Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation

Samuli Laine Tero Karras
NVIDIA Research*

Abstract

This technical report extends our previous paper on sparse voxel octrees. We first discuss the benefits and drawbacks of voxel representations and how the storage space requirements behave for different kinds of content. Then, we explain in detail our compact data structure for storing voxels and an efficient ray cast algorithm that utilizes this structure, including the contributions of the original paper: additional voxel contour information, normal compression format for storing high-precision object-space normals, post-process filtering technique for smoothing out blockiness of shading, and beam optimization for accelerating ray casts.

Management of voxel data in memory and on disk is covered in more detail, as well as the construction of voxel hierarchy. We extend the results section considerably, providing detailed statistics of our test cases. Finally, we discuss the technological barriers and problems that would need to be overcome before voxels could be widely adopted as a generic content format.

Our voxel codebase is open sourced and available at
http://code.google.com/p/efficient-sparse-voxel-octrees/

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. Contrary to common belief, there is no fundamental requirement that voxel data would need to be volumetric.

A compelling reason for using triangles has been their compactness for representing planar surfaces. Only a handful of triangles are required for representing flat-sided objects such as buildings, no matter how large they are spatially. 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.

*e-mail: {slaine,tkarras}@nvidia.com

This technical report is revised and extended from ”Efficient Sparse Voxel Octrees” published in the proceedings of I3D 2010 [Laine and Karras 2010].

NVIDIA Technical Report NVR-2010-001, February 2010.
© NVIDIA Corporation. All rights reserved.

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?

Data resolutions. The goal of streaming data is to achieve a constant texel-per-pixel ratio so that enough resolution will always be available for image synthesis. To facilitate discussion, we will call this the *target resolution* of the data. It may be different from the resolution available on the storage medium (*storage resolution*), and it may vary across the data. For example, target resolution of terrain is smaller in the distance than near the camera. If the storage resolution is less than target resolution, the synthesized image will be coarser than desired. In this case, measures must be taken to avoid the blockiness of voxels from becoming visible.

No deformation. Deformation is probably the most important benefit of separating the base mesh from fine detail. Deforming the base mesh is usually easy, and ideally the detail layer follows it naturally. If this is not possible, other deformation methods need to be used, none of which is quite as intuitive and powerful as vertex-based deformation. Therefore, it seems that currently voxels would be feasible only for static parts of the scene. Possible methods for deforming voxel-based geometry are discussed briefly in Section 3.4, but in this paper we will restrict our scope to non-deforming geometry only.

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], where a principal axis is chosen and a slice of voxels, corresponding to the beam of rays, is propagated within the volume. As with other packet traversal algorithms, the benefit ob-

tained on a CPU-based implementation seems unlikely to translate readily to massively parallel GPUs.

Gobbetti et al. [2005] discuss a framework for rendering massive voxel datasets on graphics hardware. Their rendering utilizes the graphics pipeline by drawing antialiased point primitives, and hence does not support ray tracing. However, their system is notable for employing a generic shading method that is based on sampling the original data from various directions. They then choose, on a per-voxel basis, a shader that can best express the observed results and compute the shading parameters based on the observations. A method like this could be very useful in real-world applications, because no manual tuning is required and shading of different LOD levels is handled automatically.

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 or fuzzy 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. While traversing the octree, 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.

Our approach is different from Crassin et al. in several ways. We aim for storing representations of large-scale scenes in GPU memory with enough detail for high-quality rendering, which necessitates a small memory footprint. Also, our primary interest is in representing surfaces instead of volumes, because surfaces arguably play a more significant role in most real-world content. Because of these considerations, we use of a single hierarchical structure instead of separate schemes for coarse and fine data. We also introduce additional contour data to allow accurate surface placement within individual voxels.

3 Triangles vs. Voxels

In this section, we consider the main differences and similarities between triangle- and voxel-based geometry representations. For triangle-based representation, we assume a (coarse) base mesh equipped with unique color textures as well as displacement maps for achieving geometric detail. In this way, both representations are more or less equally powerful for representing static, highly detailed objects. There are major differences in other aspects, though, as we will see in a moment.

3.1 Memory Usage

Voxels are generally assumed to consume huge amounts of memory. This is true if the data is truly volumetric 3D such as an MRI scan, but if we only encode surfaces, the memory usage drops to $O(N^2)$ where N is the resolution along one spatial dimension. The resolution could be defined, for example, as the maximum number of distinguishable features per unit length. In terms of complexity, this is the same as with triangles, where the total surface area is also the factor that most affects memory usage, assuming unique texturing.

Triangles. Let us assume that the base meshes are coarse enough so that we can ignore them altogether. What remains is the color and displacement detail maps. Assuming DX texture compression, colors can be encoded using 4 bits per sample, and for three-axis displacement 8 bits per sample might be sufficient using e.g. DXT5 compression [van Waveren and Castaño 2008], totalling 1.5 bytes per sample. Normals are not required, because it should be possible to derive them from the displacement map.

	triangles	**voxels** with contours
geometry	1	1
color	0.5	1
normals	0	2
contours	–	1
total	1.5	5

Table 1: *Memory usage per voxel for uniquely textured, displaced triangles, and voxels. All numbers are in bytes.*

Voxels. A comprehensive analysis of voxel memory consumption is presented later in Section 5.6. In total, voxels with colors, normals and contours (Section 5.2) require approximately 5 bytes per voxel. The memory consumption is therefore 3.33 times as high as with triangle-based representation. The relevant values are summarized in Table 1. If we consider memory usage per leaf voxel, i.e. target-resolution sample, we need to multiply per-voxel cost by $\frac{4}{3}$, yielding 6.67 bytes per sample. For triangles, the same multiplier is required for taking mipmaps into account, which yields 2 bytes per highest-resolution sample.

It is important to note that texture resolution cannot be controlled on a fine-grained basis, but voxel resolution can. For instance, if a color texture contains a large patch of constant color, textures cannot easily take advantage of this, but with hierarchical voxel encoding it is possible to simply leave out further detail levels where they would not make any difference. The same applies to displacement maps, and therefore to geometric detail as well. In this sense, direct comparison between textures and voxels is somewhat crude, as the nature of the content determines how much voxels can benefit from local resolution fine-tuning.

To estimate how much data one can fit in e.g. 4GB video memory available on NVIDIA Quardo FX 5800 board, let us assume 1 mm $\times$ 1 mm constant resolution for geometry. This gives us one sample per pixel when content is viewed not closer than approximately one meter away using 90° FOV in 1080p display resolution (1920 $\times$ 1080). For triangles, we could fit $2^{32}/2 = 2^{31}$ samples into GPU memory, giving $2^{31}/10^6 = 2147$ sq.m ($\approx$ 23000 sq.ft). For voxels, the same calculation gives 644 sq.m ($\approx$ 6900 sq.ft).

Note that this estimation is inaccurate because it does not account for diminishing resolution requirements as distance to camera grows. In the scenario above, assuming that one sample per pixel is enough, the full resolution would only be needed in a sphere with radius of two meters, and after this the memory requirements would start to drop drastically. This effect will be analyzed in detail in Section 4.

3.2 Downsampling

Downsampling of triangle meshes is fairly straightforward, and individual displacement and color maps can also be trivially downsampled. A major remaining problem—that is to our knowledge not properly solved yet—is how to make high-frequency geometric occlusion (e.g. fur, grass, dense grating) transform into partial transparency when downsampled. There are also situations where triangle meshes cannot represent downsampled data appropriately,

e.g. when fur should turn into partially transparent volumetric mass after enough downsampling.

Downsampling voxel data is theoretically very simple. The appearance of a parent voxel needs to match the combined appearance of its child voxels with occlusion and transparency taken into account. This is much easier to achieve than generic triangle-based level of detail, and volumetric transparency can also be easily represented. During rendering, the level of detail can be efficiently decided on the fly. For example, our ray caster always uses a level of detail that is appropriate for the distance that the ray has traversed.

However, there are practical difficulties in downsampling voxel data. Firstly, partial binary occlusion in finer detail should translate to transparency in coarser detail level. We have not yet attempted to incorporate this into our builder, so there could be unforeseen difficulties along the way. Also, some detail is invariably lost when downsampling the data. For example, if child voxels have highly specular BRDFs and different normals, the parent's BRDF should have multiple spikes according to the child voxels' normals. It would not be practical to have BRDFs with variable amount of detail, so an optimization problem arises: how to set the parent BRDF (or more generally, shading model) parameters so that they mimic the combined BRDFs of the child voxels as well as possible.

Our builder computes colors and normals by spatially filtering input data so that filter size is chosen according to voxel scale. Handling each parameter separately works well in most cases, but obviously there are situations that are not properly handled. An example is a highly specular surface with very high-frequency displacement. When such a surface is downsampled, the high-frequency bumpiness is lost. The proper way to compensate for the lost frequencies would be to increase the surface roughness of the BRDF. Considering downsampling, another limitation of our current builder is that it does not work bottom-up, i.e. parent data is not derived from child data. This would result in better matching between LOD levels.

It should be noted that exactly the same difficulties arise when using mipmapped normal maps and surfaces with varying BRDF parameters, both common in games today. We expect that solutions that are currently in use for handling these issues can be accommodated to handle downsampling voxel data as well.

3.3 Zooming

A common argument against voxels is that they become blocky when viewed up close. This is true in the same sense that bitmap images become blocky when zoomed enough, and the same holds for textures in traditional triangle-based graphics.

The loss of color, normal, etc. detail in surfaces can be battled by filtering the data. In triangle-based graphics this is done at texture lookup stage by performing interpolated fetches. Similar interpolation is possible for voxels as well, but the cost is higher because the data is not organized into a 2D array that can be easily addressed. We have found that post-process filtering is a better alternative, as discussed in Section 6.3. In any case, achieving smoothly textured surfaces is equally possible with voxels as with textured triangles.

A more severe problem is the blockiness of silhouettes that is not equally easily solved by means of blurring. Our experiences with this are briefly described in Section 6.4. However, the additional contour data, explained in detail in Section 5.2, reduces the blockiness of silhouettes to a great extent and also has other benefits such as lower memory usage and faster rendering. This is our preferred solution for fixing the silhouettes.

3.4 Deformation

Triangle meshes are easy to animate by applying transformations to the vertices and allowing the rest of the surface to follow. Blending multiple transformations by interpolating vertex positions is the de facto method for animating triangle meshes today.

Animating voxel data is considerably different from animating triangle data. A trivial method for animating voxel data would be having separate datasets for each animation frame. As a brute-force solution this is close to a video sequence that consists of a series of individual bitmap frames. The amount of raw data may be overwhelming, but compression methods similar to what is being used in video compression might bring the datasets to tolerable sizes. A major downside is that only pre-recorded animations can be played. All in all, this is not a particularly attractive solution.

There is an increasing amount of research targeted at more generic deformation of spatial data. The model is usually enclosed in a hull that is deformed so that the object follows as naturally as possible while taking constraints such as volume conservation into account. Rendering the deformed dataset is a concern. For ray tracing, there would need to be inverse mapping for the final deformation function so that the rays can be bent while the model remains in a reference pose.

The main problem with deforming voxel data is that voxels cannot be easily moved from one place to another. Because of this, another sample-based representation, *surface splats* [Zwicker et al. 2001; Weyrich et al. 2007], might be better suited for deformable objects. Splats can be easily organized into a standard BVH tree that can be updated in linear time when splats are moved according to deformation. The required BVH size also grows linearly with respect to number of splats, so even though the memory usage is higher than for voxels, it is only a constant multiplier away.

3.5 Authoring

There is a well established content pipeline for triangle-based 3D modeling. Automated tools can be used to generate UV atlases for textures, and artists can freely paint on the models themselves without modifying the textures directly.

A different methodology is usually employed when the content needs to be geometrically detailed. In this case, the content is typically generated using sculpting software such as ZBrush or Mudbox. ZBrush utilizes voxel-like data structures whereas Mudbox works with meshes with locally adjustable resolution. Both tools allow the artist to modify the geometry using sculpting tools and to paint textures on the model. The finished model can then be reduced into low-polygon base mesh detailed with normal or displacement maps.

Considering voxels, the sculpting-like interface seems to be currently the only method for authoring highly detailed data directly. Conversion from triangle meshes is another approach, and the only one we so far support in our system. Unfortunately, converting triangle meshes into voxels has the drawback of combining the limitations of both paradigms. Unless the source data is extremely detailed, voxels are not used to their full potential.

More generally, authoring content with extremely high geometric detail seems to be in its early stages. Digital sculpting, acquisition using laser scanning devices, procedural generation, and several ad hoc methods for e.g. scanning faces are currently in use. Of these, only the sculpting-to-mesh pipeline can be considered mature enough for wide adoption. It remains to be seen how the field develops when real-time rendering of highly detailed geometry becomes possible.

3.6 Acquisition

Acquiring real-world data using cameras or scanning devices is where voxels and other sample-based representations excel. It is difficult to imagine a device that would produce something other than discrete data points over the 2D or 3D domain being scanned. Point clouds can certainly be converted to detailed, colored triangle meshes, which can in turn be converted to a base mesh and a displacement map. However, for structures that are badly suited for such representation, such as thin branches or undersampled data, it can be very difficult to perform either stage of the conversion.

Such conversion chain is unnecessary if we can directly translate the raw input data into splat or voxel data. If we have e.g. a badly undersampled blade of grass that has only a handful of samples that do not form a continuous surface, it is not a problem for sample-based representation (as long as the object is not viewed from too close). We could still render the samples we have, and there would be little risk of catastrophic failures.

3.7 Rendering

Displacement-mapped triangles can be directly rasterized using GPU hardware tessellation. This keeps the rendering process similar to current triangle-based graphics. Ray casting such meshes is, however, much more difficult. The power of displacement maps lies in their ability to compactly encode the equivalent of millions of triangles, and therefore it would be unfeasible to "decompress" them into individual triangles and place them into an acceleration hierarchy for efficient ray casting. In practice, the ray caster would therefore need to consider the two detail levels separately, first intersecting the rays against bounding volumes of displaced geometry, and then against the displacement maps themselves.

Voxels can also be rasterized directly, but as explained in Section 6.4, this does not seem to be as efficient as ray casting. On the other hand, ray casting a voxel hierarchy is very simple and efficient thanks to the regular structure of an octree. Very little arithmetic has to be carried out during ray casts, except for the contour evaluation which requires intersecting the ray against a pair of parallel planes. Benchmark results from our implementation can be found in Section 8. As mentioned earlier in Section 3.2, rays cast into voxel structure can work on appropriate level of detail according to distance from ray origin. This reduces the amount of computation and memory bandwidth usage, and simultaneously decreases aliasing caused by high-frequency content.

3.8 Summary

Considering memory usage, triangles and displacement maps are more compact than voxels, but only by a multiplier of 3.33. Taking into account that voxel representation can adapt to data resolution on much finer scale than displacement maps, the difference is smaller in practice. In Section 5.7, we also propose additional methods for decreasing memory usage of voxels.

