TU WIEN Informatics

3D Visualization of GPX Tracks in AlpineMaps

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software und Information Engineering

eingereicht von

Jakob Günther Maier
Matrikelnummer 11809618

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr.in techn. MSc Manuela Waldner
Mitwirkung: Univ.Ass. Dipl.-Ing. BSc Adam Celarek

Wien, 10. Mai 2024

Jakob Günther Maier	Manuela Waldner

TU WIEN Informatics

3D Visualization of GPX Tracks in AlpineMaps

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Jakob Günther Maier
Registration Number 11809618

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr.in techn. MSc Manuela Waldner
Assistance: Univ.Ass. Dipl.-Ing. BSc Adam Celarek

Vienna, May 10, 2024

Jakob Günther Maier	Manuela Waldner

Erklärung zur Verfassung der Arbeit

Jakob Günther Maier

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. Mai 2024

Jakob Günther Maier

Danksagung

Ich möchte mich an dieser Stelle bei Manuela Waldner und Adam Celarek für all die Unterstützung und Hilfe bedanken, die ich während dem Implementieren und Schreiben dieser Arbeit erhalten habe.

Kurzfassung

Die Integration von GPS-Daten in digitale Systeme zur Geländedarstellung ist eine wichtige Aufgabe für Bereiche, wie etwa Outdoor-Sport, Forschung oder Bildung. Viele Nutzer, von Hobbysportlern bis hin zu Forschern, benötigen hochwertige und leistungsstarke Lösungen für die Visualisierung von räumlichen Daten sowie Daten, die daraus abgeleitet werden können. Das GPS Exchange Format (GPX) ist ein Standard für die Speicherung von solchen GPS-Track-Daten und ermöglicht den Austausch von Tracks, Routen und Wegpunkten zwischen verschieden Anwendungen und Geräten.

In dieser Arbeit wird eine effektive Methode zum Rendern von solchen 3D GPX-Daten in das interaktive 3D Terrain Rendering Programm *AlpineMaps* integriert. Dabei wird eine Raytracing basierte Methode verwendet, um auch große Mengen an Datenpunkten performant rendern zu können.

Diese Arbeit legt speziellen Fokus auf 3D Tracks, die zumindest teilweise weit über der Erdoberfläche verlaufen. Solche Tracks, wie sie etwa beim Paragleiten entstehen, werden von traditionellen GPS-Track Rendering Methoden vernachlässigt. Das Digitale Höhenmodell, dass von AlpineMaps verwendet wird, macht keinen Unterschied zwischen der Höhe des Terrains und der Höhe von Vegetation von Bauwerken. Tracks, die dadurch verdeckt werden, müssen daher besonders betrachtet und dargestellt werden.

Die Ergebnisse unserer Arbeit zeigen, dass GPU-Ray-Tracing von Kapsel Geometrie eine effektive Methode zum Erstellen von hochqualitativen Visualisierungen von Liniendaten auf 3D Terrain ist.

Abstract

Integrating Global Positioning System (GPS) data into terrain rendering systems is a vital task for various fields such as outdoor recreation, research or education. Many users, from hobbyist athletes to researchers, need high-quality, high-performance solutions for visualizing both spatial data as a well as contextual data derived from the temporal axis provided by many capturing devices. The GPS Exchange Format (GPX) has become the standard way of storing GPS track data, providing a convenient way to share tracks, routes and waypoints between various devices and applications.

This thesis explores the implementation of a state-of-the-art GPU-based ray tracing method for rendering three-dimensional (3D) GPX tracks on interactive 3D terrain. By generating tight-fitting bounding geometry and then ray tracing rounded capsules in the fragment shader stage, we implement a rendering method that is scalable to real-world requirements and provides insights into geospatial data. Our solution is integrated into the open-source, multi-platform renderer *AlpineMaps*.

This thesis especially focuses on fully 3D tracks that can also be high above the surface level. Such tracks, which can be generated by anything from drones to sailplanes and paragliders, are often neglected by traditional GPS track rendering methods, which focus mostly on surface level tracks. As the Digital Elevation Model used by *AlpineMaps* includes vegetation and buildings, some tracks may be occluded below terrain. This case is handled and visualized with partial transparency of the terrain.

The results of this study show that GPU ray tracing capsule primitives is an effective way of creating high-quality visualization of 3D line data on 3D terrain, offering a valuable tool for analysis.

Contents

CHAPTER 1

Introduction

1.1 Motivation

With the increasing popularity of GPS recording devices among both amateur and professional athletes, there is a growing demand for tools and applications that can visualize GPS tracks on terrain maps realistically and accurately. Such visualizations are vital in learning, training and general decision making, be it when reviewing a past hike, paraglider flight or when planning a future activity. Such data may allow a hiker to avoid a specifically steep part of a route in the future or a paraglider to identify excellent thermals.

The *GPS Exchange Format* (GPX) has become a popular format for storing such GPS track data, providing a convenient way to share routes, tracks and waypoints between different devices and applications. Existing rendering solutions [ces, tra] generally expect track data to be at surface level, however this is not always the case. With the growing popularity of aerial sports, fully 3D tracks are becoming more common and need to be handled by applications.

As has been shown by Hsu and Zhen [ZH21], interactive 3D terrain renderings often also allow for better spatial orientation when compared to traditional, flat maps. Especially users with low map-literacy can benefit from using such applications. For this reason, providing GPX track renderings in an application such as *AlpineMaps* can also have a safety aspect.

In recent years, much work has been done in the field of line set rendering for scientific visualization ([KRW18, GG20, HWU+19]) with ray tracing of geometry primitives proving to be an effective method for rendering such data sets. To our knowledge, these methods have not been widely used in geographic visualization applications yet. However, they appear to be very promising due to their high performance and rendering quality.

1.2 Aim of the Work

The aim of this thesis is to implement a high-quality, high-performance method for rendering GPX tracks on an interactive terrain model. In addition to the pure location data, we also want to visualize derived metrics that can be computed from the spatial as well as the temporal data from the GPX files.

As mentioned before, we specifically want to focus on tracks that are not often limited to the surface level. Traditional approaches ([SGK05, SK07, SZT+16, OC11]) to rendering tracks on terrain generally focus on tracks that are right on the surface of the terrain, as would be expected from tracks generated while hiking, cycling, etc.

Due to the nature of the Digital Elevation Model used by *AlpineMaps*, which does not differentiate between the actual terrain height and vegetation trees or buildings, even ground-level tracks may be occluded by the rendered terrain. It is therefore necessary to handle this type of occlusion in some way to preserve visual legibility.

The implementation of our rendering algorithm is done in the context of an existing terrain renderer, *AlpineMaps*. *AlpineMaps* is a cross-platform, Qt application, available for desktop, mobile and as a web app. It provides interactive high resolution 3D rendering of the Digital Elevation Model (DEM) of Austria. [alpa]

To sum up, we formulate a list of requirements:

1. Read and parse GPX files in *AlpineMaps*.
2. Render one or multiple GPX tracks on Terrain in a readable and aesthetically pleasing manner.
3. Handle occlusion behind terrain, vegetation and other structures.
4. Visualize speed, steepness, etc. as special shading of the track.

1.3 Methodology

To find a technique suitable for the task, we first look at existing methods for line and line-set rendering, some of which have already been used in the context of terrain rendering, and techniques not commonly used in this space. Here we especially focus on methods based on ray tracing.

This leads us to the implementation, where we integrate the chosen method into *AlpineMaps*. We will give a detailed description of the capabilities of *AlpineMaps*, as well as our line-set rendering implementation.

Finally, we evaluate our results by looking at performance measurements and comparing the results of our visualization with an existing, state-of-the-art solution.

CHAPTER 2

Previous Work

We will now discuss some of the previous work done in line rendering on 3D terrain as well as 3D line rendering in general.

2.1 Traditional Methods

In general, traditional methods for rendering vector/line data on height maps can be divided into four groups: texture-based, geometry-based, volume-based, and screen-based methods. With texture-based methods, the vector data is first rasterized and stored in a texture. The terrain is then rendered to a buffer and the texture is projected onto it, as is done by Schneider et al. [SGK05], for example. This is easy to implement but can lead to visible artifacts stemming from aliasing, level-of-detail (LOD) changes and distortions from steep terrain.

