Single-Pass GPU Solid Voxelization for Real-Time Applications

Elmar Eisemann*
ARTIS-INRIA / Grenoble University

Xavier Décoret†
Phoenix Interactive

Abstract

In this paper, we present a single-pass technique to voxelize the *interior* of watertight 3D models with high resolution grids in real-time during a single rendering pass. Further, we develop a filtering algorithm to build a density estimate that allows the deduction of normals from the voxelized model. This is achieved via a dense packing of information using bitwise arithmetic. We demonstrate the versatility of the method by presenting several applications like translucency effects, CSG operations, interaction for particle simulations, and morphological operations. The speed of our method opens up the road for previously impossible approaches in real-time: 300,000 polygons are voxelized into a grid of one billion voxels at $> 90Hz$ with a recent graphics card. **This is the authors' version of the paper. The definitive version has been published in the Proceedings of Graphics Interface 2008.**

Keywords: GPU, solid voxelization, real-time, applications

1 Introduction

Geometric complexity in computer scenes is constantly increasing. As a consequence, rendering is becoming more expensive, but is counterbalanced by the advancements of graphics hardware. Nevertheless, many tasks go beyond pure display. Interaction becomes more expensive when the number of primitives increases. This makes alternative representations important, one of which are voxels.

The popularity of voxels comes from their simplicity, regularity, and from general advantages of volumetric representations [13, 17]. A variety of fields exploit voxel representations including effects like shadows [21], CSG operations [12], visibility queries [32], and collision detection [25, 15].

For a long time, voxelization has been a costly task often performed in a preprocess. A model is placed in a volumetric grid and approximated by storing information representing the geometry in each grid entry. The particular binary voxelization only uses a boolean (indicating the presence of matter).

Recent binary GPU voxelization algorithms [8, 10] provide better performance, but solely derive a boundary voxelization. For polygons at a grazing angle, this introduces holes, which imposes several passes [8] and a reduced resolution or conservative boundary voxelization [38]. We present a fast method that delivers a solid voxelization where voxels are marked if they lie in the interior of the model. This is important in many contexts like simulations, path finding routines, or visibility computations.

2 Contributions

In this paper, we explain how to efficiently convert a *watertight* model into a high definition binary volume representation with solid interior. Our approach has advantages that no other method shares. It is faster than any previous work (up two 2 orders of magnitude even on older hardware), and the memory consumption is usually 8 times lower than for competitors. In contrast to others, an accurate voxelization is assured for any watertight model in a single rendering pass.

We show that this representation is well adapted for many contexts. Further, we propose a GPU adapted scheme (respecting parallelism and memory constraints) to derive a density function which resembles a local distance field. Distance fields are not only useful for physics simulations [34]; they allow the definition of normals for the sampled surface. We avoid storing floats and create a dense packing of information using bits in integers. This allows resolutions that were not possible before. The resulting normal information can be exploited in many contexts like particle simulations or illumination effects. It enables the interaction with the scene whilst involving geometry just during a fast voxelization pass.

More specifically, our contributions are the following:

- **Single-Pass Solid Voxelization** - an extension of the original slicemap algorithm. [10]
- **Conservative Voxelization** - for solids. Boundary voxelization also benefits from our solution.
- **Density/Normal Estimation** - efficient *computation* and *storage* of a local density function of the voxelization

After explaining our method, we illustrate its use in a variety of applications:

- Translucency Effects
- CSG and Object Intersection
- Large Volume Visualization
- Particle Collision Detection and Interaction
- Mathematical Morphology Operations

Each application illustrates the versatility of our real-time solid voxelization and improves previous work in some aspects.

We were motivated by two goals: fast solid voxelization based on slicemaps (explained in Section 4) and the derivation of normal information. Consequently, the method's description is divided into two parts. Section 5 introduces our binary solid voxelization. Section 6 describes how to derive a density (resembling a local distance field) and the normal extraction. We then present several applications (Section 7) and our general results(Section 8). Finally, we discuss the method (Section 9) before concluding (Section 10).

3 Previous Work

Early approaches voxelized based on point queries against the model [24]. Even nowadays, this is not practical for larger models. Haumont and Warzee [16] presented a robust approach that deals with complicated geometry. They report computation times of the order of seconds for typical models. The same holds for [30]. Here, a layered depth representation is derived to count the parity of intersections from each voxel center in the projection direction. We avoid extracting and storing the layered depth images and achieve real-time performance with accurate point sampling.

In our approach, we sample a binary response at the resolution of the voxelgrid. This leads to aliasing artifacts any binary sampling has. It relates to aliasing of standard rasterization where mostly super-sampling is used to hide this problem. In the same spirit, the high resolution of our voxel grid allows to capture most details and we derive a smoother density estimate. Nevertheless, super-sampling can only hide, but not solve the problem. An alias free

*Elmar.Eisemann@inrialpes.fr
†Xavier.Decoret@inrialpes.fr

voxelization is presented in [36], involving an expensive distance calculation for all primitives. In practice (even for medical applications), a binary representation of $< 256^3$ seems often sufficient [31]. This situation cannot be generalized because other sets might need a precision of several billion voxels.

Recently, approaches have been suggested that benefit from the tremendous performance increase of graphics hardware in these last years. Chen and Fang [4] store binary voxels in a bit representation using clipping planes and a transfer into bits of an accumulation buffer. Normal estimation is done by replacing the accumulation texture by slices of a 3D texture (bits become floats). Our approach gives normal information without this costly (both, in time and in memory) process. Even on the latest cards with 1 GB of memory, no Giga Voxel volumes would be possible.

Previous work could not extract large amounts of slices with normal information in real-time [9]. Modern hardware can extract a small amount of property layers in one pass (currently 8 RGBA buffers in DX10). This decreases the grid resolution or increases the number of passes. Further, their solution uses REPLACE blending to avoid incoherent results, thus keeping only the last value of a written fragment in a voxel. Blending operations are not programmable and evidence suggests they will not be for a long time because of optimization issues. Thin elements can thus have front, as well as back faces, fall into the same clipping region and only one normal is stored. *Thin* refers to a 16th of the scene because these techniques rely on around 16 layers due to performance issues. This is problematic when particles interact from all sides. Our attribute extraction is limited but uniform and normals are based on the voxelized shape, leading to coherent values.

