Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs, T. Kuhlen, and F. Marton (Editors)

Polygonization of implicit surfaces on Multi-Core Architectures with SIMD instructions

P. Shirazian[1], B. Wyvill[1] and J-L. Duprat[2]

[1]University of Victoria, BC, Canada
[2]Intel Corporation.

Abstract

In this research we tackle the problem of rendering complex models which are created using implicit primitives, blending operators, affine transformations and constructive solid geometry in a design environment that organizes all these in a scene graph data structure called BlobTree. We propose a fast, scalable, parallel polygonization algorithm for BlobTrees that takes advantage of multicore processors and SIMD optimization techniques available on modern architectures. Efficiency is achieved through the usage of spatial data structures and SIMD optimizations for BlobTree traversals and the computation of mesh vertices and other attributes. Our solution delivers interactive visualization for modeling systems based on BlobTree scene graph.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

We present a parallel method for speeding up the generation of a polygon mesh from an implicit model. Although the method is applicable to many types of implicit surfaces, we focus on surfaces generated from fields surrounding geometric primitives, known as skeletal implicit surfaces, [BCB*97]. The model data structure is a tree whose leaf nodes are primitives, and internal nodes are operators; the *BlobTree*, [WGG99]. Currently the *BlobTree* supports operations such as; arbitrary blends, boolean operations, warping at a local and global level including contact deformations. Geometric transformation matrices are also stored as nodes in the tree so the data structure is also a scene graph.

A *BlobTree* is typically visualized by polygonization to produce a triangle mesh to be rasterized by the graphics processor. Direct ray tracing [BCB*97] can also be used, to produce high quality images. Both methods require computation of the field value which can only be evaluated by traversing the *BlobTree* structure. The field due to each operator depends on its child nodes and the leaves are the primitives which can be any implicitly defined function; e.g. distance field due to geometric skeletal elements.

Implicit modeling using the *BlobTree* has several advantages over other modeling methods. Various different blends are simple to represent, as are free-form volume deformations and constructive solid geometry operations (CSG) [GVJ*09]. Other operators such as detecting contact, and warping surfaces accordingly (see [CGD97]), can easily be represented as nodes in the *BlobTree*. The models created with *BlobTree* are resolution independent and the definition can be very compact, making the *BlobTree* a good candidate to use for network based cooperative design.

An incremental, sketch based *BlobTree* system was built by Schmidt et al. [SWSJ06], promoting flexibility and modular design for the creation of complex models, and most of the earlier problems with the methodology have been overcome [BBCW10]. Although direct manipulation is possible [SWSJ06], very complex models can only be visualized interactively as coarse meshes. Hence the need for a faster polygonizer. The *BlobTree* facilitates incremental modeling, a strategy that promotes flexibility and modular design for creating complex models.

The main contribution of our research is a high performance polygonization algorithm that scales well with the number of physical cores and SIMD vector width available on modern processors.

As opposed to previous work that attempted to render implicit surfaces defined by static algebraic surfaces or volu-

DOI: 10.2312/EGPGV/EGPGV12/089-098

metric scanned data, our method is data-driven where the definition of the surface can change over time. This feature is particularly useful in collision detection applications such as surgical simulations where the interaction of the surgical tools and deformable tissues should be visualized in real-time [LD07].

In addition we have improved the performance of the algorithm that finds the intersection of a cube edge and the surface, by making use of the SIMD architecture, to find the intersection in a single run of a field evaluation kernel.

The paper is organized as follows; related work is reviewed in section 2, background information on implicit surfaces, skeletal primitives and the polygonization process is given in section 3. In sections 6 our algorithm is explained along with the improvements made to the distance-field computation process. Our performance results and future work are presented in sections 7 and 8, respectively.

2. Related Work

Several methods for polygonization of implicit surfaces have been proposed which can be classified based on speed, accuracy of the output mesh or quality. Comparing these methods in terms of performance reveals that space partitioning methods are the fastest and the most popular. The paper [WMW86] was the first to introduce a method for finding iso-surfaces using uniform space subdivision into cubic cells. A seed cell on the surface was found by starting at a vertex close to each primitive and evaluating the field at cell vertices along each of the three axes to find a surface crossing. Vertices inside the volume were classified as ‘hot’ and ‘cold’ outside. A hash table was used to keep track of processed cells to avoid redundant field evaluations and to avoid storing any cells that did not contain part of the surface. Only adjacent cells that share an intersecting edge with their parent were processed, and a second cubic subdivision served to reduce the number of primitives considered in each field evaluation. Ambiguous cases were ameliorated by taking another sample from the centre of the face. A similar method was later introduced as Marching Cubes in [LC87]. The main difference between the two algorithms was that Lorenson et al. applied their method to discrete volume data instead of sampling a continuous function and in Lorenson’s method the space was completely partitioned into cubic voxels and all cubes were visited.

Bloomenthal showed that the ambiguous cases can be dealt with by subdividing cells into tetrahedra [Blo94], and also that a six tetrahedron subdivision was superior to subdividing into five [GH95]. The fact that tetrahedral simplices have 4 vertices reduces the total number of configurations to 16 (or 3 by symmetry), however, the number of redundantly generated triangles as a result of this decomposition increases significantly. We will refer to marching cubes and tetrahedra, with MC and MT respectively throughout this paper.