Geometry-based methods work by using the vector data to generate vertices and geometry, which is then rendered in the standard 3D way. One example of this work is by Ohlarik and Cozzi [OC11]. These techniques do not generate as many artifacts as texture-based techniques, however, they do not scale well. Rendering a lot of vector data quickly requires large amounts of geometry, as well as special consideration concerning z-fighting and handling of LOD. They are also highly dependent on the LOD handling of the underlying terrain.

Volume-based techniques, as introduced by Schneider and Klein [SK07], create shadow-volumes from vector data by extruding the points into polyhedral meshes. These meshes are then rendered to a shadow stencil, which can then be intersected with the terrain. This technique can still create artifacts where the intersection is smaller than a single pixel.

In screen-based approaches, for example the work by She et al. [SZT+16], each pixel is inversely projected onto the terrain heightmap and tested for intersection with the vector

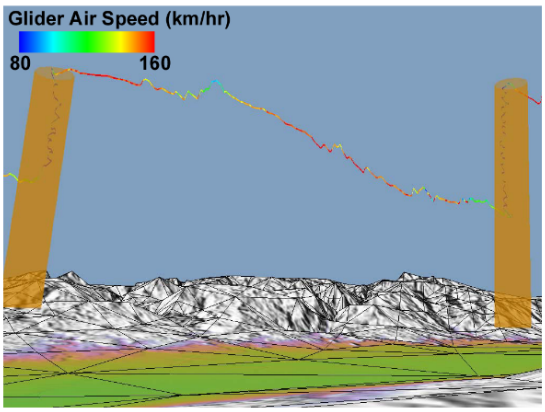

Figure 2.1: Glider track with colormap showing speed [PSH21].

data. If there is such a hit, the pixel is rendered. Using this technique avoids z-fighting artifacts and the artifacts stemming from texture mapping. However, there is the problem of testing every pixel. This can be significantly sped up by using acceleration structures and parallel rasterization algorithms. Most modern approaches use the screen-based vector visualization technique as it can handle large amounts of data, produces very little in the way of artifacts and is largely independent of the terrain rendering technology.

However, all of these techniques have one thing in common: They are either optimized for or even require surface level tracks, as they are either projecting onto or intersecting with the terrain. As such, they are not sufficient for our needs, as our application is required to also support fully 3D tracks.

2.2 3D Methods

We will now look at some previous work that looks beyond traditional vector/line rendering on terrain. First, we look at some previous work that focuses on some of the same problem space, namely visualizing flight tracks with a focus on data relevant to glider pilots. After that, we will look at previous work in the space of 3D line set rendering, mostly in connection with scientific data visualization.

Hsieh et al. [HKH11] propose a method of visualizing local weather characteristics interpreted from GPS flight logs. Along with rendering the flight path and thermals, they also introduce a number of metrics of interest to glider pilots, namely altitude gain per second and velocity. As can be seen in Fig. 2.1, velocity is represented as color coding of the flight track. The exact method of rendering the tracks is not mentioned in the paper. Palenik et al. [PSH21] build on some of the work of Hsieh et al. to visualize atmospheric convection as a set of spheres (see Fig. 2.2).

As line set rendering is an important task in many scientific visualization applications, ranging from rendering fluid flow to turbulence and geographic applications, much has

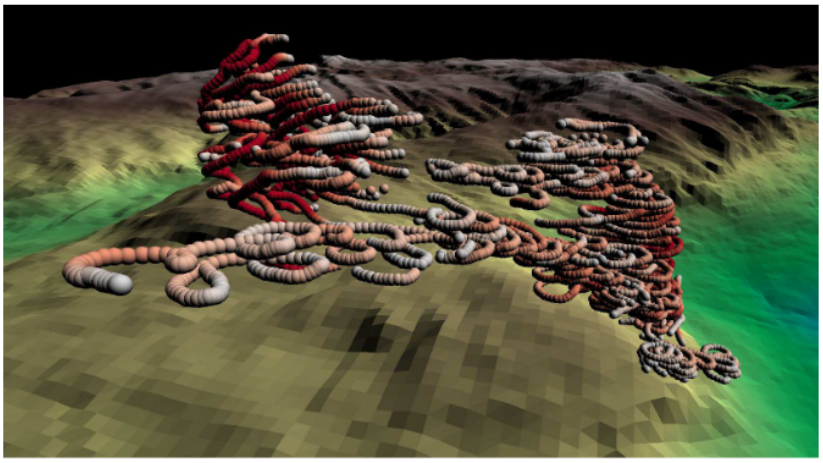

Figure 2.2: Atmospheric turbulence rendered as spheres [PSH21].

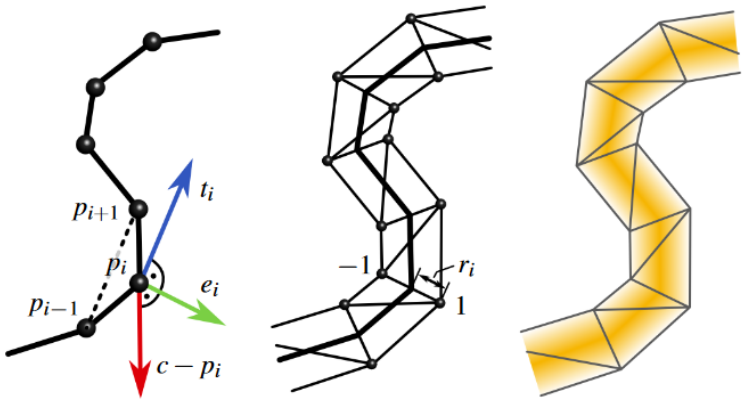

Figure 2.3: Polyline approach [KLG+13].

been written about the topic. The main focus of the work in this field is scaling to large data sets as well as shading and (partial) transparency. [Com21]

Kuhn et al. [KLG+13] create polylines by generating a triangle strip from the original line data with an initial width of zero and expanding it in the fragment shader so that it is camera aligned (see Fig. 2.3). So for each point in the original line, two vertices are created. Each of these vertices holds the location information as well as a tangent vector and the radius of the line at this point. For shading, a drop-off function is used that reduces opacity with increased distance from the original line data, which gives the lines the appearance of actual 3D geometry. This drop-off factor is called density in their work. By layering multiple lines on top of each other with the OpenGL blending functions, areas of increased density can be made visible. As the blending functions used are not dependent on depth order, no depth test is needed.

While the method proposed by Kuhn et al. works well for visualizing local density, it does

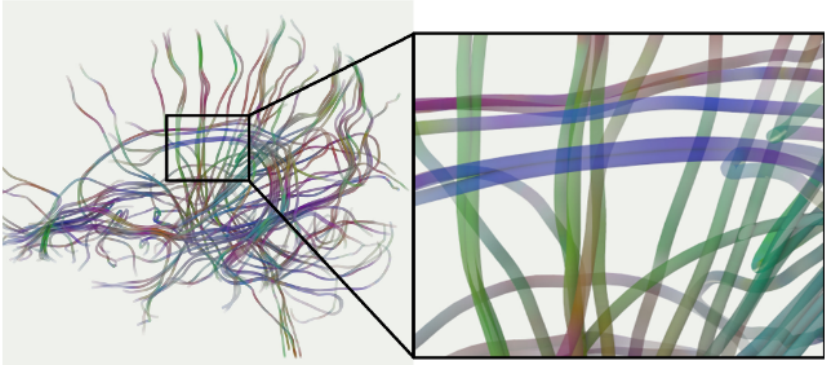

Figure 2.4: Rendering with "Generalized Tubes" [HWU+19].

not allow for rendering actual three-dimensional geometry or more advanced shading techniques, such as ambient occlusion and view-dependent transparency. For this reason, various approaches using ray traced geometry have been proposed.

Han et al. [HWU+19] introduce a rendering primitive which they call the "Generalized Tube", which uses ray traced cones and spheres to render line sets as semi-transparent tubes with varying radii (see Fig. 2.4). In their work, they also propose a method of removing interior surfaces, which would otherwise be visible as artifacts when rendering base primitives with transparency. They do so by using a constructive solid geometry approach and handling the line geometry as a union of the base primitives, i.e. cylinders and spheres. Unfortunately, their implementation is CPU based and implemented for the OSPRay ray tracing library and therefore not directly applicable to our rendering pipeline.

An approach for integrating GPU ray casting into a GPU rasterization pipeline was found by Kanzler et al. [KRW18]: They propose a voxel-based method that works by encoding lines into a voxel grid and then ray casting on the GPU using the voxel grid as an acceleration structure. This also allows for global illumination effects by providing an efficient way to sample points along the ray.

