# O-Buffer: A Framework for Sample-Based Graphics

Huamin Qu and Arie E. Kaufman, *Fellow*, *IEEE*

**Abstract**—We present an innovative modeling and rendering primitive, called the *O-buffer*, as a framework for sample-based graphics. The 2D or 3D O-buffer is, in essence, a conventional image or a volume, respectively, except that samples are not restricted to a regular grid. A sample position in the O-buffer is recorded as an offset to the nearest grid point of a regular base grid (hence the name O-buffer). The O-buffer can greatly improve the expressive power of images and volumes. Image quality can be improved by storing more spatial information with samples and by avoiding multiple resamplings. It can be exploited to represent and render unstructured primitives, such as points, particles, and curvilinear or irregular volumes. The O-buffer is therefore a unified representation for a variety of graphics primitives and supports mixing them in the same scene. It is a semiregular structure which lends itself to efficient construction and rendering. O-buffers may assume a variety of forms including 2D O-buffers, 3D O-buffers, uniform O-buffers, nonuniform O-buffers, adaptive O-buffers, layered-depth O-buffers, and O-buffer trees. We demonstrate the effectiveness of the O-buffer in a variety of applications, such as image-based rendering, point sample rendering, and volume rendering.

**Index Terms**—Sample-based rendering, image-based rendering, hybrid rendering, irregular sampling, hierarchy, offset, frame buffer, layered depth image.

## 1 Introduction

Sample-based primitives, such as images, volumes, and points, have been widely used in computer graphics and visualization. Sample-based graphics employs pixels, voxels, or points and renders them using image-based rendering (IBR), volume rendering, or point sample rendering, respectively. Image-based modeling and rendering methods use an image (e.g., texture, depth image [10], [12], layered depth image (LDI) [3], [8], [20]) as the modeling and rendering primitive. All these images are digital, have a finite resolution, and are stored in a regular 2D gridded buffer. Similarly, 3D volumes from medical imaging, scientific simulation, or voxelization of a geometric model [6] also have a finite resolution and are usually stored in a regular 3D gridded buffer.

The regular sampling pattern and limited resolution of images and volumes result in limited spatial precision for sample points. This makes conventional images and volumes inefficient in precisely representing high-frequency details (e.g., edges of 3D objects) and in handling multiple samples in one grid point. Also, there is an abundance of irregular samples in computer graphics, such as point clouds from range scanners, curvilinear and irregular grids from scientific computations, and samples from adaptive or jittered sampling for antialiasing. However, regular images and volumes have accuracy limitation in handling these irregular samples.

As graphics primitives diversify, one challenge for sample-based rendering is how to organize and mix different sample primitives in one scene and render them in correct visibility order, especially when volumes are involved. Typically, geometric models, images, volumes, and point clouds represent some objects in 3D space. Therefore, one may think that a volume representation is an ideal framework to represent all these models. However, when pixels and points are projected to 3D space or when geometric models are discretized, they have to be resampled into a limited resolution 3D grid. Some information in the original samples may then be lost.

We believe that the regularity of conventional images and volumes limits their modeling power and might actually be unnecessary. For example, some image representations, such as LDIs, are assembled from pictures of the real world. When pixels in these pictures are warped to a new viewpoint, they no longer fall on a regular grid. Usually, a resampling process is needed. However, these representations only serve as intermediate models and are not used directly for display. Therefore, it is unnecessary for pixels in these images to be defined on a regular grid. In the past, the regular pattern of images and volumes was critical for rendering. However, depth images, layered depth images, point samples, and volumes can all be rendered now by some kind of 3D warping or splatting algorithm [5], [10], [20], [21], [22]. When samples are projected onto an image display plane, they do not fall on the grid points, requiring reconstruction and resampling anyway. Therefore, whether or not samples of the original images and volumes are on a regular grid is not that important.

To face these challenges, we propose a new modeling and rendering primitive, the *O-buffer*. It is essentially a conventional image or volume, except that samples are no longer restricted to grid points on a regular grid and their position is recorded as an offset to a grid point of a regular base grid. The offset is usually quantized for compact

• *The authors are with the Center for Visual Computing (CVC) and Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400. E-mail: {huamin, ari}@cs.sunysb.edu.*

*Manuscript received 1 Oct. 2003; revised 12 Nov. 2003; accepted 17 Nov. 2003.*
*For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-0095-1003.*

storage and efficient rendering. Therefore, the O-buffer can represent the position of samples more precisely without increasing the sampling rate of the image or the volume, can eliminate their need for a regular grid, and can serve as a container for intermediate and final results to avoid multiple resamplings during transformation and to reduce aliasing. In addition, the O-buffer can be naturally utilized to represent and render unstructured primitives, such as points, particle systems, and curvilinear and irregular volumes. Consequently, the O-buffer can be regarded as a uniform representation for a variety of primitives and can thus support the mixing of several different primitives (e.g., images, points, volumes) in the same scene.

O-buffers are related to the concepts of subpixels, subvoxels, subgrids, and offset buffers, which have been used mostly for antialiasing in geometry-based rendering [2], [18]. The O-buffer, however, is primarily designed for sample-based models. The O-buffer was inspired by the pioneering work of Popescu et al. [12], [13], who used an offset buffer to resolve the visibility of pixels and for antialiasing in image-based rendering. However, their work is limited to 2D and used mainly for better reconstruction after image warping. In our earlier work, we used an offset buffer for consecutive warping that resulted in an IBR method with stable frame rates for a real-time system [15]. Botsch et al. [1] attached offset attributes to the samples of their point sampled geometry to guarantee water tight surfaces. Sen et al. [19] used offsets to record the locations of representative points on the geometric silhouette in a shadow depth map for antialiasing.

In contrast, the O-buffer is a general 2D or 3D sample-based primitive. The forms of O-buffer include 2D/3D O-buffer, uniform/nonuniform O-buffer, regular/adaptive O-buffer, and O-buffer tree. Our main contribution in this paper is the introduction of the O-buffer as a new modeling and rendering primitive, a uniform framework for sample-based graphics, a unified way for hybrid rendering, and a versatile representation which can be exploited to solve a wide range of problems in computer graphics and visualization.

This paper extends our previous work [14] with a more detailed exposition of the concept, algorithms, and applications of the O-buffer. A more systematic classification of O-buffers and a more accurate analysis of storage requirements for O-buffers are presented. We also give a high-level overview of motivations and methods to convert other primitives to O-buffers. More importantly, we introduce the O-buffer tree as a structure to compress sparse O-buffers and to organize samples in a hierarchy. In addition, we explore three new applications of O-buffers in image-based rendering and point sample rendering.

In the next section, we introduce the O-buffer representation and related quantization and data structures. We present the O-buffer tree in Section 3. In Section 4, we describe motivations and algorithms to convert other primitives to O-buffers. The rendering algorithms for O-buffers are presented in Section 5. Section 6 introduces some applications for O-buffers. We discuss advantages and limitations of O-buffers and suggest some future work in Section 7.

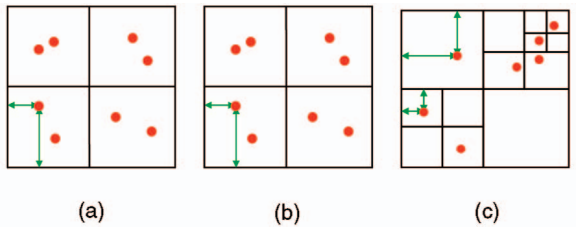


Fig. 1. (a) A uniform O-buffer, (b) a nonuniform O-buffer, (c) an adaptive O-buffer.

## 2 O-Buffer Representations

The O-buffer is basically a conventional image or volume except that the sample points in the O-buffer are no longer on a regular grid. A sample point position is recorded by its offset to a grid point in a regular base grid. This grid can be rectangular, cylindrical, spherical, etc. For simplicity, we use, in this paper, a rectangular grid as the base grid, which is the most common case. We further define the following terms: *regular samples* for samples on a regular grid; *irregular samples* for unorganized samples; *offset samples* for samples whose positions are recorded by offsets to a regular base grid.

