Preprint: Precomputed Safety Shapes for Efficient and Accurate Height-field Rendering

Lionel Baboud, Elmar Eisemann, and Hans-Peter Seidel

Abstract—Height-fields have become an important element of realistic real-time image synthesis to represent surface details. In this paper, we focus on the frequent case of static height-field data, for which we can precompute acceleration structures. While many rendering algorithms exist that impose trade-offs between speed and accuracy, we show that even accurate rendering can be combined with high performance. A careful analysis of the surface defined by the height values, leads to an efficient and accurate precomputation method. As a result, each texel stores a safety shape inside which a ray cannot cross the surface twice. This property ensures that no intersections are missed during the efficient marching method. Our analysis is general and can even consider visibility constraints that are robustly integrated into the precomputation. Further, we propose a particular instance of safety shapes with little memory overhead, which results in a rendering algorithm that outperforms existing methods, both in terms of accuracy and performance.

Index Terms—I.3.7.b Computer Graphics – Three-Dimensional Graphics and Realism – Raytracing; I.3.7.f Computer Graphics – Three-Dimensional Graphics and Realism – Color, shading, shadowing, and texture

1 Introduction

Surfaces in the real-world are often complex and show many details at different scales. For real-time applications, it is currently infeasible to represent such small scale variations with separate geometric primitives. On the other hand, there are many possibilities to approximate such appearance, the most common one being textures.

Standard textures only account for color, but the increasing capacities of graphics hardware allow us to shift increasing amounts of surface properties into such image-based representations. Such techniques are usually more efficient than using a geometric equivalent. One application of textures is to add detail to the appearance of a surface, as is achieved with bump-mapping, normal mapping and other techniques now standard in real-time applications, but texture-based methods can also be used to define geometric modifications of the surface. One possibility is to interpret texture values as a displacement. In other words, the texture itself becomes a height field. This will be the main aspect investigated in this paper.

There are many applications for fast height-field rendering, such as terrain rendering [1], impostors [2], or physical simulations [3]. Some applications need to take dynamic changes into account, for others the data is constant. We will focus on the latter case, were a precomputation can be used to derive information to accelerate the display. Such performance improvements are important because despite the simple definition of a height field, efficient and accurate rendering is not straightforward. Even standard texture resolutions give rise to a dramatic number of virtual primitives and it is of importance to be able to skip unneeded elements. Today, height-field rendering on the GPU is most effective with ray marching. It consists in looking for intersections at successive positions along a view ray. For accuracy, it is necessary to adapt the marching distance to not miss an intersection.

We want to ensure the correctness of our output and present an accurate, yet efficient ray-marching method for height-field rendering. After a review of existing techniques (Sec. 2), we present our motivations (Sec. 3) and summarize our contributions (Sec. 4). We then give an overview of our technique (Sec. 5) and the basic definitions. Here, we define the general procedure and the principle of our acceleration. We then detail how to accurately and efficiently precompute the needed data (Sec. 6). Potential constraints on viewpoints can be exploited to further optimize the result. Section 7 justifies one specific definition, namely the way we interpret the height-field data as a surface. Here, we also propose alterations to the algorithm that lead to higher performance if some minor quality sacrifice is acceptable. As will be shown in Section 8, our approach compares favorably in terms of precomputation times, accuracy and rendering performance.

2 Related works

Height-field rendering is useful for many contexts. Oliveira et al. [4] used a cube with height-field-augmented faces to represent complex objects. Height-field-based rendering was also used with a few layers [5], [6], [7], patch-based representations [8] or view-interpolations [9]. Such scenarios particularly benefit of

- *L. Baboud and H.-P. Seidel are with the Max-Planck-Institut für Informatik, Department 4: Computer Graphics, Campus E1 4, 66123 Saarbrücken, Germany.*
  E-mail: {lbaboud, hpseidel}@mpi-inf.mpg.de
- *E. Eisemann is with Telecom ParisTech, TSI, 46 rue Barrault, 75013 Paris, France.*
  E-mail: elmar.eisemann@telecom-paristech.de

rendering precision addressed by our method, but are considered out of the scope of this paper. Instead we will focus on the rendering process itself and address most related work. More methods for height-field rendering can be found in [10].

One possibility to render height fields is to tesselate a surface and apply a per-vertex displacement [11]. The complexity of this *displacement mapping* can be reduced with hierarchical representations [12], controlled by the deviation, general adaptive subdivisions [13], or predefined and adaptively selected patterns [14].

Image-based methods avoid the geometric workload. Oliveira et al. [4] use a prewarping, but need to transform the entire texture, preventing the technique to be output sensitive. Parallax mapping [15], [16] shifts texture coordinates to simulate deformations and Kautz and Seidel [17] rely on slicing in order to render the information. Such approximations can result in confusing appearances for deep or high-frequency structures.

On today's hardware, it is possible to execute height-field ray casting directly in the fragment shader [18]. Baboud and Décoret [19] followed Amanatides and Woo [20] and present an algorithm that leads to accurate results on dynamic height fields. We will adapt this algorithm and present a more efficient, but still naive solution in Section 5.2.1. At the expense of accuracy, higher performance can be reached, as shown by Policarpo et al. [21], possibly inspired by root finding processes. The idea is to advance along the ray with constant steps, until it falls below the height field. From there, a binary interval search (the bisection method) delivers an intersection point. Similar solutions were applied to reflection and refraction approximations [22]. Even though such methods are relatively fast, many initial steps might be needed before arriving underneath the surface and artifacts often appear for grazing views.

One simple way to increase the initial steps is an on-the-fly computed structure, inspired by classical min-max mipmaps [23], that hierarchically store lower/upper elevation limits. It can be interesting for large dynamic height fields, but not for static data because especially near-silhouette rays become costly.

For static height fields, the key to a fast rendering with maintained accuracy lies in the use of precomputed acceleration data. It is common to encode space above the relief's surface in form of some *safety shape*. During rendering this information allows for a safe ray marching (*i.e.* without missing intersections) and ensures large steps along the ray (Fig. 1).

The most general encoding consists in storing a distance value for each possible viewing ray [24], [25]. This requires a dense sampling of the five dimensional set of rays [26], related to the plenoptic function [27], inducing large memory costs and allowing only small relief textures, even after compression.

To reduce the dimensionality of the acceleration data, Donnelly [28] approximates the distance function on a regular 3D grid. Geometrically, this defines spheres that

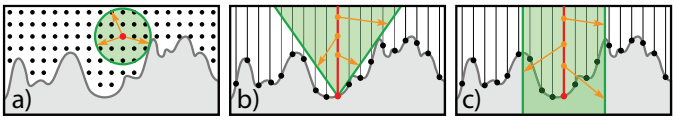


Fig. 1: Different empty space encodings; 3D grid: (a) empty spheres; 2D sampling of the surface: (b) empty cones, (c) safety cylinders. Arrows depict possible rays concerned by the corresponding safety shape. They span the maximum step length allowed by the safety volume.

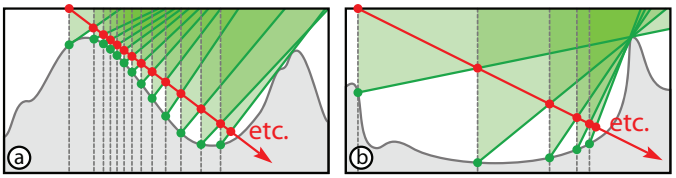


Fig. 2: Limitations of cones: (a) marching silhouette rays is slow; (b) concave zones cannot be reached in a finite number of steps (for clarity, only half-cones are drawn).

are stored in a 3D texture (Fig. 1(a)). The solution is approximate because the minimization only considers texel centers and interpolating such distances is not necessarily meaningful. Furthermore, due to the still-high memory cost, only smaller textures can be processed.

A better solution is to only use one value (or small set of values) per height-map texel which constitutes the best tradeoff between memory and efficiency. Further, the data can then be stored as a simple 2D texture of the same size as the height map.

Paglieroni et al. [29] precompute empty inverted cones (Fig. 1(b)) which enable large marching steps when far above, but only small steps when near the surface where cones inevitably shrink near their apex (Fig. 2).

Baboud and Décoret [19] define a *safety-volume property* (denoted $\mathcal{P}_{SV}$) as follows: any possible ray starting above a certain texel τ intersects the surface at most once within the safety volume at τ. They observe that the precomputed shape is actually allowed to grow past the relief surface, as long as $\mathcal{P}_{SV}$ is satisfied. With this broader definition, at most one surface intersection can occur between two stepping positions, which can be obtained accurately with an efficient binary search (Fig. 3).

In [19], this principle is used to derive a *safety radius* which defines an associated *safety cylinder* for each texel (Fig. 1(c)). The method enables a faster ray marching than previous methods, but has the drawback that the step sizes do not depend on the ray's height.

Policarpo and Oliveira [30] apply this idea to compute *relaxed cones*. These are wider than the classical (strict) ones which results in a faster rendering. Their costly precomputation algorithm results in a fast, but error-prone rendering (see Section 3 for a more detailed analysis). We will avoid inaccuracies in the precomputation and present a hybrid solution that increases effectiveness and allows us to produce accurate results.

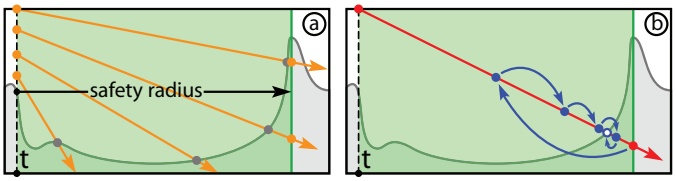


Fig. 3: Safety distance: (a) every ray originating above the texel τ intersects the surface at most once within the safety cylinder; (b) in this example, one marching step using the safety radius is enough to cross the surface: the bisection (blue) then finds the (unique) intersection.

