GPU-Assisted Z-Field Simplification

Alexander Bogomjakov Craig Gotsman

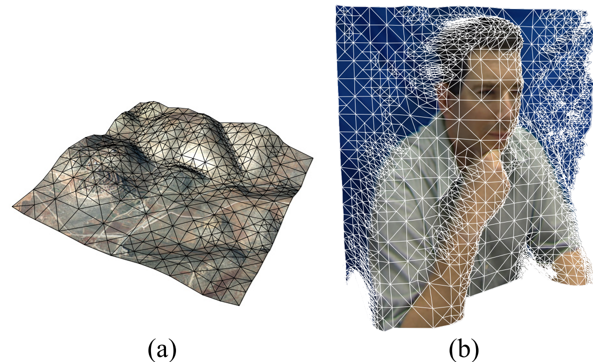
Center for Graphics and Geometric Computing, Computer Science Dept.
Technion – Israel Institute of Technology
Haifa 32000, Israel
{alexb|gotsman}@cs.technion.ac.il

Abstract

Height fields and depth maps which we collectively refer to as z-fields, usually carry a lot of redundant information and are often used in real-time applications. This is the reason why efficient methods for their simplification are necessary. On the other hand, the computation power and programmability of commodity graphics hardware has significantly grown. We present an adaptation of an existing real-time z-field simplification method for execution in graphics hardware. The main parts of the algorithm are implemented as fragment programs which run on the GPU. The resulting polygonal models are identical to the ones obtained by the original method. The main benefit is that the computation load is imposed on the GPU, freeing-up the CPU for other tasks. Additionally, the new method exhibits a performance improvement when compared to a pure CPU implementation.

1. Introduction

The performance of the graphics hardware increases at a rapid pace. The nature of the processed data allows for the computation to be organized according to the Single Instruction Multiple Data (SIMD) principle, which, among others, has the advantage of increased throughput via parallelized execution. Each of the input vertices undergoes exactly the same computation as all others. The independence of the computations allows further improvement of the performance by physical replication of the parallel execution units. Analogously to the vertices, this is also done for the frame buffer fragments. Today, high-performance GPUs provide developers with the possibility to utilize this ever-growing power not only quantitatively but also qualitatively. GPUs such as NVIDIA GeForce FX provide hardware support for the user to intervene in the rendering pipeline, replacing the fixed-function vertex and fragment processing with flexible shader programs. Instead of just using more and more polygons, the image quality and realism is significantly increased by using advanced rendering effects that now can be programmed for execution, in real time, on the geometry and the raster processing levels. As the programmability features extend the GPU functionality towards a general purpose CPU, it becomes more obvious that the graphics hardware can now be utilized to solve a wider class of problems, not necessarily related to rendering effects. In this work we describe how to use the programmable fragment engine to run a specific mesh simplification algorithm.



(a) (b)

Figure 1. *Simplified height- (a) and depth- (b) fields.*

The algorithm we explore was initially developed for real-time simplification of height fields. Later it was adopted for simplification of depth maps. Although coming from different sources, there is no essential difference between these two kinds of data. Both are represented by a regular rectangular grid with a height or a depth, i.e. *z*-value, at each grid point, relative to the *x-y* plane. Consequently, we use the generalized term *z-field*. Figure 1 shows examples of a height field and a depth field after simplification.

2. Related Work

The complexity of the polygonal representation of the *z*-field data can be reduced using a variety of geometric simplification methods. Most mesh simplification techniques [1,2,3] are not applicable to the problem of real-time resolution reduction of *z*-fields due to the relatively high computational cost of the optimization process that they use.