Karabassi et al. [20] and Kolb and John [22] use simple depth maps to deduce voxel information and cannot handle concavities correctly. Depth peeling [11] is used in [25]. Arbitrary surface attributes can be trivially obtained. The number of peeling passes is unknown because they are object *and* viewpoint dependent. This implies the need for a high amount of texture memory- especially if extra attributes are retrieved. It also involves a costly occlusion query after each peel. Typically, low depth complexity can be handled. Furthermore, to evaluate the voxels efficiently, all pixels are reprojected from the extracted layers into a uniform representation. ≈ 250.000 vertices are scattered per layer for a 512×512 resolution. Holes may appear for grazing polygons because fragments define one voxel, whereas the depth range they represent might be larger. Depth peeling from several viewpoints tries to address this problem, but it also overwrites concurrent information. Our method handles grazing angles automatically.

Solid voxelization on the GPU has first been presented by Dong et al. [8], who propose a flood fill along the third dimension. Their algorithm fails in cases where two fragments fall in the same voxel. Eisemann and Décoret [10] perform solid voxelization in a single geometry rendering pass. Voxelizations of front and back-facing polygons and a special texture allows to derive the enclosed space without explicit flood fill. Ambiguous situations occur when several front and back-facing triangles fall in one voxel.

Recently, the importance of solid voxelization was shown in the context of interaction with fluid/ gas simulations [5, 26]. The stencil buffer alternates between one and zero to find the parity of the number of intersections towards the eye. This was also used in [4] where the stencil buffer is not reinitialised when passing from one voxel slab to the next. Both approaches are limited to the extraction of one voxel layer per rendering path. Our solution provides ≈ 1000 binary layers in a single pass (and 128 even on older hardware (Geforce 6 series)).

GPU Conservative voxelization is a problem that was solved recently in [38]. The authors derive depth extents for fragments and transform them into bitmasks using a 2D texture lookup. Conservative rasterization is also used in [18]. The approach is accurate, but slow even for small resolutions, as only one slab of voxels is created per render pass. We focus on conservative *solid* voxelization. In this case, two conservative possibilities exist: voxels lie entirely in the interior, or voxels lie partially in the interior. Both are presented in section 5.3. By default, our technique performs artifact free and accurate sampling of the voxel centers.

4 Background: GPU Boundary Voxelization

We build upon the work in [8, 10] that we briefly review here. A binary grid of voxels is represented via a single 2D texture. The voxel (i, j, k) can have value 0 or 1, and is encoded in the k-th bit of the RGBA representation of pixel (i, j). Figure 1 provides an illustration.

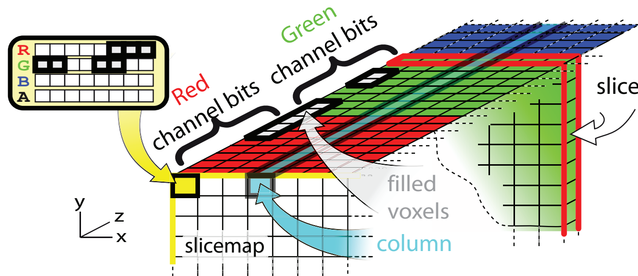

Figure 1: Slicemaps: a binary 3D grid represented using bits of a 2D texture. Instead of seeing color components as separate information, their bits are interpreted as one concatenated boolean vector, where each entry corresponds to a position in space.

The three dimensions of the grid are not treated equally. The texture extents define the xy dimensions. Bits of the RGBA representation of a texel form a *column* of the 3D grid, defining the z-dimension. The k-th bits of all pixels form a "slice", hence the term *slicemap* coined by [10] to refer to the texture.

To obtain a slicemap, it suffices to render a scene. If a fragment's depth lies in the kth slice, the shader outputs a color having only the kth bit activated. By setting the blending mode to a bitwise `OR` operation, all fragments which project to the same pixel are correctly combined [10]. This approach is interesting because it allows to keep track of all fragments in a single pass rather than discarding all but the closest.

The drawback is that depth is quantized, whereas multi-pass methods (such as depth peeling) have full precision and arbitrary information per "slice" (e.g. normals). Advantages include that the result is a single texture and no number of peeling steps needs to be predefined (an often neglected issue).

In [10], the authors further discuss how the choice of the camera/voxel frustum influences the grid's shape, how non-uniformity can be achieved to locally optimize columns (e.g. on a per-pixel basis), and how Multiple Render Targets (MRT) lead to a higher resolution in depth. Our method is compatible with these techniques.

5 Single-Pass Solid Voxelization

This section explains our solid voxelization algorithm. We will precisely define the input of our algorithm (Section 5.1) before providing our solution to the problem of solid voxelization (Section 5.2).

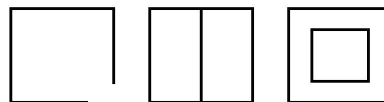

Figure 3: Non watertight models: Models with holes (left), inner walls(middle). In both cases, the interior based on the Jordan theorem is not well-defined, or the symmetric difference is forced (right).

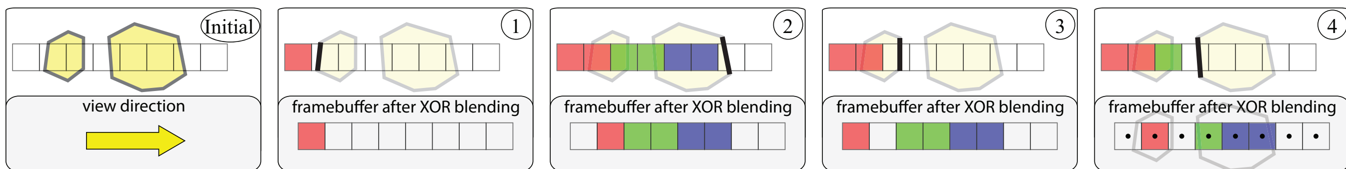

Figure 2: Solid Voxelization for a column in the slicemap. To simplify the illustration, only one framebuffer with two bit color channels is shown. Left: The scene, consisting of two watertight objects, is voxelized in the column along the view direction. 1-4): During rendering, fragments can arrive in an arbitrary order. For each fragment, a bitmask (upper row) is computed in the shader which indicates all voxels that lie in front of the current fragment. This mask is accumulated in the framebuffer (bottom, initialized at zero) using a XOR operation. Once the rendering is complete (4), the framebuffer contains a center sampled solid voxelization in a grid shifted by half a voxel.