### 2.1 Forms of O-Buffers

The O-buffer is a very general primitive and goes beyond specific graphics primitives, such as images, volumes, and points. Thus, O-buffers have a variety of forms. According to the dimension of the data, O-buffers can be classified into *2D O-buffers*, which are offset images, and *3D O-buffers*, which are offset volumes. The 3D O-buffer has important applications in hybrid rendering which will be described in Section 6.4.

The number of samples in every grid cell of an O-buffer can be the same or different. According to the uniformity of the sample distribution, O-buffers can be classified into *uniform O-buffers*, where the number of offset samples stored in every grid cell of the O-buffer is the same, and *nonuniform O-buffers*, where this number is different. Fig. 1a and Fig. 1b show a small portion of a uniform and a nonuniform O-buffer in 2D. Nonuniform O-buffers provide more flexibility than uniform ones, thus they may be more useful in some applications. A typical 2D nonuniform O-buffer is a layered depth O-buffer, which will be presented in Section 6.1.

According to the type of the base grid, O-buffers can be classified into *regular O-buffers*, where the base grid is uniform, and *adaptive O-buffers*, where the base grid is adaptive [4]. Fig. 1c shows a small portion of an adaptive O-buffer with a quadtree style adaptive base grid. Each nonempty node of the quadtree contains offset samples. The positions of the offset samples in a certain level of the quadtree are encoded relative to the base grid corresponding to that level of the quadtree. The adaptive O-buffer can be used to organize samples with nonuniform spatial precision so that small features can have small scale offsets while large features have large scale offsets. The adaptive distance field (ADF) representation proposed by Frisken et al. [4] also uses an adaptive grid, but it is designed to represent surfaces. In contrast, the adaptive O-buffer is a

sample-based representation. It is designed to organize irregular samples and accelerate the rendering of curvilinear and irregular grids.

According to the dimension of the offset, O-buffers can be classified into O-buffers with 1D offsets, O-buffers with 2D offsets, and O-buffers with 3D offsets. By definition, we can see that many familiar computer graphics primitives are just special cases of O-buffer. Regular images and volumes are just O-buffers with zero offset. Z-buffers, relief textures, depth images, height fields, layered depth images are simply O-buffers with offsets only in one direction.

We will use the 2D O-buffer to demonstrate the O-buffer concept and algorithms in the remainder of this paper unless explicitly stated otherwise. The extension to the 3D case is usually straightforward.

### 2.2 Offset Quantization

By using the O-buffer, we can record the position of a sample point much more precisely than a conventional image with the same resolution. The offset can be recorded by using two floating-point numbers, which represent the offsets along two axes of the image coordinates. However, in most cases, a floating-point number is overkill. Usually, the offset can be quantized for compact representation and efficient rendering. We find that quantizing the offset into 16 levels in each axis can achieve a good trade off between spatial accuracy and storage space. In this case, we can use only one byte ($16 \times 16 = 256$) to represent offsets for 2D O-buffers. A one-byte offset for every pixel does not increase the storage requirement for an image much. A typical depth pixel needs 7 bytes (3 bytes for color, 4 bytes for depth). Therefore, a one-byte offset only gives a 12.5 percent increase of the storage requirement for the O-buffer.

One-byte offsets can provide reasonable spatial precision. Suppose the 2D base grid is $512 \times 512$. O-buffers with one-byte offsets can provide spatial precision for sample points just as that of a grid of $8,192 \times 8,192$. We find that this is usually good enough for many applications. It can reduce the position error to be less than $1/32$ pixel distance in each direction in the image plane. Thus, the quantization error in the source image plane is less than half the diagonal length of a subpixel, which is $\sqrt{2}/32$ pixel. Then, the quantization error after warping the source image to a new image plane is less than $\sqrt{2}f/32$ pixel in the new image plane, where $f$ is the zoom factor of projecting the source image to the new image. Similarly, the quantization error for 3D is less than $\sqrt{3}f/32$ pixel, where $f$ is the zoom factor of projecting a volume to an image plane. This means that, even when the zoom factor is 15, which is unusually high, the error caused by the quantization is still less than a pixel. A more detailed discussion of the quantization error can be found in our previous paper [14]. In the remainder of this paper, we assume that the offset is quantized into 16 levels in each axis.

### 2.3 Features of O-Buffers

The O-buffer is a semiregular structure, which is essentially a two-level hierarchical sample container. The first level is a uniform spatial partition grid. This level keeps the overall regularity of the O-buffer. At the second level, each cell of this grid contains a sorted or unsorted list of samples. Rather than being resampled to the grid locations, these samples retain some original spatial information. This level introduces some irregularity for the O-buffer and thus makes it more flexible. For example, the O-buffer can now handle the case of multiple samples falling into the same cell by recording the subpixel positions of these samples. Thus, the O-buffer strikes the middle ground between a regular and irregular representation.

The O-buffer is also an efficient way to encode spatial information. The position of a sample in the O-buffer consists of two parts: the base position and the offset position. The base position is implicitly defined and thus does not need to be stored. Therefore, the storage and memory space for the O-buffer can be dramatically reduced. For example, if we want to maintain 12 bits of precision in each direction and choose the base grid to be $256 \times 256$, we only need 4 bits, instead of 12 bits, in each direction to record the position in an O-buffer.

### 2.4 Data Structures

The data structures for a uniform O-buffer and a nonuniform O-buffer are different. For a uniform O-buffer, we can use the same data structure as a conventional image or volume, which is just a 2D or 3D array of offset samples.

For a nonuniform O-buffer, we use different data structures for construction and for rendering. During the construction stage of an O-buffer, it is more important to efficiently insert and delete offset samples. Therefore, we use a 2D or 3D array of linked offset sample lists to store the O-buffer. During the rendering stage, it is more important to maintain the spatial locality of offset samples in order to most effectively take advantage of CPU cache coherency. We reorganize the offset samples into a linear array ordered bottom up and left to right in the image space. Then, we present two auxiliary data structures to access this array: a double array structure for random access and a run length structure for sequential access. Fig. 2 shows the double array structure. We first calculate the location of the beginning of each scanline in the data array and store them into a 1D array. Within each scanline, for each cell location, we store the offset from the beginning of the scanline to that location. We then use the double array of offsets to locate each offset sample. In order to find offset samples stored at a specific cell, we can simply use one offset to find the beginning of the scanline and then further use another offset to find the first offset sample at that location. This data structure is somewhat similar to the one proposed by Shade et al. [20] for layered depth images. The double array structure allows quick access to a specific cell in a nonuniform O-buffer. However, if we always access a nonuniform O-buffer in scanline order, which is the most common case for rendering, then the storage can be further reduced by using a structure similar to that used by run length encoding. Fig. 3 shows the run length structure. For each cell, we first store the number of samples in this cell, followed by a list of samples in this cell.

Now, we can compute the storage requirement for a nonuniform O-buffer. Let the resolution of the base grid be $w \times h$ and the total number of the offset samples be $N$. Each offset sample needs $B$ bytes. Then, the total storage for these

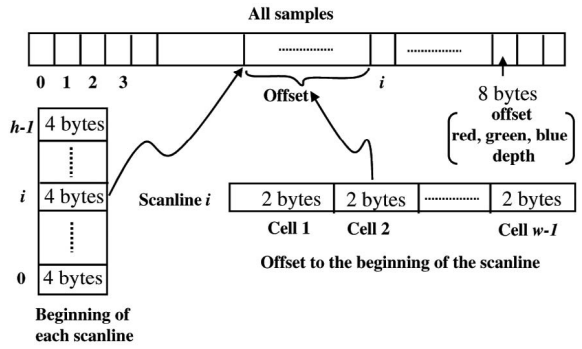


Fig. 2. A double array structure for nonuniform O-buffers.

