Ray Tracing of Signed Distance Function Grids

Herman Hansson Söderlund
NVIDIA

Alex Evans
NVIDIA

Tomas Akenine-Möller
NVIDIA

Figure 1. A path traced image of two orange signed distance function models mixed with triangle geometry.

Abstract

We evaluate the performance of a wide set of combinations of traversal and voxel intersection testing of signed distance function grids in a path tracing setting. In addition, we present an optimized way to compute the intersection between a ray and the surface defined by trilinear interpolation of signed distances at the eight corners of a voxel. We also provide a novel way to compute continuous normals across voxels and an optimization for shadow rays. On an NVIDIA RTX 3090, the fastest method uses the GPU's ray tracing hardware to trace against a bounding volume hierarchy, built around all non-empty voxels, and then applies either an analytic cubic solver or a repeated linear interpolation for voxel intersection.

1. Introduction

Rendering signed distance functions (SDFs) is becoming more and more popular due to their simplicity and expressiveness with a small amount of data. Impressive three-dimensional scenes can be defined using only code, where SDFs are used with operators, such as union, intersection, smooth subtraction, and smooth union [Bloomenthal et al. 1997], and are often demonstrated on websites, such as ShaderToy. Entire

 ISSN 2331-7418

games, such as *Dreams* [Evans 2015] and *Claybook* [Aaltonen 2018], use SDFs extensively. In those cases, the SDF primitives and operators are sampled onto a three-dimensional grid, where each corner of a grid cell holds a signed distance. We call these *SDF grids*. The bounding box of such a grid can be rasterized and the content sphere traced [Hart 1995] to reveal its containing geometry. Sphere tracing has a weakness in that it takes smaller and smaller steps the closer the iteration gets to the actual surface. This reduces performance just before a ray hits a surface, when a ray is close to a silhouette, and when secondary rays are to be shot from a hit point on an SDF surface, when using path tracing, for example. One may use acceleration techniques for sphere tracing [Keinert et al. 2014; Bálint and Valasek 2018], but those often only pay off when the number of iterations is large. Segment tracing [Galin et al. 2020] can provide generous speedups, but this method has not yet been extended to SDF grids.

We present a method to analytically ray trace the surface inside each voxel in an SDF grid. The surface inside a voxel is a third-order polynomial as was derived by Parker et al. [1998]. Our method uses substantially fewer operations than theirs. In addition, Parker et al. did not provide continuous normals between voxels, which gives poor visual quality when the viewer is close to the voxels. We present a remedy to this.

Before we describe our work, we want to emphasize the surprising (at least to us) expressive power of possible surfaces of the third-order polynomial inside a voxel. When triangulating SDF grids, one may use the marching cubes algorithm [Lorensen and Cline 1987], which states that 14 topological situations (excluding empty) can occur in a voxel. However, when using trilinear interpolation of the signed distances inside a voxel, the surface becomes a third-order polynomial, which has a much wider set of topologies. Lopes and Brodlie [2003] showed that there are 31 different topologies. We show a few in Figure 2.

2. Analytic Voxel Intersection

Our data structure is a three-dimensional grid, consisting of $n_x \times n_y \times n_z$ locations, where each location holds a signed distance. This data structure is called a *signed distance function* (SDF) grid, or simply a grid. A *voxel* within such a grid is a box in three-dimensional space with $2 \times 2 \times 2$ signed distance values specified at its corners.

In this section, we formulate how to intersect a ray with the surface defined by the zero level set of the signed distance function inside a voxel, as shown in Figure 3. This part of our paper is similar to the work of Parker et al. [1998], but with constants grouped for fewer computations and better use of fused-multiply-and-add (FMA) operations. Given $2 \times 2 \times 2$ distance values s_{ijk} with $i, j, k \in \{0, 1\}$, in a single voxel, the standard equation for trilinear interpolation is

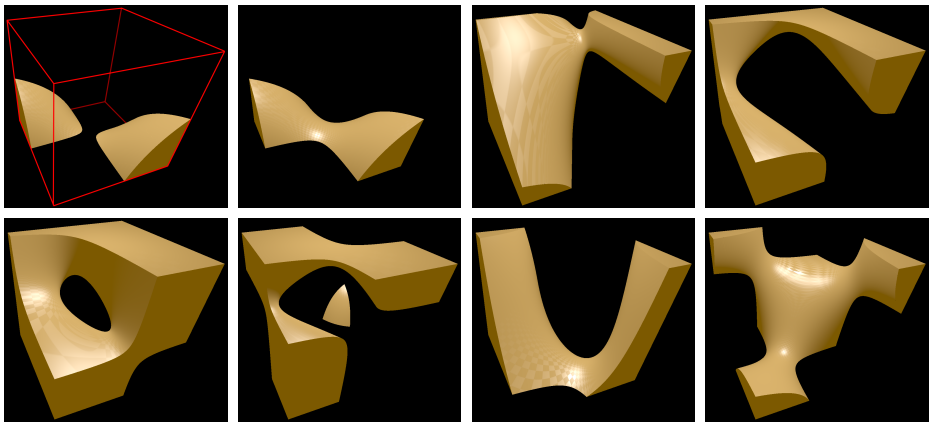

Figure 2. Each image shows the surface inside a single voxel, which has a signed distance at each of the $2 \times 2 \times 2$ corners of the voxel. In the upper left image, all signed distances are positive, except for two opposing corners of the bottom quadrilateral. This case is part of the marching cubes algorithm. However, the second image on the top row has the same configuration, except the numbers are a bit different. In this case, a single surface is shown inside the voxel. Marching cubes does not generate geometry correctly for this case. The rest of the images show various cases that marching cubes does not handle correctly.