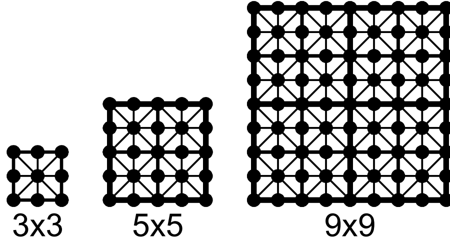


Figure 2. *Z-field triangulation without simplification.*

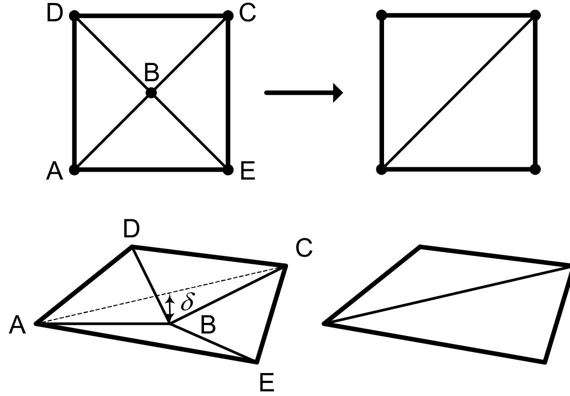


Figure 3. *Vertex removal. The triangles ABD and BCD are merged into one triangle. The same happens for the triangles AEB and BEC.*

Several techniques have been developed specifically to address the issue of fast simplification of height field data. Gross et al. [4] produce adaptive surface meshing using wavelet transforms. Lindstrom et al. [5] developed a real-time height field simplification algorithm. Their method uses quad-tree partitioning of the height field domain and constructs a level-of-detail hierarchy, defined through a specific type of vertex removal operation. With some modifications, Chai et al. [6], proposed using the Lindstrom simplification for depth map data, e.g. the output of a range-scanner. These come from real-time stereo reconstruction and must be processed as quickly as possible, preferably with as little computational effort as possible, at least on the CPU part.

Employing the graphics hardware, whenever possible, to assist in the computation is gaining popularity, even with the fixed-function graphics pipeline [7]. With the introduction of the programmable GPU [8] the applicability of this technique has been significantly extended. Purcell et al. [9] as well as Carr et al. [10] demonstrated the feasibility of ray-tracing with graphics hardware. The GPU has also been used for physical simulation [11], matrix multiplication [12] and even solving sparse linear systems [13]. The interested reader is referred to the web-site www.gpgpu.org, which is specifically dedicated to the general purpose computations using graphics hardware.

The technique presented here is based on the simplification method of Lindstrom et al. [5]. It generates the same output, but is adapted to run its main parts in the programmable graphics hardware. By doing that we are able to achieve some speed-up and relieve the CPU of most of the computation load. The latter may be important in a client-server configuration where maximal utilization of the available resources is important to provide the best performance. Another case is a view-dependent scenario, where the simplification *must* be done on the client. If the client's graphics subsystem is more powerful, it makes sense to take advantage of this in order to free-up additional CPU resources.

3. Z-Field Simplification

The surface triangulation of the *z*-field before the simplification is defined as follows. The smallest sub-mesh is defined to be a mesh over 3x3 vertices, and successively larger sub-meshes are formed from the smaller ones by arranging them into 2x2 arrays, as shown in Figure 2. Defined in this way, a sub-mesh of level l has the dimensions (2^l+1)x(2^l+1) and is the highest resolution *block* that can be defined over these vertices. A block is a discrete level of detail sub-mesh, defined over a square region of vertices. Discarding every other row and column of four higher resolution blocks creates one block at the next resolution. The triangulation before the simplification consists of the highest resolution blocks. The discrete level of detail block hierarchy corresponds to a quad-tree structure.

The simplification is performed in two phases. At the coarse-grained phase, entire blocks are considered, and the blocks of the lowest possible resolution, whose maximum error does not yet exceed a certain threshold, are chosen to represent the mesh. At the fine-grained phase, individual vertices are considered for removal. When a vertex is removed, two adjacent triangles, sharing this vertex at the same resolution level are merged into one triangle. The resulting triangle and its co-triangle are then considered for further simplification (Figure 3). The view-independent simplification error at the removed vertex is defined as:

$$\delta_B = \left| B_z - \frac{A_z + C_z}{2} \right|$$

The view-dependent version is the error projected to the viewer image plane.