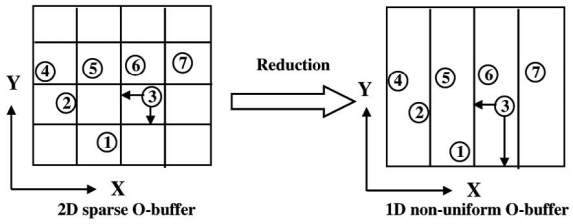


Fig. 4. Two-to-one reduction: projecting a 2D sparse O-buffer into a 1D nonuniform O-buffer and recording the offset of samples to the new low dimensional base grid.

samples needs $N \times B$ bytes. Suppose we use four bytes to store the location of the beginning of each scanline and two bytes to store the offset from that location to the beginning of each cell. Then, the total storage overhead for the double array data structure is $w \times h \times 2 + h \times 4$ bytes. Suppose there are at most 256 samples in each cell, then the storage overhead for the run length structure is only $w \times h$ bytes. The storage overhead for a nonuniform O-buffer is pretty small for typical applications. In the remainder of this paper, we will assume that the run-length structure is used for nonuniform O-buffers.

### 2.5 Comparisons

There are different ways to organize irregular samples with the same spatial precision. For example, we can use a regular buffer, a sample list, an O-buffer, or a quadtree. We now compare the storage requirements for these representations. Suppose there are $N$ irregular samples we want to organize. Each sample has $B$ bytes of attributes. If we use a nonuniform O-buffer with a base grid of $2^w \times 2^h$ to organize them, the total storage will be $N \times (B+1) + 2^w \times 2^h$ bytes. If we use a regular buffer to organize them, we will need a $2^{w+4} \times 2^{h+4}$ high resolution buffer to achieve the same spatial precision provided by the O-buffer. The total storage is $2^{w+4} \times 2^{h+4} \times B$ bytes. It is easy to see that the high resolution regular buffer is sparse and wastes substantial space for most applications. For example, if $w = h = 9$, $N = 2 \times 512 \times 512$, and $B = 8$ bytes, the high resolution regular buffer will consume 512MB space, while the O-buffer only needs 4.75MB. If a sample list or a quadtree is used, the positions of the samples will have to be recorded as absolute positions in a coordinate system. The absolute positions need $(w + 4 + h + 4)$ bits to achieve the same spatial precision provided by the O-buffer. Thus, if we use the sample list representation, we will need $N \times (B + 1 + \frac{w+h}{8})$ bytes. If we use the quadtree to organize irregular samples, the storage space will equal to the storage of the sample list plus the overhead of the pointers from lower level nodes to higher level nodes. Thus, we can safely say that the storage requirement for the quadtree is greater than or equal to that of the sample list. It is easy to see that, as long as

$$N > \frac{2^w \times 2^h \times 8}{w + h}, \tag{1}$$

the O-buffer representation will be more compact than the sample list and the quadtree representations. In other words, if there are, on average, $8/(w+h)$ samples in each cell of the base grid, the O-buffer will use less space than the sample list and the quadtree representations. For a $256 \times 256$ base grid, this number of samples is 0.5. For a $512 \times 512$ base grid, this number is 0.45. Thus, the O-buffer is the most compact data structure among these representations for relatively dense irregular samples. We will discuss some compression techniques for sparse O-buffers in the next section.

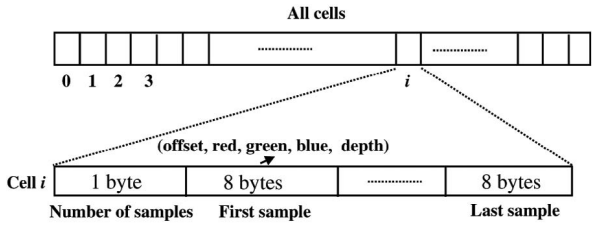


Fig. 3. A run-length structure for nonuniform O-buffers.

## 3 O-Buffer Tree

For 3D O-buffers that represent surfaces, there are very likely many empty cells. In this case, the O-buffer may consume more space than the sample list and quadtree structures. Also, the existence of many empty cells slows down rendering. In this section, we introduce the O-Buffer tree (OB tree) to compress sparse O-buffers and to organize irregular samples in a hierarchy.

### 3.1 Three-to-Two Reduction

For 3D sparse O-buffers, if rendering is our only concern, we can use a 3-to-2 reduction method to compress them. Fig. 4 shows this technique in 2D. For a sparse 3D O-buffer, we can project samples in the 3D O-buffer onto one of the six planes of the bounding box of the O-buffer and organize these samples into a 2D nonuniform O-buffer. The offsets of samples to the new lower dimensional base grid are computed and recorded accordingly. The nonuniform O-buffer used here is similar to an LDI. However, there are two fundamental differences between these two representations. First, samples of the LDI are on a regular grid, while samples in the 2D nonuniform O-buffer are not. Second, samples in the same cell of the 2D nonuniform O-buffer are not necessarily in the different layers of the model. They may belong to the same layer because O-buffers can now handle multiple samples from the same

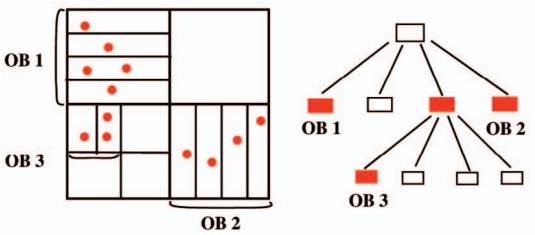


Fig. 5. An O-buffer tree, where every nonempty leaf node of the quadtree is an O-buffer.

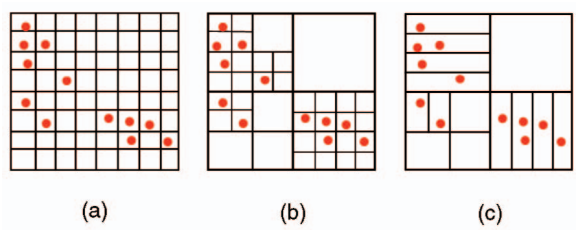


Fig. 6. Different ways to organize irregular samples: (a) a sparse 2D O-buffer, (b) a quadtree, (c) an OB tree.

layer falling into the same cell by recording their subpixel positions. For more details, see Section 6.1.

We can now compare the storage space for 3D and 2D nonuniform O-buffers resulting from the 3-to-2 reduction so that we can decide whether to do this reduction in practice. Suppose there are $N$ irregular samples in a sparse 3D O-buffer with a $2^w \times 2^h \times 2^d$ base grid. Each sample has $B$ bytes of attributes. The storage space for the 3D nonuniform O-buffer is $N \times (B + 1.5) + 2^w \times 2^h \times 2^d$ bytes. After a 3-to-2 reduction, a $2^w \times 2^h$ nonuniform O-buffer only needs $N \times (B + 1.5 + \frac{d}{8}) + 2^w \times 2^h$ bytes. Thus, as long as $N < (8 \times 2^w \times 2^h \times (2^d - 1))/d$, the 3-to-2 reduction will save storage space.

The criterion to choose which of the six planes to use for the 3-to-2 reduction is the uniformity of the sample distribution in the resulted 2D nonuniform O-buffer. We want a final 2D nonuniform O-buffer that has the most uniform distribution of samples and the least number of empty cells. For example, in Fig. 4, we project along the Y direction instead of X because the final nonuniform O-buffer has a more uniform distribution of samples in each cell and no empty cells.

