Vision, Modeling, and Visualization (2012)
M. Goesele, T. Grosch, B. Preim, H. Theisel, and K. Toennies (Eds.)

Generating Smooth High-Quality Isosurfaces for Interactive Modeling and Visualization of Complex Terrains

F. Löffler and H. Schumann

Institute for Computer Science, University of Rostock, Germany

Abstract

In modeling and rendering of complex procedural terrains the extraction of isosurfaces is an important part. In this paper we introduce an approach to generate high-quality isosurfaces from regular grids at interactive frame rates. The surface extraction is a variation of Dual Marching Cubes and designed as a set of well-balanced parallel computation kernels. In contrast to a straightforward parallelization we generate a quadrilateral mesh with full connectivity information and 1-ring vertex neighborhood. We use this information to smooth the extracted mesh and to approximate the smooth subdivision surface for detail tessellation. Both improve the visual fidelity when modeling procedural terrains interactively. Moreover, our extraction approach is generally applicable, for example in the field of volume visualization.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms

1. Introduction

The procedural modeling of complex terrains (including 3D features) is often based on isosurfaces (cf. [Gei07, BFO*07]) as they are not restricted to any initially defined surface topology. Isosurfaces are implicitly defined. That means that the function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ in conjunction with an isovalue α implicitly defines the surface $S_\alpha = \{x \in \mathbb{R}^3 \,|\, f(x) = \alpha\}$. However, for the rendering process we usually need an explicit surface representation.
Although, rendering of such surfaces is common in different domains e.g. volume visualization, existing surface extraction algorithms either focus on performance or on mesh quality. Extraction of high-quality meshes requires complex computations. Thus, most of these algorithms are not suitable for interactive applications. In contrast, algorithms focusing on performance make compromises with regard to quality. This might cause visual artifacts or might lead to meshes that are unsuitable for further processing, e.g. smoothing or detail tessellation.

However, for the generation, editing and rendering of procedural content (e.g. modeling complex terrains) we need both high quality and high performance. That means we have to address the following aspects:

- Manifold surfaces and connectivity information (including *1-ring* neighborhood) are required to apply geometric processing algorithms and to export the mesh to external modeling tools.
- Interactive frame rates are needed for quick interaction-feedback cycles when changing model parameters (e.g. noise parameter) or when adjusting the isovalue.
- A high-quality surface mesh (i.e. well-shaped and well-distributed quadrilaterals) is needed to minimize visual artifacts.

To fulfill these requirements we propose a novel processing pipeline with well-balanced parallel computational kernels and different lookup tables. Our solution integrates steps to *extract* the isosurface, *smooth* the resulting mesh, and apply *detail tessellation* to enhance visual fidelity.

The extraction of manifold surfaces is based on *Dual Marching Cubes* introduced by Nielson [Nie04]. We extend this approach and propose a novel parallel processing scheme. In contrast to straightforward parallelizations, we generate a quadrilateral mesh with full connectivity infor-

DOI: 10.2312/PE/VMV/VMV12/079-086

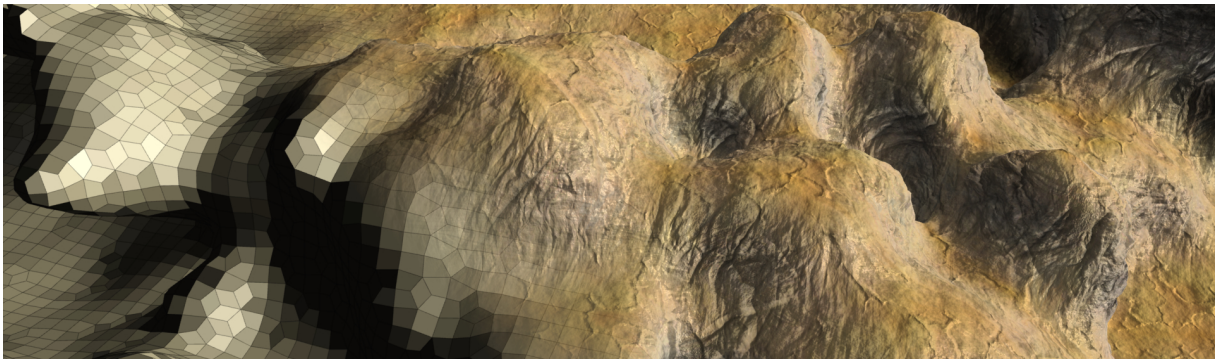

Figure 1: *A complex terrain surface extracted in* 12*ms with the proposed algorithm. We improve the mesh quality by applying the surface diffusion flow and tangent movement (left). For the rendering the 1-ring vertex neighbors are used to approximate the Catmull Clark subdivision surface as proposed in [LS08]. Displacement mapping enhances the visual fidelity (right).*

mation and *1-ring* vertex neighborhood.
To adapt to sharp features we present an alternative vertex relocation solution based on the *Quadric Error Metric* [GH97] (QEM) which is computationally efficient.
For smoothing we use a combination of the *surface diffusion flow* and *tangent movement*. For this purpose, we need the 1-ring vertex neighbors, which are usually gathered by an additional step that analyzes the mesh. Our solution is different though, as it gathers the 1-ring neighbors directly during the surface extraction. In this way, smoothing operations can be integrated with the surface extraction.
Detail tessellation is applied by approximating the smooth subdivision surfaces (e.g. *bicubic patches*) and displacement mapping. Since we provide the 1-ring neighbors, such a high order patch representation can be easily generated.

