HistoPyramids in Iso-Surface Extraction

Christopher Dyken Gernot Ziegler

Christian Theobalt Hans-Peter Seidel

MPI-I-2007-4-006 August 2007

Authors'Addresses

Christopher Dyken
Department of Informatics
University of Oslo
PO Box 1080 Blindern
N-0316 Oslo
Norway

Gernot Ziegler
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
D-66123 Saarbrücken
Germany

Christian Theobalt
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
D-66123 Saarbrücken
Germany

Hans-Peter Seidel
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
D-66123 Saarbrücken
Germany

Abstract

We present an implementation approach to high-speed Marching Cubes, running entirely on the Graphics Processing Unit of Shader Model 3.0 and 4.0 graphics hardware. Our approach is based on the interpretation of Marching Cubes as a stream compaction and expansion process, and is implemented using the HistoPyramid, a hierarchical data structure previously only used in GPU data compaction. We extend the HistoPyramid structure to allow for stream expansion, which provides an efficient method for generating geometry directly on the GPU, even on Shader Model 3.0 hardware. Currently, our algorithm outperforms all other known GPU-based iso-surface extraction algorithms. We describe our implementation and present a performance analysis on several generations of graphics hardware.

Keywords

GPU, graphics hardware, marching cubes, iso-surface extraction, HistoPyramids, volume rendering, voxelization

HistoPyramids in Iso-Surface Extraction

Christopher Dyken[1,2], Gernot Ziegler[3], Christian Theobalt[3] and Hans-Peter Seidel[3]

[1] Department of Informatics, University of Oslo, Norway
[2] Centre of Mathematics for Applications, University of Oslo, Norway
[3] Max-Planck-Institut für Informatik, Germany

Abstract

We present an implementation approach to high-speed Marching Cubes, running entirely on the Graphics Processing Unit of Shader Model 3.0 and 4.0 graphics hardware. Our approach is based on the interpretation of Marching Cubes as a stream compaction and expansion process, and is implemented using the HistoPyramid, a hierarchical data structure previously only used in GPU data compaction. We extend the HistoPyramid structure to allow for stream expansion, which provides an efficient method for generating geometry directly on the GPU, even on Shader Model 3.0 hardware. Currently, our algorithm outperforms all other known GPU-based iso-surface extraction algorithms. We describe our implementation and present a performance analysis on several generations of graphics hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

Iso-surfaces of scalar fields are used in a wide range of applications. In particular, scalar fields represented as three-dimensional grids of scalar values are very common, e.g. in medical imaging, geology, and computational geometry. However, the number of elements in such a grid grows to the power of three with respect to sample density, and therefore the sheer amount of data associated is already a computational burden. Thus, any operation dealing with volumetric grids of data puts tough requirements on CPU memory bandwidth and processing power.

The Graphic Processing Units (GPUs) of graphics cards are designed for huge computational tasks with large requirements for memory bandwidth, building on simple and massive parallelism instead of the CPU's more sophisticated serial processing. Not unexpectedly, there has been a lot of interest in various volume-data processing algorithms on graphics hardware. For example, volume ray-casting is a visualization technique for scalar fields that has been successfully implemented on GPUs.

However, applications such as for example surface fairing, free-form modeling, and simulations usually needs an explicit representation of the iso-surface. Marching Cubes (MC) is an efficient algorithm for extracting explicit iso-surfaces from volumetric grids of scalar values. In this paper we present a formulation of the MC algorithm suitable for GPU implementation. The formulation is a novel, yet well-investigated approach to MC on both Shader Model 3 (SM3) hardware (hardware prior to the introduction of geometry shaders) and on Shader Model 4 (SM4) [LB07] hardware (hardware with geometry shaders). It outperforms the known SM4-based geometry-shader approaches, yet requires hardly more implementation effort. Since the geometry is extracted directly by the graphics hardware and is independent of view-point, we can store it in graphics memory and use it for low-burden rendering, or it could be fed directly into a particle system running on the GPU and be used to spawn particles evenly over the iso-surface. In Figure 1 we used such a particle system to set the surface "on fire", without any transfer of geometry to or from the CPU.

The main element of our approach is the *Histogram Pyramid (short: HistoPyramid)*, a hierarchical data structure recently introduced in GPU programming [ZTTS06]. The local nature of its associated algorithms allow for parallel data expansion and compaction, which has traditionally been

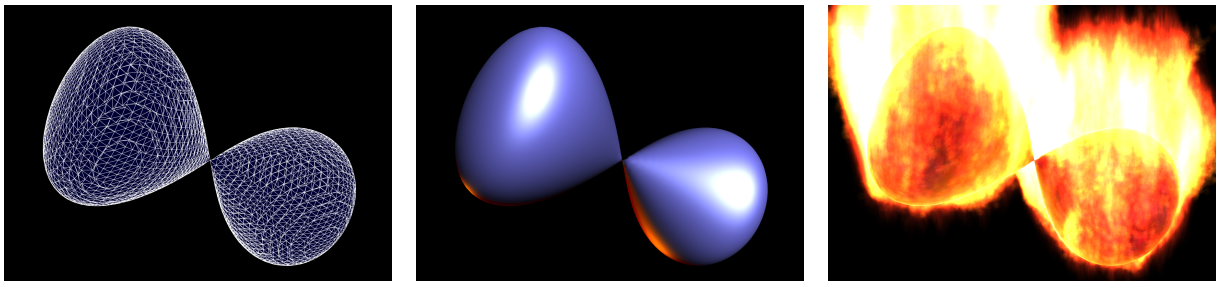

Figure 1: *Many applications require explicit iso-surfaces from scalar fields, e.g. in the form of a list of triangles (left). Such an explicit representation can be used to visualize the iso-surface (middle), or, for example, be used by the geometry shader to spawn particles evenly over the iso-surface (right). In all three images, the GPU has autonomously extracted the mesh from the scalar field, where it is kept in graphics memory to source subsequent rendering passes.*

seen as a hard task for stream processors. Here, it is put to use for volume analysis and, in one of the two presented algorithmic variants, even for generating the output geometry — directly on graphics hardware.

We begin by describing the general strategy of HistoPyramids in Section 3, and continue with approaches to an OpenGL implementation on SM3 and on SM4 hardware. Please note that we describe the HistoPyramid technique for data compaction and expansion of cells stored in *2D textures*, and therefore use the term *texel* for single data elements. But, as already pointed out in [ZTTS06], this does not restrict the algorithm to being used on 2D arrays only. With the according mapping, arrays of *any* dimensionality can be processed. As an example, in our particular context, we process a 3D array by mapping it into the 2D domain, where its cells can temporarily be regarded as texels.

Please do also note that instead of using the term voxels, we use the term *Marching Cube cell* (or: *MC cell*) for the 3D cells of geometry stemming from the input voxel data.

2. Previous and related work

In the last years, iso-surface extraction algorithms for voxel data on stream processor architectures (like GPUs) have been a topic of intensive research. Even though the particular computations for each MC cell can be done in parallel, the variable amounts of triangles produced have to be merged into one continuous sequence of triangles. This is not trivial to do in parallel, and poses a major performance problem for iso-surface extraction on the GPU.

Graphics hardware prior to SM4, which introduced geometry shaders, lacked functionality for this kind of stream compaction and expansion. The graphics hardware couldn't produce triangles directly, and the triangle data had to be provided by the CPU or streamed off a vertex buffer object. Also, the only method for discarding triangles in the graphics pipeline was to let the triangles be culled right before rasterization. A straight-forward approach to stream compaction and expansion was thus to assume a fixed number of triangles for each 3D cell and let the superfluous output elements be discarded by degenerating the corresponding triangles to points in the vertex shader.

