New Scalable SIMD-Based Ray Caster Implementation for Virtual Machining

Alexander Leutgeb, Torsten Welsch$^{(\boxtimes)}$, and Michael Hava

RISC Software GmbH, Softwarepark 35, 4232 Hagenberg, Austria
{alexander.leutgeb,torsten.welsch,michael.hava}@risc-software.at

Abstract. We present a highly efficient ray casting system for the visualization of subtractive manufacturing, combining state-of-the-art results of various active research fields. Besides popular techniques like acceleration structures, coherent traversal and frustum culling, we integrated the novel surface cell evaluation (SCE) algorithm, allowing the elimination of surfaces that have no effect on the final workpiece's shape. Thus, our ray caster allows an interactive, non-approximate visualization of thousands of Boolean subtraction operations between a stock and arbitrary triangular swept volumes. Compared to image-space based approaches for virtual machining, such as *z-maps* [3], *dexels* [4] or *layered depth images* [10], our scalable SIMD-based implementation offers a higher rendering performance as well as a view-independent workpiece modeling. Hence, it is perfectly suited for of both simulation and verification in computer-aided manufacturing (CAM) applications.

Keywords: Ray casting · SIMD · CAM · Boolean subtraction operations · Subtractive manufacturing · Space partitioning strategy · Multi-axis milling

1 Introduction and Related Work

With CAM being extensively applied in industry nowadays, simulations of subtractive manufacturing - material removal processes, such as milling, turning or drilling - have become complex tasks, requiring efficient systems for their computation and visualization. In our development scenario, for example, complex multi-axis milling processes, that result in workpieces with a high-quality surface finish, consist of thousands of material removal steps. Hence, the manufacturing industry has a strong need for both real-time and offline visualization as well as verification during the whole machining process. Especially in terms of verification, not only a continuous but an interactive simulation is essential. To develop a system that is capable of meeting the outlined requirements, one has to ensure that state-of-the-art research results of various fields are combined in an efficient manner, lending itself to both the available infrastructure and the application scenario.

With that respect, our implementation is related to: the STAR about ray tracing by Wald et al. [9], the coherent grid traversal by Wald et al. [8], the SIMD frustum culling by Dmitriev et al. [2], the fast ray/triangle intersection test by Moeller and Trumbore [5] and the very efficient SCE algorithm presented in this paper. As the generation of swept volumes is not the concern of this paper, we assume that all swept volumes mentioned in the following have triangulated, closed ("watertight") and well oriented surfaces.

2 Basic Idea

In the context of material removal simulations, the workpiece is visualized while a machine tool virtually cuts layer-by-layer into it. Along its path, the tool's movement is discretized and combined to swept volumes that are further applied to the original stock geometry as Boolean subtraction operations.

Roth [7] firstly presented ray casting as an algorithm for rendering Boolean operations between volumes. To apply this algorithm to our subtractive manufacturing scenario, an intersection counter i_r has to be added per ray r, recording whether r enters or leaves a swept volume along its way through the virtual three-dimensional (3D) space. After all intersections between the rays and the volumes have been determined, we sort the intersections per r in ascending order, depending on their distance to r's origin. Given an arbitrary r and the $k+1$-th intersection along its way, we update i_r according to the following rule:

$$\begin{aligned} i_r^0 &:= -1 \\ i_r^{k+1} &:= \begin{cases} i_r^k - 1 \text{ , } r \text{ enters a swept volume} \\ i_r^k + 1 \text{ , } r \text{ leaves a swept volume.} \end{cases} \end{aligned} \tag{1}$$

Note that we model the stock geometry as a subtraction from the completely solid 3D space as well, enabling us to treat all inserted volumes equally. Thus, each i_r has to be initialized with -1 so that the final surface hit of r is found iff $i_r = 0$ holds. In this case, r does not lie inside a swept volume any more, as all entered swept volumes have been left.