The downsampling of voxels is attractive because a single algorithm should be able to handle all possible content, properly transforming dense partially occluding geometry into volumetric transparency. Also, combining shading models of eight child voxels into parent voxel is probably more straightforward than doing the same for triangles. Voxels have the ability to represent downsampled dense opaque data, e.g. fur, correctly as partially transparent, volumetric "cloud". This is something that displacement maps cannot do.

Triangle data handles zooming better than voxels, and parametric patches handle it even better. The blockiness of voxel data can usually be removed to great extent, but sharp features cannot always be represented well without increasing the resolution.

Generic deformation of voxel data is, with current knowledge, impossible to do efficiently, whereas triangles have no problems with it. However, fracturing voxel data, i.e. separating chunks of it, is easy, and this could be enough for e.g. simulating destructible walls and buildings.

Authoring is an open question for both triangles and voxels. While manual triangle modeling is the de facto method of content creation today, the transition to extremely detailed geometry is non-trivial. The key issues are the kind of data that can be represented, and the set of tools required by artists. The winning data structure is the one that fulfills the real-world requirements best.

The acquisition pipeline from scanned samples to voxels appears simpler than conversion of raw data into triangles and displacement maps. Especially the handling of imperfect data should be more easily achieved in samples-to-samples conversion.

Finally, rendering of triangle data enjoys the benefits of immensely fast rasterization. On the other hand, ray casting geometrically detailed surfaces is probably more efficient with voxels than with triangles, as our results suggest (Section 8). Voxels would therefore seem more suitable for rendering with advanced illumination effects such as reflections and path tracing.

4 Dimensional Analysis of Memory Consumption

When rendering images from voxel data, it is desirable to keep in GPU memory high resolution data near the camera, and lower resolution data for things that are far away. While this is clear, it is not obvious how much data is needed for obtaining a given viewing distance, or how the memory requirements vary according to scene resolution, image resolution, and viewing distance. In this section, we will provide a theoretical framework for analyzing these issues.

Let us consider the contents of the world around a cube centered at an arbitrary point and having side length of 2^l. Suppose we are inspecting voxels, i.e. non-empty cubical pieces of space, of scale s, meaning that each voxel cube has side length of 2^s. Neglecting the position of the cube, we define *voxel count* function $N(l,s)$ to give the number of scale-s voxels in a scale-l cube. Because we only consider non-empty voxels, this is generally much less than the upper bound of $2^{3(l-s)}$.

To examine how the number of voxels changes when changing voxel scale s or cube size l, we define two auxiliary functions. Let us first consider how the number of voxels changes when voxel scale is doubled. This yields the *local dimensionality function* D_{local} defined as:

$$D_{local}(l,s) = \log_2 \frac{N(l,s)}{N(l,s+1)},$$

The logarithm converts the ratio of voxels into a dimensionality, i.e. 3 if the number of voxels increases eight-fold, 2 if it increases four-fold, and so on. It should be noted that the dimensionalities are not necessarily integer numbers but may assume any real in range $]0,3]$.

The local dimensionality tells how the data behaves when its resolution is increased or decreased. Therefore, a planar surface would have local dimensionality of 2, and a volumetric chunk would have a local dimensionality of 3. If most of the content we are representing consists of surfaces, we can expect the local dimensionality to

hover around 2. Figure 24 shows the measured dimensionalities of our test scenes on different scales.

Let us now define the *global dimensionality function* which tells how the number of voxels changes when cube size l is doubled, i.e. l is increased by 1. This is defined as

$$D_{global}(l, s) = \log_2 \frac{N(l+1, s)}{N(l, s)}.$$

The global dimensionality tells how expanding our observed region affects the amount of data. For instance, in a world where the content is located above a ground plane, the global dimensionality would be about 2 for scales that encompass the entire vertical range of the data. This holds even if there is e.g. a layer of volumetric fog above the ground—doubling the radius of our observed universe does not increase the amount of data eight-fold, which would be the local dimensionality of the fog data, but only four-fold.

The combined effect of doubling the cube size and doubling the voxel scale can now be expressed as a combination of local and global scale changes. We call this the *scale dimensionality* because it tells how the amount of data changes along with viewing scale.

$$\begin{aligned} D_{scale}(l, s) &= \log_2 \frac{N(l+1, s+1)}{N(l, s)} \\ &= D_{global}(l, s) - D_{local}(l+1, s). \end{aligned}$$

Decomposing the dimensionality of the scaling into local and global parts is useful, because these parts are governed by different features of the voxel content. The local dimensionality depends on the shapes of small neighborhoods, whereas the global dimensionality depends on large-scale features of the scene.

The rationale for calculating the combined effect of the two scales is as follows. Assume that we have some part of the world in memory with resolution that is sufficient for high-quality rendering. Now assume that we want to double the radius (or side) of the area we can see, i.e. double the viewing distance. If we kept the resolution the same, we would require $2^{D_{global}}$ as much data, but because the new content is on the average twice as far as the data we originally had, we can decrease the resolution according to $2^{D_{local}}$ for that part.

We can see that especially the case where local dimensionality is lower than global dimensionality causes the information content to grow rapidly as a function of the view distance. Vegetation is a good example of this kind of data, because leaves and grass blades are two- or perhaps only one-dimensional, but they (more or less) fill the entire space until a certain scale is reached. On the other hand, a layer of fog behaves much more nicely, because the resolution requirement drops eight-fold when distance grows, but only four times as much content is contained in the larger region. Therefore, doubling the viewing distance gets cheaper as the distance grows.

Assuming constant scale dimensionality D, we can obtain the complexity of the total amount of data required to obtain view distance of r by taking the integral of x^D up to $\log_2(r)$, i.e.

$$\int_0^{\log_2 r} x^D \, \mathrm{d}x = \begin{cases} O((\log r)^{D+1}) & , D > -1 \\ O(\log \log r) & , D = -1 \\ O(-(\log r)^{D+1}) & , D < -1 \end{cases}$$

We can see that the memory usage complexity always grows sublinearly with respect to radius. The usual situation with $D_{scale} = 0$ yields logarithmic complexity with respect to view radius, whereas when $D_{global} > D_{local}$, the complexity is logarithm raised to a power.

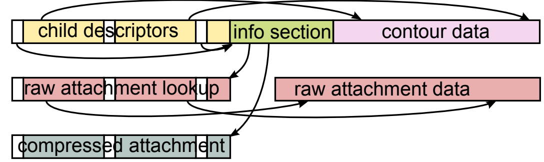


Figure 1: *Single block of octree data. The child descriptor and attachment arrays are addressed with the same index. The gaps in the arrays are due to page headers and far pointers. Page headers are placed at every 8 kilobyte boundary, and point to the info section.*

The effect of increasing display resolution is simpler to analyze. If the display resolution is multiplied by two, this means that D_{local} and D_{global} will be evaluated with scale parameter $s-1$ instead of s. This is equivalent to applying scaling by $2^{D_{local}}$. Therefore, it is only the local dimensionality (in appropriate scales due to distance) that affects the memory usage when display resolution is varied. Locally low-dimensional data such as surface fragments and hair thus behaves more nicely than high-dimensional volumetric data.

5 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 correspond to localized portions of voxels in the tree, storing the octree topology along with voxel geometry and shading attributes. All memory references within a block are relative, making it easy to reorganize blocks in memory. This facilitates dynamic octree updates, for example when streaming data from disk.

Each block consists of an array of child descriptors, an info section, contour data, and a variable number of attachments. This is illustrated in Figure 1. The child descriptors (Section 5.1) and contour data (Section 5.2) encode the topology of the octree and the geometrical shape of voxels, respectively. Attachments (Section 5.5) 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.

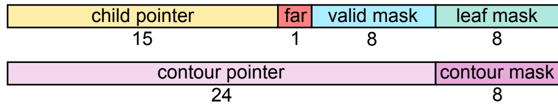


Figure 2: *64-bit child descriptor stored for each non-leaf voxel.*

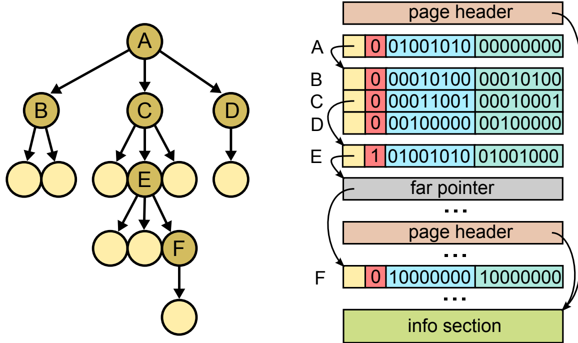


Figure 3: *Layout of the child descriptor array. Left: Example voxel hierarchy. Right: Child descriptor array containing one descriptor for each non-leaf voxel in the example hierarchy.*

5.1 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 5.2).

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, and is therefore empty.
- The bit in *valid mask* is set: the slot contains a non-leaf voxel that is subdivided further.
- 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, 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 as the parent 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.

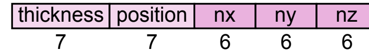


Figure 4: *Bitfields in a contour value. Components of the normal vector as well as spatial position are encoded as signed integers. Thickness is encoded as an unsigned integer.*

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.

5.2 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 5.

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 slabs that define a tight bounding volume for the surface, as illustrated in the middle and bottom rows of Figure 5. 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 6.

We use 32 bits to store the contour corresponding to one voxel. The value is divided into five components: three to define the normal vector of the two planes, and two to define their positions within the voxel. See Figure 4 for details.

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.

5.3 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.

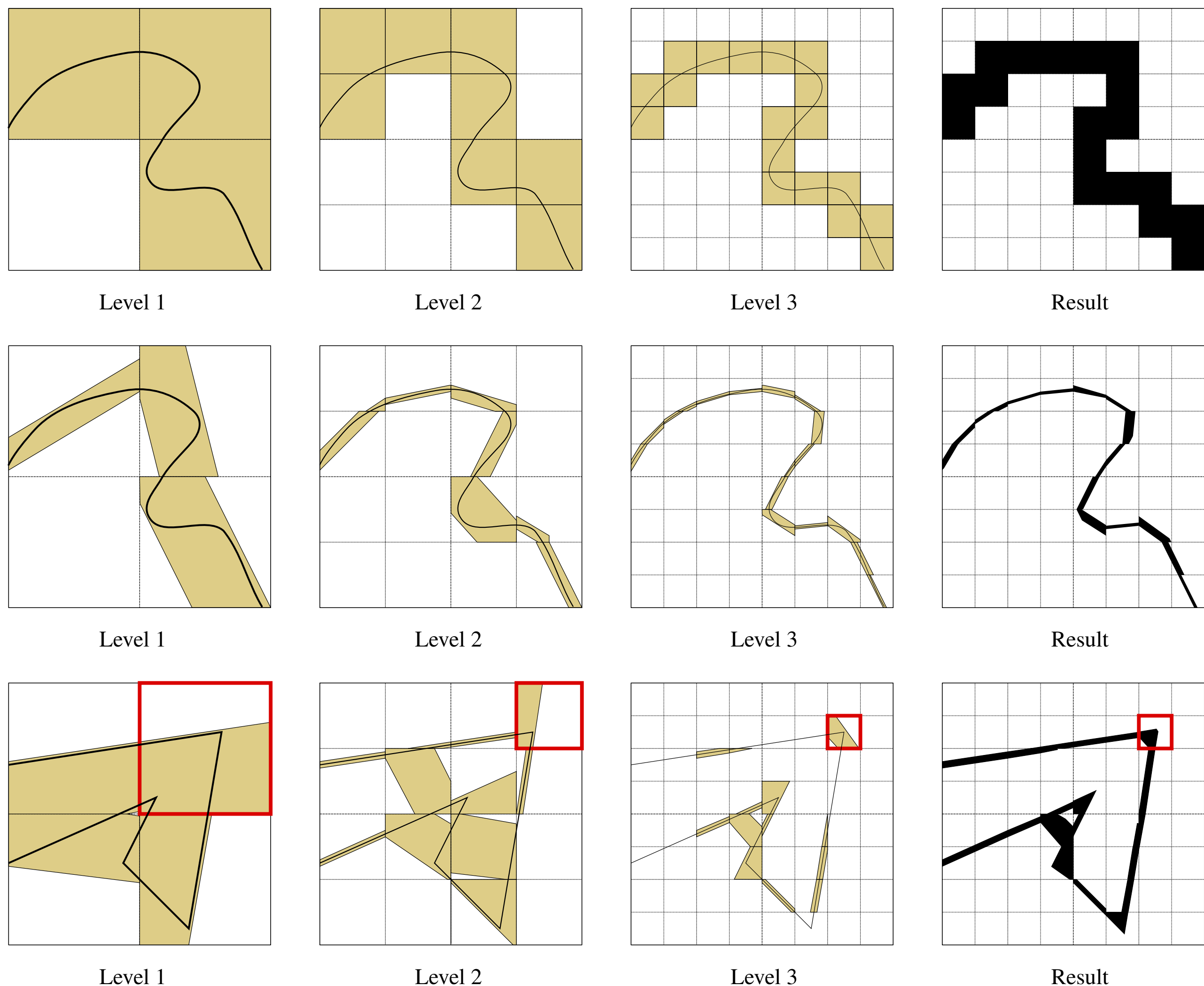


Figure 5: *Effect of contours on surface approximation. Top row: cubical voxels. Middle row: voxels enhanced with contours. The resulting approximation follows the original surface much more closely than in the top row, with some jaggedness remaining in the areas of high curvature. Note that in absence of sharp corners, contours on the most detailed level would be sufficient for obtaining the final result. Bottom row: surface with sharp edges. The result is obtained by intersecting the overlapping contours of each level. In the highlighted area, all three levels contribute to the final shape of a voxel.*

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 5.

5.4 Uncompressed Shading Attributes

In addition to the geometrical shape of voxels, we also need to store a number of material attributes for shading. Such attribute data is included within octree blocks in the form of one or more attachments. Attachments are auxiliary data buffers whose interpretation is specific to their type. The info section of each block stores a directory of attachments contained by the block, each identified by a type ID and a relative data pointer. The access model is such that once a ray hits a surface during rendering, the shader can query the attachments available for the particular voxel and decode the attributes it needs.

The Phong shading model we use needs a color and a normal vector for each voxel. We have two different schemes for storing these attributes. In this section, we describe a straightforward storage format for uncompressed, i.e. raw, attribute data, and the next section (Section 5.5 will introduce a more compact block-based compression scheme. Even though we only consider colors and normals, we expect the same ideas to apply to a wide range of other material attributes as well.