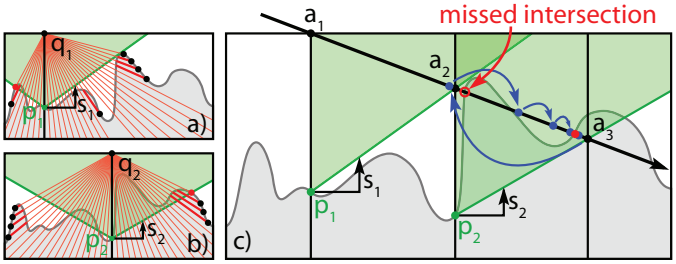


Fig. 4: A fail case example for relaxed cones [30]: (a),(b) relaxed cones slopes s_1,s_2 at positions p_1,p_2 are computed by considering second intersections of rays leaving from q_1,q_2; (b) marching along a ray using these cones leads to miss a big peak; the final bisection phase (blue) on interval $[a_1, a_3]$ (containing 3 intersections) then converges to the wrong intersection point (red dot).

3 Motivation

To motivate our work and explain the importance of our contributions, we will first analyze existing issues with recent methods.

Ensuring precomputation correctness

The preprocess of most algorithms (*e.g.* , [30], [19]) boils down to the sampling of a set of rays. This always leads to a slow and approximate computation. One of our contributions is to obtain an efficient *and* accurate result.

In particular, it is noteworthy that even the setup of such a sampling process is difficult. In [30], it was suggested to sample rays originating from the top of the relief's bounding box, above the considered texel τ and find the second intersection point along the ray. The slope stored in τ is then the largest cone not containing any of the second intersection points[1]. Fig. 4(a) and (b) show two examples.

The issue however is that no guarantee exists that cones computed via such a process will satisfy the aforementioned safety-volume property. Figure 4(c) shows a case where computed cones are too large and intersections can be missed during rendering.

1. There actually exist small inconsistencies between the preprocess algorithm described in [30] and the accompanying preprocess shader code (mainly the direction in which the sampled rays are traversed). Here, we base our discussion and figures on the algorithm described in the text. With minor adaptations, the same issues persist for the preprocess shader.

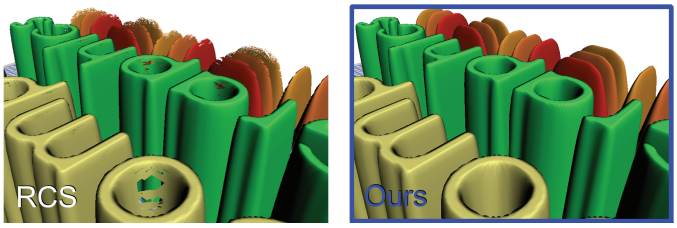


Fig. 5: Typical hole artifacts obtained with RCS [30].

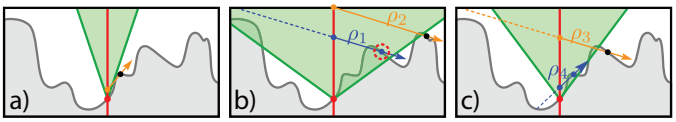


Fig. 6: Different cones definitions: (a) classical strict cone; (b) relaxed cone as defined by [30] (exterior ray ρ_1 intersects the relief twice *within* the cone, contradicting $\mathcal{P}_{SV}$); (c) ideal relaxed cone, taking all exterior rays into account (being interior, ray ρ_4 does not contradict $\mathcal{P}_{SV}$).

The problem lies in the assumption that rays originate from the top of the relief's bounding box. This choice is motivated by the idea that rays initially will be considered to originate outside the bounding box (viewpoints inside were excluded) which is a common assumption for meso-scale relief rendering. However, during rendering, after one marching step, the tested position will lie inside the bounding box. Because this location was not considered by the preprocess, the corresponding relaxed cone can be too wide (texel p_2 in Fig. 4(b)). This results in artifacts, particularly affecting reliefs with sharp features (see Fig. 5 and accompanying video). Other authors reported this problem and showed illustrations [23], but no explanation was previously found for these issues.

In order to perform a correct preprocessing, the question should be: which set of rays needs to be considered to enforce $\mathcal{P}_{SV}$? In fact, this strongly depends on the viewing context.

The most general assumption allows viewpoints anywhere above the surface (*i.e.* including locations inside the relief's bounding box). For example, this happens if the relief represents a large-scale terrain with an observer located on the ground. Interestingly, in that case, it can be shown that relaxed cones satisfying $\mathcal{P}_{SV}$ are equivalent to strict cones because rays originating from the surface need to be accounted for (see Fig. 6(a)).

If all viewpoints are located outside of the relief's bounding box, one can consider less rays, resulting in larger relaxed cones. We call *exterior* any ray originating outside the bounding box or exiting it without intersecting the relief when traced backwards (non exterior rays are called *interior*). Exterior rays are those produced by exterior viewpoints. As shown in Fig. 6(b), exterior rays (*e.g.* ρ_1, ρ_3) do not necessarily enter the bounding box above the considered texel, an assumption exploited by the computation of relaxed cones. For a correct computation [19], one needs to consider all rays originating from densely sampled positions above the considered texel,

along densely sampled directions, and disregard interior ones. This strategy allows us to compute accurate (up to sampling issues) relaxed cones (Fig. 6(c)), but would require modifying the preprocess and adding many sampling dimensions and a backtracking for a potential intersection with the surface before each ray.

Avoiding the limitations of sampling

Sampling the 5D ray space has two fundamental limitations:

1) Prohibitive computational cost for large maps. Preprocessing a n^2 height map by sampling n_z elevations and $n_\theta n_\phi$ directions costs $O(n^3 n_z n_\theta n_\phi)$ (each ray needs to be marched backwards and forwards in $O(n)$). Even with insufficient sampling [30], it takes over 8 hours for 1024^2 height maps and gets impractical for larger ones [23];
2) No exactness guarantee because –even for smooth surfaces– visibility exhibits high frequencies.

These shortcomings motivated us to develop a preprocess that is both (provably) accurate and more efficient (exploiting properties of *limiting triangles*, defined in Sec. 6). In particular, we enable a fast preprocessing by showing that 2D visibility considerations are sufficient.

Another strength of our method is its ability to integrate knowledge about viewing restrictions in form of polygonal viewing regions (*e.g.* if the relief is integrated in a scene with occlusion) and/or directional constraints. For example, if it is known in advance that the relief cannot be observed from directions above a certain angle (*e.g.* previous techniques often assume downward-directed rays), less rays need to be considered during precomputation, leading to larger safety shapes that still guarantee accurate rendering.

Improving safety shapes

Finally, even if computed correctly, one issue inevitably remains with relaxed cones; cones are infinitely thin at their apex, which can prevent the ray marching from converging when approaching the surface. Remember that height-field intersections are found by binary search once a point below the surface is reached during the ray-marching phase. There are situations however (e.g. concave parts) where relaxed cones never allow us to reach the surface (see Fig. 2(b)). Policarpo and Oliveira [30] try to deal with this issue by fixing a maximum amount of marching steps. This decision can result in wrongly-located intersection points, leading to (view-dependent) warping artifacts that are very visible near silhouettes (see Figure 7). This inaccuracy is also problematic when rendering shadows (Figure 8) because marching along a shadow ray from the light source to the surface point is likely to terminate early (shadow rays cannot be traced in the other direction, as such upwards directed rays are forbidden by the relaxed-cone technique). This early ray termination leads to virtual occlusions between the source and the surface point, that even thresholding cannot address easily.

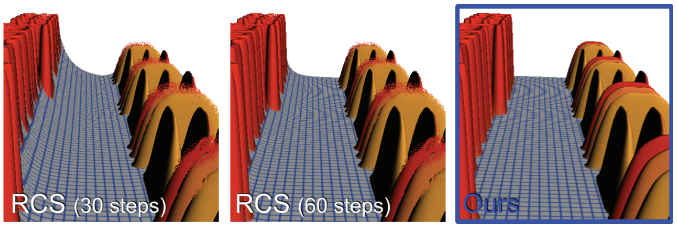


Fig. 7: Warping-like artifacts produced by RCS [30].

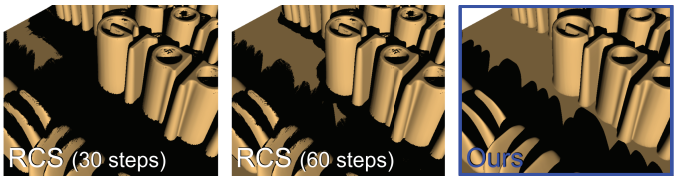


Fig. 8: Artifacts obtained with RCS [30] on shadows.

Cylinders [19] with their strictly positive radius do not have this problem (Fig. 3). To keep the advantages of both approaches, we propose a hybrid cylinder-cone shape. It enables large steps far above the relief, while the surface is reached very quickly when approaching it (Fig. 9). We will further show that the marching position update for this shape can still be done very efficiently, leading to high performance. Nonetheless, our fast precomputation is not restricted to a particular safety shape: any shape can be tested for the $\mathcal{P}_{SV}$ efficiently using our algorithm.

Improving ray traversal

More subtle, but also related, is the question of how to interpolate safety shapes between neighboring texels. Many approaches (*e.g.* [30]) compute their safety shapes on texel centers only, while marching positions can fall anywhere else inside a texel, where $\mathcal{P}_{SV}$ is not ensured. The difficulty comes from the fact that safety shapes for neighboring texels can be bound by unrelated distant parts of the relief, preventing exactness for any local interpolation scheme (see Fig. 10). To address this point, we propose a careful yet efficient ray-marching algorithm, that forces stepping positions to fall on restricted locations inside texels, for which we ensure $\mathcal{P}_{SV}$. This also requires a careful consideration of how height-map samples define the underlying relief surface and we will address this question in detail (Fig. 11 shows a challenging relief with thin features rendered accurately with our method, while existing ones fail).

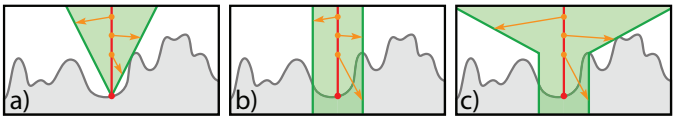


Fig. 9: Advantage of a hybrid shape: (a) cones are better suited for elevated rays; (b) cylinders perform well for low rays; (c) the hybrid shape keeps both advantages.

