Computing Terrain Multi-visibility Maps for a Set of View Segments Using Graphics Hardware*

N. Coll, M. Fort, N. Madern, and J. A. Sellarès

Universitat de Girona, Spain
{coll,mfort,nmadern,sellares}@ima.udg.es

Abstract. A multi-visibility map is the subdivision of the domain of a terrain into different regions that, according to different criteria, encodes the visibility concerning a set of view elements. In this paper we present an approach for computing a discrete approximation of a multi-visibility map of a triangulated terrain corresponding to a set of view segments, for weak and strong visibility, by using graphics hardware.

1 Introduction

Visibility information of terrain areas is necessary in many Geographic Information Systems applications, such as path planning, mobile phone networks design and environmental modelling.

The visibility map is a structure that encodes the weak or strong visibility of a terrain concerning a view element (point, segment, triangle, etc.) belonging to or above the terrain. Visibility structures for several view elements, that we generically call multi-visibility maps, can be defined by combining the visibility map of such elements according to some operators, for example intersection, union or counting. Taking into account that the representation of a terrain is a rough approximation of the underlying terrain and that the combinatorial complexity of exact multi-visibility maps is too complex to compute them exactly, an approximation of a multi-visibility map is often considered sufficient.

Algorithms for computing the visibility map corresponding to a point are available in [3, 7, 8, 11]. There also exist some recent papers dealing with multi-segment visibility [5] and inter-region visibility [5, 12]. In [5] a multi-visibility map is approximated using an algorithm based on an approach that reconstructs an approximation of an unknown planar subdivision from information gathered from linear probes of the subdivision [4]. The multi-visibility map is obtained by repeatedly executing an algorithm that computes segment-segment visibility in an exact fashion: the visible parts of a segment on the terrain from a view segment.

The increasing programmability and high computational rates of graphics processing units (GPUs) make them attractive as an alternative to CPUs for

* Partially supported by the Spanish Ministerio de Educación y Ciencia under grant TIN2004-08065-C02-02. Marta Fort is also partially supported by the Spanish Ministerio de Educación y Ciencia under grant AP2003-4305. Narcís Madern is also partially supported by the Ministerio de Educación y Ciencia under grant BES2005-9541.

general-purpose computing. Recently, different algorithms and applications that exploit the inherent parallelism and vector processing capabilities of GPUs have been proposed [16]. In computational geometry there exist several algorithms that have a fast hardware-based implementation [1, 6, 9, 10, 13, 14].

In this paper we address the problem of computing approximate multi-visibility maps for a terrain modelled by a TIN concerning a set of view segments, for weak and strong visibility. Since we consider discrete approximations of multi-visibility maps, it suffices to compute solutions on a grid covering the domain of the terrain. For computational purposes, we discretize the domain of the terrain into pixels. This discretization allows us to exploit graphics hardware capabilities. Our approach repeatedly uses the skewed projection for representing segment-segment visibility in a bounded two-dimensional space. Once again, the discretization into pixels of this two-dimensional space allows us to quickly compute segment-segment visibility using graphics hardware.

2 Preliminaries

We model a terrain as the graph of a *Triangulated Irregular Network* (TIN), $(\mathcal{T}, \mathcal{F})$, formed by a triangulation $\mathcal{T} = \{T_1, \cdots, T_n\}$ of the domain D (in the xy-plane), and by a family $\mathcal{F} = \{f_1, \cdots, f_n\}$ of linear functions such that: a) function $f_i \in \mathcal{F}$ is defined on triangle T_i, $i = 1..n$; b) for any pair of adjacent triangles T_i and T_j, f_i and f_j coincide in $T_i \cap T_j$. For any triangle $T_i \in \mathcal{T}$, $f_i(T_i)$ is a triangle in space called a *face* of the terrain, and the restriction of f_i to an edge or a vertex of T_i is called an *edge* or a *vertex* of the terrain.

A *view element* v (point, segment, triangle, etc.) is an element belonging to or above the terrain, i.e. its projection onto the xy-plane is contained in D, and for any point (x, y, z) of v, $z \geq z'$, where (x, y, z') is a point on the terrain. A point P on the terrain is: *visible* from a view point V if the interior points of the line segment VP with endpoints V and P lie above the terrain; *weakly visible* from a view segment or polygon v if P is visible at least from a point of v; *strongly visible* from a view segment or polygon v if P is visible from any point of v.