To ensure validity of the resulting triangulation, the removal process is ruled by the vertex dependencies, induced by the parent-child relations (Figure 4).

The removed vertex, unless it is on the boundary, always has exactly four neighbors, of which two are its parents and two its non-parent neighbors. If the vertex has a degree greater than four, some of its children are

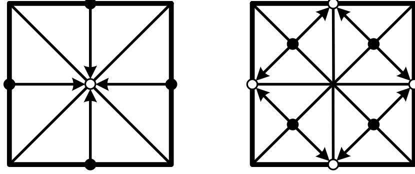


Figure 4. *The two kinds of the parent-child relationship. The arrows point from children to parents.*

present and hence the vertex cannot be removed. After removal of the vertex, the resulting quadrilateral is triangulated by connecting the two non-parent vertex neighbors, which can be viewed as two pair-wise merges of the four triangles containing the vertex.

Two attributes are associated with every vertex: *enabled* and *activated*. A vertex is enabled if it is activated or any of its children is enabled. A vertex is activated if its simplification error exceeds the threshold. During the fine-grained simplification, every vertex is evaluated to determine whether it is enabled or not. It is for acceleration of this process that we propose using the programmable graphics hardware.

After the correct *enabled* states of all the vertices are determined, the final construction of the mesh triangles can be easily performed by a DFS traversal of the vertex dependency tree [5, 6].

4. Modified Algorithm

According to the algorithm of Lindstrom et al [5], when a vertex is evaluated, if its *enabled* state is changed, a notification is sent to both of its parents. A notified vertex checks whether its *enabled* state should be altered, and if it does, sends, in turn, a notification to both of its parents, creating a recursion. If none of the children of the notified vertex is enabled, its state is dictated by its *activated* attribute. Computation of the correct value of the *activated* attribute is relatively expensive. For this reason, in the original formulation it is not performed during the notification, rather only during the vertex evaluation if none of its children is enabled. This allows avoiding unnecessary computations but can result in a one-frame delay before the *activated* attribute is corrected. The delay is possible if at least one of the vertex children is evaluated after the vertex itself. Consider the case where vertex u is the only enabled child of vertex v. Suppose that v is evaluated before u. For the current frame, *activated*(v) will not be computed because one of its children is enabled. Suppose that after the evaluation u became disabled. Now *activated*(v) is important, but it will not be computed until next frame. For similar reasons, during the evaluation for the same frame, notifications may travel the same route more than once.

A key observation is that these events may be avoided by arranging the vertices considered for the simplification in such a way that children are always evaluated before their parents. Thus, when a vertex is considered, all necessary information about its children is available, up-to-date and no notification is necessary. This can be achieved by grouping the vertices according to their level in the simplification hierarchy, starting with the leaves at level 0, which are the vertices whose parents are their immediate neighbors above and below or on the left and on the right. For a grid of n = (2^k+1)x(2^k+1) vertices there are $\lfloor n/2 \rfloor$ leaves and $2k+3$ levels. The highest level consists of a single vertex. All vertices in the same level are independent and can be processed in parallel. The simplification process proceeds level by level, starting at level 0. With the given viewing parameters and the error threshold, the correct simplification of the z-field for the current frame is achieved after a single sweep of all levels.

