Informatics

Interaktive Visualisierung von Vektordaten auf Höhenfeldern

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Visual Computing

eingereicht von

Silvana Zechmeister, BSc
Matrikelnummer 01327455

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr.techn. Daniel Cornel

Wien, 13. Oktober 2020

Silvana Zechmeister Eduard Gröller

Informatics

Interactive Visualization of Vector Data on Heightfields

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Visual Computing

by

Silvana Zechmeister, BSc
Registration Number 01327455

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing. Dr.techn. Daniel Cornel

Vienna, 13th October, 2020

Silvana Zechmeister Eduard Gröller

Erklärung zur Verfassung der Arbeit

Silvana Zechmeister, BSc

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, 13. Oktober 2020

Silvana Zechmeister

Danksagung

An dieser Stelle möchte ich mich bei allen Menschen in meinem Leben bedanken, die mich während meiner Studienzeit begleitet und diese zu einer unvergesslichen Zeit gemacht haben. Der Austausch und die gemeinsamen Projekte mit meinen Mitstudierenden haben mir immer viel bedeutet, besonders mit Klara und Michaela in der Endphase, in der wir uns auch gegenseitig motiviert und geholfen haben unser Studium abzuschließen. Meinen Freund Johannes möchte ich besonders hervorheben, der mich immer unterstützt und mich in sehr harten Zeiten mit Schokolade versorgt hat. Ein großes Dankeschön geht auch an meine Familie, die mich immer darin bestärkt hat, weiterzumachen und mich mit der Frage über den Zeitpunkt meines Studienabschlusses angetrieben hat.

Ich möchte mich auch bei meinem Betreuer Eduard Gröller für die Beantwortung all meiner Fragen und für das wertvolle Feedback zu dieser Arbeit bedanken. Des Weiteren möchte ich der Integrated Simulations Group des VRVis für das unterstützende Arbeitsumfeld danken und ein besonderes Dankeschön geht dabei an Daniel Cornel, der sein Fachwissen mit mir geteilt und sich immer die Zeit genommen hat, mich von der Anfangsphase über die konkrete Umsetzung bis zur endgültigen Fertigstellung der Arbeit, zu unterstützen.

Danke euch allen!

Diese Diplomarbeit wurde durch das Kompetenzzentrum VRVis ermöglicht. Die VRVis Forschungs-GmbH wird im Rahmen von COMET – Competence Centers for Excellent Technologies (854174) durch BMK, BMDW, Land Steiermark, Steirische Wirtschaftsförderung – SFG und Wirtschaftsagentur Wien – Ein Fonds der Stadt Wien gefördert. Das Programm COMET wird durch die FFG abgewickelt.

Acknowledgements

At this point, I would like to thank all the people in my life who accompanied me during my studies and made it an unforgettable time. The exchange and joint projects with my fellow students have always meant a lot to me, especially with Klara and Michaela in the final phase, in which we also motivated and helped each other to complete our studies. I want especially emphasize my boyfriend Johannes, who always supported me and provided me with chocolate in very hard times. A big “thank you” also goes to my family, who always encouraged me to continue and pushed me with the question about the time of my graduation.

I would also like to thank my supervisor Eduard Gröller for answering all my questions and for the valuable feedback on this work. Furthermore, I want to thank the Integrated Simulations Group of VRVis for the supportive working environment and a special thanks goes to Daniel Cornel, who shared his expertise with me and always took the time to support me, from the initial phase over the concrete implementation to the final completion of the thesis.

Thank you all!

This thesis was enabled by the Competence Centre VRVis. VRVis is funded by BMK, BMDW, Styria, SFG and Vienna Business Agency in the scope of COMET - Competence Centers for Excellent Technologies (854174) which is managed by FFG.

Kurzfassung

Die präzise Visualisierung großer Mengen georeferenzierter Vektordaten auf Höhenfeldern in Echtzeit ist ein häufiges Problem im Bereich von geographischen Informationssystemen (GIS). Vektordaten bestehen in der Regel aus Linien und Polygonen, die Objekte wie Straßen, Flüsse, Gebäude und Parks darstellen. Die interaktive Erkundung dieser Vektorobjekte in weitläufigen virtuellen 3D Umgebungen und der daraus resultierende große Zoombereich stellen eine zusätzliche Leistungsherausforderung für deren Visualisierung dar. In solch weitläufigen Umgebungen ist es schwierig, eine klare Sichtbarkeit aller Objekte von Interesse sowohl in der Gesamtübersicht als auch ihrer Details in Nahansichten zu gewährleisten.

In dieser Arbeit wird eine bildschirmbasierte Visualisierungsmethode für Vektordaten vorgestellt, die zwei verschiedene Ansätze kombiniert, einen statischen und einen dynamischen Ansatz, um das Verhalten und die Sichtbarkeit der entsprechenden Vektorobjekte kontrollieren zu können. Die Vektordaten können Objekte aus der realen Welt darstellen, und um ihre relative Größe zum Rest der 3D Szene zu erhalten, wird für den statischen Ansatz eine konstante Objektgröße verwendet. Dieses statische Verhalten kann jedoch dazu führen, dass Vektorobjekte beim Herauszoomen verschwinden. Da Linien aufgrund ihrer geringen Breite besonders betroffen sind, werden sie beim dynamischen Ansatz entsprechend der aktuellen Ansicht skaliert, um auch aus der Ferne gut sichtbar zu sein.

Die Evaluierungsergebnisse zeigen, dass beide bildschirmbasierten Visualisierungsansätze in realen Anwendungsfällen eines raumbezogenen Entscheidungsunterstützungssystems mit weitläufigen Umgebungen und Vektordaten, die aus mehreren Millionen von Eckpunkten bestehen, angewendet werden können und dennoch eine Echtzeitleistung bieten. Die Ergebnisse zeigen auch, dass die vorgeschlagene bildschirmbasierte Visualisierungsmethode im Vergleich zu einer volumenbasierten Visualisierung einen größeren Render-Overhead erzeugt, aber bei großen Vektordatensätzen die neue Methode diese übertrifft.

Abstract

The accurate visualization of huge amounts of georeferenced vector data on heightfields in real-time is a common problem in the field of geographic information systems (GIS). Vector data usually consist of lines and polygons, which represent objects such as roads, rivers, buildings, and parks. The interactive exploration of these vector entities in large-scale virtual 3D environments and the resulting large zoom range pose an additional performance challenge for their visualization. Ensuring clear visibility of all objects of interest in overview and of their details in close-up views is difficult in such large-scale environments.

In this thesis, a screen-based visualization method of vector data is proposed, which combines two different approaches, a static and a dynamic approach, to control the behavior and the visibility of the corresponding vector entities. The vector data can represent real-world objects and to preserve their relative size to the rest of the 3D world, a constant object size is used for the static approach. But, this static behavior can cause vector entities to disappear when zooming out. Since lines are especially affected due to their small width, the dynamic approach scales them according to the current view in order to be clearly visible even from far away.

The evaluation results show that both screen-based visualization approaches can be applied in real-world use cases of a geospatial decision support system with large-scale environments and vector data consisting of several millions of vertices and still provide real-time performance. The results also highlight that the proposed screen-based visualization method produces larger render overheads compared with a volume-based visualization, but for large vector data sets, the new method outperforms it.

Contents

Acronyms

AABB axis-aligned bounding box 22, 34, 43, 46, 47, 49, 50, 61, 62, 65

BVH bounding volume hierarchy 13, 14, 22–25, 34, 36, 43, 44, 49, 50, 52, 60–62, 64, 65, 80–82, 84, 87, 89, 92, 93, 97, 100, 102

CPU Central Processing Unit 41–43, 45–47, 49, 51, 52, 56, 80, 81, 89

FOV field of view 54

FXAA fast approximate anti-aliasing 17, 37, 66, 85, 87, 97

GB Gigabyte 78, 79

GHz Gigahertz 79

GIS geographic information systems 1, 3, 5–8, 23

GPU Graphics Processing Unit 41–43, 50–52, 78–83, 86, 88–94, 96–100, 102

HORA Natural Hazard Overview & Risk Assessment Austria 71, 73

LOD level of detail 7, 8, 17

MB Megabyte 93

min minute 78

MLAA morphological anti-aliasing 17

ms millisecond 78, 85, 87, 93, 95

MSAA multi-sampling anti-aliasing 15, 36, 37, 66, 79, 85, 87, 99, 102

RAM Random-Access Memory 79

SSAA super-sampling anti-aliasing 15, 66, 85

STL Standard Template Library 49

TAA temporal anti-aliasing 17

CHAPTER 1

Introduction

1.1 Motivation

Natural disasters cause a lot of damage and can hardly be controlled, so it is desirable to be prepared as well as possible for these unavoidable events. Floods are the natural disaster that involves the most people worldwide and affects more than two billion people [WH18]. In addition to the danger flood disasters pose to people, they also lead to high costs and an economic loss of US $ 656 billion. One way to be better prepared for such disasters is the use of decision support systems for flood management [KJ19]. These systems enable the analysis of diverse crisis scenarios and the computation of possible outcomes with different measures. This way broad evaluations can be made in advance, which should make it easier for flood managers to make reasonable decisions and better prepare for a real disaster situation.

The visualization of spatial data is important for flood management to communicate location-relevant information, such as the positions of flooded areas and their proximity to buildings and roads to identify and locate potential risk areas. There are different types of spatial data that are important in this context, the general and the domain-specific geographical data and data provided by engineers to define, for example, building footprints or protection walls. The general geographical data describe the landscape, for example with land use data and the infrastructure, with road, rail and sewer networks. The domain-specific geographical data include flood and catchment areas to support visual analysis for flood management. In the field of geographic information systems (GIS) all these different data types are commonly stored and provided by vector data, which describe spatial objects by using two-dimensional points, lines, and polygons [SZT$^+$16, TBP17].

There is a shift from 2D virtual environments towards 3D and one reason for this is the omnipresence of 3D scenes in games and other media [Pre15]. This trend is also noticeable in the GIS sector, where traditional 2D landscape representations are sometimes replaced

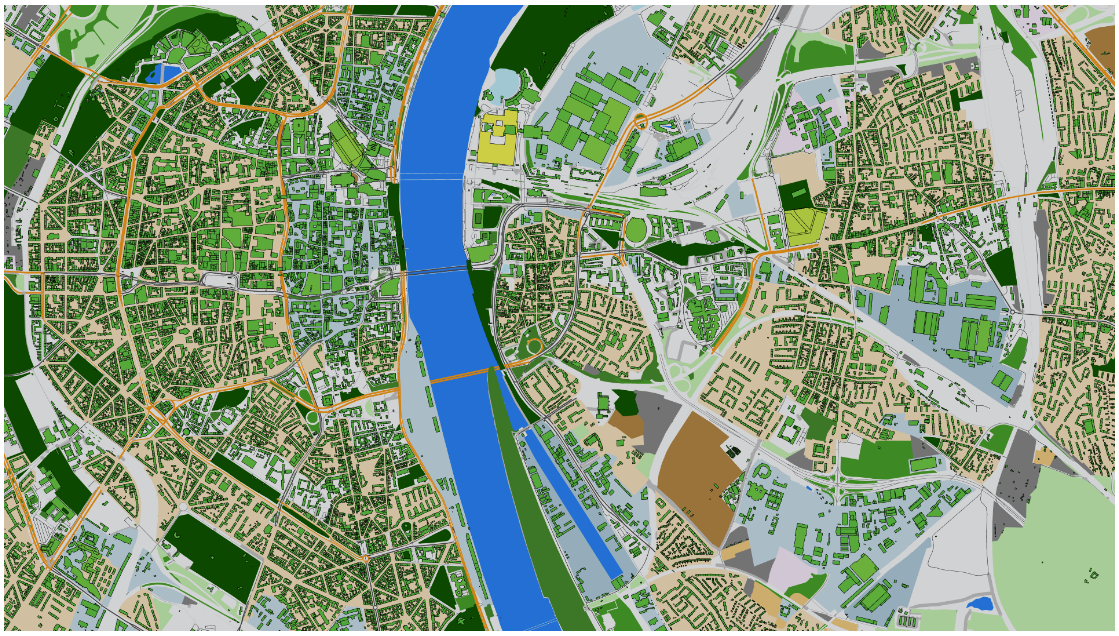


Figure 1.1: Visualization of various vector data representing land use, building footprints, roads and water bodies in Cologne.

or extended by 3D representations. An advantage of 3D scenes is that they are more realistic and therefore it is intuitively understandable even for non-domain experts when georeferenced data are displayed in their natural 3D shape. Furthermore, the user study of Leskens et al. [LKT^{+}17] shows the importance of the third dimension, especially in the field of flood analysis. This is because the impact of flood hazards can also be shown from the side and the water height and its damage can be directly visualized. Compared to 2D visualizations where these data are displayed with abstract color coding, the direct 3D visualizations allow the user to interpret flooded scenes intuitively. The participants of their user study were able to better understand the consequences of a flood in terms of damages, loss of life, and the urgency to evacuate.

The downside of using the third dimension is that occlusions and perspective distortions can occur. Furthermore, the navigation through a 3D virtual world is more complex than in 2D. The additional dimension also introduces a conflict between the 2D vector data and the 3D scene and a mapping from the vector data onto the underlying terrain is necessary to resolve it. Previous research led to different mapping approaches, which have their limitations when it comes to visualizing vector data for dynamic flood management systems. This thesis tries to overcome these limitations and focuses on the optimization of vector data visualization in the context of the geospatial decision support system for flood management called *Visdom* [vis]. In Figure 1.1 a scene in *Visdom* is shown where vector data are used to visualize land use, building footprints, roads, and water bodies of Cologne.

1.2 Problem Statement

There are different challenges that come with the visualization of geospatial big data for decision support systems. Despite the large amount of data, interactivity should be provided to enable the user to effectively direct the system and explore different scenarios from variable views. To accomplish this interactive behavior, a highly flexible visualization of vector data that is suitable for different perspectives and zoom levels is necessary. This interactivity combined with a large-scale environment can make the visualization of large amounts of vector data a performance bottleneck.

A way to reduce the complexity of vector data are simplification algorithms, which eliminate unnecessary geometric detail. Line simplification has a particularly important role to reduce the geometric complexity of vector data and thus accelerate the visualization process. The challenge is that the simplification needs to reduce the size of vector data while it should preserve the visual impression of the original data. This means that for lines further away more data can be reduced without noticeable changes in their appearance, while near lines need more details and less simplification can be applied. To do so, the simplification process has to dynamically adapt to different views. This dynamic simplification process is not trivial in real time, which leads to a lack of these techniques that provide both efficiency and accuracy.

The three-dimensionality, which is beneficial for flood analysis also complicates a fast visualization. The advantage of an intuitive interpretation and an improved spatial impression of a scene, comes along with occlusions, perspective distortions, and a generally higher complexity due to the additional dimension. Furthermore, it introduces a dimensional conflict between 2D vector data and the 3D environment. To integrate the vector data they have to be mapped onto the heightfield of their underlying terrain. Existing techniques have problems if they are used to map large amounts of vector data in dynamic real-time applications with wide scenes. The limitations include the occurrence of visual artifacts and the need to generate additional geometries and complex data structures, which costs time and memory. The techniques also have limited flexibility, as some are only designed for specific use cases and they are not intended for use in applications with vector entities that are interactively manipulated and dynamically changing. One application for dynamically changing vector entities are large-scale environments that cause objects far away to disappear, even if they are important and the user wants to interact with them. A dynamic scaling of the vector entities according to the current view would be desirable in such cases, to be able to perceive them at every distance.

1.3 Research Question

This thesis tries to answer the question of the feasibility of visualizing vector data under challenging conditions required for interactive GIS applications. The main question is how to display a million of partially dynamic lines and polygons integrated into a large-scale

3D environment in real time with pixel accuracy and without visual artifacts. To answer this question, different optimization algorithms for line and polygon visualization need to be applied and analyzed for their speed and accuracy. In this way it should be determined whether both a real-time performance and a high accuracy can be achieved at the same time.

1.4 Contributions

In this thesis a new vector data visualization method is introduced that provides interactivity even for large amounts of vector data that are visualized in large-scale environments. Line and polygon data can be rendered with interactive frame rates in challenging use cases while providing pixel accuracy. The feasibility and efficiency is proven by case studies, where vector data are visualized in real world use cases. The proposed method provides two ways to render vector lines, a static and a dynamic variant, which have both their strengths and limitations. Static lines are fast and memory-efficient, but there are applications where a dynamic scaling of lines to remain visible has a higher priority. Depending on the application, a different line type can be chosen to obtain line visualizations that meet individual requirements. Additionally, different polygon and line styles offer the possibility to change the data representation to highlight and differentiate vector entities.

1.5 Structure of the Thesis

The work related to this thesis and domain-specific terminology are discussed in the next Chapter 2, *Related Work*. Then the whole process of visualizing vector data, from the data input to the final display of different decal types is described in Chapter 3, *Visualization Process*. The following Chapter 4, *Implementation Details*, covers details of the concrete implementation and limitations of the two screen-based visualization approaches for static decals and dynamic lines. Afterwards, the strengths and weaknesses of the proposed visualization method are analyzed in real-world use cases of a flood management system and compared with a volume-based visualization technique. In Chapter 5, *Evaluation*, also the results of performance tests on time and memory consumption are presented and different influence factors are discussed. The final Chapter 6, *Summary*, concludes the thesis by summarizing the main contributions of the thesis and giving an overview of open topics for future work.

CHAPTER 2

Related Work

The chapter covers the work that is related to this thesis and addresses different aspects that are relevant to visualize vector data. In order to get clarity in the use of different terms, domain-specific terminology is discussed first. Different vector data visualization techniques that map 2D vector data onto 3D geometry are introduced afterwards. These techniques can be divided into four main categories, i.e., texture-based, geometry-based, volume-based, and screen-based techniques, which are covered in the respective sections. They are followed by a short overview of different anti-aliasing methods to prevent visual artifacts on the displayed vector data. Afterwards, existing algorithms to reduce the data complexity by simplifying lines are discussed. The last section of this chapter covers common styling methods used to visualize vector data.

2.1 Terminology

There are different naming schemes in the field of computer graphics and GIS. The projection of 2D structures onto 3D surfaces is usually called *decal rendering* or *decaling* in computer graphics literature because the process is like attaching a decal to a surface [AHH+18]. Decals can represent versatile things and therefore they are used in different application areas. In computer graphics decal rendering is known as the projection of a 2D decal texture onto 3D surfaces to represent fine structures, such as eyes, fur, and skin structures as shown in Figure 2.1 [DWB+13].

Decal rendering is also known under the synonym *draping* in the area of GIS and, as one can see in Figure 2.2, it is used to highlight road networks, rivers, and building footprints for instance [TD19]. In this field, the applied 2D structures are usually described by vector data instead of textures and they can have different shapes, which are not limited to rectangles like textures are. In this thesis decals are consistently defined by vector data and therefore in the next section only visualization techniques using vector data as input are discussed.

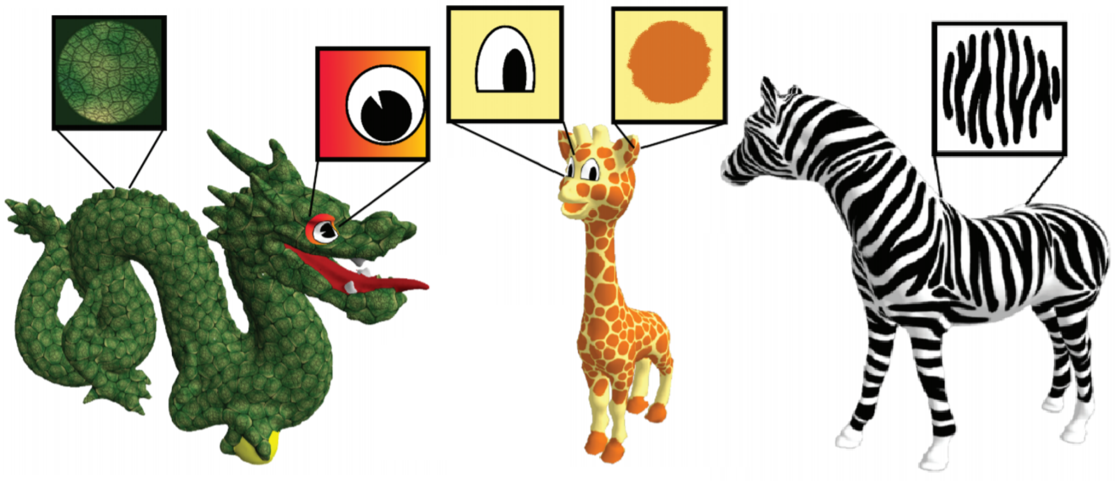


Figure 2.1: Decals based on 2D textures used to apply eyes and different surface structures for skin and fur.

Image source: Groot et al. [DWB+13]

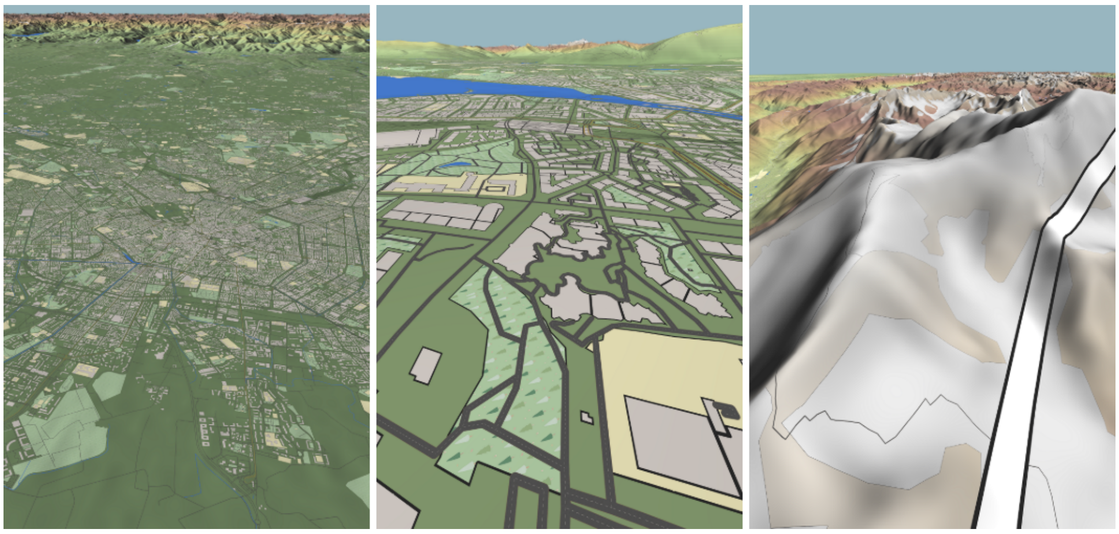


Figure 2.2: Decals based on 2D vector data and used in GIS for representing road networks, rivers, and building footprints on virtual terrains.

Image source: Frasson et al. [FEP18]

2.2 Texture-Based Techniques

For texture-based techniques, textures are generated to represent the given vector-data. Therefore, the vector data are rasterized at a given resolution and stored in a 2D texture beforehand. After rendering the terrain in an offscreen buffer, the decal texture is projected onto it and the result is displayed [AHH+18]. Figure 2.3 illustrates this process and shows

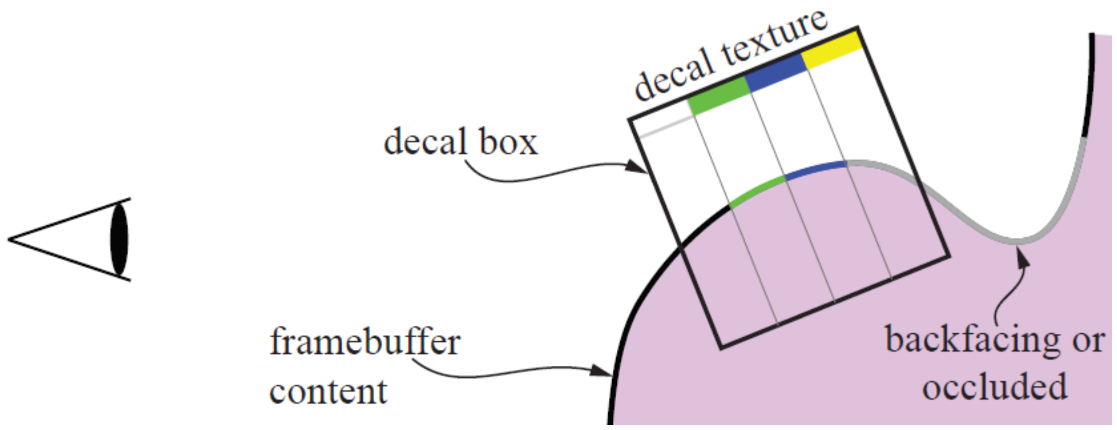


Figure 2.3: The texture mapping principle used to apply 2D decal textures on 3D surfaces.
Image source: Akenine et al. [AHH^{+}18]

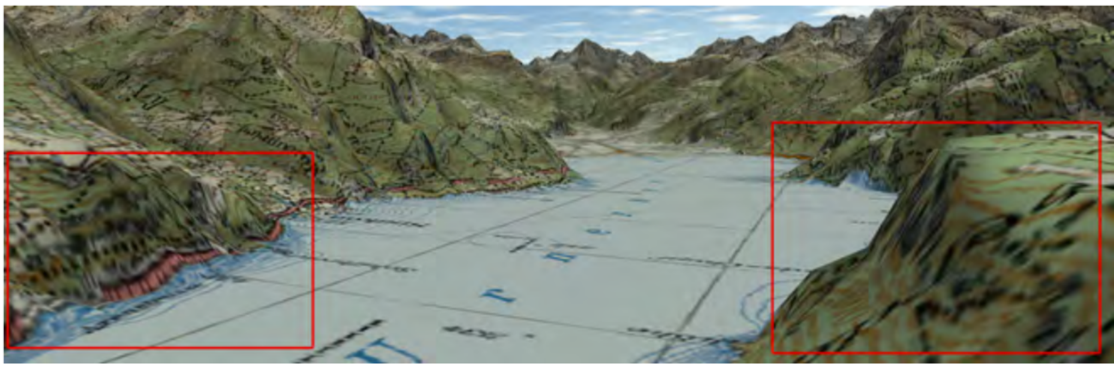


Figure 2.4: Projective distortion on a steep slope (left) and perspective aliasing near the virtual camera (right) are artifacts produced by the texture-based technique.
Image source: Thöny et al. [TBP17]

opportunities for accelerated decal rendering by skipping fully transparent, backfacing, or occluded decals. Through graphics hardware support for texture mapping the approach is fast and easy to implement [SGK05]. A further advantage is the independence of the decals from the underlying geometry. This independence property of decals is especially advantageous in combination with level of detail (LOD) terrain rendering because the dynamically changing terrain geometry does not affect the decals [TD19].

The negative sides of texture-based techniques are that they suffer from projective distortions on steep slopes and perspective aliasing artifacts caused by insufficient resolution near the viewer. Examples of these artifacts can be seen in Figure 2.4 left and right, respectively. To provide appropriate texture resolutions, the vector data can be rasterized in different resolutions to generate a multi-resolution texture pyramid in a preprocessing step [WLB09]. In GIS high flexibility for dynamic user interaction is expected, but this static approach does not support an individual selection, highlighting or manipulation of decals [WKW+03]. Zhi et al. [ZGW+13] avoid the early rasterization of vector data and propose a dynamic texture generation on-demand. That way decals can be dynamically changed and visualized based on the user's needs. This dynamic method is less memory intensive than the static pyramid approach and it can be used to create view-dependent resolutions of textures to reduce perspective aliasing artifacts [SGK05]. Nevertheless, the dynamic updates can be time-consuming and hamper dynamic user interaction.

2.3 Geometry-Based Techniques

The geometry-based techniques subdivide the given vector data based on intersections with the underlying terrain mesh and insert additional vertices. The vertices and the terrain's height information are used to generate 3D geometry-overlays as shown in Figure 2.5 [DXZS13]. These approaches do not scale well with respect to the size and complexity of their corresponding terrain. The number of generated primitives grows with the terrain complexity and is getting high even if the initial vector entities are very simple [SK07].

One advantage over texture-based approaches is that no aliasing or distortion artifacts are created, which is at the cost of independence. The approach is highly dependent on the terrain geometry and the geometry-overlays need to dynamically adapt to the terrain's LOD to avoid artifacts such as z-fighting caused by geometry inconsistencies. For a fast terrain matching Schneider et al. [SGK05] and Qiao et al. [QWS+11] generate multi-resolution geometry overlays based on the terrain's LODs in a preprocessing step. This static data structure provides a direct mapping from terrain LOD to the corresponding decal geometry during runtime, but at the expense of memory consumption and interactive manipulation of decals. Furthermore, the approach is only suitable for terrain rendering with predefined LODs. Sun et al. [SLL08] limited the application of their method to static terrain meshes to avoid geometry changes and to support interactive manipulation, which includes vertex editing, moving, rotating, or resizing of decals. Since the typically large-scale terrains in GIS need efficient LOD rendering techniques for real-time display, the approach is hardly applicable in this area.

To provide a more flexible and dynamic geometry-based technique, Deng et al. [DXZS13] organize the terrain geometry and the vector data with the same quad-tree data structure. The quad-tree makes it possible to skip terrain mesh quads without vector data early in the rendering process. Otherwise, the tree delivers the quad storing the terrain mesh used to produce the tailored decal mesh. They use parallel computing to accelerate the on-demand geometry matching and make it run in real time. But, their per-fragment intersection tests between terrain and decals produce wrong results at the terrain's silhouettes. It leads to a roofing effect with lines drawn along silhouettes in cases where they should lie behind, as shown in Figure 2.6 (left). Ohlarik and Cozzi [OC11] handle this problem by detecting and omitting the silhouette fragments and rendering them with a volume-based technique instead. The resulting line in Figure 2.6 (right) is then

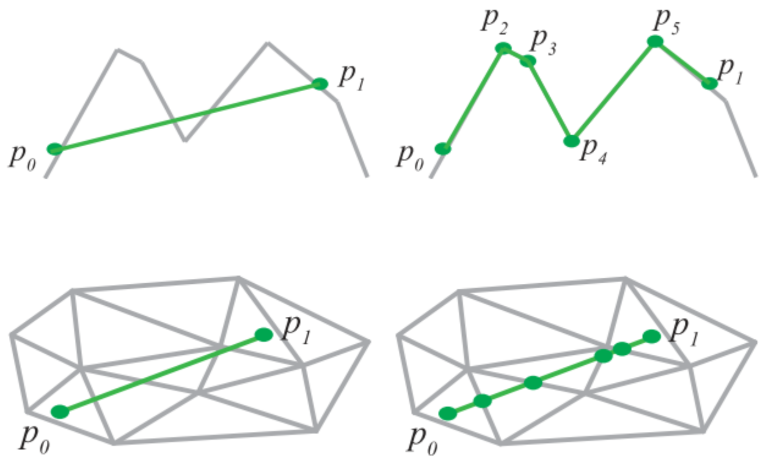


Figure 2.5: Geometry-overlay generation in side view and top-down view: The initial line and it's corresponding terrain mesh (left) and the subdivided line with additional vertices (right).

Image source: Thöny et al. [TBP17]

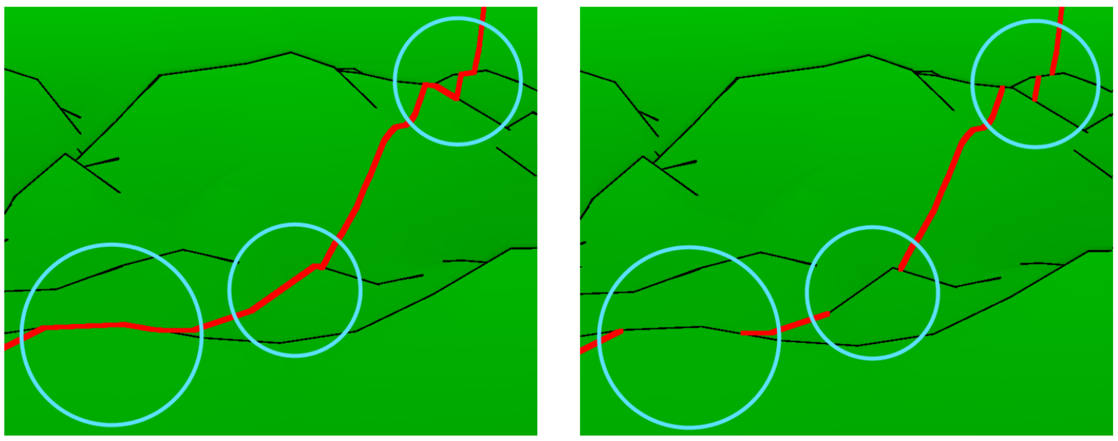


Figure 2.6: Geometry-based line visualization proposed by Ohlarik and Cozzi [OC11] without consideration of the terrain's silhouettes (left) and with (right).

Image source: Ohlarik and Cozzi [OC11]

displayed correctly, but at the cost of runtime caused by the additional render pass.

Another geometry-based technique is proposed by Vaaraniemi et al. [VTW11], which uses only the center lines for the geometry matching between vector lines and terrain. The procedure reduces the computational effort, which speeds up the matching process and facilitates dynamic line changes. This opportunity is used for a view-dependent scaling of lines to enhance their visibility in large-scale environments. Since the lines are matched only by their center lines, the runtime width scaling algorithm for lines may lead to

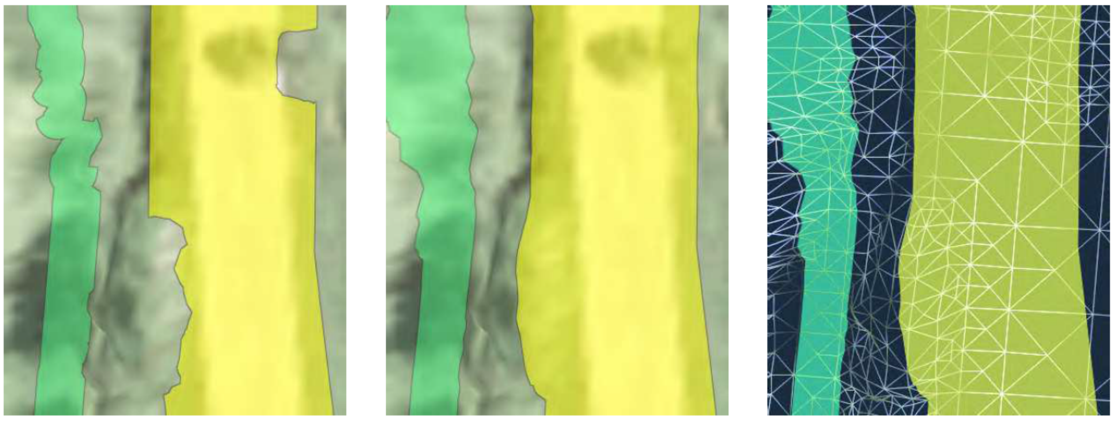


Figure 2.7: A geometry-based technique, using only center lines, causes the vector lines to disappear into the terrain (left). A volume-based technique (middle) prevents this error and is independent of the underlying terrain geometry (right).
Image source: Vaaraniemi et al. [VTW11]

errors on non-planar terrains, because through the scaling they can intersect the terrain and partially disappear. In Figure 2.7 one can see such an error case compared with a volume-based approach. Vaaraniemi et al. [VTW11] implemented also a volume-based variant of the view-dependent line scaling without lines disappearing into the terrain. But, their volume-based approach leads to performance issues for wide scenes and high numbers of scaled vector lines.

2.4 Volume-Based Techniques

Schneider and Klein [SK07] introduced the decal rendering technique based on a stencil shadow volume algorithm, initially used to render shadows [AHH+18]. Therefore, these techniques are often referred to as shadow-volume-based approaches, although they have nothing to do with shadows. The general procedure can be divided into three parts: polyhedra construction, stencil buffer mask generation, and mask application. During the first step, a 3D geometric representation is created by a vertical extrusion of the 2D vector data, which can be done during preprocessing for static vector data. Figure 2.8 shows the polyhedra construction for a blue line with vertices A, B, C and D in side view. The vertices are replicated and shifted to the side according to the line width and above and below the terrain surface to form the extruded 3D geometry. The obtained geometry is then rendered into a stencil buffer by counting the front and back faces to create a decal-terrain intersection mask. Finally, the mask is applied to the scene by coloring pixels accordingly.

The extrusion step leads to a higher primitive count, which can affect rendering efficiency and leads to more expensive geometry updates for dynamically changing vector maps [XSWJ10]. But, these screen-space techniques provide pixel-accurate decal render-

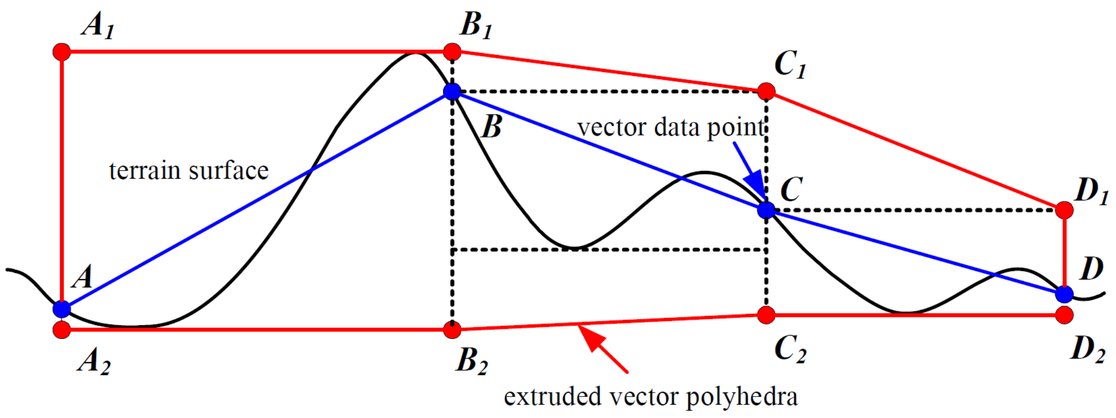


Figure 2.8: Polyhedron generation for volume-based approaches: The blue line is extruded to create the red polyhedron, which intersects the black terrain surface only vertically at the line endpoints A and D.

Image source: Dai et al. [DZY08]

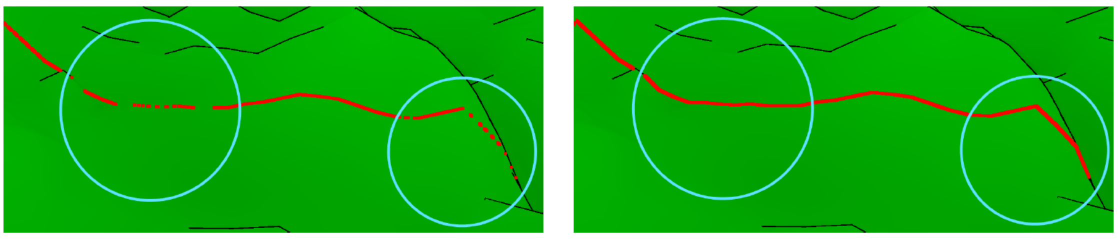


Figure 2.9: Rendering a narrow red line with a volume-based approach leads to dashing artifacts (left) and rendered with a screen-based approach without artifacts (right).

Image source: Ohlarik and Cozzi [OC11]

ing without resolution artifacts as they occur with texture-based approaches. Another advantage of this method is the easy integration into any terrain rendering system caused by the independence of the underlying terrain geometry. Furthermore, the performance depends only on the complexity of the vector data instead of the terrain complexity, as is the case with geometry-based methods [SK07].

