Volume 31 (2012), Number 8 pp. 2348–2361 COMPUTER GRAPHICS *forum*

Low-Complexity Intervisibility in Height Fields

Ville Timonen†

Åbo Akademi University

Abstract

Global illumination systems require intervisibility information between pairs of points in a scene. This visibility problem is computationally complex, and current interactive implementations for dynamic scenes are limited to crude approximations or small amounts of geometry. We present a novel algorithm to determine intervisibility from all points of dynamic height fields as visibility horizons in discrete azimuthal directions. The algorithm determines accurate visibility along each azimuthal direction in time linear in the number of output visibility horizons. This is achieved by using a novel visibility structure we call the convex hull tree. The key feature of our algorithm is its ability to incrementally update the convex hull tree such that at each receiver point only the visible parts of the height field are traversed. This results in low time complexity; compared to previous work, we achieve two orders of magnitude reduction in the number of algorithm iterations and a speedup of 2.4 to 41 on 1024^2 height fields, depending on geometric content.

Keywords: height field, intervisibility, global illumination

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Computer Graphics—Color, shading, shadowing, and texture

1. Introduction

The two major applications for visibility algorithms in computer graphics rendering are geometry culling and lighting. Firstly, culling is used to accelerate rendering by determining which geometry should not be sent down the rendering pipeline, i.e., which geometry is invisible from the viewer. Secondly, lighting algorithms need to know the visibility of light sources from each illuminated receiver point. As generally all scene geometry acts as light sources, global illumination algorithms need to determine the visibility of the whole scene from each receiver point. This problem is similar to that in geometry culling, except orders of magnitude more complex, as visibility from millions of receiver points needs to be determined. The problem is largely solved for static geometry as most of the computation can be performed as a pre-pass, but dynamic geometry remains an open problem.

In this paper we present a method to determine intervisibility of height field geometry. Height maps describe geometry by defining the elevation of a plane as a function of $N \times N$ surface coordinates. They can be used as standalone objects or to describe meso- and micro-structure on the surfaces of a larger-scale object. Another recently popular application for height field algorithms is producing lighting effects in screen space, where the depth buffer of a rendered scene is treated as a height field with effects applied in post-processing. For non-graphics related applications of height field visibility algorithms, see the survey [Nag94].

Current interactive height field intervisibility methods determine approximate or local visibility in order to produce effects such as soft global illumination [NS09] or color bleeding [RGS09]. Current methods are limited to local, approximate, or noisy effects mainly due to poor scaling of visibility calculations. The approach that current methods use involves sampling, for each receiver independently, the surrounding height field where in order to test n sender points for one receiver point, n iterations are performed. Intervisibility searches based on this approach are variations of what we in this paper will call the *naïve method*: for each receiver point K azimuthal directions are chosen and, for each direction, the height field is traversed outwards from the receiver one unit length step at a time.

† e-mail: vtimonen@abo.fi

Table 1: *The average number of visible points per direction for the 1024^2 height fields in Figure 7.*

Height field	Visible points	Visible/total
fractal terrain	27.1	5.6 %
brick surface	8.94	1.8 %
sine grid	22.8	4.7 %
blocks	5.56	1.1 %

The main problem in the naïve method is that it scales linearly with respect to the search distance, while visibility in all practical height fields should scale *sub-linearly*. In Table 1 we counted the average number of visible points for a single azimuthal direction for different types of height fields ($N = 1024$) and compared it to the number of evenly spaced points that were tested for visibility (roughly $N/2$). In Section 6 we will measure the scaling to be between $O(N^{0.01})$ and $O(N^{0.65})$.

In this paper we present a new way of calculating visibility in height fields. The key feature of our method is its ability to traverse only the visible geometry by effectively culling the non-visible geometry. Another advantage of our method is that it produces a more compact description of the visibility than a simple enumeration of the visible points: for each receiver point we determine a list of local visibility horizons where two consecutive horizons always enclose all adjacent visible height field points. Our algorithm runs in time linear in the number of output visibility horizons and is dependent on the height field content. Compared to previous algorithms, we achieve two orders of magnitude reduction in the number of iterations required to extract accurate intervisibility on 1024^2 height maps bringing the complexity to manageable levels, and a speed up of 2.4 to 41 compared to the naïve method, depending on the height field content.

It can be argued that in rendering there is a trend towards performing an increasing portion of shading and lighting as a screen-space pass. It will be interesting to see what the full potential of such methods are. Current screen-space methods are not able to effortlessly scale to full illumination solutions, but rather settle for producing a limited set of effects that are computationally feasible. While there are many obstacles to overcome before global illumination with arbitrary materials and integrated light sources is feasible in screen-space, our contribution is to overcome the one with the highest computational complexity: intervisibility.

2. Previous work

Determining height field intervisibility is essentially a problem of computational geometry. Finding the visible areas of a terrain from a single viewpoint [KZ02] [FHT09], a line [CS89], or a region [BWW05] is a problem extensively researched and largely solved. However, algorithms that find terrain intervisibility at all surface points are significantly less studied, and applying single-viewpoint methods to each height field point is intractable for interactive applications.

Global illumination methods for height fields require intervisibility determination, but unlike our method they so far exclusively use a scheme where the same visibility search procedure is performed independently for each height field point. The naïve approach [CoS95] [SLYY08] is to solve the visibility in a set of azimuthal directions where each direction is traversed from the receiver point outwards in unit length steps, and each time the previous slope maximum is exceeded the new point is known to be visible from the receiver point. Our method produces results identical to the naïve method.

The naïve approach can be accelerated by applying level of detail (LOD) methods: [NS09] generate multiple levels of detail of the height field and use the lower resolution levels and sparser sampling when traversing farther from the receiver, as first introduced in [SN08]. While faster, the approximated visibility favors using the method only for soft effects.

Ambient occlusion methods with falloff terms need to know the distance of occluding geometry and therefore have to solve the intervisibility problem as well. Screen-space ambient occlusion methods [Mit07] [DBS08] approximate *local* scene visibility in image-space by treating the depth buffer as a height field. The same approach has been used to produce global illumination effects such as color bleeding [RGS09], which is extended with LOD in [SHR10]. Intervisibility in these methods is determined from sender and receiver normals only and any occluding geometry in-between is ignored. While fast and sufficient for approximate near-field effects, scaling to far-field is problematic: an occlusion search between sender and receiver is required as suggested by the authors of [RGS09], making the intervisibility search the same as used by the naïve method.

Implementations of the naïve method usually trade banding for noise by randomizing sampling patterns per receiver. In future work, in Section 10, we discuss ways to apply LOD and to trade banding for noise with our method as well.

More exotic ways to sample the height field have also been introduced, such as performing a very sparse randomized occlusion search per pixel and gathering the final occlusion values from a small neighborhood around a receiver [HBR*11]. Alternatively, instead of taking simple height samples, line and area samples can be taken to approximate the overall occlusion of the near-field geometry [LS10]. While these methods produce fast results, they don't scale well to occlusion effects of arbitrary length and are non-trivial to extend to indirect illumination.

Global illumination methods for *generic* geometries are diverse [DBBS06] and also need to address the intervisibility problem. Excluding various approximations, these methods traverse all scene geometry for each receiver primitive.

The problem then becomes making sure that only the effect of the frontmost layer around the receiver is accounted for. This has been tackled in [Bun05] and [DSDD07] by running several iterations of the algorithm where each iteration removes extraneous contribution of the overlapping layers. Another solution to overcome this problem is to use shadow maps (see the survey [HLHS03]) to determine the receivers from the point of view of point light sources. Fast approximate shadow maps [RGK*08] [ADM*08] can be used to make this approach feasible when many light sources need to be considered. None of these approaches make use of the characteristics of height field geometry, and their visibility solutions become prohibitively expensive for anything but very small height fields when accurate results are preferred.