3 High-Level Optimizations

3.1 Acceleration Structure

Considering only volumes with triangulated surfaces, we accelerate the original boolean ray casting algorithm by reducing the number of triangles per ray that have to be tested for intersections. This is done by subdividing the 3D scene into multiple cells that refer to triangles lying inside them. Hence, a ray has to test only those nearby triangles for an intersection that lie inside cells the ray traverses. With respect to the requirements of virtual machining, consisting of thousands of successive Boolean subtraction operations, we need a space partitioning strategy that allows us to apply new swept volumes to the workpiece

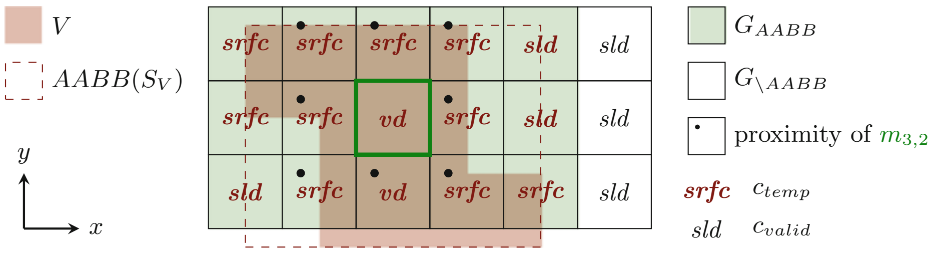

Fig. 1. Illustration of the temporary classification algorithm.

rapidly at low computational cost. As regular grids are fastest to build [9], we implement a coarse regular grid whose cubical cells we call *macrocells*.

Additionally, we add a second regular grid level per macrocell that consists of so-called *microcells*, supporting the elimination of triangles in macrocells that are enclosed by the union of two or more swept volumes, see Sect. 3.3. The rays' grid traversal, however, is not affected by this extension as its computation takes place on the macrocell level only, see Sect. 3.5.

3.2 Cell Classification

As the correct evaluation of the rays' intersection counters (1) depends on "watertight" swept volumes, we have to keep even those triangles available for intersection tests that have no effect on the final workpiece's shape. To deal with this performance issue, we first have to introduce a grid cell classification: A grid cell can lie completely inside (*void*) or outside (*solid*) of a swept volume or it can contain triangles of this volume (*surface*). For simplicity's sake, we describe the following algorithms in the 2D space only.

Given the grid's microcells, $G = \{\text{microcells } m_{x,y} \mid 1 \leq x \leq k \ \wedge \ 1 \leq y \leq l\}$, with k = the number of microcells in x- and l = the number of microcells in y-direction. Given further an arbitrary, triangulated swept volume V that has to be inserted into that grid. The swept volume's triangles, $S_V = \{\text{triangles } t \mid t \text{ belongs to } V\}$, are mapped into the grid by initially computing the axis-aligned bounding box (AABB) of S_V. Then, we determine all microcells that are affected by the insertion of V, $G_{AABB} = \{\text{microcells } m_{x,y} \mid m_{x,y} \in G \wedge m_{x,y} \cap AABB(S_V) \neq \emptyset\}$. Finally, the G_{AABB} store references to all $t \in S_V$ that (partially) lie inside them, if any.

Depending on the 8-connected proximity relationship between the $m_{x,y} \in G$ and on the microcells that are not affected by the insertion of V, $G_{\setminus AABB} = G \setminus G_{AABB}$, the grid cells are classified as described in the following: Starting with the $m_{x,y} \in G_{AABB}$, we determine their individual temporary classification, $c_{temp} \in \{void, solid, surface\}$ (or *vd*, *sld* and *srfc*) by labeling all $m_{x,y}$ that store $t \in S_V$ as *surface*. Afterwards, if a $m_{x,y}$ has a neighbor $n \in G$ of classification type either *void* or *solid*, it gets the same label. Otherwise, if either $m_{x,y}$ has no neighbors or the label *surface* is the the only classification available in $m_{x,y}$'s proximity, we take one $t \in S_V$ and cast a ray r_{mt} from the center of $m_{x,y}$ toward that t. Given the intersections with other $t \in S_V$ on its way, if any, we use the one nearest to the origin of r_{mt} to determine the temporary classification c_{temp} of $m_{x,y}$: If r_{mt} and the normal of that t point into the same half-space of t's plane, $m_{x,y}$ lies inside V (*void*), otherwise outside (*solid*). Note that we define all triangle normals of swept volumes as outward-pointing. For further illustration, see Fig. 1.