The benefits come with higher rendering costs, because there are multiple render passes necessary. After the mask creation, decals can no longer be distinguished from each other and all masked pixels get the same color and transparency. Therefore, decals can only be differentiated if they are grouped by their style and rendered separately. Another drawback of volume-based approaches is that large parts of the screen may be covered by the extruded geometry, which produces a massive overdraw [TBP17]. Visual artifacts can appear in special cases, such as at steep slopes, where the decals are distorted, similar to texture-based techniques [SK07]. Another problem case mentioned by Ohlarik and Cozzi [OC11] are thin lines viewed lengthwise, which can lead to dashing, as at the red

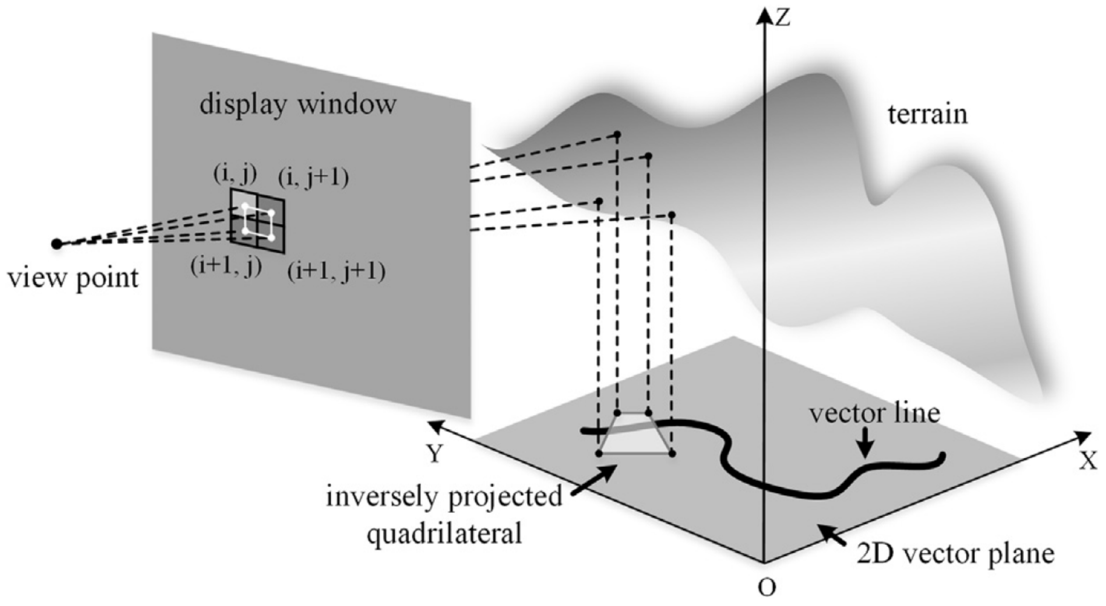


Figure 2.10: For the screen-based technique of She et al. [SZT$^+$16] each pixel is inversely projected to world space by using the height information of a virtual terrain. The resulting pixel quadrilateral is then tested for intersections with the vector data.
Image source: She et al. [SZT$^+$16]

line in Figure 2.9 (left) [OC11]. They use a screen-space approach with vertical walls instead of polyhedra to render the same line without dashing artifacts, as shown in Figure 2.9 (right).

2.5 Screen-Based Techniques

The latest and most promising techniques are the screen-based approaches because they overcome limitations of previous methods, such as static map properties, high terrain dependency, and additional geometry generation. That is the reason why this thesis is based on screen-based techniques. These techniques operate on a per-pixel basis for a direct mapping of the given vector data to corresponding pixels without transforming the data into an intermediate representation, such as rasterized textures, geometry-overlays, or shadow volumes [TBP16].

In Figure 2.10 the basic concept of the screen-based approach of She et al. [SZT$^+$16] is illustrated. Each pixel is inversely projected to world space by using the height information of the terrain. In world space the pixels are tested for intersections with vector data. In case of a hit the pixel is colored according to its intersecting decal. This pixel-precise rendering does not suffer from resolution artifacts such as texture mapping and also prevents artifacts such as z-fighting and dashing. To avoid that for every pixel

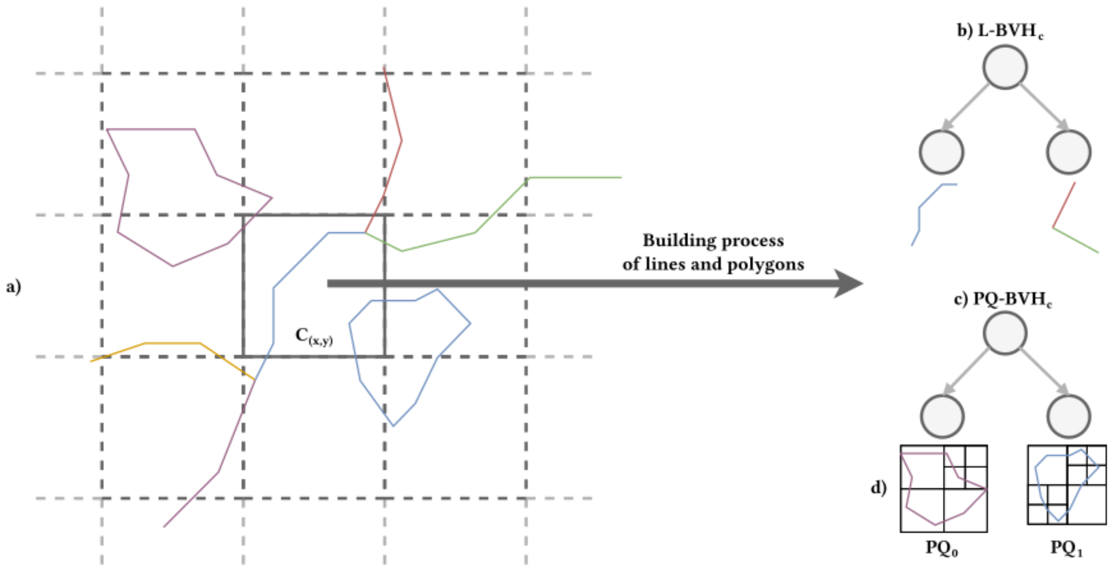
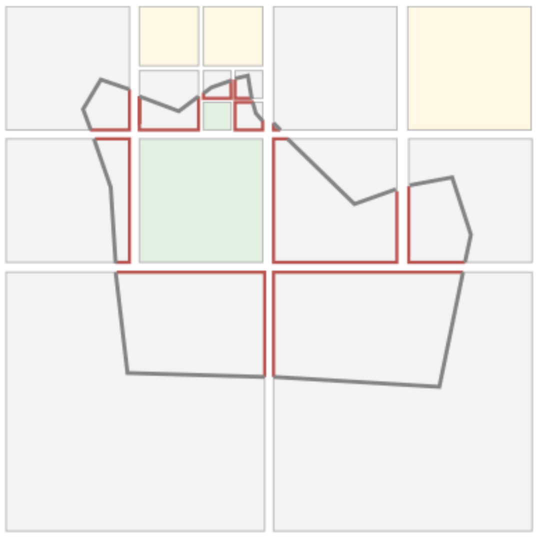


Figure 2.11: Spatial data structures used for the screen-based decal rendering technique of Frasson et al. [FEP18]: (a) A regular grid for spatial indexing of lines and polygons. Per grid cell two bounding volume hierarchys (BVHs) for (b) lines and (c) polygons are stored. (d) Polygons are further organized in quad-trees and stored in the leaf nodes of their BVH.

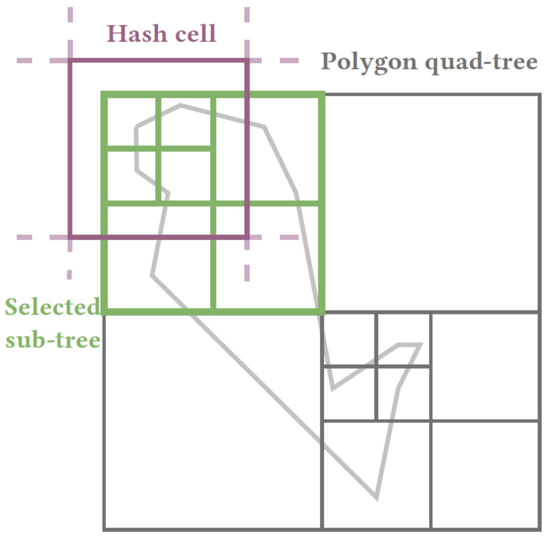
Image source: Frasson et al. [FEP18]

all lines have to be tested for intersection, Thöny et al. [TBP17] accelerate the line search by using spatial data structures constructed during preprocessing. All line segments are assigned to a regular grid as shown in Figure 2.11a. A segment is added to a grid cell if it intersects a cell after adding the corresponding line width. Thus, a line segment is stored at least in one cell but it can be assigned to several cells. To avoid testing many segments per cell, their number should be as small as possible by using a fine grid. Since the memory consumption grows with the grid resolution, it is limited to the available memory and a grid size of 256×256 cells is proposed by Thöny et al. [TBP17]. To keep the per-pixel computation cost low, the line segments are further stored in a fully balanced binary BVH per grid cell, which is sorted according to a space-filling curve (see Figure 2.11b). After this preprocessing step each inversely projected pixel can be located inside the regular grid and the corresponding BVH is traversed by testing if the pixel's distance to a leaf node segment is smaller than half the line width. The pixel color can then be composited according to all detected line segments.

The approach of Thöny et al. [TBP17] is developed for lines only, therefore Frasson et al. [FEP18] extended it to be able to render more complex shapes with polygons as well. They store polygons in quad-trees to achieve fast pixel-in-polygon tests. The tree is build during preprocessing by splitting a quad until only few polygon segments are left to be stored in a leaf node. As shown in Figure 2.12a, the leaf node segments



(a)



(b)

Figure 2.12: (a) Leaf nodes of a polygon quad-tree categorized in fully-in (green), fully-out (yellow) and partial nodes (gray). For point-in-polygon tests per partial node, additional segments (red) are added to close the small leaf polygons. (b) The quad-tree culling process, where the smallest sub-tree that contributes to a grid cell is detected.
Image source: Frasson et al. [FEP18]

(dark gray) are then used to create small leaf node polygons closed on the quad borders with additional segments (red). The resulting quad-tree then consists of three types of nodes, the fully-in, fully-out and partial nodes, which are the green, yellow and light gray nodes, respectively. The created polygon quad-trees are then stored in intersecting cells organized in BVHs the same as line segments are. The data preparation with a regular grid for lines and polygons and a separation of the data into different per-cell BVH can be seen in Figure 2.11. For decal rendering the polygon quad-trees can be efficiently traversed with point-in-rectangle tests between pixels and quad boundaries until a leaf node is reached. To avoid the traversal of whole quad-trees in cases where only a sub-tree contributes to a cell of the regular gird, the root node of this sub-tree is stored per cell. Through this process, called *quad-tree culling* by Frasson et al. [FEP18], the traversal of polygon quad-trees can be limited to relevant sub-trees only. Figure 2.12b visualizes the quad-tree culling process with a quad-tree covering only a small part of a hash cell, which is a cell of the regular grid. If fully-in or fully-out leaf nodes are reached, the contribution of the polygon to a pixel can be determined immediately. For partial leaf nodes a point-in-polygon test has to be executed with the pixel and the prestored leaf polygon.

Spatial search structures, such as those used by Thöny et al. [TBP17] and Frasson et al. [FEP18], accelerate decal rendering but also limit the flexibility. The static structures are not suit-

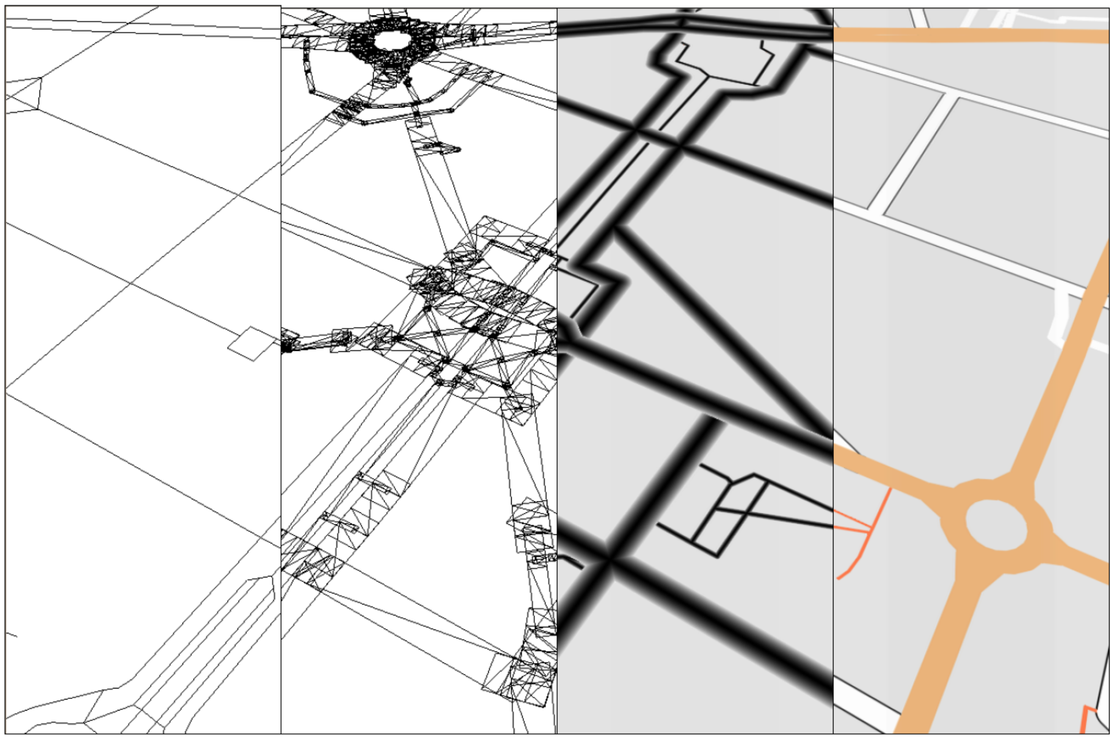


Figure 2.13: A screen-based technique which uses the initial vector lines (left) and their width to generate polygons (second left) and distance field textures per segment (third left). The textures are then used to composite the final decal color (right).
Image source: Trapp et al. [TSD15]

able for decals that are dynamically changing, because it would lead to time consuming updates of the spatial data structures during runtime. The motivation of the use of these static structures is to limit the intersection tests of a pixel to decals nearby, instead of testing against all decals, which quickly becomes inefficient with an increasing number of decals. Trapp et al. [TSD15] propose a more flexible method without using such static structures, which provides view-adaptive rendering of lines, interactive filtering, and highlighting. Therefore, they generate a polygon per line segment by using its width and store a distance field texture per polygon, as shown in Figure 2.13. The pixel color is then composited by using all distance field textures at the pixel location. Thus, only decals contributing to a pixel are processed but the disadvantage of this screen-based technique is that it is limited to planar terrains.

2.6 Anti-Aliasing

The rasterization of decals can lead to aliasing artifacts in the form of jagged edges, as shown in Figure 2.14(left) [AHH+18]. Different anti-aliasing techniques exist to avoid

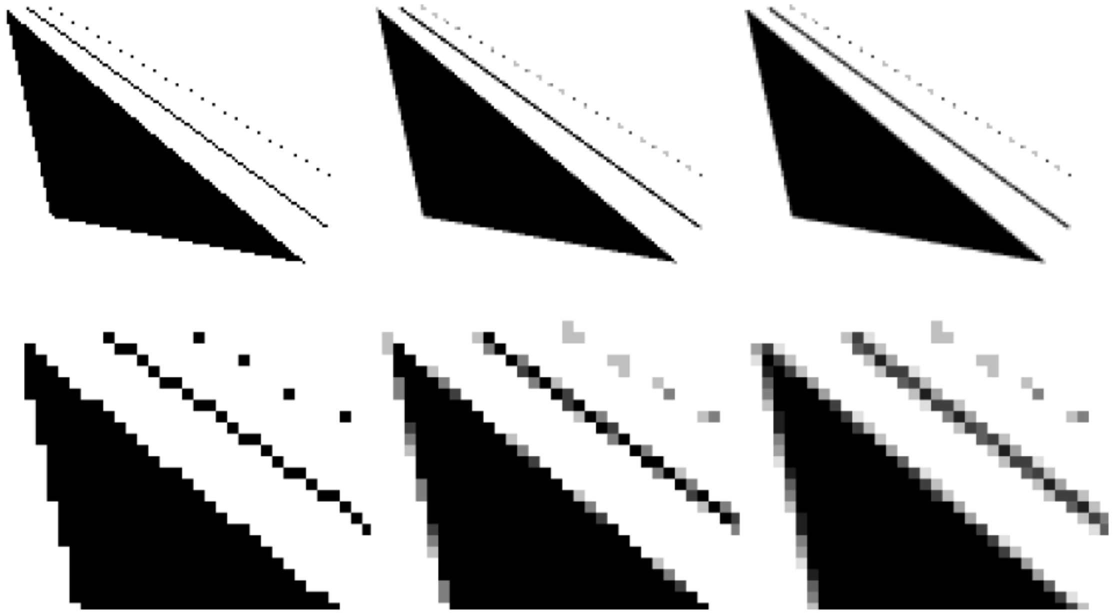


Figure 2.14: Three levels of anti-aliasing of a triangle, a line and points (top) and their magnifications (bottom). Without anti-aliasing by using only one sample (left) and anti-aliasing with four samples (middle) and eight samples (right).

Image source: Akenine et al. [AHH+18]

staircase effects. To get smoother edges super-sampling anti-aliasing (SSAA) and multi-sampling anti-aliasing (MSAA) use multiple samples per pixel and use this gathered information to adjust the pixel color. MSAA is an optimization of SSAA, which does not process all samples. It detects pixels at edges, where anti-aliasing occurs and processes the samples only if necessary. In Figure 2.14 the anti-aliased result with four (middle) and eight (right) samples can be seen. These anti-aliasing techniques have a high memory and bandwidth consumption because multiple samples have to be processed and combined per pixel [CR12].

One way to reduce the anti-aliasing costs is by using analytical methods, such as the prefiltering method proposed by McNamara et al. [MMJ00]. The analytical anti-aliasing concept for lines is based on the distance of a pixel to the center of a line segment and a filter used to smooth the pixel color according to this distance. The radius of the filter should relate to the pixel size to avoid too blurry or too jagged edges [Rou13]. While this information is implicitly given if operating in screen space, for lines embedded into a 3D environment it is not [SLLW18]. Thöny et al. [TBP17] use an analytical anti-aliasing approach for their vector lines, which is based on the estimation of the pixel's coverage. The estimated coverage is used to compute a blending factor applied to the corresponding pixel. Their approach is suitable for lines but can become computationally expensive for polygons [FEP18]. Lines with outlines or with glyphs inside, consist of different colors and therefore need anti-aliasing also inside. She et al. [SLLW18] process the edges individually to determine the resampling region. This region is used to apply a color resampling operation based on Hermite spline interpolation for a smooth transition of colors.

With an image-based anti-aliasing filter, decals can be rendered aliased and filtered afterwards to reduce aliasing artifacts. The anti-aliasing as a post-process is memory-efficient and has low computational cost [CR12]. The idea of morphological anti-aliasing (MLAA) is to detect and smooth only stair-stepped patterns to avoid unnecessary blurring of the scene. A similar technique as MLAA is fast approximate anti-aliasing (FXAA) [fxa], which is optimized and developed by Nvidia. It is fast and easy to integrate in different applications because it relies only on color input and does not need additional information, such as depth and normals [Gre19].

The anti-aliasing filter techniques can be further improved by using temporal anti-aliasing (TAA). It additionally takes samples of the previous frames into account and blends them to obtain the final result [FEP18]. Using the history of frames is not suitable for every application. It cannot be used by systems like *Visdom*, where the change of views does not need to be achieved by continuous navigation. Thus, there is no consistent image flow over sequential frames, which is necessary for TAA to work.

2.7 Line Simplification

Simplification algorithms for lines can be used to reduce the complexity of vector data and therefore accelerate the rendering process. The challenge is that even if the data is reduced, there should not be a noticeable change in the visual appearance of the lines.

Line simplification methods have a long history, going back to the well-known line simplification algorithm proposed by Douglas and Peucker [DP73]. It is based on the selection of a subset of representative line points by omitting points with small distance to the original line. According to Visvalingam and Whyatt [VW93], the Douglas-Peucker algorithm is only suitable for minimal simplification and not for the generalisation of complex lines. Thus, they propose a line simplification method that iteratively eliminates points spanning the triangle with the smallest area. But, both algorithms may lead to unwanted topological changes, such as intersections between lines and self-intersections. Therefore, there exist several methods to prevent topological changes, such as the approach of Mantler and Snoeyink [MS00], which defines safe sets with ε-Voronoi diagrams to avoid that. The method of Shin-Ting and Márquez [SM03] also preserves topological properties of lines by defining star-shaped subsets. The Douglas-Peucker-based approach of Amiraghdam et al. [ADP20] avoids line intersections by detecting points that cause intersections and excludes them from simplification. The computational costs increase with the number of lines and their complexity. To avoid such time-consuming calculations at runtime, Amiraghdam et al. [ADP20] store different LODs of the lines in advance. Then only the distance between a line and the virtual camera has to be calculated to determine the LOD to use for rendering.

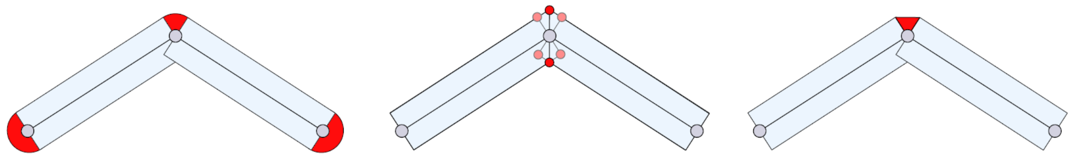


Figure 2.15: Three methods to deal with the gaps between line segments: Adding rounded caps at segment endpoints (left), the connection of corners of adjacent segment quads (middle), and the use of additional triangles (right).
Image source: Vaaraniemi et al. [VTW11]

All these line simplification algorithms use a metric to determine the smallest simplification error in world space. The problem is that the metric is losing its validity in screen space after the perspective transformation of the lines. This leads to an additional error through the world-space-based metric used to simplify lines, which are displayed in screen space.

2.8 Styling Techniques

Vector data can be used to represent various types of objects and to enable the viewer to distinguish them, it is desirable to vary their visual appearance according to the data they represent. Common attributes that are changed per object type are color, line, outline, and filling type to differentiate road and railway lines and park and building footprint areas, for instance. The system of Frasson et al. [FEP18] support dashed, outlined, and textured lines and polygons by storing additional material information. This information is then used for style-dependent fragment shading. It also includes the drawing order to avoid intersection artifacts between overlapping decals caused by distance-based line shading. She et al. [SLLW18] experimented with different line styles to produce variations of solid, gradient, and dashed lines with different symbols. They parameterize each line segment and apply different symbol functions according to the prestored line type. For different line styles, the joints need a special treatment to avoid gaps or broken symbols between segments. Vaaraniemi et al. [VTW11] identify three basic methods to deal with line joints, either rounded caps at all segment endpoints are added or vertices are shifted to connect the quads or the gap between two segment quads is closed by an additional triangle. In Figure 2.15 the basic idea of all three methods is shown. These methods result in different corner styles that are round, pointed, or flattened, which are referred to as round, miter, and bevel corners [ZPYL16]. The round caps have the advantage that they can be analytically computed based on the distance to the segment endpoints without changing vertices or increasing the number of geometric primitives. Furthermore, the caps are able to remove small cracks between lines, which are caused by incomplete line data [ZPYL16].

CHAPTER 3

Visualization Process

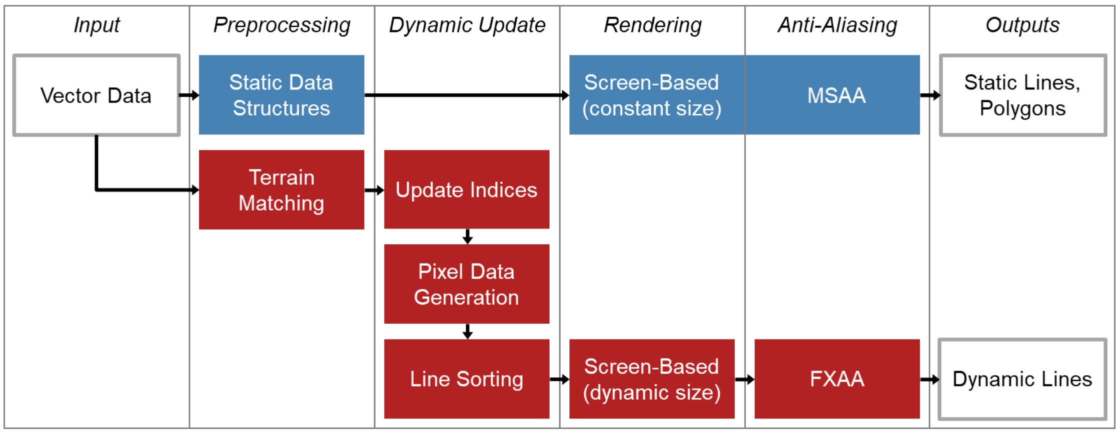


Figure 3.1: Overview of the visualization process of static decals in blue and dynamic lines in red.

The whole process of visualizing vector data, from the data input to the final display of different decal types is described in this chapter. Figure 3.1 shows an overview over the main steps that are executed to visualize static decals, which are static polygons and lines and the main steps for the visualization of dynamic lines. The different decal types and the individual steps of their screen-based visualization processes are described in more detail in the respective sections.

3.1 Decal Types

In this thesis, 2D vector data are used to visualize three different types of decals: static polygons, static lines, and dynamic lines. The lines can belong to polygons and represent their outlines or they are separate lines.

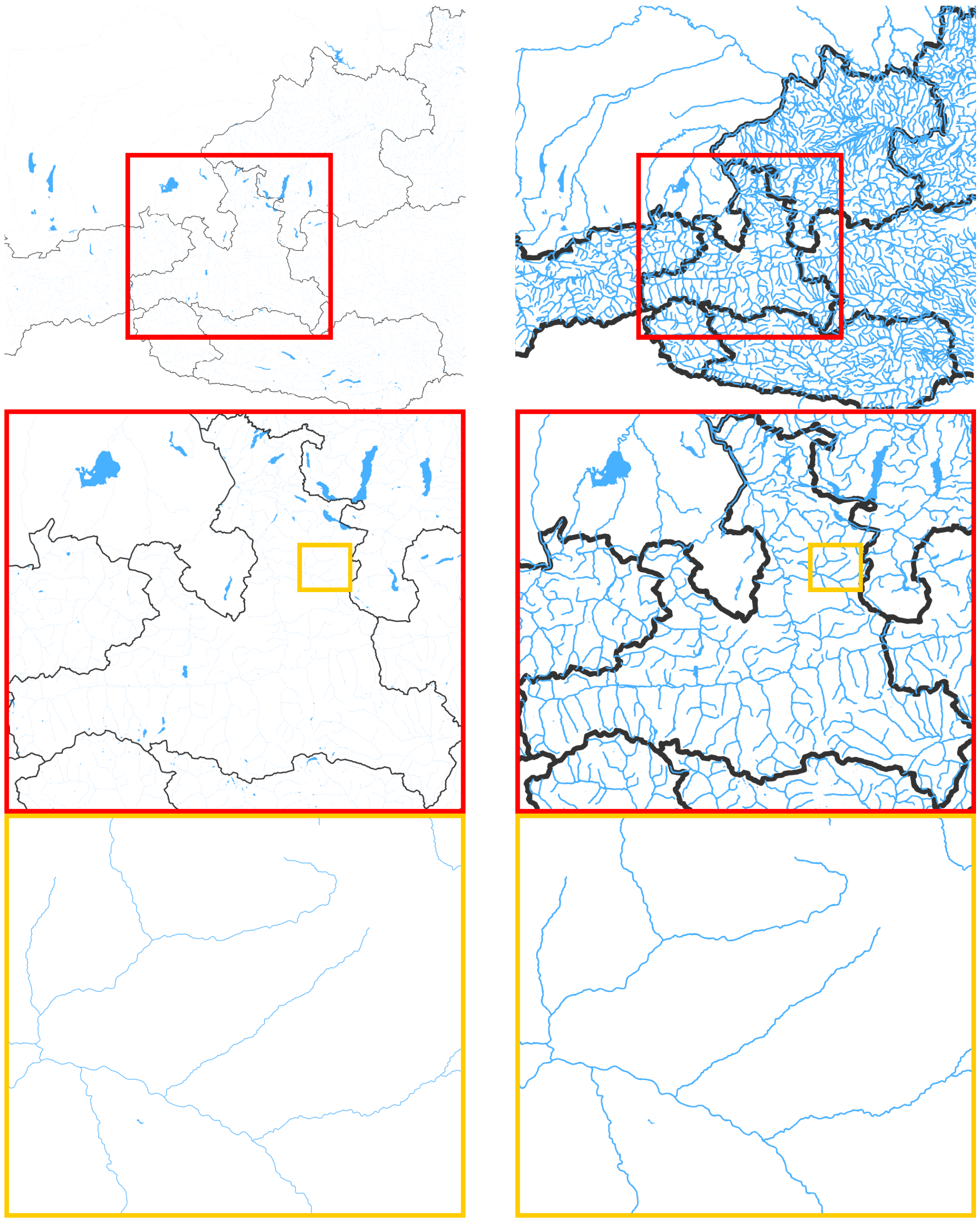


Figure 3.2: Comparison of three different zoom levels of blue river lines and black borders in Salzburg represented by (a) static lines and (b) dynamic lines whereby the static lines have a constant size in world space and the dynamic lines in screen space.

All decal types are embedded into the 3D environment whereby the static decals have a constant size in world space. The decals can cover several square kilometers and are able to display such large objects on the screen. A virtual camera with perspective transformation can be used, which projects the world-space decals onto the image plane of the screen. The decal size in screen space is determined by two factors, the decal size in world space and the distance between decal and camera. Since the decals represent real-world objects with a fixed world-space size, they are getting smaller and larger on the screen under perspective transformation, depending on the zoom level. The optimal size of decals on the screen after the perspective transformation is given only in a small distance range of the camera to the decal. It is desired that all objects of interest are clearly visible and outside the optimal distance range the decals are getting too large, when zooming in, or too small, when zooming out. This is a problem for large-scale environments where both, an overview of all objects and detailed views of individual objects are relevant. Figure 3.2a shows an example of river lines and borders in Salzburg, which are not or only barely visible in overview and become slowly visible when zooming in. By increasing the line width in world space, the static lines can be made visible in overview, but in close-up views, the lines would become too large and their details would no longer be perceptible.

To enable the visibility of lines in overviews and close-up views of such large-scale scenes, the lines are dynamically scaled according to the current view. This dynamic scaling of lines is shown in Figure 3.2b, where the river lines are clearly visible in overview and after zooming in, one can still see the detailed courses of the rivers. These dynamic lines have a constant size on the screen, but since they also have to be embedded into the 3D environment, their visualization is particularly demanding. It is easy to render 2D lines with a constant size in screen space by projecting the lines directly onto the image plane. But, the 2D decal lines have to lie on the 3D surface of a terrain and should be projected onto this surface instead. The integration of the 2D lines into the 3D world has to be done in world space. To still get lines with a constant screen-space size, the line size is fixed in screen space and then the lines are transformed back into world space with the inverse perspective transformation to embed them into the 3D scene. As a result, the world-space size of the initial lines implicitly changes when zooming and varying the distance between lines and camera. This dynamic change of the world-space size of the lines leads to a view-dependent scaling of the area on the terrain that is covered by the lines. Therefore, the lines have to be continuously re-embedded into the 3D scene and a data preparation in advance is only feasible to a limited extent. Thus, the line data are prepared according to the current view in dynamic updates, which are executed each time before the lines are rendered.

Figure 3.1 shows the main steps executed during this dynamic update process, which are only performed for dynamic lines before rendering. Since the initial size of static decals does not change, their data can be completely prepared during preprocessing. The further steps of the static and dynamic visualization process also differ due to the different data preparation and the different zoom behavior.

3.2 Preprocessing

During preprocessing different data structures are generated based on the initial vector data. The data structures provide a more efficient access to the vector data and are used to accelerate the render process. In Figure 3.1 one can see that the generation of the static data structures for static decals and the matching of the dynamic lines with the terrain are executed during preprocessing. All further data preparations of the dynamic lines are done during dynamic updates.

3.2.1 Static Data Structures

The basic concepts of the screen-based visualization techniques of Thöny et al. [TBP17] and Frasson et al. [FEP18] to visualize lines and polygons, are already described in the previous chapter in Section 2.5. The concepts are also used for the vector data visualization method of this thesis to render static decals. The same spatial data structures are generated, which includes a regular grid, bounding volume hierarchies (BVH) for lines and polygons per grid cell and polygon quad-trees, shown in Figures 2.11 and 2.12. These data structures are used during rendering to speed up the determination if the world-space position of a pixel is covered by a static decal.

Decal Grid

The regular grid divides the spatial domain covered by static decals into small cells and the grid cells refer only to lines and polygons that are inside the cell or intersect it. Before the lines are assigned to the decal grid, their individual segments are extended to rectangles according to the predefined line width to perform a correct intersection between the segments and the grid cells. To avoid the assignment of whole polygons and lines, polygons are divided into smaller parts by quad-trees and only intersecting tree nodes and intersecting segments of a line are assigned to cells. During rendering, this grid limits the number of line and polygon parts that possibly contribute to a given pixel to those located in the same cell as the pixel's world-space position. Thöny et al. [TBP17] propose a grid resolution of 256×256 cells and Frasson et al. [FEP18] 1000×1000 cells. A higher resolution has the advantage that the decals are divided into more grid cells, which leads to less decals stored per cell. This is beneficial for rendering because less decals need to be tested for pixel coverage. But, a higher decal grid resolution also leads to a higher number of cells that have to be processed during preprocessing, causing a higher time and memory consumption due to more per-cell data.

Bounding Volume Hierarchies

In order to efficiently search through all line segments and polygon parts of a cell during runtime, they are organized in separate BVHs. The per-cell BVHs are tree structures that facilitate skipping whole sub-trees with several decals at once, if a pixel lies outside the common coverage area of the decals. The common areas are defined per tree node by 2D axis-aligned bounding boxs (AABBs), which are boxes that enclose all decal parts assigned to the sub-tree of the node. In the leaf nodes, the line BVHs refer to line segments and the polygon BVHs refer to other tree-based data structures, namely polygon quad-trees.

The order in which the decal parts are stored in the BVH determines their drawing order. For a consistent drawing order of lines and polygons over all decal grid cells, every line and polygon gets a different rank in the drawing order of decals. The individual parts of a cell are sorted according to the order of the corresponding lines or polygons before they are assigned to leaf nodes of a BVH. Therefore, different sorting criteria can be used, such as the line width, the line length, or the polygon area. Also the importance of the objects corresponding to the decals can be used to define the drawing order, for example if lines represent roads, the main roads are more important and should be drawn on top. By default the line lengths and polygon areas are used as sorting criteria for lines and polygons, respectively, because long lines and large polygons are usually more important.

Polygon Quad-Trees

The leaf nodes of polygon BVHs do not refer to whole polygons but to polygon quad-trees. The quad-trees are used to divide polygons into smaller parts with reduced complexity and only the part in which a pixel is located has to be processed at runtime. The individual polygon parts are stored in the leaf nodes of the quad-tree. Such a leaf node is either fully inside, fully outside or partially inside the corresponding polygon. The different leaf node types are already defined by Frasson et al. [FEP18] and presented in Chapter 2, *Related Work*, in Figure 2.12a. The leaf node types fully inside or fully outside facilitate a fast classification of a pixel status during rendering. Only for the partially inside leaf nodes additional point-in-polygon tests have to be executed between the pixel and the polygon part stored inside the leaf node, to determine if the pixel is inside the polygon.

To reduce the traversal time of the polygon quad-trees during rendering, the quad-tree culling process proposed by Frasson et al. [FEP18], presented in Section 2.5, and visualized in Figure 2.12b, is also applied. All quad-trees that are intersecting a decal grid cell are further analyzed by searching for the sub-tree that actually contributes to the grid cell, which is referred to as hash cell in Figure 2.12b. Only this sub-tree is assigned to the cell and the rest of the quad-tree can be omitted during rendering.

Interior Polygons

In addition to the static decal types that are used by Thöny et al. [TBP17] and Frasson et al. [FEP18], the proposed visualization method also includes interior polygons. These are polygons inside other polygons. Interior polygons are elements that often appear in GIS data, for example to represent inner courtyards of buildings. To display such data correctly, it is necessary to be able to visualize interior polygons appropriately. The outer polygons are referred to as exterior polygons. One interior polygon is assigned to exactly one exterior polygon but one exterior polygon can have multiple interior

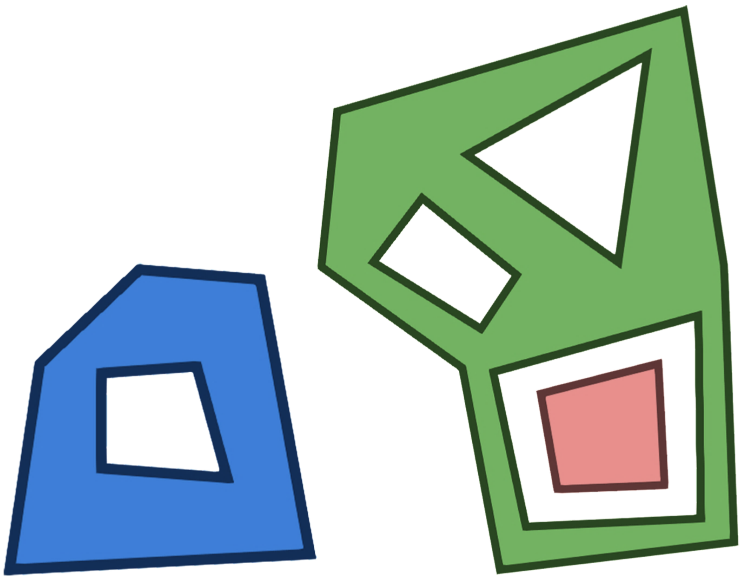


Figure 3.3: Two exterior polygons in blue and green with four interior polygons that lead to cutouts and one nested interior polygon in red.

polygons that may also be nested multiple times. Figure 3.3 shows two examples of exterior polygons in blue and green with multiple interior polygons. The blue polygon contains only one interior polygon, which leads to a cutout. The green polygon contains four interior polygons, whereby three also lead to cutouts and one of them is nested and produces an additional red polygon inside.

Interior polygons are also divided into smaller parts and stored in quad-trees, such as exterior polygons, but interior polygons are not organized in BVHs. They are only assigned to their corresponding exterior polygons and to their intersecting decal grid cells. The resulting data structures for polygons are shown in Figure 3.4. One can see that both exterior polygons, the blue and the green one, intersect the selected decal grid cell and are therefore assigned to the polygon BVH of this cell. Only the first interior polygon of the blue exterior polygon and the second interior polygon of the green exterior polygon intersect the cell and are referenced by it. During rendering, the interior polygons can be limited to those that belong to an already detected exterior polygon and to a certain grid cell.

