Incremental Ray Traversal through OpenVDB Frustum Grids

Manuel N. Gamito
Framestore

Figure 1. One frame from the film ”The Midnight Sun” (Copyright Netflix 2020), showing several volumes stored as OpenVDB frustum grids.

Abstract

The rendering of volumes with modern path-tracing techniques requires the sampling of collision distances within the media to determine where scattering events occur. For collision sampling, delta-tracking algorithms traverse a ray through clusters of voxels, while querying the minimum and maximum extinction values within each cluster. Such ray traversals can be performed easily on uniform grids but face difficulties due to the non-linearity of frustum grids. Frustum grids, however, can be very useful to distribute voxels efficiently, relative to a perspective camera, while keeping memory requirements low. We present an incremental technique for traversing frustum grids in OpenVDB—a widely adopted format for volume storage in production path tracing. Our technique is a generalization of the digital differential analyser algorithm that is the standard for ray traversal in uniform grids.

1. Introduction

Path tracing in the presence of scattering media employs a technique, called *delta tracking*, that samples the free path distance in a medium before a scattering or absorption collision occurs. A companion tracking technique, called *ratio tracking*, is used to estimate transmittance within the medium, for next event estimation from light sources. Although the delta-tracking algorithm and its variants are beyond the scope of this paper, we address here the issue of efficiently traversing voxel grids during tracking. For more details on the tracking algorithms themselves, we refer to the survey by Novák et al. [2018].

A required feature for ray tracking is that voxels must be grouped into clusters, and the maximum extinction coefficient within each cluster must be recorded. Some versions of the tracking algorithm also employ the minimum extinction within a cluster and so it is useful to store these extinction extrema as part of the metadata of a voxel grid. For OpenVDB, this partitioning of voxels corresponds naturally to the leaf nodes of a grid hierarchy [Museth 2013]. An OpenVDB grid is a lightweight hierarchy of nodes, commonly four levels deep, and the leaf nodes hold the actual voxel data that is loaded into memory on request. A leaf node stores an array of $8 \times 8 \times 8$ voxels, although leaf arrays with other powers of two in size can also be configured.

The ray traversal through the leaf nodes of a uniform OpenVDB grid is done with a 3D digital differential analyzer [Amanatides and Woo 1987], which efficiently finds all successive ray intersections with the leaf boundaries. We call this procedure *ray traversal* with respect to a grid to differentiate it from *ray marching*, which is the increment of the ray distance by constant steps, irrespective of any grid boundaries. The 3D DDA is an extension of the old 2D line drawing DDA algorithm of Bresenham [1965]. OpenVDB offers a DDA implementation, and we routinely employ it for ray tracking in our uniform grids.

Frustum grids offer significant advantages, compared to uniform grids, in placing voxels where they matter most relative to the camera [Wrenninge 2012]. They have been used successfully to render cloudscapes with OpenVDB [Miller et al. 2012]. We present a generalization of the DDA ray-traversal algorithm that accounts for the non-linearity of the perspective transform in OpenVDB frustum grids.

2. Previous Work

Tracing camera rays through camera-aligned frustum grids can be done with fixed distance increments—a technique that amounts to a one-dimensional DDA over the depth axis [Wrenninge 2012]. For shadow rays or indirect rays, this is no longer possible and different approaches are needed.

Wrenninge et al [2013] store the volume data in a sparse kd-tree format in frustum space, and the ray is traversed through this kd-tree. During ray traversal, each successive kd-tree separating plane is mapped from frustum space to world space, and an explicit ray-plane intersection is performed to yield the correct world-space distance.

Museth [2014] proposes using an auxiliary uniform grid, superimposed over an OpenVDB frustum grid. This auxiliary grid stores binary information that acts as a mask for the presence of the frustum. The DDA is performed over the uniform grid, and the frustum grid is sampled for points where the mask indicates that it is present.

As we will show, our technique requires neither a dedicated volume format nor an auxiliary grid and can perform a fully incremental 3D DDA traversal over an OpenVDB frustum grid for any ray.

3. OpenVDB Frustum Grids

A frustum grid, in OpenVDB, is a voxel grid that is mapped to 3D world space through a transform that creates a rectangular frustum. This frustum is intended to be aligned exactly with the view frustum of a perspective camera. The transform proceeds in two steps. First, a non-linear mapping $f : \mathbb{R}^3 \mapsto \mathbb{R}^3$ takes the index space of the voxel grid into a canonical frustum space, shown in Figure 2. This step is then followed by an affine transform, represented by a 4×4 homogeneous matrix $\mathbf{M}$, which adds the final mapping into world space.

