SCI INSTITUTE
TECHNICAL REPORT

Multi-Fragment Effects on the GPU using the *k*-Buffer

Louis Bavoil, Steven P. Callahan, Aaron Lefohn, João L. D. Comba and Cláudio T. Silva

UUSCI-2006-032

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

November 2, 2006

Abstract:

Many interactive rendering algorithms require operations on multiple fragments (i.e., ray intersections) at the same pixel location; however, current Graphics Processing Units (GPUs) capture only a single fragment per pixel. Example effects include transparency, translucency, constructive solid geometry, depth-of-field, direct volume rendering, and isosurface visualization. With current GPUs, programmers implement these effects using multiple passes over the scene geometry, often substantially limiting performance. This paper introduces a generalization of the Z-buffer, called the *k*-buffer, that makes it possible to efficiently implement such algorithms with only a single geometry pass, yet requires only a small, fixed amount of additional memory. The k-buffer uses framebuffer memory as a read-modify-write (RMW) pool of k entries whose use is programmatically defined by a small *k*-buffer program. We present two proposals for adding k-buffer support to future GPUs and demonstrate numerous multiple-fragment, single-pass graphics algorithms running on both a software-simulated k-buffer and a *k*-buffer implemented with current GPUs. The goal of this work is to demonstrate the large number of graphics algorithms that the *k*-buffer enables and that the efficiency is superior to current multipass approaches.

THE UNIVERSITY OF UTAH

Multi-Fragment Effects on the GPU using the k-Buffer

Louis Bavoil [1] Steven P. Callahan [1] Aaron Lefohn [2] João L. D. Comba [3] Cláudio T. Silva [1]
[1] Scientific Computing and Imaging Institute, University of Utah
[2] Neoptica [3] Instituto de Informática, UFRGS

Figure 1: Example effects using the *k*-buffer for multi-fragment processing. The Lucy model (28,055,742 triangles) is rendered with transparency on the left and with translucency on the right. These effects captured 8 fragments per pixel in a single geometry pass and were rendered with a current hardware implementation that avoids read-modify-write hazards. With our proposed extension to hardware, these hazards can be automatically avoided and performance improved.

Abstract

Many interactive rendering algorithms require operations on multiple fragments (*i.e.*, ray intersections) at the same pixel location; however, current Graphics Processing Units (GPUs) capture only a single fragment per pixel. Example effects include transparency, translucency, constructive solid geometry, depth-of-field, direct volume rendering, and isosurface visualization. With current GPUs, programmers implement these effects using multiple passes over the scene geometry, often substantially limiting performance. This paper introduces a generalization of the Z-buffer, called the *k*-buffer, that makes it possible to efficiently implement such algorithms with only a single geometry pass, yet requires only a small, fixed amount of additional memory. The *k*-buffer uses framebuffer memory as a read-modify-write (RMW) pool of *k* entries whose use is programmatically defined by a small *k*-buffer program. We present two proposals for adding *k*-buffer support to future GPUs and demonstrate numerous multiple-fragment, single-pass graphics algorithms running on both a software-simulated *k*-buffer and a *k*-buffer implemented with current GPUs. The goal of this work is to demonstrate the large number of graphics algorithms that the *k*-buffer enables and that the efficiency is superior to current multi-pass approaches.

Keywords: fragment processing, graphics hardware, visibility ordering, blending, volume rendering, transparency, CSG

1 Introduction

Raster-based graphics algorithms simulate many effects by operating on multiple fragments in the same pixel. Several existing algorithms keep all the fragments for each pixel [Carpenter 1984; Mark and Proudfoot 2001; Wittenbrink 2001] so that they can be sorted and composited in either front-to-back or back-to-front order for transparency. However, the unbounded memory requirements for these types of algorithms is a limiting factor for practical applications.

Recent work by Callahan *et al.* [2005] proposed the *k*-buffer—a fixed size buffer of fragments per pixel that is maintained in GPU memory. This *k*-buffer was shown to be effective for sorting and compositing fragments in the special case of direct volume rendering with current graphics hardware. However, many applications other than direct volume rendering require access to multiple fragments simultaneously. Here, we generalize the definition of the *k*-buffer to be a pool of fragments per pixel that can be read, modified, and written at the fragment level. The benefit of the *k*-buffer is that it allows fragments to be compared, ordered, blended, and discarded in a streaming manner. Thus, many effects that normally require multiple passes over the scene geometry can instead be streamed through the *k*-buffer in one pass. Note that in this paper, we refer to a pass as a rendering pass over the scene geometry, not image processing passes, such as deferred shading, which render a screen-aligned quadrilateral with a pixel shader. For large or complex scenes, the geometry passes are much more expensive than the image processing passes.

Whereas a traditional Z-buffer-based framebuffer saves fragment results for a single depth per pixel (the front-most fragment), the *k*-buffer can save up to *k* fragments with only a small increase in memory requirements. By giving access to multiple ray intersections along a viewing ray, the additional information in a *k*-buffer provides algorithms with a more global view of the scene, in turn opening up a number of new algorithmic possibilities for raster graphics. For example, the first *k* fragments per pixel in front-to-back order can be stored in a *k*-buffer, effectively performing depth peeling in a single pass. Since the *k*-buffer supports programmable RMW operations, it can also be used to implement a Z-Buffer, a stencil buffer, or arbitrary blending.

The *k*-buffer can be implemented in current hardware using read-modify-write (RMW) operations on textures at the fragment level. Currently, this feature is allowed in hardware, though the results are undefined. Because of the highly parallel nature of fragment processing on the GPU, there is no guarantee that artifacts will not

appear from overlapping geometry in screen space. We address this issue by describing solutions that avoid the race conditions that can occur with overlapping fragments. This current implementation is used for validating our k-buffer applications, generating images, and producing experimental results of the data structure. However, with a few modifications to the current GPU pipeline, we believe that full k-buffer support is possible. We propose two such modifications that would avoid RMW hazards in future hardware.

We believe that the k-buffer has important implications for interactive graphics and visualization because of the number of applications that it enables or simplifies. The contributions of this paper include:

- we discuss a general definition of the k-buffer and its implementation in current hardware;
- we suggest modifications to the current hardware pipeline to enable full hardware support of a k-buffer on future GPUs;
- we outline efficient single-pass algorithms using the k-buffer that normally require expensive, multi-pass algorithms using a traditional Z-buffer; and
- we provide experimental results of our applications of the k-buffer on current GPUs and compare them to multi-pass approaches.

