




LOOPS: LOcally Optimized Polygon Simplification

Alireza Amiraghdam[†] , Alexandra Diehl , Renato Pajarola 

Department of Informatics, University of Zürich, Switzerland

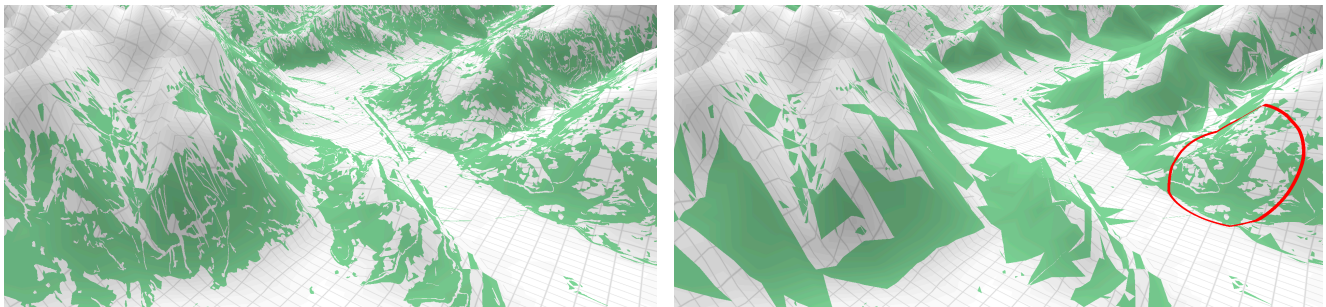


Figure 1: (left) Forest regions of Switzerland visualized over a shaded terrain model using LOOPS. (right) The red circle shows a refinement lens that demonstrates how flexibly LOOPS can simplify or refine polygons inside an area interactively controlled by the user.

Abstract

Displaying polygonal vector data is essential in various application scenarios such as geometry visualization, vector graphics rendering, CAD drawing and in particular geographic, or cartographic visualization. Dealing with static polygonal datasets that has a large scale and are highly detailed poses several challenges to the efficient and adaptive display of polygons in interactive geographic visualization applications. For linear vector data, only recently a GPU-based level-of-detail (LOD) polyline simplification and rendering approach has been presented which can perform locally-adaptive LOD visualization of large-scale line datasets interactively. However, locally optimized LOD simplification and interactive display of large-scale polygon data, consisting of filled vector line loops, remains still a challenge, specifically in 3D geographic visualizations where varying LOD over a scene is necessary. Our solution to this challenge is a novel technique for locally-optimized simplification and visualization of 2D polygons over a 3D terrain which features a parallelized point-inside-polygon testing mechanism. Our approach is capable of employing any simplification algorithm that sequentially removes vertices such as Douglas-Peucker and Wang-Müller. Moreover, we generalized our technique to also visualizing polylines in order to have a unified method for displaying both data types. The results and performance analysis show that our new algorithm can handle large datasets containing polygons composed of millions of segments in real time, and has a lower memory demand and higher performance in comparison to prior methods of line simplification and visualization.

CCS Concepts

• **Human-centered computing** → **Geographic visualization**; Visualization techniques; • **Theory of computation** → Computational geometry; • **Computing methodologies** → Rendering; Rasterization;

1. Introduction

Polygonal line and area feature vector maps are a major and central type of data in any *geographical information system* (GIS), pre-

dominantly to visualize cartographic information. Common area features found in GIS, including lakes, forests or settlement boundaries, are defined by closed polygons in 2D. We refer to such area features simply as *polygons*.

In the context of interactive 3D geo-visualization systems and rendering detailed digital elevation models, large scale polygon

[†] Authors emails: {amiraghdam,diehl,pajarola}@ifi.uzh.ch

vector data is most commonly displayed by converting them to other data types or by creating proxy data structures. Then, to support scalable visualization of large scale polygon datasets, a set of discrete levels of detail (LODs) with different accuracy are generated. However, LOD visualization is not continuous in this approach and only the closest available LOD can be chosen. Additionally, it is challenging to assign different LODs to individual classes of data, e.g. lakes and forests. Even if fetching different classes of data from different LODs is possible, conflicting LOD simplifications cannot easily be resolved. Finally, independent adaptation of the LOD of different zones on screen is becoming increasingly important in 3D geo-visualization systems. This scenario suffers from obvious seams and discontinuities in the displayed information when discrete LODs are used.

Only recently, real-time simplification and display of large polyline vector datasets has been made possible by LOCALIS [ADP20]. Nevertheless, LOCALIS is limited to non-filled polyline vector data and one specific method of line simplification. To optimally support the visualization of vector data in GIS applications and address the aforementioned challenges, real-time generalization and display algorithms are required for managing the LOD of a wider range of vector data directly. In this paper we introduce LOOPS, a novel polygon simplification approach that supports locally optimized LOD simplification and visualization of large static polygonal vector data in real-time. In particular, the contributions include:

- A novel locally-adaptive polygon simplification algorithm with a point-inside-polygon test suitable for data-parallel processing environments such as GPUs.
- Demonstration of the independence of our algorithm from the underlying simplification technique, i.e. supporting any technique that recursively discards vertices based on an error metric, like Douglas-Peucker [DP73] or Wang-Müller [WM98].
- A novel technique for drawing the outline of triangulated 2D polygons which enables us to use the same algorithm and data structure uniformly for simplification and visualization of both open line and closed polygonal vector data.
- A customizable parametric spatial indexing data structure to optimize the per-pixel LOD search, as well as an evaluation of how the performance is affected by adjusting its settings.

2. Related Work

Real-time simplification of vector map data such as lines and polygons is a well studied problem in on-the-fly cartographic generalization [WB08]. Also, real time 3D rendering of large vector map data is a well studied problem in interactive geographic visualization [DMK05]. However, combining these two disciplines, i.e. interactive exploration and display of vector data at continuously varying LODs poses new challenges. To the best of our knowledge, LOCALIS [ADP20] is the only work so far that offers efficient algorithms and data structures to satisfy local LOD demands and line-intersection queries during rendering time. While it brings these two worlds one step closer, it is limited to line simplification. In this paper we take the next step by introducing an algorithm for real-time simplification of both lines and polygons. In the rest of this chapter, we discuss the related work from these two disciplines.

2.1. Real-time Simplification Algorithms

Automatic line simplification algorithms are developed to reduce the manual work of map generalization. Douglas-Peucker [DP73] is one of the early algorithms that simplifies lines based on a distance error metric and retains the most critical points. Other simplification algorithms have different properties such as Wang-Müller [WM98], Visvalingam-Whyatt [VW93], and Zou-Jones [ZJ05] that retain critical bends, effective areas, or weighted effective area respectively, as well as H-tree [ALL05] which is particularly suitable for progressive transmission. In addition to these general purpose line simplification algorithms, there are algorithms for special use cases, e.g. for simplification of building footprints [HW10] or contours [GS04]. Automatic line simplification introduces extra problems such as unwanted self intersections or intersections with other lines which is addressed by algorithms such as Star-shaped DP [WM03], or gaps appearing where adjacent polygons share edges, which is addressed by [SC13].

In addition to a fundamental line simplification algorithm, appropriate data structures are needed for improving the performance of such algorithms when applied to large-scale datasets. The data structure can be a dynamic convex hull created at run-time [HS94], or a binary line generalization tree (BLG-tree) created in the preprocess [VOVDB89]. While these data structures work on individual entities, higher level data structures are used to enhance access and spatial queries on a larger scale of a dataset, such as Reactive-tree [VO92], GAP-tree [VO95] or Multi-VMMap [VMPR06].