Marching Tetrahedra (MT) is particularly suitable in such a setting. While MC produces up to four (original triangle table [LC87]) or five (modified triangle table [MSS94]) triangles per MC cell, MT only produces at maximum two per MT tetrahedron. In addition, each MC cell needs eight scalar values to classify its cubic interior, while a MT tetrahedron only requires the scalar values at its four corners. Thus, both the amount of inputs per element and the amount of fixed expansion per element is limited. Pascucci [Pas04], a pioneer of this technique, used triangle strips, arranged in a 3D space-filling curve, to feed as little geometry as necessary to the GPU. However, since one single cubic 3D cell has to be split into at least five MT tetrahedra, the total number of triangles is usually larger than for the result of MC.

Another prime example of GPU based MT is the work of Klein, Stegmaier, and Ertl [KSE04], which renders vertex arrays and peaks 7.7 million tetrahedra per second, rendering on an ATI Radeon 9800 Pro. Kipfer and Westermann [KW05] improved upon this by observing that edges are shared in-between tetrahedra, and thus should be used as the basic data structure in the evaluation. Further, Buatois [BCL06] used multiple stages and vertex texture lookups instead to reduce redundant calculations.

All of the mentioned algorithms suffer from the fact that the GPU cannot easily create or discard geometry. The fixed expansion causes a considerable amount of unnecessary vertex processing. Kipfer and Westermann [KW05] try to reduce this vertex processing load via an interval tree, by identifying the volume regions which produce geometry at all. However, this requires CPU-based pre-processing of the 3D cells. Another CPU-assisted method is given by Johannson [JC06], who circumvents the GPU geometry generation restriction by letting the CPU do the MC cell classifi-

cation and only feeds the GPU with an MC cell if it actually produces geometry. But Johannson also notes that this preprocessing on the CPU limits the speed of the algorithm.

Another drawback of these approaches is that none of the named algorithms is capable of creating a compact list of the iso-surface triangle mesh in GPU memory. Thus, part of or the whole of the algorithm has to be run each time the resulting triangle mesh is rendered. Also, it is difficult to use the set of triangles as input to other computations.

A first solution for GPU-based stream compaction is given in [Hor05]. It first uses a prefix sum method to generate the correct output indices for each input element. Afterwards, for each output element in the compacted output, it gathers the corresponding input element via a binary search, ignoring unnecessary elements in the process. Unfortunately, their approach has a complexity of $n\log(n)$, and does not perform well on large datasets. Prefix Sum (Scan) [Har07] improved on this, and provides an efficient implementation in CUDA. During a pyramid-like up-sweep and down-sweep phase, it creates, in parallel, a table that associates each input element with an offset in the output stream, at considerably reduced algorithmic complexity. Then, using this table and GPU scattering, it can directly place each relevant input element in the output list, while it ignores the irrelevant elements.

Another approach to data compaction is provided by Ziegler et.al. [ZTTS06] with the introduction of HistoPyramids, running entirely on the GPU of SM3 hardware. As demonstration, the algorithm is used to extract a compact sequence of points from volumetric data. Despite a more complicated gather process for the output elements, the HistoPyramid is surprisingly fast when a considerable amount of input elements shall be discarded. In [DRS07], Dyken et.al. use this HistoPyramid for creating a compact list of silhouette edges, and can thus reduce the amount of data transferred from GPU to CPU. The result is a silhouette detection algorithm faster than any CPU implementation for meshes larger than a few thousand triangles.

The introduction of SM4 hardware provided data compaction and expansion on the hardware level. The *geometry shader*, a new programmable stage in the graphics pipeline, has the ability to create and discard geometry on the fly. Uralsky [Ura06] present how MT can be implemented using geometry shaders, with an implementation given in the NVidia OpenGL SDK-10. We have benchmarked our approach against this implementation in Section 5. Another feature of SM4, the *transform feedback buffers*, provide a simple method for tapping into the pipeline before rasterization starts, and to record all the primitives submitted. In combination, these two new features can also create a compact sequence of triangles in GPU memory.

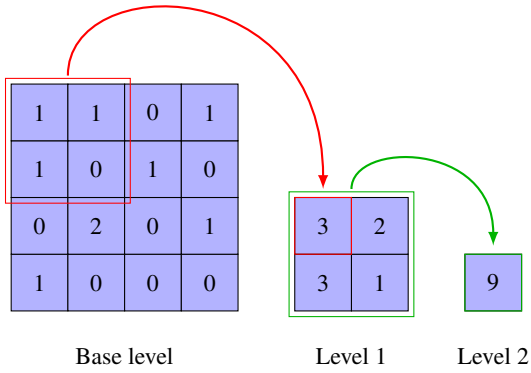

Figure 2: *Bottom-up build process, adding the values of four texels repeatedly. The top texel contains the total number of output elements in the pyramid.*

3. HistoPyramids

The core component of our MC implementation is the HistoPyramid algorithm, which is used to compact and expand data streams on the GPU. The input is a stream of data input elements called the *input stream*. Each input element may *allocate* a given number of elements in the output stream. If an input element allocates zero elements in the output stream, the input element is discarded and the output stream becomes smaller (data compaction). On the other hand, if the input element allocates more than one output element, the stream is expanded (data expansion). The input elements individual allocation is determined by a *predicate function*.

The HistoPyramid algorithm consists of two distinct phases. In the first phase, we create a HistoPyramid, a pyramid-like data structure very similar to a MipMap. In the second phase, we extract the output elements by traversing the HistoPyramid top-down to find the corresponding input elements. In the case of stream expansion, we also determine which copy of the input element we are currently generating.

3.1. Construction

The first step is to build the HistoPyramid. The HistoPyramid is a stack of 2D textures. At each level, the texture size is a quarter of the size of the level below, resembling the exact same layout as the MipMap pyramid of a 2D texture. We call the largest texture in the bottom of the stack the *base texture*, and the single texel of the 1×1 texture in the top of the stack the *top element*. Figure 2 show the levels of a HistoPyramid laid out from left to right. The number of texels in the base texture is the number of input elements the HistoPyramid can handle. For simplicity, we assume that the base texture is square and the size of the sides is a power of two (arbitrary sizes can be accommodated with suitable padding).

Each texel in the base layer corresponds to one input element, and contains the number of its allocated output elements. For example, in Figure 2 we have an input stream of 16 elements, laid out from left to right and top to bottom. Thus, elements number 0,1,3,4,6,11, and 12 have allocated one output element each (stream pass-through). Element number 9 has allocated two output elements (stream expansion), while the rest of the elements have none allocated (stream compaction). This rest will be discarded. To generate this allocation, we apply the predicate function to the base layer, letting the predicate function do the mapping from the dimension of the input stream (in our MC application, the input stream is 3D) to a 2D layout in the base layer.

The next step is to build the rest of the layers from bottom-up, layer by layer. According to the MipMap-principle, each texel in a level corresponds to four texels in the in the level below. This texel is assigned the sum of the four corresponding elements in the layer below. Even this is similar to the construction of MipMap pyramids, the only difference is that we *sum* the four elements instead of averaging them. The example in Figure 2 shows this process. The sum of the texels in the 2×2 block in the upper left of the base layer is three, and stored in the upper left texel of Level 1. The sum of the texels in the single 2×2 block of Level 1 is nine, and stored in the single texel of Level 2, which is the top element of the HistoPyramid.