The rest of the paper is outlined as follows. Section 2 provides an overview of related research. Section 3 describes the general k-buffer framework including future and current hardware implementations. Section 4 describes specific algorithms for single-pass effects facilitated by the k-buffer. We provide experimental results of the k-buffer in Section 5. Finally, we provide a general discussion in Section 6 and conclude our paper in Section 7.

2 Related Work

Single-Pass Approaches. There are many relevant publications on storing and processing multiple fragments per pixel. The traditional image-based algorithm for fragment sorting is the Z-Buffer [Catmull 1974]. It is a streaming algorithm—for every pixel, the fragment with lowest (or greatest) depth is kept and the others are discarded. The A-Buffer [Carpenter 1984] is an extension of the Z-Buffer which stores all the fragments rasterized per pixel in a list, which is then sorted according to depth. Fragments that belong to the same surface and that have very close depth values are merged. This algorithm is not suitable to current graphics hardware because of its unbounded memory per pixel. The R-Buffer [Wittenbrink 2001] is a variation of the A-Buffer. All the fragments for the scene are stored in a single FIFO queue in memory. Although the R-Buffer was designed for hardware implementation, storing a large number of transparent fragments is not feasible in practice. Another streaming approach for fragment processing is the Z^3 algorithm [Jouppi and Chang 1999]. Z^3 uses a fixed number of fragments per pixel and thus requires less memory than the A-Buffer or R-Buffer. When the maximum number of fragments per pixel is reached, it selects the two closest fragments and merges them together using a set of heuristics based on pixel coverage. Of these streaming methods, only the Z-Buffer has an actual hardware implementation in current GPUs. Recently, Eisemann and Decoret [2006] showed that an approximate partitioning of the scene can be performed in a single pass on the GPU by voxelizing the scene. Although this technique is very efficient for volumetric effects such as transmittance shadow mapping, it does not allow effects which require the exact locations of the fragments, such as transparency. Of these algorithms, the k-buffer is most similar to the Z^3 architecture because it stores a fixed number of fragments per pixel. The k-buffer can be seen as a generalization of Z^3 where the storage and insertion of the fragments has been made programmable.

Multi-Pass Approaches. Due to memory resources on graphics hardware, multi-pass rendering is often required to achieve many effects. The F-Buffer [Mark and Proudfoot 2001] is very similar to the R-Buffer, except that it does not sort the fragments. The F-Buffer requires semi-transparent surfaces to be rendered in depth order, although it may be possible to sort an F-Buffer using a bitonic sort. Implementations of the F-Buffer which require rendering the whole geometry for every pass are available on ATI's graphics hardware [Houston et al. 2005]. The original depth peeling algorithm by Mammen [1984] proposes a solution for sorting fragments by *peeling* the layers in depth order in separate passes. A hardware implementation was more recently described by Everitt [2001]. Depth peeling has been used for rendering order-independent transparency [Everitt 2001; Wexler et al. 2005; Luft and Deussen 2006], volume rendering [Nagy and Klein 2003; Bernardon et al. 2006], global illumination [Hachisuka 2005; Mendez et al. 2006], and multi-layer shadow maps [Woo 1992; Bavoil et al. 2006]. Kelley *et al.* [1994] proposed a hybrid solution that stores four RGBAZ fragments per pixel, sorted front-to-back, and handles overflow with multiple passes. In each pass, the four layers are composited into a single layer and the three remaining layers are used to capture the next fragments. A recent approach similar to depth peeling is the Vis-Sort algorithm [Govindaraju et al. 2005] which sorts 3D primitives using occlusion queries on the GPU, assuming there are no visibility cycles and no intersecting primitives. With Vis-Sort, the number of passes required to composite transparent fragments is equal to the depth complexity of the scene. The k-buffer simplifies multi-pass approaches by allowing k fragments to be operated on in a single pass. Since it operates in image-space, it also avoid problems with visibility cycles and intersecting primitives.

3 The k-Buffer

The k-buffer is a generalization of the traditional Z-buffer-based framebuffer. Instead of restricting framebuffers to a single depth value, a single stencil value, and n color values, the k-buffer uses framebuffer memory as a RMW pool of k entries whose use is programmatically defined by k-buffer operations. In essence, the recent addition of multiple render targets (MRTs) to GPUs already allows multiple fragments to be stored, albeit in textures. We take this a step further and suggest that the programmable combination of these fragments is as important as their storage to achieve many advanced effects in a single pass. The general structure of the k-buffer algorithms for each fragment f that is rasterized is as follows:

1. **Read** the k-buffer elements for this pixel from memory. These values, along with the incoming fragment f, are now available.
2. **Modify** the k-buffer elements using f.
3. **Write** the k-buffer elements back to memory and discard f.

Many effects can be performed using different types of modify operations on the k-buffer values. Generally, these modify operations fall into two types. The first type is to use the k-buffer to accumulate up to k fragments for a post-processing pass such as deferred shading. Examples of this type of algorithm are depth-peeling and depth-partitioning, which can be used to perform effects such as transparency, translucency, midpoint shadow mapping, constructive solid geometry, and depth-of-field. The second type is to use the k-buffer as a fragment stream processor and programmable blender. An example of this type of algorithm is fragment ordering, which can be used to perform isosurfacing, direct volume rendering, and transparency of geometry with large depth complexity.

The first type of algorithm generally requires a fixed number of fragments to perform the desired effect. The k-buffer is used as temporary storage of the most significant fragments to be used in a post-processing pass. These fragments can be rasterized in any particular order with no change in the final result. However, the second type of algorithm generally needs to consider all fragments to achieve the desired effect. In this case, when a new fragment is inserted, a blending operation is performed and a fragment is discarded. Thus, the rasterization order of the fragments will affect the output. The k-buffer is capable of sorting a k-nearly sorted sequence (k-NSS) of fragments. Given a sequence S of fragment depths, S is a k-NSS if no depth in S is more than k positions out of place. Therefore, when fragment ordering is required, the geometry needs to be rasterized in at least a partial order so that the k-buffer can complete the ordering and blend the results all in one pass (see [Callahan et al. 2005] for a more formal definition of a k-NSS). The partial ordering of the geometry occurs in object-space prior to rasterization. The extent of the ordering that is required is dependent both on the depth complexity and the available k size.