The grid is initially defined over a region $[I_0, I_1] \times [J_0, J_1] \times [K_0, K_1]$ of the $\{(i, j, k) : \mathbb{R}^3\}$ index space. The grid can be indexed at any continuous location within this region by interpolation from the nearest integer grid points. The grid is then mapped to the rectangle $[-1/2, +1/2] \times [-a/2, +a/2]$ on the near plane at coordinate $z = 0$ in frustum space, with the aspect ratio $a = (J_1 - J_0)/(I_1 - I_0)$. The i- and j-grid indices map to the x- and y-frustum axes, respectively, while the k-index is mapped along the z-depth axis. The frustum transform f is parameterized by a taper τ and a depth d. The far plane is set at $z = d$ and contains the rectangle

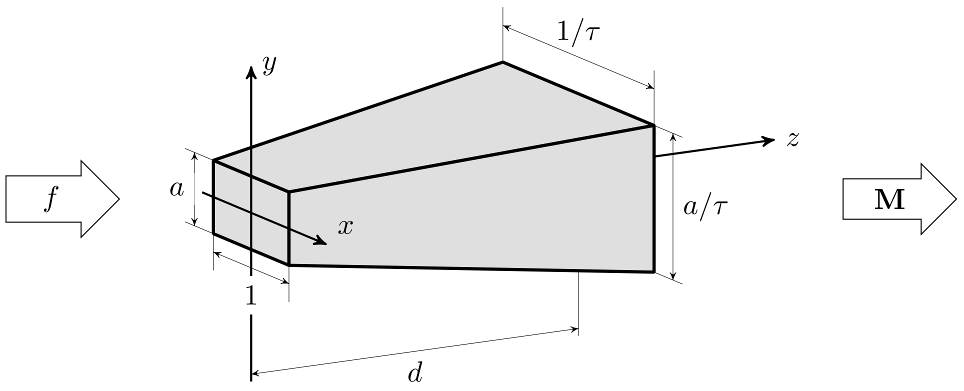

Figure 2. The OpenVDB frustum space is the middle stage of the index space to world space transform. The non-linear map f and the matrix $\mathbf{M}$ move points into and out of this space.

$[-1/(2\tau), +1/(2\tau)] \times [-a/(2\tau), +a/(2\tau)]$. The taper parameter τ is therefore the ratio between the sizes of the near rectangle and the far rectangle. The matrix $\mathbf{M}$ can be any general affine transform. Usually, it will contain scaling components that change the size of the near rectangle to match the image-sensor size of the camera. A translational component can also move the near plane $z = 0$ toward the camera's near clipping-plane distance.

4. Ray Traversal in Index Space

For ray traversal of the grid, we seek to go in the opposite direction of the transform in Figure 2. Given a ray $\vec{r}(t) = \vec{r}_0 + \hat{\omega}t$ in world space, parameterized by distance $t > 0$ and with unit direction vector $\hat{\omega}$, we first transform it linearly into frustum space and obtain an equivalent frustum ray $\vec{s}(t) = \vec{s}_0 + \vec{\delta}t$ such that

$$\begin{aligned} [\vec{s}_0\ 1]^\top &= \mathbf{M}^{-1}[\vec{r}_0\ 1]^\top, \\ [\vec{\delta}\ 0]^\top &= \mathbf{M}^{-1}[\hat{\omega}\ 0]^\top. \end{aligned} \tag{1}$$

The direction vector $\vec{\delta}$ is not renormalized after the inverse matrix multiplication. In this way, values of t found during traversal can be directly applied in world space.

Expressing the frustum ray in its three coordinates $\vec{s}(t) := (s_x(t), s_y(t), s_z(t))$, we can now invert the frustum map f and obtain parametric expressions for the coordinates of the ray in index space. Based on the definition of f^{-1} from OpenVDB, the ray indices are given in Equations (2):

$$\begin{aligned} i(t) &= \frac{I_1 - I_0}{\gamma s_z(t) + 1} s_x(t) \;+\; \frac{I_1 + I_0}{2}, \\ j(t) &= \frac{I_1 - I_0}{\gamma s_z(t) + 1} s_y(t) \;+\; \frac{J_1 + J_0}{2}, \\ k(t) &= \frac{K_1 - K_0}{d} s_z(t) \;+\; K_0, \\ \gamma &= (1/\tau - 1)/d. \end{aligned} \tag{2}$$

The dependence of $j(t)$ on $I_1 - I_0$ is not an error. It accounts for the aspect ratio of the y-axis, relative to the x-axis. For illustration, when $s_z(t) = 0$ and with $s_y(t)$ in the range $[-a/2, +a/2]$, we get the correct index range $j(t) \in [J_0, J_1]$. Given that all three components of $\vec{s}(t)$ are binomials, the $i(t)$ and $j(t)$ indices are rational functions, while $k(t)$ is a linear remapping: $[0, d] \mapsto [K_0, K_1]$. A ray in frustum space, therefore, becomes a rational curve in index space, after the inverse map f^{-1} has been applied. It is this curve that will be tested against all intersections with the internal voxel boundaries of the grid, for increasing t.