Even though the aforementioned algorithms support data structures for on-the-fly generalization of vector maps, such as GiMoDig [SSS*05], their performance is far from being usable in real-time 3D rendering systems. This is true especially when the algorithm is running in a highly parallel computing environment such as a GPU, where for each pixel contributing to a line, the corresponding hierarchical data structure should be traversed once.

2.2. Vector Data Visualization

There are four categories of approaches for overlaying 2D vector data over a 3D terrain. In the first category, the vector data is rasterized and used as a texture over the terrain. Since the resolution of a single texture is limited and can appear pixelated when zoomed in, the texture-based approaches employ hierarchical texture pyramids. Such textures can be generated in a preprocess [SLL08] or at render time [KD02, WLB09]. The second category of approaches creates 3D meshes for the vector data and renders them normally as a part of the scene [WKW*03, QWS*11, WSFL10, SLL08]. These techniques adjust the generated meshes to match the terrain elevation. The terrain can also be adjusted for a better match [SLT*20]. Such matching is challenging when the terrain is not static, e.g. in variable-LOD terrains, and results in artifacts where the vector mesh and the terrain intersect. The matching can be done at rendering time, e.g. by draping the vector mesh over the terrain [TD19].

The third category of approaches creates shadow volumes by extruding the vector data into polyhedral meshes [DZY08, YZK*10, KS13]. By applying the shadow stencil technique [DZY08, YZK*10, KS13], it can be determined where they intersect with the

terrain. These methods are limited in the number of shadow volumes they create and produce artifacts when the intersections are smaller than a pixel. The fourth category draws the vector data in screen space [SZT*17, TBP16, TBP18, FEP18, ADP20] using a deferred fragment shader pass. Each pixel is back-projected into the vector data space and is tested for an overlap with any vector entity.

We use such a screen-space vector visualization technique in our work because it can handle large vector datasets, produce pixel-precise results, and is more flexible by directly accessing the vector data. Therefore, this approach is also suitable for algorithms that rely on processing the vector data while rendering them, such as the one presented in this paper.

3. Locally Adaptive Line Simplification

We first review LOCALIS [ADP20] as preliminary technology which combines a screen-space display technique called deferred vector rendering with individually refinable line segments which can be queried quickly via a 2D bounding volume hierarchy (BVH) and can be overlaid over a 3D terrain, see also Fig. 2.

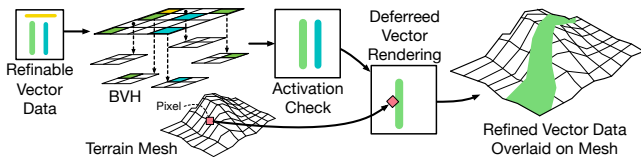


Figure 2: Refinable vector data is stored in a BVH. For each pixel of the terrain, the relevant line segments are queried from the BVH and their activation is evaluated. If an active line segment overlaps the pixel, the pixel is colored accordingly.

In the first step, a binary LOD refinement tree is created for each polyline as illustrated in Fig. 3. Each node of such a tree contains a point together with an error e , which corresponds to the error introduced by removing that point from the polyline representation, and a distance d_{max} which is the maximum distance between the point and all other points in its subtree.

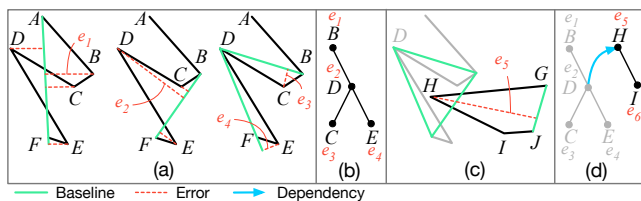


Figure 3: Process of creating refinement trees. (a) Finding the farthest points from a line segment step by step. (b) Generated binary tree. (c) Resolving intersection collisions, and (d) connecting the dependent and dependee nodes.

In the second step of the initialization, any collisions are resolved that might happen during the simplification of a polyline between line segments of itself or other polylines. Furthermore, according

to the tree hierarchy and the dependencies created from resolving collisions, the saturated values \hat{e}_s and \hat{d}_{max} are calculated such that no node has higher values than its children or any dependees.

In the third step, LOCALIS breaks down the individual line-LOD trees into individual line segments and attributes them with two special points: a *generator* and a *splitter*. The generator is one of the endpoints of an individual line segment and that line segment becomes active when this point is included in the LOD representation of the polyline. The splitter is a point that causes a line segment to be replaced by two other line segments and thus becomes inactive. In the final step of the preprocess, all the attributed line segments are stored in the BVH which is later used during runtime for fast LOD querying of line segments.

During run-time, LOCALIS integrates its locally adaptive line-LOD selection method into a deferred line rendering approach [TBP16, TBP18]. A line segment is active if its generator is included and its splitter is not included and can be determined using the formula

$$\hat{e}_g > \epsilon(d_g - \hat{d}_{maxg}) \wedge \hat{e}_s \not> \epsilon(d_s - \hat{d}_{maxs}) \quad (1)$$

where \hat{e}_g is the saturated error of generator, \hat{e}_s is the saturated error of splitter, ϵ is a LOD function that maps distance to error threshold, d_g is the distance between camera and generator, d_s is the distance between camera and splitter, \hat{d}_{maxg} is the saturated maximum distance between the generator and its underlying points in the tree, and \hat{d}_{maxs} is the saturated maximum distance between the splitter and its underlying points in the tree.

The BVH of LOCALIS is a grid which has a quadtree in each of its cells. The grid helps to limit the query quickly to a smaller set of lines around the point of interest. The quadtrees narrow down the search to a part of the set.

4. Locally Optimized Polygon Simplification

4.1. Polygon Simplification and Rendering

As polygons are defined by a closed polyline loop, they can be simplified using similar techniques as used for lines. However, displaying polygons is very different from lines in that all pixels in the interior have to be detected and shaded accordingly. Point-inside-polygon (PiP) tests include techniques such as stabbing (ray casting) or winding numbers. However, these techniques depend on a certain subset of line segments which can be far from the pixel of interest. To improve the performance of such tests, additional data structures can be employed, such as using a quadtree for each polygon [FEP18]. Nevertheless, traversing trees on the GPU is expensive and if a polygon covers many pixel, the same tree is traversed in parallel by many fragment shaders.

We explain our novel algorithm for PiP tests with the example polygon simplification illustrated in Fig. 4(a). At each step of the simplification, a vertex is removed until we reach a triangle which is the simplest non-degenerate polygon. Correspondingly, a triangle is added to or subtracted from the polygon at each step when refining the polygon. Given that structure, we can determine the PiP state by checking the type of triangles which are activated for the current LOD and cover the query point. If the number of additive is more

than the number of subtractive triangles, then the point is inside the polygon at the current LOD, otherwise it is outside.

As shown in Fig. 4(c), each pixel must only be tested for the triangles that overlap it: for P_1 , one triangle and for P_2 , three triangles need to be tested. Since the number of additive triangles for these two pixels is higher than the number of subtractive triangles, both are inside the polygon. When the polygon is simplified in Fig. 4(d) by deactivating the triangle of Step 1, an equal number of additive and subtractive triangles cover P_2 , hence, it is outside of the polygon and will be colored as background.

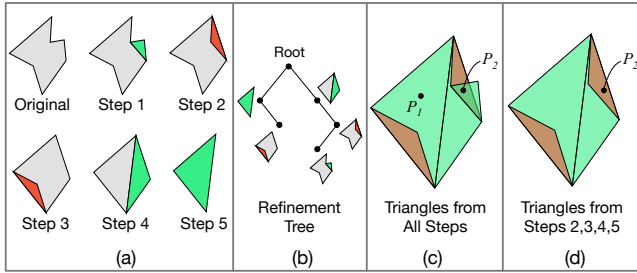


Figure 4: (a) Simplifying a polygon in 5 steps, showing additive and subtractive triangles in green and red respectively. (b) The refinement tree is created based on the inverse removal order of the vertices. (c) For activation evaluation involving all triangles, at P_1 one and at P_2 three triangles need to be tested. (d) With one step of simplification, only two triangles need to be tested for P_2 .

