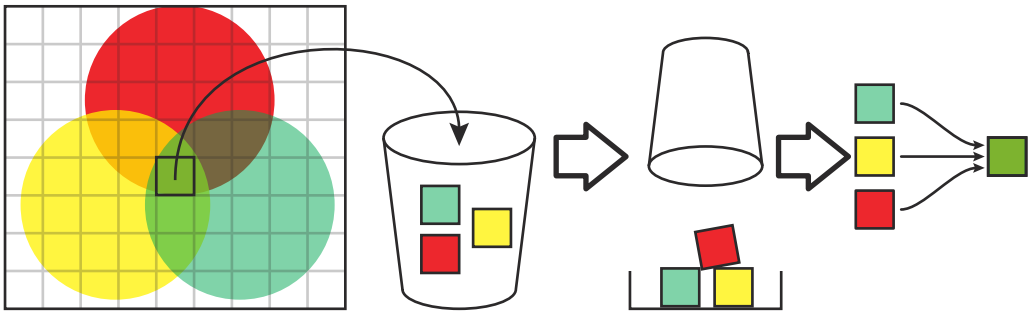


**Figure 2.7.:** Schematic depiction of buffer based fragment sorting. All fragments are stored in a designated data structure capable of holding enough per segment and assigning them to a pixel. Afterwards all fragments are sorted and blended.

This method guarantees pixel perfect and correct renderings with no regard to the number of transparency layers in the image because every fragments pixel is stored. The only limiting factor is the memory size, symbolized by the buckets total capacity. Also, the sorting is time limiting as it is costly when having significantly more layers, but this will be addressed later in Section 2.4.

### 2.1.3. Ray tracing

Ray tracing uses a different approach. Instead of rendering the objects directly by rasterizing them as the approaches described before, also called rasterization based, it uses view rays starting from the image plane hitting the scenes objects. For each view ray, the hit fragments are collected. These fragments are blended from front to back to obtain the correct image as illustrated in Figure 2.8. The ray tracing stops when the rays color is opaque because the used FTB blending increases the $\alpha$-value while blending new fragments, which is the reason for stopping when reaching the green line in Figure 2.8.

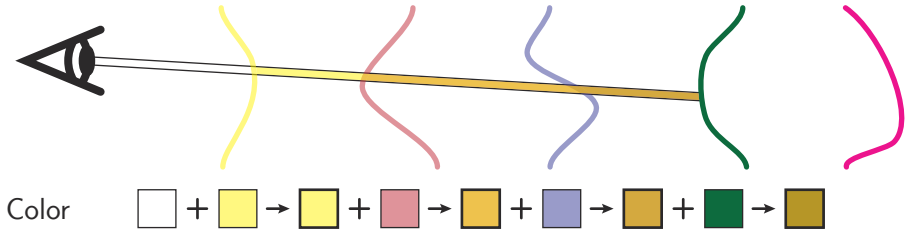



**Figure 2.8.:** Schematic depiction of ray tracing. The color is computed along the viewing ray until it is no more transparent (after hitting green line) starting with a transparent ray. The rays color shows the color while blending like the overview below.

The biggest problem using ray tracing is the collision detection of object and ray. Because of the trajectory data's size, the trajectories cannot be efficiently iterated through and tested whether they do hit the ray or not.

As a conclusion, a more efficient data structure, for example, an Oct-Tree has to be used. This kind of structure divides the whole view space into smaller sections. Each of them contains a list of traversing lines. Now it can be first tested in little time which sections are hit by the ray. Thus only these lines must be tested against the ray, which significantly speeds up ray tracing. [BGZ02]

The ray tracing either ends when a ray finished traversing the data structure or when the colors opacity is saturated. Hence raytracing is also capable of returning an entirely correct image like the buffer based fragment approach.

### 2.1.4. Comparison of the different approaches

The demands put up by the data have to be compared, to select a suited algorithm for implementation into Met.3D.

Met.3D's trajectory actor uses the Lagrangian Trajectories (path-lines). A time step has roughly $3.6 \cdot 10^6$ trajectory segments in total. Each of those segments is extruded to a tube with 8 points of support each side, yielding a total of 16 fragments per tube. Most of the time only a subset of the trajectories is shown, but already a small subset partially has over 100 transparent layers.

Only algorithms which have the ability to return an entirely correct rendering of the scene were considered. Hence weighted sum or average algorithms as presented by McGuire and Bavoil [MB13] are not taken into account. These algorithms avoid sorting by guessing the correct impact of fragments by using their alpha value and depth. Guessing leads mostly to an accurate rendering, but especially when having a high amount of different opacities in a small region, it can result in wrong results. Additionally, the result may change when zooming in or out, which is undesirable.

These algorithms are suitable for usage in video games, where the number of transparent layers is not high, and their outcome is not crucial. Hence they are not considered since the outcome should be a representative and correct rendering of the scene.

For the same reason the Geometry Sorting approach is sorted out, as it also cannot guarantee correct images as already described in detail in Section 2.1.1.

The three remaining approaches are compared by weighing their positive and negative aspects regarding the concrete use case.

**Depth Peeling** On the one hand this approach is intuitive and elegant as no sorting is required. Furthermore blending is done straightforward because the resulting images are directly blended over each other. On the other hand, it only handles a fixed number of transparency layers, which correlates to the number of passes. As a scene can have a more than 100 layers, the over 101 necessary passes are not applicable.

**Buffer Based Fragment Sorting** This approach can handle the varying depth flexible throughout the scene. Furthermore it can use most parts of the already existing pipeline with minor changes. On the downside, it needs enough memory to store all fragments and their depth, regardless of their visibility, because sorting and blending is done as the final step.

**Ray tracing** It is an elegant approach as the transparency is obtained simultaneously while collecting the fragments. Furthermore, it automatically stops when the fragments behind do not affect the color anymore. On the other hand, finding the colliding triangles is costly, especially since the tubes are implicitly generated by the geometry shader. Furthermore, an Oct-Tree or a respective alternative has to be set up before displaying the first fragment, which needs memory and preparation time.

As a conclusion, buffer based fragment sorting and ray tracing will fulfill the demands. Both have the same disadvantage, the memory consumption. However, the data structure needed for ray tracing has to be generated before the rendering starts. The buffer for fragment sorting is generated on the fly, which leads to less pre-computation needed.

Furthermore, most of the existing geometry rendering steps need not to be changed like the tube generation. Using ray tracing would require a rewrite of those components. Thus buffer based fragment sorting is chosen for integration into Met.3D.

## 2.2. Comparison of different OIT variations

As stated in the previous section, the buffer based fragment sorting approach will be integrated into Met.3D. A data structure to temporarily store all the scenes fragments between rasterization and blending is needed.

The foundation for OIT was laid in 1984 by Carpenter [Car84] for rendering movies at Lucasfilm. He describes the idea along with the *A-Buffer* (anti-aliased, area-averaged, accumulation buffer) as a significant improvement over the *Z-Buffer* which was invented ten years before. [Cat74] The A-Buffer is a data-structure fitting exactly the needs of the algorithm. It is a hidden surface mechanism which resolves the visibility pixel based for intersecting or transparent objects.

The data structure must be able so store a tuple of the fragments depth and color per pixel. Additionally, it must be capable of assigning each fragment to an XY coordinate representing the screen pixel. The A-Buffer can be imagined as a data-structure representing the bucket for every pixel in Figure 2.7.

Since there was no GPU more than 30 years ago, the algorithm was initially implemented in plain C on the CPU. Hence a matching GPU implementation has to be found.

### 2.2.1. 3D-Texture Buffer

A 3D-texture is an easy way to assign storage to a 2D index structure. Therefore, a 2D-texture in the screens size is expanded in the third dimension. Each place in the texture holds a

fragment and its depth.

This method is getting problematic when the amount of overlapping transparent layers highly varies through the image because the depth of the texture is globally uniform. Thus, to generate a correct rendering, the depth of the texture must match the maximum of overlapping transparent layers. [Mau+11]

When having more hundred transparent layers, the texture must have a minimum of hundred layers in z-direction. This depth is a massive wastage of GPU memory because most of it will stay empty as the high amounts of transparency mostly will not fill the whole viewport.

### 2.2.2. Linked List

The *Linked List* (LL) data-structured was proposed by Yang et al. in 2010 and is called the first "true implementation of the A-Buffer on the GPU" by the authors. [Yan+10] It features an implementation of a concurrent true linked list on the GPU synchronized by using atomic operations[3].

Creating a LL data structure on the GPU is similar to the CPU implementation of singly linked lists. A head pointer buffer is needed, to assign a pixel to the start of a linked list. This buffer is stored as a 2D texture and contains the pointers to the associated linked lists start node.

Furthermore, every fragment has to be stored as a node of the linked list, which is done in the node buffer. Each node consists of the color RGB$\alpha$-tuple and the depth value. Additionally, a next pointer is saved which links the node to the linked lists next elements. When being the last item, the next pointer is set to a value representing the end of the list (EOL).

Generally speaking, a struct is used to store the fragment as a node in the buffer. Hence the node buffers' internal structure can be chosen freely as long as it is able to appropriately store the fragments information.

In order to practically reserve a space in memory for each incoming node, a global atomic counter is needed. The counters value is interpreted as the next free space in the node buffer. Hence the counter has to be atomically incremented every time a new fragment is added.

When rendering an image using OIT, a linked list storing all fragments for each pixel is needed. Thus the head pointer is the same size as the screen. It is initialized with the value representing EOL. Hence, in the beginning, each head pointer points at the lists end which stands for a blank image. The atomic node counter is initialized with zero, meaning that the first node will be inserted at location zero in the node buffer.

Now all fragments are rendered and instead of directly writing them to the screen buffer, they are stored as nodes. Figure 2.9 depicts this process. It has the following stages for each fragment:

[3] Atomic Operations assure that, even at parallel execution, no run-time conditions occur which may cause data-loss.

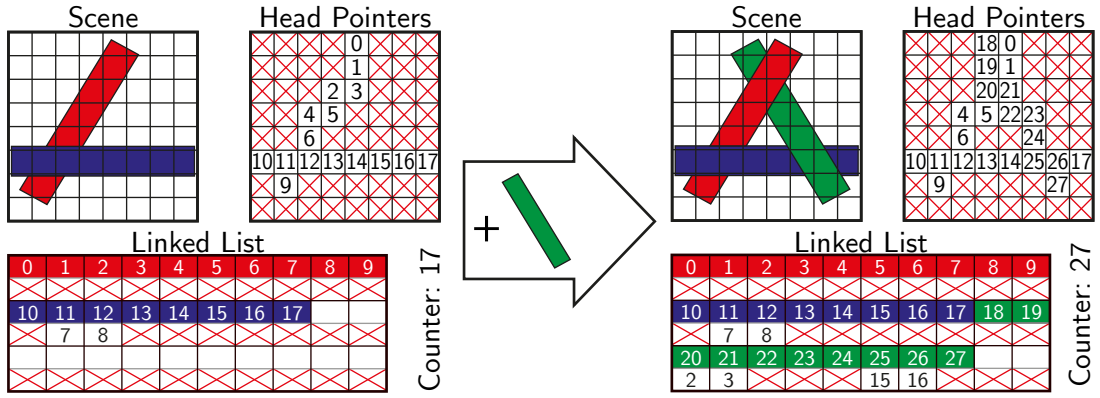



**Figure 2.9.:** Generation of a scene using a linked list. The green rectangle is added to the right scene. Each black number is a pointer and refers to the position in the linked list marked with a white number. Red crosses indicate the EOL flag. The counter holds the number of the last added fragments. For simplicity, all rectangles are rendered after each other beginning from the left top.

1. Retrieve the atomic node counters current value (node address) after incrementing it, representing the allocation of memory
2. Exchange the pixels' current head pointers value atomically with the new nodes address. Now the head pointer points to the new node.
3. Use the retrieved value of the old head pointer as the next pointer and write the finished node element to the buffer at the right location. The next pointer will now either point to the end of the list or the next node.

The node buffer must have enough place to store all fragments which are rendered or adapt its size. Since OpenGL does not support dynamical memory allocation and assuring that all fragments must fit is not achievable in praxis, the maximum number of fragments is limited. Therefore the number of nodes, the buffer is able store is stored and compared against the node address between step 1 and 2. When the address is bigger, the fragment is discarded. When the total number of fragments from the atomic counter is greater than the total places in the buffer, a flag is set indicating that the rendering output cannot be correct because not all fragments were stored.

### 2.2.3. Dynamic Fragment Buffer

The *Dynamic Fragment Buffer* (DFB) is a storage approach developed by Maule. [Mau+13] Knowles independently developed a similar structure under the name *Linearized Layer Fragment Buffer* (LLFB). [Kno15] The nomenclature of Maule is used in this thesis solely to avoid confusion.

Like the LL, the DFB is specially designed for rendering scenes with multiple transparent layers. Its main advantage is the storage compactness of the fragments because it utilizes a pre-pass to optimize the memory layout.

The algorithm needs two different buffers to work properly. The offset buffer is filled during the first of two geometry passes and stores the individual offsets for each fragment. The eponymous Dynamic Fragment Buffer stores all fragment at the pre-calculated offsets in memory. In total the following four stages are cycled through for every rendering:

**Fragment counting** This stage starts the first geometry pass gathering data for the memory-efficient storage. All primitives are rasterized, and the number of fragments per pixel is counted. Therefore a screen size 2D texture is accessed via atomic operations. These assure that the counting result is correct and is not affected by race conditions among the concurrent GPU threads.

**Prefix sum** With the total amount of layers for every pixel known, this stage generates the indexing structure to store all fragments consecutively into the buffer. For each pixel, a base index is defined which points to the address in the buffer where the associated list of fragment starts. For the exact method to calculate the prefixes, it is referred to the original paper by Maule or Knowles. At the end of this pass, the exact amount of needed memory space is known. Hence the dynamic fragment buffer is resized when it is too small to hold all fragments.

**Fragment shading and storing** This stage starts the second geometry pass which finally shades all fragments and stores them in the DFB. Therefore, the per-pixel base pointers from the previous stage are used to address the fragment list. This base-pointer added with the fragment count returns the position to store the fragment in the buffer.

**Sorting and blending** Finally the algorithm sorts and blends the fragments of each pixels list. To find the corresponding list, the base pointer from stage two is used again. In combination with the fragment count, the list is found and processed. Empty lists are interpreted as transparent pixels.

### 2.2.4. Conclusion

As shown in runtime and quality comparisons by Kubisch [Kub14] and Maule [Mau+12; Mau+13] all data structures are suitable to render pixel perfect images. However, the 3D Texture has the biggest restriction as its layer count directly affects the number of maximum transparent layers. Hence only the Dynamic Fragment Buffer and the Linked List are compared.

According to the measurements of Maule, her DFB data-structure will slightly outperform the linked list in some cases, although they mostly perform equally well. Kubisch draws the opposite image, finding an advantage of the Linked List in one scenario.