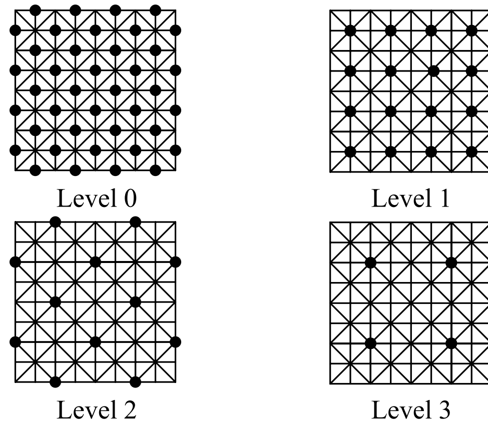


Figure 5. *Vertex levels.*

5. Simplification in Graphics Hardware

The fact that the processing of the vertices of the same level can be done independently and in parallel and the fact that the problem domain naturally corresponds to the fragments of the frame buffer allows natural mapping of our solution onto programmable graphics hardware. Each vertex of the z-field is associated with a fragment of the frame buffer. This way the z-field can be viewed as a grayscale image where each pixel represents a z-value. The unsimplified input data is thus represented by a 2D texture. The output of the algorithm is written into the color buffer as a binary mask, specifying for each z-field vertex, associated with the corresponding pixel, its *enabled* state. Essentially, the output is a 1-bit (monochrome) image.

The algorithm works in a multi-pass fashion. Every five passes compute the *enabled* state for all vertices of one particular level. The first level is an exception and

is computed in a single pass. The level to be processed is chosen with the help of a special frame arrangement. The effect of individual vertices on the final result is controlled with the depth test mechanism.

5.1. Frame Arrangement

In order to avoid the unnecessary rasterization of vertices belonging to levels other than the current, we rearrange the vertices inside the frame. Instead of the initial interleaved arrangement, the vertices of each level are grouped as illustrated on the Figure 6. As a result, each level is covered by a single continuous rectangle, which does not cover any vertices of other levels. Thus, in order to process a given level, it suffices to render a quad covering the corresponding vertices. Only vertices of the current level will be rasterized. The arrangement mapping is precomputed and stored in a 2D texture which is used for computing the correct texture coordinates when accessing the *z*-field texture, see Figure 7(b).

During the computation of the arrangement, it is necessary to know, for each vertex, which level it belongs to. This can be precomputed by a simple algorithm, similar to the notification mechanism described above. Starting with the leaves at level 0, the vertex level is marked, and then the procedure is called recursively for the parents of the marked vertices, with the level number incremented by one.

5.2. Rendering Passes

The computation is independent and identical for every vertex and thus is particularly suitable for the SIMD approach, which is implemented very efficiently in the latest graphics hardware. However, this computational model imposes a number of constraints, one of which is linear control flow. Currently, conditional execution, even if supported, imposes a significant overhead. In order to eliminate conditional code we had to split the computation into several passes. A depth test is used for selection of the vertices that will participate in a pass. The total number of passes is less than five times the number of levels and is logarithmic in the *z*-field resolution.

In the first four passes, for each vertex the *enabled* flags of its children are tested; these flags were computed at the previous level. During any of these passes, if an enabled child of a vertex is detected, the vertex is marked as enabled and its fragment depth is modified, so that it will not participate in the subsequent passes. Only the vertices with all children disabled make it to the fifth pass of the current level. In this pass the delta for each vertex is computed and compared against the threshold. If this test fails, the vertex becomes disabled, otherwise it is enabled.

This core computation on the *z*-field vertices is performed in two fragment programs. Processing of one level consists of the following steps.

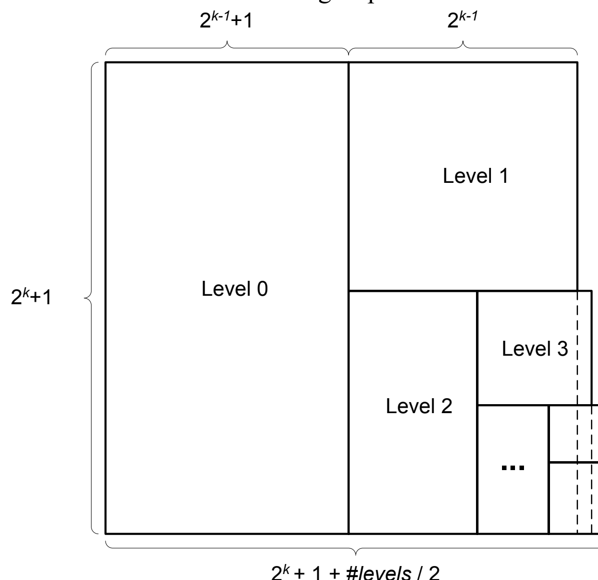


Figure 6. *Level arrangement in the frame.*

1. Prepare the *enabled* flags, computed at the previous level, for read access at the current level.
2. Perform the four children checking passes using the first fragment program. At each pass, another child of every vertex is tested.
3. Perform the delta test pass, executing the second fragment program.