Another common approach to solving visibility of scene geometry is ray tracing, where visibility is queried by shooting independent rays from a receiver and determining the closest geometry the rays intersect. Let K be the number of azimuthal directions in which rays from each point are cast, and let p_a be the average number of visible height field points from one receiver point in one azimuthal direction. Then on an $N \times N$ height field at least $N^2 K p_a$ optimally chosen rays have to be traced to determine intervisibility at the same accuracy as our method. Not including the complexity of tracing one ray, this already is higher than the time complexity of our method, $O(N^2 K m_a)$, where $m_a \leq p_a$ denotes the average number of visibility horizons.

We use a compact way to unambiguously describe intervisibility on a line by local horizons as introduced in [DFM94]. In this model, a local horizon from a viewpoint is defined at each transition from visibility to invisibility. From local horizons it is possible to produce the *casting set* as used in [NS09], which is the set of points visible from the receiver point.

Incidentally, it was shown in [TW10] that a visibility horizon is defined by the points of a convex hull, and that a line sweep algorithm can incrementally determine global visibility horizons for n points in $O(n)$ time by using a convex hull stack. In order to determine intervisibility, we extend the ideas of [TW10]: we track *a set of convex hulls* instead of only one and introduce a novel tree structure to hold them. Through efficient tree update operations, we maintain the same linear complexity: n local horizons are extracted in $O(n)$ time.

3. Height field processing

In this section we describe the highest level framework of our method with a focus of the process on the scale of the whole height field. In Section 4 we describe the process of solving visibility on the scale of a single line. Section 5 defines in detail the algorithm that is executed for each point on the line. Section 6 establishes the complexity of the algorithm and analyzes its scaling with respect to the height field size and content. An implementation on the GPU and optimizations on the code are covered in Section 7 and the implementation efficiency with respect to available hardware resources is discussed in Section 8. Section 9 showcases the actual performance and Section 10 discusses the accuracy of our method and ways to utilize the visibility information.

The input to our algorithm is a height map consisting of a regular grid of $N \times N$ height values. For each height field point our algorithm determines visibility in K discrete azimuthal directions by performing K sweeps through the height field. For each direction $\phi_k = \frac{k}{K} 2\pi, 0 \leq k < K$, the height field is swept through in parallel lines that are unit length apart thus calculating the given azimuthal direction for all height field points in one sweep.

These lines are stepped through one unit length step at a time, and visibility backwards along the line is determined for each step in turn, as demonstrated in Figure 1. At each step lighting contribution is gathered from the visible segments of the line and the result is written into the sweep's output buffer. The output buffers are axis-aligned such that the processed lines map to vertical lines in each output buffer, as demonstrated in Figure 2. As the maximum number of lines as well as the maximum length of a line in an arbitrary direction can be at most $\sqrt{2}N$, the output buffers are of size $\sqrt{2}N \times \sqrt{2}N$.

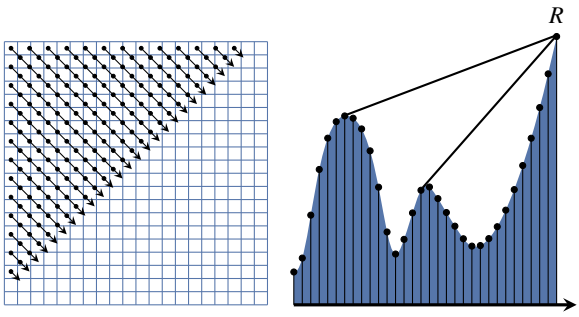

Figure 1: *For one azimuthal direction, the height field is processed in parallel lines (left). For each line, the visibility horizons are extracted backwards along the line for the most recent step, receptor R (right).*

After the sweeps have been performed, each of the K output buffers contain N^2 result values. Finally, results across the output buffers are accumulated into a single result buffer the size of the input height field, $N \times N$, shown in Figure 2. At this point visibility of an average of roughly $K\frac{N}{2}$ height field samples have been considered for each of the N^2 sampled receptor points. Unlike previous methods, we require substantially fewer than $K\frac{N^3}{2}$ iterations to achieve this, as shown later in this paper.

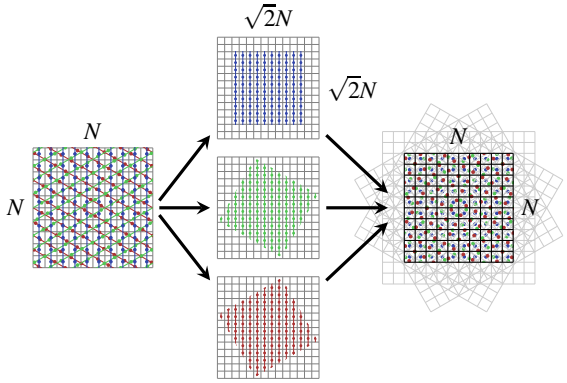

Figure 2: *Three sweeps ($K = 3$), denoted by different colors, are performed over the input height field (left), their results are written to axis-aligned output buffers (center), and finally accumulated in the result buffer (right).*

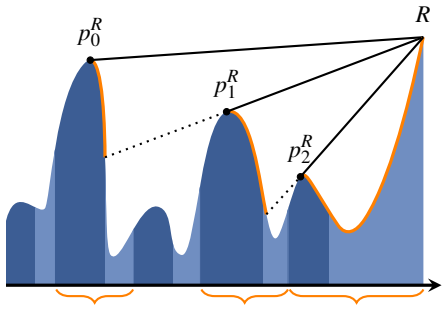

Figure 3: *The local horizons formed between points p_i^R and R unambiguously describe visible segments (denoted at the bottom) and the extent of their visibility (the surface in orange) at the receptor R. Convex sections are dark and concave light.*

4. Line processing

Visible points of the height field along a line are naturally grouped into continuous parts. Such a part can only start at a point in the *convex* section of the line: when viewed from the receptor R, the point is a local maximum with neighbors that are below it. Convex sections are separated from each other by *concave* sections, and therefore the line can be split into strictly alternating convex-concave sections. We call a pair of convex-concave sections a *segment*, and use these segments to determine visibility along a line.

From now on we refer to a height field sample at step i as point p_i, denoting its height by h_i and its distance from the start of the line by d_i. In this notation p_0 is the first point on the line and p_n is the receptor R. When traversing a line, each point p_i is determined to belong to either a convex or a concave section of the line by the following function:

$$C(p_i) = \begin{cases} \textit{convex} & \text{if } i = 0 \text{ or } 2h_i > h_{i-1} + h_{i+1}, \\ \textit{concave} & \text{if } i \neq 0 \text{ and } 2h_i < h_{i-1} + h_{i+1}, \\ C(p_{i-1}) & \text{otherwise} \end{cases} \quad (1)$$

Note that in case the three points p_{i-1}, p_i, and p_{i+1} forming a straight line the convexity status is inherited from the previous point.

The *visible* segments can be found using local horizons as demonstrated in Figure 3. The local horizons are lines-of-sight formed between R and points p_i^R on the surface such that each line-of-sight is locally tangent to the surface at p_i^R and does not intersect the surface between R and p_i^R. The line-of-sight between p_0 and R is also a horizon if it does not intersect the surface. There are as many local horizons as there are visible segments along the line, and each horizon lies on the convex part of a visible segment. The beginning of the visibility is available as the endpoint of the horizon, p_i^R. The end of the segment's visibility generally lies between two surface points and its coordinate is not directly available from the horizons. Instead, the next horizon (through p_{i+1}^R and R) intersects the visible segment exactly at the end of the visibility. The last visible segment includes R and does not have a following horizon, in which case the visibility reaches all the way to p_{n-1}. The least amount of information required to describe the complete line visibility from R is an array of the unsigned integer distance values d_i of p_i^R.

