Efficient Data-Parallel Tree-Traversal for *BlobTrees* (revised)

Herbert Grasberger, Jean-Luc Duprat, Brian Wyvill, Paul Lalonde, Jarek Rossignac

Abstract

The hierarchical implicit modelling paradigm, as exemplified by the *BlobTree*, makes it possible to support not only Boolean operations and affine transformations, but also various forms of blending and space warping. Typically the resulting solid is converted to a boundary representation, a triangle mesh approximation for rendering. These triangles are obtained by evaluating the corresponding implicit function (field) at the samples of a dense regular three-dimensional grid and by performing a local iso-surface extraction at each voxel. The performance bottleneck of this rendering process lies in the cost of the tree traversal (which typically must be executed hundreds of millions of times) and in the cost of applying the inverses of the space transformations associated with some of the nodes of the tree to the grid samples.

Tree pruning is commonly used to reduce the number of samples for which the field value must be computed. Here, we propose a complementary strategy which reduces the costs of tree both the traversal and of applying the inverses of the blending and warping transformations that are associated with each evaluation.

Without blending or warping a *BlobTree* can be reduced to a CSG tree only containing Boolean nodes and and affine transformations, which can be reordered to increase memory coherence. Furthermore, the cumulative effects of the affine transformations can be precomputed via matrix multiplication. We propose extensions of these techniques from CSG trees to the fully general *BlobTrees*. These extensions are based on tree reordering, bottom-up traversal, and caching of the combined matrix for uninterrupted runs of affine transformations in the *BlobTree*.

We show that these new techniques result in an order of magnitude performance improvement for rendering large *BlobTrees* on modern Single Program Multiple Data (SPMD) devices.

1. Introduction

1.1. Motivation

The *BlobTree* [46] as a data-structure for Implicit models, has many advantages over other modelling methods, such as blending [17]. In order to visualize an implicit model the value of the field function has to be found for many points in space. Each function evaluation requires a tree traversal, in which a significant proportion of the traversal time is spent calculating which is the next node.

Several researchers have shown that visualization methods can indeed be fast enough to re-create a new mesh object from a *BlobTree* at interactive frame-rates (e.g. [36, 37]). Other approaches have shown, that ray-tracing times can be reduced, for example using interval-arithmetic [24] to find the intersection of a ray with the *BlobTree* surface. These approaches have in common that the speed up is achieved by reducing the number of implicit (*BlobTree*) evaluations at points in space, but little work has been done to improve the time for a tree evaluation.

Recent advances in *BlobTree*-modeling introduce more complex operators to expand the capabilities of the *BlobTree* [5, 17] , solving four long-standing problems (bulging, blending at distance, small details are lost in blending and undesired filling of holes), and offer greater control to the user. Unfortunately, these advantages come at the cost of increased computation time over the standard operators. As a result, it is even more important that tree traversal time is reduced to preserve modelling interactivity. Additionally, by accelerating the traversal itself, any method based on the tree traversal is accelerated as well. Even basic user interaction, such as calculating a point on the *BlobTrees* surface, will benefit from a faster *BlobTree* traversal.

1.2. The BlobTree

BlobTree modelling is a derivative of Constructive Solid Geometry (CSG) [31]. Much work has been done to improve CSG tree traversal [22, 19, 34], as well as novel rendering techniques [32]. Unfortunately, approaches improving tree traversal for CSG are not necessarily applicable in our case, due to the different mathematical formulation of the *BlobTree*. Depending on the application, CSG evaluation algorithms classify points, line segments, or surfaces, but always return point sets (possibly augmented with set membership maps for their neighborhood), while *BlobTree* evaluation algorithms return a scalar value at a query point. Direct CSG classification algorithms classify (i.e., trim) these candidate sets against the leaves of the CSG tree (i.e., the primitive solids) and then merge the classified subsets up-the-tree, according to the Boolean operations [25]. CSG trees that support such algorithms are limited to nodes that represent regularized Boolean set operators (Union, Intersection, Difference) and affine transformations. Offsetting, blending, and Minkowski operations require evaluating a boundary representation of the solids associated with the argument nodes of such operations and hence do not lend themselves to a direct CSG evaluation. In contrast, *BlobTrees* evaluate scalar values

at the query point, one per *BlobTree* leaf (primitive) and then blend, filter and combine these values according to various formulae [3, 17], which may reproduce Boolean operations and also their blended versions. This includes bending, twisting, tapering and even warping based on Free-Form Deformation approaches. Recently a subset of the *BlobTree* has also been used to improve skinning [42]

1.3. BlobTree Traversal Scenarios

Typically *BlobTree* field-values are calculated multiple times along one ray, where a large number of rays are cast, or along a dense grid of points (e.g. 32^3 for the use case presented in Section 7). The final object's surface is placed where field-value samples cross a given iso-value. Whenever a *BlobTree* is traversed to create a field-value, the time of such a evaluation can be decomposed into time spent calculating field-values at leafs, combining field-values at operators, calculating the next node in the traversal, memory transfers loading node data and saving and loading the intermediate computation results. The first two measures depend on the primitives and operators used in the model and thus cannot be influenced by a *BlobTree* traversal change. In contrast, the last three measures are directly influenced by a traversal algorithm and are addressed in this work.

1.4. Contributions

We show how to accelerate the tree traversal for the *BlobTree* using an approach that results in an $O(n)$ traversal time (n nodes in the *BlobTree*), where every node in the tree is only accessed once, compared to three times for interior nodes and once for leaf nodes in the default recursive traversal. This allows us to exploit predictable memory access patterns, important for performance on modern SIMD and SPMD architectures, such as GPUs using OpenCL, or using vector instructions on multi-core CPUs. In our approach the tree information is stored in a linear memory pattern, preferred by modern CPU/GPU architectures. This technique can improve traversal speed by as much as an order of magnitude, compared to previous approaches running on an SPMD architecture.

1.5. Outline

The remainder of the paper is structured as follows. Section 2 discusses related work in SPMD programming, as well as *BlobTree* and CSG acceleration. In section 3 we summarize the most efficient CSG traversal accelerations and in section 4 we describe how some of these changes can be applied to the *BlobTree*. Section 6 introduces our implementation, we discuss our results in section 7 and conclude the paper with future work in section 8.

2. Related Work

The publications related to this paper can be divided in three distinct topics: using the SPMD programming model to accelerate parallel calculations on the same data, accelerating *BlobTree* rendering and accelerating CSG rendering. Section 3 summarizes the CSG rendering accelerations based on optimized an tree traversal in greater detail as a basis for this work.

2.1. The SPMD programming model

Current computer architectures provide two main paths for accelerating floating-point heavy workloads: SIMD (Single Instruction Multiple Data) units, which evaluate the same floating point operator on multiple (typically 4, 8, or 16) elements of data; and GPUs, re-purposed to general computation using a large set of SIMD-like processors, with hardware predication for divergent control flow. This last model is better known as the Single Program Multiple Data (SPMD) [9] programming model. In an SPMD program a single execution stream is applied over a large number of independent data elements. Applying the SPMD model usually involves rethinking the algorithm with respect to optimal memory transfer and problem blocking [2] to allow maximum data independence. Several high-level programming languages help the programmer to create code to run on SPMD architectures, such as OpenCL [26] or CUDA [27] for GPUs [12], or ISPC [28] for CPUs. Tree Traversal on GPUs in general, using the SPMD programming model has been discussed by [16], who propose storing additional data ("Autoropes") on a stack during traversal to accelerate the process. The traversal, however is still done top-down, compared to our advocated bottom-up approach. In addition to autoropes, the paper also discusses interaction between GPU threads in order to optimize performance, something left to future work for our proposed approach.

