Efficient Rendering of Volumetric Motion Blur Using Temporally Unstructured Volumes

Magnus Wrenninge
Pixar Animation Studios

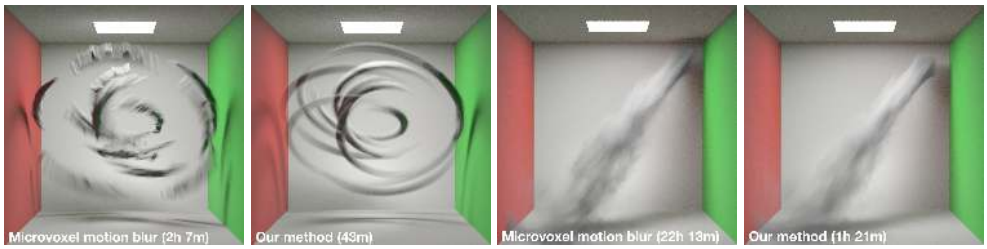

Figure 1. From left: A single volume generated procedurally using our Reves algorithm, rendered with vector-based motion blur (2h 7m); and rendered using our temporal volume method (43m). A smoke simulation using vector-based motion blur (22h 13m); and with our method (1h 21m). Our method improves the rendering speed by 3.0x and 16.4x, respectively, and shows correct blur with smooth, curved motion, while the current state-of-the-art approach exhibits linear overshoot throughout the volume.

Abstract

We introduce *temporally unstructured volumes* (TUVs), a data structure for representing 4D volume data that is spatially structured but temporally unstructured, and *Reves*, an algorithm for constructing those volumes. The data structure supports efficient rendering of motion blur effects in volumes, including sub-frame motion. Reves is a volumetric extension to the classic Reyes algorithm, and produces TUV data sets through spatio-temporal stochastic sampling. Our method scales linearly and maximizes data locality and parallelism both during construction and rendering of volumes, making it suitable for ray tracing, and supports arbitrarily large input models. Compared to the current state-of-the-art in volumetric motion blur, our approach produces more accurate results using less memory and less time, and it can also provide high-quality re-timing of volumetric data sets, such as fluid simulations.

 ISSN 2331-7418

1 Introduction

In production rendering, motion blur is a crucial element. Without it, rendered frames appear to stutter when played back as an animation. The ability to handle motion blur effects accurately and efficiently, along with proper antialiasing, is a fundamental requirement for production rendering. Algorithms that produce correct motion blur for surface rendering are well known, but there are fewer available options for volume rendering.

Volumetric motion blur techniques generally have to balance correctness, speed, and memory use. The current state-of-the-art, microvoxel motion blur [Clinton and Elendt 2009] (see Figure 24 and Section A.2), achieves high visual quality and its memory overhead is low for coherent rays, but performance and memory use degrade as the amount of motion blur increases and as secondary rays require larger portions of the scene to reside in memory. With path tracing becoming a more common production rendering technique, it is desirable to find an approach that is both more robust in the face of heavy motion blur and also handles incoherent ray and screen sampling patterns.

Current techniques generally consider the temporal aspect of volumetric data as being due to motion within the volume, where the motion is defined by a separate volume containing *motion vectors*. We propose a new approach that shifts the viewpoint of volumetric motion blur from a geometric interpretation (an $\mathbb{R}^3$ volume deformed by a vector field) into a purely Eulerian interpretation (a fixed $\mathbb{R}^4$ volume). This change in perspective is fundamental, and it allows for new, more correct, and more efficient ways of rendering volumetric motion blur effects.

The data structure that supports our method is called a *temporally unstructured volume* (TUV). Compared to three-dimensional volume data, our four-dimensional TUVs naturally support motion blur and can store arbitrarily complex motion while maintaining a small memory footprint, even for incoherent rays. In order to achieve high performance, we limit the scope of our work to fixed data sets; once created, there is no dynamic insertion or removal of data. This limitation suits the needs of motion blur in rendering, but is not intended for general storage of temporal volume data.

When volume-rendering complex production scenes, it is often more efficient to combine multiple volume primitives into a single volume, such that the sum of all contributing primitives can be queried with a single lookup. This step, known as volume modeling or volume construction, can also be used to pre-compute complex procedural volume primitives. Present solutions to the volume construction problem provide various ways of generating volume data, but none offer a robust, general-purpose method that supports a wide variety of input model types, nor do they handle motion blur accurately.

Our algorithm for producing TUV data sets, Reves, is a novel solution to the volume construction problem that addresses all of the limitations of current methods: it supports a wide variety of modeling primitives along with SIMD programmable shaders; it has robust antialiasing and texture filtering support; it supports motion blur with an arbitrary number of motion samples; and it scales linearly (in terms of time and memory) with input-model quantity, concurrent execution threads, and output-model resolution.

The shift from seeing volumetric motion as "3D plus motion" to 4D functions also has implications for fluid simulation retiming. Because our method is able to store complex temporal changes in volumetric properties, it is able to capture sub-frame motion in simulation data and later replay the data at novel frame rates without artifacts.

2 Background and Related Work

Motion blur was first introduced in computer graphics by Korein and Badler [1983] and Potmesil and Chakravarty [1983]. Both computed solutions that were continuous and analytic, where Korein and Badler used a method of parameterized motion, and Potmesil and Chakravarty used image-space convolution. Along with their analytic approach, Korein and Badler also mention a non-continuous method whereby images are repeatedly rendered at different time samples and accumulated into a buffer. Although this multi-sample approach works poorly when applied to entire images, it can be seen as a precursor to Cook et al. [1984], who effectively solved motion blur for production rendering, and whose method uses a multitude of point samples distributed in multiple dimensions in order to produce the appearance of motion blur and many other effects.

The types of scene primitives that could be motion-blurred using these early techniques were largely limited to geometry. Korein and Badler and Potmesil and Chakravarty do not mention the rendering of participating media, and although the method described in Cook et al. [1984] can achieve motion blur due to the temporal change in a volume, it does not mention this application.

In fact, when it comes to the context of realistic rendering of participating media, motion blur has often been ignored. Volumes are inherently Eulerian in nature, and the techniques used to describe motion for geometry generally do not apply. Fedkiw et al. [2001] computed the motion of a gaseous fluid and rendered it realistically, but without consideration for the motion of the fluid. Enright et al. [2002] simulated the motion of water using a two-phase fluid approach and signed distance volumes and mentions that sub-frame fluid states may be found by simple interpolation between two signed distance volumes. Zhu and Bridson [2005] later reported that this approach destroys details that are smaller than one simulation voxel.

Kim and Ko [2007] described a technique for computing the sub-frame state of signed distance volumes based on a fluid's velocity information, and, recently, Kulla and Fajardo [2012] mention the use of a similar approach in producing volumetric motion blur effects within a path tracing framework. The last two techniques are both referred to as Eulerian motion blur, in the sense that they produce motion blur of Eulerian volumes. We note, however, that both address a volume-plus-motion problem, rather than a true $\mathbb{R}^4$ representation of volumes. Appendix A.1 provides an outline of these existing Eulerian motion blur techniques.

The field of sound rendering is also related to our method of constructing temporal data sets, and Funkhouser et al. [2003] gives a survey of current methods. In time-resolved imaging, recent work by Jarabo et al. [2014] provides a rendering framework using density estimation of transient signals.

Shader execution systems are a key component of production rendering systems. The Reyes algorithm, described by Cook et al. [1987], not only introduced an efficient rendering system, but also showed how an intermediate rendering primitive (the *micropolygon*) could provide both efficient rasterization as well as opportunity for SIMD shading. Recently, Elendt [2005] provided implementation details of a production-grade shading system.

For production rendering, volumetric motion blur is handled differently in various types of renderers. Several of the systems described by Wrenninge et al. [2011] and Wrenninge and Bin Zafar [2011] use pre-integrated or *smeared* volume data. These produce inaccurate visual results, since shading and illumination are computed on already convolved data. More recently, the microvoxel technique described by Clinton and Elendt [2009] produces high-quality motion blur both due to transformation and deformation of volume data, but it requires a unique position differential at each sample location. Procedurally-generated volumes often include overlapping primitives with differing motion vectors, so storing only a single density and velocity sample per voxel limits its use to fluid simulation data.

A general introduction to the production volume rendering subject can be found in Wrenninge's *Production Volume Rendering* [2012].

Related to production volume rendering is the procedural construction of volume data, often referred to as *volume modeling*. Several different methods exist, as described by Wrenninge [2011] and Bin Zafar [2011]. Most often, these methods are based on direct rasterization of simple primitives, such as spheres, with the addition of procedural, noise-based detail. Ideally, the process that produces TUV data sets should also address the limitations of those volume-modeling systems, such as the lack of support for wide ranges of input primitives and robust motion blur.

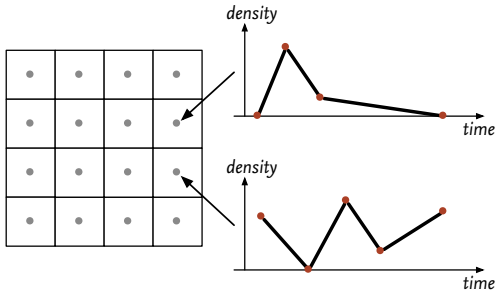

Figure 2. Each voxel in the data structure stores an independent set of sample points, specifying density as a function over time. The number of samples as well as their placement in time can vary from voxel to voxel.

3 Volume Data Structure

Our approach to volumetric motion blur is based on the following observation: When rendering a volume using some form of ray-marching method, we answer queries of the type $\sigma = f(P, t)$, i.e., the property of the medium is defined by some function that varies with both position and time. However, the rendering algorithm never directly queries the *motion* of the volume, i.e., $\partial P/\partial t$, and as such our method focuses on describing volumetric properties where time is simply another sampling dimension.