There have been many enhancements proposed for both MC and MT. Some gain advantage by classifying cubes according to different criteria and surface edge intersection calculation and number of field function evaluations. For example, Dietrich et al. [DSC*09], did a statistical analysis of cube configurations in MC that are responsible for most of the degenerate triangles in the output mesh. Their algorithm avoids those cube configurations by inserting an extra vertex into the cell when generating triangles as was done in [WMW86] where an extra sample was taken. This reduces the statistical occurrence of the problem.

Triquet et al. [TGMC03] enhanced MT by applying time-stamps on calculated values and using hash tables for retrieving them. They also pre-computed surface vertices along crossing edges which are shared with adjacent voxels and referenced previously calculated values to avoid re-evaluating them. This latter enhancement was also done in Bloomenthal’s polygonizer [Blo94] and was a fairly common feature of implicit surface polygonizer’s of the 1990s.

Beside enhancing serial algorithms some attempts were made to increase the performance of MC by dividing the workload between multiple CPUs or on a network grid of computers. Mackerras proposed an MIMD implementation of MC algorithm [Mac92]. The bounding volume is divided into uniform blocks and each processor runs a serial implementation of MC on one or more blocks. They reported that because of efficient usage of cache their method showed a speed-up greater than the total number of physical processors involved. Hansen and Hinker presented a parallel implementation of MC [HH92]. They labeled each cube with a virtual processor identifier to avoid complexities in communicating between processors, then each cube is processed independently. They reported linear speed-up by increasing the number of physical processors. Their method spends constant time on each processor regardless of the number of polygons in a cubic cell.

The advent of shader programs and GPGPU computing interested some to port serially computationally intensive programs to the GPU. Space partitioning methods like MC and MT are good candidates for these devices since each cell (either tetrahedra or cube) can represent an independent volume to be processed on a separate SIMD core.

Kipfer and Westermann proposed a GPU-accelerated Marching Tetrahedra algorithm that stores the topology of the surface on the GPU [KW05]. They used a span-space interval tree to cull tetrahedral elements that don’t intersect with the surface. Caching edge-surface intersections helped them to avoid redundant calculations. For computing edge-surface intersections they used linear interpolation for finding roots along each edge which is less accurate and degrades the quality of the output mesh.

Johansson et al. accelerated iso-surface extraction using graphics hardware [JC06]. They stored MC cases on the

GPU and used a vertex program to compute surface intersection points. They used a span-space data-structure similar to [CMM*97] to enhance the cell classification phase in MC. Their method shows a speedup order of 13 over the naive algorithm.

Tatarchuk et al. presented an iso-surface extraction algorithm implemented using DirectX [TSD08]. They used graphics hardware to visualize medical data. They maximized utilization of SIMD units on the GPU by separating their polygonization (which is a hybrid of marching cubes and marching tetrahedra) into two phases: Fast cube tetrahedralization and a marching tetrahedra pass. Each input voxel position is dynamically computed in the vertex shader, then they used the geometry shader and stream-out features of DirectX 10 to tessellate voxels into six tetrahedra spanning the voxel cube. However their method is limited to medical volume datasets.

An adaptive parallel polygonization method is proposed by Yang et al. [YCP10]. They enhanced the Araujo's method [RA05] by dividing the bounding box of the model into eight parts and then processing them in parallel. For each part, their method find a seed point on the surface and increasingly expand it to form a local mesh for the part by using the surface tracking approach. Using local curvature of the implicit surface, their method produces triangles of varying sizes. However, their method is not scalable since it can not guarantee finding a seed point per each sub box in case the number of sub boxes increases. They reported very slow rendering times even for the simple models that they have tested their system with.

In a similar work Knoll et al. [KHH*07] proposed interactive raytracing of arbitrary implicits with SIMD interval arithmatic. They used SSE instructions for fast computation of interval arithmatic and ray traversal algorithm. However, their method is restricted to static implicit functions and algebraic surfaces.

None of the proposed methods above used a modeling framework to define their input data in a hierarchical structure similar to the *BlobTree*. Their method is either limited to volume data or an algebraic implicit function to represent the underlying volume. In a closely related work, Schmidt et al. [SWG05] used a field caching mechanism inside the *BlobTree* to perform fast potential field reconstruction without traversing the entire tree. They used a trilinear reconstruction filter for field value and a triquaratic filter for gradient interpolation. They evaluated cache efficiency by polygonizing a *BlobTree* model once using cache nodes and the other time without. They reported upto 16 times speedup for polygonizing a model with different resolutions. However, their method is not scalable since the cache nodes cannot be updated from different processing threads without using locking mechanisms or a data race condition can occur.

3. Background

Implicit surfaces can be defined based on discrete data, radial basis functions, offset surfaces, algebraic surfaces, level sets or distance fields to skeletal geometric primitives [BCB*97]. Independently of its origin, an implicit surface can be defined as a level-set function $F : \mathbb{R}^3 \rightarrow \mathbb{R}$ where the surface can be defined for instance as the set of points $\left\{M(x,y,z) \in \mathbb{R}^3 | F(x,y,x) = c\right\}$. c is a constant and is called the *iso-value* which is set to 0.5 in our system. For each point in space if the field is greater than c the point is considered inside the model otherwise outside.