2.2. Accelerating BlobTree rendering

The *BlobTree* [46] is a hierarchical modelling approach with a unified structure in which nodes can represent arbitrary combinations of operators and primitives. It builds on the notation of skeletal implicit surfaces [7] as the primitives and exhibits operators that go beyond the classical CSG union, intersection and difference, starting with the widely known summation blend, the Ricci-blend [31, 38], as well as more complex operators [3, 5]. All visualization approaches, including polygonization [47, 6, 1] and ray tracing [23, 40] rely on iterative root-finding by sampling the field-value and gradient (based on differencing) functions.

Previous work on improving the traversal time of *BlobTrees* include work by [14], which aggregates nodes in the tree, to reduce the overall node count. Furthermore a simple approach using spatial subdivision together with pruning the tree for each subdivision node is also suggested and explored further in [10]. Caches within the tree structure were introduced by [36], in order to reduce the number of field value calculations, in favour of interpolation of field-values as soon as enough values are found within the cache structure. This work enabled interactive editing through fast polygonization, however for edit operations it still relied on polygonizing at a coarser resolution to allow for fast interactive feedback. Polygonization itself is an algorithm that lends itself to an implementation using SIMD [37] which makes use of linearizing a tree structure into continuous memory and calculating multiple field-values in parallel, since each calculation is independent.

In all of the above approaches rendering times were improved significantly, but none include methods to reduce the time a single tree traversal takes. It is not obvious that improving the time

taken by the calculation of the field-values is possible. The formulas for primitives and operators are constant in that they cannot be changed without changing the resulting values. Thus, it is a very important condition, that any acceleration approach does not alter the result of the field-value calculation at a point in space. We will show that it is possible to accelerate the field calculations without altering the underlying formulas, by taking certain hardware properties, especially those of modern GPUs, into account.

In another GPU approach [30], the model is built by altering the GLSL shader code, that renders models directly to the screen, but is less practical than our approach as a new shader is required whenever the model changes. Other approaches such as [18], present a method to ray-trace meta-balls on the GPU. Neither of the above deal with tree structures similar to the *BlobTree*, and thus these are not comparable systems.

2.3. Accelerating CSG rendering

Most of the current state of the art GPU rendering techniques for CSG models are based on the formulation of a CSG tree as a *Boolean list (Blist)* [33]. *Blister* [19] and subsequently [20] use this Blist formulation of a CSG tree in a rendering technique based on depth peeling. The depth peels are classified for each primitive based on stencil bit true/false values that are then combined using the Boolean expression given by the CSG tree. The optimization methods in these approaches resulted in the formulation of *Ordered Boolean Lists* [34] which can evaluate a Boolean expression in $O(loglogn)$ space.

Other approaches to render CSG models on GPUs include [32], where CSG objects are subdivided until each child is simple enough (one primitive or Boolean operator of two primitives) for rendering on the GPU. This subdivision is done on the CPU. It is also possible to use a face representation precomputed on the CPU to directly render the CSG objects using boundary representations [21].

Using a tree structure efficiently in an SPMD context requires linearizing the tree into a continuous block of memory. Many approaches for linearizing tree structures store the tree nodes in top down order, with offset pointers used to do the traversal [39] and [8]. Some approaches insert additional offset pointers so it is possible to directly go to the in-order predecessor or successor by one single offset pointer (threaded tree [45]), instead of reading the parent pointer again in order to find the neighbouring child node. Linearizing and traversing trees has been important in the context of acceleration structures [43], [39], [8], [4] and [29]. Unfortunately we cannot use these methods in our case, since these acceleration structures try to avoid traversing the entire tree, which is a requirement for a complete *BlobTree* traversal.

3. Methods to accelerate CSG tree traversal

The most efficient methods to accelerate the traversal of CSG trees involve writing the tree as a Boolean expression. In such an expression, the leaves of the tree correspond to the literals, which at any 3D point either evaluate to *true* or *false*. The operators (notation as per Rossignac [34]) in the expression can be of the limited set of union, expressed as a Boolean OR ("+"), intersection as AND ("•") and the difference operator as AND NOT ("•!"), where "!" denotes the complement. Operators that can create more complex transitions between surfaces, such as the ones described in [11] and [44]) are usually not considered in these approaches. Any tree only consisting of the three simple operators stated above can be described using such a Boolean expression which can be evaluated in parallel for any input points and classifies this point against the CSG model surface.

A CSG tree that uses only these three operators ("+, •, !") and that has the "!" operator pushed to the literals using the de Morgan laws is called a *Positive Form Expression* (PFE). In this expression, which is the basis of the *Blist* wiring process for optimization, only the $+$ and $\bullet$ operators exist; the "!"-operator being expressed as a parameter to the actual primitives. Both operators in this PFE are commutative, so they can often be swapped to make the tree left heavy, which can reduce the footprint of the expression (see section 4.2 for a discussion on left heavy trees). The Blist wiring process uses the metaphor of an electrical circuit, where every literal is expressed as a switch reading its Boolean value. Each switch has a top output, representing it being true, and the bottom output, representing it being false.

Two switches A, B can be connected together to form either the expression $A+B$ or $A \bullet B$, by altering the connections between input and output. This allows the introduction of connections that can skip the evaluation of nodes, e.g. in the expression of $A+B$ the result is already set to *true* as soon as A evaluates to *true*. There is no need to evaluate B in this case. Similarly for the $A \bullet B$ case, if A evaluates to *false*, the whole expression results in *false*, not requiring B to be evaluated. In an Ordered Boolean List the Blist structure is then stabilized (reducing the nodes width, as defined in [34]) by continually swapping nodes to effectively re-order the tree resulting in the smallest memory footprint possible for each expression. This results in OBLs, that can be created from any expression and can be evaluated in $O(loglog\ n)$ space.

4. Techniques applicable to the *BlobTree*

Compared to CSG which can be reduced to Boolean values, the *BlobTrees* skeletal implicit primitives are based on a modified distance field $d(P)$ (for a set of points P) to a given skeleton (defined by a distance function). In order to bound the field to finite space, this distance d is modified using a filter-fall-off-function, often defined as $f(d) = (1-d^2)^3$ [38]. A surface is defined by the set of points, whose field-values match a given iso-value c. Any binary operator node within the *BlobTree* is a function having two field-values f_1 and f_2 (calculated at the two child nodes for the same input coordinates) as a parameter, such as for the simplest operators ([31]):

- union: the maximum of f_1, f_2
- intersection: the minimum of f_1, f_2
- difference: the minimum of f_1 and $1-f_2$

- blend: a variation of the $\sum(f_1, f_2)$.

More sophisticated blend operators provide the user with greater control over the resulting shape, at the cost of computationally more expensive operator functions (eg. [3] and [17]).

Optimization approaches which rely on the simplification of Boolean expressions are not applicable to the *BlobTree*. Unlike Boolean CSG operators, both sides of a *BlobTree* operator have to be evaluated for blending, especially when blends are based on the child gradients [17]. All nodes return scalar values, that are combined as described above, potentially changing the inside/outside classification for a point P based on its numerical value. For example using the Ricci operations, union is $\max(f_1, f_2)$. Even if $f_1 > c$ (CSG-true) it still has to be evaluated as f_1 may be greater than f_2.

Looking at the difference operator, we emphasize that pushing an *invert* operator to the leaf nodes, as done in OBL to simplify the expression, does not work for *BlobTrees*: Given that in the *BlobTree* the right child's field-value is not just negated, but the complement is calculated using $1 - f$, an inversion at the child node level is not possible, since f can be the result of a non-associative operator. For these reasons only a subset of the aforementioned CSG acceleration methods can actually be applied to the *BlobTree*.

4.1. Hardware Considerations

Memory reads and writes can have a significant impact on the performance on modern processors, both for CPUs and GPUs, which accelerate applications using prefetching mechanisms and hardware caches. In case the memory footprint of the *BlobTree* is very small, it might be possible to fit a large portion or all of it into the hardware cache, avoiding the more expensive read from main memory.

