# Interpreting OpenEXR Deep Pixels

Florian Kainz, Industrial Light & Magic

Updated August 26, 2018

## Overview

Starting with version 2.0, the OpenEXR image file format supports deep images. In a regular, or flat image, every pixel stores at most one value per channel. In contrast, each pixel in a deep image can store an arbitrary number of values or samples per channel. Each of those samples is associated with a depth, or distance from the viewer. Together with the two-dimensional pixel raster, the samples at different depths form a three-dimensional data set.

The open-source OpenEXR file I/O library defines the file format for deep images, and it provides convenient methods for reading and writing deep image files. However, the library does not define how deep images are meant to be interpreted. In order to encourage compatibility among application programs and image processing libraries, this document describes a standard way to represent point and volume samples in deep images, and it defines basic compositing operations such as merging two deep images or converting a deep image into a flat image.

# Contents

# Definitions

## Flat and Deep Images, Samples

For a single-part OpenEXR file an ***image*** is the set of all channels in the file. For a multi-part file an image is the set of all channels in the same part of the file.

A ***flat image*** has at most one stored value or ***sample*** per pixel per channel. The most common case is an RGB image, which contains three channels, and every pixel has exactly one $R$, one $G$ and one $B$ sample. Some channels in a flat image may be sub-sampled, as is the case with luminance-chroma images, where the luminance channel has a sample at every pixel, but the chroma channels have samples only at every second pixel of every second scan line.

A ***deep image*** can store an unlimited number of samples per pixel, and each of those samples is associated with a depth, or distance from the viewer.

A pixel at pixel space location $(x,y)$ in a deep image has $n(x,y)$ samples in each channel. The number of samples varies from pixel to pixel, and any non-negative number of samples, including zero, is allowed. However, all channels in a single pixel have the same number of samples.

The samples in each channel are numbered from $0$ to $n(x,y)-1$, and the expression $S_i(c,x,y)$ refers to sample number $i$ in channel $c$ of the pixel at location $(x,y)$.

In the following we will for the most part discuss a single pixel. For readability we will omit the coordinates of the pixel; expressions such as $n$ and $S_i(c)$ are to be understood as $n(x,y)$ and $S_i(c,x,y)$ respectively.

## Channel Names and Layers

The channels in an image have names that serve two purposes: specifying the intended interpretation of each channel, and grouping the channels into layers.

If a channel name contains one or more periods, then the part of the channel name that follows the last period is the ***base name***. If a channel name contains no periods, then the entire channel name is the base name.

[Examples: the base name of channel $R$ is $R$; the base name of channel $L1.L2.R$ is $R$.]

If a channel name contains one or more periods, then the part of the channel name before the last period is the channel's ***layer name***. If a channel name contains no periods, then the layer name is an empty string.

[Examples: the layer name of channel $R$ is the empty string; the layer name of channel $L1.L2.R$ is $L1.L2.$]

The set of all channels in an image that share the same layer name is called a ***layer***.

The set of all channels in an image whose layer name is the empty string is called the ***base layer***.

If the name of one layer is a prefix of the name of another layer, then the first layer ***encloses*** the second layer, and the second layer ***is nested in*** the first layer. Since the empty string is a prefix of any other string, the base layer encloses all other layers.

A layer ***directly encloses*** a second layer if there is no third layer that is nested in the first layer and encloses the second layer.

[Examples: layer $L1$ encloses layers $L1.L2$ and $L1.L2.L3$. Layer L1 directly encloses layer $L1.L2$, but $L1$ does not directly enclose $L1.L2.L3$.]

## Alpha, Color, Depth and Auxiliary Channels

A channel whose base name is $A$, $AR$, $AG$ or $AB$ is an ***alpha channel***. All samples must be greater than or equal to zero, and less than or equal to one.

A channel whose base name is $R$, $G$, $B$, or $Y$ is a ***color channel***.