Skeletal implicit models are constructed from combinations of geometric skeletal elements. An implicit model A is generated by summing the influences of N_A skeletal elements: $F_A(x,y,z) = \sum_{i=1}^{i=N_A} F_i(x,y,z)$ The field value due to an skeletal element at a point in 3D space is computed as filtered distance to its skeleton where the filter function (i.e. falloff function) is defined as follows [WGG99]:

$$g_{\text{wyvill}}(x) = \begin{cases} 1 & \text{if } x \leq 0 \\ (1-x^2)^3 & \text{if } 0 < x < 1 \\ 0 & \text{if } x \geq 1 \end{cases} \tag{1}$$

Normals can be derived from gradients which are computed by evaluating 4 field values and performing a numerical approximation:

$$\nabla F(x,y,z) = \begin{cases} F(x+\delta,y,z) - f \\ F(x,y+\delta,z) - f \\ F(x,y,z+\delta) - f \end{cases} \tag{2}$$

Where $f = F(x,y,z)$ is the field at point (x,y,z).

Each skeletal primitive has a bounded region of influence in space. For each node in the tree an axis-aligned bounding box is computed which is used to trivially reject those field queries that are outside the box. The bounding box of the entire model is computed as the union of all primitive nodes bounding boxes.

For evaluating the field at a point P in a *BlobTree* model such as the one shown in figure (1), the tree structure should be traversed from root to leaves recursively. Each operator combines the values of its children according to its type. For example, for a simple blend the values are summed. A leaf node represents a primitive, and returns the value by applying equation 1 to the distance of P from the primitive.

For visualization purposes the *BlobTree* is queried numerous times to evaluate the field. As suggested in [SWG05] accelerating field computation will have a large impact on the overall surface extraction process.

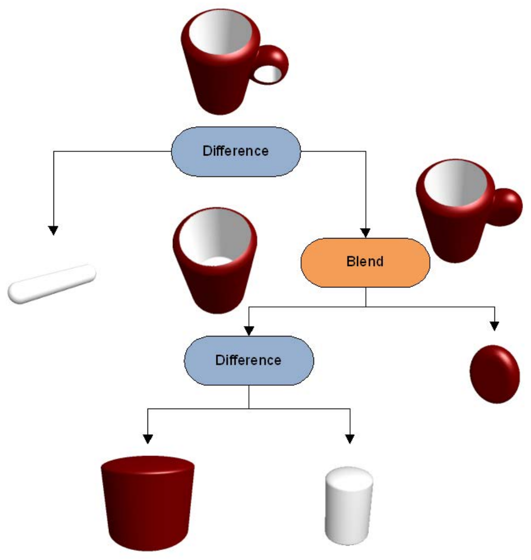

Figure 1: *BlobTree structure of a coffee mug created with CSG and skeletal implicit primitives.*

4. Architecture Constraints

In this section we define some processor architecture constraints, i.e. minimum requirements from the hardware side to implement our algorithm as efficiently as possible. The algorithm scales with the number of physical cores and the SIMD vector width available on the processor. See results section 7.

Our current implementation leverages both Intel SSE with 4 float wide and Intel AVX with 8 float wide SIMD instruction sets. Using a cache-aware technique our algorithm is designed to minimize the movement of cache lines in and out of the processor's on-chip memory. To this end the technique requires at least 256 kilobytes of last level cache memory per each processor core. The input data structures take about 192 kilobytes of memory in our implementation.

Although our test environment was Intel based, our algorithm should be implementable on any multicore machine with SIMD instructions and sufficient cache.

5. Naming Conventions

The polygonization method used, is a space partioning algorithm based on [WMW86], which uses a uniform grid of a user defined cell size (*cellsize*). In order to leverage the SIMD parallel computation capabilities of the processor, the bounding box of the model is divided into axis-aligned grids of 8x8x8 vertices where each grid is called model partitioning unit (*MPU*).

An *MPU* is $7 * cellsize$ as shown in figure 2. Each *MPU* contains 7*7*7 or 343 cubic cells. An *MPU* is called *empty* if it does not intersect with the iso-surface of the model. The list of all *MPU*s is called the *MPUSET* and a half open interval $[a,b)$ over *MPUSET* is called an *MPURANGE* which contains consecutive *MPU*s from a to $b-1$.

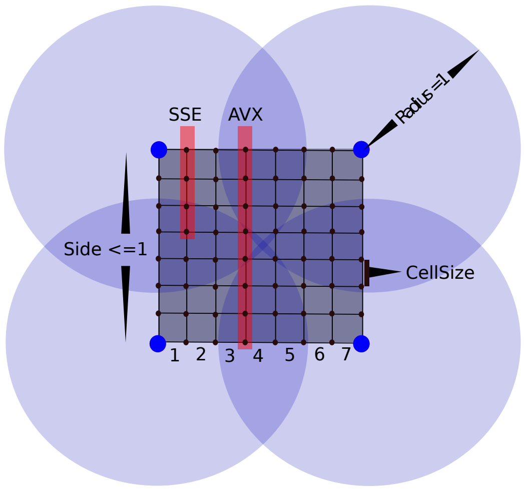

Figure 2: *The MPU is our unit of computation per each core illustrated as a 2D cross section here. Field-values due to every 4 or 8 points are computed in parallel with SSE or AVX instructions, respectively. When the field at a vertex is zero no iso-surface will pass in the neighbourhood of a unit circle (sphere in 3D) centred at that vertex.*

6. Algorithm

The input to our algorithm is a *BlobTree* data structure, representing an implicit model whose iso-surface we wish to find. Output is a triangle mesh. The model bounding box and the cellsize parameter supplied by the user to control the resolution of the final mesh. The *BlobTree* structure is first converted into a compact, linear structure required for SIMD optimization techniques, then the model bounding box is divided into the *MPUSET* with respect to the *cellsize* parameter. The *MPUSET* is processed in parallel using multiple cores; with a fast empty *MPU* rejection method and SIMD surface extraction algorithm the mesh contained within intersecting *MPU*s is extracted. The algorithm has no synchronization points except after all *MPU*s are processed and the triangle mesh is sent to the GPU for rasterization. The following sections describe this whole process in detail.