In order to iteratively derive the local horizons for each R (p_n) along the line without having to go over points $p_0 \cdots p_{n-1}$ each time, we track a convex hull for each segment, starting from the beginning of the segment (p_j) and ending in p_n as demonstrated in Figure 4. The convex hulls

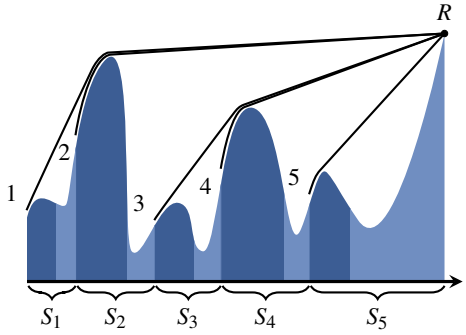

Figure 4: *A set of convex hulls ($1 \ldots 5$) are formed from the convex sections of segments $S_1 \ldots S_5$ to the receptor R (p_n) at the far right.*

therefore are the groups of points ordered by their distance:

$$\left\{ p_{S_j}, \{j, \{i_c\}, n\} \in S_j \text{ and } j < i_c < n \text{ and } \frac{h_{S_j[i]} - h_{S_j[i-1]}}{d_{S_j[i]} - d_{S_j[i-1]}} > \frac{h_{S_j[i+1]} - h_{S_j[i]}}{d_{S_j[i+1]} - d_{S_j[i]}} \right\}, \quad (2)$$
$$j = 0 \text{ or } \Big(C(p_j) = \textit{convex} \text{ and } C(p_{j-1}) = \textit{concave} \text{ and } 0 < j < n \Big)$$

In other words, we defined the set of upper convex hulls that start from the first point in each convex section and end in R. The first convex section always starts from p_0.

Convex hulls are efficient in determining occlusion between a receiver that is included in the hull and geometry behind the hull: points on the hull can have a direct line-of-sight only to their neighbors and thus the highest occlusion is cast by the point previous to the receiver in the hull. The last point of each convex hull defined in Equation 2 is the receptor R, and the points second to last are p_i^R. Therefore the edges between the last two points of the hulls are the local horizons for R. Duplicate horizons, however, emerge from this definition: the convex hull of each *invisible* segment produces the same horizon as the convex hull of one visible segment. This is due to the possibility of convex hulls sharing points close to R. In fact, the vast majority of the segments are usually invisible.

To overcome this problem we maintain a *convex hull tree* instead of maintaining a separate convex hull for each segment. Points included in the convex hull tree are the union of points in the separate convex hulls as defined in Equation 2. Each separate convex hull is still existent as part of the convex hull tree and we call such a part a *convex hull path*. A path will start from p_j (a *leaf node* of the tree) and end up in R (the *root node*). The branches are ordered such that the first child of a parent is the one farthest away from the parent or—equivalently—having the highest height of the children. There are no redundant points in the tree and the direct children of the root node are exactly the points p_i^R. Therefore, when the convex hull tree is up-to-date, extracting the local visibility horizons involves nothing more than going over the children of the root node. Figure 5 demonstrates a convex hull tree in a fractal terrain height field along one line.

When a new step along the line is taken, a new R is introduced and becomes the new root of the tree. The core algorithm for determining visibility using the convex hull tree therefore adjusts the tree after the introduction of a new root such that each convex hull path is valid (convex). This algorithm is recursive in nature and described next.

5. Point processing in the convex hull tree

In this section we describe how a new point is added to the present convex hull tree. The input of this phase is the next height field point p_n along the line being traversed. The point is first added as the new root of the convex hull tree, making the previous point p_{n-1} (the old root) its only child. The tree is then processed using a recursive algorithm until all paths from the root to the leaves are convex. As the algorithm is applied incrementally it can be assumed that the paths were valid before the addition of the new root. Therefore, it is enough to process paths only to the point where convexity once again holds.

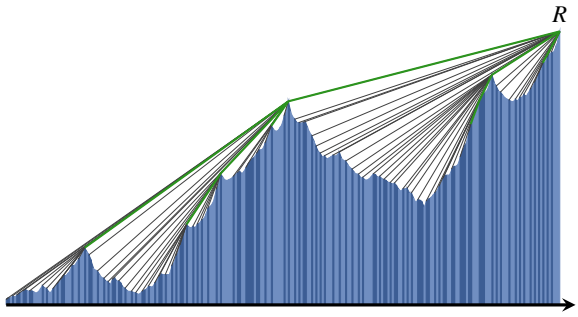

Figure 5: *A convex hull tree along one line during a sweep on a fractal terrain. Green links are shared by multiple convex hull paths.*

The algorithm is first invoked using the triplet (root's first child's first child $\rightarrow$ root's first child $\rightarrow$ root) as its parameter. We are naming the elements of such a triplet ($child_T \rightarrow parent_T \rightarrow root$). The last element of the triplet will always be the root element of the tree, and $child_T$ the first child of $parent_T$. First, the algorithm checks whether the vertices of the triplet are convex. If they are, no action is needed and the call returns. If the convexity check fails, then $parent_T$ needs to be removed from this path, causing $child_T$ to be connected directly to *root*. The edge from *root* to $child_T$ will be above the edge from *root* to $parent_T$, and therefore the correct position for $child_T$, as a *root*'s child, is before $parent_T$. If $child_T$ was also the last child of $parent_T$, $parent_T$ gets orphaned and is removed. Otherwise the second child of $parent_T$ takes the place of the first child. After these changes it is necessary to proceed both one step deeper and one step wider from $child_T$ in the tree. Figure 6 illustrates the described process.

When proceeding deeper, the previous $child_T$ becomes the new $parent_T$ and the first child of $child_T$ becomes the new $child_T$. When stepping wider, $parent_T$ stays the same and its (newly assigned) first child becomes the new $child_T$. The process continues recursively until the convexity checks for each branch pass and the algorithm stops, at which point all convex hull paths are valid. Algorithm 1 lists the pseudocode for the recursive function. After the convex hull tree is valid again, the visibility information is available as *root*'s direct children (as a linked list) as described in Section 4.

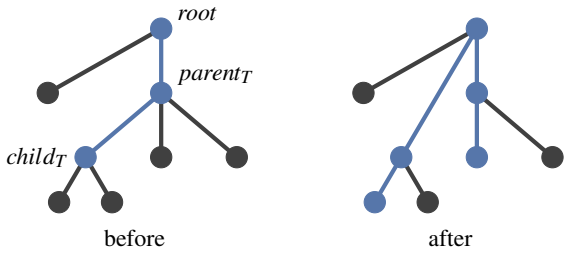

Figure 6: *Triplet (child$_T$ → parent$_T$ → root) fails the convexity check (left), causing child$_T$ to disconnect from parent$_T$ and connect to root$_T$ (right). Two new triplets are then processed, shown in blue.*

Algorithm 1 RecConvexity(child$_T$, parent$_T$, root)

```
if !convex(child_T → parent_T → root)
    connect child_T to root before parent_T
    if child_T has a next sibling
        first child of parent_T ← next sibling of child_T
        // Step wider
        RecConvexity(next sibling of child_T, parent_T, root)
    else
        delete parent_T

    if child_T has a first child
        // Step deeper
        RecConvexity(first child of child_T, child_T, root)
```

Finally, after the convexity has been established, we determine whether the previous step started a new segment. When the beginning of a segment is detected, the segment's first element is duplicated in the tree and set to be the last child of the root. The duplicated elements form the leaf nodes of the convex hull tree and are permanent throughout the line.

