Deep Images and Cutouts

Janne Kontkanen

April 2010

Abstract

This document describes the basic compositing operations possible for deep image data format. We explain the idea of continuous alpha blending, and present formulas for computing *cutout images* that can be accurately composited by an ADD operation in the post-process compositing stage.

The last section also talks about direct generation of *cutout images* from the renderer.

1 What is a Deep Image?

Consider a single pixel in a deep image. Each pixel contains four functions corresponding to the color and alpha. All of these functions depend on the depth: $r(z)$, $g(z)$, $b(z)$, $v(z)$. The value of each function equals to the value of that channel as if you would discard everything beyond that distance. In other words, if you evaluated $r(z)$ channel at a certain z' for all the pixels in an image, you would get a red channel similar to if you put a far plane to that z' when rendering.

2 Basic operations – Adding Elements During Rendering

Adding an element into a pixel of a deep image is the basic operation that happens a lot during rendering. As a prerequisite for a simple algorithm, we need to assume that we can process the elements falling within each pixel in a front-to-back order. Fortunately withing DWAs system this straightforward since our particle renderer can sort the particles prior to rendering, and our micropolygon renderer constructs deep files where the surfaces falling within a pixel can be processed in depth order. We will also assume that z increases as we go farther away from the camera (opposite to the convention of our software).

So, assume the rendering happens in front-to-back order into a standard (flat) R, G, B, A framebuffer. Each time we composite a particle, surface (or any element), into a pixel, that denotes an instantaneous change at that depth. So, we simply perform an update operation to the deep image such that $r(z)$,

DreamWorks Animation Technical Report 2010-362

$g(z)$, $b(z)$, and $a(z)$ jump from value $(R,G,B,A)_{previous}$ to $(R,G,B,A)_{new}$. In the future we may also talk about rendering more volumetric elements, so despite this explanation, don't assume the r,g,b,a really are 'staircase shaped'-functions – it is beneficial to also support smoother functions. So in the following we continue talking about these functions in abstract manner, until we move to discussing a the specific case of piecewise linear functions.

What ever renderer does to blend results into the framebuffer, we treat as an abstract operation – we do not care about that here.

3 Merging and Cutouts using Deep Images

Deep images store enough data to allow something that we can call continuous alpha blending, i.e. merging two images together such that the depth relationships are accurately resolved.

Consider two images, with color and visibility channels c_1, v_1, and c_2 and v_2. You can think of c referring to any of the three possible color channels.

When merging image 1 into image 2, the contribution of the color channels from the image 1 is dimmed by the occlusion caused by the image 2. We denote the resulting color $c_{1cutout2}$

This can be written down as follows:

$$c_{1cutout2}(z') = \int_0^{z'} (1 - v_2(z)) \frac{\partial c_1(z)}{\partial z} dz \tag{1}$$

This is *continuous alpha-blending* – we apply the occlusion described by v_2 to each differential change of color c_1. If we apply the above to all the pixel of the image, we get something that we typically call a *cutout*.

Consider merging the two images. You can directly reconstruct a combined color channel from the two deep images as follows, by talking into account the color contributions from both, and applying the above cutout-formula two ways:

$$c_{1merge2}(z') = \int_0^{z'} (1 - v_2(z)) \frac{\partial c_1(z)}{\partial z} + (1 - v_1(z)) \frac{\partial c_2(z)}{\partial z} dz \tag{2}$$

The above equation works for any combination of surface or particle renderings: you can use it to combine two particle renderings, two surface renderings, or surface rendering and a particle rendering. This is the beautiful thing about the accumulated visibility function.

You can use 2 for merging two images directly, or you can use 1 to compute a cutout-image. If you compute cutout images two ways by swapping the roles of the two images and later composite the resulting flat raster images together with an ADD-operation you get the same result as if you merged the images directly.

4 Practical Formulas for Piecewise Linear c and v

In practice, we currently have two modes to store the functions of depth: piecewise constant and piecewise linear representation. Our deep shadow maps are piecewise linear and deep images piecewise constant. There is probably no fundamental reason for this, but since we haven't yet implemented merging for the piecewise linear case, there is really no use for piecewise linear deep images.

Evaluating cutouts/merging in the piecewise constant case is almost trivial (at least mathwise), so we go directly into discussing piecewise linear $c(z)$ and $v(z)$.