In order to efficiently sample four dimensions, our representation uses an independent function-over-time for each voxel in the data structure. That is, we preserve the spatial discretization of an ordinary, structured voxel buffer, but add a fourth, temporal dimension that is unstructured, storing an arbitrary number of $\langle$time, value$\rangle$ pairs. By allowing both the sample count and spacing to vary, our data structure can capture arbitrarily fast or slow variations in volumetric properties. Figure 2 illustrates the configuration.

Lookups into our data structure take place in two steps: First, the enclosing 3D voxel coordinate is found, similar to an ordinary voxel buffer. Next, the temporal coordinate t is used to interpolate a value from the function curve stored in the voxel. The sample pairs are sorted, so finding the appropriate interval involves an $O(\log N)$ search; however the number of temporal samples is generally low, resulting in faster lookup performance than both Eulerian and microvoxel motion blur. Appendices A.1 and A.2 describe these differences in more detail.

Our TUV data structure has several implications for volume rendering. First, the temporal dimension is stored in an adaptive fashion that avoids excessive memory use. Second, it stores temporally continuous volumetric data, which can encode motion blur of all types, including material appearing or disappearing over time. Also, for fluid simulations, TUVs can be used to efficiently store every sub-frame simulation time-step, including material that appears and disappears.

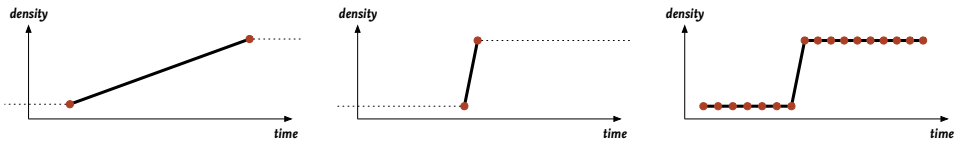

Figure 3. A slow-varying and fast-varying function (left and center) represented using the same number of sample points. Without the ability to specify a time value per-sample, a large number of fixed-time samples (right) would be required to capture the fast-varying function.

3.1 Representation of Data Structure

Our implementation structures the temporal data into N^3-sized *blocks* of voxels. The set of blocks can be organized into either a hash map or a 3D array, permitting both bounded and unbounded volumes to be handled. Blocks are only allocated when in use, so the overall structure is sparse. Using a block structure provides a good tradeoff between spatial adaptivity and fast access, and popular implementations exist in the open source projects Field3D [2009] and, more recently, OpenVDB [Museth 2013]. However, neither of these offer storage of temporally continuous volumes.

In order to accurately capture motion blur effects, a system must be able to process high temporal frequencies. Figure 3 shows that two samples may be sufficient for storing a linearly varying parameter, irrespective of its actual slope and implied frequency, as long as the samples can be placed arbitrarily in time. Without this adaptive placement, the sharp transition would require a large number of samples (and a large memory footprint). We also note that voxels with zero samples are valid (indicating zero value) and provide a good way to take advantage of sub-block sparsity.

Our data structure stores voxel data for a given block of voxels according to the following layout: We maintain one array of time values and a second array of density values, each of which varies in length-per-block from zero upwards. We also store an offset table of length $N^3 + 1$, which indicates the memory location at which the time and density values for a given voxel may be found. We avoid storing an explicit sample count for each voxel by computing it as the difference between consecutive offset values. Figure 4 illustrates this configuration.

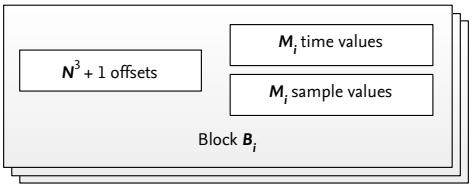

Figure 4. Memory layout for each *block*. Each block $\mathbf{B}_i$ stores a total of N^3+1 offset values, but each block contains a varying number of $M_i \geq 0$ time and sample values.

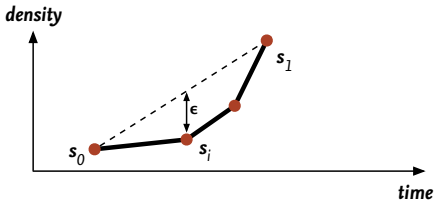

Figure 5. Line simplification step. The error ϵ is measured relative to the magnitude of the interval end points.

3.2 Compression of Temporal Functions

In order to further reduce the memory use of our data structure, we implement two compression steps. Since we assume linear interpolation between temporal samples, any repeated values are removed. As a second step, we apply line simplification to remove the least salient vertices. The data contained in each voxel is similar in structure to that of Lokovic and Veach [2000], but differs in that the functions are non-monotonic. As such, we implement a technique similar to Ramer [1972] and Douglas and Peucker [1973], in which vertices are removed in the order by which they cause the smallest error.

Our functions are one-dimensional, but we treat them as lines in 2D defined by the ⟨time, value⟩ pairs as coordinates. Also, because our method must support volumetric properties in the range $(-\infty, \infty)$ (for example, motion vector fields), we compute an error threshold ϵ_r relative to the magnitude of each interval. That is, for a point $S_i = (t_i, v_i)$ which lies between end points S_0 and S_1, the error threshold is $\epsilon_r = \epsilon_u(v_1 - v_0)$, where ϵ_u is the user-specified threshold (see Figure 5). For zero-value vertices, we set $\epsilon_r = \infty$, as removal of such vertices leads to the introduction of material at a time where none should be present. Apart from these alternative error metrics, our implementation is standard, and the accompanying video (`http://www.jcgt.org/published/0005/01/01/JCGT-TUV.mp4`) illustrates the type of visual artifacts that an excessively large ϵ_u causes.

3.3 Interpolation Methods

We have explored two distinct methods for interpolating the 4D data in TUV data sets. The simplest is of quadrilinear type, where the eight voxels that surround a sample point are found, the temporal curve for each is linearly interpolated, and the results are combined using standard trilinear interpolation.

The second interpolation method is a 4D distance-weighted average. Similarly to the first method, we first find the eight surrounding voxels. Within each voxel, we then find the two nearest temporal samples and combine the resulting 16 values by

weighting each by the 4D distance to the sample point:

$$v = \frac{\sum_{i=1}^{8} \dfrac{v_i}{\max(|(P - P_i)|, \epsilon)}}{\sum_{i=1}^{8} \dfrac{1}{\max(|(P - P_i)|, \epsilon)}}.$$

Higher-order schemes, such as cubic splines, could be implemented along the same lines. However, in our experience within the context of production volume rendering, it is more time-efficient to up-sample data sets using a smooth filter before rendering, and then linearly interpolate the result, since the number of point samples N taken by a path tracer from a voxel buffer of resolution R is generally $N \gg R^3$.

Overall, the cost of the weighted average is $40 \pm 2\%$ greater than the quadrilinear method, and the visual difference between the two approaches is small. In all of our examples we use the quadrilinear method.

4 The Reves Algorithm

Before it is possible to leverage the benefits of TUVs for rendering, we must have a method for constructing them.

Our approach, dubbed Reves, is inspired by the classic Reyes algorithm [Cook et al. 1987]. Our approach allows for the input of both volumetric data (e.g., fluid simulations), as well as most common types of geometry (points, curves, and surfaces). A novel feature, compared to existing volume modeling techniques, is the introduction of an intermediate representation that decouples the input primitives from the process of rasterization into the output voxel buffer, much as micropolygons do in the Reyes framework. All inputs are handled uniformly in the shading and rasterization pipelines, and shaders may be run on all primitive types. The rasterization process itself uses spatiotemporal stochastic sampling rather than simple voxelization, and the separate shading and rasterization steps allow for orthogonal user controls of visual detail versus motion blur- and antialiasing quality.

Overall, the Reves modeling algorithm offers several benefits for volume modeling: The framework allows for a multitude of primitive types as inputs; multi-segment motion blur can be handled efficiently and allows for close approximation of curved motion paths; motion blur effects, being view-independent, are pre-computed and stored in our temporal data structure, which increases rendering efficiency; there is robust antialiasing and texture filtering; and finally, displacement shaders are fully supported. To our knowledge, no previous systems offer the union of all these features.

4.1 List of Terms

Input primitives are the models provided to the system. They may be geometric (e.g., a point cloud or a set of curves) or volumetric (e.g., fluid simulation densities). The

primitives are accompanied by a set of *uniform* (constant for the entire model) or *varying* (unique per model vertex) properties, which are called *primvars* (primitive variables), following RenderMan's convention. The primvars drive the appearance of the model either directly or by binding to shader parameters. Elendt [2005] provides a good resource for how primvars are handled in a shading pipeline, and we follow a similar approach.

Output volumes are the discrete voxel data sets into which the final computed density values are stored. In our case, these are the TUVs described in the previous section. The number of output volumes can vary, from a single one (usually density), to multiple ones. Output volume names are matched by the rasterization engine against *output primvars* (see *Shading vertices* below).

Bucket. A bucket is a contiguous region of voxels in the output volume that is computed concurrently. Generally, the size is set to match the tile size of the output volume's sparse structure.

Voxel sample. Each voxel in a bucket is subdivided into multiple sub-voxels, from which a single point sample is drawn. The point sample has a unique time, t, assigned and is referred to as a *voxel sample*. During rasterization, a set of M^3 voxel samples are used to construct the temporal function curve in each voxel. The value of M influences the quality of both antialiasing and motion blur. Voxel samples are conceptually similar to *pixel samples* in Reyes.

Multi-segment. A multi-segment is a property that has multiple temporal samples. This applies both to input primitives (where geometric primitives express motion by providing multiple time samples of the vertex positions) and to internal, computed properties (e.g., bounding boxes).