We see that the computation of a texel in one level only depends on the texels in the previous one. This allows us to compute all texels in one level in parallel, without any data inter-dependencies.

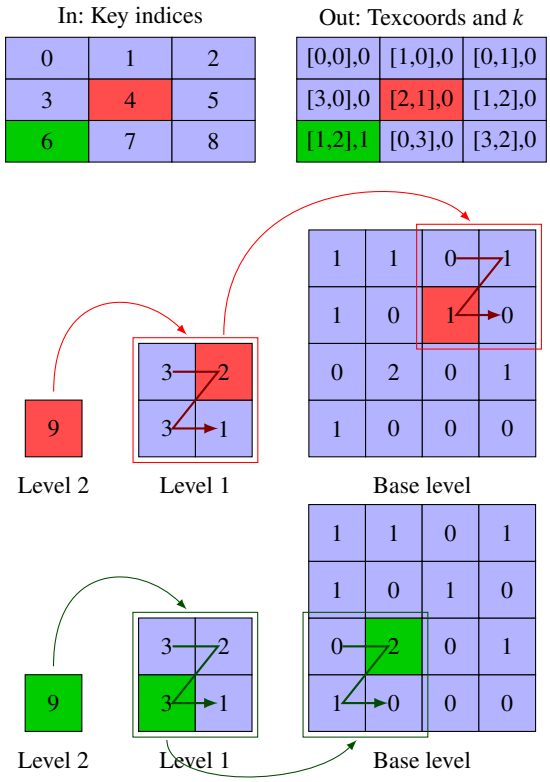

Figure 3: *Element extraction, interpreting partial sums as interval in top-down traversal. Red traces the extraction of key index 4 and green traces key index 6.*

3.2. Traversal

In the second phase, we generate the output stream. The number of output elements is given by the top element of the HistoPyramid. To produce the output stream, we iterate over the output elements and traverse the HistoPyramid once per output element. We enumerate the elements in the output stream, and use this enumeration as each element's *key index* k. The traversal requires several variables: We let m denote the number of levels in the HistoPyramid. The traversal maintains a texcoord $\mathbf{p}$ and a current level l, which always refer to a texel in the HistoPyramid. The traversal starts from the top level $l = m$ and goes recursively down, terminating at the base level $l = 0$. During traversal, k and $\mathbf{p}$ is continuously updated, and when the traversal terminates $\mathbf{p}$ points to a texel in the base layer. In the case of stream pass-through, k is always zero when the traversal terminates. However, in the case of stream expansion, the value in k determines which numbered copy of the input element this particular output element is.

Initially, $l = m$ and $\mathbf{p}$ points to the center of the single texel in the top level. We subtract one from l, descending one step in the HistoPyramid, and now $\mathbf{p}$ refers to the center of the 2×2 block of texels in level $m-1$ corresponding to the single texel $\mathbf{p}$ pointed to at level m. We label these four texels in the following manner,

$$\begin{matrix} a & b \\ c & d \end{matrix}$$

and use the values of these texels to form the four ranges A,B,C, and D, defined as

$$\begin{aligned} A &= [0\ ,\ a), \\ B &= [a\ ,\ a+b), \\ C &= [a+b\ ,\ a+b+c), \quad \text{and} \\ D &= [a+b+c\ ,\ a+b+c+d). \end{aligned}$$

Then, we examine which of the four ranges k falls into. If, for example, k falls into the range B, we adjust $\mathbf{p}$ to point to the center of b and subtract the start of the range, in this case a, from k. We then recurse by subtracting one from l and repeating the process until $l = 0$, when the traversal terminates. Then, from $\mathbf{p}$ we can calculate the index of the corresponding input stream element, and the value in k enumerates the copy.

Figure 3 show two examples of HistoPyramid traversal. The first example, labeled red, is of the key index $k = 4$, is a case of stream copy. We start at level 2 and descend to level 1. The four texels at level 1 form the ranges

$$A = [0\ ,\ 3), \quad B = [3\ ,\ 5), \quad C = [5\ ,\ 8), \quad D = [8\ ,\ 9).$$

We see that k is in the range B. Thus, we adjust the texcoord to point to the upper left texel and subtract 3 from k, which leaves $k = 1$. Then, we descend again to the base level. The four texels in the base level corresponding to the upper left texel of level 1 form the ranges

$$A = [0\ ,\ 0), \quad B = [0\ ,\ 1), \quad C = [1\ ,\ 2), \quad D = [2\ ,\ 2).$$

The ranges A and D are empty. Here, $k = 1$ falls into B, and we adjust $\mathbf{p}$ and k accordingly. Since we're at the base level, the traversal terminates, $\mathbf{p} = [2,1]$ and $k = 0$.

The second example of Figure 3, labeled green, is a case of stream expansion. Here the key index $k = 6$. We begin at the top of the HistoPyramid and descend to level 2. Again, the four texels form the ranges

$$A = [0\ ,\ 3), \quad B = [3\ ,\ 5), \quad C = [5\ ,\ 8), \quad D = [8\ ,\ 9),$$

And k falls into the range C. We adjust $\mathbf{p}$ to point to c and subtract the start of range C from k, resulting in $k = 1$. Descending, we inspect the four texels in the lower left corner of the base layer, which form the four ranges

$$A = [0\ ,\ 0), \quad B = [0\ ,\ 2), \quad C = [2\ ,\ 3), \quad D = [3\ ,\ 3),$$

Here, k falls into range B, and we adjust $\mathbf{p}$ and k accordingly. Since we're at the base level, we terminate the traversal, and have $\mathbf{p} = [1,2]$ and $k = 1$. This means that output element 6 is the second output element (since $k = 1$) of the input element corresponding to the position $[1,2]$ in the base texture.

The traversal only reads from the HistoPyramid and there are no data dependencies between traversals. Therefore, the output elements can be extracted in any order, for example, at the same time in parallel.

3.3. Comments

The 2D texture layout of the HistoPyramid fits graphics hardware very well. It can intuitively be seen that in the domain of normalized texcoord calculations, the texture fetches overlay with fetches from the layer below. This allows the 2D texture cache to assist HP traversal with memory prefetches, and thus increase its performance.

At each descend during the traversal, we have to inspect the values of four texels, which amounts to four texture fetches. However, since we always fetch 2×2 blocks, we can use a four-channel texture and encode these four values as a RGBA value. This halves the size of all textures along both dimensions, and lets us thus build four times larger HistoPyramids within the same texture size limits. In addition, since we quarter the number of texture fetches, and graphics hardware is quite efficient at fetching four-channel RGBA values, this usually yields a significant speed-up. For more details, see *vec4-HistoPyramids* in [ZTTS06].