A channel whose full name is $Z$ or $ZBack$, is a ***depth channel***. All samples in a depth channel must be greater than or equal to zero.

A channel that is not an alpha, color or depth channel is an ***auxiliary channel***.

## Required Depth Channels

The base layer of a deep image must include a depth channel that is called $Z$.

The base layer of a deep image may include a depth channel called $ZBack$. If the base layer does not include one, then a $ZBack$ channel can be generated by copying the $Z$ channel.

Layers other than the base layer may include channels called $Z$ or $ZBack$, but those channels are auxiliary channels and do not determine the positions of any samples in the image.

## Sample Locations, Point and Volume Samples

The depth samples $S_i(Z)$ and $S_i(ZBack)$ determine the positions of the front and the back of sample number $i$ in all other channels in the same pixel.

If $S_i(Z) \geq S_i(ZBack)$, then sample number $i$ in all other channels covers the single depth value $z = S_i(Z)$, where $z$ is the distance of the sample from the viewer. Sample number $i$ is called a ***point sample***.

If $S_i(Z) < S_i(ZBack)$, then sample number $i$ in all other channels covers the half open interval $S_i(Z) \leq z < S_i(ZBack)$. Sample number $i$ is called a ***volume sample***. $S_i(Z)$ is the sample's ***front*** and $S_i(ZBack)$ is the sample's ***back***.

Point samples are used to represent the intersections of surfaces with a pixel. A surface intersects a pixel at a well-defined distance from the viewer, but the surface has zero thickness. Volume samples are used to represent the intersections of volumes with a pixel.

## Required Alpha Channels

Every color or auxiliary channel in a deep image must have an ***associated alpha channel***.

The associated alpha channel for a given color or auxiliary channel, $c$, is found by looking for a ***matching*** alpha channel (see below), first in the layer that contains $c$, then in the directly enclosing layer, then in the layer that directly encloses that layer, and so on, until the base layer is reached. The first matching alpha channel found this way becomes the alpha channel that is associated with $c$.

Each color our auxiliary channel matches an alpha channel, as shown in the following table:

| Color or auxiliary channel base name | Matching alpha channel base name |
|---|---|
| $R$ | $AR$ if it exists, otherwise $A$ |
| $G$ | $AG$ if it exists, otherwise $A$ |
| $B$ | $AB$ if it exists, otherwise $A$ |
| $Y$ | $A$ |
| (any auxiliary channel) | $A$ |

[Example: The following table shows the list of channels in a deep image, and the associated alpha channel for each color or auxiliary channel.

| Channel name | Associated alpha channel |
|---|---|
| $A$ | |
| $AR$ | |
| $AG$ | |
| $R$ | $AR$ |
| $Z$ | |
| $L1.A$ | |

| | |
|---|---|
| $L1.AR$ | |
| $L1.R$ | $L1.AR$ |
| $L1.G$ | $L1.A$ |
| $L1.L2.G$ | $L1.A$ |

]

## Sorted, Non-Overlapping and Tidy Images

The samples in a pixel may or may not be sorted according to depth, and the sample depths or depth ranges may or may not overlap each other.

A pixel in a deep image is ***sorted*** if for every $i$ and $j$ with $i < j$,

$$S_i(Z) < S_j(Z) \vee \left(S_i(Z) = S_j(Z) \wedge S_i(ZBack) \leq S_j(ZBack)\right).$$

A pixel in a deep image is ***non-overlapping*** if for every $i$ and $j$ with $i \neq j$,

$$\left(S_i(Z) < S_j(Z) \wedge S_i(ZBack) \leq S_j(Z)\right) \vee \left(S_j(Z) < S_i(Z) \wedge S_j(ZBack) \leq S_i(Z)\right) \vee ¿$$

$$\left(S_i(Z) = S_j(Z) \wedge S_i(ZBack) \leq S_i(Z) \wedge S_j(ZBack) > S_j(Z)\right) \vee ¿$$

$$\left(S_j(Z) = S_i(Z) \wedge S_j(ZBack) \leq S_j(Z) \wedge S_i(ZBack) > S_i(Z)\right).$$

A pixel in a deep image is ***tidy*** if it is sorted and non-overlapping.