Volumetric primitives (vprims) are the internal representation of an *input primitive*. A vprim knows how to compute its own bounds, how to subdivide itself (*splitting*),

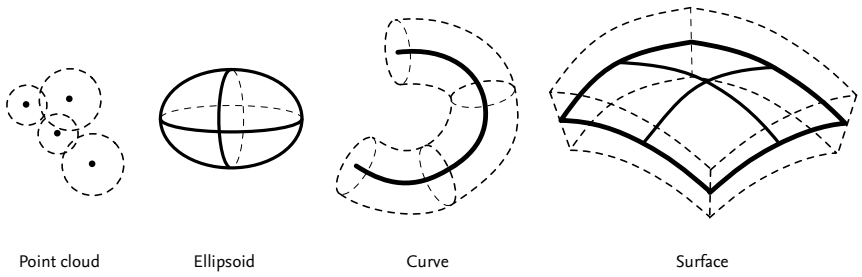

Figure 6. Examples of different types of volumetric primitives (vprims). The point cloud, the curve, and the surface use a *radius* property to define their volumetric extents (dashed lines), whereas the ellipsoid uses its natural volume.

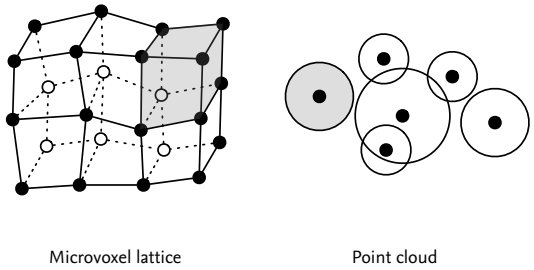

Figure 7. The two types of rasterization primitives (rprims). Each rprim type is made up of *rasterization elements*, which are shown shaded. The rasterization elements are in turn made up of individual *shading vertices*.

and how to *dice* itself into a rasterizable representation. Vprims that are the result of splitting are referred to as *sub-vprims*. Geometric primitives such as point clouds and curves will generally include a *radius* property that define their three-dimensional extents. Vprims are not seen by the rasterization process, and, as such, do not need to provide a point sampling method. Figure 6 shows some examples of vprim types.

Rasterization primitives (rprims) are the intermediate representations that make rasterization of volumetric primitives efficient. They have the primary requirement of being fast compared to point sample, but do not need to perform any of the vprim's duties, such as splitting. Although Reves uses multiple types of rprims, they fall in the same conceptual category as *micropolygons* in the Reyes algorithm. In our implementation, there are two types of rprims: *microvoxel lattices* and *point clouds*, and each of the two is considered to be made up of *rasterization elements*: individual microvoxels and points, respectively. Figure 7 shows the two types.

Shader. A shader is a procedural description of a primitive's appearance. Shaders are generally used to provide detail at a finer level than is practical using geometric subdivision. They are assigned to individual *vprims* but are executed only after the vprim is diced into an *rprim* (see below). Shaders may affect the shading vertex's primvars and also its position, in which case they are referred to as a *displacement shaders*.

Microvoxel lattices are the most common form of rprim present in Reves and are analogous to micropolygon grids in Reyes. Most vprims can easily be broken down into a three-dimensional lattice, on which shaders can be executed, and which can efficiently be point-sampled. Because of its connected nature, the microvoxel lattice provides the shading engine with trivial SIMD opportunities and simple shading-derivative calculations.

Dicing is the process of converting a vprim into its rasterizable form. This step is generally performed in the local space of the vprim and involves transformations from parametric to world space, but not vice versa. For this reason, it is inexpensive even for geometrically complex types. Without an intermediate rasterization primitive, vprims would need to be point-sampled directly, involving expensive inverse transformations.

Shading vertices are the vertices that make up an rprim. They carry information about each vertex's position and can also have an arbitrary number of *output primvars*. The output primvars are read by the rasterizer and are the actual values that are point-sampled and recorded in the temporal functions of each voxel. We note that Reves executes its shaders on rprims, not on vprims. We also note the difference between shading vertices (which relate to the shading sub-system) and rasterization elements (which interact with the rasterization sub-system, and are made up of individual shading vertices).

4.2 Overall Algorithm

The Reves algorithm works in a series of steps that turn an arbitrary number of input primitives into one or more output volumes. For acceleration purposes, we first insert the vprims into an octree. Then, the output volume is divided into *buckets*. A lookup in the octree provides a fast way to bounds-test the input vprims against the bucket's extents. These two steps efficiently partition the inputs and the outputs, such that only the minimal set of inputs that affect a given bucket is considered.

Vprims that intersect the bucket are given the opportunity to *split* into smaller sub-vprims as needed. Once sufficiently small sub-vprims are produced, each one is *diced* into an *rprim*, and its *shader* is executed, which produces and/or modifies an arbitrary number of output primvars at each shading vertex. The shaded rprims are then *rasterized* using spatiotemporal point sampling in order to construct the temporal function in each voxel of the bucket. Once rasterization is complete, the temporal function curves are compressed, and the whole bucket is stored into the block-based output volume. The pipeline is well-suited to threading, and our implementation parallelizes work at the granularity of individual buckets. Algorithm 1 shows a pseudocode implementation of the process, and the following sections outline the details of each step.

4.3 Bounding

Our scene database stores vprims in an ordinary octree structure. In order to handle motion blur efficiently, indexing is performed in raster space rather than world space coordinates, such that the leaf-level nodes of the octree perfectly align with each bucket in the output volume. Because the number of rprims that overlap any given

Algorithm 1 The Reves algorithm.

Precondition: list of input vprims v, list of output buckets b

```
1:  function REVES(v, b)
2:      s ← voxelSize(b)
3:      o ← ⟨empty octree⟩
4:      for each v_i in v do
5:          insert (o, v_i)
6:      for each b_i in b do
7:          r ← ⟨empty list⟩                          ▷ Storage of rprims
8:          for each p_i in lookup(o, b_i) do
9:              if shouldSplit(p_i, s) then
10:                 n ← split(p_i)                    ▷ List of sub-vprims
11:                 erase(o, p_i)
12:                 insert (o, n)
13:             else
14:                 insert(r, dice(p_i, s))           ▷ Insert new rprim
15:         for each r_i in r do
16:             r_i ← shade(r_i)                      ▷ Shade each rprim in place
17:         for each i, j, k in b_i do
18:             c ← rasterize(r, i, j, k)             ▷ Construct temporal curve
19:             b_i(i, j, k) ← compress(c)            ▷ Apply compression
```

bucket may be large, tight bounds-testing is needed in order to reduce memory use and rasterization time. In order to provide tight intersection tests during insertion,

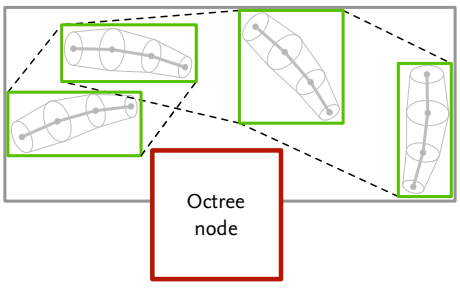

Figure 8. Intersection between an octree node and a multi-segment axis-aligned bounding box (MSAABB, green) for a vprim with multi-segment motion. Although the regular AABB (gray) encompassing the full motion intersects the octree node, the swept bounds of the MSAABB (dashed) provide a tighter and more accurate intersection test.

vprims use a multi-segment axis-aligned bounding box (MSAABB), with one sample per motion sample of the input primitive. The number of motion samples used affects how accurately curved motion can be approximated but does not affect the temporal quality of the final TUV. During octree insertion, vprims are added to any octree node that intersects the swept motion of the MSAABB, as illustrated in Figure 8. We use the moving bounding box test described by Ericson [2004] to test each segment of the MSAABB. Algorithm 2 outlines the insertion process. During lookups, candidate vprims are gathered trivially from the octree by testing each octree node's extents against the the current bucket.

Algorithm 2 Insertion of a vprim into the octree.

Precondition: Octree node o, list of vprims v

```
1: function INSERT(o, v)
2:     for each v_j in v do
3:         for each i in [0, 8) do
4:             if intersect(o_i, v_j) then                ▷ o_i denotes child node
5:                 if size(v_j) < size(o_i) then
6:                     insert(o_i, v_j)                   ▷ Recurse if vprim is small
7:                 else
8:                     addItem(o_i, v_j)
```

4.4 Splitting

Once a vprim is found in the octree, it is queried to see whether a reasonable-resolution rprim would be produced. We use several measures to determine when the appropriate size is found. Ideally, for minimizing cross-task dependencies, we want each shading lattice to intersect as few processing buckets as possible, which suggests small rprims with a small number of shading vertices per rprim. At the same time, splitting excessively results in more individual vprims to manage and insert into the octree and smaller batches for the SIMD shading system to process. Through testing, we have found lattices with 50–150 vertices to be optimal. Figure 9 shows the performance of the system with different average vertex counts per rprim, and Algorithm 3 shows the implementation. From the graph, we draw the conclusion that execution time suffers as vertex counts exceeds 200–300, but that shading and octree lookups become a bottleneck with small vertex counts, negating other gains. The data was captured from the data set in Figure 1, with motion blur disabled. For more complex shaders, the shading time would increase, making very small grids expensive. With motion blur enabled, the octree lookup and the acceleration structure dominate, but they are less influenced by the size of individual shading grids.

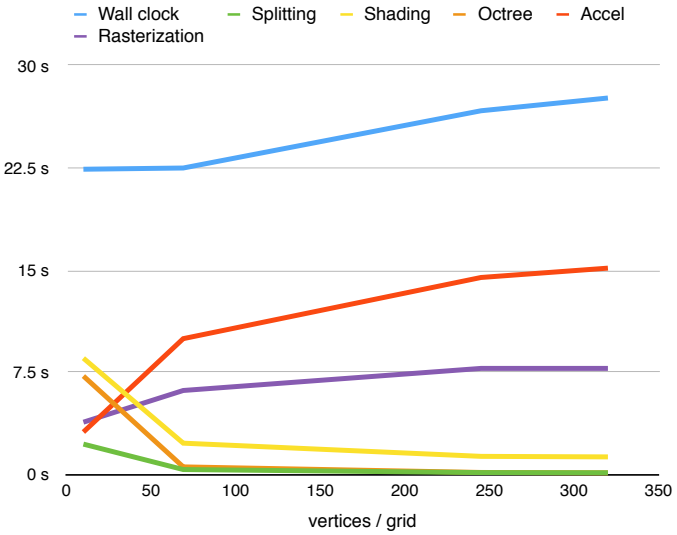

Figure 9. Plot of microvoxel lattice size versus run time for various parts of the Reves pipeline. (All times in seconds.)