We start by describing the initialization phase and continue with the surface extraction details in the next section. The algorithm starts by computing the size of an *MPU* side (7 cells) and dividing the bounding box of the model into a 3D grid of *MPU*s, where each *MPU* is assigned a unique global identifier. The main idea of our algorithm is parallel processing of the set of all *MPU*s (*MPUSET*) using multi-core and SIMD processing techniques.

Our algorithm recursively splits *MPUSET* into disjoint *MPURANGE*s where each *MPURANGE* is assigned to an idle core on the processor. The granularity of the divisions can be determined by the average amount of machine cycles spent to process an *MPU*, however, in our implementation we resort to the solution provided by Intel Threading Building Blocks (TBB) [Rei07], which provides a non-preemptive task scheduling system to take care of the differences in task loads by monitoring processors and starting new tasks on idle cores automatically (work-stealing) [Rei07].

6.1. BlobTree Linearization

The first step in our algorithm is the *BlobTree* reduction and pruning as suggested by Fox et al. [FGW01]. In the second step, using the same linearization algorithm proposed for quadtrees [LS00]; the *BlobTree* is converted into a pointerless representation to achieve cache-memory efficiency by keeping all input data structures at aligned memory addresses and fitting the entire *BlobTree* model into the last level cache memory of the processor. The final linearized *BlobTree* is in the format cache-line aligned structure of arrays. With this format several computations can be optimized with SIMD instructions, e.g. applying a transformation matrix on a vector of 4 or 8 vertices as opposed to scalar computation. The output mesh is also in the format of cache-line aligned structure of arrays which is the key to compute colors and normals in SIMD fashion.

6.2. Surface Extraction

In our algorithm we assign field values for every vertex of every *MPU* that is not trivially rejected with the method explained in the following, and compute the triangular mesh representing the iso-surface. This approach combines elements of several algorithms ([WMW86, LC87, Blo94]).

We extended the method proposed by Zhang et al. [ZWW06] to trivially reject all empty *MPU*s. The observation made is that according to equation 1, if the field value at a given vertex is zero then the shortest distance from that vertex to the iso-surface is greater than or equal to one (See figure 2). Using this fact empty *MPU*s can be identified very fast by evaluating the fields at the 8 vertices of each *MPU* and rejecting it of all 8 fields are zero. However, this test is only applicable when the cellsize parameter is smaller than or equal to $1/7$ or 0.1428. For larger *cellsize*s the iso-surface may still intersect with the *MPU* while the fields at vertices of the *MPU* are zero. For a discussion on *cellsize* versus performance see section 7.

For larger *cellsize*s we shoot 8 rays from the centre of the *MPU* to its eight vertices, using the technique of Zhang et al. per each step we march $0.866c$ (0.866 is half of the diagonal of an *MPU* with side one and c being the cellsize parameter) along each ray. At each step we compute the fields for the 8 vertices along the rays; if a non-zero field is found then the *MPU* is further processed, otherwise we march along the rays until we reach the vertices of the *MPU*.

If an *MPU* is not rejected then it is further processed for surface extraction. A local copy of the linearized *BlobTree* is provided per each core in order to avoid *false-sharing* among cores [BS93]. Using SIMD processing techniques field values for all 512 vertices of *MPU* are computed. With SSE or AVX instructions this step requires 128 or 64 field evaluation kernel runs, respectively (figure 2).

All the fields are stored in a memory aligned array of 512 floating points. This technique avoids reevaluating field values while processing cells in the next step. Storing field values from a SIMD register into memory aligned address can be accomplished with a SIMD instruction in parallel. After this step all 343 cells of the *MPU* are processed. Per each cell, the 8 vertex field values are gathered in SIMD fashion. Each vertex with a field greater than or equal to *iso-value* is labeled one otherwise zero. The configuration index of the cell is computed using the SIMD method shown in algorithm 1. A configuration index is computed to access the table as in [LC87]. We used the modified marching cubes table proposed by Dietrich et al. that eliminates many of the degenerate triangles produced in the original MC algorithm [DSC*09]. For the ambiguous cases we take another sample from the center of the cell [WMW86, DSC*09].

Algorithm 1 SIMD computation of cell configuration. Pseudo code provided for AVX SIMD computation. Similar code can be written in SSE.

```
1: Gather the 8 vertex field values of the cell
2: simd $index = cmp\_ge8(fields, simd(0.5))$
3: $index = and8(index, simd(1.0))$
4: $index = mul8(index, maskPower)$
5: $index = hadd8(index, index)$
```

In algorithm 1 fields is an array of 8 vertex field values, line 2 performs a parallel comparison between *iso-value* and fields. In line 4 maskpower shifts the field values into the appropriate slot in the SIMD array and finally line 5 performs a horizontal add operation on the values to compute the configuration index.