As a result, I consider both algorithms in respect to pixel perfect rendering and performance as coequal. The DFB has the advantage using less memory because it pre-computes the total

needed size, which leads to its biggest disadvantage, the pre-computation. It requiring two full geometry passes and has to calculate the offsets every step. Because a geometry pass in Met.3D is sophisticated as, for example, the tube primitives generated on the fly, and building the offset list further delays the execution, I chose the Linked List which only needs one geometry pass.

## 2.3. Implementation

This section will explain how the basic OIT integration using Linked Lists into Met.3D is made using at least OpenGL 4.2 according to the author's ideas [Yan+10] combined with the common OpenGL textbook [Shr+13].

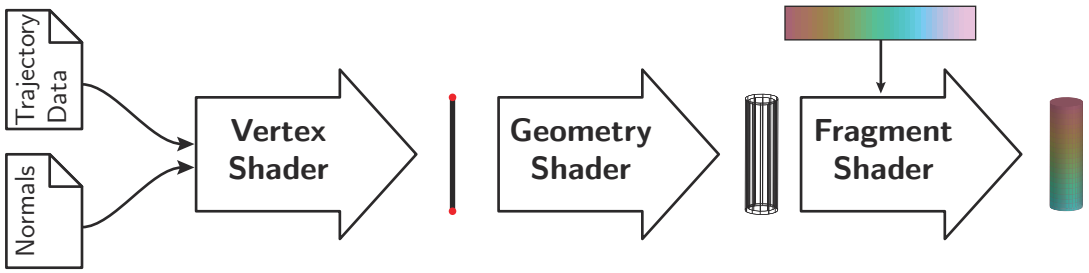



**Figure 2.10.:** Current Met.3D trajectory rendering pipeline

The current Met.3D drawing pipeline is shown in Figure 2.10. The path-lines are uploaded to the GPU, where one shading pass is displaying them. First, the vertex shader passes all elements to the geometry shader. This shader extrudes the lines to 3D-tubes using the method described by Bürger [Bür10] and projects the coordinates from object to screen space. Afterwards, the vertices are passed to the fragment shader for rasterization. Now the color is determined using a transfer function and assigned to the fragment. This fragment is now finished and, after a depth test, passed to the frame buffer for displaying.

My goal is to change this pipeline to an LL pipeline similar to Figure 2.7 which depicts the process for OIT. Therefore the necessary buffers are first integrated into Met.3D.

### 2.3.1. Buffers

In total three buffers are needed for the LL approach.

An atomic counter is used to allocate memory concurrent without a mutex or memory fences. It is realized in OpenGL by using an atomic counter which was introduced in Version 4.2. The existence of this type of buffer type is crucial as it is needed to synchronize the memory access for the Linked List Buffer. Hence the code requires at least OpenGL version of 4.2. Since most of Met.3D's code also requires at least this version, this does not affect the compatibility.

The next key piece in the algorithm is the head buffer. It is essentially for assigning the linked list to the screens pixels. I am using an `uimage2D` in the `r32ui` format to store all the head pointers. This type of texture also features atomic operations since OpenGL version 4.2. Mainly `imageAtomicExchange` is used. This function atomically exchanges the stored value with the value passed by argument. Its return value is the head pointers previous value, which will come in handy when appending nodes to the LL.

Finally, a buffer to store all nodes is needed. Therefore a buffer texture with format `GL_RGB32UI` is used. Therein three unsigned integers are stored, which means each incrementation of the atomic buffer allocates space for three unsigned integers.

These three allocated memory spots are filled with RGB$\alpha$-color (`vec4`), depth (`float`) and the next pointer (`uint`). To fit the data type, color is packed by `packUnorm4x8`, which stores four normalized floating point numbers in a 32-bit unsigned Integer. Respectively the depth is encoded using `floatBitsToUint`. This preserves its bit-level representation and is reversible because it only changes the type.

### 2.3.2. Shaders

In order to use the newly generated buffers, the default trajectory pipeline depicted in Figure 2.10 has to be altered, as this pipeline only uses one shader pass. According to Chapter 2.2.2 a second pass is needed for sorting and blending.

The first pass's shaders nearly remains the same except for the fragment output. Instead of writing the output directly to the default frame buffer, the fragments are stored in the linked list. Therefore a place in memory is reserved by incrementing the atomic counter buffer. The value returned when incrementing the buffer also serves as the pointer to access the linked lists node.

In order to insert the new node, the pointer lying in the head pointer buffer is atomically exchanged with the new pointer. Now the new fragment is the start of the linked list but does not link the fragments behind. The missing link breaks the structure for the moment, which is not problematic because the linked list is never traversed while building it.

To fix the LL structure, the previous head is saved in the node as the next pointer. The whole appending process is illustrated in Figure 2.11.

The second pass has to resolve the linked list for the whole screen. Therefore a full-screen quad is defined on which the head pointer texture is spanned which ensures that every pixels linked list is used.

At first, the LL is traversed, and all fragments are stored in a temporary array. To sort the array, as proposed by the book, bubble sort is used. Afterwards, the fragments are blended using the OpenGL mix function.

Blending the fragments is performed in front to back order, using the FTB blending equations. This has the substantial advantage that the blended fragments can be partially transparent. When using BTF blending, this cannot happen as it has to be started with a solid background.

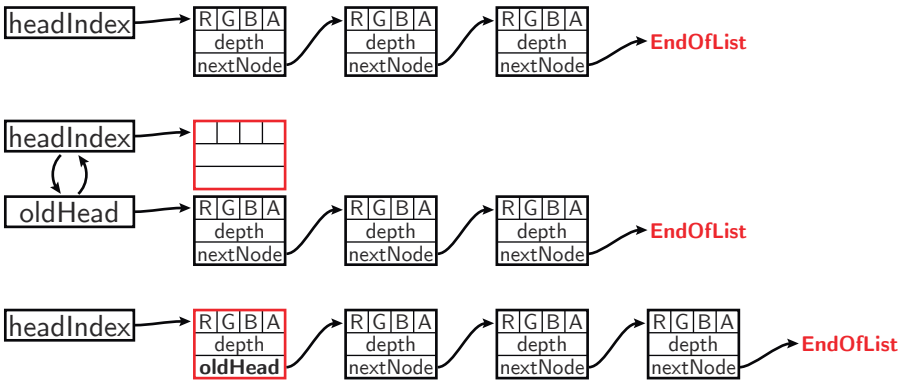



**Figure 2.11.:** Appending a new fragment to the pixels linked list. The node marked in red is newly generated and inserted at the top of the linked list. To obtain the structure, the old head is appended using the new fragments next pointer.

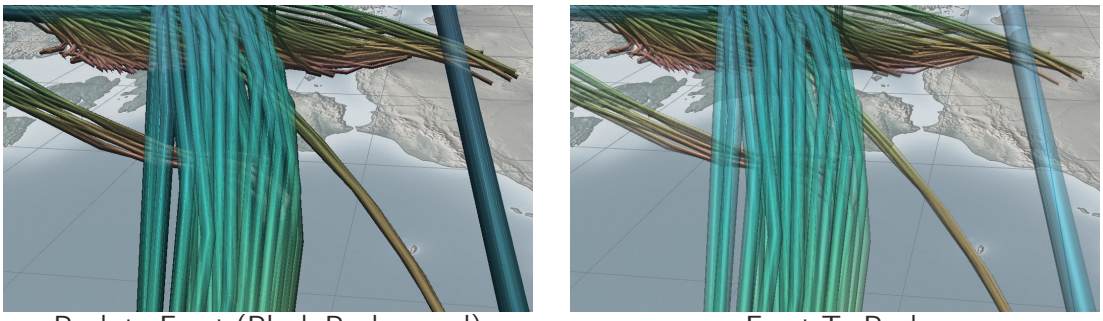



**Figure 2.12.:** Comparison of BTF Blending with a black background and FTB which does not need a background color to start. Hence the resulting blended color be transparent itself, which lets the background shine through.

Afterwards, some of the used buffers have to be set back. The atomic counter is set to zero. Every element of the head buffer has to be restored to contain the EOL flag. Because traversing the buffer for deletion is a costly operation a trick is used. A pixel buffer object is prepared to hold the needed EOL flags in every cell. This buffer is now written on the head pointer texture, which is done by OpenGL's embedded function `glSubImage2D`.

## 2.4. Optimization

In the development of Met.3D, the value was placed on real-time data exploration. However, when using the recently implemented OIT algorithm, the smoothness when exploring the image scene decreased, because OIT reduced the frames per second (fps) significantly.

To prove this lemma, several runtime comparisons between the original Met.3D trajectory actor and the OIT implementation were made using the three demo scenes depicted in Figure 2.13.

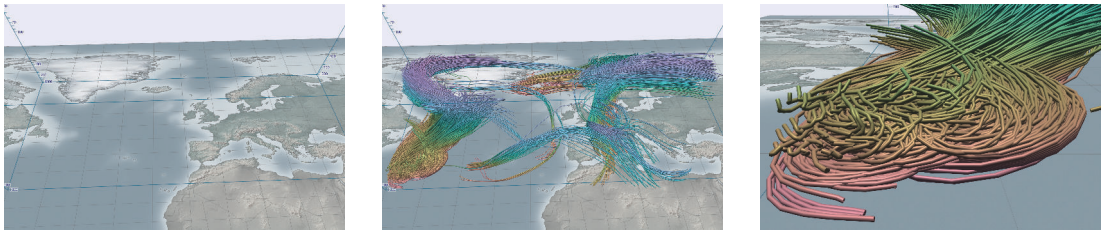



**Figure 2.13.:** Overview of three demo scenes for performance testing

These demo scenes are simulating the most common situations when using Met.3D as explained in detail as following:

**Scene 1 (blank)** This scene mainly serves as a reference for evaluating the base load when rendering. The viewport is the same like in Scene 2, only the line selection threshold is set to 800hpa which results in no trajectories fulfilling this condition. Hence this scene contains zero fragments. Only base map, graticule and the bounding box actor are active.

**Scene 2 (overview)** This scene simulates a user looking at the whole dataset. The viewport is chosen to have all trajectories in the scene. It contains $14,787,048$ fragments.

**Scene 3 (detail)** This scene simulates a user having a closer look at the dataset. Therefore the region with the densest occurrence of trajectories is chosen, because this is the worst case speaking in terms of performance. It contains $47,563,127$ fragments, which is almost four times the amount of the overview scene.

Shadow rendering was disabled for all scenes. The fragment count was measured by reading the value of the atomic counter implemented to allocate new nodes to store the fragments.

The fps and rendering time values were measured by using Met.3D's built-in possibility for profiling the OpenGL performance over 30 seconds. Therefore *Vertical Sync* (VSync) was disabled in the GPU driver to get undistorted values. The time represents the average rendering time per fragment.

All measurements were performed on the same computer using the specified three scenes on a monitor with a total resolution of 1920x1200 pixels. The OpenGL Context has a dimension of 1531x1029 pixels since the Met.3D user interface covers a part of the screen. The used computer is equipped with an i7 6700 processor (4x 3.40GHz), 16GB of DDR3 RAM and a Nvidia GeForce GTX 970 with 4GB GDDR5. It runs on OpenSuse 13.2 with the proprietary Nvidia Driver in version 367.27.

| *Total Time* | **Scene 1 (blank)** | **Scene 2 (overview)** | **Scene 3 (detail)** |
|---|---|---|---|
| **No OIT** | 0.594 ms (1685 fps) | 2.68 ms (374 fps) | 4.57 ms (219 fps) |
| **OIT (Bubble Sort)** | 1.96 ms (509 fps) | 698 ms (1.43 fps) | 1579 ms (0.633 fps) |

**Table 2.3.:** Runtime comparison of the trajectory actor with and without OIT enabled

As seen in Table 2.3, using OIT results in a significant performance drop. This sections goal is not to optimize OIT to be coequal in respect of performance but optimize it to allow the user to explore the data set again interactively. Therefore it is measured how much time each of the necessary OIT steps needs.

Because OpenGL does not provide a possibility to measure the execution time in the shader directly, the step times have to be calculated. Therefore the runtime of a shader executing everything except the wanted task is measured. The difference between the measured runtime and the value in Table 2.3 is a good estimator for the runtime of the step. The complete listing of time measurements which were used to calculate the values in Table 2.4 is given in Table B.1. As runtime for the shading task, the runtime of the no OIT algorithm is used, as it only performs the shading step.

| *Time* | **Scene 1 (blank)** | **Scene 2 (overview)** | **Scene 3 (detail)** |
|---|---|---|---|
| **Shading** | 0.593578 ms | 2.67738 ms | 4.56552 ms |
| **LL construction** | 0 ms | 10.89577 ms | 24.01373 ms |
| **Head index access** | 0.01879 ms | 0.0853 ms | 0.0554 ms |
| **Collecting** | 0.82642 ms | 15.3117 ms | 37.5255 ms |
| **Sorting** | 0.02412 ms | 668.291 ms | 1512.5783 ms |
| **Blending** | 0 ms | 0 ms | 0 ms |
| **Rest** | 0.500952 ms | 0.41285 ms | 0.41285 ms |
| **Total** | 1.96386 ms | 697.674 ms | 1578.95 ms |

**Table 2.4.:** Runtime comparison of the trajectory actor with and without OIT enabled

The values listed in Table 2.4 and visualized in Figure 2.14 show that sorting is the dominating operation. Hence it will be optimized in Section 2.4.2. The time-consume of the

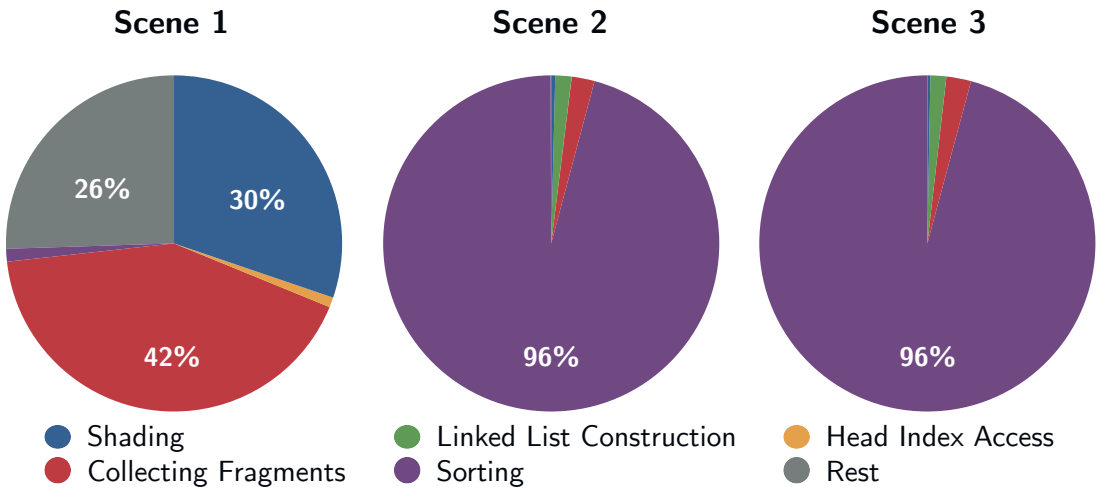



**Figure 2.14.:** Visualization of the needed rendering time in percent per step and scene

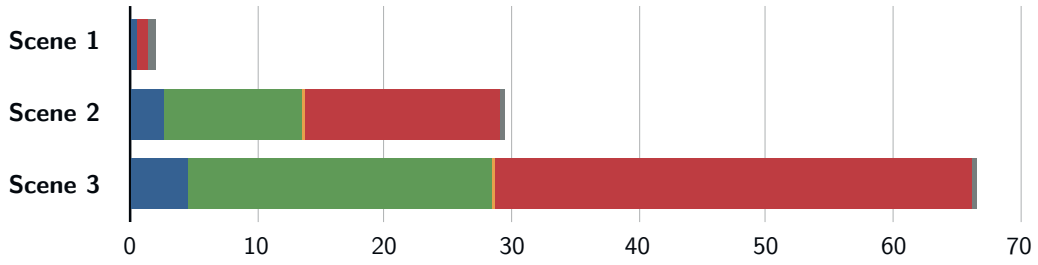



**Figure 2.15.:** Visualization of the needed rendering time in ms per step and scene without sorting

remaining steps is visualized in Figure 2.15. Constructing the LL at the end of the first pass and traversing the list at the beginning of the second pass make up most of this time. The bottleneck for the construction is the atomic counter which is used by all fragments to allocate memory. [Mau+13]

Collecting the fragments is slowed down due to random memory access, which is necessary to traverse all the lists node. [Mau+13] The next longest step is the actual shading process which cannot be avoided as it is required to colorize the fragments.