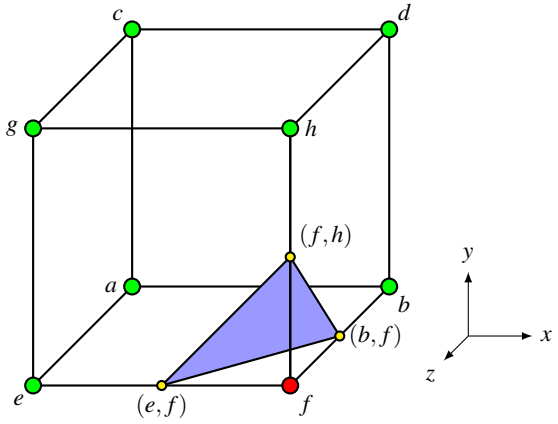

Figure 4: *A marching cube cell (MC cell). In this case, only the corner f is inside the iso-surface, and thus, the edges (e,f), (b,f), and (f,h) intersect the iso-surface.*

4. Marching cubes

The Marching Cubes (MC) algorithm [LC87] of Lorensen and Cline is probably the most used algorithm for extracting iso-surfaces from scalar fields. From a 3D grid of $N \times M \times L$ scalar values, we form a grid of $(N-1) \times (M-1) \times (L-1)$ cubic "MC cells", where the centers of the MC cells are in-between the position of the scalar values. Thus, each MC cell have a scalar value associated with each of its eight corners. The basic idea is to "march" through all the cells one-by-one, and for each cell, produce a set of triangles that approximates the iso-surface locally to that particular cell.

It is assumed that the topology of the iso-surface inside a MC cell is completely determined from classifying the eight corners of the MC cell as *inside* or *outside* the iso-surface, see Figure 4. Thus, the topology of the local iso-surface can be encoded into an eight-bit integer, which we call the *class* of the MC cell. For example, in Figure 4 the corner f is inside and the rest of the corners are outside. Encoding "inside" with 1 and "outside" with 0, we get the MC class %00000100 in binary notation, or 32 in decimal. If any of the twelve edges of the MC cell have one endpoint inside and one outside, the edge is said to be *piercing* since it intersects the iso-surface. The MC cell in Figure 4 have three piercing edges: (b,f), (e,f), and (f,h). The set of piercing edges is completely determined by the MC class of the cell.

For each piercing edge of a MC cell, we determine the *intersection point* where the iso-surface and the edge intersects. By triangulating the intersection points we get an approximation of the iso-surface inside the MC cell. And with some care, the triangles of two adjacent MC cells fits to-

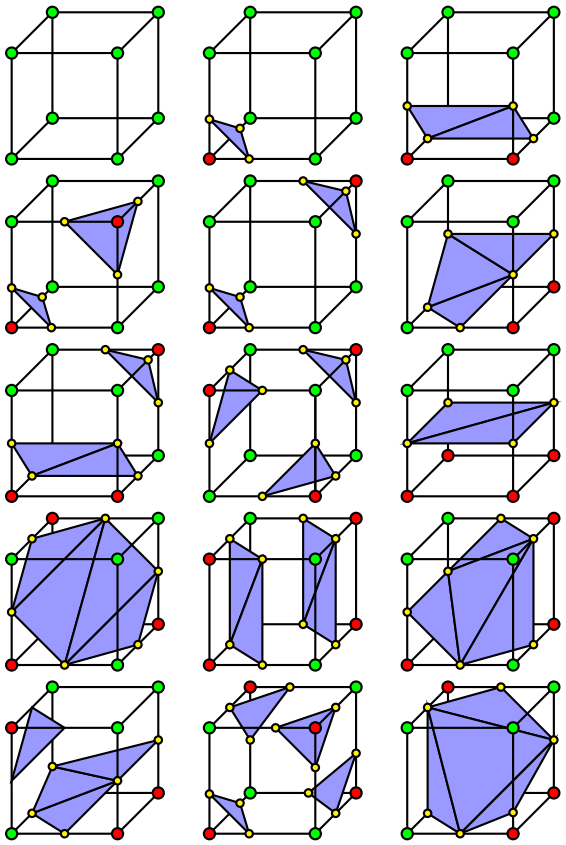

Figure 5: *The 15 basic predefined triangulations [LC87] of edge intersections, which by symmetry provide in principle triangulations for all 256 MC cases. However, a few of the cases are ambiguous, and we handle this by adding some extra triangulations [MSS94].*

gether. Since the intersection points only slides along the piercing edges, there are essentially 256 different triangulations we use, one for each MC class. From 15 basic predefined triangulations, depicted in Figure 5, we can create triangulations for all 256 MC classes du to inherent symmetry [LC87]. However, some of the MC classes are ambiguous, which may result in a discontinuous surface. Luckily, this is easily remedied by modifying the triangulations for some of the MC cases [MSS94]. On the downside, this also increases the maximum number of triangles emitted per MC cell from 4 to 5.

We only know the value of the scalar field at the end-point of the MC cell edges. Thus, to determine the intersection of a piercing edge and the iso-surface, the scalar field must be defined along the edges. The simplest choice is to let the intersection be at the midpoint of the edge. The result of this choice is depicted in the left part of Figure 6, which shows that this strategy leads to an excessively "blocky" appearance. A considerably better choice is to use a linear polynomial that interpolates the scalar values at the edge end-points to approximate the scalar field. By finding the intersection of the edge and this approximation, we get a considerably better estimate of the intersection. The result is shown in the left part of Figure 6.

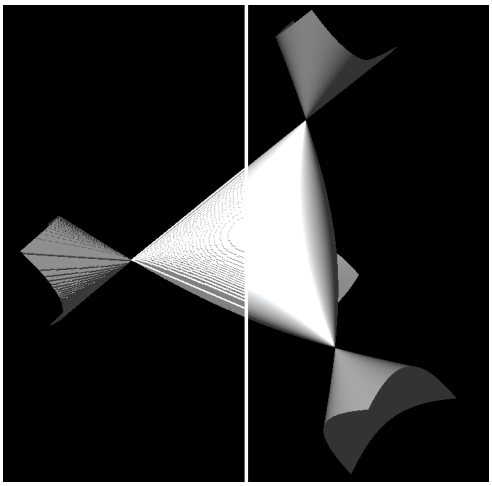

Figure 6: *Difference between assuming that edges pierce the iso-surface at the middle of an edge (left) or using an approximating linear polynomial to determine the intersection (right).*

4.1. Mapping Marching Cubes to Stream and HP Processing

Our approach is to implement MC as a sequence of data stream operations, with the input data stream being the cells of the 3D scalar field, and the output stream being a set of vertices, forming the triangles of the iso-surface. The data stream operations are executed via the HistoPyramid and (in one variant) the geometry shader, which compact and expand the data stream as necessary.

Figure 7 shows a flowchart of our algorithm. We use a texture to represent the scalar field, and the first step of our algorithm is to update this scalar field. The scalar field can stem from a variety of sources, and thus for example originate from storage or CPU memory, or simply be the result of GPGPU computations. For static scalar fields, this update is of course only conducted once.

The next step is to build the HistoPyramid. First, we build the base layer. Our predicate function maps base level texels to MC cells, and calculates the original 3D coordinates. Then, it samples the scalar field via these 3D texcoords to classify the MC cell corners. By comparing against the iso-level, it can determine which MC cell corners are inside or outside the iso-surface. This determines the MC class of the

cell and a lookup in the vertex count texture yields the number of vertices needed to triangulate this class. We store this number in the base layer, and can now proceed with HistoPyramid reductions to build the rest of the levels, as described in Section 3.1.

After the HistoPyramid has been completed, we read back the single texel on its top level. This makes the CPU aware of the required number of vertices needed for the complete iso-surface. Dividing this number by three yields the number of triangles.