Step 1 is not performed for the first level because it does not use any previously computed flags. For any other level it amounts to copying the color buffer rectangle, corresponding to the previous level, into a dedicated 2D texture which serves as a read-only memory for the fragment program. This relatively expensive pixel copying operation is needed because, currently, in the graphics hardware there is no such memory unit which is accessible both for reading and writing at the same time by the fragment program. Precisely for this reason the existing “render texture” OpenGL extensions do not seem to be directly applicable here (see the Discussion section). However, due to the arrangement of the levels in the frame, no redundant copying is performed and each pixel is copied only once.

A pass is performed by rendering a single quad which covers vertices of the corresponding level. After the quad is rasterized, the selected fragment program is executed for each fragment of the rasterization. For efficiency, the coordinates of each of the four children are precomputed and stored in four different 2D textures. The two 16-bit coordinate values are packed into

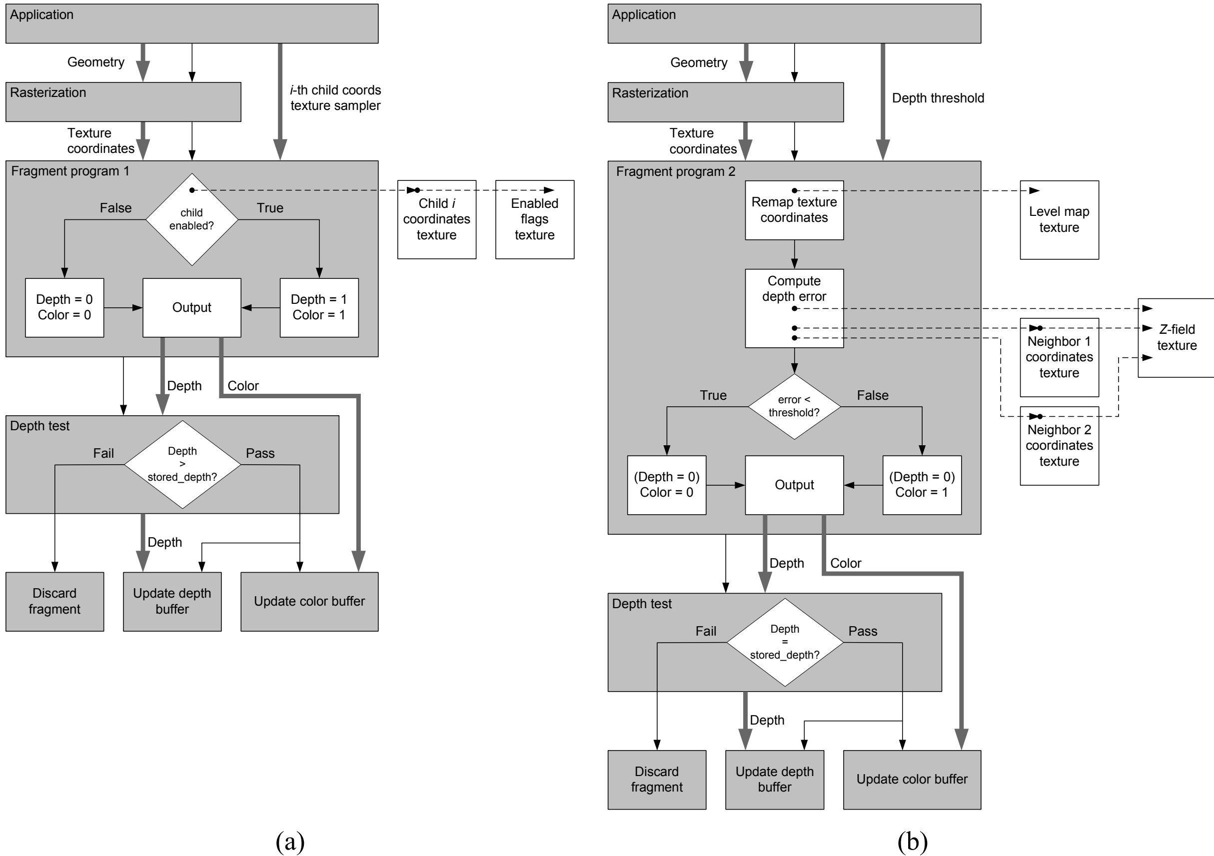


Figure 7. *Rendering pass execution diagrams. Solid lines denote control flow, fat dark-gray lines denote data paths, dashed lines – texture accesses. (a) The first fragment program – first four passes; i runs from 1 to 4. (b) The second fragment program – the fifth pass.*

a 4-byte RGBA texture word. The first program reads child coordinates from the corresponding texture and uses them to access the texture of the *enabled* flags. The second program uses the same scheme for accessing the precomputed neighbor coordinates. In order to mark a vertex as enabled, the fragment programs issue the color value of 1.

To prevent participation, in the subsequent passes, of a vertex for whom an enabled child is detected, the first fragment program issues the depth value of 1. For both programs the default fragment depth is zero, which is also the clearing value for the depth buffer. The depth test for the first program will succeed only if the incoming fragment depth is greater than that stored. For the second program the depth test succeeds if it is equal.

Figure 7 summarizes the execution of the two types of rendering passes.

