Computers & Graphics 34 (2010) 125–135

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

Parallel GPU-based data-dependent triangulations

Michal Červeňanský [a,*], Zsolt Tóth [a], Juraj Starinský [a], Andrej Ferko [a], Miloš Šrámek [b]

[a] Faculty of Mathematics, Physics and Informatics, Comenius University, Slovakia
[b] Austrian Academy of Sciences, Vienna, Austria

ARTICLE INFO

Article history:
Received 22 April 2009
Received in revised form
30 December 2009
Accepted 8 January 2010

Keywords:
Data-dependent triangulation
Image reconstruction
Graphics hardware
GPGPU

ABSTRACT

In this paper we introduce a new technique for data-dependent triangulation which is suitable for implementation on a GPU. Our solution is based on a new parallel version of the well known Lawson's optimization process and is fully compatible with restrictions of the GPU hardware. We test and compare the quality of our solution in an image reconstruction problem. In comparison with the standard implementations we achieve significant speed-up (eight times on average) with comparable quality of the reconstructed image. Further, several other improvements and optimizations are introduced and tested, and the results are discussed in detail.

1. Introduction

Parallelization of various computationally expensive problems is today made possible by the architecture of mainstream processors. Simultaneously, generalization of the GPU architectures to non-graphic applications results in their wide usage in non-standard areas, which benefit from the inherent parallelism of GPUs. This observation leads us to the idea of accelerating computation of optimal triangulations by such a GPU implementation. Nowadays, generation of optimal triangulations is done mainly on the CPU, and, depending on the type of the required property, the optimality can be achieved in a very time-consuming optimization process.

Optimal triangulations are widely used in different branches of science and technology. We focus our attention on the generation of locally optimal meshes by iterative improvement (optimization), which can be used in various geometrically defined problems, as, for example, in finite element simulations and image reconstruction. Specifically, we selected the image reconstruction (namely, edge preserving magnification) problem, which can be solved by a special case of optimal mesh, called *data-dependent triangulation* (DDT) [8]. The main advantage of this technique resides in its ability to fit the mesh structure to the underlying data. We do not generate the triangulation, but only optimize it. As an input we require an arbitrary triangular mesh and with the help of special cost functions and topological operations we generate an optimal one from it. With different choices of these functions we can obtain different properties of the resulting mesh.

Image reconstruction is only one of the application areas of data-dependent triangulations. The possibility of visual representation of results was the reason for our selection. In comparison with the convolution based techniques, the DDT based approaches produce visually more pleasant results in high-frequency areas. As an example, see the blocky artifacts in the edge areas in Fig. 1(a), obtained by convolution, compared with results obtained by non-convolution approaches in Fig. 1(b) and (c). For creation of data-dependent meshes we choose an approach called Lawson's optimization process. Effective implementation of this technique on a GPU requires its parallelization. The processing pipeline of a graphics card, however, is not directly suitable for representation and maintenance of the data structures usually used in this type of computations. Therefore, the main challenge was to avoid the hardware restrictions and to design a fully parallel approach implementable on a GPU.

The main contribution of this paper is the solution of the above-mentioned problem. We introduce a parallel version of Lawson's optimization process which is completely implementable on a GPU. We further present several optimizations and improvements of the basic parallel version. In Fig. 1(c) we can see that our approach gives visually similar results to those of the original CPU-based approach Fig. 1(b), but its run time on a GPU is about eight times shorter. While the presented results are oriented to the image reconstruction problem, the technique is nearly general and can be applied to any DDT related task and to arbitrary data distribution.

This paper is structured as follows. In Section 2 we briefly survey previous work on data-dependent triangulations and on

* Corresponding author.
E-mail address: cervenansky@sccg.sk (M. Červeňanský).

0097-8493/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2010.01.001

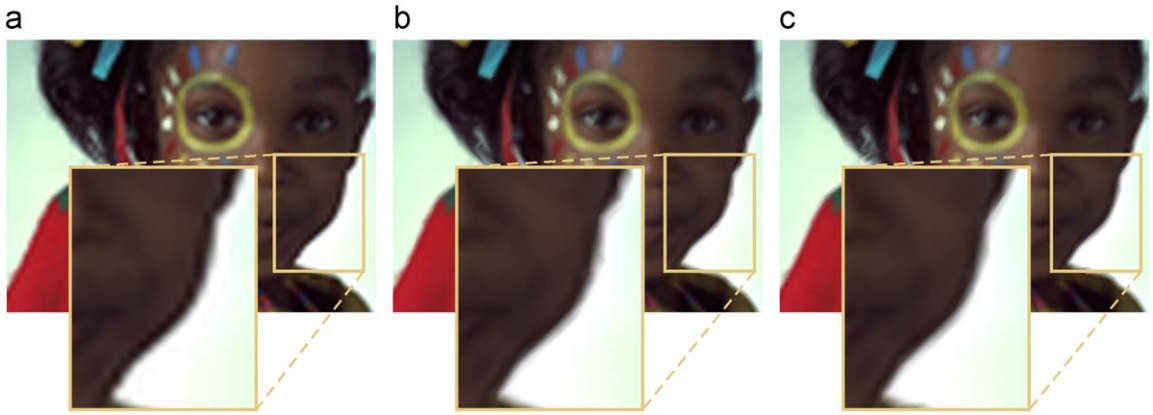

Fig. 1. Image magnification at 800 percent using: (a) lanczos filter; (b) a CPU-based data-dependent triangulation and (c) our GPU-based technique. Results (b) and (c) show improved reconstruction in the high-frequency areas.

GPU usage in this field. Section 3 introduces the basic concepts of parallel data-dependent triangulation. Section 4 deals with GPU-specific implementation details, and presents our data structures. In Section 5 we present additional modifications and improvements of the original approach. Section 6 analyzes the results and, finally, in Section 7 we draw conclusions from the results and outline future work possibilities.

2. Related work

Digital images are 2D arrays of pixels, which are represented by a pair (x,y) of Cartesian coordinates associated with a color function value $f(x,y)$. The goal of reconstruction is to provide the means to evaluate this function at an arbitrary point of the domain. The main aim is to satisfy the human visual system, which is a very subjective and generally complex task, since results with feature preservation and without significant artifacts are expected. In other words, correct reconstruction of high-frequency areas (edges), which play an important role in quality decision by human observers, is required.

The most commonly used techniques are the convolution based methods known from image processing. The *sinc* function is the ideal reconstruction filter, but it is not suitable for practical use because of its infinite width [13]. In practice, this results in usage of different, size-limited convolution filters, which, however, may introduce artifacts as jaggies, blurring, ringing, etc. The computational complexity of these approaches is dependent on the size of the reconstruction kernel. In general, convolution techniques can be considered as computationally cheap operations compared with other techniques. Efficient implementations of interpolation by means of various convolution kernels on the GPU were described e.g. by Hadwiger et al. [14] and Bjorke [5].

A completely different geometrically based reconstruction approach, the *data-dependent triangulation*, was introduced by Dyn et al. [8]. It fits the processed data values with a triangle mesh, thus resulting in piecewise linear interpolation. Contrary to other mesh generation methods, DDT aligns edges to the underlying data and organizes the triangular structure into a feature-preserving mesh. Here, the quality of the resulting triangulation is defined through a special *cost function* in an optimization process. DDT was applied to image reconstruction by Yu et al. [35], while improving the image reconstruction quality was the topic of our previous work [32]. Battiato et al. [4] used the concept of DDT to produce vector images from raster data.