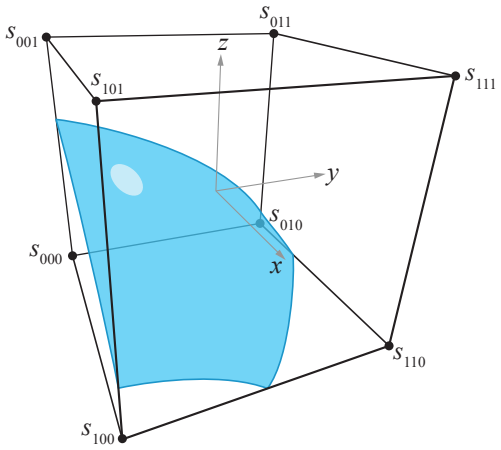

Figure 3. A voxel with signed distances s_{ijk} at the $2 \times 2 \times 2$ corners. A possible surface is shown in blue.

$$
\begin{aligned}
f(x,y,z) = \\
(1-z)\Big((1-y)\big((1-x)s_{000} + xs_{100}\big) + y\big((1-x)s_{010} + xs_{110}\big)\Big) \\
+ z\Big((1-y)\big((1-x)s_{001} + xs_{101}\big) + y\big((1-x)s_{011} + xs_{111}\big)\Big),
\end{aligned}
\tag{1}
$$

where $x, y, z \in [0, 1]$. This equation is evaluated at each step in algorithms based on sphere tracing [Hart 1995].

The surface inside a voxel is defined by $f(x, y, z) = 0$ (Equation (1)), which can be rewritten as

$$f(x, y, z) = z(k_4 + k_5x + k_6y + k_7xy) - (k_0 + k_1x + k_2y + k_3xy) = 0, \quad (2)$$

which is a polynomial of degree three because the highest-order term is xyz. The constants k_i are functions of the s_{ijk} distances:

$$\begin{aligned} k_0 &= s_{000}, & k_4 &= k_0 - s_{001}, \\ k_1 &= s_{100} - s_{000}, & k_5 &= k_1 - a, \\ k_2 &= s_{010} - s_{000}, & k_6 &= k_2 - (s_{011} - s_{001}), \\ k_3 &= s_{110} - s_{010} - k_1, & k_7 &= k_3 - \big(s_{111} - s_{011} - a\big), \\ & & a &= s_{101} - s_{001}. \end{aligned} \quad (3)$$

A ray is defined by $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, where $\mathbf{o} = (o_x, o_y, o_z)$ and similar for $\mathbf{d}$. The intersection between the surface and the ray is found by replacing x, y, and z with the ray components, e.g., replacing x with $r_x(t) = o_x + td_x$, in Equation (2). This results in

$$\begin{aligned} &(o_z + td_z)\big(k_4 + k_5(o_x + td_x) + k_6(o_y + td_y) + k_7(o_x + td_x)(o_y + td_y)\big) \\ &- \big(k_0 + k_1(o_x + td_x) + k_2(o_y + td_y) + k_3(o_x + td_x)(o_y + td_y)\big) = 0, \end{aligned} \quad (4)$$

This expression can be rewritten as

$$c_3t^3 + c_2t^2 + c_1t + c_0 = 0, \quad (5)$$

where

$$\begin{aligned} c_0 &= (k_4o_z - k_0) + o_xm_3 + o_ym_4 + m_0m_5, \\ c_1 &= d_xm_3 + d_ym_4 + m_2m_5 + d_z(k_4 + k_5o_x + k_6o_y + k_7m_0), \\ c_2 &= m_1m_5 + d_z\big(k_5d_x + k_6d_y + k_7m_2\big), \\ c_3 &= k_7m_1d_z, \end{aligned} \quad (6)$$

and

$$\begin{aligned} m_0 &= o_xo_y, & m_3 &= k_5o_z - k_1, \\ m_1 &= d_xd_y, & m_4 &= k_6o_z - k_2, \\ m_2 &= o_xd_y + o_yd_x, & m_5 &= k_7o_z - k_3. \end{aligned} \quad (7)$$

The method presented by Parker et al. [1998] for computing the c_i in Equation (5) is rather brute-force, though mathematically elegant, in their presentation. Their method uses about 161 operations, where fused-multiply-and-adds were counted as

one operation. By sharing repeated calculations (the k_i and m_i) where possible, our method uses only 37 operations, again counting each FMA as one operation.

To intersect a ray with the surface inside a voxel, we first compute the intersection between the ray and the box. This intersection point is used as the new origin of the ray, unless the ray origin is inside the box. The origin of the ray is then transformed to the canonical voxel space, which is $[0, 1]^3$, as we have derived our surface function in that space. The distance from this point to the exit point of the ray on the box is denoted t_{far}.

Next, we compute the intersection distance, t, by using Equation (5) with the constants in Equations (3), (6), and (7). Note that we are only interested in the first real root inside the box, i.e., with $t \in [0, t_{\text{far}}]$. There are several viable approaches to solving that third-order polynomial. First, one may use an analytic cubic polynomial solver based on Vieta's approach [Press et al. 2007], or any other analytic method for solving cubic polynomials.

Second, one may use a numerical approach instead of an analytic solver for the third-order polynomial. As a first step, we use the approach by Marmitt et al. [2004]. Let us denote our cubic polynomial by $g(t) = c_3t^3 + c_2t^2 + c_1t + c_0$. Marmitt et al. differentiate this equation and solve for equality to zero, which gives us $g'(t) = 3c_3t^2 + 2c_2t + c_1 = 0$. This can be solved analytically by splitting the interval $[0, t_{\text{far}}]$ into subintervals wherever $g'(t) = 0$. The subintervals are processed in order, from $t = 0$ toward $t = t_{\text{far}}$. As soon as we find a subinterval $[t_{\text{start}}, t_{\text{end}}]$, where $g(t_{\text{start}})g(t_{\text{end}}) \leq 0$, i.e., $g(t_{\text{start}})$ and $g(t_{\text{end}})$ have different signs, we know that there will be a root of $g(t)$ in $[t_{\text{start}}, t_{\text{end}}]$. This is illustrated in Figure 4.

We also found that this type of computation can be used to optimize shadow ray testing. For opaque geometry, shadow ray testing can be terminated as soon as a hit is found in $[0, t_{\text{light}}]$. This means that we can exploit the previous step, i.e., if we find that there is a root in $[t_{\text{start}}, t_{\text{end}}]$ and this interval fully overlaps with $[0, t_{\text{light}}]$, then we can report a hit without numeric iteration. This is evaluated in Section 4.