Because current GPUs make use of cache line aligned reads of memory, they prefer memory access patterns that are easier predictable (e.g. linear reads compared to random access). An example outlined by [2] is that linear memory reads are hardware accelerated, whereas random access is not. Provided that the data-structures are stored in a way to support cache aligned reads and writes, a coherent memory access pattern can be achieved, resulting in a significant performance improvement. A good performing *BlobTree* traversal algorithm, has a reduced and cache line size aligned memory footprint (see section 4.2), a decreased number of reads/writes to the temporary storage (see section 4.3) and (or) a reduced number of memory access direction changes (see section 7).

4.2. Linearizing a BlobTree

In order to use a tree data structure efficiently on modern SPMD architectures (GPUs or CPU vector instruction sets) it is necessary to store the whole tree in contiguous blocks of memory. The usual implementation, where links in the tree are represented by pointers, can lead to bad performance, since memory reads are harder to predict for the hardware, and in many cases the pointers wouldn't lead to a cache aligned memory distribution either. Previous approaches to improving tree traversal are based on *linearizing* the tree structure (information about parent - child relation) and data (information on the type of node) into arrays of contiguous memory [45], [39], [8], [43], [37].

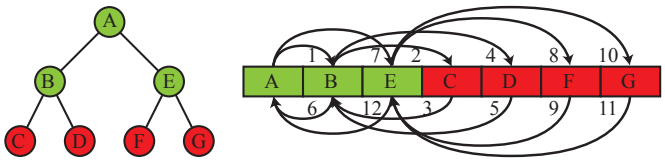


Figure 1: The order of the tree nodes in memory as proposed by [43], traversed iteratively. The numbers describe the order of memory reads. Top arrows: reads going deeper in the tree; numbers are closer to arrow head. Bottom arrows: reads going back up in the tree; numbers closer to arrow base.

A simple application that traverses the tree top-down iteratively and stores child relations only in the parent node requires that the parent node is read after visiting the first child. More generally, any top-down traversal algorithm of a tree maintains two stacks of temporary memory:

- the traversal state (information about the previous visited node), m
- the temporary results at the nodes, to be used in the parents, t

Both of these stacks depend on the number of nodes n within the *BlobTree* and its structure.

Figure 1 shows how memory is read (not including the temporary storage stack) in such an approach, resulting in 12 memory reads for the traversal, with 5 changes in the read direction for this small example. In this case, we make use of a tree storage optimization proposed by [43] that stores child nodes in neighbouring array elements, removing the need to store 2 index offsets per parent node pointing to the children. Whenever our traversal moves deeper down the tree, the current node is pushed to our traversal stack m. In case of a primitive, we compute the data with the corresponding primitive function for p and store it in our temporary stack t. If the current node is an operator, we have to check if both children have been visited and if a child is still to be processed we push the current node to the stack and make the current node the child node. As soon as both children of an operator have been calculated we can use their results stored on stack t to calculate the data values of the operator, write it to the stack t and use the information in m to move back up the tree towards the root node.

Apart from the fact that the tree itself is read in a non predictable way (figure 1), both of our stacks are involved in a lot of reads and writes. Stack m has the size $|m|$ of the maximum height found in the tree. In order to quantify the size of stack t, we have to define the property *right-branching depth* r_d, which can be calculated recursively. If, an interior node N has two leaf nodes L, its right-branching depth r_d is 0. For the case that N's *right child* is an operator node N, then the right-branching depth is $\max(r_d(L), r_d(N) + 1)$. The resulting value, incremented by 2, corresponds to the size of the temporary storage stack t. For any tree $|m| \geq |t|$.

Figure 2: The order of the tree nodes in memory as proposed by [43], traversed using a threaded tree approach. Top arrows: reads going deeper in the tree; numbers are closer to arrow head. Bottom arrows: reads going back up in the tree; numbers closer to arrow base.

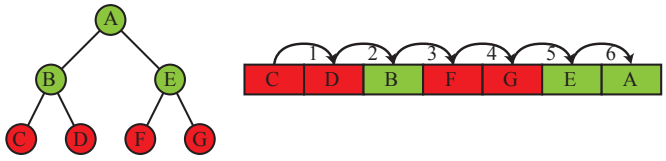


Figure 3: The order of how the tree nodes are stored optimally for reading the tree bottom up. The numbers describe the order of memory reads for the full tree. It results in: less memory reads, memory is only read in one direction and only once.

We use this approach as our performance base line, in which every interior node N is visited three times, every leaf L visited once. Optimizing the tree storage itself, and as a result the number of stacks/stack frames needed, can lead to a much better tree traversal performance. The number of interior node N visits can be reduced, as demonstrated below.

4.3. Removing the need for a traversal stack

Much work has been done how to represent parent child relations of a tree in memory in order to avoid unnecessary memory reads, e.g. in theory it is not necessary to visit the parent N node to change from child L to child R. As a result a *threaded* tree stores this relation directly in every child node by storing the offset to the in-order predecessor and successor [45]:

- node L stores an array offset to N and to R
- node R stores an array offset to L and N.

In cases where L or R are trees and not leaves these offsets would be to the appropriate leaf nodes instead. Since for a *BlobTree* the parent nodes combine the results of the child nodes, these need to be visited after processing the right child, something not needed for the general threaded tree approach. The memory layout of the tree and the corresponding reads using the threaded approach are shown in figure 2, which reduces the number of memory reads to 9. Even though this approach already removes the need to keep a traversal stack m, as presented by [29], this approach still relies on changing the read direction in memory 3 times, and is not optimal for all SPMD architectures [2].

Looking at the architecture of modern SPMD hardware, especially GPUs, non-cached reads from main memory can often be very slow. This makes finding an approach that does not need to store offset pointers to traverse the tree a desired goal. In fact, not having to store offsets for parent-child relations means that only the tree data is needed.

A *BlobTree* can be treated as a mathematical expression, with the literals being the leaf nodes that are combined using operators, we can apply the same approach as already done in a Blist for CSG, as well as for abstract syntax trees in the context of compilers [13]: rewriting the expression in reverse Polish notation. In this notation both operands precede the corresponding operator, thus the expression $A \circ B$ would be rewritten as $AB\circ$. This corresponds to a post-order / bottom-up tree traversal, shown in figure 3. Compared to a top-down approach, the memory access pattern of the tree data is simpler, reads are only done in one memory direction, independent of tree size and structure, and as a result easier to predict for current hardware. This results in 6 reads from memory, compared to the 12 in the original approach. If a certain tree node is of a fraction of the size of a cache line, it can happen that if one node is read into memory, the following node is read as well.

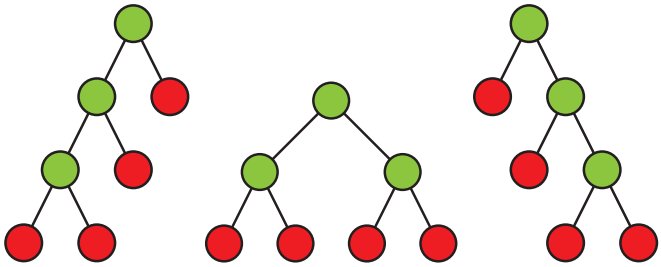
For every field-value computation, we also need to store the intermediate calculation results, in order to combine them at operator nodes. In a recursive approach, storage for these intermediate results is implicitly given by the recursion stack, which the bottom-up approach does not require. Since the traversal is based on the reverse Polish notation of an expression, the storage layout for the temporary results is a stack, where every intermediate result is pushed on, and at an operator the two last results are popped from the stack, so they can be combined. This results in algorithm 1 (for a single thread/field-value calculation).