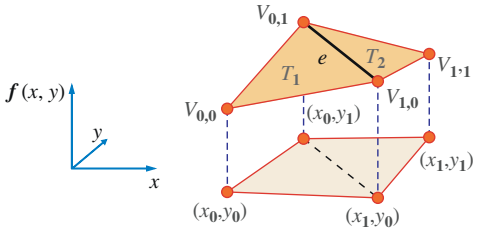

Fig. 2. Illustration of geometric dependencies in 2D data-dependent triangulation.

2.1. Data-dependent triangulation

The DDT approach is less well known than the convolution techniques. Therefore, we introduce first the necessary definitions and basic facts about the technique.

Let a set of distinct (and not all collinear) points $\mathbf{V} = \{V_{i,j} = (x_i, y_j); i,j = 1,\ldots,n\}$ is given in E^2 (Euclidean two-dimensional space). Triangulation of $\mathbf{V}$ is a set of triangles $T(\mathbf{V})$ which satisfy the following properties:

- each vertex of an arbitrary triangle from $T(\mathbf{V})$ is from $\mathbf{V}$ and each element from $\mathbf{V}$ is a vertex of at least one triangle of $T(\mathbf{V})$,
- each edge from the triangulation contains exactly two elements from $\mathbf{V}$,
- the union of all triangles from $T(\mathbf{V})$ is the convex hull of the set $\mathbf{V}$, and
- an intersection of two arbitrary triangles from $T(\mathbf{V})$ is an empty set or their common vertex or their common edge.

Data-dependent triangulation of the above-mentioned set $\mathbf{V}$ is a triangulation whose topology is dependent on a function $f(x,y)$ with

$$z_{i,j} = f(x_i, y_j), \quad i,j = 1,\ldots,n,$$

where $z_{i,j}$ are data values, for example, luminance values converted from RGB color components by standard conversion.

We restrict our considerations to edge based DDTs, where we assign cost exclusively to edges of the triangulation. There are also other possibilities, for example, vertex based DDTs [7]. The relation between the edges and the function $f(x,y)$ can be arbitrary, but we limit our consideration to the following. Each interior edge is dependent on the four vertices forming a quadrilateral (created from two adjacent triangles), which contains the edge as a diagonal. The situation is illustrated in Fig. 2,

where T_1 and T_2 are triangles on a piecewise linear surface, the common edge is denoted as e and $V_{i,j}=f(x_i,y_j)$, $i,j\in\{0,1\}$. The linear polynomials $P_1(x,y)$ and $P_2(x,y)$ define planes which contain the triangles T_1 and T_2:

$$P_1(x,y)=a_1x+b_1y+c_1,$$

$$P_2(x,y)=a_2x+b_2y+c_2.$$

These polynomials can be used to define several geometrically based cost functions. Sederberg's cost function (SCF) depends on four vertices and uses projection of the T_1 and T_2 triangle normals into the plane xy:

$$c^{SCF}(e)=\|\nabla P_1\|\cdot\|\nabla P_2\|\text{-}\nabla P_1\cdot\nabla P_2,$$

where

$$\nabla P_i=(a_i,b_i),\quad i=\{1,2\}$$

are the gradients of $P_i(x,y)$. Their magnitude is defined by

$$\|\nabla P_i\|=(a_i^2+b_i^2)^{1/2},\quad i=\{1,2\}.$$

Several other approaches exist, which assign cost to components of the triangulation (vertices, edges, triangles, etc.) according to various geometrically and non-geometrically based criteria [1,7,8].

Generation of DDTs involves a mesh optimization process. The goal is approximation of a minimum weight triangulation (MWT), which is a triangulation of the set $\mathbf{V}$ with the minimal cost among all possible triangulations:

$$c(MWT(\mathbf{V}))=\sum_{e\in MWT(\mathbf{V})}\|c(e)\|=\min\{c(T(\mathbf{V}))\},\quad \forall T(\mathbf{V}),$$

where $c(T(\mathbf{V}))$ marks the cost of the triangulation. NP-hardness of the MWT problem was proved by Mulzer and Rote [25]. In DDT, however, the number of the existing heuristics and approximations [9] of this problem for planar triangulations is reduced owing to the existence of the edge cost function.

Most of the existing optimization approaches rely on a common topological operation called *edge flip*. Every interior edge is an intersection of two adjacent triangles, which form a quadrilateral. If the quadrilateral is convex and non-degenerate, then the edge can be replaced by the other diagonal of the quadrilateral. This operation is the edge flip. The diagonal edge of the quadrilateral with the mentioned properties is called locally optimal if the quadrilateral is optimally (MWT) triangulated. If it is concave or degenerate then it is also locally optimal. We note that local optimality of the edge depends on the sum of SCF costs of five edges (the edge plus four edges of the quadrilateral). Thus, in its evaluation, information from 12 vertices is processed (Fig. 2).

The most popular technique for obtaining optimal DDTs is Lawson's optimization [8] (also called local optimization) which works with the edge flipping operation. It processes the edges in iterations and proves their local optimality. The pseudo-code of this method is presented in Fig. 3. It is similar to the Delaunay triangulation [3] for the planar case and, if the cost function satisfies the well known empty circle property [3], then they are identical.

An enhancement of this technique, the look-ahead approach, was presented by Yu et al. [35]. A computationally cheap modification, the pixel level DDT, was introduced by Su and Willis [31]. In this very fast triangulation, however, only slightly better results than in the standard bilinear or bicubic interpolations are obtained. Apart from the aforementioned deterministic methods, stochastic optimization techniques were also used in DDT heuristics of MWT. A simulated annealing approach was described by Schumaker in [30]. Its application to image compression was presented by Kreylos et al. [23]. Genetic optimization is another stochastic process, introduced to DDT by Kolingerová [21]. Recent results can be found in [22]. Theoretical results dealing with the number of simultaneously flippable edges in triangulations were published in [6].

GPU-based approximation of the Voronoi diagram in discrete space was published in [16,11,27]. The work by Rong et al. [28] deals with generation of the Delaunay triangulation from the discrete Voronoi diagram. They use a combination of GPU and CPU for the computation of the Delaunay triangulation in the continuous space.

To the best of our knowledge, no work on DDT implementation in the GPU has been published yet.

3. Parallel data-dependent triangulation

In this section we present a parallel version of the data-dependent triangulation approach based on Lawson's optimization process introduced in Section 2. We take into account the possibilities and restrictions of the GPU during the algorithm design. Technical and implementation details are discussed in Section 4.

First, we need to define some terminological notions. Under *neighbors* of an edge e in triangulation $T(\mathbf{V})$ we mean a set of such edges from $T(\mathbf{V})$, which create with the edge e a triangle. The set of neighbors of edge e is denoted as region of degree 1 (1-*region*). Following this notion we define the *n-region* ($n>1$) of an edge e in $T(\mathbf{V})$ as a union of:

- edges from the (n-1)- region of e and
- neighbors of edges from the (n-1)- region of e.

Local optimality of an edge e in DDT triangulations depends on the selection of the cost function. Therefore, we define region-of-influence (ROI) of e as a set of such edges, the change (flipping) of which affects the local optimality of e. For the SCF cost function introduced earlier this region is of degree 2, and contains 12 edges—see Fig. 4(a).

Edges should be in a parallel version of the triangulation algorithm processed concurrently, from which some major restrictions follow. We propose such an iterative algorithm, in which every iteration consists of three steps: *creation of candidates*, *acceptance and rejection of candidates* and *edge flipping*.

```
1  make an arbitrary triangulation
2  repeat
3    for (each edge e from triangulation)
4     if (e is not locally optimal)
5       flip (e)
6     endif
7    endfor
8  until the locally optimal triangulation is calculated
```

Fig. 3. Pseudo-code of Lawson's optimization process.

