UCRL-ID-153185

Alpha Dithering to Correct Low-Opacity 8 Bit Compositing Errors

P. L. Williams, R. J. Frank, E. C. LaMar

March 31, 2003

Alpha Dithering to Correct Low-Opacity 8 Bit Compositing Errors

Peter L. Williams, Randall J. Frank, and Eric C. LaMar

Lawrence Livermore National Laboratory
Livermore, CA 94551

March 31, 2003

Abstract

This paper describes and analyzes a dithering technique for accurately specifying small values of opacity (α) that would normally not be possible because of the limited number of bits available in the alpha channel of graphics hardware. This dithering technique addresses problems related to compositing numerous low-opacity semitransparent polygons to create volumetric effects with graphics hardware. The paper also describes the causes and a possible solution to artifacts that arise from parallel or distributed volume rendering using bricking on multiple GPU's.

Keywords: Transparency, volume rendering, alpha compositing.

1 INTRODUCTION

This paper addresses problems related to compositing numerous low-opacity semitransparent polygons to create volumetric effects with graphics hardware that uses limited precision arithmetic. This includes the low-opacity fog problem and the bricking artifact problem, both of which are described in detail in Section 2. The low-opacity fog problem has to do with rendering polygons whose opacities are too low to register in 8 bit graphics hardware. The bricking artifact problem occurs in parallel or distributed volume rendering using bricking on multiple GPU's.

In Section 2, we discuss the background for our paper, give some of the assumptions regarding our calculations, and describe the low-opacity fog problem and the bricking artifact problem. Then in Section 3 we present the alpha dithering technique. In Section 4 we show results of applying this dithering technique to different instances of the low-opacity fog problem and the bricking artifact problem. In Section 5, we discuss dithering color. Then in Section 6 we discuss how we implement dithering in the transfer functions. In Section 8 we cover the effect of dithering on image quality and rendering performance, discuss front to back dithering, present an unsolved problem in regard to our dithering method, describe an alternative dithering technique using chunks, and some related issues. Section 9 discusses some limitations of our method and gives our conclusions.

2 BACKGROUND

Semitransparent objects can be rendered by specifying an opacity (α) value in addition to a color. This α value which ranges between 0.0 (totally transparent) and 1.0 (totally opaque) is used to blend the color of an incoming fragment with the color of a pixel stored in the frame buffer. Assuming the objects are rendered from back to front in visibility order, the blending equation normally used [18], is:

$$NewPixelColor = (\alpha \times c) + (1 - \alpha) \times OldPixelColor. \tag{1}$$

Where α and c are the opacity and color, respectively, of the incoming fragment.

Graphics hardware generally has an alpha channel which allows the specification and storage of an opacity value as a fourth color component (in addition to R, G and B). The hardware also implements several methods that control how color values in the incoming fragment are combined with those already stored in the frame buffer. One of these methods implements Equation 1. We refer to this process as *compositing*. However, graphics hardware may offer a limited number n of bits for the specification and storage of color and alpha values, thus limiting these values to integers in the range $[0, 2^{n-1}]$ which maps to 2^n different allowable colors and opacities in the range $[0, 1.0]$, with a minimum possible value of $\frac{1}{n-1}$ which we refer to as the *input threshold*. Typically, $n = 8$ bits, then the input threshold is $\frac{1}{255}$, and there are 256 allowable color and opacity values. This can be a problem if we need to input an opacity value lower than the input threshold or a value that lies between two consecutive allowable input values. This situation can arise in volume rendering or in creating volumetric effects such as rendering fire and fog. We will focus on volume rendering applications.

In volume rendering, each data value is mapped to a color and an opacity by transfer functions. Volume rendering may be accomplished in a number of ways, such as: (a) by slicing the data perpendicular the view direction to form slices, where each slice is composed of a set of polygons formed by the intersection of the slice with the cells in the mesh, and then rendering the slices from back to front using hardware compositing, as described in [24, 28], (b) by rendering in visibility order the projection of each cell as a set of polygons, using hardware compositing, as described in [20, 22], or (c) by resampling the data to a set of 2D textures or a 3D texture, as described in [6, 10, 13, 19, 25], which can be rendered from back to front, using compositing, as 2D textures or as one 3D texture. In the case of textures, each 2D texture can be thought of as a slice for the purpose of this paper.

Until the release of the R300 based graphics card from ATI Technologies Inc, and the NV30 based card from NVIDIA Corporation, both of which have 128 bit floating point frame buffers (32 bits per channel), most graphics hardware offered limited precision, usually 32 bit frame buffers, with

8 bits per channel. On some high-end SGI machines, the frame buffers have 48 bits, 12 bits per channel. In this paper, we focus on dealing with the limitations presented by having only 8 bits per channel. While the R300, NV30 and later cards have 128 bit floating point frame buffers, our work dealing with graphics hardware with only 8 bits per channel is still relevant for at least three reasons. First, there will be many legacy 32 bit graphics cards in use for a number of years. Second, the R300 and NV30 generation chips have certain limitations on the use of floating point compositing that may make the use of 8 bit compositing still relevant. For example, in the current generation of cards, no blending operations are supported when targeting a floating point frame buffer, instead the frame buffer must be copied to a texture where blending is accomplished by a user provided fragment program. Third, images created using these new graphics cards, which involve compositing for volumetric effects, may appear different than images created with 32 bit frame buffers using the same data and viewing parameters — this paper explains why this may be the case.

2.1 Preliminary Assumptions and Definitions

By *n bit compositing*, we mean that the opacity and color values are stored, and specified as input, (as integers), using n bits. However, the blending equation, Equation 1, is assumed to be evaluated using floating point arithmetic, with n bit integer inputs, and the result rounded to an n bit integer. So if the equation to be evaluated is $c = f(a, b)$, where a, b, c are 8 bit integers, then we would evaluate it as $c = round(f(float(a), float(b)))$. The results presented in this paper were generated using a graphics hardware simulator based on these assumptions. We have compared the output from our simulator with the output from an NVIDIA GeForce3 GPU, and found them to be in general agreement. The GeForce3 has a 32 bit frame buffer with 8 bits per channel. In this paper we focus on 8 bit compositing, and use 32 bit compositing as a high accuracy frame of reference. In Section 4.1, we briefly discuss 12 bit compositing.

Normally 8 bit color and opacity values are specified either as (a discrete set of 256) floating point values in the range $[0.0, 1.0]$, or as integers in the range $[0, 255]$. In this paper, we have chosen the latter representation, but with the modification that values are expressed in floating point notation in the range $[0.0, 255.0]$. We made this choice for two reasons. We wish to be able to describe opacity and color values that are below the input threshold or that lie between two consecutive allowable input values. Second, this representation makes it obvious which are the allowable input values. Thus an opacity given as 5 in this paper will be equivalent to an opacity of 5/255 on a range of $[0, 1.0]$, and an opacity of 1.25 will be equivalent to an opacity of 1.25/255 on a range of $[0.0, 1.0]$. Obviously, an opacity of 5 is an allowable input value, whereas 1.25 is not. We emphasize again that unless otherwise specified, all opacity and color values v given in this paper will be values in the range $[0.0, 255.0]$ that correspond to $v/255$ in the range $[0.0, 1.0]$. Finally, when we specify a color value, e.g. $c = 128$, we mean the value of any one of R, G, or B.

2.2 Low-Opacity Fog Problem

For slice-based rendering, an image may be generated using progressive refinement: initially a few slices are rendered, then as time permits the resolution of the image is improved by rendering it with more slices. However, when the number of slices is increased, the opacity values need to be decreased. To understand this, we need to look at what we mean by a slice and how opacity is defined.

Although we speak of slices, conceptually the volume is partitioned into slabs perpendicular to the viewing direction and each slab is represented by a slice, centered in the slab. See Figure 1. When we render a slice, conceptually we are rendering a slab. It is often loosely said that a slice has an opacity or that the opacity needs to be adjusted as the slice spacing or slice density changes. This can be confusing; it is preferable to think in terms of rendering slabs rather than slices.

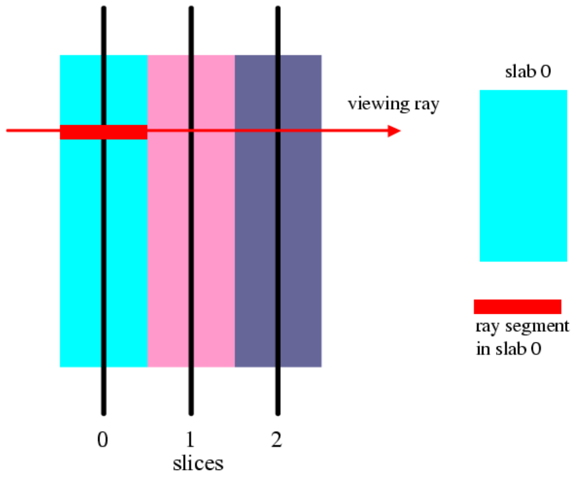

Figure 1: A volume partitioned into three slabs, showing the three slices that represent them, and a viewing ray through the slabs.

Based on the optical model that is often used for volume rendering, the *absorption plus emission* model described in [14, 22, 23], the opacity α is defined as:

$$\alpha = 1 - e^{-\tau d}. \tag{2}$$

where τ is the optical density or extinction coefficient[1] of the slab along the ray segment, and d is the length of the ray segment in the slab, see Figure 1. For orthographic viewing, d is the width of the slab. We will confine our discussion to the orthographic case. The perspective case is more

[1]The optical density or extinction coeffi cient τ is considered to be a physical property of the volume that varies with the scalar fi eld. The values of τ are inferred from the initial opacity settings selected by the user for the scalar fi eld values. For the model used here, τ is considered to vary linearly along a ray segment through a slab, as does the scalar fi eld. Therefore, τ in Equation 2 is the average value of τ along the ray segment.

complicated and is the topic of a paper in progress by our group. As explained below, Equation 2 is only used to calculate opacity values when the number of slices (slabs) is changed. Initially, the user sets the colors and opacities (α's) for the transfer function directly, empirically, by adjusting them to give the desired image. However, when the number of slices (slabs) is changed, the slab width d changes and so according to Equation 2, the opacity must be modified. To do this, we do the following for each of the original opacities α_{orig} in the transfer function. We first calculate the optical density τ corresponding to α_{orig} using Equation 2, giving:

$$\tau = -ln(1 - \alpha_{orig})/d_{orig}$$

where d_{orig} is the original slab width. Then using that value of τ in Equation 2, we get the new opacity α_{new} corresponding to the new slab width d_{new}:

$$\alpha_{new} = 1 - e^{(d_{new}/d_{orig})ln(1-\alpha_{orig})}.$$

Let s_{orig} be the number of slices (slabs) for which the user initially set the opacity, and s_{new} be the new number of slices (slabs), then $d_{new}/d_{orig} = s_{orig}/s_{new}$. Therefore, each time the number of slices is changed, the new opacity value is given by:

$$\alpha_{new} = 1 - e^{(s_{orig}/s_{new})ln(1-\alpha_{orig})}. \tag{3}$$

In most scientific visualization volume renderings, the majority of polygons which we refer to as *background polygons* have a very low opacity in order to create sufficient transparency to give a penetrating view of the object. This allows the field values of interest, which are mapped to higher opacity values, to stand out. It is important that the background polygons not be mapped to zero opacity as these background polygons create a diaphanous haze or fog-like appearance that provides a spatial context in which to properly interpret the structure of the field values of importance. See Figure 2. A problem arises however when the number of low-opacity polygons (or slices) per pixel is large, a situation commonly encountered in large data sets or as might be required to resolve high frequency spatial structures in data sets.

For cell projection volume rendering, the number of polygons is determined by the number of cells in the data set. If the number of cells is large, the opacity required for the low-opacity polygons may be smaller than the input threshold. For slice-based rendering, or texture-based volume rendering, the number of slices is variable and may increase to a point where the opacity as calculated by Equation 3 for the low-opacity portion of the slices may be reduced below the opacity input threshold. For example, in the case of slice-based volume rendering, consider original alpha values of 1 and 3, which are reduced so the new values become 1.5 and 0.5; converting them to allowable input values (integers) will result in values of either 2 and 1, or 1 and 0. Neither of these sets of values results in semitransparent images that are comparable with the original image. We refer to the problem described in this paragraph as the *low-opacity fog problem*. What we need is some way to preserve the precision of the new low-opacity values.

One approach to this problem, proposed by Kniss et al in [10], is to scale up input alpha values while rendering, and then scale down the finished image. This is a good method, however, it requires that the scaled-up values and their composited values remain in range — which is not always possible. In our paper, we present a method that involves dithering small opacity values, thus effectively extending the range of allowable input values for opacity without the limitation of the scaling method.

Briefly, our method works by dithering low opacity values, either randomly or using a regular pattern, over a certain number of slices or polygons. So for example, if we desire to dither an opacity of 0.75 over say 4 slices, then we set the opacity to 1 for three slices and to 0 for one slice. Thus, over 4 slices, an effective opacity of 0.75 will be used. Our method is explained in detail in Section 3.

Figure 3 shows six volume rendered images of a scientific data set that illustrate the low-opacity fog problem. The images were generated using a 32 bit frame buffer, with 8 bits per channel. The left images were generated using 128 slices, the middle pair using 275 slices, and right ones using 600 slices. The bottom set of images was created without alpha dithering, the top set with alpha dithering using a regular pattern over a period of 16 slices. The image with 128 slices was chosen as the reference image and the initial alpha values set to give it the desired transparency.

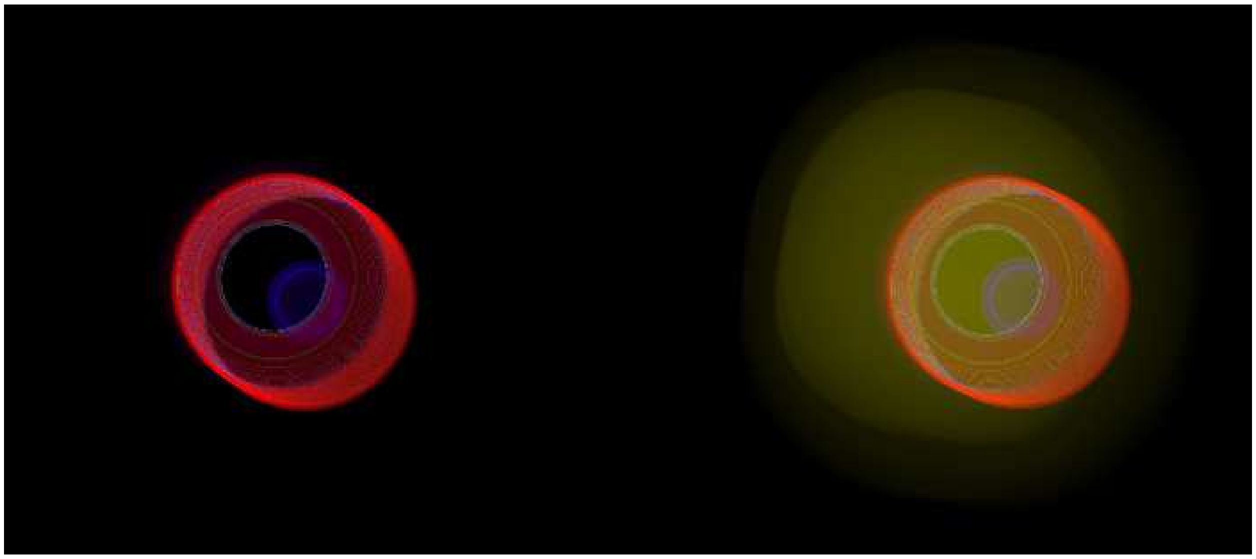

Figure 2: Two volume rendered images generated using a 32 bit frame buffer, with 8 bits per channel. The right one has a yellow haze formed by compositing polygons with low opacity values. The left image has no haze because the opacity of the low-opacity polygons is below the input threshold (1/255) for 8 bit alpha values. The haze is important because it gives the image a spatial structure in which to interpret the fi eld values with higher opacity values.

2.2.1 Previous Work

Wittenbrink et al [26] describe an opacity-weighted color interpolation scheme that exactly reproduces material based interpolation results for voxel-based volume rendering when performing material classification, e.g. bone versus muscle. However, their method does not deal with the problems related to low opacity values when using fixed precision graphics hardware, nor is it rel-

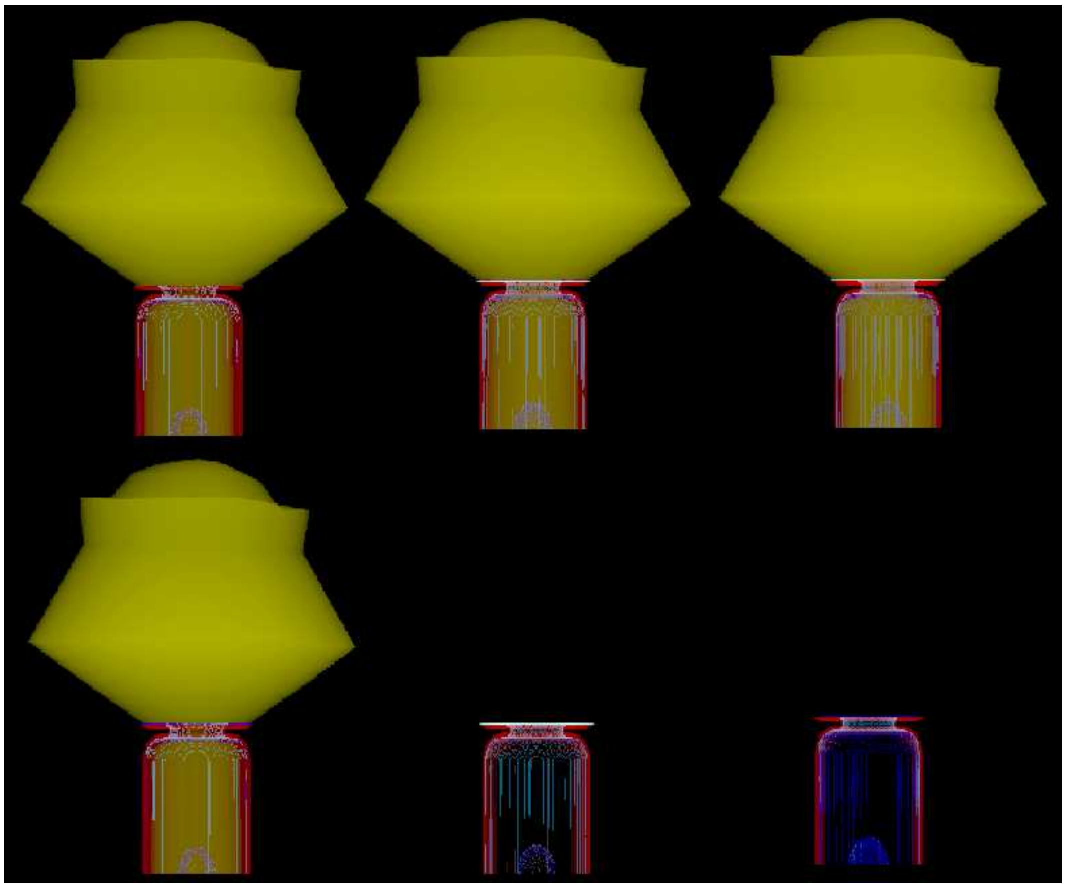

Figure 3: Volume rendered images of a scientifi c data set generated with, from left to right, 128 slices, 275 slices, and 600 slices. The images on the bottom were generated without alpha dithering, those on the top with alpha dithering. As the number of slices increases, the alpha values decrease to the point where small alpha values vanish unless dithering is used.

evant to volumes with a continuous medium. Engel, Kraus and Ertl [9] describe a pre-integrated volume rendering method applicable to cell projection techniques. Their method achieves very high image quality, however, the blending equation, Equation 1, is still implemented in hardware by the frame buffer. Therefore, the low-opacity fog problem described above is still relevant, and their pre-integrated method can be improved by adding our dithering technique. Lacroute [11] describes a slicing-based volume renderer however, it does not utilize graphics hardware. We have found no references in the literature either to what we refer to as the low-opacity fog problem, nor to any solutions to it.

2.3 The Bricking Artifact Problem

Another problem caused by limited precision compositing can occur when bricking is used. *Bricking* means that the overall volume is partitioned, for example by an oct-tree decomposition, into bricks. The bricks are then sliced and rendered in parallel, using multiple GPU's. This process results in several semitransparent images, one from each brick. These images are then gathered and composited two at a time, in back to front order, in software, to form the *final image*. A slice may be a texture [6, 10, 13, 19, 25], or consist of a set of polygons [24, 28]. When bricking is used, it is necessary to accumulate opacity as well as color during rendering. The accumulated opacity is needed so the image of a brick can be composited with the image of another brick. This is why we call these *semitransparent images* — they have a color and an opacity at each pixel. Most graphics hardware allows opacity values to be accumulated in the frame buffer in addition to color. We assume the alpha channel has the same number of bits as each of the color channels. The bricking artifact problem, which we will explain in detail, arises due to a combination of two factors: the limited precision of the accumulated RGBA values in the frame buffer, and the formation of a region of overlap created when the bricks are rotated. See Figure 4 which shows two rotated bricks with a region of overlap. In this region, between pixels $P1$ and $P3$, the number of slices accumulated in each brick varies. This can cause a noticeable artifact when the images of the two bricks are composited. An orthographic projection is assumed in Figure 4; however, this type of artifact can occur with perspective projection also. We have not seen any reference to this problem, nor any solutions to it, in the literature. However, we have had conversations with other researchers using graphics clusters who have experienced this problem. We now explain how the artifacts in the final image arise.

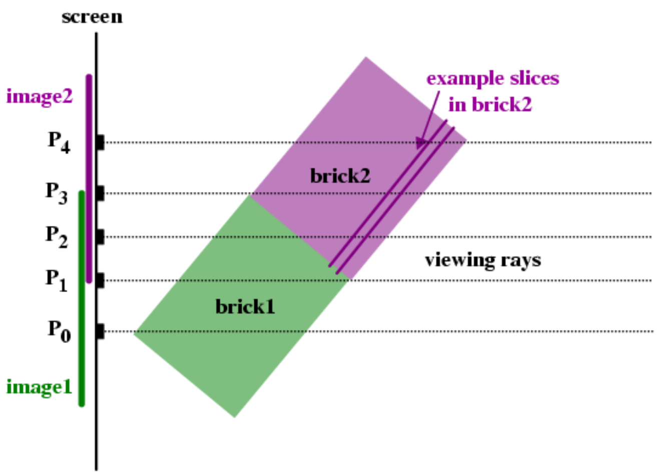

Figure 4: Two bricks sliced from upper right to lower left. Each brick is rendered separately to form images 1 and 2, and then the two images are composited. The rotated bricks form a region of overlap in the two images across which the number of slices composited by each individual brick varies. In this particular example, pixel P_2 of image 1 composites half the slices in brick 1. That same pixel location in image 2 composites half the slices in brick 2. Pixels P_0 and P_1 composite all their slices from brick 1.