In the simple but inefficient uncompressed storage format, we use a 64-bit encoding to store the color and normal vector associated with a single voxel. The 64-bit value is divided into 32 bits for the color and 32 bits for the normal as illustrated in Figures 7a and 7b, respectively. We use a standard ABGR-encoding for the color value, storing each channel as a separate 8-bit integer. For the normal vector, however, we have to use a more specialized encoding.

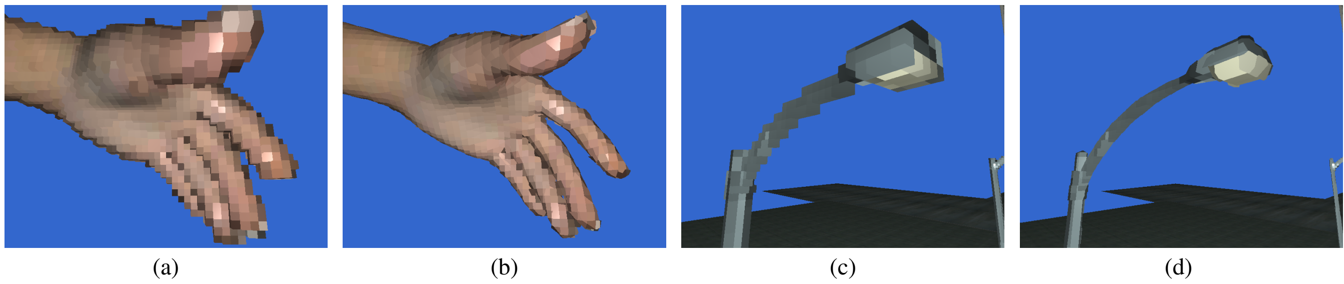


Figure 6: *(a) and (c): Cubical voxels. (b) and (d): The same voxels with contours. In this kind of situation where surfaces are reasonably smooth, contours can provide several hierarchy levels' worth of geometric resolution improvement. Note that the models have been deliberately undersampled to illustrate the effect. In practical content, the improvement would be seen in details, not in the overall shape.*

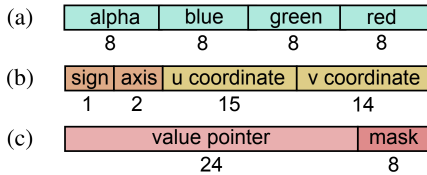


Figure 7: *Bitfields related to storing uncompressed attributes. (a) Color value. (b) Normal vector. (c) Lookup entry.*

Contrary to how normal maps are traditionally stored in triangle-based representations, we have to use object-space normals instead of tangent-space normals. Experimentation shows that on large smoothly curved surfaces, object-space normals need at least 12 bits of precision to avoid visible quantization artifacts. To maximize precision, we store each normal as a point on a cube. The face is identified using 3 bits (sign and axis), 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.

The attachment storing uncompressed colors and normals consists of two parts. The first part is an array of lookup entries matching the layout of the child descriptor array. The second part is a collection of 64-bit attribute values referenced by the lookup entries. Every entry in the child descriptor array has a direct correspondence to an entry in the lookup array, including the gaps due to far pointers and page headers. This makes it possible to map child descriptor pointers to their corresponding lookup entries without the need for additional data structures. See Figure 1 for an illustration.

The encoding of a lookup entry is illustrated in Figure 7c, and is similar to the second part of a child descriptor. The 8-bit attribute mask tells whether a child voxel in each slot of the voxel has an attribute value. Child slots whose corresponding bit is not set inherit their attributes from the parent voxel, allowing attributes to be omitted in areas where they do not vary significantly. The *value pointer* references a list of consecutive attribute values in the second part of the attachment, one for each bit that is set in the attribute mask. The pointer is relative to the beginning of the attachment.

The example code in Figure 8 summarizes the process of finding the uncompressed attribute value corresponding to a given voxel. Even though the process involves multiple indirect memory lookups, it is relatively cheap compared to the cost of ray casts.

```
int lookupUncompressedAttributeAddress(
    int childDescAddr,
    int childSlotIdx,
    int attachmentIdx)
{
    // Find the info section.

    int pageSize = 8192;
    int pageAddr = childDescAddr & ~(pageSize - 1);
    int infoAddr = pageAddr + *((int*)pageAddr) * 4;
    InfoSection* info = (InfoSection*)infoAddr;

    // Determine index of child descriptor in block.

    int blockAddr = infoAddr + info->blockPtr * 4;
    int childDescIdx = (childDescAddr - blockAddr) / 8;

    // Find and decode the lookup entry.

    int attachPtr = info->attachPtr[attachmentIdx];
    int attachAddr = infoAddr + attachPtr * 4;
    int lookupAddr = attachAddr + childDescIdx * 4;
    int lookupEntry = *(int*)lookupAddr;
    int attribMask = lookupEntry & 0xFF;
    int valuePtr = (lookupEntry >> 8) & 0xFFFFFF;

    // Is the attribute omitted for the voxel?

    int childBit = 1 << childSlotIdx;
    if ((attribMask & childBit) == 0)
        return 0;

    // Find value for the requested child slot.

    int valueIdx = popc8(attribMask & (childBit - 1));
    return attachAddr + valuePtr * 4 + valueIdx * 8;
}
```

Figure 8: *Example code for finding uncompressed attribute value for a given voxel. The voxel is specified using a child slot index and the memory address of the corresponding child descriptor.*

5.5 Compressed 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 can 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.

To do this, we encode colors and normals together, employing a block-based compression scheme that spends an average of 1 byte for colors and 2 bytes for normals per voxel. 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 val-

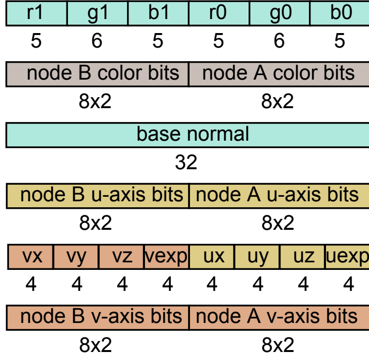


Figure 9: *Compression block consisting of six 32-bit words. The first two words encode the colors of 16 child slots, while the remaining four words encode the corresponding normals.*

ues 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 lookup table. It also has the benefit that individual values can be represented more accurately, since there is less competition within each compression block. An alternative would be to introduce a lookup table similar to the one used with uncompressed attributes, and encode each consecutive run of 16 attributes as a single block.

The encoding of the compression block is shown in Figure 9.

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 $\{\mathbf{c_0}, \mathbf{c_1}, \frac{2}{3}\mathbf{c_0} + \frac{1}{3}\mathbf{c_1}, \frac{1}{3}\mathbf{c_0} + \frac{2}{3}\mathbf{c_1}\}$.

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 10. 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 using the same encoding as with raw attributes (Figure 7b). 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

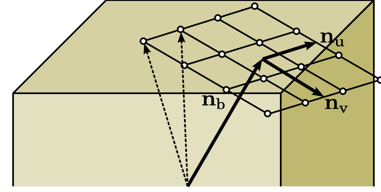


Figure 10: *Illustration of our normal compression scheme. Base normal $\mathbf{n_b}$ specifies a point on a unit cube, and two axis vectors $\mathbf{n_u}$ and $\mathbf{n_v}$ define an arbitrary 4×4 grid around this point. There are no orthogonality requirements between the three vectors, and the axis 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.*

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.

5.6 Memory Usage Analysis

Let us first consider the voxel hierarchy and later the attributes (colors, normals, contours). For every non-leaf voxel, our hierarchy encoding requires two 8-bit masks and a 15+1 -bit child pointer. For offsets larger than what the 15-bit field can accommodate, we need separate far pointers, but the amount of memory taken by these is negligible. For leaf voxels, which correspond to finest-resolution samples, no hierarchy data is stored. With average branching factor of four, we thus have 1 byte of hierarchy data per voxel on the average.

Contours require 32 bits for each voxel where they are used. However, contours are only required in voxels where they enhance the surface quality, which in most cases is a small subset of all voxels. Nonetheless, a 32-bit lookup entry (contour pointer and mask) is required per non-leaf voxel, which makes the total cost of contours slightly above 1 byte per voxel.

Let us now consider the attribute data sizes. As described above, colors can be compressed into 4 bits per slot, making it preferable to not use attribute data pointers and instead allow empty space in attribute data buffers. All eight children will always need to be encoded, so we need 32 color bits per non-leaf voxel, yielding 1 byte per voxel. Our normal compression format requires 8 bits per sample, which yields 2 bytes per voxel in total, assuming an average branching factor of 4.

Counting together hierarchy, contours, colors, and normals, we arrive at approximately 5 bytes per voxel (Table 1).

5.7 Potential Improvements

In this section we list a number of potential but yet untested improvements to the data structure. Some of these have easily pre-

dictable outcomes, while the impact of others cannot be evaluated without empirical tests. Note that some of the following ideas are not compatible with each other.

Combined indirect attribute lookups. Using lookup entries (Figure 7c) is beneficial when the size of the attribute is large enough. With average of 4 children per voxel, the breakeven point is 1 byte, so neither colors or normals, compressed down to 4 and 8 bits per voxel, benefit from the additional indirection. However, since we always store color and normal together, it would make sense to have the lookup entry.

Currently color-normal pair takes effectively 24 bits per voxel due to wasted slots in compressed tiles, but with lookups we could cut this down to 12 bits for payload and approximately 8 bits for the lookup entry, totalling 20 bits. Furthermore, the lookup entry would allow omitting attributes where they are not required, unlike the direct addressing scheme. This could allow some amount of compression near the leaf voxels without sacrificing quality.

Repurposing unused child pointers. Voxels that contain only leaf voxels as children waste the 16 bits reserved for *far* and *child pointer* in the child descriptor (see Figure 2). Because this is obviously a common situation near the leaves of the tree, it would make sense to repurpose the unused fields. One interesting possibility would be storing the contour pointer and contour mask in the vacant space.

Since there would be only 8 bits left for the contour pointer this way, we would need to interleave the contour data with the child descriptors, unlike in the current approach. Also, having some child descriptors take 32 bits and some take 64 bits would complicate the memory fetches. Whereas we currently always fetch 64 bits, we would then need to fetch 32 bits first and check the contents of *leaf mask*. If not, another 32 bits would need to be fetched. Note that always reading 64 bits would not be possible, because the child descriptors would not be suitably aligned anymore.

If all leaf voxels were on the same level, we could save approximately 1 byte per leaf with this scheme. Taking the rest of the levels into account, this translates to about 0.75 bytes per voxel, thus removing most of the contour-related memory consumption. In practice, the savings would be less than this, because a voxel can have both leaf and non-leaf children, and in this case the optimization would not be possible. We have not measured how common it is for a voxel to contain only leaves as children, so the true impact of this optimization is still somewhat speculative.

Repurposing unused contour pointers. This potential optimization is a major reorganization of child descriptors and contour and attachment lookup entries. The idea is to store attribute lookup entries within the node array, utilizing unused contour pointers whenever possible.

In the following, we assume that all attributes are accessed indirectly using lookup entries. Let us consider a list of child descriptors laid out consecutively in GPU memory. Currently each voxel has a 64-bit entry, half of which encodes the hierarchy and half of which points to the contour data. Attachment lookup entries are stored in a separate array.

To improve the memory utilization, the child descriptor span could be laid out as follows. Each child descriptor of a non-leaf voxel still consumes 64 bits, allowing fast lookup based on the index of the child in the parent voxel. However, the latter 32 bits of the child descriptor may or may not contain the contour lookup entry. Contour lookup entries are identified by e.g. setting their highest bit, whereas other possible contents for this space have their highest bit cleared. This way, the ray caster can peek at the bit and if the entry is not a contour lookup entry, none of the voxel's children

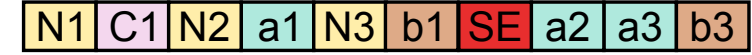


Figure 11: *The hypothetical interleaving scheme for child descriptor entries and lookup entries. Illustrated is a span with three non-leaf voxels* N1–N3*. All boxes correspond to 32 bits. Only the first voxel has a contour lookup entry* C1*. Attachment lookup entries* a1–a3 *and* b1,b3 *are stored in an unintuitive but efficient way on both sides of span entry field* SE*. See text for discussion.*

have contours. The amount of data touched by the ray caster would therefore not change from the current encoding.

After the child descriptor entries, the span continues with a 32-bit span entry. The span entry acts as a directory for the child descriptor span, indicating which voxels have contour pointers, and which attribute lookup entries are present in each child voxel. These attribute lookup entries can then be stored partially into the unused halves of child descriptors, partially after then span entry.

Consider the situation depicted in Figure 11. There are three voxels in the span, labeled N1–N3. Only N1 has children with contours, so it has a contour lookup entry C1 next to it. Attribute lookup entries a1 and b1 are stored in the unused contour pointer slots.

What is most important in this scheme is that it would allow omitting contour and attachment lookup entries selectively. For example, if we had an attachment that had data for very few voxels, the current encoding would always need to store at least the lookup entry, yielding 1 byte per voxel. This overhead could be removed with the suggested encoding.

The unused combination. We currently use only three out of four possible combinations in *valid mask* and *leaf mask* (Figure 2). We have so far refrained from taking this combination in use, because the best way to utilize it is not obvious.

One interesting extension to the current scheme would be indicating partially transparent voxels using the unused combination. This would allow e.g. accumulating occlusion factor or performing color accumulation and attenuation during raycasting. A slight complication is that one cannot distinguish transparent leaf voxels from transparent non-leaf voxels. Therefore, all transparent voxels would need to have a corresponding child descriptor entry, even when they are leaf voxels.

6 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 voxel that the ray is currently traversing 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. We express the cube 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

$$t_x(x) = \left(\frac{1}{d_x}\right)x + \left(\frac{-p_x}{d_x}\right)$$

for the x-axis, and similar formulas for the y and z axes. With precomputed ray coefficients this amounts to a single multiply-

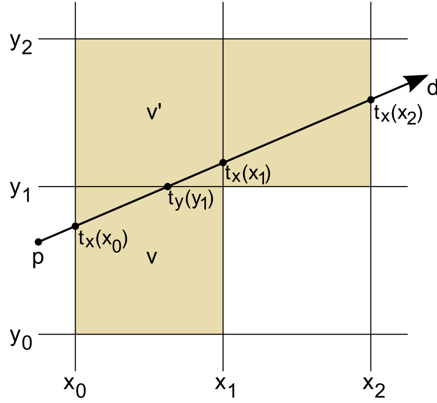


Figure 12: *Advancing through voxels of equal scale. The ray is defined by the origin vector* $\mathbf{p}$ *and direction vector* $\mathbf{d}$*. It enters* v *at* $t = t_x(x_0)$ *and exits it at* $t = t_y(y_1)$*, proceeding to* v'*.*

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 $tc_{max} = \min(t_x(x_1), t_y(y_1), t_z(z_1))$.