If we detect that there is a root in $[t_{\text{start}}, t_{\text{end}}]$, using the method in Figure 4, one alternative is to use a numerical solver to find the root in that subinterval. Marmitt et al. applied repeated linear interpolation to find the root. Another alternative is to first refine the current t into $t = (g(t_{\text{end}})t_{\text{start}} - g(t_{\text{start}})t_{\text{end}})/(g(t_{\text{end}}) - g(t_{\text{start}}))$, which provides a refined initial guess of the root. Next, we use the Newton–Raphson method [Press et al. 2007] to find the root. This is shown in Listing 1. We evaluate the analytic solution, the method of Martmitt et al., and the Newton–Raphson method in Section 4.

If desired, one can add an additional test, which makes the voxel surface solid, as shown in Figure 2, before using the cubic solver. When the ray origin is located on a face of the voxel's box, we evaluate Equation (2) once at the ray origin. If $f(o_x, o_y, o_z) < 0$, then the ray is deemed to have hit a side of the voxel, and we return

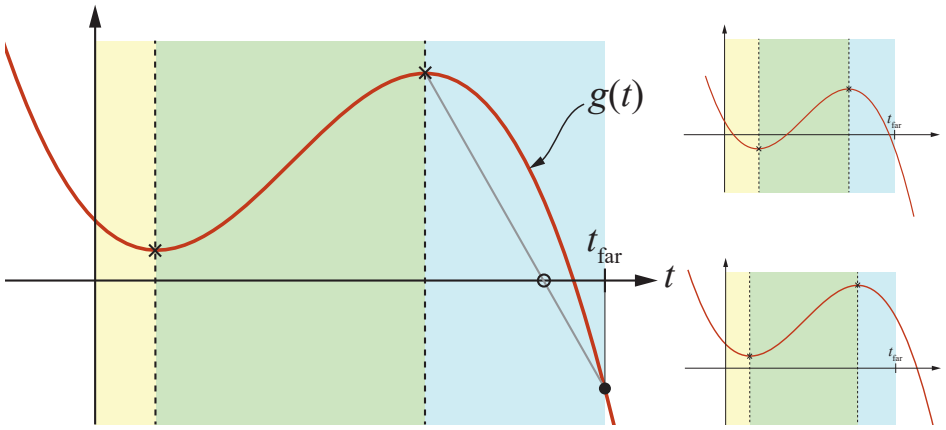

Figure 4. Illustration of how the solutions to $g'(t) = 0$ help with numeric root finding. The roots to $g'(t) = 0$ are marked with vertical dashed lines, which split the t-axis into three intervals. Since the curve is above the t-axis in the first two intervals (yellow and green), there cannot be any real roots there. For the last interval, the curve is above the t-axis at the beginning of the interval and below at t_{far}. Using linear interpolation, we then compute the t-value at the non-filled circle and use that as a starting point for Newton–Raphson iteration (NR). The top right example shows how there are three roots in $[0, t_{\text{far}}]$, but NR will start in the first interval due to opposite signs of $g(t)$ at the start and end of the first (yellow) interval. The bottom right example shows that there are no roots, and so no iteration is needed.

```
for(int q=0; q < maxNumSteps && abs(t-tPrev) >= kNumericEpsilon; q++)
{
    float gt = evalCubic(coefficients, t);              // 3 FMAs
    float gt_deriv = evalQuadratic(quadraticCoeffs, t); // 2 FMAs
    tPrev = t;
    t -= gt / gt_deriv;                      // Newton-Raphson step
}
// At this point, the approximation of the root is t.
```

Listing 1. Refining the root using Newton–Raphson iteration, where `evalCubic()` evaluates the cubic polynomial $g(t)$, while `evalQuadratic()` evaluates the derivative of $g(t)$, i.e., $g'(t)$.

a hit there immediately. An advantage of adding this test is that rays which "sneak" between the surfaces of two neighboring voxels due to floating-point imprecision are likely to immediately hit such a box face, which avoids visible cracks.

3. Normals

For any continuous surface defined by a differentiable implicit function f, a vector $\mathbf{n}$, which is normal to the surface, can be computed from the gradient of f, i.e., $\mathbf{n} = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})$. Common numerical methods for computing $\mathbf{n}$ include central differencing [Frisken et al. 2000], forward and backward differencing, or the tetra-

hedral method, using only four function evaluations, by Falcão [2008]. All of these numerical methods are based on some epsilon value to determine offset points.

We compute normals using the analytic derivative of f, as described in Section 3.1. Because f is a cubic polynomial, the normal will be C^2 continuous inside each voxel. However, over an SDF grid, the normals will not be continuous, since two neighboring voxels' surfaces are only C^0 continuous where they meet. To remedy this, we present a method (Section 3.2) that interpolates normals from neighboring voxels' surfaces, which is similar in spirit to how normals are computed for PN triangles [Vlachos et al. 2001]. To clarify, our approach computes continuous normals that appear to be reasonable, but they do not correspond to the normals of the surface, similar to the method by Vlachos et al.