Complex geometry is visualized by de Toledo and Levy [dTL08]: They use implicit GPU primitives to visualize industrial structures such as power plants and oil drilling rigs. These primitives are reverse engineered from triangular meshes to allow the rendering of arbitrary geometry. To enable effective rendering, they also develop a method for computing bounding billboards for 3D cylinders. This is similar to the work by Zhaocong et al. [ZWD19], who propose a GPU raytracing pipeline for visualizing pipelines in virtual urban terrain. In their work, the pipe data is split into adaptive LOD tiles and is then rendered by ray tracing cylinder and torus slices.

Groß and Gumhold [GG20] introduce a method of rendering line sets with GPU-based ray tracing of cone primitives, a cylinder with a half-sphere on each end. This method allows for rendering of large line sets with ambient occlusion and transparency using a

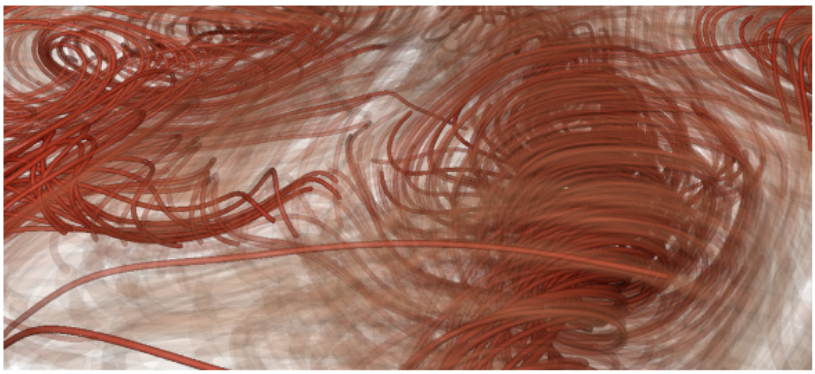

Figure 2.5: GPU ray traced cones[GG20].

voxel ray tracing approach (see Fig. 2.5). They also elaborate on methods for generating tight-fitting bounding geometry, which is crucial when ray tracing in a rasterizing pipeline. Additionally, they propose a technique for determining fragment visibility order from a single geometry pass.

In our work, we will focus mostly on a cone-based method similar to that developed by Groß and Gumhold, as this method is both simple and very powerful for rendering fixed-radius tubes. However, their method has no way of handling occlusion or intersection with some other geometry, so this is an issue that we will have to solve.

CHAPTER 3

Implementation

We will now outline the background knowledge necessary for our implementation and give a detailed description of *AlpineMaps* before describing our implementation of track rendering.

3.1 Background

3.1.1 AlpineMaps

The track rendering solution presented in this thesis builds on top of *AlpineMaps* (see Fig. 3.1), a real-time terrain renderer that enables interaction with a high-resolution DEM of Austria. It is developed by Adam Celarek et al. at Technische Universität Wien. The application is built with Qt 6.6 and OpenGL and is available for Linux and Windows desktop devices, Android mobile devices and as a web application [alpa].

AlpineMaps loads terrain information from remote servers with the same protocol used by tiled web maps, also called "slippy maps", which display maps by dynamically requesting tiles of differing zoom levels and joining them seamlessly. This allows *AlpineMaps* to retrieve only the necessary data at the appropriate time and in the lowest required resolution, optimizing network throughput, as well as GPU and RAM memory usage [alpa].

The basic function of tiled web maps is as follows: the world is divided into square tiles of usually 256x256 pixels, with the entire world visible in a single tile at zoom level 0. When zooming in, each tile is replaced by 4 higher resolution tiles, leading to 1 tile at zoom level 0, 4 tiles at zoom level 1, 16 tiles at zoom level 2 and so on. Most servers do not provide any zoom level greater than 18, at which point 68,719,476,736 tiles are needed to represent the entire earth. Tiles are served with URLs conforming to the pattern `http://.../Z/X/Y.png`, where Z identifies the zoom level, and X and Y are integers

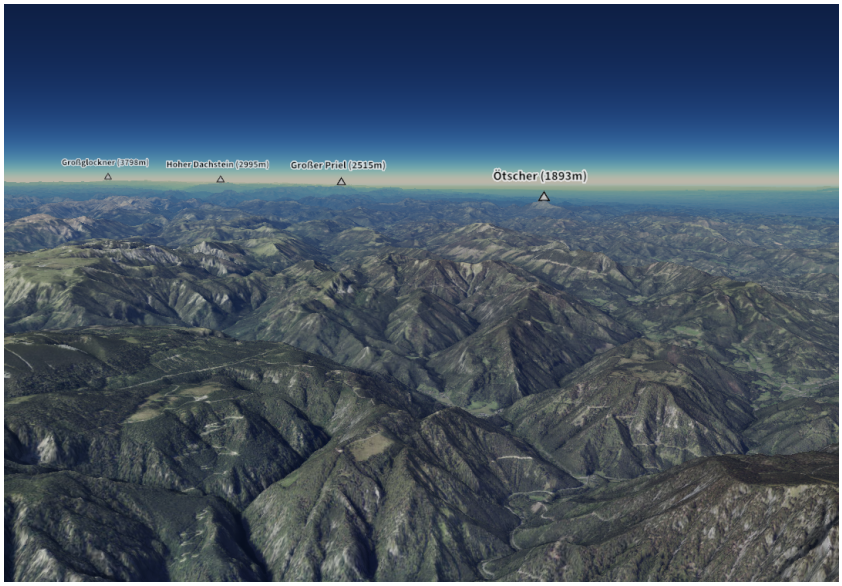

Figure 3.1: AlpineMaps [alpa].

depending on the latitude and longitude of the tile's center. *AlpineMaps* supports several different protocols for requesting tiles, including Web Map Service (WMS), Web Map Tile Service (WMTS) and Tile Map Service (TMS). The main difference between these is the orientation of the Y-axis, i.e. whether it points north or south [ope].

WMS and WTMS use the Web Mercator projection system (EPSG:3857), a modified version of the widely used mercator map projection system (WGS84). Web Mercator features spherical projection, meaning the earth is assumed to be a perfect sphere, with latitude and longitude lines intersecting at perfectly right angles [ope].

AlpineMaps loads two different types of raster tiles from the remote server: orthophoto texture tiles, used for shading the terrain in realistic colors, and heightmap tiles that contain elevation data. These heightmap tiles will be discussed in more detail in Section 3.1.2, when discussing the DEM. The orthographic texture tiles are either satellite or aerial imagery that has been corrected to account for camera view angle, optical distortion and other effects. All tile data used by *AlpineMaps* is provided by *basemap.at*, a service provided by the Austrian government, and is therefore limited to the area of Austria.

When rendering very large grids of tiles, it is essential to have some way of managing the LOD to enable real-time performance. This is essential to minimize rendering time, GPU memory requirements and the number of requests to the server. Tiles closer to the camera shall be of higher resolution and therefore higher zoom level, while tiles further away can be of lower resolution. If a tile is not visible at all, it shall be culled and will not be rendered at all.

For this purpose, *AlpineMaps* uses a quadtree. A quadtree is a tree data structure where

every non-leaf node has four children. This property lines up perfectly with the data model used by the tiled web maps, which simplifies the implementation significantly. Quadtrees can be built by recursively splitting nodes using some heuristic until the desired state is reached. Quadtrees have been used for geometry simplification and LOD management in terrain rendering for a long time, as they are easy to implement and offer good speedup [Paj02]. This quadtree implementation is then used to generate a list of tiles that need to be requested from the server and rendered [alpa].

For increased visual appeal, *AlpineMaps* supports advanced shading techniques such as screen space ambient occlusion (SSAO) and cascaded shadow maps in addition to basic Blinn-Phong [Bli77] shading. To achieve this, *AlpineMaps* uses deferred shading, a rendering technique where color, position, normal and depth information are rendered into an intermediate "G-Buffer", which can then be accessed in a compose shader stage, where the final illumination is computed [Kim23].

When rendering shadows in very large scenes, there is often the problem of not having a sufficient resolution in the shadow maps, which causes undesirable artifacts. One solution to this problem is "cascaded shadow mapping", an approach where multiple shadow maps centered around the camera focus point are used. *AlpineMaps* actually uses four shadow maps, each with a resolution of 4096x4096 pixels. Terrain farther away from the camera is rendered with lower resolution shadows, while terrain closer to the camera has the highest shadow resolution [Kim23]. *AlpineMaps* also provides realistic sun locations depending on location and time, which is then used for shading [alpa].