Each vertex of the output *z*-field corresponds to exactly one output pixel. Therefore, the resolution of the frame buffer and all the textures must be sufficient to represent the finest underlying resolution of the *simplified* *z*-field. This resolution, however, does not necessarily have to be the resolution of the *input* *z*-field. In fact, just like any discrete image, the input *z*-field can be filtered and resampled, and the graphics hardware provides efficient means, e.g. mipmaps, for doing that.

5.3. Meshing

The final meshing is done on the CPU. After the *enabled* flags are computed on the GPU they need to be read from the frame buffer into the main system memory. To minimize the time that is spent on the expensive pixel-read operation, we pack the *enabled* bits, in sets of eight, into bytes. These bytes are stored in one of the four color channels of the color buffer. This way we only require one eighth of the total number of pixels and for each pixel we only need to read a singe one-byte channel.

The packing is performed in one additional pass, after all the computation passes. In this pass the third fragment program is executed. Its function for each fragment amounts to reading the corresponding consecutive eight *enabled* flags, starting at an *x* coordinate which is a multiple of 8, and packing them together into one channel. The rendered rectangle spans the entire

Z-field data-set	Initial resolution	Retained vertices	Simplification time (sec)	
			CPU	GPU
seliun	257x257	511	0.0088	0.0066
seliun	257x257	1223	0.0093	0.0076
elevh	257x257	821	0.0087	0.0064
elevh	257x257	2035	0.0093	0.0070
k.baram	513x513	641	0.0333	0.0215
k.baram	513x513	1113	0.0337	0.0218
k.baram	513x513	5172	0.0356	0.0250
shomeron	513x513	1024	0.0333	0.0220
shomeron	513x513	4177	0.0355	0.0243
c1AvgHand	513x513	6153	0.0364	0.0249

Table 1. *Simplification runtimes. The CPU column refers to the pure CPU implementation of the algorithm.*

height and one eighth of the width of the frame, although the entire *enabled* flags texture is sampled.

The meshing procedure is exactly the same as in the CPU counterpart, but with a slight overhead from remapping and unpacking when reading the values of the *enabled* flags.

6. Results

We implemented the technique described above using OpenGL and Cg – a high level programming language for graphics. The code runs in hardware on graphics cards based on the GeForce FX or any other GPU whose driver supports the “NV_fragment_program2” OpenGL extension. Table 1 summarizes the runtime results. The CPU used in our tests is a 2.8GHz Xeon with 1GB RAM. The GPU used is a 400MHz GeForce FX 5900 with 128MB RAM.