We can obtain a cutout image for a piecewise linear case, consider a single span $[z_0, z_1]$ where both images are purely linear. So we can denote the derivatives with constants: $\frac{\partial c_1(z)}{\partial z} = K_{c_1}$, $\frac{\partial v_1(z)}{\partial z} = K_{v_1}$ within this span.

This gives us the following:

$$span(z_0, z_1)_{1cutout2} = \int_{z_0}^{z_1} (1 - [v_2(z_0) + (z - z_0)K_{v_2}])K_{c_1} dz \tag{3}$$

Integrating and reorganizing a bit, this gives:

$$span(z_0, z_1)_{1cutout2} = z_1 \left[1 - v_2(z_0) + K_{v_2} z_0\right] - \frac{K_{v_2} z_1^2}{2} - z_0 \left[1 - v_2(z_0) + K_{v_2} z_0\right] + \frac{K_{v_2} z_0^2}{2} \tag{4}$$

To do the full image, walk through all the spans $[z_i, z_{i+1}]$, in which all the involved functions are linear. Then sum the contributions evaluated according to the above equation. To obtain the other cutout image, swap the roles of the images.

5 Visibility Channel and Deep Shadow Maps

The visibility channel should go through a similar treatment as the color channels above. So when computing the visibility channel for a cutout image, Equation 1 can be used to apply the cutout from v_2 to v_1 and the other way around. Deep shadow map is a deep depth map that stores only visibility channels. If the deep shadow map is monochrome, there is a single visibility channel. If the deep shadow map is colored, there are three visibility channels.

6 Generating Cutouts During Rendering

All the explanation in this document so far has been written in context of deep images, where we assume that functions r, g, b, and v are explicitly stored. However, direct rendering of cutout images is also possible, and in fact, this is what we have done (with different approximations) for years in **pa_render** via

dm and **dm_soft** particle shaders. Since we are adding particle and volume rendering support to **light**, it is natural to allow applying cutouts arbitrarily regardless of what type of elements we are rendering. For instance, we may want to *cut out* a volume from a group of particles and surfaces. Some of these procedures are not directly needed in the production workflows, but still, it does not make sense to restrict this procedure to certain types of primitives and develop special purpose cutout code for those. For code maintenance, it seems like it would be the best if the cutouts were natively supported in the renderer.

Recall that applying image 2 into image 1 as a cutout means multiplication with $(1 - v_2(z))$ as we integrate/composite (Equation 1). To evaluate this directly in the **/lib/dshade** compositing code, we to maintain $v_2(z)$ as we proceed compositing front-to-back. To do this, we need to introduce an auxiliary alpha channel that is used during the front-to-back compositing loop. In **d2r** we call this a *cutout channel*. We also simplify the problem this time, by not thinking about linear interpolation, but assuming that each surface/particle or volume element causes an instantaenous change to r, g, b, v or v_2. The cutout channel (v_2) functions as follows.

Any surface, volume or particle can tag itself as a cutout (a special material shader output variable). This is currently done by **cutout** material and **cutout transparency** material, but more shaders may use this flag if particles and volumes are used as cutouts.

If material is tagged as a cutout, the compositor will treat the alpha channel of that material as a cutout, meaning that the alpha gets blended into the cutout channel v_2, and not into the alpha channel v_1. As the compositing proceeds front-to-back, the v_2 will be always available in the cutout channel.

 # The front-to-back compositing loop in pseudo-code. In practice
 # the implementation looks different due to optimizations and support
 # for colored alpha, but the basic idea remains.
#
 # NOTE: Prior to entering this loop the following is carried out for
 # any fragment tagged as cutout:
$c_{src}[ALPHA] \rightarrow c_{src}[CUTOUT]$
$c_{src}[ALPHA] = 0$
#
for *each subpixel* c_{dst} **do**
    clear(c_{dst})
    for *each fragment,* c_{src}*, in front-to-back order* **do**
        for $i := RED, GREEN, BLUE, ALPHA, CUTOUT$ **do**
            $c_{dst}[i] = c_{dst}[i] + (1 - c_{dst}[CUTOUT])(1 - c_{dst}[ALPHA])c_{src}[i]$
        end