3.1 Future Hardware Implementation

Complete k-buffer support in hardware would enable fast single-pass effects without RMW hazards that may occur with our current implementation. Although the specifics of hardware implementations are not publicly available, we propose two possible high-level solutions based upon available information about the current hardware pipeline. In both solutions, we implement the k-buffer as a set of floating-point renderable buffers, and write to these buffers using Multiple Render Targets (MRTs). The main difference between the two solutions is where we execute the k-buffer operations (read, modify, and write). The first solution involves changes at the fragment scheduling stage of the pipeline and the second solution involves changes at the blending stage of the pipeline. Figure 2 shows a simplified version of the GPU pipeline with annotations that specify the areas that require modification for our hardware proposals.

3.1.1 Fragment Scheduling

In this solution, the k-buffer operations are implemented in fragment programs, and the reads are performed using the connection between the memory partition and the texture cache. However, this approach has issues with RMW hazards because current GPUs process multiple fragments at the same time per fragment pipeline, with multiple parallel pipelines. When an output of a fragment is computed, it is not immediately written to memory, but written to a reorder buffer which reorders the fragments in the order in which the primitives have been rasterized. This is necessary for the correctness of RMW raster ops such as blending and stencil buffer. This asynchronous write to memory means that if two overlapping fragments are processed concurrently, and they modify a value of the k-buffer for the same pixel, one fragment will read an obsolete value, and overwrite the value of the other fragment with an incorrect value. A solution to this issue—like for CPUs—would be to add dynamic scheduling of fragments to GPUs to detect and avoid pipeline hazards.

The Unified Render Architecture proposed by ATI in the next generation GPU architecture [Doggett 2005] aims for better load balancing on vertex and fragment shaders by considering them as a single shader unit that is managed by a thread arbiter (or the fragment dispatcher) . The thread arbiter controls the data being passed to the shader units, and finds the best possible way to ensure that all of the shader units are busy. If the thread arbiter can be configured, or even programmed in the future, it would be possible to divide fragments into non-overlapping groups that are processed by

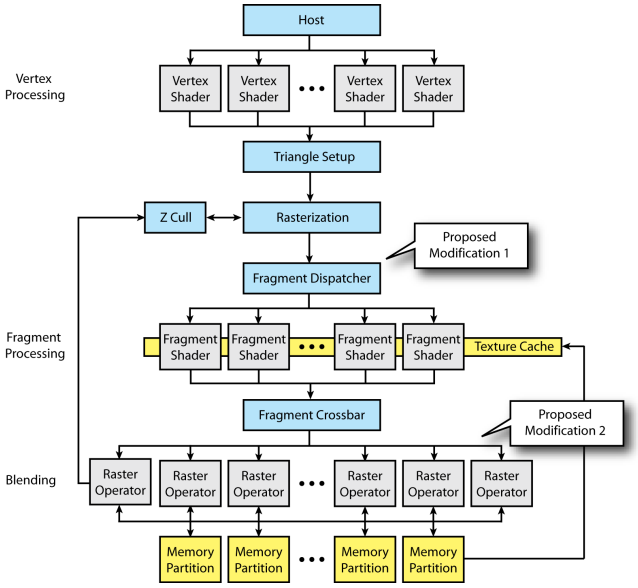


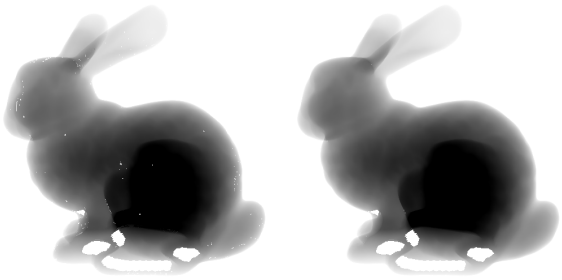
Figure 2: The current GPU pipeline showing where our proposed modifications will occur. Figure adapted from [Kilgariff and Fernando 2005].

different shader units. Similarly to early-Z culling [Kilgariff and Fernando 2005], the hardware could keep a scoreboard [Thornton 1964] in the form of a coarse image of what fragments are currently in the pipeline. For a given incoming fragment packet (fragments are currently packed to perform derivative operations), the scoreboard would be checked for overlaps with a fragment already in the pipeline. If an overlap is detected, the packet could be inserted in a buffer, and the next packet coming from the rasterizer could be tested. An overflow of this temporary buffer would result in a stall until a pipeline becomes available.

The main advantage to k-buffer support at the rasterization stage is that it leaves the current pipeline relatively unchanged. Another advantage is that k-buffer programs can use all the features of fragment shaders, including texture accesses (*e.g.*, for lookup tables). Note that the k-buffer access does not need to be a texture access. It may be more efficient to implement it as varying arguments like texture coordinates. In any case, k-buffer programs could mix texture accesses with k-buffer accesses. However, there are several disadvantages to this proposed solution. First, by modifying depth in fragment programs, early-Z tests are invalidated. Since the k-buffer generally requires all the fragments anyway, this is not a major issue. Second, the fragment scheduler may adversely affect performance by reducing the parallelism during fragment processing. Finally, full-screen antialiasing presents some challenges because fragment shaders operate on pixel fragments while multisample antialiased blending operations occur on multiple samples per pixel [Blythe 2006]. To support antialiasing with the k-buffer would likely require supporting the more costly supersample antialiasing rather than the more efficient multisample antialiasing.