### 3.2 OB Tree

We can combine the 3-to-2 reduction with the octree structure to develop a more powerful compression method for sparse 3D O-buffers. We term the new representation an *OB tree (O-Buffer tree)*. Fig. 5 shows an OB tree. An OB tree is essentially an octree. Every nonempty leaf node of the OB tree stores a 2D nonuniform O-buffer. The OB tree can also be used to organize samples in a hierarchy by storing some low resolution 2D O-buffers in the nonleaf nodes. In this case, the OB tree is similar to the LDI tree of Chang et al. [3] and the LDC tree of Pfister et al. [11]. The differences between the OB tree and the LDI or LDC tree are similar to the differences between the O-buffer and the regular buffer. The OB tree is more general, more accurate, more flexible, and thus more powerful than the LDI or LDC tree.

### 3.3 Comparisons

We now compare the OB tree with the conventional octree and the QSplat representation [17]. Fig. 6 shows some irregular samples organized by a sparse O-buffer, a quadtree, and an OB tree. Compared with the quadtree representation, the OB tree has two major advantages. First, the OB tree is more compact. At a certain level, the 3D block of samples is collapsed into a 2D representation and does not need further subdivision. This may dramatically reduce the height of the tree. Also, the positions of the samples in the OB tree are recorded as offsets, while the positions of samples in the conventional octree are recorded as absolute positions. This can further reduce the storage. Second, the OB tree can be rendered faster. Octrees and OB trees are all rendered by traversing the trees from top to bottom and, at a certain level, the samples are projected. However, the octree projects each sample individually, while the O-buffer projects a block of samples well organized into an O-buffer whose rendering can be accelerated by using incremental computation and lookup tables (see Section 5.1).

QSplat is also a data structure used to organize irregular samples. QSplat encodes a very large number of point samples in a hierarchy of bounding spheres. Each node of the QSplat tree contains the sphere center and radius. However, the node of the QSplat stores a single sample, while the node of the OB tree stores an O-buffer. Thus, the OB tree may have fewer levels than the QSplat tree for some applications. The QSplat is rendered by traversing the hierarchy and projecting individual samples at a certain level, while the OB tree is rendered by traversing the hierarchy and projecting an O-buffer. Samples in the O-buffer are well-organized and can be rendered efficiently using incremental computation. To summarize, the QSplat is a highly optimized structure for point clouds and, in come applications, may be more compact and flexible than the OB tree. However, the OB tree is better organized and can be constructed and rendered efficiently. Thus, the OB tree is a more regular structure than the octree and the QSplat. Fig. 7 shows primitives with different level of regularity for sample-based graphics.

## 4 Construction of O-Buffers: A Conceptual Overview

O-buffers can be constructed from both discrete representations (e.g., images, points, volumes) and continuous representations (e.g., triangle meshes, NURBs, curvilinear and irregular grids). This section gives a high-level

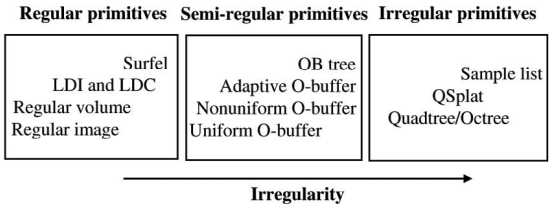


Fig. 7. Primitives with different levels of irregularity for sample-based graphics.

overview of the general principles, motivations, and methods to convert other primitives to O-buffers.

### 4.1 Principles

The conversion algorithms for uniform and nonuniform O-buffers are different. When we convert primitives to a uniform O-buffer, we want to minimize the error between the original primitive and the O-buffer, given a predetermined resolution for the O-buffer. When we convert primitives to a nonuniform O-buffer, we want to minimize the number of samples in the O-buffer, given a predetermined error between the original primitive and the O-buffer.

### 4.2 Motivations

The general motivations to convert other primitives to O-buffers instead of leaving them in their original forms or converting them to regular buffers include:

- Better organization: The semiregularity of the O-buffer lends itself to efficient rendering. Thus, converting other less organized representations, such as point clouds and curvilinear and irregular grids, into O-buffers can accelerate the rendering of these primitives.
- Avoiding resampling: If the sample-based models only serve as intermediate representations, then using O-buffers to cache them can avoid resampling on a regular grid (see examples in Sections 6.1 and 6.5).
- More accuracy: Some sample-based models may contain high frequency details that need more accuracy. These details can be recorded into an O-buffer without the overhead of a high resolution grid.
- Better multiresolution: O-buffers open the possibility of developing more accurate and flexible multiresolution representations for regular images and volumes. O-buffers provide the flexibility for the number and position of samples in low resolution O-buffers. By optimizing these two variables, the low resolution O-buffers can be used to better preserve the information of high resolution primitives. We discuss this issue in our previous paper [14] and its thorough study is left for future work.
- Hybrid rendering: Suppose there are multiple computer graphics primitives in one scene and we need to mix them. After converting these primitives to O-buffers, we only need to render a single primitive, the O-buffer, whose rendering can then be highly optimized and can even be realized in hardware. Also, we can determine the visibility of samples in a 3D O-buffer very easily. Therefore, different samples can be rendered and composited in correct visibility order, which is critical especially for volume rendering. For more details, see Section 6.4.

### 4.3 Methods

The general methods to convert other primitives to O-buffers include:

- Ray tracing: Ray tracing can be used to convert other primitives to 2D O-buffers. Adaptive ray tracing can locate high frequency details (silhouettes, feature boundaries, creases) in original models more precisely. Jittered ray tracing can be used for antialiasing. Ray tracing can also discretize surfaces to 3D O-buffers.
- Warping: Warping is used to transform other 2D primitives to 2D O-buffers. After warping, samples can be recorded into an O-buffer to avoid resampling. For example, a single nonuniform O-buffer can be constructed by warping multiple depth images into a common image plane (see Section 6.1).
- Projection: Projection is used to convert other sample-based primitives to 2D/3D O-buffers. For example, a 3D O-buffer can be constructed by projecting samples in depth images, LDIs, and point clouds to 3D space and computing their offsets to a regular base grid.
- Simplification: Low resolution 2D or 3D O-buffers can be constructed by simplifying regular images, depth images, volumes, and even high resolution O-buffers.
- Discretization: We can discretize surfaces and curvilinear and irregular grids into 3D O-buffers. We presented a vertex clustering algorithm to discretize triangle meshes to O-buffers in our previous work [14].

## 5 Rendering of O-Buffers

There are two basic methods to render O-buffers: either by splatting or by ray tracing. The structure of the O-buffer lends itself well to being rendered by splatting. Considerable speedup can be gained for splatting by exploiting the semiregular structure of O-buffers and the quantization of offsets.

Splatting is an object order approach and memory coherence of data can be exploited. Incremental computation has been used to speed up the rendering of samples on a regular grid [5], [10]. O-buffers with quantized offset can also be rendered rather efficiently by using incremental computation and lookup tables [13], [15]. Splatting consists of two steps: warping samples to the image plane followed by reconstruction and resampling. There are several ways [5], [10] to warp samples in depth images and volumes to a new image plane. In this paper, we base our presentation on a method and notation proposed by Shade et al. [20]. We use 2D O-buffers (i.e., offset images) to demonstrate our method.

### 5.1 Warping of 2D O-Buffers

Let $T$ be the $4 \times 4$ matrix to transform a point from an offset image plane to a destination image plane and $(x_1, y_1, z_1, 1)$ be the homogeneous coordinates of samples in the offset depth image. Then, the warped sample position at the destination image plane can be computed by:

$$T\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = T\begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + z_1 T\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \mathbf{start} + z_1.\mathbf{depth}. \quad (2)$$

To compute the position of the next sample along the same scanline, **start** is simply incremented. For the O-buffer, the next sample along the same scan line is $(x_1 + 1 + u, y_1 + v, 0, 1)$, where $(u, v)$ is the offset of this sample. If the offset is quantized to one byte, we can

precompute $T[u, v, 0, 0]$ for all possible levels of the offset (e.g., 256 levels) and store them in a lookup table, **table**. Before rendering a new image, we use the new camera information to precompute the table indices. Then, we use the following incremental computation:

$$T\begin{bmatrix} x_1+1+u \\ y_1+v \\ 0 \\ 1 \end{bmatrix} = T\begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + T\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + T\begin{bmatrix} u \\ v \\ 0 \\ 0 \end{bmatrix} \quad (3)$$
$$= \mathbf{start} + \mathbf{xincr} + \mathbf{table}[u, v].$$

Compared to regular images, there is only a slight increase in the rendering time for O-buffers.