Equation 1 is used to composite color during rendering, and to composite the color values of two semitransparent images. (In practice, a subtle modification is required for the latter which we explain later). The following equation is used to composite opacity:

$$CompositedOpacity = \alpha_{front} + (1 - \alpha_{front}) \times \alpha_{back} \tag{4}$$

When accumulating two images, α_{back} and α_{front} are the pixel opacities of the images of the bricks that are farthest and nearest from the viewer, respectively. When used for rendering, α_{back} is the existing opacity at a pixel, and α_{front} is the incoming α. The equation for compositing color is intuitive. However the equation for opacity is easier to understand if we consider *transmittance* T, defined as $(1 - \alpha)$. When a polygon with transmittance T_1 is blended with another polygon whose transmittance is T_2, the combined transmittance T' is:

$$T' = T_1 \times T_2. \tag{5}$$

This agrees with our intuition: if a polygon transmits $\frac{1}{2}$ of the light from behind it, and we composite it with another polygon whose transmittance is also 50%, the combined transmittance will be $\frac{1}{4}$. Translating Equation 5 into terms of α, we get the blending equation for opacity shown above:

$$(1-\alpha') = (1-\alpha_1)\times(1-\alpha_2) = 1-\alpha_2-\alpha_1+\alpha_1\times\alpha_2,$$
so
$$\alpha' = \alpha_2+\alpha_1-\alpha_1\times\alpha_2 = \alpha_2+(1-\alpha_2)\times\alpha_1.$$

The OpenGL function *glBlendFunc()*, that selects how the graphics hardware blends the color and opacity of an incoming fragment with the color and opacity of a pixel stored in the frame buffer, has an option that implements Equation 1, and this option is what is normally used to achieve transparency, when opacity is not accumulated. However, *glBlendFunc()* does not have an option that implements both Equations 1 and 4 at the same time, unless the incoming color is first premultiplied by the incoming opacity in software. The *glBlendFunc()*[2] that correctly composites opacity, implements the following equations:

$$NewPixelColor = c + (1-\alpha)\times OldColor. \tag{6}$$

$$NewPixelOpacity = \alpha + (1-\alpha)\times OldPixelOpacity \tag{7}$$

Where α and c are the opacity and color, respectively, of the incoming fragment. Hence we see the need for the premultiplication of color by opacity in order to make Equation 6 agree with Equation 1. The premultiplication does not contribute to the problem under discussion, but we need to be aware of it. We now return to the subtle difference, referred to above, in applying Equation 1 to blend two images, as opposed to perform compositing during rendering. In the process of creating a semitransparent image of a brick, every incoming color value has been multiplied by the incoming opacity. Therefore, when a semitransparent image is blended with another such image, it is not necessary to multiply the color of the front image by the opacity of the front image. Hence the blending equation for color for two semitransparent images is:

$$CompositedColor = c_{front} + (1-\alpha_{front})\times c_{back} \tag{8}$$

2.3.1 An Example

Recall from Section 2.1 that by *n bit compositing*, we mean that the opacity and color values are stored, and specified as input, as integers, using n bits, and that the blending equation, Equation 1, is evaluated using floating point. The goal of most of our examples in this paper is to compare the results of 8 bit compositing (with or without dithering) with ideal or exact compositing. For the purpose of contemporary graphics, we consider 32 bit compositing to be exact. Please do not confuse 32 bit compositing with the use of a 32 bit frame buffer. Think of 32 bit compositing as ideal or exact compositing.

For this example, we are going to composite a single brick with 200 slices (or polygons) each of which has a uniform color of 255.0 and opacity of 1.0 (both out of a total of 255.0). Equation 1 will be used to composite the colors of the slices. Figure 5 shows the composited color at a pixel in the frame buffer as a function of the number of polygons (or slices) composited. Two cases are shown: a) 32 bit compositing — the red line, and b) 8 bit compositing — the blue line. In this

[2]i.e. when $sfactor = GL\_ONE$ and $dfactor = GL\_ONE\_MINUS\_SRC\_ALPHA$. See [15] for more details. There is an extension to OpenGL, called *EXT_blend_func_separate* that allows for separate opacity and color blend function specifi cation, however is not widely supported at this time.

figure we see that after compositing 100 polygons using 8 bits, the pixel value is 100, whereas the use of 32 bit compositing gives a value of 82.9. After accumulating all 200 polygons using 8 bits, the composited color is 128, whereas using 32 bits, the color is 138.8. The green line shows the error due to 8 bit compositing. Note that when using 8 bits, the composited color reaches a *plateau* at 128 slices. From 0 to 128 slices, we say the composited color is in its *linear ramp stage*. In Section 3, we explain in detail why a plateau occurs when using 8 bit compositing.

Our goal in this work is to dither the input color and/or opacity in such a way that the composited color tracks the 32 bit (red) curve as closely as possible. (Note that for this choice of polygon color and opacity (255 and 1), the curve of accumulated color and the curve of accumulated opacity are identical, since the color is premultiplied by opacity, i.e. $color \times \alpha = 255 \times (1/255) = 1 = \alpha$. Therefore we have not shown a separate plot of accumulated opacity. This is true for any value of α when $color = 255$.)

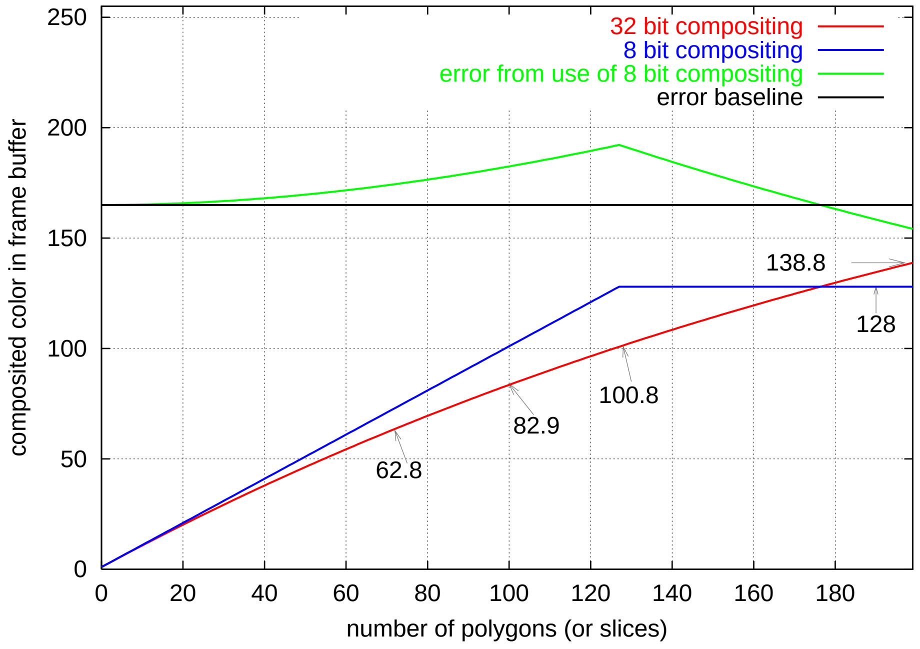

Figure 5: The red and blue curves show the composited color at a pixel in the frame buffer as the number of polygons composited increases. Each polygon has a uniform color of 255 and opacity of 1. The red curve is for 32 bit compositing (our high accuracy standard), the blue for 8 bit compositing. The green curve shows the magnitude of the difference between the two.

Now consider the case where two bricks, each with 200 slices, are rotated as shown in Figure 4. Each brick is rendered separately to a different frame buffer resulting in images 1 and 2. Pixel P_2 in image 1 and pixel P_2 in image 2 will each have accumulated 100 slices and so have a color and opacity of 100 when 8 bit compositing is used — see Figure 5. Now we composite the two images using Equation 8. When applied to pixel P_2 of each image we get:

$$CompositedColor(P_2) = 100 + (1 - 100/255) \times 100 = 161.$$

However the nearby pixel P_1 (see Figure 4) will have accumulated all 200 slices from brick 1, so its value in the final image is 128 (see Figure 5). So while both pixels P_1 and P_2 have composited the same number of identical slices, their final values will be different: 128 vs. 161. Thus an artifact is introduced into the image. We refer to this as the *bricking artifact problem*. This problem can occur when numerous low-opacity polygons or texels project onto a pixel, it doesn't require the entire brick to have a low opacity. Figure 6 shows an example of an artifact due to this problem in an actual image created by volume rendering two bricks positioned similarly to those shown in Figure 4, but with less rotation.

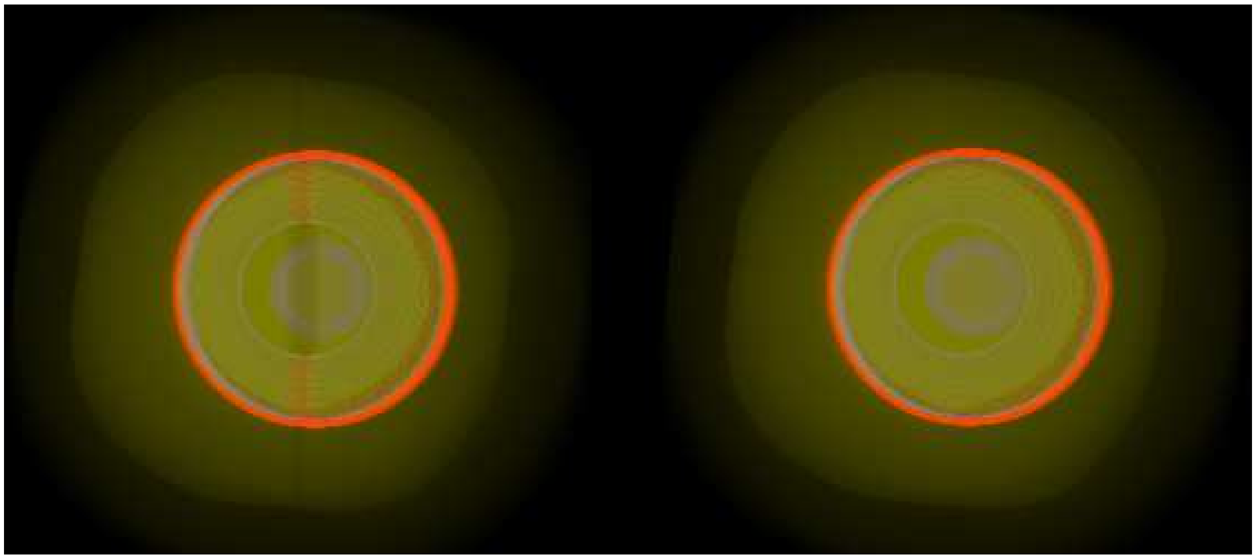

Figure 6: The left image shows an artifact due to the bricking artifact problem. The two bricks are positioned similarly to those shown in Figure 4, but with less rotation. Each brick has 400 slices, and each slice is composed of polygons, most of which have an opacity of 1. The right-hand image shows the same data and viewing parameters, but the data is not bricked.

Figure 7 shows the composited pixel values across a scan line spanning the overlap region (i.e. between P_1 and P_3 in Figure 4) of two bricks oriented as shown in Figure 4. The distance across the scan line is parameterized by the number of slices X in brick 1, starting from P_1. The slices in each brick have a color of 255 and opacity of 1. Each brick has 200 slices. The far left of the plot, $X = 0$, corresponds to pixel P_3 in Figure 4 where brick 1 has 0 slices and brick 2 has 200 slices. The far right of the plot corresponds to P_1 where brick 1 has 200 slices and brick 2 has 0 slices. Pixel P_2 occurs in the middle of the plot where each brick contributes 100 composited slices to the final image. The red line shows the results when 32 bit compositing is used to composite color and opacity in each brick, the blue line when 8 bit is used. The composition of the two images to form the final image is done in software using Equations 4 and 8 with floating point arithmetic. The correct result as shown by the red line is flat across the image.[3] However, when 8 bit compositing

[3]Referring to Figure 5, we see that the 32 bit composited value at 72 slices is 62.8, and at 128 slices is 100.8. Using these values in Equation 8, we get:

$$CompositedColor = 62.8 + (100.8 \times (1 - 62.8/255) = 138.8.$$

Thus the compositing formula gives the correct answer when 32 bit compositing is used, regardless of brick size.

is used, the intensity varies across the image resulting in an artifact — this is the bricking artifact problem.

Given two bricks each of which has the same uniform color and opacity throughout. Let C_1 be the composited color in the image of one of the bricks which accumulates all the slices in that brick, e.g. the composited color between P_0 and P_1 in Figure 4; and let C_X be the composited color at any pixel X on a scan line in the final composited image of the two bricks. We define the *artifact amplitude percentage* (AAP) *error* to be $AAP(X) = ((C_X - C_1)/C_1)/100$, the percentage change in the image intensity due to the bricking artifact problem. The dashed line at the bottom of Figure 7 shows this error metric, which has a maximum value of of 28% in this case. We can better understand the reason for the shape of the blue 8 bit curve by looking at Figure 8 which superimposes the composited color of brick two on top of that for brick one. Between $X = 0$ and $X = 72$, brick one is in its linear ramp stage and brick two is at its plateau, therefore the composited color of the two bricks shown in Figure 7 is increasing. From $X = 73$ to $X = 126$, both bricks are in their linear ramp stages, therefore the combined value of the two bricks shown in Figure 7 is nearly flat. The reason the curve in this region is not completely flat is due to the error from 8 bit compositing as explained in the caption to Figure 8. If the color and opacity had been accurately composited in each brick, the pixel values would be constant (flat) across the scan line since the color and alpha values are the same in each brick. Figure 9 shows the bricking artifact problem in the context of a scan line from P_0 to P_4 across the image formed by compositing images 1 and 2 in Figure 4, but rotated much less so as to correspond to the configuration of the two bricks used to generate Figure 6. Note how the artifact shown in Figure 9 corresponds to the artifact in Figure 6.

The bricking artifact problem becomes less significant as the opacity of the polygons increases. This is because the 8 bit curves more closely track the accurate 32 bit composited values as the input opacity increases. See Figures 10 and 11, which compare 8 bit and 32 bit compositing for alpha values of 5 and 10 respectively. As can be seen in Figures 12 and 13, the maximum value of the AAP error has fallen significantly (from 28% for an opacity of 1, to 8.6% for an opacity of 5, and to 4.5% for an opacity of 10.) In general, the lower the plateau value reached by the composited colors in each brick, the worse the bricking artifact. As we will see, dithering reduces the AAP error.

When the number of slices is increased, the maximum absolute value of the AAP error increases when compositing low opacities ($\alpha < 3$). Figure 14 shows the bricking artifact problem when 2 bricks each with 1000 slices are composited where the background color is 255 and opacity is 1. Figure 15 shows the superimposed composited values for the two bricks. The AAP error has increased to 49%, whereas for 200 slices it was 28%. The AAP error changes only slightly with increased slice density for an input opacity of 5, and not at all for an input opacity of 10.

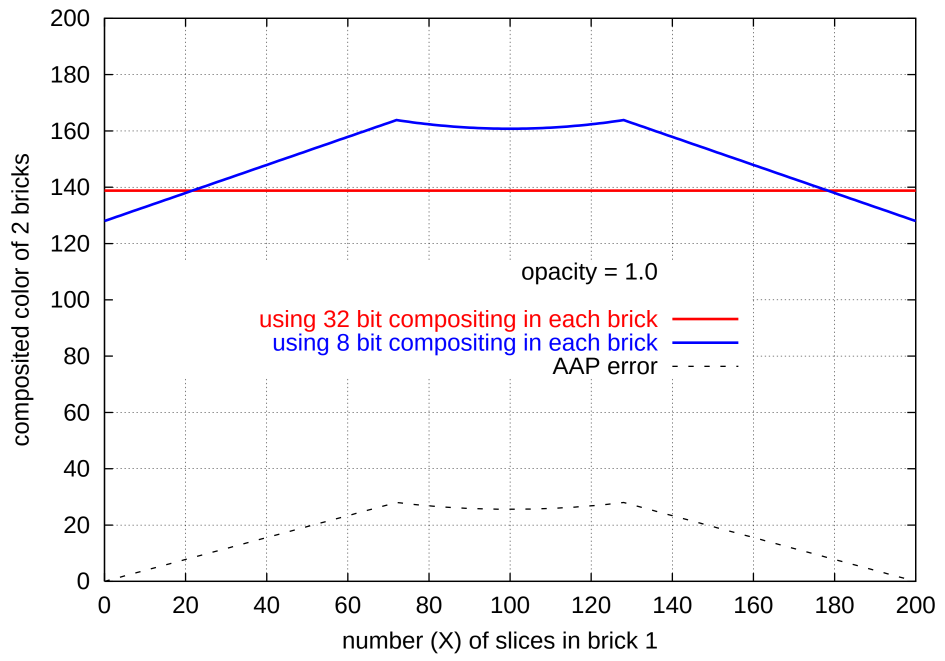

Figure 7: The red and blue curves show the pixel values across a scan line of the composited final image of two bricks oriented as shown in Figure 4. The scan line spans the overlap region (i.e. between P_3 and P_1. The distance across the scan line is parameterized by the number of slices X in brick 1, starting from P_3. The slices in each brick have a color of 255 and opacity of 1. Each brick has 200 slices. Pixel P_1 corresponds to the far right of the plot. Pixel P_2 occurs in the middle of the plot. There are two different compositing operations going on here and we have to be careful not to confuse them. First, each brick is composited in hardware to form a volume rendered image of each brick. Second, these two images are composited in software using maximum precision to form the final image. The plotted red and blue curves show the pixel values in the final image. The red line shows the results when 32 bit hardware compositing is used to composite color and opacity in each brick, the blue line when 8 bit hardware compositing is used. The dashed line shows the AAP error as defined in the text —note this error is not the difference between the red and the blue curves. The reason the blue curve dips down in the middle and is not flat can be understood by looking at Figure 8.

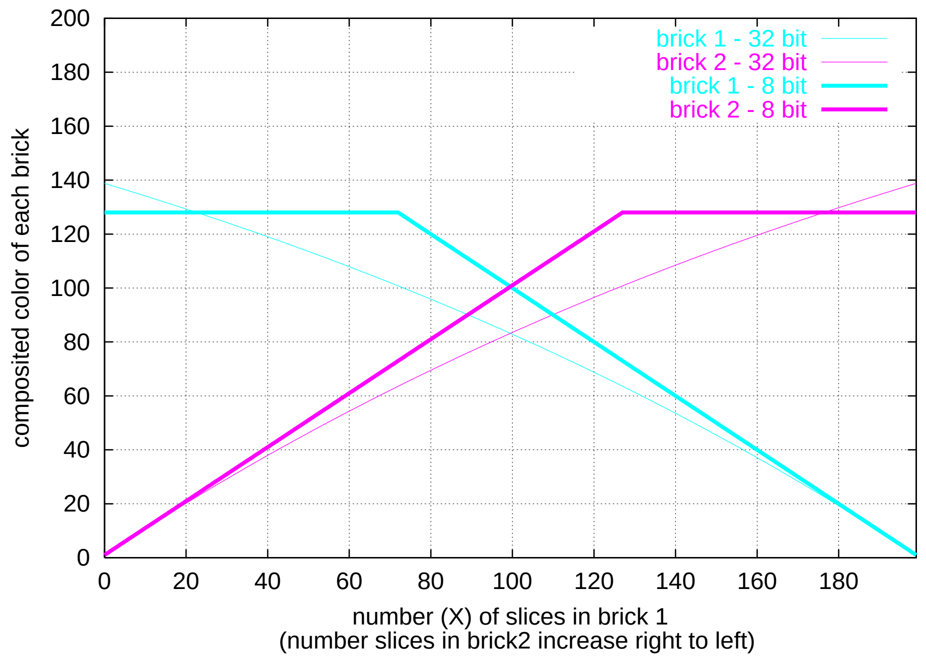

Figure 8: The heavy lines show the composited color, using 8 bit compositing, for each brick oriented as shown in Figure 4. The number of slices for brick 1 increases from left to right, whereas the number of slices for brick 2 increases from right to left. The lighter lines show the results if 32 bit compositing had been used. The slices in each brick have a uniform color of 255 and opacity of 1. There are 200 slices in each brick. In order for the the composited color of the two bricks, shown by the blue line in Figure 7, to be flat between $X = 73$ and $X = 126$, the composited color of each brick would have to follow the 32 bit lighter curves. Since this is not the case, the blue line in Figure 7 is not flat.

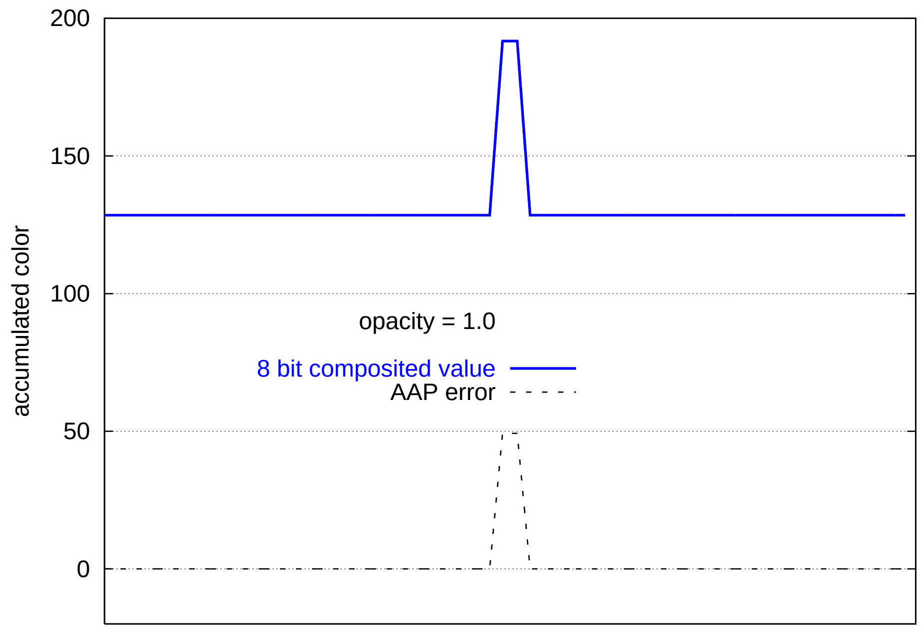

Figure 9: The bricking artifact problem in the context of a scan line from P_0 to P_4 across the image formed by compositing images 1 and 2 in Figure 4. Note how this corresponds to the artifact in Figure 6.

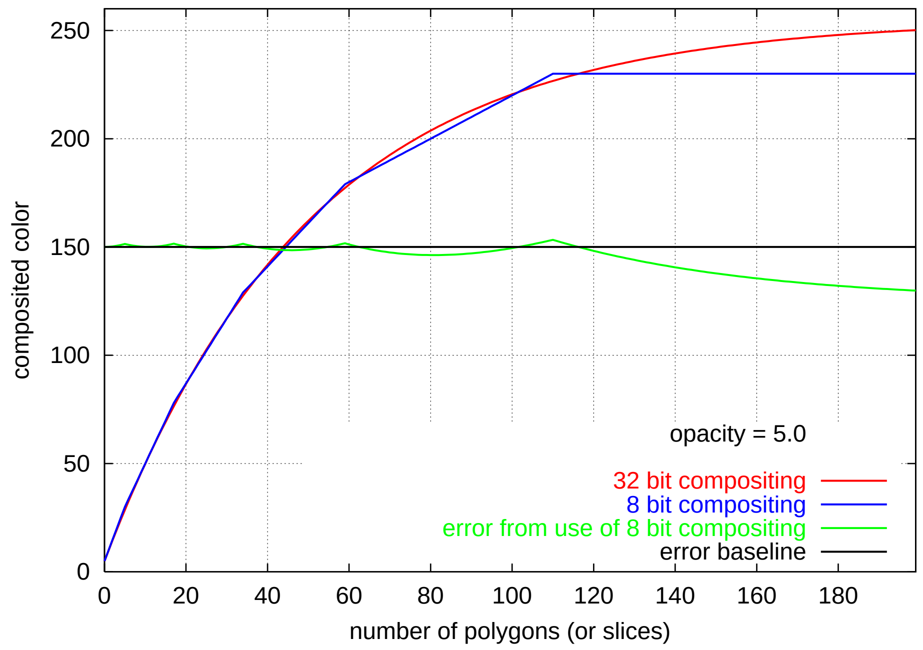

Figure 10: Accumulated pixel values for compositing polygons with opacity 5 and color 255.