A deep image is sorted if all of its pixels are sorted; it is non-overlapping if all of its pixels are non-overlapping; and it is tidy if all of its pixels are tidy.

The images stored in an OpenEXR file are not required to be tidy. Some deep image processing operations, for example, flattening a deep image, require tidy input images. However, making an image tidy loses information, and some kinds of data cannot be represented with tidy images, for example, object identifiers or motion vectors for volume objects that pass through each other.

Some application programs that read deep images can run more efficiently with tidy images. For example, in a 3D renderer that uses deep images as shadow maps, shadow lookups are faster if the samples in each pixel are sorted and non-overlapping.

Application programs that write deep OpenEXR files can add a deepImageState attribute to the header to let file readers know if the pixels in the image are tidy or not. The attribute is of type DeepImageState, and can have the following values:

| Value | Interpretation |
|---|---|
| MESSY | Samples may not be sorted, and overlaps are possible. |
| SORTED | Samples are sorted, but overlaps are possible. |
| NON_OVERLAPPING | Samples do not overlap, but may not be sorted. |

| | |
|---|---|
| TIDY | Samples are sorted and do not overlap. |

If the header does not contain a deepImageState attribute, then file readers should assume that the image is MESSY. The OpenEXR file I/O library does not verify that the samples in the pixels are consistent with the deepImageState attribute. Application software that handles deep images may assume that the attribute value is valid, as long as the software will not crash or lock up if any pixels are inconsistent with the deepImageState.

## Alpha and Color as Functions of Depth

Given a color channel, $c$, and its associated alpha channel, $\alpha$, the samples $S_i(c)$, $S_i(\alpha)$, $S_i(Z)$ and $S_i(ZBack)$ together represent the intersection of an object with a pixel. The color of the object is $S_i(c)$, its opacity is $S_i(\alpha)$, and the distances of its front and back from the viewer are indicated by $S_i(Z)$ and $S_i(ZBack)$ respectively.

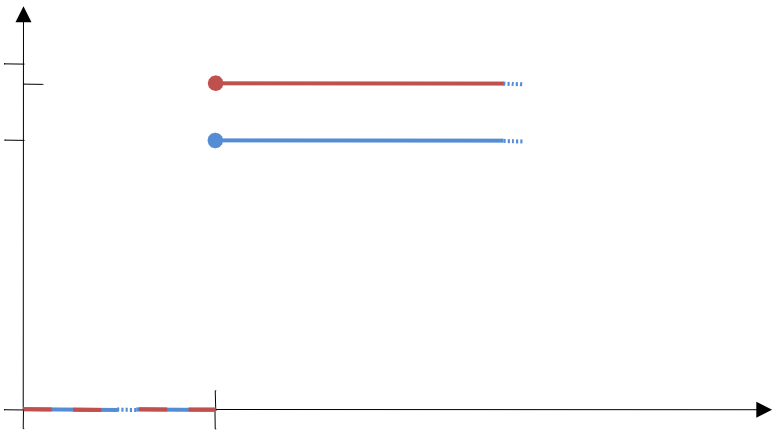
### One Sample

We now define two functions, $z \longmapsto \alpha_i(z)$, and $z \longmapsto c_i(z)$, that represent the opacity and color of the part of the object whose distance from the viewer is no more than $z$. In other words, we divide the object into two parts by splitting it at distance $z$; $\alpha_i(z)$ and $c_i(z)$ are the opacity and color of the part that is closer to the viewer.