4.1. Ray-Frustum Clipping

The ray must be clipped against the frustum to determine the range of distances $[t_0, t_1]$ over which the traversal will proceed. For that purpose, we invert the index functions of Equations (2) relative to t, writing out the full expressions for the ray components and obtain three possible distance functions, depending on which index is used:

$$\begin{aligned} t_\mathrm{I}(i) &= \frac{x'(\gamma s_{0z}+1) - s_{0x}}{\delta_x - x'\gamma\delta_z}, & x' &= \frac{i-(I_1+I_0)/2}{I_1-I_0}, \\ t_\mathrm{J}(j) &= \frac{y'(\gamma s_{0z}+1) - s_{0y}}{\delta_y - y'\gamma\delta_z}, & y' &= \frac{j-(J_1+J_0)/2}{I_1-I_0}, \\ t_\mathrm{K}(k) &= (z - s_{0z})/\delta_z, & z &= d(k-K_0)/(K_1-K_0). \end{aligned} \tag{3}$$

The coordinates (x', y', z) result from the partial projection of $\vec{s}(t)$ in frustum space onto the $z = 0$ plane, leaving the depth $z = s_z(t)$ unchanged.

Clipping is done by testing all distance functions against the boundaries of the grid. We first intersect with the slab of vertical planes $k = K_0$ to $k = K_1$ (correspondingly, $z = 0$ to $z = d$) to produce an initial range $[t_0, t_1]$, obtained with

$$\begin{aligned} t_0 &= t_\mathrm{K}(0), \\ t_1 &= t_\mathrm{K}(d). \end{aligned}$$

The other four separating planes are then tested in turn. Considering the right frustum plane first, for which $i = I_1$ and $x' = 1/2$, the outward orthogonal vector to the plane is $\vec{n} = (1, 0, -\gamma/2)$ and one of the points on the plane is $\vec{p}_0 = (1/2, 0, 0)$. The distance to the plane is

$$t_\mathrm{R} = \frac{(\vec{p}_0 - \vec{s}_0)\cdot\vec{n}}{\vec{\delta}\cdot\vec{n}} = \frac{(\gamma s_{0z}+1)/2 - s_{0x}}{\delta_x - \gamma\delta_z/2} = t_\mathrm{I}(1/2).$$

The dot product $\vec{\delta}\cdot\vec{n}$ tells us if the ray is entering or leaving the frustum through this plane. If $\delta_x < \gamma\delta_z/2$, the dot product is negative, the ray is entering, and we update $t_0 = \max\{t_0, t_\mathrm{R}\}$; otherwise the ray is leaving, and we update $t_1 = \min\{t_1, t_\mathrm{R}\}$.

The left frustum plane has $\vec{n} = (-1, 0, -\gamma/2)$ and $\vec{p}_0 = (-1/2, 0, 0)$, leading to

$$t_\mathrm{L} = \frac{(\gamma s_{0z}+1)/2 + s_{0x}}{-(\delta_x + \gamma\delta_z/2)} = \frac{-(\gamma s_{0z}+1)/2 - s_{0x}}{\delta_x + \gamma\delta_z/2} = t_\mathrm{I}(-1/2).$$

The ray enters the left plane when $-(\delta_x + \gamma\delta_z/2) < 0$, or $\delta_x > -\gamma\delta_z/2$, and we update t_0 with t_L if that condition is true; otherwise we update t_1. The top and bottom frustum planes are clipped similarly but using the factor $a/2$ instead of $1/2$.

By checking if $t_1 > t_0$, after all six planes have been tested, we know when a valid distance range for traversal was found. Otherwise the ray does not enter the frustum, and the ray traversal routine returns immediately.

4.2. An Iterative Formulation for Ray Traversal

For ray traversal of a frustum grid, we need to find the distances corresponding to the intersections with the internal boundaries of the OpenVDB leaf nodes. Assuming an infinite grid, the leaf boundaries are defined by the index coordinates $\{(iN, jN, kN) : i, j, k \in \mathbb{Z}\}$, where N is the number of voxels across each dimension of a leaf. In practice, this infinite space is restricted to the grid-boundary coordinates, which will also be multiples of N. We insert these coordinates in Equations (3) and introduce four auxiliary binomials that define the two rational functions for i, and j:

$$\begin{aligned}
t_\mathrm{I}(iN) &= \frac{P_\mathrm{I}(iN)}{Q_\mathrm{I}(iN)}, \quad \begin{cases} P_\mathrm{I}(i) &= x'(\gamma s_{0z}+1) - s_{0x} \\ Q_\mathrm{I}(i) &= \delta_x - x'\gamma\delta_z \end{cases}, \quad x' \text{ as in (3)} \\
t_\mathrm{J}(jN) &= \frac{P_\mathrm{J}(jN)}{Q_\mathrm{J}(jN)}, \quad \begin{cases} P_\mathrm{J}(j) &= y'(\gamma s_{0z}+1) - s_{0y} \\ Q_\mathrm{J}(j) &= \delta_y - y'\gamma\delta_z \end{cases}, \quad y' \text{ as in (3)} \\
t_\mathrm{K}(kN) &= (z - s_{0z})/\delta_z, \quad z = d(kN - K_0)/(K_1 - K_0).
\end{aligned} \tag{4}$$