6. Complexity

In this section we observe the complexity of our convex hull tree processing algorithm on a line of n steps. When sweeping through an $N \times N$ height field, $1 \leq n \leq \sqrt{2}N$. Let m_i denote the number of visible horizons and t_i the number of iterations of Algorithm 1 at step $i, 0 \leq i < n$ on the line. Then the total number of horizons on a line is given by $m = \sum_{j=0}^{n-1} m_j$ and the total number of iterations by $t = \sum_{j=0}^{n-1} t_j$. We first prove that our algorithm's complexity is linear in the total number of produced output horizons ($t = O(m)$), and then analyze the complexity of the horizons.

The total number of horizons m on all points of a line of length n is at least n. We distinguish between three types of iterations of the algorithm and show that these are either $O(n)$ or $O(m)$:

1. An iteration that fails the convexity check with a child that does not have a next sibling results in the deletion of the parent node. As there are at most n elements in the tree, there can be at most n iterations of this type.
2. An iteration that passes the convexity check will return without further invocations of the algorithm. Each iteration of this type corresponds to exactly one visible horizon formed by the parent node (directly connected to the root). Also, no other iteration can produce the same horizon because such an iteration would need to have the same parent, and all such iterations would need to emanate from this iteration. Therefore the number of iterations of this type will be exactly m.
3. The last iteration type is the one for which the convexity check fails, but results in the child being detached from its previous parent and connected to the root without the parent being deleted. As the parent and its previous sibling were directly connected to the root and formed consecutive horizons before this iteration, the iteration will introduce a new horizon (formed by the child) between the two. As this inevitably increases the number of horizons, there can be at most m iterations of this type.

The total number of iterations t is therefore at most $2m+n$, or, of complexity $O(m)$.

Table 2 presents empirical results for the number of visible horizons and iterations of Algorithm 1 for height fields shown in Figure 7. The figures are measured from sweeps in 256 directions ($K = 256$) and averaged for one height field point and for one azimuthal direction. The height fields are of size 1024^2 ($N = 1024$) and the naïve method performs 484 iterations per point on average. Compared to this figure, the number of iterations required to produce the visibility information is reduced by two orders of magnitude.

Table 2: *The average number of horizons m_i and iterations t_i for one azimuthal direction at a given point (denoted by m_a and t_a, respectively).*

Height field	m_a	t_a	t_a/naïve
fractal terrain	9.25	11.7	2.4 %
brick surface	3.06	4.80	0.99 %
sine grid	1.68	3.02	0.62 %
blocks	2.55	3.76	0.78 %

As a visibility description, horizons are in all cases at least as compact as a point-to-point description: if the visibility consisted of individual scattered points there would be one horizon (that can be expressed by a single surface coordinate) for each point. In practice it is common that the visibility of each segment spans multiple consecutive points for which only one horizon is required. This can be seen by comparing Tables 1 and 2.

Determining visibility for a line of n samples using our method has the worst-case complexity equal to that of the

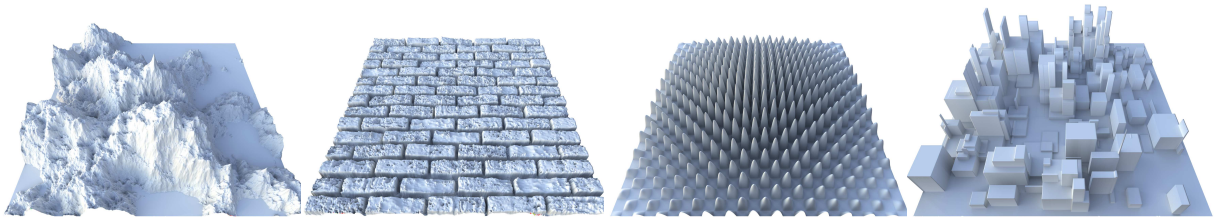

Figure 7: *The four 1024^2 height fields used as test data.*

naïve method defined in Section 1, $O(n^2)$. This happens, for instance, in a bowl-shaped height field where every other point is depressed. On the other hand, if the height field is dome-shaped, the complexity is $O(n)$. We measure practical complexity by presenting iteration counts for various height field examples and by measuring scaling in n.

In order to measure scaling, we differentiate between two types of content scaling which most practical height fields exhibit a combination of. First, we measure increasing *detail* using fractal terrains: every time n is doubled, more resolution is added between the existing points using a uniform distribution with half the range. Second, we measure increasing *extent* with a constant level of detail using random data: when n increases, more height data is randomized from a fixed finite height range. Figure 8 shows the average number of iterations t for a line of length n as a function of n. We measure scaling by k in $t = O(n^k)$. For the naïve method $k = 2$, and for linear scaling $k = 1$. As the axes are logarithmic, we fit first-order polynomials to the graph (for $n \geq 512$) and attain k from their slopes. Visibility of the fractal terrains seem to exhibit a scaling of roughly $O(n^{1.65})$, whereas the random data quickly settles to near-linear scaling of $O(n^{1.01})$. The data suggests that the scaling in most practical cases is well below the quadratic scaling of the naïve method.

We noticed that the average segment length in these two height field examples is the same for all n, indicating that the scaling is due to changes in the amount of visible horizons, not due to changes in the average coverage of one horizon. This means that the number of visible points as listed in Table 1 would scale similarly, and the observed complexity applies generally to visibility and not just to our visibility description. An increase in detail could, however, also cause an increase in segment lengths. This would be the case if the sine grid shown in Figure 7 were to scale up without the grid size changing (the number of sine "domes" staying the same): every point would continue to have the same number of visible segments but the number of points belonging to them would increase. This type of increase in detail would

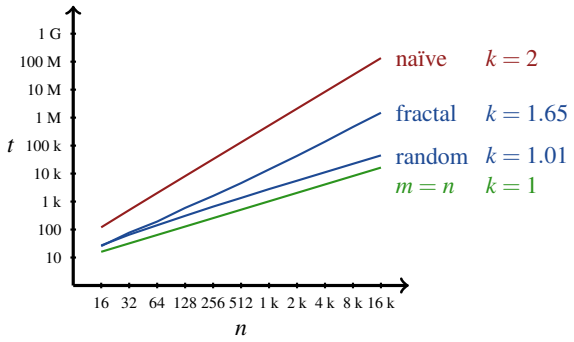

Figure 8: *The number of required iterations t for a line of n steps for fractal terrains and random data. The naïve method (in red) and linear scaling (in green) as references.*

yield linear scaling in our algorithm, and demonstrates the power of the compactness of our visibility description.

7. Implementation

As current hardware accelerated graphics libraries have a fixed rasterization stage that does not allow writing lines into a framebuffer, we have chosen to use GPGPU. We use OpenGL 4 and CUDA 3 in our implementation, and in this section use CUDA terminology. Notes on performance apply to the NVidia Fermi architecture [NVI09].

The high-level framework begins with the passing of the source height field as a texture from OpenGL to CUDA. Visibility calculations are then performed in one kernel, one thread mapping to one line in the height field. As many azimuthal directions are processed simultaneously as allowed by the amount of available graphics memory. One output buffer for each K is produced, of which $\pi/2$ rotations are linearly accumulated in CUDA reducing the number of buffers to a quarter. Once the resulting $K/4$ buffers have been passed back to OpenGL as textures, they are sampled and additively blended in a floating point frame buffer.

In the actual visibility algorithm we implement a node of a convex hull tree by allocating the following:

i one half-precision (16 bit) float for the height h_i
ii one 15+1 bit unsigned integer for the distance d_i (also index of self, i)
iii one 16 bit unsigned integer for the index of the first child (null if a leaf)
iv one 16 bit unsigned integer for the index of the next sibling

A node allocated this way fits into 8 bytes. Elements are stored in global memory using indices that correspond to element distances, with the exception of leaf nodes. Leaves are stored using indices one smaller than their distance which are known to be empty when the leaf is forked, and the 1 bit flag is set to denote the offset.