3.2.2 Terrain Matching

The preparation of vector data for an efficient rendering of dynamic lines is limited due to their view-dependency and the resulting dynamic behavior. It is not possible to assign dynamic lines to intersecting cells of a static regular grid, as it is done for the static decals, because the lines can be scaled arbitrarily and the areas covered by them vary. Thöny et al. [TBP17] define a maximum line width to assign the lines to all covered cells

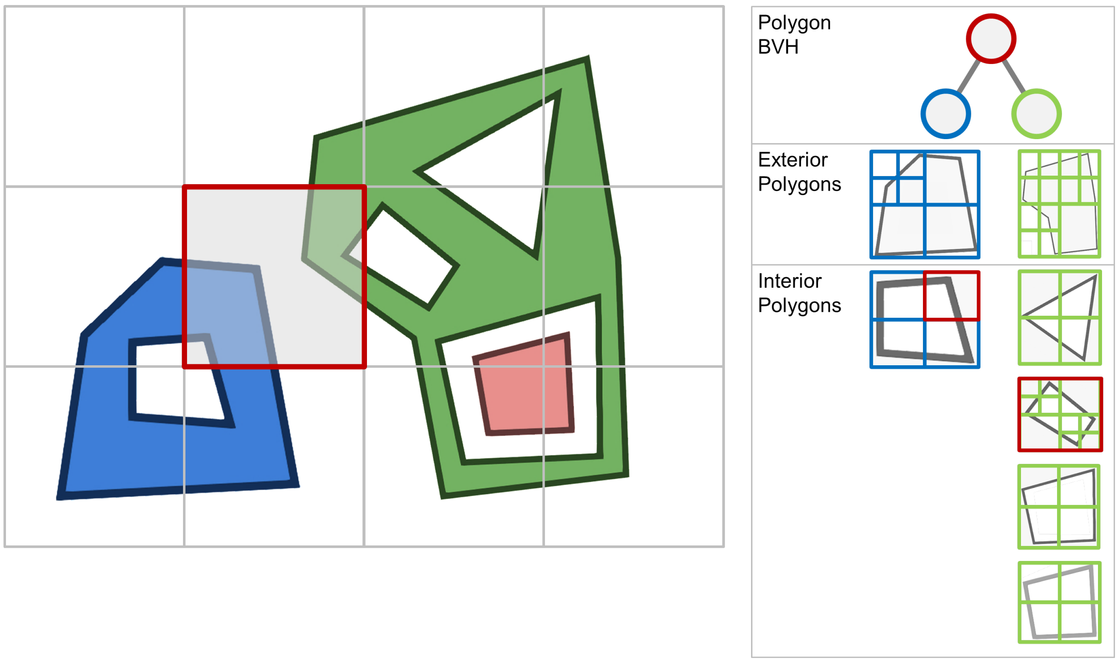


Figure 3.4: The selected decal grid cell (red) refers to a polygon BVH containing the intersecting blue and green exterior polygons. They refer further to their interior polygons whereby only the first interior polygon of the blue and the second of the green exterior polygon are contained in and assigned to the detected grid cell.

with this maximum width. The use of this approach would introduce an unclear user parameter that effectively restricts the dynamic line scaling behavior to a certain zoom range.

To still be able to limit the number of lines during rendering, 3D line segment boxes are generated during the dynamic update process that enclose the whole terrain area that can be covered by their corresponding line segments. This rough determination of the affected area limits the search space for lines during rendering only to the box area. Thus, the number of lines that have to be processed per pixel is reduced, making the render process more efficient. To make the segment boxes cover the affected terrain area, they must be positioned on the terrain surface onto which the lines are projected afterwards. Since the segment boxes cannot be precalculated, because their size depends on the dynamically changing line width, only the corresponding line segments can be positioned on the terrain surface.

Therefore, line segments are matched to the terrain surface by subdividing the 2D segments and locating the newly created line vertices on the 3D terrain surface, as shown in Figure 3.5. The terrain matching is also a typical part of geometry-based visualization techniques, but the difference is that for the dynamic lines only the center lines are used and not the static contours of a line with predefined width. The pixel-based

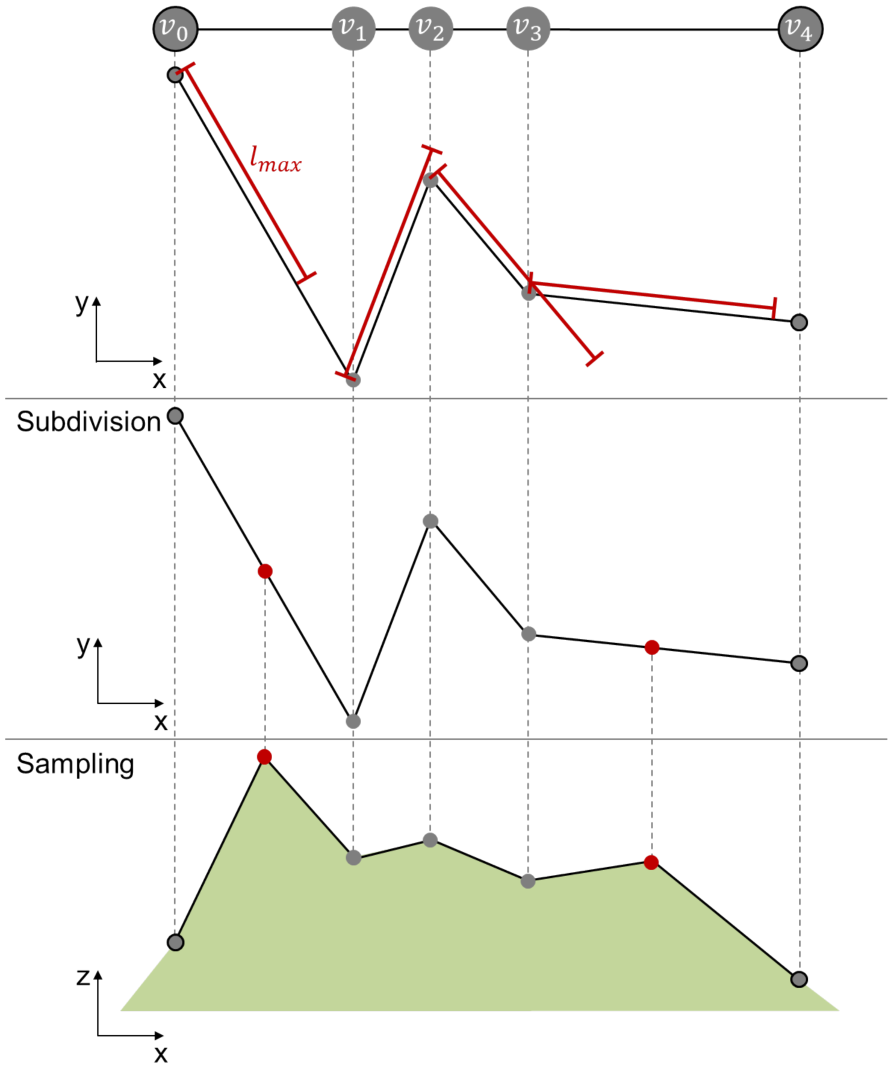


Figure 3.5: The terrain matching process takes the initial 2D line segments, which are defined by vertices v_{0-4} and subdivides the first and last segment because their length is larger than a predefined maximum segment length l_{max}. The initial 2D vertices (gray) and the new 2D vertices (red) are then sampled on the terrain's heightfield to determine the height and the position of the vertices in the 3D environment.

focus of the visualization approach of the dynamic lines is characteristic for screen-based visualization techniques and thus it is rather assigned to them and not to the geometry-based techniques.

To be able to use dynamic lines for different terrain types, only a minimum terrain resolution is required to determine the maximum segment length l_{max} for the line

Algorithm 3.1: subdivideLineSegment()

```
Input: A line segment with vertices $v_{start}$, $v_{end}$ and a maximum length $l_{max}$
Output: New line segment vertices $v^{(n)}$
1  $n = 1$
2  $v^{(0)} = v_{start}$
3  $\vec{s} = v_{end} - v_{start}$
4  if $||\vec{s}|| > l_{max}$ then
5      $n = \text{floor}\left(\frac{||\vec{s}||}{l_{max}}\right) + 1$        // $n \geq 2$
6      $\vec{o} = \frac{\vec{s}}{n}$                                      // Calculate vertex offset
7      for $i \leftarrow 1$ to $(n-1)$ do
8          $v^{(i)} = v_{start} + (i \cdot \vec{o})$                  // Add new vertex
9      end
10 end
11 $v^{(n)} = v_{end}$
```

subdivision. An adaptive terrain does not cause to repeat this step every time the terrain is changing. In Figure 3.5 only the lengths of the first and last line segment are larger than the maximum segment length and therefore two new vertices are added during the subdivision process, which are highlighted in red. For the subdivision process, only a line segment with its start and end vertices v_{start} and v_{end} and the maximum segment length l_{max} are required to subdivide the segment and to determine new segment vertices, as shown in Algorithm 3.1. For line segments that are shorter than the predefined maximum length no subdivision is executed and its start and end vertices stay the same. Otherwise, the segment is divided in smaller line segments with equal length that are shorter than the maximum length.

The subdivision can lead to a much higher number of line vertices depending on the maximum segment length. The larger amount of line data is an additional challenge for the rendering process, but the terrain matching is necessary to be able to restrict the search space for lines during rendering. After subdividing the individual line segments, the new vertices are sampled on the terrain surface by accessing the height information of the terrain's height field at the vertex locations. This height information is stored in the third dimension of the 2D line vertices. The new 3D line vertices are then used during the dynamic update process to position the 3D line segment boxes on the terrain and limit the terrain area that can be affected by the dynamic lines.

3.3 Dynamic Update

Compared to the previous steps, which can be done during preprocessing, this part of the visualization process depicted in Figure 3.1 has to be performed always before rendering. It is only necessary for dynamic lines because their data have to be continuously updated

Algorithm 3.2: updateLineSegmentIndices()

```
Input: All vertices of a line $v^{(n)}$ and the image width $i_w$ in pixels
Output: New line segment indices $i^{(m)}$
for $i \leftarrow 0$ to $n$ do
    $vs = \mathsf{transformToScreenSpace}(v^{(i)})$
    $p^{(i)} = \mathsf{floor}(vs_y) \cdot i_w + \mathsf{floor}(vs_x)$    // Calculate pixel index of $v^{(i)}$
end
$m = 0$
for $j \leftarrow 1$ to $n$ do
    if $p^{(j-1)} \neq p^{(j)}$ then
        if $\mathsf{segmentInFrustum}(v^{(m)}, v^{(j)})$ then
            $i^{(m)} = (m, j)$    // Add new line segment index
            $m = m + 1$
        end
    end
end
```

due to the view-dependent scaling of the lines. The dynamic update can be divided into three different steps, which are used to reduce the line data and to organize them for a quick access during rendering. At first, the line segment indices are updated, in the update indices step, to refer only to segments that are relevant for the current view. During the second step, the pixel data generation, all lines that cover are stored in per-pixel lists. The last step of the dynamic update is the line sorting, where the lines stored in the pixel lists are sorted for a consistent line drawing order over all pixels.

3.3.1 Update Indices

Since the terrain matching step performed during preprocessing produces a high number of line vertices, it is desirable to reduce the vertices as much as possible before the lines are rendered. To achieve that the lines are simplified and a view frustum culling is executed.

The aim of line simplification is the reduction of the line complexity without creating a visible difference. The proposed approach exploits the fact that only the change over two or more pixels can be distinguished by the viewer and the removal of consecutive line vertices located at the same pixel cannot be recognized. The vertices are not actually removed, because it is more efficient to update only the segment indices to refer only to vertices that are not on the same pixel as their previous vertex. This update process of line segment indices is shown in Algorithm 3.2. To sort out unwanted vertices, the screen-space positions of all line vertices and corresponding pixel indices are determined. If two consecutive line vertices are assigned to different pixel indices, a line segment index referring to the two vertices as start and end vertex is added. Otherwise, the vertices with predecessors belonging to the same pixel are omitted without changing the visual appearance of the line. Thus, only line segments are added that cover more than one pixel.

The proposed line simplification has a low effect in close-up views. Depending on the density of the line vertices, they do not or rarely lie on the same pixel and the number of reduced vertices is limited. Especially in close-up views there are many line vertices that are irrelevant for the current image, because they are outside the visible area. Therefore, all line segments are tested if they lie inside the view frustum and only if this is the case, they are passed further to the next steps.

The line simplification and view frustum culling are view-dependent and have to be performed in every image update. To accelerate the execution time of the processes, they are parallelized. This is accomplished by performing a parallel stream compaction, a common programming function that filters out only wanted elements from a given array [BRSC17]. All line vertices and their corresponding pixel indices represent the input array, which is filtered by removing all consecutive occurrences of the same pixel index. The proposed screen-based line simplification approach deals only with the special case of consecutive line vertices located at the same pixel. A more general simplification could achieve a greater reduction of line data and further improve the performance, but there is a lack of such screen-based line simplification approaches applicable during runtime.

3.3.2 Pixel Data Generation

After the reduction of line vertices, it is also necessary to reduce the search space for lines to a certain area on the screen. Otherwise, all line segments would be possible candidates for all pixels, which cannot be processed in real time for a larger number of segments.

The 3D line vertices produced in the terrain matching step and reduced in the index update step, are used to generate 3D line segment boxes. The boxes are used for a rough determination of the area where a line segment can be, to reduce the number of line segments that possibly cover a pixel and to process only these segments per pixel. The location and the length of the line segment boxes, shown in red in Figure 3.6, are defined by the corresponding segment vertices v_{0-6}, represented by gray circles. The box width is determined by a predefined line width, scaled according to the current view. For the scaling, a view-dependent scale factor is calculated according to the projection type of the virtual camera, which can be perspective or orthographic. The calculation of the box width b_w is shown in Algorithm 3.3, which is the first step of the generation process of 3D segment boxes, presented in Algorithm 3.4. The orthographic scale factor depends only on camera internal parameters and is therefore the same for all segment boxes. The perspective scale factor also depends on the segment vertex v_{start} or v_{end}, further away from the camera is also relevant. The more distant vertex defines the maximum scale factor of the whole line segment and is therefore used to scale the width of the 3D line segment box.

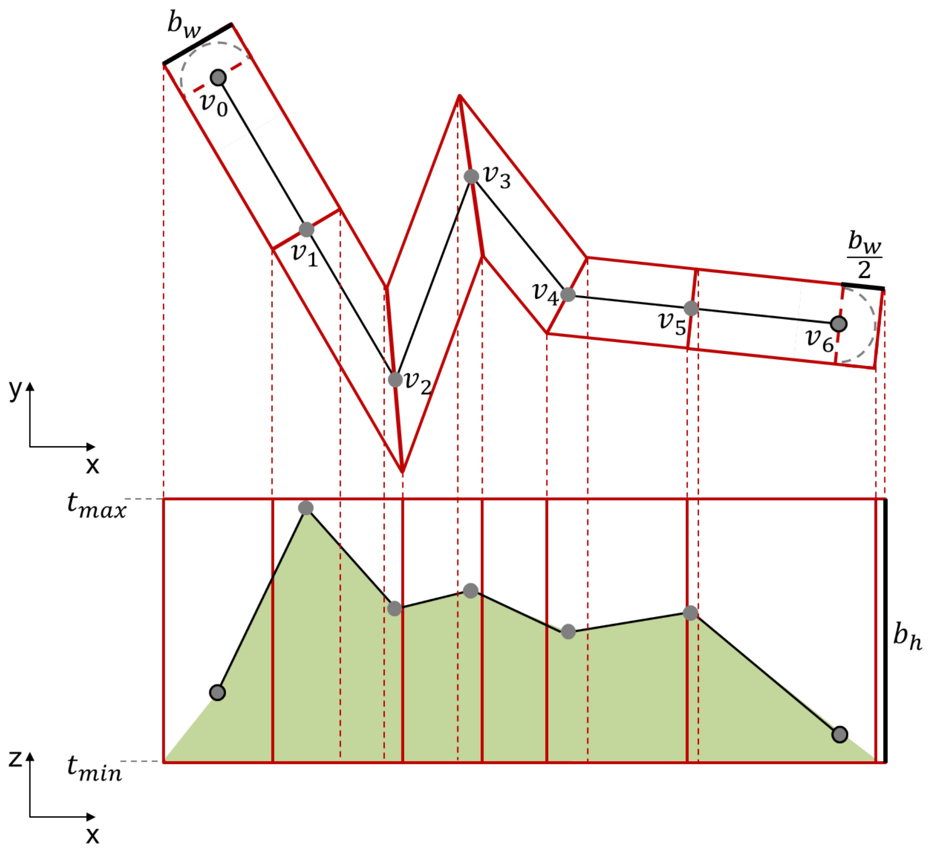


Figure 3.6: For the 3D segment box generation, the 3D line vertices v_{0-6} obtained from terrain matching are used to determine the locations and lengths of the boxes (red). The box width b_w corresponds to the scaled line width, the box offsets at the line ends to half of the scaled line width $\frac{b_w}{2}$, and the box height b_h to the minimum and maximum terrain height t_{min} and t_{max}.

The 3D segment boxes are defined by eight corner positions, generated by shifting the line segment vertices v_{start} and v_{end}. The front surface of the segment box is defined by four corners that are created by shifting the start vertex in four different directions. The directions are defined by an offset vector $\vec{o}_{frontSide}$ for left and right shifts and the height range of the terrain is used for up and down shifts. For the back surface the end vertex is shifted up also according to the terrain's height range but by a different offset vector $\vec{o}_{backSide}$. To keep the area covered by 3D segment box as small as possible, the overlaps between consecutive segment boxes are avoided. Therefore, the side offsets depend on the previous line segment for the front surface and on the next line segment for the back surface.

If one of the segment vertices is located at a line end, the corresponding surface is

Algorithm 3.3: getBoxWidth()

```
Input: A line width $l_w$, line segment vertices $v_{start}$, $v_{end}$, and the camera $c$
Output: Scaled line width $b_w$
1  if orthographic view then
2      $b_w = l_w \cdot c.\mathsf{getOrthographicScaleFactor}()$
3  else if perspective view then
4      $d_{start} = ||c.\mathsf{position} - v_{start}||$
5      $d_{end} = ||c.\mathsf{position} - v_{end}||$
6      $d_{max} = \mathsf{max}(d_{start}, d_{end})$
7      $b_w = l_w \cdot c.\mathsf{getPerspectiveScaleFactor}(d_{max})$
8  end
```

Algorithm 3.4: generate3DSegmentBox()

```
Input: A line segment with vertices $v_{start}$, $v_{end}$, the predecessor and successor line
       vertices $v_{prev}$, $v_{next}$, the line width $l_w$, the minimum and maximum terrain
       height $t_{min}$, $t_{max}$, and the camera $c$
Output: Corner positions of the segment box $p^{(0-7)}$
1  $b_w = \mathsf{getBoxWidth}(v_{start}, v_{end}, l_w, c)$
2  $\vec{s} = (v_{end} - v_{start})_{xy}$
3  $\vec{d} = \frac{\vec{s}}{||\vec{s}||}$
4  $\vec{n} = (\text{-}\vec{d}_y, \vec{d}_x)$
5  if $v_{start}$ is first line vertex then // Calculate front surface offsets
6      $v_{start}.\mathsf{subdractLineEndOffset}(\vec{d} \cdot \left(\frac{b_w}{2}\right))$
7      $\vec{o}_{frontSide} = \vec{n} \cdot \left(\frac{b_w}{2}\right)$
8  else
9      $\vec{b} = \mathsf{getAngleBisector}(v_{prev}, v_{start}, v_{end})$
10     $\vec{o}_{frontSide} = \vec{b} \cdot \left(\frac{b_w}{2}\right)$
11 end
12 if $v_{end}$ is last line vertex then // Calculate back surface offsets
13     $v_{end}.\mathsf{addLineEndOffset}(\vec{d} \cdot \left(\frac{b_w}{2}\right))$
14     $\vec{o}_{backSide} = \vec{n} \cdot \left(\frac{b_w}{2}\right)$
15 else
16     $\vec{b} = \mathsf{getAngleBisector}(v_{start}, v_{end}, v_{next})$
17     $\vec{o}_{backSide} = \vec{b} \cdot \left(\frac{b_w}{2}\right)$
18 end
19 $p^{(0-3)} = ((v_{start})_{xy} \pm \vec{o}_{frontSide}, t_{min/max})$     // Front surface corners
20 $p^{(4-7)} = ((v_{end})_{xy} \pm \vec{o}_{backSide}, t_{min/max})$       // Back surface corners
```

additionally shifted by half the box width in the direction of the segment $\vec{d}$. Thus, the segment boxes belonging to the start and end segments of a line are extended to include the line end caps. To obtain the desired box width b_w, the start vertex is also shifted to the side by half the box width in the direction of the normal vector $\vec{n}$ of the segment if it is the first vertex of the line. The same shift operation is executed for the end vertex if it is the last vertex of the line. For intermediate vertices, the angle bisector of the previous and the current line segment determines the direction in which the start vertex is shifted and the angle bisector of the current and the next line segment is used for the end vertex. After defining the box width by shifting the segment vertices, the height b_h is determined according to the minimum and maximum terrain height t_{min} and t_{max}.

The so defined segment boxes enclose the area on the terrain where a line segment can be. After projecting the segment boxes onto the screen, only the pixels covered by the boxes can be part of a line and only for these covered pixels point-in-line tests have to be executed. If a test is successful, the corresponding line index is stored with all other covering lines per pixel. Additional to the line index also the line color is stored in a linked list per pixel to reduce the number of memory accesses during sorting and rendering.

The reason for the storage of lines per pixel is to be able to sort them afterwards and to ensure a consistent drawing order. This is necessary because the parallel execution of this pixel data generation process leads to lines being output in arbitrary order. Therefore, the line indices are stored in linked lists per pixel and sorted according to the drawing order of the lines. Linked lists are used because they support the storage of elements in random order, which can be sorted by changing only the references.

3.3.3 Line Sorting

After the pixel generation step, the indices of all lines covering a pixel are stored in linked lists in random order. To ensure a consistent drawing order of the individual lines, the pixel lists are sorted accordingly. Without this process different lines would be drawn on top and different line colors would be displayed by pixels that are part of line overlaps. Figure 3.7a shows a visualization of dynamic lines without line sorting. One can see artifacts at the overlap area of the red and green line, which are caused by skipping the sorting step. In Figure 3.7b the lines are sorted consistently according to their length, with the red line on top followed by the pink and the green line. This prevents visual artifacts and a randomly changing drawing order of lines during runtime.

The sorting criterion can be different, but by default the length of the lines is used as for static lines. For the sorting a selection sort algorithm is applied because it supports an early termination as the first N elements already have their fixed position after the N^{th} pass. The sorting algorithm re-sorts two lines if their order does not match their drawing order. The sorting process is terminated if the composited color of the already sorted lines is opaque or if all lines of the list are in the correct order. This means if the line colors are all opaque only the line rendered on top has to be found in the list and the

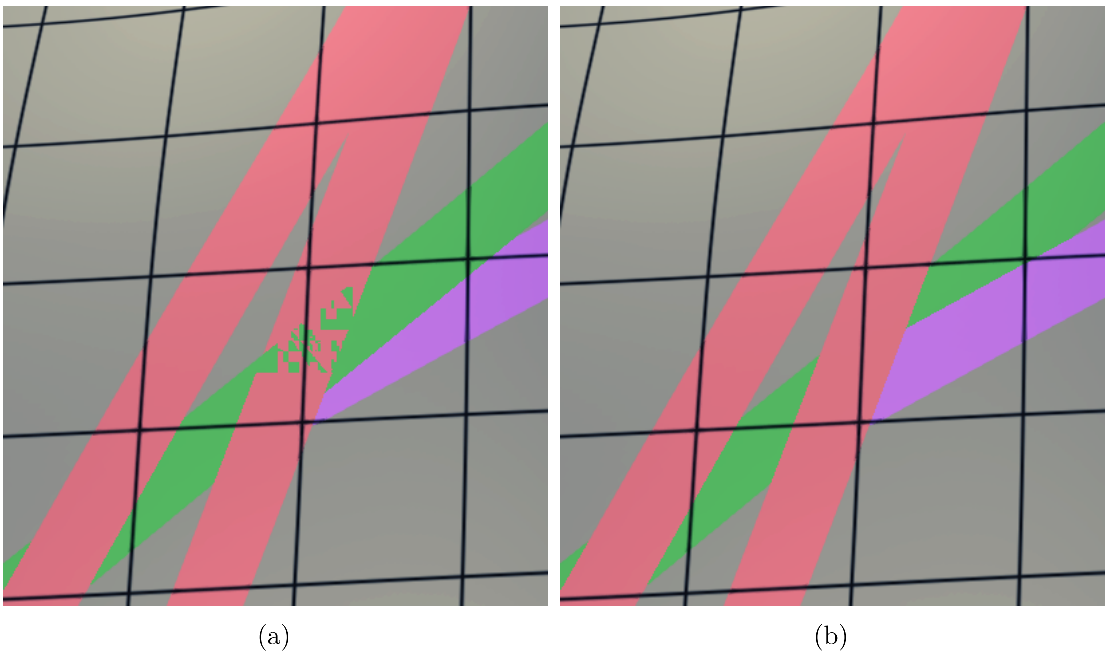


Figure 3.7: Visualization of dynamic lines (a) without a consistent drawing order and (b) with a consistent order achieved by sorting the linked lists per pixel.

sorting process can be stopped. Only if all lines have the same line color with the same transparency, this step can be completely skipped because the sorting would not change the pixel colors.

After the execution of this dynamic update the lines are simplified and the vertices outside the visible area are discarded. The lines are stored with their index in per-pixel lists and the list elements are sorted according to the line drawing order. The resulting per-pixel data are passed further to the next step of the visualization process where the dynamic lines are rendered.

3.4 Rendering

The rendering process of static and dynamic decals is screen-based and executed after the terrain is rendered to be able to project them onto the terrain surface. During decal rendering, the terrain color of pixels that are covered by decals is replaced with the corresponding decal color, but the pixel depth determined by the terrain remains the same.

To accelerate the rendering, different data structures are generated during preprocessing for all decal types and during the dynamic update for dynamic lines. An overview of this data preparation process and the resulting data structures are shown in Figure 3.8. The data structures are deployed to find all decals that contribute to a pixel and to determine

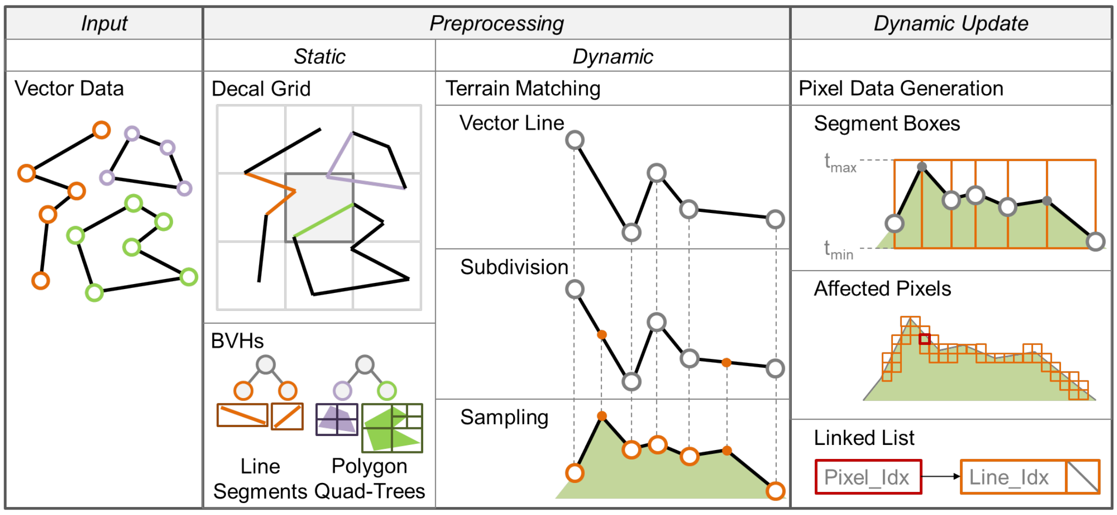


Figure 3.8: The data structures that are used to accelerate the rendering process are generated during preprocessing and additionally during dynamic updates for dynamic lines. For static decals a decal grid and BVHs for line segments and polygon quad-trees are created and for dynamic lines linked lists containing all lines covering a pixel are built up.

the pixel color accordingly. While the static decals are rendered with constant size, the dynamic lines are scaled according to the current view.

3.4.1 Static Decals

The render approaches of static lines and polygons are similar, because both use the decal grid and corresponding BVHs to determine if the decals contribute to a pixel. The main steps that are executed during the render process of static decals to determine the color of a pixel are shown in Algorithm 3.5.

The first step of the screen-based rendering of static decals is to calculate the world-space position of the current pixel center position by using the terrain depth t_d. The world-space pixel position p_{ws} can then be located on the decal grid and this location belongs to a certain grid cell. The corresponding cell index i_{cell} delivers all static decals $d^{(m)}$ that potentially cover the current pixel. The decals of a cell are organized in a BVH to accelerate the search for decals that really cover the current pixel and contribute to the pixel color p_c. The nodes of this BVH $n^{(m)}$ are traversed from top to down and from left to right according to the containment of the pixel position in the node AABBs n_{AABB}. If the pixel position p_{ws} is outside the nodes's AABBs, the whole sub-tree of the node can be skipped and the next right node, which has not already been processed is tested. The search for pixel-relevant decals may end at one or more leaf nodes, depending on the number of decals that cover the current pixel. A leaf node of the BVH refers either to an exterior polygon quad-tree or to a line segment.

Algorithm 3.5: renderStaticDecalsofPixel()

```
Input: A pixel position $p$, and the data structures of static decals
Output: Pixel color $p_c$
$t_d$ = getTerrainDepthAtPosition($p$)
$p_{ws}$ = getWorldSpacePosition($p, t_d$)
$i_{cell}$ = getGridCellIndex($p_{ws}$)
$d^{(m)}$ = getDecalsOfCell($i_{cell}$)
$n^{(m)}$ = getBVHNodesOfCell($i_{cell}$)
i = 0
while $i < m$ do
    if $p_{ws}$.inside($n^{(i)}_{AABB}$) then
        if $n^{(i)}$is internal node then
            $i$ = getLeftChildIndex($i$)          // Go to left child node
            continue
        else if $p_{ws}$.inside($d^{(i)}$) then
            $p_c$.addColor($d^{(i)}_c$)            // Add decal color
            if $p_c$ is opaque then
                break
            end
        end
    end
    if nodes left to process then
        $i$ = getNextRightNodeIndex($i$)      // Go to next right node
    else
        break
    end
end
```

The quad-trees also have to be traversed until a leaf node is reached but the traversal ends exactly at one leaf node, which is determined by the pixel position. Depending on the leaf node type, fully inside, fully outside, or partially inside, the pixel can be immediately classified as inside or outside the exterior polygon or an additional point-in-polygon test has to be performed. For pixels that are detected to be inside of an exterior polygon, it has to be tested if the pixel is located at a cutout produced by an interior polygon, before the corresponding polygon color is applied to the pixel. Therefore, the list of interior polygon parts assigned to a cell is searched for those that belong to the detected exterior polygon. If an interior polygon is found, its corresponding quad-tree is also traversed to a leaf node. The determination whether a pixel is covered by an interior polygon is done the same way as for exterior polygons. After this process the pixel is either inside or outside of an interior polygon. For the first case, the exterior polygon covers the pixel and its color can be added to the pixel color. Otherwise, the pixel is located at a cutout or at a nested interior polygon and the color remains unchanged or is adjusted accordingly.

For line segments a point-in-line test can be performed directly to determine if the current pixel belongs to a line. If the pixel is inside of a static decal, the decal color d_c is added to the pixel color p_c with front-to-back compositing. The search process is continued until the pixel color is opaque or if all nodes are processed and the BVH is fully traversed. Then there can be no more static decals that can change the final pixel color.

3.4.2 Dynamic Lines

The render pass of dynamic lines is simple and fast because the line data is already prepared according to the current view and organized per pixel after the dynamic update phase. The line data can be efficiently processed during rendering by accessing the linked list of a pixel. There is a linked list for each pixel containing all required lines in their drawing order. It is only necessary to look up the linked list of the current pixel and if the list is empty, there are no lines covering the pixel. For pixels with non-empty lists, the pixel color has to be determined. This is done by going through the linked list and performing front-to-back compositing of the individual line colors until the color is opaque or the end of the list is reached.

3.5 Anti-Aliasing

The described rendering process of all decal types produces hard and jagged edges. This is caused by the point-in-line and point-in-polygon tests, which use only the pixel center position to determine if the whole pixel is inside or outside the corresponding decal. Pixels that are touching a decal are either fully colored with the decal color or left completely empty, depending on the location of the pixel center. The resulting jagged edges can be seen in Figure 3.9a for a static outline and in Figure 3.9c for a dynamic outline. To avoid such unpleasant edges and to get a smooth transition from decal color to terrain color, anti-aliasing has to be applied.

Analytical anti-aliasing methods require additional computations during rendering to continuously increase the transparency to the outside of the decals to create smooth decal edges. Fading out the edges, which are embedded into the 3D environment, requires additional information such as the pixel size in world space to determine if a pixel is part of the decal edge or not. To avoid this additional effort and to keep it simple, other methods are used.

In order to create smooth edges, different anti-aliasing methods are applied to static decals and dynamic lines. MSAA delivers high quality anti-aliasing and can be easily applied during a render pass by taking multiple samples per pixel to calculate an average pixel color. Therefore, this anti-aliasing method is used to improve the visual appearance of static decals. A black outline anti-aliased with MSAA is shown in Figure 3.9b. MSAA is not suitable for the pixel-based approach of the dynamic lines because the affected pixels are already determined and stored before rendering in the dynamic update process.

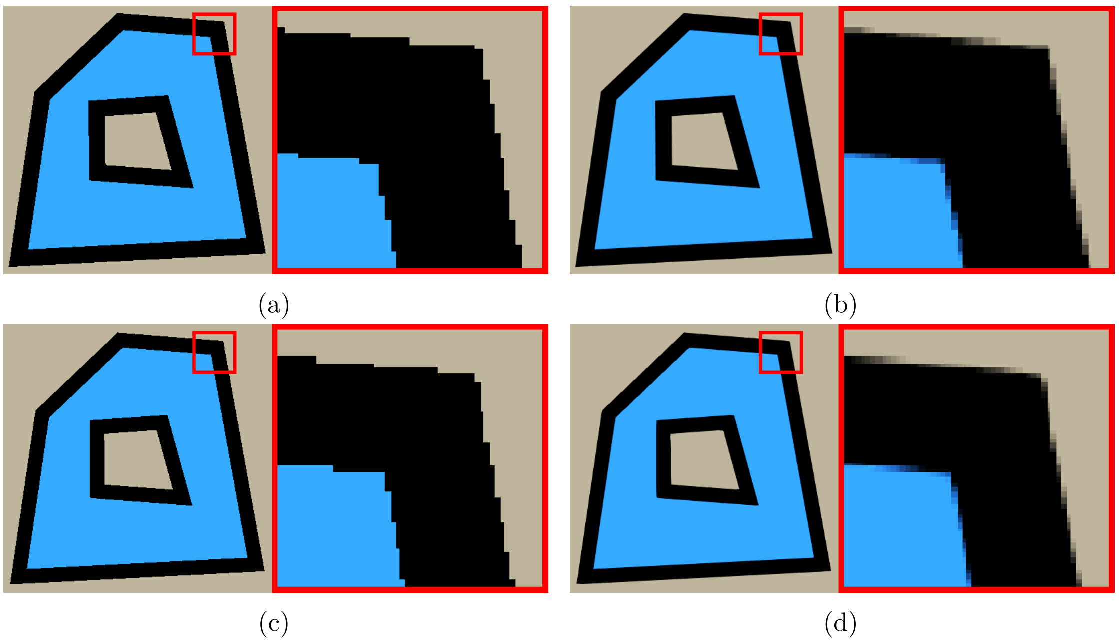


Figure 3.9: A blue polygon with static black outlines (a) without anti-aliasing and (b) with MSAA. The same polygon with dynamic black outlines (c) without anti-aliasing and (d) with FXAA.

For the anti-aliasing of the lines with MSAA additional samples are required, which determination and storage are too time- and memory-consuming for real-time rendering. Thus, the dynamic lines are rendered without anti-aliasing and FXAA [fxa] is applied afterwards as a post-process. This anti-aliasing technique is described in Section 2.6 and a comparison between the aliased and anti-aliased dynamic outlines is shown in Figure 3.9. A disadvantage of using FXAA is that it leads to artifacts applied on thin lines, which can be reduced by MSAA.

3.6 Line Styles

Different styles for lines and outlines can be used to influence their visual appearance. Both static and dynamic lines can be displayed with different corner styles, which are round, miter, and bevel. As Figure 3.10 shows, one line segment is defined by two vertices v_0 and v_1 and has two corners. If the line segment is not at the line end, the corners to the previous segment and to the next segment can produce a gap, such as in Figure 3.10a. The corner styles close this gap between two consecutive line segments with different techniques and influence the style of the line.

For a correct computation of the corner styles, the neighborhood information of a line vertex is necessary. To avoid the storage and access of this information during rendering, all additional positions required for the corner styles are precalculated for static lines.

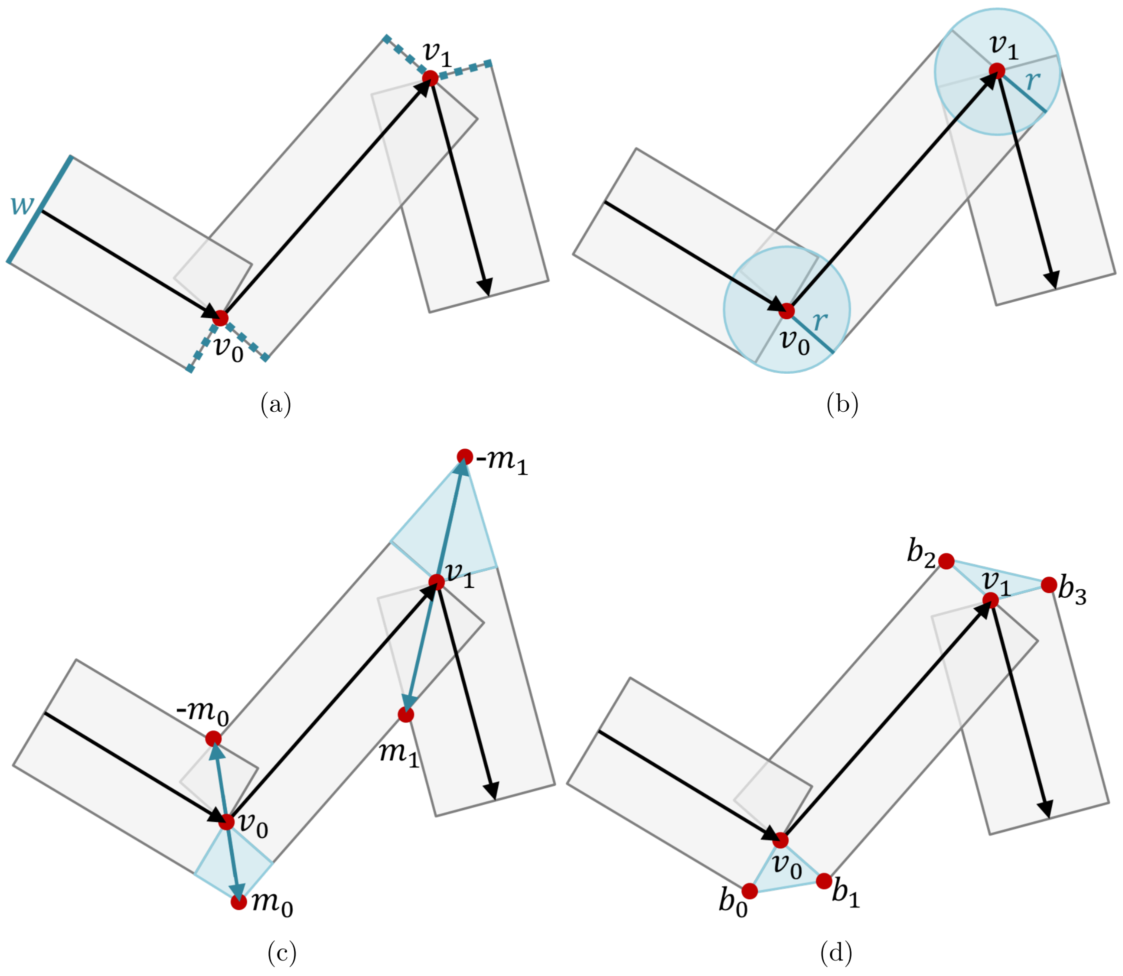


Figure 3.10: (a) Two corners created by three consecutive line segments represented by gray rectangles defined by black vector lines with vertices v_0 and v_1 and line width w. All necessary positions for the different corner styles (b) round, (c) miter, and (d) bevel are highlighted in red.

The corner positions of dynamic lines are changing with the view as the lines do. Since the lines are only scaled while their orientation stays the same, the direction vectors of the corner positions could be calculated during preprocessing. But, due to the simplification of dynamic lines by removing vertices, the predecessor and successor vertex may change and the direction vectors with them. Thus, all information that is necessary for the corner styles is determined during the dynamic updates.