The input to the render pass is a sequence of increasing key indices. The length of this sequence is the number of vertices in the iso-surface. For each vertex, we use the key index to conduct a HistoPyramid traversal, as described in Section 3.2. After the traversal, we have a texel position in the base texture and a key index remainder. From the texel position in the base texture, we can determine the corresponding 3D coordinate. Using the MC class of the cell and the key index remainder, we can do a lookup in the triangulation table texture, which is a 16×256 table where entry (i, j) tells which edge vertex i of a class j cell corresponds to. We then sample the scalar field at the two end-points of the edge, determine a linear interpolant of the scalar field along the edge, find the exact intersection, and emit the corresponding vertex.

In effect, the algorithm has transformed the stream of scalar field values into a stream of vertices, which can directly be used to render iso-surface geometry. Still, the geometry can be stored in a buffer on the GPU if so needed, either using transform feedback buffers or through a render-to-vertex-buffer pass.

4.2. Implementation details

In detail, the actual implementation of our MC approach contains some noteworthy caveats which we describe in this chapter.

The scalar field texture for the input data would ideally be stored as 3D texture. But since we want to generate its content on the GPU, and render-to-3D-texture is not yet possible, we map the 3D domain to a 2D tiling. For this purpose, we create a large tiled 2D texture, where each tile corresponds to a slice of the 3D volume, an approach known as a Flat 3D layout [HISL03]. The HistoPyramid algorithm performs better for large amounts of data. Therefore, we use the Flat 3D layout on the HistoPyramid base level as well, and process the entire volume using one HistoPyramid.

We used a four-channel HistoPyramid, where the RGBA-values of each base layer texel correspond to the analysis of a tiny $2 \times 2 \times 1$-chunk of MC cells. The analysis begins by fetching the scalar values at the common $3 \times 3 \times 2$ corners of the four MC cells. We compare these values to the iso-value to determine the inside/outside state of the corners, and from this determine the MC class of the MC cells. The MC class corresponds to the MC template geometry set forth by the Marching Cubes algorithm. It is needed in the extraction process, and therefore we use some of the bits in the base level texels to cache it. To do this, we let the vertex count be the integer part and the MC class the fractional part of a single float32 value. This is sound, as the maximum number of vertices needed by an MC class is 15, and therefore the vertex count only needs 4 of the 32 bits in a float32 value.

HistoPyramid texture building is implemented as consecutive GPGPU-passes of reduction operations, with special handling for the base level. By viewing it as reduction operation in a MipMap-like pyramid, the current texel receives the sum of the four corresponding texels in the level below. The top level contains exactly one entry. The reduction passes are mostly done as exemplified in "render-to-texture loop with custom MipMap generation" [JS05], but instead of using one single framebuffer object (FBO) for all MipMap levels, we use a separate FBO for each MipMap level, which gave a speedup on some hardware. Since we use a four-channel HistoPyramid, the top level texel actually contains four values, not only one, and the sum of these four values yields the number of vertices in the iso-surface. We read these four values back and add them on the CPU. We can then commence rendering of the iso-surface triangles.

Iso-surface rendering is triggered by processing the given number of vertices in the vertex shader. The only vertex attribute provided by the CPU is a sequence of key indices, streamed off a static vertex buffer object (VBO). In theory, on SM4 hardware, we could use the built-in `gl_VertexID` variable directly, but since OpenGL cannot trigger processing of vertices without any vertex attribute data, we would have to provide a VBO anyway. The vertex shader is executed for each vertex, using the provided key index to traverse the HistoPyramid, determining which MC cell and which of its edges this vertex corresponds to. It then samples the scalar field at both end-points of its edge, and forms a linear approximation which it intersects with the edge. The shader can also find an approximate normal vector for this point, which it does by interpolating the forward differences of the scalar field at each edge end. In this variant, it is thus the vertex-shader which generates the iso-surface on the fly.

Another approach to iso-surface extraction is to let the geometry shader generate the vertices required for each MC cell. In this variant, the HistoPyramid is only used for data stream compaction, discarding MC cells that do not intersect the iso-surface. After retrieving the number of geometry-producing MC cells from the top level of the HistoPyramid, the CPU triggers the geometry shader by rendering one point per geometry-producing MC cell. For each invocation, the geometry shader first traverses the HistoPyramid and determines which MC cell this invocation corresponds to. Then, based on the stored MC class, it emits the required vertices and, optionally, their normals by iterating through

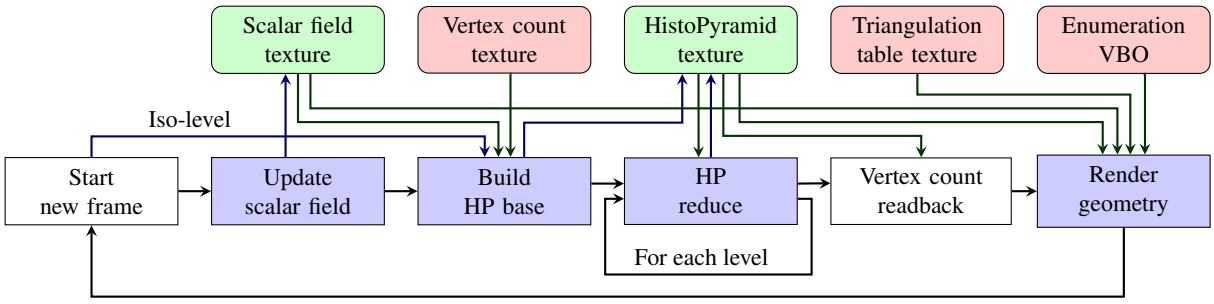

Figure 7: *A flowchart of our algorithm (vertex-shader variant, aka VS variant). Black arrows describe the temporal flow of the algorithm, blue arrows are writes, and green arrows are reads. White boxes are operations done on the CPU, blue boxes are done on the GPU, red boxes are static data, and green boxes are dynamic data.*

the triangulation table texture. This way, this variant reduces the number of HistoPyramid traversals from once per every vertex of each iso-surface triangle, to once per geometry-producing MC cell. Despite this theoretical advantage, it showed in the timings of Section 5 that the overhead of this additional GPU pipeline stage is still considerably larger than the partially redundant HistoPyramid traversals.

Instead of direct rendering, the generated iso-surface mesh might be required by the CPU. In that case, it can be downloaded as a 1D buffer stream. The most direct way for this on recent hardware is the transform feedback extension, which captures the newly generated stream of vertex data before it actually is rendered. On older hardware, a simple fragment shader and render-to-texture (NVidia/ATI) or render-to-vertex buffer (ATI) can be used to extract the geometry to a static data array in graphics memory, from where it can be downloaded to the CPU.

5. Performance analysis

The number of voxels processed per second is the usual measure of performance for iso-surface extraction implementations. But since the processing of a MC cell intersecting the iso-surface is higher than for MC cells that do not intersect, this number alone is not a good enough indicator of performance. Therefore, we have chosen to introduce the term *density*, defined as the percentage of the MC cells that do produce geometry.

We used six datasets with varying complexity at four different resolutions, thus measuring the performance of the algorithms under a wide range of conditions. Iso-surfaces of the datasets are depicted in Figure 8. The "Bunny" and "CT-head" datasets were obtained from the Stanford volume data archive [Sta], the "MRbrain", "Bonsai", and "Aneurism" datasets were obtained from volvis.org [Vol], while the "Cayley" is the implicit equation $f(x,y,z) = 16xyz + 4(x + y + z) - 1$ sampled over $[-1,1]^3$.