Our solution allows for modeling and visualizing procedural complex terrains interactively at a high visual fidelity (cf. Figure 1). The solution is general and might be used in other application domains as well.

In the following section we provide related work and discuss the implications with regard to algorithm requirements. In Section 3 we describe our general approach followed by a detailed description of our novel surface extraction pipeline (see Section 4). In Section 5 we discuss the results. We conclude our work in Section 6 and provide ideas for future work.

2. Releated Work

The extraction of isosurfaces has been a research topic for decades. Usually we distinguish between *primal* and *dual* contouring methods (cf. [SW04]). Primal methods generate a polygonized surface by connecting intersection points of the isosurface with a structured grid. Dual methods generate the topologically dual of primal surfaces. Both methods have been implemented successfully on recent graphics hardware.

Primal contouring methods are often equated with the *Marching Cubes* (MC) algorithm (see [LC87]) the quasi standard algorithm for polygonizing implicit surfaces. The algorithm is general, robust and simple and forms the basis for many other widely used algorithms (cf. [Nie03, NY06]). Since some of the 15 canonical configurations are ambiguous, the original algorithm possibly generates non-manifold surfaces. Nielson and Hamann address this problem in [NH91] and resolve the ambiguities by enhancing the classification to 23 canonical cell configurations. In contrast to that, other MC derivatives e.g. *Marching Tetrahedron* (MT) do not have topological ambiguities (see [NB93]). Nonetheless, MC surfaces lack sharp features and quality. These issues have been examined thoroughly over past decades. For instance, the *Extended Marching Cubes* [KBSS01] detects sharp features by taking the normals into account. Labsik et al. [LHMG02] rely on a hierarchical approach to improve the mesh quality. However, such methods require significantly more computations in contrast to the original MC (see [NY06]).

Dual contouring methods: locate vertices within a cell and connect them with adjacent ones. By doing so, dual methods reproduce sharp features and tend to avoid the poorly shaped triangles that are often appearing in MC surfaces. The *Dual Contouring* [JLSW02] (DC) algorithm uses a octree decomposition of the domain. To consider sharp features, vertices are relocated in an octree cell by minimizing the *Quadratic Error Function* (QEF). With a set of <point,normal> pairs $(\mathrm{p_i}, \mathbf{n}_i)$ corresponding to the intersection of the isosurface with the octree cell the QEF is defined as: $E(\mathrm{v}) = \sum_i [\mathbf{n}_i^T(\mathrm{v} - \mathrm{p_i})]^2$. In matrix form a solution is found by using the *pseudoinverse* of a $n \times 3$ matrix (e.g. QR decomposition) where n is the number of intersection points (cf. [JLSW02]). For more information we refer to [SW02].
Just as the original MC, the DC possibly produces non-manifold surfaces. Schaefer et al. solve this problem

by using a topology-preserving vertex clustering algorithm [SJW07], whereas [ZHK04] represent isosurfaces by an *enhanced cell* representation.
The *Dual Marching Cubes* (DMC) algorithm proposed by Nielson [Nie04] always generates topological manifold surfaces. Nielson unifies MC surface fragments to polygonal patches where the vertices of these patches are located on the lattice edges. Since each lattice edge is adjacent to four cells, each patch vertex is touched by four patches. The dual surface is now defined (1) by replacing each patch by a vertex and (2) by replacing each patch vertex by a quadrilateral face. In contrast to DC, this approach results in a classification of 23 cell configurations that are dual to the 23 MC configurations required for extracting topological manifolds. Each configuration may create up to 4 vertices and the connectivity is well defined via the lattice edges. More precisely, when a lattice edge intersects the isosurface, this edge is associated with four vertices forming a quadrilateral surface fragment. In [SJW07] this property is exploit to define the QEF. We rely on this algorithm for two reasons: First, the algorithm generates manifold surfaces. Second, the configuration-based approach provides many advantages for parallelization.

Other contouring methods: In [SW04] both a primal method and a dual method are used in combination. First, a grid dual to the primal grid (octree) is generated. Vertices of this *dual grid* are located at features of the implicit surface. In a second step MC is applied to the dual grid to create a high-quality surface mesh.
Advancing front algorithms have also been applied to extract isosurfaces [Blo94, Har98]. In [SSS06] a so-called guidance field is computed from the implicit surface which drives the advancing front algorithm. The algorithm produces high-quality smooth triangle meshes. Even though many parts of the algorithm have been implemented in parallel this approach is far away from interactive rates.

GPU-based variations: Both MC and DC provide real-time capabilities on programmable graphics hardware (cf. [TSD07, Gei07, SDC09, LMS11]). In sum, such implementations (a) lack mesh quality, (b) provide topology ambiguities and (c) generate a "primitive soup". Especially the last point causes huge memory and computational overhead since vertices are generated and stored multiple times. This also makes such meshes unsuitable for further geometric processing and high order smooth surface approximations (e.g. approximation of *Catmull Clark surfaces* [LS08]). Loeffler et al. [LMS11] take advantage of general purpose GPU shaders and generate a mesh with connectivity information for tiny parts of the volume. This procedure has two drawbacks: (1) it does not generate a connected mesh for the whole isosurface, and (2) due to the usage of Dual Contouring it possibly generates non-manifold surfaces.

