Efficient Sparse Voxel Octrees

Samuli Laine Tero Karras
NVIDIA Research

Figure 1: *Sibenik cathedral ray-traced using voxels. Voxel data was created with high-resolution surface displacement, and ambient occlusion was calculated as a pre-process step. All geometry and shading data is stored on a per-voxel basis, i.e. there are no instantiated objects, textures, or materials. The resolution is approximately 5mm throughout the entire building, including outer walls that are not visible from the inside. The total size of the data in GPU memory is 2.7 GB. Our ray caster is able to cast 60.9 million primary rays per second for this data, and 122.0 million rays per second for the non-displaced version of the same scene. For comparison, the fastest triangle-based GPU ray caster to date ([Aila and Laine 2009]) achieves 107.1 million rays per second for the non-displaced variant on the same hardware.*

Abstract

In this paper we examine the possibilities of using voxel representations as a generic way for expressing complex and feature-rich geometry on current and future GPUs. We present in detail a compact data structure for storing voxels and an efficient algorithm for performing ray casts using this structure.

We augment the voxel data with novel contour information that increases geometric resolution, allows more compact encoding of smooth surfaces, and accelerates ray casts. We also employ a novel normal compression format for storing high-precision object-space normals. Finally, we present a variable-radius post-process filtering technique for smoothing out blockiness caused by discrete sampling of shading attributes.

Our benchmarks show that our voxel representation is competitive with triangle-based representations in terms of ray casting performance, while allowing tremendously greater geometric detail and unique shading information for every voxel.

1 Introduction

Voxels can be seen as a simpler alternative to the triangle pipeline that has become relatively complicated in the current GPUs. Traditionally, voxels have been used for representing volumetric data such as MRI scans, but in this paper we concentrate on using them as a densely sampled representation of opaque surfaces.

A compelling reason for using triangles has been their compactness for representing planar surfaces. This advantage is less significant today, because memory consumption is already dominated by color textures and normal maps that are required for realistic look. Displacement maps can be used to obtain true geometric detail, but only the latest GPUs are able to rasterize them efficiently. Displaced geometry is also more difficult to ray trace than flat triangles.

It is customary to use the same textures over and over to conserve GPU memory. Unfortunately, this results in repetitive look for materials and makes it difficult to add variation in the scene, although small details can be easily added using decal texture patches. *id Software* pioneered the use of a single large texture for terrain in its *id Tech 4* engine, and the current *id Tech 5* engine extends the technique to all textures. Only a subset of this so-called megatexture is kept in memory, and missing parts are streamed from disk as they are needed.

Assuming that such megatexturing will become commonplace in the future, we need to store a color value per resolution sample for all surfaces. If megatextured displacement mapping is used to achieve higher geometric complexity, we effectively need to also store some amount of geometry data per sample. At this point, we may quite reasonably ask why a separation between coarse geometry (base mesh) and fine detail (color and displacement maps) is necessary in the first place. If we have to store color and geometry data per resolution sample anyway, why not use a simpler representation that utilizes the same data structure for both purposes?

2 Previous Work

There is a vast body of literature on visualizing volumetric structures, so we will focus on papers that are most directly related to our work. We specifically omit methods that are restricted to height fields (see e.g. [Dick et al. 2009] for a recent contribution) or are based on a combination of rasterization and per-pixel ray casting in shaders (see [Szirmay-Kalos and Umenhoffer 2008] for an excellent survey) because these are not capable of performing generic ray casts.

Amanatides and Woo [1987] were the first to present the regular grid traversal algorithm that is the basis of most derivative work, including ours. The idea is to compute the t values of the next subdivision planes along each axis and choose the smallest one in every iteration to determine the direction for the next step.

Knoll et al. [2006] present an algorithm for ray tracing octrees containing volumetric data that needs to be visualized using different isosurface levels. Their method is conceptually similar to kd-tree traversal, and it proceeds in a hierarchical fashion by first determining the order of the child nodes and then processing them recursively. The algorithm is not as such well suited for GPU implementation. An extension to coherent ray bundles is given by Knoll et al. [2009].

Crassin et al. [2009] present a GPU-based voxel rendering algorithm that combines two traversal methods. The first stage casts rays against a regular octree using kd-restart algorithm to avoid the need for a stack. The leaves of this octree are bricks, i.e. 3D grids, that contain the actual voxel data. When a brick is found, its contents are sampled along the ray. Bricks typically contain 16^3 or 32^3 voxels, yielding a lot of wasted memory except for truly volumetric data. On the other hand, mipmapped 3D texture lookups supported by hardware make the brick sampling very efficient, and the result is automatically antialiased. An interesting feature of the algorithm is the data management between CPU and GPU. The renderer detects when data is missing in GPU memory and signals this to the CPU, which then streams the needed data in. This way, only a subset of nodes and bricks needs to reside in GPU memory at any time.

Ju et al. [2002] augment an octree structure with auxiliary data to improve fine geometric details. While the Hermite data utilized by their representation is flexible in its ability to support dynamic CSG operations, a separate triangulation pass is required to render the resulting surface. Our contour-based representation, on the other hand, aims for efficient rendering and compact storage given the assumption of static geometry.

3 Voxel Data Structure

We store voxel data in GPU memory using a sparse octree data structure where each node represents a voxel, i.e. an axis aligned cube that is intersected by surface geometry. Voxels may be further subdivided into smaller ones, in which case both the parent voxel and its children are included in the octree. The data structure has been designed to minimize the memory footprint while supporting efficient ray casts. Sometimes both can be achieved at the same time, because more compact data layout also reduces the memory bandwidth requirements.

To this end, we adopt a scheme where most of the data associated with a voxel is stored in conjunction with its parent. This removes the need to allocate storage for individual leaf voxels and makes compression of shading attributes more convenient.