As with any renderer that aims to render very large terrain, AlpineMaps needs some way of handling reduced floating point precision when rendering geometry far away from the origin. This is done by computing all coordinates relative to the camera before passing them to the shader for rendering [alpa].

3.1.2 Digital Elevation Model

A Digital Elevation Model (DEM) is a representation of the elevation data of some terrain, usually created using remote sensing methods such as Lidar, Radar or stereo photogrammetry [Hir15]. There are two types of DEMs: Digital Terrain Models (DTM) and Digital Surface Models (DSM). The DTM represents only the actual terrain height, while the DSM also includes surface features, such as foliage and buildings [Hir15].

Terrain renderers commonly use 'heightmaps' as a representation of the DEM to generate terrain geometry. These heightmaps encode the elevation values for each point on the terrain in a pixel value that can easily be accessed by using the X and Y values on the terrain as UV coordinates. Height maps are commonly grayscale images, which can lead to artifacts stemming from low precision when using only 8-bit values. To work around this, elevation values can be encoded in multiple channels of the heightmap image tiles.

Such a heightmap tile, used by the AlpineMaps renderer, can be seen in Fig. 3.2. The elevation value is encoded in the red and green channels. To decode this RGB value into

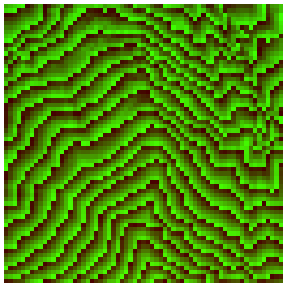

Figure 3.2: Example heightmap tile [alpb].

an elevation value the function from Listing 3.1 is used, where `color` is a floating point vector in the range [0.0, 1.0].

Heightmap-based digital elevation models have the limitation of not being able to represent overhanging terrain. For this reason, this type of terrain model is sometimes also called 2.5D.

```
float altitude_from_color(vec4 color) {
    return (color.r + color.g / 255.0) * 32.0;
}
```

Listing 3.1: Decode elevation value from color [alpa].

3.1.3 The GPX Format

The GPS Exchange Format (GPX) is a simple XML-based format that is used to describe location data. It is often used as a format to share GPS data between applications and devices, for example a GPS smartwatch and an analysis application.

A GPX file can consist of waypoints, routes and tracks. A waypoint consists of coordinates in WGS84 format as well as other optional data. A route is an ordered list of points that usually leads to some destination. A track is an ordered list of coordinate points with an optional timestamp in ISO 8601 format. Every point can also contain elevation data. GPX files can also contain additional data, with many companies defining their custom extension. One such example is TCX, published by Garmin [top].

In our implementation, tracks are the only part of GPX that is visualized, as that is what is generally used by recording devices like smartwatches to record activities. For 3D rendering, coordinates as well as elevation data are necessary. Timestamps are optional for basic rendering, but allow the calculation of speed and vertical speed, which can represented with the use of a colormap.

3.2 Track Rendering

The simplest approach for rendering a track, which is essentially a sequence of lines, is to use a line primitive such as `GL_LINES` available in OpenGL. However, there are

some unacceptable drawbacks to this method, such as only having a fixed width for the line all along the track. This makes perspective-dependent rendering impossible [GG20]. `GL_LINES` also does not support the setting of a line width on all devices, such as WebGL, which makes it unacceptable for *AlpineMaps*.

Lines can also be represented as a strip of geometry, often called a polyline, and then rendered using `GL_TRIANGLE_STRIP`. This polyline is usually rotated so that it is screen-space-aligned. This approach offers good performance and is easy to implement, but has some drawbacks. Firstly, the shading that can be done is limited, as the geometry is entirely flat. Secondly, there can be artifacts if the camera view axis is aligned with the axis of the track. There are some ways of improving the appearance of such a rendering approach, as was done by Kuhn et al. [KLG+13], but their approach still lacks actual shading.

There is also the approach of naively generating explicit tubular meshes for rendering, but this would quickly lead to unacceptably large numbers of vertices and therefore significant performance problems.

A possible solution to many of these problems is to do what is proposed by Han et al. [HWU+19], Kanzler et al. [KRW18] and others mentioned in Section 2.2: to use ray tracing instead of traditional rasterization for line rendering. This can be done by generating camera rays and then using these rays to perform intersection tests against basic geometry primitives such as spheres, cylinders, toruses and capsules. The results of these tests can then be used to determine the final shape and color of the geometry. More complex geometry can be created by combining primitives using a constructive solid geometry (CSG) approach. In this way not only can we render curves and joints in tubes, but also more complex geometry such as entire power plants and oil drilling rigs as shown by Toledo and Levy [dTL08].

This method also allows for pixel perfect representation of tube surfaces and significantly reduces the amount of vertices needed for equivalent geometry while also providing surface normals for shading in the fragment shader stage. Ray-tracing-based methods also do not require any more vertices than polyline approaches.

For this reason, we settle on the following rendering pipeline: Before any rendering is done, GPX tracks are parsed, preprocessed and uploaded to a one-dimensional texture for easy access in the shader. The tracks are then rendered as 3D tubes without the need for explicit tubular geometry by first creating a vertex buffer with bounding quad vertices, which are then aligned with the camera in the vertex shader. Next, the actual geometry is ray traced in the fragment shader, with the track data passed to the fragment shader as a one-dimensional texture. We choose rounded cones as our geometry primitive as it allows us to have rounded corners in our geometry without having to do any explicit handling of joints. The information produced in this step is then combined with terrain geometry information, especially terrain distance from the camera, to calculate occlusion and shade the geometry accordingly. Additionally, metadata, such as timestamps passed to the fragment shader, is used for colormapping to visualize additional data and provide

more context to the individual tracks.

3.2.1 Preprocessing

Consumer GPS devices are reported to have an accuracy of around 5-10m in perfect conditions, although this can get significantly worse in actual use. The GPS satellite signal can be blocked, disturbed or reflected, causing increased Positional Dilution of Precision (PDoP). This is especially noticeable in urban environments, where buildings and other structures cause the so-called "urban canyoning errors". Track points obtained from GPS signals may therefore be distributed around the actual track captured by the recording device [SA09]. For this reason, it is necessary to perform some data preprocessing before any rendering is done in order to improve the appearance of our render.

In our implementation, we first parse the GPX file and convert the point coordinates from the WGS84 coordinate system to the internal coordinate system used by *AlpineMaps* for rendering. Additionally, we enforce a minimum distance between consecutive points, which reduces the number of points that need to be rendered, while also reducing artifacts when a large number of points are in close proximity.

To reduce the noise in the GPS track, we filter the points by applying a Gaussian kernel to the entire track. This has been shown by Schüssler and Axhausen [SA09] to be a simple but effective approach to reducing noise in such data. We do this by first generating a one-dimensional Gaussian kernel, as can be seen in Equation 3.1:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}}. \tag{3.1}$$

This kernel of size N is then applied to each dimension x, y and z of point P at time t:

$$P_t = \sum_{i=-N}^{N} P_{t+i} \cdot G(i). \tag{3.2}$$

After some experiments we found $N = 5$ to be a reasonable choice. In Fig. 3.3, the track was rendered without any preprocessing. In Fig. 3.4 a Gaussian blur with $\sigma = 1.0$ and $N = 5$ was applied. As can be seen from these results, this simple filtering technique is effective in reducing noise in the rendered track.

After preprocessing, we pass the track data to the GPU as an OpenGL texture in `RGBA32F` format, which allows for access to arbitrary points in the fragment shader. The RGB channels are used for the point coordinates, the alpha channel is used for the timestamp.

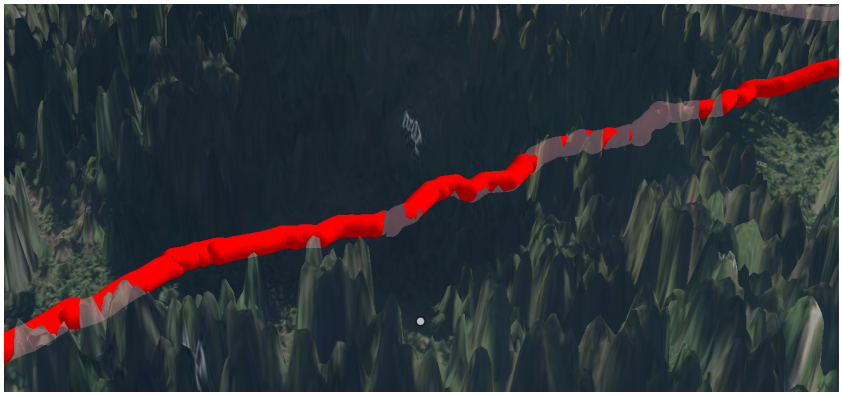

Figure 3.3: Without preprocessing.