Given a microcell's c_{temp} and its original valid classification c_{valid}, the updated valid classification for that microcell, c^+_{valid}, is given by (2). Note that all microcells of the grid are initially labeled with $c_{valid} = solid$ as the grid is considered to be completely solid before applying the first swept volume onto it.

$$c^+_{valid} := \begin{cases} sld & , c_{temp} = sld \wedge c_{valid} = sld \\ vd & , c_{temp} = vd \vee c_{valid} = vd \\ srfc & , else \end{cases} \tag{2}$$

After c^+_{valid} has been determined for all $m_{x,y} \in G_{AABB}$, the classification of their corresponding macrocells is assigned in the following way: Every macrocell containing at least one $m_{x,y}$ of the type *surface* is labeled as *surface* as well and stores references to all $t \in S_V$ inside its $m_{x,y}$. Otherwise, as one can easily see, a macrocell can contain only *void* or *solid* microcells exclusively and therefore gets the same attribute as its uniform microcells.

3.3 Triangle Elimination

Given both the new valid macro- and microcell classification for all cells that are affected by the insertion of a swept volume, all triangles inside *void* macrocells can be dismissed as they reside completely inside swept volumes and therefore have no effect on the current workpiece's shape. With this elimination, we get a first grid level that holds triangles in *surface* macrocells only, significantly reducing the number of triangles per ray that have to be tested for intersections during the ray casting, see Fig. 6/2–3 and Sect. 5. As we handle *surface* macrocells isolatedly during the ray's grid traversal, there is one fundamental condition for the ray casting of Boolean subtraction operations that has to hold if we want to eliminate triangles that way: The ray caster must be able to determine if the intersection point between an entering ray and a *surface* macrocell lies inside or outside of swept volumes with triangles stored in that cell. Otherwise, as this triangle removal results in non-watertight swept volumes, there is no way to evaluate the ray's intersection counter (1) correctly.

3.4 Surface Cell Evaluation

The basic idea is to determine if the point p_{rM} where a ray r enters a *surface* macrocell $M_{x,y}$ lies inside or outside of swept volumes V with triangles $t \in S_V$ stored in $M_{x,y}$. Given such an r that enters an $M_{x,y}$, we start with the computation of all intersection points p_{rt} between r and all triangles stored in

$M_{x,y}$. If r intersects with at least one triangle per V, we can determine whether r enters or leaves these V by comparing r with the normals of those intersected $t \in S_V$ with the shortest distance to r's origin, see Sect. 3.2.

A problem occurs when there is a V with $t \in S_V$ stored in $M_{x,y}$ not intersecting with the ray r: To solve this, we chose an arbitrary triangle $t \in S_V$ stored in $M_{x,y}$ and determine a reference point p_V that is guaranteed to lie on t and inside $M_{x,y}$. Given p_V, we create a so-called positioning ray r_{pos} from p_{rM} toward p_V. As r_{pos} is guaranteed to intersect with t at least, we can use it to determine if p_{rM} lies inside or outside of V, again by comparing r_{pos} with the normal of the intersected $t \in S_V$ nearest to p_{rM}, see Sect. 3.2.

After thus proceeding with all V with $t \in S_V$ stored in $M_{x,y}$, we know the relative location of r in p_{rM} in relation to all these V, allowing us to preset the initial intersection counter i_r^0 as follows: This time starting from $i_r^0 := 0$, i_r^0 is decreased by one for each V with p_{rM} inside, as r must have entered these V anywhere before the current $M_{x,y}$ during its grid traversal. For further illustration, see Fig. 2.

Given the initial i_r^0, we sort the determined intersections between r and the V with $t \in S_V$ stored in $M_{x,y}$ in ascending order, depending on their distance to r's origin. Starting with the first intersection, we update i_r according to (1) while checking these intersections one by one. Finally, at the point k where $i_r^k = 0$ holds, we have found the final surface hit of r. Otherwise, if no such k exists in the current $M_{x,y}$, we continue with the SCE algorithm in the next *surface* macrocell along the ray's way through the grid.