On the highest level, our octree data is divided into *blocks*. Blocks are contiguous areas of memory that store the octree topology along with voxel geometry and shading attributes for localized portions

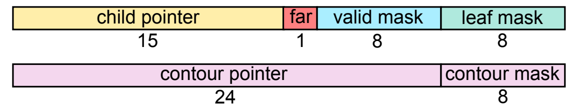

Figure 2: *64-bit child descriptor stored for each non-leaf voxel.*

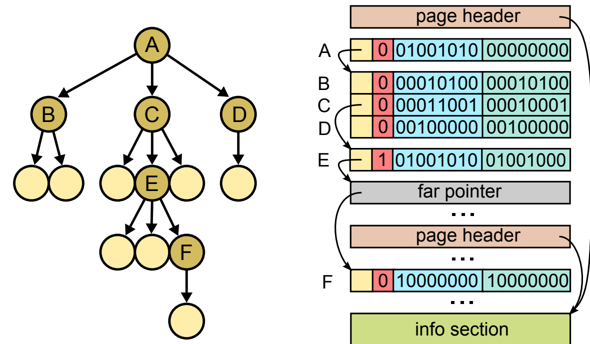

Figure 3: *Layout of the child descriptor array. Left: Example voxel hierarchy. Right: Child descriptor array containing one descriptor for each non-leaf voxel in the example hierarchy.*

of the octree. All memory references within a block are relative, making it easy to reorganize blocks in memory. This facilitates dynamic memory management necessary for out-of-core rendering.

Each block consists of an array of child descriptors, an info section, contour data, and a variable number of attachments. The child descriptors (Section 3.1) and contour data (Section 3.2) encode the topology of the octree and the geometrical shape of voxels, respectively. Attachments (Section 3.5) are separate arrays that store a number of shading attributes for each voxel. The info section encompasses a directory of the available attachments as well as a pointer to the first child descriptor.

We access child descriptors and contour data during ray casts. Once a ray hits surface geometry, we execute a shader that looks up the attachments contained by the particular block and decodes the shading attributes. For the datasets presented in this paper, we use a simple Phong shading model with a unique color and a normal vector associated with each voxel.

3.1 Child Descriptors

We encode the topology of the octree using 64-bit child descriptors, each corresponding to a single non-leaf voxel. Leaf voxels do not require a descriptor of their own, as they are described by their parents. As illustrated in Figure 2, the child descriptors are divided into two 32-bit parts. The first part describes the set of child voxels, while the second part is related to contours (Section 3.2).

Each voxel is subdivided spatially into 8 child slots of equal size. The child descriptor contains two bitmasks, each storing one bit per child slot. *valid mask* tells whether each of the child slots actually contains a voxel, while *leaf mask* further specifies whether each of these voxels is a leaf. Based on the bitmasks, the status of a child slot can be interpreted as follows:

- Neither bit is set: the slot is not intersected by a surface.
- The bit in *valid mask* is set: the slot contains a non-leaf voxel.
- Both bits are set: the slot contains a leaf voxel.

If the voxel contains any non-leaf children, we store an unsigned 15-bit *child pointer* in order to reference their data. These children,

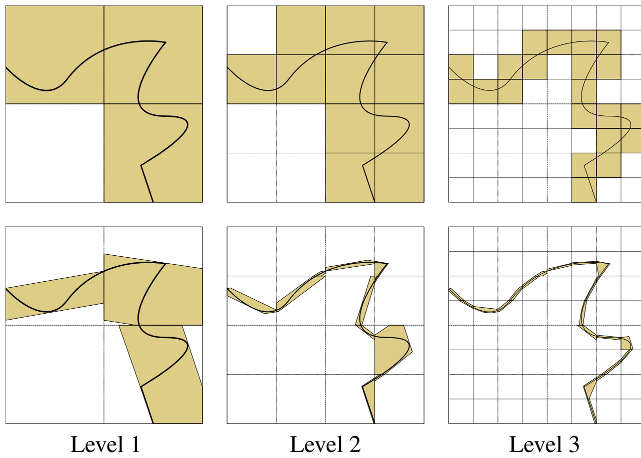

Figure 4: *Effect of contours on surface approximation. Top row: cubical voxels. Bottom row: voxels enhanced with contours. The resulting approximation follows the original surface much more closely than in the top row. Sharp corners can be approximated by taking contours on multiple levels into account simultaneously.*

in turn, store their own child descriptors at consecutive memory addresses, and the *child pointer* points to the first one of them as illustrated in Figure 3. This way, we can find a particular child by incrementing the pointer based on the bitmasks. The children can reside either in the same block or in a different one, making it possible to traverse the octree without having to consider block boundaries.

In case the children are located far away from the referencing descriptor, the 15-bit field may not be large enough to hold the relative pointer. To indicate this, we include a *far* bit in the child descriptor. If the bit is set, the *child pointer* is interpreted as an indirect reference to a separate 32-bit far pointer. The far pointer is interleaved within the same array and has to be placed close enough to the referencing descriptor. In practice, far pointers can be made very rare by sorting child descriptors in an approximate depth-first order within each block.

In addition to traversing the voxel hierarchy, we must also be able to tell which block a given voxel resides in. This is accomplished using 32-bit page headers spread amongst the child descriptors. Page headers are placed at every 8 kilobyte boundary, and each contains a relative pointer to the block info section. By placing the beginning of the child descriptor array at such a boundary, we can always find a page header by simply clearing the lowest bits of any child descriptor pointer.

3.2 Contours

The most straightforward way to visualize voxel data is to approximate the geometry contained within each voxel as a cube. The resulting visual quality is acceptable as long as the data is oversampled, i.e. the projection of each voxel on the screen is smaller than a pixel. However, if voxels are considerably larger due to undersampling, the approximation produces very noticeable artifacts near the silhouettes of objects. This is due to the fact that replacing each intersection between a voxel and the actual surface with a full cube effectively expands the surface. This introduces significant approximation error as illustrated in top row of Figure 4.