The *visibility map* related to a view element v consists of a partition of the domain D into visible and non visible maximal connected regions. If a region is labelled as visible (non visible) then all points on the terrain whose vertical projection is in the region are visible (non visible) from v, considering weak or strong visibility.

Multi-visibility maps related to a set V of r view elements are defined combining the single visibility maps of such elements. A multi-visibility map is a subdivision of the domain D of the terrain into regions according to different visibility criteria. Typical multi-visibility maps are *union*, *intersection*, *counting* and *overlay*. Union yields domain portions visible from at least one view element; intersection yields domain portions visible from all the view elements; counting yields domain portions according to the cardinal of the set of view elements from which a region is visible; overlay is obtained superimposing the visibility maps of all view elements and labelling each region in the resulting partition of the domain with the set of view elements from which that region is visible.

2.1 Graphics Hardware

The rendering pipeline [15] is divided into several stages. Each stage is implemented as a separate piece of hardware on the GPU. The input to the pipeline is a list of 3D geometric primitives, expressed as vertices, and the output is an image in the frame buffer ready to be displayed on a screen. A vertex defines a point, an endpoint of a line, a corner of a polygon, etc. The attributes associated to a vertex are 3D coordinates, color, texture coordinates, etc. The frame buffer is a collection of several hardware buffers, each of them corresponds to an uniform two dimensional grid, composed of cells called pixels. Each pixel in the frame buffer is a set of some number of bits grouped together into several buffers. There is a special buffer, the *Color buffer*, that mainly contains RGB color information. Arrays in GPU memory are called textures. The coordinates of a texture are used to access the values stored in textures. Textures are mainly used to store information that will be used in the per-fragment operations to decide or modify a pixel color. Usually the maximal size of a texture is 2048×2048 or 4096×4096 and at most 8 textures can be used simultaneously.

In the first stage of the pipeline per-vertex operations take place. Each input vertex is transformed depending on the point-of-view and the defined transformations, lighting calculations may also be performed to determine its color. The next stage is rasterization. The result of this stage is a fragment for each pixel location covered by a geometric primitive. Each of them corresponds to a pixel in the frame buffer with a depth value. The third stage, the fragment stage, computes the color for each pixel according to the fragments corresponding to it. Each fragment has to pass different tests and per-fragment operations before being placed into the frame buffer. Such operations can use values from global memory in the form of textures and include updating, blending, masking and other logical operations. Finally, the fragments that pass the tests and the operations of the previous stage are drawn or rendered on the screen with the appropriate color.

Logical operations are established by calling *glLogicOp* specifying the desired logical operation: *AND_LOGIC, OR_LOGIC*, etc. They are performed on the color of the incoming fragment and the previously stored pixel color. Information of a user defined rectangle of the color buffer can be transferred to the CPU in a summarized way with *getMinmax* or *getHistogram*. The minimum and maximum pixel colors contained in the rectangle are given by *getMinmax* and the number of occurrences of each color by *getHistogram*.

A fragment shader, also sometimes called pixel shader, is a programable function executed on a per-fragment basis. A fragment shader allows combination of fragment values, such as color and position on screen, with texture values to change the appearance of the pixels. Textures are sent as parameters to the fragment shader.

3 The Skewed Projection

Let v and s be two non coplanar segments in space parameterized by t and u in $[0, l_v]$ and $[0, l_s]$ where l_v and l_s are the lengths of v and s, respectively. Denote $T_{v,s}$ the tetrahedron determined by the endpoints of v and s.

For any point P in $T_{v,s} \setminus \{v, s\}$, let $\pi_{P,v}$ and $\pi_{P,s}$ be the planes determined by the point P and the segments v and s, respectively. Let $w_P = \pi_{P,v} \cap \pi_{P,s}$ be the unique line passing through P, v and s. Denote t_P, u_P the parameters of the points v_P and s_P where w_P intersects segments v and s, respectively.