5.1 Input to our Algorithm and Limitations

We are interested in a fast method to voxelize *watertight* models. Our definition of *watertight* follows the one in [30]. A model is watertight if for any connected component in space (separated by the geometry), all its points share the same classification: being in the interior or exterior. A point in space is considered interior/exterior if the number of intersections with the model of any ray originating at this point is odd/even (Jordan theorem).

This definition excludes some models from being usable with our technique. Figure 3 shows examples where the definition of an interior is problematic. The left object exhibits a crack in its hull and, therefore, does not define a proper interior. The middle object contains a supplementary wall that separates the inner volume into two parts. Rays shot from one inner part into the other will intersect the model in a pair amount of intersections, while shooting vertically leads to a single intersection. This model is thus not watertight in the above sense. The same holds if the wall coincides with the outer hull. Finally, the rightmost example illustrates a box englobing an inner box. Here, the definition implies that the inner box is an empty region. It is coherent, but not all models are conform to this. This is a limitation that our method shares with several previous works [36, 31, 8, 10, 5].

It is possible to use more advanced techniques in a preprocess to derive a coherent model which is adapted to our algorithm. This step could also exploit supplementary knowledge that is ignored by our solution, such as normals, if this information is accessible. Interestingly, our approach can be integrated into previous work to accelerate the derivation of a coherent mesh, e.g. [30]. Further, most correction methods derive an implicit representation of the input model. A triangulation based on marching cubes [27] is always compatible with our method.

Other elements of the scene (for example the cape of a character) might not be well suited for solid voxelization because they do not define an interior. For these elements, it is possible to perform a separate boundary voxelization [10] and combine it efficiently with our solution for the solid parts, compare section 7. This works even if these elements intersect the geometry.

5.2 Our method for Solid Voxelization

Our approach relates to closed-curve filling in the plane [3]. To achieve fast solid voxelization, we will exploit the definition of watertightness: a point lies inside an object if for any ray leaving the point, the number of intersections with the object's surface is odd. In particular, this condition holds for a view-ray and is e.g., used to test points inside a shadow volume [6]. Therefore, determining whether a voxel lies inside the model amounts to counting the fragments f_i rendered in front of it.

Figure 2 illustrates the voxelization process. Let n fragments lie in front of voxel (i,j,k). It lies inside the model if n is odd ($n \; mod \; 2 = 1$). Consider for a moment that each voxel contains an integer counter and each fragment increments all voxels situated in front of it. Instead of letting the shader output a value having only a single 1 in the kth position based on the fragment's depth (as for the boundary voxelization), it returns a 1 in all positions smaller than k. Adding this column to the corresponding column of counters in the voxel grid, increments the value in exactly all those voxels where any ray along the view direction would intersect the incoming fragment. Maintaining a real counter per slice is impractical on current graphics hardware. To make the accumulation work, we need a second observation: $n \; mod \; 2 = (\sum_{t=0}^{n} 1) \; mod \; 2 = \bigoplus_{t=0}^{n} 1$, where $\oplus$ denotes a XOR operation. In this form, the counters can be stored in a single bit maintaining an in/out status. An incoming bitmask (based on a rasterized fragment) is accumulated by blending with a XOR operation. In practice, this bitmask can be built in the fragment shader by a lookup in a small 1D *bitmask texture* based on the fragments depth.

Due to the way rasterization is performed on current cards and our choice of the bitmask, the voxelization accurately samples centers of a voxel grid shifted by half a voxel along the z-axis. There is no imprecision or aliasing introduced due to the XOR operator. The shift comes from the fact that we choose the bitmask based on the voxel the fragment falls into. Thus, the separations are naturally at the boundary between two column voxels. The offset can be counteracted though by a adding half a voxel to the fragments distance, thus virtually shifting the column.

Supplementary Details

As for shadow volumes, we must ensure that polygons are not clipped by the near and far planes of the rendering camera. *Depth clamp* (NV_depth_clamp extension) performs this operation by clamping depth values to the frustum, and thus outputting all fragments that otherwise would have been excluded by the near/far plane. As we count intersections of rays shot away from the viewpoint, the voxelization remains correct even when created from inside of a volume.

For efficient queries using the voxelization, DX10 (and recently also OpenGL) exposed the *render to texture array* extension which allows the storage of the slices in a texture stack for efficient access. Another possibility is the rendering into a 3D texture. The latter is slower in access, but was available in DX for a longer time. Unfortunately, DX does not support the bitwise blending operations needed for the correct accumulation of the fragments during the voxelization. It has been removed in a previous DX release, but indications exist that they might be reintroduced. For DX9, we use texture tiling to store the result of each pass.

Finally, using two instances, a technique recently exposed in OpenGL, allows to increase the resolution to 2192^3 with a single render call. This limit is imposed solely by memory. The shader stays almost the same. The resolution can be further increased using several passes.

5.3 Conservative Voxelization

One consequence of point sampling is that thin geometry might not pass through the voxel centers and thus remains uncaptured. Technically, the faces on each side fall in the same voxel; they create the same bitmask, which is annihilated by the XOR operation. This makes sense: the resolution of the grid is lower than the details.

For some applications, like conservative visibility testing [7], it might sometimes be necessary to fill even those voxels that lie partially in the interior. This problem is addressed by a so-called *conservative voxelization*.

To achieve this filling, the scene is rendered once with our algorithm. Then, in a second step, a conservative boundary voxelization [38] is added into this same texture. This leads to a conservative solid voxelization because either a voxel is touched by the surface, and thus detected via the conservative boundary rendering; or all points (especially the center) of a voxel lie inside/outside. The only approximation comes from the currently problematic depth range derivation (compare [38]). The process involves conservative rasterization [1] that creates fragments where polygons touch a pixel and further provides a corresponding depth interval.