### 5.2 Resampling to Regular Buffers

The O-buffers can be resampled efficiently to regular buffers for display and other purposes. Each offset sample spreads energy to the neighboring regular sample points. The weights of the contribution of an offset sample to a regular grid point can be either the inverse distance between these two samples or computed from a Gaussian kernel centered at the regular grid point. However, because the offset is quantized, we can precompute these weights for all possible offsets (256 cases for 2D) and store them in a lookup table. Then, we can quickly obtain the weights of the contribution of any offset sample to its neighboring regular samples at runtime. For each regular point, we add the weighted contributions from all neighboring offset samples and normalize the result by the summation of all weights. When the samples are transparent, we first divide the 3D object or scene into different layers [11], [17] based on depth. The samples in the same layer are resampled to a regular grid using the method just described. Then, these layers are composited together from back to front.

### 5.3 Rendering of OB Trees

The OB tree can be rendered similarly to an LDC tree [11]. Suppose there are low resolution 2D O-buffers stored at all the internal nodes of the OB tree. We traverse the tree top down. At a certain level of the OB tree which can provide an adequate sampling rate for the final image, we render the 2D O-buffer stored at that node using the method presented in Section 5.1. Please see Pfister et al. [11] for more details.

## 6 Applications of O-Buffers

O-buffers are versatile representations and can be used to solve a wide range of graphics problems. In this section, we present some typical applications of O-buffers in image-based rendering, point sample rendering, and volume rendering. Our experimental results have been generated on a Dell Dimension 8200 desktop with a 2.53GHz Pentium 4, 1GB of RAM, and an Nvidia GeForce4 with 64MB memory.

### 6.1 Layered Depth O-Buffers

The layered depth image (LDI) proposed by Shade et al. [20] is an important extension of the depth image. It provides the information for the occluded parts of an object. The representation is compact compared to multiple images because the LDI reduces the redundancy in multiple images.

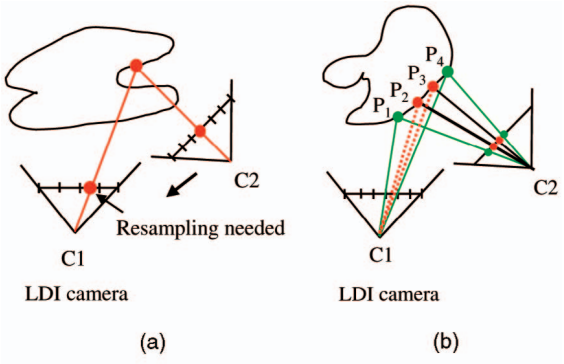


Fig. 8. (a) Multiple resampling problems of the LDI, (b) different sampling rates in the original images. Part of the surface is better sampled at the image taken from C2 than at the image taken from the LDI camera.

Having multiple depth images, an LDI can be constructed by warping these depth images into a common camera view. As Shade et al. [20] pointed out, there are some disadvantages to the LDI. First, pixels undergo two resampling steps in their journey from input image to output. This can potentially degrade image quality. Second, some information in the original images can be lost if a surface is better sampled in one of the images than it is from the viewpoint of the LDI. Chang et al. [3] proposed an LDI tree to solve the problem caused by different sampling rates in the original images.

We introduce a layered depth O-buffer (LDOB) as an extension of the LDI to overcome these two disadvantages. Fig. 8a demonstrates the two resampling steps of the LDI. When we warp the depth images to the LDI camera position, the pixels do not fall on a regular grid. Therefore, resampling is needed at the construction stage of the LDI. During the rendering stage, the LDIs are warped to an image plane. At this time, another resampling is needed. The resampling at the construction stage of the LDI is unneccesary because the LDI is just an intermediate representation and is not used for display. Therefore, this resampling can be delayed to the rendering stage with the help of an O-buffer. When we warp an image to the viewpoint of the LDI, we use an O-buffer to record the position of these pixels in the LDI instead of resampling the warped image. Because of the quantization of the offset, it may be more accurate to say that the LDOB avoids resampling to the base grid at the construction stage.

Fig. 8b demonstrates the different sampling rates problem. If the original depth image resolutions are different, then a surface can be better sampled at the original image that has a higher resolution than the LDI image. Even if the resolutions of the original images and the LDI are the same, a surface can still be better sampled at some image other than the LDI because of the orientation of the surface.

We propose a two step warping algorithm to preserve all the information in the original images. The LDOB is still constructed by warping the depth images one by one to the LDOB camera position. However, before we warp a depth image to the LDOB, we first try to reconstruct this depth image from the current LDOB by warping pixels in the LDOB to the image plane of this depth image. Only pixels in

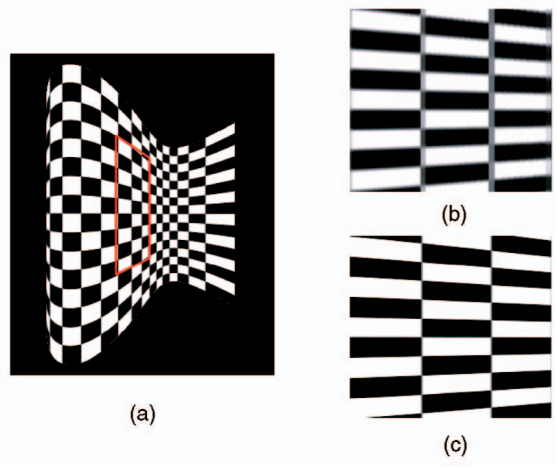


Fig. 9. Improving image quality with an LDOB: (a) a curved checkerboard, (b) an image rendered by warping an LDI, (c) an image rendered by warping an LDOB.

the original depth image that cannot be reconstructed from the LDOB are warped and stored into the LDOB. These pixels either represent new surfaces or surfaces which cannot be sampled well in the LDOB. The main steps of our algorithm are:

1. Initialize the camera and the resolution of the LDOB.
2. Sort the depth images in ascending order according to the angle between the camera of the depth images and the camera of the LDOB.
3. Repeat the following steps for every depth image in the sorted list:
   a. Warp the LDOB to the image plane of the depth image and mark all pixels in the depth image that lie near the warped pixels from the LDOB.
   b. Update the LDOB by warping all unmarked pixels in the depth image to the image plane of the LDOB and storing them in the LDOB. The positions of these pixels are recorded using offsets.

The size of the splat for pixels in an LDOB can be fixed [16] or computed from normal vectors [20]. In our case, the splat can be precomputed, quantized, and stored with the samples. For each sample, we find its neighboring samples in the same surface of the model. Then, we compute the maximum distance between this sample and its neighboring samples and use this distance as the splat size [22]. This way, we guarantee that there are no holes in the rendered images.

Fig. 9 demonstrates the LDOB advantages. We use a curved checkerboard for the experiment. We render seven depth images of the checkerboard from different camera positions around the object. The resolution for each image is $512 \times 512$. These seven images have a total of 673,993 nonbackground pixels. Then, we assemble separately an LDI and an LDOB. The LDI has 102,362 nonbackground pixels, while the LDOB has 210,855. Both the LDI and the LDOB can reduce the redundancy of the original images. However, the LDOB keeps more samples from original images of surfaces that are not well-sampled from the camera of the LDI. Fig. 9a shows the image from the viewpoint of the LDI camera. The red quadrilateral region is not well sampled from this viewpoint. Fig. 9b shows a portion of the image rendered by warping the LDI back to one of the original images which has a better sampling rate for the red quadrilateral region. In order to avoid holes, a large splat size has to be used and the image is therefore blurred. However, if the LDOB is used, the samples from the original images are kept. When we warp the LDOB back, we still have enough sampling rate and the final image is still sharp. Because the O-buffer is used to avoid multiple resampling, the image has almost the same quality as the original image (see Fig. 9c). It takes 0.13 sec to render the LDI and 0.15 sec to render the LDOB. The LDOB has more samples to warp. However, resampling is faster for the LDOB because the LDI has to use a larger splat size than the LDOB. If we do not use quantization and incremental warping, the time to render the LDOB will be 0.19 sec. Thus, compared with the sample list representation, the O-buffer can gain more than 20 percent speedup for splatting.