Let us consider the problem of determining the next voxel v' along the ray, given the current voxel v as specified by $parent$ and idx. We will start by assuming that v and v' are siblings of each other, implying that they are of the same scale as depicted in Figure 12. In this particular case, the current voxel's cube spans the range $[x_0, x_1]$ horizontally and $[y_0, y_1]$ vertically. This means that the ray must exit v through either x_1 or y_1, whichever it happens to intersect first. The values of t corresponding to the intersections are given by the functions t_x and t_y. We see that $t_y(y_1) < t_x(x_1)$, meaning that the ray intersects y_1 before x_1. With reasoning similar to [Amanatides and Woo 1987], we can thus conclude that v' lies directly above v.

Therefore, 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: Proceed to the next sibling of the highest ancestor that the ray exits.

Figure 13 shows an example of the resulting traversal order.

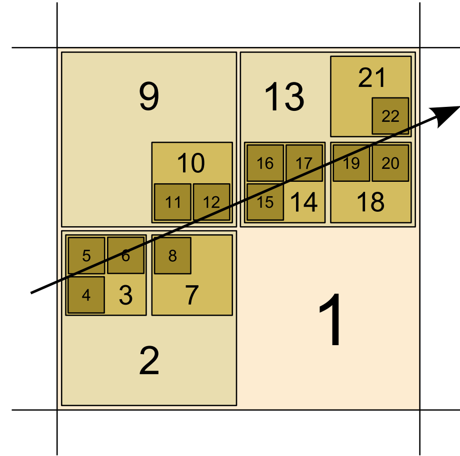


Figure 13: *Order of hierarhicy traversal. The algorithm starts at voxel 2, which is a child of the root 1. It executes* PUSH *twice to reach voxel 3 and then leaf 4. After executing* ADVANCE *twice to traverse siblings 5 and 6, the algorithm notices that the ray exits their common parent 3. It thus executes* POP*, reching voxel 7. The traversal terminates after leaf 22 when the ray exits the root. Note that the voxels in the image have been shrunk to make the numbering clearer.*

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.

In the case of POP, we can determine the next voxel by looking at the bit representations of pos and pos^*. Figure 14 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.

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 14: *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.*

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 $scale'$.

6.1 Ray Cast Implementation

Pseudocode for the complete ray cast algorithm is given in Figure 15. 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.