For a solid voxelization that only keeps entirely interior voxels, one can derive the conservative overestimate and then substract the conservative boundary voxelization by blending with a XOR operation. This delivers the conservative interior without boundary in three passes.

Concerning the implementation, we propose a slight improvement with respect to [38]. Our small 1D bitmask texture can be used instead of allocating large 2D textures (that become especially expensive for DX10 hardware) to transform the two depth extremes into a bitmask which encodes the voxelization. To conservatively activate voxels between these two extents, we shift the farther value by the distance of one voxel and leave the other unmodified. We then perform the lookup of the corresponding bitmasks and achieve a conservative filling in the column by computing a XOR in the shader (or equivalently a substraction for older cards). The offset we applied to the farther depth value ensures that all the voxels lying partially between these two extremes are activated. Finally, the result is blended into the buffer using an OR (not XOR) blending, leading to a conservative boundary. To add the interior voxels, the solid voxelization is simply kept in the framebuffer before adding the hull.

6 Our Method for Density/Normal Estimation

Our solid voxelization is useful in itself, but in some contexts the binary nature hinders its usage. In this section, we will transform the binary grid into a *density representation*.

There is a variety of literature focusing on the question of how to choose an appropriate filtering kernel for the density estimate. A good overview is given in [28]. In our situation, because speed is a concern, we opted for a box-filtering.

The contributions in this section are twofold. We provide a solution to compute and to store the density in a GPU adapted manner, respecting parallelism and the limited memory.

6.1 Overview

We filter and downsample the slicemap to construct a *density map*. Each density map voxel contains a *non-binary* value which indicates how many filled voxels of the original slicemap it represents.

Our solution is general, as any power of two kernel can be obtained dyadically. To simplify, we will concentrate on a kernel of size 2. We perform a box filtering which computes the sum of 2^3 adjacent neighbors. The result is then (just like for mipmapping) stored in a downsampled version of the volume. Formally, we compute:

$$d(i,j,k) := \sum_{l,m,n=0}^{1} v(2i+l, 2j+m, 2k+n)$$

where v is the binary value (0 or 1) of the slicemap in the considered voxel. Consequently, d takes integer values in $[0,8]$. Zero indicates that the voxel lies entirely outside of the model. Eight implies that it lies entirely inside. The boundary between these two regions takes values in the range $[1,7]$. Division by eight leads to an approximate occupancy of the model, hence the name *density map*.

The implementation is complex for several reasons. First, densities are no longer binary; we need several bits to store them. Second, we must organize computations for the GPU (e.g. minimize texture lookups, and optimize parallelism). Consequently, we need to intelligently layout the density map in texture memory.

6.2 Details of the Density Map Construction

Because of the downsampling, we will assume an initial slicemap of size $2w \times 2h$ stored in one integer texture with 32 bits per color channel. Thus the slicemap represents a voxel grid of size $2w \times 2h \times 128$. Next, we derive a density map of size $w \times h \times 64$, where each voxel contains a number in $[0..8]$. This requires 4 bits of storage instead of 1. Combined with a downsampling of a factor of 2 per axis implies a need of 256 bits per density map column. Thus, the representation no longer fits into a single texture and needs to be spread over two $w \times h$ textures representing half of the density map's column (in practice, one tiled texture). Each texture only needs half a column of the slicemap to fill it up. As a consequence, the filtered result for the slices of the first 64 bits (two color channels) will be processed for one, the remaining 64 bits for the other.

We can now focus on an input slicemap representing a $2w \times 2h \times 64$ grid whose density will be stored in a texture of size $w \times h$ with 128 bits per pixel. The storage is sufficient, but the major challenge is to fill this texture efficiently because treating voxels separately is extremely slow. We will derive the density in two steps, first along a column, then spatially in the x,y plane.

To achieve parallel execution, we observe that the sum of two integers of n bits can be stored in $n+1$ bits. Bits higher than $n+1$ will not be touched. Adding the integers i_1^n with i_2^n and j_1^n with j_2^n (where n indicates the number of bits needed to store them) can be done by putting them in subparts of integers with $2n+2$ bits. A sum then actually evaluates two sums in parallel. This holds if more number are concatenated.

With this observation, we compute an intermediate *z-density map* via an in-place operation. It stores two bits per voxel (encoding the pairwise sum of neighboring voxels in a column c) and is given by:

$$(c\& I_{10101...}) + ((c\& I_{01010...}) \gg 1),$$

where & denotes a bitwise AND, and $I_{abc...}$ an integer with bitmask $abc...$. The succeeding zero bit (introduced by the mask) after every copied bit, ensures that the summation will not pollute the solution. This operation also performs the downsampling.

The next step is to sum up voxels of the z-density map in the xy-plane. Four neighboring voxels need to be combined, where now, each voxel is represented with 2 bits. Their sum will thus need 4 bits of storage and we cannot perform an in place operation. Instead (to benefit from parallel execution), we calculate the sums of even and odd voxels seperately as:

$$ESum := \sum_{i=0}^{3} (c_i\& I_{00110011...}) \gg 2,\ OSum := \sum_{i=0}^{3} c_i\& I_{11001100...}$$

i is iterating over the four neighbors in the xy plane. The masks do not only recover the right bit pairs, but they also assure that each is followed by two zero bits which are needed to assure a correct summation. The resulting integers *ESum* and *OSum* can then be stored in the density map. Figure 4 illustrates this step.

At this point, all entries of the density map are already computed. The only catch is that the voxels alternate along the z-direction between values in *EvenSum* and *OddSum*. For normal derivation, this is not problematic, for other applications it might. To simplify usage, one can reorder the result in a parallelized process, detailed in appendix A.