The round corner style does not require additional corner positions because only the radius r, shown in Figure 3.10b, is required to determine if a pixel lies inside the corner gap. Since the radius is half the line width w, no further processing is necessary for this corner type because the width is already predefined. To be able to determine if a

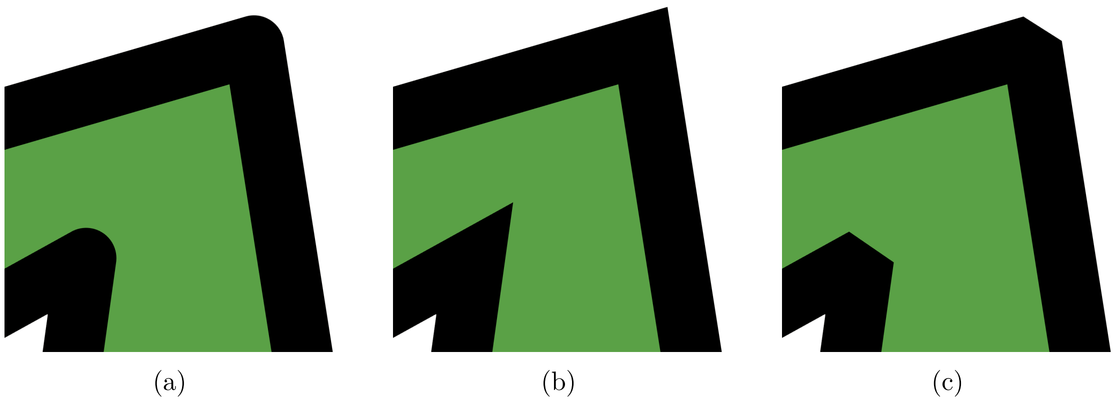


Figure 3.11: The three corner styles for lines and outlines (a) round, (b) miter, and (c) bevel shown for a green polygon with black outline.

pixel lies inside a line with a miter corner style, one additional 2D vector is required per corner, m_0 or m_1 in Figure 3.10c. This vector is the angle bisector of two consecutive line segments. The lengths of the miter vectors depend on two factors, the line width and the angle between the segments meeting at the corner. The smaller the angle is, the longer the vector. For the point-in-line tests during the render process also the inverse vectors of the miter vectors $-m_0$ and $-m_1$ are necessary, which do not have to be stored because they are implicitly given by the miter vectors m_0 and m_1. For the bevel corner style two additional 2D vectors are necessary per corner, b_0 and b_1 for the first corner and b_2 and b_3 for the second one. They are pointing in the direction of the normals of the two line segments meeting at a corner. The sign has to be chosen so that the normals point in the direction of the corner gap. The correct sign can be determined by the direction in which the second line segment points from the first segment, which is either left, right, or straight. If the gap is left or right, the right or left normal has to be used respectively, because the gap is located where the segment does not point. For straight line segments the gap does not exist and does not have to be closed and the directions of the bevel vectors are irrelevant.

The results of the three different corner styles are shown in Figure 3.11 by black outlines. They are created by using different point-in-line tests during the render process according to the selected corner style. This is done in the same way for static and dynamic lines. The advantage of the round corner style is that its point-in-line test can be performed more efficiently compared to the other corner styles. If it is of higher priority that the lines have sharp corners to represent the underlying vector data in the best possible way, the miter corner style should be preferred. This may be the case for outlines of polygons representing building footprints, which are usually angular and not round. If two line segments are very acute-angled, their shared corner can become very sharp and long with the miter corner style. Then the bevel corner style can be used, which cuts off the corner.

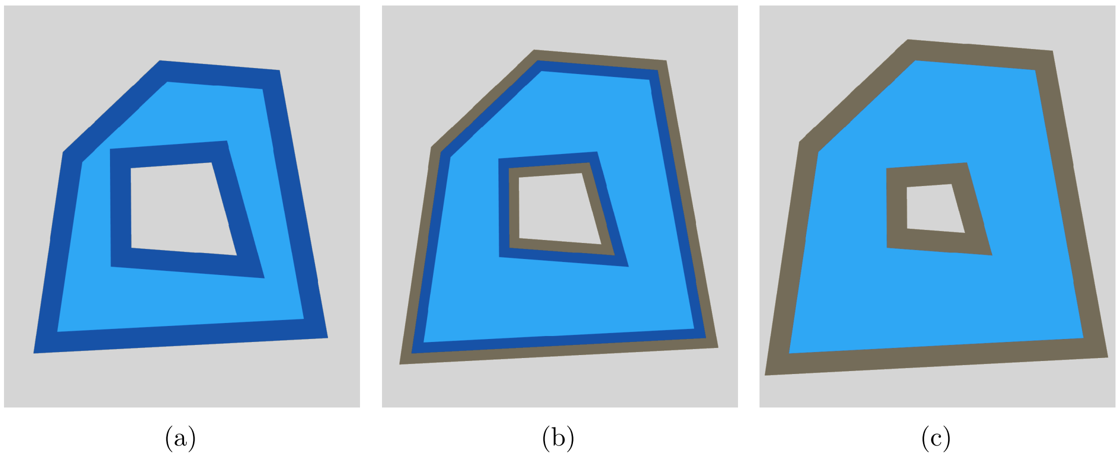


Figure 3.12: The three outline modes for static polygon outlines (a) inset, (b) normal, and (c) offset shown for a blue polygon with a semi-transparent black outline.

Another stylistic element are the three different outline modes, inset, normal, and offset for static lines, which can be seen in Figure 3.12. The outline modes create different effects for the enclosed polygons and depending on the polygons a specific mode can be useful. The inset outline mode keeps the outlines inside their corresponding polygons. For polygons that are close together, the inset mode can be used to prevent outlines to overlap. The outlines are completely outside their polygons if the offset outline mode is used. Small polygons can be a good use case for the offset mode because the outlines do not hide the enclosing polygons. The inset and offset outline modes are only available with a miter corner style, because otherwise the modes would produce gaps between the polygons and their outlines. Furthermore, the outline modes inset and offset can lead to a wrong perception of polygons, which may appear shrunk or enlarged. By using the normal outline mode, the outline is half inside and half outside of its polygon and the polygon size is rather perceived as it is.

For the normal outline mode no additional steps are necessary during preprocessing or rendering. To be able to display static lines in inset and offset modes, the line widths are doubled before the lines are assigned to the decal grid cells. In case of the inset mode the outline part that is outside of the corresponding polygon can then be omitted during rendering. For the offset mode the outline part inside the polygon is omitted and not displayed to achieve the desired effect.

CHAPTER 4

Implementation Details

The previous chapter introduced the two screen-based visualization approaches for static decals and dynamic lines. This chapter covers details of the concrete implementation of the individual steps, which are shown and classified according to their processing on the Central Processing Unit (CPU) or Graphics Processing Unit (GPU) in Figure 4.1. The implementation details also reveal some limitations of the used visualization methods, which are also presented and discussed in this chapter.

The proposed vector data visualization method is integrated into the already existing framework of the flood management system *Visdom* [vis]. For the implementation the graphics API *OpenGL* [ogl] version 4.6 and the parallel computing platform *CUDA* 10.2 [cud] with *Thrust* [thr] are utilized.

4.1 Input Data

For the screen-based decal rendering it has to be determined, which pixel world-space positions are covered by decals defined by 2D vector data. Therefore, different data structures and user settings are deployed to visualize given georeferenced vector data, as shown in Figure 4.1. These vector data consist of lines and polygons, which are available as open and closed polylines defined by 2D positions in double precision. The required vector data are collected from various open and closed data sources, such as *OpenStreetMap* [osm], which is a project that creates and provides free geographic data.

In addition to the vector data, different settings can be made by the user to control the visualization process according to the individual needs. This includes settings for the data structure generation and settings to adjust the color, width, and style of the decals. Since the quad-tree generation for particularly complex polygons has a major impact on memory consumption, preprocessing, and rendering time, the user can set limits for the quad-tree leaf capacity and tree depth according to individual requirements. Low

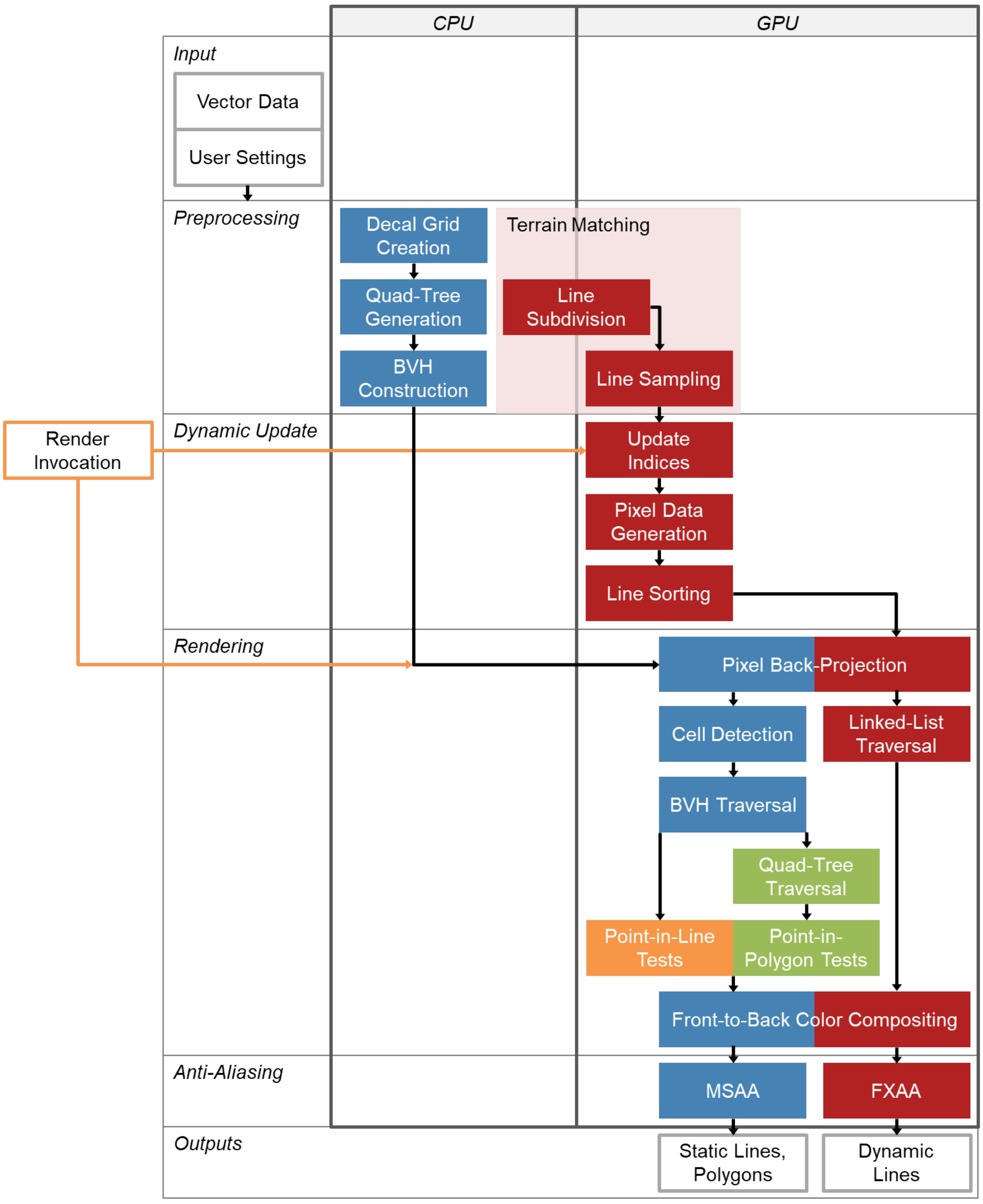


Figure 4.1: The individual steps of the visualization process of static decals in blue and dynamic lines in red divided according to processing either on the CPU or the GPU. The steps in which the static decals differ are highlighted in orange for lines and in green for polygons.

limits can accelerate the preprocessing and reduce memory requirements for quad-tree data, but at the expense of slower rendering and vice versa for high limits. Different style settings allow the user to determine the visual appearance of the decals to reflect the associated data objects, e.g. by using green color for parks or color scales to color code additional information about the objects. The use of different decal colors, line styles, and outlines for polygons can make the individual data objects distinguishable.

4.2 Preprocessing

During preprocessing the given vector data and user settings are used to generate data structures to accelerate the decal rendering and to adjust the visualization process to the user needs. In Figure 4.1 one can see that the preprocessing of static decals is done completely on the CPU and mostly on the GPU for dynamic lines. The parallel construction of the tree-based data structures is not trivial and is therefore executed mainly sequentially on the CPU instead. The preprocessing of static decals could be accelerated if it would be parallelized and transferred to the GPU, but this remains open for future work.

4.2.1 Static Data Structures

The data that are necessary for an efficient rendering of static decals are prepared and stored in this step. This includes the generation of shader storage buffers containing information about BVHs, line segments, and polygon quad-trees. Figure 4.2 shows an overview of these buffers structured according to the information they contain and the arrows indicate references from the data stored in one buffer to the data of another buffer. The first step of the data preparation of static decals is the creation of the decal grid to subdivide the area covered by all decals into small grid cells. Then the line and polygon data are created and stored in corresponding buffers, whereby the line data are based on line segments and polygon data on quad-trees. Individual line segments and polygon quad-tree nodes are then referenced to intersecting decal grid cells. After the determination of line segments and polygon quad-tree nodes belonging to individual cells, they are used to construct separate polygon and line BVHs per grid cell. All required data for the BVH traversal during rendering are also stored in different shader storage buffers.

Decal Grid

The decal grid is a 2D regular grid containing all static decals and the grid size is therefore determined by the common AABB of all lines and polygons. Then a predefined number of 250,000 grid cells is used to divide the grid area. The resulting decal grid has a dimension of 500×500 cells, which allows the decals to be rendered fast, while the preprocessing does not take too much time. As Figure 4.2 shows, the cell index i_{cell} of the decal grid cell has to be calculated during rendering to access the data of BVHs, lines, and polygons. The cell index corresponds to the grid cell containing the world-space position of a pixel

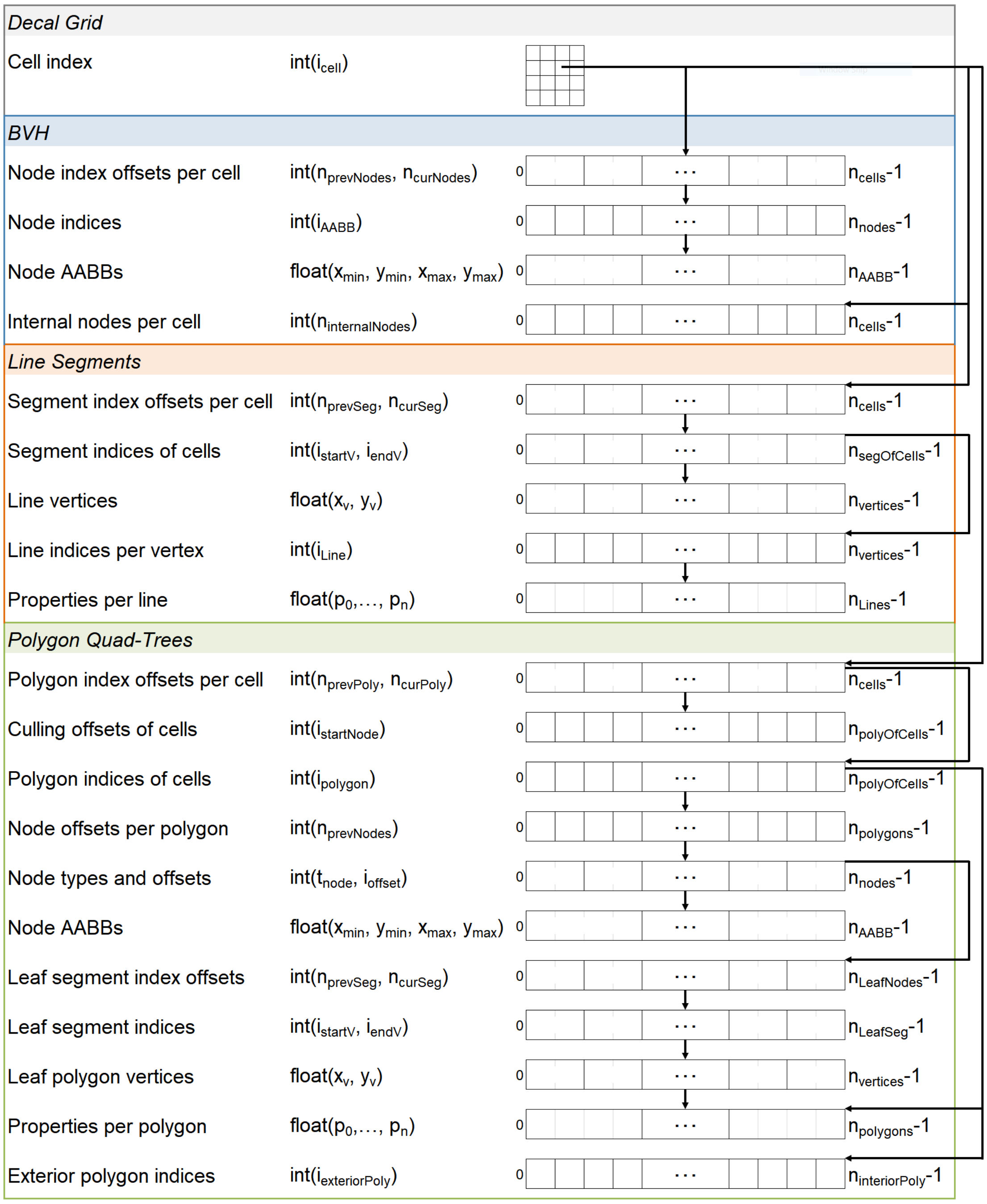

Decal Grid
Cell index — int(i_{cell})
BVH
Node index offsets per cell — int($n_{prevNodes}$, $n_{curNodes}$) — 0 … n_{cells}-1
Node indices — int(i_{AABB}) — 0 … n_{nodes}-1
Node AABBs — float(x_{min}, y_{min}, x_{max}, y_{max}) — 0 … n_{AABB}-1
Internal nodes per cell — int($n_{internalNodes}$) — 0 … n_{cells}-1
Line Segments
Segment index offsets per cell — int($n_{prevSeg}$, n_{curSeg}) — 0 … n_{cells}-1
Segment indices of cells — int(i_{startV}, i_{endV}) — 0 … $n_{segOfCells}$-1
Line vertices — float(x_v, y_v) — 0 … $n_{vertices}$-1
Line indices per vertex — int(i_{Line}) — 0 … $n_{vertices}$-1
Properties per line — float(p_0,…, p_n) — 0 … n_{Lines}-1
Polygon Quad-Trees
Polygon index offsets per cell — int($n_{prevPoly}$, $n_{curPoly}$) — 0 … n_{cells}-1
Culling offsets of cells — int($i_{startNode}$) — 0 … $n_{polyOfCells}$-1
Polygon indices of cells — int($i_{polygon}$) — 0 … $n_{polyOfCells}$-1
Node offsets per polygon — int($n_{prevNodes}$) — 0 … $n_{polygons}$-1
Node types and offsets — int(t_{node}, i_{offset}) — 0 … n_{nodes}-1
Node AABBs — float(x_{min}, y_{min}, x_{max}, y_{max}) — 0 … n_{AABB}-1
Leaf segment index offsets — int($n_{prevSeg}$, n_{curSeg}) — 0 … $n_{LeafNodes}$-1
Leaf segment indices — int(i_{startV}, i_{endV}) — 0 … $n_{LeafSeg}$-1
Leaf polygon vertices — float(x_v, y_v) — 0 … $n_{vertices}$-1
Properties per polygon — float(p_0,…, p_n) — 0 … $n_{polygons}$-1
Exterior polygon indices — int($i_{exteriorPoly}$) — 0 … $n_{interiorPoly}$-1

Figure 4.2: The data that are necessary for rendering of static lines and polygons with data types and sizes. They contain indices i, numbers of objects n, and positions with xy-coordinates and are stored in shader storage buffers. Decal grid and BVH data are required for both decal types, for lines also segment data and for interior and exterior polygons also quad-tree data are necessary.

center and the referred data belong to lines and polygons intersecting the cell. For the cell index determination the grid origin, dimension, and cell size is required, which are calculated once during preprocessing and passed to the fragment shader during a render pass.

The generated decal grid covers the whole area where static decals occur and is ready to be intersected with line segments and polygon quad-trees. Before the execution of the intersection tests, all line segments have to be extended to rectangles according to their predefined line widths, to perform correct intersection tests between the segments and the grid cells. To be able to assign only polygon parts to decal grid cells, both interior and exterior polygons are divided into smaller parts by quad-trees. These data preparation for lines and polygons is described in the following sections.

Line Segments

The line data are prepared according to predefined line widths and selected line styles to execute correct intersection tests with the decal grid, which is generated in the previous step. The line segments can then be assigned to their intersecting grid cells to reduce the number of line segments that have to be tested per pixel at runtime.

A line segment is defined by the positions of its two end vertices and the line width and for miter and bevel corner styles also by its corner positions. To make these line segment data accessible during rendering they are stored in different shader storage buffers. In Figure 4.2 one can see, that the xy-coordinates of all line vertices of the input 2D polylines are stored in a line vertices buffer in 2D float vectors $float(x_v, y_v)$. There are different line properties that are required for static and dynamic lines, which include the line width, line colors, and additional values to color code information on the line, such as its importance. The line properties are stored for both line types in corresponding properties per line buffers. Line widths are defined in meters in the 3D world and are represented by floats stored in a line width buffer per line. The RGBA colors of lines are stored in 4D float vectors $float(r, g, b, a)$ in a line color buffer. For a color coding of additional information on lines, predefined transfer functions are used to map per-line values stored as floats to a certain line color. To make the line properties accessible for a certain line segment during rendering, all line indices i_{Line} are stored per vertex in an additional line indices per vertex buffer.

Static lines and outlines can be displayed with three different corner styles, round, miter, and bevel, and three different outline modes, normal, inset, and offset, which are described in Section 3.6. In addition to the shader storage buffers listed in Figure 4.2, corner positions for miter and bevel corner styles are also stored per line vertex if one of these styles is requested by the user. The precalculation of all required corner positions for the miter and the bevel corner styles is done in parallel per line vertex on the CPU. To avoid abrupt ends of lines, null vectors are stored to the vertices at the line ends, indicating that a round end cap has to be inserted there. This is also done to avoid long sharp corners if a miter vector becomes too long. If two consecutive line segments are at an angle smaller than a predefined angle limit of 40 degrees, a null vector is added and the miter corner is replaced by a round corner during rendering.

Before the line segments are tested for intersection with the individual grid cells, they are extended according to predefined line widths and styles in parallel on the CPU per line segment. The segments are extended to rectangles for round and bevel corner styles having a width according to the corresponding line width and a length according to the line segment length. If the lines have to be displayed with a miter corner style, the segments are extended to quadrilaterals according to the already calculated miter vectors of the corresponding segment end points. The polygon outline modes inset and offset are only available for outlines with the miter corner style to avoid gaps between the outlines and their enclosing polygon. For these outline modes, the outline widths are doubled before they are used to generate segment quadrilaterals, so that the outline part that is inside or outside of the enclosing polygon can then be omitted according to the selected outline mode during the render process.

After the parallel extension of all line segments, they are assigned to all decal grid cells they cover. Therefore, the AABBs of the line segments are calculated and located on the decal grid for a rough determination of all possible line segments covering a grid cell. This is done for efficiency reasons, to limit the number of possible cell intersection candidates to certain line segments and to avoid accurate intersection tests between all segments and all cells. For an accurate intersection of the grid cells with all their candidate line segments, the segment rectangles and quadrilaterals are intersected with the cell squares in parallel on the CPU per grid cell. To make all intersecting line segments of a grid cell accessible with a cell index i_{cell}, all intersecting line segment indices are stored in a segment indices of cells buffer grouped per grid cell. The segment indices are stored in 2D integer vectors $int(i_{startV}, i_{endV})$, where the first element refers to the start vertex and the second one to the end vertex of the segment. This is memory-efficient because the use of line vertex indices avoids saving the vertex positions repeatedly for every cell they occur and the storage of 2D float vectors per cell can be replaced by integer indices referring to these vertex position vectors, which are stored only once. The segment indices are grouped per cell, but to know where the relevant indices of a cell are located in the indices buffer, an additional offset buffer is required. Therefore, the offset to the first line segment index and the number of segments assigned to a cell are stored in 2D integer vectors $int(n_{prevSeg}, n_{curSeg})$ per cell.

After this preprocessing step the line segment data can be accessed during rendering by a cell index, which refers to an offset buffer pointing to all relevant line segment indices of a cell. The line width, vertex positions, and corner positions stored at the segment indices are then used to determine if a pixel is covered by a static decal line.

Polygon Quad-Trees

The segments of the input 2D polylines of polygons are not directly assigned to decal grid cells. For a point-in-polygon test all segments of a polygon would have to be tested, which can become time-consuming for complex polygons. Thus, the interior and exterior polygons are divided into smaller parts by quad-trees and only the segments of small leaf polygons are used for testing. The shader storage buffer structure of polygon quad-trees presented in Figure 4.2 is generated for interior and exterior polygons to provide an efficient rendering. The quad-tree data for interior and exterior polygons are stored in the same buffers grouped per polygon. Only the exterior polygon indices buffer, which contains the exterior polygon indices $int(i_{exteriorPoly})$ per interior polygon, is reserved for interior polygons. The buffer is necessary to access the corresponding exterior polygons per interior polygon during rendering.

The polygon quad-tree generation is done in parallel on the CPU per polygon for all exterior and interior polygons. The quad-trees consist of internal nodes and leaf nodes, whereby all nodes refer to AABBs of the rectangular node quads, internal nodes additionally refer to their child nodes, and leaf nodes also to polygon segments. These node AABBs are stored in 4D float vectors $float(x_{min}, y_{min}, x_{max}, y_{max})$, containing the xy-coordinates of the minimum and maximum position of the bounding box. The use of square quads would be more memory-efficient because only a length has to be stored and for rectangular quads two values, the length and the width of the rectangle are required. But, rectangular quads have the advantage, that the quad-trees can adapt better to polygons. This is especially advantageous for long and thin polygons, where the trees can remain much narrower and do not cover an unnecessarily large area. The polygons are then assigned to less decal grid cells because their quad-trees are smaller.

The quad-tree generation is based on a queue and the root node is added as the first element, which contains all polygon segments and refers to the rectangular AABB of the whole polygon. The root node AABBs of a polygon can later be accessed by the node offset $int(n_{prevNodes})$ that are stored in a node offsets per polygon buffer. A root node AABB is divided into four further AABBs, which are assigned to the child nodes of the root node and added to the queue. The segments of the parent node are then assigned to its child nodes if they are inside or intersect the respective child node AABB. The child AABBs are not disjoint because they are expanded by a small offset to avoid precision problems at segments that are only touching the border between two nodes. Such segments would not be assigned to any node without this offset. To know where the child node AABBs of the current internal node are located, an index i_{offset} to the first child is stored per internal node in a node types and offsets buffer. The subdivision process of the internal nodes is continued until a predefined leaf capacity or a maximum quad-tree depth is reached.

The leaf capacity determines how many polygon segments can be assigned to a leaf node and thus influences the number of segments that have to be tested during the render process. Frasson et al. [FEP18] use only the leaf capacity as termination criterion for the polygon quad-tree generation but for complex polygons this can lead to very deep quad-trees, especially for a small leaf capacity. This should be prevented because the quad-trees are generated sequentially and have to be traversed during rendering and both processes take longer the deeper the trees become. Therefore, an additional termination criterion is introduced that prevents the quad-trees to grow deeper than a certain limit, even if the leaf capacity of the leafs is exceeded. Both termination criteria can be adjusted by the user to control the maximum resolution of the quad-trees and the time consumption during preprocessing and rendering.

So, if one termination criterion is met, a leaf node is reached, which has to be classified according to the three leaf node types already defined by Frasson et al. [FEP18], which are either fully inside, fully outside, or partially inside the polygon. If no segment is assigned to a leaf node, it is either completely inside or outside of the polygon. This is clarified with a final point-in-polygon test with the center position of the leaf node. To make this information accessible during the render process, the node type t_{node} is stored as integer value for all nodes of all polygon quad-trees, where 0 indicates an internal node, 1 a fully outside leaf node, 2 a fully inside leaf node, and 3 a partially inside leaf node. The node type t_{node} stored in a buffer for node types and offsets is used during rendering to determine if a pixel is located in a completely inside or outside leaf node and to quickly decide whether the pixel lies inside or outside the polygon.

To all other leaf nodes, polygon segments are assigned and therefore they are partially inside the corresponding polygon. For these leaf nodes, small leaf polygons are created by clipping the polygon segments, assigned to the leaf node, at the node boundary. Afterwards, closing segments are added to complete the leaf polygon. The resulting xy-coordinates of the leaf polygon vertices are then stored in 2D float vectors $float(x_v, y_v)$ in a leaf polygon vertices buffer. These vertex positions are then referenced by leaf segment indices $int(i_{startV}, i_{endV})$, containing the indices to two segment end positions, such as for the static line segments. These segment indices are not stored per cell, each leaf polygon segment is referenced only once by a 2D integer index vector. During the render process, the leaf polygon data is accessed to execute point-in-polygon tests between the pixel and the leaf polygon. Therefore, leaf segment index offsets $int(n_{prevSeg}, n_{curSeg})$ are stored per leaf node that point to all segments of the corresponding leaf polygon. The leaf segment index offsets can be accessed by the offset i_{offset} stored per partially inside leaf node together with the node type t_{node} in a 2D integer vector in a node types and offsets buffer. To be able to display polygons with individual colors, either polygon colors or values for a color mapping are stored in corresponding properties per polygon buffers. As for static lines, the RGBA colors are stored in 4D float vectors $float(r, g, b, a)$ and the values for the color mapping are stored as floats.

After the generation of the polygon quad-trees, the quad-tree data are assigned to intersecting cells of the decal grid. The indices of exterior and interior polygons intersecting a cell are grouped per cell and stored as integer values $int(i_{polygon})$ in separate polygon indices of cells buffers. To know where the interior and exterior polygons of a cell are located in these index buffers, additional index offsets $int(n_{prevPoly}, n_{curPoly})$ are stored per cell referring to the indices of both polygon types. The quad-tree culling process proposed by Frasson et al. [FEP18] enables skipping of quad-tree nodes that are not relevant for the assigned cell. This is implemented with a culling offset $int(i_{startNode})$ stored per polygon assigned to a cell, which points only to the sub-tree that is inside the cell. The offsets are stored in a culling offsets of cells buffer for interior and exterior polygons to skip the rest of the corresponding quad-tree during the render process. The quad-tree traversal can be restricted only to this relevant sub-tree.

Bounding Volume Hierarchies

After the generation of the decal grid and the assignment of the generated line segment and polygon quad-tree data to intersecting grid cells, the line segments and exterior polygon quad-trees are organized in BVHs to avoid looping over all cell decals during rendering. The interior polygons are referred by their exterior polygons and are therefore not organized in BVHs. The BVHs are fully balanced binary trees storing the line segments and exterior polygon quad-trees of a cell in the leaf nodes. Since the lines and polygons are rendered separately, the data are also stored in separate BVHs, but they are generated in the same way. Due to the independence of the individual cells, the BVHs are created in parallel per cell.

At first, the cell decals are sorted because the order in which they are stored in the BVH determines the drawing order, the first object is processed first and drawn on top. The sorting can be done according to different criteria, such as the line width or length. Thöny et al. [TBP17] sort the line segments according to their midpoints along a space filling curve. Line segments of the same line can have a different rank in the drawing order of the corresponding cells. Then it may occur that intersecting lines are drawn in a different order in different cells, because the segments of a line are not consistently sorted across the cells. To keep a certain order that is consistent for all segments of a line over all cells, the sorting should not be based on individual line segments but on line properties, such as the line importance. If there are no additional importance criteria, the line segments are sorted according to the lengths of the corresponding lines and the polygon quad-trees according to the areas of their polygons. This is done because long lines and large polygons are supposed to be more important, for example if they represent roads or buildings and therefore should be rendered on top of other decals. Furthermore, larger decals cover a larger area and thus the probability increases that a pixel belongs to them. Since they are processed first, it is more likely that a decal covering a pixel is found faster. The sorting process is done in parallel per cell for the grouped line segment indices or polygon indices of a cell on the CPU with the *sort*() operation of the C++'s Standard Template Library (STL).

After the sorting of the per-cell indices, the corresponding line segments and polygons are assigned to leaf nodes. To facilitate an easy BVH traversal during rendering by implicit indexing of the tree nodes, a full binary tree is desired, therefore empty leaf nodes are added until their number is a power of two. These empty leaf nodes are the right sibling nodes of the non-empty leaf nodes and do not refer to any decal data. The non-empty leaf nodes refer to the objects assigned to the cell, which are either line segments or polygons. The AABBs of the cell objects of two consecutive leaf nodes are merged and assigned to the parent nodes. This merging process is continued to produce all internal nodes until the root node of the BVH is reached. The root node and all internal nodes only refer to the merged AABBs of all cell objects stored in the leaf nodes below them. During rendering, multiple decals can be skipped at once by checking if a pixel lies outside of their common AABB. Therefore, the xy-coordinates of the minimum and maximum positions of all node AABBs are stored in 4D float vectors $float(x_{min}, y_{min}, x_{max}, y_{max})$ inside a node AABBs buffer. To access all AABBs associated with the line or polygon BVH of a cell during the render process, index offsets $int(n_{prevNodes}, n_{curNodes})$ referring to the BVH nodes of a cell are stored in a node index offsets buffer per cell. These index offsets point to all node indices $int(i_{AABB})$ of the BVHs, which refer further to AABBs or indicate an empty node and the end of the tree traversal with the negative index -1. Since the number of non-empty leaf nodes and the number of cell objects is the same, the leaf node data, which are the line segment and polygon indices of a cell, can be accessed by the corresponding leaf node indices. Before that, the number of internal BVH nodes $n_{internalNodes}$ has to be subtracted from the leaf node indices and thus the number of internal nodes is also stored per cell during preprocessing.

4.2.2 Terrain Matching

The data preparation during preprocessing is limited for dynamic lines due to their view-dependency. In Figure 4.3 one can see all data necessary for the rendering of dynamic lines stored in shader storage buffers, where only the line vertex positions and properties per lines are processed during preprocessing.

The initial vector lines are defined by 2D vertices and the static lines are transformed into the 3D space when they are projected onto the terrain surface during the render process. For dynamic lines the height information of the terrain is already added during preprocessing to enable the generation of 3D line segment boxes during the dynamic updates, which are used to limit the area on the terrain surface that may be covered by the lines. These segment boxes are based on the 3D segments created by the terrain matching process, because they represent the center line of the boxes. The process can be divided into two individual steps, the line subdivision and the sampling of the lines on the terrain surface, which are mainly executed on the GPU by using compute shaders.

Line Subdivision

To be able to adapt the initial vector lines to the terrain surface, they are subdivided according to the minimum terrain resolution. Therefore, a maximum line segment length is defined, which is half as large as the terrain resolution. Every segment is divided into equal sized segments until the subdivided segment lengths are shorter than the maximum length. If the line segments of the initial vector lines are larger than the maximum line segment length, the subdivision process leads to a higher number of vertices. Since the process is executed in parallel, the vertex offsets per line segment have to be determined first, to write the new vertex data to the correct memory locations. The number of subdivisions can easily be calculated by the division of the line segment length by the maximum length and the integer part of the division result then determines the number of additional line vertices. This step is done in parallel for every line segment, but the

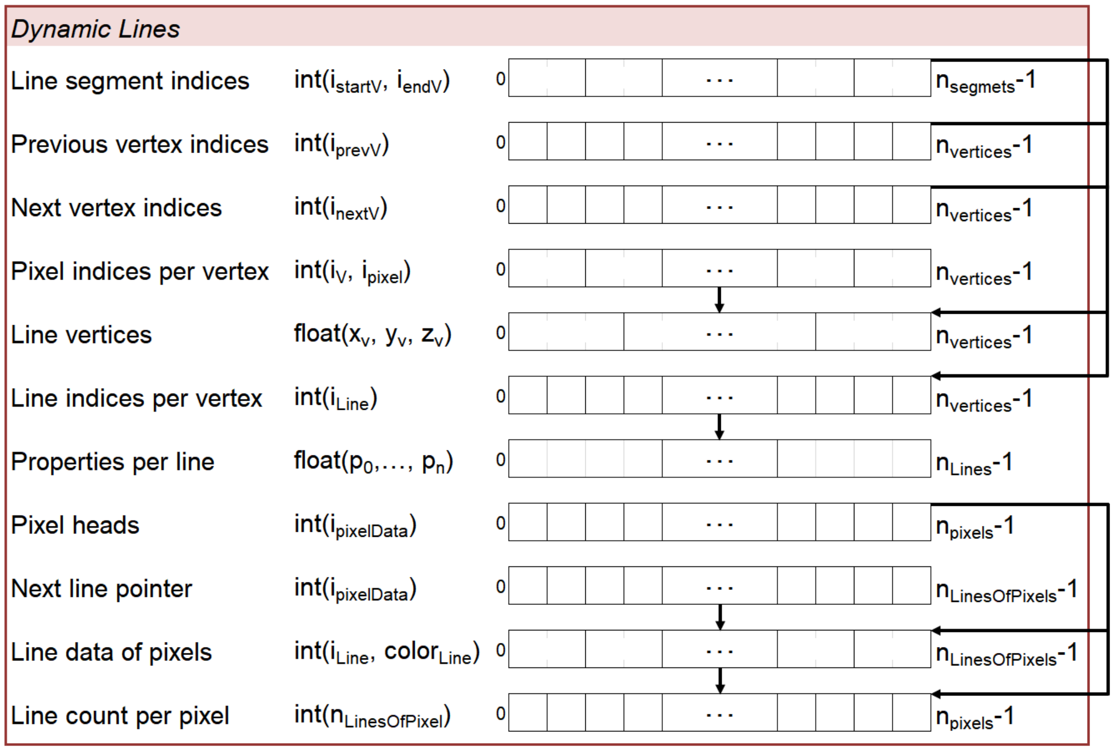


Figure 4.3: All data that are used to render dynamic lines with data types and sizes are generated and stored during preprocessing and dynamic updates in shader storage buffers. They contain indices i, positions with xyz-coordinates, and colors.

final segment offset calculation is executed sequentially on the CPU by summing up the number of subdivisions of all previous line segments. The execution of a parallel prefix sum on the GPU to calculate these offsets could further accelerate this step.

The resulting offsets are required for the next step, which is the calculation of the new intermediate 2D line segment vertices. Therefore, all dynamic line buffers of Figure 4.3 depending on the number of line vertices $n_{vertices}$ have to be resized according to the new number of vertices. This includes the buffers containing line vertices, previous and next vertex indices and line indices per vertex. The already calculated offsets then determine the buffer location where the new data are written in parallel. The initial line segments are then subdivided by inserting new 2D vertices at equal intervals along the segment, which are then stored according to the determined vertex offsets. The offsets are also used to store the index of the first vertex of polygon outlines per line in a separate line property buffer. This information is stored additionally to the line properties, i.e., width, color, and color mapping values as for static lines described in Section 4.2.1. The first vertex index of a polygon outline is required for the index update process, which is performed during the dynamic update to store the neighborhood information of a vertex. The neighborhood information is then used during the pixel data generation step to determine the orientation of the 3D segment boxes. Another property that is stored per line is the drawing order, which is implicitly stored for static lines by the order of the BVH leaf nodes, but is explicitly stored for dynamic lines by inserting the order as integer values per line in a corresponding properties per line buffer. The drawing order is then used during the dynamic updates to sort the line data of a pixel. After this step the dynamic line vertices are still in 2D, but they have the correct resolution to be matched with the terrain in the next step.