It is now possible to establish iterative rules to update all three distances during traversal. The index coordinates of the leaf being traversed are updated with fixed integer deltas $\{\Delta_\mathrm{A} = \pm N : \mathrm{A} = \mathrm{I}, \mathrm{J}, \mathrm{K}\}$. The signs of the deltas control if coordinates are incremented or decremented. We defer the determination of these signs to the following section. Given the three distances at iteration n, each can be updated in the next iteration according to a subset of the following rules:

$$\begin{aligned}
&\mathrm{I}: \begin{cases} (P_\mathrm{I})_{n+1} &= (P_\mathrm{I})_n + \Delta P_\mathrm{I},\ \Delta P_\mathrm{I} = (\gamma s_{0z}+1)\Delta'_\mathrm{I} \\ (Q_\mathrm{I})_{n+1} &= (Q_\mathrm{I})_n - \Delta Q_\mathrm{I},\ \Delta Q_\mathrm{I} = \gamma\delta_z\Delta'_\mathrm{I} \\ (t_\mathrm{I})_{n+1} &= (P_\mathrm{I}/Q_\mathrm{I})_{n+1} \end{cases} \quad \Delta'_\mathrm{I} = \frac{\Delta_\mathrm{I}}{I_1 - I_0} \\
&\mathrm{J}: \begin{cases} (P_\mathrm{J})_{n+1} &= (P_\mathrm{J})_n + \Delta P_\mathrm{J},\ \Delta P_\mathrm{J} = (\gamma s_{0z}+1)\Delta'_\mathrm{J} \\ (Q_\mathrm{J})_{n+1} &= (Q_\mathrm{J})_n - \Delta Q_\mathrm{J},\ \Delta Q_\mathrm{J} = \gamma\delta_z\Delta'_\mathrm{J} \\ (t_\mathrm{J})_{n+1} &= (P_\mathrm{J}/Q_\mathrm{J})_{n+1} \end{cases} \quad \Delta'_\mathrm{J} = \frac{\Delta_\mathrm{J}}{I_1 - I_0} \\
&\mathrm{K}: \begin{cases} (t_\mathrm{K})_{n+1} = (t_\mathrm{K})_n + \Delta'_\mathrm{K}, \quad \Delta'_\mathrm{K} = \dfrac{d}{\delta_z}\dfrac{\Delta_\mathrm{K}}{K_1 - K_0} \end{cases}
\end{aligned} \tag{5}$$

For each iteration, the ray distance to the next leaf-crossing is the nearest of the three distances: $(t)_n = \min\{(t_\mathrm{I})_n, (t_\mathrm{J})_n, (t_\mathrm{K})_n\}$. This distance is then iterated according to its own set of rules, as stated in Equations (5), while the other two distances stay unchanged. Note that the iteration for t_K follows the same rule that is normally employed in the 3D DDA algorithm of Amanatides and Woo [1987]. This is a natural consequence of the frustum map preserving linearity along the depth. If the frustum map has no tapering effect, so that $\tau = 1$ and $\gamma = 0$, the entire set of Equations (5) reverts to a 3D DDA.

4.3. Asymptotic Behavior of Rays in Index Space

To determine the direction of travel of the curved ray in the grid index space, we analyze the derivatives of $\vec{s}(t)$, with respect to the parameter t. From Equations 2, and using simple calculus rules, we obtain the following set of ray derivatives for the indices (i, j, k) in the grid as a function of the ray distance:

$$\begin{aligned}
\frac{di}{dt}(t) &= \frac{I_1 - I_0}{(\gamma s_z(t)+1)^2}\big(\delta_x + \gamma(\delta_x s_{0z} - \delta_z s_{0x})\big), \\
\frac{dj}{dt}(t) &= \frac{I_1 - I_0}{(\gamma s_z(t)+1)^2}\big(\delta_y + \gamma(\delta_y s_{0z} - \delta_z s_{0y})\big), \\
\frac{dk}{dt}(t) &= \frac{K_1 - K_0}{d}\delta_z.
\end{aligned} \tag{6}$$

The sign of these derivatives then indicates when indices should be incremented or decremented during traversal. Since the squared denominators are always positive, the signs never change along the trajectory, and the ray has fixed directions of travel, with no inflection points along the way. The grid increments are given by

$$\begin{aligned}
\Delta_{\mathrm{I}} &= \operatorname{sgn}\big\{\delta_x + \gamma(\delta_x s_{0z} - \delta_z s_{0x})\big\}N, \\
\Delta_{\mathrm{J}} &= \operatorname{sgn}\big\{\delta_y + \gamma(\delta_y s_{0z} - \delta_z s_{0y})\big\}N, \\
\Delta_{\mathrm{K}} &= \operatorname{sgn}\big\{\delta_z\big\}N,
\end{aligned} \tag{7}$$

where the sign function is understood to return 0 for zero arguments.