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.

```
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
11:     if voxel is small enough then return t_min
INTERSECT
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
18:       if voxel is a leaf then return tv_min
PUSH
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 15: *Pseudocode for the ray cast algorithm.*

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.

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.

For the sake of clarity, a few implementation details have been left out of the pseudocode. The most important of them are outlined below. The actual ray caster code is given in Appendix A.

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.

Caching the current child descriptor. Whenever the algorithm executes ADVANCE, $parent$ remains unchanged for the next iteration. It is thus reasonable to conserve memory bandwidth by fetching the corresponding child descriptor only once. This is accomplished by invalidating the local copy of it in PUSH and POP, and fetching the descriptor from memory in the beginning of the loop only if it is invalid.

Sharing computation between INTERSECT and PUSH. It is possible to organize the computation of contour projections so that it utilizes the values of t_x, t_y, and t_z evaluated at the center of the voxel. The same values are also used to determine the first child voxel that the ray enters in PUSH. It is thus beneficial to compute the values in the beginning of INTERSECT and re-use them in PUSH.

Hit position. In most cases, the position where the ray intersects voxel geometry is also needed in addition to the t value. Computing the position simply as $\mathbf{p} + t\mathbf{d}$ is not robust in practice, as floating point inaccuracies can cause the resulting point to lie outside the cube corresponding to the intersected voxel. It is thus necessary to explicitly clamp each component of the position to the minimum and maximum coordinates of the cube.

6.2 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.

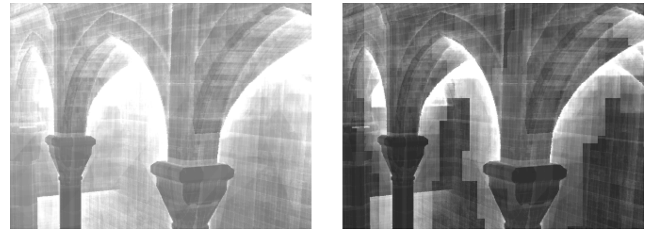


Figure 16: *Illustration of the effect of beam optimization on iteration counts. Left:* SIBENIK-D *with no beam optimization. Right: with beam optimization in 8×8 blocks. White corresponds to 64 iterations in both images.*

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. 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 16 illustrates the effect of beam optimization on iteration counts.

6.3 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 17.

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. The kernel we have used for the images in this paper is shown in Figure 18.

Pseudocode of the algorithm is given in Figure 19. 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

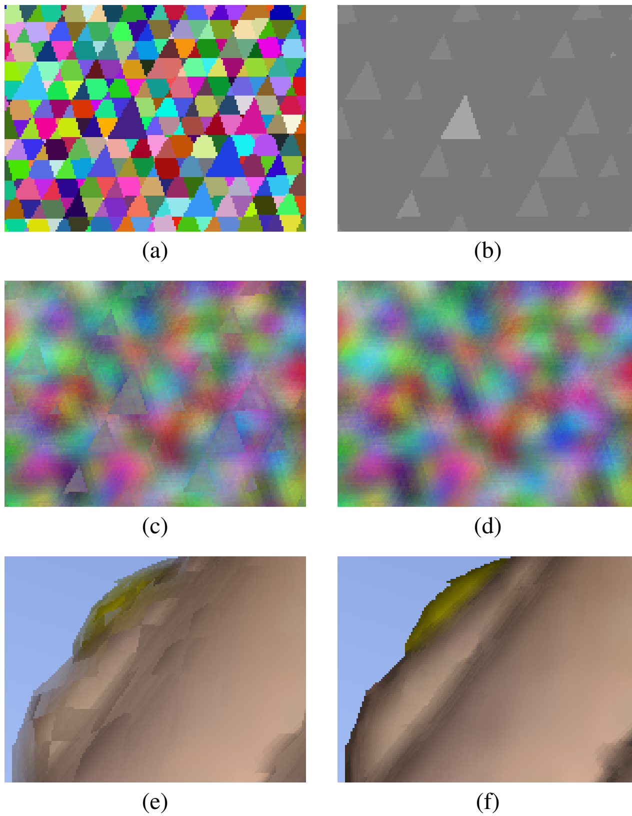

(a) (b) (c) (d) (e) (f)

Figure 17: *The problem with varying filter radii. (a) False-colored voxels on an oblique surface. (b) Octree depth of each point, brighter tone indicating shallower tree, i.e. larger voxel. (c) Filtering each pixel with a radius deduced from the size of the corresponding voxel. Seams are visible at hierarchy level changes. (d) Our method adjusts the filter radius while accumulating neighboring colors, yielding smooth transition between levels. (e) and (f): standard and our method applied to* FAIRY*'s hand built with aggressive pruning and few voxel levels.*

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.

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.

6.4 Failed Approaches

We tried out a number of ideas that seemed feasible, but ultimately proved not to be. This section collects some of the most fundamental ones in terms of expected gain and observed results.

Rasterizing voxels. We wrote a simple hierarchical voxel rasterizer that was inspired by the micro-rendering technique of Ritschel et al. [2009]. Each thread is given a beam of 4×4 or 8×8 rays to process. The thread traverses the voxel hierarchy so that voxels outside the beam are culled. This can be done quite efficiently using box vs frustum intersection tests. Whenever the traversal encounters a voxel that projects to about one pixel in size,

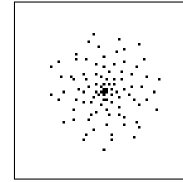


Figure 18: *Filter kernel used for the images in this paper. The samples are stored in constant memory and ordered in ascending order of distance from the center. Each sample has a weight proportional to the disc area it represents. Kernel radius is 24 pixels and it has 96 samples. This particular kernel causes some banding artifacts as can be seen in e.g. Figure 17d, but we expect that tweaking the sample positions manually or using a more sophisticated relaxation method could improve the situation considerably.*

1: $(c, r) \leftarrow \text{fetch}(x, y)$
2: **if** $r \leq 1$ **then return** c
3: $accum \leftarrow (0, 0, 0, 0)$
4: **for** each sample s in kernel **do**
5: $(c', r') \leftarrow \text{fetch}(x{+}s.x,\ y{+}s.y)$
6: $r \leftarrow \min(r, r')$
7: **if** $s.dist > r$ **then break**
8: $w \leftarrow s.weight \cdot \min(r - s.dist, 1)$
9: $accum.rgb \leftarrow accum.rgb + c' \cdot w$
10: $accum.w \leftarrow accum.w + w$
11: **end for**
12: **return** $accum.rgb / accum.w$

Figure 19: *Pseudocode for the post-process filtering algorithm.*

this pixel is plotted into a small frame buffer kept in local memory, and the children of the voxel are not visited.

The main benefit of this approach is that there is strictly less traversal work to be done, because the common part of traversal for rays in each beam is performed only once. However, there are two major complications. Firstly, the result is not quite exact because voxels are drawn in the frame buffer as points, and no exact ray vs voxel test is performed. Secondly, leaf voxels that are larger than a pixel require specific handling.

Besides these issues, it seems that there is very little coherence between beams, and this can make the execution and memory accesses diverge significantly between SIMD threads. Updating the frame buffer in local memory has no coherence between threads, making the updates expensive. Even when only updating a depth buffer and without specific handling of large voxels, the rasterizer was several times slower than the ray caster. Beating the ray caster seemed very unlikely, and therefore the development of the rasterizer was ceased.

We hypothesize that the main benefit of rasterization—shared processing of the common part of traversal—is already adequately handled by the beam optimization for the ray caster. Also, the common part is cheap to execute because of high data and execution coherence.

Short stack. Thanks to profiling (Section 8.5) we noted that our ray caster causes a significant amount of stack traffic. Therefore, it would seem like a good idea to reduce it by storing only the top of the stack in registers. This is known as *short stack* [Horn et al. 2007]. The downside of using a short stack is that whenever the stack is exhausted, the traversal has to be restarted from the root. Restarts are quite expensive, but it can be argued that the restarts are so infrequent that their total cost is small.

We have previously found short stack to be somewhat slower than full stack in case of triangle ray casting, and with voxels the penalty is even more pronounced. The reason is that the stack is used for different purposes in the two cases.

For triangle ray caster, the stack acts as a list of nodes to visit, and each pop removes the topmost item. With voxels, however, the stack is used for storing the path to the current voxel, and the level that is popped depends on which boundary is crossed by the ray while traversing. Consider, for example, the axis-aligned planes that lie at the middle of the octree. When the ray crosses any of these, we need to pop all the way to the very first, i.e. the bottom, entry of the stack. This kind of random access is much less suitable for short stack approach than the usual pops.

Re-use of ray cast stack. This optimization is related to secondary rays. Because the ray cast stack always contains the path to the current voxel, it is possible to take the final stack of a primary ray and use it as a starting point for the secondary ray, given that the secondary ray starts where the primary ray ended.

In an ideal world this would work as-is, but in practice it is necessary to move the starting point of the secondary ray a bit to account for inaccuracies. It turns out that this can be handled as long as the offset is in the same coordinate octant as the direction of the ray, but offset to other octants is complicated. Therefore, the only practical offset direction is to the direction of the secondary ray, which is fortunately fine for most purposes. However, the larger the offset, the less useful the re-using of the stack is.

We are still not entirely certain why stack re-use failed to provide speedup for rendering, but we observed that while the iteration counts for the secondary rays did decrease, the overall performance degraded a bit. A possible reason is that starting each of the secondary rays with its own stack decreases the overall coherence. Also, the initial descent in the voxel hierarchy is very coherent and cheap. It looks like avoiding it is not worth even the small amount of extra code that is required for handling the offsetting of the secondary ray starting point.

Attribute interpolation. We spent a lot of effort in trying to remove the blockiness of the surfaces in the same way that the conventional graphics pipeline of GPUs does it with textures. Initial experiments with interpolated attribute fetch were discouraging—finding the neighbors and decoding and interpolating the attributes was more expensive than performing the actual ray cast.

From this, it became apparent that some amount of redundancy is required for the attribute storage so that the costly neighbor search could be avoided. After many experiments, we converged to a scheme where each non-leaf voxel has a $3 \times 3 \times 3$ grid of attribute points, each of which could be occupied or empty. These points correspond to the corners of the child voxels, and the data points on the faces of the parent voxel are duplicated between adjacent voxels so that neighbor lookups are never required.

This scheme yields approximately 4.5 attribute points per voxel, which is quite a big expansion factor. It would be possible to decrease this by using smart compression schemes, e.g. by using fewer than 8 corners per voxel where possible, but it seems that at least twice the ideal amount of data would be required even in the best case.

Attribute lookup is quite fast in this kind of structure, because no neighbor search is needed. However, block-based texture compression cannot be easily used, because the natural unit of compression would be a set of eight children, where up to 27 attribute points may be occupied. Finally, the construction of attributes becomes non-trivial. It is easy to guarantee continuity between neighboring voxels, but enforcing continuity between different voxel levels is tricky.

Currently it seems that it is better to smooth the surfaces using the post-process blur than to perform interpolated attribute lookups. More data can be fit in memory without interpolation, and this already increases the quality more than interpolation in many cases. The post-process blur also has the advantage of smoothing the silhouettes between areas with large blur radii, and this cannot be achieved with interpolation.

Silhouette blurring. Before the advent of contours, we attempted to hide the blockiness of silhouettes by simply blurring them. This proved to be much more difficult than it initially seemed. An obvious problem is that the blur can only operate outwards from the surface, because due to occlusion we cannot know what lies behind the silhouette. This has the consequence of making objects appear to thicken as they come closer to the camera. Blurring around a silhouette also causes unwanted blurring of details that should remain sharp, e.g. distant geometry that is visible right outside the silhouette.

We tried to counter the unwanted effects in various ways, but no adequate solution was found. Ultimately, contours removed the need to hide silhouette blockiness in a neat and memory-friendly way, while simultaneously offering much better geometry resolution than cubical voxels could provide.

7 Data management

Because we are working with data sets that may be larger than what can be fit in GPU memory at one time, our data format is designed to be suitable for localized on-demand resolution adjustment. As noted in Section 4, the resolution requirements drop quickly when distance from camera increases, and we want to be able to exploit this.

Storage on disk. Our octree data is stored as *slices* that are kept in a single file that has a directory of slices it contains. A slice contains voxel data on one resolution level in a particular cubical part of the octree. To facilitate on-demand streaming, we constrain the amount of data that one slice can contain to about 1 MB. The slices are organized in a hierarchy, and if the next resolution level of a slice consumes more memory than allowed, the slice is split spatially so that it has more than one child slice. Figure 20a illustrates the storage of octree as slices on disk.

The structure of the slice hierarchy, without slice contents, is small enough to be kept in memory. This makes it efficient to find the slice that contains data in desired resolution in a particular spatial region. In our implementation, all data updates are performed by the CPU.

The contents of a slice differ considerably from the contents of a block in memory, because whenever we want to load a slice, we have all voxels above it in the octree already present in memory. In particular, we already know which voxels the slice will contain, because the parent level contains this information. Therefore, we do not need to store any spatial positions, pointers, or indices in the voxels contained in the slice.

Storage in memory. As discussed in Section 5, the loaded portion of the octree is stored in GPU memory as *blocks*. In practice, we need two kinds of blocks because of dynamic resolution changes. The highest octree levels are stored in the *trunk* which is a pool where child descriptors can be allocated and freed easily. The trunk is stored as blocks so that the ray caster does not need to handle voxels in the trunk differently.

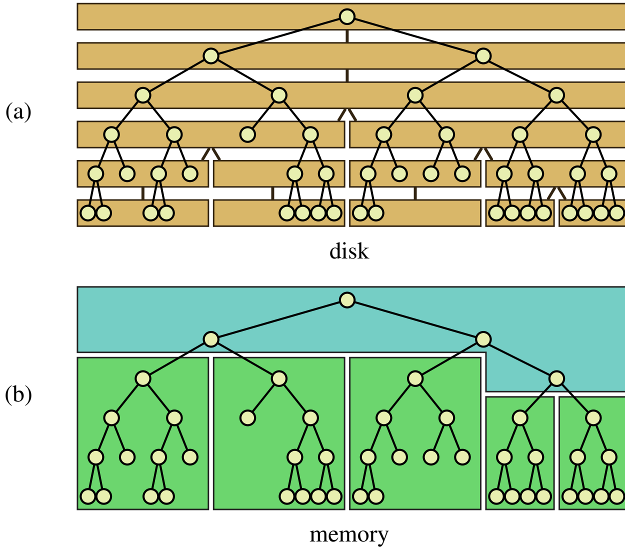


Figure 20: *Octree data on disk and in memory. (a) The octree is stored on disk as a set of slices. Each box represents a slice that is constrained to contain at most a given amount of data. The lines between the boxes indicate the slice hierarchy. (b) In memory, the octree is stored as a set of blocks. The top part is the trunk, and the bottom boxes represent leaf blocks where new slices may be added. When a leaf block has to be split, some of its contents are moved to the trunk.*

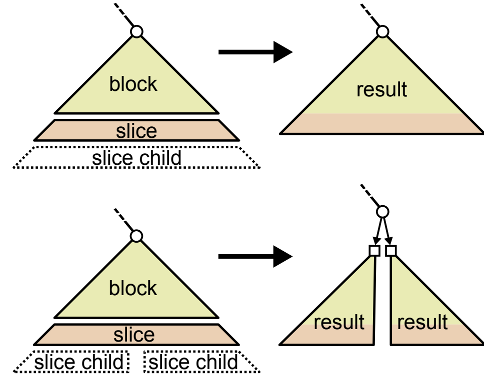


Figure 21: *The process of loading a slice. Top: When loading a slice that has only one child, the block is augmented with slice data and one result block is produced. Bottom: If the slice has multiple children, one result block is produced per slice child. The original block root is moved to the trunk.*

Child descriptors in the trunk are allocated so that they always have space for far pointers and every possible attribute. This makes the trunk blocks flexible, which is necessary because loading and unloading of slices often requires small modifications in the trunk. Because of high branching factor and large blocks, there are only a handful of trunk blocks in memory at any time.

The vast majority of voxel data is stored in ordinary, or *leaf* blocks. Voxels in a leaf block are organized in depth-first order in memory to make them as compact and efficient as possible. Figure 20b illustrates the storage of octree as blocks in memory.

Loading slices. Let us consider the situation where we have an octree in memory and wish to augment a given block by adding an extra level of resolution. For now, let us assume that the slice is not split, i.e. has only one child. We first locate the slice that contains the desired part of the octree and load the slice data from disk. We also transfer the relevant block from GPU to host memory. We then merge the block with data from the slice to produce a result block that contains the desired extra level of resolution. Finally, we transfer the result block back to GPU memory.