Algorithm 3 Split criterion for ordinary vprim.

Precondition: Vprim or sub-vprim v, voxel size s

function SHOULDSPLIT(v, s)
    $d \leftarrow \text{extents}(v)$ ▷ World space isoparm lengths
    $r \leftarrow d/s$ ▷ Relative length
    return $r_x \cdot r_y \cdot r_z > \langle\text{split threshold}\rangle$

Algorithm 4 Splitting an ordinary vprim.

Precondition: Vprim or sub-vprim v

function SPLIT(v)
    $d \leftarrow \text{extents}(v)$ ▷ World space isoparm lenghts
    $r \leftarrow d/s$ ▷ Relative length
    $v_1, v_2 \leftarrow v$ ▷ Clone v twice
    $i_1, i_2 \leftarrow \text{splitIsoparmBounds}(v, \text{longestAxis}(r))$
    $\text{assign}(v_1, i_1)$
    $\text{assign}(v_2, i_2)$
    return $\langle v_1, v_2 \rangle$

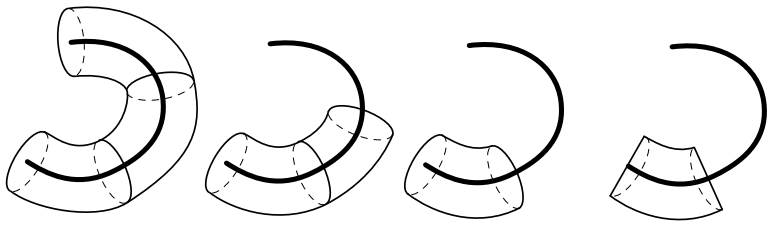

Figure 10. Three successive splitting steps. The first two split the vprim along the $\partial P/\partial u$ direction, whereas the third split happens along $\partial P/\partial v$.

For ordinary vprim types (spheres, curves, surfaces, etc.), splitting is performed in parametric space along isoparametric lines. Each vprim stores a parametric bounding box, initially $[0, 1]$ in each dimension, and each split event is binary, producing exactly two resulting primitives, which are then further split individually, as necessary. As illustrated in Algorithm 4, the axis to split is determined at each iteration by measuring the length of each dimension of the vprim or sub-vprim in world space. This ensures that the shading lattices produced are roughly isotropic. Our only exception to this rule is point clouds, which have no notion of parametric coordinates and which are bisected along the longest world space axis directly. Figure 10 shows three successive steps of splitting a curve vprim. Note that the last step's split direction is orthogonal to the first two. If a vprim is split, the resulting vprims are re-inserted into the octree.

Employing a splitting stage in the pipeline also allows for procedural primitives to perform detail amplification by emitting new geometry on-the-fly. This also allows for memory-efficient handling of input data sets that do not fit in primary memory, as primitives can be discarded from memory as soon a bucket is computed. The results (Section 6) show an example where a white-water element is generated from a point cloud by procedurally instancing line segments.

4.5 Dicing

The purpose of the dicing step is to convert a vprim or sub-vprim into an rprim, such that it's ready for shading to be applied and rasterization to take place. Spheres, ellipsoids, curves, and surfaces are all diced into $N_x \times N_y \times N_z$ resolution microvoxel lattices, such that the vertex spacing is roughly isotropic $\left(|\frac{\partial P}{du}| \approx |\frac{\partial P}{dv}| \approx |\frac{\partial P}{dw}|\right)$ and roughly equal in length to the output volume's voxel size S. The dicing happens along isoparametric lines within the vprim's parametric bounding box (as discussed previously). This yields a structure for which it is simple and efficient to transfer primvars from the vprim onto the lattice, which is required in order for the shading subsystem to work. Likewise, the connectivity of the lattice makes the calculation of shading derivatives straightforward, despite the fact that the shape of the lattice can be irreg-

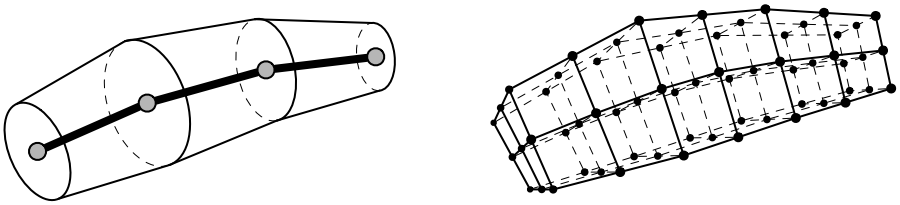

Figure 11. Left: A polygonal curve with varying radius. Right: The result of dicing the curve into a $3 \times 3 \times 7$ microvoxel lattice.

ular. Our method uses simple finite differencing to compute these derivatives. Figure 11 shows a curve vprim diced into a microvoxel lattice rprim. During dicing, the vprim's primvars are transferred to the rprim, using linear interpolation in parametric space. Algorithm 5 outlines the process, which is the same for the aforementioned vprim types.

The point cloud vprim is a special case of the dicing process, in that the vprim itself can be point sampled efficiently. Thus, the dicing step is a simple pass-through of the primvar data.

Algorithm 5 Dicing function.

Precondition: Vprim v, voxel size s

```
1:  function DICE(v)
2:      d ← extents(v)                          ▷ World space isoparm lengths
3:      r ← d/s                                 ▷ Relative length and resolution
4:      M ← newMicrovoxelLattice(r)
5:      for each (i, j, k) in (r_x, r_y, r_z) do
6:          P ← indexToParametric(i, j, k)
7:          W ← parametricToWorld(P)
8:          setLatticePosition(M, i, j, k, W)
9:          for each x in primVars(v) do
10:             x_v ← interpolatePrimVar(x, P)
11:             setPrimVar(M, x, i, j, k, x_v)
12:     return M
```

4.6 Shading

Once an rprim is diced, its vertices are passed to the shading system. The assigned shader reads uniform and/or varying values from input primvars and writes to an arbitrary number of output primvars, including new lattice point positions for displace-

ment shading. The shading system takes advantage of the microvoxel lattice structure to compute accurate derivatives of position or any primvar, as needed. Derivatives can then be used in a shader's texture lookup to perform antialiasing and, in procedural routines, to properly filter out high frequency signals. Because of the vprim/rprim separation, this is supported uniformly for all vprim types, regardless of type.

Again, Cook et al. [1987] and Elendt [2005] provide very useful details on implementing a production-grade shading system, and our method follows their approaches closely. Algorithm 6 outlines the steps.

Algorithm 6 Shading function.

Precondition: Rasterization primitive r, shader s

```
1: function SHADE(r, s)
2:     p ← interpolatePrimVars(r)
3:     d ← computeDerivatives(r, p)
4:     bindPrimvars(p, s)
5:     executeShader(s, r, d)
```

4.7 Rasterization

Our rasterization process is a volumetric extension of the method introduced by Cook et al. [1987]. This is different from simple voxelization, which tends to compute a value per output voxel center. Our technique uses spatiotemporal stochastic sampling to select sub-voxel and temporal coordinates. Any well-behaved 4D blue noise function could be used; in our tests a 4D stratified random set of samples is employed. At its simplest, the process is as follows:

1. For each voxel sample in each voxel, pick a position and time from the sampling distribution.

2. Point sample each overlapping rasterization element at the given position and time, returning a set of output primvar values. We use additive composition for overlapping rprims, but other operations such as $\max$ are also valid.

3. Insert each output primvar value into its corresponding temporal curve.

Interpolation of microvoxels. Each microvoxel in a microvoxel lattice is effectively an eight-vertex cuboid, where each face is described by a bilinear patch. Any parametric interior point $(u, v, w) \in [0, 1]^3$ can be transformed into a position P using a linear combination of its control vertices, as illustrated in Figure 12. Thus, interpolation of a value V could be performed by finding (u, v, w) given P and then computing the value $V(u, v, w)$ using the same linear combination. However, the inversion from P

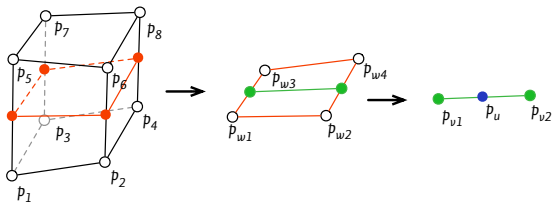

Figure 12. Finding a point P_U in a microvoxel based on (u, v, w) coordinates. Interpolating first along w yields a bilinear patch, then along v to yield a line, and finally along u to yield P.

to (u, v, w) is difficult to solve exactly, and there are many vertex configurations that are non-invertible. Instead, we use a distance-weighted average for interpolating V within the voxel, which avoids having to find (u, v, w) explicitly:

$$V = \frac{\sum_{i=1}^{8} \dfrac{V_i}{\max(|(P - P_i)|, \epsilon)}}{\sum_{i=1}^{8} \dfrac{1}{\max(|(P - P_i)|, \epsilon)}}.$$

Before interpolating values, we must first determine whether a microvoxel overlaps a given sample point. The most accurate inside/outside test would check whether a point P is behind all of the bilinear patches of each microvoxel. For purposes of efficiency, we instead treat each bilinear patch face as two triangles, and test whether P is within the enclosing triangles of the cuboid. Because the size of individual microvoxels are similar to the output volume's voxels, we have not noticed any artifacts in our tests due to the approximation.