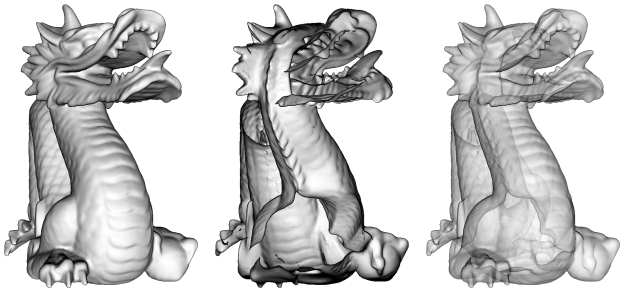
3.1.2 Programmable Blending

A more promising approach is to implement the k-buffer by allowing programmable blending. Currently, the only RMW operations on colors in the graphics pipeline are fixed-function per-pixel operations. The k-buffer can be supported by extending RMW capabilities using blending programs similar to fragment or vertex programs. Specifying different k-buffer applications could then occur with the use of a programmable blender that take as input the re-



(a) (b)

Figure 3: Artifacts that appear in our current implementation due to hazards. (a) With the original mesh layout. (b) Sorting the triangles with a depth sort by centroid.



(a) (b) (c)

Figure 4: Depth peeling two layers from the dragon dataset. (a) First layer, (b) Second layer, (c) Transparency using 4 layers.

sults of the fragment shader and outputs the values in the *k*-buffer for the pixel. Programmable blending has been discussed as a possible hardware extension in the future [Blythe 2006].

To demonstrate that this model conceptually fits into the current pipeline, we extended Mesa 6.5 with *k*-buffer support for programmable blending. We modified the OSMesa driver, which is a purely software implementation of Mesa. The following changes to the existing pipeline were made. In the software rasterizer, we added a *k*-buffer mode which changes the behavior of MRTs. When in this mode, fragments leaving the fragment program are passed to a programmable blender. A programmable blender is a specialized fragment program which takes as input the current pixel in the framebuffer, and the current *k*-buffer fragments for this pixel, passing these fragments as 4-float varying arguments. Programmable blenders are written in ARBfp1 assembly and specified using a new target `GL_KBUFFER_PROGRAM_MESA` in place of the usual `GL_FRAGMENT_PROGRAM_ARB`. This *k*-buffer program is then optionally executed instead of blending. Figure 10c was rendered using this implementation.

Implementing *k*-buffers with programmable blending has a number of benefits over implementing them in the fragment program stage. First, it does not require texture (random) memory access, which means the only memory reads are pure stream accesses from the incoming fragment and the *k*-buffer. Second, it does not require scheduling so it would not affect the parallelization of the fragment pipeline. Third, current caching strategies for pixel tiles are critical to GPU performance [Hasselgren and Akenine-Möller 2006] and would still be applicable. Fourth, no cache coherency would be required between the pixel tile caches and the texture caches. Finally, multisample antialiasing would still be possible, given that the *k*-buffer operates directly on subsamples rather than fragments. One consideration of this approach is that many compression algorithms are hardcoded for the fixed semantics of each component of the framebuffer [Hasselgren and Akenine-Möller 2006]. By generalizing the framebuffer, the hardware may no longer know which chunks of memory can be optimized for depth, stencil, color, *etc.*

3.2 Current Hardware Implementation

We created an experimental implementation of the *k*-buffer using current hardware to test *k*-buffer applications and demonstrate the flexibility of the framework. All of the effects and results shown in this paper were created using this implementation, unless specified otherwise. Our experimental *k*-buffer is implemented in OpenGL as a set of textures that can be read and written to in fragment programs using MRTs, as described in Section 3.1.1. Since current hardware does not handle RMW pipeline hazards, artifacts may appear. To perform off-screen rendering into the MRTs with OpenGL, we use a Framebuffer Object (FBO), which is a collection of logical buffers such as color, depth, or stencil. Currently, up to four color buffers can be attached to an FBO and used as MRTs. Algorithms operating on the *k*-buffer are currently implemented as fragment programs on FP32 textures with *Z*-culling disabled.

If the desired application requires streaming with programmable blending, one of the color attachments may act as an off-screen framebuffer, while the other three contain *k*-buffer entries. Otherwise, all four available color attachments may be used to store the *k*-buffer entries. These entries can be single values (*e.g.*, depth), or sets of values (*e.g.*, depth, scalar, color, *etc.*) depending on the application. Thus, the number of values per entry directly effects the size of *k* available for the application. With four MRTs, it is possible to store up to 16 fragment attributes using RGBA textures. The precision of these color attachments is application specific, thus by quantizing the values, it is possible to pack additional *k*-buffer entries into the color attachments. To minimize the number of attributes stored with each *k*-entry, we would ideally only need to store one depth value per entry. The world-space position can be reconstructed from the depth (either the clip-space depth or the distance to the eye). From the positions, normals can be estimated using centered differences so deferred shading can be performed. Using lookup table IDs can also be useful to reduce the number of attributes stored in the *k*-buffer.

Our experimental *k*-buffer reads and writes from the same textures in each fragment operation. This is available in the current OpenGL API even though the results are undefined. In practice, this may result in RMW hazards due to the parallel nature of GPU architectures (for example, see Figure 3). To reduce these hazards, we have developed two heuristics applied to scene geometry prior to rasterization to avoid screen-space overlaps. Depending on the application and scene, these heuristics may be necessary for correct images using current hardware. Our first heuristic is to sort the primitives by their centroid depth. This effectively layers the geometry in screen space, which reduces the likelihood of overlapping fragments in the pipeline simultaneously. For algorithms that require complete sorting of the fragments, this object-space sorting is already required, thus the hazards are inherently reduced without additional penalty. Our second heuristic is to batch triangles and flush the graphics pipeline after each batch by rendering a full-screen quadrilateral with a `GL_FALSE` color mask. For performance reasons, we use a simple algorithm to create the batches—triangles are added to the current batch in order, until a maximum batch size is reached. Though these heuristics adversely affect performance by reducing the caching on the GPU and stalling the GPU pipelines, they demonstrate the ability to overcome the RMW hazards that occur with the current hardware implementation.

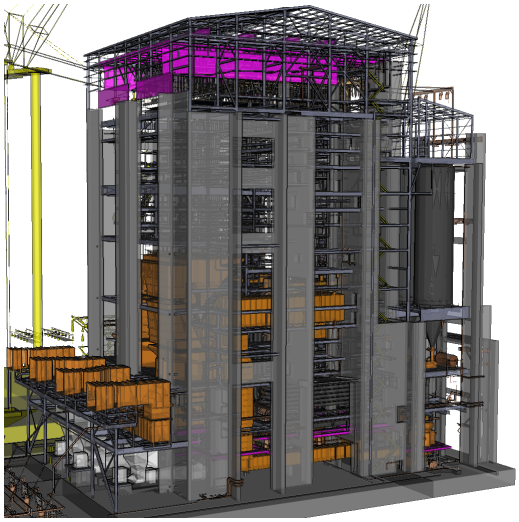


Figure 5: Single-pass transparency by depth peeling with 4 layers on a subset of the UNC Powerplant dataset (1,946,000 triangles). All surfaces are semi-transparent with $\alpha = 0.5$.