Consider the continuous function, sk, that maps a point P in the tetrahedron $T_{v,s} \setminus \{v, s\}$ to the point $(t_P, u_P) \in [0, l_v] \times [0, l_s]$. Observe that all points of the segment whose endpoints are v_P and s_P are mapped to (t_P, u_P). The map sk is essentially the restriction to the tetrahedron $T_{v,s} \setminus \{v, s\}$ of the *skewed projection* introduced in [17].

The skewed projection $sk(s')$ of a segment s' in $T_{v,s} \setminus \{v, s\}$ is a connected conic arc contained in $[0, l_v] \times [0, l_s]$. The conic is a hyperbola or a parabola and, in degenerate cases, a line or a point. Consequently, the skewed projection $sk(T)$ of the region bounded by a triangle T in $T_{v,s} \setminus \{v, s\}$ is a region of $[0, l_v] \times [0, l_s]$ bounded by three connected conic arcs, one per edge of T. It is not difficult to prove that the region $sk(T)$ can only by defined by: (a) three conic edges with three or four vertices, (b) two conic edges with two vertices (an edge is mapped to a point). In Figure 1a) we can see (in white) the skewed projection of a triangle that yields a region with three vertices, and in Figure 1b) we can see the skewed projection of a triangle yielding a region with four vertices.

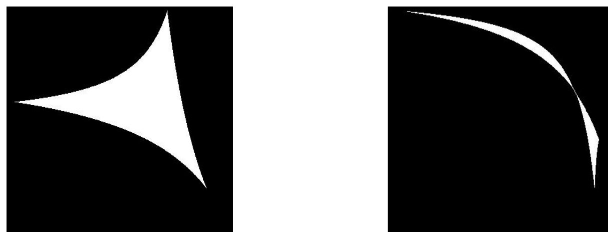

Fig. 1. : a) Region with 3 boundary vertices. b) Region with 4 boundary vertices.

3.1 Extending the Skewed Projection

Let l be a segment in $T_{v,s} \setminus \{v, s\}$ coplanar with s. Then $sk(l) = \{(t_l, u) | u \in I \subset [0, l_s]\}$ is a vertical segment in $[0, l_v] \times [0, l_s]$, where t_l is the parameter of the point where v intersects with the plane through s and l. Let l be a segment in $T_{v,s}$ with an endpoint B in s. We know that $sk(l \setminus B) = \{(t_l, u) | u \in I \subset [0, l_s]\}$ and we continuously extend it to the whole segment l taking $sk(B) = (t_l, u_B)$, where u_B is the parameter of B in s. Let T be a triangle with a vertex B in s, and denote b the edge of T opposite to B. We can express T as the union of segments with endpoints in the edge b of T and B respectively. We extend the skewed projection from $sk(T \setminus B)$ to the whole T by defining $sk(T) = \bigcup_{C \in b} sk(CB)$, so that $sk(B) = \{(t, u_B) | t \in I \subset [0, l_v]\}$ is an horitzontal segment in $[0, l_v] \times [0, l_s]$.

In a similar way we can extend the skewed projection to the endpoint of a segment or to the vertex of a triangle located in v.

4 Segment-Segment Visibility Computation

Given a view segment v and a segment s on a face of the terrain, we want to compute the visible parts of s from v. We focus on the case in which v and s are non coplanar, so that they determine the tetrahedron $T_{v,s}$. The study of the simpler case in which v and s are coplanar is omitted due to space limitations.

To simplify computations we apply a preprocess in two stages. First we delete the subsegment of v not facing the front side of the face containing s. If s is an edge of the terrain, we delete the subsegment of v that does not face the front side of any of the two faces containing s. If the entire segment v is deleted, then s will be not visible at all from v. Otherwise, we take as view segment the part of v not deleted, that we still denote v. If segment v belongs to the terrain an equivalent study has to be done. This first step outputs two segments s and v whose faces do not block each others visibility.