For each intersecting edge there is one inside and one outside vertex. Using a root finding method the point of intersection of the iso-surface is computed and stored in a hash table to be reused by the neighbouring cells that share that vertex.

For the root finding methods that do not require gradient information such as regula falsi or bisection method, the field value should be evaluated multiple times along the edge, which will degrade the performance of the system. Other methods such as Newthon-Raphson require gradient information, and as mentioned in section 3 each gradient computation involves 4 extra field evaluations. We describe a root finding technique based on SIMD instructions that computes the root with only one extra field evaluation in AVX (two with SSE) with adequate precision. By subdividing the intersecting edge into 8 vertices and evaluating the field values, the exact interval containing the final root can be identified. Performing linear interpolation in that interval will produce the final root (figure 3), it is trivial to show when the number of intervals increases the interpolation error decreases [Mat87].

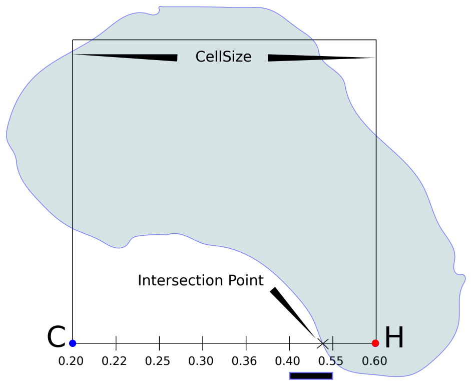

Figure 3: *Top: A cell edge is intersected with part of the surface shown in blue. By performing one field evaluation using AVX or two with SSE instructions the interval containing the intersection point can be identified. The final root is computed using linear interpolation within the interval marked with bold line segment.*

Algorithm 2 summarizes the process of surface extraction which is run per each *MPU*. Lines 1 through 25 are related to the *MPU* discard method explained earlier in this section. Lines 26 through 42 shows the cell processing technique which is optimized using SIMD cell configuration computation and our root finding method. Since color and normal attributes should only be computed for final mesh vertices, this step is performed last to fully leverage SIMD optimizations by performing every 4 or 8 attribute computations in one SIMD call which greatly enhances the throughput of the system and minimizes *BlobTree* traversals.

7. Results

We have implemented our algorithm using Intel threading building blocks in C++ on a Linux platform. We used two systems with different configurations. On the first system which has Intel i7-3960X processor with Sandy Bridge architecture, there are 6 physical cores given that each core runs in hyperthreaded mode; up to 12 threads can run in parallel on this machine. This processor supports both SSE and

Algorithm 2 Algorithm for surface extraction of an *MPU* using AVX SIMD instructions, Similar code can be written for SSE instruction set. Input is linearized BlobTree T, lower vertex of *MPU* and the *cellsize* parameter. Output is the local mesh contained in the *MPU*

```
1:  $side \leftarrow cellsize * 7$
2:  simd $v \leftarrow$Compute $MPU$ vertices
3:  if $side \leq 1$ then
4:      simd $f \leftarrow T.compute\_field8(v)$
5:      if $f == 0$ then
6:          return;
7:      end if
8:  else
9:      $flag \leftarrow true$
10:     $incr = 0.866 * cellsize$
11:     $d = incr$
12:     while d < side * 0.866 do
13:         Shoot rays from center of $MPU$ to its 8 vertices
14:         simd $v \leftarrow$Travel along the rays for distance $d$
15:         simd $f \leftarrow T.compute\_field8(v)$
16:         if $f != 0$ then
17:             $flag \leftarrow false$
18:             $break$;
19:         end if
20:         d = d + incr;
21:     end while
22:     if $flag == true$ then
23:         return;
24:     end if
25: end if
26: float fieldCache[512];
27: for all simd $vertex$ in $mpu$ vertices do
28:     simd $f \leftarrow T.compute\_field8(vertex)$
29:     Store $f$ in appropriate location in $fieldCache$
30: end for
31: for all $cell$ in $mpu$ do
32:     $f \leftarrow gather8(cell, fieldCache)$
33:     $edges \leftarrow$Compute cell config from $f$ to access table
34:     for $i = 1 \rightarrow$count of edges do
35:         if root for $i$th edge is not stored in edge table then
36:             Compute root associated with $i$th edge
37:             Store the root in the edge table
38:             Add root to mesh vertices
39:         end if
40:     end for
41:     Add $cell$ triangles to mesh
42: end for
43: for all simd $vertex$ in mesh vertices do
44:     simd $n \leftarrow T.compute\_normal8(vertex)$
45:     simd $c \leftarrow T.compute\_color8(vertex)$
46:     Add $n$ to mesh normals
47:     Add $c$ to mesh colors
48: end for
```

AVX instructions and there is a last level cache memory of 15 megabytes which is shared between all cores.

The second system is a server with 4 Intel X7560 processor with Nehalem architecture. Each processor has 8 physical cores or 16 in hyperthreaded mode and has 24 megabytes of last level cache memory and it does not support AVX instructions. Together these 4 processors provide us with as many as 32 physical cores (64 when hyperthreaded) on this server. We refer to these two systems with SNB and NHM respectively.