Figure 3.4: After preprocessing.

3.2.2 Generating Bounding Geometry

Ray tracing in a rasterizer pipeline requires some way of reducing the number of primitives that need to be tested per pixel. This can be done by generating and rendering some bounding geometry that entirely contains the silhouette of the desired object in view space and passing the necessary geometry parameters to the raytracing stage, where the intersection test can then be performed for every fragment/pixel. Numerous methods for generating such proxy geometry have been proposed, such as by Sigg et al. [SWBG06] for sphere primitives and by Zhaocong et al. [ZWD19] for cylinders. The approach chosen in our implementation is based on Groß and Gumhold [GG20], who propose a method using only a single quad to entirely envelop a rounded cone/capsule.

Generating such proxy geometry would ideally be done in a geometry shader before the vertex shader, but as they are not supported in WebGL as of 2024, this option is not available to us. For this reason, vertices have to be passed to the GPU with an initial offset to the track of zero. The final offset and orientation towards the camera are then computed in the vertex shader.

To calculate this offset, we need the capsule end points $\vec{x_1}$ and $\vec{x_2}$, the camera position $\vec{e}$ and the tube radius r. First, we calculate the track tangent $\vec{t}$ and the view direction vectors from the camera position to the track points:

$$\begin{aligned} \vec{t} &= \vec{x_2} - \vec{x_1}, \\ \vec{d_0} &= \vec{e} - \vec{x_1}, \\ \vec{d_1} &= \vec{e} - \vec{x_2}. \end{aligned} \tag{3.3}$$

Next, we compute two vectors that are orthogonal to the view direction and the track tangent as well as vectors $\vec{v_0}, \vec{v_1}$ that are orthogonal to that plane:

$$\begin{aligned} \vec{u_0} &= \vec{t} \times \vec{d_0} \cdot ||\vec{t} \times \vec{d_0}||^{-1}, \\ \vec{u_1} &= \vec{t} \times \vec{d_1} \cdot ||\vec{t} \times \vec{d_1}||^{-1}. \end{aligned} \tag{3.4}$$

$$\begin{aligned} \vec{v_0} &= \vec{u_0} \times \vec{d_0} \cdot ||\vec{u_0} \times \vec{v_0}||^{-1}, \\ \vec{v_1} &= \vec{u_1} \times \vec{d_1} \cdot ||\vec{u_1} \times \vec{v_1}||^{-1}. \end{aligned} \tag{3.5}$$

These vectors are then added to the original line points to form the vertices p_0 to p_3 of our bounding quad, as can be seen in Equation 3.6 and Fig. 3.5. In our shader implementation this is done by passing two offset factors to the vertex shader that identify the offset of the vertex that needs to be computed. These factors are then always either 1 or -1.

$$\begin{aligned} \vec{p_0} &= \vec{x_1} - (\vec{v_0} \cdot r) - (\vec{u_0} \cdot r), \\ \vec{p_1} &= \vec{x_1} - (\vec{v_0} \cdot r) + (\vec{u_0} \cdot r), \\ \vec{p_2} &= \vec{x_2} + (\vec{v_1} \cdot r) - (\vec{u_1} \cdot r), \\ \vec{p_3} &= \vec{x_2} + (\vec{v_1} \cdot r) + (\vec{u_1} \cdot r). \end{aligned} \tag{3.6}$$

This simple algorithm succeeds in generating tight-fitting bounding geometry, even when the camera view axis is aligned with the main axis. Fig. 3.5 shows the geometry of this calculation, when viewed along the view direction vector $\vec{d}$.

3.2.3 GPU Ray Casting

Older approaches to render tube primitives use a combination of cylinder and sphere primitives [HWU+19], but this can be improved and simplified. As Groß and Gumhold [GG20] show, tubes with a fixed radius can be rendered using only a single primitive: rounded cones.

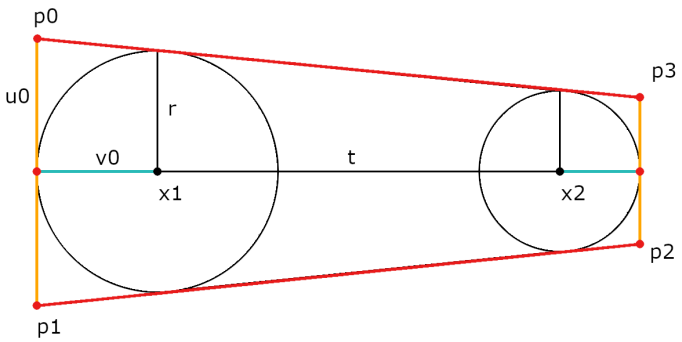

Figure 3.5: Bounding quad calculation.

A rounded cone, often also called “capsule”, is a cylinder with hemispherical ends, which can be defined by two points and a radius. Ray-capsule intersection is well known from ray tracing and collision detection literature. The algorithm used in our implementation, published by Inigo Quilez on his website [int], actually has optimization: the intersection with the hemispherical “cap” only needs to be computed a single time, improving performance.

So using the correct pair of points, which are passed to the fragment shader via a texture and indexed with a vertex id passed from the vertex shader, as well as a user-defined radius, we can now define a capsule primitive, which we can then be intersected with the camera ray. If this intersection test returns a hit, we can calculate the exact hit point as well as a surface normal, which is crucial for realistic shading.

Consecutive capsules share their starting and ending point and therefore overlap. This can lead to visible artifacts, which need to be removed. This issue can be resolved by introducing two clipping planes as proposed by Groß and Gumhold [GG20].

A plane can be defined by a normal $\vec{n}$ and a distanced d from the origin. Using not just our cone end points $\vec{x_1}$ and $\vec{x_2}$ but also $\vec{x_0}$ and $\vec{x_3}$, the track points before and after our cone, we can compute the clipping plane normals $\vec{n_0}, \vec{n_1}$ and distances d_0, d_1. During ray tracing, every hit point is tested against these planes using the signed distance function [Eri05], to determine if the fragment needs to be discarded. This way, only a single ray-capsule intersection needs to be computed for every fragment, and only the geometry between x_1 and x_2 (see Fig. 3.6) is rendered [GG20].

$$\begin{aligned} \vec{n_0} &= \vec{x_0} - \vec{x_2} \cdot ||\vec{x_0} - \vec{x_2}||^{-1}, \\ \vec{n_1} &= \vec{x_1} - \vec{x_3} \cdot ||\vec{x_1} - \vec{x_3}||^{-1}. \end{aligned} \tag{3.7}$$

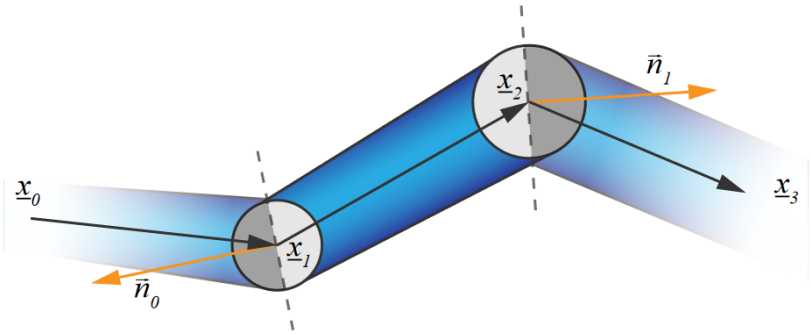

Figure 3.6: Clipping planes [GG20].

$$\begin{aligned} s_0 &= \vec{x_1} \cdot \vec{n_0}, \\ s_1 &= \vec{x_2} \cdot \vec{n_1}. \end{aligned} \tag{3.8}$$

3.2.4 Occlusion Handling

Track occlusion can be divided into two different types: occlusion by terrain and occlusion by vegetation. Occlusion by terrain may have several different causes: Tracks can be hidden behind terrain simply due to the chosen camera view, they can be hidden due to inaccuracies in the terrain model or track data or, finally, a track very low to the surface can cause occlusion if the chosen track width is too large. Occlusion by vegetation is of course natural for surface or near-surface level tracks when a track goes through a forested area.