Binding and resetting the buffers introduced to use OIT mainly consumes the rest time. Hence the time has a nearly constant value and is unlikely to be reduced.

### 2.4.1. Reducing the Number of Fragments

A common way to speed up the rendering process is reducing the amount of fragments which have to be blended. As this thesis goal is to keep the image quality up, fragments cannot be discarded without a reason. However, there are some possibilities to get rid of some fragments, without making compromises in image quality.

#### Backface Culling

The geometry shader generates each tube by rotating the normal in several steps around each trajectories vector [Bür10]. As a reason, almost half of the generated fragments are not directly visible. These fragments lie on the back of each trajectory tube as depicted in Figure 2.16.

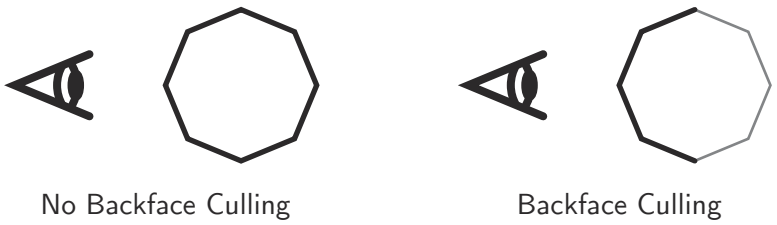
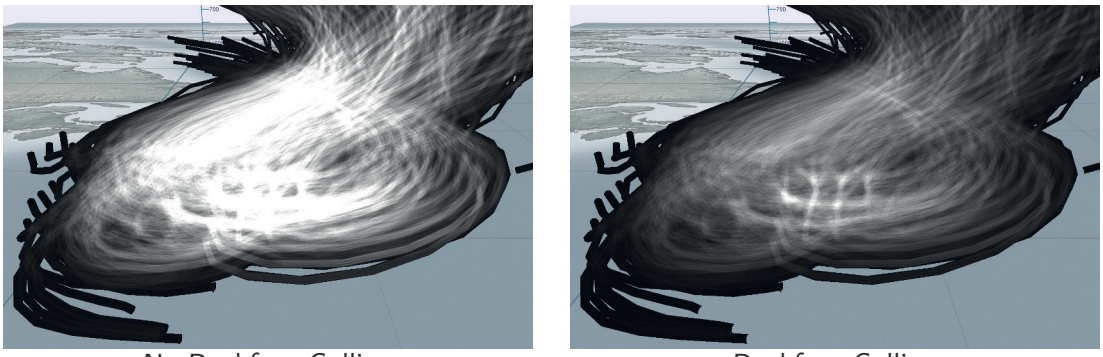



**Figure 2.16.:** Schematic depiction of backface culling. Black fragments are retained, gray fragments are discarded because their back is directed to the viewpoint

When enabling backface culling, OpenGL automatically discards these fragments by checking the direction of the fragment normals against the camera position. When the normal direction points towards the camera, the front face is visible and it will not be discarded. The fragment is discarded when the back face would be visible, which means the normal vector is pointing away from the camera.

The coverage[4] of the detail view scene is shown in Figure 2.17. The right image, where background culling is enabled looks significantly darker, which indicates far less overlapping fragments.

[4]Coverage is the number of overlapping layers in each pixel.



No Backface Culling Backface Culling

**Figure 2.17.:** Amount of overlaying transparent layers without and with backface culling enabled. Transparent sections have 0 layers, black stands for 1 layer and white for at minimum 100 overlapping transparency layers.

The fragment reduction effect is also shown by counting the scenes total number of fragments. This is done by reading the value of the atomic counter buffer after all fragments were stored in the linked list.

| *Fragment Count* | **Scene 1 (blank)** | **Scene 2 (overview)** | **Scene 3 (detail)** |
|---|---|---|---|
| **No Backface Culling** | 0 | 14,787,048 | 47,563,127 |
| **Backface Culling** | 0 | 7,395,949 | 23,830,197 |
| **Reduction** | – | 49.98% | 49.90% |

**Table 2.5.:** Fragment count with and without backface culling

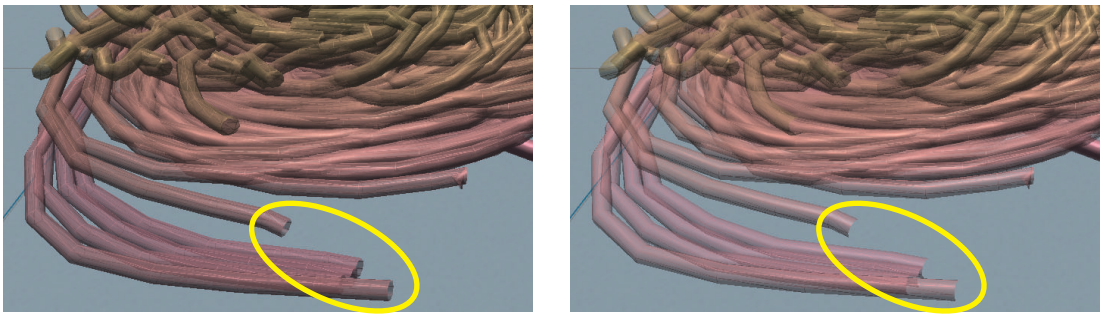
When comparing the amount of total overlaying fragments in the detail scene, it is observable that the numbers of fragments are almost halved, which affects the performance in a positive way.

However, enabling backface culling tests affects the rendered image in two different manners.

**Transparency** The assigned alpha value will influence the transparency more direct as only one layer per tube is visible. Without using culling, each tube has a back and a front side which is blended by the OIT algorithm. Thus each tube is blended twice what significantly decreases the perceived transparency.

**Missing backface** In some rare occasions, mainly when zoomed in, some discarded fragments would be visible. Hence an unclosed tube is seen. Because the tubes are symmetrical, even when the background fragment is missing, no information is lost.

The effects described above are shown in Figure 2.18. A higher perceived transparency throughout the image recognizable. Hence more background structure is seen when looking through the tubes. The discarded fragments, which leave unfinished tubes, are marked in yellow.



No Backface Culling Backface Culling

**Figure 2.18.:** Detail comparison of rendering without and with backface culling, showing unclosed tubes and less perceived opacity.

#### Early Fragment Test

Fragment tests are performed by OpenGL to determine whether a fragment could be visible or not. Hence shadowed fragments are discarded because the cannot affect the rendering outcome.

Typically OpenGL performs fragment visibility tests after executing the fragment shader when the fragment color is already determined. However, since the fragments are stored in the linked list after the execution and processed afterward, this does not discard any fragments when using OIT. Enabling early fragment tests enforces OpenGL to perform the fragment test before running the fragment shader. This ensures that covered fragments are not passed to the LL. [Shr+13]

However, there is a pitfall as enabling early fragment tests also writes to the depth buffer before appending the fragments to the linked list. This step is handy for solely opaque scenes to discard the shadowed fragments. To the contrary in transparent scenes, this discards fragments which are visible, as more than only the frontmost fragment matters. As a consequence writing to the depth buffer must be disabled temporarily.

Furthermore, it positively affects the image quality as the already shadowed fragments should not be displayed to avoid artifacts. This is shown in Figure 2.19. Specifically, the bounding box or the colormap are not visible without using early fragment tests.

### 2.4.2. Comparison of different Sorting Algorithms

Table 2.4 shows that sorting is, with almost 96% of the run time, the most time-consuming operation when performing OIT blending. Since the bubble sort algorithm with a complexity of $O(n^2)$ is used, the sorting performance can be optimized.

Therefore several comparison based sorting algorithms were implemented on the GPU and compared against each other. The selection of algorithms followed the fact, that GLSL, the OpenGL shading language, does not support recursion. Hence quick sort, the CPU standard, is sorted out as it heavily depends on recursion.

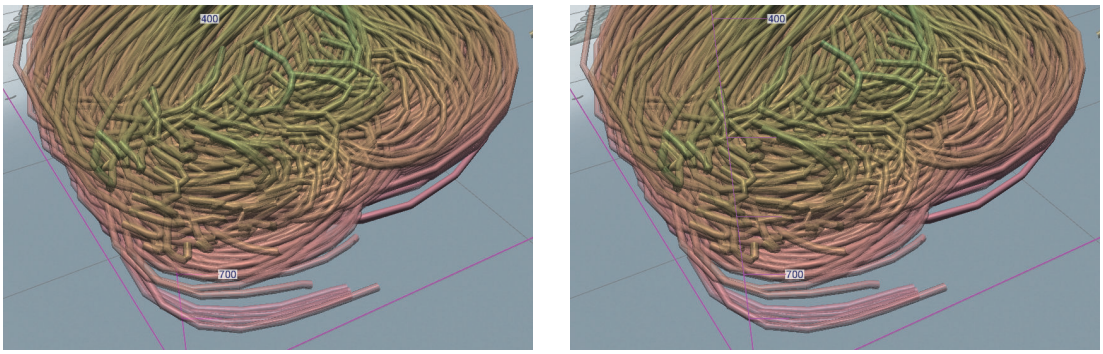



**Figure 2.19.:** Comparison of rendering without and with early fragment tests

The compared sorting algorithms are a selection of comparison based sorting algorithms which all do not depend on recursion. The selection is taken from Algorithms by Sedgewick and Wayne [SW11]:

**Bubble sort** A classic sorting algorithm which only swaps fragments directly next to each other. Not changing distant elements makes it un-performant for larger datasets.

**Insertion sort** An optimized sorting algorithm for a small to medium number of fragments. One fragment after the other is inserted at the correct position by traversing the array.

**Shell sort** An optimized insertion sort variation which does multiple gapped insertion sort passes. The swap operation optimized gap sequence is taken from Ciura [Ciu01]. Using gaps, fragments change their position faster in the first passes which efficiently pre-sorts a list.

**Heapsort** In contradiction to all previously defined sorting algorithms, heapsort defines its data structure upon the array, a heap. Hence it is a non-trivial sorting algorithm which can sort long lists efficiently.

The runtime measurements are performed for the known three scenes with backface culling and early fragment tests enabled.

| *Total time* | **Scene 1 (blank)** | **Scene 2 (overview)** | **Scene 3 (detail)** |
|---|---|---|---|
| **No sort** | 1.97 ms | 16.6 ms | 35.5 ms |
| **Bubble sort** | 1.98 ms | 227 ms | 476 ms |
| **Insertion sort** | 1.95 ms | 152 ms | 333 ms |
| **Shell sort** | 1.95 ms | 73.9 ms | 185 ms |
| **Heapsort** | 1.96 ms | 70.4 ms | 166 ms |

**Table 2.6.:** Runtime comparison with different sorting operations

Scene 1 contains no relevant information when comparing the sorting algorithms, as there are no fragments to sort. Hence this scene is left out for all following run time comparisons based on sorting. However, for now, it is shown that enabling backface culling or early fragment tests has almost no effect on this particular scene.

| *Only sorting* | **Scene 2 (overview)** | **Scene 3 (detail)** | **Average speed up** |
|---|---|---|---|
| **Bubble sort** | 211 ms | 441 ms | – |
| **Insertion sort** | 136 ms | 298 ms | +50% |
| **Shell sort** | 57.3 ms | 150 ms | +215% |
| **Heapsort** | 53.8 ms | 130 ms | +254% |

**Table 2.7.:** Runtime comparison and speed up of different sorting operations (Bubble Sort is reference for speed up)

In contradiction, you see the clear benefit of the fragment reduction when looking at the performance in Scene 2 and 3. Bubble sort performs almost four times better, which indicates its quadratic complexity. Insertion sort performs slightly better than bubble sort, although it has the same complexity.

Gapped insertion sort, called shell sort, is significantly faster than regular insertion sort. The gaps lead to a more rapid pre-sorting of the list because fragments traverse the list quickly. An optimized gap size brings shell sort to an average complexity of $O(n \log(n))$. [Ciu01]

Heapsort is the fastest of the compared algorithms, despite it first builds up a heap data structure, what can be time-consuming. However, for longer lists, the structure leads superior sorting performance because a complexity of $O(n \log(n))$ is assured.

### 2.4.3. Varying the Heap Width

Heapsort is as the name already calls it based on heaps. A heap is a tree with a fulfilled invariant; all direct children must be smaller than the parent. Hence all descendants are smaller than their ancestors. As a result, the root is the biggest node in the heap.

Typically when implementing heapsort, a binary tree is chosen for simplicity reasons. However, since comparing is cheaper than swapping tree nodes, flatter trees are faster on a GPU. As a result, a broader tree like a quaternary tree is faster as it is only half as deep as a binary tree. The heap construction phase is also shorter because 3 out of 4 nodes are leafs and need not sink. So sinking must only be performed on a quarter of all nodes. This leads to slightly faster runtimes than before as shown in Table 2.8. I did the theoretical background for this optimization in previous work [Ban12]. Heapsort is when comparisons are cheap, optimal on a quaternary heap. When using more than four children per node, the number of comparisons is too high to achieve competitive runtimes.

Additionally, with an increased amounts of comparisons, the temporary fragment array is split up in two separate arrays; one for color and one for all depth values. This leads to two benefits:

| *Only sorting* | **Scene 2 (overview)** | **Scene 3 (detail)** | **Average speed up** |
|---|---|---|---|
| **Binary** | 53.8 ms | 130 ms | – |
| **Ternary** | 51.6 ms | 124 ms | +4.8% |
| **Quaternary** | 49.8 ms | 119 ms | +9.0% |
| **Quinternary** | 56.0 ms | 134 ms | −3.1% |

**Table 2.8.:** Runtime comparison and speed up of different tree widths using heapsort (binary heapsort is reference for speed up)

**Code Readability** Since there is a `depthList` array containing all depths, its elements are directly compared. When it comes to changing the element positions, the swap procedure will swap both color and depth in the respective arrays, so it does not affect the readability.