DX9 Implementation

On a DX9 system, the realization is actually simpler. All the above steps are encoded in lookup textures. We use a 256^2 RGBA texture

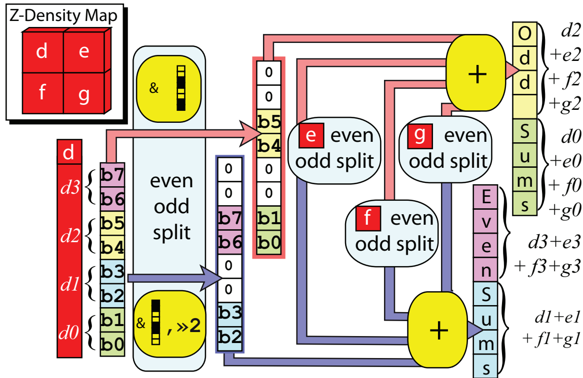

Figure 4: The columns of the the z-density (d,e,f,g) contain the sum of neighboring voxels along the z-axis (encoded on 2 bits). The final density is computed by summing four neighboring z-densities in the xy-plane. This is done in parallel by splitting the vectors into even and odd entries that are summed separately. (To simplify the illustration each column is represented with only 8 bits). It has to be noted that the splitting leads to a result that is ordered 0,2/1,3 and not 0,1/2,3.

that, based on two 8 bit voxel columns, gives the same configuration we obtain with the above algorithm, namely four 8 bit components that contain neighboring bit sums separated by two zero bits. These lookup values of the neighboring columns can then be safely added to yield the final density.

6.3 Normal Derivation from Density

Following the definition for an implicit surface, we can compute the gradient ∇d along the three axis via finite differences and derive the normal n from it by $\nabla d/||\nabla d||$. Each component has a value in $[-8,8]$, thus about 17^3 directions are possible (removing those for which normalization leads to the same result gives a total of $4034 \approx 2^{12}$ distinct directions).

We use a symmetric kernel needing 6 values around the center voxel. Five lookups are sufficient (the two neighbors in the same column are retrieved together). A simpler kernel, with only the center and three neighbors, would require 3 lookups, but the normals are of lower quality for an insignificant speedup. In [29] alternative normal derivations are discussed.

7 Applications

Translucency

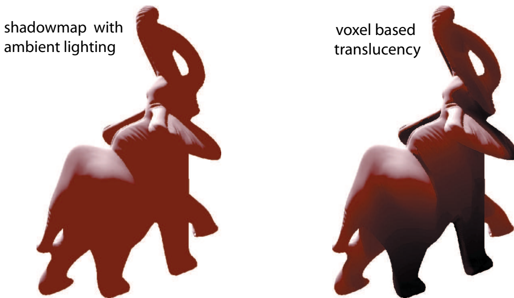

Figure 5: Translucency effect: Even for more complex models with 60,000 triangles, the framerate exceeds $200Hz$ on a Geforce 6.

Solid voxelization can be used for translucency effects. As for Transmittance Shadow Maps [10], the number of filled voxels between a scene point and the light position is calculated. Due to the high grid resolution, bit counting with textures is expensive. Instead, we solve this arithmetically using a dyadic approach [2]. Figure 5 shows a result. The volume approximation is quite accurate and leads to better solutions than simply taking a difference between front and back depth maps [37]. This was pointed out in [10], but the solution was more complex and could contain artifacts.

CSG and Inter-object intersection

To test object penetration, many works rely on complicated data structures or numerous occlusion queries. With solid voxelization, general CSG is straightforward. Each object is voxelized in a separate slicemap. Then we render the first over the second, blending with the desired boolean operation. Figure 6 shows an example.

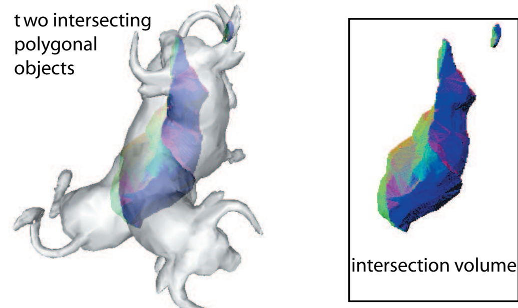

Figure 6: Example of a CSG Operation: Discretized intersection of two complex meshes. The cost is directly linked to the voxelization.

We want to underline that it is simple, but works at a precision of several *billion* (10^9) voxels in real-time with low dependence on the input geometry's complexity. Furthermore, the density computation we presented in section 6 allows to recover the intersection's volume rapidly. This is useful for collision detection, or haptic feedback. An extension to CSG trees is possible by storing intermediate results (intuitively $\log n$ where n is the height of the tree). Rearranging the tree could optimize this number [14].

Visualization

Compactness is one interest of slicemaps. Estimating normals compensates to some extent for the lack of other than binary information, and allows to reflect the surface's appearance in real-time. This becomes useful when visualizing level sets of large CT scans (1024^3). Usually, this amount of memory would not fit in the card making interactive visualization via slicing [19] impossible. It would stall the pipeline with texture transfers. Marching cubes [27] are not feasible at this resolution either . A slicemap of a level set is small and can be created by one slicing pass on the GPU or transferred directly from the CPU. The display is interactive and several level sets can be kept at the same time and blended together. This technique was used to display the CSG result in Figures 6 and 8 (false colors represent our normal estimate).

Particle collision

We apply our technique to a GPU particle system similar to [23], although we have not optimized the particle rendering and simulation. We detect collisions and make particles bounce based on the solid voxelization with normals. Figure 7 shows some examples.

Our approach seamlessly treats concave regions, like the complex toboggan scene (Figure 7 (right)). Dynamic deformation is possible since we recompute density and normals at every frame. The whole scene is queried via a single representation and the computation is efficient: normals are evaluated only when a particle enters in contact with a surface. Surprisingly the actual bottleneck is the particle display via billboards. The simulation runs entirely on the GPU. The precision is high and even particles crossing a boundary (due to high velocity or large time steps) can be detected since we represent a solid volume. In these situations, we perform

Figure 7: Two examples we used to show the benefit of using our voxelization in the context of particle simulations. The stone in the left example contains several tunnels and holes. The right example shows the high resolution of the voxelization as the fine geometry of the toboggan is captured.

back-integration and estimate the actual impact point. Of course, theoretically, particles can still cross very thin volumes.