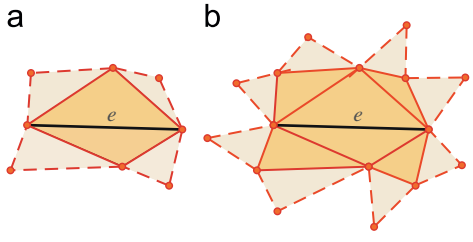

Fig. 4. (a) Region of degree 2 and (b) region of degree 3 for the edge e.

3.1. Creation of candidates

In the first step of each iteration we evaluate the cost function for all edges of the triangulation and classify them in two sets. The first set consists of locally non-optimal *candidate* edges, flipping of which results in decrease of the cost value for the whole triangulation. The rest of the edges will not be flipped in the current iteration of the algorithm. As cost evaluations are not influencing each other, they can be executed concurrently on all edges.

3.2. Acceptance and rejection of candidates

The local optimality of an edge e is influenced by all topological changes (edge flippings) in its ROI. In Lawson's optimization process the edge flippings are done sequentially and therefore it is not necessary to take this fact into account. The situation is different, however, in parallel edge processing.

Consider that we have two candidate edges e and f, where f is in the ROI of e and vice versa. Once we flip one of these, the local optimality of the other edge also may be changed, i.e. a conflict occurs (Fig. 5). In order to avoid these conflicts in the algorithm, we have to select from the candidate set a subset of independent edges, the ROIs of which do not intersect and which can thus be flipped concurrently.

In other words, we split the set of candidates into two disjunctive sets of

- *rejected* edges, which will not be flipped, and
- *accepted* edges, which can be flipped concurrently.

The edge classification is based on comparison by means of edge identification numbers (*ID* s). Each edge has its own, unique *ID*, which is assigned to it at the beginning of the algorithm. If an edge is flipped, it keeps the original *ID*.

We run the classification process on all edges from the candidate set. For each candidate edge e it is necessary to test all edges in its ROI. If at least one accepted edge in the ROI exists, the edge e is rejected. If there are no accepted edges in such ROI, then, if its *ID* has minimal value among all of the candidate edges in its ROI, the actual edge is added to the set of accepted edges and removed from the candidate set. If the *ID* of the edge e is not minimal then this edge e remains as a candidate and will be processed in the next iteration of Step 2 of the algorithm.

This process is iterative and continues as long as there are edges in the candidate set. The set of candidate edges is a finite set. In each iteration step of this process it is possible to find the edge with the minimal *ID*. It is obvious that at least one such edge is either accepted or rejected in every iteration, and therefore the number of edges in the candidate set is decreased at least by one in each iteration step. Hence, this iterative process is always finite.

The described process can be adjusted to the restrictions of the GPU, as we will describe in Section 4. We note that it is possible to consider different, more sophisticated techniques, if the CPU usage is allowed, but that is not our province.

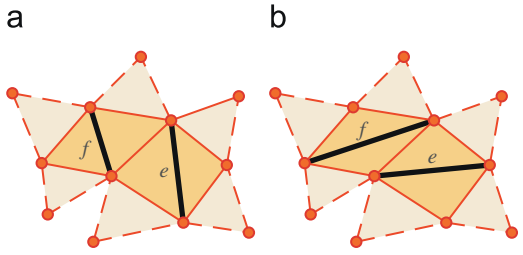

Fig. 5. Conflicting edges e and f before and after parallel edge flipping (in case of e, edge f should be unaffected and vice versa).

```
1  repeat
2    AcceptedSet.clear()
3    CandidateSet.clear()
   // – Step 1 –
4    for ( each edge e from triangulation )
5      if ( e is not locally optimal )
6        CandidateSet.add(e)
7      endif
8    endfor
   // – Step 2 –
9    while ( exists edge e from CandidateSet )
10     if ( exists accepted edge f in the ROI of e )
11       CandidateSet.remove(e)     // reject e
12     else
13       if ( e.ID < min ( ID of all candidate edges in ROI of e ) )
14         AcceptedSet.add(e)     // accept e
15         CandidateSet.remove(e)
16       else
17             // e remains candidate, unchanged
18       endif
19     endif
20   endwhile
   // – Step 3 –
21   for ( each edge e )
22     if ( e from AcceptedSet )
23       flip(e)
24       update data structure(e)
25     else
26       if ( exists accepted edge in 1-region of e ) // neighbor
27         update data structure (e)
28       endif
29     endif
30   endfor
31 until the locally optimal triangulation is calculated
```

Fig. 6. Pseudo-code of the GPU-friendly parallel Lawson's optimization algorithm.

3.3. Edge flipping

In the last step of the algorithm we flip in parallel all edges from the set of accepted edges created in the previous step of the algorithm. We also update relevant data structures of the edge being processed, as well as of all edges from its ROI.

The whole iterative process is then repeated until a locally optimal mesh is obtained. The pseudo-code of the algorithm is detailed in Fig. 6.

4. GPU implementation

In the previous section we proposed an algorithm for parallel data-dependent triangulation. It has been implemented for execution in a fragment shader, similar to most of the GPGPU (*general-purpose computation on GPUs* [24]) computations. Owing to this, it is possible to execute this algorithm also on older programmable graphics accelerators.

One of the main problems in implementation was to map the triangulation algorithm to the fragments. The most flexible input data structure for a fragment program is a texture, which can hold up to four values in its elements—*texels*. Each texel is represented by one RGB color channel and a transparency channel A. GPUs can read from multiple textures and can access arbitrary number of texels during fragment computations. The main restrictions are in writing (output), where only a constant number of values can be written to one framebuffer and the number of these framebuffers is also limited. Moreover, writing is restricted to a fixed position defined beforehand, and this position is the same for all output framebuffers.

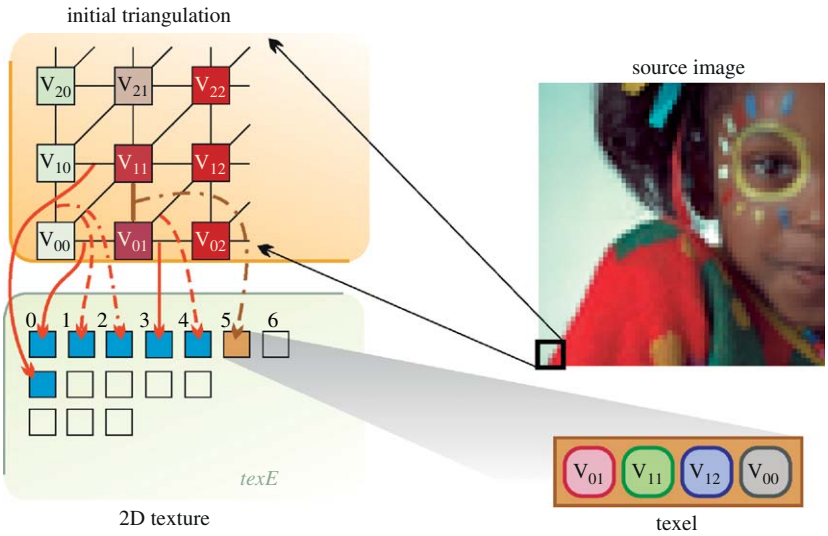

Fig. 7. Initial triangulation (upper left); creating the texture texE from initial triangulation (left) and one texel of the texture texE (right).