At all times, we maintain one-to-one correspondence between non-trunk blocks in memory and next-level slices on disk. To accomplish this, we must be able to split blocks when loading slices. If the slice that was loaded has more than one children, we split the block by contructing one result block for each of the child slices. This ensures that each result block again has exactly one slice corresponding to next resolution level. When a block is split into multiple blocks, there are one or more voxels near the root of the original block that are not in any of the result blocks. These voxels are transferred to the trunk. See Figure 21 for illustration.

Dynamic loading. We use simple heuristics for dynamically selecting slices to load into memory. Based on the distance from the camera to a block and the resolution of the block, we can approximate the size of a leaf voxel on the screen. Our heuristics attempts to maintain a constant voxel-to-pixel ratio, utilizing all available memory resources. For every slice in memory and on disk, we can calculate a score that approximates how big quality impact the slice has for the current camera position. Slices are loaded in the order determined by the scores. When GPU memory becomes full, slices are unloaded from memory to make room for new ones as long as this improves the overall score. Asynchronous I/O is used for streaming multiple requests to the operating system, increasing the total data throughput.

We use a custom memory manager that allocates a large arena of GPU memory and handles the allocation and freeing of blocks. Due to loading and unloading of variable-sized blocks, the memory space will gradually become fragmented. When fragmentation prevents memory allocation from succeeding, the blocks are compacted so that a larger continuous free space is obtained. To avoid major hiccups, the compactor only moves as much data as is necessary at any given moment.

7.1 Voxel Hierarchy Construction

The octree hierarchy is constructed in a top-down fashion one slice at a time. We insert special *unbuilt* slices in the slice hierarchy in places where we want to continue construction. Unbuilt slices contain all data required to construct the voxels for the slice, referred to as the *build data*. There are no dependencies between slices, making it trivial to use all CPU cores for the construction simultaneously. The build data contains the following information for each voxel: flags (REFINE-GEOMETRY, REFINE-ATTRIBUTES), voxel position, indices of input triangles and displacement primitives (Section 7.4) intersecting the voxel, and the most important ancestor contours. The flags indicate whether the shape or attributes of the voxel are outside given error tolerances. When variable build resolution is enabled, at least one of these flags is therefore always active, because otherwise the voxel would be fine as it is and would not need to be subdivided further.

The construction begins by taking the input mesh and creating an unbuilt root slice covering all input triangles. When the builder processes an unbuilt slice, it reads the build data and replaces the contents of the slice with actual voxel data. One or more child slices are also created with their respective build data. Processing of a single voxel in the build data consists of filtering the triangles and dis-

placement primitives to eight child voxels, determining appropriate contours (if requested by flag REFINE-GEOMETRY) and attributes (if REFINE-ATTRIBUTES is set) for them, and finally, creating the build data entries for the child voxels.

Each unbuilt slice contains a description of how the slice should be split into child slices. It is important to allow a slice to be split into more than eight child slices. Otherwise we would need to prepare for the worst-case data expansion, which in turn would lead to very small slices on the average. Consider, for example, a situation where we have increased the voxel resolution for many levels without splitting the slice, as the data has been low-dimensional in large scales. As a consequence, there is a large gap between the voxel level and slice split level, making the slice spatially large compared to the size of the voxels it contains. Now, assume that the data starts to behave in a volumetric fashion so that the number of voxels grows eight-fold on each level. Remembering that the data was low-dimensional in large scales, splitting the slice once does not decrease the amount of data as much as increasing the resolution increases it. Therefore, the amount of data per slice could grow uncontrollably.

Even if this example may sound a bit artificial, the problem is severe in practice unless splits to more than eight children are allowed. Note that the parent slice must decide the split strategy for each of its child slices before seeing how much data the child slices will actually contain after being built, so there is some chance that the estimate is overrun and a slightly larger slice than what was hoped for is generated.

7.2 Contour Construction

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. It should be noted that even though we do not enforce smoothness, the resulting contour surface tends to be relatively smooth.

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, as illustrated in Figure 22. 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.

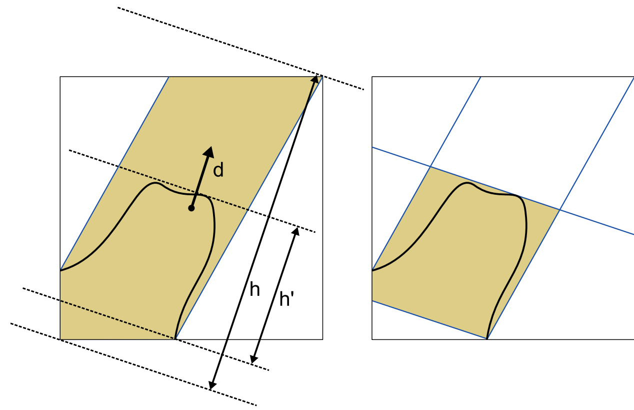


Figure 22: *Evaluation of candidate direction in cooperative contour construction. Left: one contour has already been defined (blue lines) and candidate direction* $\mathbf{d}$ *is being considered for the next contour. Outermost dashed lines indicate current voxel extent* h *in candidate direction, whereas inner dashed lines show the resulting extent* h' *if a contour is inserted in this direction. The score of the candidate direction is determined by difference* $h - h'$*. Right: situation if the candidate direction on the left is chosen.*

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. The contour quality threshold is currently specified as an effective cubical octree level, meaning that the shape of the voxel is considered to be good enough if the geometrical error is smaller than the size of a cubical voxel on the given level.

7.3 Attribute Construction

Attribute values are computed by integrating over all input surfaces within the voxel. We experimented with weighted filters, e.g. pyramidal filter that extends beyond the voxel boundaries, but the results were not any better than with the simple box filter. Filters with larger support tend to blur the content. We also tried taking the midpoint of the attribute extents, reasoning that this would minimize the error metric, but the resulting quality was bad. An obvious problem in treating all surfaces within a voxel as equally important is that for close-by surfaces even the parts that are not visible to the outside can influence the attributes. The correct solution would be to somehow figure out which are the ”outermost” surfaces and take only them into account, but this seems non-trivial and is left as future work.

The decision whether the attributes are represented adequately is made by looking at the input geometry in the voxel and checking if it contains values that are too far from the encoded ones. Attribute quantization and compression is performed before the test. This ensures that artifacts caused by them are properly taken into account. Note that different error tolerances are used for different attributes.

The way textures are sampled has a significant effect on the quality of the results. We build mip-maps of the textures and take one trilinear sample per triangle within the voxel. This is not particulately accurate, but works satisfactorily. More accurate solutions might provide some quality improvement, especially if highly anisotropic texture parameterizations are used.

When attribute compression is enabled, the attributes are encoded into every voxel. This is necessary because our current implementation does not allow leaving DXT compression blocks out. Furthermore, because color and normal are encoded together, leaving one out is not possible either. Note that compression artifacts are detected by the attribute error metric, as the actual compressed value is used as the reference. Adding more resolution makes the compression blocks spatially smaller and more likely to contain similar colors, so encoding with variable resolution automatically fixes the compression artifacts where they occur, unless the maximum level limit is reached.

7.4 Displacement Mapping

Displacement-mapped triangles are handled by first converting them into displacement primitives. These are processed similarly to ordinary triangles, with the exception that they may be split when necessary. The displacement primitive is represented as a pair of a power-of-two square in displacement map space and the original triangle. As voxels are subdivided, the displacement primitives are split until they become smaller than the voxel in world space, and primitives that are certainly outside the voxel are culled away. Operations such as determining geometry extents or integrating over the contents of the voxel are approximative, as they are carried out conservatively. For example, the geometry extents are calculated based on minimum and maximum displacement values fetched from a mip-map.

8 Results

The main 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.

Additional rendering tests were performed on NVIDIA GeForce GTX 285 with 1 GB of RAM installed in a PC with 2.66 GHz E6750 Intel Core2 Dual CPU and 2 GB of RAM. The operating system in this machine was 32-bit Windows Vista Enterprise.

Figure 23 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.

8.1 Memory Usage and Build Time

Table 2 shows the amount of GPU memory consumed by the voxel datasets using various voxel levels. We can see that the representations with contours (lower rows) are much more compact than representation without them (upper rows). SIBENIK-D and HAIRBALL are exceptions, because there contours cannot represent the surface well enough to allow omitting finer levels. Naturally, the contour version of the scene is also a much more faithful representation of the original mesh. The encoding of cubical voxels currently wastes about one byte per voxel, because we do not omit the contour pointer even when contours are not enabled. Nevertheless, this accounts for only about 20% of the total memory usage.

scene	10	11	12	13	14	15	16	bytes
CITY	8	34	152	655	2724	–	–	4.99
	13	39	131	432	1368	–	–	5.44
SIBENIK	33	132	530	2120	–	–	–	5.18
	41	141	440	1034	1857	–	–	5.68
SIBENIK-D	47	184	744	2734	–	–	–	5.52
	79	314	1192	2806	–	–	–	8.10
HAIRBALL	262	1157	–	–	–	–	–	5.27
	442	1552	–	–	–	–	–	7.48
FAIRY	10	36	145	606	2669	–	–	5.56
	11	35	99	239	376	639	1109	5.62
CONFERENCE	21	89	362	1459	–	–	–	5.09
	17	40	96	220	512	1328	–	5.16

Table 2: *GPU memory usage of the test scenes with different octree depths (MB). For each scene, the upper row denotes dataset with cubical voxels (no contours) and the lower row denotes one with contours. The* bytes *column on the right shows the average memory consumption per voxel in the largest built dataset.*

scene	8	9	10	11	12	13	14	15	16
CITY	5	6	8	16	45	141	505	–	–
	9	11	14	23	48	115	311	–	–
SIBENIK	1	3	7	28	104	395	–	–	–
	2	4	10	31	90	194	336	–	–
SIBENIK-D	9	21	56	190	839	2079	–	–	–
	10	24	70	274	1342	2541	–	–	–
HAIRBALL	45	70	136	335	–	–	–	–	–
	77	132	294	628	–	–	–	–	–
FAIRY	1	2	4	12	42	153	592	–	–
	2	3	6	15	38	86	121	178	282
CONFERENCE	1	2	4	14	55	222	–	–	–
	3	4	7	13	24	51	115	288	–

Table 3: *Voxel hierarchy construction time for various level counts. Values are in seconds, and each value denotes the total build time up to the level indicated. The cost of building a single level is thus the difference between adjacent numbers. For each scene, the upper row denotes cube voxelization (no contours) and the lower row with contour voxelization.*

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. Comparing to the theoretical figures (Section 5.6) and taking the one wasted byte for cubes into account, we see that these measured values are about 0–0.5 bytes higher than expected without contours, and 0.2–0.7 bytes higher with contours, excluding HAIRBALL and SIBENIK-D where a disproportionally large fraction of voxels require contours.

The differences are explained by variance in branching factor, gaps in child descriptor array, and variance in the amount of contour data. Our attribute compression method requires that every span of child voxels is aligned at a 128-bit boundary, which causes a bloat of about $1/8$ of data size, assuming an average of 4 children per voxel. There is also a slight overhead in page headers, far pointers, etc. that are included in this figure.

Table 3 lists the build times for the test scenes. Note that each number is the total build time up to the level indicated, and the time required to load the mesh is not included in these figures. We can see that for most scenes, the build time with contours falls below the build time without contours. This happens because voxels cease to subdivide when contours start to approximate the surfaces well enough.

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

Figure 23: *Test scenes used in measurements.* SIBENIK *is the same as* SIBENIK-D *but with flat triangles.*

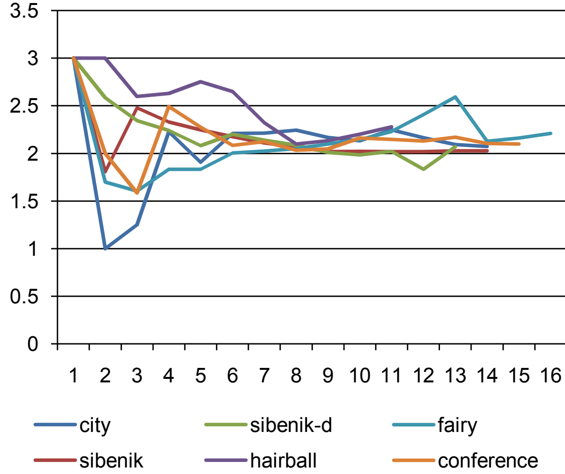


Figure 24: *Local dimensionality of our test scenes on different voxel levels. We can see that all scenes converge towards the two-dimensional setting.*

Scene	resolution	triangle caster *(Mrays/s)*	cubical voxels	con-tours *(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 4: *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.*

8.2 Scene Dimensionalities

Figure 24 illustrates the local dimensionality D_{local} (Section 4) of our test scenes depending on voxel level. The dimensionality curve has been calculated by taking base-2 logarithm of the average branching factor of each level. As the voxels get smaller, the scenes mostly converge towards two-dimensional setting, i.e. average of about 4 children per voxel. HAIRBALL looks almost volumetric for a long while, until the surface structure of the individual hairs starts to dominate. It is worth noting that in this case, the one-dimensional nature of individual hairs never leads to dimensionality less than two. This is because at the point where the surface of the individual hairs is not yet discernible, the object is so dense that the volume-like jumble near the center dominates. In FAIRY, the dimensionality remains less than two until the surface of the individual limbs becomes visible.

8.3 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 4 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 6.2). 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—it is not possible to render SIBENIK-D using the triangle ray caster.

It is worth noting that the triangle ray caster is painstakingly optimized for fast processing of triangles, and the performance is severely affected by the slightest addition of complexity. It is therefore likely that adding support for displacement maps would degrade its performance below the voxel ray caster. Admittedly, this claim remains unproven as currently there exists no such implemen-

scene	octree depth	MB	Quadro FX 5800	GeForce GTX 285	speedup %
CITY	13	432	109.5	126.6	15.6
SIBENIK	12	440	110.1	127.4	15.7
HAIRBALL	10	442	32.1	39.3	22.4
FAIRY	15	639	152.7	182.2	19.3
CONFERENCE	14	512	124.4	144.8	16.4

Table 5: *Comparison between two boards with identical GPUs. The values are in million rays per second, and measured by rendering 1024×768 primary rays using contours and beam optimization. To make comparison fair, the memory usage of both runs were limited to 1 GB available on our GTX 285 board. This explains why the reported numbers for Quadro are higher than in Table 4.*

tation optimized for current GPUs. In any case, we hypothesize that in a static scene with extremely detailed geometry, the ray cast performance is greater when ray casting in a voxel hierarchy instead of triangles and displacement maps.

8.4 GTX 285 Experiments

Because Quadro boards are generally slightly slower than GeForce boards with the same GPU, we wanted to measure how different our results would be on a GeForce GTX 285 board. Because our GTX board has only 1 GB of RAM, we ran the tests with smaller datasets that fit in 1 GB. For instance, in CITY we could only use 13 levels instead of the 14 used in other tests.