Asymptotically, the derivatives converge to a value proportional to $(0, 0, \delta_z)$ as t increases. This means that, provided a ray travels for a sufficiently long distance inside the frustum, it will progressively align itself with two orthogonal planes of constant i and j, respectively. This then creates a type of situation that is illustrated in Figure 3. There, a ray has crossed a voxel boundary with coordinate jN, at a distance $t_{\mathrm{J}}(jN)$, but it will never cross the next boundary with coordinate $(j+1)N$ because its vertical rate of increase dj/dt has become too small from that point onward. If one were to solve for the intersection with the next boundary, the distance would be $t_{\mathrm{J}}((j+1)N) < t_{\mathrm{J}}(jN)$ and it would actually be negative, corresponding to an

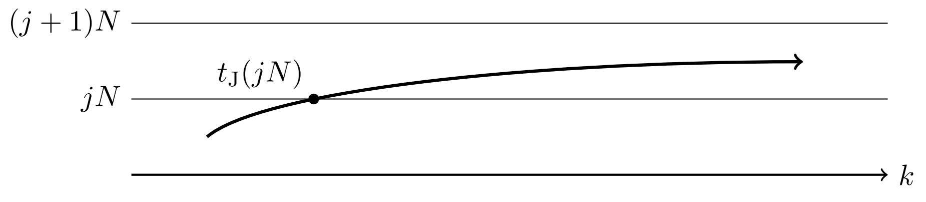

Figure 3. A ray that crosses an internal voxel boundary but does not cross the next one.

intersection point behind the ray. This is not a valid distance so we set $(t_{\mathrm{J}})_n = +\infty$, for the nth iteration where it happens, to signal to the algorithm that no further intersections along the J-axis will occur in the following iterations. Unlike the other two axes, we always have $(t_{\mathrm{K}})_{n+1} > (t_{\mathrm{K}})_n$, for all n, because

$$(t_{\mathrm{K}})_{n+1} - (t_{\mathrm{K}})_n \;=\; \frac{d}{\delta_z}\frac{\Delta_{\mathrm{K}}}{K_1 - K_0} \;=\; \frac{d}{\delta_z}\frac{\operatorname{sgn}\{\delta_z\}N}{K_1 - K_0} \;=\; \frac{d}{|\delta_z|}\frac{N}{K_1 - K_0} \;>\; 0$$

and $(t_{\mathrm{K}})_0 > 0$ by construction. In the event of both I and J becoming invalid for traversal, the depth axis K will be valid and the algorithm will always progress through the frustum.

Camera rays have the characteristic that they all start from $\vec{s}_0 = (0, 0, -1/\gamma)$ in frustum space. If this origin is inserted in the derivatives of Equations (6), we obtain $di/dt = dj/dt = 0$. The consequence is that camera rays only progress along the K-axis in index space, while keeping their i and j grid coordinates constant.

4.4. Initialization of the Stepping Variables

The initial states $(P_{\mathrm{I}})_0$, $(Q_{\mathrm{I}})_0$, $(P_{\mathrm{J}})_0$ and $(Q_{\mathrm{J}})_0$ are obtained from Equations (4), using the entry grid coordinates:

$$i_0 \;=\; \begin{cases} (\lfloor i(t_0)/N \rfloor + 1)N & \text{if } \Delta_{\mathrm{I}} \geqslant 0, \\ \lfloor i(t_0)/N \rfloor N & \text{otherwise.} \end{cases}$$

$$j_0 \;=\; \begin{cases} (\lfloor j(t_0)/N \rfloor + 1)N & \text{if } \Delta_{\mathrm{J}} \geqslant 0, \\ \lfloor j(t_0)/N \rfloor N & \text{otherwise.} \end{cases}$$

with t_0 being the ray entry distance into the frustum that feeds the coordinate functions in Equations (2). The increments Δ_{I} and Δ_{J} come from Equations (7). The corresponding increments ΔP_{I}, ΔQ_{I}, ΔP_{J} and ΔQ_{J} are obtained from Equations (5) and (7).

If we find that the initial per-axis intersection distances are less than the entry distance and, specifically for the I axis, if $(t_{\mathrm{I}})_0 = (P_{\mathrm{I}}/Q_{\mathrm{I}})_0 < t_0$ then $(t_{\mathrm{I}})_0$ must be set to $+\infty$. This corresponds to a situation where the very first iteration will already fail to reach the next leaf boundary crossing, due to the non-linearity of the frustum mapping. A similar procedure is used for $(t_{\mathrm{J}})_0$. The previous considerations also apply to a camera ray because in that case

$$(P_{\mathrm{I}})_0 \;=\; x'(\gamma s_{0z} + 1) - s_{0x} \;=\; x'(-\gamma/\gamma + 1) - 0 \;=\; 0. \tag{8}$$

This then implies $(t_{\mathrm{I}})_0 = 0$, which is certainly less than the positive distance t_0 the camera ray must travel from the apex of the frustum to the $z = 0$ plane. The J-axis also obeys its own version of Equation (8), and we start with $(t_{\mathrm{I}})_0 = (t_{\mathrm{J}})_0 = +\infty$, which effectively means that the traversal algorithm will only ever take steps along the depth axis.