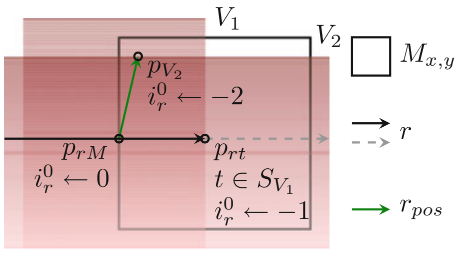

Fig. 2. Illustration of the intersection counter presetting algorithm.

3.5 Coherent Grid Traversal

Given the standard 3D-DDA traversal algorithm [1] for a single ray, we have two seemingly opposing problems to solve: Firstly, we want the grid's macrocell level to be as fine as possible to decrease the number of triangles that have to be tested per ray. Secondly, we want the grid to be as coarse as possible to reduce the time spent on grid traversal. To solve both problems at once, we join nearby rays into disjunct ray packets similarly to Wald et al. [8]: As the four corner rays of a ray packet define a frustum, we can incrementally compute the overlap of that frustum with the grid's macrocells, resulting in a slice-wise traversal of the grid along the packet's major traversal axis. In combination with our SCE algorithm in Sect. 3.4, we test a complete ray packet against all swept volumes' triangles that are stored in *surface* macrocells of that slice. Hence, we are able to utilize the SIMD paradigm, see Sect. 4.4: We load the *surface* macrocell's triangles only once and compute the intersection tests for all rays inside the packet in parallel afterwards. If all the packet's rays either leave the grid or hit the final workpiece's surface after all *surface* macrocells of the current slice have been tested, we stop the ray casting for that packet. Otherwise, the next slice along the major traversal axis has to be determined and investigated in the same way. Furthermore, we can apply the well-known frustum culling on these triangles, see Sect. 4.2. Thus, we can reduce the number of triangles to be tested against a ray packet without refining the macrocell level of the grid, preserving the fast packets' macrocell traversal.

4 Low-Level Optimizations

4.1 Single Instruction, Multiple Data (SIMD)

Intel introduced MMX in 1997, enabling CPUs to simultaneously perform the same operations on multiple data. Nowadays, the advanced vector extensions (AVX) extend both the original MMX instructions and register width (AVX: 256 bit). For performance reasons, we utilize SIMD explicitly with C/C++ intrinsics instead of letting the compiler decide where to use SIMD, for example:

```
#include <immintrin.h>

#define VECTOR3F_SUB(operand0, operand1, result) do { \
  result.x = _mm256_sub_ps(operand0.x, operand1.x); \
  result.y = _mm256_sub_ps(operand0.y, operand1.y); \
  result.z = _mm256_sub_ps(operand0.z, operand1.z); \
} while (0)

struct Vector3f {__m256 x; __m256 y; __m256 z;};
```

We arrange our data so that it scales with the registers' width, meaning that eight 3D vectors are stored vertically in three registers. In our implementation, we use this layout triangle-parallel during the frustum culling, see Sect. 4.2, and ray-parallel while computing the ray/triangle intersection tests, see Sect. 4.4.

4.2 SIMD Frustum Culling

Given a *surface* macrocell that is traversed by a ray packet and the packet's four corner rays defining its frustum, we compute the frustum culling for triangles stored in that macrocell similarly to the SIMD-based approach by Dmitriev et al. [2]: We use a triangle's barycentric coordinates to determine whether it lies inside the packet's frustum or not, reducing the number of ray/triangle intersection tests in *surface* macrocells that are not completely overlapped by the frustum. Note that we apply frustum culling to all swept volumes' triangles S_V stored in a traversed *surface* macrocell $M_{x,y}$ before testing the triangles $t \in S_V$ against any ray r of a ray packet P. As our SCE algorithm may need a valid reference point p_V per swept volume V to determine whether p_{rM}, the point where r enters $M_{x,y}$, lies inside or outside of these V, see Sect. 3.4, we have to distinguish between three cases here:

```
**if** $p_V$ lies outside of $P$'s frustum (a $t \in S_V$ between $p_{rM}$ and $p_V$ may have been culled)
    **if** $\exists t \in S_V : t$ intersects with $P$'s frustum **then**
        use $t$ to determine a new $p_V$ inside both $M_{x,y}$ and $P$'s frustum
    **else** use $p_V$ and all $t \in S_V$ inside $M_{x,y}$ to determine whether $P$ lies inside $V$ or
        not, and if so, skip all remaining intersections tests with other $V$ inside $M_{x,y}$
**else** $p_V$ is valid as it lies inside the $P$'s frustum
```