4 Applications

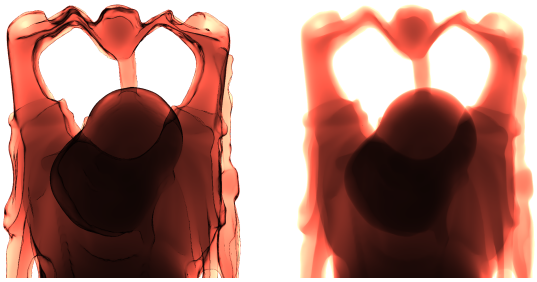
Many effects that normally require multiple geometry passes can be simplified using the k-buffer by simply changing the modify portion (step 2) of the k-buffer algorithm. In this section, we detail a few of the many possible applications of the k-buffer.

4.1 Depth Peeling Applications

Depth peeling captures all the depth layers by stripping the visible layers of fragments in multiple peeling passes using Z-buffer tests. The k-buffer makes it possible to perform up to k Z-buffer tests in a single geometry pass and therefore capture the first k fragments along a viewing ray. Effects that use depth peeling include transparency, translucency, constructive solid geometry, midpoint shadow mapping, and volume rendering. A k-buffer can be used to perform single-pass depth peeling by storing depth-sorted fragments (see Figure 4). The depth values of the k-buffer entries are initialized with the largest possible depth value. Upon rasterization, each fragment is inserted into the k-buffer in increasing depth order. This insertion sort tests the depth of the incoming fragment with the nearest depth in the sorted k-buffer. If the incoming depth is less than the nearest depth, it shifts all the elements in the k-buffer, and sets the nearest fragment to be the incoming fragment. Otherwise, the second nearest fragment is tested, and so on.

Transparency. In real-time applications, transparency is usually simulated by compositing fragments in depth order, ignoring refraction at material interfaces. A common way to do this is to perform depth peeling to generate fragments in depth order and composite them into the framebuffer using α-blending [Everitt 2001]. For simplicity, we use a uniform α, but a non-uniform α can also be used. Figures 1 and 5 show the results of single-pass transparency rendering using depth peeling with the k-buffer.

Translucency. Another application of depth peeling with the k-buffer is rendering translucency effects. We implemented a translucency algorithm that accounts only for absorption and does not simulate any scattering effects [NVIDIA 2005]. Assuming a bright ambient light, and ignoring reflection, translucency can be rendered by computing an ambient term I_a using Beer-Lambert's law: $I_a = exp(-\sigma_t l)$ where σ_t is the absorption coefficient, and l is the distance that the light travels through the material. The thickness l can be computed in one pass without the k-buffer on current hardware by summing the depths of the front and back faces separately using additive blending and taking the difference of the sums [James and Green 2004]. This approach is suitable for computing thickness, but makes more advanced blending difficult. As an example, light rays are attenuated based on their incidence angle with the surface (Fresnel's effect). For this attenuation, it is common to use Schlick's approximation to the Fresnel's transmittance: $F_t = 1-(1-cos(\theta))^5$. These terms can be computed at each fragment and multiplied together using blending in a separate geometry pass. With the k-buffer, a transmitted intensity can be computed in a single geometry pass, taking into account both thickness and Fresnel's terms. Figure 6 shows the result of depth peeling using the k-buffer, with and without Fresnel's effect.



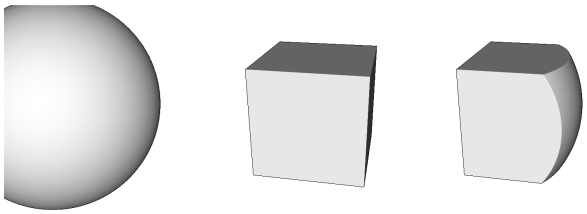
(a) (b)

Figure 6: Translucency effect on the Happy Buddha (1,087,000 triangles) by depth peeling from the eye with a k-buffer of 8 layers. (a) Attenuating with Fresnel's terms for every fragment and (b) without Fresnel's terms.

Constructive Solid Geometry. Many complex shapes can be easily represented using constructive solid geometry (CSG). CSG operations on arbitrary objects can be represented as a boolean function, which is true for a point inside the new object and false otherwise. To render a CSG object with a boolean function, Kelley *et al.* [1994] use a front-to-back depth ordering and encode the CSG function using a lookup table. With current programmable pixel shaders, the CSG function can be evaluated efficiently without the need of lookup tables. To perform CSG, we use our single-pass depth peeling to capture all the fragments of the scene. In the k-buffer, we store a linear depth and an object ID for each fragment. In a post-processing pass, the fragments are traversed from front-to-back for each pixel. For each valid fragment, the state of the object (inside or outside) is updated. The first time the boolean function returns true, the depth of the fragment is stored and used in a deferred shading pass to construct normals and shade the object using centered differencing [Livnat and Tricoche 2004]. Figure 7 shows the results of a CSG operation using the k-buffer.

4.2 Depth Partitioning Applications

Similarly to depth peeling, the k-buffer can also be used to route partition fragments into multiple depth ranges using a single rendering pass. Rather than storing the first k fragment along a ray like in depth peeling, depth partitioning keeps one fragment in each depth partition. One effect that can be achieved with depth partitioning is depth-of-field by separating foreground fragments from the background fragments, and keeping the nearest fragment to the eye in each partition. Up to k depth ranges can be captured in a single pass using the k-buffer. This is done by comparing the depth of an incoming visible fragment with constants that define the ranges and placing the fragments in their correct range location in the k-



(a) (b) (c)

Figure 7: Example of a CSG operation using the k-buffer. (a) A = sphere, (b) B = cube, (c) $A \cap B$.

buffer. Other effects based on depth partitioning, such as transmittance shadow maps, soft shadows, and refraction have been demonstrated by [Eisemann and Décoret 2006].