Algorithm 1 bottom-up traversal

```
1:  function DataAtLinear(Point p, TreeArray a, TemporaryResults t)
2:      n ← a.length
3:      for i = 1 → n do
4:          curData ← a[i]
5:          if curData is a primitive then
6:              curRes ← dataAtPrim(p, curData)
7:              push curRes to t
8:          else
9:              childResults[1] ← pop from t
10:             childResults[0] ← pop from t
11:             curRes ← dataAtOp(p,curData,childResults)
12:             push curRes to t
13:         end if
14:     end for
15:     return pop from t
16: end function
```

4.4. Optimize the tree to require less temporary storage

Given, that our implementation targets SPMD architectures, and multiple field-values will be calculated in parallel, the temporary results stack t is needed for every thread. The more threads are run, the more storage is needed. The maximum size of the stack depends on the structure of the tree. A tree with n_l leaf nodes can be classified as left heavy (figure 4a), balanced



(a) Left Heavy (b) Balanced (c) Right Heavy

Figure 4: The three different extreme structures for a tree with n leaf nodes.

(figure 4b) or right heavy (figure 4c), or any combination in between. Depending on this structure, trees with the same number of leaves (primitives) can need a bigger or smaller stack to store the intermediate results. The best case, for left-child first traversal, is a left heavy tree, as shown below in section 7, since the extreme left heavy example in figure 4a can be traversed with a constant stack size of 2, independent of the number of leaf nodes n. Optimizing a tree, so that it is as left heavy as possible is desired, in order to reduce the size for the temporary variable stack. The basis of this optimization is that the height at every node in the tree is computed, which can be done, when the tree is built, as shown by [19]:

1. A new primitive has height 0
2. The height of a new internal node is the maximum of the height of its children, plus 1

The process of swapping the tree nodes to produce a left heavy tree is integrated into the linearization of the tree into the reverse Polish memory layout. This process is only done once and sets up the tree ready for traversal for each field value query. If the height of the right child is bigger than the left, then the algorithm has to traverse the right child first, otherwise the left, essentially swapping left and right in this case. With the exception of the difference operator, all other operators in the *BlobTree* are commutative so that swapping has no effect on the resulting field value. A flag is set if a difference operator node is swapped and the algorithm assigns the child results in the opposite order.

This means, that the absolute worst case arrangement, in terms of temp storage requirements, of n_l leaf nodes, a perfectly right-heavy tree can be transformed into the best case, resulting in the temporary storage stack t of 2. In general, any right-heavy representation can be converted to a corresponding left heavy one, effectively making the previous average case, a balanced tree, to the new worst-case. A perfectly balanced tree has the t stack size requirements (right-branching depth $+2$) of $log(n_l)+2$, where n_l is the number of leaf nodes. Any other (already left-heavy transformed) tree with n_l leaves needs a stack size that is in between and 2 and $log(n_l)+2$.

In some cases, the structure of the tree can be converted to a left-heavy version by reordering parent and child relations. This does not work for all combinations of operators, only for ones that are associative and commutative in limited arrangements. The algorithm starts at a node N with children L and R that have been pivoted to be left-heavy. The leaves of L and R may be primitives or roots of subtrees with non-compatible operators. In addition, E is the left most child of R, F the parent of E and N one child of P. To pivot the nodes, the following steps have to be applied to the tree:

1. Replace R by E in N.
2. Replace E by N in node F.
3. In P replace N by R.

See Figure 5 for an illustration of before and after. This algorithm can only be applied when the path between N and F only consists of the same operator type supporting the pivot: the union, intersection and summation blend operators.

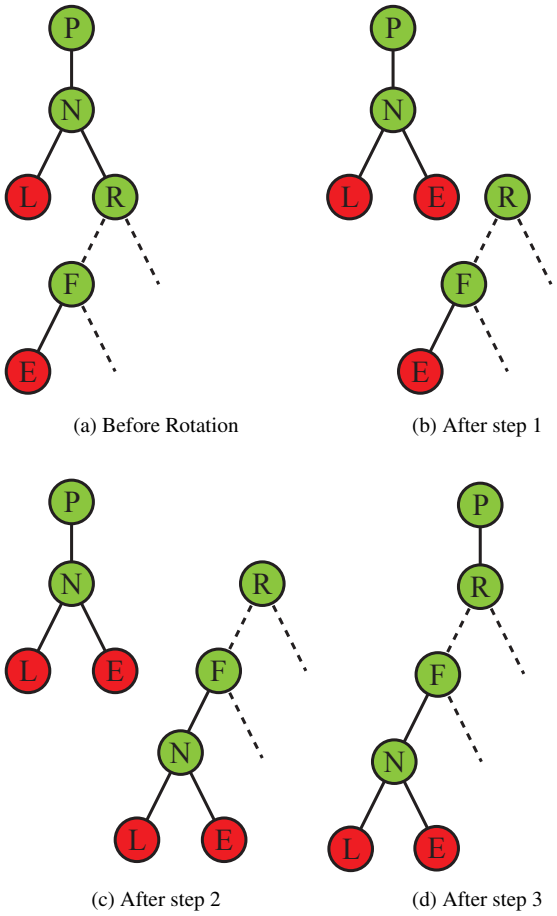


Figure 5: Conversion to left-heavy tree. Nodes N and F have to be compatible to allow the rotation. Dashed lines represent either subtrees or nodes with operator types supporting the pivot.

5. Incorporating Warp Transformations

In addition to affine transformations, the *BlobTree* also supports a series of warp transformation, such as the *Barr Warps*

[46] and more recently *Warp Curves* [41] as a form of Free-Form Deformation. Like affine transformations, these warp transformations are represented in the *BlobTree* as unary nodes at any point in the tree, transforming the whole subtree underneath. Figure 6 shows an example, where the blue nodes represent warp nodes. These nodes apply the inverse transformation to the specific input point coordinates, moving the input point according to the warp to its new location used for any further calculations. When the *BlobTree* is traversed top-down, the warps can be applied to the input point when encountered. The transformed input point will be used for the sub-tree of the warp node. In contrast, the bottom-up tree traversal, as proposed by this work, requires special handling of warp transformations in order to optimize their performance.

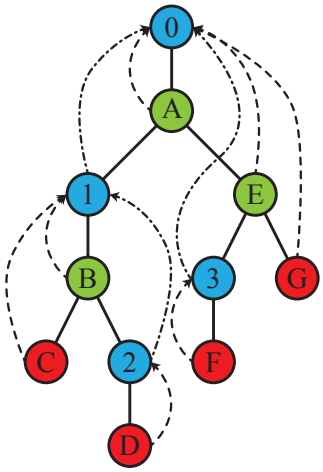


Figure 6: A *BlobTree* including warp transformations shown in blue. Stippled lines show the connections for each node to its parent warp. Stippling with line-dots connect the warp nodes

It has been shown by [43] that pushing the affine transformations to the leaf nodes can cause a large performance improvement simply by reducing the number of matrix-vector calculations during a tree traversal (not limited to *BlobTrees*), independent from the traversal method used. Often, linearizing the tree and pushing the affine transformations to the leaf nodes, where the transformations are stored as properties, is done at the same time. This makes it possible that the input point coordinates can be transformed into every leaf node's local coordinate system with only one matrix-vector multiplication, done once at every leaf node. As a result, the input point does not have to be transformed along the path from the root to the leaf, something required by our bottom-up traversal.

As soon as one warp transformation exists, that cannot be represented using a $4x4$ matrix, along the path from the root node to the leaf, all the affine transformations can't be pushed to the leaf anymore. However, pushing affine transformations and aggregating them from the *BlobTree* root downwards until the first warp node is reached is possible. Similarly, pushing and aggregating all affine transformations below a warp transformation is possible until either another warp node or the leaf is reached. As a result, aggregated affine transformations can still be stored as parameters to the leaf, instead of nodes within the *BlobTree*, with the addition that the warp nodes also store an affine transformation as additional property.

In a top-down tree traversal, this optimization would result in a *BlobTree* that contains unary warp nodes and leaf nodes, both with affine transformation properties, and binary operator nodes. As mentioned above, the inverse of these (affine and warp) transformations is applied to the input points before the tree traversal continues, until the leaf nodes are reached. Field values are calculated and then combined bottom up to produce the final traversal result.