The distance based allocation yields sparse arrays, but according to our benchmarks produced better overall performance than having separate distance and index data and packing the elements densely. The optimal data layout depends on the content of the height field and the target GPU architecture due to memory coalescing and caching efficiency. We found good overall performance from a quasi-parallel allocation in which the same indices of 2 to 8 consecutive threads are sequentially laid out in memory. Using the maximum amount of 48 kB of L1 cache on each multiprocessor for global memory accesses also produced the best performance. A cacheless architecture would not benefit from the temporal coherence of memory accesses and we expect a fully parallel allocation to yield best results on such GPUs.

To implement the recursion in Algorithm 1, we use a stack stored in global memory. The quasi-parallel allocation scheme used for the convex hull trees produced the best performance for the stacks as well.

In Algorithm 1, when the convexity check fails and $child_T$ is connected to *root*, it is necessary to adjust both $parent_T$ and its previous sibling. However, as we use a single linkage scheme in which each node has links to its next sibling and to its first child only, we need to carry both the parent and its previous sibling (called $parent_{prev}$ from now on) by pushing them onto the stack. The use of single linkage also makes breadth-first traversal preferable for the following reason: current $child_T$ will be the $parent_{prev}$ of the next breadth iteration, however if depth is processed first and all children of $child_T$ are connected to the root, then $child_T$ is removed making the previous $parent_{prev}$ obsolete. Tracking the changing of $parent_{prev}$ would require another stack.

An optimized implementation of Algorithm 1 is listed as Algorithm 2. As established, each time the convexity check fails $child_T$ is connected to the root causing changes to linkage. With some state tracking (variables *regression* and *history*) these changes can be postponed and consolidated to the fail block (lines 32 – 38) by pushing extra entries onto

Algorithm 2 OptimizedConvexity(root, inSafeZone)

Functions self(n), child(n), and next(n) return memory locations of node n, node n's first child, and node n's next sibling, respectively. Function read(p) returns the node from memory location p and write(n) stores node n to memory location self(n). Operators are C style, and calls requiring memory accesses are underlined.

```
preSet = true                                                          1
regression = null                                                      2
                                                                       3
// { child_T, parent_T, parent_prev, history }                         4
s = { child(child(root)), child(root), null, 1 }                       5
                                                                       6
while stack not empty || preSet                                        7
  if !preSet                                                           8
    s = stack.pop()                                                    9
  preSet = false                                                       10
                                                                       11
  if self(s.parent_T) == self(regression)                              12
    s.parent_T = regression                                            13
                                                                       14
  if self(s.child_T) &&                                                15
      !convex(s.child_T → s.parent_T → root)
    if !self(s.parent_prev)                                            16
      child(root) = self(s.child_T)                                    17
                                                                       18
    if !inSafeZone && (child(s.child_T) ||                             19
        child(s.parent_T) == self(s.child_T))
      stack.push(child(s.child_T) ? read(child(s.child_T))             20
          : null, s.child_T, s.parent_prev, s.history >> 1)
                                                                       21
    if next(s.child_T) || next(s.parent_T)                             22
      s.parent_prev = s.child_T                                        23
      if next(s.parent_prev)                                           24
        s.child_T = read(next(s.parent_prev))                          25
      else                                                             26
        s.child_T = null                                               27
        s.parent_T = next(s.parent_T) ?                                28
            read(next(s.parent_T)) : null
      s.history = 2                                                    29
      preSet = true                                                    30
  else                                                                 31
    // & is bitwise and, ^ is bitwise xor                              32
    if self(s.child_T) && s.history & 2                                33
      child(s.parent_T) = self(child_T)                                34
      write(s.parent_T)                                                35
    if self(s.parent_prev) && s.history ^ 1                            36
      next(s.parent_prev) = self(s.parent_T)                           37
      write(s.parent_prev)                                             38
    regression = s.parent_prev                                         39
```

the stack that have a null child. This also allows more efficient hardware scheduling as threads do the expensive global writes in one place instead of blocking the execution in several places. As a result, reads were halved and writes were cut down to about a tenth as compared to an algorithm that uses single linkage and flushes changes to memory immediately. Also, as pushes onto the stack are large and stress the memory system, the latest stack element (breadth traversal) can be passed on in a register variable, cutting down stack accesses to a quarter on average.

There are two cases when running the convexity algorithm is unnecessary. The first is when the new sample forms the new root and inherits the old root as its child without further changing the tree. The second and more important case is when the new sample replaces the old root without affecting the rest of the tree. Without specially treating this case all children of the old root are traversed, and, as their convexity checks fail, their first children are also traversed, whose convexity checks all pass. This is an expensive operation, and depending on height field content, may occur frequently. The first condition for the old root being straightforwardly replaced is when all its children get connected to the new root. This can be easily tested for by checking if the convexity check for (the last child of the old root $\rightarrow$ the old root $\rightarrow$ the new sample) fails, in which case all previous siblings of $child_T$ also fail. The second condition is that there will be no other changes to the children of the old root, which is a little harder to test. Essentially it is necessary to know whether the convexity checks on the first grandchildren of the old root will pass with the new root as well.

We address these situations by maintaining a *safe zone* between two distance boundaries that enclose the current and some future steps. Edges from the first grandchildren of the root to their parents are then projected against the distance boundaries. The line between the lowest intersection points at the boundaries forms a conservative upper bound for the safe zone, demonstrated in Figure 9. When the old root is determined to be replaced by a new sample landing within the safe zone, height and distance of the root can be safely renewed by those of the new sample.

The expense of being able to handle this special case efficiently is the need to keep the safe zone up-to-date by iterating over the first grandchildren of the root whenever the convex hull tree changes or the second boundary is stepped over. Algorithm 3 describes the process of updating the safe zone at boundaries b_1 and b_2. Although the performance impact of this optimization depends on the height field content, it was always beneficial in our benchmarks and improved the performance on average by 50 %. As a minor optimization, the safe zone information is used to limit the convexity algorithm (line 19 of Algorithm 2) in cases where the root is not being completely replaced and the convexity algorithm has to be invoked, but no grandchildren are affected.

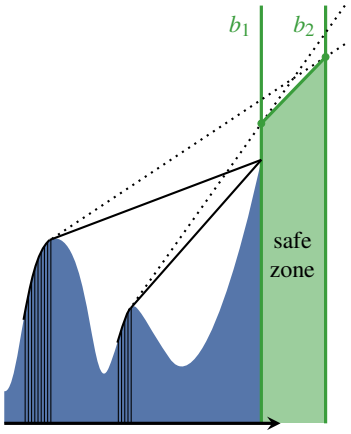

Figure 9: *The lowest projected heights of the second level horizons on the boundaries b_1 and b_2 form a zone where it is safe to replace the root.*

Algorithm 3 SafeZoneUpdate(root, b1, b2)

$\{b1_h, b2_h\} \leftarrow \infty$
c $\leftarrow$ first child of root
do *// Loop over root's children*
  if exists gc $\leftarrow$ first child of c
    // Project grandchild−child line on boundaries
    $b1_h \leftarrow \min(b1_h$, line(gc $\rightarrow$ c) at distance $b1_d)$
    $b2_h \leftarrow \min(b2_h$, line(gc $\rightarrow$ c) at distance $b2_d)$
while exists c $\leftarrow$ next sibling of c

8. Efficiency

In this section we discuss the efficiency of our implementation on an NVidia GTX 480 graphics card in order to give a frame of reference to the previous and the following sections. The two aspects we focus on are the utilization of computational resources and the efficiency of memory accesses. The naïve method is highly efficient from both aspects: neighboring height field points undergo almost exactly the same processing, allowing high utilization, while requiring sampling on neighboring height field coordinates at all times, allowing for optimal texture cache efficiency.