For axis K, we would like to recast its uniform DDA steps into an iterative procedure that is consistent with the other two axes. This can be achieved simply by introducing $(Q_\mathrm{K})_0 = 1$ and $\Delta Q_\mathrm{K} = 0$. The ray trajectory along the depth axis is, therefore, a degenerate case of a rational curve with a binomial denominator that is always unity. We can then define $(P_\mathrm{K})_0 = (d(k_0 - K_0)/(K_1 - K_0) - s_{0z})/\delta_z$, using the entry coordinate

$$k_0 = \begin{cases} (\lfloor k(t_0)/N \rfloor + 1)N & \text{if } \delta_z \geqslant 0, \\ \lfloor k(t_0)/N \rfloor N, & \text{otherwise.} \end{cases}$$

Finally, ΔP_K is the Δ'_K term from Equations (5). It is always true that $(t_\mathrm{K})_0 = (P_\mathrm{K}/Q_\mathrm{K})_0 = (P_\mathrm{K})_0 \geqslant t_0$, unlike the I and J case where the inequality may not always hold. This is a property inherited from the original DDA algorithm, where the distance toward the first intersection after t_0, for any axis, is never negative.

```
class FrustumDDA
{
public:

  // Constructor
  FrustumDDA(const openvdb::math::NonlinearFrustumMap& map)
    : m_map(map) {}

  // Initialise traversal
  bool init(openvdb::math::Ray& ray, double tmax);

  // Take step
  bool step(double& t, openvdb::Coord& ijk);

private:

  double m_tn; // Distance traversed so far
  double m_tM; // Maximum traversal distance

  openvdb::Coord m_ijk; // Current leaf coordinates
  openvdb::Vec3i m_dijk; // Coordinate updates per step
  openvdb::Vec3d m_t; // Per-axis intersection distances
  openvdb::Vec3d m_p, m_q, m_dp, m_dq; // Stepping variables

  const openvdb::math::NonlinearFrustumMap& m_map; // The frustum

};
```

Listing 1. The definition of our FrustumDDA class.

```
bool FrustumDDA::step(double& t, openvdb::Coord& ijk)
{

  // Check end of traversal

  if (m_tn < m_tM)
  {

    // Find axis with nearest intersection

    size_t a = openvdb::math::MinIndex(m_t);

    // Pass current traversal state to the outside,
    // also updating the distance travelled so far

    t = m_tn = std::min(m_t[a], m_tM);
    ijk = m_ijk;

    // Take step along axis 'a'
    // To avoid branching in the K axis case,
    // init() sets m_q[2] = 1.0 and m_dq[2] = 0.0

    m_p[a] += m_dp[a];
    m_q[a] -= m_dq[a];

    double ta = m_p[a]/m_q[a];

    // An invalid increment, to occur,
    // will only be on the I or J axes

    assert(ta > 0.0 || a < 2);

    m_t[a] = (ta > 0.0) ?
      ta : std::numeric_limits<double>::infinity();

    // Update leaf coordinates with increments from eqs. 7

    m_ijk[a] += m_dijk[a];

    return true;

  }

  return false;

}
```

Listing 2. The implementation of the stepping function.

5. Source Code

Listing 1 shows the C++ definition of the `FrustumDDA` class we implemented. It is lightweight, which allows each render thread to maintain a separate object of the class at no significant cost. This also removes any concerns about thread-safety of the code. An `init()` function initializes the traversal of a new ray. A `step()` function provides the stepping functionality during the traversal.

The listing for the `init()` function is not given. This function first transforms the ray from world space to frustum space, according to Equation (1). It then clips the ray against the frustum, as explained in Section 4.1 and returns `false` if no intersection exists. Otherwise, the clipping range is stored in the ray object with `ray.setTimes(t0,t1)`. This allows calling code to query the limits of the traversal range by invoking `ray.t0()` and `ray.t1()` after return from `init()`.

The function then initializes the necessary stepping variables of Section 4.4, together with the initial distances $(t_{\mathrm{I}})_0$, $(t_{\mathrm{J}})_0$ and $(t_{\mathrm{K}})_0$. Traversal is done up to a user-specified maximum distance `tmax`, which could be the distance to the nearest surface intersection, if it exists. Internally, this distance is clipped against the frustum exit distance so that `m_tM = std::min(tmax, t1)`. The `init()` function finally returns `true` to signal a successful initialization.

The stepping function is shown in Listing 2. It returns `true` while the traversal is ongoing and `false` when there are no more steps. This allows the function to be used as the control logic for a `while` loop. The distance to the next grid leaf is returned by argument. The integer ijk-coordinates of the OpenVDB leaf currently being traversed are also returned in the same way. These are, in fact, the coordinates of the lower-left back voxel of the leaf, which uniquely identify it, and will always be divisible by N.