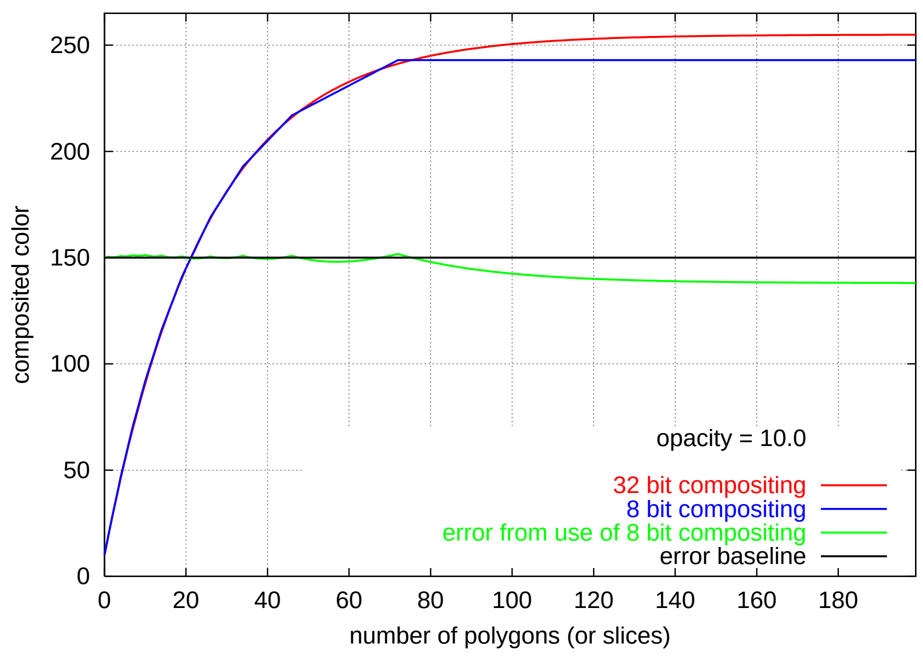

Figure 11: Accumulated pixel values for compositing polygons with opacity 10 and color 255.

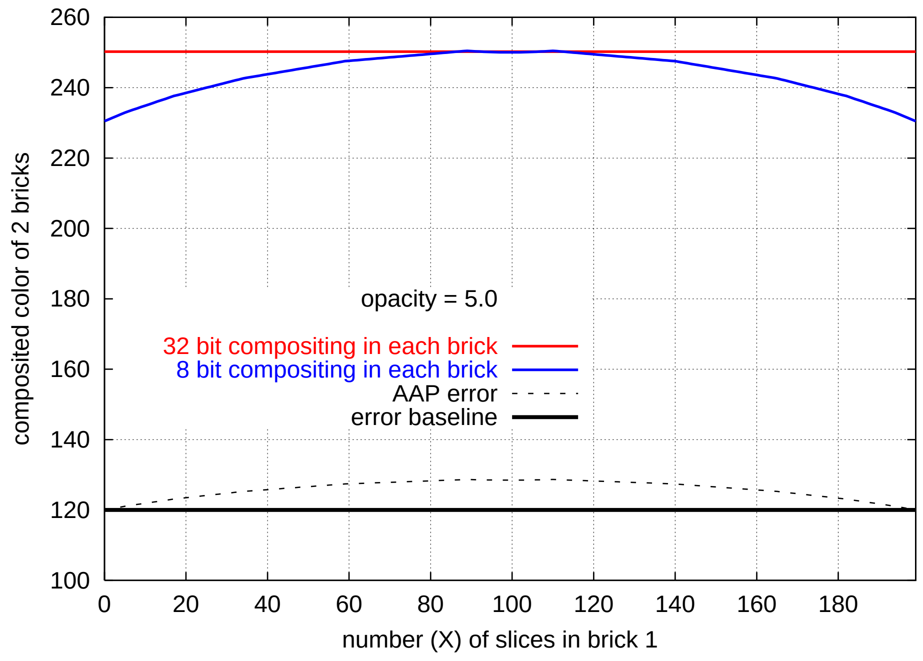

Figure 12: Bricking artifact problem for slices with opacity of 5 and color of 255. The AAP error, the dashed line at the bottom, has decreased signifi cantly, so its maximum is now less than 9%. Compare that with Figure 7 where the slice opacity was 1 and the maximum error was 28%.

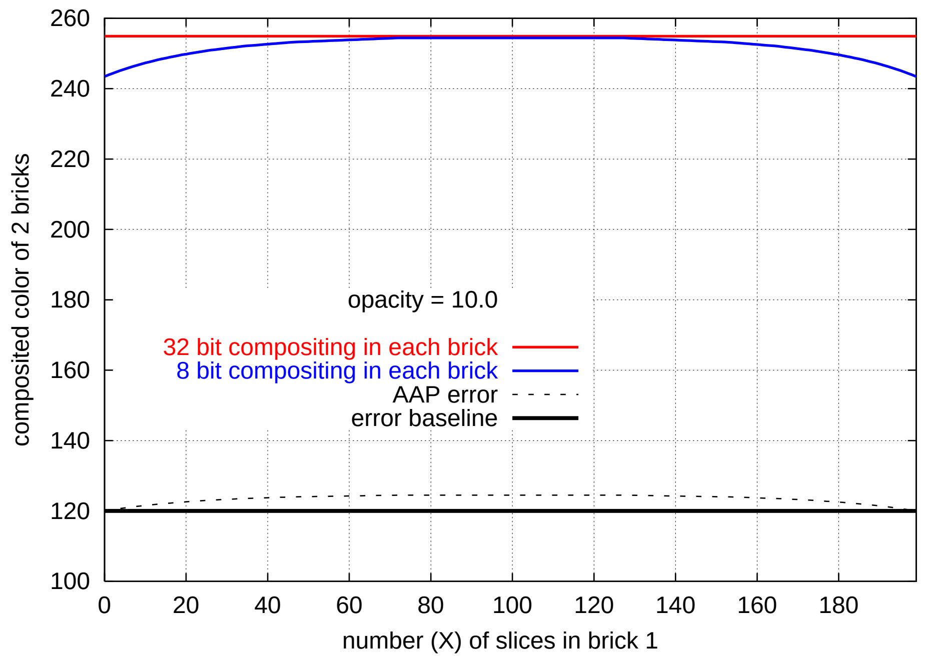

Figure 13: Bricking artifact problem for slices with opacity of 10 and color of 255. The AAP error, the dashed line at the bottom, has now decreased so the maximum is less than 5%.

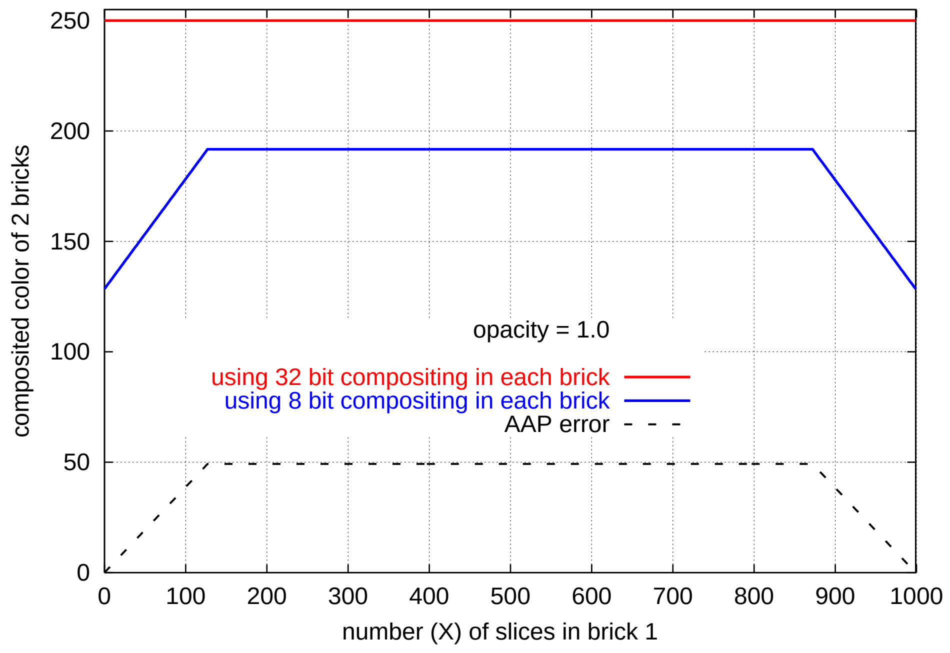

Figure 14: Bricking artifact problem for slices with opacity of 1 and color of 255, for 1000 slices. Note that the AAP error has increased to 49%, whereas for 200 slices it was 28%.

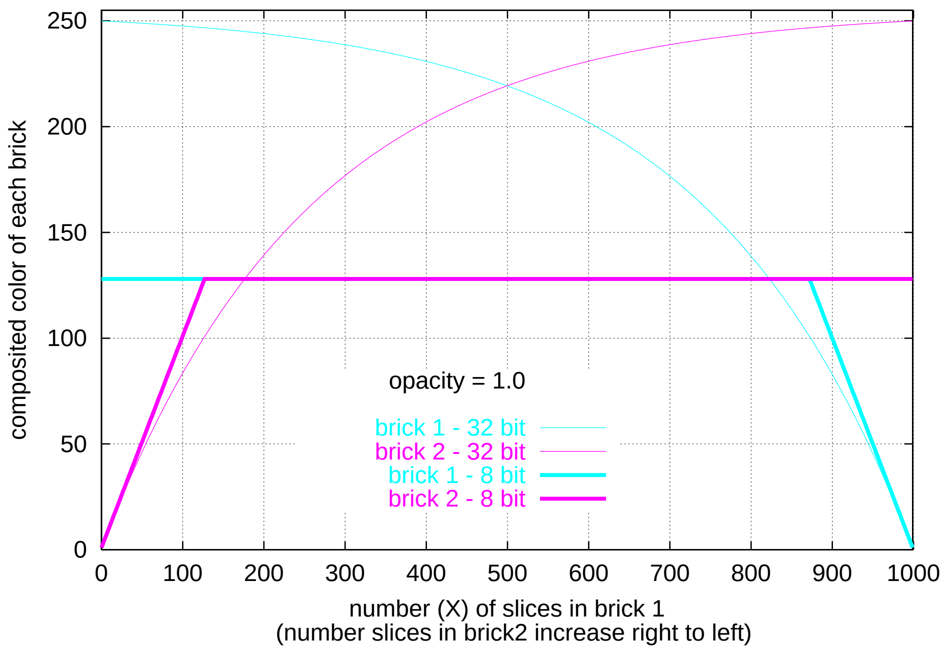

Figure 15: The heavy lines show the composited color, using 8 bit compositing, for each brick oriented as shown in Figure 4. The lighter lines show the results if 32 bit compositing had been used. The slices have a uniform color of 255 and opacity of 1. There are 1000 slices in each brick.

2.3.2 Previous Work

We have found no references in the literature either to what we refer to as the bricking artifact problem, nor to any solutions to it. Lacroute [12] describes a parallel algorithm for slice-based volume rendering, however it does not use graphics hardware. Porter and Duff [18] describe an algebra of compositing and related arithmetic including premultiplication. Blinn [1, 2, 3] discusses a number of interesting compositing applications and techniques using software. In [4], Blinn discusses 8 bit arithmetic, and in [5] he describes how to use the Intel Pentium MMX to do 8 bit arithmetic. However, these reference do not address the problems we are dealing with. Of course the basic concept of dithering [7, 8] is well known, and is applied extensively in the areas of computer graphics, printing and image processing to extend the available range of colors or grey scale values. However, it is primarily used spatially, i.e. from pixel to pixel across a scan line or from one scan line to another. A technique called *screen door transparency*, that uses pixel masks [21] or polygon stippling [15], is used to simulate transparency when the graphics hardware does not have an alpha channel. However, we are not aware on any published application of dithering that deals with compositing low opacity values.

3 THE ALPHA DITHERING TECHNIQUE

In this section, we discuss a software technique for dithering low alpha values that helps overcome the low-opacity fog problem and the bricking artifact problem. This results in improved volume rendering results using progressive refinement, and helps reduce or eliminate artifacts caused by the bricking artifact problem.

Recall from Section 2, that the problem is this: We need to be able to input small noninteger values of opacity, e.g. 0.50, 1.37, etc., to avoid the low-opacity fog problem, however 8 bit graphics hardware only allows integer opacity values to be input. So to accomplish this, we dither small alpha values over a set of P slices. Our basic dithering method is that for every set of P slices, we render $P \times (1 - (\alpha - \lfloor\alpha\rfloor))$ slices with an integer alpha value of $\lfloor\alpha\rfloor$ and $P \times (\alpha - \lfloor\alpha\rfloor)$ slices with an integer alpha value of $\lfloor\alpha\rfloor + 1$. We call P the *period*. So for example if we choose a period of 16 slices, and the alpha to be dithered is 2.25, then for 12 slices we input an alpha of 2, and for 4 slices we input an alpha of 3. The alphas may be alternated in a regular pattern such as $\{3, 2, 2, 2, 3, 2, 2, 2, 3, 2, 2, 2, 3, 2, 2, 2\}$ which we refer to as an *ordered dither*, or randomly. If alpha had been 2.3, then there is no dither pattern with a period of 16 that will exactly simulate it, so we round down to the closest dither pattern, in this case the pattern for alpha of 2.25. (For maximum dithering accuracy, we attempt to set undithered alphas in the transfer function table to rational numbers, $\alpha = n/P$ where n is an integer, since in the dithering process, alpha will get rounded to such a value in any case.) We refer to the low integer in the above alternation, $\lfloor\alpha\rfloor$, as the *base*, and the difference between the higher integer and the base, as the *bump*. So in this example, the $base = 2$ and the $bump = 1$. We call an instance of the dithered value which is equal to $base + bump$ a *hit*, so over one period P there are on average $P \times (\alpha - \lfloor\alpha\rfloor)$ hits.

Given an opacity value val that we wish to dither, we calculate the number of hits $numHits$ required as:

$$numHits(val) = round(P \times ((val - base)/bump)) \tag{9}$$

Later in this section we will discuss the use of bumps greater than one. The above formula is general enough to deal with that possibility. Equation 9 is used by both the random and the ordered dithering routines to determine whether to output a hit. The random dithering routine uses the result of Equation 9 in the following logic:

```
if ((random()% P) >= numHits) return base;
else return base+bump;
```

Note that `(random() % P)` returns a value in the range $[0, P - 1]$. For ordered dithering, we basically set up a 2D table indexed by opacity val and slice number which returns the dithered opacity in accordance with Equation 9. Then for each slice number, the pattern table for val is accessed to see whether to output $base$ or $base + bump$. Further details on the implementation of ordered dithering are given in Section 6. Dithering is not essential for large alpha values, therefore we set an alpha value, called the *dithering cutoff value,* above which we do not dither.

Due to the unusual nature of the results of compositing using only 8 bits (discussed below), we have found that one dithering method is not suitable for dealing with all low opacity values. However, with some creativity, the desired 32 bit behavior can be closely approximated. Note that we do not yet know how to implement all the details necessary for certain of the techniques we will introduce, such as exponential bumping and multiple slope modifiers. This is discussed further in Section 9.2. To study the problem, we assume the slices have constant color and opacity throughout.

Figure 16 shows the effect of random and ordered dithering on 8 bit compositing where each slice has a uniform opacity of 0.10 and color of 255. In this case $base = 0$, $bump = 1$, and the period is 32. Both random and ordered dithering give excellent results all the way out to 1000 slices when compared to 32 bit compositing. Output for undithered 8 bit compositing is not shown because it is zero. While it is possible to specify opacity as a floating point value with $glColor4f()$, when the frame buffer has only 8 bits per channel, such input values get rounded to integers in the range $[0, 255]$. So an opacity of 0.10 will get rounded to 0. (Recall again that opacities in this paper are specified as values in the range $[0.0, 255.0]$.) Figure 17 shows the results of compositing the images of two bricks, each of which was created using these same parameters. The AAP error is negative in this case since the initial and final values (i.e. at 0 and 1000 slices) of both dithered curves are above their values at most intermediate points. The maximum absolute value of the AAP error is 8.6%.

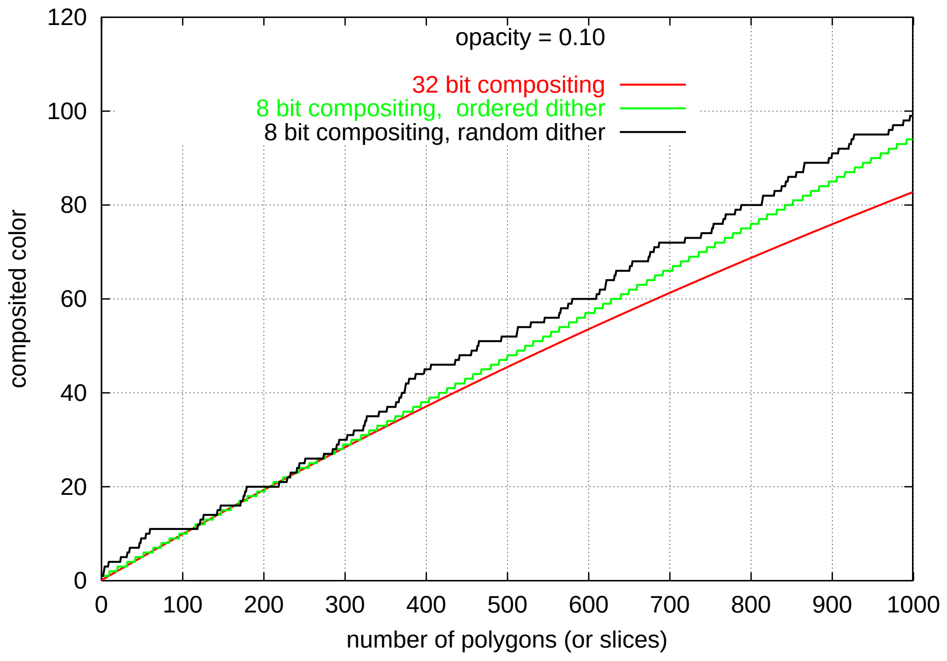

Figure 16: Accumulated pixel values when using random and ordered dithering for slices with opacity 0.10 and color 255 over 1000 slices. The period is 32, $base = 0$ and $bump = 1$.

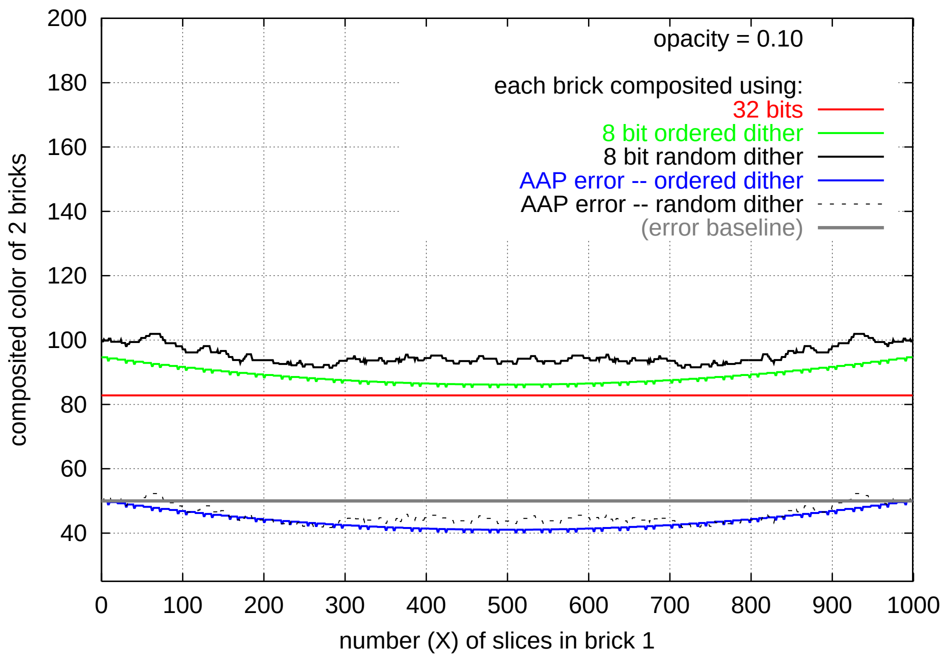

Figure 17: Bricking artifact problem for slices with opacity of 0.10 and color of 255, using random and ordered dithering, for 1000 slices. The period is 32, $base = 0$ and $bump = 1$. The AAP error shown at bottom indicates a maximum error of -8.6%.

Now compare the plot for $\alpha = 0.50$, Figure 18, with the previous plot for $\alpha = 0.10$, Figure 16. The undithered input opacity values of 0.5 are rounded by the hardware to 1, hence the blue curve in Figure 18 increases with a higher slope than the other three curves. Also note that the undithered as well as both the random and ordered dithered pixel values all reach a *plateau* at 128.

The reason for the plateau is as follows. When the blending equation (Equation 1) is used with an 8 bit buffer, certain pixel values occur that can not be increased unless the input opacity value is greater than some minimum value. For example, if the current pixel value is 243, then the input opacity must be at least 11 in order to increase the pixel value. These *pixel plateau values*, shown in Table 1, can be calculated as a function of alpha ($\alpha \geq 0.5$), for $IncomingColor = 255$, from:

$$plateau(\alpha) = \lceil 255.0 - (127.5/round(\alpha)) \rceil. \qquad (10)$$

For example, assume a current pixel value of 230, an input color of 255 and an input opacity of 5. Plugging these values into Equation 1, using our 8 bit evaluation convention, we get:

$$NewColor = round((5/255 \times 255) + (1 - 5/255) \times 230) = 230.$$

Thus an input opacity of 5 does not increase a pixel whose current value is 230. For an incoming color less than 255, even greater input opacity values are required to overcome the plateau. Given an 8 bit pixel plateau value pv, we call the minimum input opacity value necessary to overcome that plateau $AlphaMin(pv)$.

As the opacity increases, the plateaus increase the bricking artifact problem. See Figure 19 which shows this problem for an input opacity of 0.50 and a color of 255 for 1000 slices. The max AAP error reaches 49% even with dithering. Figures 20 and 21 shows the results for input opacities of 0.75 and 1.5, where the max AAP error is 49% and 24% respectively.