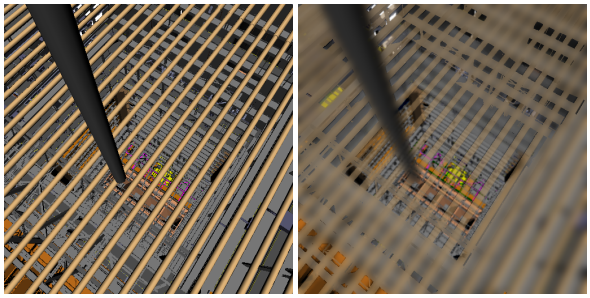
Depth-of-Field. Given depth partitions of the visible fragments, depth-of-field can be performed on the GPU by applying a depth-based blur that is weighted by the distance from the focal plane (i.e., a spatially-varying blur based on the circle-of-confusion). The drawback of this approach is that fragments from the background bleed onto the foreground and a sharp background cannot be seen behind blurry, transparent foreground objects. These problems are largely ameliorated by partitioning the scene into foreground, midground, and background depth layers [NVIDIA 2005; Lefohn 2006], blurring each layer separately, and compositing them together. This approach requires rendering the entire scene three times with different near and far planes. With the k-buffer, we can route the foreground, midground, and background fragments into separate buffers based on their depth values. These buffers can then be blurred separately and composited into the final image. Figure 8 shows an example of depth-of-field using the k-buffer.

4.3 Sorting and Blending Applications

For effects such as transparency or volume rendering that require visibility ordering with arbitrary depth complexity, depth peeling a fixed number of layers may not be sufficient to render the effect properly. In case of overflow of the k-buffer, one can either merge fragments in the k-buffer [Carpenter 1984; Jouppi and Chang 1999], or blend one fragment with the current color buffer [Callahan et al. 2005]. This later blending approach assumes that the fragments are generated in a front-to-back, nearly-sorted order.

To perform programmable blending with the k-buffer, a RMW framebuffer is required for compositing. For every fragment that is rasterized, the following steps take place. First, the k-buffer entries are read and compared along with the incoming fragment to find the two fragments closest to the eye (f_1 and f_2). A value such as color or depth is then computed using f_1, f_2, and the distance between them. This value is then composited into the framebuffer. Finally, the f_2 fragment along with the unused fragments are written back into the k-buffer and the f_1 fragment is discarded.

To ensure that the fragments are rasterized in a nearly-sorted visibility order, some object-space sorting is usually required. We perform the object-space sorting using a Least Significant Digit Radix Sort [Sedgewick 1998], which operates in linear time on floating point values with a simple float-to-int conversion [Callahan et al. 2005]. The k-buffer finalizes the order in image-space by selecting the fragments closest to the eye from the k stored fragments. For scenes with many objects of low depth complexity, this object-space ordering can be accomplished by simply rendering these objects in depth order. For scenes with complex objects, the renderable primitives may require sorting. A less conservative approach can be used in most scenes by sorting clusters of triangles. The depth of the centroid of the primitive or set of primitives is used to perform the ordering after every view transformation that effects the depth order.



(a) (b)

Figure 8: Single-pass depth-range partitioning. Partitioning the fragments into foreground and background is necessary to render a sharp background underneath a blurry foreground. (a) Without depth-of-field (pinhole camera). (b) With foreground depth-of-field. The foreground, midground, and background are rendered into three separate images using an RGBZ k-buffer.

Depending on the k-buffer size and the quality of the primitives, this approximate order results in a correct visibility ordering for the fragments. Scenes with geometry that overlap in depth and have high variance in size or aspect ratio can result in artifacts due to incorrect visibility order with a small k. There are several ways to avoid these artifacts. First, using a larger k-buffer will lower the chances of incorrect fragment ordering. Second, subdividing the triangles such that the centroids are better approximations of the depth of the fragments will improve the order and reduce inaccuracies. This is done by prioritizing the triangles based on their aspect ratios (*i.e.*, how long and skinny they are), which is directly related to the error they will cause when centroid sorting. Then, the triangles are recursively subdivided until a triangle budget is reached. Subdivision schemes on a single triangle, such as Loop subdivision [Loop 1987], can propagate bad aspect ratios to subdivided triangles. Instead, we use a new subdivision scheme that reduces the total area of triangles with bad aspect ratios, while maintaining the original vertex locations of mesh. Given a triangle, our scheme first finds the longest edge e_{max} with length l_{max} and the shortest edge e_{min} with length l_{min}. A new vertex is then added on e_{max} at a distance $min(l_{min}, l_{max}/2)$ from the vertex connecting e_{max} and e_{min}. The resulting subdivided triangles are recursively subdivided as needed. Figure 9 demonstrates this subdivision on one triangle.

Isosurface Rendering. A texture-based approach for isosurface rendering of tetrahedral meshes was proposed by [Weiler et al. 2003] which projects the tetrahedra in screen-space to triangles. The method uses a texture lookup to determine if interpolated texture coordinates correspond to an iso-value or not. Using the k-buffer, similar isosurface extraction can be performed directly on the triangles that compose the mesh. Our technique extracts the isosurface without the need to update a texture for each iso-value, and works with an arbitrary number of isosurfaces. For each fragment,

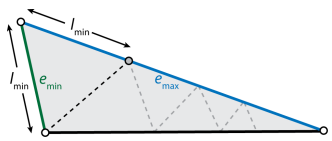
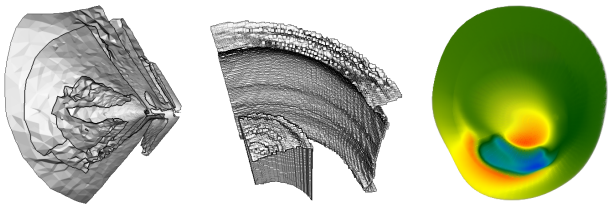


Figure 9: Our triangle subdivision scheme for reducing the total area of triangles with bad aspect ratios.