The core clock speed in GTX 285 is about 14% faster than in the Quadro board, and the peak memory bandwidth is a staggering 56% greater. Table 5 gives a summary of rendering results with contours and beam optimization enabled. Based on the results, the rendering is 15–22% faster on GTX 285 with an average speedup of 18%. The observed speedup indicates that the ray caster is not strictly limited by memory bandwidth, because the difference in core clock speed is almost sufficient to explain the results. If we were severely limited by memory performance, the speedups should be higher.

8.5 Detailed Execution Profile

Table 6 lists a number of profiling counters from rendering the test scenes in 1024×768 resolution using contours. All values are per-ray averages except for those that consider the entire frame. As can be seen by contrasting the Mrays/s figures with Table 4, there is some (less than 1%) fluctuation between benchmark runs that we have not been able to remove.

The top row *Mrays/s* indicates the measured rendering performance in millions of rays per second. The next row *simulated* shows a simulated performance figure obtained by counting the warp-wide execution counts of each code block and weighting with the corresponding instruction counts. The effects of warp divergence have been included in this figure, and it is therefore an upper bound for how fast the ray casting could optimally be executed. This assumes that every instruction that could possibly dual-issue does so, and there are no memory latencies or other hiccups of any kind. The same technique has been used before for estimating theoretical ray cast performance on GPUs [Aila and Laine 2009].

The next row shows how much of the simulation performance we reach in our measurements. The triangle ray caster consistently performed over 80% of simulated performance for primary rays, and without beam optimization we obtain close to similar figures with voxels. However, with beam optimization enabled, our figures fall to 60–80% range and even below that in FAIRY. This suggests that with beam optimization we are suffering from limited memory bandwidth to some degree.

Memory bandwidth *bw GB/s* is calculated by adding the global and local memory bytes accessed per ray, multiplying this with rays per second, and dividing by 2^{30}. This figure does not take coalescing or any other real-world memory subsystem features into account. We can see that the beam optimization reduces bandwidth requirements significantly. The final row in the top section shows how much of the total rendering time was spent in rendering the coarse frame for beam optimization.

The next section details the execution counts of various parts of code. The first row shows the average number of instructions executed per ray, not taking SIMD execution into account. Not included in the table is the overall SIMD efficiency, i.e. how many threads in a warp are enabled on the average. This hovers usually around 70% when no beam optimization is used, and 60% with beam optimization enabled.

The *iterations* row tells the number of main loop iterations in the ray caster, and *intersect* shows how many voxel-ray intersection tests (INTERSECT in pseudocode of Figure 15) were performed. Row *push* tells the number of times the PUSH branch was executed, and *store* shows the number of actual stack writes. Rows *advance* and *pop* correspond to their respective parts of the pseudocode as well. All figures are per-ray, and SIMD execution is not taken into account. We can see that by far the most common case is descending to a child voxel (*push*), and the least frequent is ascending in the hierarchy (*pop*). Comparing rows *push* and *store*, we can see that the optimization to eliminate unnecessary stack writes was able to remove over half of them.

The bottom section in the table lists memory transfer statistics. The first row *glob acc* counts the total number of global memory fetch instructions executed per ray, and the next row *glob bytes* counts the number of bytes that were fetched. Coalescing is not taken into account in these figures. However, the next row *glob trans.* counts the number of fetches after coalescing. This figure is calculated by taking all fetch instructions executed in parallel and counting how many requests these generate after the coalescing logic in the GPU is applied. Again, the grand total is divided by the number of rays to get a per-ray figure. It is easy to see that the fetches are quite well coalesced, and the actual transaction count is mostly between 12% and 16% of fetch instruction count, except for HAIRBALL where the ratio is as high as 37%.

The next three rows show the same statistics for local memory. All local memory accesses are due to stack traffic, as there are no local memory spills in the ray cast kernel. We have assumed that local memory is striped so that the same local memory address in each thread's address space is stored consecutively in GPU memory. Calculated this way, the coalescing reduces the request count to around 13–20%. HAIRBALL is in its own league with transactions per request ratio of 44%.

Low transaction/request ratios thanks to coalescing are explained by high coherence between rays. It is worth keeping in mind that these figures were measured for primary rays only, and other types of rays will probably exhibit less coherent traversal. However, most of the per-ray figures will be the same or almost the same for other types of long rays, too.

9 Adoption of Voxels

In this section, we take a look at the feasibility of voxels as a generic geometry representation in quantitative terms and make rough extrapolations into the future regarding the remaining bottlenecks.

statistic	CITY		SIBENIK		SIBENIK-D		HAIRBALL		FAIRY		CONFERENCE	
	no beam	w/beam	no beam	w/beam	no beam	w/beam	no beam	w/beam	no beam	w/beam	no beam	w/beam
Mrays/s	89.1	106.0	94.1	103.6	38.7	48.3	27.9	28.4	145.4	150.8	110.5	124.3
simulated	96.5	133.8	107.1	142.2	46.4	73.5	38.9	41.3	182.4	268.0	122.2	163.0
% of sim.	92.4	79.3	87.9	72.8	83.4	65.8	71.8	68.7	79.7	56.3	90.4	76.3
bw GB/s	33.8	25.5	32.1	23.2	25.6	16.3	12.0	10.3	26.7	14.5	32.3	23.1
coarse %	–	15.9	–	15.5	–	10.3	–	3.6	–	15.3	–	18.2
instructions	2238	1370	1983	1256	3866	1953	2627	2220	1103	563	1766	1109
iterations	29.4	19.1	25.8	17.7	52.8	27.2	34.2	29.5	14.8	8.1	22.4	15.3
intersect	25.4	16.7	22.6	15.2	38.8	21.5	28.0	23.9	11.5	6.4	20.0	13.3
push	19.5	14.7	17.0	13.4	30.6	18.3	18.4	16.7	8.0	5.6	14.1	11.2
store	8.8	7.3	8.0	6.7	12.3	8.3	7.6	7.0	3.5	2.6	6.1	5.0
advance	8.9	3.4	7.8	3.2	21.2	7.9	15.3	12.2	6.5	2.1	7.3	3.0
pop	4.1	1.5	3.9	1.4	10.3	3.7	7.4	6.1	3.3	1.1	3.6	1.3
glob acc	51.4	33.1	45.8	31.0	89.2	47.6	58.7	50.3	23.9	13.5	40.4	27.1
glob bytes	303.9	192.6	270.9	179.5	528.9	272.5	340.0	289.4	142.2	77.1	236.2	153.8
glob trans.	6.0	4.5	6.0	4.9	16.3	11.4	19.7	18.7	3.2	2.5	4.8	3.9
local acc	12.9	8.8	11.9	8.1	22.5	12.0	15.1	13.1	6.8	3.6	9.7	6.3
local bytes	102.8	65.7	95.5	60.6	180.4	90.3	120.5	101.9	54.6	26.4	77.8	45.7
local trans.	1.7	1.4	1.7	1.5	4.7	3.5	6.0	5.8	1.0	0.8	1.3	1.1

Table 6: *Profiling statistics from runs in 1024×768 resolution. Only contour variants have been included in this table. For each scene, the left column corresponds to rendering without beam optimization and the right column with beam optimization. All values are per-ray averages except for the top section that concerns the entire frame. When beam optimization is enabled, the values are calculated by summing the counters over both the coarse frame and the full-resolution frame, and dividing by the number of rays in the full-resolution frame. Note that code execution statistics are calculated on a per-ray basis, and actual values are higher due to SIMD execution of 32 rays in parallel. See text for description of individual rows.*

9.1 Rendering Performance

According to the benchmark results, we can cast approximately 50 million primary rays per second in a moderately detailed scene (SIBENIK-D) using a single GT200 class GPU. This is enough for first-hit rendering in 1080p resolution with 25 frames per second, assuming that shading costs are negligible.

The simplest and reasonably good-looking antialiasing takes 4 samples per pixel, which according to our tests costs about three times as much as taking one sample per pixel. In addition, shadow rays and reflections require additional rays. These secondary rays are less coherent than the primary rays, but based on preliminary tests they seem to cost roughly as much as primary rays. Incoherent secondary rays would undoubtedly be more expensive, but we have not tested those yet.

Assuming that, on the average, four secondary rays are generated per primary ray sample, we end up having five times as expensive samples as in the first-hit-only case. Brute-force antialiasing triples this figure, yielding per-pixel cost multiplier of 15. Extrapolating GPU performance with 50% growth per year, doing 25 frames per second with such rendering would take about 7 years to reach.

This estimate disregards the cost of shading, post-process filtering, and other operations such as data transfers. On the other hand, it is likely that algorithmic improvements will emerge.

9.2 Memory Size

Considering that we can fit over 600 square meters of surface data in 1mm×1mm resolution into 4 GB of memory (Section 3.1), and that memory consumption increases only logarithmically with respect to viewing distance (Section 4), the GPU memory size appears to be less of a concern than might seem at first. However, more complex shading models can increase per-voxel memory usage significantly. Also, it cannot be assumed that every voxel in memory would be stored at exactly the right resolution. On the other hand, occlusion-based on-demand streaming [Crassin et al. 2009] could offer huge savings in terms of memory usage.

In any case, it is comforting that the memory requirements are relatively sensible compared to memory sizes available today. GPU memory size does not seem to be the worst bottleneck in potential adoption of voxel rendering.

9.3 Media Size and Speed

While the amount of GPU memory in today's boards is relatively high, the same cannot be said about the media used for deploying digital content. A dual-layer Blu-Ray disc has capacity of 50 GB, which is just 12.5 times the GPU memory size. Obviously, if the goal is to make the content so detailed that a couple of gigabytes worth of data is used for rendering a single viewpoint, the total amount of data must be much higher.

Unsurprisingly, larger storage media are being researched. *Holographic Versatile Disc* (HVD) is a technology being developed by a consortium of several companies, and it will theoretically have storage capacity of 6 TB. However, the first releases with 1 TB capacity are not planned until 2016. Also, the planned transfer rate is only 125 MB/s, so it would take 32 seconds to completely refresh the contents of a 4 GB video memory assuming zero time wasted in seeking, and much longer when seek time is taken into account.

This estimate assumes that the size of the data on the disk is the same as in GPU memory, and thus neglects the possibility of compressing the data on the disk. It is difficult to estimate what kind of (lossy) compression ratios could be achieved without intolerable decrease in data quality. This is an important practical issue that would deserve a closer look.

Considering the situation today, the only storage media large enough to accommodate a reasonable amount of content is the hard disk drive, where 1 TB of storage can be currently bought for well below $0.10/GB. The data transfer rate is typically less than 100 MB/s, which remains a problem. Solid-state drives can have much better transfer rates, so this problem might get resolved if their price drops significantly.

Without digging deeper into the issue of storage technology, it seems that the size and speed of distribution media are among the biggest challenges in data-intensive content representations.

9.4 Remote Rendering

There is an interesting possibility of sidestepping the storage and media problems by running the game in a dedicated server farm and streaming the rendered graphics to the client. There are ventures in this direction, such as OnLive, OTOY, and Gaikai, who promise high-quality gaming over broadband connection. Despite all kinds of technological challenges associated with such approaches, they could provide something that home gaming cannot by storing tremendously large assets in high-performance RAID clusters.

10 Future Work

An obvious step forward would be experimenting 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.

We have used the simple Phong shading model due to availability of material data in this form, and when downsampling, we combine the shading attributes of multiple primitives by averaging them. This produces reasonably good results in most cases, but it would be better to take occlusion into account when calculating aggregate shading attributes for large voxels. Moreover, it is not at all clear if the Phong model is a good choice for representing the appearance of such voxels. For instance, the approach taken by Gobbetti et al. [2005]—employing a generic shading method that is based on sampling the original data from various directions—seems eminently suitable for our purposes as well.

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 Ingo Wald, University of Utah. Conference room model courtesy of Anat Grynberg and Greg Ward, Lawrence Berkeley Laboratory.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of 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 *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 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.

GOBBETTI, E., AND MARTON, F. 2005. Far Voxels – a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Transactions on Graphics 24*, 3, 878–885.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 167–174.

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.

LAINE, S., AND KARRAS, T. 2010. Efficient sparse voxel octrees. In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*.

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.

RITSCHEL, T., ENGELHARDT, T., GROSCH, T., SEIDEL, H.-P., KAUTZ, J., AND DACHSBACHER, C. 2009. Micro-rendering for scalable, parallel final gathering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia 2009) 28*, 5.

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*.

WEYRICH, T., HEINZLE, S., AILA, T., FASNACHT, D. B., OETIKER, S., BOTSCH, M., FLAIG, C., MALL, S., ROHRER, K., FELBER, N., KAESLIN, H., AND GROSS, M. 2007. A hardware architecture for surface splatting. *ACM Transactions on Graphics (Proc. SIGGRAPH) 26*, 3 (Aug.).

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. 2001. Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 371–378.

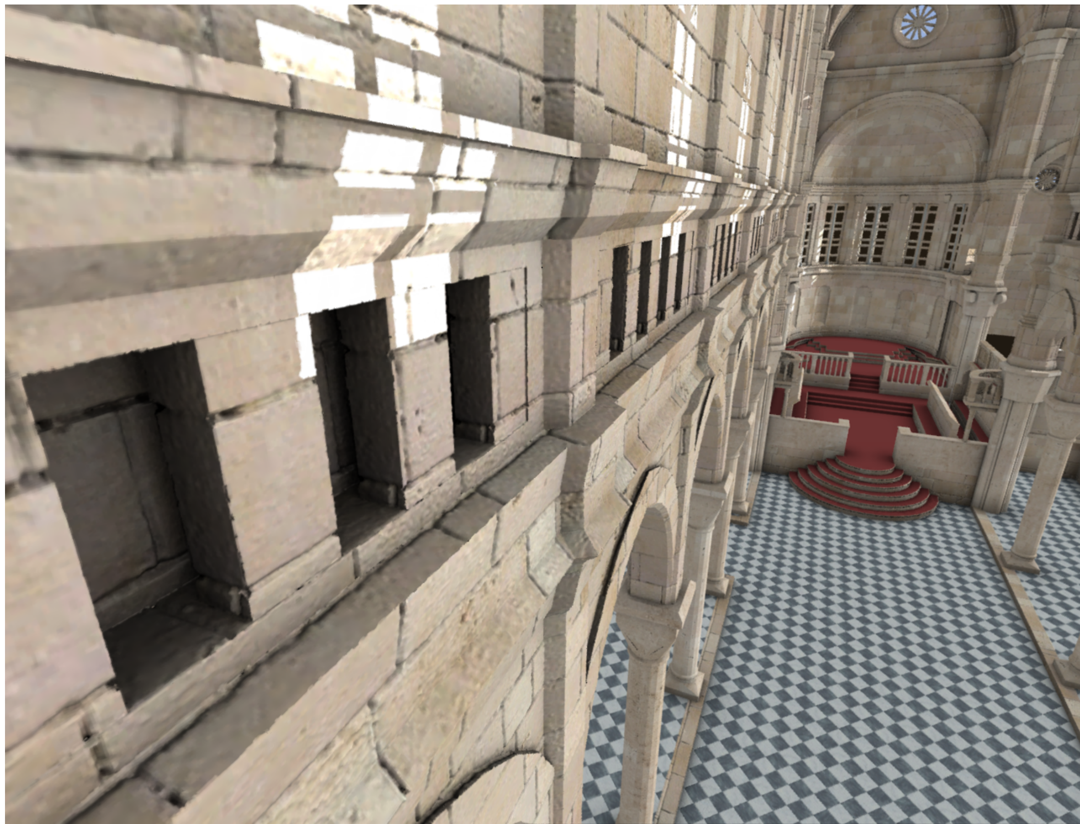
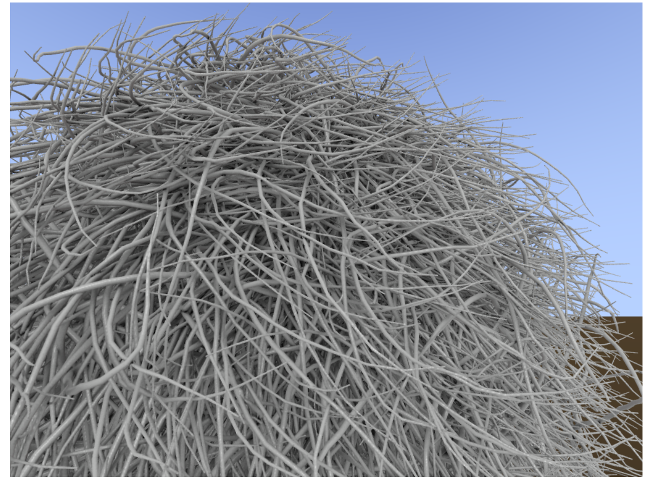


Figure 25: SIBENIK-D *represented as voxels with high-resolution surface displacement and rendered with precomputed ambient occlusion and a single shadow ray per primary ray. Four samples per pixel antialiasing was used with a reconstruction filter that uses 12 samples, including 8 from neighboring pixels. 13 octree levels could be fit in memory, corresponding to voxel spacing of about 5mm.*

City



Hairball

Fairy

Conference

Sibenik-D

Figure 26: *Test scenes used in this paper.*

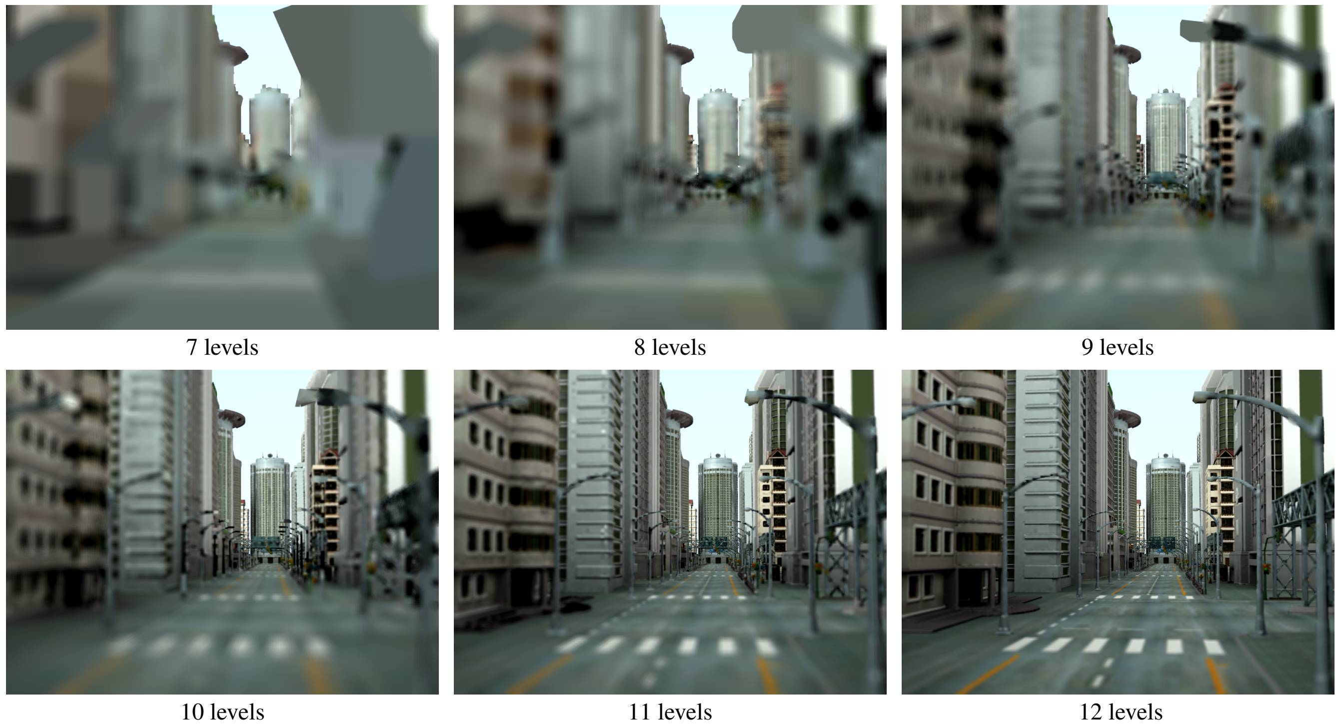


Figure 27: *Voxel renderings of* CITY *using octrees of different depths. Note how only a few levels are required to make the overall appearance adequate for e.g. far-field gathering for global illumination.*

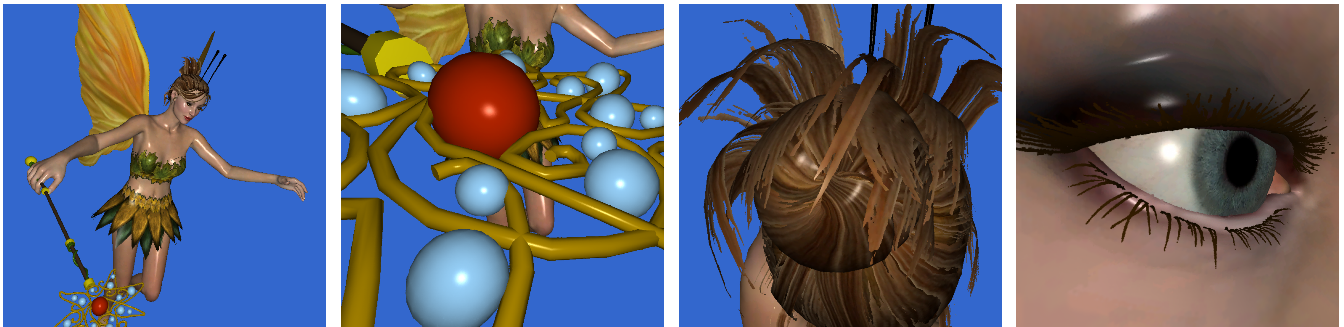


Figure 28: *Voxel renderings of* FAIRY. *The hair close-up highlights the limitations of the polygonal source data.*

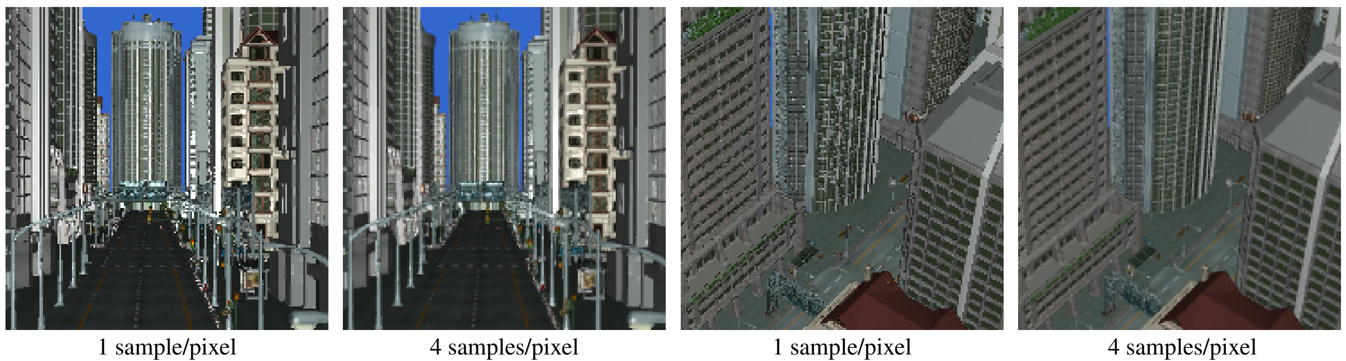


Figure 29: *Effects of antialiasing in* CITY *close-ups. The images are* 180×180 *pixels in size. As can be seen, automatic downsampling of voxels does not remove the need for antialiasing.*

Figure 30: *Effects of post-process filtering. All images are* 768×768 *pixels in size. Voxel resolution is limited in order to make the effect stand out better. The leftmost image is from a car scene that was not used in benchmarks.*

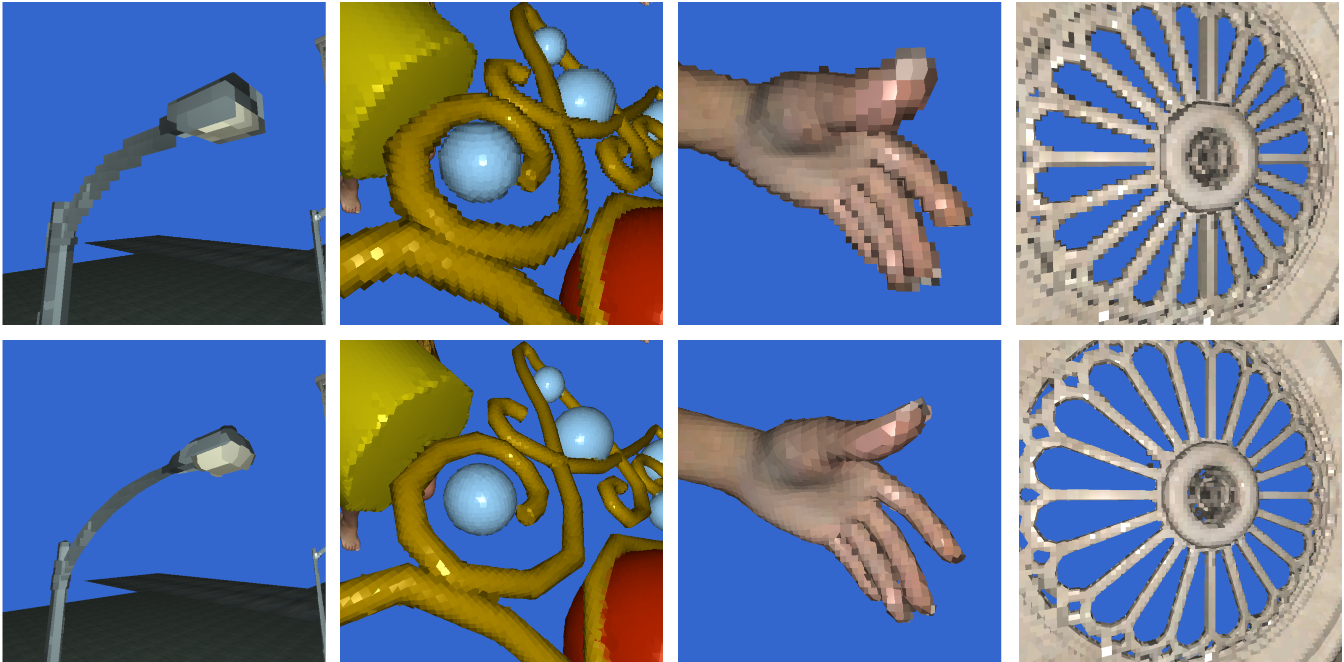


Figure 31: *Voxel renderings without (upper row) and with (lower row) contours. Post-process filtering is disabled to better visualize the individual voxels. Voxel resolution is limited in order to highlight the differences. For example, in* FAIRY *it would be impossible to find a badly undersampled region if all voxel levels were used.*

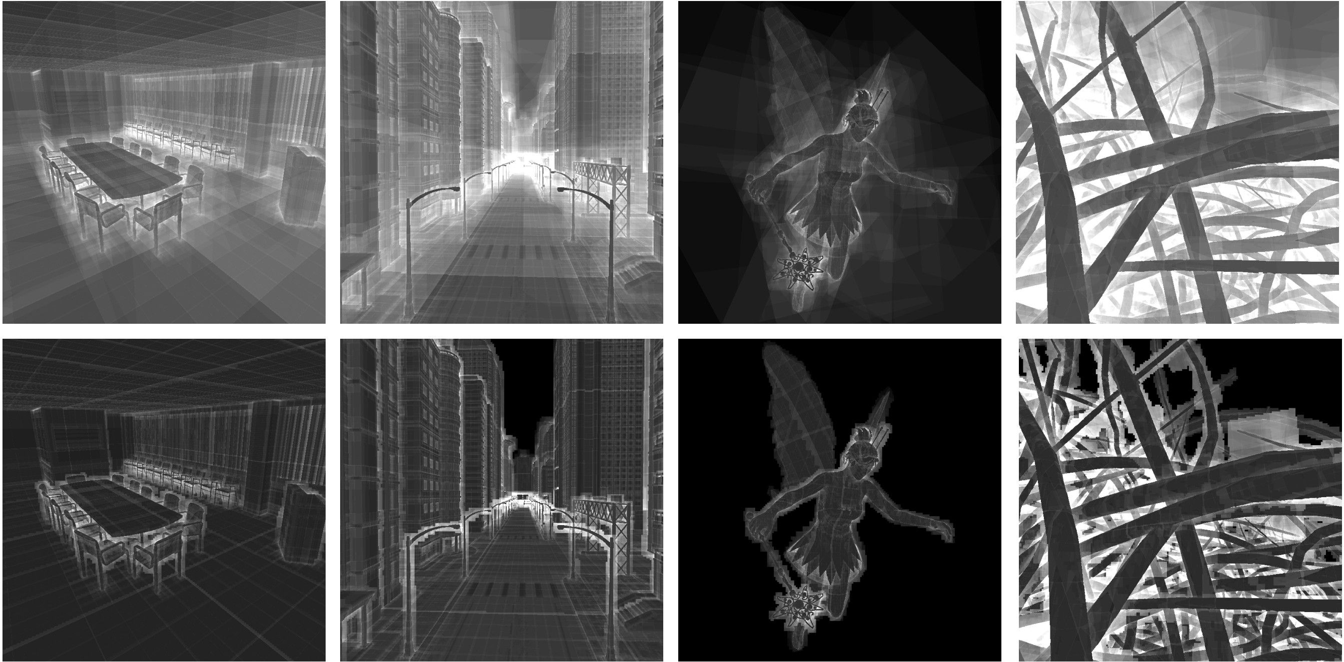


Figure 32: *Iteration count images without (upper row) and with (lower row) beam optimization. The scale from black to white corresponds to 0–64 iterations.*

Appendix A CUDA source for the ray cast algorithm