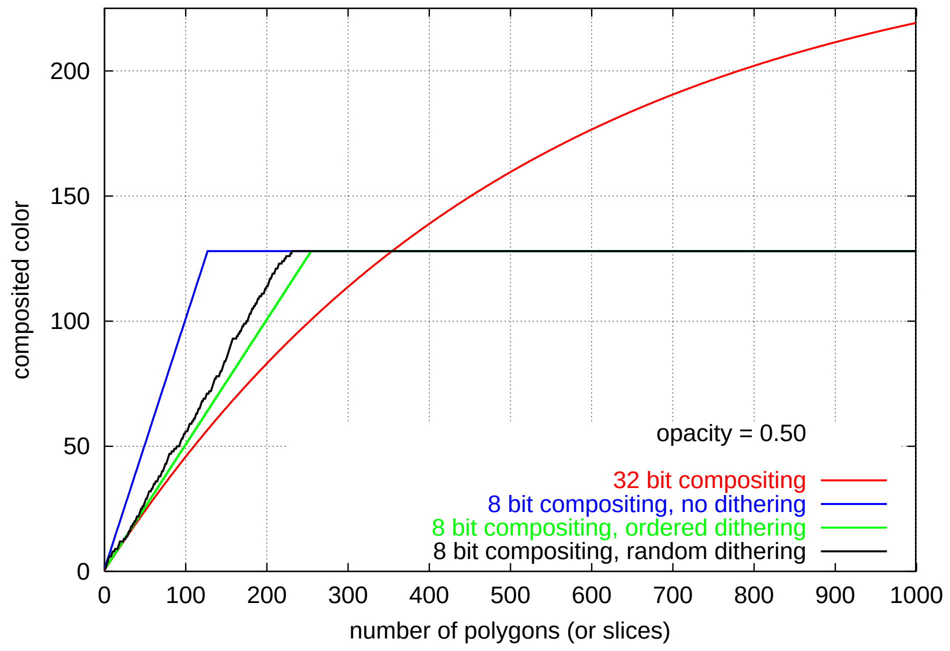

Figure 18: Accumulated pixel values when using random and ordered dithering for slices with opacity 0.50 and color 255 over 1000 slices. The dithering period is 32.

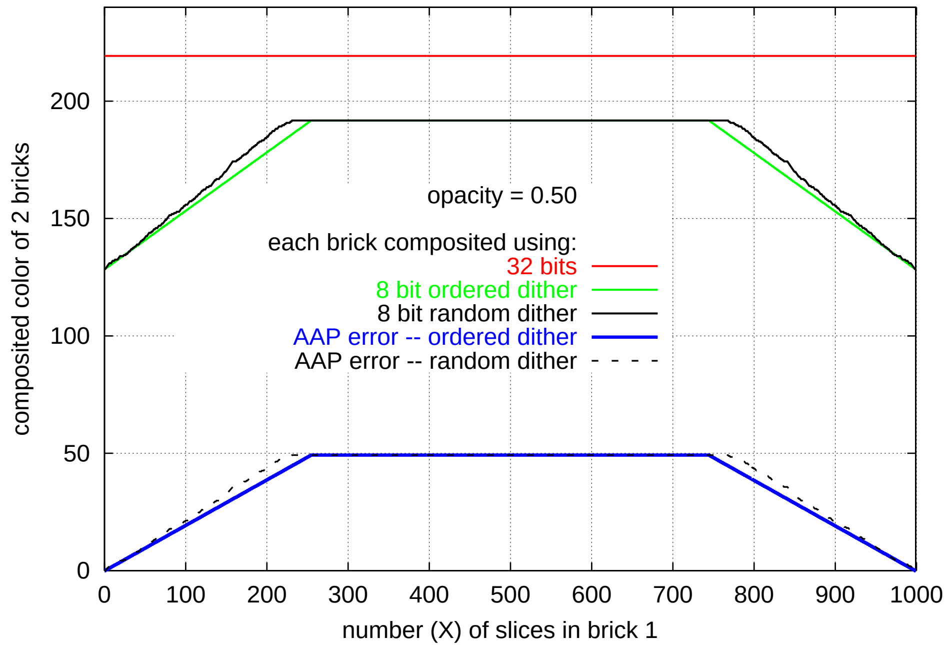

Figure 19: Bricking artifact problem for slices with opacity of 0.50 and color of 255, using random and ordered dithering, for 1000 slices. The period is 32, $base = 0$ and $bump = 1$. The AAP error reaches a maximum error of 49%.

Pixel Plateau Value	Minimum Input Opacity To Overcome Plateau
128	2
192	3
213	4
224	5
230	6
234	7
237	8
240	9
241	10
243	11
244	12
245	13
246	15
247	16
248	19
249	22
250	26
251	32
252	43
253	64
254	128

Table 1: This table shows the minimum input opacity values required to climb above the various 8 bit pixel plateau values. The plateau values were calculated using Equation 10. The minimum input values were determined using Equation 1 with an *IncomingColor* value of 255, setting *ExistingColor* to a pixel plateau value. Note that for pixel values over 230-240, rather large input opacity values are required to change the pixel value. For *IncomingColor* values less than 255, even greater input opacity values are required to overcome the plateau.

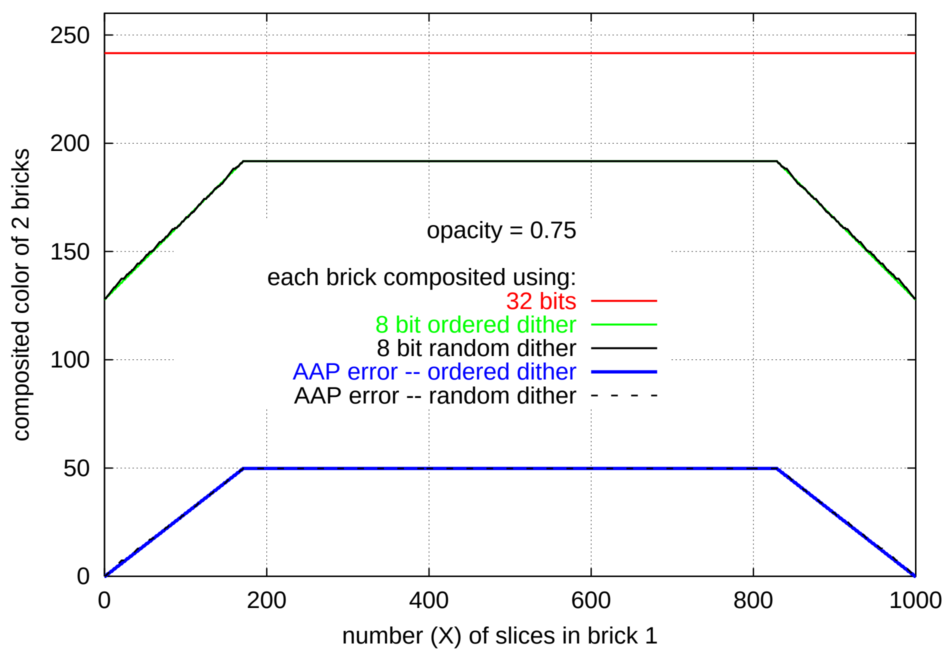

Figure 20: Bricking artifact problem for slices with opacity of 0.75 and color of 255, using random and ordered dithering, for 1000 slices. The period is 32, $base = 0$ and $bump = 1$. The AAP error reaches a maximum error of 49%.

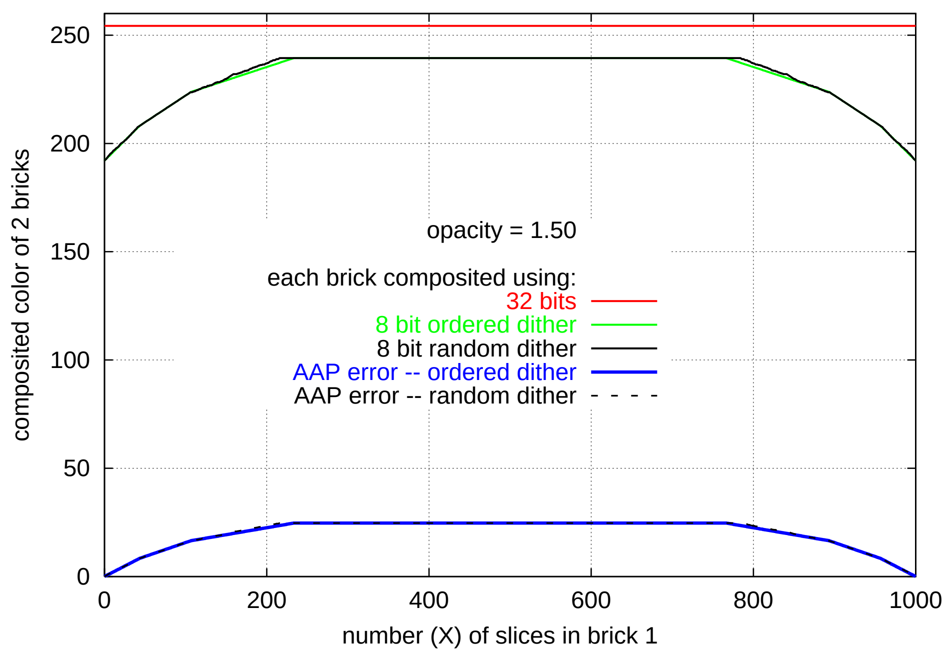

Figure 21: Bricking artifact problem for slices with opacity of 1.5 and color of 255, using random and ordered dithering, for 1000 slices. The period is 32, $base = 0$ and $bump = 1$. The AAP error reaches a maximum error of 24%.

Once a plateau has been reached, it is necessary to increase the bump so $(base + bump) \geq AlphaMin(pv)$. In Figure 22 we use *exponential bumping*, described below, to dither $\alpha = 0.50$. As can be seen in this figure, exponential bumping causes the 8 bit composited values to better track the ideal 32 bit composited values. In the next paragraph we indicate how to further improve this tracking. Our exponential bumping method works by increasing the bump value whenever the accumulated pixel value reaches a plateau for the current value of $base + bump$. Note that Equation 9 takes into consideration the bump size so the number of hits is correctly adjusted. We discuss the problem of determining when the accumulated pixel values reach a plateau in Section 9.2.

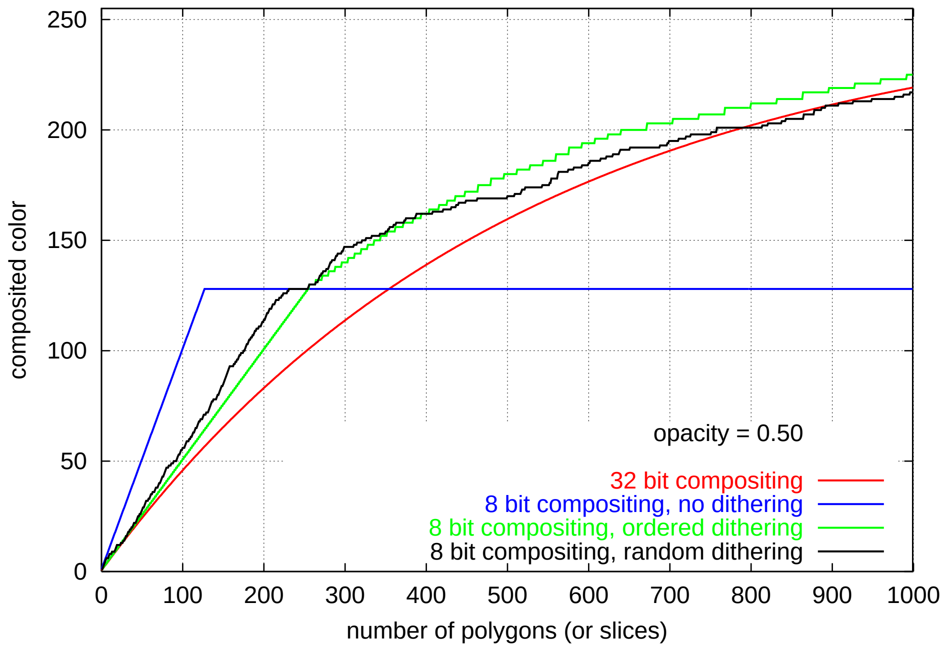

Figure 22: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 0.50 and color 255. The dithering period is 32.

Returning to Figure 22, we see that the initial slopes of the curves for both dithering methods are too steep, relative to the 32 bit ideal. For a constant input color and opacity, as is the case in our examples presented here, the 32 bit composited color varies with the number of slices x as $c(x) = 1 - e^{-sx}$, where s is a function of the input opacity. Therefore the slope of the 32 bit composited color is:

$$\frac{d}{dx}[c(x)] = se^{-sx} \tag{11}$$

Whereas, the slope of the dithered color up to the first plateau is just the input opacity value, for example if the input opacity is 0.75 then the slope of the composited dithered color is $0.75x$. For small input opacity values (less than about 1.0), prior to $x = 128$, the first plateau value, the slope of the dithered opacity is always higher than that of Equation 11. As the input values increase above 1.0, this upward bias decreases rapidly. The dithered values have two distinct slopes in Figure 22, one region is from slice 0 to about slice 250 (where the dithered value reaches the undithered plateau value) and the other from slice 250 to slice 1000. Our solution is to use a slope modifier in each of these regions to correct these biases. The predithered alpha is multiplied by the slope modifier, and then dithered. When the slope modifier is changed, a new dithering pattern is computed. The value of the slope modifiers are determined empirically by inspection of the simulator output. Figure 23 shows the results of dithering $\alpha = 0.50$, using exponential bumping and two slope modifiers. The first slope modifier is 0.75 for the lower region, and the second is 1.0 for the higher region. The resulting curve is very close to the desired 32 bit (red) curve. After introducing some new terminology in Section 4, we state the criteria for setting the slope modifiers.

4 RESULTS

Dithering results for opacity values of 0.03, 0.25, 0.75, 1.0, 1.10, 1.25, 1.5, 1.75 and 2.0 are shown in Figures 24 through 32. Unless otherwise noted, the dithering period is 32 for all plotted results. These plots show that the dithering methods give good results over this range of input opacity values. The figure captions give details on the dithering parameters used, and comment on the results. Note that dithering an opacity of 1.0 or 2.0 is problematic when alternating between a base and a bump value since both the base and the bump must be integers. See Figure 33 where the dithering uses exponential bumping but no slope modifiers. The use of slope modifiers nicely solves this problem as can be seen in Figures 27 and 32.

Figure 34 shows that when exponential bumping and slope modifiers are not used, a *knee* occurs when the composited value reaches *plateau(base)*. After reaching the knee, $base$ input values have no effect, only the $base + bump$ input values increase the pixel value until *plateau(base+bump)* is reached. This suggests that it should be possible to even further improve the plot shown in Figure 31 by the use of three slope modifiers, one to the knee, one from the knee to the plateau, and one for the remainder of the curve. We have not implemented this optimization. For our two slope modifiers, we change from the first to the second as follows: (a) when undithered $\alpha \leq 1$, we change when the composited color is equal to $plateau(1) = 128$; (b) otherwise, we change when the composited color is equal to *plateau(base)*.

Figure 35 shows the the bricking artifact problem for an input opacity of 1.0 and a color of 255 for 1000 slices, using dithering with exponential bumping and slope modifiers as in Figure 27. The maximum absolute value of the AAP error is now 3.9% for random dithering and 0.66% for ordered dithering.

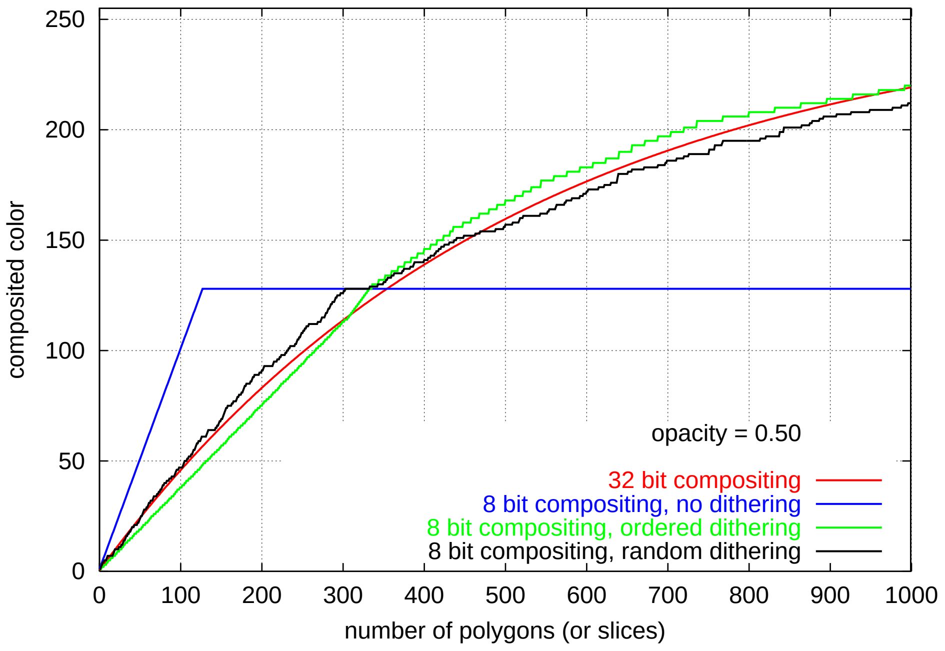

Figure 23: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 0.50 and color 255. The fi rst slope modifi er has a value of 0.75 and the second a value of 1.0.

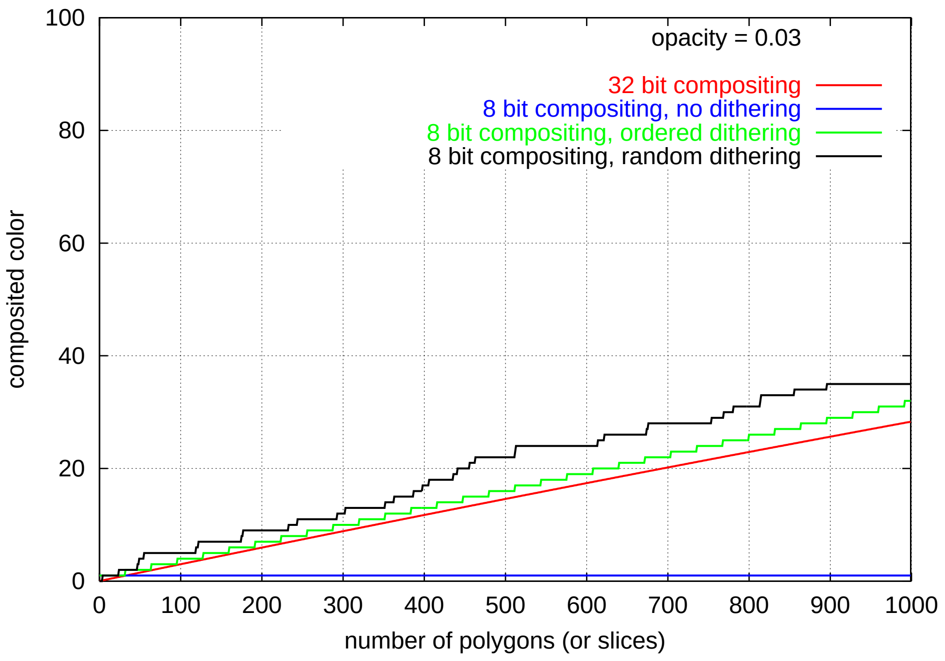

Figure 24: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 0.03 and color 255. The fi rst slope modifi er is 0.75 and the second 1.0. Since the slope here is small, the curve is not sensitive to slope modifi ers, and setting both to 1 gives equally good results.

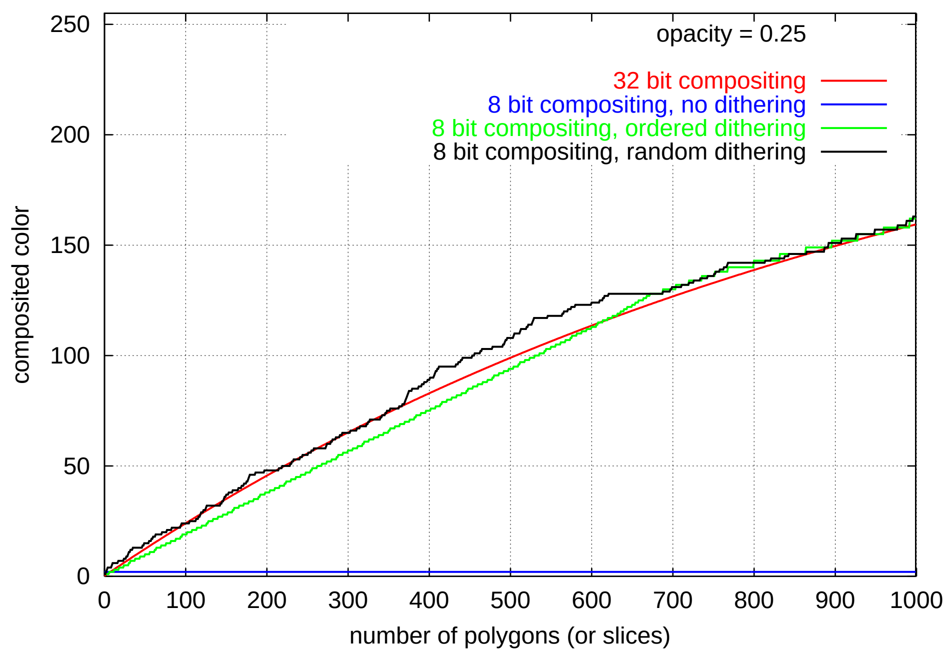

Figure 25: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 0.25 and color 255. The first slope modifier has a value of 0.8 and the second a value of 0.9.

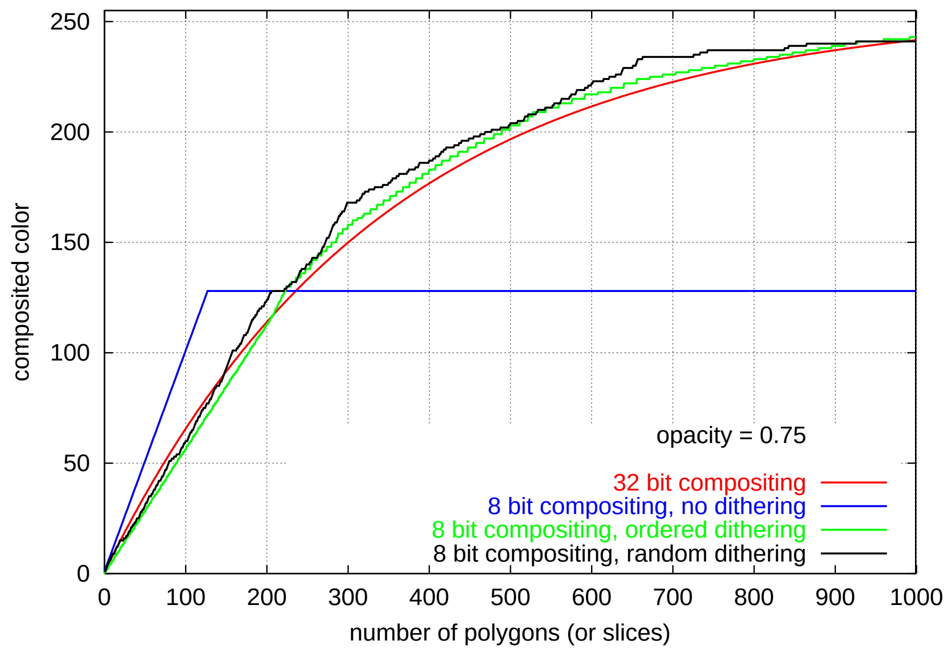

Figure 26: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 0.75 and color 255. The first slope modifier has a value of 0.75 and the second a value of 1.0.