This being said, it shows notable advantages compared to previous work. Voxelizations from depth peeling might not have a sufficient resolution or holes in certain directions and clipping plane approaches have only restricted information along the z-axis. Consequently, these simulations rely on particles having some privileged direction, whereas our voxelization is uniform in the sense that all directions share the same quality. None can easily perform consistent back integration. They do have the possibility to capture object motion directly. On the other hand, our solution can be combined with a movement extraction step like in [9]. Motion can typically be sampled at very coarse levels. Rigid motion is a constant and can be passed directly in the shader.

Mathematical Morphology

Finding an eroded interior is useful for many applications like path finding or visibility (e.g., [32, 7]), and are often obtained in a lengthy preprocess [32]. Dilation creates a hierarchical structure which allows rapid queries on whether a neighborhood contains filled voxels. In a binary context, erosion/dilation are simple logical operations (AND/OR). They are separable, so one can first erode/dilate along the x, y, then along the z direction. Finally, larger sizes are obtained by iteration. This implies that arbitrary rectangular kernels are possible. Bit shifting (like in section 6) allows to treat columns efficiently. Care has to be taken as some bits are needed from adjacent integers. Figure 8 shows an example.

Figure 8: Erosion(left) and Dilation(right) can be obtained directly from the solid voxelization which represents the only step involving the actual scene geometry making the solution fast because it is purely image-based.

8 Results

Resolution and Storage: We tested our approach (implemented in OpenGL) for DX9 cards on a simple G6 NVidia 6800 (*non* GT/ULTRA), and on DX10 hardware with NVidia's G80 (8800 GTS). The latter supports 32 bit integer RGBA textures of resolutions up to 8192×8192, and can write into 8 MRT's in a single pass. We can represent a grid whose resolution in x and y is 8192 pixels, and whose theoretical resolution in z is $4 \times 32 \times 8 = 1024$ bits. With the current drivers, it seems though that the unsigned integer format reserves the 32nd bit (possibly for exceptions). Values are not correctly compared via equality if it is activated. This could be an issue for applications using our technique. We discuss the needed minor changes in the appendix B. In our applications, we primarily use grids of around $1024^2 \times 992$. The memory footprint of these 8×1M pixel textures amounts to 128 MB.

Prior to the G80 series, the integer types were not supported. Therefore, the behavior can only be emulated (in the shader, all values are floating point, no matter the texture type). As a result, only 8 bits per color channel are possible, leading to 32 bits total per texture. From the G6 series on, 4 MRTs are possible, thus allowing 128 bits that can be used per rendering pass. The texture resolution is limited to 4096^2.

One important implication of storing the information in bits is, that in both implementations, the memory consumption is at least 8 times lower than for other approaches, e.g. [9, 5].

Performance: As a test scene, we used a Stanford dragon model with 262,078 triangles which almost filled the whole frustum. In a second test, we added four copies (leading to 1,310,390 triangles). Timings are shown in Table 1. In the case, where the interior is less dense and contains empty parts between objects, the framerate increases. For five dragons at 1024^3, the cost drops below 6 ms if only a fifth of the grid is occupied (which is the case when placing them with small separations). The timings on for the DX9 card are shown in Table 2.

resolution	512^3	1024^3
262,078 tris	1.6 ms	10.65 ms
1,310,390 tris	5.29 ms	41.5 ms

Table 1: DX10: Solid voxelization timings on NV8800 GTS

resolution	64^3	128^3	256^3	512^3
500 tris	0.19 ms	0.22 ms	0.6 ms	2,5 ms
5,000 tris	0.26 ms	0.33 ms	0.9 ms	3.9 ms
12,500 tris	0.36 ms	0.4 ms	1.1 ms	5 ms
25,000 tris	0.58 ms	0.6 ms	1.6 ms	6.3 ms
262,078 tris	3.5 ms	3.6 ms	7 ms	23.3 ms

Table 2: DX9: Solid voxelization timings on NV6800

With respect to previous work, our algorithm performs at higher speed (see Figure 9). In all tests, the frustum was fit to a bounding sphere. For a fair comparison, we optimized [5] to work directly on the texture using logical operations (without using the stencil buffer). We also followed their future work suggestions, and tried instancing and texture tiling to accelerate the approach. In practice, it turns out that instancing is less interesting from a performance standpoint because the shaders are slightly more complicated.

res.	64^3	128^3	256^3	512^3	256^3	512^3	1024^3
ms	0.25	0.6	2.9	20.9	0.28	0.9	6.5

Table 3: DX9| DX10 : Density computation timings

For DX10 the timings in Table 3 depend solely on the resolution of the initial slicemap and include the reordering, which in comparison is not very expensive. In contrast, for DX9 the reordering is for free, but the content of the slicemaps play a role because the cache comes into play due to dependent lookups. In practice, we realized that results seem to vary little (around 10%). Table 3 summarizes the timings as an average of several models.

Finally Table 4 shows the timings for our DX10 particle demo including the response and normal derivations.

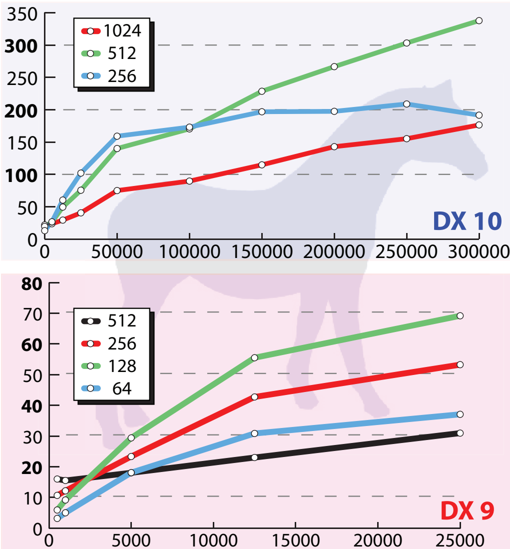

Figure 9: Speedup with respect to [5] for several LOD's of the horse model (with originally 300,000 triangles). Top: DX10 implementation (G80) Bottom: DX9 implementation (G6)

Nb. particles	256^2	512^2	1024^2
Collision management	0.32 ms	1.0 ms	4.0 ms

Table 4: Particle System collision response timings. It is voxelgrid independent (here 512^3) because computations are local per particle

9 Discussion

Once exposed, the approach for solid voxelization seems simple, but efficient single pass solutions do not exist. Furthermore the approach in this paper is accurate (point sampled or over/under conservative), and its simplicity makes it appealing.