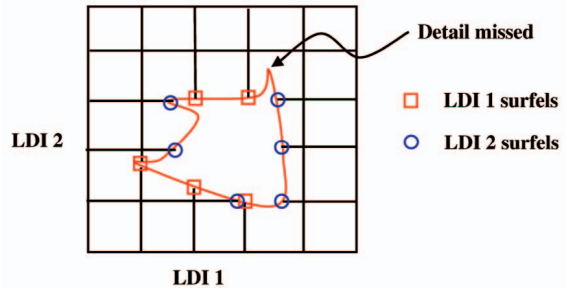


Fig. 10. Surfel representation.

### 6.2 Adequately Sampled Point Representations with O-Buffers

A point sample representation is adequately sampled at the given resolution and magnification if surface point samples are dense enough so that, when the object is viewed, there will be no holes, independent of the viewing angle [5], [8], [11]. A typical example is the surfels method proposed by Pfister et al. [11] (see Fig. 10). Surfels sample geometric models from three sides of a cube into three orthogonal LDIs, called a layered depth cube (LDC) [8]. The LDC can guarantee that the surface is adequately sampled [8], [11]. With the introduction of the OB tree, the O-buffer can be used to enhance the surfel representation and provide a compact, yet adequately sampled point representation for surfaces.

A fundamental limitation of LDC sampling is that thin structures that are smaller than the sampling grid cannot be correctly represented and rendered. For example, spokes, thin wires, or hair are hard to capture. Fig. 10 illustrates this problem. If the O-buffer is used, this problem can be easily solved. The O-buffer can capture much smaller features than a regular buffer without the overhead of a high resolution grid. We can use adaptive ray tracing to locate small features missed by LDC sampling and record the positions of these small features in the O-buffer.

Also, using three LDIs causes more storage space and rendering time. Pfister et al. [11] used a 3-to-1 reduction method which reduces the LDCs to one LDI. However, this approach resampled the surfels to the regular grid. This may

degrade image quality and smooth out some fine details. Actually, we can easily combine three LDIs into one LDOB and reduce any existing redundancy by deleting points that are getting too close. For example, in Fig. 10, we can project the surfels from LDI 2 to the image plane of LDI 1 and combine two LDIs into one LDOB. Similarly, the LDC tree used in surfels can be enhanced with the OB tree.

### 6.3 Image-Based Edge Antialiasing

Image-based models consist of images which are either pictures of the real world or rendered images of some synthetic scenes. Edges in 3D objects cannot be sampled adequately and precisely due to the discrete nature of images. This may cause two problems for image-based rendering: coarse edges when we warp a depth image to a new viewpoint and aliasing around edges when we combine multiple images together because the edges in these images cannot match each other precisely due to the limited resolution of regular images.

We present an image-based edge antialiasing method using O-buffers. Our method first extracts edges either by adaptively sampling geometric models to locate edges precisely or by edge detection from high resolution images taken from the real world. The edge of an object can be detected by finding the depth and color discrepancies of its image. The pixels near the depth discrepancies are the boundary pixels and are treated as edges. Then, we build multiresolution representations for both the original images and the edge images. The low-resolution images are generated by low-pass filtering the high-resolution images. The edges (3D lines in the edge images) are simplified by using a geometric simplification method. Generating low-resolution edge images is equivalent to using fewer vertices to approximate the original 3D lines. The new vertices of the edge images are recorded by an O-buffer.

There are two ways to suppress aliasing around the edges. One way is to combine a low-resolution image and a high-resolution edge image into one nonuniform O-buffer. In this case, the edges have better sampling rates. Another way is to combine the image and the edge image at the same resolution into one uniform O-buffer in which the samples along edges have a better spatial precision, even without a higher sampling rate. More accuracy for the edge pixels can provide better image quality, more accurate registration of multiple depth images, and thus suppress aliasing.

The edge detection and the construction of the O-buffer can be done as preprocessing. After that, the edge-antialiases image-based representations can be used in real-time systems.

Fig. 11 shows an example of suppressing aliasing around edges of the Buddha model using nonuniform O-buffers. We first generated a depth image of the model. The image resolution is $512 \times 512$. We constructed a low-resolution $256 \times 256$ image by box filtering the original image. Fig. 11a shows an image rendered by warping the low-resolution image. The edges are fuzzy and blocky. Then, we extracted the edges of the high resolution image and stored them with the low-resolution image in a $256 \times 256$ nonuniform O-buffer. Fig. 11b shows the image rendered by warping the O-buffer. The edges look much better.

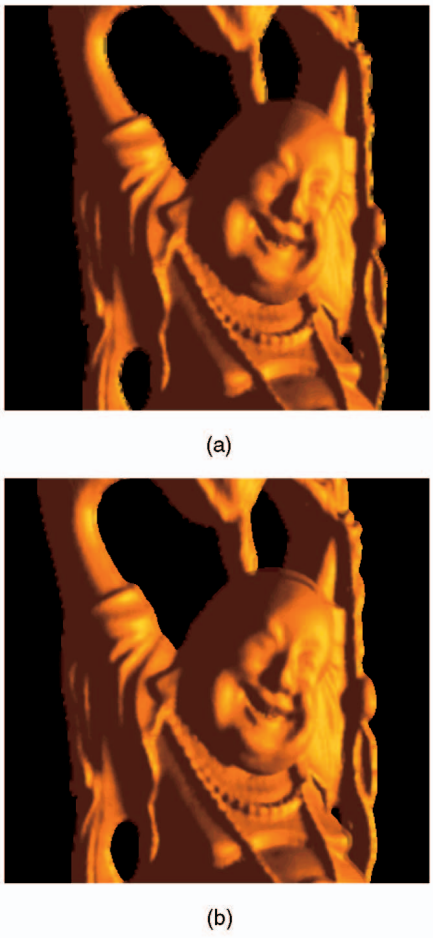


Fig. 11. Image-based edge antialiasing: (a) a Buddha rendered from a depth image, (b) a Buddha rendered from an O-buffer.

### 6.4 Hybrid Rendering with O-Buffers

Volumetric data constructed from CT and MRI scans are widely used in medical virtual systems and they are usually rendered using volume rendering. With the increasing requirement for more realistic simulations, hybrid volume rendering becomes even more important. For example, some medical operations involve penetrating the human organ with a medical device such as a scalpel or an implant such as a pace maker. These objects may be modeled by different graphics primitives. The human body is usually modeled by volumetric data, while the medical devices can be modeled by polygonal, point-based, or image-based models. These models need to be mixed in the same scene in the virtual system. Another example for hybrid volume rendering is a polygonal plane flying through a volumetric cloud over an image-based terrain.

Hybrid volume rendering is a challenging problem. All primitives in a scene must be drawn in topologically depth sorted order because compositing with the over operator for volume rendering is not commutative [7]. Hybrid volume rendering methods can be categorized into two classes: All primitives are either rendered separately and composited later or first converted into a common primitive and rendered together. As mentioned before, if we first convert all primitives in a scene into one common primitive, we only need to render one primitive whose rendering can then

be highly optimized and can even be implemented in hardware. Handling only one primitive makes hardware units smaller and, thus, more efficient. This approach is used in graphics boards, where all geometric models are first converted into triangles so only one primitive is processed by the hardware. Therefore, we believe that converting all primitives into one common primitive has its advantages and may be a starting point to design hardware for hybrid rendering.