To reduce the approximation error, we constrain the spatial extent of each voxel by intersecting it with a pair of parallel planes matching the orientation of the approximated surface. We refer to such a pair of planes as a contour. The result is a collection of oriented

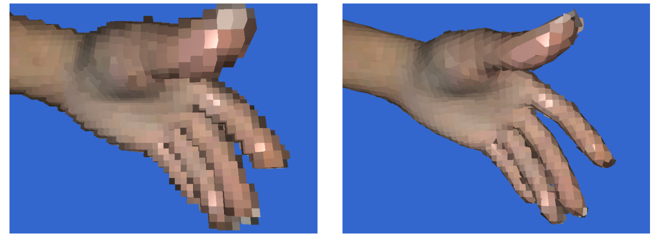

Figure 5: *Left: cubical voxels. Right: the same voxels with contours. In this kind of situation where surfaces are reasonably smooth, contours can provide several hierarchy levels' worth of geometric resolution improvement. Note that the model has been deliberately undersampled to illustrate the effect.*

slabs that define a tight bounding volume for the surface, as illustrated in the bottom row of Figure 4. For flat and relatively smooth surfaces, the planes can be oriented with the average surface normal to obtain a good fit. For curved and undersampled surfaces, the planes can still be used to reduce the approximation error, as can be seen in Figure 5. Previously, linear bounding volumes have been constructed by e.g. Peters et al. [2004] for NURBS surfaces.

We use 32 bits to store the contour corresponding to one voxel. The value is divided into five components: three 6-bit integers to define the normal vector of the two planes, and two 7-bit integers to define their positions within the voxel.

The mapping between voxels and their corresponding contours is established by two fields in the child descriptor (Figure 2). *contour mask* is an 8-bit mask telling whether the voxel in each child slot has an associated contour. Storing a separate bitmask allows omitting contours in voxels where they do not significantly reduce the approximation error. Similar to the *child pointer*, the unsigned 24-bit *contour pointer* references a list of consecutive contour values, one for each bit in *contour mask* that is set.

3.3 Cooperation Between Contours

While using only one contour per leaf voxel would be enough to represent smooth surfaces, it would also introduce distracting artifacts near sharp edges of objects. This is because the orientation of the surface varies a lot within voxels containing such an edge, and no single orientation can be chosen as a good representative for all the points on the surface.

Fortunately, we can utilize the fact that we are storing a full hierarchy of overlapping voxels. To enable cooperation between multiple contours, we define the final shape of a voxel as the intersection of its cube with all the contours of its ancestors. This way, we can incrementally augment the set of representative surface orientations by selecting different normal vectors for the contours on each level of the octree, yielding significant quality increase. The effect can be seen near the areas of high curvature in the bottom row of Figure 4.

3.4 Construction of Contours

To simplify the task of approximating a given surface with contours, we observe that the result does not necessarily need to be smooth. As long as we ensure that the original surface is fully enclosed by each contour, we are guaranteed to get an approximation that contains no holes. While discontinuities at voxel boundaries may introduce problems such as false self-shadowing or interreflections in ray tracing, these can be usually worked around by offsetting the starting positions of secondary rays by a small amount. Thus, the construction process can be defined in terms of minimizing the spatial extent of each voxel, regardless of its neighbors.

We employ a greedy algorithm that constructs a contour for each voxel in a hierarchical top-down manner. The construction is based on the original surface contained within a given voxel, as well as the ancestor contours that have already been determined. We first construct a polyhedron by taking the intersection between the voxel's cube and each of its ancestor contours. We then pick a number of candidate directions and determine how much the original surface is being overestimated by the polyhedron in each direction. Overestimation is calculated as the difference between the spatial extents of the polyhedron and the original surface along the given candidate direction. Finally, we select the direction with the largest overestimation and construct a contour perpendicular to it so that it encloses the original surface as tightly as possible.

Due to the greedy nature of the construction process, the quality of the resulting approximation depends heavily on the chosen set of candidate directions. Since we have a limited number of hierarchy levels, we want to avoid choosing directions that would only reduce the spatial extent locally without contributing to the final shape of leaf voxels. We thus restrict the set of candidate directions to normals of the original surface as well as perpendiculars of surface boundaries, since these directions are most likely to contribute to the final shape. In practice, we have found that it is enough to consider only a relatively small subset of these directions in order to speed up the processing.

In addition to constructing contours, we also want to detect the case where the shape of the voxel as defined by its ancestor contours already approximates the original surface well enough. This is a common situation with smooth input geometry, and omitting unnecessary contours generally yields significant memory savings. One way to perform the test is to check whether the distance from every point within the polyhedron to the original surface is below a fixed threshold. In practice, an efficient approximation can be obtained by considering only the vertices of the polyhedron.

3.5 Shading Attributes

Assuming an average branching factor of four, the child descriptor array takes approximately 2 bytes per voxel (taking leaf voxels into account), which is quite compact. However, raw encoding of shading attributes could easily ruin this compactness. Since the majority of rendering time is spent in ray casts which do not access the attribute data at all, it makes sense to trade attribute decoding performance for reduced memory footprint.

We employ a block-based compression scheme that spends an average of 1 byte for colors and 2 bytes for normals per voxel, yielding a total of 5 bytes including geometry. As in DXT (see e.g. [van Waveren and Castaño 2008]), each compression block is able to represent 16 values. Since voxels have 8 child slots, we assign the voxels described by two consecutive child descriptors to the same compression block to establish a direct correspondence between compressed values and child slots. Roughly half of the child slots are empty on average, resulting in 8 used and 8 unused values per block. We avoid placing values from different parts of the hierarchy into the same compression block by simply leaving gaps in the child descriptor array to ensure that we only pair descriptors having the same parent voxel.

Even though our scheme wastes approximately half of the capacity available in the compression blocks, it avoids the cost of an additional indirection table. It also has the benefit that individual values can be represented more accurately, since there is less competition within each compression block.