The implementation of `step()` contains the main results of this paper. The simplicity of `step()` is noteworthy. It is scarcely more complex than the original 3D DDA algorithm, the only difference being the update rule for the per-axis ray distances. All the remaining bookkeeping code is the same. Once all internal member variables are properly set in `init()`, the `step()` function has very little work to do.

6. Results and Discussion

Figure 4 shows two renders of one of several volume assets that were used in the scene shown in Figure 1. All volumes were composited together in Nuke to produce the final shot. The top render shows the volume as seen from its intended view point. The bottom render shows clearly the geometry of the frustum grid that was used. This render demonstrates the ability of our frustum-traversal technique to handle rays coming from any direction, using solely the information that is stored in the OpenVDB grid. The frustum grid in Figure 4 has 31 million active voxels. By comparison, a uniform grid that covers the same spatial extent requires 110 million voxels. The uniform grid

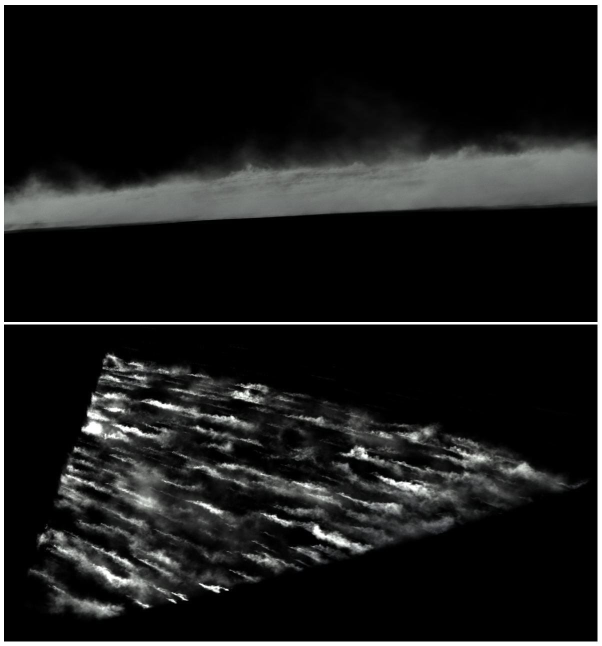

Figure 4. An OpenVDB volume asset, used in the rendering of the scene shown in Figure 1. The volume is stored in a frustum grid and seen from the point of view of the render camera (top) and a camera outside the intended view frustum (bottom).

uses a voxel size that matches the voxel size at the front of the frustum grid so that no visible sampling degradation occurs. This illustrates the memory advantages of frustum grids over uniform grids, when working with volumes that span large distances relative to the camera.

Figure 5 shows a close up of the Disney cloud data set, resampled onto a frustum grid. This render shows how light leaks can occur along the image boundaries when the frustum grid matches exactly the view frustum of the camera [Wrenninge 2012]. The frustum grid clips out-of-view data that would have otherwise contributed to internal scattering and shadowing from the light sources. The effect of the missing volume data is to create an artificial brightening around the image edges due to insufficient shadowing. This effect is not restricted to the image edges, as Figure 5 shows, but can also affect internal shadow terminators. Boundary artefacts can be minimized by padding the grid with a voxel layer of sufficient thickness to allow lighting contributions originating outside the view frustum to be propagated inside. If the frustum grid is sufficiently wide to encompass the entire extent of volume then no artefacts will result. Figure 6 shows the same padded volume of Figure 5, visualized from a view point outside the intended view frustum. The red lines indicate the frustum that was previously used. The padding of the volume relative to this frustum is evident.

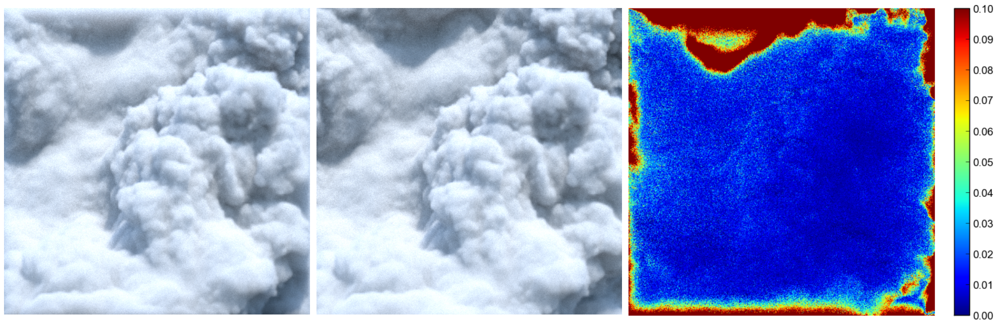

Figure 5. A frustum grid that exactly matches the view frustum (left), showing light leaks around the edges. The same grid with a 16-voxel-wide padding (middle) removes this artefact. The difference image (right) also highlights changes in internal shadowing.