Looking at this, a full *BlobTree* traversal in this situation consists of two steps:

1. top-down tree traversal: transforming the input points
2. bottom-up tree traversal: combining the field-values and potentially gradients and colour.

In both cases, the full *BlobTree* information is traversed.

It has already been shown, that in the case of a *BlobTree* only containing affine transformation, leaving out step one can provide significant performance improvements, since the full traversal can be done only within the bottom-up part. Now that warp transformations are involved, a single leaf node does not have all the information in a bottom-up traversal situation to calculate the correct values yet. A simple solution can store a list of all affine and warp transformations encountered along the path from the root node at every leaf. Such a list of transformations, however, will store the same warp transformations multiple times and can potentially also transform the same input multiple times, resulting in unnecessary work.

For this reason, a subset of the original *BlobTree*, the *Warp Tree* can be used as an optimization. It only contains the warp transformation nodes, to simplify the top-down traversal and transforms the input points and stores their respective results. Every leaf node stores a reference (index, offset, etc. depending on the implementation) to its closest warp parent node. The result of this warp transformation is then used as the input point for the bottom-up tree traversal, done without changing the algorithm described above. Figure 7 shows the tree of Figure 6 split into its *Warp Tree* and the corresponding *BlobTree*. Their memory layouts are illustrated underneath, with memory traversals and lookups shown by arrows. The stippling of the arrows corresponds to Figure 6.

In the same way, any warp node stores a reference to its closest warp parent in order to pick the right input values in the case that several warp transformations occur along the path to a leaf. Linearizing this reduced *Warp Tree* in depth-first-order into a contiguous array allows for a linear traversal algorithm. This traversal order ensures that at any node, the previous transformation node is already applied to the input points and the result can be read and used for the consecutive calculation. Applying the same data layout techniques described above for the bottom-up, field-value "gather" step, will also result in a cache-efficient data layout to store the warp transformations, which, depending on their type, are more (Barr warps) or less (Warp Curves) trivial. In the same way, the metrics to optimize are memory usage, memory read direction and random memory access.

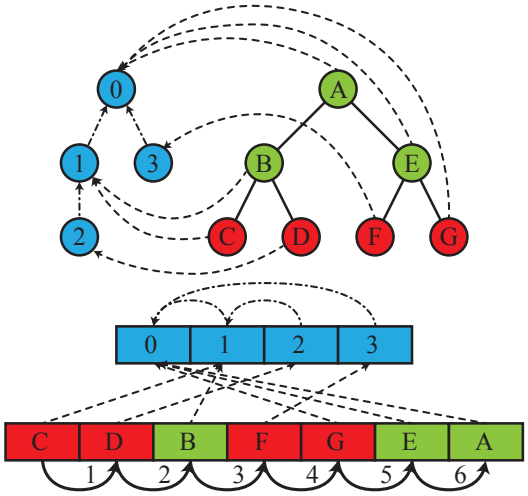


Figure 7: Separation of tree from Figure 6 into *Warp Tree* and the *BlobTree* nodes, including their corresponding memory layout and access patterns.

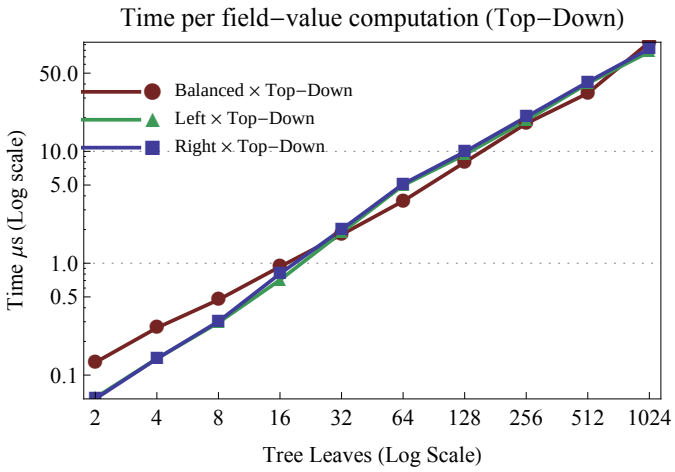


Figure 8: The running times for the computer-generated test scenes, traversed with the top down approach. Note that the "Left" and "Right" cases overlap.

6. Implementation

The implementation of the algorithms use OpenCL 1.2 and are run on a GeForce GTX 780 M. We separate the tree into data and structure (section 4.2). Only the tree data is needed for the bottom-up measurements, but the top down also requires the the structure. To minimize the number of cycles it takes to load the nodes into variables, it is desirable, that an integer number of tree nodes fit into a cache line (equivalent).

For the node data we currently store the type of the node and node-type specific data. Since our node types are mutually exclusive, but are very close in size, we decided to store them as a C-union of types, to make memory allocation, cache line alignment and array handling easier. The current size of our nodes are 64 bytes (an integer fraction of the cache line equivalent on our GPU). When the tree is traversed, node-type specific functions are called depending on the node type using a switch-case statement. In the case of our base-line top-down traversal, we store the memory as suggested by *Wald* [43], where child nodes are stored next to each other (see figure 1). This means that for storing the tree structure, we only need one index, for the first child, and the second child can be found by incrementing this index, resulting in a 4 byte small value for every tree node. If needed this approach can easily be extended to n-ary tree nodes, but in this work we restrict our modeling tree to binary.

Since the field-value calculations are used in polygonization and ray-tracing to calculate surface points, a single field-value calculation also returns compressed, signed, gradients and 8-bit RGB color values, for a total size of 16 bytes, stored for every intermediate calculation result. Given that gradient and color calculations at a tree node depend on the field-value, this avoids re-calculating the same field value multiple times.

The number of intermediate results stored is dependent on the input model's tree structure (see figure 4), and the chosen algorithm, and therefore calculated during linearization. Temporary memory is allocated so that every thread is assigned a sub-block, so that each thread accesses its own region, offset in the larger block. According to GPU vendors, this is a desired memory access patterns [2], resulting in better performance. The top down approach has access to two memory blocks, one for each stack, with the traversal stack m having stack frames of size 4. For all the other approaches, the size of a t-stack frame is 16 bytes (see 4.2).

In our implementation, the input values are stored in OpenCLs constant variable scope. Originally we wanted to store the intermediate results (and the traversal stack when needed) in local scope, since this is faster memory, however we realized, that for many large *BlobTrees*, the local memory per thread was not big enough. Given these limitations, all the cases store the stacks in global memory for consistency reasons.

7. Results

We test and compare the performance on a variable sized computer generated test scene. It contains n_l cylinder primitives, all combined with an advanced blend operator[17]. The bounding volume of this object is sampled 32^3 times along a regular grid (representing polygonization), thus requiring that many stacks. Given the tree traversal focus of the paper, we don't count the remaining time for the polygonization algorithm. We use the same resolution for all the models and algorithms so that every test case has the same constant OpenCL scheduling overhead and every performance result is calculated as the average of 32 runs to get rid of outliers due to other system operations.

7.1. Top-Down Traversal

The graphs for our computer generated test scene plot the average time in μs for a single field-value calculation (the tree traversal, and the evaluation of the nodes) for increasing number of leaf nodes. This approach requires the two stacks m and t. Figure 8 shows that the left tree and the right tree have similar performance characteristics, with the balanced case being slightly faster. Given that traversing the left and right heavy tree needs the same amount of memory, and the balanced case

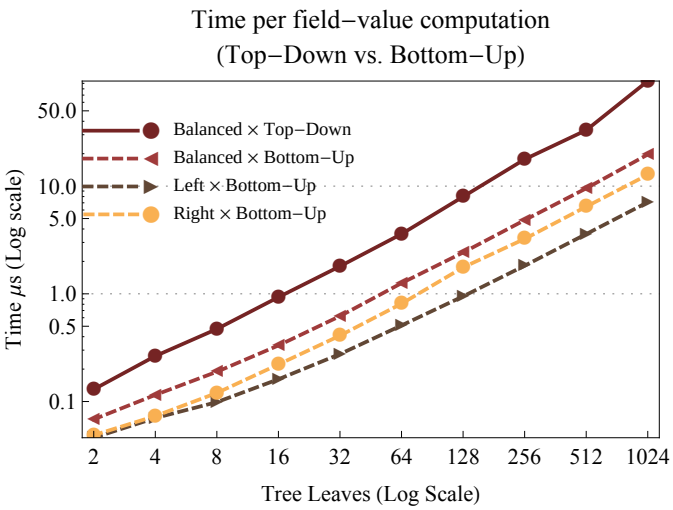


Figure 9: The running times for the linear array-based traversal algorithm, compared to the best tree structure based one as a reference.