**Cache Coherence** The color values a needed in first part solely when swapping fragments and not for comparing. When both are stored in one list, both are fetched when using the depth due to adjacent memory locations. In the second part only color values and not depths are needed for blending, where this also comes in handy.

This is shown by the measurement presented in Table 2.9. The quaternary heapsort is 3% faster than before, which makes it in total 12.2% faster than the binary heapsort. In conclusion, the optimized heapsort sorts the fragments in 25% of the originally needed time of bubble sort, which makes it perform four times as fast.

| *Only sorting* | **Scene 2 (overview)** | **Scene 3 (detail)** | **Average speed up** |
|---|---|---|---|
| **Binary** | 52.8 ms | 127 ms | +2.0% |
| **Ternary** | 50.5 ms | 122 ms | +6.5% |
| **Quaternary** | 48.0 ms | 116 ms | +12.2% |
| **Quinternary** | 50.2 ms | 121 ms | +7.7% |

**Table 2.9.:** Runtime comparison and speed up of different tree widths using heapsort with two separate Arrays (binary heapsort is reference for speed up)

### 2.4.4. Front To Back Blending using a Priority Queue

Due to its nature, a heap can be used as a priority queue[5]. This fact is an advantage because FTB blending is used. FTB blending sums up the color and opacity values until the alpha value is saturated. Hence every additional fragment blended will not affect the resulting color.

By combining FTB blending with a priority queue, the blending is optimized to only extract as many fragments as needed to obtain a correct result. As a base, the optimized quaternary Heapsort is used.

[5]Heapsort is the typical implementation of a priority queue because you can read the minimum in $O(1)$ and remove it in $O(\log(n))$

Hence in the worst case, the whole array is sorted, which results in the same performance as measured there. As the number of steps to saturate the $\alpha$-value depends on how opaque the fragments are, Scene 2 and 3 are measured with six different opacity levels.

| *Only sorting* | **Scene 2 (overview)** | **Scene 3 (detail)** | **Average speed up** |
|---|---|---|---|
| **Transparent** | 49.0 ms | 118 ms | −2% |
| **6.25% Opacity** | 47.6 ms | 118 ms | −1% |
| **12.5% Opacity** | 37.2 ms | 95.5 ms | +23% |
| **25% Opacity** | 24.3 ms | 59.4 ms | +96% |
| **50% Opacity** | 15.0 ms | 34.1 ms | +135% |
| **75% Opacity** | 11.0 ms | 24.6 ms | +361% |
| **100% Opacity** | 6.54 ms | 14.5 ms | +678% |

**Table 2.10.:** Runtime comparison and speed up (quaternary heapsort with two arrays is reference for speed up)

Table 2.10 shows that the priority queue implementation performs better the more opaque the fragments are because the lists need not traversed as wide as before. When solely having fragments with an opacity of 50%, only a maximum of six fragments has to be blended according to Equation 2.4. This amount increases when lowering the alpha value.

Blending only transparent fragments is slightly slower with a priority queue than without because an extra comparison is made for every fragment to check whether the $\alpha$-value is saturated. The significant speed up makes this fact in practical use negligible. Furthermore the nearly transparent fragments are discarded before adding them to the linked list in the actual implementation. Hence it can be said that the priority queues worst case scenario takes as long as every scenario fully sorting the lost with heapsort.

### 2.4.5. Conclusion

After optimizing the sorting process (Figure 2.20), the time distribution when rendering a fragment has significantly changed in comparison to Figure 2.14. Now the two Linked List related phases (Construction and Collecting) are the dominating phases, needing about 60% of the runtime in Scene 2 and 3.

Figure 2.21 shows how the runtime was optimized during the thesis. The greatest optimization is the fragment discarding process explained in Section 2.4.1. It reduced the run-time by more than a third. A similar speed up was achieved by using quaternary heapsort instead of bubble sort. The total run time was decreased by factor 22.3. Hence it performs more than twenty times faster and runs in most cases with at least 10 fps, which is sufficient for interactive data exploration.

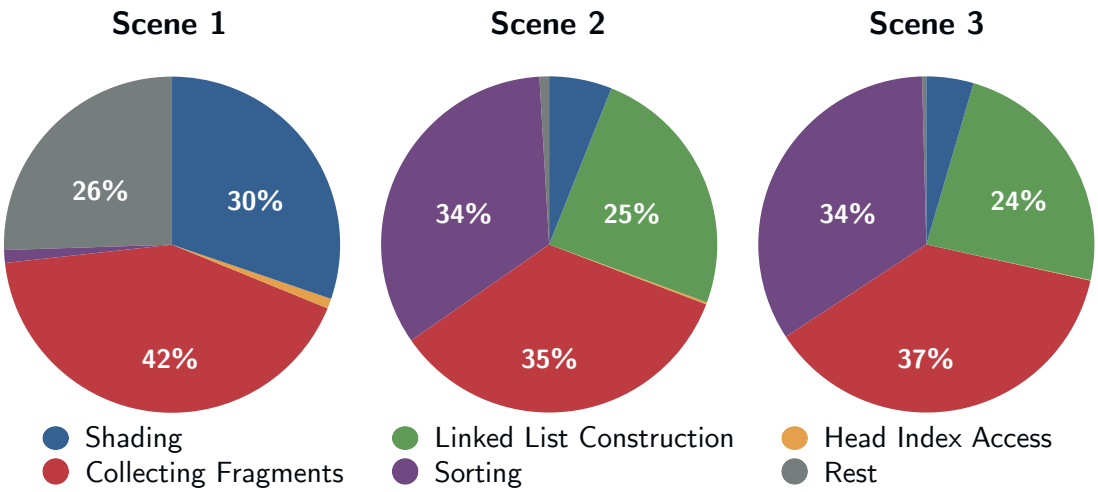



**Figure 2.20.:** Visualization of the optimized rendering time in percent per step and scene

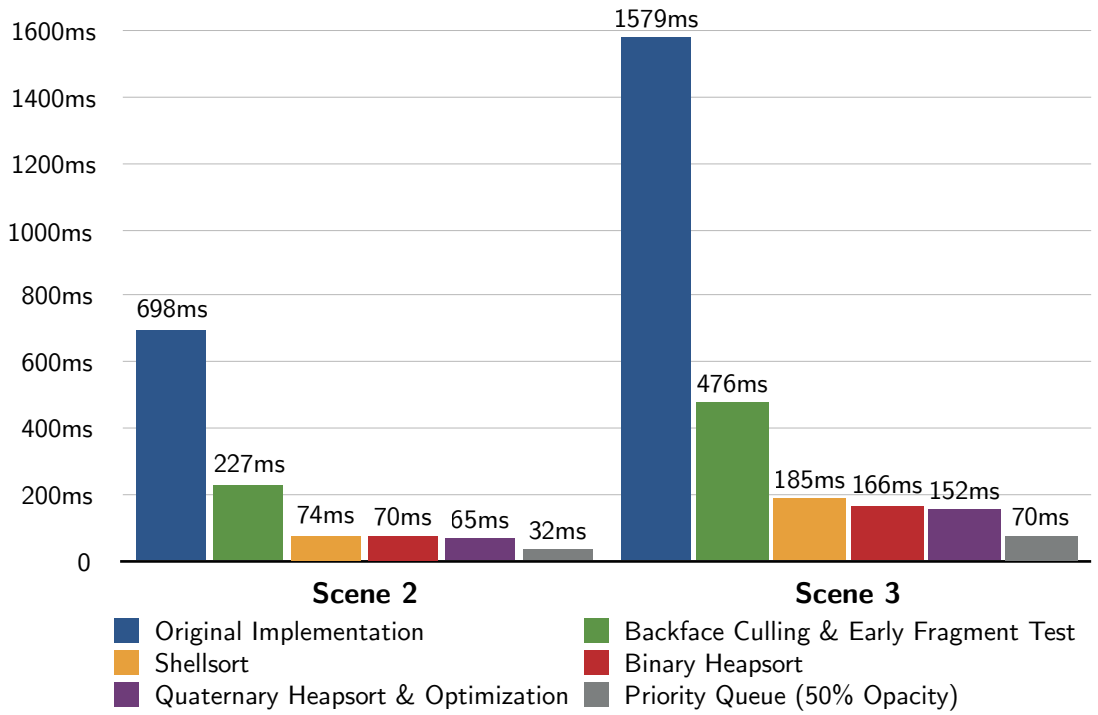



**Figure 2.21.:** Visualization of the rendering time optimization steps per scene

# 3. Opacity Mapping

In this chapter, I discuss how to implement mappings from various trajectory parameters to a relevance criterium, which will be used for determining the alpha component. This criterium allows the user of Met.3D to weigh each parameter and highlight the most important structures.

## 3.1. Non-Viewpoint Dependent Parameters

Non-viewpoint depending parameters are easier to handle. They stay constant as long as the scene is not changed. All of these parameters are not directly contained in the data as attributes, as they are derived data. It will be described how they are calculated and what they can be used for.

### 3.1.1. Total Trajectory Pressure Difference

The trajectories are, as explained in Section 1.2, computed numerically from an existing dataset. They originally only contain the XYZ-coordinates of each point and are uploaded to an array on the GPU consisting of three-dimensional vectors to hold the spatial data.

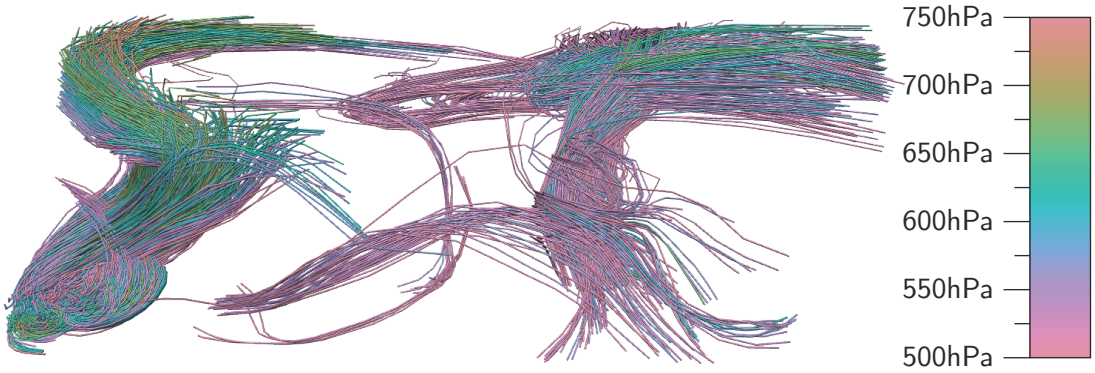



**Figure 3.1.:** Total trajectory pressure difference over 48 hours (Line Selection > 500hPa)

The trajectory object in the GPUs memory is expanded by a fourth vector component, which contains the total trajectory pressure difference, or in other words how much the trajectory rises or falls over the given time span; 48 hours are pre-set. It is not possible to calculate this value on the fly in the geometry shader because all the trajectories vertices are needed therefore and the shader only receives four vertices at a time.

This parameter is an expansion to the already implemented line selection criterium which was solely usable as a hard filter criterium. Now the parameter is available in the shader for each vertex. Because the value is assigned to whole trajectories, it is constant over the entire trajectory. As a consequence, only entire trajectories can be matched with this criterium.

This factor can be used to find WCBs. A common criterium therefore is an ascent of over 600 hPa in 48 hours [Rau+15a].

### 3.1.2. Trajectory Pressure Difference per Segment

This parameter enhances the filtering possibilities on the pressure difference by defining a filter criterium on a smaller domain. Using the geometry shader, the pressure difference per segment is calculated, which is done by finding the difference between the start and the end pressure of the given segment.

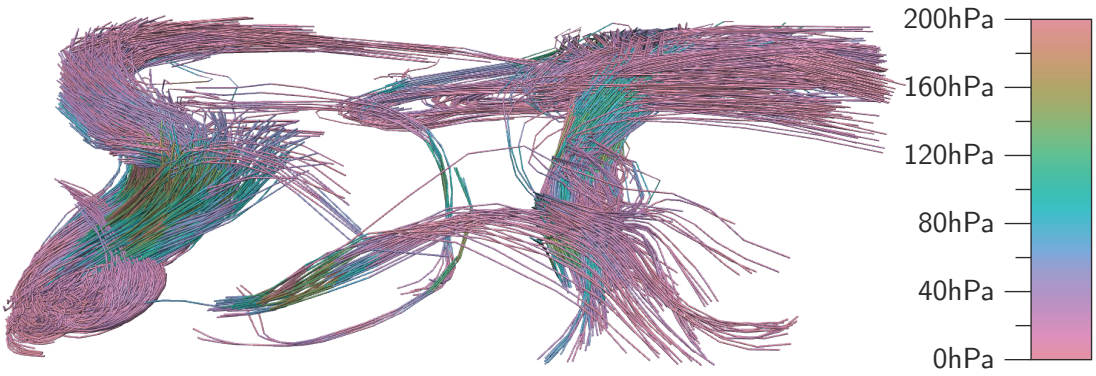



**Figure 3.2.:** Trajectory pressure difference per segment (Line Selection > 500hPa)

Since the difference is calculated per segment, the returned values are more fine-graded than when using the total difference, although they are not interpolated. This leads to one value per trajectory segment as clearly seen in Figure 3.2 when looking at the borders between the segments. Looking at this scene, it is highlighted where trajectories have the phase of their highest ascent when having, which could be a signal for a weather front or a clear air turbulence (CAT). [EUM14]

### 3.1.3. Curvature

The curvature is defined as the amount a line deviates from being a straight line. As the trajectories consist of several straight lines which are joined at an angle, the curvature is calculated as the size of every angle between the segments.

This parameter is computed on the fly in the geometry shader using the mathematical interpretation of the scalar product. Therefore gradients for both ingoing and outgoing line segments of the join point are needed. These are obtained by a normalized vector representing the gradient of both adjacent segments. The angle $\varphi$ between both is calculated using the

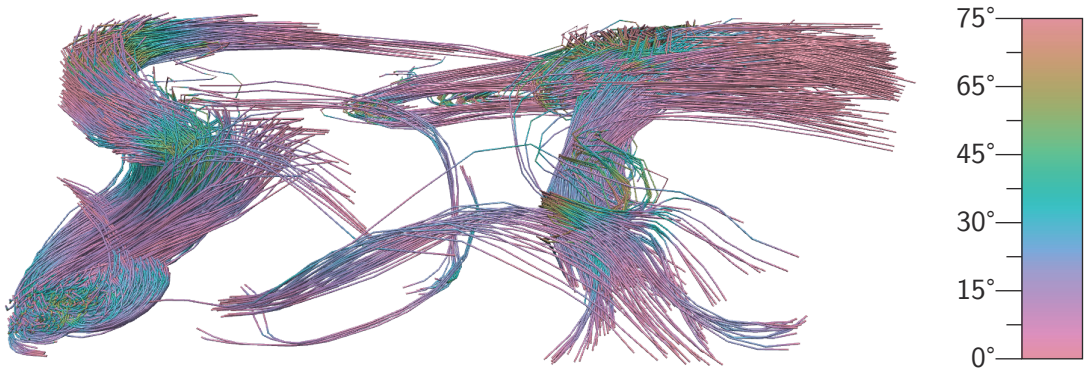



**Figure 3.3.:** Curvature (Line Selection > 500hPa)

scalar product of the two vectors, which are directly interpreted as $\cos(\varphi)$ because the vectors were normalized before.