Finding the order in which the vertices of a polygon are removed for simplification is a classic problem with many solutions as discussed later in Section 4.3. When this order is known, a refinement tree similar to the concept of BLG-trees [VOVDB89] is built. The tree root corresponds to the the first vertex that partitions the line into two sets of vertices associated with the left and right subtrees. Recursively, the next vertex will be added to the right or the left subtree. The tree nodes store the error value of the vertex defined by the simplification algorithm. In Fig. 4, the corresponding refinement tree for the simplification process of (a) is shown in (b).

4.2. Optimized LOD Polygon Model

Although refinement trees improve the performance of refining a polygon, traversing them on the GPU in parallel for all pixels is inefficient. To address this, we employ the concept of splitters from [ADP20] which allows the activation of each node of the refinement tree being evaluated individually without knowing the rest of the tree. For polygon simplification in Fig. 4(a), the splitter of each triangle is the vertex for which its removal caused the triangle to be added or subtracted. By doing so, all refinement triangles can be stored individually in a spatial data structure, i.e. a BVH, and indexed by their splitter, to be queried at rendering time.

To better illustrate how the refinement triangles are processed independently, we use the two examples in Fig. 5. In the first example (1), we have a polygon composed of three vertices A , B , and C . This polygon can be presented in two LODs. In the first representation

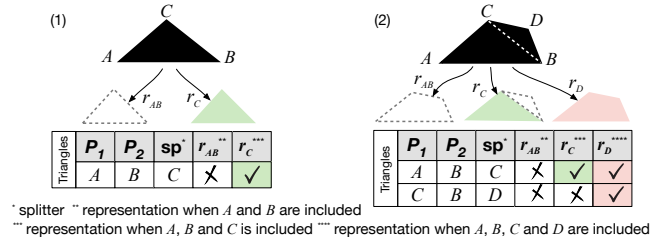


Figure 5: Two example polygons and all possible representations of them (r_{AB} , r_C , and r_D) that can emerge during refinement depending on the inclusion state of C in (1,2) and D in (2). The tables contain the refinement triangles created by LOOPS. Each representation corresponds to one of the last columns and is color coded.

r_{AB} , vertices A and B are included and C is not included. Hence the triangle is not active and the polygon is not visible. In the second representation, r_C , all three vertices are included and the polygon is visible as the triangle in green. The table shows the only primitive that LOOPS stores for this polygon. If we assume that AB is the baseline of triangle ABC , C is its splitter. The reason is that when C is not included, the polygon turns into a line with no area and is not visible. The column r_{AB} corresponds to the first representation in which the splitter is not included. The last column r_C corresponds to the green representation in which the splitter C is included.

In the second example (2) of Fig. 5, we have an additional vertex D . This polygon can thus be presented in three LOD levels: (i) as r_{AB} , when only A and B are included and nothing is visible. (ii) as r_C , when C is added and the polygon is one triangle shown in green. (iii) as r_D , when all vertices are included and the polygon is the combination of two triangles ABC and CBD shown in red. For this polygon, LOOPS stores two triangles as listed in the table. C is the splitter of triangle ABC and D is the splitter of triangle CBD . The column r_{AB} corresponds to the first representation with none of the splitters C or D included and no triangle active. Column r_C corresponds to the green representation in which only the splitter of ABC is included, but the splitter of CBD is not. The last column r_D corresponds to the red representation in which the splitters C and D of both triangles are included.

In summary, we maintain all triangles from the binary refinement trees with attributes including the splitter, saturated error of splitter $\hat{\epsilon}_s$, and saturated maximum distance of splitter \hat{d}_{max_s} . At run time, for each pixel and its corresponding real-world coordinate, we retrieve the closest triangles from the BVH. For each triangle, we then check the following inequality. If it is valid, the triangle is active and we include it in the number of additive/subtractive triangles.

$$\hat{\epsilon}_s > \epsilon(d_s - \hat{d}_{max_s}) \quad (2)$$

The difference between Eq. (2) and the second part of Eq. (1) is the inequality condition. That is because when drawing a polygon, the triangle is active and filled when its baseline is split. However, when drawing a line segment, the baseline is drawn when the segment is *not* split, thus the inequality changes from $\not>$ to $>$.

A common way of representing a hole in a polygon is to model the hole as a polygon with an opposite orientation of its vertices. We store whether a triangle is a part of a clockwise or counter-clockwise polygon as a property of the triangle. When counting additive/subtractive triangles, the counter-clockwise triangles are counted towards the opposite category, i.e. an additive counter-clockwise triangle is counted as subtractive.

Furthermore, when simplifying polygons, unwanted self intersections or intersections with other polygons can happen. To avoid these, it is sufficient to prevent intersections of the outline of the polygons. LOOPS avoids unwanted intersections based on a similar mechanism as in [ADP20]. All potential vertices which can cause intersections are identified as dependencies during the LOD generation preprocess. These dependencies between intersecting vertices are then included in the error and distance saturation process. In this way, the node corresponding to an intersecting vertex is assumed to be a child of the node which its removal causes the intersection. After saturation, the assumed parent node will have higher or equal values for \hat{e} and \hat{d}_{\max} than the assumed child node. Therefore, the intersecting node is not removed before the intersected node.

4.3. Line Simplification Algorithms

LOOPS is not restricted to a single simplification algorithm. Simplification trees can be created based on any ordered list of tuples $[t_1, \dots, t_{n-2}]$ where n is the number of vertices in a line and $t_i = (v_k, e_k)$, where v_k is a vertex and e_k its corresponding error, given that all vertices except the two endpoints appear exactly once in the list. The goal is to suggest a gradual simplification of the line by removing the vertices one by one. Each entry in the list represents the next point to be removed, assuming that the previous entries have already been removed. While in general this would follow an order of introducing the least error in each step, the error values in the list are not required to be in ascending order. The error metric can arbitrarily be chosen according to the intended purpose.

Following the inverse order of simplification, the last entry in the list is assigned as the root node of the binary LOD refinement tree. Afterwards, each entry is recursively added to the left or right subtree based on its index in the original polyline.

To demonstrate the independence of the simplification algorithm, we adapted the Wang-Müller line simplification algorithm which is known for retaining critical bends [WM98]. We first partition the line into *bends* at points where the inflection angle sign changes. In Fig. 6(a), the line has negative inflections at P_2 , P_3 , P_4 , and positive inflections at P_5 and P_6 . Therefore, the inflection sign changes between P_4 and P_5 and the bends will be $P_1P_2P_3P_4P_5$ and $P_4P_5P_6P_7$. To each identified bend, we assign an error value equal to the area of the polygon created by the bend if it was closed. The bend with the smallest error will be removed first in the Wang-Müller simplification. In this example, the bend $P_4P_5P_6P_7$ is smaller and removed first. When it is removed, in Fig. 6(b), the first point (P_4) and last point (P_7) of the bend remain in the line and are directly connected as a new segment.

Removing a bend affects its adjacent bends, so we update the neighboring bends accordingly. Removal of the red bend changes the green bend and together they turn into the blue bend in

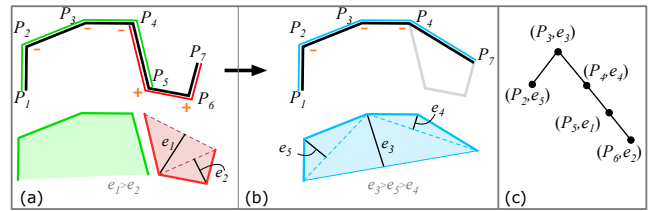


Figure 6: Using Wang-Müller for simplification and Douglas-Peucker for fine details. (a) Initial bends. (b) Recalculated bends after removing the red bend. (c) The resulting refinement tree.