4.1. Data structure design

We investigated two possibilities of representing triangulation data structures on a GPU. The first possibility—referred to as *point-centric approach in* [12]—was to represent each vertex of the triangulation by a fragment. This approach leads to a problem with the number of ingoing and outgoing edges to each vertex, since it changes during the optimization process. Such dynamic structures are difficult to handle effectively on a GPU in general. For this reason we decided to represent by fragments—referred to as *edge-centric approach in* [12]—the edges of the triangulation. The edge-centric method described in [12] is dealing with the same issue, but the proposed solution is different from our approach. Each edge in the triangulation is defined by its two endpoints—vertices (denoted as *main vertices*). We extend this *edge data element* by the other two vertices from the 1-region of the edge (denoted as *adjacent vertices*). The main and adjacent vertices are those vertices of the triangles which share the selected edge as a common edge.

The number of edges and vertices in the triangulation is constant during the optimization process. For each edge its corresponding data element contains four unique vertex *ID* s (the main and adjacent vertices of the edge). The edge data elements are stored in the texture labeled as *texE* (see Fig. 7). The vertex *ID* s are used for the computations of vertex positions in the input image and for obtaining luminance information for cost function evaluation. For this evaluation it is also necessary to store information about the neighboring edges. The *edge-neighbors data element* for an edge *e* thus consists of four *ID* s of the neighboring edges of *e*. These data elements are stored in the texture labeled as *texN* (see Fig. 8). The initial triangulation is created by connecting the neighboring pixels, as depicted in Fig. 7 (upper left part).

The textures *texE* and *texN* hold the necessary topological information about the triangulation. Additional textures are used to store information about the candidate edges for the second step of the algorithm (accepting and rejecting candidates). In that part we need to read and write from/to texture in each iteration. Owing to the limitations of the GPU it is necessary to have two such textures marked as $texC_1$, $texC_2$. These textures are interchanged as source and destination (read, write), in every iteration. In each texel of these textures information is stored, whether the corresponding edge is candidate, accepted or rejected, together

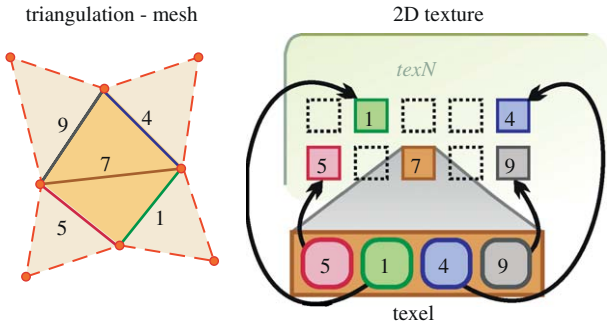

Fig. 8. Representation of the neighboring edges in texN.

with the ID of that edge. The data from the input image are stored in another texture, which is necessary for the cost computations. For general purpose computations it is also necessary to use an additional texture to store real vertex coordinates. In image reconstruction, however, the vertices form a grid and thus such texture is not necessary, since the vertex coordinates can be calculated from the vertex *IDs*.

In the following we give implementation details for all three steps of the algorithm, as introduced in Section 3.

4.2. Creation of candidates

In this step of the algorithm, candidates for edge flipping are selected. In the fragment shader the cost function of an edge is evaluated, accessing its vertices (from the ROI) by means of their *ID* s stored in *texN* and its neighboring edges using their *ID* s stored in *texE*. This is done simultaneously for each element of the texture, and thus for all edges in the triangulation.

According to the result of the cost function evaluation the edge becomes a candidate for flipping or it is locally optimal, in which case it remains unchanged. If the edge becomes a candidate then this state is stored (written) to the texture $texC_1$. If the tested edge is not a candidate, then the given fragment is not written to texture $texC_1$ (is discarded) and the corresponding location remains unchanged. In this process we can also obtain a number of candidate edges. We use a feature of the GPU called occlusion

query to count the written fragments. The obtained number of candidates is used in the next step of the proposed algorithm (accepting and rejecting candidates). If the number of candidates is zero, the calculated triangulation is locally optimal and the algorithm finishes.

4.3. Accepting and rejecting candidates

In the second step of the technique the candidate edges are either accepted or rejected. It is an iterative process where the textures $texC_1$ and $texC_2$ are mutually exchanged for reading and writing.

First, we copy the content of $texC_1$ to $texC_2$ to maintain candidate information for fragments, which are later discarded. To handle a candidate edge e, the necessary information (*ID*, acceptation status) is in a fragment program obtained from the untreated candidate edges in its ROI. By treated edge we mean an edge which has been marked as either accepted or rejected. If at least one edge from the ROI is marked as accepted, the tested edge is marked as rejected and this information is written to the texture. If the ROI of the processed edge does not contain any accepted edges, we compare the *ID* s of these untreated edges. If the processed edge has the minimal *ID*, then it is accepted and this information is written to the output. When there is an untreated edge with a lower *ID*, the fragment is discarded.

In the above described process some of the candidates are labeled as accepted or rejected, but untreated edges can still exist in the candidate set. Therefore this process is repeated until all candidates are treated. Occlusion query is used in this process to find the number of the treated candidates. The result is stored, according to the number of iterations, either in the texture $texC_1$ or in $texC_2$.

4.4. Edge flipping

The edge flipping operation causes the adjacent vertices to be exchanged with the main vertices. Afterward, information about the neighbors should also be updated. These changes have to be written to the textures *texE* and *texN* concurrently. This is done by the technique of multiple render targets. For this process, however, it is also necessary to read information from both textures *texE* and *texN*. Therefore, we make a copy of these textures and the copies are used as an input for the fragment program. Another input texture is one of the textures $texC_1$, $texC_2$, where information about the accepted candidates is stored.

Owing to architectural restrictions of the GPUs it is not possible to write to arbitrary positions in a texture, but rather only to a fixed position. In edge flipping three cases for a processed edge can occur. The first situation is the simplest, when neither the tested edge nor its neighbors are flipped. In this case the untouched values are written to the output (Fig. 9(a)). The second configuration is the case when only the tested edge is flipped (Fig. 9(b)). In this case the main vertices are exchanged with the adjacent vertices in the texture *texE*. Naturally, the correct values should be set also in *texN* to reflect the changes of the neighborhood of the flipped edge. The last configuration occurs when one edge from the neighborhood of the processed edge e is flipped (Fig. 9(c)). If a neighboring edge is flipped, it results in a change in the neighborhood information of the edge e. To handle this situation it is necessary to search the neighborhood of the tested edge e. If such a change occurs, *ID* s of the new neighbors are obtained and written to the output.

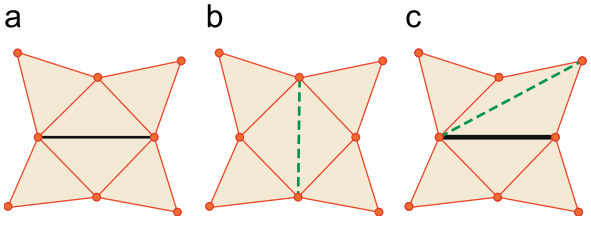

Fig. 9. Types of edge flipping (the tested edge is marked by the thick solid line, the flipped edge is marked by the dotted line): (a) no flip; (b) flip of the tested edge and (c) modification of the neighborhood.

4.5. Visualization

As soon as the DDT is calculated it is possible to download the data structure (textures *texN, texE*) to the CPU and save the triangulation in an appropriate format. Visualization of the triangulation is in this case straightforward. However, there is also a possibility to use the geometry shader to convert and render the triangulation directly without downloading the data structure to the CPU. The resulting triangulation from the geometry shader can be processed in the fragment units or directly written to a vertex buffer object for further processing and visualization. Moreover, the triangulation can be processed using the transform feedback technique [19]. In the presented approach both visualization techniques (CPU download based and geometry shader based) were tested and used.