On the first experiment our goal was to prove the scalibility of our algorithm. Figures 4, 5 show the average running time of the algorithm when rendering towers model (figure 8) on SNB and NHM systems, respectively. The *BlobTree* of the towers model has 7360 operators and 7296 primitives and a depth of 64 levels. In this test the cellsize parameter kept as a constant value of 0.14 which we found it to be a balance between number of triangles produced and the quality of the output mesh. In order to show the effect of SIMD optimizations we have tested our algorithm with scalar, 4-wide SSE and 8-wide AVX instructions. SSE being on average 4.58x faster than scalar and AVX being on average 7.35x faster than scalar run. As illustrated in figure 4 when the number of threads increases past 6, two threads run on every core; sharing hardware resources on the hyperthreaded cores. The slope is reduced because each thread gets less resources than it would if it ran alone on the core. Past 12 threads, we schedule multiple threads per core, and they start to thrash the cache; making the algorithm memory bound.

Figure 5 shows the performance of our algorithm when running on the NHM system. Doubling number of threads, doubled the performance of the algorithm on this machine up to 33rd thread. The same behaviour is shown and hyperthreaded cores start to compete for memory access when having more than 32 threads running on this machine.

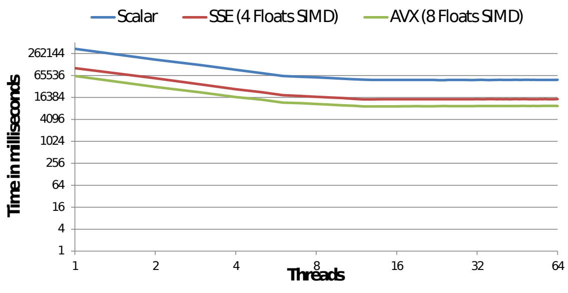

Figure 4: *Average polygonization time of the towers model when running on SNB processor. Horizontal axis is the number of threads. Vertical axis is time measured in milliseconds.*

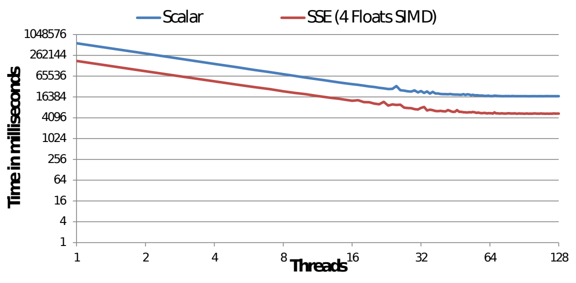

Figure 5: *Average polygonization time of the towers model when running on NHM processor. Horizontal axis is the number of threads. Vertical axis is time measured in milliseconds.*

Table 1: *Comparison of speedups and field value evaluations per triangle (FVEPT) for polygonization of Tower model with different SIMD instruction sets. Note that FVEPT was 17 before adding SIMD optimizations.*

Processor	SIMD Method	Speedup	FVEPT
SNB	SSE	4.58x	4
SNB	AVX	7.35x	2
NHM	SSE	4.25x	4

Table 1 shows the effect of using SIMD optimizations in our algorithm. With SSE and AVX the theoretical speedups are 4 and 8 times, respectively. Due to memory alignment techniques and proper caching mechanisms the speedup with 4-wide SSE is greater than 4. The AVX speedup can be improved more once scatter/gather instructions are implemented on the SNB processors which will improve the performance of surface extraction algorithm. Number of field evaluations per triangle shows the average amount of times the field evaluation kernel called to compute a single vertex in the output mesh.

In another experiment we studied the effect of our early discard method when the side of each *MPU* is less than one (figure 6). Starting from a large cellsize, we reduced the cellsize in uniform steps and measured the polygonization time. The red curve shows the polygonization time when the discard method described in section 6.2 is not being used and the blue curve is the timing when that method is in effect. Note that with the blue curve as soon as the *MPU* side is less than one; (*cellsize* = 0.14) empty *MPU*s started to get discarded efficiently thus the constant part of the time value is reduced at that point.

Figure 7 shows the polygonization time breakdown when rendering the towers model on SNB processor. Horizontal axis is the core number for a total of 12 cores on that system. As can be seen from the top of this chart; the idle time is very short and the cores are active almost all the time. This

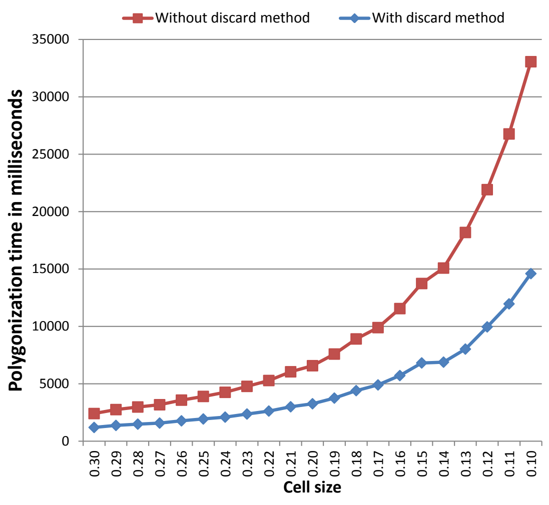

Figure 6: *Reducing cellsize parameter results in more MPU generation and increase in polygonization time. However, at a certain cellsize our early discard method stops polygonization time increase by rejecting all empty MPUs more efficiently.*