Interpolation of point clouds. Our interpolation method for point clouds is straightforward. The user selects one of several kernel functions $k(P)$ (we have found box, triangle, and Gaussian to be sufficient), and the interpolated result is a sum according to

$$v = \sum_{i=1}^{N} v_i \cdot k(P - P_i).$$

In practice, there is no summation over the full set of points in the point cloud; an acceleration structure is built such that only the rasterization elements that overlap a given output voxel are considered. The structure is described below.

Acceleration structure. In order to achieve acceptable rasterization speeds, we employ an acceleration structure to quickly determine which rasterization elements overlap a given output voxel. For each output voxel in a bucket, we store a list of temporal slices, with each slice representing a fraction of the shutter interval. In effect, this forms a dense array of pointers to fixed-length lists of slices. Our goal is to use only

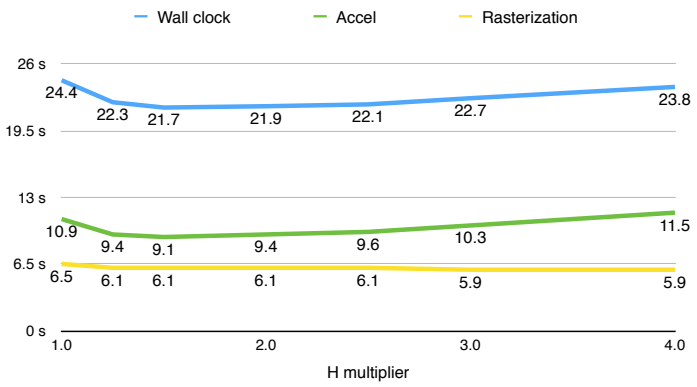

Figure 13. The effect of the square-root heuristic multiplier H on run time

one slice where the input scene is static, but to use progressively more slices as motion is introduced. Each slice stores a list of rasterization elements that need hit-testing for the time segment represented.

Each microvoxel lattice carries bounding information that is multi-segment, i.e. containing a set of ⟨time, AABB⟩ pairs. Given this bounding information, we first find the largest motion magnitude in the rasterization elements, $M_{\max}$. With knowledge of the output voxel size S, we compute a heuristic for how many temporal slices to use according to

$$N = H\sqrt{\frac{M_{\max}}{S}}.$$

The square-root metric balances the amount of work required for acceleration-structure construction against the actual efficiency of the acceleration structure. Through empirical testing, we have found that fewer slices allow too many entries for the acceleration structure to be effective, and a larger number of slices increases the construction time more than it decreases rasterization time. Figure 13 shows the performance of different multipliers on the square-root metric.

To fill the acceleration structure given N slices, we use the following steps: for each slice s, we consider the corresponding time interval T. We first perform an intersection test between the extents of output voxel (i, j, k) and m, which is the portion of the multi-segment bounds of each rasterization element over the interval T. If an intersection occurs, then the element is added to the acceleration structure for the given voxel.

This acceleration structure adapts well to both static and dynamic scenes and ensures that only relevant rasterization elements are tested against each output voxel. In the example shown in Figure 1, the acceleration structure improves performance by roughly 400x with motion blur disabled and 1300x with motion blur enabled. We have

Algorithm 7 Constructing the acceleration structure.

Precondition: List of rprims r, acceleration structure a

```
1:  function CONSTRUCTACCEL(r, a)
2:      M ← findMaxMotion(r)
3:      N ← H√(M/S)
4:      for each s in [0, N) do
5:          T ← [s/N, (s+1)/N]                          ▷ Time range
6:          for each r_i in r do
7:              for each e in rasterizationElems(r_i) do
8:                  b ← discretebounds(e, T)            ▷ Voxel range
9:                  m ← msaabb(e, T)
10:                 for each (i, j, k) in b do
11:                     if movingBBoxIsect((i, j, k), m) then
12:                         append e to a(i, j, k, s)
```

not found any failure cases in our testing, so the acceleration structure is generated and used at all times. Algorithm 7 illustrates our method for construction.

Sampling. Once the list of potential rasterization elements has been determined, stochastic point sampling is used to construct the temporal function that describes how each output primvar evolves over time. By spatiotemporally sampling the microvoxels and point clouds, motion blur effects are captured both due to motion and due to change in the output primvars with time.

Figure 14 illustrates the sampling process in 2D, where a single microvoxel is sampled, and the resulting temporal curve is built. We note that the temporal function is noisy, due to the fact that samples are distributed both spatially and temporally. Ideally, in cases where the rasterization elements are static, no temporal stratification

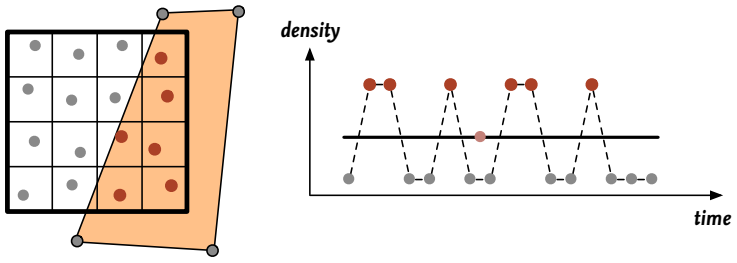

Figure 14. 2D illustration of sampling. A single, static microvoxel is sampled at 4×4 sub-voxel locations and the resulting temporal curve is constructed (dashed). Based on the measured motion, which is zero, the resulting values are filtered down to a single value (solid).

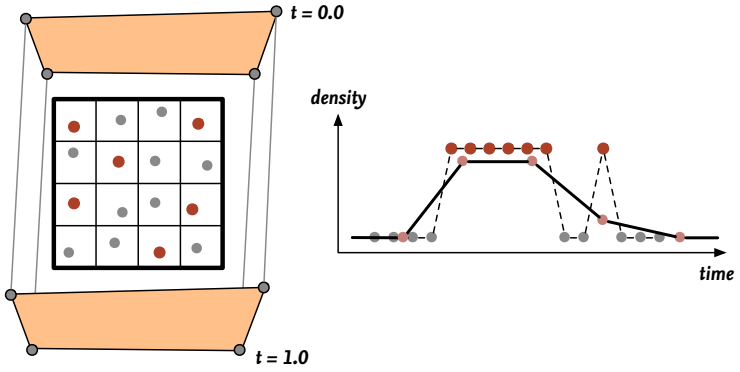

Figure 15. 2D illustration of a microvoxel moving a length of two voxels is sampled at 4×4 sub-voxel locations with the resulting temporal curve constructed (dashed). Based on the motion, five samples ($\frac{M_{\max}}{S_r S_v} = 2$) are used to reconstruct the output temporal curve (solid).

would be necessary, and the supersamples would be used only for spatial antialiasing, and vice versa in cases with fast motion.

We solve this problem by post-processing the temporal curve using information from the acceleration structure. We place N new temporal samples over the shutter interval and re-sample the temporal curve using a $\frac{1}{N}$-wide box filter. In selecting N, we note that the number of temporal samples, N, needed to capture the motion of a rasterization element is a Nyquist-like problem. Given a voxel of size S_v, a rasterization element size S_r, and the maximal motion length $M_{\max}$, we can select the number N to ensure that the element is sampled before, during, and after entering the voxel:

$$N = 2\frac{M_{\max}}{S_r S_v} + 1.$$

Figure 15 shows the before-and-after result of post-processing the temporal function for a moving microvoxel. Compression of the curve, which was discussed in Section 3.2, is applied after the resampling stage.

5 Application to Fluid Simulation Re-timing

We also apply our temporal volume data structure to the area of fluid simulation re-timing. Here, the objective is to construct novel data sets based on existing fluid simulations such that a simulation may be played back at a slower or faster speed. The challenge lies in producing data that lies in-between the pre-computed animation frames. Common techniques for this include approaches such as direct interpolation, and forward-backward advection and blending of adjacent frames. However, even in the best case there is no way to accurately reconstruct the sub-frame state of the simulator, which may have used any number of time steps to compute one frame;

the information is simply lost. In contrast to these techniques, TUVs are able to represent a continuously-varying temporal dimension, avoiding the need to synthesize in-between frames altogether.

In order to produce temporal volume data from fluid simulations, we maintain a cache of all computed sub-frame simulation fields for the duration of one animation frame. When completing a full simulation step, we construct a temporal curve per voxel by taking a number of time samples, determined by a CFL condition[1] [Lewy et al. 1928], and for each time sample computing backward and forward advection of adjacent sub-frame simulation states, such that smooth motion is reproduced. Each sample is recorded into a temporal curve, which is then compressed using the algorithm described in Section 3.2. We note that the CFL condition can be computed per voxel, such that only areas with high velocities require a large number of samples. Algorithm 8 illustrates our method.

Algorithm 8 Constructing a temporal curve from fluid data.

Precondition: List of simulation states s, coordinate P

```
1:  function RECONSTRUCT(s, P)
2:      c ← ⟨empty temporal curve⟩
3:      v ← velocity(s_first, P)              ▷ Velocity from first simulation state
4:      d ← |vΔt|/Δx                          ▷ Length of motion in voxels; CFL
5:      for each n in [0, d) do
6:          t ← (n+ξ)/d                       ▷ Time offset
7:          (s_0, s_1, f) ← findNeighborStates(s, t)   ▷ f is fractional distance
8:          V_0 ← density(s_0, P + v(t0 − t))          ▷ Advect previous state forward
9:          V_1 ← density(s_1, P + v(t1 − t))          ▷ Advect next state backward
10:         record(c, t, lerp(V_0, V_1, f))
11:     return c
```

Using TUVs to store fluid simulation data has several benefits (see Figure 16):

- Motion effects are pre-computed and continuous time is recorded, increasing the speed of rendering.
- There is no need to store a velocity field, which is typically a dense data set that compresses poorly.