Line Sampling

After the subdivision step, the dynamic lines can easily be sampled on the 3D terrain surface. Therefore, one sample from the terrain's heightfield is taken at the position of each newly created line vertex. The terrain height is stored together with the 2D line vertices in 3D vectors $float(x_v, y_v, z_v)$, which are ready to be accessed in the dynamic update process to generate 3D line segment boxes. As most other steps of the terrain matching, this line sampling step is also executed in parallel on the GPU by performing one compute shader invocation per vertex. This makes the preprocessing of dynamic lines much more efficient than the CPU processing of the static decals.

After the terrain matching the 3D line vertices and the line properties, i.e., width, color, color mapping values, outline start indices, and drawing order, are stored in corresponding shader storage buffers and can be accessed during dynamic updates and rendering of the dynamic lines.

4.3 Dynamic Update

Due to the view-dependency of dynamic lines they have to be continuously updated and the dynamic update is executed every time before the lines are rendered. As Figure 4.1 shows, the process consists of three steps, which are all performed on the GPU and described in more detail in the following sections.

4.3.1 Update Indices

Due to the subdivision of the dynamic lines during preprocessing, they consist of more vertices than the static lines do, which leads to a higher memory consumption and to a more time-consuming rendering. The 3D line vertices produced during the terrain matching process are therefore reduced by line simplification and view frustum culling, which remove indices to vertices that are irrelevant for the current view.

The lines are simplified by detecting consecutive vertices that are projected to the same pixel after perspective transformation in the current view. Then all indices to points whose predecessors are on the same pixel are removed. This simplification does not produce a visual difference of the lines because the change inside one pixel cannot be recognized anyway. The first step to achieve this screen-based simplification is to determine the pixel positions of all vertices. This is done in parallel per vertex using a compute shader, which calculates the pixel index i_{pixel} and stores it together with the vertex index i_v in a pixel indices per vertex buffer, shown in Figure 4.3, for the next step. Although, the vertex index is implicitly given by the position, where the corresponding pixel index is stored, the vertex index is stored explicitly to the pixel index, because the positions change by the next step.

The next step is a stream compaction, which filters the line vertices based on their corresponding pixels. To execute this stream compaction the *unique*() operation of the *Thrust* library [thr] provided by *CUDA* [cud] is applied to the pixel indices of the pixel indices per vertex buffer. It removes all consecutive entries with the same pixel index and therefore discards all vertices that are located at the same pixel as their previous vertex. Afterwards, the line segment indices buffer has to be updated by adding only line segment indices $int(i_{startV}, i_{endV})$ that refer to remaining vertices, which are not removed by the stream compaction. This is executed in parallel per vertex by another compute shader that writes only remaining line segment indices into the line segment index buffer. In order to have access to the neighbors of vertices during the pixel data generation process, the indices for predecessors i_{prevV} and successors i_{nextV} are also stored per vertex. This neighborhood information is necessary to generate 3D segment boxes without overlaps with previous and next segment boxes and to determine the corner positions for the different corner styles. To store the neighborhood information of polygon outlines, which are closed lines, correctly, the last vertex index has to be stored as predecessor i_{prevV} of the first line vertex and the first vertex index as the successor i_{nextV} of the last line vertex. Due to the stream compaction, the first line vertices are always the same but the last line vertices may vary. A last line vertex is found if the next vertex, stored in the pixel indices per vertex buffer, belongs to another line. If a last line vertex is detected, the line start index stored during preprocessing per polygon outline is used to store the predecessor and successor vertex indices accordingly.

During this update process, all vertices that are not already removed by the line simplification, are also tested if they are inside the view frustum. The corresponding line segment indices are only stored if at least one of the two vertices is inside. This prevents line segments outside the visible area of the current view to be processed by the next steps of the dynamic update and they are also excluded from the render process.

Unfortunately, the described view-dependent line simplification could not be integrated properly into the target framework because of pre-existing interoperability problems between the used APIs *OpenGL* and *CUDA* that we cannot fix. The pixel indices per vertex buffer of Figure 4.3 is an *OpenGL* shader storage buffer, because its data are used by *OpenGL* compute shaders to prepare the vertex index data for rendering, which is also performed by *OpenGL* shaders. *CUDA* is used to execute the stream compaction on the same data, which cannot be used properly by *OpenGL* afterwards, unless the contents of the shared buffer are copied to another buffer afterwards, which is costly and negates the benefits of the line simplification. A solution to this problem would be an own implementation of the stream compaction to avoid the *unique*() operation of *CUDA*. Since the implementation of a stream compaction goes beyond the scope of this thesis, it is a topic for future work.

Even if this line simplification cannot be applied, the view frustum culling is still executed and can eliminate line vertices early in the visualization process. The line vertices can be especially reduced in close-up views, where many vertices are outside the visible area. Such a view frustum culling is not necessary for static decals because the render process of static decals is reversed compared to the one for dynamic lines. The dynamic lines approach is based on the line segments, which are passed to the pixel data generation process, where it is determined which pixels they cover. The render process for static decals starts with the individual pixels to which line segments and polygon parts are assigned by checking all possible candidates located at the same decal grid cell. Thus, all static decals corresponding to decal grid cells outside the view frustum are directly omitted.

4.3.2 Pixel Data Generation

After sorting out the line segments outside the view frustum, the pixels covered by the individual line segments have to be determined. To avoid checking all line segments per pixel, the area which can be affected by a line segment is restricted first. Therefore, the remaining 3D line vertices from the index update step are passed to a geometry shader, which generates 3D boxes for each line segment to roughly limit their coverage area. The 3D line segment boxes, shown in Figure 4.4a, enclose the terrain areas that can be affected by the corresponding line segments. Thus, an affected area can be limited to pixels inside a box and accurate point-in-line tests have to be executed only for those pixels. These tests are performed in a fragment shader that stores all lines covering a pixel in linked list for rendering.

3D Segment Boxes

The 3D segment boxes are defined by a width, height, and length, which are calculated in parallel per line segment in a geometry shader. The width of the segment boxes is determined by view-dependent scale factors of the two segment vertices. The scale factors are defined by the distances of the vertices to the virtual camera and therefore represent the minimum and maximum width of a line segment because the width changes continuously between them. The concrete calculation depends on the projection type of the virtual camera, which can be perspective or orthographic. In both cases the camera field of view (FOV) and the distance between line and camera are taken into account to get an appropriate scale factor at every location of the 3D scene. The FOV of the camera defines the visible area of the current scene by an horizontal and vertical angle. In Equation 4.1 the scale factor $orthoScale$ is calculated for orthographic views by dividing the camera zoom factor $zoom_c$ by the tangent of half of the longer side of the FOV, which is indicated by FOV_{max} in the formula. The resulting term is then multiplied by the tangent of half of the shorter side of the FOV, specified with FOV_{min} in the formula. The zoom factor is determined by the distance between the virtual camera and its focus point.

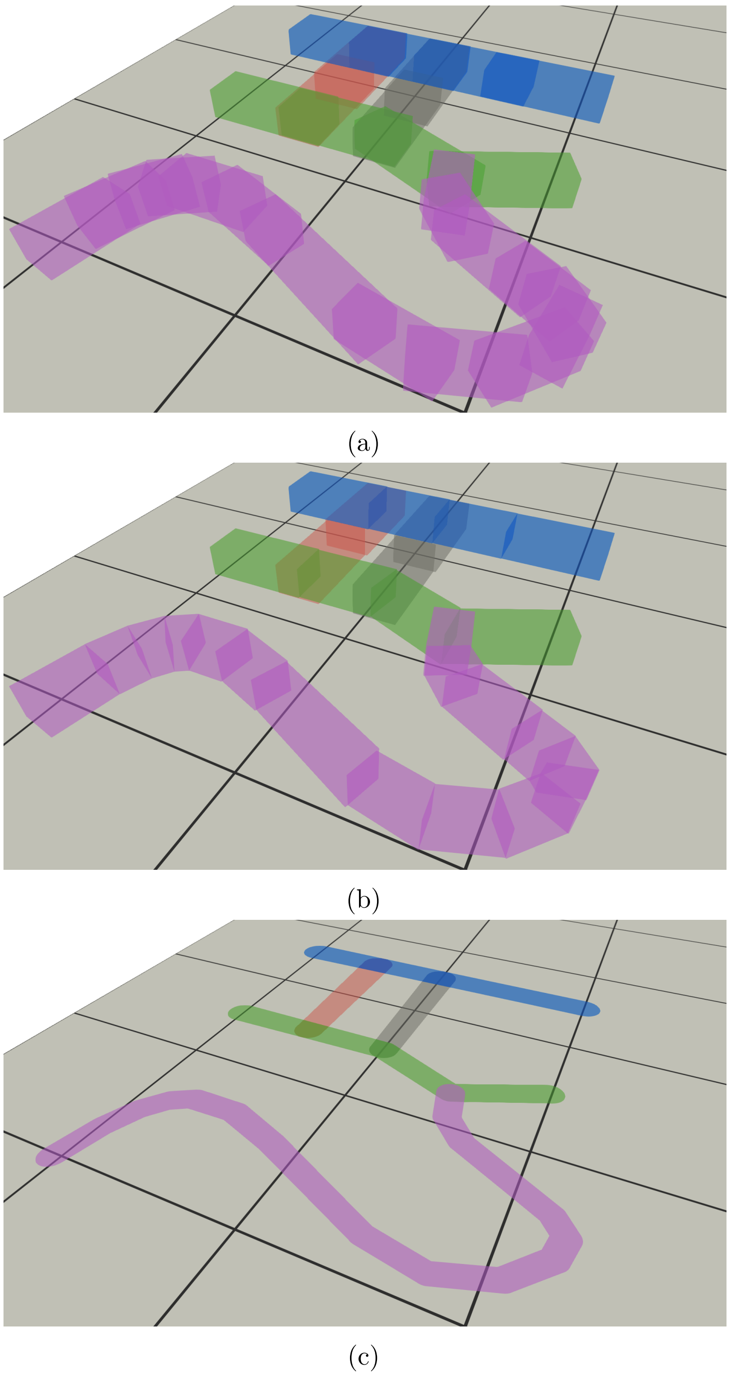


Figure 4.4: 3D segment boxes of semi-transparent dynamic lines generated (a) with a corner offsets, which cause large overlapping areas and (b) connected with common miter surfaces to avoid box overlaps. (c) The resulting line segments without overlaps between line segments of the corresponding line.

$$orthoScale = \frac{zoom_c}{tan\left(\frac{FOV_{max}}{2}\right)} \cdot tan\left(\frac{FOV_{min}}{2}\right) \tag{4.1}$$

One can see that this scale factor depends only on the current view and remains constant for all positions on the screen. This is expected because with orthographic projection the relative object sizes are preserved and the decals have to be scaled constantly without perspective distortion. Therefore, the orthographic scale factor is calculated once per view on the CPU and passed to the geometry shader and fragment shader that use the same scale factor for the pixel data generation. The segment boxes are scaled constantly with respect to the screen by multiplying the orthographic scale factor with a predefined line width stored in a properties per line buffer.

The scale factor $perspectiveScale$ cannot only be calculated for the end points of the line segments and interpolated linearly, because the perspective distortion of perspective views and the depth change of the terrain along a line segment are not linear. The perspective scale factor has to be calculated for each reference point p on the line with the formula of Equation 4.2. But, for the segment boxes only the maximum scale factor of a line segment is used to ensure that the whole line segment is contained after its perspective projection. The maximum scale factor of a line segment is determined by the segment vertex furthest away from the camera, because the perspective scale factor grows with the distance between the reference point p and the virtual camera c.

$$perspectiveScale = dist\,(p, c) \cdot tan\left(\frac{FOV_{min}}{2}\right) \tag{4.2}$$

To ensure that the terrain is included in the segment boxes, their minimum and maximum height corresponds to the height range of the terrain. A small additional offset is used to avoid segment boxes with zero height for flat terrains and to avoid z-fighting between the boxes and the terrain surface. The length of a segment box corresponds to the segment length plus an offset of half the scaled line width to include the gap at the corners. This offset produces large overlapping areas at the corners of interior segments, which can be observed at the semi-transparent 3D segment boxes in Figure 4.4a. Since a line should affect a pixel at most once, overlaps of line segment boxes of the same line have to be avoided. Therefore, a common front and back surface for two consecutive segment boxes is determined. For this purpose, the miter vectors of the corners are calculated as for the miter corner style described in Section 3.6 by using the predecessor and successor vertex of the segment. The neighbor vertices are accessed with the indices i_{prevV} and i_{nextV} stored at the current vertex index position in the previous and next vertex index buffers. A miter vector defines the orientation and influences the size of the touching surfaces of the two adjacent boxes. In Figure 4.4b one can see that the use of these miter surfaces prevents overlaps between the individual line segment boxes in world space and reduces the overlapping areas in screen space.

A so generated segment box consists of eight vertices and with the optimization for fast rendering of triangle strips by Evans et al. [ESV96], a triangle strip with fourteen indices is generated in the geometry shader and emitted to the fragment shader. Besides the vertices of the segment box, the corner positions for the miter or bevel corner styles are also passed to the fragment shader if the dynamic lines should be displayed in one of these styles. The miter vectors are already calculated for the miter surfaces of the segment boxes and therefore only need to be passed. The bevel vectors have to be calculated additionally, which is done the same way as for the static lines described in Section 3.6, which are then also passed to the fragment shader.

Linked Lists

The fragment shader is not an ordinary fragment shader that writes pixel colors to an output framebuffer, but instead stores the indices of all lines covering a pixel in a linked list realized by three shader storage buffers, i.e., the pixel head, next line pointer, and line data of pixels buffers of Figure 4.3. Therefore, the fragments corresponding to the 3D segment boxes from the geometry shader are further analyzed if they are part of the line projected onto the terrain. This is done by point-in-line tests according to the corner style by using the pixel center position and the line segment, which is defined by two vertices, a scaled line width, and corner positions described in Section 3.6. Since the test is done in world space, the pixel center position has to be projected back from screen space to world space first. This is accomplished by using the terrain depth, which is obtained from its depth texture at the screen-space pixel position.

For the round corner style, the two segment vertices and the scaled line width are required. The predefined line width stored in a properties per line buffer is scaled by using the scale factor of the current pixel that is calculated with the same equations as the scale factor for 3D segment boxes by Equation 4.1 for orthographic views and by Equation 4.2 for perspective views. Instead of using the more distant line segment vertex, the distance between the world-space pixel position and the camera is used to determine a correct scale factor for every pixel. The scale factor is then used to scale the line width, which is used to determine if a pixel lies inside the line segment. This is the case if the distance between the world-space pixel position and the segment center line is smaller than half the scaled width. Two point-in-triangle tests are necessary to determine if the world-space pixel position is located inside a line segment in the miter corner style. Therefore, the segment is divided into two triangles that are spanned by the two miter vectors of the two segment vertices. The length of the miter vectors is also scaled by the view-dependent scale factor. Three tests are performed for the bevel corner style. The first test checks if a pixel lies inside the segment rectangle without the corner gap, the other two tests are only executed if the pixel lies outside this rectangle to determine if it is part of one of the two triangles at the bevel corners. The corner triangles are defined by the segment vertices and their corresponding bevel vectors, calculated in the geometry shader. The view-dependent scale factor is also used for this corner style to scale the segment rectangles and the corner triangles appropriately.

If the corner tests prove that a pixel is part of a line segment, the corresponding line index i_{Line} is saved by adding it to the linked list of this pixel, otherwise the pixel is not covered by the line segment and the fragment can be discarded. Additional to the line index i_{Line} also its color $color_{Line}$, packed into an integer, is stored in the linked list to reduce the number of memory accesses during sorting and rendering. Furthermore, the storage of line colors per pixel can be used by an analytical anti-aliasing method to adjust the transparency of individual colors per pixel, which is another topic for future work.

Since the number of lines covering a pixel is not known in advance and the data are stored in the corresponding shader storage buffer in parallel, an atomic counter is used to count the number of already stored line fragments. The counter then determines the index to the next free storage location by incrementing the fragment counter with an *atomicAdd*() operation for every stored line data entry of a fragment. This procedure can lead to a higher index than possible for the allocated line data buffer. The process is stopped in this case to resize the buffer. To avoid frequent buffer resizing, it is enlarged by one and a half times of the already counted line data entries. This resize process is usually executed during the first frames and does not lead to noticeable waiting times. After allocating enough memory to store all line data of all pixels, the data are stored according to the atomic fragment counter at the next free memory location.

To be able to access all line data corresponding to a pixel during rendering, the individual entries are connected by linked lists per pixel. The start of the list is indicated by a pointer to the first list entry, called head. The pointer is represented by the index $i_{pixelData}$ of the first list entry and is stored in a pixel head buffer per pixel. The pixel head is updated for every new fragment of a line segment covering the pixel. Since every fragment corresponding to a pixel writes to the same location in the pixel head buffer, an *atomicExchange*() operation with the new and the previous pixel head is used to ensure synchronous updates of the pixel head. The new pixel head is determined by the index provided by the atomic fragment counter that points to the last processed line data. To prevent losing the pointer to the previous line data stored in the old head, the old head index is stored in the next line pointer buffer at the location where the new head points to. The pixel head buffer, the next line pointer buffer and the line data of pixels buffer represent the linked list containing all lines covering a pixel. The end of the linked list is reached if the next line pointer points to the negative index -1, which is the initial value for all elements of the buffer. Since the number of list entries is required for the sorting of the linked list to determine the number of sorting passes, another atomic counter is used to count the number of list entries per pixel $n_{LinesOfPixels}$. This information is stored in the line count per pixel buffer.

The generated line data lists of individual pixels can be accessed during rendering by a pixel index that refers to the pixel head $i_{pixelData}$ stored in the pixel head buffer. The pixel head points to the first line data of pixels entry $int(i_{Line}, color_{Line})$ of the list stored in the line data of pixels buffer. The index $i_{pixelData}$ of the next line data entry is stored in the next line pointer buffer at the same index as the current line data is stored. Before

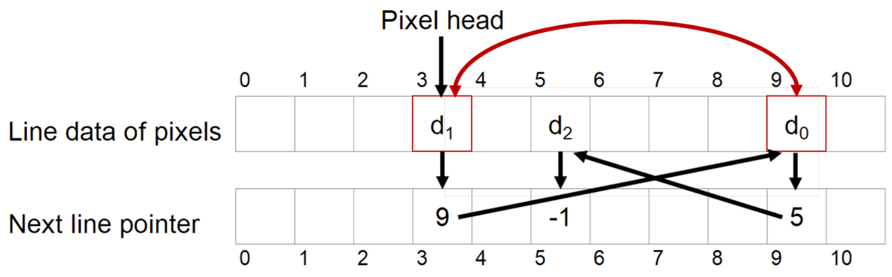


Figure 4.5: The sorting process of a linked list with three entries in the line data of pixels buffer with the corresponding line drawing orders d_{0-2}. During the sorting process only the data entries of the lines with drawing order 1 and 0 are swapped for a consistent line drawing order and the pixel head and next line pointer stay the same.

the generated per-pixel linked lists are used for rendering, they are sorted in the last step of the dynamic update process.

4.3.3 Line Sorting

After storing all lines that possibly contribute to a pixel in random order, they have to be sorted to ensure a consistent line drawing order over all pixels. Only if all lines have the same color and opacity, the sorting can be skipped and the dynamic lines can be directly rendered. A selection sort is performed for each linked list as described in Section 3.3.3. The sorting of all line data of a pixel is executed by a compute shader in parallel for each list.

Figure 4.5 shows an example of a linked list with three entries in the line data of pixels buffer, which are not stored in the drawing order d_{0-2} of their corresponding lines. The first entry of the list can be accessed by the index stored in the pixel head buffer at the pixel index location. All indices to the following line data entries are stored in the next line pointer buffer and can be accessed by the index of the previous line data entry. The next line pointer of the last list element contains only -1 to indicate the end of the list and does not reference any data. To reduce the number of memory accesses it is avoided to look up this last pointer in every sorting pass by using the number of list entries stored in the line count per pixel buffer. The sort iterations are stopped after processing the last line data entry if the number of list entries is reached. The line count per pixel buffer is only accessed once before the sorting process and it is no longer necessary to look up the next line pointer of the last list element in every sorting pass.

In the example of Figure 4.5 two list entries have to be swapped. This is done by re-sorting the line data directly and not their references because this would cause an update of the head and of two next pointers even in this simple example. The sorting procedure can be executed in real time because the re-sorting is only done inside a linked list and no expensive atomic operations are necessary. Furthermore, the overlaps between dynamic lines are rather small and do not lead to linked lists with many entries. An unnecessary iteration through all list elements should still be avoided. Therefore, after each re-sorting, it is tested if the combined color of all already sorted list elements corresponding to different lines is already opaque. Then the sorting can be terminated and all other list elements can remain unsorted because they are not used during rendering. The sorting of the linked lists concludes the dynamic update process of dynamic lines and the prepared line data can be passed on to the render process.

4.4 Rendering

The preprocessing and dynamic update have already prepared the vector data so that the static and dynamic decals can be rendered efficiently. The render pass of the decals is executed after rendering the terrain to be able to render the decals on it and to have access to its color, depth, and normal textures.

4.4.1 Static Decals

The main steps of the render process for static lines and polygons and their inputs and outputs are presented in Figure 4.6. The steps in blue are performed for both, lines and polygons, the step in orange is only executed for lines, and the steps in green only for polygons. The polygons are rendered in a separate render pass. Afterwards the outlines are rendered over them.

The first step for both static decal types is the determination of the location of the pixel center position on the decal grid in a certain cell. The detected grid cell is then used to access all relevant line and polygon data of this cell. To accomplish that, the terrain depth t_{depth} is obtained from its depth texture at the screen-space pixel center position $pixel_{ss}$. This depth is used to project the pixel back from screen space to world space. The world-space pixel position $pixel_{ws}$ is then used together with the grid origin and the cell size to determine the 2D grid index i_{grid} with the Equation 4.3. The cell index i_{cell}, which is necessary to access the line and polygon BVH data, can then be calculated with the grid index and the grid dimension according to Equation 4.4.

$$i_{grid} = floor\left(\frac{pixel_{ws}.xy - gridOrigin}{cellSize}\right) \tag{4.3}$$

$$i_{cell} = i_{grid}.x + gridDim \cdot i_{grid}.y \tag{4.4}$$

If the determined cell index is negative or larger than the total number of grid cells, the pixel lies outside the grid and is not covered by static decals. Such empty fragments can be discarded. Otherwise, the grid cell index refers to all line segments $segment_L(i_{startV}, i_{endV})$ and to all polygon indices $i_{polygon}$ covering the grid cell. The cell index is also used to access the line and polygon BVH data stored in shader storage buffer during preprocessing.

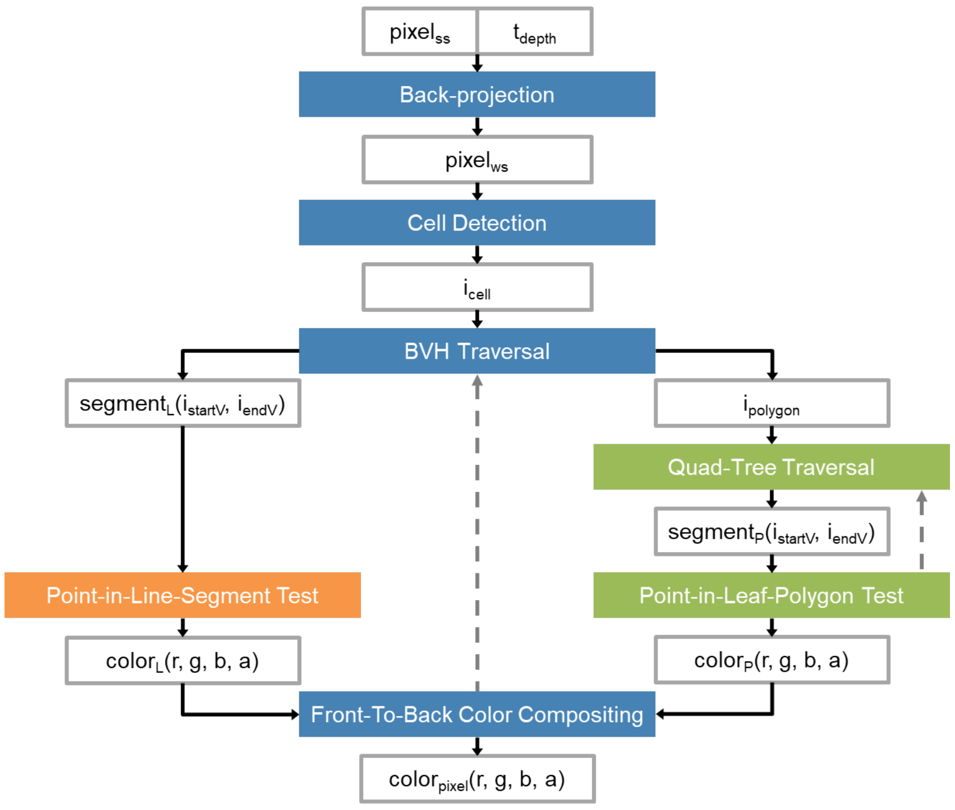


Figure 4.6: The individual steps of the render process of static decals with their input and outputs (gray). The steps that are performed for lines and polygons are highlighted in blue, line-specific steps in orange, and polygon-specific steps in green. Dashed arrows indicate a possible repetition of the steps.

Since the line and polygon BVHs contain all line and polygon candidates that may contribute to the current pixel, they have to be traversed to determine the lines and polygons that actually lie on the pixel. Frasson et al. [FEP18] use a stack-based algorithm for the BVH traversal, which is reported to be faster by Áfra and Szirmay-Kalos [ÁSK14]. The implementation of their algorithm is based on *CUDA* but since its usage combined with *OpenGL* led to problems, we use a stack-less implementation with an *OpenGL* fragment shader based on a while-loop.

All required data of the BVH nodes are accessed by the grid cell index i_{cell} as described in Section 4.2.1. The BVH traversal starts with the root node and its corresponding AABB, which is the merged AABB of all relevant line segments or polygon parts of the detected grid cell. In a while-loop the BVH is traversed along the left child nodes as long as the pixel lies inside the corresponding child node AABBs. Since the BVH is a fully balanced binary tree, the left child node indices are determined by Equation 4.5, which are used to access the node AABBs and the leaf node data. If the pixel does not lie inside of a node AABB, the whole sub-tree can be skipped and the traversal is continued at the next unprocessed right node. For the case that the current node has a right sibling node, it is the next unprocessed right node. Since the BVH is a fully balanced binary tree, all left nodes have right sibling nodes. Left nodes can be detected by their indices because they have odd node indices, so only the node index needs to be checked to determine if a node has a right sibling. The node index of a right sibling can be simply calculated by Equation 4.6. To find the next unprocessed right node starting at a right node it is necessary to go up the tree with Equation 4.7 until a node is found that has a right sibling.

$$leftChildIndex = 2 \cdot parentNodeIndex + 1 \tag{4.5}$$

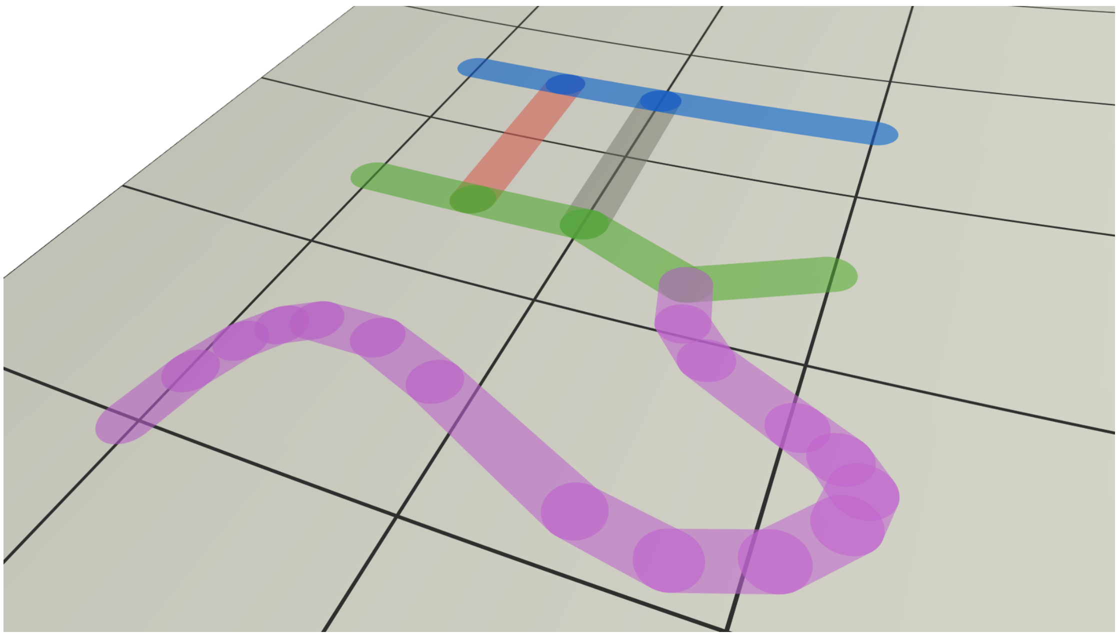
$$rightSibling = leftNodeIndex + 1 \tag{4.6}$$

$$parentNodeIndex = floor\left(\frac{childNodeIndex - 1}{2}\right) \tag{4.7}$$

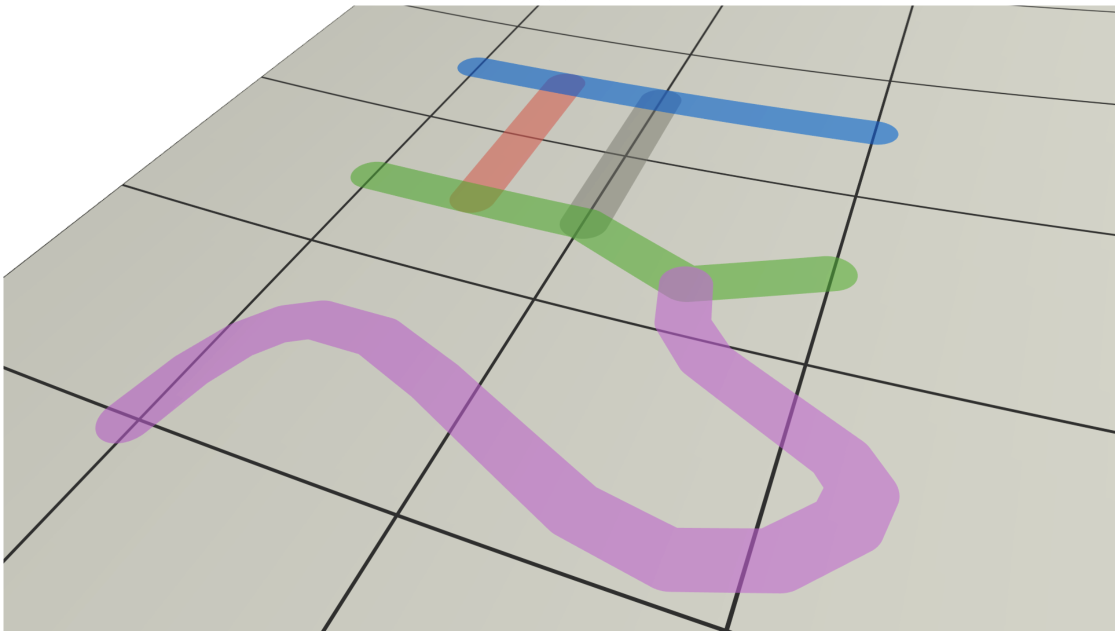
Since the the decal parts stored in the BVH leaf nodes may overlap, the BVH traversal can end at zero or more leaf nodes. If no leaf node is found, the traversal is already terminated at the root node because the pixel center position $pixel_{ws}$ lies already outside the AABB of the root node, which contains all lines or polygons stored in the BVH. If a leaf node is reached, it refers to a line segment $segment_L(i_{startV}, i_{endV})$ or a polygon index $i_{polygon}$ that refers further to the corresponding quad-tree data prepared during preprocessing and shown in Figure 4.2.

Static Lines

For all detected line segments $segment_L(i_{startV}, i_{endV})$ during the BVH traversal, point-in-line tests are performed according to the set line style. The different corner styles are handled the same as for the dynamic lines described in Section 4.3.2. The difference is that for the static lines precalculated corner positions can be used for the miter and bevel corner styles and do not have to be calculated continuously as for the dynamic lines. Since the round corner style is based only on the distance to the center line of the corresponding line segment, it leads to overlaps in the corner area of two segments of the same line, as can be seen in Figure 4.7. To avoid that a line contributes twice to the pixel color at this overlapping area, it has to be checked if there is already a line segment of the same line that covers the current pixel. Only the line index of the previous line segment has to be compared with the line index of the current line segment, because all line segments of the same line are stored consecutively. This is a result of sorting the segments according to their corresponding line drawing order. By comparing the line indices of two consecutive segments, it can be determined if they belong to the same line. In case the line has not yet been processed, it can be tested if the distance between



(a)



(b)

Figure 4.7: Semi-transparent static lines with round corner style and (a) with overlapping segments within a line and (b) without line segment overlaps, which are prevented by checking whether the segments belong to the same line.

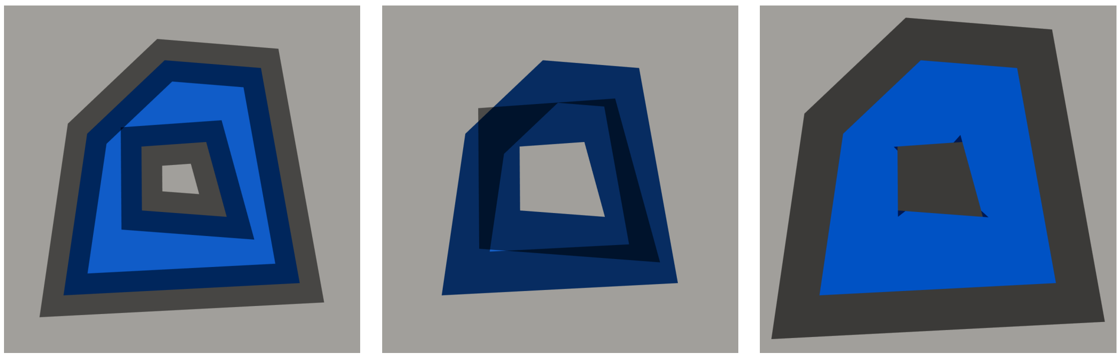

(a) (b) (c)

Figure 4.8: A blue polygon with a wide black semi-transparent outline using the outline modes (a) normal, (b) inset, and (c) offset. The inset and offset modes cause artifacts because, due to the large width, the outline reaches the outside and inside of the corresponding polygon, respectively.

the pixel and the line segment is smaller than half the line width to produce the desired round corner style, as shown in Figure 4.7b.

For the outline modes inset and offset it is tested if the pixel is located to the left or right of the outline. Since the winding order of the polygons is always the same, so is the order of its vertices used for the outline and this left-right-test can be used to determine if a pixel lies inside or outside the corresponding outline. The test delivers correct results in most cases, but especially for wide outlines and thin polygons, this is not the case. In Figure 4.8a one can see a blue polygon with a thick outline in normal mode. Figures 4.8b and 4.8c show two cases where inset and offset outline modes produce visually unpleasant results. For the inset mode this is caused by outlines of the exterior and interior polygon that cover the whole exterior polygon and extend beyond it. The interior polygon is fully covered by its outline in offset mode and it overlaps also the exterior polygon because it is too wide. The correction of this behavior is a topic for future work and can currently only be avoided by the use of thin outlines or by the normal outline mode.

After the execution of all line style tests the corresponding line color $color_L(r, g, b, a)$ is added to the pixel color $color_{pixel}(r, g, b, a)$ with front-to-back compositing if the pixel is detected to belong to a line segment. If the line color is opaque, the pixel color is determined and no further steps are necessary. In case of transparent line colors, the BVH is searched further for line segments covering the pixel. This search process is terminated if the composited pixel color is opaque or the BVH is fully traversed.

Polygons

The BVH traversal provides polygon indices $i_{polygon}$ that give access to the polygon quad-tree data. With the quad-tree node offset $i_{startNode}$ provided by the quad-tree culling process, the traversal of the polygon quad-tree can be limited to the sub-tree that is part of the detected decal grid cell. This sub-tree is also traversed by testing the world-space pixel position $pixel_{ws}$ against the AABBs of the quad-tree nodes until a leaf node is reached. Such a quad-tree leaf node is then either fully inside, fully outside, or partially inside of its corresponding polygon, which can be determined by the node type t_{node} stored in the node type buffer. For the first two cases it can be directly determined if the pixel lies inside the detected polygon or not. The third case leads to an additional point-in-polygon test between the pixel and the leaf polygon, which decides if the polygon covers the pixel or not. The segments $segment_P(i_{startV}, i_{endV})$ of the leaf polygon are tested by using the common even-odd rule for point-in-polygon tests [FVFH90]. The leaf polygons created during preprocessing avoid looping over all segments of the whole input polygon for point-in-polygon tests and limit the number of test iterations to the number of leaf polygon segments.

In the case a pixel is inside a polygon, it is necessary to check if there are interior polygons, which correspond to the current decal grid cell and to the detected exterior polygon. The interior polygons that fulfill these criteria are also stored in quad-trees and they are traversed the same way as their corresponding exterior polygons. To deal with nested interior polygons, the status of a pixel switches between inside and outside with every positive interior polygon test. After all interior polygon candidates are tested and the pixel is classified as inside, the corresponding polygon color $color_P(r, g, b, a)$ is added to the pixel color $color_{pixel}(r, g, b, a)$ by front-to-back color compositing. The final pixel color is determined the same way as it is done for static lines by traversing the BVH until the end of the BVH is fully traversed or the composited color is opaque.

4.4.2 Dynamic Lines

The only data structures that are used for rendering of dynamic lines are the linked lists containing the line data of all pixels sorted according to their drawing order. In the fragment shader a look-up at the line count buffer is made to determine if the pixel has a non-empty linked list and belongs to a line, otherwise the pixel can be discarded. The pixel color of such pixels remains unchanged and corresponds to the terrain color. For pixels covered by one or more lines, the pixel color is determined by the traversal of the line list and front-to-back compositing of the corresponding line colors.

Before a line color is added to the pixel color it has to be checked, if a line already contributes to the color. Despite the use of a common miter surface between line segment boxes to avoid overlaps, it is still possible that lines are added multiple times to the same pixel list. The 3D segment boxes do not overlap in world space, but if they are not parallel to the virtual camera, they overlap in screen space. In Figure 4.4b one can see these overlapping areas indicated by a darker line color. It cannot be avoided to add segments of the same line to the pixel list during the pixel data generation step because they are added in parallel. Since the list entries are already sorted according to their drawing order, which is unique across all lines, all entries of the same line are stored consecutively. Thus, only the previous list element has to be tested if it belongs to the same line. If this is not the case, the line colors are added to the pixel color with front-to-back compositing until it is opaque or the end of the linked list is reached.