The other parts of the terrain that may hide s must be contained in $T_{v,s}$. The second preprocessing stage allows us to determine these parts. Observe that the intersection of a face with $T_{v,s}$ is a convex polygon of constant complexity that can be easily triangulated. We find the faces intersecting $T_{v,s}$ by considering the m triangles of the triangulation $\mathcal{T}$ of the domain D that intersect the projection of $T_{v,s}$ onto D [2]. In the worst case we have $m = O(n)$. Next we compute the polygons obtained intersecting these faces with $T_{v,s}$. Finally, we triangulate all these polygons obtaining a set F of triangles. Abusing language, the obtained triangles are still called faces. Clearly in the worst case is $|F| = O(m)$ and the total cost of the whole preprocess is $O(m)$.

After the preprocessing step: a) all parts of the view segment $v(s)$ face the front side of the face/s containing $s(v)$; b) we consider only the set F of triangular faces contained in $T_{v,s}$. Observe that may exist some triangle in F with a vertex in an endpoint of s or v.

A point of s is visible from a point of v if the segment connecting them does not go through any face. Segments holding this condition are called *non blocked* segments. Thus, by determining the set of non blocked segments we are able to obtain the visible parts of s from v. We use the skewed projection sk to map the triangular faces T of F to $[0, l_v] \times [0, l_s]$. Let $U = \bigcup_{T \in F} sk(T)$ be the union of the images of the faces of F under sk. Clearly, points in $\overline{U} = [0, l_v] \times [0, l_s] - U$ correspond to non blocked segments. Let $Y = \{(0, u) | u \in [0, l_s]\}$ and define the orthogonal projection of point $(t, u) \in [0, l_v] \times [0, l_s]$ onto Y by $pr_Y(t, u) = (0, u)$. The weakly visible parts of s, points that are visible from at least one point of v, are represented by $pr_Y(\overline{U})$, whereas the strongly visible parts of s, points that are visible from the whole segment v, are represented by $\overline{pr_Y(U)} = Y - pr_Y(U)$.

The computation of U can be performed exactly using a sweep line algorithm in worst-case time $O(|F|^2)$. However, U can be also approximately computed with a simple algorithm by discretizing $[0, l_v] \times [0, l_s]$ into pixels and rendering each region $s(T)$ using graphics hardware. We use a parameter μ that defines the number of pixels per unit length, so that the number of pixels per row and per column are $\mu \cdot l_v$ and $\mu \cdot l_s$, respectively. All regions $sk(T)$ are drawn in white onto an initial black background. Final black points correspond to

non blocked segments. In this way we avoid analytic computations and some robustness problems common in geometric algorithms.

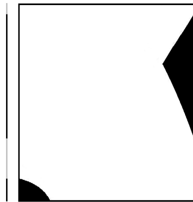

Fig. 2. Black pixels represent non blocked lines. White pixels represent the union of the skewed projections. In the left vertical segment, the weakly visibles parts of segment s are painted black and the non visible ones in grey.

4.1 Rendering the Skewed Projection of a Triangle

We know that the skewed projection $sk(T)$ of a triangle T in $T_{v,s} \setminus \{v, s\}$ is a region of $[0, l_v] \times [0, l_s]$ bounded by: (a) three conic edges and three or four vertices, (b) two conic edges and two vertices. In a preprocessing step we split regions of type (a) into two or three parts in the following way. A region with four vertices is split into two regions: one formed by two edges and two vertices and another by three edges and three vertices. Regions formed by three vertices bounded by three non straight line edges are split into two regions by a vertical or horizontal line through one of its vertices. We denote $\mathcal{R}$ the set of all the current regions. After the preprocessing stage each region $R \in \mathcal{R}$ can be described by a system I_R of quadratic or linear inequalities, with one inequality per edge.

To determine the part of $[0, l_v] \times [0, l_s]$ covered by $U = \bigcup_{R \in \mathcal{R}} R$ we use graphics hardware. We discretize $[0, l_v] \times [0, l_s]$ into pixels and we paint each region $R \in \mathcal{R}$ in white on an initially black background. To determine the pixels that correspond to a region R we consider a collection of geometric primitives $\mathcal{C}_R$ whose union contains R, and a fragment shader whose input parameters are the coefficients of the system I_R. The fragment shader assigns white color to the fragments that fulfill I_R and black color to the others. Since the time required by the fragment shader is proportional to the number of fragments obtained by the rasterization of $\mathcal{C}_R$, we choose this region so that the number of fragments is reasonably small. We associate to each non straight convex edge e of R a rectangle R_e that has one side with endpoints in the vertices of e and its opposite parallel side tangent to e. We define $\mathcal{C}_R$ as the union of the rectangles associated to the convex edges of R and, in case that R has three vertices, the straight triangle defined by the vertices of R (See Figure 3).