Concurrent threads in our algorithm, however, may perform significantly different amounts of computation per step. In current GPU architectures this yields low SIMT utilization as threads within a warp will have to wait until the longest running thread is finished. Memory accessing in our algorithm is not quite optimal either: elements of convex hull trees are accessed in a pattern where concurrent threads rarely access consecutive memory locations. Also, one thread rarely accesses consecutive indices of its own convex hull tree in a temporally local manner, but rather switches from a branch to the next in subsequent iterations of the algorithm. Therefore there is little *immediate* coherency to be exploited for either efficient memory coalescing or caching. Fortunately, however, the data set for one thread in an N^2 height field is $\sqrt{2}N \times 8$ bytes at most, significantly less on average, and only a small portion of the tree is tra-

versed at each iteration, making transparent caches useful afterall.

To quantify the aspects of computational utilization and memory access, we use seven separate metrics:

Utilization in convexity algorithm (U_C) measures the ratio of active threads in a warp (that consists of 32 consecutive threads) during the while loop in Algorithm 2. This measure reaches 100 % when all threads in a warp execute the same number of iterations.

Utilization during safe zone update (U_S) measures the ratio of active threads in a warp during the safe zone update which requires looping over the children of the root. This measure reaches 100 % when all threads simultaneously decide to update the safe zone and do so in the same number of iterations, e.g. have the same number of visible horizons.

Instruction rate (ipc) measures the average number of instructions issued per clock cycle. A multiprocessor in the GF100 architecture is able to issue at most 2 warp-wide instructions per cycle.

L1 cache hit ratio (C_{L1}) measures the ratio of memory requests that were satisfied by the L1 cache. This includes accesses to both the convex hull trees and the stacks.

16 kB L1 cache hit ratio (C_{L1}^{16k}) measures the ratio of memory requests that were satisfied by the L1 cache when configured to 16 kB.

L2 cache hit ratio (C_{L2}) measures the ratio of memory requests that could not be satisfied by the L1 cache but were satisfied by the L2 cache.

DRAM bandwidth (mem) measures the amount of bytes per second read or written by the memory controller relative to the maximum throughput as measured by the *bandwidthTest* tool in NVIDIA GPU Computing SDK.

Data for the first two metrics are gathered internally by our algorithm. Only the number of loop iterations is measured; the loss in utilization caused by divergence during if-else branching is not included in the measurement. Of these two metrics U_C is more important as the convexity algorithm dominates the overall execution time. Data for the last five metrics are extracted using NVIDIA Compute Visual Profiler [NVI11]. Unless otherwise stated, the L1 cache is configured to 48 kB.

There are four main configurables in our algorithm. Optimal values for them depend on the height field content, but as the values are currently not automatically adjusted at runtime, we use the same values for all height fields. The configurables and the values used are:

- Thread block size, 64
- The number of neighboring height field lines packed together in a thread block, 64
- The number of consecutive threads for which the same indices in the convex hull tree are laid out sequentially in memory, 4
- The number of consecutive threads for which same indices in the stack are laid out sequentially in memory, 8

In Figure 10 we observe the seven metrics for the four test cases shown in Figure 7. We consider the algorithm efficient when measured by ipc, however the utilization U_C is relatively low and has a high impact on the effective instruction rate *per thread*. A work queueing approach might be able to balance load and improve efficiency considerably, but is beyond the scope of our paper. Our algorithm is also memory intensive, and caching plays an important role in overall performance. An increase in the amount of L1 cache per thread would further reduce the strain on the memory subsystem, and would likely improve performance.

9. Performance

We measure our visibility calculation performance against the naïve method implemented as an OpenGL fragment program. Table 3 lists the average time taken to perform a single sweep on the test height fields. The performance of the naïve method does not depend on the height field content. Despite the shortcomings in the efficient mapping of our algorithm to current GPUs as discussed in Section 8, our method is still significantly faster than the naïve method in all test cases.

Table 3: *The average sweep time for the naïve and our method, and relative speedup.*

Height field	Time	Speedup
naïve	21.2 ms	
fractal terrain	8.85 ms	*2.4x*
brick surface	5.05 ms	*4.2x*
sine grid	1.61 ms	*13x*
blocks	0.52 ms	*41x*
Figure 11	3.14 ms	*6.8x*
Figure 12	0.59 ms	*36x*

Choosing the number of azimuthal directions K is a balancing act between performance and approximation accuracy. In principle, to cover every single height field point for each receiver point, $O(N)$ directions need to be processed. A lot fewer are usually adequate, but the appropriate value of K depends on e.g. the geometric content, BRDF, and frequency and intensity of the surface exit radiance. Different values of K for the indirect lighting component are demonstrated on a diffuse monochromatic surface under outdoor lighting in Figure 14. Solving visibility with $K = 32$ over the varying geometry in Figure 11 is achieved at 10 fps.

10. Discussion

10.1. Lighting

Although lighting methods that utilize the visibility information produced by our visibility algorithm are future work,

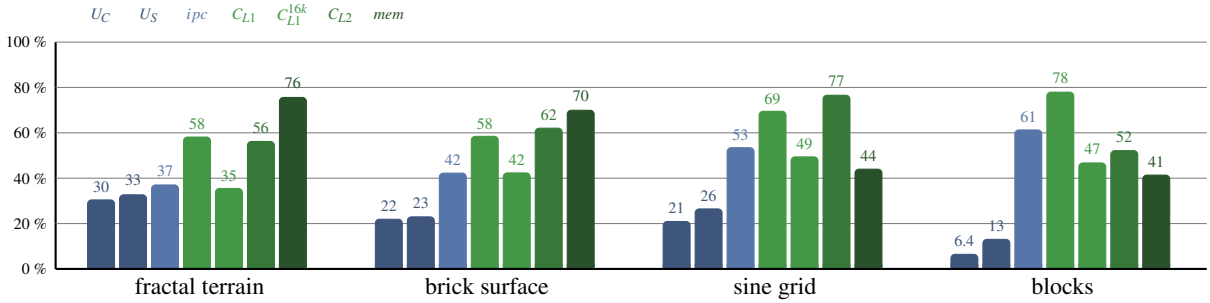

Figure 10: *Efficiency measurements for four test height fields. From the left, the metrics are: utilization in convexity algorithm (U_C), utilization during safe zone update (U_S), instruction rate (ipc), L1 cache hit ratio (C_{L1}), 16 kB L1 cache hit ratio (C_{L1}^{16k}), L2 cache hit ratio (C_{L2}), and DRAM bandwidth (mem).*

in this section we outline ways to utilize our visibility description and also demonstrate, through trivially gathering the surface radiance, that also lighting within the lowered complexity is possible.

The visibility horizons as produced by our method provide exact angular coordinates for both the beginning and the end of a segment's visibility. The Cartesian coordinate is only available for the beginning. We suggest three ways to utilize this type of visibility information, in an order of increasing computational complexity:

1. As a line on the height field is traversed, record its accumulated exit radiance in a way that allows sampling the total segment radiance using the angular coordinates.
2. Record the accumulated exit radiance so that it can be sampled using distance coordinates. To find the exact Cartesian end coordinate of the visibiliy use any of the various existing intersection search algorithms. Two properties might prove useful: (i) there is exactly one intersection point between the horizon line and the segment and (ii) the behaviour of the derivative is known due to the convexity/concavity requirements.
3. Traverse the visible points one at a time by starting from each beginning coordinate and stepping towards the receiver point until below the next horizon (cf. Figure 3). This allows sampling of only the visible points (plus the partially visible at each horizon boundary). The visible segments can be traversed in parallel.

The potentially visible height field points and their exit radiance along the line have already been traversed when it is time to determine incident radiance for a receptor. Therefore we find promise in fast lighting methods that accumulate a description of exit radiance per segment such that it can, ideally, be sampled directly with the information available in our visibility horizons (alternative 1).