We also used three different NVidia GeForce graphics cards: a 6600GT with 128MB RAM, a 7800GT with 256MB RAM, and a 8800GTX with 768MB RAM. The 6600GT was part of a Linux workstation with an AMD Athlon 64 3500+ CPU at 2.2 GHz and 1 GB of RAM, using the 97.55 release of the NVidia display driver. The 7800GT and the 8800GTX were part of another Linux workstation with an Intel Core2 CPU at 2.13 GHz and 1 GB of RAM, using the 97.51 release of the display driver. The 6600GT and 7800GT did not have enough RAM to handle the largest datasets, and thus had to be skipped in large dataset rendering.

Table 5 shows the results of our experiments. For each dataset, at each resolution, we have calculated the number of MC cells and the density. We have measured the performance of three different algorithms, two of our own design and one intended for comparison. Our own ones are HistoPyramid extraction in the vertex shader (HP-VS), and extraction in the geometry shader (HP-GS). For comparison, we benchmarked a geometry shader-based MT implementation, provided in the NVidia OpenGL SDK-10 (NV-SDK10). The performance is measured in million MC cells processed per second, with frames per second given in parentheses.

The results show that the HistoPyramid algorithms achieve a considerably higher throughput for increasing amounts of volume data. This is expected, since the HistoPyramid is especially suited for sparse input data, and applied on large datasets, large amounts of data is culled early in the traversal. However, some increase in throughput is also likely caused by the fact that larger chunks of data give increased possibility of data-parallelism, and require fewer GPU state-changes in relation to the data processed. This probably explains the (moderate) increase in performance for the NV-SDK10 comparison implementation.

HistoPyramid building speeds are highly dependent on memory bandwidth. The 7800GT has twice the memory bandwidth of the 6600GT, and the 7800GT run is thus also about twice as fast as the 6600GT run.

On the whole, The HP-VS algorithm outperforms all other algorithms except for some of the tiny and dense datasets,

	Model	MC cells	Density	6600GT HP-VS	7800GT HP-VS	8800GTX HP-VS	8800GTX HP-GS	8800GTX NV-SDK10
Bunny	255x255x255	16581375	3.2%	–	–	538.6 (32.5)	77.6 (4.7)	–
	127x127x127	2048383	5.6%	5.4 (2.6)	11.8 (5.8)	309.5 (151.1)	41.9 (20.4)	–
	63x63x63	250047	9.1%	4.0 (16.1)	8.5 (34.1)	163.4 (653.5)	26.2 (104.7)	28.3 (113.2)
	31x31x31	29791	13.6%	2.5 (82.8)	5.0 (167.9)	25.5 (857.0)	13.0 (434.9)	21.9 (734.0)
Cthead	255x255x128	8323200	3.7%	–	16.3 (2.0)	437.6 (53.0)	64.0 (7.8)	–
	127x127x63	1016127	6.3%	5.4 (5.3)	11.6 (11.5)	288.1 (283.6)	37.8 (37.2)	–
	63x63x31	123039	9.6%	3.7 (29.9)	7.7 (62.2)	97.3 (791.0)	23.2 (188.6)	25.3 (205.9)
	31x31x15	14415	14.5%	2.3 (161.3)	4.5 (311.5)	12.9 (896.4)	10.5 (729.0)	17.1 (1187.0)
mrbrain	255x255x128	8323200	5.8%	–	10.5 (1.3)	309.0 (37.4)	35.7 (4.3)	–
	127x127x63	1016127	7.4%	4.6 (4.5)	9.9 (9.7)	257.7 (263.6)	29.7 (29.2)	–
	63x63x31	123039	10.0%	3.5 (28.6)	7.4 (60.0)	96.8 (786.5)	20.5 (166.8)	26.4 (214.9)
	31x31x15	14415	14.9%	2.2 (155.0)	4.3 (300.9)	12.7 (879.7)	10.0 (695.0)	18.2 (1257.4)
Bonsai	255x255x255	16581375	3.0%	–	–	560.8 (33.8)	77.0 (4.6)	–
	127x127x127	2048383	5.1%	5.9 (2.9)	13.0 (6.3)	329.8 (161.0)	42.3 (20.7)	–
	63x63x63	250047	6.7%	5.4 (21.5)	11.4 (45.5)	186.5 (745.9)	31.0 (124.1)	28.9 (115.6)
	31x31x31	29791	8.2%	4.1 (136.8)	8.0 (268.8)	25.1 (843.0)	18.3 (613.7)	24.0 (804.6)
Aneurism	255x255x255	16581375	1.6%	–	–	892.5 (53.8)	125.1 (7.5)	–
	127x127x127	2048383	2.1%	12.6 (6.1)	29.1 (14.2)	557.6 (272.2)	92.8 (45.3)	–
	63x63x63	250047	3.7%	9.1 (36.2)	19.2 (76.7)	190.5 (761.9)	49.8 (199.3)	32.9 (131.5)
	31x31x31	29791	6.8%	4.5 (149.7)	8.6 (289.1)	25.0 (839.3)	18.8 (632.6)	25.5 (856.6)
Cayley	255x255x255	16581375	0.9%	–	–	1112.3 (67.1)	233.2 (14.1)	–
	127x127x127	2048383	1.9%	13.5 (6.6)	31.2 (15.2)	581.3 (283.8)	112.5 (54.9)	–
	63x63x63	250047	3.9%	8.5 (33.9)	17.9 (71.6)	198.0 (791.9)	51.6 (206.6)	32.1 (128.5)
	31x31x31	29791	8.1%	3.7 (123.8)	7.3 (245.8)	25.8 (866.2)	17.5 (588.3)	24.7 (827.9)

Table 1: *The performance of full extraction and rendering of iso-surfaces, measured in million MC cells processed per second, with frames per second given in parentheses. The algorithms measured were HistoPyramid extraction in the vertex shader (HP-VS), extraction in the geometry shader (HP-GS), and the Marching Tetrahedra implementation of the NVidia OpenGL SDK10 (NV-SDK10).*

where NV-SDK10 has the best performance. We also see that HP-GS, using the geometry shader for stream expansion, and with the theoretical advantage of reducing the number of HistoPyramid traversals to about one sixth on average, performs four to eight times slower than HP-VS in practice on the same hardware. This shows that the introduction of an additional stage in the graphics pipeline is considerably more expensive than the extra fetches from the HistoPyramid texture. However, this ratio is likely to change in future hardware generations with improved geometry shaders.

The HP-VS algorithm on the 8800GTX is again ten to thirty times faster than the 7800GT, peaking at over 1000 million MC cells processed per second. But on the whole, even though the performance of the 6600GT is miniscule compared to the 8800GTX, we see that in contrast to DX10-based approaches, the algorithm still works, even on low-end hardware. In addition, NV-SDK10 on an 8800GTX is only two to four times faster than HP-VS on a 7800GT.

We also experimented with various detail changes in the algorithm. For example, positioning the vertices at the edge midpoints removes the need for sampling the scalar field in the extraction pass, as mentioned in Section 4. In theory, this should increase performance, but experiments show that the speedup is marginal and visual quality drops drastically, see Figure 6. In addition, we benchmarked performance with different texture storage formats, including the new integer storage format of SM4. However, it showed that the storage type still has relatively little impact in this hardware generation.