Colors. Voxel colors are encoded using a simplified variant of DXT1 that omits the semantics related to transparency. The first 32 bits of a compression block store two reference colors, $\mathbf{c_0}$ and $\mathbf{c_1}$, using 16-bit RGB-565 encoding. The remaining 32 bits store two-bit interpolation factors to choose each of the 16 colors from the set $\{\mathbf{c_0}, \mathbf{c_1}, \frac{2}{3}\mathbf{c_0} + \frac{1}{3}\mathbf{c_1}, \frac{1}{3}\mathbf{c_0} + \frac{2}{3}\mathbf{c_1}\}$.

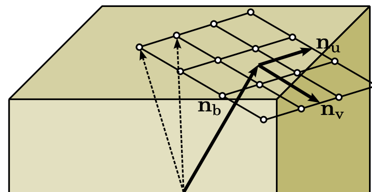

Figure 6: *Illustration of our normal compression scheme. Base normal $\mathbf{n_b}$ specifies a point on a unit cube, and two axis vectors $\mathbf{n_u}$ and $\mathbf{n_v}$ define an arbitrary 4×4 grid around this point. There are no orthogonality requirements between the three vectors, and the axis vectors are not constrained to lie on the face of the cube, allowing maximal flexibility. The dashed arrows indicate two of the 16 normals defined by this set of vectors.*

Normals. Most of the existing literature on normal compression considers only tangent-space normals (e.g. [ATI 2005; Munkberg et al. 2006; Munkberg et al. 2007]), and therefore is not applicable in our case because no tangent frame can be implicitly derived. For voxel normals, one could utilize existing normal map compression techniques such as object-space DXT5 [van Waveren and Castaño 2008]. Unfortunately, the 8-bit precision provided by such methods is insuffficent for smooth highlights and reflections. We thus employ a novel compression scheme that provides up to 14 bits of precision for smoothly varying normals.

Our normal compression scheme is based on placing a linear 4×4 grid of points in the 3-dimensional normal space and selecting each individual normal from the 16 candidates. The grid is defined using a base normal $\mathbf{n_b}$ and two axis vectors $\mathbf{n_u}$ and $\mathbf{n_v}$, as illustrated in Figure 6. Each candidate normal is defined as $\mathbf{n_b} + c_u\mathbf{n_u} + c_v\mathbf{n_v}$, with c_u and c_v selected independently from the set $\{-1, -\frac{1}{3}, \frac{1}{3}, 1\}$. For better adaptation to various kinds of data, we do not make any orthogonality requirements for $\mathbf{n_b}$, $\mathbf{n_u}$, and $\mathbf{n_v}$.

The base normal $\mathbf{n_b}$ is encoded as a point on a cube. The face is identified using 3 bits, and the coordinates on the face are stored using two fixed-point integers, one with 14 bits and one with 15 bits, totalling 32 bits. Axis vector $\mathbf{n_u}$ is stored using three signed 4-bit integers and a single 4-bit exponent, i.e. a floating-point vector with common exponent. The $\mathbf{n_v}$ axis is stored in a similar fashion, yielding a total of 32 bits for the axis vectors. The remainder of the compression block contains two u-axis bits and two v-axis bits for each individual normal specifying the values of c_u and c_v, respectively.

The compression scheme is flexible in its ability to handle different kinds of cases. If the variance of the normals within a block is small, $\mathbf{n_b}$ can be used to store the average normal with a high precision while using small exponents for $\mathbf{n_u}$ and $\mathbf{n_v}$ to minimize quantization errors. If the normals vary only in one direction, $\mathbf{n_u}$ and $\mathbf{n_v}$ can be set to have the same direction but different length in order to maximize the precision along that particular direction. Finally, if the normals are oriented in entirely different directions, one of the directions can be selected as $\mathbf{n_b}$ while using $\mathbf{n_u}$ and $\mathbf{n_v}$ to approximate two other directions.

4 Rendering

The regularity of the octree data structure is the key factor in enabling efficient ray casts. As most of the data associated with a voxel is actually stored within its parent, we need to express the

current voxel using its parent voxel *parent* and a child slot index *idx* ranging from 0 to 7. Since we do not store information about the spatial location of voxels, we also need to maintain a cube corresponding to the current voxel. The cube is expressed using position vector *pos* ranging from 0 to 1 in each dimension, and a non-negative integer *scale* that defines the extent of the cube as $\exp_2(scale - s_{max})$. The entire octree is contained within a cube of scale s_{max} positioned at the origin.

Basics. Let our ray be defined as $\mathbf{p_t}(t) = \mathbf{p} + t\mathbf{d}$. Solving t for an axis-aligned plane gives $t_x(x) = (1/d_x)x + (-p_x/d_x)$ for the x-axis, and similar formulas for the y and z axes. With precomputed ray coefficients this amounts to a single multiply-add instruction per axis. We can represent an axis-aligned cube as a pair of opposite corners (x_0, y_0, z_0) and (x_1, y_1, z_1) so that $t_x(x_0) \leq t_x(x_1)$, $t_y(y_0) \leq t_y(y_1)$, and $t_z(z_0) \leq t_z(z_1)$. Using this definition, the span of t-values intersected by the cube is given by $tc_{min} = \max(t_x(x_0), t_y(y_0), t_z(z_0))$ and $tc_{max} = \min(t_x(x_1), t_y(y_1), t_z(z_1))$.

Traversing voxels along the ray, we can determine the next voxel of the same scale by comparing $t_x(x_1)$, $t_y(y_1)$, and $t_z(z_1)$ against tc_{max} and advancing the cube position along each axis for which the values are equal. Assuming that the two voxels share the same parent, we obtain the new child slot index idx' by flipping the bits of *idx* corresponding to the same axes.