Contrary to the methods in [8, 10], our solution is accurate even if an arbitrary number of fragments fall in a single voxel. Furthermore, we do not need a front and back face separation as in [10]. Like for many CPU based methods, we do not need a manifold or a topologically coherent mesh; a watertight input suffices. The method derives the interior defined by the Jordan Theorem.

If two watertight models intersect, our algorithm assumes the symmetric difference of the two. Meaning that the two concentric boxes of Figure 3 will lead to a hollow representation, not the union. This is consistent with the definition of watertight (Section 5.1). In section 7, we showed how to perform general CSG operations (also the union).

Solid Voxelization

Concerning the performance for the DX10 implementation, the difference to [5] is remarkable for complex models (up to a factor of > 300, thus about 2 orders of magnitude). This stems from the fact that several layers (up to 1024) can be extracted in a single geometry pass, and thus the theoretical speedup is 1024. In practice, the fragment output still has some cost and the blending comes at an expense too. Nevertheless, even for a simple cube (12 triangles) we gain a factor of around 20. The cube is a kind of worst cases scenario because the geometric complexity is low and fill-rate high. Standard rendering is very cheap in this case and already runs at $> 100Hz$ for a 256^3 volume, while the framerate drops quickly for high polygon models and voxel resolution.

For the DX9 implementation, we also obtain strong speedups even for less complex models. The cards are less powerful in treating geometry than the recent generation which comes in part also from the architecture. Shading units are now general stream processors that can be reassigned to the fragment or vertex shader according to the workload. In the old generation, the number of vertex units was fixed and thus our algorithm compares quickly favorable with respect to [5] (see Figure 9).

Finally, our method is also interesting for even older cards without MRT support (e.g. Geforce 3). To show the advantage, we ran the tests deactivating the MRT feature of the G6. Without, 32 layers can still be extracted per rendering. Surprisingly, performance remained about twice the cost compared to four MRTs for models up to 12,500 triangles and converged towards a speedup factor of 23 with respect to [5]. Generally MRTs come at some cost, but one reason why the difference is not closer to four could be that the G6 had a sweet spot concerning its MRTs. Already, four MRTs impose more than the expected 33% supplementary workload with respect to three MRTs and could explain the behavior. Exact performance depends on many factors including the chosen model, but the measured tendencies remained the same throughout all our tests.

Density and Normals

Our density computation is fast and memory efficient because all the information is tightly packed in bits and evaluated in parallel. In contrast to other methods [9, 25], we do not rely on the mesh's normals, but those defined by the voxelized surface. This is key to obtaining high resolution and normals coherent with respect to the voxelized volume. As described in Section 6.1, the kernel we use for our density estimate is of size 2^3. We found, just like [30], that even this small kernel gives acceptable quality in practice.

Our DX9 implementation is well adapted to those cards, because all four components of a color will be treated equally during the process, which reflects the vector capacities of its processors. It would *not* scale for DX10 hardware. A lookup texture for a 32 bit integer would need 2^{32} entries and does not fit into the memory of the GPU. Breaking the integers down into smaller parts of 16 bit lengths would still imply two lookups per channel, leading to a total of $8 \times 4 \times 2 = 64$ lookups per 1024 bit voxel column. In this case, our bitwise arithmetic proofs more efficient.

The density can be seen as a localized version of distance fields. Much work has been published in this area and in particular GPU implementations exist [33, 34, 35]. The larger support of these distance functions allows a more general usage and can make them preferable to local density. On the other hand, our method is faster to compute and interesting for applications that need only limited distance information, some of which were presented in section 7.

Our method does not need to store normals explicitly because the gradient computation is not very costly and in the context of a (one million) particle system, the number of issued queries is much lower than the size of the density map (512^3). In practice, the proposed simple scheme leads to sufficient precision and allows us to evaluate the normals on the fly.

10 Conclusion and Future Work

In this paper, we presented a method for solid voxelization of dynamic scenes at high frame-rates. It outperforms previous approaches in terms of speed and resolution. We showed how to obtain a sampled voxelization, as well as a conservative solution. The basic method is easy to implement, does not need knowledge about the scene geometry (only a depth value has to be produced), and is compatible with shader based animation. The density function is a quick way to derive something resembling a local distance field. The normal estimate based on it, is more efficient and simpler to handle than depth peeling.

This work allows a large variety of new applications besides the presented ones. Algorithms for advanced collision detection could benefit from this representation. Particle or ray-tracing algorithms (e.g. for refraction) could make use of our hierarchical representation (Mathematical Morphology, section 7).

Acknowledgements: We would like to thank W.J. Willet, H. Bezerra, P.-E. Landes, K. Subr, E. Turquin and M. Eisemann for their comments on the paper. Further, F. Moulin (stone), Cyberware and INRIA ISTI for the models. We thank the reviewers for their helpful suggestions that substantially improved structure and content.

References

[1] T. Aila and T. Akenine-Möller. Conservative and tiled rasterization. *Journal of Graphics Tools 10(3)*, 2005.

[2] S. E. Anderson. Bit twiddling hacks. graphics.stanford.edu/~seander/bithacks.html.

[3] C. Baldazzi and A. Paoluzzi. From polyline to polygon via xor tree, 1996.

[4] H. Chen and S. Fang. Fast voxelization of 3d synthetic objects. In *ACM Journal of Graphics Tools, 3(4):33-45*, New York, NY, USA, 1999. ACM Press.

[5] K. Crane, I. Llamas, and S. Tariq. *GPU Gems 3*, chapter Ch. 30: Real-Time Simulation and Rendering of 3D Fluids. Addison-Wesley, 2007.