Fig. 6(b). The blue bend is the only remaining at this stage, and will be removed next. Removing the red bend, removes P_5 and P_6 , and removing the blue bend removes P_2 , P_3 and P_4 . For each bend individually we run the Douglas-Peucker algorithm, or some other alternative, to find an order of the vertices to be removed. In the red bend, P_5 will be removed after P_6 as it is farther from the baseline, giving rise to error e_1 . The order of removal is thus $[(P_6, e_2), (P_5, e_1)]$. Likewise, the order of removal in the blue bend will be $[(P_4, e_4), (P_2, e_3), (P_3, e_3)]$. Considering that the red bend is removed first, the final ordered list is $[(P_6, e_2), (P_5, e_1), (P_4, e_4), (P_2, e_3), (P_3, e_3)]$, and the corresponding simplification tree is shown in Fig. 6(c).

Similarly, we can integrate other simplification algorithms such as Visvalingam-Whyatt [VW93] that retains the effective areas or Zou-Jones [ZJ05] that retains the weighted effective areas.

4.4. Triangle-based Line Drawing

The number of triangles that LOOPS store for a polygon is significantly lower than the number of line segments of the polygon's outline. For instance, in the first example of Fig. 5, the polygon has three line segments but LOOPS stores one triangle. In the second example, two triangles are stored for five line segments. To benefit from this fact and to have a unified algorithms for drawing both data types, we generalized LOOPS also for line drawing.

To correctly draw the outline of a polygon, only the sides of the outer refinement triangles should be drawn without drawing the sides that are inside the polygon. While splitters are enough for drawing filled polygons, generators are additionally needed for drawing the outlines. We have defined six rules based on the triangle being generated and/or split as illustrated in Fig. 7: (1) not generated: draw none of the sides. (2) generated, but not split: draw the base only. (3,4,5,6) generated and split: do not draw the base, draw any side that has no splitter. As the rules suggest, we need to store two additional attributes about the line segments: if the left/right side has a splitter or not. We do not need to know if the left/right side is actually split or not, we just need to know if the splitter exist.

Fig. 8(a) shows an example of applying these rules. We have a polyline $ABCDEF$ and its refinement triangles based on the Douglas-Peucker algorithm. Fig. 8(b) shows the triangles after applying the above rules assuming that B and C are excluded from the representation of the polyline. The triangles are shown in dif-

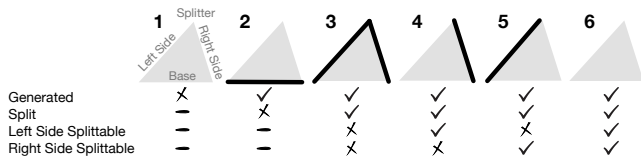


Figure 7: The six rules for correctly drawing the sides of the refinement triangles of a polygon that are needed for drawing its outline.

ferent colors only to be distinguishable. The red triangle AFD is generated since it corresponds to the base line, and split since D is included. Its left and right sides have splitters, so it falls into rule 6: no sides are drawn. For this triangle, we do not need to know if the sides (AD and DF) are actually split or not. The blue triangle DAB is generated since D is included but it is not split since B is not included. So it falls into rule 2: only the base is drawn. The orange triangle BDC is not generated since B is not included. So it falls into rule 1: no sides are drawn. The green triangle DFE is generated since D is included and split since E is included. None of its sides have a splitter, so it falls into rule 3: left and right sides are drawn. Eventually the drawn sides together form the expected line in Fig. 8(c). The table is divided into two parts: The static part is created during the preprocess and does not change during the visualization. The dynamic part is calculated in run time and its content in this example is specific to the representation shown in Fig. 8(c).

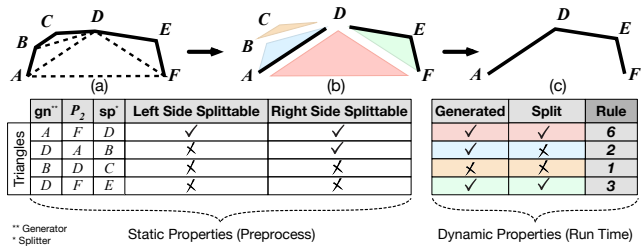


Figure 8: (a) A line and its refinement triangles. (b) The rules of triangle-based drawing applied to the refinement triangles in case B and C are excluded. (c) The simplified line without B and C .

4.5. Customizable Parametric BVH

The BVH is a major component of LOOPS to manage the attributed LOD triangles and efficiently query them during the deferred rendering phase, and it directly influences the performance and memory consumption. Designing an efficient BVH that yields a good ratio between performance and memory is not trivial and often involves some compromises. Therefore, instead of incorporating a fixed structure of the BVH into our algorithm, we designed a parameterized BVH that can be customized according to a use case.

The 2D BVH of LOOPS is composed of a primary tree, and a secondary tree nested in each node of the primary tree. The maximum depth of the primary tree is the first parameter of the BVH that can be adjusted. At each level, a primary node can partition

the space into four or any $n \times n$ regions. With four children at each level, it is thus equivalent to a quadtree. However, the branching factor of the primary tree is flexible, and with maximum depth of D_p corresponds to an array $BF_p = [b_1, \dots, b_{D_p}]$. A tree of depth 1 has a root and its direct children as leaves. A primary tree node is only subdivided if the number of its primitives is higher than some threshold PN_p which is the third parameter of the BVH.

The fourth parameter of the BVH is the maximum depth of the nested secondary trees which depends on its level in the primary tree. Secondary trees in nodes closer to the root of the primary tree cover more area and thus benefit from a higher depth. Hence, this parameter is an array $D_s = [d_0, \dots, d_{D_p}]$. The branching factor of the secondary trees $BF_s = [b_1, \dots, b_{max(d_0, \dots, d_{D_p})}]$ is the fifth parameter.

The primary tree serves the purpose of efficient space subdivision and the attributed triangles are maintained in the secondary trees of its leaves. All nodes in the secondary trees can store LOD primitives based on a coverage measure. This coverage measures the percentage of the child nodes that overlap with the primitive up to a certain depth CD_s from the current node. For instance, in case of a quadtree, there is a maximum of 16 children at 2 levels depth ($CD_s=2$). If a primitive overlaps with 8 of them, the coverage is 50%. The minimum coverage needed to store a primitive in an intermediate node is MC_s and is the last parameter of the BVH.

Optimization of these parameters depends on the size of the dataset, its distribution of data, and the available memory. In Section 6, we present the parameter settings of our experiments. They can be used as a starting point for other datasets and hardware.

5. Implementation

The transformation of the input vector data and terrain into a visualization is divided into two stages: preprocess and run time. During the preprocess, the input data is transformed and stored in the buffers on the GPU as shown in Fig. 9. In the first rendering pass, the G-buffer is filled with normal, world position, depth and terrain color for each pixel. In the second rendering pass, the world position from the G-buffer is used as the input of the LOD function to calculate the error tolerance and draw the vector data (lines and polygons) accordingly. In the last pass, the whole G-buffer is used to shade the terrain and integrate the drawn vector data.