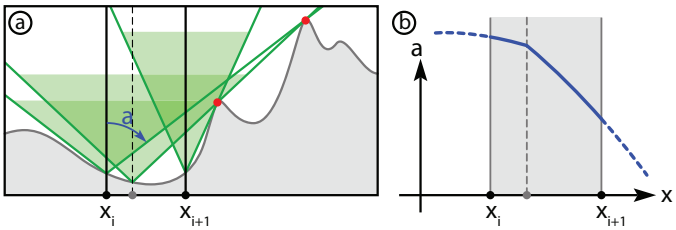


Fig. 10: Interpolation of cones inside a texel: (a) cones are bound by pivot points, changing arbitrarily when the cone position moves between two neighboring samples; (b) this results in a discontinuous derivative for the cone's angle.

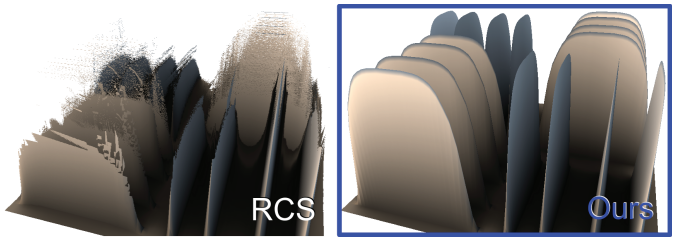


Fig. 11: A 64^2 height field with 1-texel wide features.

4 Contributions

Our paper makes the following contributions:

- **Accurate rendering**: Our solution is fast and exact.
- **Efficient and general precomputations**: Our solution is practical. We accelerate and improve the precomputation of previous state-of-the-art algorithms [19], [30].
- **Improved acceleration structure**: We present a precomputation that accelerates ray marching significantly while requiring little extra memory.
- **Exploitation of visibility**: We show how to exploit prior knowledge concerning the set of viewpoints observing the surface, leading to an acceleration at no extra memory cost.
- **Fast ray marching**: We present a novel accurate and efficient ray-marching algorithm.
- **Surface definition**: We show how to rely on a special surface interpretation to enable faster, yet accurate results.

5 Method Overview

This section presents an overview of our general rendering algorithm. We will present a simple and accurate algorithm before proceeding to our main contributions that involve precomputations in Section 6.

5.1 Surface definition

A height map is a 2D array of height values $h_{i,j}$ sampled on a regular grid (for simplicity we assume an $N \times N$-pixel square). Such a point-wise definition can be interpreted in various ways. Here, we will give a first surface definition that we rely on for the rest of the paper before discussing alternatives in Section 7.

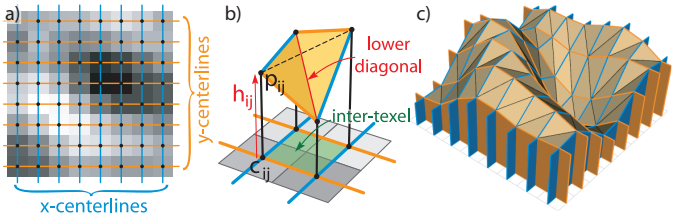


Fig. 12: Surface definition from a sampled height map: (a) height-map samples and centerlines; (b) triangulation for an inter-texel; (c) complete surface.

We start with a few definitions that can be followed along in Fig. 12. The positive height value $h_{i,j}$ defines a surface point $p_{i,j}$ above the center $c_{i,j}$ of the texel located at the integer coordinates (i, j). We define two sets of lines: the *x-centerlines* (y-centerlines) parallel to the y-axis (x-axis) and passing through the texel centers. This network of lines creates square cells that we refer to as *inter-texels*. For the sake of simplicity, we will use these expressions also for their extension along the z-axis. Consequently, we say that a ray intersects a centerline, if its projection in the x,y-plane does. Similarly, a 3D point is said to be located in an inter-texel, if its projection is.

The *surface* $\mathcal{S}$ is a triangular mesh defined by the $p_{i,j}$ vertices[2]. Above each inter-texel are exactly two triangles. There are two options for the common diagonal edge: our surface is defined by always taking the lower one (Fig. 12(b)). The resulting surface $\mathcal{S}$ is the graph of a continuous height function h, concave in each inter-texel.

5.2 Accurate rendering algorithm

The rendering is performed entirely on the GPU using a ray-casting fragment shader. A polygonal bounding volume $\mathcal{V}$ is used to initialize view rays from the eye. The challenge is to compute the first intersection of this ray with $\mathcal{S}$ encoded in a 2D texture, the so-called *height map*.

5.2.1 Naïve algorithm

The simplest way to find the first intersection consists in traversing successive inter-texels (as illustrated in Fig. 13). More precisely, a testing point p is shifted along the ray, stopping at each x- or y-centerline. Here, its corresponding height p_z is compared to the surface elevation $h(p_{xy})$. The marching stops when p passes below the surface, *i.e.* when $p_z \leqslant h(p_{xy})$, which indicates that an intersection occured. Because our defined height function is concave in each inter-texel, the intersection point has to lie inside the last traversed inter-texel, *i.e.* between the two last test positions. Finding the intersection amounts to testing against the inter-texel's two triangles.

2. We discuss other surface definitions in Sec. 7

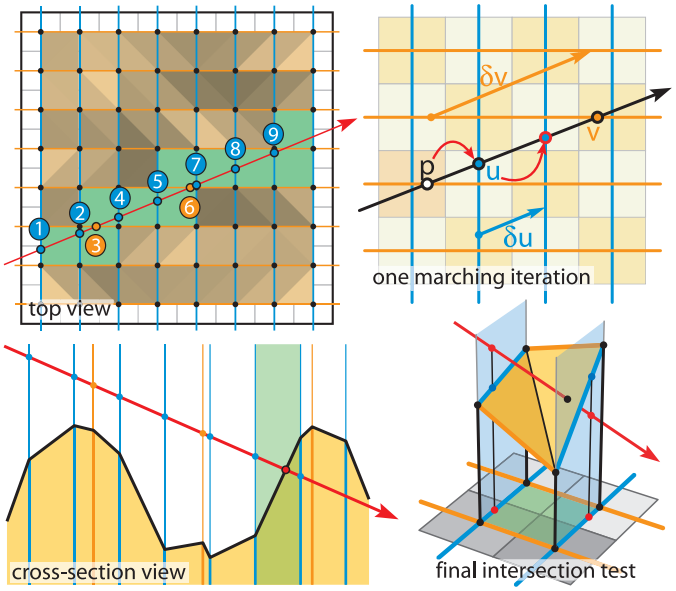


Fig. 13: Successive steps of the naive marching algorithm.

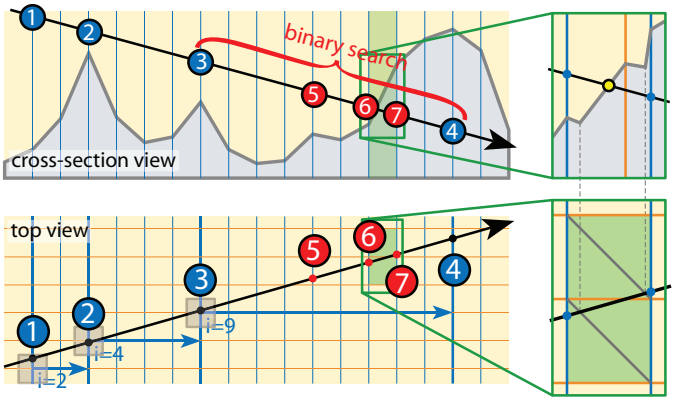


Fig. 14: The improved algorithm: the three first steps fetch a value i from the precomputed safety distances map to jump on x-centerlines, the fourth step falls below the surface, which switches to the bisection phase.

The grid traversal algorithm by Amanatides and Woo [20] provides an efficient way to update the test positions. Along a given ray, there exists a constant translation vector δu (δv) to jump from one x-centerline (y-centerline) to the next one (Fig. 13, top right). To determine successive test positions in the correct order, these two separated sets of centerline intersections need to be considered in an interleaved fashion.

Concretely the algorithm works as follows: A first initialization stage places p on the first intersected centerline. From here, we keep track of two *step points* u, v which represent the intersection with the next x-, y-centerline, respectively. During the marching stage the test position p is shifted to the closest amongst u and v. The chosen step point is updated using its corresponding translation vector and the process is reiterated.

An interesting property of our surface definition is that above centerlines, the triangulated surface matches a bilinear interpolation (and usually only there): as p is always located on a centerline, the surface elevation $h(p_{xy})$ can be determined via a bilinear texture lookup in the height map.

This algorithm requires $2N$ iterations in the worst case, which can become costly for very large height maps.

5.2.2 Improved algorithm

To reduce the number of steps in the algorithm, we are inspired by approximate solutions that rely on fast root finding and precomputed data (whose computation is detailed in Section 6). The latter will enable bigger steps during the marching phase, but ensures that only one surface intersection occurs between two successive test positions. Under such conditions, the bisection method delivers the accurate intersection point.

We start by associating the ray to one of four sets ($\mathbb{R}^{x+}$, $\mathbb{R}^{x-}$, $\mathbb{R}^{y+}$, $\mathbb{R}^{y-}$) depending on its direction. These four sets indicate the axis in the x, y-plane that the ray's direction mainly follows. E.g., let $r(t) := o + td$ be a ray, then $r \in \mathbb{R}^{x+}$ iff $d_x \geqslant |d_y|$.

In a preprocess, we derive a 2D texture for each of the four ray sets, defining for each texel of the height map an integer value called *safety distance* that we use to accelerate the marching phase.

In the marching phase, test positions will always be located on one type of centerline (x-centerlines for $\mathbb{R}^{x+} \cup \mathbb{R}^{x-}$, y-centerlines for $\mathbb{R}^{y+} \cup \mathbb{R}^{y-}$). In the following, we will assume that the ray we trace is in $\mathbb{R}^{x+} \cup \mathbb{R}^{x-}$.