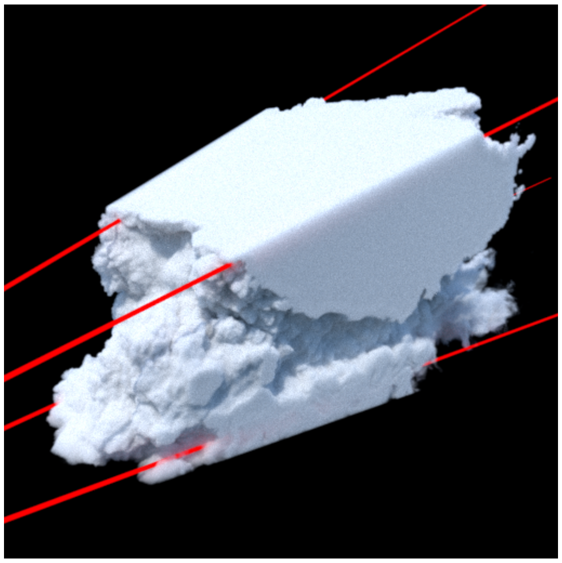

Figure 6. The frustum grid of Figure 5, visualized with a camera that is not aligned with the grid. The red lines indicate the frustum of the camera that was originally used.

7. Future Work

The OpenVDB frustum transform features a linear mapping along the depth axis. By contrast, the OpenGL perspective transform adds a non-linear mapping along this dimension as well, with the intention of improving the numerical precision of depth values [Kessenich et al. 2016]. In OpenGL, geometric elements are compressed towards the near clipping plane, after a perspective transform, and, conversely, they are placed more sparsely towards the far clipping plane. It could be useful to have a traversal algorithm that works with OpenGL perspective matrices. This is outside the scope of our path-tracing work but, nevertheless, we consider it to be an interesting extension of the current algorithm.

Another possible development could lead to a hierarchical frustum-traversal algorithm that exploits the full configuration of an OpenVDB grid hierarchy, analogous to the hierarchical DDA of Museth [2014] for uniform grids. Such an algorithm would not be restricted to traversing leaf nodes only and could take advantage of grid sparsity to quickly step through regions that are either empty or hold a constant value. This could be achieved by managing several traversal states simultaneously, one for each level of the grid hierarchy. The algorithm would then transition between traversal states in the same way that the ray would move upwards or downwards through the hierarchy during traversal.

Acknowledgements

The author would like to thank Daniel Jenkins, Nicolas Sauval and Ruben Villoria Serrano for preparing the cloud datasets, and Laurianne Proud'hon for helping with the complexities of Nuke.

References

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics 87*. Eurographics, Air-la-Ville, Switzerland, 3–10. 50, 54

BRESENHAM, J. E. 1965. Algorithm for computer control of a digital plotter. *IBM Syst. J. 4*, 1 (Mar.), 2530. URL: https://doi.org/10.1147/sj.41.0025. 50

KESSENICH, J., SELLERS, G., AND SHREINER, D. 2016. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*, 9 ed. Addison-Wesley Professional, Hoboken, NJ. 61

MILLER, B., MUSETH, K., PENNEY, D., AND ZAFAR, N. B., 2012. Cloud modelling and rendering for "Puss In Boots". ACM SIGGRAPH Talk. URL: http://www.museth.org/Ken/Publications_files/Miller-etal_SIG12.pdf. 50

MUSETH, K. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph. 32*, 3 (July). URL: https://doi.org/10.1145/2487228.2487235. 50

MUSETH, K. 2014. Hierarchical digital differential analyzer for efficient ray-marching in OpenVDB. In *ACM SIGGRAPH 2014 Talks*. ACM, New York, NY. URL: https://doi.org/10.1145/2614106.2614136. 51, 62

NOVÁK, J., GEORGIEV, I., HANIKA, J., AND JAROSZ, W. 2018. Monte Carlo methods for volumetric light transport simulation. *Computer Graphics Forum (Proceedings of Eurographics - State of the Art Reports) 37*, 2 (May), 551–576. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13383. 50

WRENNINGE, M., KULLA, C. D., AND LUNDQVIST, V. 2013. Oz: the great and volumetric. In *ACM SIGGRAPH 2013 Talks*. ACM, New York, NY. URL: https://doi.org/10.1145/2504459.2504518. 50

WRENNINGE, M. 2012. *Production volume rendering: Design and implementation*. A K Peters/CRC Press, Natick, MA. 50, 60

Author Contact Information

Manuel N. Gamito
Framestore
28 Chancery Lane
London WC2A 1LB
Manuel.Gamito@framestore.com

Manuel N. Gamito, Incremental Ray Traversal through OpenVDB Frustum Grids, *Journal of Computer Graphics Techniques (JCGT)*, vol. 10, no. 1, 49–63, 2021
http://jcgt.org/published/0010/01/03/

Received: 2020-03-05
Recommended: 2020-10-30 Corresponding Editor: Christophe Hery
Published: 2021-03-05 Editor-in-Chief: Marc Olano