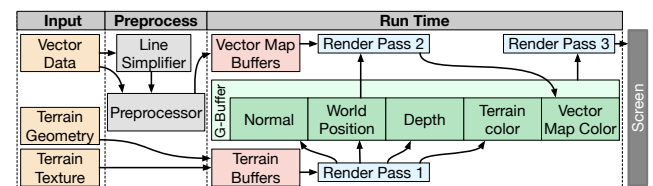


Figure 9: Overall architecture of LOOPS

Fig. 10 depicts the steps of how the input data is transformed into what is used in the shaders. Both line and polygon primitives are treated in the same way. In the first step, the simplification function generates a sequence of points and errors for each primitive. Then, for each sequence, a refinement tree is generated. After that, the

collisions between the nodes of the trees are detected which results in the trees being connected by dependencies. Using all the connections, the e and d_{max} values are saturated. Next, the refinement triangles are extracted and stored in a BVH. Finally, the refinement triangles are stored in three blocks of memory: *Points*, *Triangles*, and *Trees*, and each block is loaded to a texture buffer.

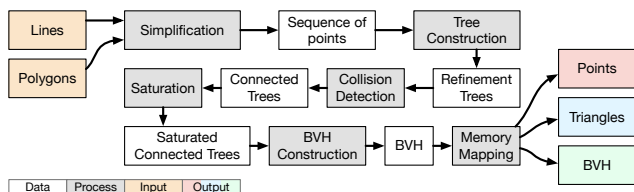


Figure 10: Data flow during the preprocess of LOOPS

Fig. 11 shows how the three blocks of memory are structured. Each point is stored in two blocks of four float values, in which six values are stored: (x, y) coordinates of the point, (x, y) coordinates of the proxy, \hat{e} , and \hat{d}_{max} . For the points without a proxy, the coordinates of the point is stored for the proxy too. With this structure, a part of the memory space is wasted. Instead, all data needed for evaluating the activation of a triangle can be read in a single texture access at run time which retrieves four values. If the triangle is active, the last two values are read for drawing.

Each triangle is stored in one block of four integer values: pointers to the two base points and the splitter. The fourth value is used bit-wise to store the following information: which one of the base points is the generator (1 bit), the drawing style (5 bits), if the triangle is clockwise or counter-clockwise (1 bit), if it is the root of the primitive (1 bit), if the line between P_1 and splitter is splittable (1 bit), and if the line between the splitter and P_2 is plittable (1 bit).

The BVH buffer contains three types of data structures: primary nodes, secondary nodes, and triangle arrays. As shown in Fig. 11, the primary node is composed of a fixed-size and a free-size part. The fixed-size part contains the index of the root of the corresponding secondary tree, and the number of the children. The free-size part contains the indices of the children nodes and its size depends on the number of the children indicated in the fixed-size part. The secondary nodes have the same structure as the primary nodes but they store the index of a triangle array instead of the index of a secondary node. In a triangle array, we store the number of triangles at the first position followed by the triangle pointers consecutively.

6. Results

In our experiments we used the following three datasets. (1) Water dataset: representing all water bodies of Switzerland (20,230 polygons consisting 1,900,164 line segments in total). (2) Forest dataset: representing areas registered as forest in Switzerland (212,466 polygons consisting 13,390,598 line segments in total), and (3) Street dataset: containing the whole street network of Switzerland (1,323,900 lines consisting 11,156,345 line segments in total). Experiments are done at 1920×1080 on a 3.5GHz Core i7-3770K, 16GB RAM, GeForce GTX 1080Ti running Windows.

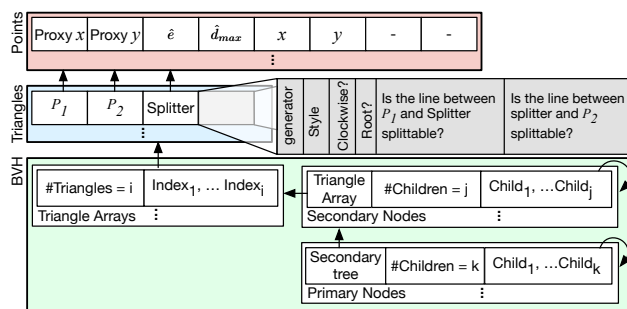


Figure 11: Structure of the data buffers for points, refinement triangles, and BVH.

6.1. Polygons

Our tests with both water dataset and forest dataset, show that LOOPS can pixel-precisely visualize highly detailed polygons. It can handle polygons with multiple holes no matter how fine the details are. The simplification happens interactively as expected with no artifacts. Fig. 12 shows an example of a complex polygon, lake harbor area, from the water dataset.



Figure 12: A part of lake Zurich with very fine polygonal details visualized over a textured terrain by LOOPS.

Fig. 13 shows an experiment demonstrating how the simplification affects the polygons when the LOD level is related to the distance from the camera, such that the error on screen equals one pixel as projected from world coordinates. Using two cameras, the main camera (Cam 1) controlling the LOD and consequently the amount of simplification, while the second (Cam 2) is fixed and located closer to the terrain. We move the main camera gradually, from Zoom 1 to Zoom 4, towards the secondary camera allowing us to observe the changes. In Zoom 4, the output of both cameras are finally, having reached the same position. Due to the error tolerance set for this experiment, the LOD changes that are visible in Cam 2 happen at subpixel level and are not visible in Cam 1. The heatmaps show the number of active triangles (additive/subtractive) increasing as the camera Cam 1 gets closer to the terrain.

6.2. Wang-Müller Simplification

Wang-Müller (WM) simplification algorithm [WM98] is known for preserving curves. Fig. 14 shows a lake from the water dataset with

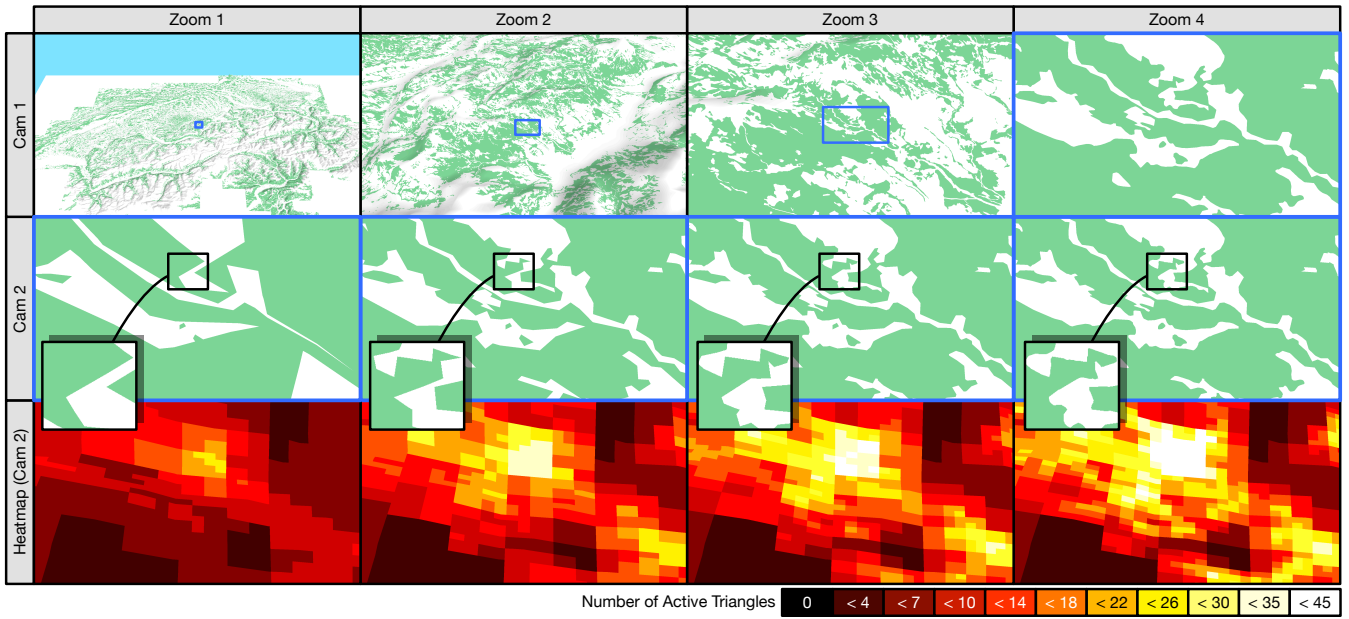


Figure 13: Main camera (Cam 1) moving towards a fixed secondary camera (Cam 2). The adaptive LOD is related to the distance from Cam 1, therefore, the LOD changes visible in Cam 2 are not visible in Cam 1. The heatmaps show the number of active triangles.

one main polygon which has two small holes and a small separate polygon. The lake is simplified both with DP and WM in four different ways. In the two *uniform* columns, the polygons are uniformly simplified for two different thresholds. WM keeps smoother curves compared to DP, however, according to the heatmaps, the number of active triangles for the same LOD threshold is higher when using WM than DP.