The algorithm is illustrated in Fig. 14. Initially, from the ray's origin, we advance exactly as for the naive algorithm, until we reach the first x-centerline (Pos. 1 in Fig. 14). Here, we will start the real marching process, we fetch the safety distance value i corresponding to the current test position p from the precomputed data, then advance by i x-centerlines. The new position $p+i\delta u$ is located again on an x-centerline and we reiterate the process.

In the special case that i is zero, we proceed as for the naive algorithm and advance to the next x-centerline: if a y-centerline is crossed between p and $p + \delta u$ (it can happen at most once because the ray is in $\mathbb{R}^{x+} \cup \mathbb{R}^{x-}$), an extra test is performed.

The marching stops when the test position is below $\mathcal{S}$. We then use the bisection method between the two last test positions to determine the precise intersection. Here again, the bisection is restricted to x-centerlines, which is achieved by dividing the search interval at integer positions only. The bisection stops when the two last test positions are exactly one x-centerline apart. Again, because the ray is in $\mathbb{R}^{x+} \cup \mathbb{R}^{x-}$, the interval spans at most two inter-texels, leaving at most four separate triangles to test (Fig. 14, insets).

6 Precomputation

The previous section presented an efficient ray marching, but relied on a precomputed safety distance whose accu-

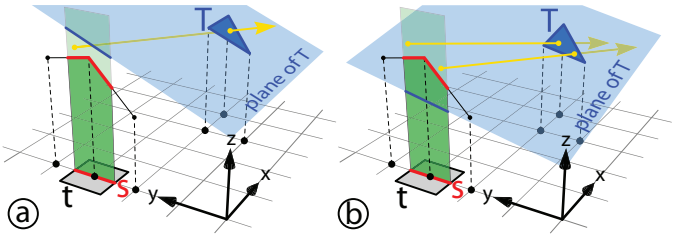


Fig. 15: Limiting triangle: (a) a ray hitting the back face of T is found, making it a limiting triangle; (b) T is not limiting as any ray leaving above $h(s)$ sees his front face.

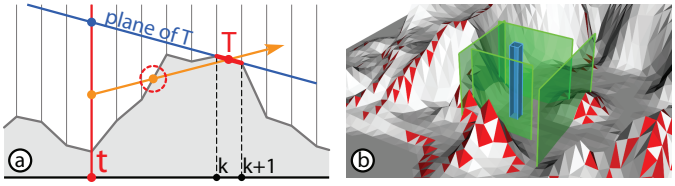


Fig. 16: Limiting triangles bound the safety distance: (a) a ray hitting the back face of a limiting triangle necessarily crosses the surface earlier, so that the safety distance is less than or equal to k; (b) safety planes for each of the four ray classes (limiting triangles are shown in red).

rate and efficient computation will be investigated in this section. We then show how we can improve these precomputed values while maintaining rendering accuracy if we have prior knowledge about the viewpoints from which $\mathcal{S}$ will be observed. Finally, we generalize this acceleration data and describe a hybrid cone-cylinder safety shape to achieve further speedups.

6.1 Simple safety distance

We will focus on the class $\mathbb{R}^{x+}$, but by rotation of the height field, the description applies to the other ray classes too.

Let's focus on a specific texel τ of the height map. Please remember that for rays in $\mathbb{R}^{x+}$, the marching algorithm only stops on x-centerlines. We call s_τ the segment that is the intersection of the x-centerline through τ with τ itself. The curve on $\mathcal{S}$ whose footprint is s_τ will be referred to as $h(s_\tau)$. We need to consider all rays in $\mathbb{R}^{x+}$ passing through a point p above $h(s_\tau)$ (*i.e.* , $p_{xy} \in s$ and $p_z > h(p_{xy})$). Let i be the safety distance that we want to compute, then the bisection method is exact only if the ray cannot pierce $\mathcal{S}$ twice between p and $q := p + i\delta u$. In other words i should be less than the distance to the *second intersection* of the ray with $\mathcal{S}$. Hence, i can be defined as the largest integer value such that this condition holds for all rays in $\mathbb{R}^{x+}$ passing above $h(s_\tau)$ (we denote this set $\mathcal{R}_\tau^{x+}$).

A simple solution is to sample this four-dimensional set of rays, but this is costly and inaccurate. Instead, we consider all discrete triangles of $\mathcal{S}$ and compute i accurately. Some observations will help us solve this problem.

Our main observation is that the safety distance is bound by special triangles of $\mathcal{S}$ that we call *limiting triangles*. A triangle T is said to be *limiting* iff there exists a ray $r \in \mathcal{R}_\tau^{x+}$ hitting its back-face (Fig. 15).

It can be shown that the safety distance i is given by the closest limiting triangle. More precisely, we show that if k is the smallest integer such that there exists a limiting triangle T between x-centerlines at distance k and $k+1$, then $i = k$.

Indeed, let $r \in \mathcal{R}_\tau^{x+}$ be a ray hitting the back face of T at a point p. Because $\mathcal{S}$ is continuous, p is at least the second intersection of r with $\mathcal{S}$. Thus, $i \leqslant k$ (Fig. 16).

Conversely, there exists a ray $r \in \mathcal{R}_\tau^{x+}$ whose second intersection p is located strictly between the x-centerlines at distances i and $i+1$ (otherwise the safety distance would be at least $i+1$). As p is a second intersection, it necessarily belongs to a triangle T', back-facing for r. By definition, T' is a limiting triangle, located between x-centerlines at distance i and $i+1$. It follows that $i \geqslant k$.

Following this observation, we compute the safety distance by traversing $\mathcal{S}$'s triangles in increasing distance from τ and stopping at the first limiting triangle. Only the subset of triangles reachable from $h(s_\tau)$ by a ray in $\mathcal{R}_\tau^{x+}$ needs to be considered (Fig. 17(a)).

Remains the question of evaluating wether a given triangle is limiting or not. This can be done very simply using the following equivalent definition: a triangle T is *limiting* iff there exists a ray $r \in \mathcal{R}_\tau^{x+}$ crossing it and lying in its plane $\mathcal{P}_T$ (such a ray will be called a *limit ray*). Equivalence of the two definitions follows from the observation that a limit ray can always be tilted into a ray intersecting the back-face of T, and vice versa.

Consequently, testing whether a given triangle T is limiting can be done by comparing $h(s_\tau)$ with $\mathcal{P}_T$. Let's call $\sigma_{\tau,T}$ the segment in $\mathcal{P}_T$ above s_τ from which rays in $\mathbb{R}^{x+}$ cross T (usually the vertical projection of s_τ onto $\mathcal{P}_T$, but not always, see Fig. 17(b)). Then T is a limiting triangle iff $\sigma_{\tau,T}$ contains a point strictly above $\mathcal{S}$.

Because s_τ is on a centerline, $h(s_\tau)$ consists of two meeting segments. Consequently, three points at most of $\sigma_{\tau,T}$ need to be tested: its two extremities and the point above the center of s_τ (Fig. 17(c)).

The simplicity and parallelizability of the resulting algorithm (one value per texel, all computations being independent) allow us to implement it on the GPU, leading to small precomputation times (Sec. 8).

6.2 Restriction to exterior rays

The previous computation did not impose any restrictions on the rays, *i.e.* we considered arbitrary rays emanating from above the surface. However, in many situations this set of rays can be narrowed down, implying larger safety distances (Fig. 18), hence, faster rendering.

As already discussed (Sec. 3), the most common assumption is that the viewpoint remains outside the height field's bounding volume, *i.e.* , view rays are

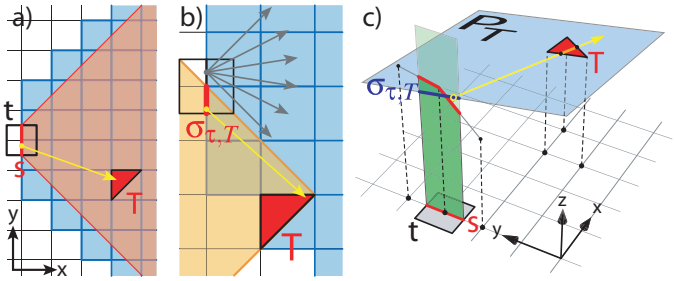


Fig. 17: Limiting triangles: (a) the $\pm 45°$ quadrant leaving from s_τ defines possible locations of limiting triangles; (b) definition of $\sigma_{\tau,T}$ in a case where it does not entirely cover s_τ; (c) here a point of $\sigma_{\tau,T}$ above $\mathcal{S}$ exists, thus T is limiting (a limit ray leaves this point).

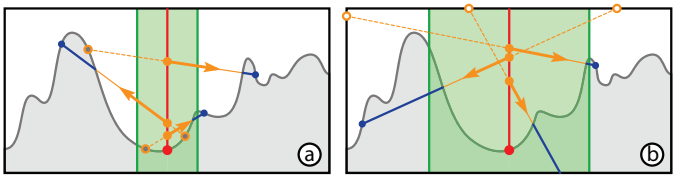


Fig. 18: Advantage of visibility restriction: (a) the safety distance defined for any ray is smaller than (b) the one for exterior rays only.

exterior rays (such as those to test for shadows by an exterior light source).

To make use of such restrictions, we will no longer consider all rays in $\mathcal{R}_\tau^{x+}$, but restrict ourselves to those that are exterior. This operation usually requires visibility computations involving high dimensionality and complex constructs [31]. For our specific context, we propose a simple and accurate solution.

We keep the basic algorithm unmodified: we process the triangles by increasing distance until a limiting triangle is found. Only, this time the definition of *limiting* will take the *reduced* ray set into account.

Denoting the limit rays of a triangle T as $\mathcal{L}_{\tau,T}$, we will reduce this set to $\mathcal{L}_{\tau,T}^{ext}$. The latter will only contain exterior rays in $\mathcal{L}_{\tau,T}$ that do not intersect the surface $\mathcal{S}$ before reaching s_τ. Only if $\mathcal{L}_{\tau,T}^{ext}$ is not empty, the triangle is limiting.

6.2.1 Testing for Limiting Triangles

The difficulty of properly handling $\mathcal{L}_{\tau,T}^{ext}$ resides in its 5D nature. However to evaluate limiting triangles, only a 2D subset needs to be dealt with.