Hierarchy traversal. We will now extend the idea of incremental traversal to a hierarchy of voxels. This is necessary since our octree data structure is sparse in the sense that we do not include the subtrees corresponding to empty space. Doing the traversal in a hierarchical fashion also has the benefit of being able to improve the performance by using contours as bounding volumes of their corresponding subtrees.

Our algorithm traverses the set of voxels intersected by the ray in depth-first order. In each iteration, there are three distinct cases for selecting the next voxel:

- PUSH: Proceed to the child voxel that the ray enters first.
- ADVANCE: Proceed to the next sibling voxel.
- POP: Otherwise, proceed to the next sibling of the highest ancestor that the ray exits.

The algorithm incorporates a stack of parent voxels and contour t values associated with the ancestors of the current voxel. The depth of the stack is s_{max}, making it possible to address its entries directly using cube scale values. Whenever the algorithm descends the hierarchy by executing PUSH, it potentially stores the previous parent into the stack at *scale* based on a conservative check. When the ray exits the current parent voxel, the algorithm ascends the hierarchy by executing POP. It first uses the current position *pos* to determine the new pos', $scale'$, and idx' as described below. It then reads the stack at $scale'$ to restore the previous parent.

Determining the child voxel that the ray enters first in the case of PUSH is similar to selecting the next sibling in ADVANCE. We evaluate t_x, t_y, and t_z at the center of the voxel and compare them against tc_{min} to determine each bit of the new idx'.

To differentiate between ADVANCE and POP, we need to find out whether the ray stays within the same parent voxel. We start by assuming that it does, and compute candidate position pos^* and child slot index idx^*. We then check whether the resulting idx^* is actually valid considering the direction of the ray. As described previously, we obtain idx^* by flipping one or more bits of *idx*, each corresponding to an axis-aligned plane crossed by the ray. For idx^* to be valid, the direction of the flips must agree with sign of the corresponding component of ray direction $\mathbf{d}$. For example if $d_x > 0$, the bit corresponding to the x-axis is only allowed to increase. If all of the flips agree with the ray direction, we execute ADVANCE by using pos^* and idx^* as the new pos' and idx', respectively. If we encounter any conflicting flips, we proceed with POP.

scale		9	8	7	6	5	4	3	2	1	0
pos.x	0.	1	0	1	1	0	0	1	0	0	0
pos.y	0.	0	0	0	1	1	1	0	0	0	0
pos.z	0.	0	1	1	1	0	1	1	0	0	0
idx		1	4	5	7	2	6	5			

Figure 7: *Connection between pos and child slot indices. Each bit position of pos corresponds to a cube scale value. Interpreting the bit triplet corresponding to scale as an integer yields idx. Bits above scale define a progression of child slot indices that forms a path from the root to the current voxel. Bits below scale are zero.*

In the case of POP, we can determine the next voxel by looking at the bit representations of *pos* and pos^*. Figure 7 illustrates the connection between cube positions and child slot indices. Each triplet of bits at a given bit position forms a child slot index corresponding to a particular cube scale. Starting from the highest bit position, the child slot indices define a path in the octree from the root to the current voxel.

Let us denote the paths corresponding to *pos* and pos^* with p and p^*, respectively. We know that the traversal must have visited all the voxels along p in order to reach the current voxel, and that p^* must diverge from this set of voxels at some point along the path. The fact that a ray can never re-enter a voxel after exiting it implies that the first differing voxel in p^* is necessarily unvisited. In a depth-first traversal it is also the voxel that we should visit next.

Therefore, we determine the next voxel as follows. We first obtain the new $scale'$ by finding the highest bit that differs between *pos* and pos^*. We then find child slot index idx' by extracting the bit triplet of pos^* corresponding to $scale'$. To obtain pos', we take pos^* but clear the bits below $scale'$. This yields a cube with the correct scale that contains pos^*. Finally, we restore the parent voxel from the stack entry at $scale'$.

4.1 Ray Cast Implementation

Pseudocode for the complete ray cast algorithm is given in Figure 8. The code consists of initialization phase followed by a loop traversing each individual voxel along the ray.

The algorithm starts by initializing state variables on lines 1–7. The active span of the ray is stored as an interval between two t-values, t_{min} and t_{max}, and is initialized to the intersection of the ray with the root. h is a threshold value for t_{max} used to prevent unnecessary writes to the stack. The current voxel in the octree is identified using *parent* and child slot index *idx*. It is initialized to a child of the root by comparing t_{min} against t_x, t_y, and t_z at the center of the octree. Finally, *pos* and *scale* are initialized to represent the corresponding cube.

The loop on lines 8–39 is iterated until the ray either hits a voxel or leaves the octree. Each iteration intersects the current voxel against the active span on lines 11–16 and potentially descends to its children on lines 18–25. If the voxel is not intersected by the ray, the algorithm executes ADVANCE on lines 28–30, potentially followed by POP on lines 32–37.

Line 9 computes the span tc corresponding to the current cube to be used by INTERSECT and ADVANCE, and line 10 checks whether to process the current voxel or skip it. If the bit corresponding to the voxel in *valid mask* is not set, or the active span t is empty, the code determines that the ray cannot intersect the voxel and skips directly to ADVANCE. Otherwise, the voxel may intersect the ray and is thus processed further.

```
INITIALIZE
 1: (t_min, t_max) ← (0,1)
 2: t' ← project cube(root, ray)
 3: t ← intersect(t, t')
 4: h ← t'_max
 5: parent ← root
 6: idx ← select child(root, ray, t_min)
 7: (pos, scale) ← child cube(root, idx)
 8: while not terminated do
 9:   tc ← project cube(pos, scale, ray)
10:   if voxel exists and t_min ≤ t_max then
INTERSECT
11:     if voxel is small enough then return t_min
12:     tv ← intersect(tc, t)
13:     if voxel has a contour then
14:       t' ← project contour(pos, scale, ray)
15:       tv ← intersect(tv, t')
16:     end if
17:     if tv_min ≤ tv_max then
PUSH
18:       if voxel is a leaf then return tv_min
19:       if tc_max < h then stack[scale] ← (parent, t_max)
20:       h ← tc_max
21:       parent ← find child descriptor(parent, idx)
22:       idx ← select child(pos, scale, ray, tv_min)
23:       t ← tv
24:       (pos, scale) ← child cube(pos, scale, idx)
25:       continue
26:     end if
27:   end if
ADVANCE
28:   oldpos ← pos
29:   (pos, idx) ← step along ray(pos, scale, ray)
30:   t_min ← tc_max
31:   if idx update disagrees with ray then
POP
32:     scale ← highest differing bit(pos, oldpos)
33:     if scale ≥ s_max then return miss
34:     (parent, t_max) ← stack[scale]
35:     pos ← round position(pos, scale)
36:     idx ← extract child slot index(pos, scale)
37:     h ← 0
38:   end if
39: end while
```