The runtimes refer to the complete simplification process, which for the GPU-assisted case includes texture upload, execution of the fragment programs, frame buffer reading and meshing.

The runtime seems to be dependent almost entirely on the resolution of the input *z*-field, although we have some times experienced a very slight decrease in speed with an increase in the output mesh resolution.

The *z*-field texture upload consumes about 16% of the total simplification time, which is dominated by the execution of the fragment programs, see Table 2. This texture upload time could be reduced by using different extensions, e.g. “compressed textures”. Many redundant fragment program executions could be avoided by early fragment rejection from depth or stencil tests, if supported and correctly employed. Although our implementation still leaves a lot of room for further optimization, the GPU already outperforms the CPU in the *z*-field simplification.

Component	Share
Fragment programs execution	62%
Z-texture upload	16%
Meshing	14%
Texture bind & copy	3%
Frame buffer read	3%
Cg parameter manipulations	2%
Total	100%

Table 2. *Time distribution between the components of the GPU-assisted simplification for a typical run.*

7. Discussion

Our initial Cg implementation contained a single fragment program and executed only one rendering pass per simplification level. However, this program included a great deal of conditional code, which affected the performance significantly. The reason for this is that the GPUs do not, yet, support conditional execution at the instruction level. Although Cg allows conditional statements, it is implemented by executing all the conditional code and then conditionally assigning the result. Elimination of all of the conditional code led to splitting the main program into two and multiplying the number of passes by a factor of 5. We believe that this significant increase in the number of passes, will be avoided in the future when GPUs with a more advanced instruction set are available.

A possible alternative to the five passes per level is a two-pass approach, where the four children testing passes are unified into one. We could determine whether a vertex has enabled children by testing the sum of their *enabled* bit values. While not requiring any conditional code, this approach would perform many redundant texture accesses, which could not be avoided even by employing early fragment rejection. In our experiments, without early fragment discards, the four-pass and the one-pass children testing performed similarly.

The time cost of the pixel copying is quite low, less than one millisecond. Still, it could be eliminated almost completely by using a double p-buffer. A P-buffer is an OpenGL extension for accelerated off-screen rendering. When processing one level, one p-buffer could be written to, while the other could be read from, by selecting it as a texture. For the next level the two p-buffers would be swapped.

In the context of our work, we were particularly interested in simplification of depth maps that can change at every frame and have limited resolution. However, our method can be used for simplification of large height fields as the fine-grained step, complementary to the coarse-grained block-based simplification.

Acknowledgements

Some of the depth map data was provided by 3DV Systems Ltd. This work was partially supported by

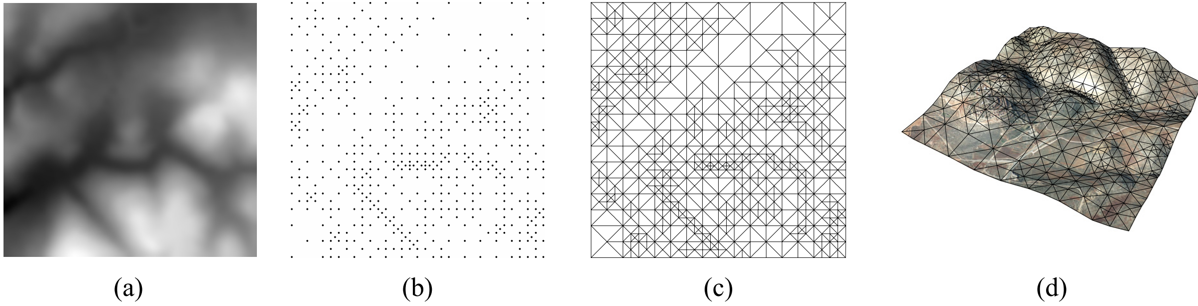


Figure 8. *(a) A height field. (b) The enabled mask, computed on the GPU. (c) and (d) The resulting triangulation.*