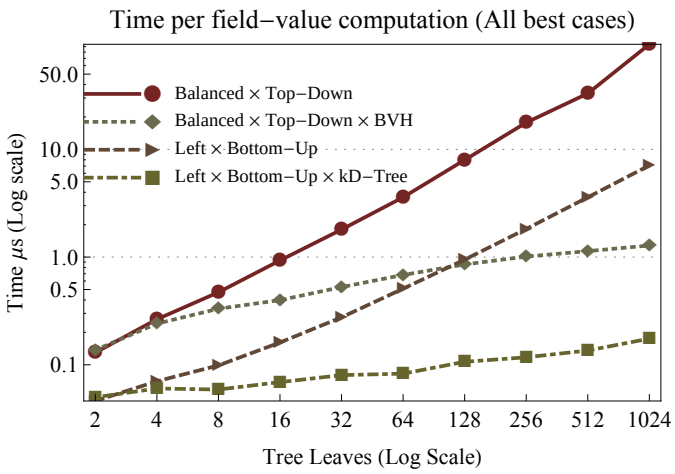


Figure 10: A comparison of the best case running times.

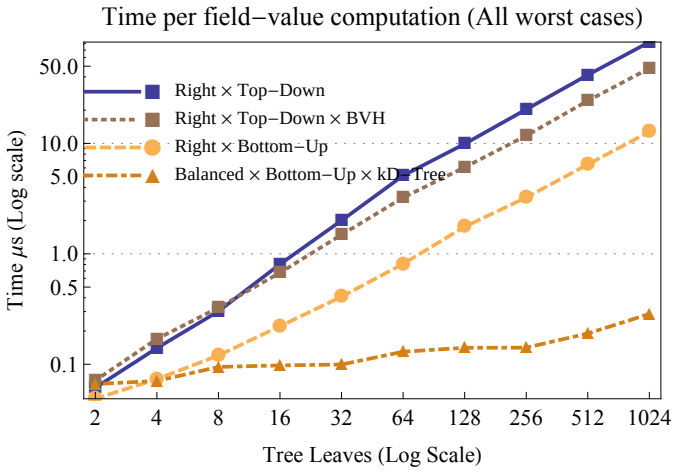


Figure 11: A comparison of the worst case running times.

needing less, the performance characteristics is not a surprise. This means that the balanced tree should be preferred for the top-down traversal.

7.2. *Bottom-Up Traversal*

Traversing the tree bottom-up creates a better memory access pattern, since every thread loads the tree array from start to end, potentially resulting in several memory load multicasts if threads try to access the same node at the same time. This approach does not require a stack m, only the stack t for the temporary results is needed.

Figure 9 includes the best case from figure 8 (balanced - shown as a solid line) for reference, and compares it to the run times for the bottom-up traversal. In all cases bottom-up is faster than top-down. On the other hand, the left-heavy tree case only needs a constant temporary results stack of 2 and shows the best performance of all of them (dotted line with triangle marker). The second fastest case is the right-heavy one, using the largest t stack of the tree. It seems that temporary memory size on this GPU is not that much of an issue, and the downfalls of large memory needs can be compensated by a better memory access pattern. For the right heavy case, the t stack (implemented as an array) is first filled from start to end, then read from end to start, only changing the access direction once. We conclude that one should favour the production of left-heavy trees for efficient traversal, as already shown for CSG by [19]. Figure 9 shows a time difference of one order of magnitude between the best top-down case and the best bottom-up.

7.3. *Acceleration Structures*

Based on work by [14], we investigated the effect that two acceleration structures, Bounding Volume Hierarchy (BVH) [35] and Binary Space Partition (BSP) trees [15], have on the performance of a field-value calculation. In our top-down traversal case, it is very easy to add a BVH to the traversal algorithm, since only a point-in-bounding-volume check has to be added.

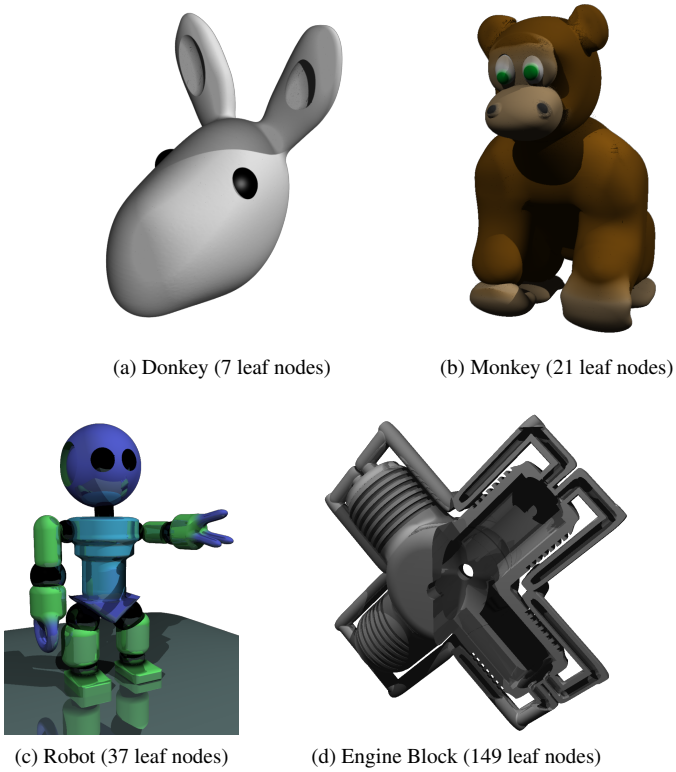
However, the early discard property of a BVH cannot be used efficiently in our bottom-up traversal. An acceleration structure that prunes the *BlobTree* for each space subdivision leaf, is the BSP Tree, which has been used widely to improve the visualization times of mesh scenes, resulting in real-time speed for raytracing on the CPU [4]. We use axis aligned BSP trees (kD trees), for the performance reasoning given by Wald in his thesis, [43]. Since in our case we only need to find the kD node for a given point in space, we do not need to be able to backtrack into neighbouring kD nodes, as demonstrated by [29]. To avoid storing duplicate nodes, we adapted the skip pointers from [39], and created index arrays for each kD leaf, that work on the array storing the full tree data.

Figure 10 compares the four best cases of each algorithm(top-down, top-down plus BVH, bottom-up, bottom-up plus kD), showing two orders of magnitude difference between the worst-best algorithm and the absolute best. Adding an acceleration structure to any of the approaches changes the overall slope of the graph, whereas the pairs of accelerated and unaccelerated curves have approximately a parallel slope, once the node count is higher than 16. For

the worst case the difference in running time is even bigger. Figure 11 shows that the worst case run time for the top-down approach for 1024 leaf nodes is close to 30μs. On the contrary the worst run-time for the bottom-up traversal using a kD-tree is 0.15μs, being two orders of magnitude faster than top-down. Without the acceleration structures, we are still looking at one order of magnitude difference.