Since the cosine does not change linear over the angle, the value is converted to degrees where 0 equals 0° and 1 equals ±180°. The curvature at the endpoint is defined as 0° which leads to a fade out of the factor at the end of each path-line.

As described the curvature is calculated and assigned per vertex, which is why the values are linearly interpolated over each segment. This leads to a smooth transition over the trajectory.

This factor can be used to highlight tightly curved structures inside the data. Hence it can be used to find areas with high vorticity like for example tornados in combination with a steep ascent. The curvature was also used as relevance criterium by Mathias Kanzler for Figure 1.6 when introducing the approaches.

### 3.1.4. Length

This parameter also uses the length of the trajectories segments gradient. This factor is a generalization of the segment-wise pressure difference because it takes all three spatial dimensions into account, not only the pressure.

Because all trajectory support points have the same timespan in between, the faster the particle moves, the longer the respective segment is. As a result, the segments length directly correlates with its velocity. When having a look at Figure 3.4, it can clearly be seen where the jet stream is located in the upper troposphere [Pic84]. Furthermore, some similarities with the segment ascending are visible, because as stated before both take the length into account.

### 3.1.5. Angle of Ascent

A metric for finding steep rising particle is gathered by looking at the angle of the trajectory concerning the ground. The higher this angle is, the steeper the particle will rise. This parameter returns similar results as the other parameters used to highlight the segments

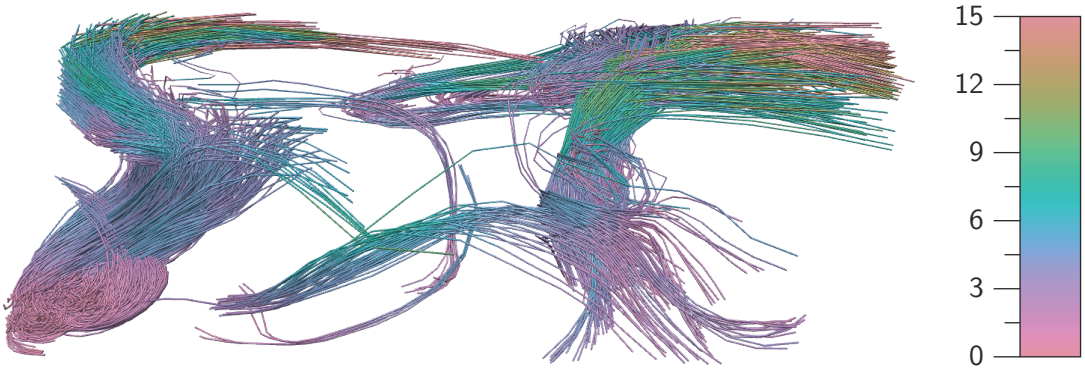



**Figure 3.4.:** Length/Velocity (Line Selection > 500hPa)

ascent but is still different because it is independent of the segments length. Hence this parameter also will highlight particles which have a steep ascent but are not fast.

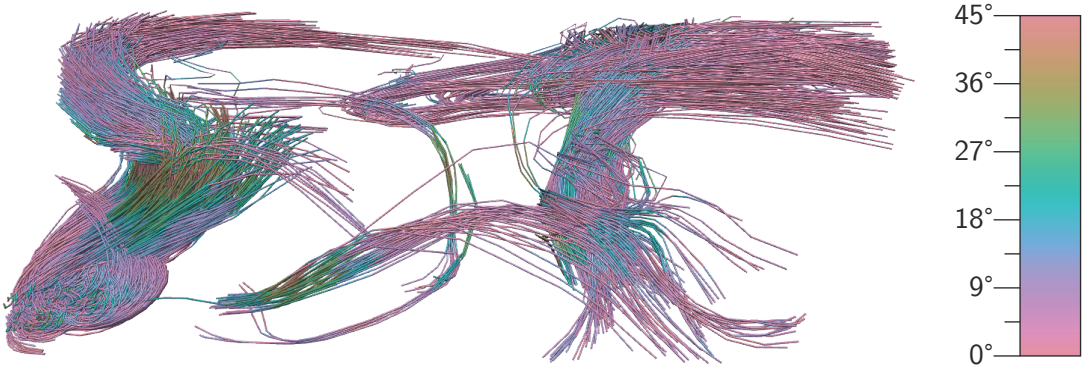



**Figure 3.5.:** Angle of ascent (Line Selection > 500hPa)

To calculate this parameter, the scalar product is used in the same manner as when calculating the gradient. The angle is determined between the gradient vector and the x- and y-components of the same vector which leads to results from 0° to 90°. When having particles rising with an angle of nearly 90° this can be a signal for thunderclouds. [Pic84].

### 3.1.6. Trajectory height difference per segment

This parameter is, as the name already states it, is similar to the pressure difference per segment, but the height is used to calculate the difference. Height and pressure cannot be linearly converted because the pressure changes faster, the nearer the particle is to the ground. It is a consequence of the greater stack of air, lying on the particles, in lower sections. Because this results in a higher pressure gradient in the lower regions, it leads to higher pressure differences in lower than higher levels. [LC79]

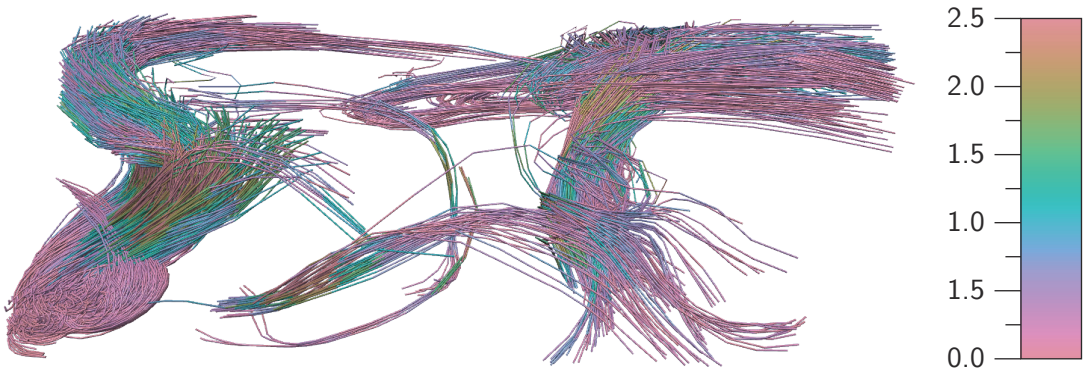



**Figure 3.6.:** Trajectory height difference per segment (Line Selection > 500hPa)

Hence when looking at Figure 3.2 there is almost no pressure difference in the higher levels visible. When looking at the same section in Figure 3.6 there is a difference in height shown.

## 3.2. Viewpoint Dependent Parameters

The parameters introduced until now were all not viewpoint dependent, which makes them constant while rotating or zooming the scene. Since various viewpoints result in different amounts of occluded layers, the set of parameters must be expanded to use them for avoiding occlusion.

As stated in the introduction in Section 1.3, both Günther and Kanzler presented an approach to avoid occlusion.

According to the argumentation in Chapter 1.3.3 a opacity optimization approach as presented by Günther [GRT13] is not applicable as it needs to much computation power. However, the line hierarchy approach by Kanzler [Kan16] features some viewpoint dependent parameters, which are directly computable without solving a linear system of equations.

Although the whole approach by Kanzler is implemented, the parameters are still usable and handy and introduced into Met.3D. To be more specific only the right part highlighted in red of Figure 1.7 is implemented which features four viewpoint dependent parameters.

### 3.2.1. Coverage

The coverage criterium is technically a benefit of the linked list approach since it is automatically retrieved when resolving the LL as seen in Section A.1. When iterating through the list of gathered fragments, a variable counting the number of fragments is summed up. Once all fragments have been stored in their respective array, the variable holds the total count of segments.

This value is the coverage of the area. The higher it gets, the more layers are rendered over each other and in conclusion, this also leads to a higher possible occlusion.


**Figure 3.7.:** Coverage (Line Selection > 500hPa)

### 3.2.2. Maximum Pixel Importance

These parameters are useful to emphasize the visual contribution of the most important segment in the list. Therefore an additional buffer is introduced to the linked list approach. This buffer has, like the head buffer, the format `R32UI`. It is filled throughout the geometry pass in the linked list construction phase along the other buffers. It stores two parameters, the maximum importance as well as the depth of the fragment with the maximum importance.


**Figure 3.8.:** Maximum Importance (Line Selection > 500hPa)

To store the value without losing updates or using critical sections, atomic operations are used, namely `imageAtomicMax`. This operation only overwrites the current textures value, when the new value is greater. Using this, the maximum importance can be determined because it is the value which the buffer holds in the end.

Since it is crucial also to get the depth of this fragment, both parameters are stored combined. Therefore `packUNorm2x16(vec2(depth, importance))` is used which packs the two unsigned normalized floats into one unsigned integer. The sorting-relevant maximum importance is stored in the most significant bits position. This trick ensures that when changing the value, both the importance value and the depth are written. Furthermore, the importance only is

considered for finding the maximum, as it is stored in the most significant bits.

On the downside, packing two 32-bit precision floating point numbers into an unsigned value of the same size decreases the accuracy and the domain. Because both are normalized floats, the range is not changed; only the precision is decreased. This loss of significance is not grave, as it is solely used for comparison and 16-bit precision ($\pm 1.5 \cdot 10^{-5}$) is accurate enough.

To avoid hard borders, which occur when the depth greatly varies, the resulting outcome is blurred. This is accomplished by a 5x5 filter which, sums up all 25 pixels and takes the average as the result. It leads to a smooth transition as depicted for comparison in Figure 3.9. As a result the now visible back features are surrounded by a narrow white border which is caused by the smoothed depth values.

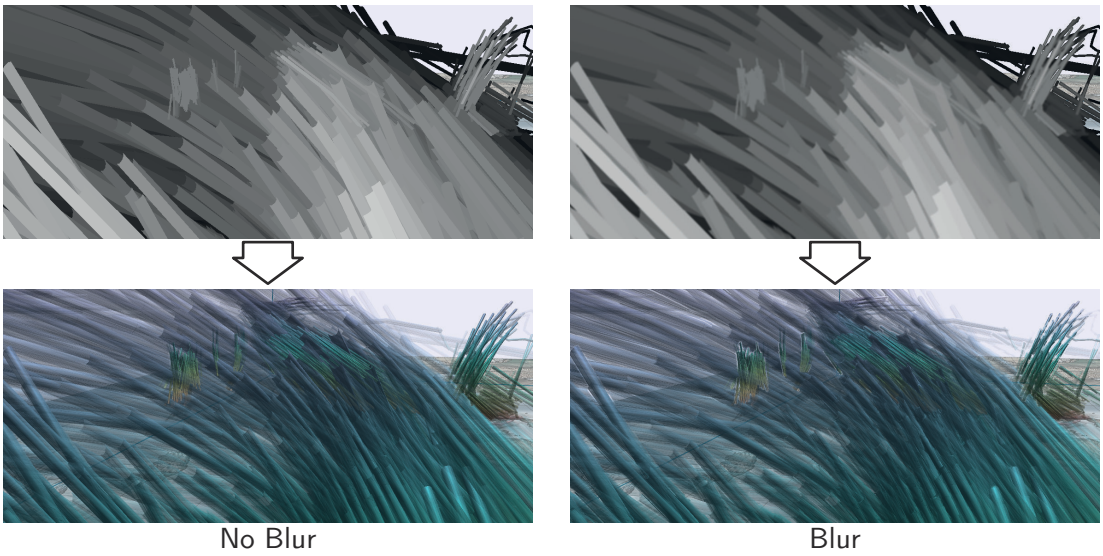



**Figure 3.9.:** Impact of blurring the maximum importance depth. The maximum importance and the resulting image using the penalty factor introduced Section 3.3.2 is shown.

### 3.2.3. Directional Variance

The directional variance introduces a way to analyze the image scene whether the pixels fragments mainly share their direction or do not. To get the direction of the trajectory segments, the linked list structure is expanded by a field to contain the gradient of the rendered line segment. This normalized gradient is calculated, as most of the other geometric values, in the geometry shader.

To store this gradient in the linked list, the vector is packed. Because the three vector components range from -1 to 1, `packSNorm4x8` is used which packs signed normals. This value is gathered from the linked list when unpacking it like the other components. Directly after

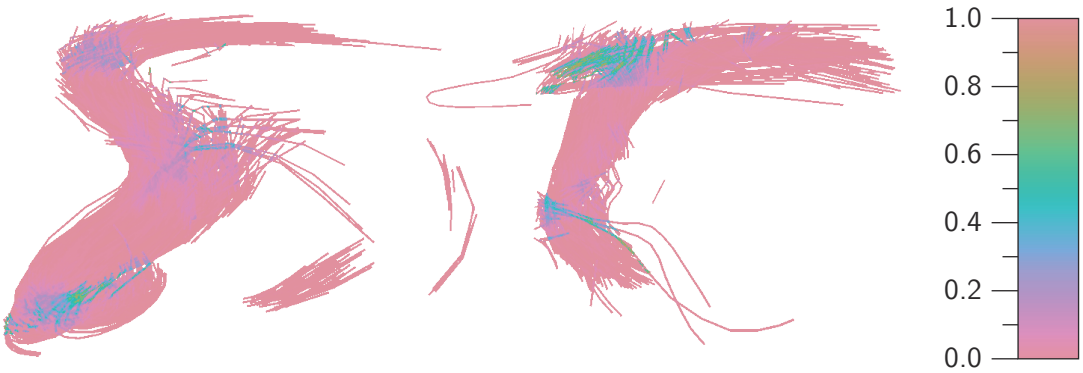



**Figure 3.10.:** Directional variance (Line Selection > 500hPa)

unpacking the values they are summed up to perform the calculation from Equation 3.1.

$$\text{variance} = 1 - \frac{1}{n} \cdot \| \sum_i \vec{g}_i \| \tag{3.1}$$

The value indicates the variance with zero meaning there is no variance at all and one standing for entirely different directions of the summed up vectors.

## 3.3. Parameter Mapping

Using these ten parameters, the focus on the displayed data set can individually be set. As a base, the already built in line selection filter works by only letting lines pass having a pressure difference of at least the given value. This reduces the amount of lines in the dataset which are considered for being rendered and as a result increases the performance.