4.3 Ray Packet Repacking

As mentioned by Wald et al. [8], the slice-wise grid traversal yields the disadvantage of testing rays for intersections with triangles stored in *surface* macrocells they would never have traversed in the non-packet case. Hence, we repack the rays of a packet after the frustum culling so that only rays that really intersect with the current *surface* macrocell have to be checked for triangle intersections, resulting in a reduced number of tests while preserving SIMD compatibility.

Additionally, we record the intersecting rays per swept volume inside the current *surface* macrocell. Before using positioning rays to determine the relative location for rays that have not intersected with one or more swept volumes, see Sect. 3.4, we repack the packet's rays - with respect to the swept volumes they have missed - a second time: Given a swept volume and a list of rays that have no intersection with this volume, we have to build positioning rays to the valid reference point of that swept volume only for rays on the list. Again, we get rid of some dispensable work while supporting the simultaneously computation of the same operations on all repacked rays.

4.4 SIMD Intersection Test

Considering its good parallelization capability, we decided to adapt the algorithm presented by Moeller and Trumbore [5] to the SIMD paradigm. In their algorithm, the triangle is translated to the origin and transformed to a unity triangle in y and z, where the rays are aligned with x. In this state, the test whether the rays intersect with the triangle or not is easy to compute, see Fig. 3.

5 Results

The experimental results presented in this section are based on the following hardware configuration: Intel Core i5–2400 CPU (four cores @ 3.1 GHz, four threads, Turbo disabled, 198 GFlops) and 16 GB DDR-1333 RAM.

Using a ray packet size of eight times eight rays, 100 macrocells in the longest grid dimension and zero microcells in one macrocell dimension, (100,0), our implementation is tested against a simple scene first, see Fig. 5/1. Although being no reproduction, this scene is used to demonstrate our performance speedup compared to the recently published approximative image-space based modeling approach by Zhao et al. [11] that uses layered depth images. By adding the times for mapping, classification and rendering in Table 1, one can see that our implementation outperforms [11] by factor 10.4 with respect to [#triangles/(second · GFlops)] ([11]: 525 k triangles, 585 ms, Nvidia GeForce GTX 480, 1344 GFlops).

```
__forceinline void determineRaysTriangleIntersection(__m256& mask, const
     Vector3f& origin, const Vector3f& directions, const Vector3f& vertex0,
     const Vector3f& edge1, const Vector3f& edge2, const Vector3f& normal)
{
  Vector3f  tVec, qVec, pVec, uvw;
  __m256 det;
  mask = c_false;

  VECTOR3F_SUB(origins, vertex0, tVec);
  VECTOR3F_CP(directions, edge2, pVec);
  VECTOR3F_DP(edge1, pVec, det);
  VECTOR3F_DP(tVec, pVec, uvw.x);
  if ((MOVEMASK(uvw.x) ^ MOVEMASK(det)) == 255) return;

  VECTOR3F_CP(tVec, edge1, qVec);
  VECTOR3F_DP(directions, qVec, uvw.y);
  if ((MOVEMASK(uvw.y) ^ MOVEMASK(det)) == 255) return;

  uvw.z = SUB(SUB(det, uvw.x), uvw.y);
  if ((MOVEMASK(uvw.z) ^ MOVEMASK(det)) == 255) return;

  mask = XOR(c_true, XOR(CMPLT(uvw.x, c_zero), CMPLT(det, c_zero)));
  mask = ANDNOT(XOR(CMPLT(uvw.y, c_zero), CMPLT(det, c_zero)), mask);
  mask = ANDNOT(XOR(CMPLT(uvw.z, c_zero), CMPLT(det, c_zero)), mask);
}
```

Fig. 3. SIMD intersection test between eight rays (`origin`, `directions`) and one triangle (`vertex0`, `edge1`, `edge2`, `normal`). A valid intersection for the ray at position `index` is found iff `int (mask.m256_f32[index]) != 0` holds.