For a point sample, $\alpha_i(z)$ and $c_i(z)$ are step functions:

$$\alpha_i(z) = \begin{cases} 0, & z < S_i(Z) \\ S_i(\alpha), & z \geq S_i(Z) \end{cases}$$

$$c_i(z) = \begin{cases} 0, & z < S_i(Z) \\ S_i(c), & z \geq S_i(Z) \end{cases}$$



For a volume sample, we define a helper function $x(z)$ that consists of two constant segments and a linear ramp:

$$x(z)=\begin{cases} 0, & z\le S_i(Z) \\ \dfrac{z-S_i(Z)}{S_i(ZBack)-S_i(Z)}, & S_i(Z)<z<S_i(ZBack) \\ 1, & z\ge S_i(ZBack) \end{cases}$$

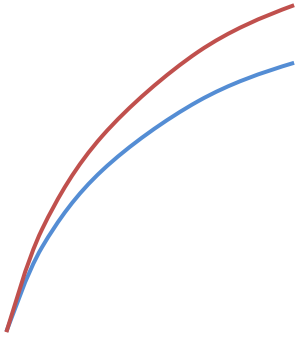
With this helper function, $\alpha_i(z)$ and $c_i(z)$ are defined as follows:

$$\alpha_i(z)=1-\left(1-S_i(\alpha)\right)^{x(z)}$$

$$c_i(z)=\begin{cases} S_i(c)\cdot\dfrac{\alpha_i(z)}{S_i(\alpha)}, & S_i(\alpha)>0 \\ S_i(c)\cdot x(z), & S_i(\alpha)=0 \end{cases}$$

Note that the second case in the definition of $c_i(z)$ is the limit of the first case as $S_i(\alpha)$ approaches zero.

The figure below shows an example of $\alpha_i(z)$ and $c_i(z)$ for a volume sample. Alpha and color are zero up to $Z$, increase gradually between $Z$ and $ZBack$, and then remain constant.



## Whole Pixel

If a pixel is tidy, then we can define two functions, $z \longmapsto A(z)$, and $z \longmapsto C(z)$, that represent the total opacity and color of all objects whose distance from the viewer is no more than $z$: if the distance $z$ is inside a volume object, we split the object at $z$. Then we use "over" operations to composite all objects that are no further away than $z$.

Given a foreground object with opacity $\alpha_f$ and color $c_f$, and a background object with opacity $\alpha_b$ and color $c_b$, an "over" operation computes the total opacity and color, $\alpha$ and $c$, that result from placing the foreground object in front of the background object:

$$\alpha = \alpha_f + (1-\alpha_f)\cdot\alpha_b$$

$$c = c_f + (1-\alpha_f)\cdot c_b$$

We define two sets of helper functions:

$$A_i(z) = \begin{cases} 0, & i<0 \\ A_{i-1}(S_i(Z)) + (1-A_{i-1}(S_i(Z)))\cdot\alpha_i(z), & i\geq 0 \end{cases}$$

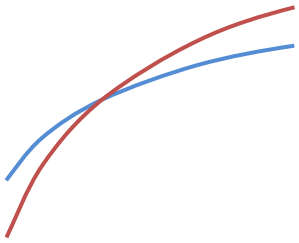
$$C_i(z) = \begin{cases} 0, & i<0 \\ C_{i-1}(S_i(Z)) + (1-A_{i-1}(S_i(Z)))\cdot c_i(z), & i\geq 0 \end{cases}$$

With these helper functions, $A(z)$ and $C(z)$ look like this:

$$A(z) = \begin{cases} A_{-1}(z), & z<S_0(Z) \\ A_i(z), & S_i(Z)\leq z<S_{i+1}(Z) \\ A_{n-1}(z), & S_{n-1}(Z)\leq z \end{cases}$$

$$C(z)=\begin{cases} C_{-1}(z), & z<S_0(Z) \\ C_i(z), & S_i(Z)\le z<S_{i+1}(Z) \\ C_{n-1}(z), & S_{n-1}(Z)\le z \end{cases}$$

The figure below shows an example of $A(z)$ and $C(z)$. Sample number $i$ is a volume sample; its $ZBack$ is greater than its $Z$. Alpha and color increase gradually between $Z$ and $ZBack$ and then remain constant. Sample number $i$+1, whose $Z$ and $ZBack$ are equal, is a point sample where alpha and color discontinuously jump to a new value.