First, we consider oriented lines instead of rays because exterior rays cannot have intersections before their origin (*i.e.* they can be arbitrarily translated backwards), which eliminates one dimension. In the following we will, hence, use the terms *line* and *ray* interchangeably.

Second, all limit rays of triangle T lie in $\mathcal{P}_T$, which eliminates two more dimensions. Therefore, we only need to deal with the two-dimensional set of oriented lines contained in $\mathcal{P}_T$.

Practically, continuous sets of 2D lines can be manipulated algebraically using a 2D parametrization. Each 2D line is thus represented by a point in this 2D *dual space*, where two helpful properties hold:

1) the set $\pi(b)$ of lines intersecting a segment b covers a 2D polygonal area;
2) the set π^{x+} of lines corresponding to rays from $\mathbb{R}^{x+}$ also covers a 2D polygonal area.

These properties reduce the computation of $\mathcal{L}_{\tau,T}^{ext}$ to a sequence of 2D polygonal CSG operations.

Before detailing the dual space, we can already describe how our visibility algorithm determines if a triangle T is limiting: In $\mathcal{P}_T$, we consider only the rays in $\mathbb{R}^{x+}$. Further, we restrict the ray set to those above[3] s_τ. This amounts to an **intersection**: $\pi(s_\tau) \cap \pi^{x+}$. Out of these rays, we want those that intersect T, *i.e.* passing through one of T's edges $(t_i)_{i=1,2,3}$. The set of rays passing through the edges is the **union** $\pi(T) := \bigcup_i \pi(t_i)$ (for a 2D triangle, two out of the three edges are sufficient to define $\pi(T)$). Our ray set is then initialized as $R := \pi(T) \cap \pi(s_\tau) \cap \pi^{x+}$. All these sets are polygonal areas in dual space, the intersections can thus be handled geometrically.

To obtain $\mathcal{L}_{\tau,T}^{ext}$ from R, we must exclude interior rays, *i.e.* those stopped by $\mathcal{S}$ before reaching T. Potential blockers are the segments (b_k), found at the intersection $\mathcal{P}_T \cap \mathcal{S}$ and strictly before s_τ. Removing blocked rays requires a **subtraction** for each blocker b_k: $R \leftarrow R \backslash \pi(b_k)$. At the end of this process, R is a representation of $\mathcal{L}_{\tau,T}^{ext}$: T is a limiting triangle iff $R \neq \varnothing$. In the case where R is not empty, a limit ray, *i.e.* an exterior ray contained in $\mathcal{P}_T$, can be disclosed by simply picking a point from R.

Next, we will detail use of the dual space which makes the solution practical. Finally, we explain how to avoid numerical issues and enable efficient computations in dual space, how to integrate other visibility constraints, and how to accelerate the entire precomputation.

6.2.2 2D Line Duality

Let ℓ be a line in $\mathcal{P}_T$, non-orthogonal to the x axis. It can be represented in a unique way by a 2D point (u, v) in *dual space* where $(0, u)$ and $(1, v)$ are its respective intersections with the planes of equations $x = 0$ and $x = 1$. This is a two-dimensional equivalent of the parallel slabs parametrization commonly used for light-fields [32], whose interesting properties have been studied and exploited to address 2D visibility problems [33], [34].

The benefit of using this parametrization comes from the underlying duality between lines and points. By definition, a line ℓ in *primal space* is represented by a point $\pi(\ell)$ in *dual space*.

Our rays in $\mathbb{R}^{x+}$ map to points (u, v) in dual space such that $|v - u| < 1$, covering a polygonal area π^{x+} in the form of an unbounded slanted stripe (Fig. 19).

3. To simplify notations, we will use $\pi(s_\tau)$ in place of $\pi(s'_\tau)$, where s'_τ is the vertical projection of s_τ onto $\mathcal{P}_T$.

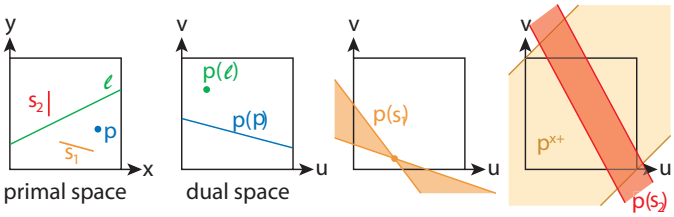


Fig. 19: Dual parametrization of 2D lines.

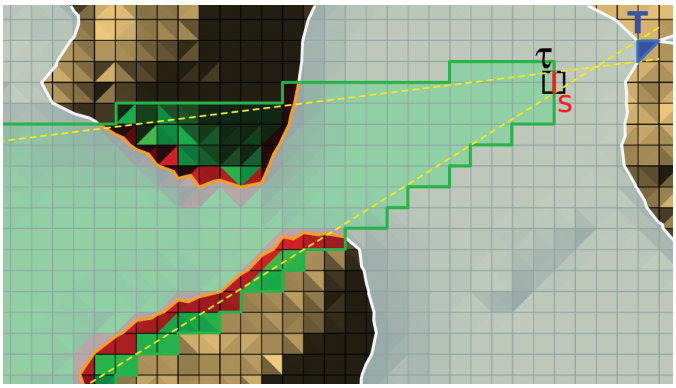


Fig. 20: Potential blockers (orange) for rays going from s to T are found in the shaft-like shape (green).

As indicated before, we are further interested in sets of lines intersecting segments. Let s be a segment in primal space, and let p be a point on s. The pencil of concurrent lines meeting at p corresponds to the set of dual points $\pi(p) := \{\pi(\ell) \ / \ \ell \ni p\}$. It is a line in dual space. Consequently, the dual set of all lines passing through s is $\pi(s) = \bigcup_{p \in s} \pi(p)$. It can be shown that these lines all meet in a point $q = \pi(\ell)$, where ℓ is the line containing s. In the special case where s is orthogonal to the x axis, the corresponding dual lines are parallel (q is at infinity).

Now, it can be seen that our algorithm does not need to handle unbounded polygons: the set of rays in $\mathcal{P}_T$ crossing s_τ and belonging to $\mathbb{R}^{x+}$, *i.e.* $\pi(s_\tau) \cap \pi^{x+}$, covers a parallelogram in dual space (Fig. 19).

6.2.3 Blocker traversal

To test for a limiting triangle, we treat its segments independently. For each, the initial rayset R is represented by one or two convex polygons. Whenever we subtract a blocker from a convex polygon, it results in one, two convex polygons, or an empty set. This follows from the particular shape of $\pi(b)$, the dual of a blocker segment b (Fig. 19). Operations on convex polygons are very simple, which makes the whole process efficient.

To achieve further acceleration, we scan the triangles of $\mathcal{S}$ for blockers by decreasing x-coordinate and only consider those in the shaft of rays going from s_τ to T. (Fig. 20).

6.2.4 Basic computations

Although our algorithm conceptually works in dual space, an explicit conversion is actually not needed because a convex polygon can be equally represented by a list of points or a list of half-planes. By duality, representing a dual half-plane can be done using a single primal point: given a point p and a line ℓ in primal space, determining the side of the line $\pi(p)$ that the point $\pi(\ell)$ is located on, is equivalent to determining this relationship for ℓ and p.

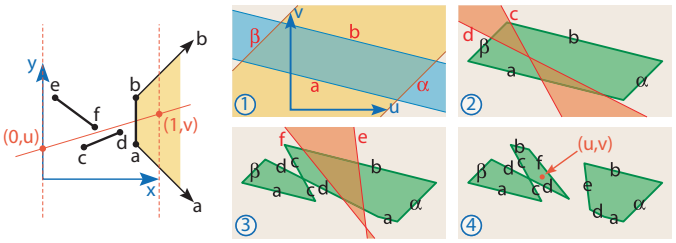


Fig. 21: Accurate rayset computation in the dual parametrization: (1) initialization with rays hitting $[ab]$, directed between $\alpha = (1,-1)$ and $\beta = (1,1)$: $R_0 \leftarrow \pi([ab]) \cap \pi^{x+} = (a, \alpha, b, \beta)$; (2) removal of rays blocked by $[cd]$: $R_1 \leftarrow R_0 \setminus \pi([cd]) = ((a,c,d,\beta),(a,\alpha,b,c,d))$; (3) further with $[ef]$: $R_2 \leftarrow R_1 \setminus \pi([ef]) = ((a,c,d,\beta),(d,f,b,c),(a,\alpha,b,e,d))$; (4) a point (red) in the residual dual area discloses a ray from $\mathbb{R}^{x+}$ hitting $[ab]$ wihout hitting $[cd]$ and $[ef]$.

Thus a convex dual polygon can be represented by an ordered list of primal points. The advantage is that now intersection operations become simple list-splitting operations (Fig. 21). Another advantage is that we can avoid numerical issues that can usually arise from duality transforms because the rayset R is now represented by a subset of the original vertices of the blockers and those of T. Only the two half-planes forming π^{x+} map to points at infinity in primal space (*i.e.* directions $(1,-1)$ and $(1,1)$). Homogeneous coordinates spare a specific treatment of this case.

6.2.5 Other visibility restrictions

For now we showed how to take self-occlusions of $\mathcal{S}$ into account to optimize safety distances. Other cases exist where visibility restrictions are known a priori.

The first case consists in occlusions due to other objects of the scene. As our algorithm handles any kind of polygonal blocker, it can be used unmodified to benefit from occlusion by static polygonal objects surrounding the height field.

The second case concerns restrictions on the viewing angle. For example, if a height field is used to represent details of a floor, the viewpoint will usually stay above some known altitude, so that an angle ϕ_{max} bounds the slope of viewing rays. Such a constraint is also easily handled by our visibility algorithm, as shown next.