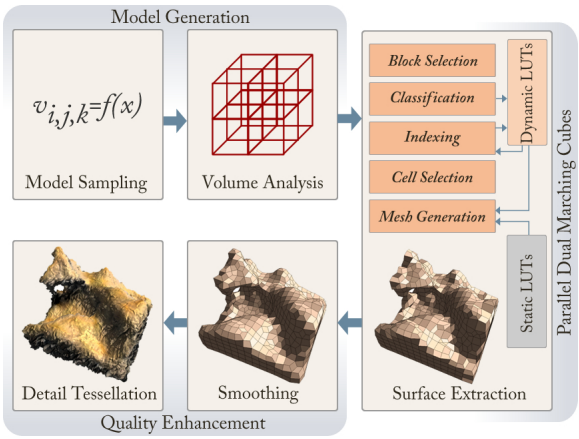

Figure 2: *Illustrates the overall modeling process.*

3. General Approach

In this section we focus on the generation of smooth high-quality isosurfaces. Our goal is the interactive modeling of procedural complex terrains (with 3D features) at a high visual fidelity. Our solution relies on the following three principle components (see Figure 2):

1. The *model generation* which samples the density function and analyzes the volume for further processing.
2. The *surface extraction pipeline* which generates the explicit isosurface representation.
3. The *quality enhancement* which improves the mesh quality and the visual fidelity.

During the modeling it is not necessary to pass through the whole pipeline. For instance, it can be sufficient to only adjust the model parameters to generate a new model. Isovalue changes though, merely affect the surface extraction and thus lead to new shapes. By doing so the modeling process is accelerated. Nonetheless, achieving interactivity is not a trivial task and requires well-matched components. In addition, each component puts several demands on previous components. In the following we briefly describe the components and discuss design choices.

The first component is the model generation. We sample the values on a regular grid because (a) in advance we have no information about the surface (b) the sampling can be efficiently implemented on parallel hardware and (c) most parallel surface extraction algorithms are optimized for this representation. The sampling is general. However, for our purposes we define the density function $f(x)$ as a combination of different types of noise and specific terrain functions as described in [Gei07, GT07].
Interactivity requires the efficient processing of the volume. Hence, we need a method to select relevant parts of the volume efficiently. We rely on a block-based acceleration structure. We subdivide the volume into blocks with a fixed num-

ber of cells. For each block we compute the *min* and *max* density values by a parallel reduction algorithm. This acceleration structure is simple but (a) analysis and selection can be efficiently implemented in parallel and (b) processing of single blocks can be mapped well to the parallel processing concept (i.e. in analogy to thread groups).

The second component - the surface extraction pipeline - selects relevant blocks and extracts the isosurface. To avoid visual artifacts and unequally distributed details we need well-shaped and well-distributed primitives. Hence, we need to smooth the extracted surface mesh in the last component. Applying smoothing operators requires (1) a manifold surface, (2) connectivity information, and (3) the 1-ring vertex neighborhood. Usually, a parallel surface extraction algorithm does not fulfill these requirements. This is caused by the parallel processing itself. In parallel all cells are analyzed and surface fragments are produced based on cell characteristics. This results in redundant computations and, more importantly, it results in unconnected primitives. This is not suitable for our approach.
We need a manifold surface. We, therefore, rely on Dual Marching Cubes byNielson [Nie04]. The algorithm fits well our requirements in terms of quality, but not in terms of interactivity. We developed an extended version of DMC that follows a novel parallel processing scheme. Our multi-staged processing pipeline generates a quadrilateral surface mesh with full connectivity information. This is possible due to a smart combination of static and dynamic look-up tables as well as parallel reduction algorithms. Section 4 is devoted to the pipeline in detail.
Usually, 1-ring neighbors are not supplied by surface extraction algorithms. In fact, it demands an extensive analysis of the mesh which is not suitable for interactive applications. We solve this problem by precomputing adjacency information for each cell configuration. During the surface extraction we evaluate this information to gather the 1-ring neighborhood directly.
In many application domains surface extraction needs to response to sharp features. Usually, features extraction is carried out by minimizing the quadric error function (cf. Section 2). Due to the limited register count on recent parallel hardware this complex computation leads to performance penalties. Our solution is based on the quadric error metric [GH97]. In comparison to common approaches, our algorithm does not achieve the same quality but is less computation complex. Since we merely want to preserve the volume of the terrain, this is not problematic.

The last component - quality enhancement - (optionally) applies smoothing operators and detail tessellation to the surface mesh. Usually, in order to smooth a mesh, a discrete *laplace operator* is used (cf. [WMKG08]). For quadrilateral meshes Zhang et al. [ZHK04] proposes (a) the use of surface diffuse flow for smoothing and (b) to add tangent movement for quality improvements. Due to the mesh representation, both can be implemented efficiently in parallel.

For detail tessellation we follow the real-time capable approach of Loop and Schaefer [LS08] and approximate the smooth *Catmull Clark subdivision* surface with *bicubic patches* in combination with displacement maps (via tessellation shader). For the displacement maps and the texture maps we use *tri-planar texture mapping* [Gei07, GT07]. Figure 5 shows some results.