Figure 8: *Pseudocode for the ray cast algorithm.*

Line 11 checks whether the voxel is small enough to justify termination of the traversal. This provides a way to pre-filter the geometry by dynamically adapting voxel resolution to match the screen resolution, and is accomplished by comparing $\exp_2(scale)$ against a linear function of tc_{max}. The check can be executed before it is known for sure whether the voxel actually intersects the ray, since the exactness of the result is not relevant for very small voxels.

Lines 12–16 compute the span tv as the intersection of the current cube with the active span and voxel contour. The effect of the contours corresponding to the ancestor voxels is included in the active span, so tv represents the exact intersection with the geometric shape of the current voxel. Line 17 checks whether the intersection is non-empty, and if so, proceeds to execute PUSH. Otherwise, the voxel is skipped by executing ADVANCE.

If the current voxel is a leaf, as seen from the *leaf mask* of *parent*, line 18 terminates the traversal because the desired intersection has been found. Line 19 stores the old values of *parent* and t_{max} to the stack if necessary. The decision is based on the limit h as follows:

- Normally, h corresponds to the t value at which the ray exits *parent*. $tc_{max} = h$ means that the ray exits both the voxel and its parent at the same time, in which case we do not need to store *parent* as it will not be accessed again.
- If *parent* has already been stored to the stack, we set h to 0. As $tc_{max} \geq 0$ is always true, this has the effect of preventing the same parent from being stored again.

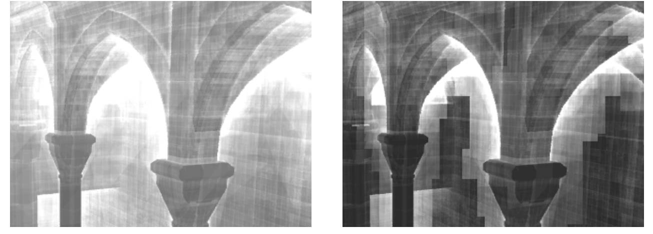

Figure 9: *Illustration of the effect of beam optimization on iteration counts. Left:* SIBENIK-D *with no beam optimization. Right: with beam optimization in 8×8 blocks. White corresponds to 64 iterations in both images.*

Descending the hierarchy, lines 20–24 replace *parent* with the current voxel and set *idx*, *pos*, and *scale* to match the first child voxel that the ray enters. Finally, line 25 restarts the loop to process the child voxel.

Lines 28–30 execute ADVANCE. The current cube position is first stored into a temporary variable. *pos* and *idx* are then advanced to the next cube of the same scale along the ray. All t values required for deciding the axes to advance along have already been computed on line 9, and can be reused here. Finally, the active span of the ray is shortened by replacing t_{min} with the value at which the ray enters the new cube. Line 31 checks whether the child index bit flips agree with the direction of the ray, i.e. whether the traversal stays in the same parent voxel. If so, the loop restarts. Otherwise, the algorithm proceeds to execute POP.

Lines 32–36 execute POP as described previously. If the new *scale* exceeds s_{max}, line 33 determines that the ray exits the octree and terminates the traversal with a miss. Finally, line 37 sets h to 0 to prevent the parent that was just read from the stack from being stored again.

Alternate coordinate system. The algorithm contains multiple operations that have to explicitly check the signs of the ray direction. These checks can be avoided by mirroring the entire octree to redefine the coordinate system so that each component of $\mathbf{d}$ becomes negative. In practice, we accomplish this by determining an octant bitmask based on the ray direction during initialization. We then use this mask to flip the bits of *idx* whenever we interpret the fields of a child descriptor. In the same vein, we can also offset the origin of *pos* to make its components range within $[1,2]$ instead of $[0,1]$. This has the benefit of enabling us to operate directly on the corresponding floating point bit representations in POP.

4.2 Beam Optimization

There is a relatively simple way to accelerate the ray casting process for primary rays. With cubical voxels, it is possible to render a coarse, conservative distance image and then use it to adjust the starting positions of individual rays. This has the effect of making the individual rays skip majority of the empty space at their beginning before intersecting a surface.

For many acceleration structures this kind of approach is not feasible, because it is generally impossible to guarantee that the coarse grid of rays does not miss features that would be important for the individual rays. However, with voxel data we can make this guarantee by terminating the traversal as soon as we encounter a voxel that is not large enough to certainly cover at least one ray in the coarse grid. Note that contour tests must be disabled in the coarse pass in order for this to work.

In practice, we divide the image into 4×4 or 8×8 pixel blocks and cast a distance ray for the corners of these blocks in the coarse pass.