3.1. Analytic Normal

Differentiation of Equation (1) with respect to x gives

$$\begin{aligned}\frac{\partial f(x,y,z)}{\partial x} = &\quad (1-y)\,(1-z)\,(s_{100}-s_{000})\\ &+\quad y\quad (1-z)\,(s_{110}-s_{010})\\ &+(1-y)\quad z\quad (s_{101}-s_{001})\\ &+\quad y\quad\quad z\quad (s_{111}-s_{011}).\end{aligned} \tag{8}$$

This is a bilinear interpolation of distance differences in the x-direction, which we can rewrite as

$$\begin{aligned}y_0 &= \texttt{lerp}(y, s_{100}-s_{000}, s_{110}-s_{010}),\\ y_1 &= \texttt{lerp}(y, s_{101}-s_{001}, s_{111}-s_{011}),\\ \frac{\partial f(x,y,z)}{\partial x} &= \texttt{lerp}(z, y_0, y_1),\end{aligned} \tag{9}$$

where $\texttt{lerp}(u,a,b) = a + u(b-a)$ is linear interpolation on an FMA-friendly form. Similary, we get

$$\begin{aligned}x_0 &= \texttt{lerp}(x, s_{010}-s_{000}, s_{110}-s_{100}),\\ x_1 &= \texttt{lerp}(x, s_{011}-s_{001}, s_{111}-s_{101}),\\ \frac{\partial f(x,y,z)}{\partial y} &= \texttt{lerp}(z, x_0, x_1),\end{aligned} \tag{10}$$

and

$$\begin{aligned}x_0 &= \texttt{lerp}(x, s_{001}-s_{000}, s_{101}-s_{100}),\\ x_1 &= \texttt{lerp}(x, s_{011}-s_{010}, s_{111}-s_{110}),\\ \frac{\partial f(x,y,z)}{\partial z} &= \texttt{lerp}(y, x_0, x_1).\end{aligned} \tag{11}$$

The analytic normal for the surface inside a voxel is then $\mathbf{n} = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})$, using Equations (9)–(11).

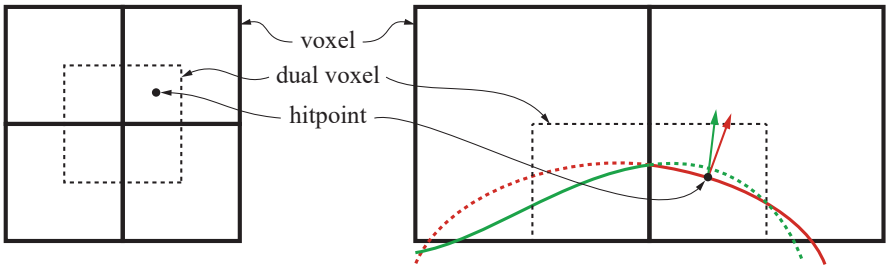

Figure 5. Left: A 2×2 voxel grid and its corresponding dual voxel in the middle. Each voxel has its own surface, and normals can be evaluated outside its voxel. For a hit point, we compute the normals for each of the four voxels at the same hit point and then weight these based on the position of the hit point inside the voxel. Right: To avoid cluttering the illustration, we only show two voxels here. The illustration shows a green (respectively, red) surface defined by the signed distance values at the corners of the left (respectively, right) voxel. Normals, shown in red and green, can be computed at the hit point based on the implicit function defined in both voxels. These normals are then weighted based on the position of the hit point inside the dual voxel. In this simple illustration, only the x-component of the hit point would be used for weighting. Because the hit point is closer to the right border of the dual voxel than the left border, the red normal will get a larger weight than the green normal.

Our method uses about 30 operations for the normal (without normalization), while the method of Parker et al. uses 54 operations.

3.2. Continuity Across Voxels

One could imagine that normals, which are computed and shared at voxel corners, are trilinearly interpolated inside the voxel volume. Though this would give normal continuity at the shared borders between voxels, the quality would suffer and such a method would not be sufficient to compute reasonable normals for the voxels in Figure 2, for example.

Instead, we take the following approach. We introduce the concept of a *dual voxel*, which is just a voxel shifted in location by half the voxel dimensions. This is shown in Figure 5. Any hit point will fall inside a single dual voxel, which in turn overlaps $2 \times 2 \times 2$ voxels. We evaluate the analytic normal $\mathbf{n}_{ijk}$ in each voxel at the hit point and interpolate the results. Our interpolation scheme takes place inside the dual voxel, weighted using the triplet $(u, v, w) \in [0, 1]^3$, which represents the hit point's position within the dual voxel. Our formula for interpolation resembles that of trilinear interpolation in that we use

$$\begin{aligned}
\mathbf{n} = \quad & (1-u)\,(1-v)\,(1-w)\,\mathbf{n}_{000} + u\,(1-v)\,(1-w)\,\mathbf{n}_{100} \\
& + (1-u) \quad v \quad (1-w)\,\mathbf{n}_{010} + u \quad v \quad (1-w)\,\mathbf{n}_{110} \\
& + (1-u)\,(1-v) \quad w \quad \mathbf{n}_{001} + u\,(1-v) \quad w \quad \mathbf{n}_{101} \\
& + (1-u) \quad v \quad w \quad \mathbf{n}_{011} + u \quad v \quad w \quad \mathbf{n}_{111},
\end{aligned} \tag{12}$$

with the subscript indices indicating from which voxel a normal is computed. We chose to write the expression in the clearest form in Equation (12), but in reality, it is better to evaluate it using a form similar to that of Equation (1). Our method differs from standard trilinear interpolation in that we do not compute normals at the corners of the dual voxel, but instead compute the (normalized) analytic normals, using Equations (9)–(11), evaluated at the hit point. This is illustrated to the right in Figure 5. Note that this means that normals are computed outside the usual domain of a voxel in seven out of the eight cases, and only inside for the voxel where the hit point is located. As can be seen to the right in Figure 5, the green normal is not computed on the green surface, but rather at the hit point on the red surface.

The resulting interpolated normals are smoothly varying across the whole field; however, they do not correspond exactly to the original normals of the intersected surface, which are discontinuous at voxel boundaries. In other words, we trade a small deviation from the geometric normal to the surface for smoothness across voxel boundaries.

4. Results

All tests were run on an Intel Core i9 10980XE clocked at 3.00 GHz with 128 GB of DDR4 system memory clocked at 2.4 GHz. The graphics card was an NVIDIA RTX 3090 with 24 GB of memory using driver version 471.68. Our implementation was done inside an in-house path tracer running in Falcor [Kallweit et al. 2021].

We have used four test scenes called Cheese, Goblin, Heads, and Ladies, which can be seen in Figure 6. All rendering was done with full path tracing using up to three bounces and with a single square light source. Performance was evaluated using different camera paths for our four scenes, with 900 frames per scene. After 30 seconds of warm-up rendering on the GPU, we ran the camera path five times and computed the average frame time.