## Basic Deep Image Operations

Given the definitions above, we can now construct a few basic deep image processing operations.

### Splitting a Volume Sample

Our first operation is splitting volume sample number $i$ of a pixel at a given depth, $z$, where

$$S_i(Z) < z < S_i(ZBack).$$

The operation replaces the original sample with two new samples. If the first of those new samples is composited over the second one, then the total opacity and color are the same as in the original sample.

For the depth channels, the new samples are:

$$S_{i,new}(Z) = S_i(Z) \qquad S_{i+1,new}(Z) = z$$

$$S_{i,new}(ZBack) = z \qquad S_{i+1,new}(ZBack) = S_i(ZBack)$$

For a color channel, $c$, and its associated alpha channel, $\alpha$, the new samples are:

$$S_{i,new}(\alpha) = \alpha_i(z) \qquad S_{i+1,new}(\alpha) = \alpha_i(S_i(Z) + S_i(ZBack) - z)$$

$$S_{i,new}(c) = c_i(z) \qquad S_{i+1,new}(c) = c_i(S_i(Z) + S_i(ZBack) - z)$$

If it is not done exactly right, splitting a sample can lead to large rounding errors for the colors of the new samples when the opacity of the original sample is very small. For C++ code that splits a volume sample in a numerically stable way, see page 19 of this document.

### Merging Overlapping Samples

In order to make a deep image tidy, we need a procedure for merging two samples that perfectly overlap each other. Given two samples, $i$ and $j$, with

$$S_i(Z) = S_j(Z)$$

and

$$max\left(S_i(Z), S_i(ZBack)\right) = max\left(S_j(Z), S_j(ZBack)\right),$$

we want to replace those samples with a single new sample that has an appropriate opacity and color.

For two overlapping volume samples, the opacity and color of the new sample should be the same as what one would get from splitting the original samples into a very large number of shorter sub-samples, interleaving the sub-samples, and compositing them back together with a series of "over" operations.

For a color channel, $c$, and its associated alpha channel, $\alpha$, we can compute the opacity and color of the new sample as follows:

$$S_{i,new}(\alpha) = 1 - (1 - S_i(\alpha)) \cdot (1 - S_j(\alpha))$$

$$S_{i,new}(c) = \begin{cases} \dfrac{S_i(c) + S_j(c)}{2}, & S_i(\alpha) = 1 \wedge S_j(\alpha) = 1 \\ S_i(c), & S_i(\alpha) = 1 \wedge S_j(\alpha) < 1 \\ S_j(c), & S_i(\alpha) < 1 \wedge S_j(\alpha) = 1 \\ \dfrac{S_i(c) \cdot v_i + S_j(c) \cdot v_j}{w}, & S_i(\alpha) < 1 \wedge S_j(\alpha) < 1 \end{cases}$$

where

$$u_k = \begin{cases} -\log(1 - S_k(\alpha)), & S_k(\alpha) > 0 \\ 0, & S_k(\alpha) = 0 \end{cases}$$

$$v_k = \begin{cases} \dfrac{u_k}{S_k(\alpha)}, & S_k(\alpha) > 0 \\ 1, & S_k(\alpha) = 0 \end{cases}$$

with $k = i$ or $k = j$, and

$$w = \begin{cases} \dfrac{S_{i,new}(\alpha)}{u_i + u_j}, & u_i + u_j \neq 0 \\ 1, & u_i + u_j = 0 \end{cases}$$

Evaluating the expressions above directly can lead to large rounding errors when the opacity of one or both of the input samples is very small. For C++ code that computes $S_{i,new}(\alpha)$ and $S_{i,new}(c)$ in a numerically robust way, see page 20 of this document.