6. Conclusion and future work

We have presented a fast and general method to extract iso-surfaces from volume data, running completely on the GPU. It is based on the MC algorithm and uses the HistoPyramid technique to handle geometry generation. We have described a SM3 version using HistoPyramids for both culling of MC cells and geometry generation, and a SM4 version that uses the geometry shader for geometry generation. We have done a performance analysis on both versions alongside the MT implementation provided in the NVidia SDK-10. For reasonable data sizes, our algorithms outperforms all other known GPU-based iso-surface extraction algorithms. Surprisingly, the SM3 variant of the algorithm is also the

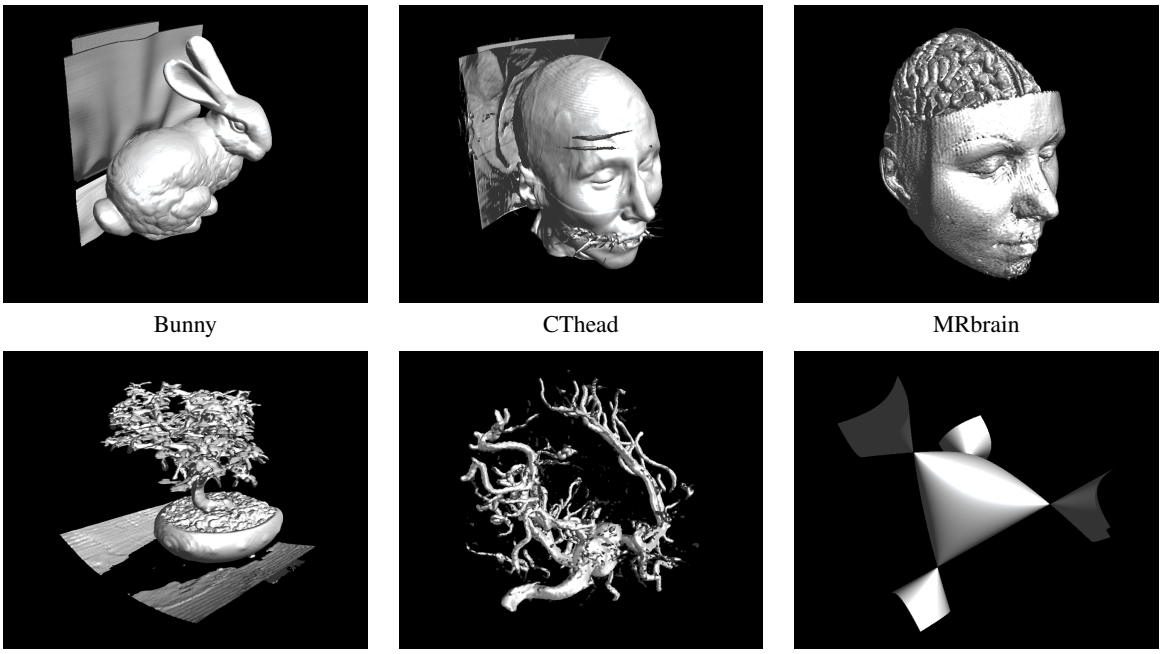

Figure 8: *Images of the voxel volumes used in the performance analysis.*

fastest one on recent SM4 hardware, even though it actually ignores recent geometry shader capabilities.

In direct comparison, Scan and HistoPyramids have some similarities (the up-sweep phase and the HistoPyramid construction is closely related), while the difference lies in the extraction process. Scan has the advantage that only one table lookup is needed, as long as scatter-write is available. For HistoPyramids, each output element extraction requires a $\log(n)$-traversal of the HistoPyramid. But despite that algorithmic complexity, the HistoPyramid algorithm can utilize the texture cache very efficiently, reducing the performance hit of the deeper traversal. A second difference is that Scan's output extraction iterates over *all input elements* and scatters the relevant ones to output, while HistoPyramid iterates on the *output elements* instead. Thus, if a lot of the input elements are to be culled (which is the case with MC), the HistoPyramid algorithms can play out its strengths, despite its tree traversal.

It is worth noting that the presented geometry generation approach is *not* specific to MC. Its data expansion principles are general enough to be used in totally different areas, allowing the use of older SM3 hardware for GPU-based geometry generation and processing.

Future work might concentrate on out-of-core applications which can benefit greatly from high-speed MC implementations. Multiple Rendering Targets (MRT) might allow us to generate multiple iso-surfaces or to accelerate HistoPyramid processing (and thus geometry generation) even further. A proper view-dependent layering of volume data extractions could allow for immediate output of transparency sorted iso-surface geometry.

References

[BCL06] BUATOIS L., CAUMON G., LEVY B.: GPU accelerated isosurface extraction on tetrahedral grids. In *International Symposium on Visual Computing* (2006).

[DRS07] DYKEN C., REIMERS M., SELAND J.: Real-time GPU silhouette refinement using adaptively blended Bézier patches. *Computer Graphics Forum*, to appear (2007).

[Har07] HARRIS M.: Parallel prefix sum (scan) with CUDA. NVIDIA CUDA SDK 1.0, 2007.

[HISL03] HARRIS M. J., III W. V. B., SCHEUERMANN T., LASTRA A.: Simulation of cloud dynamics on graphics hardware. *Proceedings of Graphics Hardware* (2003).

[Hor05] HORN D.: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005, ch. Stream Reduction Operations for GPGPU Applications, pp. 573–589.

[JC06] JOHANSSON G., CARR H.: Accelerating marching cubes with graphics hardware. In *CASCON '06: Proceedings of the 2006 conference of the Center for Ad-*

vanced Studies on Collaborative research (2006), ACM Press.

[JS05] JULIANO J., SANDMEL J.: GL_EXT_framebuffer_object. OpenGL extension registry, 2005.

[KSE04] KLEIN T., STEGMAIER S., ERTL T.: Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. *Pacific Graphics 2004 Proceedings* (2004).

[KW05] KIPFER P., WESTERMANN R.: GPU construction and transparent rendering of iso-surface. In *Proceedings Vision, Modeling and Visualization 2005* (2005), Greiner G., Hornegger J., Niemann H., Stamminger M., (Eds.), IOS Press, infix, pp. 241–248.

[LB07] LICHTENBELT B., BROWN P.: GL_EXT_gpu_shader4. OpenGL extension registry, 2007.

[LC87] LORENSEN W., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (SIGGRAPH 87 Proceedings) 21*, 4 (1987), 163–170.

[MSS94] MONTANI C., SCATENI R., SCOPIGNO R.: A modified look-up table for implicit disambiguation of Marching Cubes. *The Visual Computer 10* (1994), 353–355.

[Pas04] PASCUCCI V.: Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. *Joint Eurographics - IEEE TVCG Symposium on Visualization* (2004), 292–300.

[Sta] The stanford volume data archive. http://graphics.stanford.edu/data/voldata/.

[Ura06] URALSKY Y.: DX10: Practical metaballs and implicit surfaces. GameDevelopers conference, 2006.

[Vol] Volvis volume dataset archive. http://www.volvis.org/.

[ZTTS06] ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.-P.: *GPU Point List Generation through Histogram Pyramids*. Tech. Rep. MPI-I-2006-4-002, Max-Planck-Institut für Informatik, 2006.