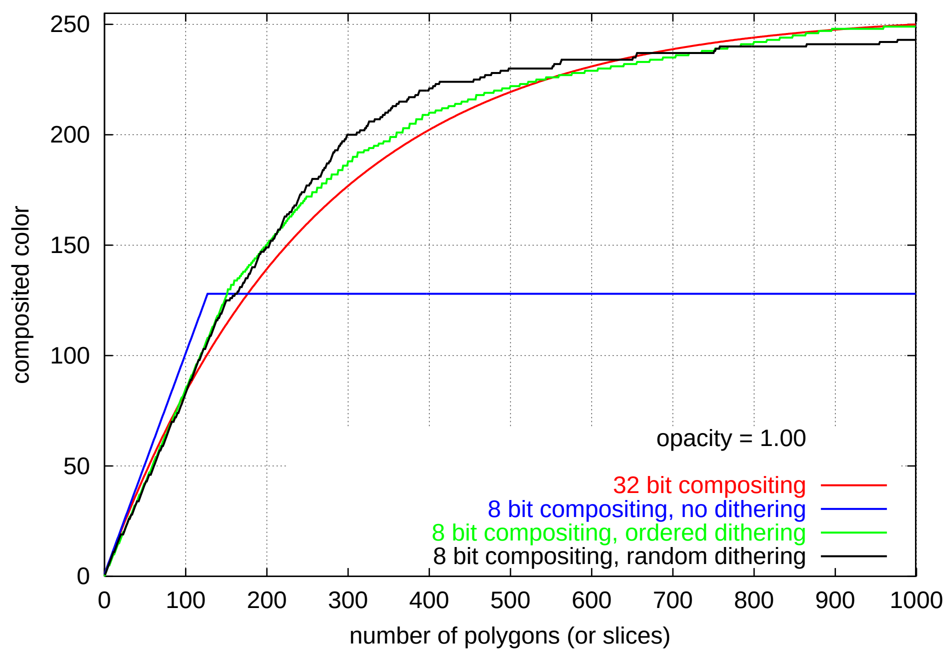

Figure 27: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 1.0 and color 255. The first slope modifier has a value of 0.85 and the second a value of 0.9.

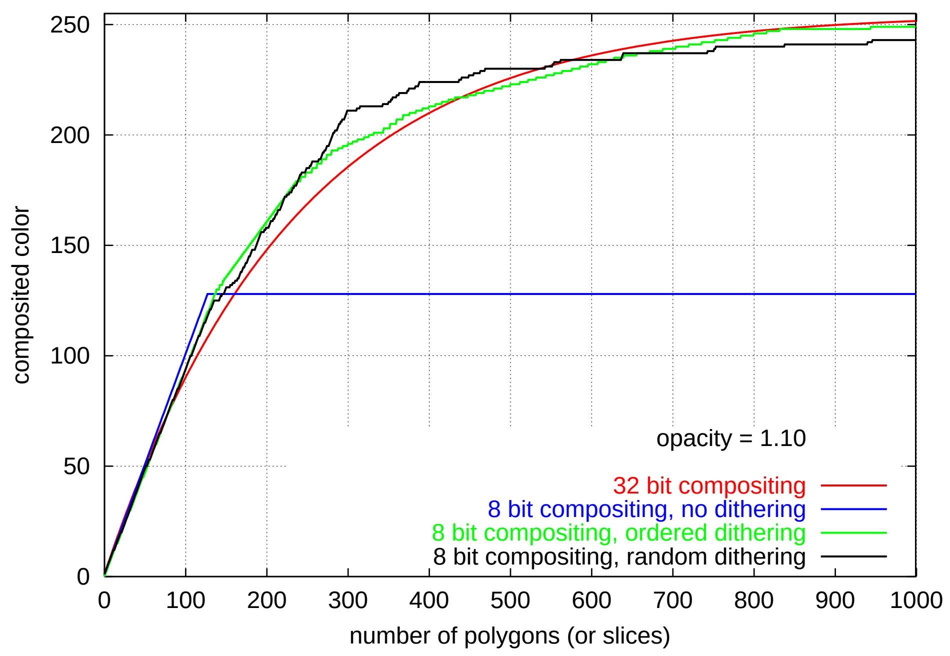

Figure 28: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 1.10 and color 255. The first slope modifier has a value of 0.85 and the second a value of 0.9.

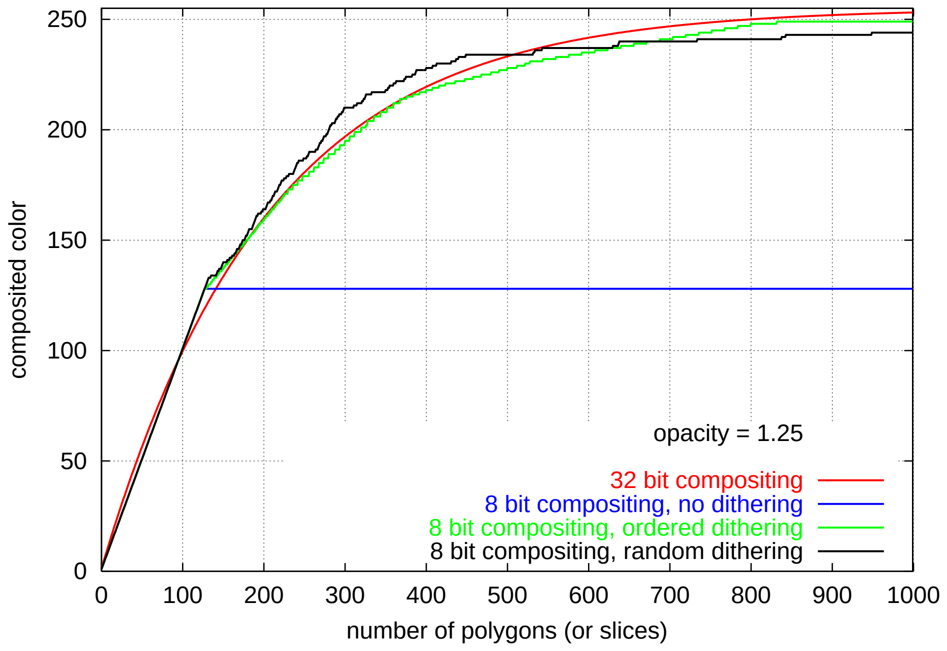

Figure 29: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 1.25 and color 255. The first slope modifier has a value of 0.8 and the second a value of 1.5. Note the need for a slope modifier which is greater than one for this case.

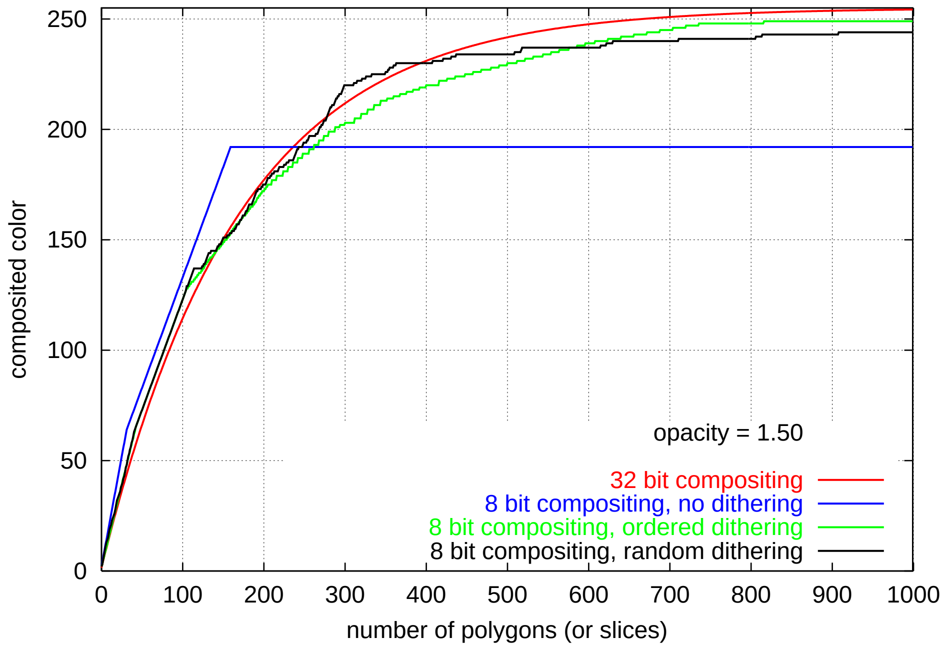

Figure 30: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 1.5 and color 255. The first slope modifier has a value of 1.0 and the second a value of 1.3.

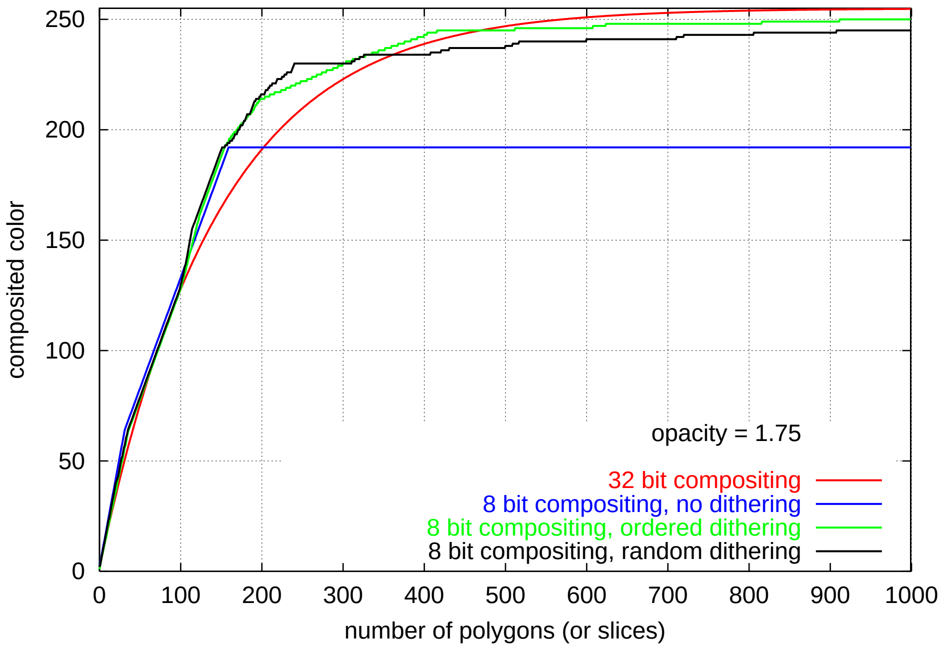

Figure 31: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 1.75 and color 255. The first slope modifier has a value of 1.0 and the second a value of 1.686. The second slope modifier is carefully chosen so that it will not cause the input alpha value to increase to 3 or above, since the dithering cutoff value, defined in Section 3, is 3.

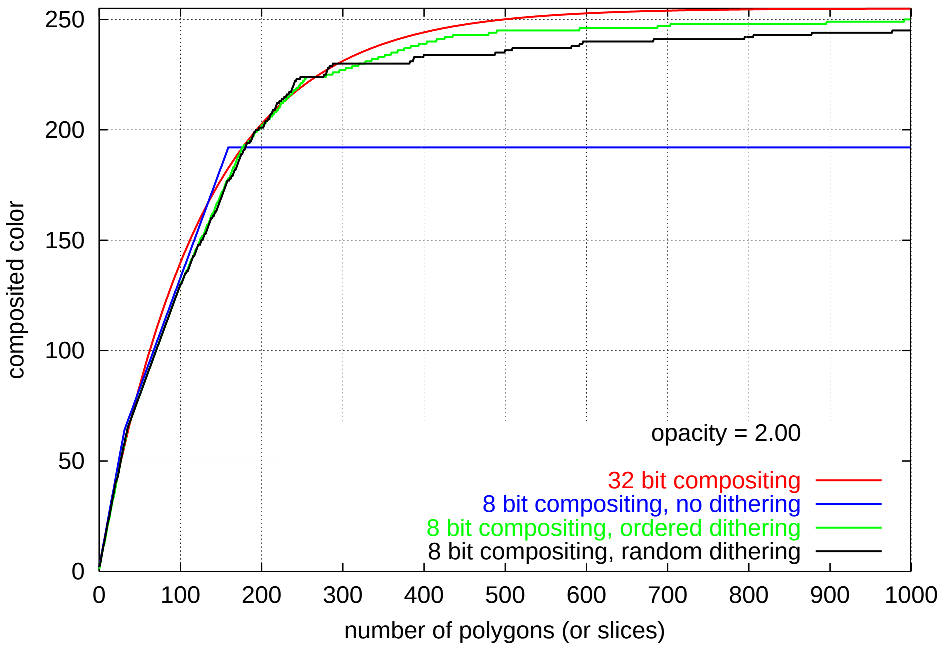

Figure 32: Accumulated pixel values when using random and ordered dithering with exponential bumping. Polygons have an opacity 2.0 and color 255. The first slope modifier has a value of 0.9 and the second a value of 1.4.

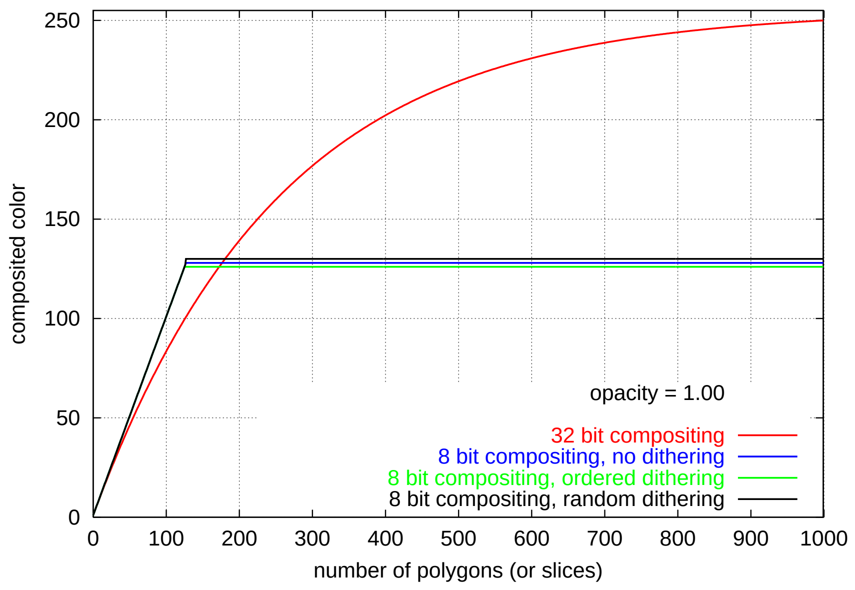

Figure 33: Accumulated pixel values when using random and ordered dithering with exponential bumping and no slope modifi ers. Polygons have an opacity 1.0 and color 255. Note in comparison to Figure 27, which is also for an opacity of 1.0, how important the use of slope modifi ers are in enabling dithering for an integer input opacity value.

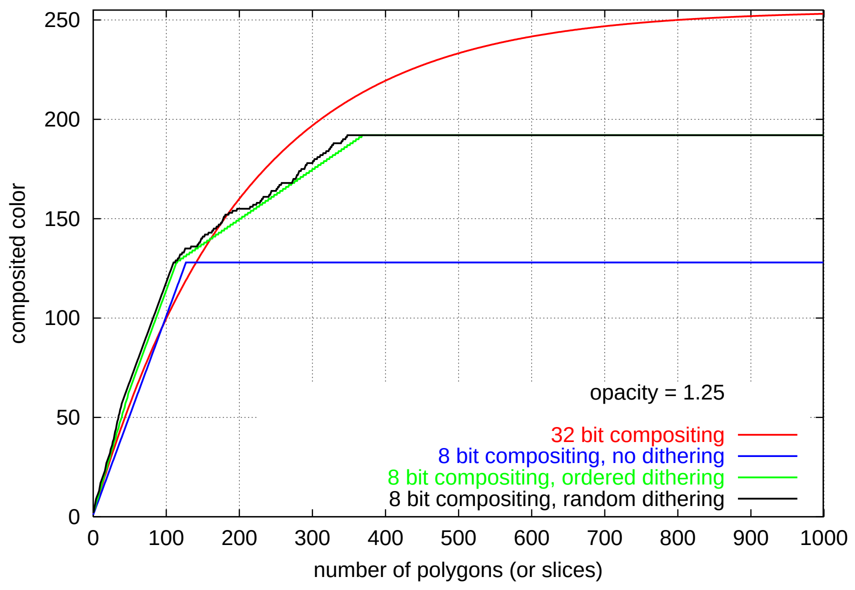

Figure 34: Accumulated pixel values when using random and ordered dithering but with no exponential bumping and no slope modifi ers, i.e. $base = 1$ and $base + bump = 2$ throughout. Polygons have an opacity 1.25 and color 255. A *knee* occurs when the composited color reaches $plateau(base) = 128$. The $base + bump$ input values continue to increase the pixel value until $plateau(base + bump) = 192$.

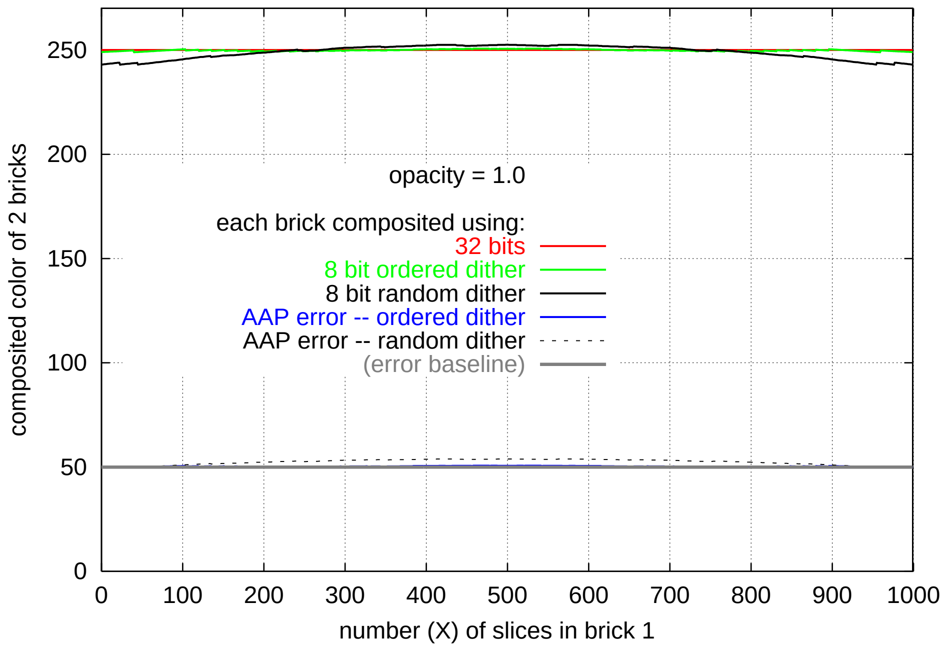

Figure 35: Bricking artifact problem for slices with an opacity of 1.0 and a color of 255, using random and ordered dithering, with exponential bumping and slope modifi ers as in Figure 27. The maximum absolute value of the AAP error is now 3.9% for random dithering and 0.66% for ordered dithering, a big improvement over the undithered case where the AAP maximum error was 49%.

4.1 48 Bit Frame Buffers

Certain graphics hardware has 48 bit frame buffers, 12 bits per channel. Those additional four bits per channel make a big difference in the accuracy of compositing as can be seen in Figures 36 through 38 which show the results of 12 bit compositing, without dithering, to composite slices with opacities of 1/4095, 0.1/255 and 1/255 respectively.

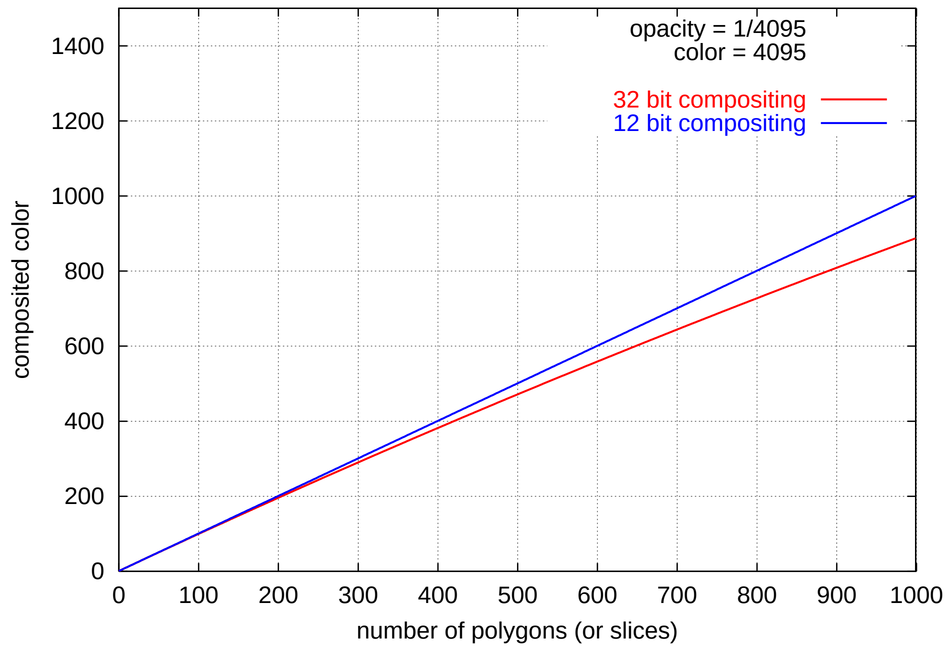

Figure 36: Compositing slices with $\alpha = 1/4095$ and color = 4095 using 12 bit compositing and no dithering.

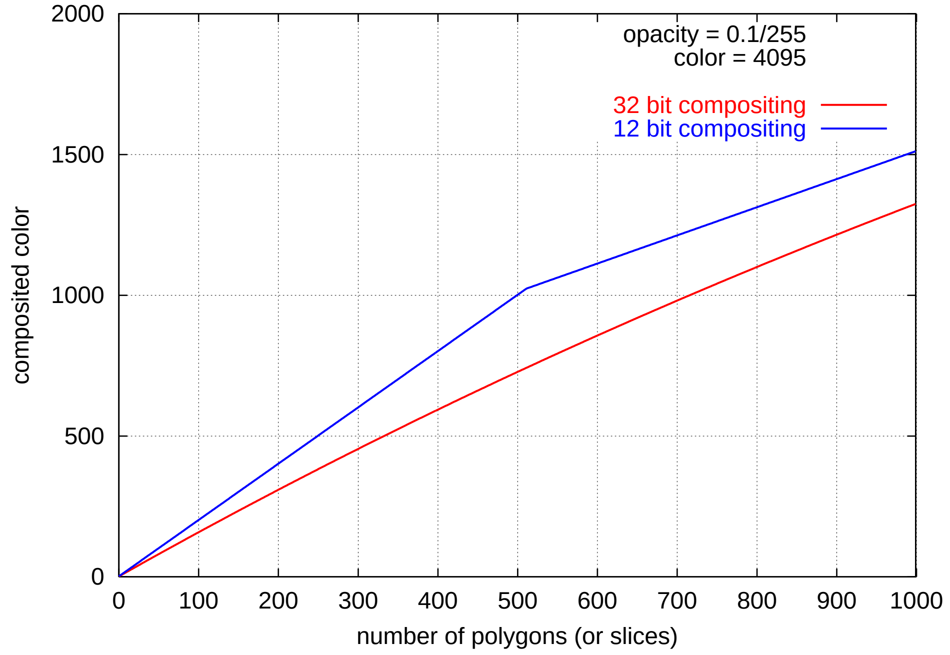

Figure 37: Compositing slices with $\alpha = 0.1/255$ and color = 4095 using 12 bit compositing and no dithering.