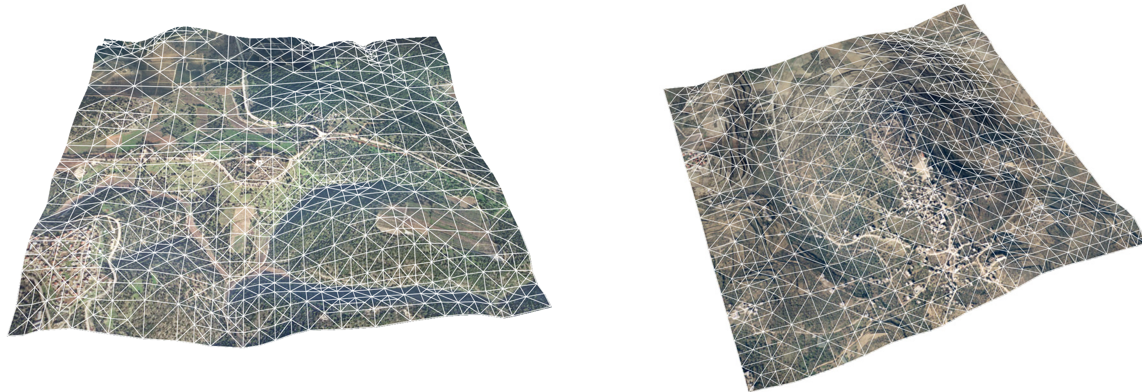


Figure 9. *Simplified height fields.*

Israel Ministry of Science grant 01-01-01509, German-Israel Fund (GIF) Grant I-627-45.6/1999, and European FP6 NoE grant 506766 (AIM@SHAPE).

References

[1] Hoppe, H., Progressive meshes. In *Proceedings of SIGGRAPH '96*, pages 99-108, 1996.

[2] Schroeder, W. J., Zarge, J. A. and Lorenson, W. E., Decimation of triangle meshes. In *Proceedings of SIGGRAPH '92,* Computer Graphics 26(2), pages 65-70, July 1992.

[3] Kobbelt, L., Campagna, S. and Seidel, H.-P., A General Framework for Mesh Decimation. In *Graphics Interface '98*, pages 43-50, 1998.

[4] Gross, M., Gatti, R. and Staadt, O., Fast multiresolution surface meshing. In *Proceedings of Visualization '95*, pages 135-142, October 1995.

[5] Lindstrom P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N. and Turner, G., Real-Time Continuous level of detail rendering of height fields. In *Proceedings of SIGGRAPH '96,* Computer Graphics, pages 109-118, October 1996.

[6] Chai, B.-B., Sethuraman, S. and Swahney, H. S., A depth map representation for real-time transmission and view-based rendering of a dynamic 3D scene. In *1st International Symposium on 3D Data Processing Visualization and Transmission*, pages 107-114, 2002.

[7] Hoff, K. E., Keyser, J., Lin, M., Manocha, D. and Culver, T., Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH '99*, pages 277-286, 1999.

[8] Lindholm, E., Kilgard, M. J. and Moreton, H., A user-programmable vertex engine. In *Proceedings of SIGGRAPH '01*, pages 149-158, 2001.

[9] Purcell, T. J., Buck, I., Mark, W. R. and Hanrahan, P., Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH '02*, pages 703-712, 2002.

[10] Carr, N. A., Hall, J. D. and Hart, J. C., The Ray Engine. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2002.

[11] Doug, L. J. and Dinesh, K. P., DyRT: Dynamic tesponse textures for real time deformation simulation with graphics hardware. In *ACM Transactions on Graphics*, 21(3):58285, 2002.

[12] Larsen, E. S. and McAllister, D. K., Fast matrix multiplies using graphics hardware. In *Supercomputing*, 2001.

[13] Bolz, J., Farmer, I., Grinspun, E. and Schröder, P., Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proceeding of SIGGRAPH '03*, 2003.