While a track rendering solution could reasonably discard any tracks that are hidden behind terrain, in our case we do not have this option, since we have no way of identifying which occlusion type is happening. A hiking track that is partially below the rendered terrain due to it going through an area of tall vegetation, such as a forest, must still be clearly visible to the user, while also conveying information on what is happening. For this reason, we chose to render occluded tracks with transparency, as this allows the viewing of the track, while clearly showing it where it is occluded.

The deferred rendering pipeline used by *AlpineMaps* writes the terrain position and distance from the camera to the terrain surface in camera space into a texture, which can then be accessed in our track shader. With the hit point returned by the ray-capsule intersection, we can then compute if the track is below the terrain and by how much. This information is then used for blending the track color with the terrain color. Fig. 3.7 shows an illustration of this technique. t is the distance along the ray of intersection with the capsule, d is the distance from the camera to the terrain surface and $\vec{e}$ is the location of the camera.

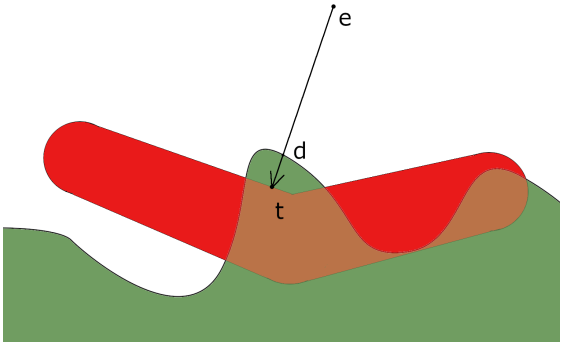

Figure 3.7: Ray tracing track with occlusion.

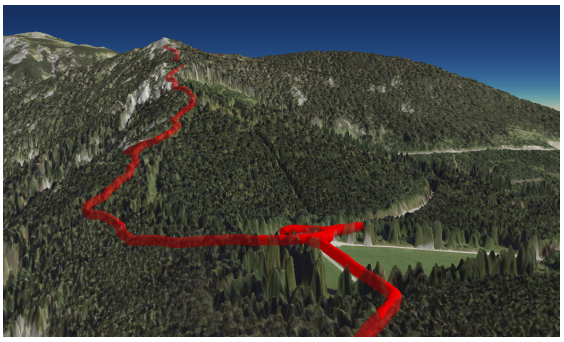

Figure 3.8: Track occluded by forest.

The result of this can be seen in Fig. 3.8, where a hiking track is partially hidden below the trees. We found it useful to render occluded tracks with a fixed alpha value up to a certain depth and then to slowly fade out. This was done because tracks that are hidden far behind whole mountains can otherwise be confusing to users.

3.2.5 Shading

Our track rendering implementation uses the same deferred shading pipeline used for rendering tiles in *AlpineMaps*. This allows us to apply the advanced shading techniques implemented in *AlpineMaps*, including SSAO, to the tracks as well. Deferred shading pipelines often have problems with transparency, which is vital for occlusion handling in our implementation. We solve this issue by reading the terrain color directly from the "G-Buffer" and computing the blended value in the track fragment shader. This is done with the function from Listing 3.2:

```
vec3 blending(vec3 track, vec3 terrain, float alpha) {
    return track * alpha + terrain * (1.0 - alpha);
```

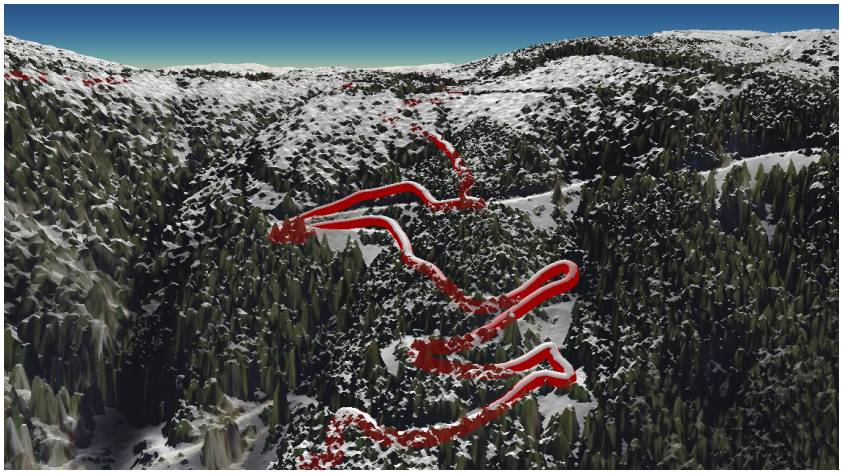

Figure 3.9: Track with simulated snow.

}

Listing 3.2: Additive blending function.

As mentioned before, track rendering can be used to convey additional information along with location data. Using the timestamp recorded along with the coordinates, it is possible to compute the velocity of the recording device, while the angle between consecutive points can be used to calculate the steepness of the track.

In addition to shading the track in a single color, we therefore added the ability to visualize data computed from temporal or spatial properties of the track via a colormap. A colormap is a function that maps a range of values to a set of colors, with the aim of helping the viewer to interpret data. Usually, normalized values are mapped to some color. In our implementation, we used *Turbo* [goo], a colormap developed at Google as an improvement over the traditional, low accuracy, "rainbow maps".

AlpineMaps also supports the simulation of snow as an overlay. As our track shading is integrated into the deferred rendering pipeline, this also affects our tracks (Fig. 3.9).

CHAPTER 4

Results

In this section, we will present the results and findings of our implementation. The evaluation of the results of our work is split into two parts: First, we will look at the performance achieved when rendering both real-world and generated data sets, and in the next step we will compare our renderings with a state-of-the-art solution.

4.1 Performance

All measurements were done using an AMD Ryzen 7 4000 series with integrated AMD Radeon graphics card on Windows 10. When testing the web application, both the Google Chrome (Version 123.0.6312.124) and Firefox (Version 124.0.2) browsers were used. *AlpineMaps* has built-in support for frame profiling, which provides a breakdown of how much time was spent for rendering the terrain, shadows, labels and our tracks. This profiling unfortunately does not work on WebGL, which is why this measurement was only done on the native application.

Consumer GPS devices do not produce very large amounts of data. Garmin devices, which are very popular among athletes, record the GPS location about every second, but can also be configured to do so only after changes to speed, location or elevation are detected [gar]. So when assuming a recording frequency of about 1 Hz, such a device would produce approximately 3600 points per hour. A recording from a typical consumer would likely not be longer than a few hours, so the most common use case for our application would be to render a track consisting of a few thousand points. Our implementation supports loading and rendering multiple GPX tracks, but even then, the number of points is quite low compared to datasets routinely rendered for scientific visualization, which can consist of millions of points. Even so, we tested our implementation with datasets that are significantly larger than what would be expected in normal use.

The datasets used in our performance measurement include a hiking track, a paragliding track, as well as a randomly generated track. We use a randomly generated track to control the amount of points that need to be rendered and also because GPX tracks containing hundreds of thousands to millions of points are not very common.

In Table 4.1 we compare the rendering times for different data sets. These rendering times were calculated when all optional rendering steps and effects (SSAO, shadows, etc) were enabled.

Table 4.1: Rendering datasets of different sizes.

Dataset	Number of Points	GPU Total (ms)	CPU Total (ms)
Hiking Track	7,532	29.97	8.62
Paragliding Track	11,031	39.89	8.9
Generated Track	100,000	47.97	15.48

As mentioned before, *AlpineMaps* has a built-in frame profiler, which allows us to measure and compare the average render time per frame for the individual rendering steps. As can be seen in Table 4.2, when rendering 11,031 points, the track rendering stage makes up only a small fraction of the total rendering budget. The result of this render can be seen in Fig. 4.2. Table 4.3 shows the distribution of rendering times when rendering 1,000,000 points. Here, track rendering takes up a significantly larger proportion of total rendering time per frame.

Table 4.2: Rendering 11,031 points in the native application.

Step	Average Time (ms)
Atmosphere	0.87
Tiles	6.85
Tracks	0.56
Compose	4.0
Labels	0.02
Shadowmap	19.97
SSAO	13.68

In conclusion, our implementation is easily able to handle data sets far in excess of the expected workloads. The results of the performance even suggest our solution may allow us to visualize not just a few GPX tracks, but to create some kind of mass visualization, similar to the heat maps available on many fitness statistics and tracking websites [str, the], but in 3D.

Table 4.3: Rendering 1,000,000 points in the native application.