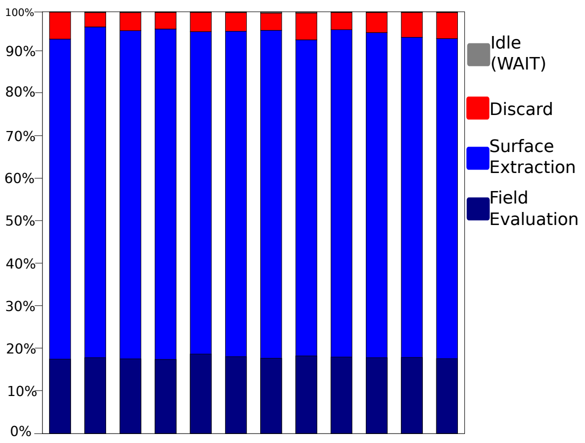

Figure 7: *Towers model per-core time breakdowns. Each bar represents a logical core on the processor for a total of 12 cores. Vertical axis is the total polygonization time. 190463 MPUs processed with 12 cores in 9283 milliseconds. This chart shows the portion of time spent in each step of the algorithm when rendering the towers model on the SNB processor with 8-wide AVX instructions.*

shows that the work stealing algorithm scales well. 190463 *MPU*s are processed and 116723 of them are intersected with the iso-surface (40 percent were empty). 40 percent of the *MPUSET* has been processed in less than 10 percent of the total polygonization time.

These results demonstrate the scalibility of our algorithm both in the number of SIMD vector lines and the number of cores available on each processor.

Finally, we compare our method against Schmidt et al. 's [SWG05] using the Medusa model provided by them which has 2920 primitives and 11 operators and the tree structure has a depth of 6 (figure 9). In this experiment, we divided polygonization timings reported in [SWG05] by 8 as the best AVX optimized version of Schmidt's method. Then we ran our polygonization algorithm optimized with AVX instructions on a single core for Medusa model (See table 2).

The results shows that our algorithm outperforms that of Schmidt et al. by a factor of 6 when running on a single core in lower resolutions.

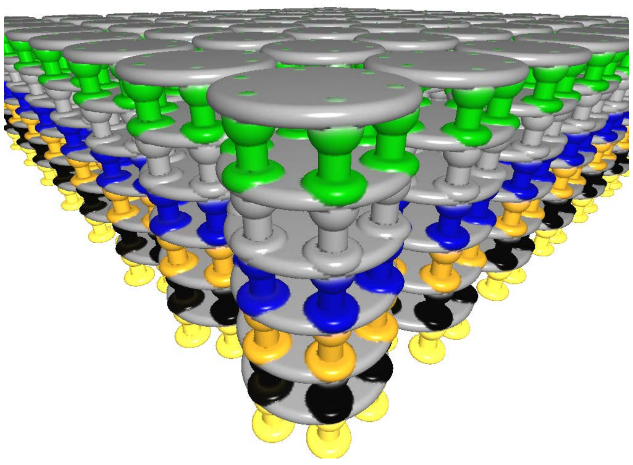

Figure 8: *Towers model created with skeletal primitives and binary operators in our incremental designing system. The model is a grid of 8 by 8 towers for a total of 7360 operators and 7296 primitives.*

8. Conclusions and future Work

We have presented a new parallel polygonization algorithm using SIMD processing techniques that takes advantage of a multi-core machine. Our main contribution is a scalable algorithm both in terms of the number of cores available on multicore architectures and the number of SIMD vector width as shown in the results section. We also presented a SIMD technique for finding the intersection of an iso-surface and a cube edge.

For future work we are interested in improving the perfor-

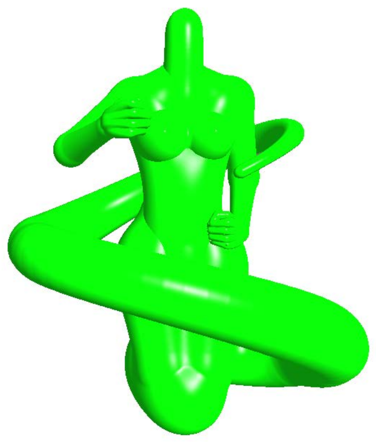

Figure 9: *Medusa model courtesy of Schmidt et al. [SWG05].*

Table 2: *Comparison of our polygonization method against Schmidt et al. 's [SWG05] when rendering Medusa model at 5 different resolutions on one single core with AVX instructions. All timings are in milliseconds.*

CellSize	Our method	Schmidt's method	Speedup
0.01	5220	6228	1.19x
0.03	3441	3653	1.06x
0.05	1071	2175	2.03x
0.10	264	1292	4.89x
0.14	108	721	6.67x

mance of our system on Many Integrated Cores architecture (MIC) such as the one shown in Larrabee [SCS*08]. We are also interested in testing our algorithm for precise contact modeling, where the *BlobTree* model is deformed over time while a collision detection algorithm is being evaluated. Finally, we are also interested in improving the visualization of a haptics enabled simulator [LD07], based on an implicit surface.

9. Acknowledgement

We would like to thank Intel Corporation for their support and providing us with their cutting-edge server and processors. This work is partly supported by the GRAND NCE foundation of Canada, and the Natural Sciences and Engineering Research Council of Canada.