We demonstrate a trivial use of the 3. (slowest) alternative by sampling through the exit radiance one point at a time and computing the contribution on the receiver analytically. While this method is slow compared to visibility determination, it performs in the complexity of Table 1 and Figure 8 while producing exact lighting. Figure 11 demonstrates global lighting with one indirect bounce under direct lighting of a single point light. The complete lighting performed this way is faster than only finding the visible points using the naïve method.

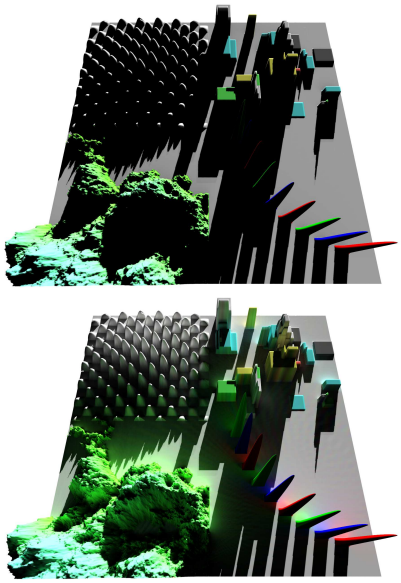

Figure 11: *Top: direct illumination from a point light source on a 1024^2 height field. Bottom: an additional indirect light bounce. Accurate illumination in 64 directions is achieved in 1.21 seconds per frame (19 ms per direction).*

Figure 12 demonstrates global lighting (direct lighting in a dim 256×256 lighting environment plus one indirect

bounce) with parts of the height field emitting light by themselves. In this case, we achieve pixel-perfect lighting with an unbound extent at a per-direction rate higher than that of the approximative method in [NS09]. Multiple indirect bounces can be simulated by repeating the indirect lighting phase using the output of the previous iteration as the new surface radiance.

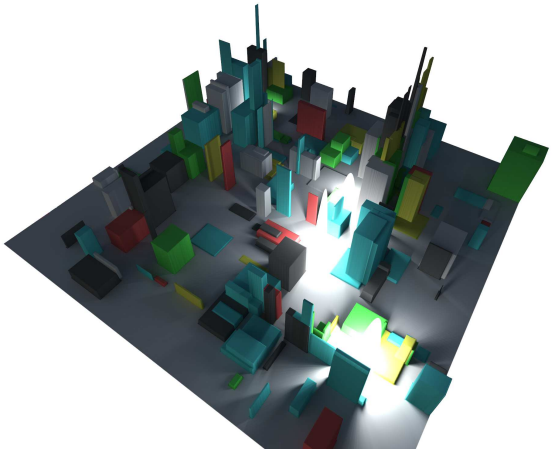

Figure 12: *A 1024^2 height field with self-illuminating parts. Illumination in 256 directions achieved in 1.68 seconds per frame (6.6 ms per direction).*

10.2. Error analysis

Errors in our method come from (i) sampling along each line, and (ii) the discretization of the visibility search into K azimuthal directions. All image-space methods suffer from the first issue: geometry is visited at sampled points, which generally are not the original height field points. Using heavy supersampling along a line is considered to produce the ground truth. The more identifiable approximation in our method is the discretization into azimuthal directions. Visibility solved using a large value of K represents the ground truth in this aspect.

In this section we analyze the error introduced by these two issues as compared to the respective ground truths. As the test case, we use a diffuse height field under outdoor lighting, shown in Figure 13. The error is visualized in Figure 14 and average error plotted in Figure 15. Generally the error in indirect illumination is amplified by high-frequency details in geometry and glossiness of the surface.

Azimuthal undersampling results in banding that is especially apparent on flat regions of a height field, however less visible on uneven regions. One might argue that our method is imbalanced for determining visibility along each line accurately while crudely undersampling azimuthally, however quantitative analysis implies that accurate visibility determination is necessary even for a small number of azimuthal directions: Figure 15 shows that $K = 16$ introduces an error that is much smaller than that coming from unit length sampling ($S = 1$). Qualitatively though, banding can be the most outstanding visual artefact and in the next section we discuss a way to trade it for noise.

10.3. Future work

As future optimizations, it may be possible to trigger a simplification of the convex hull tree for parts far away from the receptor every few dozen steps. Also because the algorithm has traversed the data of the line already, the simplification process can utilize the exit radiance and geometry information along the line to estimate the impact of the simplification in order to control the error in the resulting approximation. Any such simplification can also be performed while processing height field samples for the first time. For example, it might be a reasonable optimization to ignore fine levels of roughness on otherwise flat surface regions and favor extrusions over depressions when determining exit radiance from the simplified regions. Other, non-sweep based methods, are unaware of the contents of a line before taking the actual samples, and therefore cannot trivially apply said data-aware optimizations.

Previous methods based on the naïve method that perform an independent visibility search for each receiver point can randomize azimuthal directions per receiver, and therefore trade banding for noise. Trading banding for noise is also possible in our method: one can perform a sparse sweep in each direction (process every n-th line) while simultaneously increasing K. Afterwards, when gathering results for a point in the result buffer, one accumulates lighting only from directions that have a line that crosses the receiver point. Furthermore, a line does not have to strictly cross the receiver, but a configurable distance epsilon can be used to provide

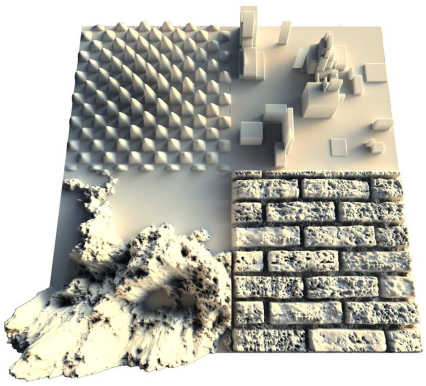

Figure 13: *The height field used for error testing. The surface exhibits diffuse reflection and is lit under outdoor lighting.*

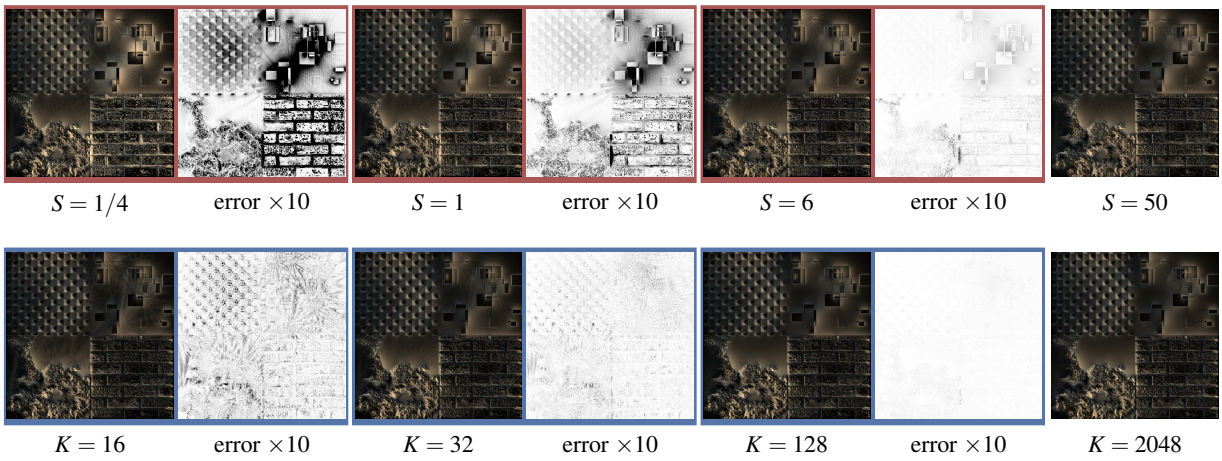

Figure 14: *The indirect illumination component from Figure 13 and the corresponding error (white = 0 %, black = 10 %) against ground truth (right) for different levels of visibility supersampling (on red) and values of azimuthal directions (on blue).*

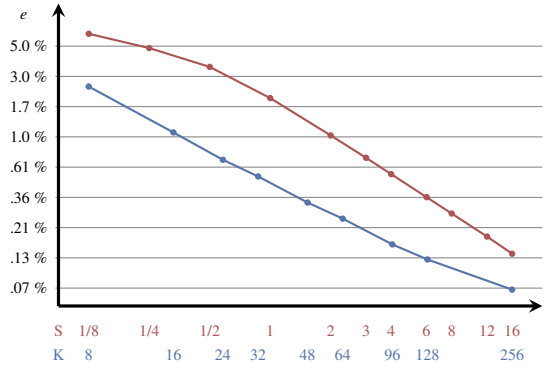

Figure 15: *The average error e per pixel for different values of azimuthal directions K and supersampling S. The axes are logarithmic.*

a way to trade noise for blur. Randomizing azimuthal directions in previous methods incurs a penalty of lowered texture cache hit ratio, the impact of which can be significant. Our method takes very few height field samples per receiver and is not bottlenecked by sampling, making it practically immune to this penalty.