4.1. Algorithms

To provide a comprehensive study of the performance of path tracing of SDF grids, we have divided the algorithm into a *grid traversal* part and a *voxel intersection* part.

4.1.1. Grid Traversal

The grid traversal methods we evaluate are

1. grid sphere tracing (GST),
2. sparse voxel set (SVS),
3. sparse brick set (SBS), and
4. sparse voxel octree (SVO).

Figure 6. Three frames from each of our test scenes at different viewpoints from our camera paths.

Grid sphere tracing GST is sphere tracing through a dense grid of SDF values. The grid that GST uses stores all voxels for a given resolution, this includes those voxels that do not contain the SDF surface. Values are formatted and clamped to $[-1, 1]$ such that a value of 1 encodes 2.5 interior voxel diagonals. This allows the SDF values to be stored as 8-bit `snorm` integers. To accelerate traversal of empty regions of space, a hierarchy of SDF grids is utilized. Lower-resolution levels of the hierarchy encode larger SDF values. An algorithm adaptively selects a grid resolution to safely maximize the step size taken by the sphere tracing traversal. The traversal is done entirely in shader code. A similar technique is used in *Claybook* [Aaltonen 2018].

Sparse voxel set SVS creates an axis-aligned bounding box (AABB) for each voxel that intersects part of the surface and builds a bounding volume hierarchy (BVH) for use in DirectX Raytracing (DXR). Hence, BVH traversal is done using the GPU's RTX ray tracing hardware. When a voxel is reached, a custom intersection shader is invoked and any of the following methods can be used. Instead of storing $2 \times 2 \times 2$ signed distance values per voxel, SVS stores $4 \times 4 \times 4$ signed distance values around a voxel. This allows easy access of neighboring voxel values, which is necessary for some of the normal evaluation methods that follow.

Sparse brick set SBS is similar to SVS, except that each AABB stores the data of $7 \times 7 \times 7$ voxels ($8 \times 8 \times 8$ values), called *bricks*. When the custom intersection shader is invoked, a 3D digital differential analyzer (DDA) is used to visit the relevant voxels in order. SBS stores the distance values of each 8^3 brick as slices in a 2D texture and provides a lookup buffer to access the bricks. This allows eight voxel corner values to be fetched using only two HLSL `gather` operations. This traversal method is most similar to that used by *Dreams* [Evans 2015] and OpenVDB [Museth 2013; Museth 2021]; OpenVDB uses a forest of trees to store leaf bricks with data for $8 \times 8 \times 8$ voxels. OpenVDB does not directly make use of GPU `gather` operations and therefore does not need to duplicate the voxel data at brick borders.

Sparse voxel octree SVO is a sparse voxel octree [Crassin et al. 2009] with implementation details following the work by Laine and Karras [2010]. All traversal occurs in shader code for SVO. In general, we store SDF values as 8-bit `snorm` integers. SDF values are formatted such that a value of 1 encodes 0.5 interior voxel diagonals. After formatting, the SDF values are clamped to $[-1, 1]$, allowing them to be stored as `snorm` integers. The size of a voxel varies at different levels of the octree, which allows lower-resolution levels to encode larger distances at the expense of precision.

4.1.2. Voxel Intersection

We evaluate the following methods:

1. grid sphere tracing (GST),
2. sphere tracing (ST),
3. analytic (A),
4. Marmitt et al. (M), and
5. Newton–Raphson (NR).

Grid sphere tracing GST for voxel intersection is only combined with GST for grid traversal, meaning that the same algorithm is used until an intersection is found.

Sphere tracing ST performs classical sphere tracing only inside the voxel and can be combined with other traversal methods. To prevent extremely long runs, GST and ST stops after 512 and 128 iterations,[1] respectively, or when there is a crossing from outside to inside the surface.

[1] We use a larger value for GST because it uses sphere tracing for both traversal *and* voxel intersection, while ST only uses it for voxel intersection. Note that these numbers are mostly there to prevent extremely long runs, but we seldom see this happening.

Cheese						**Goblin**					
	GST	ST	A	M	NR		GST	ST	A	M	NR
GST	15.4	22.6	14.6	15.7	15.7	GST	16.6	22.2	16.0	16.9	17.0
SVS	–	16.6	**9.7**	**9.9**	10.4	SVS	–	14.9	**9.7**	**9.8**	11.2
SBS	–	30.3	17.8	19.1	18.6	SBS	–	25.5	17.3	18.1	18.0
SVO	–	28.4	16.7	17.3	17.3	SVO	–	24.3	16.3	16.6	16.6
Heads						**Ladies**					
	GST	ST	A	M	NR		GST	ST	A	M	NR
GST	50.8	63.9	47.9	50.8	50.8	GST	79.7	93.2	82.9	86.3	86.3
SVS	–	27.2	**17.3**	**17.1**	18.0	SVS	–	23.6	**16.1**	**16.2**	17.3
SBS	–	53.8	32.8	34.8	33.9	SBS	–	44.6	30.9	32.3	31.3
SVO	–	59.4	40.7	41.4	41.6	SVO	–	66.1	54.8	54.8	55.3

Table 1. Main performance results, in milliseconds, for our four scenes called Cheese, Goblin, Heads, and Ladies. The traversal method (GST/SVS/SBS/SVO) is identified by the row and the voxel intersection method (GST/ST/A/M/NR)) by the column. The fastest two methods per scene are marked with bold numbers.

Analytic We solve the cubic polynomial using an analytic cubic root solver, as mentioned in Section 2.

Marmitt et al. and Newton–Raphson M first uses the method proposed by Marmitt et al. [2004] to split the polynomial as shown in Figure 4, and then uses repeated linear interpolation for iteration. NR also uses the polynomial split, but then replaces repeated linear interpolation with Newton–Raphson iteration. Both M and NR have a maximum number of iterations that can be performed and also stops iteration if the difference between two subsequent t-values is less than a chosen epsilon. We empirically picked a maximum number of iterations of 50 and an epsilon value of $\epsilon = 4.0 \cdot 10^{-3}$, where a larger maximum number of iterations or smaller epsilon value never visually improved the images we rendered.