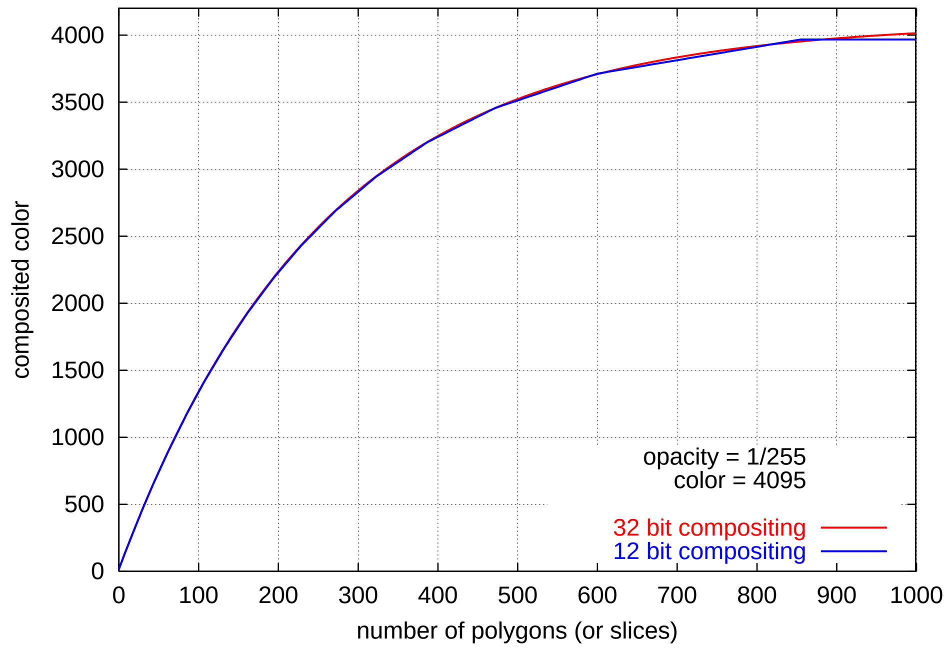

Figure 38: Compositing slices with $\alpha = 1/255$ and color = 4095 using 12 bit compositing and no dithering.

5 DITHERING COLOR

In the work presented so far, only opacity has been dithered. Let c and α be the current color and opacity to be dithered and let $\alpha' = dithered(\alpha)$, then, for premultiplied color, we submit $c_{in} = c \times \alpha'$ and $\alpha_{in} = \alpha'$. When not using premultiplied color, that is when opacity is not being accumulated, we submit $c_{in} = c$ and $\alpha_{in} = \alpha'$. We do not dither color and opacity separately because this can cause the accumulated color to oscillate significantly, causing artifacts. This is the case because the color and opacity inputs can have different dither patterns. When color is 255, this is a special case when there is no difference between dithering both color and opacity and just dithering opacity, since $255 \times dithered(\alpha) = dithered(255 \times \alpha)$. Let us take a case where opacity and premultiplied color are being submitted, and the color is not 255. If we dither color and also dither opacity, $\alpha_{in} = dithered(\alpha)$ and $(\alpha \times c)_{in} = dithered(\alpha \times c)$. Consider the case when slices have a color of 200 and an opacity of 0.75. Take an instance when the current pixel color is 157, and:

$$base_{\alpha \times c} = base_{\alpha} = 0, \qquad bump_{\alpha \times c} = 6, \qquad bump_{\alpha} = 7.$$

Suppose dithering results in a hit for opacity and not a hit for color. Then input $\alpha_{in} = 7$ and input $c_{in} = \alpha' \times c = 0$, so from Equation 1, $c_{pixel} = 157 \times (1 - (7/255) + 0 = 152.69$. The accumulated color has just gone from 157 down to 153! And this drop could be repeated several times in a row, hence we get the wide swings seen in Figure 39 where separate dithering is used for color and opacity. This same phenomena can occur if exponential bumping is not used, and it can occur when color is not premultiplied.

If only color is being used, i.e. opacity is not used, there is no reason to dither color since there is no compositing. In the case where the input color c_{in} is very small and opacity is not, if we are accumulating only c_{in}, then, as the blending equation shows, the final accumulated color can never exceed c_{in}. Therefore there is no point in dithering color in this case. In general, we recommend setting the fog or haze color for low-opacity polygons to fully or near fully saturated values (color values close to 255). Then when color is premultiplied by opacity we get the most accuracy and sensitivity.

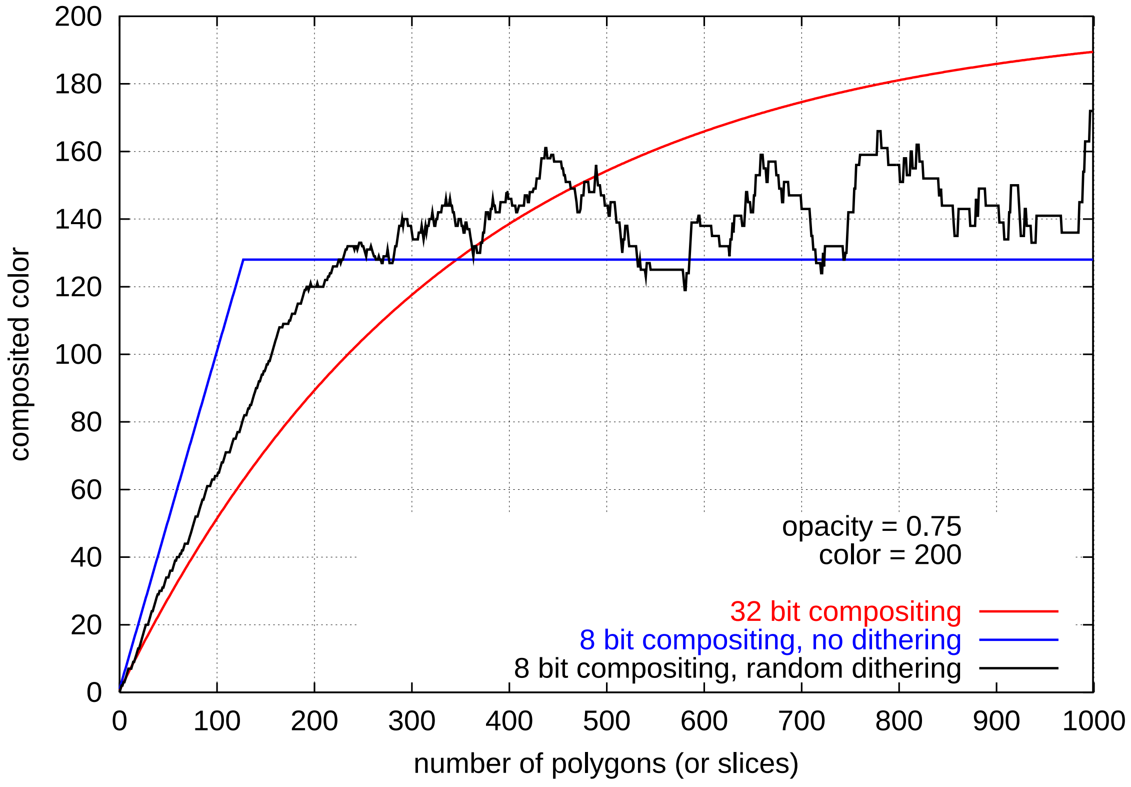

Figure 39: Here opacity and (opacity×color) are dithered separately for slices with $\alpha = 0.75$ and color = 200. To avoid the oscillation in composited color shown here (and the artifacts that result), we do not recommend dithering both color and opacity. The same phenomena can occur when using ordered dithering.

6 TRANSFER FUNCTIONS AND TEXTURE MAPS

In volume rendering, typically color and opacity are calculated from user-defined transfer functions which map scalar field values into $RGBA$ values. We keep our primary transfer functions, the ones specified by the user, in floating point tables. This allows the user to specify opacities and colors with high precision, and has two benefits. These values can be stored for future use, perhaps with a high accuracy software volume rendering system. And, we can use these accurate opacity values to calculate the dither pattern and the texture maps, as explained below.

In a preprocessing step, the scalar field values are normalized to floating point values in the range

$[0, n-1]$, where n is the size of the texture table, described next. We implement the transfer functions as a one dimensional texture table, which maps scalar field values to RGBA values. Typically the size of the texture table is 256. When a polygon is rendered, the normalized scalar value at each polygonal vertex is entered as a texture coordinate. Keeping the transfer functions in texture has a number of advantages. First, the transfer functions can be changed on the fly merely by creating and downloading a new texture map, rather than modifying all the scalar field values. Second, since the texture hardware interpolates the scalar value across the polygon, and then applies the transfer functions, this results in a more accurate image. If texture was not used, then the color and opacity would be interpolated across the polygon which might miss fluctuations in the transfer functions. Third, having the transfer functions in texture makes dithering easy to implement.

The basic opacity dither patterns are calculated once, as described in Sec 3, for a given period and a given transfer function and then stored as a set of textures. The colors, premultiplied by the appropriate dithered opacity values if necessary, are also stored in these texture tables. Table 3 shows an example set of dithered transfer functions in a 2D table, (how the transfer functions are calculated is discussed below). Each vertical column, 0 through 7, in Table 3, is a transfer function represented as a texture map. Before slice s is rendered, texture map number ($s \ \% \ P$) is downloaded into texture memory, (where P is the dithering period). To avoid the overhead of downloading a new texture map between every slice, it is possible to bind each transfer function to a texture object. If there is enough texture memory, these objects will all remain in texture memory. Then between each each slice, the appropriate texture map is selected as the current texture by:

```
glBindTexture(GL_TEXTURE_1D, texNames[slice % P]);
```

thus avoiding the overhead of downloading a new texture with each slice. Of course, the textures will have to be modified downloaded again when the bump values or slope modifiers are changed. Another alternative, which may prove faster (we did not test it) is to create a 2D texture table, where the second texture component is $(slice \ \% \ P)$. Table 2 gives timings for two of these methods. Binding the transfer functions to texture objects was significantly faster than downloading a new texture between each slice. For our 256 slot transfer functions with a dither period of 32, we observed that all 32 bound texture objects remained resident in texture memory at all times.

A histogram of the scalar field values being rendered is very useful in setting preliminary transfer functions. See Figure 40. Each bin of the histogram has an equal range of data values. Let sv_{min} and sv_{max} be the maximum and minimum scalar values in the data set, and n be the number of bins in the histogram. (Generally we make the number of bins in the histogram the same as the number of entries in the transfer functions.) Then the value in bin i is the number of scalar field values between $i \times (sv_{max} - sv_{min})/n$ and $(i+1)(sv_{max} - sv_{min})/n$. We say those scalar field values are *in* bin i. The data in the three bins with very high values, which we call *peak data*, will be mapped to a low opacity value to form the background haze or fog. Generally a histogram of the data will show one of more such peaks.

Figure 41 zooms in on the first 25 bins of the histogram in Figure 40 to show more detail as we discuss the use of the histogram to create a preliminary transfer function. We assign an opacity

Method	Total Rendering Time	Texture Change Time
Texture Unchanged Between Slices	0 0.849	0
Bound Textures	0.855	0.006
Download Texture Between Slices	0.882	0.033

Table 2: Timings for volume rendering 450 slices using an NVIDIA GeForce3 graphics card to a 1000 × 1000 window. A total of 658,660 polygons were rendered. The fi rst method bound the tranfer functions to a texture once only at the beginning, therefore no dithering was used. The second method downloaded a new texture between each slice, thus implementing dithering —we used a dither period of 32. The fi nal method bound all 32 transfer functions to 32 texture objects which remained resident in texture memory throughout. Dithering was accomplished as described in the text by changing the binding between each slice. The times shown are averages over 30 renderings each.

value to each bin in an inverse relationship to the value of the bin. Here we use a simple formula:

$$\alpha(i) = (1/ln(bin(i))) \times S, \tag{12}$$

where S is a scale factor that depends on the data set. We used $S = 100$ to bring the opacities into the range $[0.0, 22.0]$. We only use this formula for non-peak bins since, when using dithering, it is preferable that all the data in a peak bin be mapped to a single low opacity value. (Here we assume only peak data will be dithered.) This avoids problems that may arise due to interpolation between values in the transfer function texture map. Since our dithering patterns are set for a specific opacity value and occupy a specific entry in the texture map (see Table 3), interpolating between entries would not be meaningful in the context of a single dithering pattern. So we map all the data in a peak bin to a single opacity by assigning the opacity so it straddles the bin. See Figure 41, where we have assigned both peak bins an opacity of 0.25. Note that for the first peak, bin 2, the corresponding low opacity 0.25 has also been assigned to bins 1 and 3, in addition to bin 2. This ensures that all peak data values in bin 2 are mapped to the same opacity. In order that the data in bins 1 and 3 is not neglected, we use the average value of bins 0 and 1 to set the opacity for bin 0, similarly, the opacity of bin 4 is based on the average value of bins 3 and 4. From bin 5 through bin 8, and after bin 13 the opacity of bin i depends only on the value of bin i using the above formula. The transfer functions for this portion of the data is shown in Table 3. Note that the dither patterns have been permuted to provide spatial dithering. This transfer function, with minor modifications, was in fact used to create the images shown in Figures 2, 3, and 6.

The above strategy implies the use of the texture filter option GL_LINEAR. An alternative to assigning opacity to straddle peak bins is to set the option to $GL\_NEAREST$, which has the effect of snapping all values in a bin to the marked points on the blue curve in Figure 41. Therefore Equation 12 could be used for all bins. There is another advantage to using $GL\_NEAREST$. When using GL_LINEAR, certain graphics cards do the texture look-up for each vertex, followed by interpolation, which may cause a color artifact. Other cards do the interpolation first, and then the look-up, which may be more appropriate.

It is preferable that all peak bins be mapped to the same low opacity or at least to a limited number of low opacities. Peaks can be mapped to different colors, but as discussed before, preferably to saturated color values, i.e. values close to 255.0. In this way, specific dither patterns can be created for each low alpha value that do not interfere with each other.

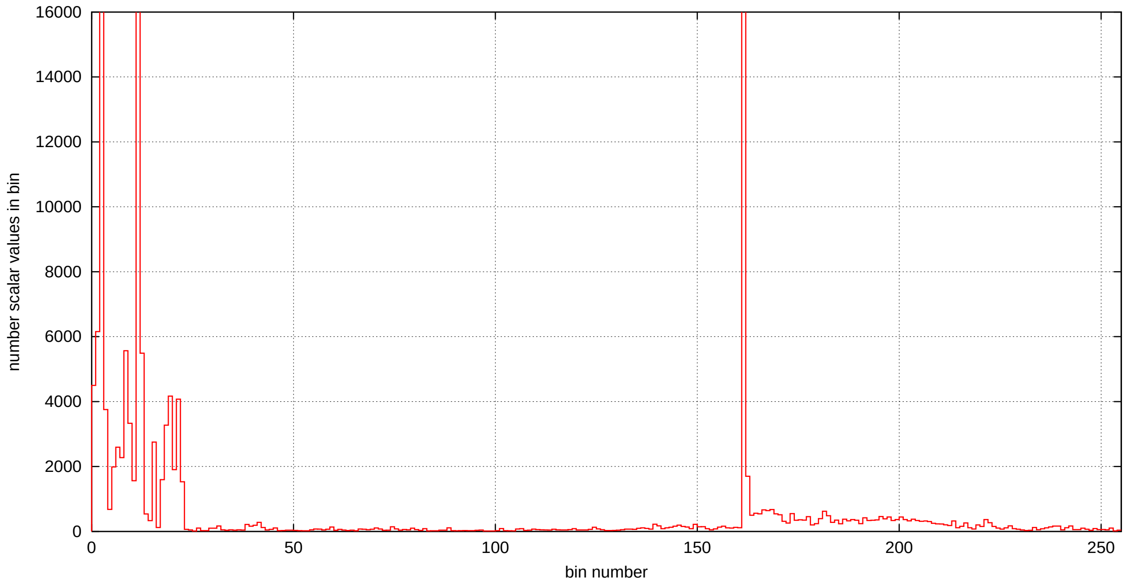

Figure 40: Histogram of scalar fi eld values in a typical scientifi c data set. Each bin has an equal range of data values. The histogram has 256 bins since our transfer function table has 256 entries. The data in the three bins with very high values will typically be mapped to a low opacity haze value.

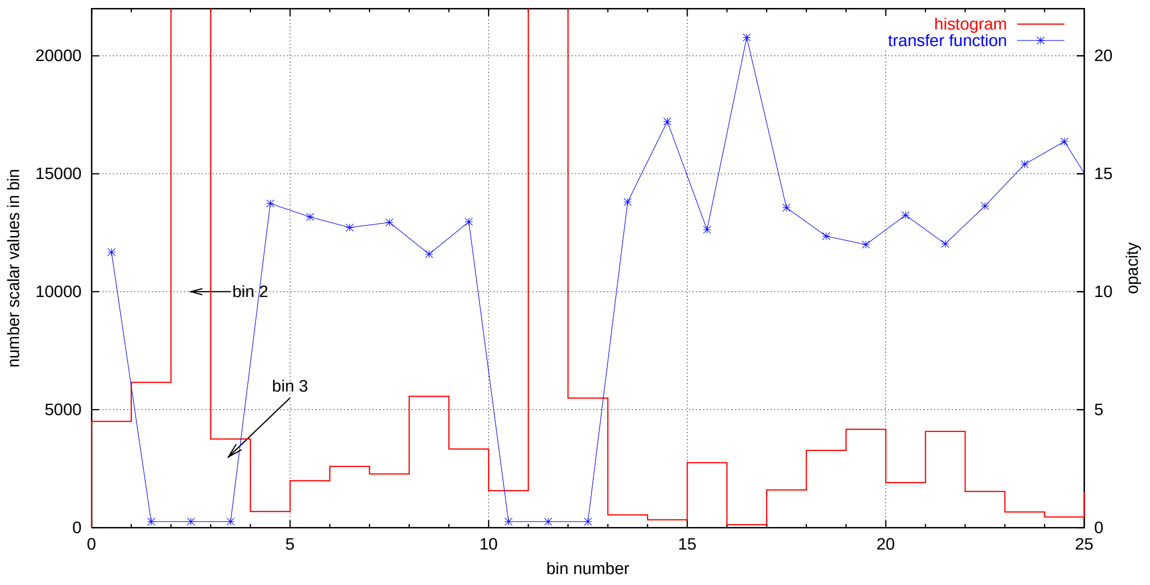

Figure 41: Enlarged portion of the histogram shown in Figure 40. The blue line shows the transfer function. The methodology for calculating the transfer function using the histogram is described in the text. There are two peak data bins.

	Texture map numbers 1 to 8 holding dithered transfer functions								
texture index	0	1	2	3	4	5	6	7	comment
0	23	23	23	23	23	23	23	23	$\alpha = 23$
1	0	1	0	0	0	1	0	0	$\alpha = 0.25$
2	1	0	0	0	1	0	0	0	$\alpha = 0.25$
3	0	0	0	1	0	0	0	1	$\alpha = 0.25$
4	28	28	28	28	28	28	28	28	$\alpha = 28$
5	27	27	27	27	27	27	27	27	$\alpha = 27$
6	26	26	26	26	26	26	26	26	$\alpha = 26$
7	27	27	27	27	27	27	27	27	$\alpha = 27$
8	23	23	23	23	23	23	23	23	$\alpha = 23$
9	27	27	27	27	27	27	27	27	$\alpha = 27$
10	0	1	0	0	0	1	0	0	$\alpha = 0.25$
11	0	0	1	0	0	0	1	0	$\alpha = 0.25$
12	0	0	0	1	0	0	0	1	$\alpha = 0.25$
13	28	28	28	28	28	28	28	28	$\alpha = 28$
14	34	34	34	34	34	34	34	34	$\alpha = 34$
15	25	25	25	25	25	25	25	25	$\alpha = 25$
16	42	42	42	42	42	42	42	42	$\alpha = 42$
17	28	28	28	28	28	28	28	28	$\alpha = 28$
18	25	25	25	25	25	25	25	25	$\alpha = 25$
19	24	24	24	24	24	24	24	24	$\alpha = 24$
20	27	27	27	27	27	27	27	27	$\alpha = 27$
21	25	25	25	25	25	25	25	25	$\alpha = 25$
22	28	28	28	28	28	28	28	28	$\alpha = 28$
23	31	31	31	31	31	31	31	31	$\alpha = 31$
23	33	33	33	33	33	33	33	33	$\alpha = 33$
254									
255									

Table 3: Example set of transfer function texture maps for the transfer function shown in Figure 41. These tables implement ordered dithering with a period of 8 and a dithering cutoff value of 3.0. (Only alpha values are shown here. In a real texture map, the colors, R, G, and B, premultiplied by dithered opacity values if necessary, would also be included.) Note that the dither pattern is permuted to give spatial dithering.

7 Simple Dithering Using Chunking

Since, as discussed in the Conclusion, it is not yet known how to fully implement our dithering technique, we here discuss a method that can be implemented. Dithering with a single slope modifier and no exponential bumping, a process what we call *simple dithering*, does an adequate job of correcting the low-opacity fog problem for very low opacities, i.e. for $\alpha < 1$. The dithered images in Figure 3 were created using simple dithering with a period of 16; the fog had an opacity of 0.5. However, the AAP error for the bricking artifact problem increases dramatically when the number of slices in a brick increases to the point where the composited color reaches a plateau value. The only way to overcome the plateau is to increase the bump value. Since the bump value does not change in simple dithering, the bricking artifact problem is not addressed. For example, see Figure 19 where the AAP error is very large when using only simple dithering. (One alternative approach that we have not yet tried is to use an initial bump value large enough to overcome the necessary plateaus.) Since simple dithering uses one slope modifier, integer opacity values can be dithered (until the plateau in reached).

Another approach is to render several subsets of the slices, *chunks*, and composite the images of each chunk, a process we call *chunking*. One way this can be done is by rendering a chunk, doing a frame buffer readback of the image of the chunk, clearing the frame buffer, and then rendering the next chunk. The chunks can be composited by the CPU while the GPU is rendering the next chunk. A more efficient method, is to render each chunk to a texture (using `GLX_ARB_render_texture`) and then composite the resulting textures using the $ARB\_multitexture$ extension, or register combiners. Another alternative is to render each chunk to a pixel buffer *pbuffer* [16] (`GLX_ARB_pbuffer`) which might be practical since the chunks are being rendered off-screen on the nodes of a cluster. Even another alternative is to write out the frame buffer periodically to texture memory using `glCopyTexImage2D()` however this involves an extra pixel copy over rendering directly to texture.

We now present some results that help determine an optimal chunk size. Figures 42 through 54 show how chunking impacts the bricking artifact problem. These figures show the use of chunks with 100, 200, and 300 slices for input opacities of 0.50. 0.75, 1.0, and 1.5, using only simple dithering. The image of each brick is created by compositing the images of each of the chunks. The final image is the composition of the images of each brick. For input opacities up to 1.0, we composited the chunks until we reached a total of 1000 slices, so for example when using 100-slice chunks, 10 chunks are composited to form the image of each brick. For an input opacity of 1.5, the composited color saturates when using 1000 slices, so in order to more clearly see the effect of chunking, we show the results for a total of 500 slices. In all cases, the dithering period is 32. It is important that the number of slices in a chunk be such that the composited value of the chunk remains under the plateau. As can be seen in Figure 55 for an input opacity of 1.0 and in Figure 56 for an input opacity of 1.5, the ordered dither plateau value occurs around 150 slices and 230 slices respectively, therefore the chunk size should be less than that.