When it comes to applications, height field methods have proven useful in producing lighting effects in screen space. The main approximations of depth buffer geometry are (i) not counting geometry outside the visible frame buffer and (ii) taking only the first (visible) depth layer into account. It is possible to alleviate these limitations [BS09], and we expect our algorithm to prove useful in producing properly occluded *global* screen-space indirect illumination effects not yet seen in interactive graphics. Thus an interesting avenue of further research is to investigate the possibility of producing global scene lighting entirely in screen space, with light sources rendered in HDR as part of the scene geometry.

11. Conclusion

Determining intervisibility on surfaces of objects is a problem that needs to be solved in various applications, including global illumination systems in computer graphics. Unfortunately, the problem is complex and its current solutions computationally expensive. For height field geometry, the problem can be reduced from 2.5D to 1.5D domain by approximating visibility in discrete azimuthal directions. The most compact way currently known to express intervisibility in the 1.5D case are local visibility horizons.

In this paper we have presented a novel algorithm that determines local visibility horizons using *incremental convex hull trees*. Visibility from every height field point is determined to a number of azimuthal directions in time that is linear in the number of output visibility horizons, making the algorithm of optimal time complexity. We have showed that the proportion of visibility to the whole height field is low, giving our algorithm an advantage over the previous methods. In practice, we achieve a reduction by two orders of magnitude in the number of iterations required to produce the accurate visibility information. Our method is also amenable for GPGPU implementations and we have demonstrated that such an implementation can achieve significantly better performance than previous work.

Acknowledgements

I would like to thank Prof. Jan Westerholm for his advice and support and Dr. Jukka Arvo for his helpful comments on the manuscript.

References

[ADM*08] ANNEN T., DONG Z., MERTENS T., BEKAERT P., SEIDEL H.-P., KAUTZ J.: Real-time, all-frequency shadows in dynamic scenes. *ACM Trans. Graph. 27*, 3 (2008), 1–8.

[BS09] BAVOIL L., SAINZ M.: Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH '09: SIGGRAPH 2009: Talks* (New York, NY, USA, 2009), ACM.

[Bun05] BUNNELL M.: *Dynamic ambient occlusion and indirect lighting*. Addison-Weseley Professional, 2005, pp. 223–233.

[BWW05] BITTNER J., WONKA P., WIMMER M.: Fast exact from-region visibility in urban scenes. In *Rendering Techniques 2005 (Proceedings Eurographics Symposium on Rendering)* (June 2005), Bala K., Dutré P., (Eds.), Eurographics, Eurographics Association, pp. 223–230.

[CoS95] COHEN-OR D., SHAKED A.: Visibility and dead-zones in digital terrain maps. *Computer Graphics Forum 14* (1995), 171–180.

[CS89] COLE R., SHARIR M.: Visibility problems for polyhedral terrains. *J. Symb. Comput. 7*, 1 (1989), 11–30.

[DBBS06] DUTRE P., BALA K., BEKAERT P., SHIRLEY P.: *Advanced Global Illumination*. AK Peters Ltd, 2006.

[DBS08] DIMITROV R., BAVOIL L., SAINZ M.: Horizon-split ambient occlusion. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), ACM.

[DFM94] DE FLORIANI L., MAGILLO P.: Computing point visibility on a terrain based on a nested horizon structure. In *SAC '94: Proceedings of the 1994 ACM symposium on Applied computing* (New York, NY, USA, 1994), ACM, pp. 318–322.

[DSDD07] DACHSBACHER C., STAMMINGER M., DRETTAKIS G., DURAND F.: Implicit visibility and antiradiance for interactive global illumination. *ACM Trans. Graph. 26*, 3 (2007), 61.

[FHT09] FISHMAN J., HAVERKORT H., TOMA L.: Improved visibility computation on massive grid terrains. In *GIS '09: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (New York, NY, USA, 2009), ACM, pp. 121–130.

[HBR*11] HUANG J., BOUBEKEUR T., RITSCHEL T., HOLLÄNDER M., EISEMANN E.: Separable approximation of ambient occlusion. In *Eurographics 2011 - Short papers* (2011), pp. 29–32.

[HLHS03] HASENFRATZ J.-M., LAPIERRE M., HOLZSCHUCH N., SILLION F.: A survey of real-time soft shadows algorithms. *Computer Graphics Forum 22*, 4 (dec 2003), 753–774.

[KZ02] KAUČIČ B., ZALIK B.: Comparison of viewshed algorithms on regular spaced points. In *SCCG '02: Proceedings of the 18th spring conference on Computer graphics* (New York, NY, USA, 2002), ACM, pp. 177–183.

[LS10] LOOS B. J., SLOAN P.-P.: Volumetric obscurance. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D '10, ACM, pp. 151–156.

[Mit07] MITTRING M.: Finding next gen: Cryengine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses* (New York, NY, USA, 2007), ACM, pp. 97–121.

[Nag94] NAGY G.: Terrain visibility. *Computers & Graphics 18*, 6 (1994), 763–773.

[NS09] NOWROUZEZAHRAI D., SNYDER J.: Fast global illumination on dynamic height fields. *Computer Graphics Forum: Eurographics Symposium on Rendering 28*, 4 (June 2009), 1131–1139.

[NVI09] NVIDIA CORPORATION: *NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Whitepaper*, 1.1 ed. Santa Clara, CA, 2009.

[NVI11] NVIDIA CORPORATION: *NVIDIA Compute Visual Profiler 3.2*. Santa Clara, CA, 2011.

[RGK*08] RITSCHEL T., GROSCH T., KIM M. H., SEIDEL H.-P., DACHSBACHER C., KAUTZ J.: Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Trans. Graph. (Proc. of SIGGRAPH ASIA 2008) 27*, 5 (2008).

[RGS09] RITSCHEL T., GROSCH T., SEIDEL H.-P.: Approximating dynamic global illumination in image space. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2009), ACM, pp. 75–82.

[SHR10] SOLER C., HOEL O., ROCHET F.: A deferred shading pipeline for real-time indirect illumination. In *ACM SIGGRAPH 2010 Talks* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 18:1–18:1.

[SLYY08] SHEN Y., LIN L., YANG M., YURONG G.: Viewshed computation based on los scanning. In *2008 International Conference on Computer Science and Software Engineering* (dec. 2008), vol. 2, pp. 984 –987.

[SN08] SNYDER J., NOWROUZEZAHRAI D.: Fast soft self-shadowing on dynamic height fields. *Computer Graphics Forum: Eurographics Symposium on Rendering 27*, 4 (June 2008), 1275–1283.

[TW10] TIMONEN V., WESTERHOLM J.: Scalable Height Field Self-Shadowing. *Computer Graphics Forum (Proceedings of Eurographics 2010) 29*, 2 (May 2010), 723–731.