Step	Average Time (ms)
Atmosphere	0.5
Tiles	5.62
Tracks	12.46
Compose	2.65
Labels	0.03
Shadowmap	25.18
SSAO	7.1

4.2 Visual Readability

One of the requirements (see Section 1.2) for the GPX track rendering method implemented for this thesis was to present GPX tracks in a readable and aesthetically pleasing way. In order to truly evaluate the visual readability of our solution a user study would need to be conducted. However, this was determined to be out of scope for this thesis. For this reason, we will look at some realistic renderings of our solution and compare the results to a state-of-the-art renderer, *Cesium* [ces].

Cesium is a collection of open source libraries for creating 3D maps and rendering various models and data sources on these maps [ces]. It is available for multiple platforms, however, when we talk about *Cesium* in this thesis, we actually mean *CesiumJS*, the JavaScript library. Among many other types of data, *Cesium* also has support for rendering GPX tracks. *Cesium* also supports polyline rendering, which enables developers to render any kind of line data. The *Cesium* polyline implementation uses screen-oriented triangle strips, which offer good performance, but only offer limited shading options.

We will now look at some example renderings done with our rendering solution. Fig. 4.1 shows a track that starts out as a hiking trail, but then transitions to a helicopter flight. This is a good example of the capabilities of our renderer, where we have segments that are obscured by foliage in the bottom part of the image and the flying section that is high above the surface. In this case, the colormap visualizes the speed.

Fig. 4.2 shows a paragliding flight, with the colormap again visualizing speed. The sections that appear bright red clearly show where the pilot descended rapidly. Here, ambient occlusion works very well to give a sense of depth and to improve spatial awareness. This effect is very visible in Fig. 4.2, especially when compared with Fig. 4.2, where SSAO is disabled.

Next, we will look at the same GPX track rendered in both *AlpineMaps* (see Fig. 4.4) and *Cesium* (see Fig. 4.5). The source was captured using a Garmin smartwatch during a short walk of approximately 6 km. Both solutions allow the user to clearly comprehend the spatial extent of the track. One additional feature that our solution provides is the

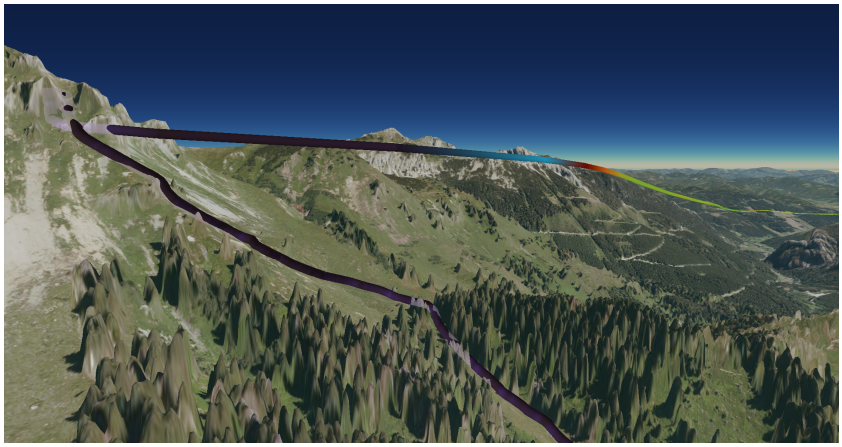

Figure 4.1: Hike and helicopter flight, colormap showing speed.

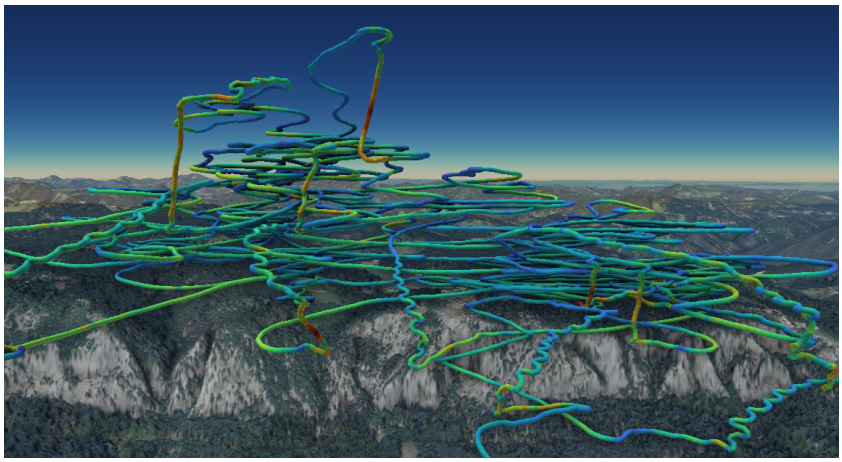

Figure 4.2: Paragliding, with SSAO.

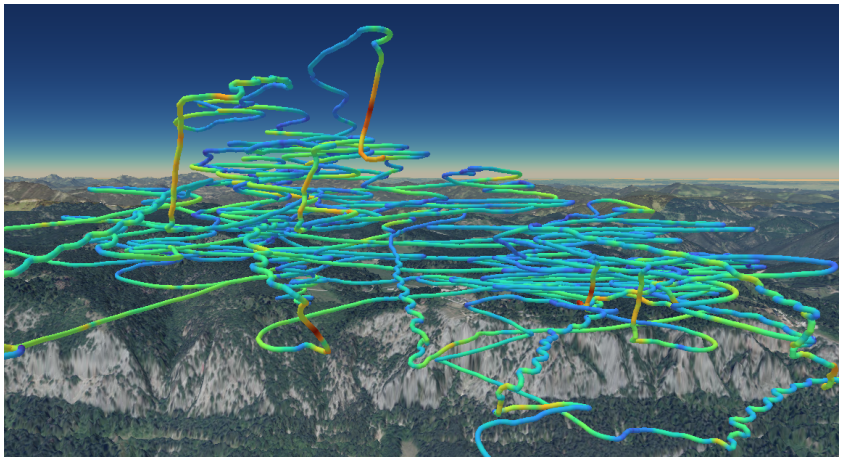

Figure 4.3: Paragliding, without SSAO.

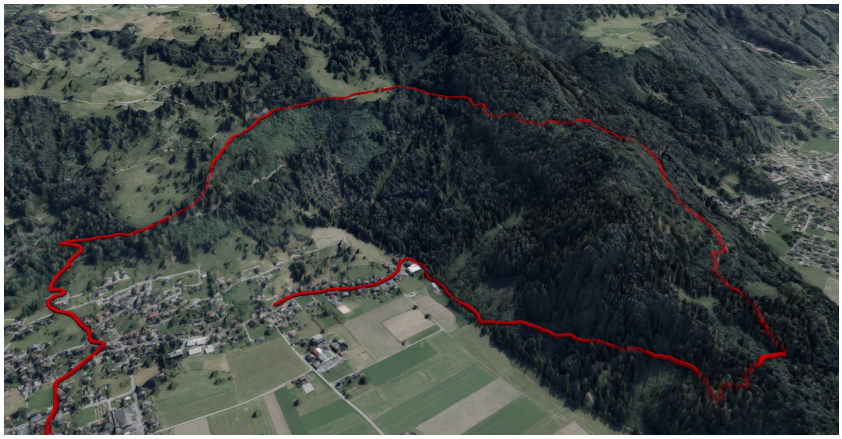

Figure 4.4: Rendering with *AlpineMaps*.

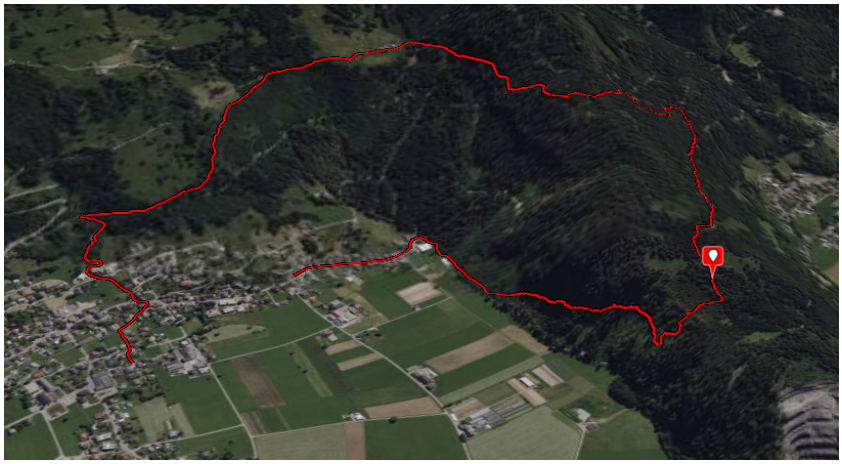

Figure 4.5: Rendering with *Cesium*.

feedback on occlusion using transparency, as can be seen in the right part of the image. In this case, our solution produces a more realistic rendering, as it clearly shows the track going through the forest (see Fig. 4.6).