In the last two columns of Fig. 14, a refinement lens is placed over the lake at two different positions to demonstrate the locality of simplification. The LOD inside the lens is at the highest and outside at the lowest level. Hence, nothing is required to be refined outside of the lens. But, still a minimum amount of detail is visible since LOOPS has a conservative approach and this detail is in fact necessary to maintain the continuity of polygons. As the lens moves, LOOPS adapts the LOD constantly. By assigning an appropriate LOD level to visible parts in a real use case like the one in Fig. 13, the changes of the LOD are not visible since they are outside of the viewing area or happen at subpixel level. The heatmaps in Fig. 14 show that LOOPS is more conservative with WM than DP resulting in more extra details outside of the refinement lens.

The results show that the use of different simplification algorithms does not affect the functionality of LOOPS, however, the visual quality of the simplified vector data varies and should be chosen according to the use case's purpose and criteria.

6.3. Triangle-based Line Drawing

As explained in Section 4.4, LOOPS uses the same data structure for both lines and polygons. Drawing lines using the refinement triangles instead of line segments enables LOOPS to avoid half of the primitives required for drawing lines using line segments,

while producing identical visualizations. For instance, LOOPS stores 9,998,600 triangles instead of 20,988,790 line segments as in [ADP20] for the street dataset. Since a primitive can appear multiple times when it overlaps multiple nodes of the BVH, the actual number of primitives stored in the BVH is different and depends on the settings of the BVH. We discuss this together with the data structure in Section 6.4.

6.4. BVH and Performance

Different settings for parameters of the BVH affect the amount of processed data that should be stored on the GPU and also the performance of the algorithm. Tab. 1 contains the parameter settings of all BVHs, named in the first column, as discussed in this section.

BVH	D_p	BF_p	PN_p	D_s	BF_s
replica	1	[16]	0	[0,10]	[4,4,4,4,4,4,4,4,4]
constant	4	[100,100,100,100]	45	[0,0,0,0]	[]
descending	4	[576,256,100,36]	45	[0,0,0,0]	[]
ascending	4	[25,64,169,324]	45	[0,0,0,0]	[]
peak	4	[25,625,625,25]	45	[0,0,0,0]	[]
flat6	6	[25,25,25,25,25,36]	45	[0,0,0,0,0,0]	[]
flat7	7	[16,16,16,16,16,25,25]	45	[0,0,0,0,0,0,0]	[]
flat8	8	[9,9,16,16,16,16,16,16]	45	[0,0,0,0,0,0,0,0]	[]
size-efficient	6	[9,9,9,9,9]	45	[8,7,6,5,4,3,2]	[4,4,4,4,4,4,4,4]
efficient	6	[9,9,9,16,36]	45	[8,7,6,5,4,3,2]	[4,4,4,4,16,16,4,4]
even	6	[9,9,9,9,9]	45	[4,4,4,4,4,4]	[4,4,4,4]

Table 1: Depth of the Primary tree (D_p), Branching Factor of the Primary tree (BF_p), maximum Primitive Number of Primary tree nodes (PN_p), Depths of the Secondary tree (D_s), and Branching Factor of Secondary tree (BF_s) of BVHs. In all BVHs Coverage Depth (CD_s) is 2 and Maximum Coverage (MC_s) is 100%.

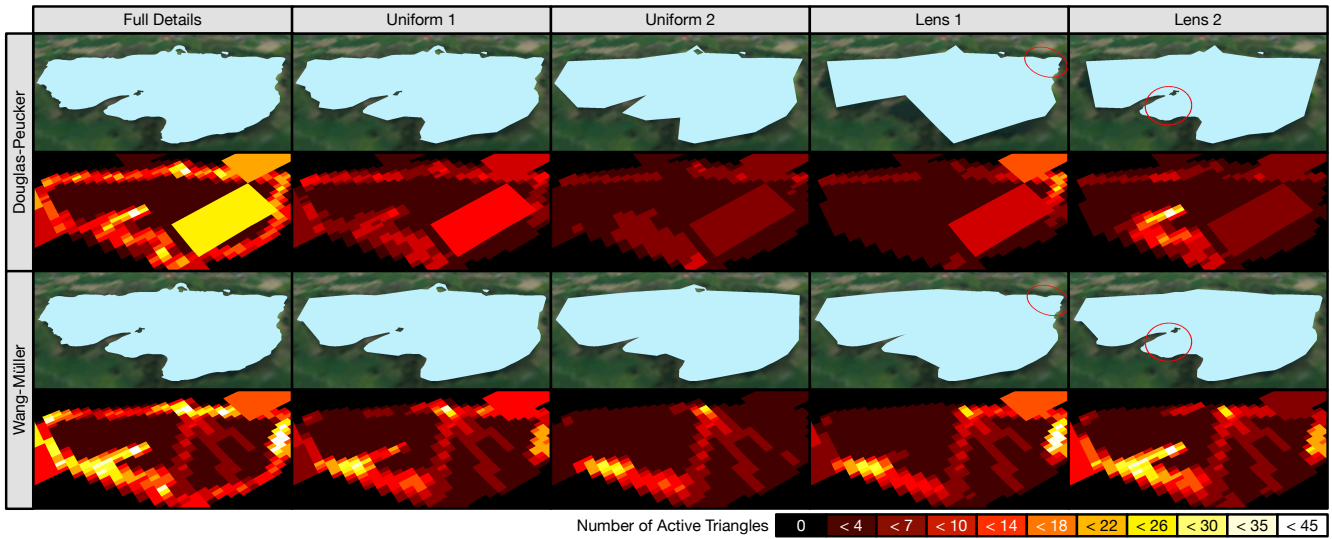


Figure 14: Lake Amsoldingen rendered by LOOPS with different simplification levels using Douglas-Peucker and Wang-Müller techniques. Left to right: with full details, uniform simplification applied at two levels, and a refinement lens placed at two positions. The locally optimized part inside the lens has the highest LOD and outside the lowest LOD. The heatmaps show the number of active triangle.

First, we compare the line simplification and visualization performance of LOOPS with the most recent method for real-time simplification of vector data [ADP20]. We replicated the test environment, using the same BVH structure and dataset, and measured the performance with the same viewing angles. The parameters for this replicated BVH (*replica*) are shown in Tab. 1. We ran this test with both techniques of line drawing: based on refinement line segments and refinement triangles. In Tab. 2, we compare memory consumption and performance of these two cases with previous work. In addition, we compared these cases to a basic BVH with constant branching factor, called *constant* in Tab. 1, to have a baseline for further comparisons. The comparison between the first and second rows shows that the line drawing of LOOPS performs better, even with a similar technique. Changing to triangle-based line drawing, reported in the third row, both memory consumption and performance improve. The last row shows that also the constant BVH structure performs better but at the cost of higher memory cost.