We are limited by GPU memory in the size of trees we can explore, but one can assume that the top-down traversal will continue to increase its run time exponentially, whereas our bottom-up version, will grow much slower. We have shown that a top-down traversal of a solid model tree in SPMD is not the most performant. By modifying the traversal algorithm, including many approaches presented for CSG [34], the traversal now works bottom up. This improves the memory access pattern and usage, and decreases the execution time of traversal algorithms significantly. With the top-down traversal, adding a BVH is fairly easy, and improves the algorithms run time, however the trend of the performance still shows an almost exponential slope (linear in a log graph), with increasing leaf nodes. On the contrary the slope of the graph is a lot shallower for the bottom-up and especially the accelerated bottom-up algorithm, showing that it will lead to better performance in most cases.

7.4. Models



(a) Donkey (7 leaf nodes)

(b) Monkey (21 leaf nodes)

(c) Robot (37 leaf nodes)

(d) Engine Block (149 leaf nodes)

Figure 12: Three of the four real world models.

The synthetic test scenes are built so that primitives are distributed equally in space, something very unlikely in real-world situations. While the synthetic models have a constant distribution of primitives in the surrounding bounding box, real world models will have their primitives clustered and distributed arbitrarily. Consequently, four real world models (see ray-traced images in Figure 12) of different sizes were used to investigate the performance of the Bottom-Up algorithm in a more realistic modelling scenario. The models were generated with an interactive modelling system that uses the Bottom-Up approach in the polygonizer. The performance graphs, as explained for the synthetic models, can be found in Figure 13. They show that the kD-tree is actually slightly slower than the basic bottom-up algorithm. One potential reason for this behaviour could be that the median split strategy for the kD-tree is not the best, as proven by [43]. Furthermore, since there is only a limited amount of splits (same maximum recursion as with the computer generated models), and all the objects are clustered at more or less the same point in space, the pruned trees contained in the leaf nodes will not be much more simplified than the original. As a result, more input data storage is needed without benefiting from the acceleration structure. It only adds traversal overhead to the running times.

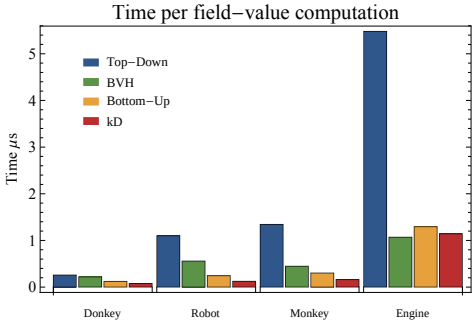


Figure 13: Average run times of a single field-value calculation using the algorithm variations for the four models, in μs.

Most of the trees used in the examples are very close to left-heavy; thus, adding this optimization did not add a significant performance impact to the models, especially the engine. The memory use for the left-heavy variants of the models stayed the same. Overall, the bottom-up traversal shows similar performance improvements with real-world models due to the reduced memory usage. Interestingly, these models don't benefit as much from the kD-tree as the synthetic *BlobTrees*.

These four models were chosen because of different distinct properties:

- The **donkey** model is built from a small number of nodes, however most of them are based on sketched primitives, combined using the summation blend. While the size of the tree is small, the sketched primitives need more time to compute than the cylinder primitives in the synthetic models of the previous section, since they are dependent on the number of control points used to create the sketched field.

- Compared to the donkey, they **monkey** model is built from more sketched primitives, that are combined using

the same Gradient Based Blend used in the synthetic case. There is a higher distribution of primitives in space compared to the donkey, which results in a better improvement of the accelerated test cases, compared to the donkey model.

- The **robot** model uses a wider variety of *BlobTree* primitives and operators, with the primitives having a good distribution in space. As a result, this model can benefit from an acceleration structure similarly as the monkey.

- Lastly, the **engine** model, is the largest example model. It contains four large parts that are copies from each other, just positioned and rotated differently. Each of the four sub-*BlobTrees* are largely left-heavy trees which should show large performance improvements in the bottom-up case. Because these sub-*BlobTrees* are rotated, the axis-oriented bounding boxes, used in the acceleration approaches, occupy larger regions of space, resulting in a non-optimal space subdivision. The performance graph shows that the bottom-up approach is almost as fast as the accelerated approaches. Because of the four similar trees the BVH case works very well at determining an early exit in the top-down traversal, whereas the kD-Tree case cannot improve performance as well.

7.5. Warp Transformations

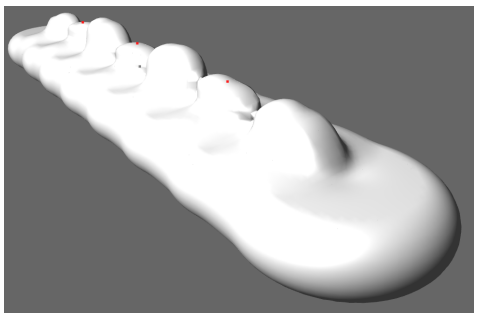


Figure 14: The WarpCurve test scene. Notice the top-most three displaced points in red.

The performance test case for the inclusion of warp transformations in the tree is fairly similar to the synthetic test scene, where the different traversal methods are compared more generally. In this case, only the Top-Down traversal and the Bottom-Up traversal are compared. Both approaches include the separation of the Warp Tree and the *BlobTree*. Comparing them to a non-separated version is not possible, since the OpenCL compiler is not able to properly unroll the required loops. The models to be used for testing are based on the number of leaf nodes (primitives in the tree) but only for the balanced tree case. Above every blend node within this balanced tree, a WarpCurve [41] spanning the underlying model with three control points is inserted, where the middle one is displaced along the surface normal. The larger the number of leaf nodes, the larger the number of interior operator nodes, which is equal to the number of warp curve nodes.

leaf nodes	Top-Down	Bottom-Up
2	1.007	0.821
4	2.027	1.839
8	4.364	3.985
16	8.990	8.018
32	17.259	16.317
64	34.640	32.772
128	70.156	64.634
256	140.729	129.534
512	279.599	260.756
1024	563.750	518.007
2048	2536.030	2360.930

Table 1: The performance numbers in μs, stating the average time for a single field-value calculation in μs for the test cases for *BlobTrees* that include various numbers of WarpCurves.

Given that Disc primitives (slightly less expensive than the previously used Cylinder primitives) are combined using the Gradient Based Blend [17] (see Figure 14), the traversal times/-calculation times for a single field value evaluation are fairly similar to the cases presented above. However, since each WarpCurve has to calculate its values based on three thin plate splines, and there are $n/2$ WarpCurves in the test object, most of the calculation time is spent on the warp calculation. The same WarpCurve calculations are done in both cases, so by looking at the difference in the traversal numbers, there is still a difference between the two traversal methods. As the warps need to be calculated for both cases, and the overhead of the warp calculation overshadows the traversal time in the first place, it can be seen, that the difference between Top-Down and Bottom-Up traversal is still there. However, most of the time is spent on calculating the warp, and not the tree traversal. Thus, the warp transformation case is computation bound, not memory bound as the previous cases were.

8. Conclusion & Future Work

We have presented an accelerated tree traversal for the *BlobTree* using an approach that results in an $O(n)$ traversal time. We have shown how to generate predictable memory access patterns, important for performance on modern SIMD architectures, such as GPUs using OpenCL, or using vector instructions on CPUs. By reinterpreting the *BlobTree* as a mathematical expression and rewriting it in reverse Polish notation, the corresponding bottom-up tree traversal results in a performance improvement of one order of magnitude. We investigated the memory usage of different tree structures, isolated the best performing structure for a given number of *BlobTree* leaf nodes n_l, and shown how to transform an arbitrary *BlobTree* into a representation from which the best performance can be achieved. Future work includes overcoming the memory limitations employed by current generation GPUs, by further reducing the need for intermediate storage. This could require reordering the tree based on the operators, to minimize temporary data stack push and pop operations. As a result we should be able to make

use of the smaller, but faster local memory found in current GPUs.

Acknowledgements

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada, the GRAND NCE, Intel Inc., and nVidia Inc.

References

[1] Akkouche, S., Galin, E., 2001. Adaptive Implicit Surface Polygonization Using Marching Triangles. Computer Graphics Forum 20 (2), 67–80.