5. Improvements

In the previous sections we introduced a new, GPU-based parallel optimization algorithm. Initial tests of this approach showed some limitations compared with the non-parallel CPU-based version. First, the GPU-optimized mesh usually had higher cost than the original CPU-based approach. From the theoretical point of view this is possible, since for an initial triangulation numerous mutually different, locally optimal configurations may exist, which naturally have different costs. Our goal, however, was not only to generate locally optimal meshes, but also to approximate a globally optimal solution. Second, we observed disturbing artifacts which often appeared in edge areas with certain properties.

The algorithm modifications introduced below were designed to eliminate these visual artifacts and to lower the cost of the triangulation. Their influence upon the resulting interpolation is presented in Fig. 10. The middle row in this figure shows the difference images (compared with the original image) computed in the perceptually linear color space.

5.1. ROI expansion

We analyzed the above-mentioned disturbing artifacts depicted in Fig. 10(c) showing the basic GPU reconstruction result. We observed that the artifacts were caused by improperly oriented triangle edges, which were uniformly distributed along the high-frequency areas in a distance approximately the size of the ROI area. Therefore, we expanded the ROI used for the candidate selection process—step 2, pseudo-code lines 9–20. In the basic approach this region was of degree 2, which comprises 12 edges. We expanded this region by additional edges to a 3-region. The situation is depicted in Fig. 4(b), where the new added edges are marked with dashed lines. This modification partially suppressed the artifacts, and the final cost of the triangulation was lowered. In the text, this improvement is referred to as *ExpROI*. Fig. 10(e) shows its visual contribution in comparison with the basic GPU algorithm.

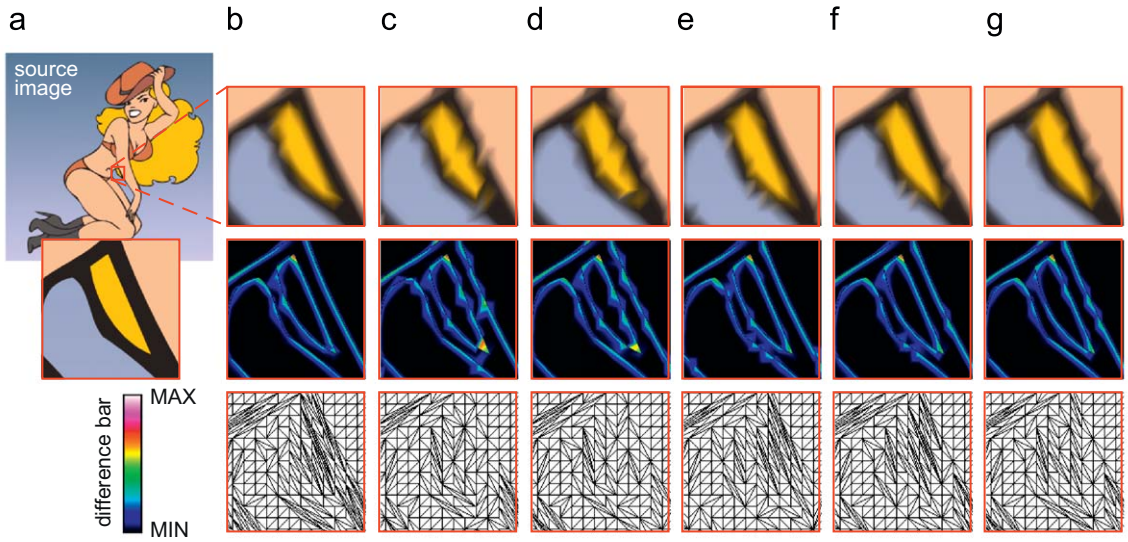

Fig. 10. Close-up of an image magnified to 1200 percent from a rasterized vector image. The emphasized parts belong to the edge with the most severe artifacts: (a) the original image; (b) the CPU-based DDT; (c) the basic GPU approach, (d) GPU MaxGain; (e) GPU ExpROI; (f) GPU ExpROI3 and (g) GPU ExpROI MaxGain. The middle row shows the difference images compared to the original image and the bottom row shows the topology of the corresponding triangulations.

A drawback of the extended ROI was its increased computational complexity, which resulted in longer execution time of the corresponding fragment program. We observed, however, that these artifacts usually originated in the first two iterations of the basic GPU version. Thus, the costly expanded ROI processing was necessary only in the first N iterations, where the N can be set by the user. After these N iterations, the basic approach is used.

In the following we refer to this technique as *ExpRoiN*. As illustrated in Fig. 10(f), this modification leads to similar visual appearance to *ExpRoi* (Fig. 10(e)).

5.2. Gain maximization

The main goal of this modification is to lower the cost of the triangulation. Unlike the basic version, where the accepted edges were selected purely on edge IDs, in this version we take magnitude of contribution of edge flipping to minimization of the triangulation cost into account. Now, we not only evaluate the local edge cost, but we also store its value and use it in the accepting and rejecting step of the algorithm. When iterating over the candidate set, the edge with the highest gain (difference between the cost before and after edge flip) from the ROI of the current edge is selected. If some of the tested edges in the ROI have equal maximal gain, then their IDs are used for the decision, similarly to the basic version.

This modification increases the runtime of the algorithm but the resulting cost is usually lower and thus better approximates the global optimum. Simultaneously, this version converges faster to minimum. Therefore, the drawback of this enhancement is that we can get stuck in a poorer local optimal configuration, compared with the situation where decrease of the cost is slower. A similar behavior was observed in minimization by simulated annealing with improperly adjusted control parameters [30]. In [15] an analogous procedure is mentioned, targeting at improvement of a non-parallel version of Lawson's optimization process.

In the following text this optimization is denoted as *MaxGain*. Fig. 10(d) shows its result in comparison with the basic GPU algorithm. With this modification the cost of the triangulation is usually lower than that of the basic approach.

The last proposed modification combines the previous approaches. In concert with our expectations this algorithm generates triangulations with the lowest cost and best suppression of visual artifacts within acceptable computation time. Fig. 10(g) depicts the visual results.

6. Results and optimization

In this section we present time, quality and cost measurements of the algorithms described in the previous sections. The approaches were tested on two datasets. The first dataset contained 12 real-life raster images, which were selected from commonly used test image sets used in image processing. The second dataset contained 8 artificial images drawn and rasterized in a vector graphic editor. Both datasets were iteratively down-scaled by a factor of two with bilinear filtering to $\frac{1}{4}$ and eventually to $\frac{1}{8}$ of the original image size. The down-scaled images had resolutions from 64×64 to 250×243 pixels. These down-scaled images were magnified back to the original size by different reconstruction approaches. The CPU version of the algorithm was implemented in the C++ language. For implementation of the GPU version we used the OpenGL API and alternatively the OpenGL Shading Language (glsl) [29] and C for graphics (Cg) [10] for shader programming. The minimal hardware requirement of our technique is the Shader Model 3.0 compatibility, which is available on most of the recent consumer GPUs. We also investigated usability of the geometry shader for direct calculation of the DDT. After a detailed analysis, however, we came to the conclusion that usage of this feature would not be sufficiently efficient. The measurements were performed on systems equipped with a Intel Pentium 4 3.0 GHz CPU, 1 GB RAM and the NVIDIA GeForce 8800 GTS GPU with 640 MB of memory, NVIDIA GeForce 9600 GT with 512 MB of memory and ATI Radeon HD 4770 with 512 MB of memory.

First, we compared runtime of the introduced techniques against the CPU-based version of the DDT algorithm. This measurement was split into three parts: *initialization*, *computation* and *finalization*. In the initialization stage an image was loaded and the appropriate data structures (initial triangulation) were generated. In the GPU version, the created textures were also uploaded to the GPU. In the next stage the computation time of the locally optimal triangulation generation was measured. In the last stage the resulting triangulation was saved to the disk.