[1] In fluid dynamics, a CFL number expresses how many difference-lengths (i.e. voxels) a simulation is allowed to advance in one time step.

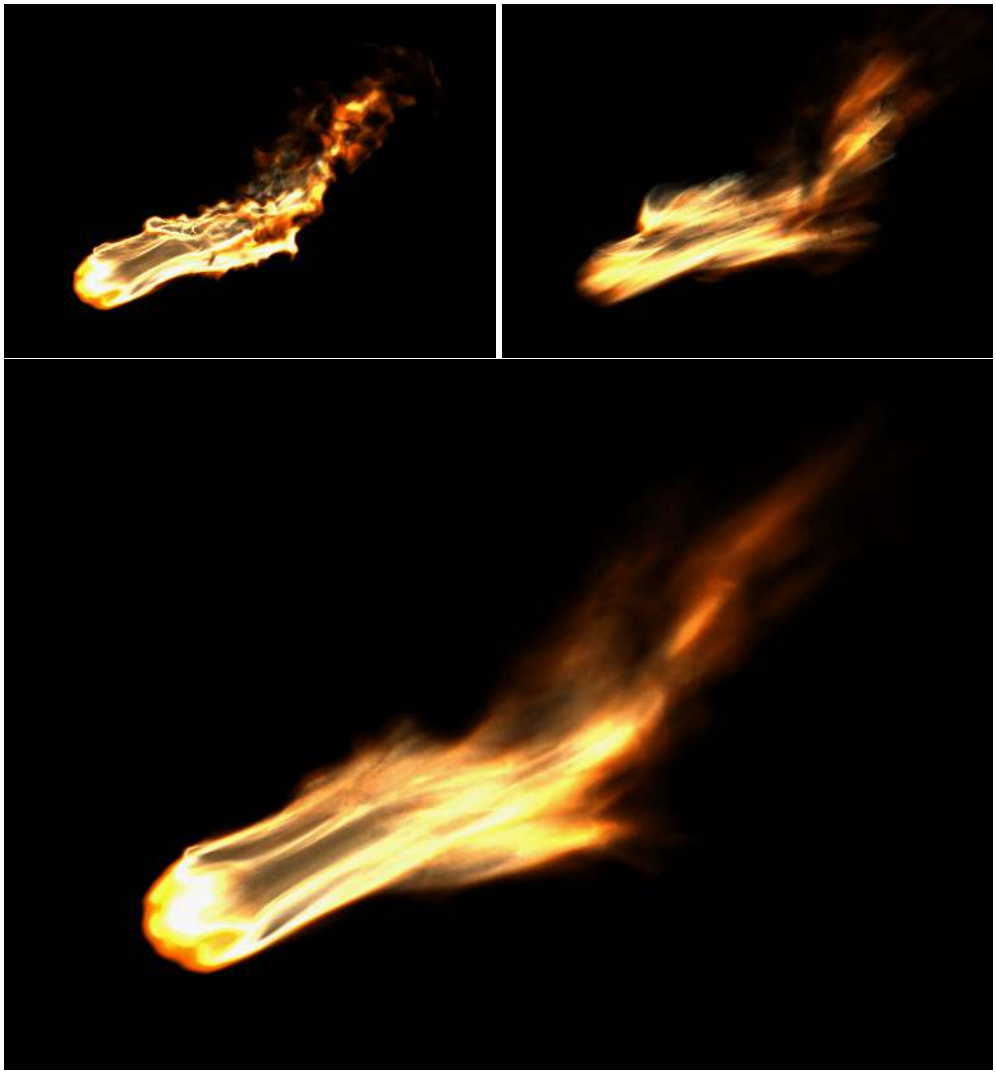

Figure 16. Fire simulation rendered without motion blur (top left, 6m 50s, 1.27GB), with microvoxel motion blur (top right, 20m 41s, 3.39GB), and with TUV data (bottom, 17m 22s, 2.29GB). Motion blur based on the fluid simulation motion vectors produces incorrect results, while our TUV approach captures both the steady state of the source, as well as complex non-linear motion during the shutter open/close interval.

- Re-timing the data is straightforward and produces high-quality results that incorporate the full motion of the original simulation. Because the data structure stores a continuous time dimension, rendering frame 4.5 is no different than rendering frame 4.0. Figure 17 shows how accurate sub-frame data can be extracted from a temporal data set by interpolating each temporal voxel at the chosen sub-frame time.

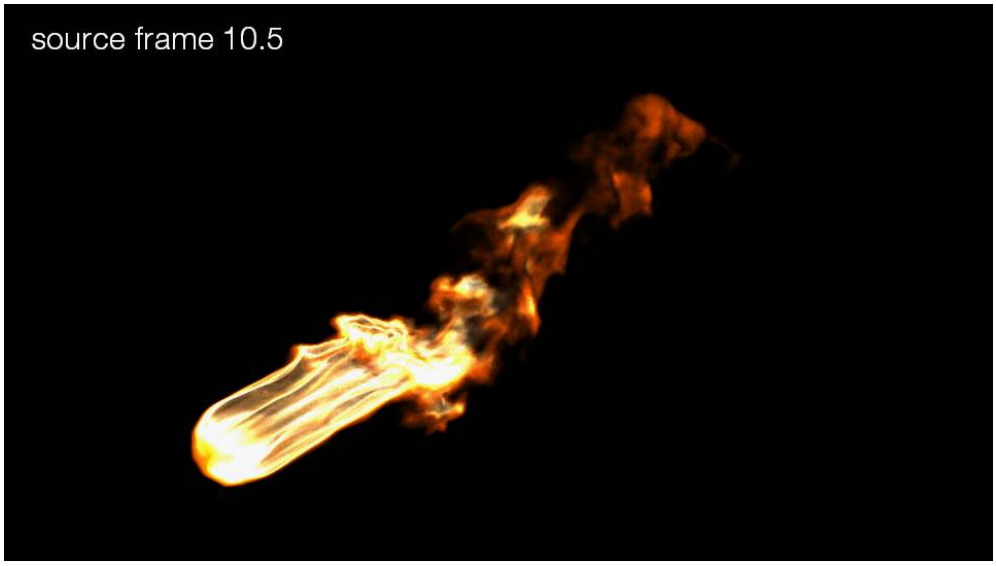

Figure 17. Extraction of sub-frame data from temporal volume. Note the lack of ghosting or other artifacts. The temporal adaptivity of our method allows for high-quality re-timing of fluid simulation data.

6 Results

The images in Figure 18 show data sets generated by our Reves algorithm. The input primitives are eight curves, each with complex, rotational motion and a unique set of shader parameters. The curves are used to produce a $292 \times 384 \times 302$ resolution voxel

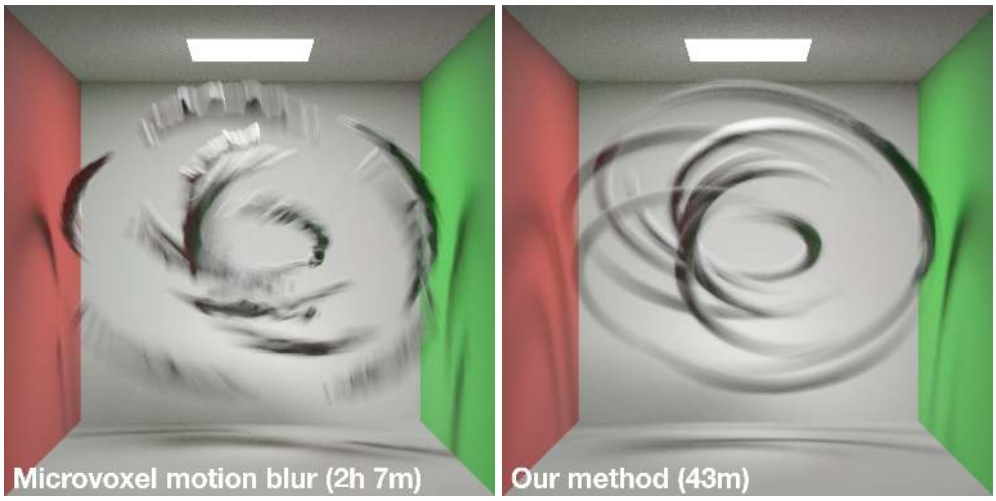

Figure 18. A single volume generated procedurally using our Reves algorithm, rendered with motion vector-based motion blur (left, 2h 7m); and rendered using our temporal volume method (right, 43m)

Figure 19. 8.5 million polygonal lines rasterized into a $8192 \times 8192 \times 4096$ resolution volume.

buffer before rendering. In the first image, motion vectors are computed using central differencing, and the volume is motion-blurred using microvoxel-based motion blur. In the second image, a temporal volume is generated using 32 motion samples and 6^3 voxel samples. Reves constructs the volume data for the first image in 1m 20s. For the second image, the construction time is 14m 21s due to the extreme motion. Rendering of the data set using microvoxel motion blur takes 2h 7m, and using our method the render time is 43m. We note that although our method involves more precomputation for scenes with volumetric motion blur, the time spent generating the temporal volume data is small relative to the time saved during rendering. Although the processing time to produce this particular TUV data set may seem high, we stress that the example is extreme, in order to provide our algorithm with a very difficult motion blur case that the current state-of-the-art method fails to render correctly. The video supplement includes a 240-frame animation illustrating both test cases.