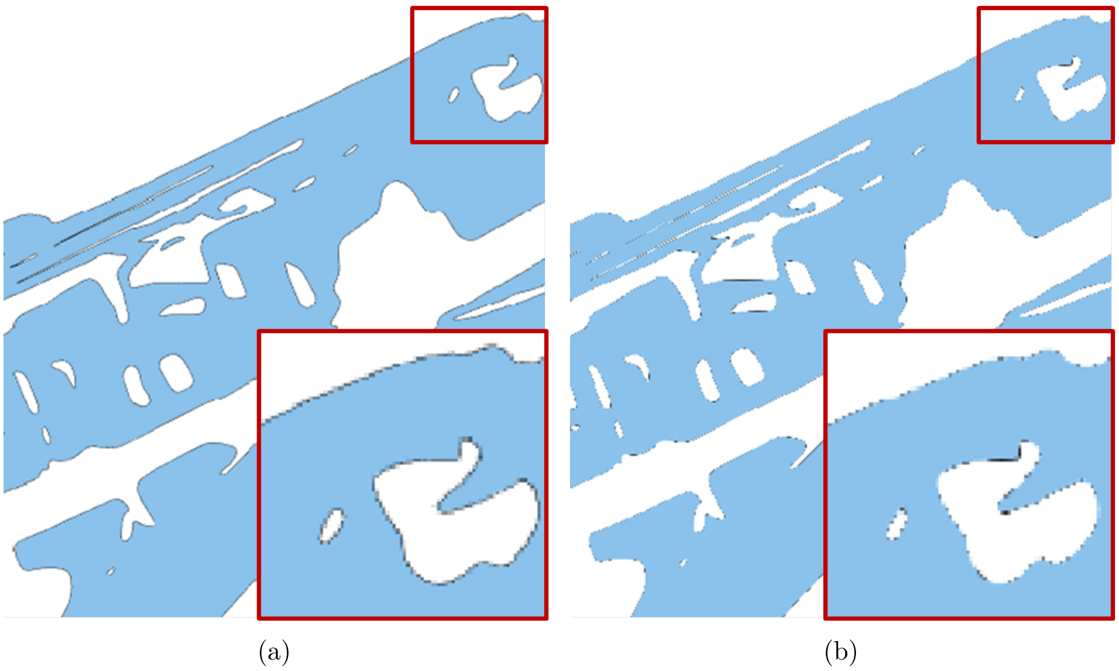
4.5 Anti-Aliasing

The final step of both screen-based visualization approaches, static and dynamic, is the anti-aliasing of the decals. Anti-aliasing is applied before the decals are displayed on the screen to prevent jagged edges at the transition from decal to terrain or to other decals.

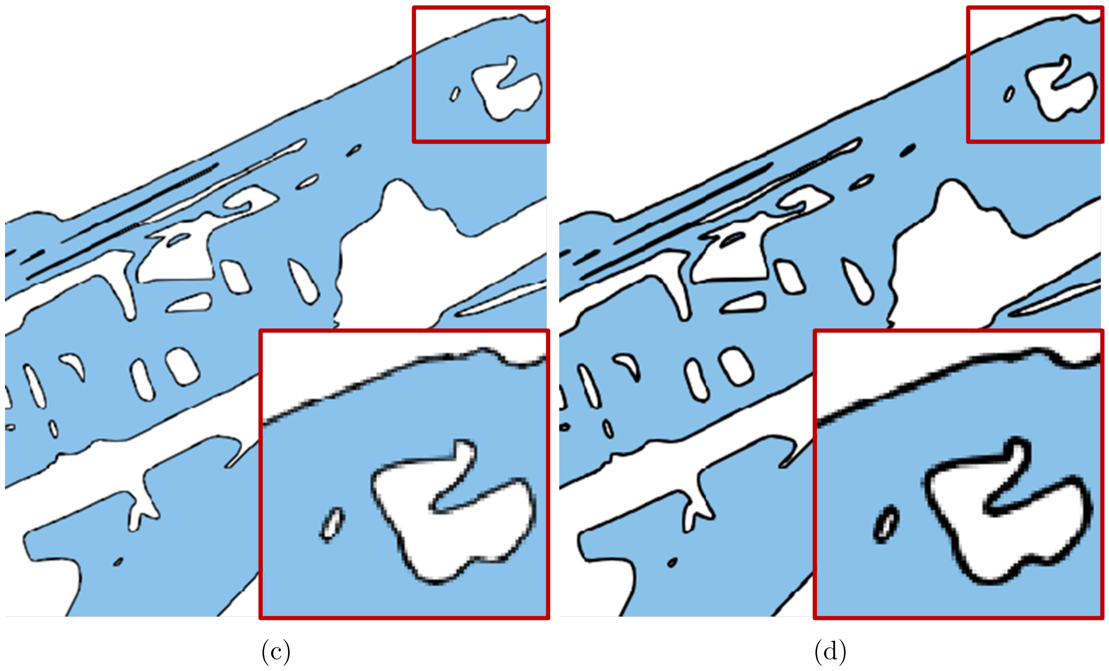
For the anti-aliasing of static decals MSAA is used by activating *OpenGL's* multisampling. This anti-aliasing technique is performed during the render process and is based on the detection of edges of geometric objects, where anti-aliasing is necessary. Only for pixels corresponding to edges, multiple samples are taken and processed and not for all pixels, such as for SSAA. This approach is not applicable to the screen-based decal rendering, because no geometric objects are passed to the *OpenGL* render pipeline and the vector data are only processed in the final render step in a fragment shader and are not available for optimizations before. Therefore, it has to be forced that multiple samples are taken at every pixel by using *OpenGL's* per-sample shading. Through this process the MSAA degenerates to SSAA, making it a less efficient method compared to MSAA applied to geometric objects. But, it produces high quality anti-aliasing results for static lines and as one can see in Figure 4.9a, MSAA is able to reduce the dashing effect produced by very thin lines. It is a common problem that thin lines appear dashed if their widths are getting smaller than a pixel. Anti-aliasing is one way to reduce this problem, but as shown in Figure 4.9b, the used anti-aliasing method for dynamic lines cannot avoid this dashing effect produced by thin dynamic lines.

MSAA can reduce this effect but it is not suitable for the pixel-based approach of the dynamic lines, because the lines and their colors are stored per pixel and not per sample. The storage of line data per sample can cause a massive memory overhead and a medium MSAA taking eight samples per pixel would result in eight times the amount of video memory required to store the data. Besides the high memory consumption, this large amount of data would not only have to be stored during the pixel data generation process, but also sorted and processed during rendering. This time-consuming processing of the data can cause the dynamic lines to no longer be rendered in real time.

To avoid performance issues, a more efficient anti-aliasing approach is chosen, but unfortunately it does not reduce the dashing effect as well as MSAA. The anti-aliasing filter FXAA [fxa] of Nvidia is applied after the rendering of dynamic lines as a post-process. It smooths the jagged line edges and leads to a low memory and time consumption. The main problem of this approach are thin lines, because the lines are rendered with the dashing effect without anti-aliasing and the anti-aliasing afterwards does not work as good as the anti-aliasing of MSAA executed directly during rendering. Therefore, pixels that are less than a pixel width away from a line are detected and classified as line pixels to prevent lines from becoming narrower than a pixel. This correction process is done by projecting the line segment vertices onto the image plane and the resulting



(a) (b)



(c) (d)

Figure 4.9: Blue polygons with thin black outlines appearing as dashed lines due to a small line width. (a) Static outlines, (b) dynamic outlines, (c) dynamic outlines with correction, and (d) dynamic outlines with a larger width.

screen-space positions are used for a distance test between the pixel center position and the line segment defined by the screen-space vertices. If this distance is smaller than half a pixel, the line covers the pixel and is assigned to its linked list. In Figure 4.9b and 4.9c one can see the comparison of dynamic outlines with the same width, without and with the correction applied. The problem with this correction is, that it does not consider the depth of a pixel and would therefore detect pixels as covered, even if the corresponding line is occluded by the terrain. To prevent lines from appearing on the terrain if they are behind, the correction of thin lines is only applied in top-down views. The use of larger line widths, as in Figure 4.9d, is suggested to avoid dashing artifacts in other views. Thöny et al. [TBP17] use a second-depth anti-aliasing method proposed by Persson [Per12] to prevent lines from becoming thinner than a pixel. The method uses a second depth buffer for silhouette detection, which can be considered for future work.

CHAPTER 5

Evaluation

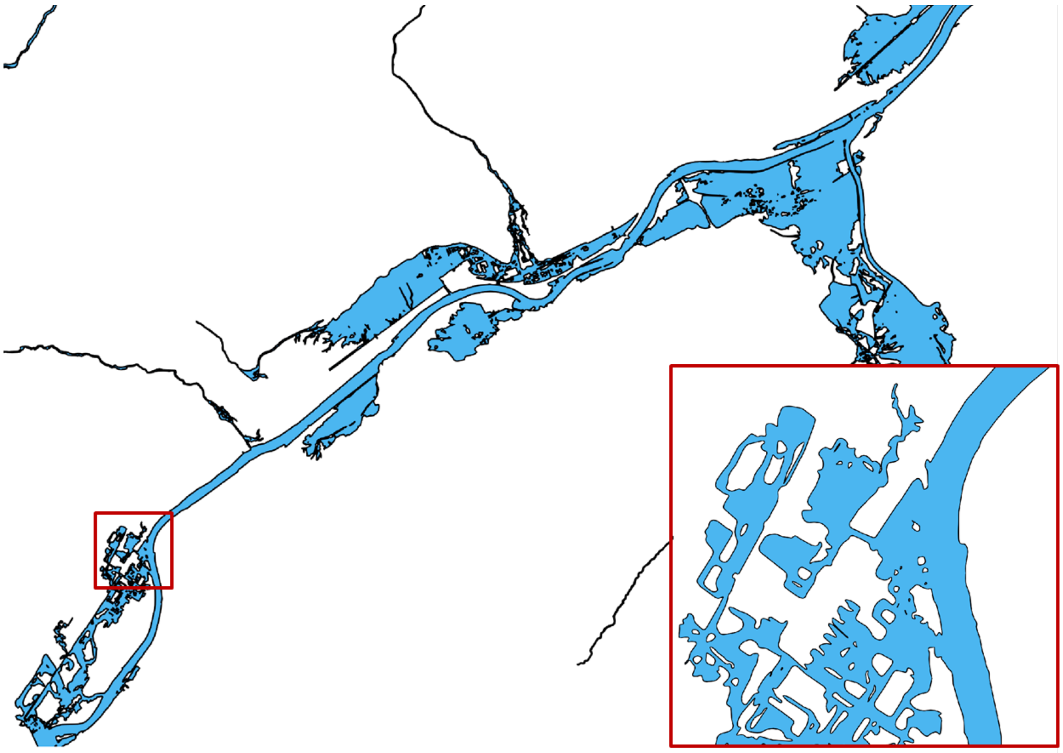
To analyze the strengths and weaknesses of the proposed vector data visualization method, it is applied in real-world use cases of a flood management system. Eight different case studies are utilized, which are introduced and described in more detail in the next section. Afterwards, the results of performance tests on time and memory consumption are presented and different influence factors are discussed.

5.1 Case Studies

The eight case studies are chosen to cover different aspects, such as the expansion area, the size, and the application of the vector data. Detailed information about the number of interior and exterior polygons and the number of lines can be found in Table 5.1. The

	Polygons			**Lines**	**Total**
	Exterior	**Interior**	**Total**		
HORA IL	19	1,605	1,624	1,624	**3,248**
Cologne OSM	7,749	856	8,605	43,985	**52,590**
Wachau	21,710	2,327	24,037	6,307	**30,344**
Graz SC	92,268	31,380	123,648	29,111	**152,759**
HORA RLL	22,330	659	22,989	2,747	**25,736**
Cologne SA	446,321	276	446,597	652,334	**1,098,931**
HORA FA	22,998	116,296	139,294	116,317	**255,611**
HORA RC	63,397	659	64,056	41,079	**105,135**

Table 5.1: Numbers of different decal types per case study. The smallest and largest numbers among all case studies are highlighted in blue and red, respectively.



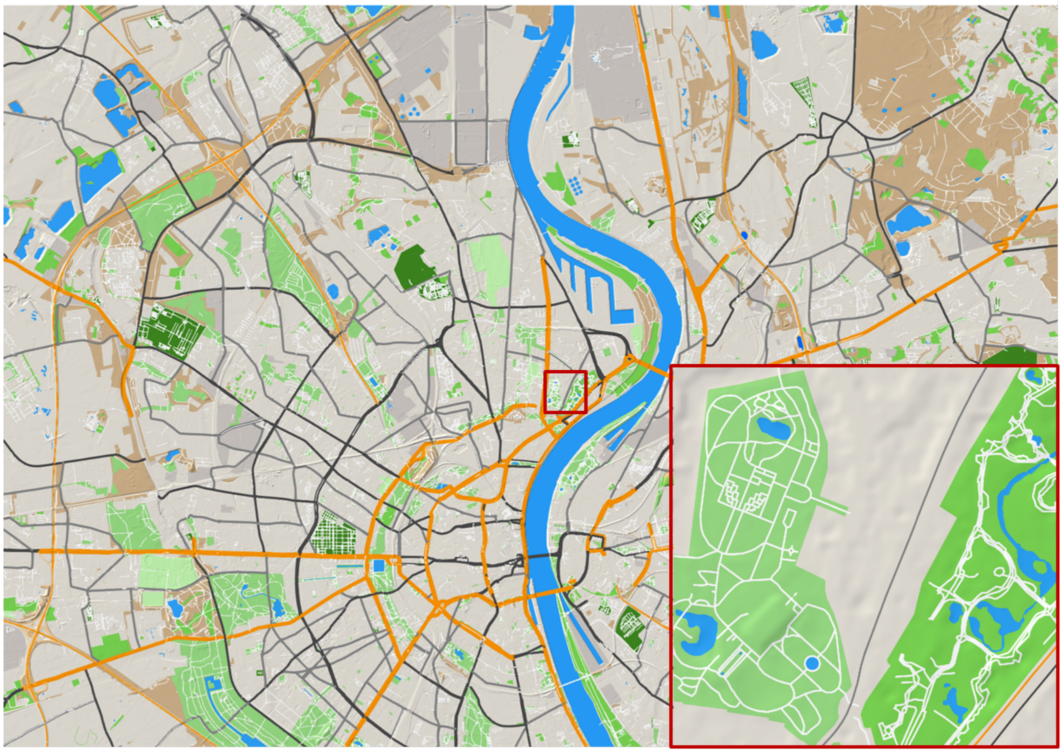


Figure 5.1: The case studies *HORA IL* (top) and *Cologne OSM* (bottom).

	Polygons			Lines	Total
	Exterior	Interior	Total		
HORA IL	53,833	69,178	123,011	123,011	**246,022**
Cologne OSM	185,720	18,686	204,406	227,219	**431,625**
Wachau	328,924	20,445	349,369	114,612	**463,981**
Graz SC	2,471,650	330,945	2,802,595	1,460,434	**4,263,029**
HORA RLL	904,271	61,666	965,937	4,802,375	**5,768,312**
Cologne SA	6,228,849	2,982	6,231,831	4,690,521	**10,922,352**
HORA FA	2,437,857	8,268,962	10,706,819	9,790,961	**20,497,780**
HORA RC	15,067,576	61,666	15,129,242	14,213,384	**29,342,626**

Table 5.2: Numbers of vertices per decal type and case study. The smallest and largest numbers among all case studies are highlighted in blue and red, respectively.

line numbers include the number of all lines and polygon outlines. One can see that the case studies differ in the ratio of different decal types and this variation enables the analysis of the visualization methods based on the distribution of the types. A major factor that influences performance is the data complexity, which is mainly determined by the number of vertices. In Table 5.2 their distribution over all case studies is listed. The case studies are sorted according to the number of vertices and this order is used for the whole chapter. The first three case studies contain smaller data sets with less than 500,000 vertices. They are followed by two case studies with medium sized data sets with less than 6 million vertices and three large case studies with more than 10 million vertices each.

The vector data of the case studies have different sources and represent different objects. The data are grouped accordingly to be able to individually manipulate the visualizations with different settings for the used data structures and the applied styles. Table 5.3 shows the distribution of data groups with their characteristics for each case study. A data group either contains polygons with outlines, polygons only, or lines only, which determines the number of draw calls. Since polygons and lines are rendered in separate draw calls, two render passes are executed for data groups that contain polygons with outlines and otherwise only one draw call is required. The table also contains information about the color settings for the individual data groups, because it is an influence factor for both, the volume-based and the screen-based approach. It shows the numbers of draw calls that apply individual colors to lines and polygons. The other draw calls use only one color for all lines and one color for all polygons.

5.1.1 HORA Isolines (IL)

In this case study vector data from *Natural Hazard Overview & Risk Assessment Austria (HORA) 3* [hor] are utilized to visualize maximum water depths via blue polygons with black outlines. *HORA* is a warning service of the Federal Ministry of Agriculture, Regions

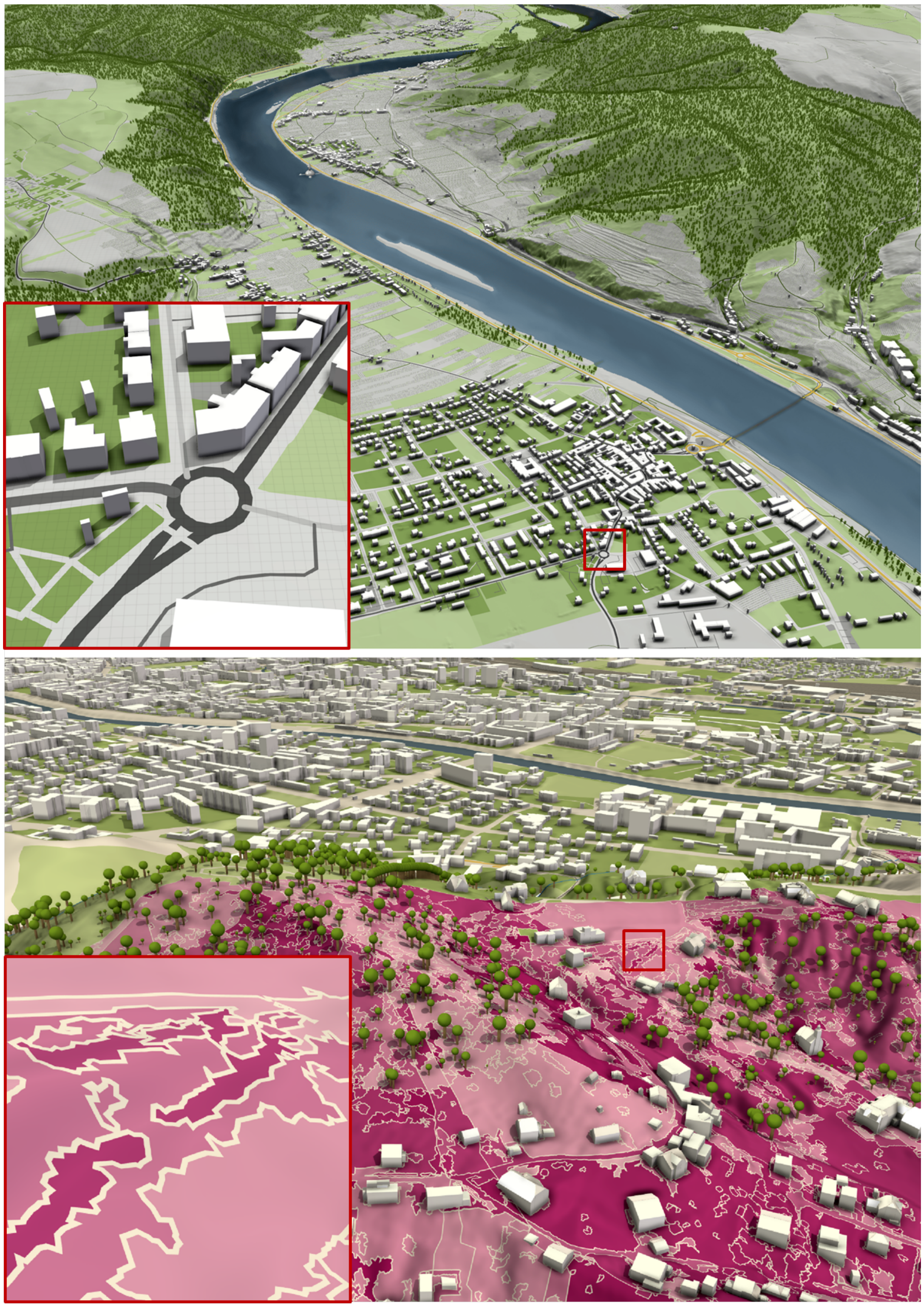


Figure 5.2: The case studies *Wachau* (top) and *Graz SC* (bottom).

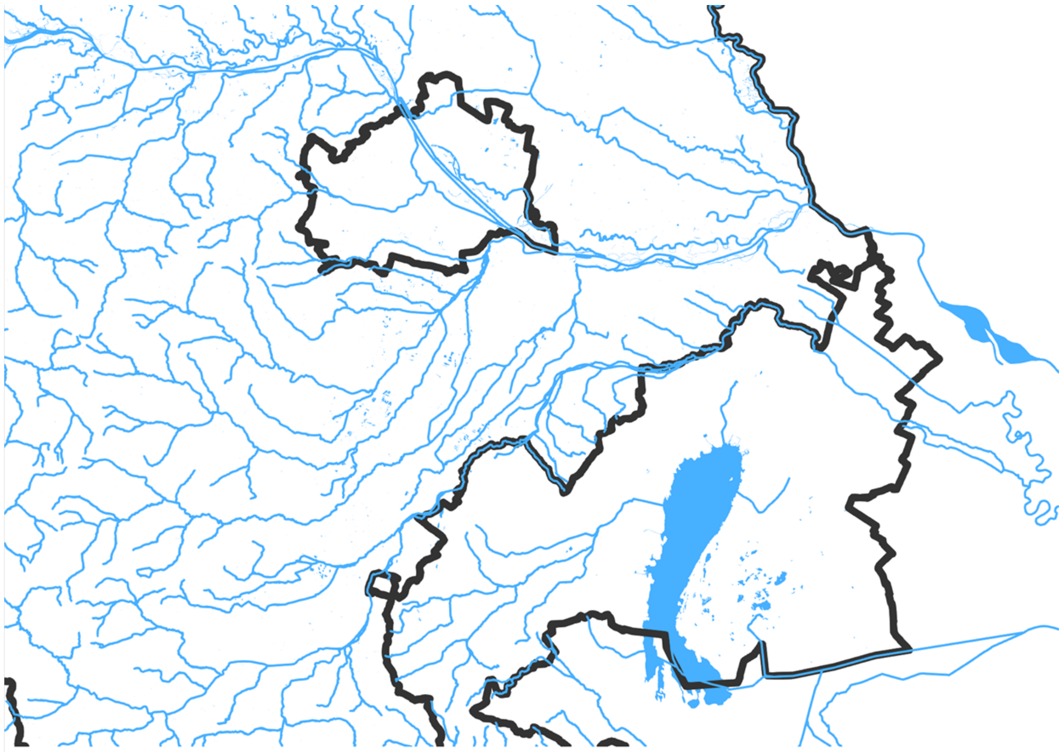
	Draw Calls	**Data Groups**				**Individual Colors**		
		P+O	**P**	**L**	**Total**	**P**	**L**	**Total**
HORA IL	2	1	-	-	**1**	-	-	**-**
Cologne OSM	3	-	2	1	**3**	2	1	**3**
Wachau	3	-	1	2	**3**	1	1	**2**
Graz SC	9	1	2	5	**8**	3	5	**8**
HORA RLL	3	-	1	2	**3**	-	-	**-**
Cologne SA	6	1	2	2	**5**	2	1	**3**
HORA FA	4	1	1	1	**3**	1	-	**1**
HORA RC	4	1	1	1	**3**	1	-	**1**

Table 5.3: The number of draw calls, the number of data groups, and the number of data groups that use different colors per case study and for polygons with outlines (P+O), polygons only (P), and lines only (L). The smallest and largest total numbers among all case studies are highlighted in blue and red, respectively.

and Tourism for natural disasters in Austria. One application for the visualization of the Austria-wide *HORA 3* data is the comparison with earlier *HORA* data to identify differences in simulation results. The case studies *HORA IL* and *HORA flood areas (FA)* are the only ones that have more interior than exterior polygons. Although the number of polygon types is very different, the number of vertices is not. This is caused by the higher complexity of the exterior polygons compared with the interior polygons, which can be seen in Figure 5.1 (top). Since the case study is the only one that contains a single data group with polygons with outlines, it is also the only case study with the same number of polygon and line vertices. Furthermore, *HORA IL* is the smallest case study with respect to the total number of decals and vertices.

5.1.2 Cologne OpenStreetMap (OSM)

The Rhine flows right through Cologne, which makes the city vulnerable to flooding. Two of the case studies deal with data of Cologne, one smaller and one larger case study. They are chosen because they represent typical use cases for flood management in urban environments. *Cologne OSM* is the smaller case study because it has less vector data and covers a smaller area. The used vector data are open data provided by *OpenStreetMap* [osm] that contain information about land use, water and road networks. An overview is shown in Figure 5.1 (bottom). One can see that individual colors are used to indicate different land use, water, and road types. Together with the *Cologne & Surrounding Area (SA)* case study, they are the only case studies that contain more lines than polygons, whereby the line number of the *Cologne OSM* case study is determined by lines only and the *Cologne SA* case study also contains polygon outlines.



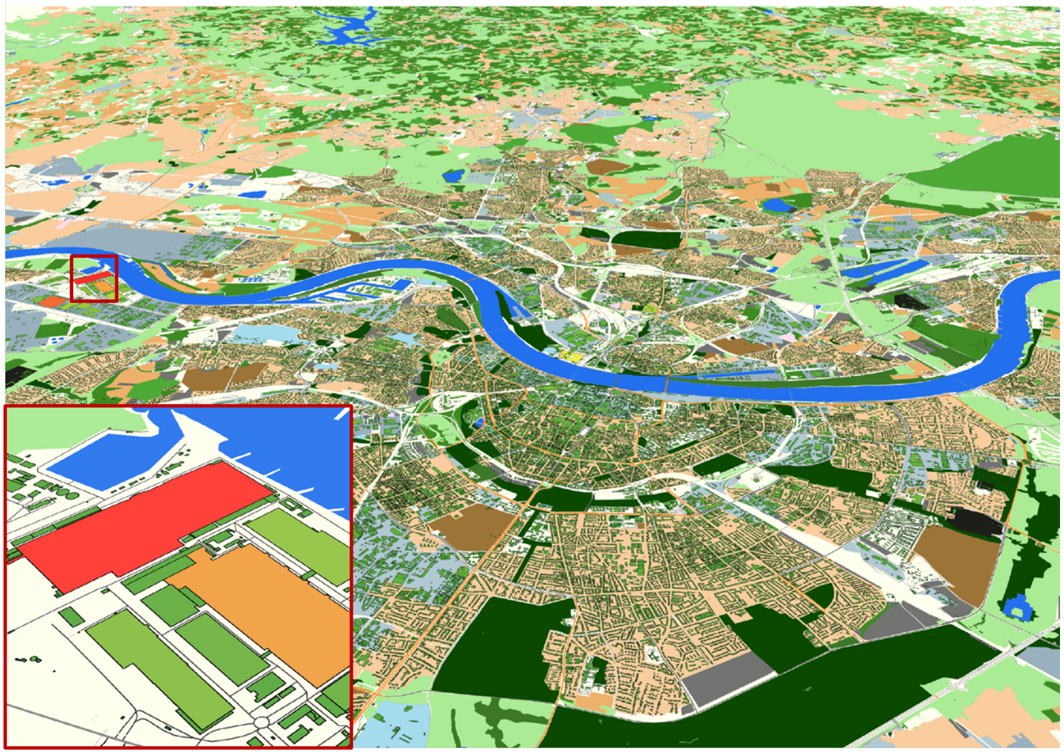


Figure 5.3: The case studies *HORA RLL* (top) and *Cologne SA* (bottom).

5.1.3 Wachau

The Wachau is an area in and around the valley of the Danube in Lower Austria, which makes the Danube a flood risk for this area. For the analysis of this risk, vector data of *OpenStreetMap* [osm] are utilized. This case study represents a typical vector data set of a rural environment in which flood management is performed. It contains only lines and polygons without outlines. As most case studies the *Wachau* case study has more polygons than lines. With 114,612 line vertices, it contains the fewest number of line vertices among all case studies. This is caused by lines that are only used to represent administrative borders and the road network, which is not so dense in this area. Polygons are used to visualize a larger amount of data to represent the land use. Figure 5.2 (top) shows the case study with lines in different colors to distinguish the road types and polygons in different shades of green to represent the land use types.

5.1.4 Graz Sewer Catchments (SC)

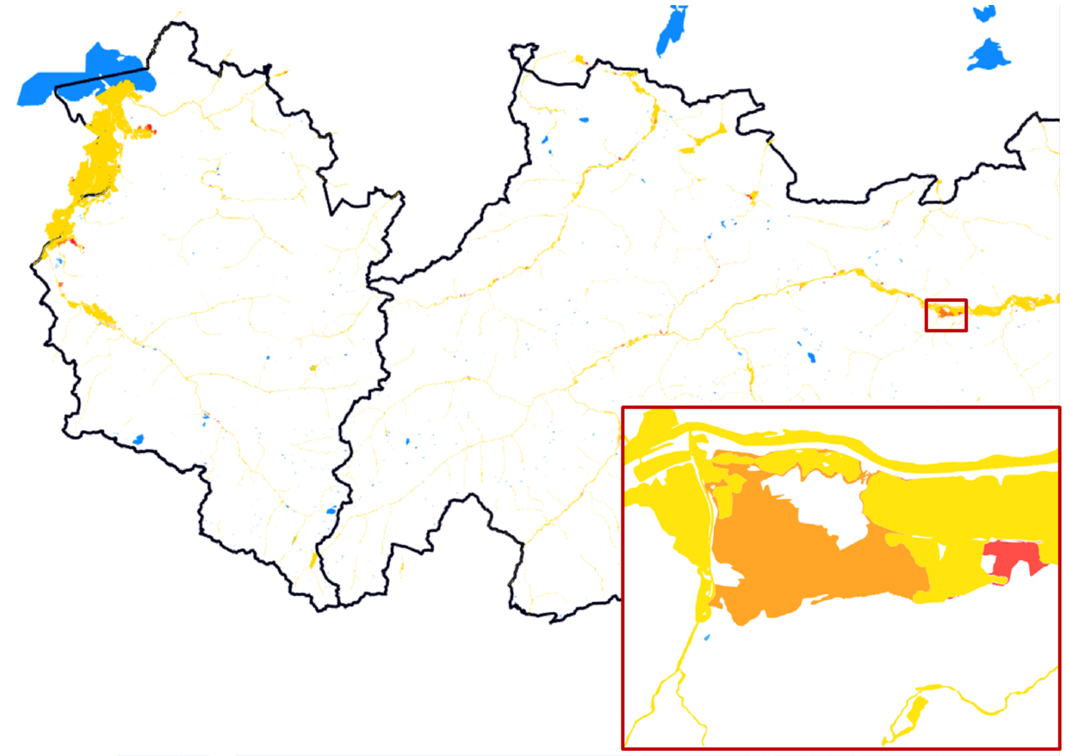
The Mur river flows through the city center of Graz and it can be seen in the upper part of Figure 5.2 (bottom). The *Graz SC* case study has eight data groups, which is the highest number of all case studies and leads to a dense occurrence of decals. The groups contain sewer catchments, land use, rivers, lakes, and the road and rail network of Graz. The sewer catchments, given as polygons with outlines, encircle the area in which all surface water runs off to the same sewer inlet, which can be precomputed from the terrain topology. In Figure 5.2 (bottom) these polygons are colored in different shades of magenta to be able to distinguish the individual sewer catchments. This can be helpful to asses the risk of a drain to overflow. Individual colors are also used for four further data groups in this case study, to highlight data relevant aspects, such as road types.

5.1.5 HORA River Lines & Lakes (RLL)

This case study shows rivers and lakes of the whole of Austria by using detailed data of *HORA* [hor]. Lakes are represented by polygons without outlines and rivers by lines. In Figure 5.3 (top) one section of the Austrian-wide vector data is shown, including a part of Lower Austria, Vienna, and the upper part of Burgenland with Lake Neusiedl. Due to the large scale of this case study, it is desirable to have lines that are still visible if they are viewed in overview and that it is still possible to perceive the detailed flow of a river in close-up views. Therefore, this case study is a good application of the dynamic line visualization approach because it fulfills these requirements. Although the data are very complex, there is also a lot of empty space. There are many rivers all over Austria, but their coverage is relatively small compared to the whole area of the country.

5.1.6 Cologne & Surrounding Area (SA)

This case study covers with more than a million decals an area of $100 \times 100\ km^2$ in and around Cologne. This makes it the largest case study regarding the number of decals with a high density. It contains five data groups with information about land use, building



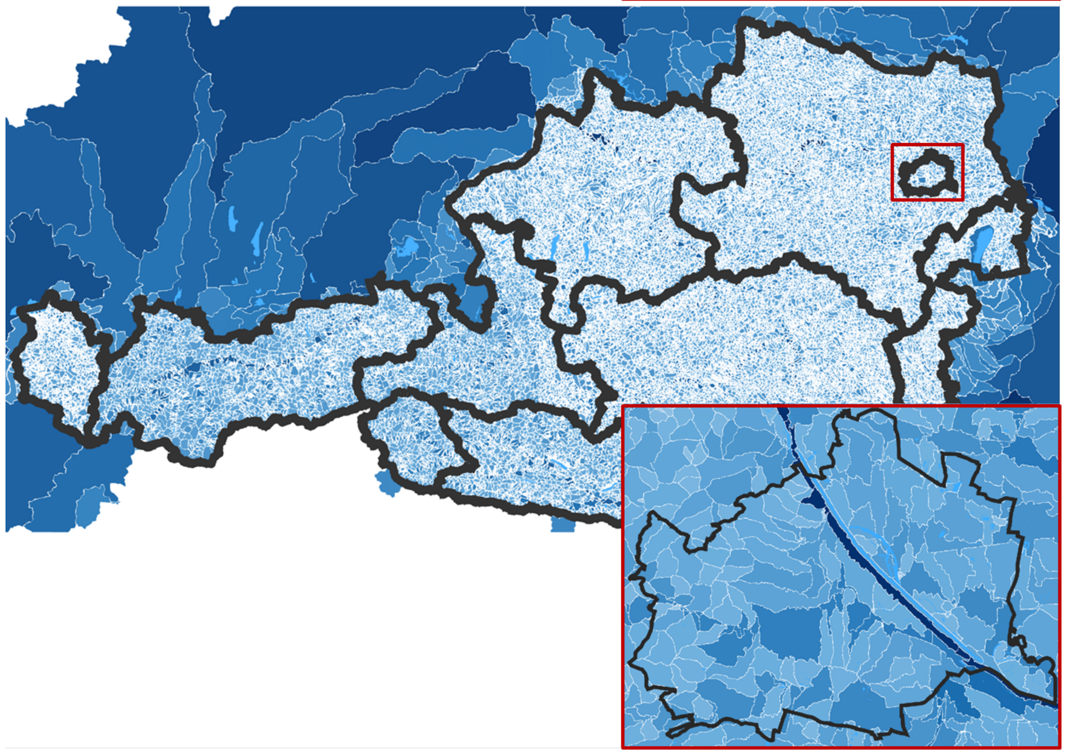


Figure 5.4: The case studies *HORA FA* (top) and *HORA RC* (bottom).

footprints, water, and roads. Together with the *HORA River Catchments (RC)* case study, they are the only ones that use a transfer function for continuous color mapping. The transfer functions produce many different colors, which are particularly expensive for the volume-based approach. In the *Cologne SA* case study a transfer function is used to map the ground area of buildings to the filling color of their corresponding polygons. The zoomed-in area of Figure 5.3 (bottom) shows a large building in red, a medium-sized building in orange and smaller buildings in different shades of green. Different colors are also used to differentiate land use and road types in Cologne and surrounding area.

5.1.7 HORA Flood Areas (FA)

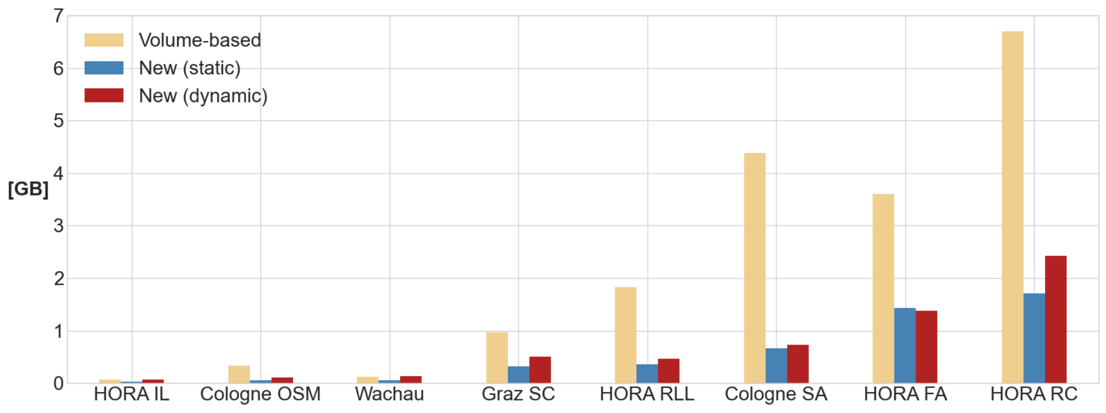
In Figure 5.4 (top) one can see the flood area data of Vorarlberg and partially Tyrol provided by *HORA* [hor] (not calibrated, preliminary results). The data are categorized into three intensities, which are visualized in different colors: yellow, orange, and red. They show the probability of a flood occurrence every 30, 100 or 200 years. The flood areas are very complex and jagged polygons without outlines. There is a lot of empty space between the flood areas, but despite their low coverage, the data have a high complexity with respect to the number of vertices. *HORA FA* is the case study with the highest number of interior polygons and the most interior polygon vertices. If the case study is compared with the case study *Cologne SA*, they differ in data complexity because *HORA FA* contains almost twice as many vertices and has less than a quarter of the decals in *Cologne SA*.

5.1.8 HORA River Catchments (RC)

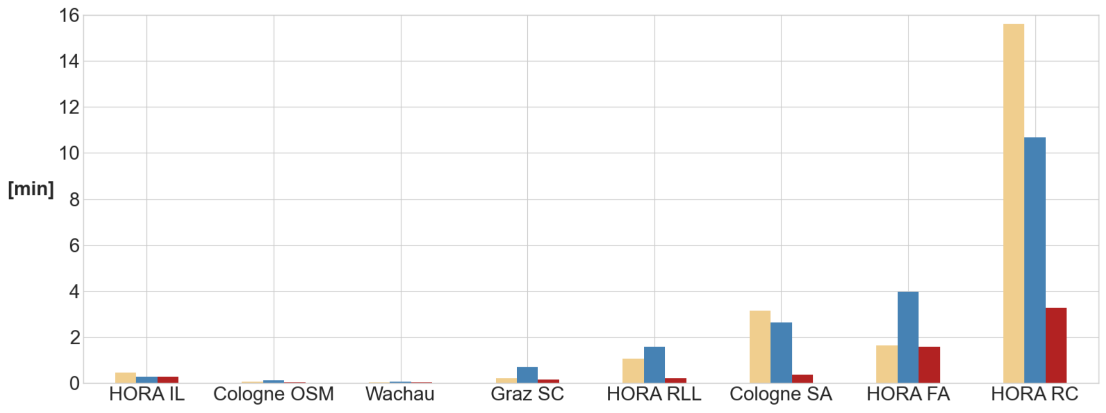
River catchments represent areas around a river, where all the landed water on this area reaches a corresponding river point. This means the border between river catchment areas assigned to different rivers are watersheds. In Figure 5.4 (bottom) the river catchments of the whole of Austria are visualized as polygons in shades of blue with white outlines to highlight the borders. The polygons are color coded by using a transfer function to map the polygon filling color according to the size of the catchment area. This case study has almost 30 million vertices, making it the case study with the most vertices in total. The data also provided by *HORA* [hor] cover the whole area of Austria with more than 80,000 km^2. These dense river catchment areas also overlap with other decals contained in this case study, which are lakes and borders.