BVH	# ap-segs	Mem. for ap-segs	# bvh-segs	Mem. for bvh-segs	time scene 1	time scene 2	time scene 3	time scene 4
ADP20	21M	320.2MB	75.1M	2.1GB	152ms	58.2ms	39.2ms	74ms
SEG	<i>replica</i> 21M	320.2MB	93M	1.1GB	84ms	38ms	33ms	33ms
TRI	<i>replica</i> 10M	152.5MB	56M	773MB	58ms	27ms	28ms	29ms
TRIcons	<i>constant</i> 10M	152.5MB	119.5M	1.81GB	57ms	24ms	18ms	19ms

Table 2: Memory consumption and performance visualizing the street data in four ways: as reported in [ADP20], segment-based (SEG) or triangle-based (TRI) LOOPS with *replica* settings, and triangle-based LOOPS with *constant* settings (TRIcons). The table reports the number and memory used for storing all-possible primitives (*ap-prims*), primitives assigned to the BVH (*bvh-prims*), and rendering time for four test scenes, as shown in Fig. 15(1-4).

In order to compare BVHs with different settings, we defined a trajectory and measured their runtime performance along it. The

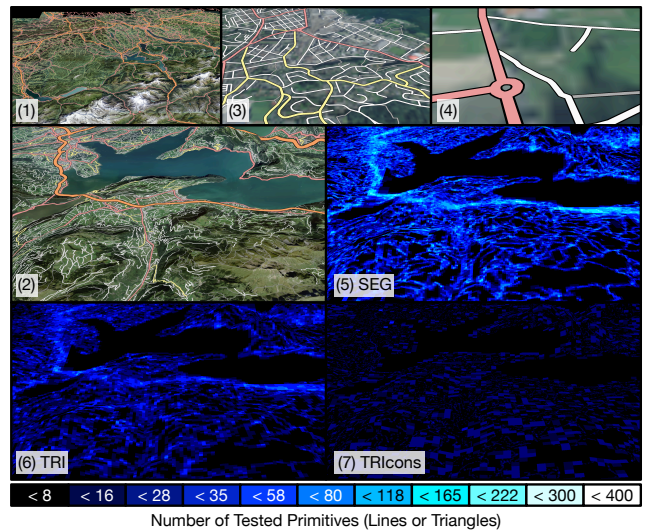


Figure 15: (1-4) Four scenes used in experiments of Tab. 2. (5,6,7) Three heatmaps for scene (2) depicting the number of primitives (lines or triangles) tested for activation, corresponding to experiments SEG, TRI, and TRIcons respectively.

trajectory is composed of 4 phases: A zoom in from far away (0-20%), panning sideways at a fixed height (20-40%), look around (40-60%), and fly through a valley (60-100%).

In the first comparison, we investigate how an ascending or descending branching factor in the primary BVH tree affects the performance. To answer this question, we changed the BF_p parameter

of *constant* and created two BVHs, called *descending* and *ascending* (see Tab. 1). Also we compared these to a BVH with a higher branching factor in middle nodes, called *peak*. The branching factors are chosen in a way to roughly use the same amount of memory. Fig. 16(a) shows the frame time along the trajectory and the memory consumption for these BVHs. The charts show that the flat branching factor, *constant* BVH, performs better than the others.

Next, we investigate how the depth of the primary tree affects the performance, i.e. comparing a deep tree with lower branching factor to a shallow tree with higher branching factor. In Fig. 16(b), we compare BVHs *flat6*, *flat7*, and *flat8* with *constant* (from Tab. 1). The branching factors are increased overall and higher in the lower layers to have comparable memory usages. Results indicate that flat trees with higher depth than *constant* are not beneficial.

By introducing the secondary tree, the number of possible BVHs configurations increases rapidly. In our experiments with different settings we encountered two useful BVHs which are shown in Fig. 16(c). The *size-efficient* BVH using only 297MB performs on the same level as BVHs with higher memory demand. The *efficient* BVH offers an excellent memory-performance ratio, while the *even* BVH is a simple yet still effective BVH.

Fig. 16(d) shows the frame times for rendering the terrain, traversing the BVH, and drawing the triangles. The chart shows that the distribution between these tasks is stable throughout the trajectory. While the time needed for rendering the terrain is insignificant at 0.9ms average, the BVH traversal is demanding. In summary, the analyzed BVH configurations provide a variety of starting points for visualizing large vector datasets. Still, the settings need to be adjusted to the dataset, available memory, and computation power.

7. Conclusions

In this paper we presented LOOPS, our locally optimized polygon simplification algorithm for real-time visualization of large-scale vector-based 2D line and polygon data. Our approach is capable of handling large datasets composed of millions of line segments forming polylines or polygons. The LOD simplification function is highly adaptive, can vary throughout the scene, and it can be set to a screen-space error tolerance below one pixel.

LOOPS is effective for two main reasons. First is the simplicity of the primitives composing the LOD data. LOOPS transforms both polylines and polygons into a set of attributed triangles that are easy to store, maintain and process. Second is the autonomy of the LOD primitives. The attributed triangles carry enough information to evaluate their activation state independently, that is without any information about other adjacent or nearby primitives. This property is ideal for data-parallel computing environments like GPUs or multi-threaded CPUs. Even though LOOPS can run on less parallelized configurations, a performance comparison with a sequential algorithm is considered for future work.

In addition, we have demonstrated that LOOPS does not depend on one specific simplification algorithm. Any simplification algorithm that exports a sequence of points and their LOD errors for a polyline or polygon, thus indicating in which order the points can be removed, can be integrated into LOOPS. Choosing a suitable

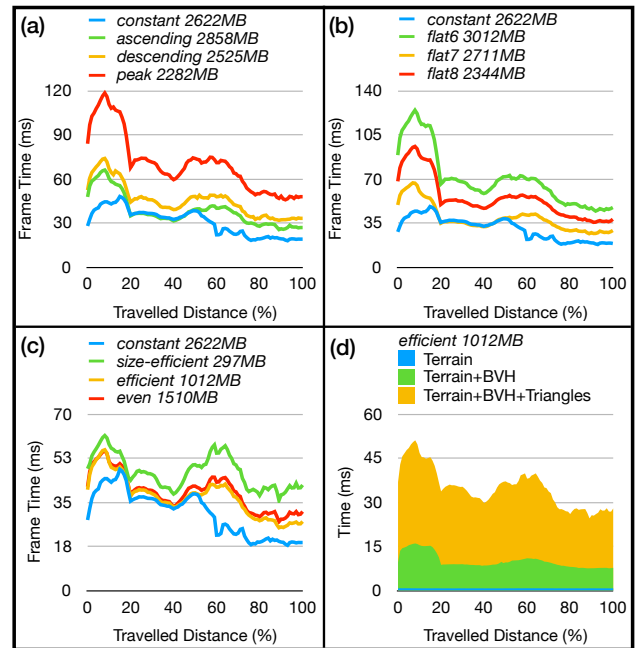


Figure 16: Performance of LOOPS along a trajectory when the BVH has (a) only a primary tree with flat, ascending or descending branching factors, (b) only a primary tree with different depths, and (c) secondary trees. (d) Frame time spent for rendering the terrain, traversing the BVH, and drawing the triangles.

simplification algorithm depends on the use case. While the collision avoidance mechanism successfully works with the data used in this paper, it is subject of improvement for datasets with a higher rate of collisions, e.g. where polygons share boundary sections.

Moreover, we have implemented a customizable BVH that allows us to explore its parameter space. By heuristically adjusting the parameters of the BVH, we have found highly performant settings that were unlikely to be found without such an available and explorable parametrization. Our method of exploration is not meant to cover the whole space and is prone to staying in local minima. Using the customizable BVH to methodically find an optimal setting, for a dataset and hardware, is a topic for future work.

With this work we are one step closer to combing on-the-fly vector map generalization and real-time, adaptive-LOD vector data rendering. However, we still require further methods that can perform other generalization operations such as selection, exaggeration or displacement interactively. LOOPS clearly demonstrates that real-time generalization and visualization of 2D vector data in a 3D environment is feasible.