Figure 19 shows the test case we use to illustrate Reves' scaling properties. A point cloud with 4.3 million vertices is used to procedurally generate 8.5 million curve primitives, illustrating Reves' capacity for run-time detail amplification. We expect the Reves algorithm to scale no worse than linearly as we increase the resolution of the output volume (Figure 20(top)). In our experiments, we find super-linear scaling for scenes without motion blur and slightly sub-linear scaling with severe motion blur. The super-linear scaling is due to our various bounding optimizations—as the output volume is refined, intersection tests become more accurate and prune more irrelevant vprims and rprims in the various stages of the algorithm. The memory use shown does not include the storage of the input primitives, which is a constant 312MB (including

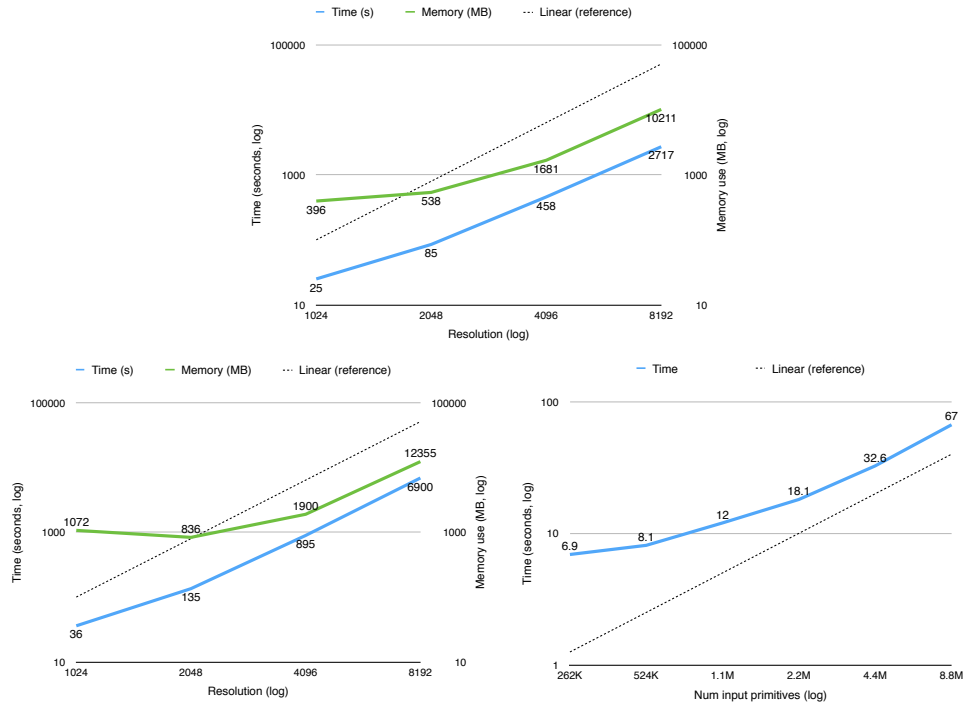

Figure 20. Reves' scaling behavior with increasing resolution without motion blur (top), increasing resolution with motion blur (bottom left), and increasing input primitive count (bottom right). The dashed line indicates linear increase.

primvars), nor the output volume, which increases linearly with the increase in voxel count. The increased memory use for the motion blur case is due to the high number of input primitives: at lower resolutions, many vprims overlap each processing bucket, but as the output volume resolution increases, bucket size decreases, and we are left with fewer vprims to process per bucket, which reduces the amount of data required to reside in memory concurrently.

We also expect Reves' performance to scale linearly with increasing numbers of input primitives. In our tests, we find slightly super-linear performance for increases where the primitive count is low, most likely due to general overhead. As the number of primitives increases, performance becomes more linear. The results in Figure 20 (bottom left) were generated using the same data as in the previous example, constructed at a resolution of 2048^3.

The images in Figure 21 show a 512^3 resolution smoke simulation with fast motion, rendered first using motion vectors (22h 13m) and then using our temporal data structure (1h 21m). In this example, the extreme motion blur causes the microvoxel rendering described by Clinton and Elendt [2009] to consume more memory (4.12GB vs 2.25GB using temporal data) and take significantly longer (16.4x) to render than

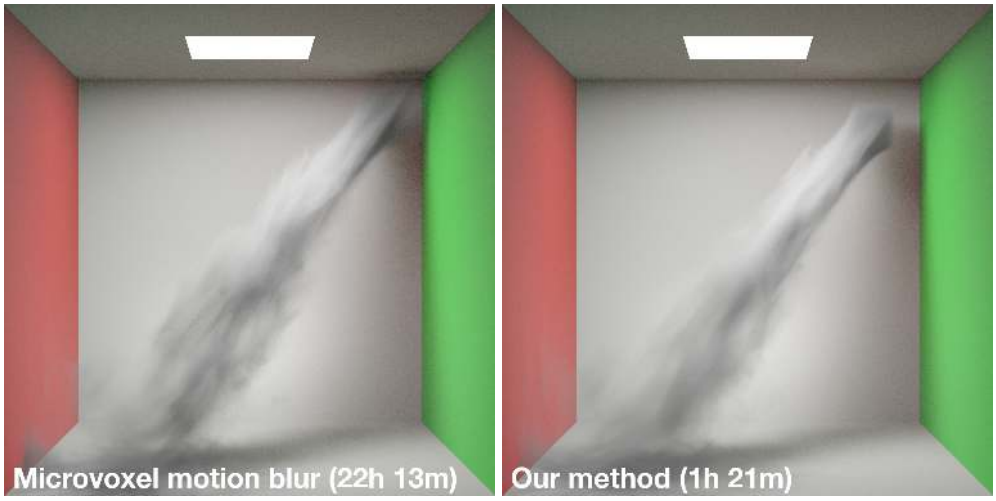

Figure 21. A smoke simulation using microvoxel motion blur (left, 22h 13m); and with our method (right, 1h 21m)

our method. Our method for producing temporal data from fluid simulations does add some overhead, on the same order of magnitude as performing advection. In this test, the simulation took an average of 39s per frame, and with temporal reconstruction the total time including simulation and reconstruction took 44s per frame. Compared to the time savings and increased correctness during rendering, the cost is small. The file size for the ordinary density and velocity volumes were an average of 1.47GB per frame, versus 1.37GB per frame for the density-only TUV using an error threshold of 0.05.

Figure 16 shows a fast-moving flame simulation simulated at 350^3 resolution. The first image (top left) is rendered without motion blur (6m 50s), the second (top right) shows microvoxel-based motion blur (20m 41s), and the third (bottom) shows our method (17m 22s). The motion vector technique shows several types of artifacts: the linear motion path of each microvoxel fails to capture the subtle and curved motion of the original simulation data; also the source itself is motion blurred, even though there is steady, continuous injection of fuel. Our method is able to correctly render the data set with smooth and curved motion and accurately preserving the static source. The video supplement includes a 48-frame animation of the tests. In this example, the speed advantage is smaller for temporal volume data, due to the large number of samples per voxel (30–40 in some voxels); however vastly more information is preserved. The resulting TUV files are larger: 158MB for two ordinary scalar and one vector field and 257MB for two scalar TUVs.

In Figure 17, we use the same data set as in Figure 16 to demonstrate our method's application to fluid re-timing. The fire simulation is slowed down by a factor of 10, and an ordinary voxel data set is extracted for each subframe time. Here, it is clear

Figure 22. Reves as applied to shots from *The Good Dinosaur*. 1000 polygonal curves were used to generate the storm funnel cloud in 2h/frame at $2000 \times 2000 \times 500$ resolution with procedural detail added through shaders (top). 100M particles were rasterized in chunks to cover a 100m tall waterfall at 1cm resolution (middle). Temporal volumes were used to re-time the snow simulation to give effects' artist exact control over evolution speed (bottom).

that our method is able to accurately capture the complex sub-frame motion of the fire. Included in the video supplement are examples of the simulation data rendered at full speed and slowed down using existing interpolation methods, as well as using our TUV structure.

The potential failure case of our temporal data structure is over-compression: it is possible to specify a compression error threshold large enough that too much data is lost to produce accurate motion blur. The last example in the supplemental video shows how the quality of our temporal data degrades as the compression factor is increased from 0.05 to 0.5.

Our implementation of TUVs was done in Field3D, as it provides a flexible framework for adding novel data structures. Rendering was performed in Mantra using physically based shading with one diffuse and one volume bounce, with reference images produced using its native microvoxel method and with our temporal data structure implemented as a procedural volume type. All tests were performed on a 16-core 2.8GHz Xeon workstation and report wall clock time.

7 Production Use

Our system is in active production use, most recently in *The Good Dinosaur*, where Reves was used extensively for volume modeling tasks and temporal volumes were used for re-timing of fluid simulation. Figure 22 shows some examples.

On the upcoming *Finding Dory*, TUVs were used for volumetric motion blur, and as of version 21.0, Pixar's RenderMan natively uses TUVs to provide volumetric motion blur in the RIS framework.

8 Conclusion and Future Work

Our work addresses efficient representation and construction of TUV data sets, most notably in the context of producing motion blur effects. The key idea is to abandon geometric interpretations of volume data when computing motion blur, and instead focus solely on the change that the volumetric properties undergo with time. We show that our technique provides a robust, general, and efficient way to generate and represent volume data with complex sub-frame motion, and that use of the temporal data leads to increased performance and correctness when rendering images with motion blur for film production. Because TUVs require no bounds padding to account for motion blur, they enable efficient path tracing, even for incoherent ray patterns. The fast random access times provided by TUVs also help in rendering of other visual effects that use path tracing, such as multiple scattering and depth of field. Finally, we also show that TUVs provide an attractive method for re-timing fluid simulation data.

Our Reves algorithm offers an efficient and robust way of constructing both TUV and ordinary volumetric data sets. It handles large input models and scales linearly with input and output model complexity. It also offers support for a wide variety of input primitives, multi-segment motion blur, shaders, displacement shaders, texture filtering, and anti-aliasing. The spatiotemporal reconstruction method provides orthogonal control over spatial resolution and the quality of motion blur and anti-aliasing.