(a) (b) (c)

Figure 10: Volume visualization with the *k*-buffer. (a) Isosurface extraction of the Fighter tetrahedral mesh (1,403,504 tetrahedra). (b) Isosurface extraction of the Bullet007 MPM dataset (549k particles) with a constant point size. (c) Direct volume rendering of the Heart dataset using our custom *k*-Buffer extension of Mesa.

the first and second nearest fragments to the eye are selected using the *k*-buffer. This forms a ray segment. If the iso-value is in the range of the current ray segment, then the depths of the fragments are linearly interpolated to find the depth of the isosurface on this ray segment, and the generated fragment goes through a depth test. (An entry of the *k*-buffer is used as a depth buffer.) To optimize the size of the *k*-buffer, we only store a depth value and a scalar value with each fragment. The normals are computed in a post-processing pass using central differencing on the depths [Livnat and Tricoche 2004]. The surface is then shaded using a Lambertian term in eye space and silhouettes are computed (without additional cost).

This approach works equally well for tetrahedral meshes as well as for particles (points). To our knowledge, this is the first GPU-based approach for interactively extracting isosurfaces from particle data. Figure 10a and Figure 10b show the results of isosurface extraction from a tetrahedral mesh as well as from Material Point Method (MPM) simulation particles.

Volume Rendering. The first *k*-buffer application was for the specific case of direct volume rendering of unstructured grids [Callahan et al. 2005]. Using programmable blending with the *k*-buffer, the ray gaps between triangles can be composited into the framebuffer for single-pass volume rendering. For each fragment, the *k*-buffer is used to find the two fragments closest to the eye. The scalar values of these two fragments, along with the distance between their depths, are used to look up the color contribution for the ray gap in a pre-computed table of volume rendering integrals. This color is then composited in front-to-back order. See [Callahan et al. 2005] for more detail. Figure 10c shows an example of volume rendering of a tetrahedral mesh using the *k*-buffer. Since this application uses a texture fetch (table lookup) in the *k*-buffer program, it would only be supported by the fragment-shader option and not by the blending option (*cf.* Section 3.1).

5 Results

For our experimental results, we rendered several large scenes using depth peeling with the *k*-buffer. We used our current experimental implementation in OpenGL based on RMW textures to demonstrate the optimal throughput of the *k*-buffer. The *k*-buffer attribute parameters were varied and a 512×512 viewport was used. To simulate a hardware *k*-buffer, we used OpenGL with GLSL shaders, 32-bit RGBA floating-point textures, and no Z culling. Our test machine was running Linux with an AMD Opteron at 2.2 GHz, 4 GB RAM, and an NVIDIA GeForce 7900 GTX with driver 1.0-8774. The scenes were rendered with three different modes: traditional multi-pass, single-pass with the *k*-buffer, and single-pass with the *k*-buffer including heuristics to decrease RMW artifacts. The last mode was used to generate most of our images and involved sorting all the triangles by their centroid on the CPU and batching them in sorted order with 32 triangles per batch. In all cases, our timing results represent the average framerates observed when rendering the scene without deferred shading (see Table 1). When the pipeline is vertex limited, we get a linear speedup with respect to geometry passes.

k-Buffer	Dataset	Num Tris	MP	SP	SPwH
4 RGBZ	Dragon	871k	41.4 fps	139 fps	3.1 fps
16 Z			10.3 fps	139 fps	2.6 fps
4 RGBZ	Powerplant	12.7M	5.1 fps	20.1 fps	0.2 fps
16 Z			1.3 fps	20.1 fps	0.2 fps
4 RGBZ	Lucy	28.0M	0.4 fps	1.7 fps	0.2 fps
16 Z			0.1 fps	1.7 fps	0.1 fps

Table 1: Timing results for depth peeling using traditional multi-pass rendering (MP), single-pass rendering with the *k*-buffer (SP), and single-pass rendering with the *k*-buffer using heuristics to avoid RMW hazards (SPwH). Several *k*-buffer layer sizes (4 or 16) and attribute combinations (RGBZ or Z) are compared.

In addition to timing statistics, we performed an analysis of the efficiency of our proposed subdivision scheme for improving visibility order on degenerate datasets. We used our software implementation to compare the maximum error that can occur at increasing subdivision steps. For our experiments, we used a tetrahedral dataset of a heart that contains many badly formed primitives and a *k*-size of 6. Table 2 shows the results of our experiment. These results demonstrate that object-space manipulation of the geometry can minimize the errors resulting from using a small *k*.

Budget	Total Tris	Max Error
1.0×	0.3M	17
1.5×	1.0M	6
6.0×	4.3M	3
12.0×	8.7M	0

Table 2: Subdivision results when volume rendering a tetrahedral mesh of a heart (359,310 triangles) with a *k*-buffer of 12 fragments per pixel at different triangle count ratios (triangle budget relative to the original number of triangles). Maximum error represents the maximum disorder over the image in the nearly-sorted sequences of fragments for every pixel.

6 Discussion

Using a *k*-buffer to implement algorithms that operate on multiple fragments per pixel in a single geometry pass has two important benefits. First, for large datasets, each rendering pass of the geometry in the scene reduces the interactivity of the system substantially. Thus, effects that require multiple passes can greatly affect the usability of the system. With the *k*-buffer, this cost can be drastically reduced by capturing the relevant fragments in the first pass. Another important benefit of the *k*-buffer is that it simplifies the rendering of effects. In large rendering engines, each effect that is incorporated will add to the complexity of the code. With the *k*-buffer, all of the raster operations are encapsulated inside a single *k*-buffer shader, rather than having part of it controlled by fixed-function in the application and other parts controlled by a shader. This makes the shaders self-contained, and simplifies effect development.

A first step toward supporting a programmable *k*-buffer in GPUs could be a simple depth peeling raster operation for MRTs. This operation could be implemented as a set of depth and associated color buffers (i.e., multiple depth buffers). At each pass, these buffers would be filled in front-to-back order with the layered fragments using an insertion sort. This approach would be much simpler

than a full k-buffer, because it does not require programmability. Instead, it would simply be enabled by the user in the API (*e.g.*, `GL_DEPTH_PEELING`). It could also possibly allow early-Z tests by comparing with all the stored depth values. Since many of the effects described in this paper can be performed with depth peeling, this change would have a high impact at a relatively low cost.