Remember that we only consider rays included in $\mathcal{P}_T$. If n is a normal vector for $\mathcal{P}_T$ pointing upwards, then the minimum slope is found along direction n_{xy} (Fig. 22(a))

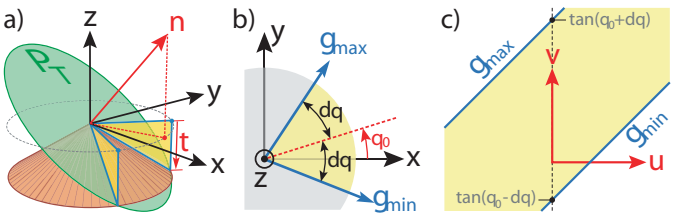


Fig. 22: Viewing angle restriction: (a) intersection of $\mathcal{P}_T$ (with normal n) with a cone of slope $t = \tan\phi_{max}$ defines a restricted angular sector; (b) 2D projection of this angular sector: θ_0 is the direction of the projection of n; (c) this corresponds to two half-planes in dual space.

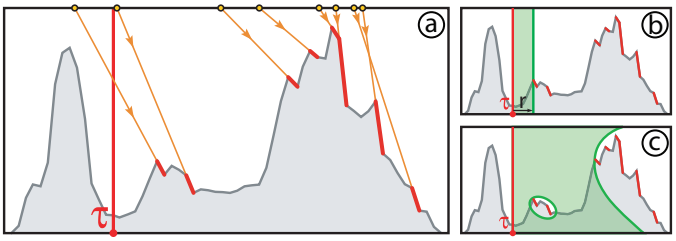


Fig. 23: Safety shapes from limiting triangles: (a) limiting triangles (red) for texel τ and associated limit rays (orange); (b) safety distance (cylinder); (c) general safety shape empty of limiting triangles.

and the general slope along direction θ is:

$$s(\theta) = -\alpha\cos(\theta - \theta_0) \quad \text{with} \quad n = \begin{pmatrix} \alpha\cos\theta_0 \\ \alpha\sin\theta_0 \\ 1 \end{pmatrix}$$

Let $t := \tan\phi_{max}$ and $d\theta := \arccos -\frac{t}{\alpha}$, then the bounded slope constraint can be expressed as:

$$s(\theta) < t \iff t > \alpha \quad \text{or} \quad \begin{cases} -\alpha < t < \alpha \\ \theta_0 - d\theta < \theta < \theta_0 + d\theta \end{cases}$$

Thus, restricting rays included in the plane to those whose vertical angle is lower than ϕ_{max} (*i.e.* whose slope is lower than t) amounts to at most two additional half-plane intersections in dual space (Fig. 22(c)).

6.2.6 Global visibility precomputation

To accelerate the precomputation, one can avoid redundant computations. First note that for a given triangle T, the rayset $\mathcal{L}^{ext}_{\tau,T}$ needs to be computed for each texel τ that "sees" it with a direction in $\mathbb{R}^{x+}$. For two such texels τ_1 and τ_2, the only difference between computations of $\mathcal{L}^{ext}_{\tau_1,T}$ and $\mathcal{L}^{ext}_{\tau_2,T}$ is the intersection with their respective x-centerline segments s_1 and s_2. Thus for each triangle T we can precompute:

$$R_T = \pi^{x+} \cap \pi(T) \cap \bigcap_k \pi(b_k)$$

thereby performing the costly blocker traversal only once. Then, $\mathcal{L}^{ext}_{\tau_1,T}$ and $\mathcal{L}^{ext}_{\tau_2,T}$ are efficiently obtained by intersecting R_T with $\pi(s_1)$ and $\pi(s_2)$ respectively. This solution requires the storage of R_T for each triangle of $\mathcal{S}$. Its size is bounded by the number of blockers b_k, *i.e.* the number of triangles of $\mathcal{S}$ intersecting $\mathcal{P}_T$. Because only visibility events (silhouettes) remain in R_T, only a very small number of vertices are left on average (typically less than five vertices per triangle), making this solution feasible even for very large height fields.

6.3 Improved safety volume

In the previous section, we showed how to compute a safety distance in order to accelerate the ray marching. More generally such a distance can be understood geometrically: for each ray set, it defines a bounding plane orthogonal to the corresponding axis. If we assume the same distance for all ray classes, we obtain a (square) cylinder. Alternatively, any other delimiting shape could be used, as long as it demarcates a volume empty of any limiting triangle (inside such a shape, a ray cannot cross $\mathcal{S}$ twice, Fig. 23). Nevertheless, there is a trade-off: a complex shape might give better safety distances, but marching and even storage can become issues. Here, we will introduce a new shape (Fig. 24) that increases the complexity of the intersection test insignificantly, while leading to a drastic performance increase.

The strongest limitation of the safety distance is that it gives the same bound for all rays, regardless of their position above a given texel. So at a location where $\mathcal{S}$ has a tiny but sharp bump, the safety distance will be small, forcing rays, even high above the surface, to slow down (Fig. 23(b)).

Previous work [29], [30] suggest the use of cones, as they provide larger distances for rays far off the surface (Fig. 23). However, as already mentioned (Sec. 3), these have an important drawback: the safety distance for rays approaching the surface converges to zero, which slows down the marching process and even prevents to eventually reach a point below the surface. For cylinders, a ray close to the surface quickly passes underneath, thus ending the marching process. Our idea is to combine the best of the two solutions.

6.3.1 Hybrid cylinder-cone shapes

To take advantage of cones, while still maintaining the accuracy and advantages of cylinders, we propose a hybrid shape, parametrized by three values: a radius r, a base height z_c and a slope s_c (Fig. 24(b)) for each ray direction class. The shape is defined by a perpendicular (cylinder) and a slanted (cone) plane.

The memory cost increases slightly (three instead of one value per texel). Nevertheless, the gain due to the large step reduction makes this solution favorable (Sec. 8). Especially, as we can derive an efficient expression to compute the safety distance for our hybrid shape at position p:

$$i = \max\left(r, \left\lfloor \frac{\max(p_z - z_0, 0)}{s_c - s_d} \right\rfloor\right) \quad \text{with} \quad \begin{cases} z_0 = z_c - r \times s_c \\ s_d = \delta u_z \mid \delta v_z \end{cases}$$

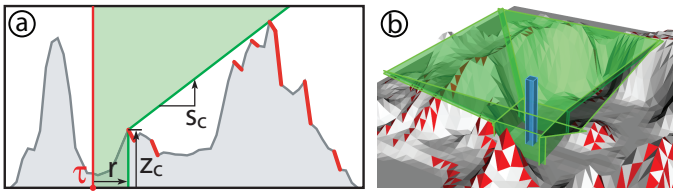


Fig. 24: Hybrid cylinder-cone safety shape: (a) the three parameters; (b) example in 3D.

6.3.2 Computation

To compute the hybrid shape parameters (as illustrated in Fig. 24b)) for a specific texel τ and the ray set $\mathbb{R}^{x+}$, we start by determining the safety distance r as described in Section 6. The second step is to compute the parameters for the slanted plane z_c and s_c. This plane has to stay above all limiting triangles for τ that are further than r.

Given the safety distance r, we chose z_c as the maximum height of the triangles at distance r to τ. This leaves us with a single degree of freedom: the slope s_c. For such a slanted plane, we find the smallest value of s_c such that all limiting triangles stay below the slanted plane. Experiments with other heuristics showed that this choice (*i.e.* fix z_c and then minimize s_c) usually yields the best results in terms of rendering speed.

Unlike the computation of the safety radius, the first limiting triangle does not necessarily define the minimum slope. Testing all relevant triangles can be costly.

Accelerating computations

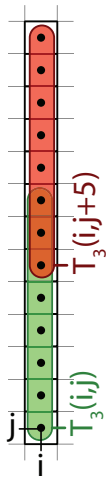
If precomputation time is an issue, the slope of the slanted plane can be conservatively approximated. Instead of individually testing each triangle, we neglect visibility and assume that all triangles beyond the safety distance r are limiting. For cones, this makes little difference in terms of rendering efficiency (Sec. 8), and allows us to accelerate the computation drastically.

To this end, we scan increasing integer abscissas x starting from r, and determine the maximal height $\bar{h}(x)$ of vertices on $\mathcal{S}$ located at abscissa x and reachable by rays in $\mathbb{R}^{x+}$ leaving above s_τ. The values of $\bar{h}(x)$ are used to compute the slopes $s(x) = (\bar{h}(x) - z_c)/(x - r)$ whose maximum is kept as s_c.

Reading all height values for a given abscissa x can be avoided using a modified version of N-Buffers [35]. This structure is derived once in a GPU preprocess. We compute $\log(N)$ textures $\mathcal{T}_l$ ($l \in \{1..\log(N)\}$), each with a resolution of N^2. Each texel (i, j) of the l^{th} texture contains the maximum of a $2^l \times 1$ window along the y-axis:

$$\mathcal{T}_l(i,j) := \max_{k \in \{0..2^l-1\}} \{h(i, j+k)\}$$

The construction of these textures is done recursively. Each texture is constructed with only two lookups from its predecessor. With these textures, $\bar{h}(x)$ can be obtained with only two lookups, chosen such that their corresponding windows cover the values that need to be tested. This makes the algorithm well-suited for the GPU.



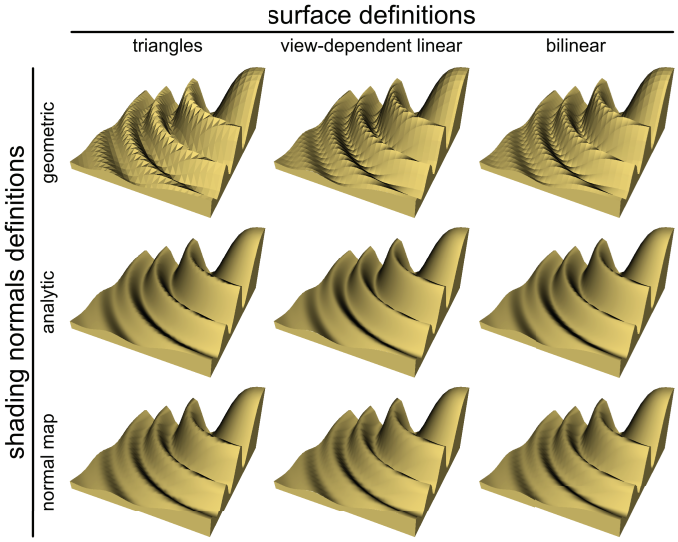


Fig. 25: Possible reconstructions for a 25^2 height field.