5.2 Results & Discussion

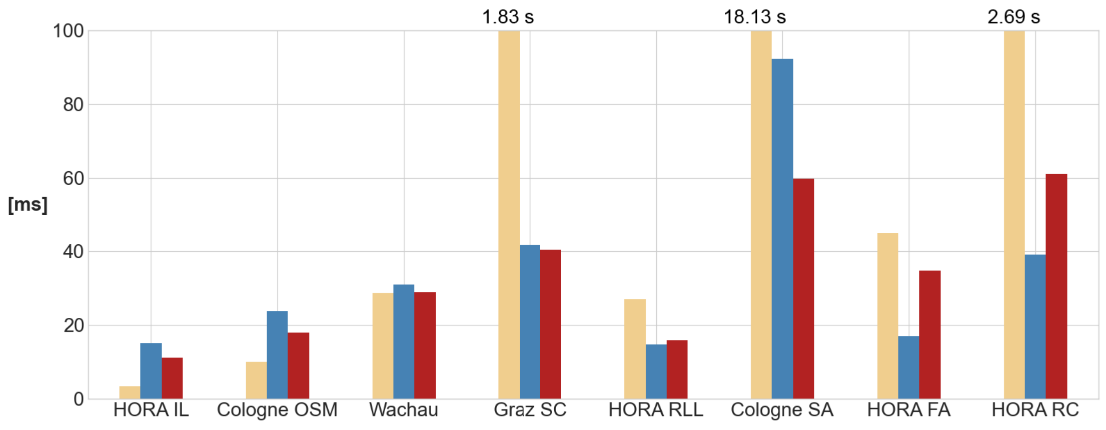
In this section the new screen-based approach is compared with a volume-based approach. The differences in time and memory consumption of polygons, static, and dynamic lines are further analyzed under different influence factors. The results are divided into sections according to the decal types that are affected by the test factors.



(a)



(b)



(c)

Figure 5.5: Comparison of a volume-based approach in beige with the new approach with static lines in blue and dynamic lines in red, under the aspects (a) GPU memory in gigabytes (GB), (b) update time in minutes (min), and (c) render time in milliseconds (ms).

5.2.1 Test Settings

The tests are performed on an Intel Core i5-8600K with 3.6 GHz and 32 GB RAM, and an Nvidia GTX 1070 Ti graphics card with 8 GB memory. To ensure reproducibility and comparability, the same starting view within a case study is used for all test cases. Afterwards, the same number of samples is taken by zooming in, to consider the zoom-dependency of the new screen-based approaches. For the anti-aliasing of static decals a medium MSAA with eight samples per pixel is applied. The value of the following influencing factors is also set to be constant for all tests and is only changed for testing their own influence. A full HD image resolution with 1920×1080 pixels is used for all test cases. A resolution of 500×500 cells is set for the decal grid of static decals. The depth of the polygon quad-trees and their leaf capacity is constantly limited to eight.

5.2.2 All Decal Types

The decal types include volume-based decals, static, and dynamic screen-based decals. In the following sections the results of performance tests for different decal types are presented and their differences and influence factors are discussed.

Comparison with Volume-Based Technique

The vector data of all eight case studies are visualized with different approaches. Figure 5.5 contains the different performance results of the three approaches, i.e., the volume-based approach and the new screen-based approach with static and dynamic lines. It reveals that the GPU memory and the preprocessing time of the new approaches grow with the amount of vector data, while the volume-based approach is more irregular. This can be explained with the different influence factors that affect the results of the individual approaches.

The main factor for all approaches is the amount of vector data, but besides this factor, the volume-based approach is mainly influenced by the number of different colors. It is necessary to group the decals according to their colors to be able to render them separately. This is memory- and time-consuming and case studies with many color variations stand out with their high memory usage and long runtimes. Since the case studies *Cologne SA* and *HORA RC* use transfer functions to map continuous colors onto the decals, they have the most color variations. This is the reason why they need the most memory, update and render time, if the volume-based approach is used.

The static screen-based approach relies also on the resolution of the decal grid and the polygon quad-trees. The dynamic approach depends also on the terrain grid resolution and the image resolution. These additional influence factors for the new approaches do not show a significant effect on the results as the different colors for the volume-based approach do. This is because they are consistent over all case studies, only the terrain grid resolution varies. The minimum cell size of the terrain grid lies under twenty meters for all case studies, except for *Cologne SA*, where a flat terrain with a minimum cell size of one kilometer is used. This size is taken to determine the sample rate of the dynamic lines on the terrain grid. Since the sampling is the main preprocessing step for the dynamic lines, it explains why the preprocessing for the *Cologne SA* case study is very fast, although a lot of data are present.

According to the results presented in Figure 5.5a the volume-based approach has a higher GPU memory consumption than the new screen-based approaches in all case studies. This is expected, because for the volume-based approach a 3D mesh has to be stored for each decal. Such a mesh consists of two polygon surfaces to close off top and bottom and additional triangles to close the sides. Storing these meshes requires more memory than is necessary for the screen-based approach. For the static variant only 2D vertices and light-weight index-based data structures have to be stored. The dynamic variant requires 3D vertices and also an index-based data structure.

Figure 5.5b shows that the approach with dynamic lines needs less time for preprocessing than other approaches. It is faster because the main preprocessing step is the sampling of lines on the terrain and this is done mostly in parallel on the GPU. Furthermore, for the dynamic lines it is not necessary to generate 3D meshes or static data structures, which also saves time. For 5 out of 8 case studies, the static approach has an higher preprocessing time than the volume-based approach. The main reason for this is the high number of decal grid cells, although they are processed in parallel on the CPU, they still lead to long waiting times. Another reason for the high update time is the generation of the tree-structures, which is not done in parallel. This effect is particularly noticeable if much vector data has to be stored per decal grid cell, which leads to large BVHs and polygon quad-trees.

In Figure 5.5c one can see that for the small case studies the volume-based approach is faster but for larger vector data sets the new approach outperforms it. The great advantage of the volume-based approach is that the rendering of 3D meshes is highly supported by modern graphics hardware. So this approach can use hardware-accelerated rasterization and per-fragment tests for a fast rendering of decals. Even though these performance advantages can be exploited, the approach is limited, especially if different colors are applied. The volume-based approach needs several seconds to render the data of three case studies, where the time bars are cut-off in Figure 5.5c. The loss of interactivity is caused by the use of many different colors. Since the volume-based approach groups the decals per color to render them in separate passes, this leads to a huge number of draw calls and a high render time. The screen-based approach cannot take advantage of hardware support such as the volume-based approach, because it is based on 2D vector data that are processed per pixel and are not sent through the hardware-supported render pipeline. This leads to a constant overhead, which is noticeable particularly in the smaller case studies of Figure 5.5c.

With the new screen-based approaches, the vector data of all case studies are visualized at interactive frame rates. An important factor for the runtime of both new approaches is the density of the vector data. There is a high number of decals with a very high density in the *Cologne SA* case study, which leads to many decals stored in the same grid cell or stored in one linked list, depending on the used line visualization method. For static lines this leads to large BVHs that correspond to these dense cells and their traversal during rendering is more time consuming than the traversal of smaller BVHs of sparse vector data. *HORA RLL* and *HORA FA* are both case studies with a low density of decals. This is one reason for the faster render time compared to case studies containing less vector data. An additional reason for the *HORA RLL* case study to be rendered relatively fast is that it contains more line vertices than polygon vertices and lines are rendered faster.

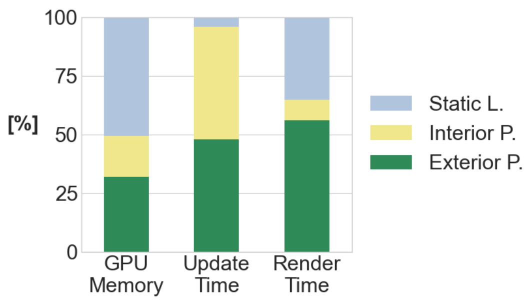
Percentage Distribution

The *HORA IL* case study is utilized to evaluate the percentage distributions of GPU memory, update, and render time for different decal types of the new approaches. This case study has the advantage that it only contains polygons with outlines, which leads to the same number of polygons and lines. Furthermore, there is a similar number of vertices of interior and exterior polygons, even if the number of interior polygons is much higher than the number of exterior polygons.

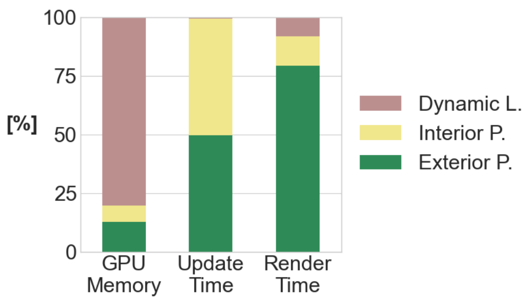
In Figure 5.6a the percentage distributions of exterior and interior polygons and static outlines are shown. One can see that static lines and polygons require similar amounts of GPU memory, although their data and data structures differ. All outlines for exterior and interior polygons and exterior polygons themselves are organized in BVHs. Interior polygons are not stored in BVHs, which is the reason why they have a lower memory consumption than exterior polygons. This leads to more line data stored in BVHs than polygon data. Lines additionally require memory to store line width and corner style information. This higher memory consumption of lines is compensated by polygons through the storage of additional quad-tree data.

The update time is highly dependent on the vector data, i.e., their complexity and density determine how long it takes to generate BVHs for lines and polygons and polygon quad-trees. In the *HORA IL* case study the main difference in preprocessing time between lines and polygons is caused by the polygon quad-tree construction. Especially the leaf node polygon generation is time-consuming. This is the reason why polygons make up almost all the update time. In cases with a higher density of line data, the update time of lines would account for a larger percentage. The preprocessing time could be considerably reduced by a parallel implementation of the tree structure generation on the GPU to reduce the workload of the CPU.

Polygons have a higher render time than lines because point-in-line tests can be executed more efficiently than point-in-polygon tests. Only one line segment has to be checked per point-in-line test but for point-in-polygon tests all edges of the leaf polygon are used. Another reason why lines are rendered faster is that the line BVHs directly store line segments at their leafs, but polygon BVHs store quad-trees. This means for polygon rendering two tree structures, a polygon BVH and a quad-tree, have to be traversed to find a candidate for the point-in-polygon test. Although the implemented quad-tree culling shortens the traversing of the tree, it still leads to an additional render overhead.



(a)



(b)

Figure 5.6: Percentage distributions of GPU memory, update time, and render time for exterior polygons, interior polygons and for (a) static and (b) dynamic lines.

The rendering of interior polygons is faster because they are not organized in BVHs, but assigned to exterior polygons. Therefore, the BVH is only traversed for exterior polygons and interior polygons are only checked after a positive exterior polygon test.

Figure 5.6b presents the percentage distributions of exterior and interior polygons and dynamic lines. It shows that dynamic lines have a higher GPU memory consumption than polygons and therefore also a higher memory consumption than static lines. Even though memory can be saved because dynamic lines do not require static data structures, the pixel-based approach is memory-intensive. On the one hand this is caused by storing data per pixel, which are usually a lot to get a certain image quality. On the other hand it is caused by increasing the line data resolution by sampling the initial lines on the terrain grid. This results in much more vertices and by adding the terrain height information, 3D vertices have to be stored instead of 2D vertices, as for static lines. The size of the per-pixel data could be reduced by the line color values, because it does not have to be stored per pixel. Since the color changes only per line, the storage per line would be sufficient, but at the cost of runtime. Many memory accesses would be necessary during line sorting and also at the final render pass, which are reduced by the current approach.

The percentage contribution of dynamic lines to the update time is very small because no static data structures have to be generated. Furthermore, the sampling of lines on the terrain is done mostly in parallel on the GPU, which is fast and also contributes to the short update time.

Figure 5.6b suggests that dynamic lines are rendered much faster than static decals, which can occur but is generally not the case. The dynamic lines are faster in smaller case studies and slower in case studies with many line vertices. For static lines the number of lines that contribute to a pixel are limited to the lines inside one decal grid cell. The dynamic lines are only limited to the line segments that are inside the visible area defined by the view frustum for which all vertex indices have to be updated dynamically. This task highly depends on the number of vertices and also leads to a higher render time particularly for scenes where all lines are visible. The render times of dynamic lines are

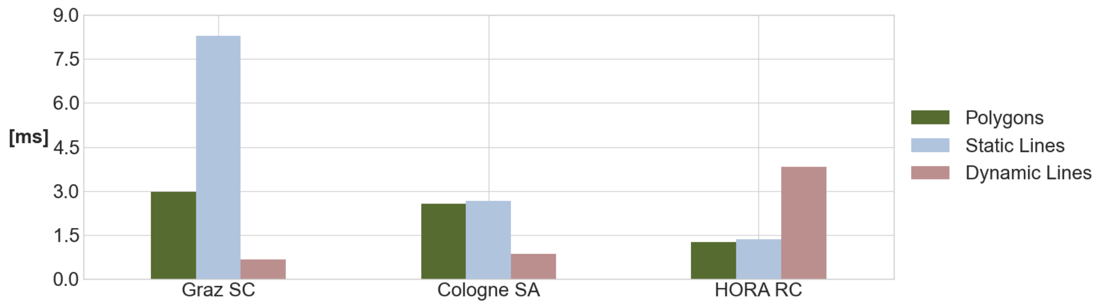


Figure 5.7: The average render overhead of each decal type, i.e., polygons, static lines, and dynamic lines, if all decals are outside the visible area.

also more variable because they are not limited to a fixed grid cell size. The lines grow in world space when zooming out, resulting in more overlaps between the lines and slower render time. Dynamic lines are, additionally to zooming, sensitive to several influence factors, such as line density, width, color, and transparency.

Render Overhead

The new screen-based approach produces more render overhead than a volume-based approach. The three case studies *Graz SC*, *Cologne SA*, and *HORA RC* are utilized to analyze this overhead, because they contain an increasing amount of vector data while the number of data groups and draw calls decreases from case study *Graz SC* to *HORA RC*. Thus, it can be evaluated if the amount of data or the number of draw calls has more impact on the overhead.

A reason for the larger render overhead of screen-based approaches is that they cannot make use of hardware-supported early discards of decals that are not visible. The decals are processed by using dynamic branching, which causes an additional render overhead. To prevent dynamic lines from being processed if they are outside the visible area, an additional view frustum culling step is executed before they are rendered. If all lines are outside the view frustum, only this step is carried out. Figure 5.7 depicts the render times for all polygons and lines being outside the visible area and therefore it shows the time that is necessary for the culling step of dynamic lines. This step is done in parallel on the GPU and in Figure 5.7 one can see, that the execution of the view frustum culling depends more on the number of vertices than on the number of data groups, because the time consumption grows with the number of vertices.

There is no such additional step for static decals to check if they lie inside the visible area before they are rendered. During the render pass the pixel has to be projected back into world space by using the terrain depth. This pixel position is then used to determine the corresponding decal grid cell. The pixel can only be discarded if the world-space pixel position lies outside the decal grid. The render overhead produced by looking up the terrain depth and decal grid grows with the number of draw calls. This can also be

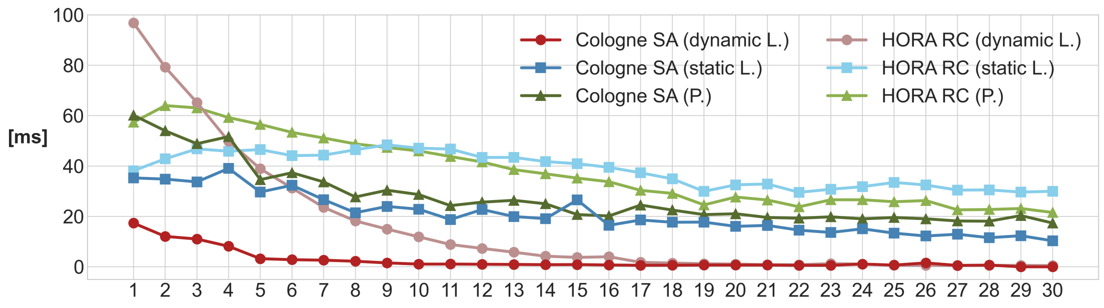


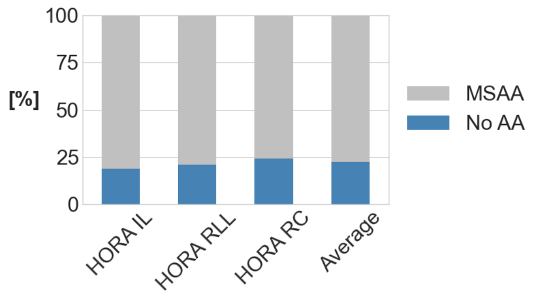
Figure 5.8: Render times of thirty consecutive frames while zooming in for polygons, static and dynamic lines in two case studies.

observed in Figure 5.7, which reflects that the case studies *Graz SC*, *Cologne SA*, and *HORA RC* execute nine, six, and four draw calls respectively, where the first case study makes six draw calls for lines and three for polygons and the other two case studies have the same number of draw calls for lines and polygons.

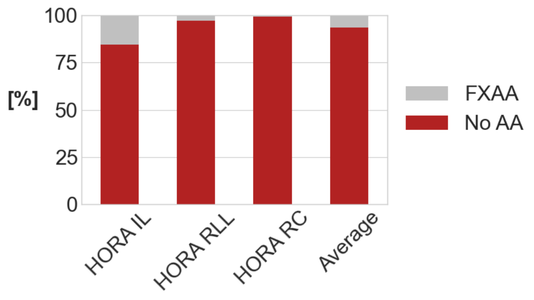
Zoom Dependency

The render times of the static and dynamic approach are zoom-dependent. In Figure 5.8 the render times of all three decal types for 30 consecutive frames while zooming in are shown. The two case studies are chosen due to their large extent and vector data density, which are properties that are expected to amplify the effect of zooming on render performance. The visible decrease in render time for polygons and static lines is mainly caused by decals covering more pixels while zooming in. This avoids a full traversal of BVHs for empty pixels, because the search for decals only stops, if a pixel is completely opaque. The results show that the traversal of the BVHs is a decisive factor for the runtime of static decals. The use of a different data structure or the optimization of its traversal and its look-ups are possibilities to further improve the render performance. Another reason for a shorter render time if zooming in, can be complex decals that move outside the visible area and do not need to be processed anymore. Since polygons are usually more complex and consist of more vertices, their render time is more affected by this effect than static lines are. This is also visible in Figure 5.8 where the polygon runtime is decreasing more with an increasing zoom factor.

The results reveal that the zoom factor has an higher impact on the runtime behavior of dynamic lines than on static decals. The reason for the render time decrease for dynamic lines is different, because zooming in leads to thinner lines with less overlaps between adjacent lines. Therefore, less data has to be stored per pixel in linked lists. This reduces the time that is necessary to dynamically generate the linked lists, to sort and to go through them for the final line rendering. This effect is clearly visible for the two case studies of Figure 5.8, which contain dense line data that produce many overlaps. For use cases with sparse line data and with a small line width, this effect will probably not be



(a)



(b)

Figure 5.9: The percentages of the total render time that are used for anti-aliasing (gray) with (a) MSAA and (b) FXAA. The rest of the render process is shown in blue for static decals and red for dynamic lines.

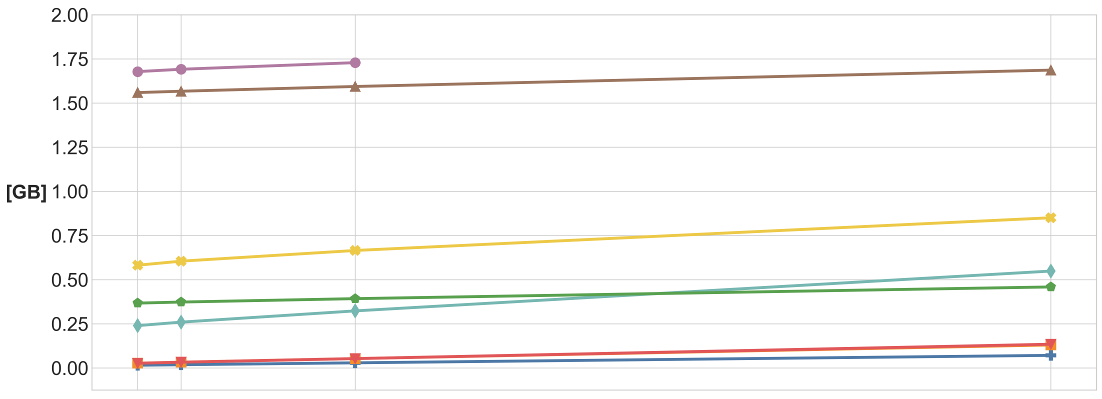
noticeable.

Anti-Aliasing

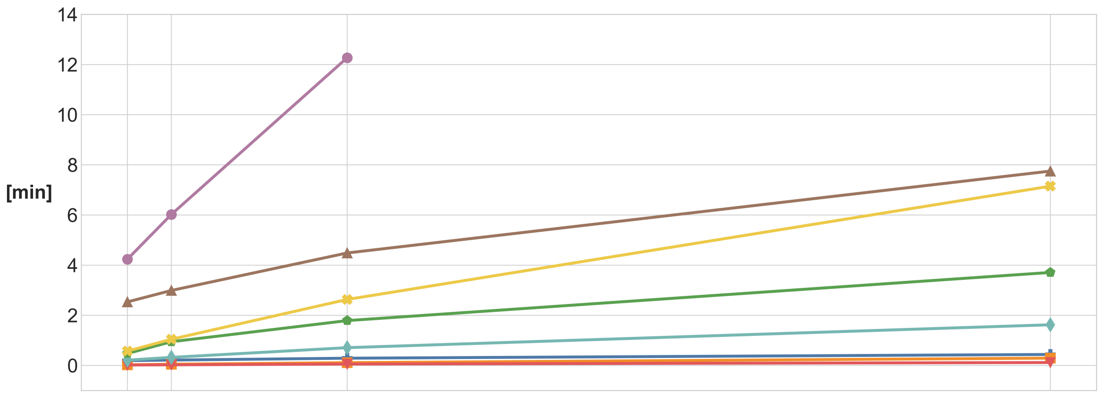
Different anti-aliasing techniques are utilized for static decals and dynamic lines. This section shows the influence of the anti-aliasing methods on the runtime of case studies *HORA IL*, *HORA RLL*, and *HORA RC*. The three case studies are chosen to be able to analyze the percentage of render time that is used for anti-aliasing of a small, medium, and large case study. Furthermore, with the *HORA RLL* and the *HORA RC* case studies the influence of a low and a high decal coverage can be tested.

For anti-aliasing of static decals *OpenGL's* multisampling (MSAA) is used, which makes up a large part of the render time for all case studies of Figure 5.9a. On average 77 % of the total render time is consumed by anti-aliasing for polygons and static lines. The reason for this high time consumption is that for the screen-based approach the optimization of MSAA is not applicable and it turns into SSAA. MSAA has to detect pixels at edges where anti-aliasing is necessary to be able to reduce the number of samples for other pixels. This is not possible, because the vector data are only processed in the final render step and are not available for optimizations before. To still be able to apply anti-aliasing to the static decals, multiple samples are taken and processed for all pixels by enforcing per-sample shading. The percentage of render time used for MSAA varies only slightly between the case studies, which means it grows with the render time and the decal coverage and varies between 12 and 30 ms for the tested case studies. To speed up the render performance of static decals an analytical anti-aliasing method or a post-processing anti-aliasing filter could be applied instead of the MSAA. Whereby other drawbacks need to be considered, such as more complex per-pixel calculations and a worse anti-aliasing result.

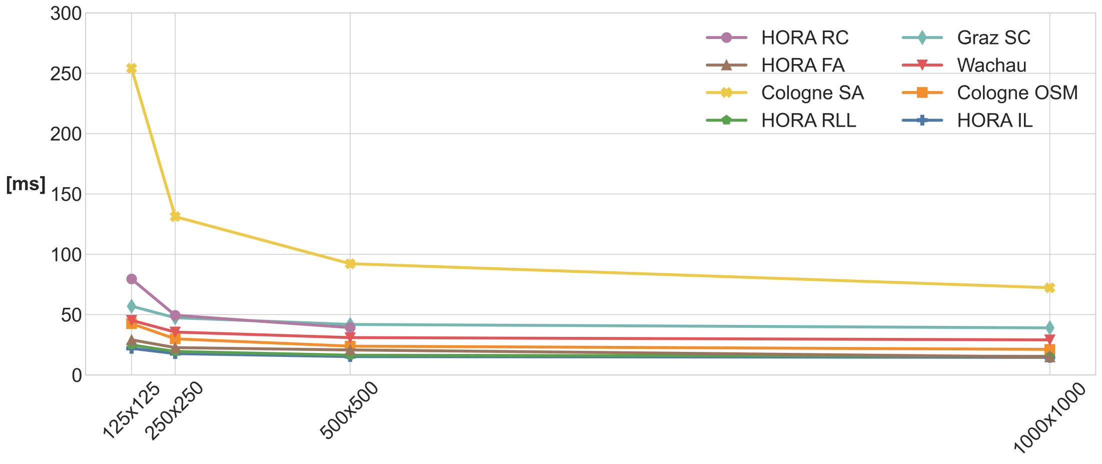
For anti-aliasing of dynamic lines the post-processing filter FXAA [fxa] is applied. It does not provide such a high quality anti-aliasing than MSAA but it is much cheaper.



(a)



(b)



(c)

Figure 5.10: The influence of different resolutions of the decal grid on (a) GPU memory, (b) update time, and (c) render time of static decals.

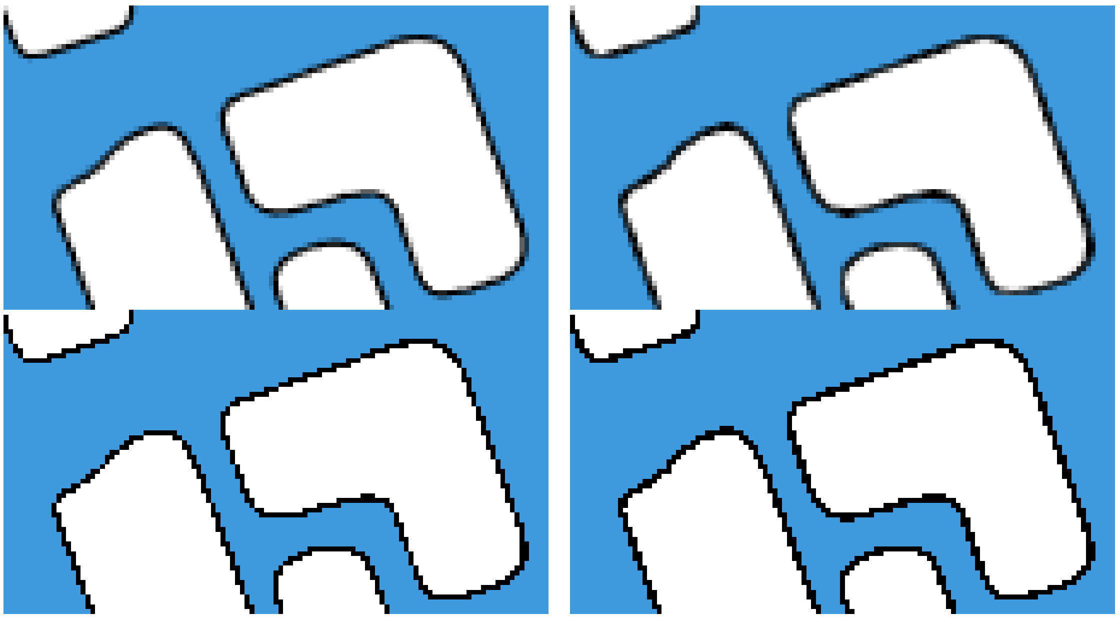


Figure 5.11: An upscaled area of interior polygons with black outlines. (a) Static outlines with MSAA (top) and without MSAA (bottom). (b) Dynamic outlines with FXAA (top) and without FXAA (bottom).

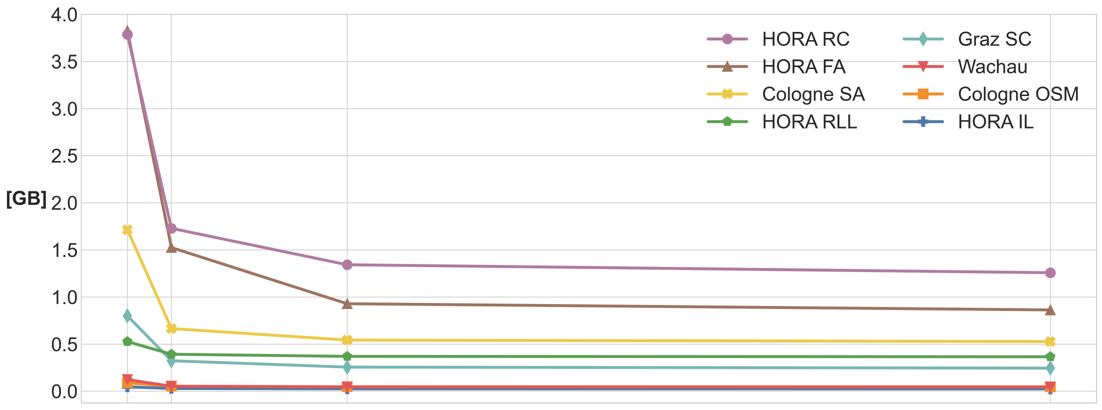
Figure 5.9b shows that FXAA makes up only 6 % of the total line render time on average. For smaller vector data sets such as used in *HORA IL* the percentage increases, because the rendering itself is faster. The execution time of FXAA is relatively constant and takes only 0.3 ms on average. Figure 5.11 shows upscaled interior polygons with black outlines of the *HORA IL* case study. In Figure 5.11a one can see a comparison of static outlines with and without MSAA and Figure 5.11b shows the difference of applying and not applying FXAA to the dynamic outlines.

5.2.3 Static Decals

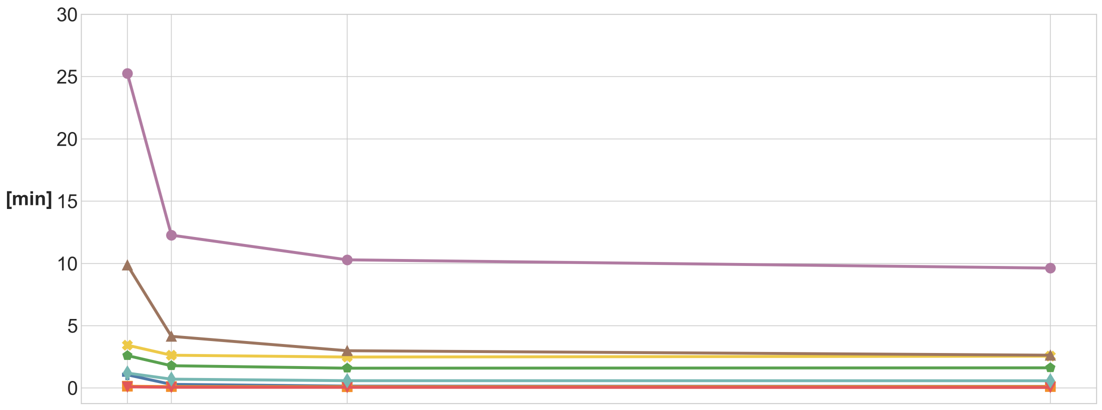
The results presented in this section only concern polygons and static lines. They show the influence of changing parameters for the static data structures. These include different resolutions for the decal grid and different settings for polygon quad-trees, such as the leaf capacity and the maximum tree depth.

Decal Grid Resolution

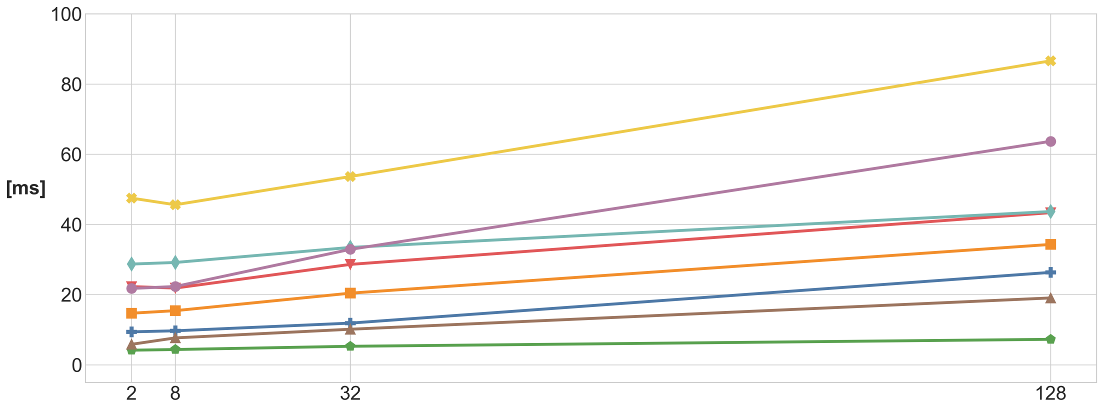
The grid resolution has a great impact on the visualization process of static decals, because it determines the amount of vector data that is stored per decal grid cell. Using a higher grid resolution means the vector data are distributed over more cells and the corresponding BVHs for lines and polygons.



(a)



(b)



(c)

Figure 5.12: The increase of the leaf capacity of polygon quad-trees leads to different results for (a) GPU memory consumption and the time necessary for (b) updating and (c) rendering polygons.

With a higher grid resolution the total number of BVHs grows. This results in a higher memory consumption because more bounding boxes and references to polygons and line segments have to be stored. The GPU memory consumption slightly increases with a higher grid resolution, as shown in Figure 5.10a. Since the data are mostly index-based, the impact of the grid resolution on the memory consumption is relatively small.

The grid resolution has a greater impact on the update time, which can be observed in Figure 5.10b. This is a result of the data processing that takes place in parallel per cell on the CPU. With a growing number of cells, more threads need to be queued, which leads to longer waiting times. If a cell is empty, it can be skipped immediately, which releases resources for other cells to be processed. For case studies with high decal coverage and therefore few empty cells, the per-cell processing becomes a performance bottleneck. The preprocessing of all case studies takes less than eight minutes, even for a high grid resolution, except the case study *HORA RC* requires a higher preprocessing time. Since the preprocessing of this case study with a grid resolution of 1000×1000 takes more than half an hour, the preprocessing was cancelled, which is the reason why Figure 5.10 does not include the results for this setting.

The grid resolution relates to the amount of vector data that is referenced per cell and it also influences the size of the polygon and line BVHs. A higher grid resolution results in smaller BVHs and the render time decreases due to a faster traversal. In Figure 5.10c one can see that the increase in grid resolution leads to a faster rendering for all case studies, but especially for *Cologne SA* and *HORA RC*. The vector data of these case studies are very dense and thus they benefit the most from dividing it further into several cells. For smaller case studies and case studies, where the data are spread out, a higher grid resolution has a smaller impact on runtime. A medium decal grid resolution of 500×500 is recommended because it offers a good trade-off between update time and runtime.

Quad-Tree Leaf Capacity

The leaf capacity determines the maximum number of polygon edges that can be stored per leaf node in the polygon quad-tree. The higher the leaf capacity, the more edges are allowed and the less often the quads have to be split. Therefore, a higher leaf capacity leads to smaller quad-trees, but also to more data stored per quad-tree leaf. Especially complex polygons consisting of many edges are strongly affected by this parameter.

The smaller the polygon quad-trees are, the fewer nodes they contain and the less GPU memory is required to store their data, such as node bounding boxes. There is a reduction of memory consumption for an increasing leaf capacity, as shown in Figure 5.12a. While the number of leaf nodes and therefore the number of leaf polygons is reduced, the number of edges per leaf polygon grows with the leaf capacity. But, the total number of edges of all leaf polygons is getting smaller. This is caused by less edges that need to be added to close the leaf polygons on the node boundaries.

Figure 5.12b shows that for a higher leaf capacity the time that is necessary to preprocess polygon data is getting shorter. Especially for the *HORA RC* case study, the capacity

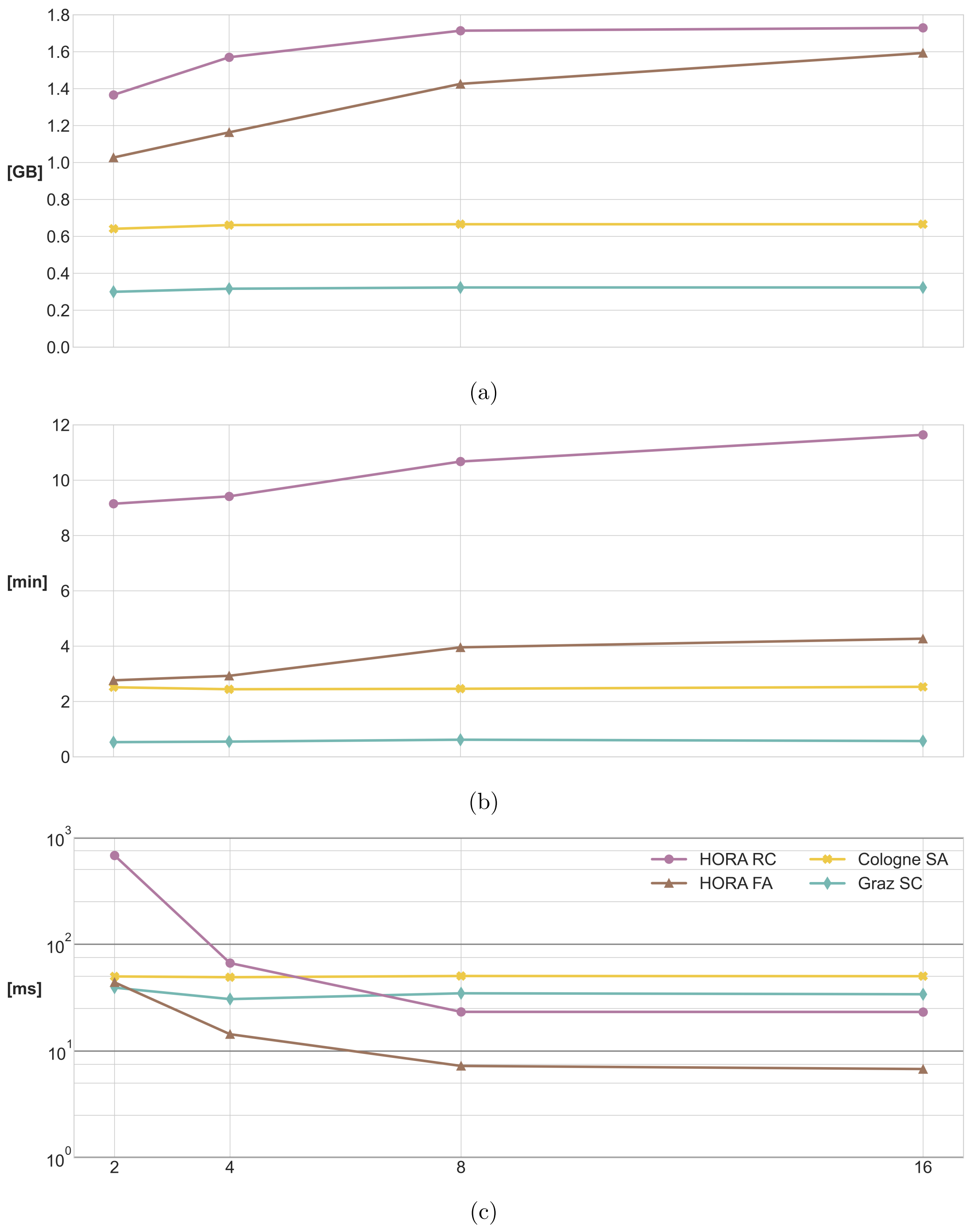


Figure 5.13: The impact of the maximum depth for polygon quad-trees on (a) GPU memory, (b) update time, and (c) render time of polygons.

has a great influence on the update time. This is caused by the high complexity of its vector data and the generally higher preprocessing time, which is further increased by the capacity. As a result of a non-parallel quad-tree generation, the time necessary for construction is much shorter for a high leaf capacity and small quad-trees. Furthermore, fewer leaf polygons have to be constructed with a higher capacity, which also saves time.

The drawback of a high leaf capacity is that having more edges in one quad-tree leaf leads to a higher runtime. As a result of more complex leaf polygons, the execution time for the point-in-polygon tests during rendering increases because more edges have to be tested. The extent of the impact on render time is presented in Figure 5.12c. It also shows that a leaf capacity of two and eight hardly makes a difference for most case studies, because the leaf polygons may have a similar number of edges again after adding the closing edges. To avoid long render times and to maintain interactivity the leaf capacity should not be too high. Therefore, a leaf capacity between 8 and 32 is suggested to avoid unnecessary memory and update time consumption with a lower capacity.