In the future, we would like to explore compression techniques that leverage knowledge about the temporal nature of the function curves, as well as GPU methods for accelerating the performance of the stochastic rasterization step. Although our method uses a uniform voxel resolution, the temporal representation should translate well to adaptive-resolution volumes, which would be an interesting area to explore. Finally, we would also like to investigate the use of two-dimensional temporal data sets for image compositing.

A Appendix

A.1 Eulerian Motion Blur

The most common motion blur scheme for ray-tracing and path-tracing renderers is Eulerian motion blur [Kulla and Fajardo 2012]. The scheme assumes that the volume properties are constant with time, and that the only change over time is due to *advection* within the volume. Given a volumetric property f, a point to sample P, a velocity field $\vec{v}$, and a time offset t, Eulerian blur makes the approximation $f(P,t) \approx f(P - \vec{v} \cdot t, 0)$. That is to say, it is assumed that the field is known at

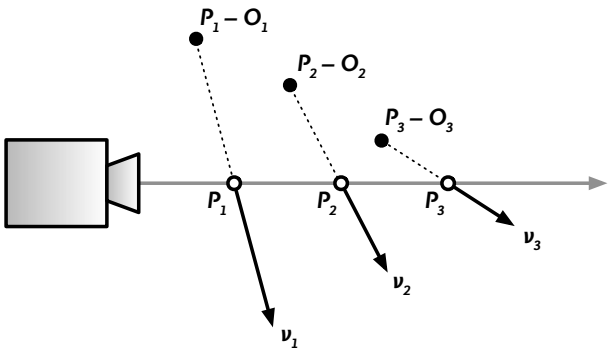

Figure 23. Eulerian motion blur. When querying the volume f at point P_i, we first find the motion vector $\vec{v}_i$, and then offset the query point by the time value $t \in [0,1]$ so that $f(P_i,t) \approx f(P_i - \vec{v}_i \cdot t, 0)$. This computation is similar to advection in fluid dynamics and could also use higher-order approximations.

$t = 0$, and it is gradually deformed by v as t increases. Figure 23 illustrates the process. The method is only approximate, because for any non-constant velocity field, $\vec{v}(P) \neq \vec{v}(P - \vec{v} \cdot t)$, and thus the material at $P' = P - \vec{v} \cdot t$ would never reach P.

When implementing Eulerian motion blur, we note that two lookups are required: First, one to query the vector field, then a second lookup to query the actual density. In extensive testing, we have found the cost of these two $O(k)$lookups to consistently be $30 \pm 3\%$ slower than a single lookup in a TUV, despite the $O(\log N)$ time complexity of the TUV lookup, since N is generally small.

We note that the process is almost exactly equivalent to the *semi-Lagrangian advection* used in fluid dynamics.

A.2 Microvoxel Motion Blur

In the microvoxel motion blur scheme [Clinton and Elendt 2009], the volume is subdivided into a connected lattice of points, with a spacing roughly equal to the projected pixel size. When tracing a ray against the lattice, a sub-frame state that depends on t is found by interpolating the lattice's animated vertex positions (see Figure 24). Although a single lookup into the free-form lattice would be expensive, the cost is amortized when traversing the lattice along a ray, as a DDA-like traversal can be used to decide through which face the ray leaves each microvoxel.

As opposed to Eulerian motion blur, there is not a fixed cost to the microvoxel scheme; it depends on the size and number of microvoxel lattices that each ray needs to traverse. The images in Figure 21 shows a 16.4x advantage for TUVs over microvoxels for an image with moderate motion blur.

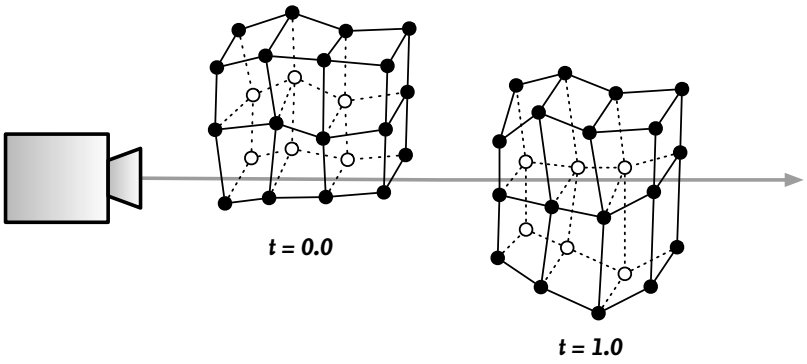

Figure 24. Microvoxel motion blur. At left, we see the lattice at $t = 0.0$, and on the right the lattice has moved and deformed at $t = 1.0$, as defined by the placement of each vertex. When tracing a ray through the volume, the sub-frame state of the volume can be found by interpolating each lattice vertex.

References

CLINTON, A., AND ELENDT, M. 2009. Rendering volumes with microvoxels. In *SIGGRAPH 2009: Talks*, ACM, New York, NY, USA, SIGGRAPH '09, 47:1–47:1. URL: http://doi.acm.org/10.1145/1597990.1598037, doi:10.1145/1597990.1598037. 2, 4, 26, 31

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '84, 137–145. URL: http://doi.acm.org/10.1145/800031.808590, doi:10.1145/800031.808590. 3

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '87, 95–102. URL: http://doi.acm.org/10.1145/37401.37414, doi:10.1145/37401.37414. 4, 8, 17

DOUGLAS, D. H., AND PEUCKER, T. K. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization 10*, 2, 112–122. 7

ELENDT, M. 2005. Shading. In *Production rendering: Design and implementation*, I. Stephenson, Ed. Springer, London, Berlin, Heidelberg, 105–136. 4, 9, 17

ENRIGHT, D., MARSCHNER, S., AND FEDKIW, R. 2002. Animation and rendering of complex water surfaces. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '02, 736–744. URL: http://doi.acm.org/10.1145/566570.566645, doi:10.1145/566570.566645. 3

ERICSON, C. 2004. *Real-Time Collision Detection*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 13

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual simulation of smoke. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, New York, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 15–22. 3

FIELD3D, 2009. v1.4.1. http://sites.google.com/site/field3d. 6

FUNKHOUSER, T., TSINGOS, N., AND JOT, J.-M. 2003. Survey of methods for modeling sound propagation in interactive virtual environment systems. URL: http://www.cs.princeton.edu/~funk/presence03.pdf. 4

JARABO, A., MARCO, J., MUÑOZ, A., BUISAN, R., JAROSZ, W., AND GUTIERREZ, D. 2014. A framework for transient rendering. *ACM Trans. Graph. 33*, 6 (Nov.), 177:1–177:10. URL: http://doi.acm.org/10.1145/2661229.2661251, doi:10.1145/2661229.2661251. 4

KIM, D., AND KO, H.-S. 2007. Eulerian motion blur. In *Proceedings of the Third Eurographics conference on Natural Phenomena*, Eurographics Association, Aire-la-Ville, Switzerland, NPH'07, 39–46. doi:10.2312/NPH/NPH07/039-046. 4

KOREIN, J., AND BADLER, N. 1983. Temporal anti-aliasing in computer generated animation. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '83, 377–388. URL: http://doi.acm.org/10.1145/800059.801168, doi:10.1145/800059.801168. 3

KULLA, C., AND FAJARDO, M. 2012. Importance sampling techniques for path tracing in participating media. *Comp. Graph. Forum 31*, 4 (June), 1519–1528. doi:10.1111/j.1467-8659.2012.03148.x. 4, 30

LEWY, H., FRIEDRICHS, K., AND COURANT, R. 1928. Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen 100*, 32–74. URL: http://eudml.org/doc/159283. 22

LOKOVIC, T., AND VEACH, E. 2000. Deep shadow maps. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '00, 385–392. doi:10.1145/344779.344958. 7

MUSETH, K. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph. 32*, 3 (July), 27:1–27:22. URL: http://doi.acm.org/10.1145/2487228.2487235, doi:10.1145/2487228.2487235. 6

POTMESIL, M., AND CHAKRAVARTY, I. 1983. Modeling motion blur in computer-generated images. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '83, 389–399. URL: http://doi.acm.org/10.1145/800059.801169, doi:10.1145/800059.801169. 3

RAMER, U. 1972. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing 1*, 3, 244 – 256. URL: http://www.sciencedirect.com/science/article/pii/S0146664X72800170, doi:http://dx.doi.org/10.1016/S0146-664X(72)80017-0. 7

WRENNINGE, M., AND BIN ZAFAR, N. 2011. In *Production Volume Rendering 1: Fundamentals*, ACM, New York, NY, USA, ACM SIGGRAPH 2011 Courses. 4

WRENNINGE, M., BIN ZAFAR, N., HARDING, O., GRAHAM, G., TESSENDORF, J., GRANT, V., CLINTON, A., AND BOUTHORS, A. 2011. In *Production Volume Rendering 2: Systems*, ACM, New York, NY, USA, ACM SIGGRAPH 2011 Courses. 4

WRENNINGE, M. 2012. *Production Volume Rendering: Design and Implementation*. CRC Press, Boston, MA, USA. 4

ZHU, Y., AND BRIDSON, R. 2005. Animating sand as a fluid. In *ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, SIGGRAPH '05, 965–972. URL: http://doi.acm.org/10.1145/1186822.1073298, doi:10.1145/1186822.1073298. 3

Index of Supplemental Materials

A video supplement illustrating the techniques described in this paper can be found at http://www.jcgt.org/published/0005/01/01/JCGT-TUV.mp4.

Author Contact Information

Magnus Wrenninge, Pixar Animation Studios, Emeryville, CA
magnus@pixar.com

Magnus Wrenninge, Efficient Rendering of Volumetric Motion Blur, *Journal of Computer Graphics Techniques (JCGT)*, vol. 5, no. 1, 1–34, 2016
http://jcgt.org/published/0005/01/01/

Received:	2015-08-11		
Recommended:	2015-10-12	Corresponding Editor:	Peter Shirley
Published:	2016-01-26	Editor-in-Chief:	Marc Olano