7 Additional Accelerations

Surface definition: The previous sections presented a new rendering algorithm and an efficient precomputation. One decision we made in the beginning concerned the surface definition and we will discuss this aspect here.

The choice of how to interpret the height-field samples was based on two interesting properties: the function remains concave in an inter-texel, allowing us to restrict test positions on centerlines. At centerlines, we can further rely on hardware-supported bilinear interpolation.

One could object that the "natural" (smoother) surface definition is an actual bilinear interpolation, which is often (more or less accurately) used in previous techniques. In fact, this definition gives a G^1 quadric surface inside each inter-texel (whereas two triangles only have G^0 continuity), but only G^0 on centerlines, which prevents its use to represent smooth surfaces with a few texels. Higher-order interpolation (*e.g.* bicubic interpolation) would be needed, implying a costly ray-intersection test [36]. As shown in Fig. 25, bilinearly interpolated height fields require high resolution to produce sharp smooth ridges for which the triangular mesh definition can be similarly faithful.

It is also important to note that the smooth appearance is comes actually from shading based on the normals. Here again, bilinear interpolation yields discontinuous surface normals (hence, discontinuous shading) at centerlines. For a smooth surface, normals need to be defined separately (*e.g.* in a precomputed normal map, whose resolution can differ from the height map), but then, only silhouettes matter. These are usually similar because differences appear only if a ray intersects the

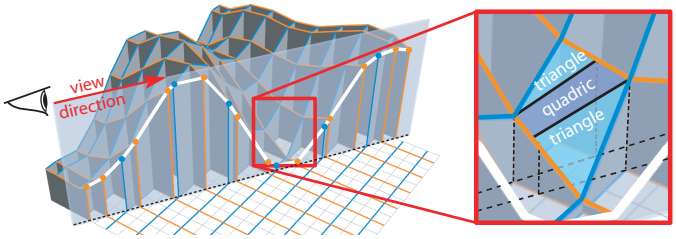


Fig. 26: View-dependent linear approximation of the height map along the viewing direction: in each texel, this corresponds to a continuous view-dependent surface, made of two triangles and one quadric patch.

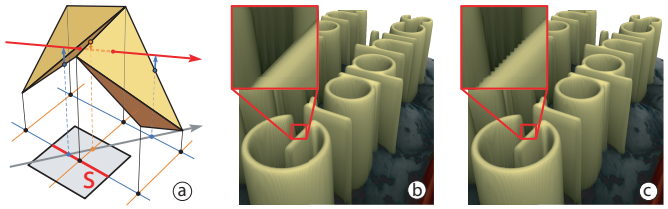


Fig. 27: Handling of null safety distances: (a) a zero safety distance is affected to the shown texel, requiring at most two simple iterations to escape from it without missing intersections; rendering (b) without and (c) with clamping of safety distances to value one (notice the small artifacts on sharp edges in the latter case).

quadric without being below the surface when traversing the centerlines.

Here, we propose an alternative surface interpretation which exploits these findings and delivers a view-dependent definition that simplifies the intersection test.

The new surface is defined as follows: on centerlines, the height still matches bilinear interpolation. But between successive centerlines we assume a linear variation along the view ray. Hence, the surface itself is view-dependent. As shown in Fig. 26, it implies that each inter-texel is decomposed into three continuously connected surface pieces: two triangles and one quadric patch. The defined surface is continuous both spatially and with respect to the viewpoint position (or the light-source position for shadow rays). This surface has the same silhouettes as our triangle definition and shares its properties needed for our precomputations. The advantage is that the final intersection test with two triangles can be replaced by a simple linear-segment intersection, which is significantly faster (Sec. 8).

Although view-dependence might seem like a potential source of visual artifacts, only in rare pathological cases it becomes visible. Typically, such cases lead to even stronger artifacts with competing methods. The accompanying video illustrates these points. The gain of this approximation is roughly 6%.

Strictly positive safety distance: If small artifacts are acceptable, an acceleration maintaining high visual quality is possible. The most costly element of our algorithm is the treatment of zero safety distances because they result in local intersection tests that imply branching in the marching loop, impairing parallelism. Forbidding zero values (*i.e.* clamping safety distances to one) avoids this behavior while the quality loss is minimal. Only silhouettes at sharp height-field discontinuities (Fig. 27) are slightly affected. The performance gain makes it a useful choice in practice.

8 Results and Discussion

In this section, we analyze the performance and quality of our approach. We explained how a distinct safety distance (SD) or hybrid cylinder-cone (CC) shape is computed for each of the four ray classes. A conservative approximation consists in keeping only the minimum of

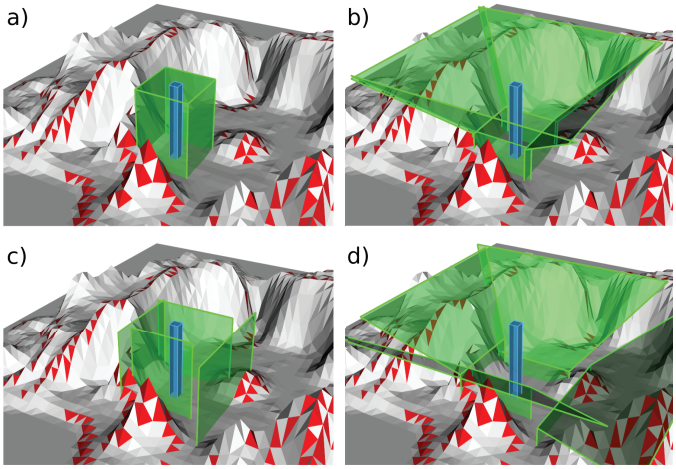


Fig. 28: The four variants of safety shapes: (a) isotropic safety distance; (b) isotropic cylinder-cone; (c) directional SD; (d) directional CC.

the four directional shapes (Fig. 28). The required storage is divided by four (*i.e.* one value for SD, three values for CC), but using four directional shapes instead of a single isotropic one increases performance by 20%, hence, this option provides an interesting trade-off between storage and rendering speed. Applying the just-mentioned surface definition results in an average speedup of 6%. For a fair comparison, the following results use only isotropic shapes and the triangle-surface interpretation.

Fig. 29 summarizes the performance of several variants of our algorithm against competing methods, measured on a 512^2 height field (Fig. 30) covering all pixels of a 1280×1024 screen using a GeForce GTX 285. Approaches without precomputation are either very error-prone (linear search (LS) [21], or much slower (minmax mipmaps (MM) [23]) and can result in less performance than our accurate naive algorithm (NA) (Sec. 5.2.1). In particular, minmax mipmaps proved less efficient even for textures of a resolution beyond 4096^2.

Higher performance is obtained with methods using precomputation. The only competing method (relaxed cones (RC) [30]) requires tuning the fixed number of marching steps to a small value to achieve performance similar to ours, thereby increasing rendering artifacts,

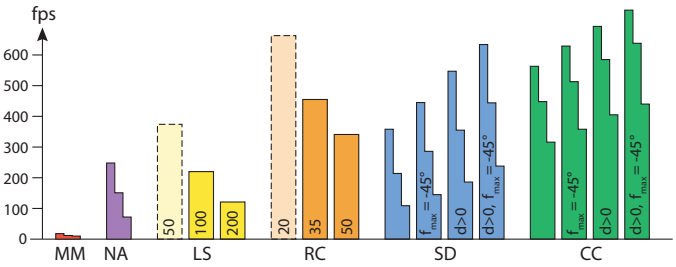


Fig. 29: Timings: MM [23], NA (our naive solution), LS [21], RC [30], SD (our safety distance), CC (our cylinder-cone). For LS and RC, numbers denote marching iterations. For SD and CC, ϕ_{max} is taken as 90° if not specified (*i.e.* no viewing angle restriction); $d > 0$ corresponds to the strictly positive distance assumption (Sec. 7). Dashed boxes denote situations where rendering artifacts are too strong to allow usefulness. For each bar, the three stair levels correspond to height-field resolutions 256^2, 512^2 and 1024^2.

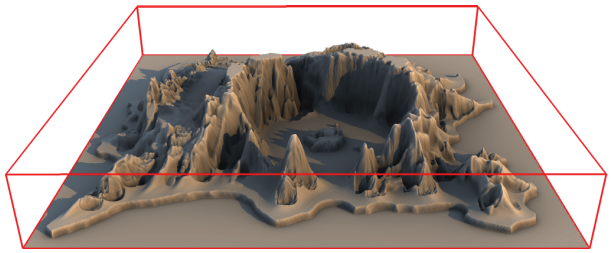


Fig. 30: The 512^2 height field used for performance comparison, rendered here with shadows, using our accurate technique.

while our technique remains accurate in all situations (only the additional no-zero-radius strategy approximates some silhouettes). The accompanying video shows that even with 35 iterations the RC method has too many artifacts for close-ups which makes it problematic to use for high-resolution height fields (see Sec. 3).

Table 1 shows computation times and rendering performance obtained for several variants of our precomputation. Exact computations involving visibility run on the CPU while conservative approximations are implementable on the GPU, leading to significantly smaller computation times. We first notice that addition of visibility has a higher influence on safety cylinders than on the hybrid cone-cylinder shape. For both shapes, an additional angular viewing constraint of $\phi_{max} = -45°$ improves rendering efficiency significantly, in which case the influence of visibility becomes negligible. It must be noted that even if not theoretically guaranteed, even below ϕ_{max}, artifacts remain very limited and less noticeable than for most competitors.