Quad-Tree Depth

Using the leaf capacity would be enough information to construct all polygon quad-trees according to the number of polygon edges. The result are quad-trees with different depths according to the complexity of the corresponding polygon. This can lead to very deep trees, which are not desired because they need a lot of memory and time resources. Deep trees have a long construction time due to their high number of nodes and they can also lead to high render times because the traversal of large trees is time-consuming. To avoid excessively deep trees the depth is limited during quad-tree generation. In this section the results of the depth limits 2, 4, 8 and 16 are presented and discussed, which limit the subdivision of the polygons to a maximum of 4^2, 4^4, 4^8 and 4^{16} parts, respectively. For this purpose, four case studies with different polygon complexity are selected. The case studies *HORA RC* and *HORA FA* contain very complex polygons, while the case studies *Cologne SA* and *Graz SC* do not.

With a higher depth limit the polygon quad-trees are allowed to grow larger. This means only polygons are affected, which would grow further. Figure 5.13 makes clear that the case studies *HORA RC* and *HORA FA* contain more such complex polygons than the other case studies because their GPU memory and time consumption significantly change with the depth limit and the results of the other case studies are not or only slightly changing with the maximum tree depth. The increase of memory consumption is due to a higher number of quad-tree nodes and the need to store their information. Figure 5.13a shows the effect on memory consumption for the mentioned four case studies.

Quad-trees of complex polygons have to be subdivided more often with a higher value for the maximum tree depth. Since this is implemented in a sequential manner, it is a time consuming process. The resulting increase in preprocessing time caused by a higher depth limit is shown in Figure 5.13b.

The smaller the depth limit, the more edges have to be stored per leaf and the more complex are the resulting leaf polygons. A small depth limit has particularly drastic effects on the *HORA RC* case study. Figure 5.13c shows this effect by using a logarithmic scale for the runtime to be able to depict all case studies. A depth limit of two leads to a runtime that is ten times longer than with a depth limit of four, for the *HORA RC* case study. The resulting leaf polygons seem to be large and complex, so that the point-in-polygon tests take such a long time. A maximum tree depth between 8 and 16 seems to be a good choice for the performance tests that have been made. The rendering is significantly faster for the two more complex case studies with a depth limit of eight and decreases only slightly with a higher limit value.

5.2.4 Lines

There are influence factors that only affect lines, such as the line width and corner styles. All tests are conducted with the same three case studies, which are *HORA IL*, *HORA RLL*, and *Cologne SA*. They are chosen because they have the fewest lines, the most line vertices, and the highest number of lines among all case studies. During the evaluation of the line width, transparency is also tested, which is an additional influence factor. Since colors and their transparencies are handled in the same way for polygons as for static lines, the effect is assumed to be the same. Due to the pixel-based data structures of dynamic lines, the image resolution plays a particularly important role for them. For this reason, the influence of different image resolutions for dynamic lines is tested and the results are presented at the end of this section.

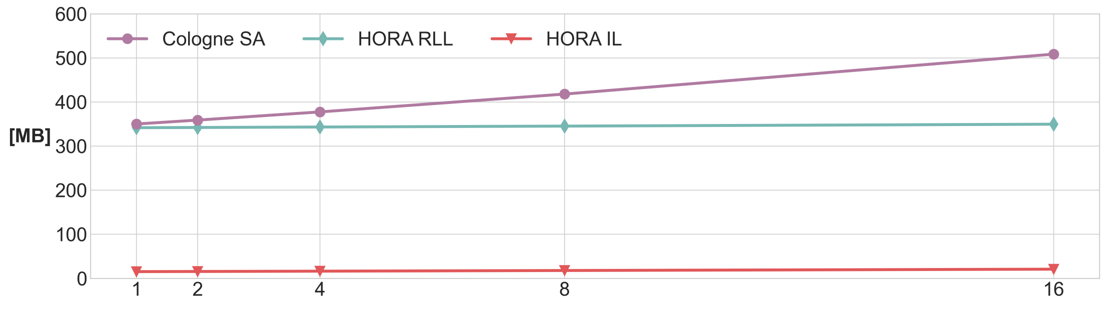
Line Sorting

To be able to compare the proposed approach with the approach of Thöny et al. [TBP17], the render times of static lines are also evaluated with different criteria for sorting the line segments within their BVHs. The test results for the sorting based on a space filling curve and based on the importance of lines do not show a significant difference in render time. Thus, an importance-based sorting is preferred to guarantee a consistent drawing order of lines.

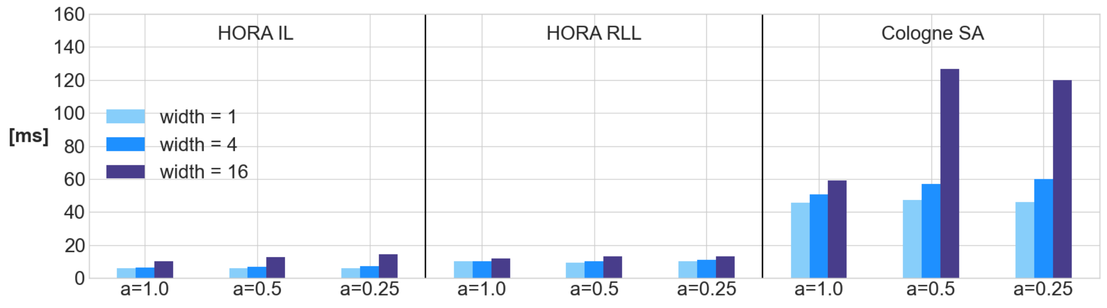
Line Width & Transparency

The width of a static line determines the extent of a line in meters in world space. For dynamic lines the line width specifies the scaling factor by which the lines are scaled in screen space.

For static lines, the line width is used during preprocessing to assign the line segments to all grid cells that they intersect. This means for a larger line width, more cells are intersected and more segments have to be referenced per cell. A consequence of more segments per cell are larger BVHs, which lead to a higher GPU memory consumption. This effect becomes visible in Figure 5.14a, especially for the *Cologne SA* case study, because the high line density causes the line BVHs to grow faster.



(a)

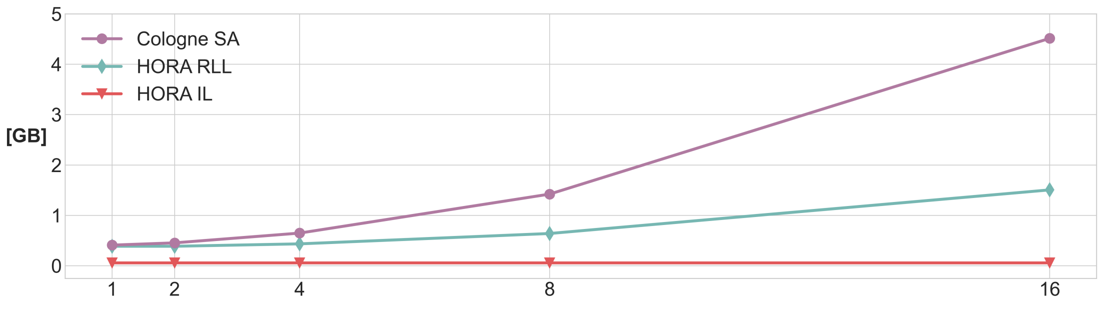


(b)

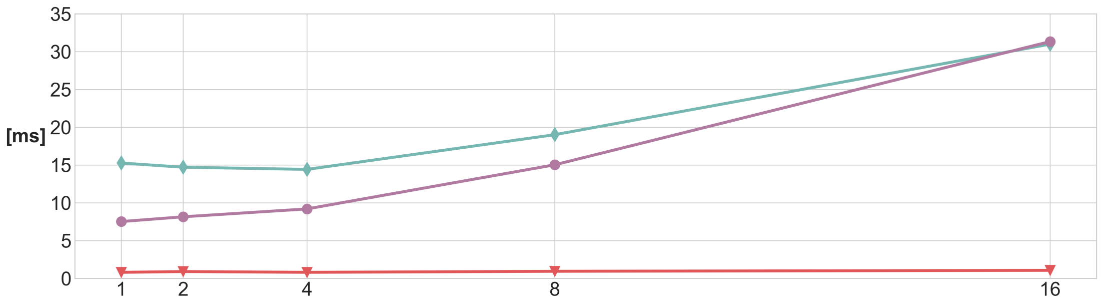
Figure 5.14: Different line widths of static lines and their impact on (a) GPU memory in megabytes (MB) and (b) render time in milliseconds (ms). The render time is also influenced by the line opacities, which is indicated by the alpha values *a=1.0*, *a=0.5*, and *a=0.25* of their RGBA colors.

Even though the line BVHs grow with the line width, there is no significant change in preprocessing time for the case studies *HORA RLL* and *HORA IL*. Only the update of the *Cologne SA* case study slightly increases with an increasing line width. This is again a result of the faster growing of the line BVHs of this case study caused by its line density.

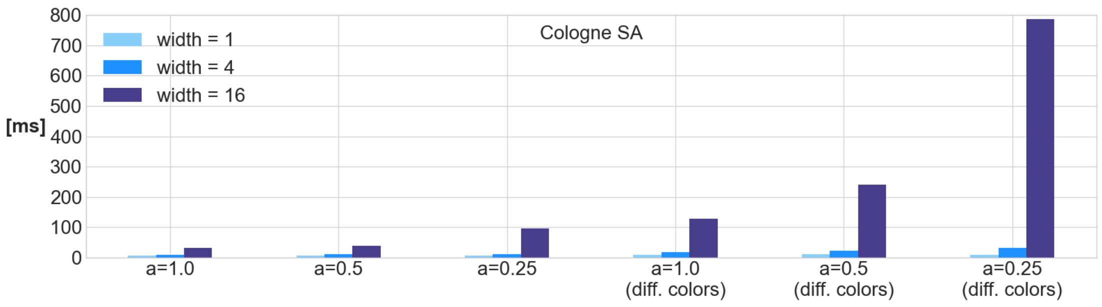
The render times also increase with higher line widths, because larger BVHs have to be traversed and the traversal can only be terminated if the composition of all detected line colors is opaque. Thus, the transparency is an additional amplification factor for the render time. It can lead to a high time consumption, especially in combination with large line widths. The extent to which the line width and opacity influence the runtime can be seen in Figure 5.14b, where the influence of the opacity of the used RGBA colors is shown by different alpha values. The alpha values 1.0, 0.5, and 0.25 represent fully, half, and one quarter opaque lines. The transparency has only a minimal effect for small line widths, but the transparency amplifies the increase in render time by an increasing width. This amplification effect can be seen particularly in the *Cologne SA* case study, where the render time for a line width of 16 doubles if transparent colors are used.



(a)



(b)



(c)

Figure 5.15: Different line widths for dynamic lines and their impact on (a) GPU memory and (b) render time. (c) The render times for *Cologne SA* also show the influence of different colors and line opacities, which is indicated by the alpha values *a=1.0*, *a=0.5*, and *a=0.25* of their RGBA colors.

The line width of dynamic lines is used differently and therefore leads to different results than the line width of static lines. As Figure 5.15a shows, it has a great impact on the GPU memory consumption, especially for the case study *Cologne SA*. Using wider lines results in more pixels covered by lines and more line data, which has to be stored per pixel. The memory consumption of the case study *HORA IL* does not change with the line width, because even the largest tested line width does not require more memory than already reserved for the pixel data. The update time is not influenced by the line width of dynamic lines because their width is dynamically changing during runtime and is therefore not used during preprocessing.

The broader the lines become, the more they overlap and the more line data contributes to a pixel. As a result of more data per pixel, the linked lists are getting longer and their dynamic generation and processing becomes more time consuming. This leads to higher render times, especially in combination with transparency and different colors. The influence of the line width, opacity, and the use of different colors on the runtime of dynamic lines is shown in Figure 5.15c on the basis of the *Cologne SA* case study. This case study is used because due to its high line density, the line width has the most influence. If the memory consumption of static and dynamic lines are compared, it becomes clear that the line width of dynamic lines has more impact on the memory usage than the width of static lines. The values increase from under one gigabyte to several gigabytes for dynamic lines and for the static lines the values only vary within a few hundred megabytes. The larger impact of the line width on dynamic lines can also be observed for the runtime, if Figure 5.14b and Figure 5.15b are compared. The greatest change of runtime happens in the *Cologne SA* case study for both line types, but for static lines the change range of about 15 ms is only approximately 30 % of the average render time and for dynamic lines the change range of about 14 ms is over 170 % of the average time. This is caused by the enlargement of the overlap areas of dynamic lines with increasing line width. The large extent of the *Cologne SA* case study additionally leads to a large scaling of the lines if the scene is viewed from further away, resulting is much thicker dynamic lines relative to the screen than static lines ever are.

If different line colors are used, an additional sorting step is necessary. After the pixel data generation step, all lines are stored in linked lists in random order. The linked lists have to be sorted per pixel according to the drawing order of the stored lines to ensure color consistency. For opaque lines only the uppermost line has to be processed and all lines below can be skipped. With transparent lines, the sorting and rendering of lines has to be executed until their composition is opaque. Depending on the number of line segments that contribute to a pixel, this can become a performance bottleneck. A line width of 16 is chosen to test extreme cases, but it is too high for most of the use cases. Such a high value can lead to a huge number of line overlaps, so that no clear lines and no details are visible anymore. The results show that larger line widths and transparent line colors should only be used if they are really required to avoid unnecessary performance drops. Especially, the use of different colors for dynamic lines should be well considered due to the high impact on the line rendering process.

Corner Styles

The style of static and dynamic lines can be modified by using three different corner styles, i.e., round, miter, and bevel.

For static lines, miter and corner positions have to be stored during prepocessing for the miter and bevel corner style, respectively. Only one 2D miter position and two 2D corner

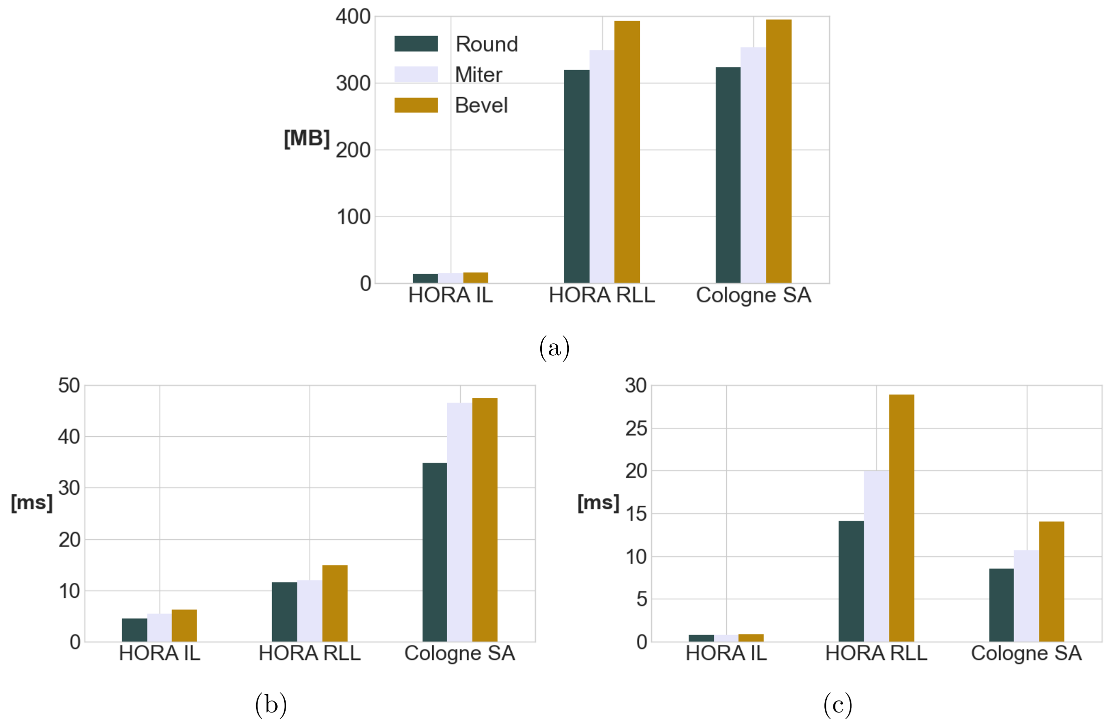


Figure 5.16: The corner styles do not have a significant effect on the update time of static and dynamic lines, but they influence (a) the GPU memory and (b) the render time of static lines, and (c) the render time of dynamic lines.

positions have to be stored per vertex, which leads to a higher GPU memory consumption for the bevel corner style, as shown in Figure 5.16a. For the round corner style no additional positions are necessary. The increase in memory consumption between the different corner styles is directly related to the number of vertices, because the vertices determine the number of corners and the required positions for the corner styles. The case studies *HORA RLL* and *Cologne SA* have almost the same memory consumption, because they contain a similar number of line vertices, while the *HORA IL* case study contains only a fraction of their number of vertices. Since the calculation of the miter and corner positions is simple and fast, there is no significant difference in the preprocessing time of lines using different corner styles.

The render times for different corner styles reflect the required number of pixel-in-line tests, which are one, two, or three for a line segment using a round, miter, or bevel corner style. Figures 5.16b and 5.16c show that this correlation holds true for both, static and dynamic, lines. One can also see that the corner style has a smaller impact on the runtime of static lines than of dynamic lines. The reason for this is that pixel-in-line tests are done for all dynamic line segments during the pixel generation process, even if the segment is occluded by another one. Static line segments are only tested if the pixel is not already fully covered. Another difference between static and dynamic lines are manifested in the case studies *HORA RLL* and *Cologne SA*, where the relative render times to each other are reversed. Static lines take longer to render in the *Cologne SA* case study, because there are more overlaps between line segments, which lead to larger line BVHs and a longer traversal. Dynamic lines are also affected by these overlaps, but the number of vertices seems to have a larger impact. The *HORA RLL* case study contains more vertices, which have to be dynamically processed during the pixel generation step. One way to reduce this additional processing time would be line simplification. Due to the constant size of static lines, such an expensive dynamic update is not required for static lines. For both line visualization methods a round corner style should be preferred over the other styles because it has the lowest memory requirements and offers the fastest line rendering of the three styles. One of the other corner styles should only be applied if sharp corners are really necessary.

Image Resolution

It is assumed that the visualization of dynamic lines with the per-pixel data depends on the image resolution. This assumption is examined in more detail in this section. The static decals are also affected by the image resolution, but mainly by the anti-aliasing and not by its general approach, such as dynamic lines. Therefore, only the influence of the image resolution on dynamic lines is further discussed.

A higher image resolution leads to a higher GPU memory consumption, which is confirmed by the results shown in Figure 5.17a. This effect is caused by the higher number of pixels, resulting in more linked lists that have to be stored per pixel. These lists contain all lines that contribute to their corresponding pixel.

The render time is also affected by the image resolution and increases with it, which can be seen in Figure 5.17b. A higher image resolution has a similar effect for all case studies, independent of the total render times and the different complexities of the line data. Since *Cologne SA* is the only case study of the three, that uses individual colors, it is used to analyze the individual render steps of dynamic lines. The distribution of the runtime to the individual render steps for increasing image resolution is shown in Figure 5.17c. The update time of the vertex indices does not change, because this step is done for all line vertices, independent of the image resolution. The pixel data generation grows with the resolution and takes the most time of all steps. The percentage increase in line sorting is the highest of all steps. This may be caused by the naive implementation of the sorting algorithm with loops and multiple memory accesses, which is another step that could be improved in the future. As a result of more pixel data that has to be processed, the render and FXAA step also increase with a higher image resolution.

5.3 Conclusion

The evaluation results show that both new screen-based approaches are able to deal with real-world use cases. The tested case studies include large-scale environments with huge

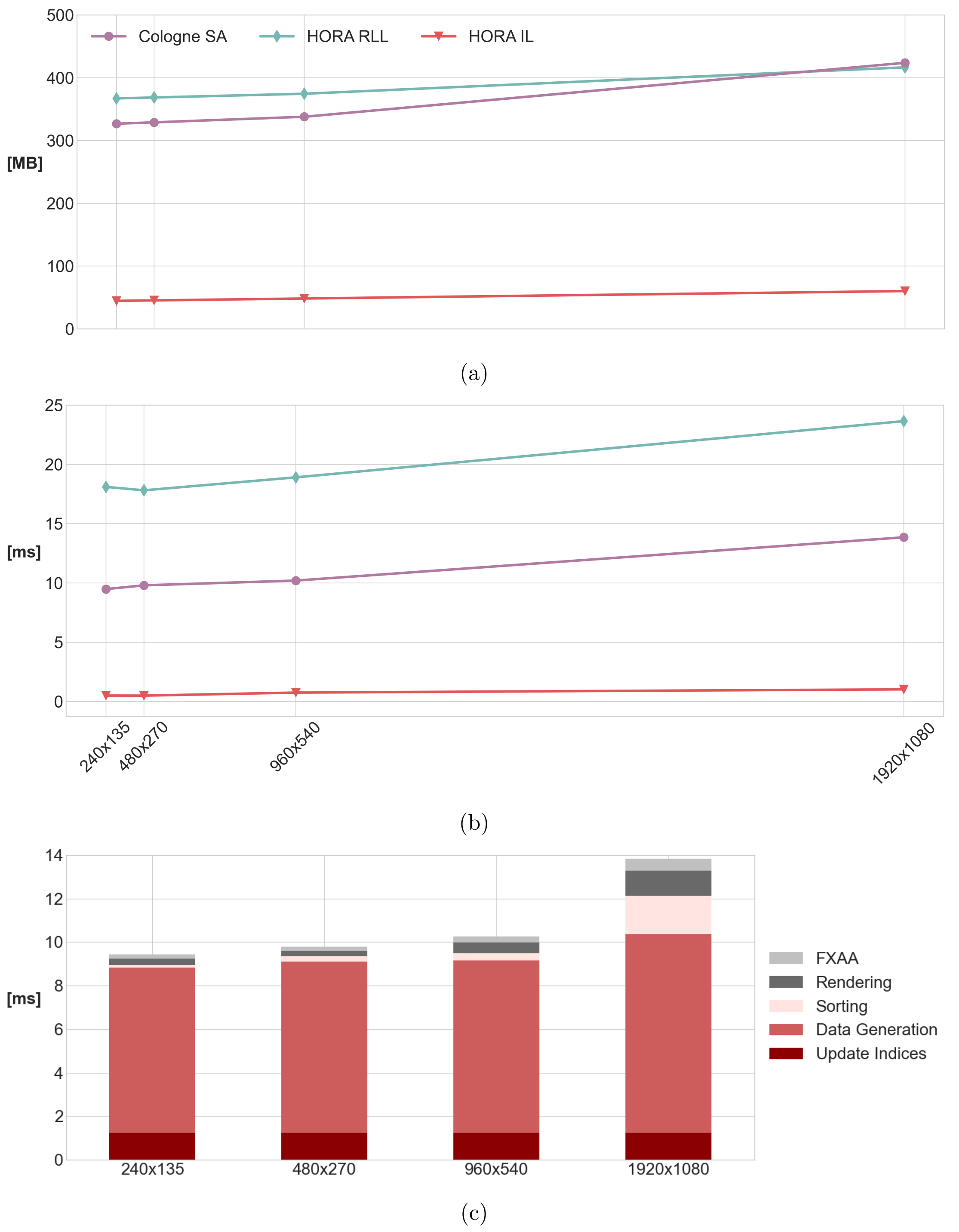


Figure 5.17: Different image resolutions for dynamic lines and their impact on (a) GPU memory and (b) render time. (c) The distribution of render time to the individual render steps for the case study *Cologne SA*.

amounts of vector data, which could still be visualized in real-time. The results also highlight that the new approaches produce larger render overheads than volume-based approaches do. But, the new approaches have the advantage that they can better handle different colors. The usage of many different colors does not lead to extremely long waiting times and makes, especially for the static decals, hardly any difference.

To conclude this chapter, Table 5.4 gives an overview on the tested influence factors of the new vector data visualization method. It also contains the impact of the factors on memory, update, and render time consumption for the different decal types. The amount and complexity of the vector data has of course an influence on all three aspects. The performance tests show, that the complexity of the line data is especially demanding for dynamic lines, because the applied line subdivision during preprocessing leads to even more data. One of the major influence factors for the static and dynamic approach is the data distribution. Both approaches are better in dealing with data that is more distributed rather than densely centered. Otherwise, more data are concentrated on one decal grid cell or pixel and the traversals of the therefore larger data structures are more expensive. By processing multiple samples per pixel, the anti-aliasing of the decals produces an additional render overhead. The anti-aliasing of static decals with MSAA requires the most part of the render time and a different approach would increase the render performance the most. The runtime of all decal types is zoom-dependent, but the zoom factor has a higher impact on dynamic lines. This is mainly caused by the view-dependent scaling that changes the degree of line overlaps.

The leaf capacity of the polygon quad-trees and their maximum depth have a higher impact on GPU memory than the decal grid resolution has. The reason for this is that polygon quad-tree data require more memory than the index-based cell data of the decal grid. By affecting polygons and lines, the grid resolution has a much higher impact on the total render time. The leaf capacity and the depth limit for quad-trees influence only polygons and particularly affect more complex polygons with a higher number of edges. For use cases with only small polygons, they can also have no effect at all.

Dynamic lines are more influenced by the line width than static lines, because they are growing larger due to the view-dependent scaling. The use of transparent colors leads to longer traversals of the data structures of both line types, because an early termination is only possible with a fully opaque color. Furthermore, the use of different colors has more impact on dynamic lines, because an additional sorting step has to be executed. For static lines only one additional memory look-up is necessary. Compared to the line width and transparency, the corner styles have a small influence on the line visualization process. The corner styles affect the GPU memory and have a small effect on the update time of static lines. Different corner styles lead also to different render times for both line types. In Table 5.4 an increasing effect on memory and render time is shown based on the corner style order round, miter, and bevel. The performance of the pixel-based approach of the dynamic lines also depends on the image resolution and becomes more expensive with a higher number of pixels.

In Table 5.4 one can see that the new screen-based approaches have many influence

	Influence Factor	**M**	**U**	**R**	**Main Reason(s)**
All Decals	Data complexity	+	+	+	More data to process
	Data distribution		−	−	Less data per cell/pixel
	Anti-aliasing			+	Multiple samples per pixel
Static Decals	Zooming in			−	More pixels covered
	Grid Resolution	+	+	−	More smaller BVHs
Polygons	Leaf Capacity	−	−	+	Fewer leaf polygons (complex)
	Quad-tree depth	+	+	−	More leaf polygons (simple)
Static Lines	Width	+	∼	+	Larger BVHs
	Transparency			+	Longer BVH traversal
	Corner Style	+	∼	+	More data, more line tests
Dynamic Lines	Zooming in			−	Less lines visible, less overlaps
	Width	+		+	More pixel data, more overlaps
	Transparency			+	Longer linked list traversal
	Corner Style			+	More point-in-line tests
	Image Resolution	+		+	More pixel data

Table 5.4: Overview of influence factors for different decal types. The symbols ∼, −, and + indicate the behaviour of GPU memory (M), update time (U), and render time (R) if the influence factor increases. They stand for *no significant effect*, *negative correlation*, and *positive correlation*, respectively. No symbol means *no effect*.

factors affecting different aspects of the visualization process. These factors can be used to further improve the visualization methods based on individual requirements. If memory is a limiting factor and the data complexity cannot be reduced in advance, the leaf capacity and the maximum quad-tree depth can be adjusted to control the memory consumption of complex polygons. For use cases with dynamic lines, the image resolution would be a good choice to lower the memory usage. If the runtime is more important, the resolution of the grid can be increased to speed up the render process of the static decals. A reduced application of transparent lines with different line colors would lead to a faster rendering of dynamic lines. For more advanced improvements, the different processes and data structures would need to be optimized to archive a more efficient rendering. The traversals of the BVHs could be further optimized to improve the static visualization approach, since traversing the tree can become expensive due to the non-disjoint branches. The visualization process of dynamic lines could be further improved by the application of a line simplification algorithm to reduce the line data that has to be dynamically processed.

CHAPTER 6

Summary

This last chapter summarizes the contribution of the thesis and gives an overview of open topics for future work. The evaluation results showed that the two developed screen-based visualization methods are capable to display vector data in interactive 3D environments in real time. The visualization method also proved to be suitable for large-scale environments with vector data sets of hundreds of thousands vector entities consisting of millions of vertices.

Compared to volume-based techniques, the proposed visualization methods scale better with the data size and are much more efficient, if many different colors are used. The new approaches have the disadvantage of a larger render overhead as compared to volume-based techniques, making it less efficient for smaller use cases with less vector data. The evaluation results also revealed that not only the data size and complexity of the individual vector entities have an impact on the performance of the proposed methods, but also the density of the vector data. The main concept of the visualization methods is the subdivision of the data into smaller parts in order to reduce the amount that has to be processed during rendering. A high data density impedes the separation, which leads to larger data chunks assigned to individual pixels and therefore to a worse render performance.

The combination of a common static screen-based visualization technique and a new dynamic approach for lines offers the flexibility to adapt the display of vector entities according to the concrete use case. Static lines are suggested for applications where the available memory is limited and the widths of vector lines should stay constant to avoid overlaps with other objects. Dynamically scaled lines are particularly suitable for large-scale scenes with a high zoom range, where static lines may disappear even if they are of interest to the viewer.

To be able to distinguish vector entities and to change the visual appearance according to the user's needs, different styles for polygons and lines are available. The color and its transparency can be used to transport additional information. Outlines with different modes can enclose polygons to highlight the boundaries. Different corner styles can be applied to change the visual appearance of lines according to the corresponding entities.

In summary, the proposed screen-based vector data visualization methods fulfill the following points, they ...

- ... are able to render large vector data sets
- ... deliver interactive frame rates
- ... are well suited for large-scale environments
- ... provide a dynamic adaption of lines to interactively changing views
- ... support different polygon and line styles

The proposed visualization methods have still some limitations, which offer interesting areas for future work. The performance of the line visualization process would benefit from an efficient screen-based line simplification. Especially, the approach of the dynamic lines has a high potential of data reduction due to the higher number of line vertices. Line simplification algorithms producing lines that are visually equivalent to their original while reducing the line data in real time are an open research problem.

Another specific problem of dynamic lines are dashing artifacts for thin lines, which can be reduced through the application of MSAA for static lines. Since MSAA is not suitable for the pixel-based approach of dynamic lines, another anti-aliasing method, such as an analytical anti-aliasing, should be applied to reduce the artifacts. Also the second-depth anti-aliasing method used by Thöny et al. [TBP17] and proposed by Persson [Per12] could prevent lines from becoming thinner than a pixel.

There are still opportunities for performance improvements of the whole visualization process, so that the preprocessing time becomes shorter and the visualization methods can be utilized for even larger data sets. Particularly, the preprocessing of static decals could benefit from a parallel generation of the tree-based data structures on the GPU. The used BVHs of the static approach could be further optimized for rendering, for example with the stack-based algorithm proposed by Áfra and Szirmay-Kalos [ÁSK14] to accelerate the BVH traversal. A different generation process or a more efficient data structure than the used BVHs should also be considered to speed up the rendering process, as their non-disjoint branches result in more time-consuming traversals.

Bibliography

[ADP20] Alireza Amiraghdam, Alexandra Diehl, and Renato Pajarola. LOCALIS: Locally-adaptive Line Simplification for GPU-based Geographic Vector Data Visualization. In *Eurographics Conference on Visualization (EuroVis)*. Computer Graphics Forum, 2020.

[AHH+18] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michal Iwanicki, and Sébastien Hillaire. *Real-Time Rendering, Fourth Edition.* CRC Press, Tayler & Francis Group, 2018.

[ÁSK14] Attila T. Áfra and László Szirmay-Kalos. Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing. *Computer Graphics Forum*, 33(1):129–140, 2014.

[BRSC17] Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Yu Chansu. Efficient Algorithms for Stream Compaction on GPUs. *International Journal of Networking and Computing*, 7(2), 2017.

[CR12] Patrick Cozzi and Christophe Riccio. *OpenGL Insights.* CRC Press, Taylor & Francis Group, 2012.

[cud] Nvidia CUDA. `https://developer.nvidia.com/cuda-zone`. Accessed: 2020-09-30.

[DP73] David H. Douglas and Thomas K. Peucker. Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[DWB+13] Erwin De Groot, Brian Wyvill, Loïc Barthe, Ahmad Nasri, and Paul Lalonde. Implicit Decals: Interactive Editing of Repetitive Patterns on Surfaces. *Computer Graphics Forum*, 33:141–151, 2013.

[DXZS13] Baosong Deng, Dong Xu, Jinxia Zhang, and Chiyang Song. Visualization of Vector Data on Global Scale Terrain. In *Proceedings of the* 2^{nd} *International Conference on Computer Science and Electronics Engineering.* Atlantis Press, 2013.

[DZY08] Chenguang Dai, Yongsheng Zhang, and Jingyu Yang. Rendering 3D Vector Data using the Theory of Stencil Shadow Volumes. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 37:643–647, 2008.

[ESV96] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing Triangle Strips for Fast Rendering. In *Proceedings of Seventh Annual IEEE Visualization'96*, pages 319–326. IEEE, 1996.

[FEP18] Alex Frasson, Tiago Augusto Engel, and Cesar Tadeu Pozzer. Efficient Screen-Space Rendering of Vector Features on Virtual Terrains. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 1–10. Association for Computing Machinery, 2018.

[FVFH90] James D. Foley, Andries Van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice, 2nd Edition.* Addison-Wesley, 1990.

[fxa] Nvidia FXAA. `https://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/opengl_samples/fxaa.htm`. Accessed: 2020-09-30.

[Gre19] Jason Gregory. *Game Engine Architecture, Third Edition.* CRC Press, Taylor & Francis Group, 2019.

[hor] Natural Hazard Overview & Risk Assessment Austria (HORA). `https://hora.gv.at`. Accessed: 2020-09-30.

[KJ19] Peter Bernard Keenan and Piotr Jankowski. Spatial Decision Support Systems: Three Decades On. *Decision Support Systems*, 116:64–76, 2019.

[LKT+17] Johannes G. Leskens, Christian Kehl, Tim Tutenel, Timothy Kol, Gerwin De Haan, Guus Stelling, and Elmar Eisemann. An interactive simulation and visualization tool for flood analysis usable for practitioners. *Mitigation and adaptation strategies for global change*, 22(2):307–324, 2017.

[MMJ00] Robert McNamara, Joel McCormack, and Norman P. Jouppi. Prefiltered Antialiased Lines using Half-Plane Distance Functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 77–85. Association for Computing Machinery, 2000.

[MS00] Andrea Mantler and Jack Snoeyink. Safe Sets for Line Simplification. In 10th *Annual Fall Workshop on Computational Geometry*, 2000.

[OC11] Deron Ohlarik and Patrick Cozzi. A Screen-Space Approach to Rendering Polylines on Terrain. In *ACM SIGGRAPH Posters*, page 68. Association for Computing Machinery, 2011.

[ogl] OpenGL. https://www.opengl.org. Accessed: 2020-09-30.

[osm] OpenStreetMap. https://www.openstreetmap.org. Accessed: 2020-09-30.

[Per12] Emil Persson. Graphics Gems for Games – Findings from Avalanche Studios. *ACM SIGGRAPH Advances in Real-Time Rendering in Games – Course Material*, 2012.

[Pre15] Nikolas Prechtel. On Strategies and Automation in Upgrading 2D to 3D Landscape Representations. *Cartography and Geographic Information Science*, 42(3):244–258, 2015.

[QWS$^+$11] Zhiyuan Qiao, Jingnong Weng, Zhengwei Sui, Heng Cai, and Xuzhao Zhang. A rapid visualization method of vector data over 3d terrain. In 19th *International Conference on Geoinformatics*, pages 1–5. IEEE, 2011.

[Rou13] Nicolas P. Rougier. Shader-Based Antialiased, Dashed, Stroked Polylines. *Journal of Computer Graphics Techniques*, 2(2), 2013.

[SGK05] Martin Schneider, Michael Guthe, and Reinhard Klein. Real-time rendering of complex vector data on 3d terrain models. In *Proceedings of the* 11th *International Conference on Virtual Systems and Multimedia*, pages 573–582, 2005.

[SK07] Martin Schneider and Reinhard Klein. Efficient and Accurate Rendering of Vector Data on Virtual Landscapes. *Journal of WSCG*, 15(1–3):59–66, 2007.

[SLL08] M. Sun, G. L. Lv, and C. Lei. Large-scale Vector Data Displaying for Interactive Manipulation in 3D Landscape Map. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 37(7):507–511, 2008.

[SLLW18] Jiangfeng She, Chuang Li, Jiaqi Li, and Qiujun Wei. An Efficient Method for Rendering Linear Symbols on 3D Terrain Using a Shader Language. *International Journal of Geographical Information Science*, 32(3):476–497, 2018.

[SM03] Wu Shin-Ting and Mercedes R. G. Márquez. A non-self-intersection Douglas-Peucker algorithm. In 16th *Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, pages 60–66. IEEE, 2003.

[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*, 99:19–27, 2016.

[TBP16] Matthias Thöny, Markus Billeter, and Renato Pajarola. Deferred Vector Map Visualization. In *SIGGRAPH ASIA Symposium on Visualization*, pages 1–8. Association for Computing Machinery, 2016.

[TBP17] Matthias Thöny, Markus Billeter, and Renato Pajarola. Large-Scale Pixel-Precise Deferred Vector Maps. *Computer Graphics Forum*, 37:338–349, 2017.

[TD19] Matthias Trapp and Jürgen Döllner. Real-time Screen-space Geometry Draping for 3D Digital Terrain Models. In 23^{rd} *International Conference Information Visualisation (IV)*, pages 281–286. IEEE, 2019.

[thr] Nvidia Thrust. `https://developer.nvidia.com/thrust`. Accessed: 2020-09-30.

[TSD15] Matthias Trapp, Amir Semmo, and Jürgen Döllner. Interactive Rendering and Stylization of Transportation Networks using Distance Fields. In *Proceedings of the* 10^{th} *International Conference on Computer Graphics Theory and Applications*, pages 207–219. SciTePress, 2015.

[vis] Visdom - Combining Simulation and Visualization. `http://visdom.at`. Accessed: 2020-09-30.

[VTW11] Michael Vaaraniemi, Marc Treib, and Rüdiger Westermann. High-quality Cartographic Roads on High-Resolution DEMs. *Journal of WSCG*, 2011.

[VW93] Maheswari Visvalingam and James D. Whyatt. Line Generalisation by Repeated Elimination of Points. *Cartographic Journal*, 30(1):46–51, 1993.

[WH18] Pascaline Wallemacq and Rowena House. Economic Losses, Poverty and Disasters: 1998-2017. *Centre for Research on the Epidemiology of Disasters United Nations Office for Disaster Risk Reduction*, 2018.

[WKW$^+$03] Zachary Wartell, Eunjung Kang, Tony Wasilewski, William Ribarsky, and Nickolas Faust. Rendering Vector Data over Global, Multi-resolution 3D Terrain. In *Proceedings of the Symposium on Data Visualisation*, pages 213–222. Eurographics Association, 2003.

[WLB09] Xianghong Wang, Jiping Liu, and Junfang Bi. Rendering of Vector Data on 3D Virtual Landscapes. In *First International Conference on Information Science and Engineering*, pages 2125–2128. IEEE, 2009.

[XSWJ10] Yunfei Xu, Zhengwei Sui, Jingnong Weng, and Xiaolu Ji. Visualization Methods of Vector Data on a Digital Earth System. In 18^{th} *International Conference on Geoinformatics*, pages 1–5. IEEE, 2010.

[ZGW+13] Ye Zhi, Yong Gao, Lun Wu, Liang Liu, and Heng Cai. An Improved Algorithm for Vector Data Rendering in Virtual Terrain Visualization. In 21st *International Conference on Geoinformatics*, pages 1–4. IEEE, 2013.

[ZPYL16] Peibei Zheng, Guoqiang Peng, Songshan Yue, and Guonian Lu. A Customizable Method for Handling Line Joins for Map Representations Based on Shader Language. *Annals of GIS*, 22:215–233, 2016.