4.2 Computing the Visible Parts of a Segment

Once the union $U = \bigcup_{T \in F} sk(T)$ has been rendered in white. We compute $pr_Y(\overline{U})$ or $\overline{pr_Y(U)}$ in order to determine the weakly or strongly visible parts of s from v, respectively. Since the OpenGL *getMinmax* function allows determination of the minimal and maximal pixel color (white is greater than black in

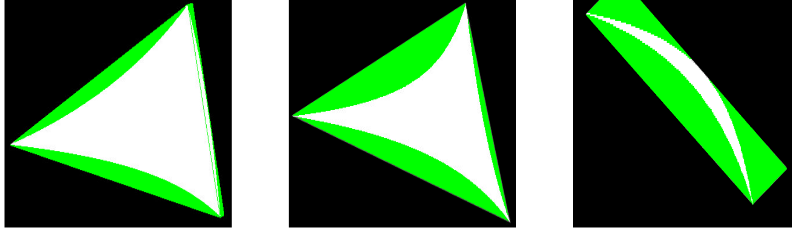

Fig. 3. In white the region determined by the conics, in green and white the region that we send to rastarize and that goes through the Fragment Shader.

RGB code) in a row of the screen, it can be used in order to compute $pr_Y(\overline{U})$ or $\overline{pr_Y(U)}$. We obtain $pr_Y(\overline{U})$ by scanning through all the rows of the screen and considering that the projection of a row is white when the minimal color of its corresponding row is. Similarly we obtain $\overline{pr_Y(U)}$ projecting into Y in black when the maximal color of the row is black. The result of the computation is stored in the CPU.

4.3 Acceleration

Tacking into account that we only need one bit of the color buffer per pixel to represent the union of skewed projections corresponding to one pair of segments (v, s), we can simultaneously represent in the color buffer the unions corresponding to a collection of pairs of segments (v, s_1) , ..., (v, s_k), with $1 \leq k \leq 24$. Then we use the per-fragment *logicOp*, specifying the *OR_LOGIC* operation, to merge the color of the already rendered fragments with the incoming one. Using the OpenGL *getHistogram* function, instead of the *getMinmax* function, we can compute the projections onto Y of the k pairs of segments (v, s_j).

5 Discrete Multi-visibility Map Computation

In this section we present an algorithm for computing discrete multi-visibility maps relative to a set of view segments using a color code and according to the type of visibility chosen (weak or strong). We give a method to visualize the following types of multi-visibility maps: union, intersection, counting and overlay.

The domain of the terrain is discretized into a rectangular grid of size $H \times W$. This size is chosen according to a parameter μ' representing the number of grid points per unit length of the terrain domain. The grid is mapped into an unidimensional array $\mathcal{A}$ of integers (8 bits) of length $4 \cdot H \cdot W$, in such a way that each grid element is mapped in 4 consecutive elements of $\mathcal{A}$. Next the array is transferred from the CPU to a texture in the GPU that stores values of 32 bits (8 bits per 4 channels). Each 4 consecutive elements of $\mathcal{A}$ are transferred to a texture value. Since only 8 textures can be used simultaneously, the maximal number of view segments is limited to $256 = 8 \cdot 32$.

In a first step, for each view segment we compute its discrete visibility map and we store it in array $\mathcal{A}$. The discrete visibility map for a view segment is

obtained by scanning all the rows of the grid and computing, using the segment-segment-visibility algorithm, the visibility of the set of segments on the terrain that correspond to a row of the grid. Each one of these segments is obtained by first computing the intersection of a horizontal line and a triangle of the triangulation of the domain and then lifting the intersection into a segment on a triangle of the terrain. For each grid position we store visibility information in an index of $\mathcal{A}$. A bit suffices to represent the visibility: this bit is set to 1 when the grid position is visible and to 0 otherwise. In fact, to increase the computational efficiency, we simultaneously store the visibility information for up to 32 view segments in $\mathcal{A}$. In the case when we have more than 32 view segments, we need a new array for every 32 view segments or fraction.