References

[BBCW10] Bernhardt A., Barthe L., Cani M.-P., Wyvill B.: Implicit Blending Revisited. *Computer Graphics Forum 29*, 2 (June 2010), 367–375. 1

[BCB*97] Bloomenthal J., Chandrajit B., Blinn J., Cani-Gascuel M.-P., Rockwood A., Wyvill B., Wyvill G.: Introduction to implicit surfaces. *Morgan Kaufmann* (1997). 1, 3

[Blo94] Bloomenthal J.: An implicit surface polygonizer. *Graphics gems IV 1* (1994), 324–349. 2, 5

[BS93] Bolosky W., Scott M.: False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems-Volume 4* (1993), vol. 1801, USENIX Association, pp. 3–3. 5

[CGD97] Cani-Gascuel M.-P., Desbrun M.: Animation of Deformable Models Using Implicit Surfaces. *IEEE Transactions on Visualization and Computer Graphics 3* (1997), 39–50. 1

[CMM*97] Cignoni P., Marino P., Montani C., Puppo E., Scopigno R.: Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (1997), 158–170. 3

[DSC*09] Dietrich C. a., Scheidegger C. E., Comba J. a. L., Nedel L. P., Silva C. T.: Marching Cubes without Skinny Triangles. *Computing in Science & Engineering 11*, 2 (Mar. 2009), 82–87. 2, 5

[FGW01] Fox M., Galbraith C., Wyvill B.: Efficient use of the BlobTree for rendering purposes. In *Proceedings of the International Conference on Shape Modelling \& Applications* (2001). 5

[GH95] Guéziec A., Hummel R.: Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Transactions on Visualization and Computer Graphics 1*, 4 (1995), 342. 2

[GVJ*09] Gomes A. J. P., Voiculescu I., Jorge J., Wyvill B., Galbraith C.: *Implicit Curves and Surfaces: Mathematics, Data Structures, and Algorithms*. Springer Verlag, 2009. 1

[HH92] Hansen C., Hinker P.: Massively parallel isosurface extraction. In *Proceedings of the 3rd conference on Visualization'92* (1992), IEEE Computer Society Press, IEEE Computer Society Press, pp. 77–83. 2

[JC06] Johansson G., Carr H.: Accelerating marching cubes with graphics hardware. *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research - CASCON '06* (2006), 39. 2

[KHH*07] Knoll A., Hijazi Y., Hansen C., Wald I., Hagen H.: Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic. *2007 IEEE Symposium on Interactive Ray Tracing* (Sept. 2007), 11–18. 3

[KW05] Kipfer P., Westermann R.: GPU construction and transparent rendering of iso-surfaces. In *Proceedings Vision, Modeling and Visualization* (2005), vol. 5. 2

[LC87] Lorensen W., Cline H.: Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (1987), vol. 87, ACM, p. 169. 2, 5

[LD07] Laycock S., Day A.: A Survey of Haptic Rendering Techniques. *Computer Graphics Forum 26*, 1 (Mar. 2007), 50–65. 2, 9

[LS00] Lee M., Samet H.: Navigating through triangle meshes implemented as linear quadtrees. *ACM Transactions on Graphics 19*, 2 (Apr. 2000), 79–121. 5

[Mac92] Mackerras P.: A fast parallel marching-cubes implementation on the Fujitsu AP1000. *Computer Science Technical Report TR-CS-92-10, The Australian National University* (1992). 2

[Mat87] MATTHEWS J.: Numerical Methods for Computer Science, Engineering and Mathematics. 6

[RA05] RODRIGUESDEARAUJO B., ARMANDOPIRESJORGE J.: Adaptive polygonization of implicit surfaces. *Computers & Graphics 29*, 5 (2005), 686–696. 3

[Rei07] REINDERS J.: *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*, vol. 23. O'Reilly Media, Inc., 2007. 5

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., OTHERS: Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers* (2008), ACM, p. 18. 9

[SWG05] SCHMIDT R., WYVILL B., GALIN E.: Interactive implicit modeling with hierarchical spatial caching. *International Conference on Shape Modeling and Applications 2005 (SMI' 05)* (2005), 104–113. 3, 8, 9

[SWSJ06] SCHMIDT R., WYVILL B., SOUSA M., JORGE J.: Shapeshop: Sketch-based solid modeling with blobtrees. In *ACM SIGGRAPH 2006 Courses* (2006), ACM, p. 14. 1

[TGMC03] TRIQUET F., GRISONI L., MESEURE P., CHAILLOU C.: Realtime visualization of implicit objects with contact control. *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Austalasia and South East Asia - GRAPHITE '03 1*, 212 (2003), 189. 2

[TSD08] TATARCHUK N., SHOPF J., DECORO C.: Advanced interactive medical visualization on the GPU. *Journal of Parallel and Distributed Computing 68*, 10 (2008), 1319–1328. 3

[WGG99] WYVILL B., GUY A., GALIN E.: Extending the CSG Tree - Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. In *Computer Graphics Forum* (1999), vol. 18, pp. 149–158. 1, 3

[WMW86] WYVILL G., MCPHEETERS C., WYVILL B.: Data structure for soft objects. *The visual computer 2*, 4 (1986), 227–234. 2, 4, 5

[YCP10] YANG B., CHEN G.-L., PANG M.-Y.: Parallel Polygonization of Implicit Surfaces. *2010 International Symposium on Intelligence Information Processing and Trusted Computing* (Oct. 2010), 220–223. 3

[ZWW06] ZHANG Y., WANG X., WU X.: Fast Visualization Algorithm for Implicit Surfaces. *Cell* (2006), 0–5. 5