Moreover, for the GPU algorithms, the textures were also downloaded from the GPU to the main memory and converted to the appropriate format. The measured times are listed in Table 1, where *time-init.* is duration of the first, *time-comp.* of the second and *time-final* of the third stage. These times are summed and listed in the last row of Table 1, marked as *time-sum*. The measured times are in seconds, and they are the averaged values for the images in the selected datasets.

The cost of a triangulation is an important aspect of quality measuring, since the aim of the introduced algorithms is to approximate the MWT. Therefore, in Table 1 the costs of the initial triangulations (marked as *cost-init.*), costs after the reconstruction (marked as *cost-final*) and improvement in percent (marked as *improvement*) are also given. The *num. of flips* is the number of flip operations performed during the optimization process.

It can be seen from Table 1 that the GPU version was around 6–10 times faster than the CPU-based solution. For the vector images this speed-up was even higher (25–30 times). Here, we observed slowdown in the CPU version of the algorithm which was caused by flipping only a negligible number of edges in the last iterations. This behavior was caused by the fact that the rasterized vector images were not corrupted by noise. In the real images this behavior was not observed.

We tested the proposed algorithms also on various GPUs which was made possible by the platform independence of the glsl implementation. The obtained runtime results for the real and artificial test set are presented in Table 2. They correspond to the average total time (time-sum) results of Table 1.

We observe that the results for the GeForce 8800 GPU for both implementations (glsl and Cg) are nearly identical—their difference is within a few percent—and that the GeForce 9600 GPU is systematically by about 35 percent faster than the 8800 version. The situation is, however, different for the implementation on the ATI GPU, which, depending on the algorithm, shows both speedup and significant slow down of up to three times longer processing time in comparison to the GeForce 8800 GPU. This can be explained by the different internal architecture of NVIDIA and ATI GPUs, which results in different efficiency of various operations.

The costs of the triangulations obtained on different GPUs differed only slightly (on decimal positions). This was probably caused by different internal precision and other architectural differences of the test GPUs.

The visual quality of the results was evaluated by several perceptual metrics which are commonly used in image processing for quality measuring. The selected measurement tools were correlation, cross-correlation, MSE, SNR [13], UIQI [33], SSIM [34]. Besides the above-mentioned algorithms standard reconstruction techniques (convolution based image processing methods) were used for quality comparison: bilinear, b-spline, and Lanczos filter. We used the ImageMagick package for the convolution [17]. Averaged values obtained in the evaluation of the results are listed in Table 3. Higher values mean better quality, except for the MSE where lower values mean a better reconstruction result. From the results it can be seen that the image reconstruction based on DDT can compete with the convolution based techniques.

Subjective visual quality represented by the human visual system cannot be measured with perceptual metrics. For this reason we include some reconstruction results obtained by different techniques, and leave the reader to decide about the quality of the reconstruction approaches—Figs. 11 and 12. Here, blocky artifacts in the high-frequency areas are noticeable for the magnifications obtained by the image processing techniques.

From our visual inspection and from the measured values we came to the following conclusions. The best-quality results of the GPU-based algorithms yield the *GPU ExpROI* 2/3, *GPU MaxGain* and

Table 1
The average cost and timing measurements for artificial and real-world scenes (Cg implementation on the NVIDIA GeForce 8800 GTS GPU).

Average of	CPU DDT	GPU basic	GPU ExpROI	GPU MaxGain	GPU ExpROI MaxGain	GPU ExpROI2 MaxGain	GPU ExpROI3 MaxGain	GPU ExpROI4 MaxGain
Speed and cost measurement—real test set								
Cost-init.	1549.901	1549.901	1549.901	1549.901	1549.901	1549.901	1549.901	1549.901
Cost-final	691.641	825.867	795.837	777.825	753.367	754.047	753.288	753.445
Improvement (%)	224.090	187.670	194.751	199.261	205.730	205.544	205.752	205.709
Num. of flips	17727	16440	17089	14808	15066	15109	15105	15092
Time-init. (s)	0.212	0.407	0.518	0.453	0.602	0.601	0.599	0.601
Time-comp. (s)	6.834	0.252	0.619	0.277	0.470	0.518	0.527	0.534
Time-final. (s)	0.221	0.089	0.093	0.087	0.085	0.091	0.089	0.085
Time-sum (s)	7.268	0.749	1.231	0.817	1.158	1.210	1.216	1.221
Speed and cost measurement—artificial test set								
Cost-init.	1549.762	1549.762	1549.762	1549.762	1549.762	1549.762	1549.762	1549.762
Cost-final	341.809	432.785	436.413	420.602	365.502	360.335	363.982	363.736
Improvement (%)	453.400	358.090	355.113	368.463	424.009	430.089	425.779	426.068
Num. of flips	8626	7263	7423	6696	7054	7161	7118	7106
Time-init. (s)	0.315	0.414	0.520	0.434	0.609	0.604	0.604	0.609
Time-comp. (s)	32.104	0.304	0.526	0.325	0.536	0.570	0.575	0.570
Time-final. (s)	0.331	0.145	0.148	0.151	0.146	0.148	0.143	0.143
Time-sum (s)	32.751	0.864	1.195	0.911	1.291	1.323	1.322	1.322

Table 2
The average timing measurements for various GPUs (glsl implementation).

Average of	GPU basic	GPU ExpROI	GPU MaxGain	GPU ExpROI MaxGain	GPU ExpROI2 MaxGain	GPU ExpROI3 MaxGain	GPU ExpROI4 MaxGain
NVIDIA GeForce 8800GTS	0.873	1.225	0.866	1.222	1.169	1.184	1.209
NVIDIA GeForce 9600GT	0.574	0.885	0.551	0.820	0.741	0.753	0.762
ATI Radeon HD 4770	0.608	1.354	0.644	3.962	4.027	4.020	4.015

Table 3
The average quality of the reconstruction results, measured by perceptual metrics.

Average of	Bilinear filter	b-spline filter	Lanczos filter	CPU DDT	GPU basic	GPU ExpROI	GPU MaxGain	ExpROI MaxGain	ExpROI2 MaxGain	ExpROI3 MaxGain
Quality measurement—real test set										
Correlation (%)	96.0142	95.3072	96.3722	96.1783	96.1418	96.1573	96.1485	96.1587	96.1600	96.1590
Cross cor. (%)	94.7434	93.3080	95.5652	95.2463	95.1506	95.1779	95.1618	95.1725	95.1693	95.1728
MSE	294.4280	377.7320	249.8240	267.5770	272.8820	270.6890	272.2040	270.8790	270.7290	270.8010
SNR (dB)	17.7444	16.6075	18.5715	18.2452	18.1580	18.1989	18.1713	18.1914	18.1952	18.1927
UIQI	0.3611	0.2966	0.4100	0.3875	0.3830	0.3848	0.3833	0.3847	0.3847	0.3847
SSIM	0.6359	0.5956	0.6609	0.6518	0.6487	0.6500	0.6491	0.6499	0.6500	0.6500
Quality measurement—artificial test set										
Correlation (%)	97.8322	97.1695	98.1455	98.2470	98.1522	98.1715	98.1570	98.1982	98.2057	98.2007
Cross cor. (%)	97.1942	95.8883	97.9042	97.7710	97.6275	97.6450	97.6162	97.6887	97.6992	97.6920
MSE	343.4650	500.5060	253.4830	268.1490	286.9160	283.8380	287.4920	278.8990	277.2380	278.3750
SNR (dB)	21.2149	19.7293	22.5276	22.3261	22.0282	22.0616	22.0196	22.1660	22.1865	22.1702
UIQI	0.6953	0.6228	0.5738	0.7806	0.7757	0.7772	0.7760	0.7782	0.7785	0.7783
SSIM	0.9019	0.8798	0.9099	0.9203	0.9156	0.9165	0.9159	0.9179	0.9183	0.9180