With this, we denote a combination of a traversal algorithm and a voxel intersection test method by the concatenation of the abbreviations, i.e., SBS-NR is sparse brick set with Newton–Raphson. We use classical instancing in the scenes (Heads and Ladies) with replicated objects: e.g., for each instance, we inverse-transform the ray and then traverse a shared object.

4.2. Performance

The main performance results using path tracing are shown in Table 1. It is clear that the fastest traversal method is SVS, combined with either the analytic voxel intersection test (A) or the numerical method by Marmitt et al. (M). Among the rest of the algorithms, it is less clear. For the simpler scenes (Cheese and Goblin), GST is the next fastest, closely followed by SBS and SVO. For the more complex scenes (Heads and Ladies), SBS is generally faster than SVO, and GST is the slowest. This is likely

	Cheese	**Goblin**	**Heads**	**Ladies**
GST	177.3	177.3	177.3	177.3
SVS	209.0	152.2	95.5	30.4
SBS	24.6	18.0	11.7	4.1
SVO	45.0	32.6	20.4	6.5

Table 2. Memory usage for different grid traversal algorithms, numbers are in megabytes. The memory usage includes both internal data buffers that are used to intersect against the SDF surface as well as the acceleration structure size.

due to more indirect rays hitting the SDF surface in complex scenes. SBS and SVO are better at handling this random access as they store data more compactly. Furthermore, SBS creates tighter bounding boxes around the SDF surface, which allows for better utilization of hardware-accelerated ray tracing. However, for simpler scenes where most rays access similar data, GST seems to traverse the grid faster than SBS and SVO.

For voxel intersection testing, the analytic test (A) is the fastest method, but with M having similar numbers for SVS. It was surprising to us that M is often faster than Newton–Raphson (NR), as NR has faster convergence rates. It may be due to NR taking longer to converge in harder cases where the starting point is on the cubic curve and the magnitude of the derivative is relatively small.

We speculated that SBS might be faster than SVS due to better use of locality because a block of $7 \times 7 \times 7$ voxels is read at a time, but we cannot see any such evidence. This may change if the data is compressed with BC4 texture compression. However, since values at the edges are replicated to avoid reading neighboring blocks, the lossy nature of BC4 will cause edge values to be different, which can give rise to cracks in the SDF surface. This may be reasonable when the camera is far away, but less acceptable when the camera is closer. We leave a thorough evaluation of texture compression for future work.

Memory usage for the different grid traversal algorithms can be seen in Table 2. The memory usage of GST is constant as it does not depend on the sparsity of surface voxels. SVS needs to store a large amount of data per voxel, but only stores voxels that intersect the SDF surface. Therefore, SVS uses more memory than GST for the Cheese SDF as it causes more voxels to intersect the surface than the other SDFs do. If we reduced the storage per voxel from $4 \times 4 \times 4$ SDF values to $2 \times 2 \times 2$ SDF values, the numbers for SVS would approximately be reduced by 50%. That would, however, make our new normal interpolation method substantially slower. SBS and SVO use the least amount of memory across all scenes due to both of them being good at exploiting sparsity, while also storing less data per voxel compared to SVS. In a memory-constraint scenario, SBS is likely the winning method because it uses the least memory and is faster than SVO.

		Cheese	**Goblin**	**Heads**	**Ladies**
Falcão normals [2008]	GST-A	+1.1%	+1.6%	+0.7%	+0.3%
Analytic normals (Section 3.1)	GST-A	+0.0%	+0.2%	+0.2%	+0.1%
	SVS-A	–0.1%	+0.3%	–2.1%	–0.1%
	SBS-A	+0.3%	–0.2%	–0.1%	–0.2%
Our method (Section 3.2)	GST-A	+4.2%	+3.7%	+1.9%	+1.1%
	SVS-A	+3.9%	+7.3%	+1.0%	+4.5%
	SBS-A	+4.3%	+5.8%	+3.7%	+3.5%

Table 3. Performance of different normal methods measured as a percent change in the overall frame time, where negative numbers indicate a reduction in total frame time to the baseline, while positive numbers indicate an increase. The baseline is the discontinuous version of the method by Falcão.

4.3. Normals

Our comparison method for the normal computations is a difference-based method that uses four evaluations of the signed distance function [Falcão 2008] and is the least expensive method that we have found. It is possible to create normals that achieve C^0 continuity of the normal field over voxel edges by sampling neighboring voxels with this method combined with GST. We call this method *Falcão normals*. However, using other traversal methods, it is not always possible to sample neighboring voxels as they may not exist. Therefore, a version of the method by Falcão [2008] that is not C^0 across voxel edges is used for those traversal methods. This method is denoted *baseline*. It extends the surface of the current voxel outside its box to allow the signed distance to be sampled for neighbors that might not exist. We compare to analytic normals (Section 3.1), which are also not C^0 across voxels, and to our proposed method (Section 3.2), which delivers continuous normals. For the baseline and Falcão methods, we empirically picked an epsilon of 20% of the voxel size as it resulted in the best visual appearance. The other methods do not require any tweaking of epsilon values. To summarize, the four normal computation methods we evaluate are the following:

1. baseline,
2. Falcão normals [2008],
3. analytic normals (Section 3.1), and
4. our method (Section 3.2).