Our main contribution is the novel surface extraction pipeline. The different stages and a detailed description of the processing scheme is discussed in the following Section.

4. Parallel Dual Marching Cube on Regular Grids

The surface extraction pipeline can be outlined as follows (cf. Figure 2): In a first step - the *Block Selection* - we select all relevant blocks in the volume. In the *Classification* we determine the cell case for all cells with regard to the isovalue. We use this information in subsequent stages to efficiently gather per cell information from static lookup tables. Then, the *Indexing* follows. Here we count the number of generated vertices/faces. The total count is used to allocate the required memory for the mesh representation. To produce a connected mesh we need to map relevant cells to unique locations in the mesh buffers. For this purpose we use dynamic lookup tables. The tables store a unique vertex/face index for each relevant cell. As a result we can refer to vertices/faces by the cell index. At this point all information has been gathered to produce the final mesh. To ensure a workload efficient generation of vertices and faces we need to process only relevant cells. These cells are selected in the *Cell Section* stage. As a result, we switch from a block-based processing to a cell-based processing for the subsequent stages. During the *Mesh Generation* we generate the vertex attributes, gather the 1-ring vertex neighbors, and generate the faces.

In the following we first introduce the terminology and some basic definitions in Section 4.1. The static lookup tables are described in the Section 4.2 and the dynamic lookup tables in Section 4.3. The different stages are described in detail in Section 4.4. In Section 4.5 we discuss our QEM-based vertex relocation approach.

4.1. Terminology

According to [Nie04] we define the input for our algorithm as a regular grid, i.e. a three dimensional grid $L = \{(i\triangle x, j\triangle y, k\triangle z) : i = 0,\ldots,N_x; j = 0,\ldots,N_y; k = 0,\ldots,N_z\}$ of values: $v_{i,j,k} = f(i\triangle x, j\triangle y, k\triangle z)$ representing a uniformly sampled function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$. The normal at a particular point is equal to the gradient direction, which is defined by the partial derivation: $\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})$. A (cubic) cell (i, j, k) is defined by the diagonal from $(i\triangle x, j\triangle y, k\triangle z)$ to $([i+1]\triangle x, [j+1]\triangle y, [k+1]\triangle z)$. With respect to an isovalue α, we can classify the grid points

$v_{i,j,k} > \alpha$ as *inside* and the grid points $v_{i,j,k} <= \alpha$ as *outside*. This classification yields 2^8 possible cases grouped into 23 canonical configurations for a single cell. These 23 configurations provide the vertices which need to be generated for a particular case (see [Nie04]). We denote these vertices as *cell vertices*. Each cell vertex is associated with a unique set of edges intersecting the isosurface. Interior edges are shared by four adjacent cells. Consequently, an edge is associated with exactly four vertices forming a quadrilateral (cf. [Nie04]). Intersection of a cell edge and the isosurface can be determined by an appropriate *root finding* algorithm. We denote these intersection points and the corresponding normals as pairs $(p_i, \mathbf{n}_i)$.

The generated mesh is represented by two buffers residing in the global memory of the processing device (GPU). The *vertex buffer* stores the attributes for each vertex. The *face buffer* stores the quadrilateral faces by indexing the vertex buffer. Both buffers are tightly packed. Stage execution on the processing device as well as memory allocations are invoked by the host (CPU).

4.2. Static Lookup Tables

To improve the performance we precompute particular cell information for the different cell cases. This information is provided by the following static lookup tables:

Vertex-Count Table: maps a cell case to the particular number of cell vertices.

Vertex-Edges Table: maps a cell case and a particular cell vertex to its unique set of associated active edges. Such an edge set can be efficiently represented by a 12 bit vector (1 bit per edge). We use the table to determine the intersection points of a cell with the isosurface.

Edge-Vertex Table: maps a cell case and a particular edge to its associated cell vertex. We encode this association into a 24 bit vector with 2 bits per edge to identify the cell vertex. We use this table to get the cell vertex for a particular edge.

Vertex-Adjacencies Table: maps a cell case and a particular cell vertex to a set of tuples $(\mathbf{V}_{zyx}, e)$ identifying the adjacent vertices with $\mathbf{V}_{zyx}$ the offset to the adjacent cell and e the associated edge. The tuples are sorted in 1-ring order.

Face Table: maps a cell case to a set of faces which are generated by a cell (see Section 4.4).

The first three tables can be directly derived from the Marching Cubes tables. The construction of the vertex-adjacencies table requires more effort and can be summarized as follows: For each possible case and each particular cell vertex we iterate through the set of intersecting edges. For each edge we collect the adjacent vertices building a quadrilateral. Vertices are declared in the form $(\mathbf{V}_{zxy}, e)$ where $\mathbf{V}_{zxy}$ specifies the offset to the neighbor cell and e the associated edge. We sort the quadrilaterals in such a way that adjacent ones are in successive order and collect the 1-ring from that sequence.