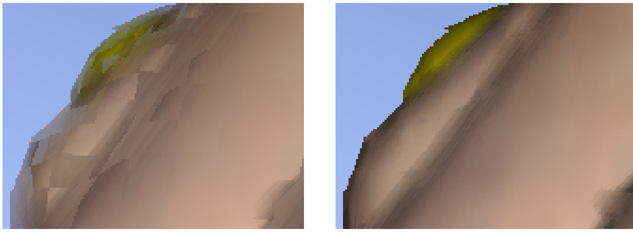

Figure 10: *The problem with varying blur filter radii. The image shows a closeup of* FAIRY*'s hand built with aggressive pruning and few voxel levels. Left: Filtering each pixel with a radius deduced from the size of the corresponding voxel. Seams are visible at hierarchy level changes. Right: Our method adjusts the filter radius while accumulating neighboring colors, yielding smooth transition between levels.*

For each ray in the actual rendering pass, we identify the corresponding block and fetch the distance values for the four corners. We then subtract an appropriate constant from their minimum to determine the starting point of the ray. Figure 9 illustrates the effect of beam optimization on iteration counts.

4.3 Post-process Filtering

To smooth out the blockiness caused by discrete sampling of shading attributes, we apply an adaptive blur filter on the rendered image as a post-processing step. Without filtering, the result would resemble the effect of nearest-sampled texture lookups. Note that silhouette edges are generally represented well by contours, so we want to avoid excess blurring around them.

The most reliable way to estimate the proper filter radius is to look at the size of the intersected voxel on the screen. However, adjacent voxels on the same surface may reside on different hierarchy levels due to the fact that the voxel resolution is allowed to vary depending on the local geometrical complexity and variance in shading attributes. As a result, the desired filter radius also varies across the surface. This can cause rendering artifacts due to abrupt changes in the filter radius at voxel boundaries, as illustrated in Figure 10.

Our method is based on a sparse set of sampling points, stored in a look-up table in ascending order according to distance from the center. We use a set of 96 samples distributed in a disc with radius of 24 pixels. The density of the samples falls as the square root of distance from the center, and each sample has an associated weight corresponding to the area of the disc it represents. Algorithms that smooth out undersampled data such as single-sample shadows or reflections tend to require two passes (e.g. [Fernando 2005; Robison and Shirley 2009]), but the ordered look-up table allows us to perform the computation in a single pass.

Pseudocode of the algorithm is given in Figure 11. To process a pixel, we start by determining the desired filter radius r based on the voxel in the pixel itself. If the radius is one pixel or less, there is no need for filtering, and we return the original color. Otherwise, we start processing sampling points in the order determined by the look-up table until their distance from the center exceeds r. For each sample, color c' and blur radius r' of the corresponding pixel are fetched. To adapt blur radius to the neighborhood, we clamp r to $\min(r, r')$. This prevents visible seams from forming by making filter radius agree between nearby regions. Accumulation weight is calculated by taking sample weight and adjusting it so that it tapers off to zero linearly between $r-1$ and r. This ensures smooth transition between different filter radii. Finally, color is accumulated according to the computed weight.

```
1: (c,r) ← fetch(x,y)
2: if r ≤ 1 then return c
3: accum ← (0,0,0,0)
4: for each sample s in kernel do
5:   (c',r') ← fetch(x+s.x, y+s.y)
6:   r ← min(r,r')
7:   if s.dist > r then break
8:   w ← s.weight · min(r − s.dist, 1)
9:   accum.rgb ← accum.rgb + c' · w
10:  accum.w ← accum.w + w
11: end for
12: return accum.rgb/accum.w
```

Figure 11: *Pseudocode for the post-process filtering algorithm.*

In practice, we store the logarithm of the voxel size into the alpha channel of the result image when casting the rays. One byte is sufficient when the value is stored as 3.5 fixed point, yielding range from 1 (no blur) to about 128 pixels.

5 Results

All tests were performed on an NVIDIA Quadro FX 5800 with 4 GB of RAM installed in a PC with 2.5 GHz Q9300 Intel Core2 Quad CPU and 4 GB of RAM. The operating system was 64-bit edition of Windows XP Professional. The public CUDA 2.1 driver and compiler was used. Figure 12 shows the test scenes used in this paper along with their triangle counts. Due to general lack of large-scale high-resolution voxel datasets, all of our voxel datasets were built from triangle meshes.

Table 1 lists the memory usage and construction time of the voxel representation for each test scene with a varying limit on the number of octree levels. The rightmost column in the table shows the average number of bytes consumed by a single voxel in the highest-resolution representation, including all overhead. We can see that the actual values are mostly near the theoretical optimum of 5 bytes per voxel (Section 3.5). The differences are explained by variance in branching factor, gaps in child descriptor array, and variance in the amount of contour data.

5.1 Rendering Performance

Arguably the most interesting piece of information is the rendering performance using voxel data. In our case, the dominant factor is the efficiency of ray casts, as shading costs are negligible and post-process filtering is one to two magnitudes faster than the ray casting. Table 2 summarizes the ray cast performance in our test scenes at various resolutions. The values in the table have been measured as averages over several viewpoints, repeating each frame several times to amortize startup and flush delays. The increase in performance as the resolution grows is therefore explained solely by better ray coherence.

The *triangle caster* column refers to the fastest GPU ray caster described in our previous paper [Aila and Laine 2009]. The *cubical voxels* column shows voxel ray cast performance with cubical voxel data, while the *contours* column shows the results for voxel data that includes contours. It should be noted that the two datasets are different in terms of their average depth, as the improved approximation provided by contours makes it possible to prune the hierarchy more aggressively. Finally, the last column shows the result with the beam optimization enabled (Section 4.2). It can be seen that the voxel ray caster consistently outperforms the triangle ray caster in the test scenes.

Obviously, the comparison between triangle and voxel ray cast performance is between apples and oranges because of the different type of data we are casting against. The triangle-based representation is able to discern every edge and corner perfectly, whereas the voxel representation may be inaccurate in such places. On the other hand, the voxel representation contains unique, i.e. non-repetitive, color and normal information on a per-sample basis, and allows representing unique high-resolution geometry.