Acknowledgements

The authors want to thank the Swiss Federal Office of Topography Swisstopo for providing the Swiss VECTOR25 and SwissTLM data sets. This project was partially supported by a Swiss National Science Foundation (SNSF) research grant (project no. 200021_169628).

References

- [ADP20] AMIRAGHDAM A., DIEHL A., PAJAROLA R.: LOCALIS: Locally-adaptive line simplification for GPU-based geographic vector data visualization. *Computer Graphics Forum* 39, 3 (June 2020), 443–453. [2](#), [3](#), [4](#), [5](#), [8](#), [9](#)
- [ALL05] AI T., LI Z., LIU Y.: Progressive transmission of vector data based on changes accumulation model. In *Proceedings Developments in Spatial Data Handling* (2005), Springer, pp. 85–96. [2](#)
- [DMK05] DYKES J., MACÉACHREN A. M., KRAAK M.-J.: *Exploring geovisualization*. Elsevier, 2005. [2](#)
- [DP73] DOUGLAS D. H., PEUCKER T. K.: 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 (1973), 112–122. [2](#)
- [DZY08] DAI C., ZHANG Y., YANG J.: Rendering 3D vector data using the theory of stencil shadow volumes. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 37 (2008), 643–647. [2](#)
- [FEP18] FRASSON A., ENGEL T. A., POZZER C. T.: Efficient screen-space rendering of vector features on virtual terrains. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2018), pp. 7:1–10. [3](#)
- [GS04] GÖKGÖZ T., SELÇUK M.: A new approach for the simplification of contours. *Cartographica: The International Journal for Geographic Information and Geovisualization* 39, 4 (2004), 37–44. [2](#)
- [HS94] HERSHBERGER J., SNOEYINK J.: An $O(n \log n)$ implementation of the douglas-peucker algorithm for line simplification. In *Proceedings Annual Symposium on Computational Geometry* (1994), pp. 383–384. [2](#)
- [HW10] HAUNERT J.-H., WOLFF A.: Optimal and topologically safe simplification of building footprints. In *Proceedings Sigspatial International Conference on Advances in Geographic Information Systems* (2010), pp. 192–201. [2](#)
- [KD02] KERSTING O., DÖLLNER J.: Interactive 3D visualization of vector data in GIS. In *Proceedings ACM SIGSPATIAL International Conference on Advances in GIS* (2002), pp. 107–112. [2](#)
- [KS13] KOLIVAND H., SUNAR M. S.: Survey of shadow volume algorithms in computer graphics. *IETE Technical Review* 30, 1 (2013), 38–46. [2](#)
- [QWS*11] QIAO Z., WENG J., SUI Z., CAI H., ZHANG X.: A rapid visualization method of vector data over 3D terrain. In *Geoinformatics* (2011), IEEE, pp. 1–5. [2](#)
- [SCI13] SHI S., CHARLTON M.: A new approach and procedure for generalising vector-based maps of real-world features. *GIScience & Remote Sensing* 50, 4 (2013), 473–482. [2](#)
- [SLL08] SUN M., LV G., LEI C.: 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 (2008), 507–511. [2](#)
- [SLT*20] SHE J., LIU J., TAN J., DONG J., BIAO W.: Local terrain modification method considering physical feature constraints for vector elements. *Cartography and Geographic Information Science* 47, 5 (2020), 452–470. [2](#)
- [SSS*05] SARJAKOSKI T., SESTER M., SARJAKOSKI L. T., HARRIE L., HAMPE M., LEHTO L., KOIVULA T.: Web generalisation services in gimodig-towards a standardised service for real-time generalisation. In *Proceedings AGILE International Conference on Geographic Information Science* (2005), pp. 509–518. [2](#)
- [SZT*17] SHE J., ZHOU Y., TAN X., LI X., GUO X.: A parallelized screen-based method for rendering polylines and polygons on terrain surfaces. *Computers & Geosciences* 99 (2017), 19–27. [3](#)
- [TBP16] THÖNY M., BILLETER M., PAJAROLA R.: Deferred vector map visualization. In *Proceedings ACM SIGGRAPH ASIA Symposium on Visualization* (2016), pp. 16:1–8. [3](#)
- [TBP18] THÖNY M., BILLETER M., PAJAROLA R.: Large-scale pixel-precise deferred vector maps. *Computer Graphics Forum* 37, 1 (February 2018), 338–349. [3](#)
- [TD19] TRAPP M., DÖLLNER J.: Real-time screen-space geometry draping for 3d digital terrain models. In *2019 23rd International Conference Information Visualisation (IV)* (2019), IEEE, pp. 281–286. [2](#)
- [VMR06] VIAÑA R., MAGILLO P., PUPPO E., RAMOS P. A.: Multi-VM: A multi-scale model for vector maps. *Geoinformatica* 10, 3 (September 2006), 359–394. [2](#)
- [VO92] VAN OOSTEROM P.: A storage structure for a multi-scale database: The reactive-tree. *Computers, environment and urban systems* 16, 3 (1992), 239–247. [2](#)
- [VO95] VAN OOSTEROM P.: *The GAP-Tree, an Approach to 'on-the-fly' Map Generalization of an Area Partitioning*. London: Taylor & Francis, 1995. [2](#)
- [VOVDB89] VAN OOSTEROM P., VAN DEN BOS J.: An object-oriented approach to the design of geographic information systems. In *Design and Implementation of Large Spatial Databases* (1989), vol. 409 of *Lecture Notes in Computer Science*, Springer, pp. 253–269. [2](#), [4](#)
- [VW93] VISVALINGAM M., WHYATT J. D.: Line generalisation by repeated elimination of points. *The cartographic journal* 30, 1 (1993), 46–51. [2](#), [5](#)
- [WB08] WEIBEL R., BURGHARDT D.: On-the-fly generalization. In *Encyclopedia of GIS* (2008), Shekhar S., Xiong H., (Eds.), Springer, pp. 339–344. [2](#)
- [WKW*03] WARTELL Z., KANG E., WASILEWSKI T., RIBARSKY W., FAUST N.: Rendering vector data over global, multi-resolution 3D terrain. In *Proceedings Eurographics Symposium on Data Visualization* (2003), pp. 213–222. [2](#)
- [WLB09] WANG X., LIU J., BI J.: Rendering of vector data on 3D virtual landscapes. In *Proceedings IEEE International Conference on Information Science and Engineering* (2009), pp. 2125–2128. [2](#)
- [WM98] WANG Z., MÜLLER J.-C.: Line generalization based on analysis of shape characteristics. *Cartography and Geographic Information Systems* 25, 1 (1998), 3–15. [2](#), [5](#), [7](#)
- [WM03] WU S.-T., MARQUEZ M. R. G.: A non-self-intersection douglas-peucker algorithm. In *Proceedings Brazilian symposium on computer graphics and Image Processing* (2003), pp. 60–66. [2](#)
- [WSFL10] WENBIN S., SHIGANG S., FENG C., LICHAO Z.: Geometry-based mapping of vector data and DEM based on hierarchical longitude/latitude grids. In *Geoscience and Remote Sensing (IITA-GRS)* (2010), vol. 1, IEEE, pp. 215–218. [2](#)
- [YZK*10] YANG L., ZHANG L., KANG Z., XIAO Z., PENG J., ZHANG X., LIU L.: An efficient rendering method for large vector data on large terrain models. *Science China Information Sciences* 53, 6 (2010), 1122–1129. [2](#)
- [ZJ05] ZHOU S., JONES C. B.: Shape-aware line generalisation with weighted effective area. In *Developments in Spatial Data Handling*. Springer, 2005, pp. 369–380. [2](#), [5](#)