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2006-5-001	M. Bender, S. Michel, G. Weikum, P. Triantafilou	Overlap-Aware Global df Estimation in Distributed Information Retrieval Systems
MPI-I-2005-5-002	S. Siersdorfer, G. Weikum	Automated Retraining Methods for Document Classification and their Parameter Tuning
MPI-I-2005-4-006	C. Fuchs, M. Goesele, T. Chen, H. Seidel	An Emperical Model for Heterogeneous Translucent Objects
MPI-I-2005-4-005	G. Krawczyk, M. Goesele, H. Seidel	Photometric Calibration of High Dynamic Range Cameras
MPI-I-2005-4-004	C. Theobalt, N. Ahmed, E. De Aguiar, G. Ziegler, H. Lensch, M.A.,. Magnor, H. Seidel	Joint Motion and Reflectance Capture for Creating Relightable 3D Videos
MPI-I-2005-4-003	T. Langer, A.G. Belyaev, H. Seidel	Analysis and Design of Discrete Normals and Curvatures
MPI-I-2005-4-002	O. Schall, A. Belyaev, H. Seidel	Sparse Meshing of Uncertain and Noisy Surface Scattered Data
MPI-I-2005-4-001	M. Fuchs, V. Blanz, H. Lensch, H. Seidel	Reflectance from Images: A Model-Based Approach for Human Faces
MPI-I-2005-2-004	Y. Kazakov	A Framework of Refutational Theorem Proving for Saturation-Based Decision Procedures
MPI-I-2005-2-003	H.d. Nivelle	Using Resolution as a Decision Procedure
MPI-I-2005-2-002	P. Maier, W. Charatonik, L. Georgieva	Bounded Model Checking of Pointer Programs
MPI-I-2005-2-001	J. Hoffmann, C. Gomes, B. Selman	Bottleneck Behavior in CNF Formulas
MPI-I-2005-1-008	C. Gotsman, K. Kaligosi, K. Mehlhorn, D. Michail, E. Pyrga	Cycle Bases of Graphs and Sampled Manifolds
MPI-I-2005-1-008	D. Michail	?
MPI-I-2005-1-007	I. Katriel, M. Kutz	A Faster Algorithm for Computing a Longest Common Increasing Subsequence
MPI-I-2005-1-003	S. Baswana, K. Telikepalli	Improved Algorithms for All-Pairs Approximate Shortest Paths in Weighted Graphs
MPI-I-2005-1-002	I. Katriel, M. Kutz, M. Skutella	Reachability Substitutes for Planar Digraphs
MPI-I-2005-1-001	D. Michail	Rank-Maximal through Maximum Weight Matchings
MPI-I-2004-NWG3-001	M. Magnor	Axisymmetric Reconstruction and 3D Visualization of Bipolar Planetary Nebulae
MPI-I-2004-NWG1-001	B. Blanchet	Automatic Proof of Strong Secrecy for Security Protocols
MPI-I-2004-5-001	S. Siersdorfer, S. Sizov, G. Weikum	Goal-oriented Methods and Meta Methods for Document Classification and their Parameter Tuning
MPI-I-2004-4-006	K. Dmitriev, V. Havran, H. Seidel	Faster Ray Tracing with SIMD Shaft Culling
MPI-I-2004-4-005	I.P. Ivrissimtzis, W.-. Jeong, S. Lee, Y.a. Lee, H.-. Seidel	Neural Meshes: Surface Reconstruction with a Learning Algorithm
MPI-I-2004-4-004	R. Zayer, C. Rssl, H. Seidel	r-Adaptive Parameterization of Surfaces

MPI-I-2004-4-003	Y. Ohtake, A. Belyaev, H. Seidel	3D Scattered Data Interpolation and Approximation with Multilevel Compactly Supported RBFs
MPI-I-2004-4-002	Y. Ohtake, A. Belyaev, H. Seidel	Quadric-Based Mesh Reconstruction from Scattered Data
MPI-I-2004-4-001	J. Haber, C. Schmitt, M. Koster, H. Seidel	Modeling Hair using a Wisp Hair Model
MPI-I-2004-2-007	S. Wagner	Summaries for While Programs with Recursion
MPI-I-2004-2-002	P. Maier	Intuitionistic LTL and a New Characterization of Safety and Liveness
MPI-I-2004-2-001	H. de Nivelle, Y. Kazakov	Resolution Decision Procedures for the Guarded Fragment with Transitive Guards
MPI-I-2004-1-006	L.S. Chandran, N. Sivadasan	On the Hadwiger's Conjecture for Graph Products
MPI-I-2004-1-005	S. Schmitt, L. Fousse	A comparison of polynomial evaluation schemes
MPI-I-2004-1-004	N. Sivadasan, P. Sanders, M. Skutella	Online Scheduling with Bounded Migration
MPI-I-2004-1-003	I. Katriel	On Algorithms for Online Topological Ordering and Sorting
MPI-I-2004-1-002	P. Sanders, S. Pettie	A Simpler Linear Time 2/3 - epsilon Approximation for Maximum Weight Matching
MPI-I-2004-1-001	N. Beldiceanu, I. Katriel, S. Thiel	Filtering algorithms for the Same and UsedBy constraints
MPI-I-2003-NWG2-002	F. Eisenbrand	Fast integer programming in fixed dimension
MPI-I-2003-NWG2-001	L.S. Chandran, C.R. Subramanian	Girth and Treewidth
MPI-I-2003-4-009	N. Zakaria	FaceSketch: An Interface for Sketching and Coloring Cartoon Faces
MPI-I-2003-4-008	C. Roessl, I. Ivrissimtzis, H. Seidel	Tree-based triangle mesh connectivity encoding
MPI-I-2003-4-007	I. Ivrissimtzis, W. Jeong, H. Seidel	Neural Meshes: Statistical Learning Methods in Surface Reconstruction
MPI-I-2003-4-006	C. Roessl, F. Zeilfelder, G. Nrnberger, H. Seidel	Visualization of Volume Data with Quadratic Super Splines
MPI-I-2003-4-005	T. Hangelbroek, G. Nrnberger, C. Roessl, H.S. Seidel, F. Zeilfelder	The Dimension of C^1 Splines of Arbitrary Degree on a Tetrahedral Partition
MPI-I-2003-4-004	P. Bekaert, P. Slusallek, R. Cools, V. Havran, H. Seidel	A custom designed density estimation method for light transport
MPI-I-2003-4-003	R. Zayer, C. Roessl, H. Seidel	Convex Boundary Angle Based Flattening
MPI-I-2003-4-002	C. Theobalt, M. Li, M. Magnor, H. Seidel	A Flexible and Versatile Studio for Synchronized Multi-view Video Recording
MPI-I-2003-4-001	M. Tarini, H.P.A. Lensch, M. Goesele, H. Seidel	3D Acquisition of Mirroring Objects
MPI-I-2003-2-004	A. Podelski, A. Rybalchenko	Software Model Checking of Liveness Properties via Transition Invariants
MPI-I-2003-2-003	Y. Kazakov, H. de Nivelle	Subsumption of concepts in DL $\mathcal{FL}_0$ for (cyclic) terminologies with respect to descriptive semantics is PSPACE-complete
MPI-I-2003-2-002	M. Jaeger	A Representation Theorem and Applications to Measure Selection and Noninformative Priors
MPI-I-2003-2-001	P. Maier	Compositional Circular Assume-Guarantee Rules Cannot Be Sound And Complete
MPI-I-2003-1-018	G. Schaefer	A Note on the Smoothed Complexity of the Single-Source Shortest Path Problem
MPI-I-2003-1-017	G. Schfer, S. Leonardi	Cross-Monotonic Cost Sharing Methods for Connected Facility Location Games
MPI-I-2003-1-016	G. Schfer, N. Sivadasan	Topology Matters: Smoothed Competitive Analysis of Metrical Task Systems
MPI-I-2003-1-015	A. Kovcs	Sum-Multicoloring on Paths