CITY, 879K tris — SIBENIK-D, 77K tris + disp. — HAIRBALL, 2.88M tris — FAIRY, 93K tris — CONFERENCE, 283K tris

Figure 12: *Test scenes used in measurements.* SIBENIK *is the same as* SIBENIK-D *but with flat triangles.*

Scene	10	11	12	13	14	15	16	bytes
CITY	13	39	131	432	1368	–	–	5.44
	14	*23*	*48*	*115*	*311*			
SIBENIK	41	141	440	1034	1857	–	–	5.68
	10	*31*	*90*	*194*	*336*			
SIBENIK-D	79	314	1192	2806	–	–	–	8.10
	70	*274*	*1342*	*2541*				
HAIRBALL	442	1552	–	–	–	–	–	7.48
	294	*628*						
FAIRY	11	35	99	239	376	639	1109	5.62
	6	*15*	*38*	*86*	*121*	*178*	*282*	
CONFERENCE	17	40	96	220	512	1328	–	5.16
	7	*13*	*24*	*51*	*115*	*288*		

Table 1: *GPU memory usage and construction times of the test scenes with different voxel level counts. All four CPU cores were utilized. For each scene, the upper row denotes memory consumption in MB and the lower row shows the construction time in seconds. The* bytes *column on the right tells the average memory consumption per voxel for the largest datasets.*

Scene	resolution	triangle caster *(Mrays/s)*	cubical voxels	contours *(Mrays/s)*	cont. w/beam
CITY	512×384	46.7	45.1	79.9	88.6
	1024×768	68.5	54.3	89.1	106.0
	2048×1536	77.1	63.9	97.4	123.8
SIBENIK	512×384	64.3	38.7	80.0	82.5
	1024×768	94.1	46.5	94.1	103.6
	2048×1536	107.1	55.1	103.9	122.0
SIBENIK-D	512×384	–	24.8	32.6	37.5
	1024×768	–	30.2	38.7	48.3
	2048×1536	–	37.1	43.6	60.9
HAIRBALL	512×384	11.6	22.4	24.1	24.1
	1024×768	20.5	22.5	27.9	28.4
	2048×1536	31.2	29.2	36.5	38.2
FAIRY	512×384	63.9	62.1	128.2	132.6
	1024×768	125.1	69.4	145.4	150.9
	2048×1536	155.8	78.6	160.4	169.2
CONFERENCE	512×384	69.1	35.8	97.9	104.4
	1024×768	111.9	43.8	110.3	124.3
	2048×1536	134.0	52.3	120.2	140.8

Table 2: *Ray cast performance for primary rays at various screen resolutions. Values are in millions of rays per second. The largest datasets that could be fit in 4 GB were used in the voxel tests.*

6 Future Work

We would like to experiment with truly volumetric effects such as fog or partially transparent materials. Our data structure is readily able to represent them, and we assume that it would be reasonably efficient to e.g. accumulate extinction coefficients or collect illumination during ray casts.

While the proposed data structure is demonstrably efficient for rendering purposes, it still requires a fair amount of storage. The memory capacity available in GPUs today is adequate for rendering purposes, but storing large amounts of high-resolution content on an optical disk or streaming it over network seems impossible without some form of compression. Finding efficient—presumably lossy—compression algorithms would make voxel-based content more feasible for practical applications.

In our benchmarks we load the entire scenes in full resolution into GPU memory, while only a small portion would be required for rendering any single image due to occlusions and resolution requirements falling with distance. Our system already supports on-demand streaming based on distance to camera, but it would be interesting to see how much visibility-based streaming (in spirit of Crassin et al. [2009]) would further reduce the memory footprint.

Acknowledgments. We thank Timo Aila and David Luebke for discussions and helpful suggestions. Sibenik model courtesy of Marko Dabrovic. Fairy model courtesy of University of Utah.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics 2009*, 145–149.

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *In Eurographics 87*, 3–10.

ATI. 2005. Radeon X800: 3Dc white paper. `http://www.ati.com/products/radeonx800/3DcWhitePaper.pdf` .

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proc. I3D '09*, 15–22.

DICK, C., KRÜGER, J., AND WESTERMANN, R. 2009. GPU ray-casting for scalable terrain rendering. In *Proc. Eurographics 2009–Areas Papers*, 43–50.

FERNANDO, R. 2005. Percentage-closer soft shadows. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, ACM, New York, NY, USA, 35.

JU, T., LOSASSO, F., SCHAEFER, S., AND WARREN, J. 2002. Dual contouring of hermite data. In *Proc. SIGGRAPH '02*, 339–346.

KNOLL, A., WALD, I., PARKER, S. G., AND HANSEN, C. D. 2006. Interactive Isosurface Ray Tracing of Large Octree Volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 115–124.

KNOLL, A. M., WALD, I., AND HANSEN, C. D. 2009. Coherent multiresolution isosurface ray tracing. *Vis. Comput. 25*, 3, 209–225.

MUNKBERG, J., AKENINE-MÖLLER, T., AND STRÖM, J. 2006. High quality normal map compression. In *Proc. Graphics Hardware 2006*, 95–102.

MUNKBERG, J., OLSSON, O., STRÖM, J., AND AKENINE-MÖLLER, T. 2007. Tight frame normal map compression. In *Proc. Graphics Hardware 2007*, 37–40.

PETERS, J., AND WU, X. 2004. Sleves for planar spline curves. *Computer Aided Geometric Design 21*, 6, 615–635.

ROBISON, A., AND SHIRLEY, P. 2009. Image space gathering. In *Proc. High Performance Graphics 2009*, 91–98.

SZIRMAY-KALOS, L., AND UMENHOFFER, T. 2008. Displacement mapping on the GPU - State of the Art. *Computer Graphics Forum 27*, 1.

VAN WAVEREN, J. M. P., AND CASTAÑO, I. 2008. Real-time normal map DXT compression. `http://developer.nvidia.com/object/real-time-normal-map-dxt-compression.html`.