4.3. Dynamic Lookup Tables

Communication between stages is carried out by a set of dynamic lookup tables, i.e. three dimensional arrays. Stages can read or write data to these tables. Usually, our tables provide cell related information and thus have a dimension of $(N_x - 1) \times (N_y - 1) \times (N_z - 1)$. The tables are essential to uncouple the different stages. For our processing pipeline we rely on the following two tables:

Case Table: provides for each cell in the volume the corresponding cell case. This table is filled in the classification stage and utilized in all subsequent stages.

Index Table: provides for each cell in the volume the corresponding index of the first face and vertex with respect to the face/vertex buffer. We generate this table in the indexing stage. The table allows to identify cell related vertices and faces in the mesh buffers. For a cell (i, j, k) the associated index $I_{i,j,k}$ (from the index table) and the x-th cell vertex, we define the location of that vertex in the vertex buffer as: $I_{i,j,k} + x$. $I_{i,j,k}$ is generated by an walk over all cells (see Section 4.4) and refers to the first vertex which is generated by the cell (i, j, k). Indices for the face buffer are computed analogously.

The tables double the memory effort with regard to the volume, but in this way mesh quality as well as interactive frame rates are ensured. Moreover, the overhead can be minimized by virtualizing the access and only allocate memory for relevant parts of the volume (cf. [M*08]).

4.4. Pipeline Stages

Block Selection: From the model generation (see Section 3) we get a block-based decomposition of the volume and the *min* and *max* density for each block. First, we mark all relevant blocks as selected by simply evaluate the condition: $(min <= isovalue < max)$. We gather the selected blocks by storing the corresponding block indices in a appropriate buffer. Both is performed in parallel. The resulting buffer acts as input for the classification.

Classification: We determines for each cell i, j, k the configuration case by inspecting the 8 corner values. The particular cases are stored in the case table (cf. Section 4.3). To reduce value fetches we map one block to one thread group and load the grid values for this block into the low latency local memory before analyzing the cells.

Indexing: The total count and the mapping can be efficiently realized by a parallel *all-prefix-sum* (aka scan) algorithm [SHG08]. A scan transforms an array $A = \{a_0, \ldots, a_{n-1}\}$ to an array $A' = \{I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})\}$ where I is the identity of the binary associative operator $\oplus$. The transformed array A' represents the mapping to the buffers. The total buffer sizes are given by summing the last element of A and A'. The total count is transferred to the host which allocates the mesh buffers.

```
0000:
0001: (V000, 6),(V010, 4),(V110, 0),(V100, 2)
0010: (V000, 5),(V100, 1),(V101, 3),(V001, 7)
0011: (V000, 6),(V010, 4),(V110, 0),(V100, 2) &
      (V000, 5),(V100, 1),(V101, 3),(V001, 7)
0100: (V000,11),(V001,10),(V011, 8),(V010, 9)
0101: (V000, 6),(V010, 4),(V110, 0),(V100, 2) &
      (V000,11),(V001,10),(V011, 8),(V010, 9)
0110: (V000, 5),(V100, 1),(V101, 3),(V001, 7) &
      (V000,11),(V001,10),(V011, 8),(V010, 9)
0111: (V000, 6),(V010, 4),(V110, 0),(V100, 2) &
      (V000, 5),(V100, 1),(V101, 3),(V001, 7) &
      (V000,11),(V001,10),(V011, 8),(V010, 9)
1000: (V000, 6),(V100, 2),(V110, 0),(V010, 4) &
      (V000, 5),(V001, 7),(V101, 3),(V100, 1) &
      (V000,11),(V001,10),(V011, 8),(V010, 9)
1001: (V000,11),(V010, 9),(V011, 8),(V001,10) &
      (V000, 5),(V001, 7),(V101, 3),(V100, 1)
1010: (V000,11),(V010, 9),(V011, 8),(V001,10) &
      (V000, 6),(V100, 2),(V110, 0),(V010, 4)
1011: (V000,11),(V010, 9),(V011, 8),(V001,10)
1100: (V000, 5),(V001, 7),(V101, 3),(V100, 1) &
      (V000, 6),(V100, 2),(V110, 0),(V010, 4)
1101: (V000, 5),(V001, 7),(V101, 3),(V100, 1)
1110: (V000, 6),(V100, 2),(V110, 0),(V010, 4)
1111:
```

Figure 3: *The face table provides the quadrilateral faces that need to be generated for the 4 classified corner values (cell-case is marked red). A tuple $(\mathbf{V}_{zyx}, e)$ identifies the offset to the adjacent cell and the corresponding (local) edge.*

Cell Selection: During indexing we count the number of relevant cells and compute successive indices for these cells. We allocate an appropriate buffer and write the indices (i, j, k) of relevant cells to this buffer. This buffer is passed to the mesh generation process. As a consequence, we switch from a block-based to a cell-based processing. This significantly accelerates the mesh generation.

Mesh Generation: Mesh generation is carried out in two phases. In the first phase we determine the vertex position. For each cell we relocate the vertices as describe in Section 4.5. The storage location for the vertices in the mesh buffers are determined via the dynamic lookup tables (see Section 4.3). The 1-ring neighbors are gathered by using the vertex-adjacencies table (see Section 4.2).