The following parameters are divided into two sections, the parameters which are directly considered for the relevance criterium, called importance, and the parameters used to resolve the overlaying layers. All non-viewpoint parameters are utilized for the importance criterium, which should remain the same while changing the viewpoint. The viewpoint dependent parameters are used for avoiding occlusion. In particular, avoiding shadowing of relevant trajectory parts.

### 3.3.1. Importance

Each of the non-viewpoint parameters has a predefined range. The user should be able to control the relevance of specific values in this range to map the parameter in a range from zero to one. Therefore each parameters control is similar as illustrated in Figure 3.11. The user can assign values for each parameter through the Met.3D user interface to define an importance value, which is clipped to the range from 0 to 1.

**Minimum** This value specifies where the importance value of 0 is set.

**Maximum** This value specifies where the importance value of 1 is set.


**Figure 3.11.:** Parameter to importance mapping using minimum and maximum

When the minimum is bigger than the maximum, the gradients direction will be switched, making minor values more important than bigger values.

All derived importance factors are multiplied by an user-controllable weighting factor, which ranges from -1 to 1. The resulting sum is divided by the sum of the weighting factors to have a total importance value between 0 and 1.

When having a negative sum of importance weights, the calculated value is interpreted as a penalty and subtracted from one to blend out the fragments which match the requirements. This is useful when the focus should be set on the opposite selection of parameters.

An exponent to the importance is introduced to alter the gradient of the importance factor. It will decrease the total opacity and increase the weighting on the higher importance values as shown in Figure 3.12. Hence it leads to a reduced number of medium transparent fragments, which is useful for focussing on the most relevant parts.

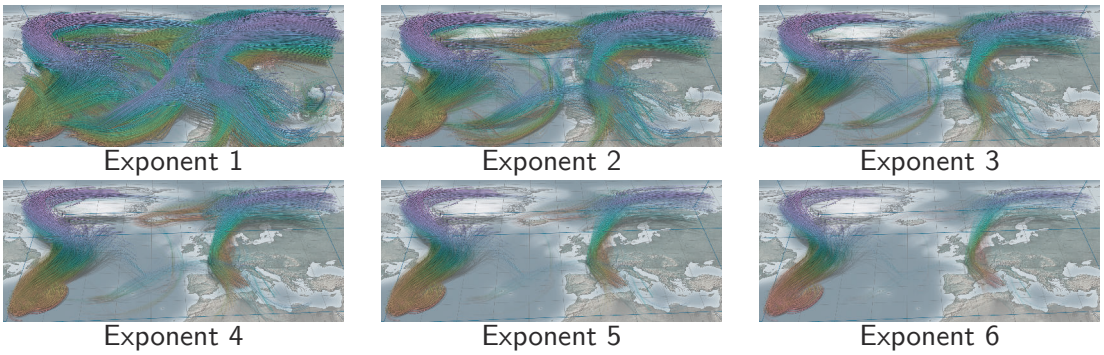



**Figure 3.12.:** Comparison of different exponents for the importance factor (solely trajectory ascent with min 300 hPa and max 800 hPa is mapped)

The calculated importance value is directly mapped onto the colors alpha components. As a reason the user can compose the alpha values directly as he likes, which leads to a direct connection between weighting and outcome. However, some important features in the middle of the domain may be shadowed by less relevant lines, lying in front. This problem is not solvable using solely non-viewpoint dependent parameters because they are not adaptive to the viewpoint.

### 3.3.2. Occlusion Avoidance

Through the direct mapping, there is a direct connection between importance and transparency, which must be loosened to avoid shadowing some important features. Therefore, the viewpoint dependent parameters are used.

These permit the system to perform different estimations for occlusion and thus find fragments which may be in the way. There are two different variations of occlusion avoidance implemented.

#### Penalty Factor

This approach introduces a factor to penalize the fragments which disturb the view at the most relevant fragments. It is probed whether the fragment lies between the most important fragment and the camera, by comparing its depth with the depth of the maximum important fragment. If it fulfills the condition, the alpha value is multiplied with a penalty factor.

The Penalty factor is user-controllable and ranging from zero to one. It is preset to 0.25, which reduces these fragments opacity by 75%. Hence the user can also decide to discard all overlaying fragments by setting it to zero.

The resulting image is, as shown in Figure 3.13, generally more transparent because the transparency of many fragments is reduced. The important structures in the back of the image which are visible now are marked with the yellow circle.

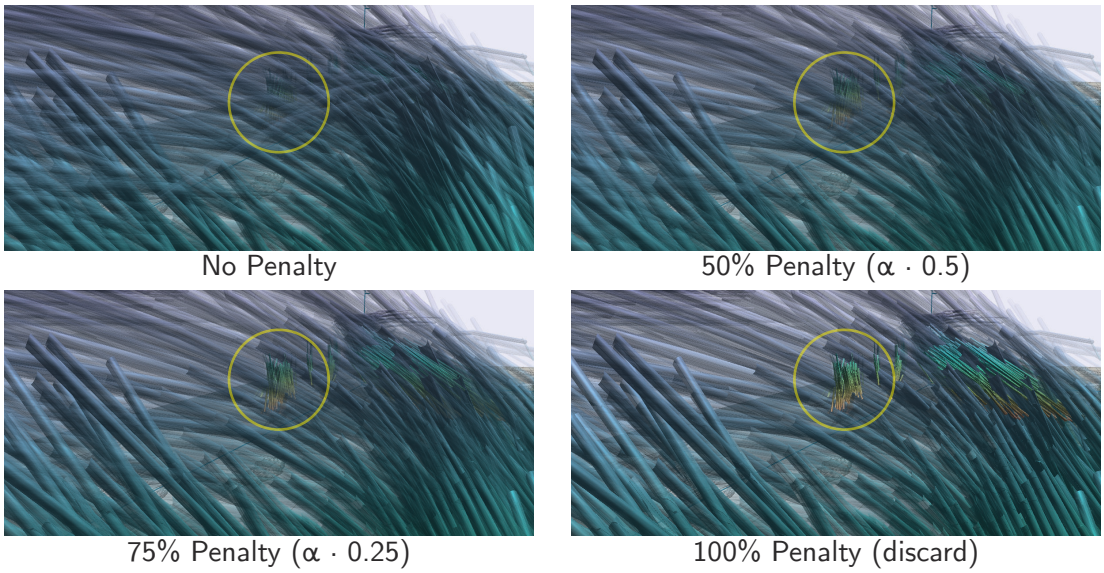



**Figure 3.13.:** Comparison of different penalty factors.

### Weighting Factor

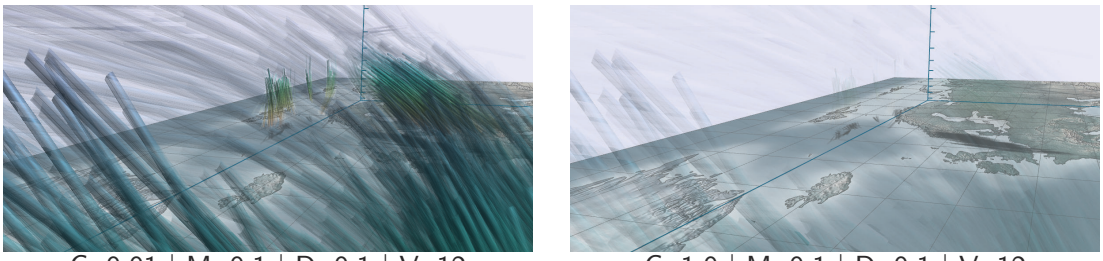
A more elaborate version is presented by Mathias Kanzler in his paper [Kan16] along with a line hierarchy. His approach combines all four parameters from Section 3.2 into one factor $\rho_i$ using Equation 3.2. The factor is calculated for each pixel $i$ in the viewing plane using the parameters at the same position $s_i$. The term $\text{less}(i, D(s_i))$ stands for a depth comparison and is set to one if the fragment lies between the camera and the most important one. Otherwise it is set to zero. The other terms refer directly to the pixels parameters: coverage (C), maximum importance (M), and directional variance (V). The factor $\lambda$ controls the impact of $P_i$.

$$\rho_i = \frac{1}{1 + (1 - g_i^{\lambda}) \cdot P_i} \tag{3.2}$$

$$P_i = m \cdot M(s_i) + c \cdot C(s_i) + v \cdot V(s_i) + d \cdot \text{less}(i, D(s_i)) \tag{3.3}$$

This factor is computed for every fragment and then, unlike to the original approach, multiplied with its opacity. This differentiation must be made, since Met.3D does not incorporate a line hierarchy along the trajectories. The different user-controllable parameters react sensitive to minor changes, especially the coverage weight because the scene contains a high number of layers.

Using this approach, the images shown in Figure 3.14 are rendered. The right image shows the outcome when using too high weights.



C: 0.01 | M: 0.1 | D: 0.1 | V: 12 C: 1.0 | M: 0.1 | D: 0.1 | V: 12

**Figure 3.14.:** Occlusion avoidance using the weighted factors by Kanzler

## 3.4. Met.3D Integration

All the previously described parameters must be accessible and be easily configurable by the user. Therefore the trajectory part of the Met.3D user interface is cleaned up, bundling the similar settings into expandable groups.

There are now several different groups containing all settings for controlling color, opacity, occlusion avoidance, and the shadow parameters. The following section explains how each of the parameters can be used.

### 3.4.1. Color

At the begin of this thesis, the color was pre-set to the pressure elevation. Now the user can change it to the importance parameter or every of the factors presented. This allows the user to set the focus on different aspects or preview his opacity filter criterium. Furthermore, the user can configure a transparent color mapping using the built-in transfer function actor.

### 3.4.2. Line Selection

All the previously implemented parameters are packed into this group because they are performing a line selection operation. This includes a switch to enable or disable the line selection and the threshold which is taken into account for discarding the trajectories.

### 3.4.3. Importance

The importance section of the side panel controls all the parameters introduced in Section 3.1 in the manner described in Section 3.3.1. Hence every parameter has its subgroup which contains all three necessary factors. So there are six subgroups containing minimum, maximum, and weight.

These parameters are implemented as decorated text-fields with a predefined range matching the parameter, for example, $0^\circ$ to $180^\circ$ for the angles. These fields are stored in vectors and passed to the GPU where the importance criterium calculation takes place.

Additionally, the importance exponent can be set which controls the factor user for exponentiating the importance value.

### 3.4.4. Occlusion Avoidance

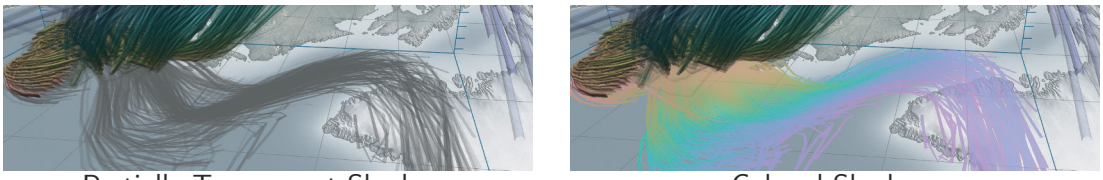
Occlusion avoidance can be controlled in several modes through the user interface. When having the simple mode activated, called penalty factor, only the corresponding factors input field is used. The user can control the penalty for overlaying fragments. The user can further control whether the maximum importance and depth should be blurred to enforce slower transitions.

When switching to weighting factor, the five parameters therefore get active. The weight of all four parameters is controllable along with the $\lambda$-factor. All weights contribute to the value of $P_i$ computed through Equation 3.3. The $\lambda$-factor controls how much these parameters contribute to the combined $\rho_i$.

An additional factor, called *global opacity* is introduced which controls the maximum possible opacity by multiplying the resulting transparency with this factor between zero and one. Furthermore the whole occlusion control can be switched off.

### 3.4.5. Shadow

The shadow generation of the trajectory actor was rewritten during this thesis and is integrated with more controllable features. Previously the user was allowed to enable or disable the shadow and switch between colored and uncolored shadows, which is also possible. The user is furthermore allowed to choose the shadow color and opacity resulting in a different visibility of the shadow.



Partially Transparent Shadows Colored Shadows

**Figure 3.15.:** Comparison of partially transparent and colored shadows

Additionally, a possibility is added to change the opacity of the already rendered shadow. This lets the base-map shine through the colored shadow as shown in Figure 3.15. Hence it allows a better spatial orientation when viewing cluttered scenes.

# 4. Discussion

This section discusses improvements gained by the integration of opacity-based techniques in Met.3D. As stated before, the user was previously only able to use line selection as shown in Figure 1.4.

## 4.1. Meteorological Benefit

Met.3D's limited possibilities were expanded by using opacity. The changes can be seen when comparing line selection with the more elaborate relevance factor. Using this factor, a meteorologist can change which of the selected fragment is more important than another and highlight it in the rendering.

### 4.1.1. Context Information

Opacity-based rendering introduces an additional channel along the color to control. It is used to grade the relevance of the shown particles, as shown in Figure 1.2 and Figure 3.12 for rising particles.

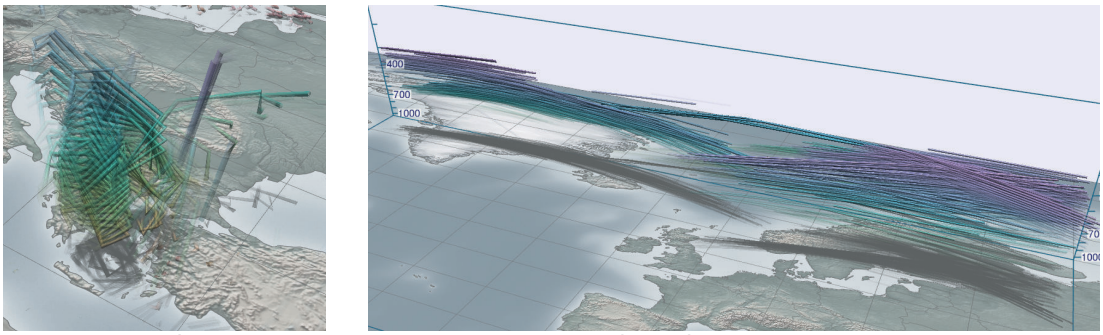
This enables the user to provide more context information throughout the scene by adding near transparent path-lines. These trajectories are useful for having a look at overall picture without loosing the important features out of sight.

It increases the possible amount of information in a scene without having too much clutter in the background. The amount is further increased by the partially transparent shadow, which also adapts its opacity according to the particles above. Hence the areas with the densest trajectory fill are visible in Figure 3.15 which can be used to perceive the particle trace in respect to their ground position.

### 4.1.2. Increased Versatility

Previously filtering was only possible by the total pressure difference. However, with the introduction of the controllable relevance factor, other structures than rising particles can be highlighted as shown in Figure 4.1.