[6] Crow. Shadow algorithms for computer graphics. In *Proceedings of SIGGRAPH '77*, 1977.

[7] X. Décoret, G. Debunne, and F. Sillion. Erosion based visibility preprocessing. In *Proceedings of the Eurographics Symposium on Rendering*. Eurographics, 2003.

[8] Z. Dong, W. Chen, H. Bao, H. Zhang, and Q. Peng. Real-time voxelization for complex polygonal models. In *Proc. of Pacific Graphics'04*, 2004.

[9] S. Drone. Advanced real-time rendering in 3d graphics and games. SIGGRPAPH course notes, 2007.

[10] E. Eisemann and X. Décoret. Fast scene voxelization and applications. In *Proc. of I3D'06*, 2006.

[11] C. Everitt. Interactive order-independent transparency. webpage sur NVIDIA developpers http://developer.nvidia.com/, 2001.

[12] S. Fang and D. Liao. Fast csg voxelization by frame buffer pixel mapping. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 43–48, New York, NY, USA, 2000. ACM Press.

[13] S. F. F. Gibson. Beyond volume rendering: Visualization, haptic exploration, an physical modeling of voxel-based objects. In R. Scanteni, J. van Wijk, and P. Zanarini, editors, *Visualization in Scientific Computing '95*, pages 9–24. Springer-Verlag Wien, 1995.

[14] J. Hable and J. Rossignac. Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1024–1031, New York, NY, USA, 2005. ACM Press.

[15] T. Harada and S. Koshizuka. Real-time cloth simulation interacting with deforming high-resolution models. SIGGRAPH Poster, 2006.

[16] D. Haumont and N. Warzee. Complete polygonal scene voxelization. *Journal of Graphics Tools*, 7(3), 2002.

[17] T. He. Volumetric virtual environments, 1996.

[18] H.-H. Hsieh, Y.-Y. Lai, W.-K. Tai, and S.-Y. Chang. A flexible 3d slicer for voxelization using graphics hardware. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 285–288, New York, NY, USA, 2005. ACM Press.

[19] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen. *GPU Gems*, chapter Ch. 39: Volume Rendering Techniques. Addison-Wesley, 2004.

[20] E.-A. Karabassi, G. Papaioannou, and T. Theoharis. A fast depth-buffer-based voxelization algorithm. *J. Graph. Tools*, 4(4):5–10, 1999.

[21] T.-Y. Kim and U. Neumann. Opacity shadow maps. *Eurographics Rendering Workshop*, 2001.

[22] A. Kolb and L. John. Volumetric model repair for virtual reality applications, 2001.

[23] L. Latta. Building a million particle system. Lecture at the GDC, 2004.

[24] Y. Lee and A. A. G. Requicha. Algorithms for computing the volume and other integral properties of solids. In *Communications of the ACM, 25(9):635 650*, 1982.

[25] W. Li, Z. Fan, X. Wei, and A. Kaufman. *GPU Gems 2*, chapter Ch. 47: Simulation with Complex Boundaries. Addison-Wesley, 2004.

[26] I. Llamas. Real-time voxelization of triangle meshes on the gpu. In *Technical sketch at SIGGRAPH*, New York, NY, USA, 2007. ACM Press.

[27] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[28] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In R. D. Bergeron and A. E. Kaufman, editors, *Proceedings of Visualization '94*, pages 100–107, 1994.

[29] T. Möller, R. Machiraju, K. Mueller, and R. Yagel. A comparison of normal estimation schemes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 19–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[30] F. S. Nooruddin and G. Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191, 2003.

[31] B. Reitinger, A. Bornik, and R. Beichel. Efficient volume measurement using voxelization. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, pages 47–54, New York, NY, USA, 2003. ACM.

[32] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion. Conservative volumetric visibility with occluder fusion. In K. Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 229–238. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[33] C. Sigg, R. Peikert, and M. Gross. Signed distance transform using graphics hardware. In *Proceedings of IEEE Visualization '03*, pages 83–90. IEEE Computer Society Press, 2003.

[34] A. Sud, N. Govindaraju, R. Gayle, and D. Manocha. Interactive 3d distance field computation using linear factorization. In *Proc. ACM Symposium on Interactive 3D Graphics and Games (I3D)*. ACM Press, 2006.

[35] A. Sud, M. Otaduy, and D. Manocha. Difi: Fast 3d distance field computation using graphics hardware. In *Computer Graphics Forum (Proc. Eurographics)*, 2004.

[36] S. W. Wang and A. E. Kaufman. Volume sampled voxelization of geometric primitives. In *VIS '93: Proceedings of the 4th conference on Visualization '93*, pages 78–84, 1993.

[37] C. Wyman. An approximate image-space approach for interactive refraction. In *SIGGRAPH 2005: Proceedings of the 32th International Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 2005. ACM Press.

[38] L. Zhang, W. Chen, D. S. Ebert, and Q. Peng. Conservative voxelization. *The Visual Computer*, 23(9-11):783–792, 2007.

A Reordering

Reordering two integers containing 4-bit groups representing voxels 0,2,4,6 and 1,3,5,7 as depicted in Figure 10 is done in a parallel manner as well. The idea is to shift groups of information at the same time, whenever it is possible.

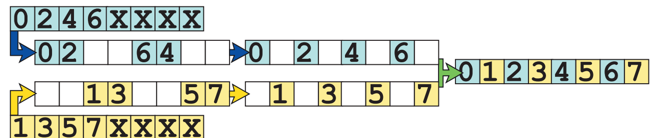

Figure 10: Parallel reordering of interleaved groups of bits

B Remarks on the 32nd bit

The modifications needed to avoid the 32nd bit that seems to be reserved for different purposes, are minor. For the solid voxelization algorithm, only the 1D texture needs to be modified to exclude the highest bit. Therefore, in practice, we use 992 slices in a single pass, instead of 1024.

For the computation of the density function, it is important to keep a size that is compatible with the downsampling. In practice, we use 28 bits per color channel, which is the biggest multiple of four below 32, leading to 960 instead of 1024 initial slices.

When comparing our performance to previous work, we took the smaller amount of slices into account.