For details on how the expressions for $S_{i,new}(\alpha)$ and $S_{i,new}(c)$, can be derived, see Peter Hillman's paper, "Deep Sample Merging."

Note that the expressions for computing $S_{i,new}(\alpha)$ and $S_{i,new}(c)$ do not refer to depth at all. This allows us to reuse the same expressions for merging two perfectly overlapping (that is, coincident) point samples.

A point sample cannot perfectly overlap a volume sample; therefore point samples are never merged with volume samples.

## Making an Image Tidy

An image is made tidy by making each of its pixels tidy. A pixel is made tidy in three steps:

1. Split partially overlapping samples: if there are indices $i$ and $j$ such sample $i$ is either a point or a volume sample, sample $j$ is a volume sample, and $S_j(Z)<S_i(Z)<S_j(ZBack)$, then split sample $j$ at $S_i(Z)$ as shown on page 11 of this document. Otherwise, if there are indices $i$ and $j$ such that samples $i$ and $j$ are volume samples, and $S_j(Z)<S_i(ZBack)<S_j(ZBack)$, then split sample $j$ at $S_i(ZBack)$. Repeat this until there are no more partially overlapping samples.

   Example (horizontal lines are volume samples, vertical lines are point samples, and the r
   e– horizontal lines are point samples:

   After splitting:

2. Merge overlapping samples: if there are indices $i$ and $j$ such that samples $i$ and $j$ overlap perfectly, then merge those two samples as shown in "Merging Overlapping Samples," above. Repeat this until there are no more perfectly overlapping samples.

   After merging:

3. Sort the samples according to $Z$ and $ZBack$ (see "Sorted, Non-Overlapping and Tidy Images" on page 6).

After sorting:

Note that this procedure can be made more efficient by first sorting the samples, and then splitting and merging overlapping samples in a single front-to-back sweep through the sample list.

## Merging two Images

Merging two deep images forms a new deep image that represents all of the objects contained in both of the original images. Conceptually, the deep image "merge" operation is similar to the "over" operation for flat images, except that the "merge" operation does not distinguish between a foreground and a background image.

Since deep images are not required to be tidy, the "merge" operation is trivial: for each output pixel, concatenate the sample lists of the corresponding input pixels.

## Flattening an Image

Flattening produces a flat image from a deep image by performing a front-to-back composite of the deep image samples. The "flatten" operation has two steps:

1. Make the deep image tidy.
2. For each pixel, composite sample $0$ over sample $1$. Composite the result over sample $2$, and so on, until sample $n-1$ is reached.

   Note that this is equivalent to computing $A\left(max\left(S_{n-1}(Z), S_{n-1}(ZBack)\right)\right)$ for each alpha channel and $C\left(max\left(S_{n-1}(Z), S_{n-1}(ZBack)\right)\right)$ for each color or auxiliary channel.

There is no single "correct" way to flatten the depth channels. The most useful way to handle $Z$ and $ZBack$ depends on how the flat image will be used. Possibilities include, among others:

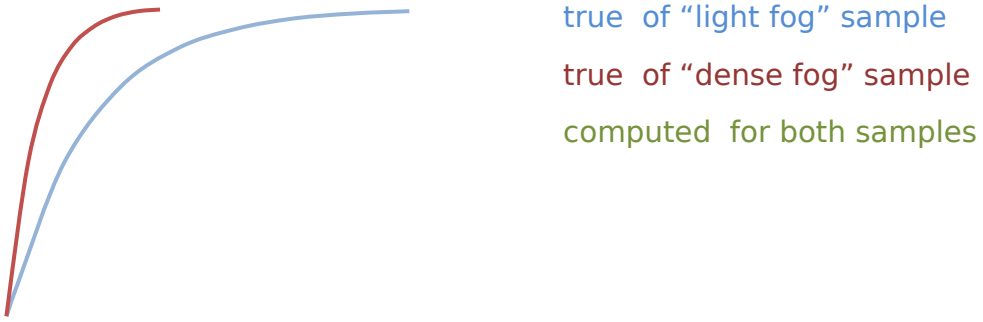
- Flatten the $Z$ channel as if it was a color channel, using $A$ as the associated alpha channel. For volume samples, replace $Z$ with the average of $Z$ and $ZBac$k before flattening. Either discard the $ZBack$ channel, or use the back of the last sample, $max\left(S_{n-1}(Z), S_{n-1}(ZBack)\right)$, as the $ZBack$ value for the flat image.
- Treating $A$ as the alpha channel associated with $Z$, find the depth where $A(z)$ becomes 1.0 and store that depth in the $Z$ channel of the flat image. If $A(z)$ never reaches 1.0, then store either infinity or the maximum possible finite value in the flat image.