Once all the visibility information is computed it is transferred to the GPU, then, by using a proper fragment shader, which has the textures as input parameters, we can visualize any multi-visibility map in a $H' \times W'$ window of the screen for any region of the terrain we are interested in. When in a texture we have k view segments, $1 \leq k \leq 32$, the fragment shader only uses the corresponding k bits of the texture values. Next we describe how the union, intersection and counting fragment shaders work.

The union fragment shader paints white a pixel corresponding to a position of the texture if it has some bit equal to 1 (visible) and black otherwise (not visible); the intersection fragment shader paints white a pixel corresponding to a position of the texture if it has all bits equal to 1 (visible) and black otherwise (not visible); the counting fragment shader paints each pixel in the appropriate color according to the number of bits equal to 1 of the corresponding position of the texture.

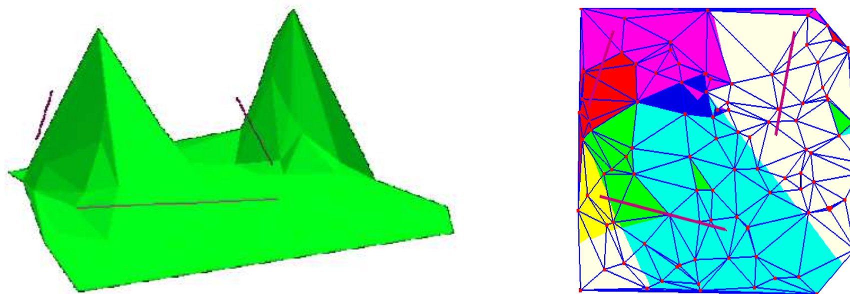

Fig. 4. a) Terrain and three view segments, v_1 on the left, v_2 on the right and v_3 on the front. b)The corresponding multi-visibility map: points visible from v_1 are painted red, from v_2 in green, from v_3 in blue, from v_1 and v_2 in yellow, from v_1 and v_3 in pink, from v_2 and v_3 in light blue and from v_1, v_2 and v_3 in white.

It is not difficult to adapt our method to visualize the union, intersection and counting for a subset of view segments; the overlay of e view segments, assuming that we are able to distinguish 2^e colors; the special case of overlay that we

describe next. Let $\mathcal{U} = \{V_1, ..V_g\}$ be a user specified collection of subsets of view segments such that $V_i \cup V_j \in \mathcal{U}$, $i, j = 1..g$. The terrain domain is subdivided into regions $R_1, \cdots, R_g, R_{g+1}$ of different colors, in such a way that the region R_i, $i = 1..g$, contains the points of the terrain that are visible from all the view segments of V_i, and region R_{g+1} contains the remaining points.

Moreover, without recomputing the information and only adapting the window to a specific region of the domain, we can visualize any multi-visibility map we are interested in.

6 Complexity Analysis

Given a terrain and a set of view segments, to analyze the cost of visualizing a discrete multi-visibility map on a grid of size $H \times W$ some important considerations have to be taken into account:

- The number of intersections between H lines and the n triangles of the domain is $O(nH)$ in the worst case. The maximal number of tetrahedra to be considered is $O(rnH)$, where r is the number of view segments.
- The faces of the terrain that intersect a tetrahedron are determined in $O(n)$ worst case time. Let c be the time needed to render the skewed projections of a triangle. Then, the total time expended in processing faces is $O(rn^2Hc)$.
- Let G be the time expended by the *getHistogram* function. Each cluster of 24 tetrahedra uses this function. Then, the total time expended by *getHistogram* is $O(rnH\frac{G}{24})$.
- Let M be the time of loading a texture of size $H \times W$ to the GPU. Each cluster of 32 view segments uses a texture. Then, the total time of loading textures is $O(r\frac{M}{32})$.
- The visualization of a region of a multi-visibility map in a $H' \times W'$ window can be done in constant time.

According to these considerations we obtain that the computational cost of visualizing a multi-visibility map is $O\left(rnH\left(nc + \frac{G}{24}\right) + r\frac{M}{32}\right)$.