To reduce the sheer amount of data, we present statistics only for GST-A, SVS-A, and SBS-A using analytic normals and our proposed method. We also present statistics on the performance of Falcão normals using GST-A. The results are presented in Table 3, where the numbers are compared against the baseline method. Falcão

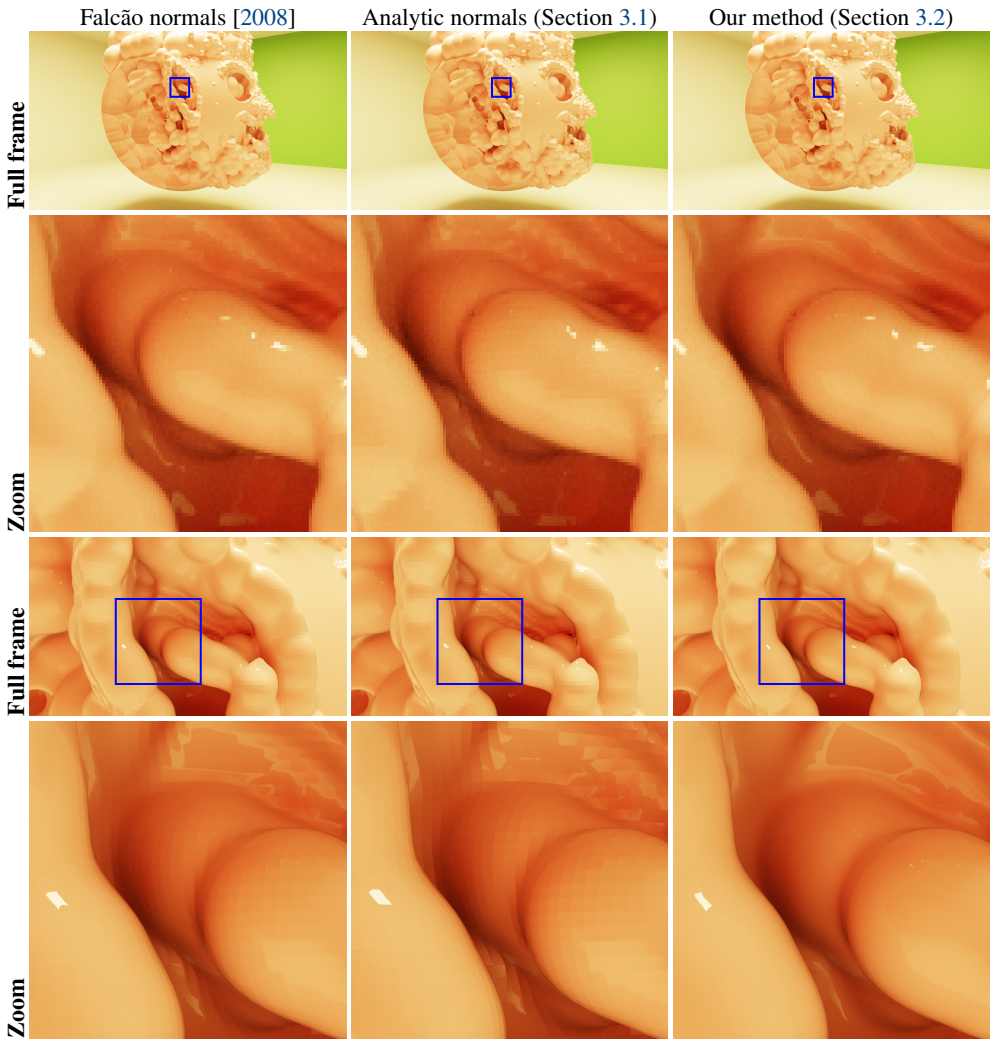

Figure 7. Image quality comparison of the three different normal methods that we evaluate.

normals perform consistently slightly worse than the baseline. This is due to it being more costly to sample additional distance values from neighboring voxels than to use the distance values of only one voxel. Analytic normals (Section 3.1) perform similar to the baseline, which is expected as both only use the distance values of a single voxel. Our method, on the other hand, uses 1.1–4.2% more time per frame on average for GST-A compared to the baseline and 1.0–7.3% more time per frame on average for SVS-A. Without storing the $4 \times 4 \times 4$ SDF values per voxel, the runtime was substantially larger. Finally, our method used 3.5–5.8% more time per frame on average for SBS-A compared to the baseline.

Even though our method used more time to compute the normals, there is also a significant difference in image quality, as seen in Figure 7. From a distance, it is

difficult to see any large differences between the three methods. In closeups, Falcão normals are somewhat smoother than the method of Section 3.1, which is to be expected since Falcão normals sometimes access neighboring voxel data. Our method provides superior results for closeups as can be seen to the lower right. An alternative would be to use the method of Section 3.1 far away from the object and switch to our method for closeups. This could be decided by tracing ray cones [Akenine-Möller et al. 2019] and selecting a method based on the ray cone radius at the hit point. This is also left for future work.

4.4. Shadow Ray Optimization

The shadow ray optimization mentioned in Section 2 is most suitable to enable when a numerical method, which splits the polynomial (Figure 4), is used. We evaluate this optimization only for the Newton–Raphson method, but the results should apply to any method that uses polynomial splitting.

We found that for GST-NR, the shadow ray optimization uses 4–6% less time across our four test scenes on average. The reduction was 6–14% for SVS-NR, 1–2% for SBS-NR, and 0.6–1.3% for SVO-NR. We conclude that it is a worthwhile optimization to do for all methods.

5. Conclusion and Future Work

We have implemented a set of traversal methods and voxel intersection techniques for path tracing on the GPU. This includes our own derivation of how to intersect a ray with the surface defined by trilinearly interpolating SDF values from the $2 \times 2 \times 2$ corners of a voxel. The fastest traversal method (SVS) is the one where a BVH is built around each non-empty voxel, which is then traced by the GPU ray tracing hardware on the RTX 3090. A custom intersection shader then handles the intersection between the surface in a voxel and a ray. In combination with SVS, solving the third-order polynomial analytically or solving it using repeated linear interpolation gave the fastest total performance.

Our new method for interpolating normals uses 1–7% more time compared to the baseline, but provides substantially smoother images when viewing surfaces from a short distance. For future work, it could be worthwhile to let ray cones [Akenine-Möller et al. 2019] control when to switch from a less-expensive, lower-quality method to our method. We recommend that the shadow ray optimization that we have proposed is always used for methods based on polynomial splitting. In the future, it would also be useful to evaluate all methods using level of detail with ray cones, so that traversal would be stopped at a level where the voxel is approximately as large as the cone diameter. DXR does not have support for this at the moment, however.