[2] AMD, Dec. 2011. OpenCL Programming Guide. Advanced Micro Devices, Inc., 1st Edition.

[3] Barthe, L., Wyvill, B., de Groot, E., Dec. 2004. Controllable binary csg operators for soft objects. International Journal of Shape Modeling.

[4] Benthin, C., 2006. Realtime ray tracing on current CPU architectures. Ph.D. thesis, Saarländische Universitäts- und Landesbibliothek, Postfach 151141, 66041 Saarbrücken.

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

[6] Bloomenthal, J., 1994. An implicit surface polygonizer. In: Heckbert, P. S. (Ed.), Graphics Gems IV. Academic Press Professional, Inc., San Diego, CA, USA, pp. 324–349.

[7] Bloomenthal, J., 1997. Introduction to Implicit surfaces. Morgan Kaufmann.

[8] Bunnell, M., Apr. 2005. Dynamic Ambient Occlusion and Indirect Lighting. In: Pharr, M., Fernando, R. (Eds.), GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional.

[9] Darema, F., George, D. A., Norton, V. A., Pfister, G. F., 1988. A single-program-multiple-data computational model for EPEX/FORTRAN. Parallel Computing 7 (1), 11–24.

[10] de Groot, E., 2008. BlobTree Modelling. Ph.D. thesis, The University of Calgary, University of Calgary.

[11] Elber, G., Dec. 2005. Generalized filleting and blending operations toward functional and decorative applications. Graphical Models 67 (3), 189–203.

[12] Fatahalian, K., Houston, M., 2008. A Closer Look at GPUs. Communications of the ACM 51 (10).

[13] Flajolet, P., Raoult, J. C., Vuillemin, J., Jul. 1979. The number of registers required for evaluating arithmetic expressions. Theoretical Computer Science 9 (1), 99–125.

[14] Fox, M., Galbraith, C., Wyvill, B., May 2001. Efficient Implementation of the BlobTree for Rendering Purposes. In: Shape Modeling and Applications, SMI 2001 International Conference on. IEEE, pp. 306–314.

[15] Fuchs, H., Kedem, Z. M., Naylor, B. F., Jul. 1980. On Visible Surface Generation by A Priori Tree Structures. SIGGRAPH '80 Proceedings of the 7th annual conference on Computer graphics and interactive techniques 14, 124–133.

[16] Goldfarb, M., Jo, Y., Kulkarni, M., 2013. General Transformations for GPU Execution of Tree Traversals. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. ACM, New York, NY, USA, pp. 10:1–10:12.

[17] Gourmel, O., Barthe, L., Cani, M.-P., Wyvill, B., Bernhardt, A., Grasberger, H., 2013. A Gradient-Based Implicit Blend. Transactions on Graphics (SIGGRAPH 2013) 32 (2).

[18] Gourmel, O., Pajot, A., Paulin, M., Barthe, L., Poulin, P., 2010. Fitted BVH for Fast Raytracing of Metaballs. Computer Graphics Forum 29 (2), 281–288.

[19] Hable, J., Rossignac, J., 2005. Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes. In: ACM SIGGRAPH 2005 Papers. ACM, New York, NY, USA, pp. 1024–1031.

[20] Hable, J., Rossignac, J., 2007. CST: Constructive Solid Trimming for Rendering BReps and CSG. IEEE Transactions on Visualization and Computer Graphics 13 (5), 1004–1014.

[21] Hanniel, I., Haller, K., Sep. 2011. Direct Rendering of Solid CAD Models on the GPU. In: 2011 12th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics). IEEE, pp. 25–32.

[22] Jansen, F. W., 1991. Depth-order point classification techniques for CSG display algorithms. ACM Trans. Graph. 10 (1), 40–70.

[23] Kalra, D., Barr, A., Jul. 1989. Guaranteed ray intersections with implicit surfaces. SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques.

[24] Knoll, A., Hijazi, Y., Kensler, A., Schott, M., Hansen, C., Hagen, H., 2009. Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. Computer Graphics Forum 28 (1), 26–40.

[25] Mäntylä, M., 1987. An introduction to solid modeling. Computer Science Press, Inc., New York, NY, USA.

[26] Munshi, A., Jun. 2011. The OpenCL Specification. Khronos.

[27] Nickolls, J., Buck, I., Garland, M., Skadron, K., 2008. Scalable Parallel Programming with CUDA. Queue 6 (2), 40–53.

[28] Pharr, M., Mark, W. R., May 2012. ispc: A SPMD Compiler for High-Performance CPU Programming. In: Innovative Parallel Computing Conference. pp. 1–13.

[29] Popov, S., Günther, J., Seidel, H.-P., Slusallek, P., 2007. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. Computer Graphics Forum 26 (3), 415–424.

[30] Reiner, T., Mückl, G., Dachsbacher, C., Jun. 2011. Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions. Computers and Graphics 35 (3), 596–603.

[31] Ricci, A., 1973. A constructive geometry for computer graphics. The Computer Journal 16 (2), 157–160.

[32] Romeiro, F., Velho, L., De Figueiredo, L., Oct. 2006. Hardware-assisted Rendering of CSG Models. In: 2011 12th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics). IEEE, pp. 139–146.

[33] Rossignac, J., 1999. Blist: A Boolean List Formulation of CSG Trees. Tech. Rep. GIT-GVU-99-04.

[34] Rossignac, J., Sep. 2012. Ordered Boolean List (OBL): Reducing the Footprint for Evaluating Boolean Expressions. IEEE Transactions on Visualization and Computer Graphics 17 (9), 1337–1351.

[35] Rubin, S. M., Whitted, T., 1980. A 3-dimensional representation for fast rendering of complex scenes. In: Proceedings of the 7th annual conference on Computer graphics and interactive techniques. ACM, New York, NY, USA, pp. 110–116.

[36] Schmidt, R., Wyvill, B., Galin, E., 2005. Interactive implicit modeling with hierarchical spatial caching. SMI '05: Proceedings of the International Conference on Shape Modeling and Applications 2005, 104–113.

[37] Shirazian, P., Wyvill, B., Duprat, J.-L., 2012. Polygonization of implicit surfaces on Multi-Core Architectures with SIMD instructions. In: Childs, H., Kuhlen, T. (Eds.), Eurographics Symposium on Parallel Graphics and Visualization (EGPGV). pp. 89–98.

[38] Shirley, P., Marschner, S., 2009. Fundamentals of Computer Graphics. A. K. Peters, Ltd., Natick, MA, USA.

[39] Smits, B., 2005. Efficiency issues for ray tracing. In: ACM SIGGRAPH 2005 Courses. ACM, New York, NY, USA.

[40] Snyder, J., Jul. 1992. Interval Analysis for Computer Graphics. SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques, 121–130.

[41] Sugihara, M., Wyvill, B., Schmidt, R., Jun. 2010. WarpCurves: A tool for explicit manipulation of implicit surfaces. Computers and Graphics 34 (3).

[42] Vaillant, R., Guennebaud, G. a. e., Barthe, L. i. c., Wyvill, B., Cani, M.-P., 2014. Robust Iso-surface Tracking for Interactive Character Skinning. ACM Trans. Graph. 33 (6), 189:1–189:11.

[43] Wald, I., 2004. Realtime Ray Tracing and Interactive Global Illumination. Ph.D. thesis, Computer Graphics Group, Saarland University.

[44] Whited, B., Rossignac, J., May 2009. Relative blending. Computer-Aided Design 41 (6), 456–462.

[45] Wyk, C. J. V., 1991. Data Structures and C Programs, 2nd Ed., 2nd Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[46] Wyvill, B., Guy, A., Galin, E., 1999. Extending the CSG tree. Warping, blending and Boolean operations in an implicit surface modeling system. Computer Graphics Forum 18 (2), 149–158.

[47] Wyvill, G., McPheeters, C., Wyvill, B., Feb. 1986. Data structure for soft objects. The Visual Computer 2 (4), 227–234.