The most costly precomputation involves exact visibility. As it gave results in reasonable time we did not try to optimize it further, though it could probably be much faster using appropriate geometrical structures (notably for the blocker search). However, if precomputation time

TABLE 1: Precomputation times and rendering performance for various cases: cylinder with all rays (cyl_{all}, CPU) or exterior rays only (cyl_{ext}, GPU), cone computed with all rays, *i.e.* with our N-Buffers solution (cone_{all}, GPU), or exterior rays only (cone_{ext}, CPU). Framerates are given without viewing restriction (left) and with a max. viewing angle of $\phi_{max} = -45°$ (right). Blue lines indicate good trade-offs between rendering speed and precomputation time.

safety shape	computation (s)			rendering (Hz)		
	256^2	512^2	1024^2	256^2	512^2	1024^2
cyl_{all}	1.8	7.1	15.2	304 / 445	190 / 284	85 / 145
cyl_{ext}	14	109	1197	358 / 450	214 / 290	109 / 147
cyl_{all}+cone_{all}	1.9	7.3	16.1	548 / 608	444 / 492	313 / 341
cyl_{ext}+cone_{all}	14.1	109	1198	563 / 612	450 / 497	316 / 348
cyl_{ext}+cone_{ext}	143	1809	24111	573 / 629	461 / 514	326 / 358

is an issue, our conservative approximation using n-buffers, coupled with a safety distance ignoring visibility (but still possibly assuming a ϕ_{max} bound) can be fully implemented on GPU, and computes results which are close to reference, in less than one second. It must be also noted that our exact visibility algorithm is already faster than a sampling solution [19] for ordinary height fields (and height fields can be designed for which sampling always fails, whatever the density used). As an indicator: a sampling with 8 positions on s_τ and 256 angular directions for a 512^2 texture, leads to the same rendering performance, but results in artifacts. The sampling-based precomputation was roughly 10 times slower than our solution, even for a fairly optimized implementation.

One final interesting comparison is the use of a standard mesh to render a height field. At a height-map resolution of 1024^2 and full screen (1280×1024) ray casting, our CC approach reaches 395 fps, compared to 170 fps for a standard rendering (*i.e.* using vertex buffer objects, with 2 triangles per texel of the height map).

Note that the bisection could be replaced by any dichotomous search (*e.g. regula falsi* [22], [37]), but the intersection-test intervals produced by our marching phase are small, making performance gains difficult.

9 Conclusion

This paper presented a novel height-field-rendering solution. We achieve high performance, while remaining accurate. The approach gives further insight into height-field rendering. We presented a method to efficiently integrate visibility into the preprocess. The precomputation is comparably fast and in many situations we outperform previous suggestions by several orders of magnitude. This makes the approach practical and height-field surfaces a very attractive rendering primitive.

Acknowledgments This work has been partially funded by the French National Research Agency (iSpace&Time) and the Intel Visual Computing Institute (IVCI) at Saarland University.

REFERENCES

[1] C. Dick, J. Krüger, and R. Westermann, "GPU ray-casting for scalable terrain rendering," in *Proceedings of Eurographics 2009 - Areas Papers*, 2009, pp. 43–50.

[2] S. Mantler, S. Jeschke, and M. Wimmer, "Displacement mapped billboard clouds," Institute of Comp. Graphics and Algorithms, Vienna Univ. of Technology, Tech. Rep. TR-186-2-07-01, Jan. 2007.

[3] L. Baboud and X. Décoret, "Realistic water volumes in real-time," in *Eurographics Workshop on Natural Phenomena*. Eurographics, 2006.

[4] M. M. Oliveira, G. Bishop, and D. McAllister, "Relief texture mapping," in *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000, pp. 359–368.

[5] J. Shade, S. Gortler, L. wei He, and R. Szeliski, "Layered depth images," in *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 1998.

[6] S. Parilov and W. Stürzlinger, "Layered relief textures," in *WSCG*, 2002, pp. 357–364.

[7] F. Policarpo and M. M. Oliveira, "Relief mapping of non-height-field surface details," in *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 55–62.

[8] R. de Toledo, B. Wang, and B. Levy, "Geometry textures," in *SIBGRAPI '07: Proceedings of the XX Brazilian Symposium on Computer Graphics and Image Processing*, 2007, pp. 79–86.

[9] C. Andújar, J. Boo, P. Brunet, M. Fairén, I. Navazo, P. Vázquez, and A. Vinacua, "Omni-directional relief impostors," *Computer Graphics Forum*, vol. 26, no. 3, pp. 553–560, Sep. 2007.

[10] L. Szirmay-Kalos and T. Umenhoffer, "Displacement mapping on the gpu - state of the art," *Computer Graphics Forum*, vol. 27, no. 6, pp. 1567 – 1592, 2008.

[11] R. L. Cook, "Shade trees," in *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1984, pp. 223–231.

[12] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner, "Real-time continuous level of detail rendering of height fields," in *Proc. of SIGGRAPH*, 1996.

[13] K. Moule and M. D. McCool, "Efficient bounded adaptive tessellation of displacement maps," in *Graphics Interface*, 2002, pp. 171–180.

[14] T. Boubekeur and C. Schlick, "Generic mesh refinement on gpu," in *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*. ACM, 2005, pp. 99–104.

[15] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi, "Detailed shape representation with parallax mapping," in *Proc. of the ICAT 2001*, 2001, pp. 205–208.

[16] T. Welsh, "Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces," Infiscape Corporation, Tech. Rep., 2004.

[17] J. Kautz and H.-P. Seidel, "Hardware accelerated displacement mapping for image based rendering," in *GRIN'01: Proceedings of Graphics interface 2001*, 2001, pp. 61–70.

[18] J. Hirche, A. Ehlert, S. Guthe, and M. Doggett, "Hardware accelerated per-pixel displacement mapping," in *GI '04: Proceedings of Graphics Interface 2004*, 2004, pp. 153–158.

[19] L. Baboud and X. Décoret, "Rendering geometry with relief textures," in *Graphics Interface '06*, 2006.

[20] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing," in *Eurographics '87*, 1987, pp. 3–10.

[21] F. Policarpo, M. M. Oliveira, and J. L. D. Comba, "Real-time relief mapping on arbitrary polygonal surfaces," in *Symposium on Interactive 3D graphics and games*, 2005, pp. 155–162.

[22] L. Szirmay-Kalos, B. Aszódi, I. Lazányi, and M. Premecz, "Approximate ray-tracing on the gpu with distance impostors," *Computer Graphics Forum (Proc. of Eurographics 2005)*, vol. 24, no. 3, pp. 695–704, 2005.

[23] A. Tevs, I. Ihrke, and H.-P. Seidel, "Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering," in *Symposium on Interactive 3D Graphics and Games (i3D'08)*, 2008, pp. 183–190.

[24] L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum, "View-dependent displacement mapping," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 334–339, 2003.

[25] X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum, "Generalized displacement maps," in *Eurographics Symposium on Rendering*, A. Keller and H. W. Jensen, Eds., 2004, pp. 227–234.

[26] E. H. Adelson and J. R. Bergen, "The plenoptic function and the elements of early vision," in *Computational Models of Visual Processing*, 1991, pp. 3–20.

[27] L. McMillan and G. Bishop, "Plenoptic modeling: an image-based rendering system," in *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, 1995, pp. 39–46.

[28] W. Donnelly, *GPU Gems 2*. Addison-Wesley, 2005, ch. Per-Pixel Displacement Mapping with Distance Functions, pp. 123–136.

[29] D. W. Paglieroni, "The directional parameter plane transform of a height field," *ACM Trans. Graph.*, vol. 17, no. 1, pp. 50–70, 1998.

[30] F. Policarpo and M. M. Oliveira, *GPU Gems 3*, 2007, ch. 18: Relaxed Cone Stepping for Relief Mapping, pp. 409–428.

[31] S. Nirenstein, E. H. Blake, and J. E. Gain, "Exact from-region visibility culling." in *Rendering Techniques*, 2002, pp. 191 – 202.

[32] M. Levoy and P. Hanrahan, "Light field rendering," in *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 31–42.

[33] X. Gu, S. J. Gortler, and M. F. Cohen, "Polyhedral geometry and the two-plane parameterization," in *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, 1997, pp. 1–12.

[34] J. Bittner, J. Prikryl, and P. Slavík, "Exact regional visibility using line space partitioning," *Computers & Graphics*, vol. 27, no. 4, pp. 569–580, 2003.

[35] X. Décoret, "N-buffers for efficient depth map query," *Computer Graphics Forum (Proc. of Eurographics 2005)*, vol. 24, no. 3, 2005.

[36] C. Loop and J. Blinn, "Real-time gpu rendering of piecewise algebraic surfaces," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 664–670, 2006.

[37] E. Risser, M. Shah, and S. Pattanaik, "Faster relief mapping using the secant method," *Journal of Graphics Tools*, vol. 12, no. 3, pp. 17–24, 2007.

Lionel Baboud is a postdoctoral researcher at the Max-Planck Institut (MPI) Informatik and the MMCI Cluster of Excellence (Saarland University). He studied Mathematics and Computer Science at ENSIMAG (École Nationale Supérieure d'Informatique et Mathématiques Appliquées de Grenoble). In 2005, he obtained a Master in Computer Graphics and Computer Vision from Grenoble Universities. He did his doctoral research at INRIA (Institut National de Recherche en Informatique et Automatique, Grenoble) in the field of real-time rendering and obtained his PhD from Grenoble Universities, in 2009. He takes an interest in real-time rendering, alternative representations and GPU algorithms.

Elmar Eisemann is an associate professor at Telecom ParisTech. Before, he was a senior scientist heading a research group in the Cluster of Excellence (Saarland University / MPI Informatik) until Dec. 2009. He studied Mathematics (Cologne) and Computer Science (Ecole Normale Superieure Paris). He obtained Master (2004) and PhD. (2008) in Mathematics / Computer Science from Grenoble Universities. He worked at MIT (2003), UIUC (2006), Adobe (Seattle - 2007, Boston - 2008). His interests include real-time rendering, shadow algorithms, global illumination, and GPU acceleration techniques. He coauthored the book *Real-time Shadows* and was local organizer of EGSR 2010. In 2011, he received the Eurographics Young Researcher Award .

Hans-Peter Seidel is the scientific director and chair of the computer graphics group at the Max-Planck-Institut (MPI) Informatik and a professor of computer science at Saarland University. He has published and lectured widely. He has received grants from a wide range of organizations, including the German National Science Foundation (DFG), the German Federal Government (BMBF), the European Community (EU), NATO, and the German-Israel Foundation (GIF). In 2003 Seidel was awarded the 'Leibniz Preis', the most prestigious German research award, from the German Research Foundation (DFG). Seidel is the first computer graphics researcher to receive this award.