Acknowledgements

Thanks to Aaron Lefohn for supporting this work, and thanks to Pontus Andersson, Marco Salvi, and Eric Haines for providing feedback. We are grateful to Jonathan Åleskog for modeling the "sad guy" in the teaser image.

References

AALTONEN, S., 2018. GPU-based clay simulation and ray tracing tech in Claybook. Presentation at Game Developers Conference, March 19–23. URL: https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc2018/presentations/Aaltonen_Sebastian_GPU_Based_Clay.pdf. 95, 103

AKENINE-MÖLLER, T., NILSSON, J., ANDERSSON, M., BARRÉ-BRISEBOIS, C., TOTH, R., AND KARRAS, T. 2019. Texture level of detail strategies for real-time ray tracing. In *Ray Tracing Gems*, E. Haines and T. Akenine-Möller, Eds. Apress, ch. 20, 321–345. URL: https://link.springer.com/chapter/10.1007/978-1-4842-4427-2_20. 109

BÁLINT, C., AND VALASEK, G. 2018. Accelerating sphere tracing. In *Eurographics Short Papers*, The Eurographics Association, 29–32. URL: http://dx.doi.org/10.2312/egs.20181037. 95

BLOOMENTHAL, J., BAJAJ, C., BLINN, J., CANI-GASCUEL, M.-P., ROCKWOOD, A., WYVILL, B., AND WYVILL, G. 1997. *Introduction to Implicit Surfaces*. Morgan Kaufmann. URL: https://dl.acm.org/doi/book/10.5555/549676. 94

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings on the 2009 Symposium on Interactive 3D Graphics and Games*, Association for Computing Machinery, 15–22. URL: https://maverick.inria.fr/Publications/2009/CNLE09/CNLE09.pdf. 104

EVANS, A., 2015. Learning from failure: A survey of promising, unconventional and mostly abandoned renderers for Dreams PS4, a geometrically dense, painterly UGC game. Advances in Real-Time Rendering in Games, SIGGRAPH Course. URL: http://advances.realtimerendering.com/s2015/AlexEvans_SIGGRAPH-2015-sml.pdf. 95, 104

FALCÃO, P., 2008. Implicit function to distance function. URL: https://www.pouet.net/topic.php?which=5604&page=3#c233266. 100, 107, 108

FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., 249–254. URL: https://dl.acm.org/doi/10.1145/344779.344899. 99

GALIN, E., GUÉRIN, E., PARIS, A., AND PEYTAVIE, A. 2020. Segment tracing using local Lipschitz bounds. *Computer Graphics Forum 39*, 2, 545–554. URL: https://doi.org/10.1111/cgf.13951. 95

HART, J. C. 1995. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 527–545. URL: https://link.springer.com/article/10.1007/s003710050084. 95, 96

KALLWEIT, S., CLARBERG, P., KOLB, C., YAO, K.-H., FOLEY, T., WU, L., CHEN, L., AKENINE-MÖLLER, T., WYMAN, C., CRASSIN, C., AND BENTY, N., 2021. The Falcor rendering framework, August. URL: https://github.com/NVIDIAGameWorks/Falcor. 102

KEINERT, B., SCHÄFER, H., KORNDÖRFER, J., GANSE, U., AND STAMMINGER, M. 2014. Enhanced sphere tracing. In *Smart Tools and Apps for Graphics—Eurographics Italian Chapter Conference*, The Eurographics Association, 1–8. URL: http://dx.doi.org/10.2312/stag.20141233. 95

LAINE, S., AND KARRAS, T. 2010. Efficient sparse voxel octrees—Analysis, extensions, and implementation. NVIDIA Technical Report NVR-2010-001, NVIDIA. URL: https://research.nvidia.com/publication/2010-02_efficient-sparse-voxel-octrees-analysis-extensions-and-implementation. 104

LOPES, A., AND BRODLIE, K. 2003. Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics 9*, 1, 16–29. URL: https://ieeexplore.ieee.org/document/1175094. 95

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (SIGGRAPH) 21*, 4, 163–169. URL: https://doi.org/10.1145/37402.37422. 95

MARMITT, G., KLEER, A., WALD, I., AND FRIEDRICH, H. 2004. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *Vision, Modeling, and Visualization*, AKA, vol. 4, 429–435. 98, 105

MUSETH, K. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics 32*, 3, 27:1–27:22. URL: https://doi.org/10.1145/2487228.2487235. 104

MUSETH, K. 2021. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*, Association for Computing Machinery, 1:1–1:2. URL: https://doi.org/10.1145/3450623.3464653. 104

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. 1998. Interactive ray tracing for isosurface rendering. In *Proceedings Visualization '98*, IEEE, 233–238. URL: https://ieeexplore.ieee.org/document/745713. 95, 97

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 2007. *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press. URL: http://numerical.recipes/. 98

VLACHOS, A., PETERS, J., BOYD, C., AND MITCHELL, J. L. 2001. Curved PN triangles. In *Proceedings of the Symposium on Interactive 3D Graphics*, Association for Computing Machinery, 159–166. URL: https://doi.org/10.1145/364338.364387. 100

Author Contact Information

Herman Hansson Söderlund
NVIDIA
Ideon Science Park
Scheelevägen 28
223 70 Lund
Sweden
hermanh@nvidia.com

Alex Evans
NVIDIA Ltd.
100 Brook Drive
READING
RG2 6UJ
United Kingdom
alexe@nvidia.com

Tomas Akenine-Möller
NVIDIA
Ideon Science Park
Scheelevägen 28
223 70 Lund
Sweden
takenine@nvidia.com

H. Hansson Söderlund, A. Evans, and T. Akenine-Möller, Path Tracing of Signed Distance Function Grids, *Journal of Computer Graphics Techniques (JCGT)*, vol. 11, no. 3, 94–113, 2022
http://jcgt.org/published/0011/03/06/

Received:	2021-09-08		
Recommended:	2022-02-17	Corresponding Editor:	Natalya Tatarchuk
Published:	2022-09-21	Editor-in-Chief:	Marc Olano