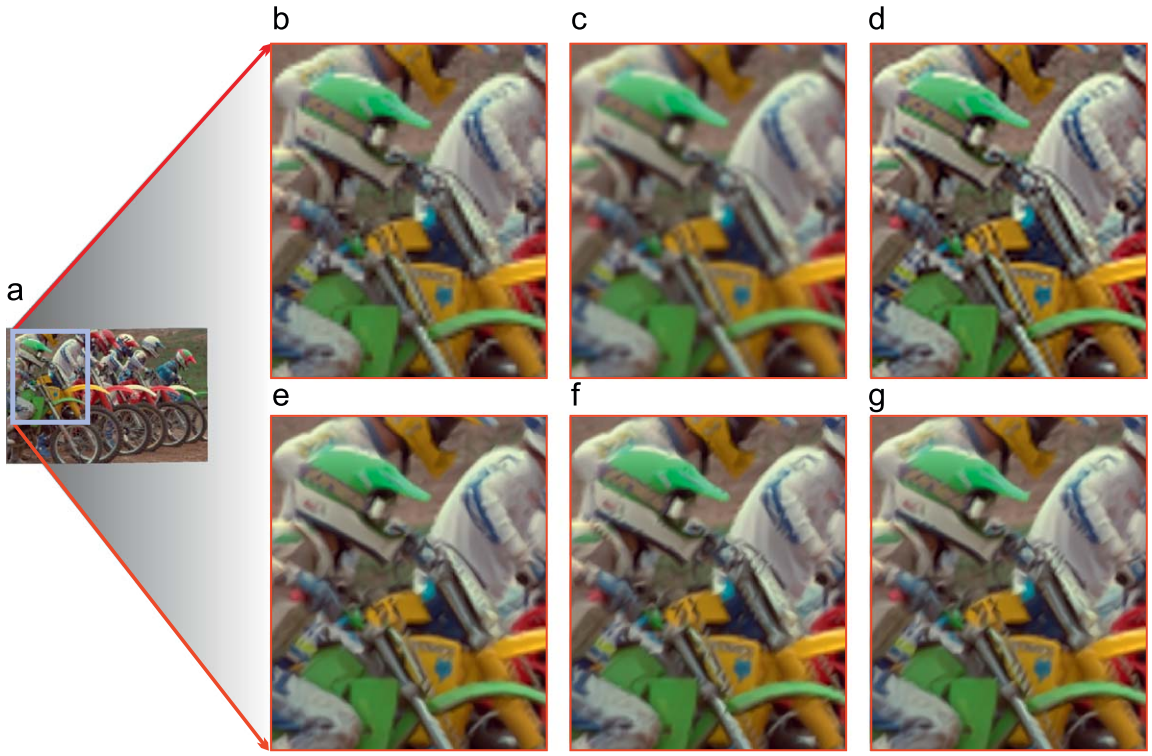

Fig. 11. Reconstruction results at 400 percent magnification: (a) original image; (b) bilinear filter; (c) b-spline filter; (d) Lanczos filter; (e) CPU-based DDT; (f) basic GPU approach and (g) GPU ExpROI MaxGain modification.

the *GPU ExpROI MaxGain* algorithms. The shorter runtime of the *GPU ExpROI MaxGain* algorithm compared with the runtime of the *GPU ExpROI* 2/3 *MaxGain* algorithm was unexpected. We expected that algorithms which used expanded region of the neighborhood control only for a couple of first iterations (*GPU ExpROI* 2/3 *MaxGain*) would be faster than *GPU ExpROI MaxGain*, which used this expanded region for all iterations. Test results are opposite to our expectations. This behavior is probably caused by fragment program switching during the computation, which is a costly operation.

Exact analytical descriptions of the time complexity of the introduced algorithms depends on numerous factors such as the type of the selected cost function, data distribution, etc. Moreover, these factors are specific for each area of application. Thus, such complex analysis exceeds the ambition of this paper. Analysis of time complexity for an approach similar to our acceptation and rejection of candidates is presented in [18].

Instead, we measured the time complexity of the image reconstruction problem in an empirical way. A set of 91 differently sized pictures was selected from a set of standardly used image processing test images. The results are presented in Fig. 13 (*Dataset* 1) where the dependency between the runtime of the *GPU ExpROI MaxGain* algorithm and the size of the source image is depicted. We observe linear growth for small and medium images and faster-than-linear growth for the largest ones. We also observe some outliers, the run time of which clearly exceeds run time of other images with similar size. This was probably caused by a "nearly worst case" distribution of the edges from the point of view of parallel processing. The number of these

Fig. 12. Reconstruction results at 400 percent magnification: (a) original image; (b) bilinear filter; (c) Lanczos filter; (d) CPU-based DDT and (e) GPU ExpROI2 Max Gain modification.

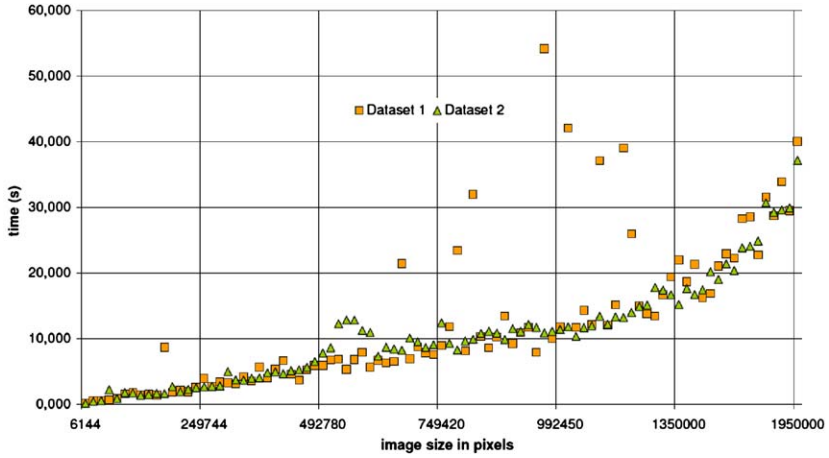

Fig. 13. Run time of the *GPU ExpROI MaxGain* algorithm (400 percent magnification) measured on *Dataset* 1 and *Dataset* 2.

outliers compared to the size of the dataset is not significant. We also used a second dataset which was generated from one selected image from *Dataset* 1 by resizing it to the same resolutions as images in *Dataset* 1. This set is denoted as *Dataset* 2 in Fig. 13. We see a similar time dependency, however, without any outliers.

7. Conclusion and future work

In this work we introduced an efficient technique for parallel optimization of data-dependent triangulations on a GPU. Specifically, we focused our attention on the image reconstruction problem—image magnification. The technique was compared using several perceptual metrics and visual inspection with traditional serial DDT implementations on a CPU, as well as with appropriate image processing based interpolation techniques by convolution. The visual quality of the results was similar to that obtained by the serial approach and superior to the results obtained by the convolution techniques. From the point of view of perceptual metrics the results were comparable. Thanks to parallelization and GPU implementation our approach to DDT was up to eight times faster on average than the serial implementation. We have proposed various modifications which lead to improved visual quality.