The O-buffer provides a unified framework for hybrid rendering because various primitives (e.g., triangle meshes, points, depth images, volumes) can all be converted into O-buffers which provide high spatial precision for samples and avoid unnecessary resampling. If these primitives have no relative motion, they can be premixed into one 3D nonuniform O-buffer. The O-buffer can easily solve the visibility order. Thus, blending and compositing of different objects can be elegantly handled by rendering the samples in the O-buffer from back to front. Furthermore, when multiple samples from different models occupy the same cell of the 3D space, the redundancy can be reduced by only storing one sample based on some predetermined overlapping rules.

There are other ways to mix triangle meshes, images, points, and volumes. For example, these objects can be separately warped and Z-buffers can be used to solve the visibility. One problem with this approach is that it is inefficient to blend multiple transparent objects. These objects have to be sliced into slabs in the depth direction and special attention must be paid to guarantee that slabs from different primitives in a similar depth zone are rendered and mixed together. Another approach is to organize all these objects in an octree-based structure which can then be splatted from back to front. This approach is similar to our method. However, O-buffers are more compact and can be warped faster than irregular samples organized as an octree.

Fig. 12 shows an example which mixes a volumetric CT lobster with an image-based skeleton hand. The resolution of the lobster is $320 \times 320 \times 34$. The image-based human hand consists of six $512 \times 512$ depth images which were generated by rendering a geometric model from the stereolithography archive at Clemson University. These images were projected to 3D space and combined with the lobster into a 3D O-buffer. Then, the 3D O-buffer was rendered by volume splatting. The rendering time is 1.2 seconds. It can be seen that these two primitives are blended nicely.

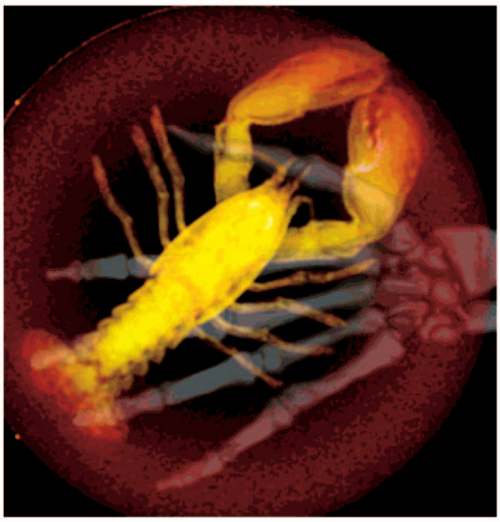

Fig. 12. Mixing a volumetric lobster with an image-based human hand into one 3D O-buffer which is then rendered by splatting.

### 6.5 Sample Caching

Image-based rendering can be used to accelerate surface and volume rendering by exploiting frame-to-frame coherence. There are two ways to use the previously rendered images. One approach is at the image level, where whole previous images are cached and reused. Another approach is at the pixel level, where only some pixels from previous images are cached and reused. The image level approach is widely used [9]. An image at some key viewpoint (key frame) is rendered by a traditional method from a high-resolution scene. Then, this image can be warped to generate a few new frames (derived frames) for this scene. If this key frame image cannot provide adequate quality, the whole image is rerendered. However, if this key frame image has to be generated on the fly and the time to render it is much longer than the time to render a derived frame, an unstable frame rate will be noticed.

A pixel-level approach only caches and reuses some pixels from previous images [15]. Therefore, an image-based rendering method with stable frame rates can be developed by updating only a fraction of the samples at each frame instead of updating the whole image once in a while. These pixels or points are assembled from some previous images. Therefore, when these pixels or points are projected onto a new image plane or 3D space, they no longer fall on a regular grid. O-buffers can be used to cache these pixels and points. We can use 2D O-buffers, 3D O-buffers, or LDOBs to cache samples based on the application. There are two common scenarios where IBR is used to accelerate rendering of a complex scene. In the first scenario, a user walks or flies through a scene. In this case, a new part of the scene comes into the view and an old part of the scene disappears. The pixels disappearing at a viewpoint are unlikely to appear again. In our previous work [15], we used 2D O-buffers to cache these pixels. However, one disadvantage to this approach is that we only cache samples which are visible at the current viewpoint. This means that, when viewpoints are changed, we always need the original models to generate new images. In the second scenario, a user spins a scene and observes it. In this case, pixels disappearing at one frame are likely to appear again in later frames. This scenario can appear in network graphics, where if we have a very complex model at a server and a user wants to see it on a client. Every time the client sends viewing parameters to the server, the server renders a depth image and sends it back. Instead, if the client side can cache these images, then, after a few frames, the client side can render images for new view parameters just from the cached images. However, directly caching all these depth images may be memory expensive because these images have substantial redundancy. The 3D

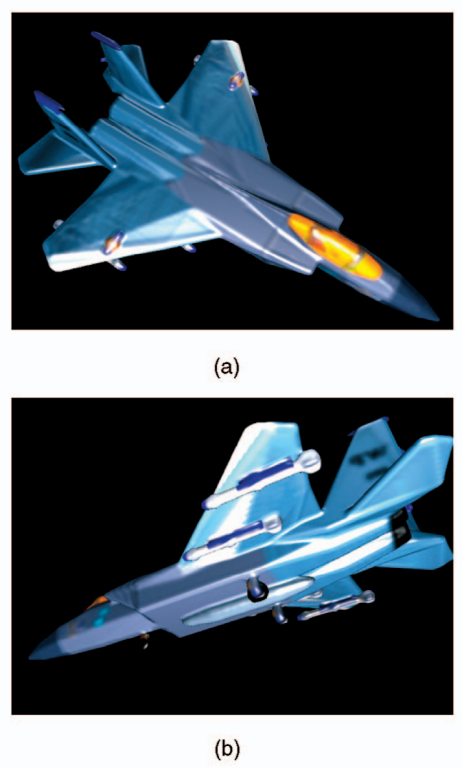


Fig. 13. Caching samples with LDOBs: (a) an LDOB, (b) an image rendered from the LDOB.

O-buffers or LDOBs are ideal ways to cache these samples. We can project pixels in these images into 3D space or to a new image plane and construct a 3D O-buffer or an LDOB. O-buffers can reduce redundancy, avoid resampling, and do not compromise image quality. O-buffers provide more accuracy than regular buffers and are more compact than sample lists.

Fig. 13 shows an example of using LDOBs to cache samples. The original model is a volumetric plane model. The resolution of the plane is $363 \times 512 \times 140$. The color of the original model is prefiltered and stored in the same resolution 3D texture volume. The storage requirements for the density volume and color volume are around 76MB. Depth images of the model are generated by VolVis, a freely distributed volume rendering software package developed at Stony Brook. We have generated 81 depth images of the plane at various camera positions around it by ray tracing. The image resolution is $512 \times 384$. Caching all these images or the original density and texture volumes consumes a large amount of space. However, we can use an LDOB to cache these 81 images and reduce any existing redundancy. The average caching time for each frame is about 0.6 sec, which is good enough for network graphics. Fig. 13a shows the LDOB. The number of total valid pixels (nonbackground pixels) in the LDOB is 94,982. The total storage requirement for the LDOB is only about 0.9MB. Fig. 13b shows an image rendered by warping the LDOB to a new viewpoint. The rendering time is about 0.08 sec.

## 7 Discussion and Future Work

Our primary contribution in this paper has been the introduction of the O-buffer—a uniform framework for modeling and rendering a variety of 2D and 3D sample-based primitives.

Our O-buffer representation is novel and unique in various ways and has the following features:

**Accuracy:** It can provide much higher spatial precision for samples than the same resolution images and volumes.

**Semiregularity:** It is a semiregular structure which lends itself to efficient construction and rendering. It is more powerful than a conventional image or volume and is more compact than other irregular sample-based representations (e.g., sample lists).