Since visualizing a multi-visibility map takes constant time, the time needed to visualize N additional multi-visibility maps is $O(N)$.

7 Conclusions and Future Work

We have presented a method for visualizing multi-visibility maps of a triangulated terrain concerning a set of view segments, for weak and strong visibility, using graphics hardware. We want to remark that our method can be easily parallelized. We have implemented the described algorithms and in our implementation parameters μ and μ' can be chosen by the user. Presently we are studying the best combinations of these parameters in order to reduce errors produced during the different discretizations. We are also working to extend our method to heterogeneous sets of view elements: segments, polygonal lines and polygonal regions.

Acknowledgements

We thank Teresa Paradinas for helping us to implement our algorithms.

References

1. P. K. Agarwal, S. Krishnan, N. H. Mustafa, S. Venkatasubramanian, Streaming Geometric Optimization Using Graphics Hardware. European Symposium on Algorithms, 544–555, 2003.
2. M.D. Berg, M.V. Kreveld, R.V. Oostrum, M. Overmars, Simple traversal of a subdivision without extra storage, *International Journal of Geographical Information Science*, Vol 11, 4(15):359–373,1997.
3. M. Bern, D. Dobkin, D. Eppstein, and R. Grossman., Visibility with a moving point of view, *Algorithmica*, 11:360–378, 1994.
4. N. Coll, F. Hurtado and J.A. Sellarès., Approximating planar subdivisions and generalized Vornoi diagrams from random sections, *19th European Workshop on Computational Geometry*, 27–30, 2003.
5. N. Coll, M. Fort and J.A. Sellarès. Approximate Multi-Visibility Map Computation, *Proc. 21th European Workshop on Computational Geometry*, 97–100, 2005.
6. Q. Fan, A. Efrat, V. Koltun, S. Krishnan, S. Venkatasubramanian: Hardware-Assisted Natural Neighbor Interpolation. Proc. 7th Workshop on Algorithms Engineering and Experimentation. ALENEX 2005.
7. L. De Floriani, P. Magillo, Visibility Algorithms on Triangulated Digital Terrain Models, *International Journal of Geographic Information Systems*, 8(1):13–41, 1994.
8. L. De Floriani, E. Puppo, P. Magillo, Applications of Computational Geometry to Geographic Information Systems, *Handbook of Computational Geometry, J.R. Sack, J. Urrutia (Eds.)*, Elsevier Science., 333–388, 1999.
9. K. E. Hoff III, J. Keyser, M. C. Lin, D. Manocha, T. Culver: Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. SIGGRAPH, 277–286,1999.
10. V. Koltun, D. Cohen-Or, and Y. Chrysanthou, Hardware-Accelerated From-Region Visibility Using a Dual Ray Space, *Proc. 12th Eurographics Workshop on Rendering*, 204–214, 2001.
11. M.J. van Kreveld, Digital Elevation Models and TIN Algorithms, *Algorithmic Foundations of Geographic Information Systems, LNCS*, 1340::37–78, 1997.
12. M.J. van Kreveld, E. Moet, R. van Ostrum, Region inter-visibility in terrains, *Proc. 20th European Workshop on Computational Geometry*, 155–158, 2004.
13. S. Krishnan, N. H. Mustafa, S. Venkatasubramanian: Hardware-Assisted Computation of Depth Contours. *13th ACM-SIAM Symposium on Discrete Algorithms*, 558-567, 2002.
14. N. Mustafa, E. Koustsofios, S. Krishnan, S. Venkatasubramanian: Hardware Assisted View-Dependent Planar Map Simplification. *Proc. of the 17th ACM Symposium on Computational Geometry (SoCG '01)*, 2001.
15. M. Segal, K. Akeley, The OpenGL Graphics System: A Specification, http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf, 2004.
16. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn and T. J. Purcell: A Survey of General-Purpose Computation on Graphics Hardware, *Eurographics 2005, State of the Art Reports*, 21–51, 2005.
17. J.W. Jaromczyk, and M. Kowaluk, Skewed projections with an application to line stabbing in $\mathbb{R}^3$, In *Proc. 4th ACM Symp. on Comp. Geometry*, 362–370, 1988.