**Figure a** Ascending air forming a vortex over the Aegean Sea is shown using the curvature and segment pressure difference parameters. The vortex consists of warm moist air ascending from the sea. [LC79]




**a:** ascending air **b:** polar jet stream

**Figure 4.1.:** Rendering improvements through new filter parameters

**Figure b** The polar jet stream is shown using the particle velocity. It meanders in upper stratospheric level with a speed of up to 160 kph. [Pic84]

In conclusion, individual weighting of transparency makes Met.3D more versatile by allowing the user more freely to set his preferences. This can also be used to focus on more than one feature like shown in Figure 4.1a.

### 4.1.3. Criterium Evaluation

The chosen criteria described in Section 3.1 are only a subset of all possible implementable parameters. Depending on what is needed for meteorological visualization, a variety of additional parameters can be defined and made accessible through the user interface.

In order to choose new parameters and examine the usefulness of the existent ones, Met.3D might be shown to several meteorologists with a different specialization. This will lead to a much more differentiate view when using and selecting the parameters. Hence this further improves the overall versatility of the system.

The now implemented parameters set the focus mainly on the particles' ascent as four, namely total pressure difference, segment pressure difference, segment height difference, ascendence angle, out of six available parameters handle this thematic range. Thus Met.3D's trajectory actor can be used for the WCB detection described by Rautenhaus et al. [Rau+15a].

## 4.2. Fragment Blending

This thesis provides an implementation of a modern highly optimized Order Independent Transparency approach, which is in most cases capable of rendering the correct scenes. Limiting factors include time and the size of the allocated GPU memory.

### 4.2.1. Overfull Buffers and Arrays

When zooming in a scene with a viewport-filling amount of trajectories being partially transparent, the buffer may run out of free places and in conclusion begin by discarding fragments[1].

A similar problem can occur when resolving the linked list, since the array for storing the fragments temporary is limited in size[2]. The resulting failures are shown in Figure 4.2 along with a correct image for reference[3].

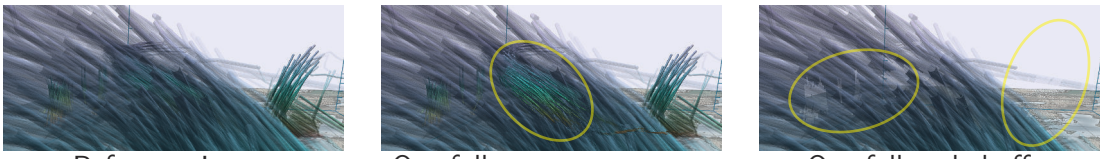



**Figure 4.2.:** Example of missing fragments due to overfull buffers

This problem might be solved by dynamically changing the buffer size depending on the atomic counter value holding the current fragment count. However, solely reading the atomic counter roughly quarters the total performance. This significant performance drop is caused by the synchronization[4] between CPU and GPU. As a reason, this is not applicable in every time-step.

As a further improvement, it might be detected when the viewport does not change for a distinct amount of time. This could trigger a re-render progress while reading the counter and adaptively changing the buffers size. Because the scene is not altered in this time, the user only would notice the better image quality afterward.

### 4.2.2. Further Optimizations

A paged extension of the algorithm, as described by Cyreal Crassin, might further reduce the total rendering time. His approach allocates the fragments in pages, which means less usage of the atomic counter. This counter is the biggest bottleneck when performing the Linked List Construction, as it must be synchronized between all parallel shading operations. Instead of allocating each fragment just before it is rendered, it is stored in an already allocated page which contains several fragments. Every time the page is full, a new page is allocated using the atomic counter. [Cra10]

[1] Modern Methods assume an average amount of two to eight transparent layers throughout the scene. [Shr+13; Kno15] The implemented approach uses an average of 32 to 64.

[2] The array limit is 512 layers. Most approaches use 32-64 layers [Shr+13]

[3] Both the Buffer for storing the nodes and for temporarily holding the fragments were reduced to one tenth of the original size to show noticeable quality losses.

[4] When reading the atomic value of the GPU on the CPU it has to be assured that nothing on the GPU affects the value at the time.

It further also addresses the main problem of resolving the linked list, namely the number of random memory accesses, by bundling the lists nodes in severals pages. This allows the GPU to prefetch the next elements and allows a faster traversing of the linked list. Both parts, assembling and traversing the linked list, make up two-thirds of the total time as seen in Figure 2.20.

### 4.2.3. Future Developments

The ongoing development of GPU hardware and software is worth monitoring. Microsoft recently presented a fundamental approach with *Raster Order Views* (ROV) which is already implemented by Nvidia [Pan15] and Intel [Dav15]. ROVs are currently exclusively supported by DirectX 12. They provide a special view guaranteeing ordering and atomicity. This makes the Linked List and the sorting step, which are the most complex parts in OIT, redundant.

## 4.3. Opacity Mapping

This thesis expanded Met.3D with about 40 new user-controllable parameters. These are all taken into account for calculating the importance criterium or control the occlusion avoidance, in respect to their assigned values.

### 4.3.1. Transfer Function Editor

When further adding possibilities for controlling the opacity, the user-interface of Met.3D may get cluttered. To avoid this, a transfer function editor can be integrated into Met.3D. This features a visual way to configure the mapping between parameter and importance, or respectively between importance and transparency in a more intuitive and versatile way than using sliders. [Aya13]

Furthermore, a transfer function editor provides a much more direct way of defining the mapping since the function curve can be drawn by hand. This may be used to highlight more than one data range or to define other than linear or polynomial growing curves.

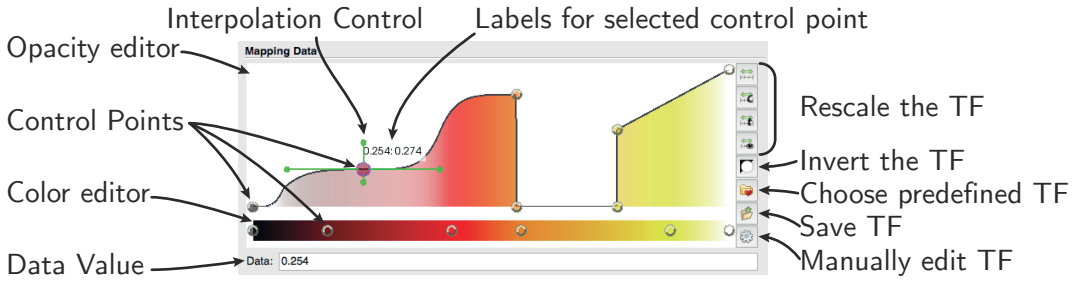



**Figure 4.3.:** ParaView's transfer function editor

The scientific visualization application ParaView has a sophisticated transfer function editor which allows the user to control the color and alpha mapping separately as shown in Figure 4.3. The user freely chooses control points for color and alpha value resulting in flexible mappings. When introducing a transfer function editor to Met.3D, I recommend it to be included system-wide, so every component will benefit.

### 4.3.2. Occlusion Avoidance

When handling more than 100 overlaying layers of trajectories, occlusion, hiding inner structures, comes into account. Therefore, several user-controllable occlusion avoidance methods are built in. The used methods are a compromise between runtime and implementation effort and help to highlight the inner structure as shown in Figure 3.13.

It might be interesting which effect an complete integration of the adaptive line hierarchy approach by Kanzler [Kan16] would have. This method is, as demonstrated by the author, capable of thinning out similar structures efficiently after the hierarchy is generated. Furthermore it is tested with an data set in similar size leading to competitive run-times.

The hierarchy could also be used as a foundation for other methods such as the opacity optimization by Günther [GRT13]. This approach is overwhelmed by the sheer number of path-lines when passing the data unfiltered. However, with a selective line hierarchy filter eliminating similar lines, it could produce optimized results in reasonable time.

# 5. Summary & Conclusion

In this thesis, new options for the visualization of Lagrangian particle trajectories in Met.3D have been realized by implementing an Order Independent Transparency (OIT) approach. OIT is a class of blending techniques, capable of rendering scenes correctly without altering the order of the contained elements in memory. Because all compared approaches are designed to handle a small to medium amount of transparent layers, it was shown the implemented algorithm adopted from Shreiner et al. was not competitive.

The high amount of over hundred transparent layers was challenging because there was no optimized OIT algorithm for this specific problem. I designed a variation of OIT using priority queues to minimize the previously dominating sorting time. Although it was shown that this approach decreases the render-time to an twentieth, it still is not capable of rendering all complex scenes with more than ten frames per second due to sometimes more than 500 transparent layers.

Based on the foundation of this OIT algorithm I introduced several user-controllable relevance factors which are assigned to the transparency value. These factors filter the trajectory domain in order to highlight the user-defined features.

In order to avoid distracting fragments between camera and features, several interactive methods for occlusion avoidance and transparency optimization are compared. The biggest challenge was again the high amount of fragments which are mostly not handleable by these algorithms. However, an algorithm is integrated which performs view-dependent filtering without affecting the performance.

Although the implemented approaches deliver solid, fast, and adjustable results, the trajectory actor can further be optimized, which is discussed in the end of the thesis.

In conclusion, the first step in bringing modern trajectory visualization approaches to Met.3D is made. The implemented techniques are a subset of the state of the art methods and Met.3D benefits from the new possibilities.

It allows meteorologists to explore the data sets in a more versatile way than before, by mapping opacity values according to different parameters. The six integrated parameters[1] are mainly chosen to highlight ascending trajectories. Hence an evaluation of the parameters among different specialized meteorologists will be useful and will further expand the possibilities.

[1] Integrated are curvature, velocity (length), ascent angle, segment height difference, segment pressure difference, and total pressure difference.

# A. Implementation

This chapter contains an excerpt of the implemented code during this thesis. The shaders were written in GLSL Version 4.3

## A.1. Linked List

This section contains the relevant parts of the Linked List Implementation in Met.3D, mainly both Fragment Shaders called `FStoLL` and `FSblend`. All relevant called methods are also listed except for the sorting methods, because they are listed separately in Section A.2.

```
/** Store the fragments in the LinkedList
 *
 * First Shader Pass ( Geometry Pass )
 */
shader FStoLL(in vec3 gs_worldSpaceNormal, in vec3 gs_worldSpacePosition, in ↩
    float gs_scalar, out vec4 fragColor)
{
   // Allocate the place
   uint nextFree = atomicCounterIncrement(fragsCount);
   // Check whether the spot is in the buffer
   if(nextFree < maxFrags)
   {
      // Color and Lighting from transfer function
      vec4 color = rgbColor(gs_worldSpaceNormal, gs_worldSpacePosition, ↩
          gs_scalar);
      // Use fix factor for alpha (debugging + test)
      color.a = 0.5;
      // Append the new fragment & Preserve the old Head
      uint oldHead = imageAtomicExchange(headIndex, ivec2(gl_FragCoord.xy), ↩
          nextFree);
      uvec4 item;
      // Fill the new node
      item.x = packUnorm4x8(color); // Color 4x 8bit float => uint
      item.y = floatBitsToUint(gl_FragCoord.z); // Depth float => uint
      item.z = oldHead; // Attach the tail
```

```
   }
   fragColor = vec4(0.0); // Write no color
}

/** Receive the fragments from the LinkedList
 *
 * Second Shader Pass ( Full Screen Quad )
 */
uint collectFragments(uint headPtr)
{
   uint fragsCount = 0; // No fragments yet
   while(headPtr != 0x00 && fragsCount < MAX_FRAGS)
   { // Extract the fragments
      uvec3 frag = imageLoad(listBuffer, int(headPtr));
      headPtr = frag.z;
      fragList[fragsCount].color = frag.x;
      fragList[fragsCount].depth = uintBitsToFloat(frag.y);
      fragsCount++;
   }
   return fragsCount;
}

vec4 frontToBack(uint fragsCount)
{
   vec4 rayColor = vec4(0.0); // Start with transparent Ray
   for(uint i = 0 ; i < fragsCount ; i++)
   {
      vec4 color = unpackUnorm4x8(fragList[i].color);
      // FTB blending equation
      rayColor.rgb += (1 - rayColor.a) * color.a * color.rgb;
      rayColor.a += color.a * (1 - rayColor.a);
      // Stop when color is saturated enough
      if(rayColor.a > 0.99)
      {
         break;
      }
   }
   rayColor.rgb = rayColor.rgb/rayColor.a; //
   return rayColor;
}
```

```
shader FSblend(out vec4 fragColor)
{
   // Receive the current head Pointer
   uint headPtr = imageLoad(headIndex, ivec2(gl_FragCoord.xy)).r;
   if(headPtr == 0x00)
   { // When unchanged => Transparent Fragment
      fragColor = vec4(0.0);
   } else {
      uint fragsCount = collectFragments(headPtr);
      bubbleSort(fragsCount);
      fragColor = frontToBack(fragsCount);
   }
}
```

## A.2. Sorting Algorithms

This section contains all sorting algorithms used for comparison in Section 2.4.2.