- Treating $A$ as the alpha channel associated with $Z$, copy the front of the first sample with non-zero alpha and the front of the first opaque sample into the $Z$ and $ZBack$ channels of the flat image.

## Opaque Volume Samples

Volume samples represent regions along the $z$ axis of a pixel that are filled with a medium that absorbs light and also emits light towards the camera. The intensity of light traveling through the medium falls off exponentially with the distance traveled. For example, if a one unit thick layer of fog absorbs half of the light and transmits the rest, then a two unit thick layer of the same fog absorbs three quarters of the light and transmits only one quarter. Volume samples representing these two layers would have alpha 0.5 and 0.75 respectively. As the thickness of a layer increases, the layer quickly becomes nearly opaque. A fog layer that is twenty units thick transmits less than one millionth of the light entering it, and its alpha is 0.99999905. If alpha is represented using 16-bit floating-point numbers, then the exact value will be rounded to 1.0, making the corresponding volume sample completely opaque. With 32-bit floating-point numbers, the alpha value for a 20 unit thick layer can still be distinguished from 1.0, but for a 25 unit layer, alpha rounds to 1.0. At 55 units, alpha rounds to 1.0 even with 64-bit floating-point numbers.

Once a sample effectively becomes opaque, the true density of the light-absorbing medium is lost. A one-unit layer of a light fog might absorb half of the light while a one-unit layer of a dense fog might absorb three quarters of the light, but the representation of a 60-unit layer as a volume sample is exactly the same for the light fog, the dense fog and a gray brick. For a sample that extends from $Z$ to $ZBack$, the function $\alpha(z)$ evaluates to 1.0 for any $z>Z$. Any object within this layer would be completely hidden, no matter how close it was to the front of the layer.




Application software that writes deep images should avoid generating very deep volume samples. If the program is about to generate a sample with alpha close to 1.0, then it should split the sample into multiple sub-samples with a lower opacity before storing the data in a deep image file. This assumes, of course, that the

software has an internal volume sample representation that can distinguish very nearly opaque samples from completely opaque ones, so that splitting will produce sub-samples with alpha significantly below 1.0.