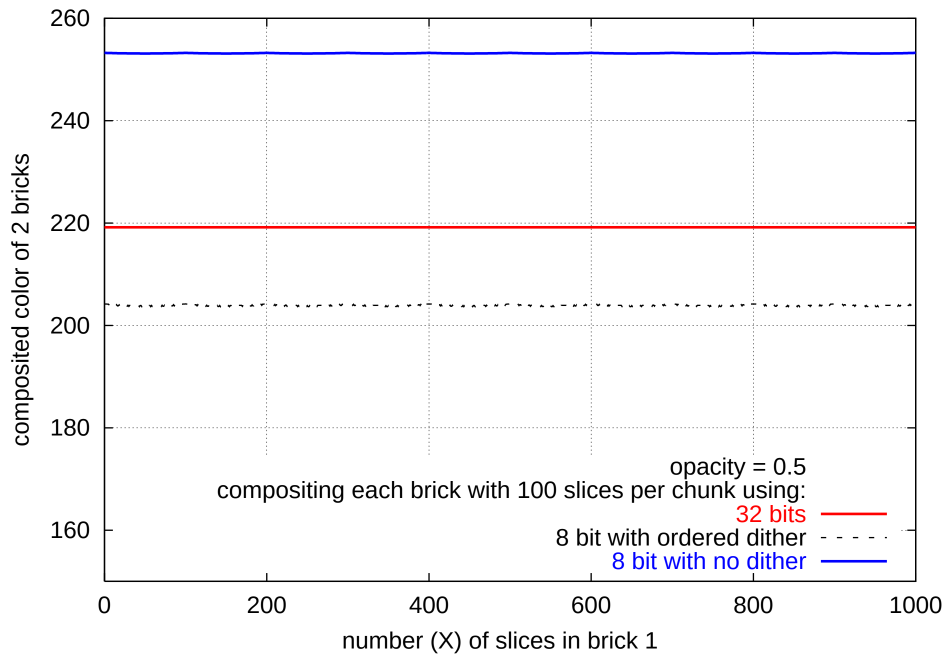

Figure 42: Bricking artifact problem where the image of each brick is created using 10 100-slice chunks with simple dithering with a single slope modifi er of 0.75. Slices have an opacity of 0.5 and color of 255.

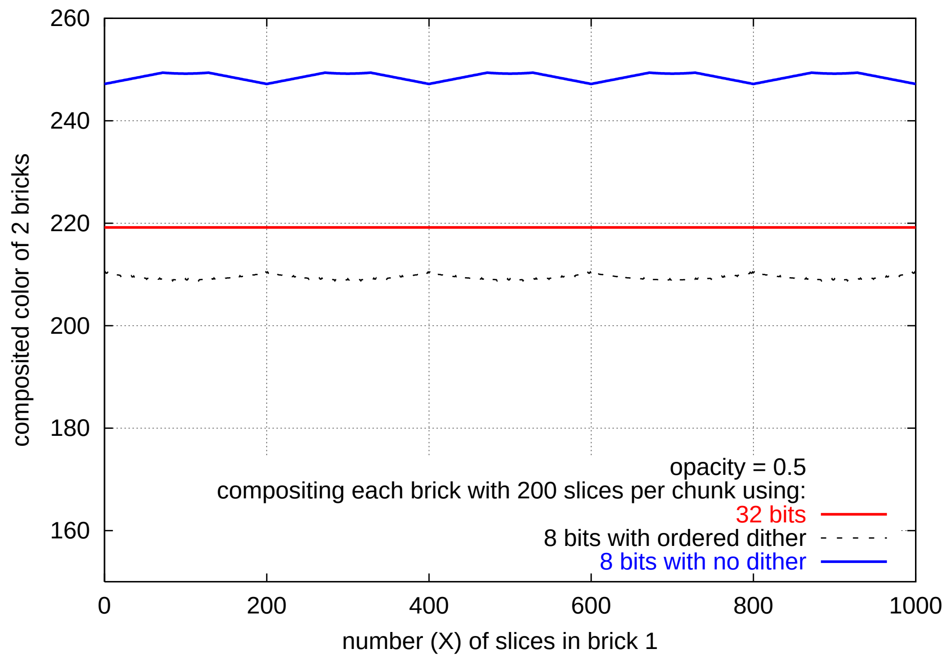

Figure 43: Bricking artifact problem where the image of each brick is created using 5 200-slice chunks with simple dithering with a single slope modifi er of 0.75. Slices have an opacity of 0.5 and color of 255.

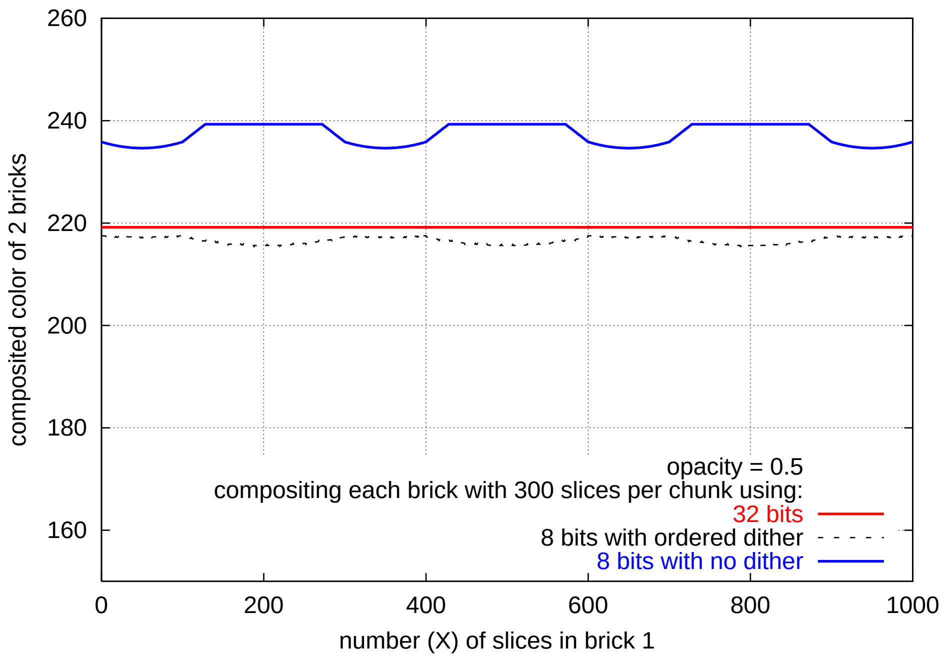

Figure 44: Bricking artifact problem where the image of each brick is created using 3 300-slice chunks (as well as a partial chunk of 100 slices) with simple dithering with a single slope modifi er of 0.75. Slices have an opacity of 0.5 and color of 255.

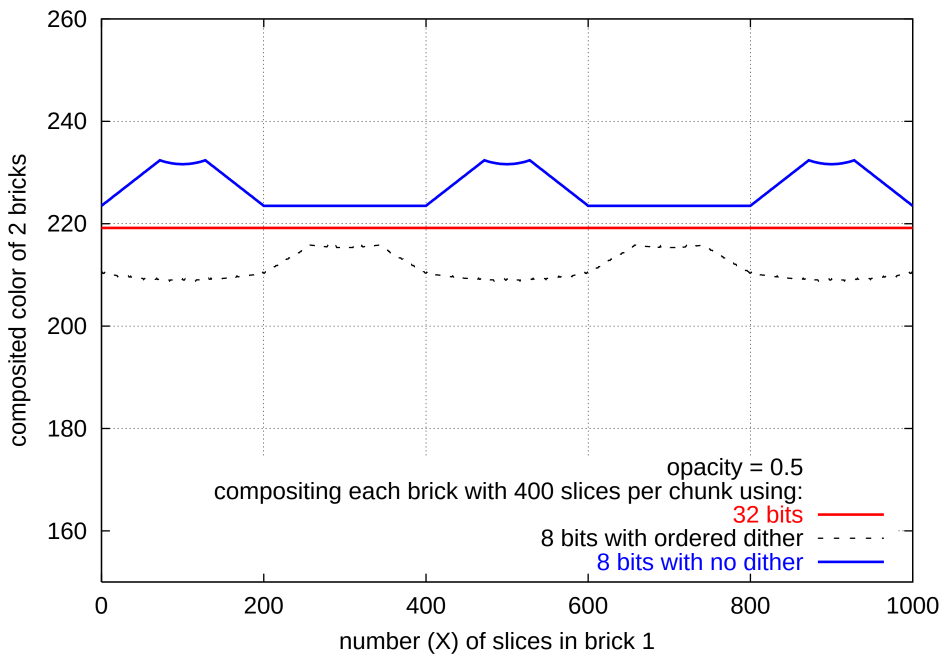

Figure 45: Bricking artifact problem where the image of each brick is created using 400-slice chunks dithering with a single slope modifi er of 0.75. Slices have an opacity of 0.5 and color of 255.

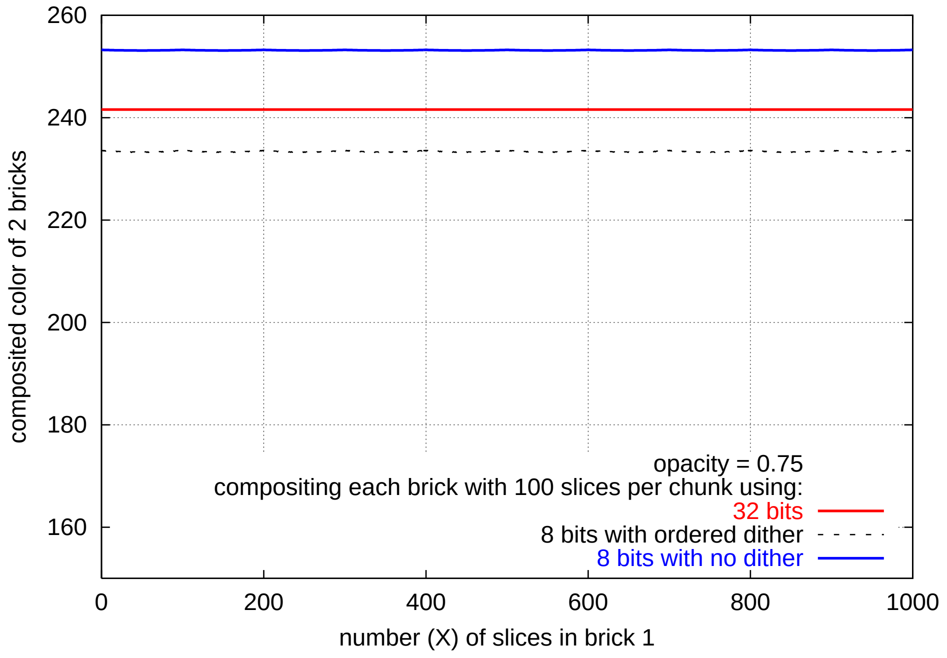

Figure 46: Bricking artifact problem where the image of each brick is created using 10 100-slice chunks with simple dithering with a single slope modifi er of 0.75. Slices have an opacity of 0.75 and color of 255.

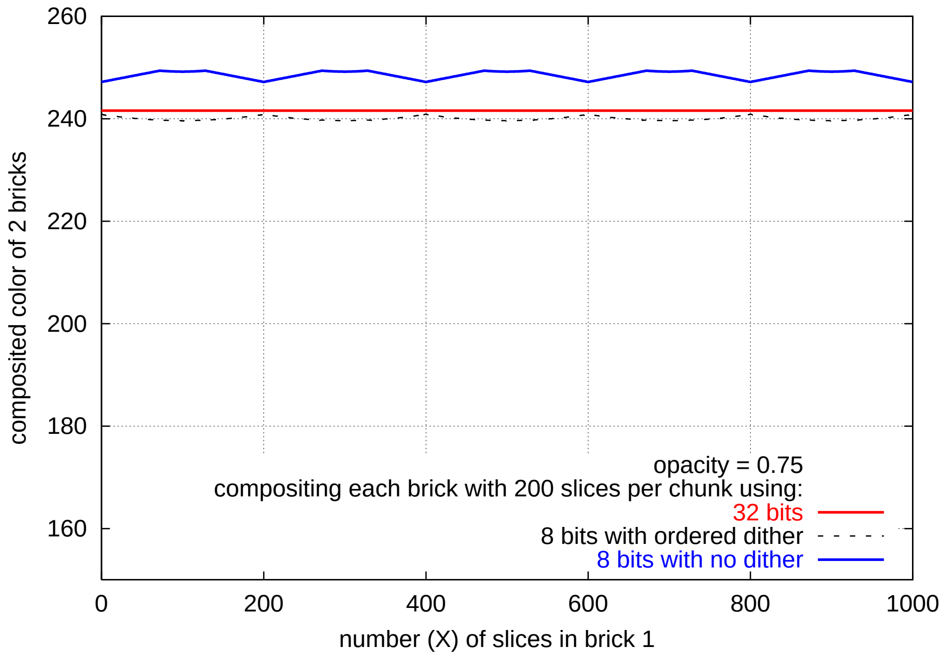

Figure 47: Bricking artifact problem where the image of each brick is created using 5 200-slice chunks with simple dithering with a single slope modifi er of 0.75. Slices have an opacity of 0.75 and color of 255.

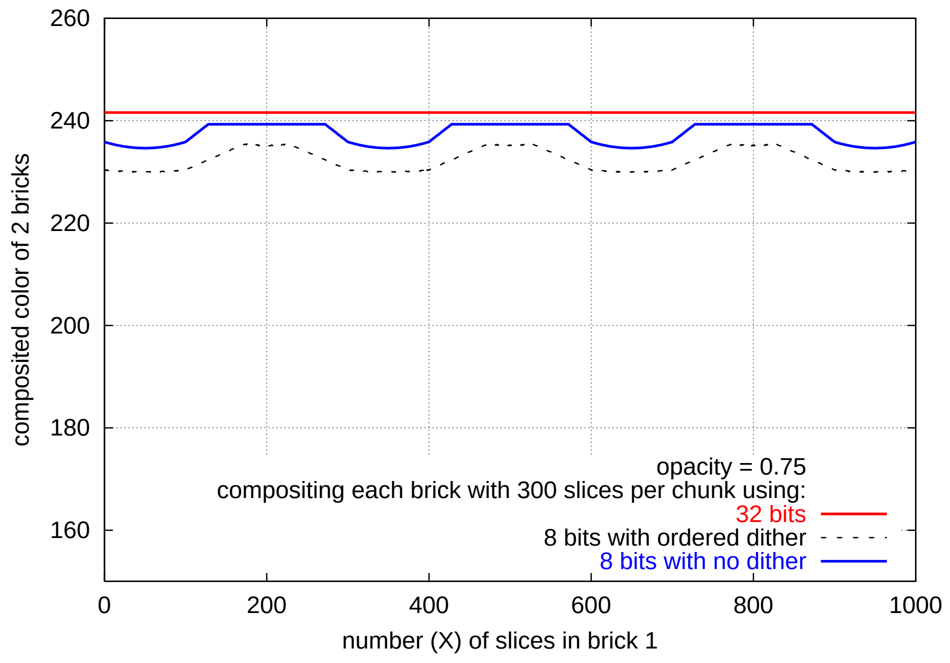

Figure 48: Bricking artifact problem where the image of each brick is created using 300-slice chunks with simple dithering with a single slope modifi er of 0.75. Slices have an opacity of 0.75 and color of 255.

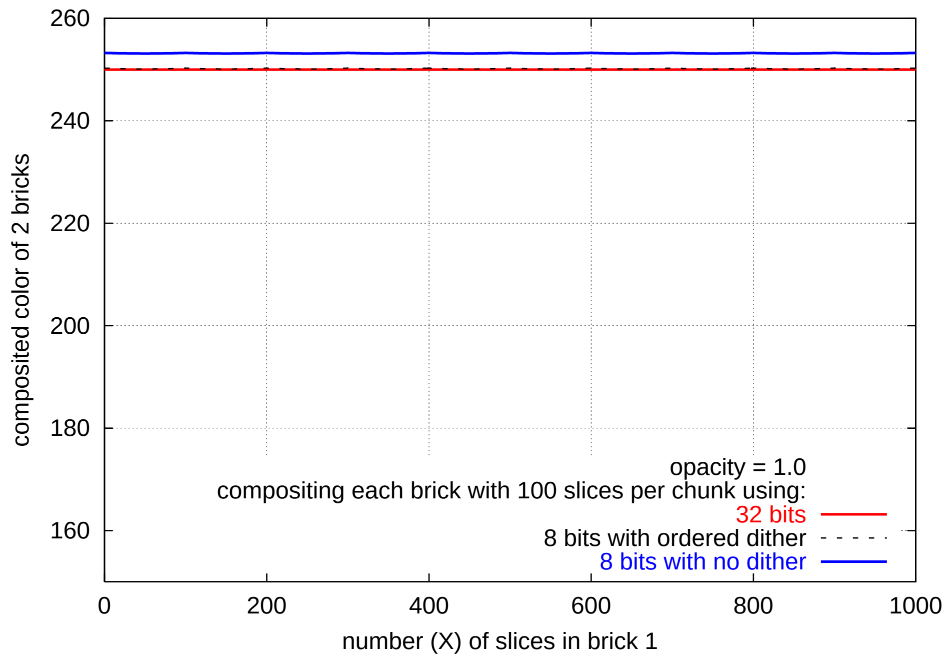

Figure 49: Bricking artifact problem where the image of each brick is created using 10 100-slice chunks with simple dithering with a single slope modifi er of 0.85. Slices have an opacity of 1.0 and color of 255.

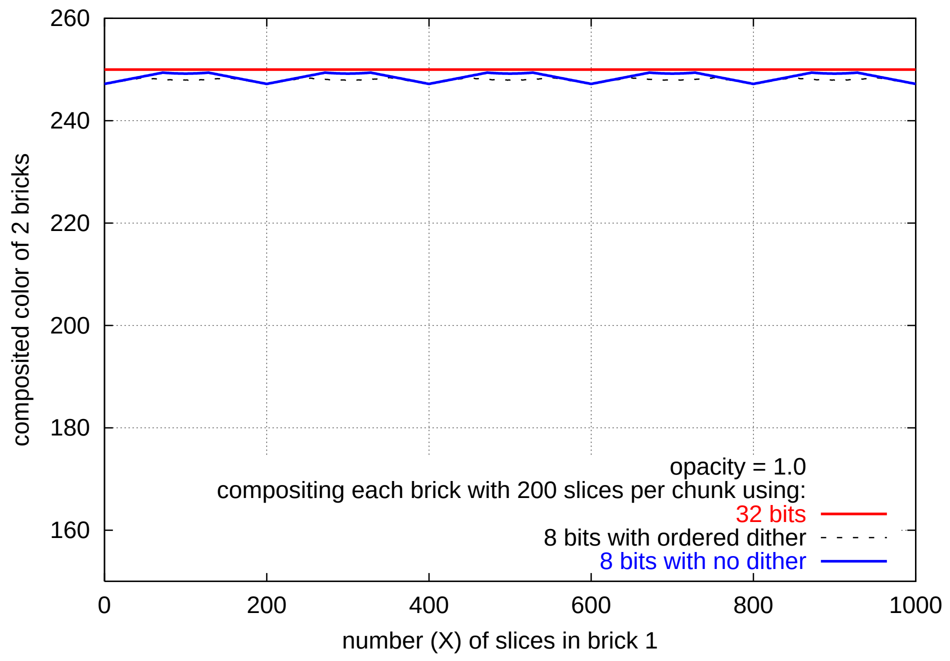

Figure 50: Bricking artifact problem where the image of each brick is created using 5 200-slice chunks with simple dithering with a single slope modifi er of 0.85. Slices have an opacity of 1.0 and color of 255.

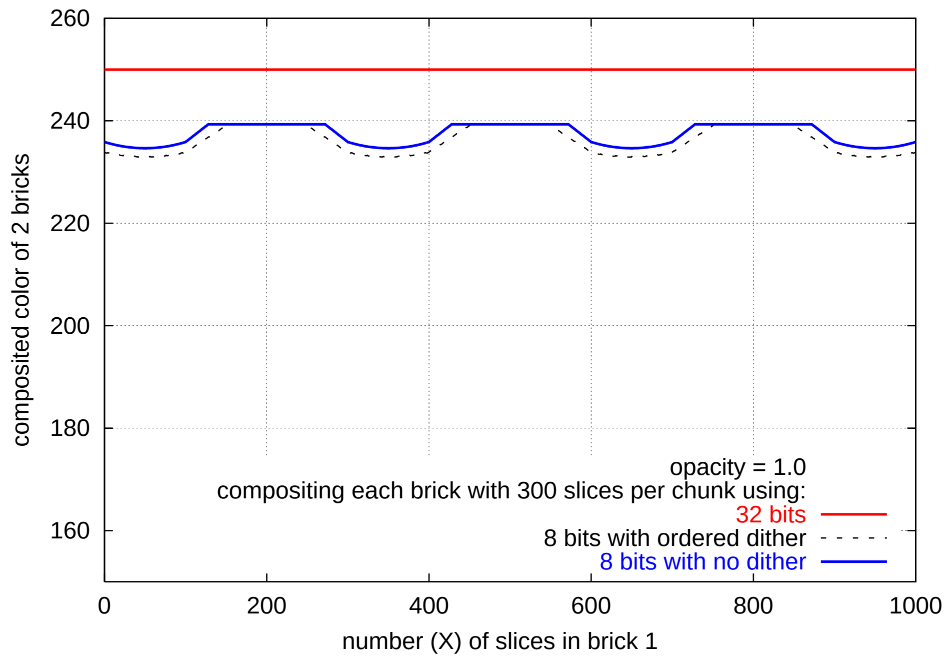

Figure 51: Bricking artifact problem where the image of each brick is created using 300-slice chunks with simple dithering with a single slope modifi er of 0.85. Slices have an opacity of 1.0 and color of 255.

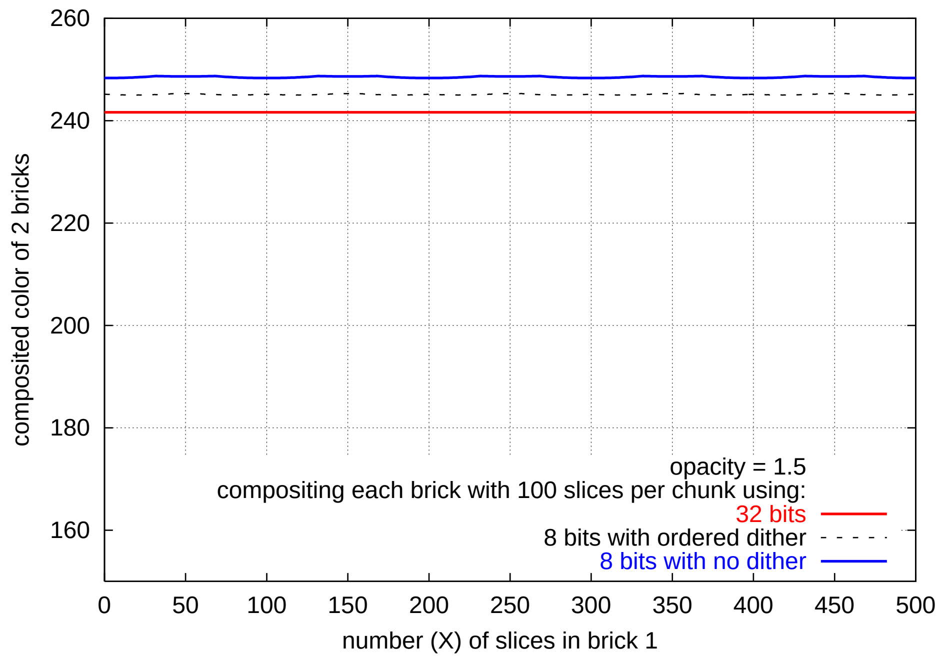

Figure 52: Bricking artifact problem where the image of each brick is created using 5 100-slice chunks with simple dithering with a single slope modifi er of 1.0. Slices have an opacity of 1.5 and color of 255.