```
__device__ void cast_ray(
    int*                root,           // In:  Octree root (pointer to global mem).
    volatile float3&    p,              // In:  Ray origin (shared mem).
    volatile float3&    d,              // In:  Ray direction (shared mem).
    volatile float&     ray_size_coef,  // In:  LOD at ray origin (shared mem).
    float               ray_size_bias,  // In:  LOD increase along ray (register).
    float&              hit_t,          // Out: Hit t-value (register).
    float3&             hit_pos,        // Out: Hit position (register).
    int*&               hit_parent,     // Out: Hit parent voxel (pointer to global mem).
    int&                hit_idx,        // Out: Hit child slot index (register).
    int&                hit_scale)      // Out: Hit scale (register).
{
    const int s_max = 23; // Maximum scale (number of float mantissa bits).
    const float epsilon = exp2f(-s_max);

    int2 stack[s_max + 1]; // Stack of parent voxels (local mem).

    // Get rid of small ray direction components to avoid division by zero.

    if (fabsf(d.x) < epsilon) d.x = copysignf(epsilon, d.x);
    if (fabsf(d.y) < epsilon) d.y = copysignf(epsilon, d.y);
    if (fabsf(d.z) < epsilon) d.z = copysignf(epsilon, d.z);

    // Precompute the coefficients of tx(x), ty(y), and tz(z).
    // The octree is assumed to reside at coordinates [1, 2].

    float tx_coef = 1.0f / -fabs(d.x);
    float ty_coef = 1.0f / -fabs(d.y);
    float tz_coef = 1.0f / -fabs(d.z);

    float tx_bias = tx_coef * p.x;
    float ty_bias = ty_coef * p.y;
    float tz_bias = tz_coef * p.z;

    // Select octant mask to mirror the coordinate system so
    // that ray direction is negative along each axis.

    int octant_mask = 7;
    if (d.x > 0.0f) octant_mask ^= 1, tx_bias = 3.0f * tx_coef - tx_bias;
    if (d.y > 0.0f) octant_mask ^= 2, ty_bias = 3.0f * ty_coef - ty_bias;
    if (d.z > 0.0f) octant_mask ^= 4, tz_bias = 3.0f * tz_coef - tz_bias;

    // Initialize the active span of t-values.

    float t_min = fmaxf(fmaxf(2.0f * tx_coef - tx_bias, 2.0f * ty_coef - ty_bias), 2.0f * tz_coef - tz_bias);
    float t_max = fminf(fminf(tx_coef - tx_bias, ty_coef - ty_bias), tz_coef - tz_bias);
    float h = t_max;
    t_min = fmaxf(t_min, 0.0f);
    t_max = fminf(t_max, 1.0f);

    // Initialize the current voxel to the first child of the root.

    int*   parent           = root;
    int2   child_descriptor = make_int2(0, 0); // invalid until fetched
    int    idx              = 0;
    float3 pos              = make_float3(1.0f, 1.0f, 1.0f);
    int    scale            = s_max - 1;
    float  scale_exp2       = 0.5f; // exp2f(scale - s_max)

    if (1.5f * tx_coef - tx_bias > t_min) idx ^= 1, pos.x = 1.5f;
    if (1.5f * ty_coef - ty_bias > t_min) idx ^= 2, pos.y = 1.5f;
    if (1.5f * tz_coef - tz_bias > t_min) idx ^= 4, pos.z = 1.5f;

    // Traverse voxels along the ray as long as the current voxel
    // stays within the octree.

    while (scale < s_max)
    {
        // Fetch child descriptor unless it is already valid.

        if (child_descriptor.x == 0)
            child_descriptor = *(int2*)parent;

        // Determine maximum t-value of the cube by evaluating
        // tx(), ty(), and tz() at its corner.

        float tx_corner = pos.x * tx_coef - tx_bias;
        float ty_corner = pos.y * ty_coef - ty_bias;
        float tz_corner = pos.z * tz_coef - tz_bias;
        float tc_max = fminf(fminf(tx_corner, ty_corner), tz_corner);
```

```
// Process voxel if the corresponding bit in valid mask is set
// and the active t-span is non-empty.

int child_shift = idx ^ octant_mask; // permute child slots based on the mirroring
int child_masks = child_descriptor.x << child_shift;
if ((child_masks & 0x8000) != 0 && t_min <= t_max)
{
    // Terminate if the voxel is small enough.

    if (tc_max * ray_size_coef + ray_size_bias >= scale_exp2)
        break; // at t_min

    // INTERSECT
    // Intersect active t-span with the cube and evaluate
    // tx(), ty(), and tz() at the center of the voxel.

    float tv_max = fminf(t_max, tc_max);
    float half = scale_exp2 * 0.5f;
    float tx_center = half * tx_coef + tx_corner;
    float ty_center = half * ty_coef + ty_corner;
    float tz_center = half * tz_coef + tz_corner;

    // Intersect with contour if the corresponding bit in contour mask is set.

    int contour_mask = child_descriptor.y << child_shift;
    if ((contour_mask & 0x80) != 0)
    {
        int   ofs    = (unsigned int)child_descriptor.y >> 8;       // contour pointer
        int   value  = parent[ofs + popc8(contour_mask & 0x7F)];    // contour value
        float cthick = (float)(unsigned int)value * scale_exp2 * 0.75f; // thickness
        float cpos   = (float)(value << 7) * scale_exp2 * 1.5f;     // position
        float cdirx  = (float)(value << 14) * d.x;                  // nx
        float cdiry  = (float)(value << 20) * d.y;                  // ny
        float cdirz  = (float)(value << 26) * d.z;                  // nz
        float tcoef  = 1.0f / (cdirx + cdiry + cdirz);
        float tavg   = tx_center * cdirx + ty_center * cdiry + tz_center * cdirz + cpos;
        float tdiff  = cthick * tcoef;

        t_min  = fmaxf(t_min,  tcoef * tavg - fabsf(tdiff)); // Override t_min with tv_min.
        tv_max = fminf(tv_max, tcoef * tavg + fabsf(tdiff));
    }

    // Descend to the first child if the resulting t-span is non-empty.

    if (t_min <= tv_max)
    {
        // Terminate if the corresponding bit in the non-leaf mask is not set.

        if ((child_masks & 0x0080) == 0)
            break; // at t_min (overridden with tv_min).

        // PUSH
        // Write current parent to the stack.

        if (tc_max < h)
            stack[scale] = make_int2((int)parent, __float_as_int(t_max));
        h = tc_max;

        // Find child descriptor corresponding to the current voxel.

        int ofs = (unsigned int)child_descriptor.x >> 17; // child pointer
        if ((child_descriptor.x & 0x10000) != 0) // far
            ofs = parent[ofs * 2]; // far pointer
        ofs += popc8(child_masks & 0x7F);
        parent += ofs * 2;

        // Select child voxel that the ray enters first.

        idx = 0;
        scale--;
        scale_exp2 = half;

        if (tx_center > t_min) idx ^= 1, pos.x += scale_exp2;
        if (ty_center > t_min) idx ^= 2, pos.y += scale_exp2;
        if (tz_center > t_min) idx ^= 4, pos.z += scale_exp2;

        // Update active t-span and invalidate cached child descriptor.

        t_max = tv_max;
        child_descriptor.x = 0;
        continue;
    }
}
```

```
        // ADVANCE
        // Step along the ray.

        int step_mask = 0;
        if (tx_corner <= tc_max) step_mask ^= 1, pos.x -= scale_exp2;
        if (ty_corner <= tc_max) step_mask ^= 2, pos.y -= scale_exp2;
        if (tz_corner <= tc_max) step_mask ^= 4, pos.z -= scale_exp2;

        // Update active t-span and flip bits of the child slot index.

        t_min = tc_max;
        idx ^= step_mask;

        // Proceed with pop if the bit flips disagree with the ray direction.

        if ((idx & step_mask) != 0)
        {
            // POP
            // Find the highest differing bit between the two positions.

            unsigned int differing_bits = 0;
            if ((step_mask & 1) != 0) differing_bits |= __float_as_int(pos.x) ^ __float_as_int(pos.x + scale_exp2);
            if ((step_mask & 2) != 0) differing_bits |= __float_as_int(pos.y) ^ __float_as_int(pos.y + scale_exp2);
            if ((step_mask & 4) != 0) differing_bits |= __float_as_int(pos.z) ^ __float_as_int(pos.z + scale_exp2);
            scale = (__float_as_int((float)differing_bits) >> 23) - 127; // position of the highest bit
            scale_exp2 = __int_as_float((scale - s_max + 127) << 23); // exp2f(scale - s_max)

            // Restore parent voxel from the stack.

            int2 stackEntry = stack[scale];
            parent = (int*)stackEntry.x;
            t_max = __int_as_float(stackEntry.y);

            // Round cube position and extract child slot index.

            int shx = __float_as_int(pos.x) >> scale;
            int shy = __float_as_int(pos.y) >> scale;
            int shz = __float_as_int(pos.z) >> scale;
            pos.x = __int_as_float(shx << scale);
            pos.y = __int_as_float(shy << scale);
            pos.z = __int_as_float(shz << scale);
            idx  = (shx & 1) | ((shy & 1) << 1) | ((shz & 1) << 2);

            // Prevent same parent from being stored again and invalidate cached child descriptor.

            h = 0.0f;
            child_descriptor.x = 0;
        }
    }

    // Indicate miss if we are outside the octree.

    if (scale >= s_max)
        t_min = 2.0f;

    // Undo mirroring of the coordinate system.

    if ((octant_mask & 1) == 0) pos.x = 3.0f - scale_exp2 - pos.x;
    if ((octant_mask & 2) == 0) pos.y = 3.0f - scale_exp2 - pos.y;
    if ((octant_mask & 4) == 0) pos.z = 3.0f - scale_exp2 - pos.z;

    // Output results.

    hit_t = t_min;
    hit_pos.x = fminf(fmaxf(p.x + t_min * d.x, pos.x + epsilon), pos.x + scale_exp2 - epsilon);
    hit_pos.y = fminf(fmaxf(p.y + t_min * d.y, pos.y + epsilon), pos.y + scale_exp2 - epsilon);
    hit_pos.z = fminf(fmaxf(p.z + t_min * d.z, pos.z + epsilon), pos.z + scale_exp2 - epsilon);
    hit_parent = parent;
    hit_idx = idx ^ octant_mask ^ 7;
    hit_scale = scale;
}
```