The second phase is the face generation which rely on the parallel processing scheme of [SDC09]. Each cell generates a particular set of faces. Each grid point $L_{i,j,k}$ has exactly three unique edges (x, y, and z axis). For a cell this means that we choose three edges sharing the same origin. Three edges result in four grid values to be classified and 16 possible configurations. Also, an edge intersecting the isosurface generates one quadrilateral face. Based on this consideration Schmitz et al. propose a face table for Dual Contouring where the entries refer to adjacent cells. In our case, however, a cell can contain more than one vertex. We, thus, need to specify a particular cell vertex. Our vertex-edges table provides for each cell vertex a associated set of edges (see Section 4.2). Yet, this association is local only. Means that for an edge e in cell i, j, k we need to find the corresponding edge e' in the adjacent cell. This correspondence is provided by our extended face table (see Figure 3).

The global index of the vertex identified by the tuple $(\mathbf{V}_{zyx}, e)$ is resolved as follows: We fetch the case of the adjacent cell $\mathbf{V}_{zyx}$ (case table) and use that case to get the index of the cell vertex associated with the edge e (edge-vertex table). Then, we retrieve the vertex index from the dynamic lookup tables as described in Section 4.3.

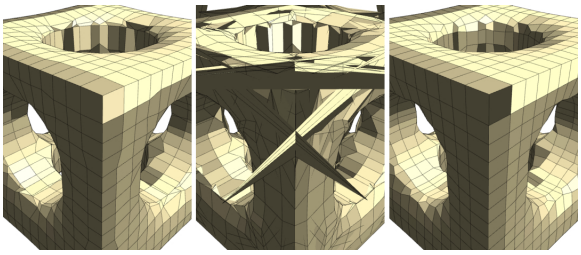

Figure 4: *Close-up of the isosurface defined by the intersection of a cube and sphere. The images illustrate the results when minimizing the QEF using (left) the pseudoinverse and (middle) QEM. Due to our supporting planes (right) QEM minimization is stable and produces accurate results.*

4.5. Vertex Relocation

There exist a couple of techniques for positioning a vertex. For sharp features, vertices are relocated at the minimizer of the QEF (cf. 2). Our solution is based on the *Quadric Error Metric* (QEM) [GH97] which represents the QEF as a sum of individual plane quadrics. Given a plane $\mathbf{n}^T \mathrm{v} + d = 0$ a *quadric* Q is written in its fundamental form as: $Q(\mathrm{v}) = \mathrm{v}^T A \mathrm{v} + 2\mathbf{b}^T \mathrm{v} + c$ with $A = \mathbf{n}\mathbf{n}^T$ a 3×3 matrix, $\mathbf{b} = d\mathbf{n}$ a three dimensional vector and $c = d^2$ a scalar value. The minimizer is given by $\bar{\mathrm{v}} = -A^{-1}\mathbf{b}$. Although QEM is computational more efficient in contrast to other QEF solver (cf. [SW02]) we need to handle numerical instabilities (see Figure 4).

We address this problem by adding *supporting planes* which stabilize the minimization process. Given the intersection point pairs $(\mathrm{p_i}, \mathbf{n}_i)$ we first compute the quadric Q_I representing the corresponding tangent planes and $\mathrm{p_{mass}}$ the *mass point* and its normal $\mathbf{n}_{mass}$, i.e. the average of the intersection points. For each intersection point we construct a supporting plane $\mathbf{n}_s^T \mathrm{v} + \mathbf{n}_s^T \mathrm{p_{mass}} = 0$ with $\mathbf{n}_s = \frac{\mathbf{n}_e \times \mathbf{e}}{|\mathbf{n}_e \times \mathbf{e}|}$ where $\mathbf{e} = \frac{\mathrm{p_{mass}} - \mathrm{p_i}}{|\mathrm{p_{mass}} - \mathrm{p_i}|}$ and $\mathbf{n}_e = \frac{\mathbf{n}_{mass} + \mathbf{n}_i}{|\mathbf{n}_{mass} + \mathbf{n}_i|}$. Please note that this is only true if $\mathbf{e}$ is non-zero and is not oriented in $\mathbf{n}_e$ direction.

The set of supporting planes is expressed as quadric Q_S. The final vertex location is computed by minimizing against the quadric $Q = Q_I + aQ_S$ with $a \in [0..1]$ scaling the contribution of the supporting planes. The factor controls the sharpness of the features: small values produce relatively sharp features but may introduce instability. For our tests we used a factor of 0.5 which leads to good results.

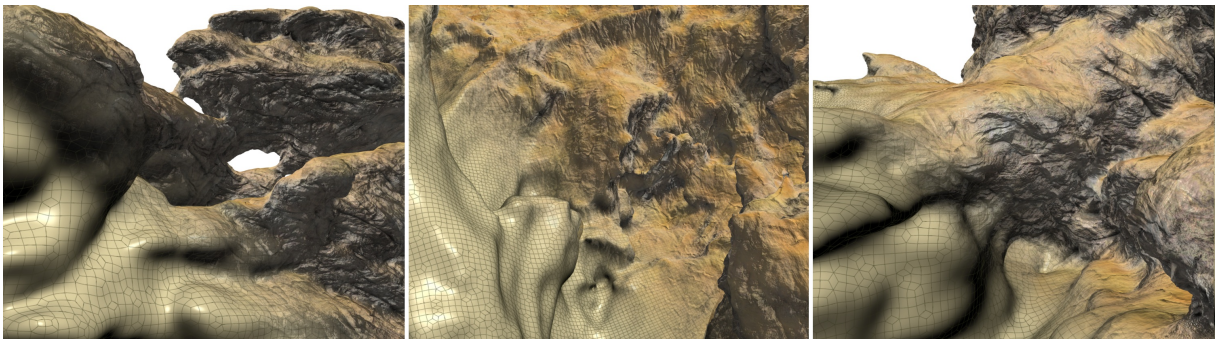

Figure 5: *Illustrates snapshots of our interactive procedural complex terrain modeling prototype. Model generation, surface extraction, smoothing and detail tessellation is performed on runtime.*