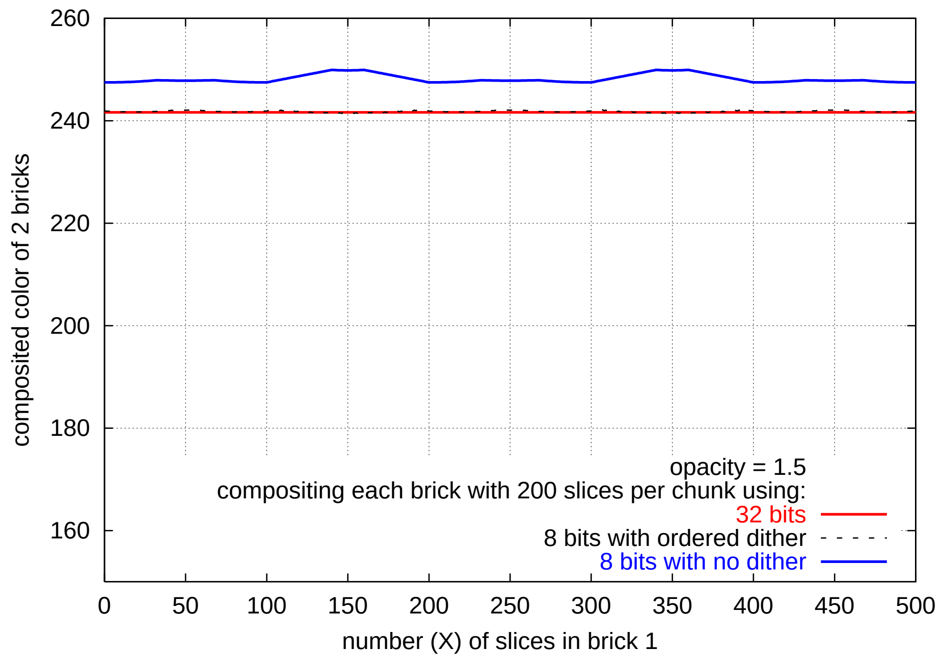

Figure 53: Bricking artifact problem where the image of each brick is created using 200-slice chunks with simple dithering with a single slope modifi er of 1.0. Slices have an opacity of 1.5 and color of 255.

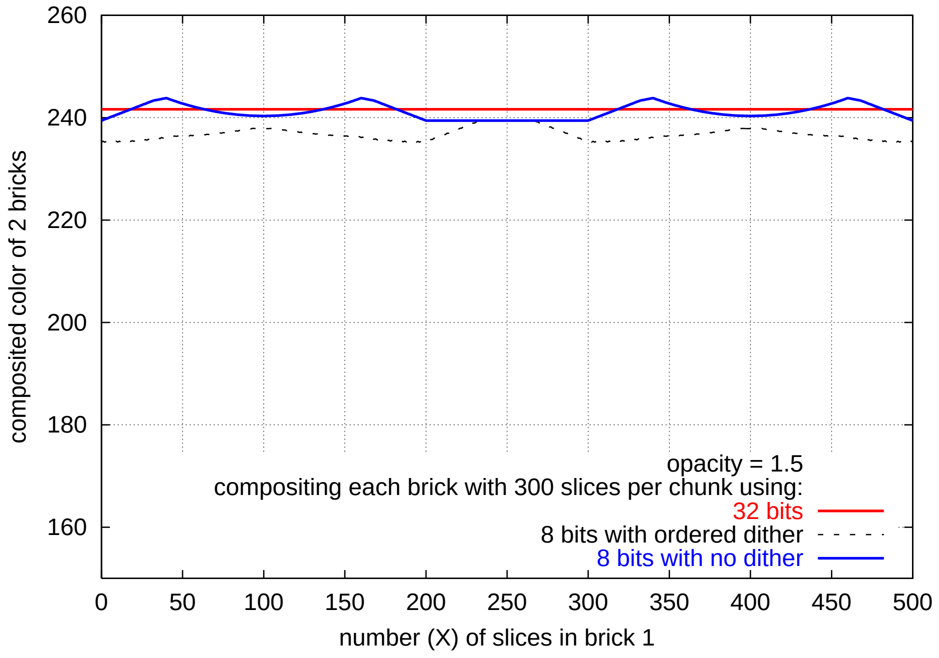

Figure 54: Bricking artifact problem where the image of each brick is created using 300 slice chunks with simple dithering with a single slope modifi er of 1.0. Slices have an opacity of 1.5 and color of 255.

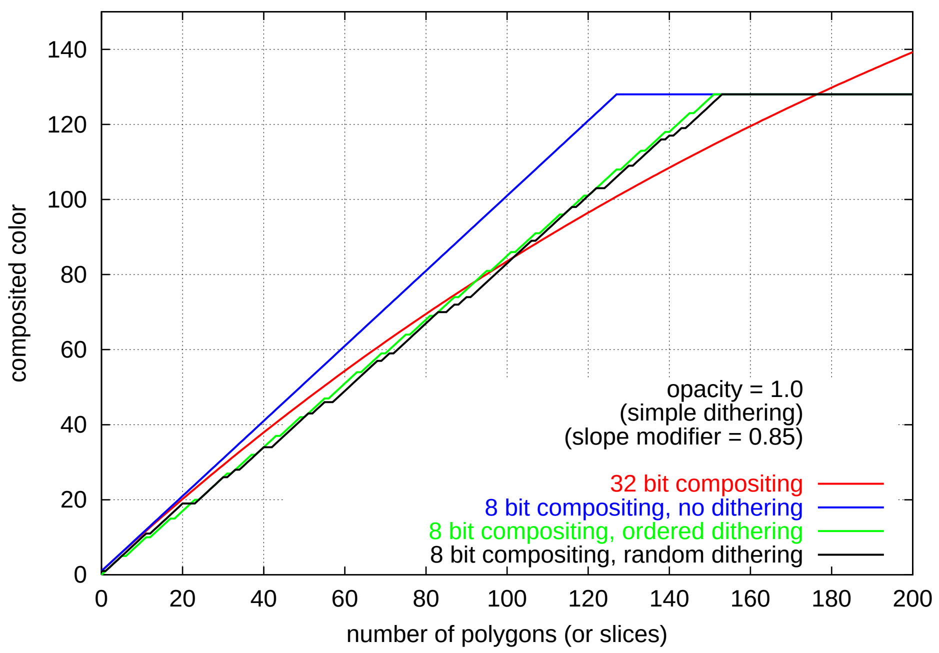

Figure 55: Accumulated pixel values using simple dithering with a single slope modifi er (0.85) for an input of opacity 1.0 and color 255. The slope modifi er results in a nice improvement over the undithered result. Since the input opacity is an integer, unless a slope modifi er is used, our dither technique would not work. Note the plateau occurs around 150 slices for the dithered case. Therefore a chunk should have fewer than 150 slices.

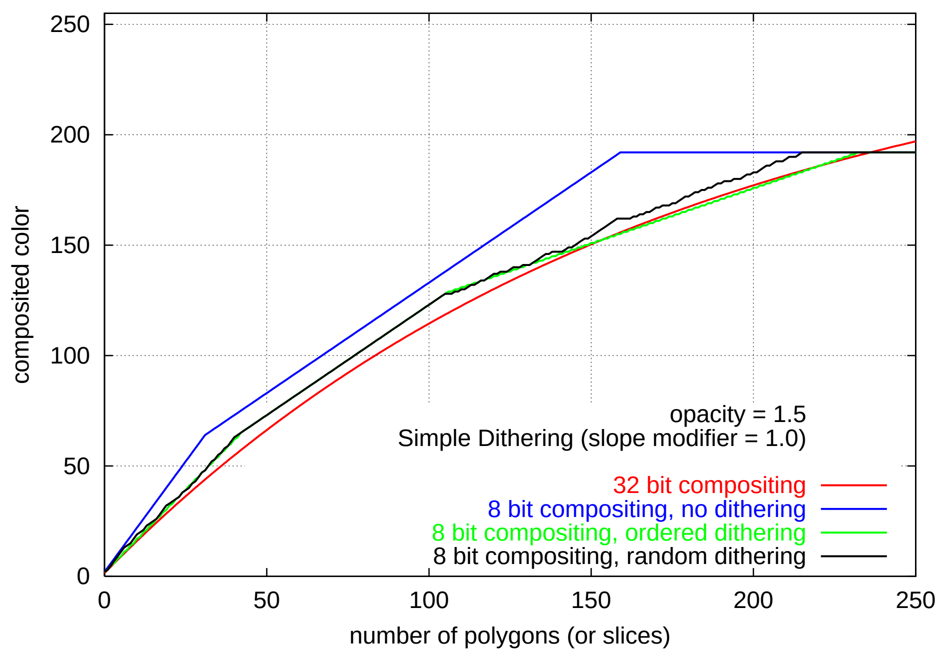

Figure 56: Accumulated pixel values using simple dithering with a single slope modifi er (1.0) for an input of opacity 1.5 and color 255. The plateau occurs around 225 slices for the dithered case. Therefore chunks for this opacity should have 200 slices.

8 OTHER CONSIDERATIONS

8.1 Effect of Dithering on Image Quality & Rendering Performance

To reveal more details of the internal structure of the volume data, the number of slices can be increased. However, this means that the opacity of the slices must be decreased. If the opacity of low-opacity polygons is reduced to the point that $\alpha < 1$, then the dither pattern for α will have $base = 0$. Then some low-opacity polygons will make no contribution to the image. For example, if $\alpha = 0.5$, 50% of the low opacity vertices will be completely transparent and not contribute to the image. So on the one hand, increasing the number of slices increases the detail in the image, but after a certain point, this may be mitigated by the creation of transparent polygons.

This phenomena also leads to a trade off in performance when rendering. Adding more slices, adds more polygons to the graphics pipe, however, if opacity testing is enabled, transparent polygons will be discarded by the pipe and not rasterized. It would be interesting to investigate the question of optimality in this regard. In addition, in the case of textured slices, texture downloads would have to be balanced.

8.2 Front-to-Back Compositing

Front-to-back compositing can be accomplished in hardware if the following rules are implemented in the graphics hardware. Let X_c be the pixel's color buffer and X_a the pixel's opacity buffer. Initially $X_c = 0$ and $X_a = 1$. The rules are:

$$\begin{aligned} &X_c \leftarrow X_a \alpha_{in} c_{in} + X_c \\ &X_a \leftarrow X_a(1 - \alpha_{in}) = X_a T_{in} \end{aligned}$$

For each slice, the color input is: $\alpha_{in} c_{in}$, and the opacity input is: $(1 - \alpha_{in}) = T_{in}$. The buffers X_c and X_a hold accumulated transmittance and color. Assuming the indices in the following increase from front to back, applying the rules to the first three slices we get:

$$\begin{aligned} &X_c = \alpha_1 c_1 + 0 \\ &X_a = T_1 \\ \\ &X_c = T_1 \alpha_2 c_2 + \alpha_1 c_1 \\ &X_a = T_1 T_2 \\ \\ &X_c = T_1 T_2 \alpha_3 c_3 + T_1 \alpha_2 c_2 + \alpha_1 c_1 \\ &X_a = T_1 T_2 T_3 \end{aligned}$$

These rules can be implemented in hardware using the OpenGL separate blending function extension, *EXT_blend_func_separate*. It remains to be seen whether front-to-back compositing will lead to improved compositing accuracy and exactly how dithering will help.

8.3 OpenGL's GL_DITHER

When *GL_DITHER* is enabled, the OpenGL specification allows dithering to occur in the graphics pipeline between the blending operation and the logic operation. This means a graphics card vendor can dither higher precision (post-blended) fragments to a lower precision frame buffer. What we have observed with our NVIDIA GeForce3 card and SGI cards is that when the frame buffer has less than 8 bits per channel, spatial dithering is used. For example, when running in $\{4, 4, 4, 4\}$ mode (as described in Section 2.3), colors that can not be realized exactly in 4 bits, are spatially dithered over alternating scan lines. Enabling *GL_DITHER* for $\{8, 8, 8, 8\}$ mode and specifying colors using floating point values does not cause spatial dithering, rather the colors are rounded and clamped to 8 bit integer values. We ran our tests with graphics hardware for this paper with *GL_DITHER* disabled. We have not seen any evidence of spatial dithering from higher order fragment resolution to 8 bits.

9 CONCLUSION

Hardware 8-bit compositing is really suited to accumulate a few semitransparent surfaces, not 1000's of semitransparent polygons as in volume rendering. Dithering is one way to extend the range of opacity accumulation. In this paper we have shown several techniques for accomplishing this. These techniques include simple dithering, simple dithering with chunking, and dithering with exponential bumping.

Based on our research, ordered dithering introduces less artifacts into the image than random dithering However, random dithering is easier to implement since ordered dither requires a pre-computed table of all possible dither patterns for a given dithering period.

In our work, we apply these dithering techniques to volume rendering slices of a 3D data set of (unstructured) meshed polyhedra — so each slice consists of a set of polygons. However, our dithering procedure is applicable to any hardware compositing technique, such as texture-based volume rendering (e.g. using static or paletted textures, or pixel programs.)

9.1 Spatial Dithering

It is possible to dither spatially in three dimensions. In the work we have presented, we dither in depth (z) only. One criticism of this is artifacts in the image may be created. For example, as the bump value increases, the number of zeros in the dither pattern increase and it is possible that these zeros may cluster spatially, resulting in an artifact. Imagine a continuous layer of blue fog covering an opaque red surface, clusters of zeros in the blue fog could result in red patches within the fog. If spatial dithering in x and y was also used this would be less of a problem. As noted in Section 6 we permute the dither pattern when applicable, however, this only applies if the two or more adjacent entries in the transfer function are the same.

It is possible to dither in x and y and render to larger image using a environment texture map and possibly a pixel program. An area averaging scheme would be used to reduce the image to the desired resolution. The drawback to this approach is that there could be a significant performance hit in rendering due to the increase in the number of pixels, as well as problems dithering at the boundary of the window.

Another approach that is more promising is to jitter, over x and y, the start of the dither pattern in z using multitexturing. This avoid the need for increasing the image resolution. It will be worthwhile to investigate this approach further.

9.2 Some Open Questions

Is there a software formula for compositing two images, where each image is created from a brick using 8 bit compositing, that will yield a result that will be close to what would be the case if all the slices were 8 bit composited in one brick? In other words, can we find a way to composite in software the images from bricks 1 and 2 in Figure 7 so the result is a flat curve whose value is the value at $P0$, i.e. 128?

Let α be the opacity of the incoming fragment. Is there a function f such if $\alpha' = f(\alpha)$ is used instead as the incoming opacity, the compositing result will be closer to exact compositing?

Can pre-integration be introduced into volume renderering using slices? And will this help the problem under discussion?

In order to determine when to increase the bump and when to change the slope modifiers, i.e. when the knee and plateau occurs, it is necessary to know or be able to estimate the accumulated pixel values for regions of low opacity (regions which are accumulating the background haze). However, the pixel values are not available when using graphics hardware unless one does a series of frame buffer read backs while rendering, and this will slow down the rendering process. Further, the accumulation of fog in the frame buffer may not be uniform. A naive approach is to estimate the current pixel values based on the number of slices rendered assuming every slice has low opacity values, or that the majority of slices contribute some low opacity values to at least a subset of the pixels. However there are two problems with this approach: (a) The volume being rendered may be such that each slice does not project to all pixels; see Figure 57. (b) The regions of low-opacity may not be uniformly distributed; see Figure 58. Is there a way to implement exponential bumping and multiple slope modifiers, possibly using current output α values?

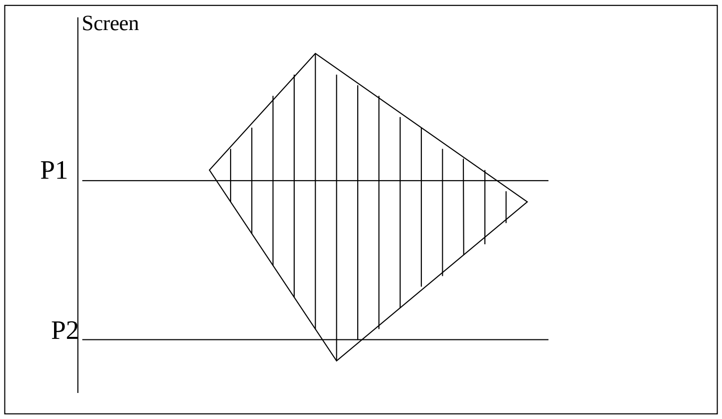

Figure 57: Why estimating pixel values based on number of slices rendered is not a good approach. Note pixel $P1$ has composited 13 slices whereas pixel $P2$ has composited only 1 slices.

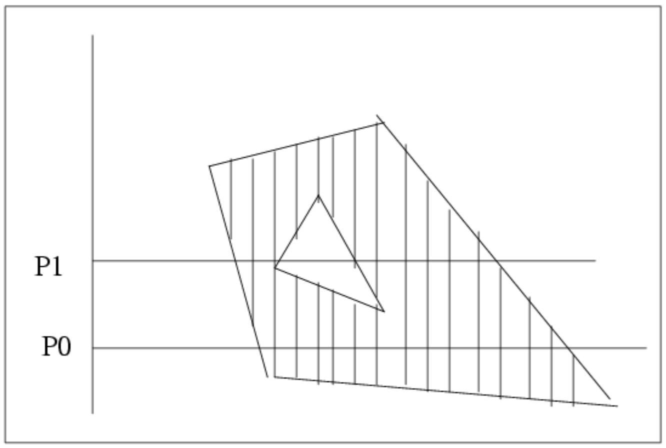

Figure 58: Another reason why estimating pixel values based on number of slices rendered is not a good approach. Note pixel $P1$ composites only 7 slices, while $P0$ composites 14 slices.

ACKNOWLEDGEMENTS

We are grateful to Nelson Max and Claudio Silva for helpful conversations in regard to dithering. This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract number W-7405-Eng-48.

References

[1] J. F. Blinn, "Compositing, Part 1: Theory," *IEEE Computer Graphics and Applications,*, 14(5): 83–87, 1994.

[2] J. F. Blinn, "Compositing, Part 2: Practice," *IEEE Computer Graphics and Applications,*, 14(6): 78–82, 1994.

[3] J. F. Blinn, "Three Wrongs Make a Right," *IEEE Computer Graphics and Applications,*, 15(6): 90–93, 1995.

[4] J. F. Blinn, "Fun With Premultiplied Alpha," *IEEE Computer Graphics and Applications,*, 16(5): 86–89, 1996.

[5] J. F. Blinn, “Fugue for MMX,” *IEEE Computer Graphics and Applications,*, 17(2): 88–93, 1997.

[6] B. Cabral, N. Cam and J. Foran, “Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware,” *ACM Symp. Volume Visualization,* A. Kaufman and W. Krueger, eds., ACM Press, New York, 1994, pp. 91–98.

[7] L. G. Roberts, “Picture Coding Using Pseudo-Random Noise,” *IRE Trans. Info. Theory,* BIT-8:145, Feb. 1962.

[8] J. F. Jarvis, N. Judice, and W. H. Ninke, “A Survey of Techniques for the Display of Continuous Tone Pictures on Bilevel Displays”, *Computer Graphics and Image Processing,* 5(1):13–40, March 1976.

[9] K. Engel, M. Kraus, and T. Ertl, “High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading,” *Eurographics Workshop on Graphics Hardware,* 2001.

[10] J. Kniss, P. McCormick, A. McPherson, J. Ahrens, J. Painter, A. Keahey and C. Hansen, “Interactive Texture-Based Volume Rendering for Large Data Sets,” *IEEE Computer Graphics & Applications,* vol. 21, no. 4, pp. 52–61, 2001.

[11] P.G. Lacroute, “Fast Volume Rendering Using a Shear-Wrap Factorization of the Viewing Transformation,” *PhD Thesis, Tech Report: CSL-TR-95-678, Stanford University,* 1995.

[12] P.G. Lacroute, “Real-Time Volume Rendering on Shared Memory Multiprocessor Using the Shear-Warp Factorization,” *IEEE 1995 Parallel Rendering Symposium,* pp 15–22, 1995.

[13] E. LaMar, B. Hamann, and K. Joy, “Multiresolution Techniques for Interactive Texture-Based Volume Visualization,” *Proc. Visualization 99,* ACM Press, New York, 1999, pp. 355-361.

[14] N. Max, “Optical Models for Direct Volume Rendering,” *IEEE Trans. on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, June 1995.

[15] OpenGL Architecture Review Board, “OpenGL Reference Manual,” D. Shreiner, ed. Addison-Wesley, Inc., Boston, MA.

[16] C. Wynn, “Using P-Buffers for Off-Screen Rendering in OpenGL,” *http://developer.nvidia.com/view.asp?IO=PBuffers_for_OffScreen.*

[17] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, “The VolumePro Real-Time Ray-Casting System,” *ACM Computer Graphics* (Siggraph 99 Proc.), ACM Press, New York, 1999, pp. 251–260.

[18] T. Porter and T. Duff, “Compositing Digital Images,” *ACM Computer Graphics* (Siggraph 84 Proc.), ACM Press, New York, 1984, pp. 253–259.

[19] C. Rezk Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, “Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Mulit-Stage Rasterization,” in *Siggraph/Eurographics Workshop on Graphics Hardware,* 2000 pp. 109-119.

[20] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics*, vol. 24, no. 5, pp 63–70, Nov. 1990.

[21] J. D. Mulder, F. C. A. Groen, and J. J. van Wijk, "Pixel Masks for ScreenDoor Transparency,", *IEEE Visualization '98*, pp 351–358, 1998.

[22] P. Williams, N. Max and C. Stein, "A High Accuracy Volume Renderer for Unstructured Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, 1998, pp. 37–54.

[23] P. L. Williams and N. L. Max, "A Volume Density Optical Model," *Proc. 1992 Workshop Volume Visualization*, Boston, pp. 61–68, Oct. 1992.

[24] P. L. Williams, M. Duchaineau, R. J. Frank, N. Max, "Slice-Based Volume Rendering of Unstructured Data," *Work in Progress,*, 2003.

[25] O. Wilson, A. VanGelder, and J. Wilhelms, "Direct Volume Rendering via 3D Textures," tech. report UCSC-CRL-94-19, Jack Baskin School of Eng., Univ. of California at Santa Cruz, CA, 1994.

[26] C. Wittenbrink, T. Malzbender, M. Goss, "Opacity-Weighted Color Interpolation For Volume Rendering," in *IEEE Symp. on Volume Visualization*, pp 135–142, 1998.

[27] C. Wittenbrink, "Cellfast: Interactive unstructured volume rendering," in *IEEE Conference on Visualization – Late Breaking Hot Topics*, 1999.

[28] R. Yagel, D. Reed, A. Law, P.-W. Shih, and N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *Proc. Volume Visualization Symp.*, IEEE 1996, pp 55–62.