### A.2.1. Bubble Sort

Bubble sort was implemented according to [SW11]. It was proposed to be used for OIT by Shreiner et al. [Shr+13]

```
void bubbleSort(uint fragsCount)
{
   bool changed; // Has anything changed yet
   do {
      changed = false; // Nothing changed yet
      for(uint i = 0 ; i < fragsCount - 1 ; ++i)
      { // Go through all
         if(fragList[i].depth > fragList[i+1].depth)
         { // Order not correct => Swap
            swapFrags(i, i+1);
            changed = true; // Something has changed
         }
      }
   } while(changed); // Nothing changed => sorted
}
```

### A.2.2. Insertion Sort

Insertion sort was implemented according to [SW11].

```
void insertionSort(uint fragsCount)
{
   uint i, j;
   fragType frag; // Temporary fragment storage
   for(i = 1 ; i < fragsCount ; ++i)
   {
      frag = fragList[i]; // Get the fragment
      j = i; // Store its position
      while (j >= 1 && fragList[j-1].depth > frag.depth)
      { // Shift the fragments through the list until place is found
         fragList[j] = fragList[j-1];
         --j;
      }
      fragList[j] = frag; // Insert it at the right place
   }
}
```

### A.2.3. Shell Sort

Shell sort sort is a gapped insertion sort. It was implemented according to [SW11] with the optimal gap sequence for comparisons from [Ciu01].

```
void shellSort(uint fragsCount)
{
   uint i, j, gap;
   fragType frag; // Temporary fragment storage
   // Optimal gap sequence for 128 elements from [Ciu01, table 1]
   uvec4 gaps = uvec4(24, 9, 4, 1);
   for(uint g=0; g < 4 ; g++ )
   { // For every gap
      gap = gaps[g]; // Current Cap
      for( i = gap ; i < fragsCount ; ++i )
      {
         frag = fragList[i]; // Get the fragment
         j = i;
         // Shift earlier until correct
         while (j >= gap && fragList[j-gap].depth > frag.depth)
         { // Shift the fragments through the list until place is found
```

```
            fragList[j] = fragList[j-gap];
            j-=gap;
         }
         fragList[j] = frag; // Insert it at the right place
      }
   }
}
```

### A.2.4. Heapsort

This classic Heapsort implementation operating on binary heaps was implemented according to Sedgewick and Wayne [SW11].

```
void maxHeapSink(uint x, uint fragsCount)
{
   uint c; // Child
   while((c = 2 * x + 1) < fragsCount)
   { // While children exist
      if(c + 1 < fragsCount && fragList[c].depth < fragList[c+1].depth)
      { // Find the biggest of both
         ++c;
      }
      if(fragList[x].depth >= fragList[c].depth)
      { // Does it have to sink
         return;
      } else {
         swapFrags(x, c);
         x = c; // Swap and sink again
      }
   }
}

void heapSort(uint fragsCount)
{
   uint i;
   for(i = (fragsCount + 1)/2 ; i > 0 ; --i)
   { // Bring it to heap structure
      maxHeapSink(i-1, fragsCount); // Sink all inner nodes
   }
   // Heap => Sorted List
```

```
   for(i=1;i<fragsCount;++i)
   {
      swapFrags(0, fragsCount-i); // Swap max to List End
      maxHeapSink(0, fragsCount-i); // Sink the max to obtain correct heap

   }
}
```

## A.3. Priority Queue

This section contains the optimized code, which is the result of this thesis and currently driving Met.3D's trajectory actor. The 1uaternary heap priority queue was implemented according to my earlier work [Ban12] and Sedgewick and Wayne Da[SW11].

```
void minHeapSink4(uint x, uint fragsCount)
{
   uint c, t; // Child, Tmp
   while((t = 4 * x + 1) < fragsCount)
   {
      if(t + 1 < fragsCount && depthList[t] > depthList[t+1])
      { // 1st vs 2nd
         c = t + 1;
      } else {
         c = t;
      }
      if(t + 2 < fragsCount && depthList[c] > depthList[t+2])
      { // Smallest vs 3rd
         c = t + 2;
      }
      if(t + 3 < fragsCount && depthList[c] > depthList[t+3])
      { // Smallest vs 3rd
         c = t + 3;
      }
      if(depthList[x] <= depthList[c])
      {
         return;
      } else {
         swapFrags(x, c);
         x = c;
      }
   }
```

```
}

vec4 frontToBackPQ(uint fragsCount)
{
   // Bring it to heap structure
   for(i = fragsCount/4;i>0;--i)
   { // First is not one right place - will be done in for
      minHeapSink4(i, fragsCount); // Sink all inner nodes
   }
   vec4 rayColor = vec4(0.0); // Start with transparent Ray
   i=0;
   while(i<=fragsCount && rayColor.a < 0.99)
   {// Max Steps = #frags Stop when color is saturated enough
      minHeapSink4(0, fragsCount - i++); // Sink it right + increment i
      vec4 color = unpackUnorm4x8(colorList[0]); // Heap first is min
      // FTB Blending
      rayColor.rgb += (1-rayColor.a)*color.a*color.rgb;
      rayColor.a += color.a*(1-rayColor.a);
      colorList[0] = colorList[fragsCount-i]; // Move Fragments up for next ↩
          run
      depthList[0] = depthList[fragsCount-i];
   }
   rayColor.rgb = rayColor.rgb/rayColor.a; // Correct rgb with alpha
   return rayColor;
}

shader FSblend(out vec4 fragColor)
{
   // Receive the current head Pointer
   uint headPtr = imageLoad(headIndex, ivec2(gl_FragCoord.xy)).r;
   if(headPtr == 0x00)
   { // When unchanged => Transparent Fragment
      fragColor = vec4(0.0);
   } else {
      uint fragsCount = collectFragments(headPtr);
      fragColor = frontToBackPQ(fragsCount);
   }
}
```

# B. Runtime Measurements

## B.1. Individual OIT steps

| *Total Time* | **Scene 1 (blank)** | **Scene 2 (overview)** | **Scene 3 (detail)** |
|---|---|---|---|
| **No sorting** | 1.94 ms (516 fps) | 29.4 ms (34.0 fps) | 66.4 ms (15.1 fps) |
| **No blending** | 1.97 ms (508 fps) | 698 ms (1.43 fps) | 1579 ms (0.633 fps) |
| **Only head access** | 1.14 ms (879 fps) | 14.1 ms (71.1 fps) | 28.8 ms (34.7 fps) |
| **No 2nd pass** | 1.12 ms (894 fps) | 14.0 ms (71.5 fps) | 28.8 ms (34.7 fps) |
| **No linked list** | 1.12 ms (892 fps) | 3.09 ms (324 fps) | 4.78 ms (209 fps) |
| **Total** | 1.96 ms (509 fps) | 698 ms (1.43 fps) | 1579 ms (0.633 fps) |

**Table B.1.:** Runtime measurement of OIT steps (without early fragment tests & backface culling)

## B.2. Different Sorting Algorithms

| *Total Time* | **Scene 1 (blank)** | **Scene 2 (overview)** | **Scene 3 (detail)** |
|---|---|---|---|
| **No sorting** | 1.97355 ms | 16.6113 ms | 35.5450 ms |
| **Bubble sort** | 1.97837 ms | 227.273 ms | 476.190 ms |
| **Insertion sort** | 1.95033 ms | 152,284 ms | 333.333 ms |
| **Shell sort** | 1.94502 ms | 73,8916 ms | 185.185 ms |
| **Heapsort (2)** | 1.96245 ms | 70,4225 ms | 165.746 ms |
| **Heapsort (3)** | 1.97200 ms | 68,1818 ms | 159.574 ms |
| **Heapsort (4)** | 1.94439 ms | 66,3717 ms | 154.639 ms |
| **Heapsort (5)** | 1.97707 ms | 72,6392 ms | 169.492 ms |

**Table B.2.:** Runtime measurement of OIT with different sorting algorithms

## B.3. Different Heap Widths

### B.3.1. One Array

| *Total Time* | **Scene 1 (blank)** | **Scene 2 (overview)** | **Scene 3 (detail)** |
|---|---|---|---|
| **Heapsort (2)** | 1.96245 ms | 70.4225 ms | 165.746 ms |
| **Heapsort (3)** | 1.97200 ms | 68.1818 ms | 159.574 ms |
| **Heapsort (4)** | 1.94439 ms | 66.3717 ms | 154.639 ms |
| **Heapsort (5)** | 1.97707 ms | 72.6392 ms | 169.492 ms |

**Table B.3.:** Runtime measurement of OIT with different heap widths

### B.3.2. Separated Arrays

| *Total Time* | **Scene 1 (blank)** | **Scene 2 (overview)** | **Scene 3 (detail)** |
|---|---|---|---|
| **Heapsort (2)** | 1.96477 ms | 69.4444 ms | 163.043 ms |
| **Heapsort (3)** | 1.97070 ms | 67.1141 ms | 157.895 ms |
| **Heapsort (4)** | 1.96980 ms | 64.6552 ms | 151.515 ms |
| **Heapsort (5)** | 1.98886 ms | 66.8151 ms | 156.250 ms |

**Table B.4.:** Runtime measurement of OIT with different heap widths and separated arrays

### B.3.3. Priority Queue

| *Total Time* | **Scene 2 (overview)** | **Scene 3 (detail)** |
|---|---|---|
| **Transparent** | 65.6455 ms | 153.846 ms |
| **6.25% Opacity** | 64.2398 ms | 153.846 ms |
| **12.5% Opacity** | 53.8600 ms | 131.004 ms |
| **25% Opacity** | 40.8719 ms | 94.9367 ms |
| **50% Opacity** | 31.5789 ms | 69.6056 ms |
| **75% Opacity** | 27.6243 ms | 60.1202 ms |
| **Opaque** | 23.1481 ms | 50.0835 ms |

**Table B.5.:** Runtime measurement of OIT with a quaternary priority queue

# List of Figures

# List of Tables

# Bibliography


[Aya13] U. Ayachit. *Using the Color Map Editor in ParaView – The Basics*. https://blog.kitware.com/using-the-color-map-editor-in-paraview-the-basics/. Nov. 2013.

[Bak07] S. Baker. *Alpha-blending and the Z-buffer*. https://www.sjbaker.org/steve/omniv/alpha_sorting.html. (last accessed September 9, 2016). Apr. 2007.

[Ban12] M. Bandle. "Heapsort – Implementierung, Vergleich verschiedener Varianten und Laufzeitanalyse." Seminararbeit, Neufahrn b. Freising, Oskar-Maria-Graf Gymnasium. Nov. 2012.

[BGZ02] H.-J. Bungartz, M. Griebel, and C. Zenger. *Einführung in die Computergraphik – Grundlagen, Geometrische Modellierung, Algorithmen*. 2nd ed. Wiesbaden: Vieweg+Teubner Verlag, 2002.

[Bür10] K. Bürger. "Particle-based Flow Visualization." PhD thesis. Munich, Germany: Technische Universität München, Nov. 2010.

[Car84] L. Carpenter. "The A-buffer, an Antialiased Hidden Surface Method." In: *Computer Graphics*. July 1984, pp. 103–108.

[Cat74] E. Catmull. "A Subdivision Algorithm for Computer Display of Curved Surfaces." PhD thesis. Salt Lake City: University of Utah, 1974.

[Ciu01] M. Ciura. "Best increments for the average case of shellsort." In: *International Symposium on Fundamentals of Computation Theory*. Springer. July 2001, pp. 106–117.

[CL96] T. F. Coleman and Y. Li. "A Reflective Newton Method for Minimizing a Quadratic Function Subject to Bounds on Some of the Variables." In: *SIAM J. on Optimization* 6.4 (Apr. 1996), pp. 1040–1058.

[Cra10] C. Crassin. *OpenGL 4.0+ ABuffer V2.0: Linked lists of fragment pages*. http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html. (last accessed September 9, 2016). July 2010.

[Dav15] L. Davies. "OIT to Volumetric Shadow Mapping, 101 Uses for Raster Ordered Views using DirectX 12." GPU Technology Conference. Mar. 2015.

[EUM14] EUMeTrain. *Clear Air Turbulence*. http://www.eumetrain.org/data/3/304/print_2.htm. (last accessed September 9, 2016). Jan. 2014.

[Eve01] C. Everitt. *Interactive Order-Independent Transparency*. June 2001.

[GR05] T. Gneiting and A. E. Raftery. "Weather Forecasting with Ensemble Methods." In: *Science* 310.5746 (Oct. 2005), pp. 248–249.

[GRT13] T. Günther, C. Rössl, and H. Theisel. "Opacity Optimization for 3D Line Fields." In: *ACM Transactions on Graphics (Proc. ACM SIGGRAPH)* 32.4 (Apr. 2013), 120:1–120:8.

[Kan16] M. Kanzler. "Line Density Control in Screen-Space via Balanced Line Hierarchies." Submitted to Computers & Graphics. July 2016.

[Kno15] P. Knowles. "Real-Time Deep Image Rendering and Order Independent Transparency." PhD thesis. RMIT University, Australia, Aug. 2015.

[Kub14] C. Kubisch. "Order Independent Transparency In OpenGL 4.x." GPU Technology Conference. Apr. 2014.

[LC79] G. H. Liljequist and K. Cehak. *Allgemeine Meteorologie*. 2nd ed. Braunschweig: Friedr. Vieweg & Sohn, 1979.

[Mau+11] M. Maule, J. L. Comba, R. P. Torchelsen, and R. Bastos. "A survey of raster-based transparency techniques." In: *Computers & Graphics* 35.6 (July 2011), pp. 1023–1034.

[Mau+12] M. Maule, J. L. D. Comba, R. P. Torchelsen, and R. Bastos. "Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer." In: *25th SIBGRAPI Conference on Graphics, Patterns and Images, SIBGRAPI 2012, Ouro Preto, Brazil, August 22-25, 2012*. July 2012, pp. 134–141.

[Mau+13] M. Maule, J. L. Comba, R. P. Torchelsen, and R. Bastos. "Memory-optimized order-independent transparency with Dynamic Fragment Buffer." In: *Computers & Graphics* (July 2013).

[MB13] M. McGuire and L. Bavoil. "Weighted blended order-independent transparency." In: *Journal of Computer Graphics Techniques (JCGT)* 2.2 (2013).

[Pan15] A. Panteleev. "New GPU Features of Nvidia's Maxwell Architecture." GPU Technology Conference. Mar. 2015.

[Pic84] H. Pichler. *Dynamik der Athmosphäre*. Zürich: Bibliographisches Institut, 1984. ISBN: 3-411-01685-X.

[Rau+15a] M. Rautenhaus, C. M. Grams, A. Schäfler, and R. Westermann. "Three-dimensional visualization of ensemble weather forecasts – Part 2: Forecasting warm conveyor belt situations for aircraft-based field campaigns." In: *Geoscientific Model Development* 8.7 (2015), pp. 2355–2377.

[Rau+15b] M. Rautenhaus, M. Kern, A. Schäfler, and R. Westermann. "Three-dimensional visualization of ensemble weather forecasts – Part 1: The visualization tool Met.3D (version 1.0)." In: *Geoscientific Model Development* 8.7 (2015), pp. 2329–2353.

[Rau15] M. Rautenhaus. "Interactive 3D visualization of ensemble weather forecasts." PhD thesis. Munich, Germany: Technische Universität München, July 2015.

[RD90] R. Rew and G. Davis. "NetCDF: an interface for scientific data access." In: *IEEE Computer Graphics and Applications* 10.4 (July 1990), pp. 76–82.

[Sch+14] A. Schäfler, M. Boettcher, C. M. Grams, M. Rautenhaus, H. Sodemann, and H. Wernli. "Planning aircraft measurements within a warm conveyor belt." In: *Weather* 69.6 (June 2014), pp. 161–166.

[Shr+13] D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane. *The OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. 8th ed. Addison Wesley, Mar. 2013.

[SW11] R. Sedgewick and K. Wayne. *Algorithms*. Fourth Edition. Boston: Addison-Wesley Professional, 2011. ISBN: 978-0-321-57351-3.

[SW15] M. Sprenger and H. Wernli. "The LAGRANTO Lagrangian analysis tool – version 2.0." In: *Geoscientific Model Development* 8.8 (Feb. 2015), pp. 2569–2586. DOI: 10.5194/gmd-8-2569-2015.

[Szo+03] E. J. Szoke, U. H. Grote, P. T. McCaslin, and P. A. McDonald. "3D Update: Is it being used?" In: *Proceedings of the 19th IIPS Conference*. Feb. 2003.

[VKG04] I. Viola, A. Kanitsar, and M. E. Groller. "Importance-Driven Volume Rendering." In: *Proceedings of the Conference on Visualization '04*. VIS '04. Washington, DC, USA: IEEE Computer Society, Sept. 2004, pp. 139–146.

[Vri15] J. de Vries. *Learn OpenGL*. `http://learnopengl.com/`. (last accessed September 9, 2016). Sept. 2015.

[Yan+10] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. "Real-time Concurrent Linked List Construction on the GPU." In: *Proceedings of the 21st Eurographics Conference on Rendering*. EGSR'10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, Aug. 2010, pp. 1297–1304.