To compare our implementation with an object-space based modeling approach as well, we created Fig. 5/2 similar to the scene Romeiro et al. [6] used in their paper. With a macro-/microcell configuration of (100,3), our overall performance - the sum of the times for mapping, classification and rendering in Table 1 - is faster by a factor of 9.2 compared to their kd-tree-based ray caster that is limited to convex primitives only; with respect to [#rays/(second · GFlops)] ([6]: 640×480 rays, 2314 ms, Nvidia GeForce 6800 GT, 67 GFlops).

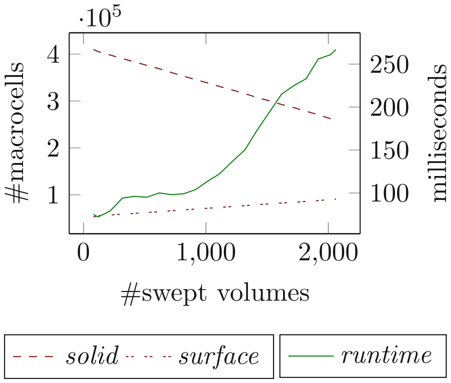

Fig. 4. Sequence of the subtractive manufacturing process of Fig. 6/4.

The last scene - as an example of complex subtractive manufacturing - is an impeller that is machined by swept volumes generated from 5-axis milling with a macro-/microcell configuration of (125,3), see Fig. 6/4. Compared to an implementation with an accelerating first grid level but without classification and elimination (68.8m grid triangles), our SCE-based approach that adds both macrocell classification and elimination reduces the number of triangles inside the grid's macrocells by 90 % (6.4 m grid triangles), resulting in a performance boost of factor 10.2. With the second

Table 1. Overview over the benchmark scenes' results. LTR: figure reference, number of used CPU cores, total number of *surface* macrocells, total number of *solid* macrocells, total number of *void* macrocells, total number of triangle references inside *surface* macrocells, used memory in [MB], average time for mapping per swept volume in [ms], average time for classification per swept volume in [ms], ray casting rendering time for 0.25 M pixels in [ms], ray casting rendering time for 1.00 M pixels in [ms], average speedup with respect to the number of used CPU cores.

fgr	#crs	#srfc	#sld	#vd	#ctrngls	mmry	mppng	clssfctn	0.25 M	1.00 M	spdp
5/1	1	4,046	7,464	374,990	7,529,401	2,359	8,019	392	1,341	2,493	-
	2						6,474	235	687	1,271	1.3
	4						5,515	157	356	657	1.7
5/2	1	63,875	22,984	413,141	1,620,027	1,283	22.6	2.9	406	978	-
	2						16.0	1.9	207	501	2.0
	4						13.8	3.0	107	259	3.6
6/4	1	90,812	259,157	449,906	4,222,313	2,794	78.9	29.6	1,043	2,554	-
	2						47.4	17.7	531	1,297	1.9
	4						29.0	10.8	270	663	3.8

grid level's microcells, the total number of triangles is reduced by another 34 % (4.2 m grid triangles), additionally increasing the rendering performance by one third. Figure 4 illustrates the generation sequence of Fig. 6/4, showing the total number of *surface* and *solid* macrocells and the rendering runtime (0.25 M pixels) of our implementation on four CPU cores, see also Table 1. The last column in Table 1 shows our implementation's scalability for all scenes: In our target scenario (Fig. 6), it is nearly linear. Furthermore, note that for a memory-bound algorithm, a higher resolution yields a higher spatial coherence of the rays which leads to an increase of cache coherent operations. Hence, the runtime of our SCE-based ray caster implementation scales sublinear with higher resolutions.