5. Experimental Results and Discussion

Our approach has been prototypically implemented with C++, OpenCL and OpenGL. The processing pipeline is entirely realized as OpenCL program whereby each stage is implemented as *kernel*. The GPU is used as processing device. All data, including the tables and the volume, are hold entirely in the device memory. The mesh buffers are provided by OpenGL. The smoothing is implemented in OpenCL, too. Detail tessellation is realized via GLSL tessellation shader. Our prototype allows for modeling complex terrain isosurfaces (volumes). Noise parameters, blending factors, and isovalues can be changed interactively. Figure 5 gives some examples.

In the following we describe the results of a series of experiments. All measurements were performed on a PC with Intel Core i7-860 (4 GB main memory) and with an AMD Radeon HD 5870 (2 GB memory) graphics adapter. For all tested models we average the result of a sequence of 100 extractions. We measure the timings for each individual stage and the total time for each iteration. In Table 1 we provide the results for the different models. Interactivity is achieved in all cases ($>$ 25 extractions per second). Consequently, for a typical modeling session (volume size: 512x512x128) we are able to achieve interactive frame rates. Our analysis reveals that - independent from model size - the main extraction stages (classification, indexing, relocation and face generation) consume $\approx$ 80% of the total time (cf. Figure 6). The remaining time is used for block as well as cell selection. In conjunction with the linear relationship between number of vertices and extraction time we are able to conclude that the processing is workload-efficient.
The timings for smoothing depend on the extracted surface complexity. For moderate model sizes, the smoothing can be applied directly after the extraction process. This has only a slight impact on the overall performance. However, interactivity is disturbed when having a high number of iterations ($>$ 10). The isosurfaces in Figure 5 have been smoothed with 5 iterations.

Model	#voxels	α	time	#faces	#verts
Terrain	1.0 M	0.48	4.8	52.5 K	53.3 K
	8.4 M		11.6	217.7 K	219.5 K
	33.6 M		27.9	601.2 K	603.2 K
	67.1 M		33.1	885.2 K	888.7 K
Bonsai	256^3	0.15	23.7	471.3 K	481.9 K
Engine	256^3	0.19	16.0	307.8 K	307.7 K

Table 1: *Experimental results of our algorithm for different test models. The table provides the size of the volume (#voxels), the isovalue α, the extraction time in ms, the number of generated faces (#faces) and the number of relocated vertices (#verts). Terrain sizes resulting in the given voxel count are:* $128 \times 128 \times 64$, $256 \times 256 \times 128$, $512 \times 512 \times 128$ *and* $512 \times 512 \times 256$.

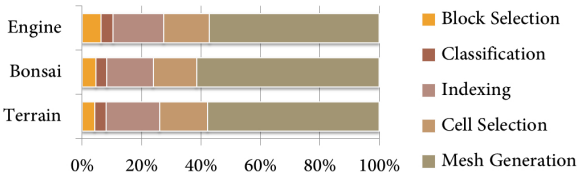

Figure 6: *Relative timings for the pipeline stages with regard to the total time.*

The approximation of the smooth subdivision surface and the displacement mapping merely effects the rendering time. This is based on the fact that bicubic patches are computed in the tessellation shader. We have not implemented an adaptive tessellation scheme yet. Thus, for large models, we are not able to guarantee interactivity when tessellation is turned on.

The results confirm a good performance. But in contrast to other GPU-based approaches our solution is much more complex. Consequently, we will never measure up to such techniques in terms of performance. However, in terms of

quality we achieve very good results. For instance, the quadrangulation created by our technique requires no or negligible downstream processing.

We have also tested the algorithm with the main processor (CPU) as processing device. The result was disappointing. For example, the processing of the Engine data-set took more than 3 seconds. Our first OpenMP-optimized prototype took merely $\approx 0.8s$. The analysis of the stages shows that indexing has consumed more that 70% of the total time. We conclude that the indexing strategy (parallel scan) works fine for GPU architectures, but not for CPUs.

6. Conclusion

We have presented a parallel surface extraction approach which generates smooth high-quality isosurfaces. Our novel parallel processing pipeline is based on Dual Marching Cubes. We guarantee manifold surfaces, full connectivity information and directly provide the 1-ring vertex neighborhood. A feature preserving smoothing operator improves mesh quality. Visual fidelity is enhanced by approximating the smooth subdivision surface and by using displacement mapping. As a result we are able to interactively generate and display procedural terrains at a high visual quality.
We see the scope of future work by finding a generic solution which performs well on CPU and GPU architectures. We have not addressed very large volumes or level of detail, yet. Last, the usefulness of our approach for other domains, e.g. volume visualization, needs to be investigated.