Our solution to the data-dependent triangulation problem, based on our parallelization of Lawson's optimization process, is general and allows us to compute locally optimal triangulations using various cost functions on the graphics hardware. These algorithms can be fitted to special optimization tasks, and can also provide locally optimal meshes with different properties outside the area of image processing. This means that it can be fitted to arbitrary data distribution and arbitrary optimization tasks related to the DDT approach.

Both the CPU and GPU versions require an initial triangulation. In the image reconstruction problem this task is simple and can be performed on a GPU in a very efficient and fast way. In other application areas, however, the initial triangulation may be implementable solely on a CPU and thus the created data structures have then to be transferred to the GPU for optimization.

The introduced technique has a big potential in image reconstruction, as we showed in Section 6. In our future work we intend to combine the new DDT technique with the standard convolution techniques. From this combination we expect more visually pleasant and faster image reconstruction. Further improvement of the introduced algorithms can enhance the quality of the generated locally optimal meshes. There is also an additional possibility how to improve performance of the computations both for the CPU and GPU versions of the algorithms. This optimization is related to the edge flipping operation in locally optimal triangulations. After the first iteration it is enough to test only the local optimality of edges from the ROI of the flipped edges in the previous iteration step. Since this change also requires changes in the data structures, we would like to investigate this feature in our future work. We further speculate on the possibility of implementing certain non-deterministic approaches like simulated annealing or genetic algorithms on a GPU.

New technologies, such as *CUDA* [26] and *ATI Stream* [2] can be used for computations of locally optimal meshes on the GPU too. These APIs reduce or eliminate some of the OpenGL restrictions and provide better programming flexibility and memory management. On the other side, the solution must be designed for a specific target hardware. Therefore, in our future work, we want to take advantage of the new *OpenCL* [20] standard, which enables to write hardware independent code to be executed on multi CPU systems as well as on various types of GPUs.

Acknowledgments

This work was supported by the Slovak Research and Development Agency under Contract no. APVV-20-056105. Part of the work presented in this publication was carried out within the VEGA project No. 1/0763/09.

References

[1] Alboul L, Kloostreman G, Traas C, Damme RV. Best data-dependent triangulations. Journal of Computational and Applied Mathematics 2000; 119(1–2):1–12.
[2] AMD, 2009. ATI stream computing—technical overview ⟨http://ati.amd.com/technology/streamcomputing/⟩, documentation.
[3] Aurenhammer F. Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Computing Surveys 1991;23(3):345–405.
[4] Battiato L, Gallo G, Messina G. SVG rendering of real images using data dependent triangulation. In: Proceedings of spring conference on computer graphics, 2004. p. 191–8.
[5] Bjorke K. 2004. High-quality filtering. In: GPU gems: programming techniques, tips, and tricks for real-time graphics. Reading, MA: Addison-Wesley. p. 391–415.
[6] Bose P, Czyzowicz J, Gao Z, Morin P, Wood DR. Simultaneous diagonal flips in plane triangulations. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on discrete algorithm. New York: ACM Press; 2006. p. 212–21.
[7] Brown J. Vertex based data dependent triangulations. Computer Aided Geometric Design 1991;8(3):239–51.
[8] Dyn N, Levin D, Rippa S. Data dependent triangulations for piecewise linear interpolation. IMA Journal of Numerical Analysis 1990(10):137–54.
[9] Ferko A, Niepel L, Plachetka T. Criticism of hunting minimum weight triangulation edges. In: Proceedings of the 12th spring conference on computer graphics, 1996. p. 259–64.
[10] Fernando R, Kilgard MJ. The Cg tutorial: the definitive guide to programmable real-time graphics. Reading, MA: Addison-Wesley; 2003.
[11] Fisher I, Gotsman C. Fast approximation of high order Voronoi diagrams and distance transforms on the GPU. Journal of Graphics Tools 2006;11: 39–60.
[12] Georgii R, Westermann R. Mass-spring systems on the GPU. Simulation Modelling Practice and Theory 2005;13:693–702.
[13] Gonzalez RC, Woods RE. Digital image processing, 3rd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.; 2006.
[14] Hadwiger M, Theußl T, Hauser H, Gröller ME. Hardware-accelerated high-quality filtering on PC hardware. In: Proceedings of vision, modeling, and visualization, 2001. p. 105–12.
[15] Hjelle O, Dahlen M. Triangulations and applications, series: mathematics and visualization. Berlin: Springer; 2006.
[16] Hoff KE, Culver T, Keyser J, Lin M, Manocha D. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. In: Proceedings of ACM SIGGRAPH 1999. New York: ACM; 1999. p. 277–86.
[17] ImageMagick, 2009 ⟨www.imagemagick.org⟩, webpage.
[18] Jones MT, Plassmann PE. A parallel graph coloring heuristic. SIAM Journal on Scientific Computing 1993;14:654–69.
[19] Khronos Group, 2008. The NV_transform_feedback specification ⟨http://www.opengl.org/registry/specs/NV/transform_feedback.txt⟩, documentation.
[20] Khronos Group, 2009. The OpenCL specification ⟨http://www.khronos.org/opencl/⟩, documentation.
[21] Kolingerová I. Genetic approach to data dependent triangulations. In: Proceedings of spring conference on computer graphics, 1999. p. 229–38.
[22] Kolingerová I, Ferko A. Multicriteria-optimized triangulations. The Visual Computer 2001;17(6):380–95.
[23] Kreylos O, Hamann B. On simulated annealing and the construction of linear spline approximations for scattered data. IEEE Transactions on Visualization and Computer Graphics 2001;7(1):17–31.
[24] Luebke D, Harris M, Krüger J, Purcell T, Govindaraju N, Buck I, et al. GPGPU: general purpose computation on graphics hardware. In: SIGGRAPH '04: ACM SIGGRAPH 2004 course notes. New York: ACM; 2004. p. 33.
[25] Mulzer W, Rote G. Minimum weight triangulation is NP-hard. In: SCG '06: Proceedings of the twenty-second annual symposium on computational geometry. New York: ACM; 2006. p. 1–10.
[26] NVIDIA Corporation, 2009. NVIDIA CUDA compute unified device architecture ⟨www.nvidia.com/cuda⟩, documentation.
[27] Rong G, Tan T-S. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In: Proceedings of the symposium on interactive 3D graphics and games. New York: ACM Press; 2006. p. 109–16.
[28] Rong G, Tan T-S, Cao T-T. Stephanus, Computing two-dimensional Delaunay triangulation using graphics hardware. In: SI3D '08: Proceedings of the 2008 symposium on interactive 3D graphics and games. New York: ACM Press; 2008. p. 89–97.
[29] Rost RJ. OpenGL(R) shading language, 2nd ed. Reading, MA: Addison-Wesley Professional; 2005.
[30] Schumaker LL. Computing optimal triangulations using simulated annealing. In: Selected papers of the international symposium on free-form curves and free-form surfaces. Amsterdam: Elsevier Science Publishers; 1993. p. 329–45.
[31] Su D, Willis P. Image interpolation by pixel-level data-dependent triangulation. Computer Graphics Forum 2004;23(2):189–202.
[32] Toth Z. Towards an optimal texture reconstruction. In: CESCG 2000–2005 best paper selection. Wien: Österreichische Computer Gesellschaft; 2006. p. 197–212.
[33] Wang Z, Bovik A. A universal image quality index. IEEE Signal Processing Letters 2002;9(3):81–4.
[34] Wang Z, Bovik AC, Sheikh HR, Simoncelli EP. Image quality assessment: from error visibility to structural similarity. IEEE Transactions on Image Processing 2004;13(4):600–12.
[35] Yu X, Morse BS, Sederberg TW. Image reconstruction using data-dependent triangulation. Computer Graphics and Applications 2001;21(3):62–8.