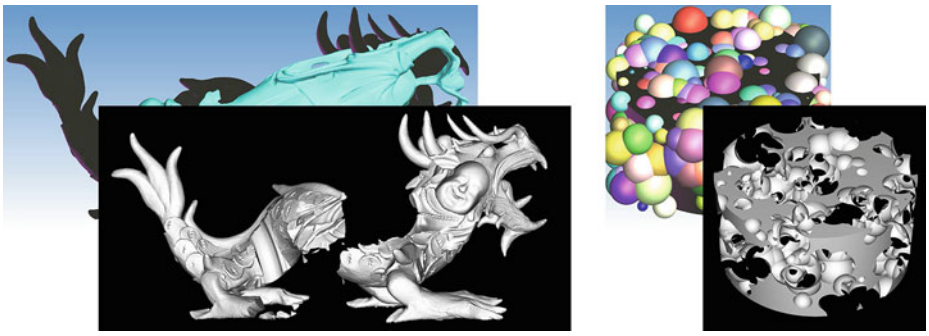

Fig. 5. LTR: *Asian Dragon* minus *Happy Buddha* (8,306,513 triangles) and cylinder minus 1,000 random spheres (5,177,870 triangles), both original volumes and ray casted Boolean subtraction result.

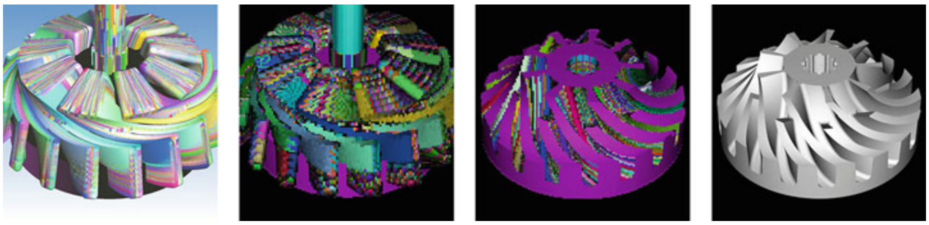

Fig. 6. Scene details: 2,062 volumes, 12,553,756 triangles. LTR: original stock and swept volumes, grid's macrocells containing all triangles (no classification) and remaining *surface* macrocells (with classification), ray casted completely machined impeller.

Acknowledgments. The models *Asian Dragon* and *Happy Buddha* were taken from *The Stanford 3D Scanning Repository*. Furthermore, this research was funded within the scope of the program *Regionale Wettbewerbsfähigkeit OÖ 2007–2013* by the *European Regional Development Fund* and the state *Upper Austria* .

References

1. Amanatides, J., Woo, A.: A fast voxel traversal algorithm for ray tracing. In: EUROGRAPHICS Proceedings, pp. 3–10 (1987)
2. Dmitriev, K., Havran, V., Seidel, H.P.: Faster ray tracing with SIMD shaft culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany (2004)
3. Goldfeather, J., Hultquist, J.P.M., Fuchs, H.: Fast constructive solid geometry display in the pixel-powers graphics system. In: ACM SIGGRAPH Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, pp. 107–116 (1986)
4. Hook, T.V.: Real-time shaded NC milling display. In: ACM SIGGRAPH Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, pp. 15–20 (1986)
5. Möller, T., Trumbore, B.: Fast, minimum storage ray/triangle intersection. J. Graph. Tools **2**(1), 21–28 (1997)
6. Romeiro, F., Velho, L., de Figueiredo, L.H.: Hardware-assisted rendering of CSG models. In: SIBGRAPI Proceedings of the 19th Brazilian Symposium on Computer Graphics and Image Processing, pp. 139–146 (2006)
7. Roth, S.D.: Ray casting for modeling solids. J. Comput. Graph. Image Proc. **18**(2), 109–144 (1982)
8. Wald, I., Ize, T., Kensler, A., Knoll, A., Parker, S.G.: Ray tracing animated scenes using coherent grid traversal. ACM J. Trans. Graph. **25**(3), 485–493 (2006)
9. Wald, I., Mark, W.R., Günther, J., Boulos, S., Ize, T., Hunt, W., Parker, S.G., Shirley, P.: State of the art in ray tracing animated scenes. Comput. Graph. Forum J. **28**(6), 1691–1722 (2009)
10. Wang, C.C.L., Leung, Y.S., Chen, Y.: Solid modeling of polyhedral objects by layered depth-normal images on the GPU. Int. J. Comput.-Aided Des. **42**(6), 535–544 (2010)
11. Zhao, H., Wang, C.C.L., Chen, Y., Jin, X.: Parallel and efficient boolean on polygonal solids. Int. J. Comput. Graph. **27**(6–8), 507–517 (2011)