# Appendix: C++ Code

## Splitting a Volume Sample

```
#include <algorithm>
#include <limits>
#include <cmath>
#include <cassert>

using namespace std;


void
splitVolumeSample
    (float   a, float   c,  // Opacity and color of original sample
     float  zf, float  zb,  // Front and back of original sample
     float   z,             // Position of split
     float& af, float& cf,  // Opacity and color of part closer than z
     float& ab, float& cb)  // Opacity and color of part further away than z
{
    //
    // Given a volume sample whose front and back are at depths zf and zb
    // respectively, split the sample at depth z.  Return the opacities
    // and colors of the two parts that result from the split.
    //
    // The code below is written to avoid excessive rounding errors when
    // the opacity of the original sample is very small:
    //
    // The straightforward computation of the opacity of either part
    // requires evaluating an expression of the form
    //
    //      1 - pow (1-a, x).
    //
    // However, if a is very small, then 1-a evaluates to 1.0 exactly,
    // and the entire expression evaluates to 0.0.
    //
    // We can avoid this by rewriting the expression as
    //
    //      1 - exp (x * log (1-a)),
    //
    // and replacing the call to log() with a call to the function log1p(),
    // which computes the logarithm of 1+x without attempting to evaluate
    // the expression 1+x when x is very small.
    //
    // Now we have
    //
    //      1 - exp (x * log1p (-a)).
    //
    // However, if a is very small then the call to exp() returns 1.0, and
    // the overall expression still evaluates to 0.0.  We can avoid that
    // by replacing the call to exp() with a call to expm1():
    //
    //      -expm1 (x * log1p (-a))
    //
    // expm1(x) computes exp(x) - 1 in such a way that the result is accurate
    // even if x is very small.
    //

    assert (zb > zf && z >= zf && z <= zb);
```

```
    a = max (0.0f, min (a, 1.0f));

    if (a == 1)
    {
        af = ab = 1;
        cf = cb = c;
    }
    else
    {
        float xf = (z - zf) / (zb - zf);
        float xb = (zb - z) / (zb - zf);

        if (a > numeric_limits<float>::min())
        {
            af = -expm1 (xf * log1p (-a));
            cf = (af / a) * c;

            ab = -expm1 (xb * log1p (-a));
            cb = (ab / a) * c;
        }
        else
        {
            af = a * xf;
            cf = c * xf;

            ab = a * xb;
            cb = c * xb;
        }
    }
}
```

## Merging two Overlapping Samples

```
#include <algorithm>
#include <limits>
#include <cmath>
#include <cassert>

using namespace std;


void
mergeOverlappingSamples
    (float  a1, float  c1,  // Opacity and color of first sample
     float  a2, float  c2,  // Opacity and color of second sample
     float& am, float& cm)  // Opacity and color of merged sample
{
    //
    // This function merges two perfectly overlapping volume or point
    // samples.  Given the color and opacity of two samples, it returns
    // the color and opacity of the merged sample.
    //
    // The code below is written to avoid very large rounding errors when
    // the opacity of one or both samples is very small:
    //
    // * The merged opacity must not be computed as 1 - (1-a1) * (1-a2).
    //   If a1 and a2 are less than about half a floating-point epsilon,
    //   the expressions (1-a1) and (1-a2) evaluate to 1.0 exactly, and the
    //   merged opacity becomes 0.0.  The error is amplified later in the
    //   calculation of the merged color.
    //
```

```
    //   Changing the calculation of the merged opacity to a1 + a2 - a1*a2
    //   avoids the excessive rounding error.
    //
    // * For small x, the logarithm of 1+x is approximately equal to x,
    //   but log(1+x) returns 0 because 1+x evaluates to 1.0 exactly.
    //   This can lead to large errors in the calculation of the merged
    //   color if a1 or a2 is very small.
    //
    //   The math library function log1p(x) returns the logarithm of
    //   1+x, but without attempting to evaluate the expression 1+x
    //   when x is very small.
    //

    a1 = max (0.0f, min (a1, 1.0f));
    a2 = max (0.0f, min (a2, 1.0f));

    am = a1 + a2 - a1 * a2;

    if (a1 == 1 && a2 == 1)
    {
        cm = (c1 + c2) / 2;
    }
    else if (a1 == 1)
    {
        cm = c1;
    }
    else if (a2 == 1)
    {
        cm = c2;
    }
    else
    {
        static const float MAX = numeric_limits<float>::max();

        float u1 = -log1p (-a1);
        float v1 = (u1 < a1 * MAX)? u1 / a1: 1;

        float u2 = -log1p (-a2);
        float v2 = (u2 < a2 * MAX)? u2 / a2: 1;

        float u = u1 + u2;
        float w = (u > 1 || am < u * MAX)? am / u: 1;

        cm = (c1 * v1 + c2 * v2) * w;
    }
}
```