We argue that our implementation is best suited for visualizing 3D tracks that are above the surface level. As we are using DSM, tracks cannot be clamped to terrain height, giving a somewhat "unclean" appearance when surface level tracks are sometimes below and sometimes above the terrain. For such tracks, *Cesium* would be the preferred solution. But when tracks consist of both surface level and above surface level sections, our solution is quite well suited. In such tracks, our terrain occlusion handling can provide information on foliage conditions as well as rendering the fully 3D parts.

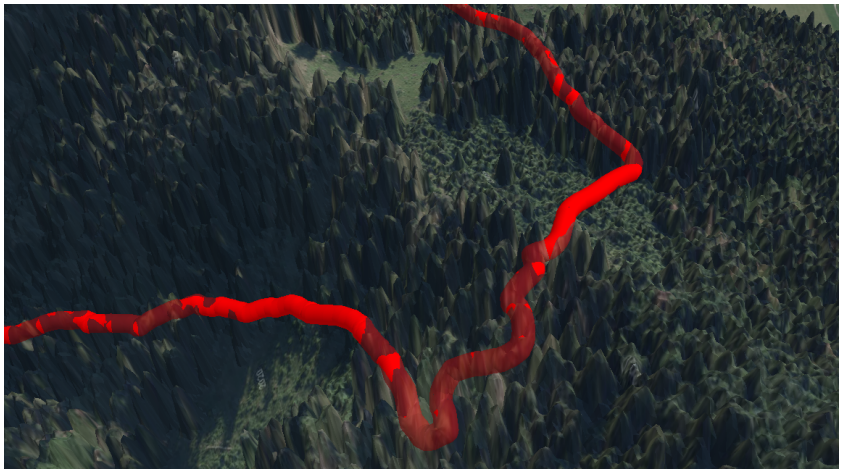

Figure 4.6: Track going through for forest.

CHAPTER 5

Conclusion and Future Work

The aim of this thesis was to implement an efficient solution for rendering GPX tracks on 3D terrain. After reviewing previous work and looking at different rendering approaches, we chose a method based on the work by Groß and Gumhold [GG20]. By implementing a GPU-based ray tracing method along with occlusion handling and additional data visualization, this has been achieved. As was shown in Section 4.1, our solution has more than sufficient performance to render the expected data sets. In Section 4.2, we evaluated the visual readability of our implementation and compared it to an established open-source renderer, *Cesium*. As this comparison shows, our implementation has strengths and weaknesses, but is especially suited for rendering full 3D tracks, which is what we set out to do. The list of requirements formulated in Section 1.2 is therefore fulfilled.

Naturally, the scope of this thesis is limited, leaving some opportunities for future work and improvement. In Section 4.2 we mentioned that, due to insufficient precision in the elevation axis, some tracks can be rendered below the surface by mistake. This problem could be solved by introducing a DTM along with the DTS that is already used, which would enable the clamping of all tracks to a minimum height over the terrain. This which would likely significantly clean up surface track rendering and alleviate some shortcomings of our method compared to established renderers, such as *Cesium*.

List of Figures

List of Tables

Bibliography

[alpa] https://github.com/AlpineMapsOrg/renderer. Accessed: 12.03.2024.

[alpb] https://alpinemaps.cg.tuwien.ac.at/tiles/alpine_png/15/17538/21242.png. Accessed: March 21, 2024.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, jul 1977.

[ces] https://cesium.com/. Accessed: 25.04.2024.

[Com21] A comparison of rendering techniques for 3d line sets with transparency. *IEEE Transactions on Visualization and Computer Graphics*, 27(8):3361–3376, 2021.

[dTL08] Rodrigo de Toledo and Bruno Levy. Visualization of industrial structures with implicit gpu primitives. In *Advances in Visual Computing*, pages 139–150, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Eri05] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[gar] https://support.garmin.com/en-US/?faq=Te47runiFR93oKcUAItwU7&productID=731136. Accessed: 24.04.2024.

[GG20] David Groß and Stefan Gumhold. Advanced rendering of line data with ambient occlusion and transparency. *IEEE Transactions on Visualization and Computer Graphics*, 10 2020.

[goo] https://research.google/blog/turbo-an-improved-rainbow-colormap-for-vis Accessed: 18.04.2024.

[Hir15] Christian Hirt. *Encyclopedia of Geodesy*, chapter Digital terrain models. Springer, 2015.

[HKH11] Tung-Ju Hsieh, Falko Kuester, and Tara Hutchinson. Visualizing Local Weather Characteristics Interpreted from Glider GPS Flight Logs. In Bing-Yu Chen, Jan Kautz, Tong-Yee Lee, and Ming C. Lin, editors, *Pacific Graphics Short Papers*. The Eurographics Association, 2011.

[HWU+19] Mengjiao Han, Ingo Wald, Will Usher, Qi Wu, Feng Wang, Valerio Pascucci, Charles D. Hansen, and Chris R. Johnson. Ray tracing generalized tube primitives: Method and applications. *Computer Graphics Forum*, 38(3):467–478, 2019.

[int] `https://iquilezles.org/articles/intersectors/`. Accessed: 02.04.2024.

[Kim23] Gerald Kimmersdorfer. `https://www.cg.tuwien.ac.at/research/publications/2023/kimmersdorfer-fancyMaps-2023/kimmersdorfer-fancyMaps-2023-report.pdf`, 2023.

[KLG+13] Alexander Kuhn, Norbert Lindow, Tobias Günther, Alexander Wiebel, Holger Theisel, and Hans-Christian Hege. Trajectory Density Projection for Vector Field Visualization. In Mario Hlawitschka and Tino Weinkauf, editors, *EuroVis - Short Papers*. The Eurographics Association, 2013.

[KRW18] Mathias Kanzler, Marc Rautenhaus, and Rüdiger Westermann. A voxel-based rendering pipeline for large 3d line sets. *IEEE Transactions on Visualization and Computer Graphics*, 25, 01 2018.

[OC11] Deron Ohlarik and Patrick Cozzi. A screen-space approach to rendering polylines on terrain. In *ACM SIGGRAPH 2011 Posters*, SIGGRAPH '11, New York, NY, USA, 2011. Association for Computing Machinery.

[ope] `https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames`. Accessed: 28.04.24.

[Paj02] Renato Pajarola. Overview of quadtree based terrain triangulation and visualization. 02 2002.

[PSH21] Juraj Palenik, Thomas Spengler, and Helwig Hauser. Isotrotter: Visually guided empirical modelling of atmospheric convection. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):775–784, 2021.

[SA09] Nadine Schuessler and Kay Axhausen. Processing raw data from global positioning systems without additional information. *Transportation Research Record*, 2105:28–36, 12 2009.

[SGK05] M K H Schneider, Michael Guthe, and R. Klein. Real-time rendering of complex vector data on 3d terrain models. 2005.

[SK07] Martin Schneider and Reinhard Klein. Efficient and accurate rendering of vector data on virtual landscapes. *Journal of WSCG*, 15:59–64, 01 2007.

[str] `https://www.strava.com/maps/global-heatmap`. Accessed: 25.04.2024.

[SWBG06] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus Gross. Gpu-based ray-casting of quadratic surfaces. 01 2006.

[SZT+16] Jiangfeng She, Yang Zhou, Xin Tan, Xingong Li, and Xingchen Guo. A parallelized screen-based method for rendering polylines and polygons on terrain surfaces. *Computers Geosciences*, 99, 10 2016.

[the] `https://thermal.kk7.ch/`. Accessed: 09.05.24.

[top] `https://www.topografix.com/gpx.asp`. Accessed: 12.03.2024.

[tra] `https://trassy.pl/trips/my`. Accessed: 21.04.24.

[ZH21] Meng-Cong Zheng and Yi-Wen Hsu. How 2.5d maps design improve the wayfinding performance and spatial ability of map users. *Informatics*, 8:88, 12 2021.

[ZWD19] Jie Shao Zhaocong Wu, Nan Wang and Guohui Deng. Gpu ray casting method for visualizing 3d pipelines in a virtual globe. *International Journal of Digital Earth*, 12(4):428–441, 2019.