References

[BFO*07] BEARDALL M., FARLEY M., OUDERKIRK D., SMITH J., JONES M., EGBERT P.: Goblins by Spheroidal Weathering. In *Eurographics Workshop on Natural Phenomena* (2007), Eurographics Association, pp. 7–14. 1

[Blo94] BLOOMENTHAL J.: Graphics gems iv. Academic Press Professional, Inc., 1994, ch. An Implicit Surface Polygonizer, pp. 324–349. 3

[Gei07] GEISS R.: *GPU Gems 3: Generating Complex Procedural Terrains Using the GPU*. Addison-Wesley, 2007, pp. 7–37. 1, 3, 4

[GH97] GARLAND M., HECKBERT P.: Surface Simplification Using Quadric Error Metrics. In *Proc. of the 24th annual conference on Computer Graphics and Interactive Techniques* (1997), ACM, p. 216. 2, 4, 6

[GT07] GEISS R., THOMPSON M.: NVIDIA Demo Team Secrets Cascades. Presentation at Game Developers Conference, 2007. 3, 4

[Har98] HARTMANN E.: A Marching Method for the Triangulation of Surfaces. *The Visual Computer 14*, 3 (1998), 95–108. 3

[JLSW02] JU T., LOSASSO F., SCHAEFER S., WARREN J.: Dual Contouring of Hermite Data. *ACM Transactions on Graphics 21*, 3 (2002), 339–346. 2

[KBSS01] KOBBELT L., BOTSCH M., SCHWANECKE U., SEIDEL H.: Feature Sensitive Surface Extraction from Volume Data. In *Proc. of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 57–66. 2

[LC87] LORENSEN W. E., CLINE H. E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proc. of the 14th annual Conference on Computer Graphics and Interactive Techniques* (1987), ACM, pp. 163–169. 2

[LHMG02] LABSIK U., HORMANN K., MEISTER M., GREINER G.: Hierarchical Iso-Surface Extraction. *Journal of Computing and Information Science in Engineering (Transactions of the ASME) 2*, 4 (2002), 323–329. 2

[LMS11] LÖFFLER F., MÜLLER A., SCHUMANN H.: Real-time Rendering of Stack-based Terrains. In *Proc. of the Vision, Modeling, and Visualization Workshop* (2011), Eurographics Association, pp. 161–168. 3

[LS08] LOOP C., SCHAEFER S.: Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Transactions on Graphics 27*, 1 (2008), 8. 2, 3, 4

[M*08] MITTRING M., ET AL.: Advanced virtual texture topics. In *ACM SIGGRAPH 2008 classes* (2008), ACM, pp. 23–51. 5

[NB93] NING P., BLOOMENTHAL J.: An Evaluation of Implicit Surface Tilers. *Computer Graphics and Applications 13*, 6 (Nov. 1993), 33–41. 2

[NH91] NIELSON G., HAMANN B.: The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes. In *IEEE Conference on Visualization* (1991), pp. 83–91. 2

[Nie03] NIELSON G.: On Marching Cubes. *IEEE Transactions on Visualization and Computer Graphics 9*, 3 (2003), 283–297. 2

[Nie04] NIELSON G.: Dual marching cubes. In *Proc. of the conference on Visualization* (2004), IEEE, pp. 489–496. 1, 3, 4, 5

[NY06] NEWMAN T., YI H.: A Survey of the Marching Cubes Algorithm. *Computers and Graphics 30*, 5 (2006), 854–879. 2

[SDC09] SCHMITZ L., DIETRICH C., COMBA J.: Efficient and High Quality Contouring of Isosurfaces on Uniform Grids. In *Computer Graphics and Image Processing* (2009), IEEE, pp. 64–71. 3, 6

[SHG08] SENGUPTA S., HARRIS M., GARLAND M.: Efficient Parallel Scan Algorithms for GPUs. *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003* (2008). 5

[SJW07] SCHAEFER S., JU T., WARREN J.: Manifold Dual Contouring. *IEEE Transactions on Visualization and Computer Graphics 13*, 3 (2007), 610–619. 3

[SSS06] SCHREINER J., SCHEIDEGGER C., SILVA C.: High-Quality Extraction of Isosurfaces from Regular and Irregular Grids. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1205–1212. 3

[SW02] SCHAEFER S., WARREN J.: Dual Contouring:" The Secret Sauce". 2, 6

[SW04] SCHAEFER S., WARREN J.: Dual Marching Cubes: Primal Contouring of Dual Grids. In *Proc. of the Computer Graphics and Applications, 12th Pacific Conference* (2004), IEEE, pp. 70–76. 2, 3

[TSD07] TATARCHUK N., SHOPF J., DECORO C.: Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline. In *SIGGRAPH courses* (2007), ACM, pp. 122–137. 3

[WMKG08] WARDETZKY M., MATHUR S., KÄLBERER F., GRINSPUN E.: Discrete Laplace operators: no free lunch. In *SIGGRAPH ASIA Courses* (2008), ACM, p. 19. 4

[ZHK04] ZHANG N., HONG W., KAUFMAN A.: Dual Contouring with Topology-Preserving Simplification Using Enhanced Cell Representation. In *Proc. of the conference on Visualization* (2004), IEEE, pp. 505–512. 3, 4