DirectX 10 [Blythe 2006] enables single-pass object-space depth partitioning by computing a render target index in a geometry shader. It can also perform fragment-level depth range culling differently for each render target using one viewport per render target. However, in this case, the geometry may need to be rasterized once per depth partition. The advantage of the k-buffer is that it only needs the geometry to be rasterized once. Geometry shaders may ease the need for deep k-buffers for visibility ordering by allowing the degenerate triangles to be split recursively on the hardware. The subdivided triangles could then be sorted on the CPU or the GPU, and finally rendered using image-space sorting and blending. Determining the trade-off between subdivision and k size needs to be addressed when these features become available.

7 Conclusions

In this paper, we have provided a definition of the k-buffer, a generalization of the traditional Z-buffer framebuffer. We describe the k-buffer data structure, operations on it, its implementation on current hardware, and two proposals for future hardware implementations. The k-buffer reduces many multi-pass algorithms to a single geometry pass, and our experimental results for a number of these effects show the performance advantage of the approach. By providing access to multiple ray intersections along a viewing ray, the k-buffer provides algorithms with a more global view of the scene, opening up a number of new algorithmic possibilities for raster graphics.

In the future, we are interested in a more theoretical foundation that describes the relationship between triangle subdivision and sorting using a fixed-size network. In particular, it would be useful to know the amount of geometry processing for a given k that is required for a complete fragment sort.

Acknowledgments. The authors thank Milan Ikits, Carlos Scheidegger and Mathias Schott for ideas and useful discussion and ATI and NVIDIA for donated hardware. The work of Louis Bavoil, Steven Callahan, and Cláudio Silva has been supported by the National Science Foundation under grants CCF-0401498, EIA-0323604, OISE-0405402, IIS-0513692, CCF-0528201, and OCE-0424602, the Department of Energy, an IBM Faculty Award, the Army Research Office, and a University of Utah Seed Grant. The work of João Comba is supported by a CNPq grant 478445/2004-0.

References

BAVOIL, L., CALLAHAN, S., AND SILVA, C. 2006. Robust soft shadow mapping with depth peeling. SCI Institute Technical Report UUSCI-2006-028, University of Utah.

BERNARDON, F. F., COMBA, J. L. D., PAGOT, C. A., AND SILVA, C. T. 2006. GPU-based tiled ray casting using depth peeling. *Journal of Graphics Tools 11.3*.

BLYTHE, D. 2006. The Direct3D 10 system. *ACM Trans. Graph. 25*, 3, 724–734.

CALLAHAN, S. P., IKITS, M., COMBA, J. L., AND SILVA, C. T. 2005. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3, 285–295.

CARPENTER, L. 1984. The A-buffer, an antialiased hidden surface method. In *Proceedings of SIGGRAPH*, vol. 18, 103–108.

CATMULL, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of Computer Science, University of Utah.

DOGGETT, M. 2005. Xenos: Xbox360 gpu. Eurographics 2005 Slides.

EISEMANN, E., AND DÉCORET, X. 2006. Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 71–78.

EVERITT, C. 2001. Interactive order-independent transparency. Tech. rep., NVIDIA Corporation.

GOVINDARAJU, N. K., HENSON, M., LIN, M. C., AND MANOCHA, D. 2005. Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, 49–56.

HACHISUKA, T. 2005. High-quality global illumination rendering using rasterization. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Mar., ch. 38, 615–633.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2006. Efficient depth buffer compression. In *Graphics Hardware 2006*.

HOUSTON, M., PREETHAM, A. J., AND SEGAL, M. 2005. A hardware F-buffer implementation. Tech. Rep. CSTR 2005-05, Stanford University.

JAMES, G., AND GREEN, S. 2004. Real-time animated translucency. (GDC 2004 Slides).

JOUPPI, N. P., AND CHANG, C.-F. 1999. z^3: an economical hardware technique for high-quality antialiasing and transparency. In *SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 85–93.

KELLEY, M., GOULD, K., PEASE, B., WINNER, S., AND YEN, A. 1994. Hardware accelerated rendering of CSG and transparency. In *Proceedings of SIGGRAPH*, 177–184.

KILGARIFF, E., AND FERNANDO, R. 2005. The GeForce 6 series GPU architecture. In *GPU Gems 2*, M. Pharr and R. Fernando, Eds. ch. 30, 471–491.

LEFOHN, A. E. 2006. *Glift: Generic Data Structures for Graphics Hardware*. PhD thesis, University of California Davis.

LIVNAT, Y., AND TRICOCHE, X. 2004. Interactive point-based isosurface extraction. In *Proceedings of IEEE Visualization*, 457–464.

LOOP, C. 1987. *Smooth Subdivision Surfaces Based on Triangles*. Master's thesis, Department of Mathematics, University of Utah.

LUFT, T., AND DEUSSEN, O. 2006. Real-time watercolor illustrations of plants using a blurred depth test. In *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*.

MAMMEN, A. 1984. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9* (July), 43–55.

MARK, W. R., AND PROUDFOOT, K. 2001. The F-buffer: a rasterization-order FIFO buffer for multi-pass rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 57–64.

MENDEZ, A., SBERT, M., CATA, J., SUNYER, N., AND FUNTANE, S. 2006. Real-time obscurances with color bleeding. In *ShaderX4: Advanced Rendering Techniques*, W. Engel, Ed. Charles River Media.

NAGY, Z., AND KLEIN, R. 2003. Depth-peeling for texture-based volume rendering. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, 429.

NVIDIA. 2005. GPU programming exposed: The naked truth behind NVIDIA's demos. (SIGGRAPH 2005 Slides).

SEDGEWICK, R. 1998. *Algorithms In C*, third ed. Addison-Wesley, 298–301,403–437.

THORNTON, J. E. 1964. Parallel operation in the control data 6600. 32–39.

WEILER, M., KRAUS, M., MERZ, M., AND ERTL, T. 2003. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics 9*, 2, 163–175.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. GPU-accelerated high-quality hidden surface removal. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 7–14.

WITTENBRINK, C. 2001. R-Buffer: A pointerless A-buffer hardware architecture. In *ACM-Eurographics Workshop on Graphics Hardware*, 73–80.

WOO, A. 1992. The shadow depth map revisited. In *Graphics Gems III*. 338–342.