**Uniformity:** It provides a uniform framework to represent various irregular and regular sample primitives in computer graphics, such as images, points, and volumes, and thus supports mixing these primitives in the same scene.

**Flexibility:** Storing more accurate spatial information with samples makes it possible for O-buffers to store multiple samples in one cell and provide more flexible and accurate multiresolution schema for images and volumes.

**Versatility:** It is a versatile representation and can solve a variety of problems, such as limited resolution of images and volumes, antialiasing for surface rendering, sample caching for image-based or point sample rendering, data mixing for hybrid volume rendering, irregular sample organization, and level-of-detail management for samples.

O-buffers also have several limitations. First, the current data structures for nonuniform O-buffers and OB trees are tailored for rendering. Dynamically changing the content of the nonuniform O-buffers may incur an overhead which may not be ignored in practice. Second, as an antialiasing method, the O-buffer method is more complicated compared to the supersampling method that is simple and general. Third, generating O-buffer representations for some applications may be time consuming. This may limit its use in real-time systems.

The introduction of the O-buffer opens up a wide range of future directions. These include the investigation of the 3D O-buffers representing density or distance fields, accelerating the rendering of curvilinear and irregular grids with adaptive O-buffers, developing more accurate and flexible multiresolution representations for regular images and volumes using O-buffers, and converting a tetrahedral mesh to an O-buffer.

## Acknowledgments

This work is partially supported by US Office of Naval Research grant N000149710402 and US National Science Foundation grant CCR0306438. The Buddha model is courtesy of Stanford University. The human hand model is courtesy of Clemson University. The authors thank Michael Ashikhmin, Manuel Oliveira, Klaus Mueller, Susan Frank, Ran Shao, and Ankush Kumar for their help.

## REFERENCES


[1] M. Botsch, A. Wiratanaya, and L. Kobbelt, "Efficient High Quality Rendering of Point Sampled Geometry," *Proc. Eurographics Rendering Workshop,* pp. 53-64, 2002.

[2] L. Carpenter, "The A-Buffer, an Antialiased Hidden Surface Method," *Computer Graphics (Proc. ACM SIGGRAPH),* vol. 18, no. 3, pp. 103-108, 1984.

[3] C.-F. Chang, G. Bishop, and A. Lastra, "LDI Tree: A Hierarchical Representation for Image-Based Rendering," *Proc. ACM SIGGRAPH,* pp. 291-298, 1999.

[4] S.F. Frisken, R.N. Perry, A.P. Rockwood, and T.R. Jones, "Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics," *Proc. ACM SIGGRAPH,* pp. 249-254, 2000.

[5] J.P. Grossman and W.J. Dally, "Point Sample Rendering," *Proc. Eurographics Rendering Workshop,* pp. 181-192, 1998.

[6] A. Kaufman, "An Algorithm for 3D Scan-Conversion of Polygons," *Proc. Eurographics,* pp. 197-208, 1987.

[7] K. Kreeger and A. Kaufman, "Hybrid Volume and Polygon Rendering with Cube Hardware," *Proc. Eurographics/SIGGRAPH Workshop Graphics Hardware,* pp. 15-24, 1999.

[8] D. Lischinski and A. Rappoport, "Image-Based Rendering for Non-Diffuse Synthetic Scenes," *Proc. Eurographics Rendering Workshop,* pp. 301-314, 1998.

[9] W.R. Mark, L. McMillan, and G. Bishop, "Post-Rendering 3D Warping," *Proc. Symp. Interactive 3D Graphics,* pp. 7-16, 1997.

[10] L. McMillan, "An Image-Based Approach to Three-Dimensional Computer Graphics," Technical Report TR97-013, Dept. of Computer Science, Univ. of North Carolina-Chapel Hill, 1997.

[11] H. Pfister, M. Zwicker, J. van Baar, and M. Gross, "Surfels: Surface Elements as Rendering Primitives," *Proc. ACM SIGGRAPH,* pp. 335-342, 2000.

[12] V. Popescu, J. Eyles, A. Lastra, J. Steinhurst, N. England, and L. Nyland, "The WarpEngine: An Architecture for the Post-Polygonal Age," *Proc. ACM SIGGRAPH,* pp. 433-442, 2000.

[13] V. Popescu and A. Lastra, "High Quality 3D Image Warping by Separating Visibility from Reconstruction," Technical Report TR99-017, Computer Science Dept., Univ. of North Carolina-Chapel Hill, 1999.

[14] H. Qu, A. Kaufman, R. Shao, and A. Kumar, "A Framework for Sample-Based Rendering with O-Buffers," *Proc. IEEE Visualization,* pp. 441-448, 2003.

[15] H. Qu, M. Wan, J. Qin, and A. Kaufman, "Image Based Rendering with Stable Frame Rates," *Proc. IEEE Visualization,* pp. 251-258, 2000.

[16] M.M. Rafferty, D.G. Aliaga, V. Popescu, and A.A. Lastra, "Images for Accelerating Architectural Walkthroughs," *IEEE Computer Graphics and Applications,* vol. 18, no. 6, pp. 38-45, 1998.

[17] S. Rusinkiewicz and M. Levoy, "Qsplat: A Multiresolution Point Rendering System for Large Meshes," *Proc. ACM SIGGRAPH,* pp. 343-352, 2000.

[18] A. Schilling, "A New Simple and Efficient Antialiasing with Subpixel Masks," *Computer Graphics (Proc. ACM SIGGRAPH),* vol. 25, no. 4, pp. 133-141, 1991.

[19] P. Sen, M. Cammarano, and P. Hanrahan, "Shadow Silhouette Maps," *ACM Trans. Graphics,* vol. 22, no. 3, pp. 521-526, 2003.

[20] J. Shade, S.J. Gortler, L.W. He, and R. Szeliski, "Layered Depth Images," *Proc. SIGGRAPH,* pp. 231-242, 1998.

[21] L. Westover, "Footprint Evaluation for Volume Rendering," *Computer Graphics (Proc. ACM SIGGRAPH),* vol. 24, no. 4, pp. 367-376, 1990.

[22] M. Zwicker, H. Pfister, J. van Baar, and M. Gross, "Surface Splatting," *Proc. ACM SIGGRAPH,* pp. 371-378, 2001.


**Huamin Qu** is a PhD candidate at Stony Brook University (State University of New York at Stony Brook). He received the BS degree (1988) in mathematics from Xi'an Jiaotong University, China, and the MS degree (2000) in computer science from Stony Brook University. His research interests include image-based rendering, point-based rendering, virtual reality, volume rendering, medical imaging, and flight simulation. For more information see http://www.cs.sunysb.edu/~huamin.

**Arie E. Kaufman** received the BS degree (1969) in mathematics and physics from the Hebrew University of Jerusalem, the MS degree (1973) in computer science from the Weizmann Institute of Science, Rehovot, and the PhD degree (1977) in computer science from the Ben-Gurion University, Israel. He is the director of the Center for Visual Computing (CVC), a leading professor and chair of the Computer Science Department, and leading professor of radiology at Stony Brook University (State University of New York at Stony Brook). He was the founding Editor-in-Chief of the *IEEE Transactions on Visualization and Computer Graphics* (*TVCG*), 1995-1998. He was the cochair for multiple Eurographics/SIGGRAPH Graphics Hardware Workshops and Volume Graphics Workshops, the papers/program cochair for the ACM Volume Visualization Symposium and the IEEE Visualization Conferences, and the cofounder and a member of the steering committee of the IEEE Visualization Conference series. He has previously chaired and is currently a director of the IEEE Computer Society Technical Committee on Visualization and Computer Graphics. He is an IEEE fellow and the recipient of a 1995 IEEE Outstanding Contribution Award, 1998 ACM Service Award, 1999 IEEE Computer Society Meritorious Service Award, 2002 State of New York Entrepreneur Award, and 2004 IEEE Harold Wheeler Award. He has conducted research and consulted for more than 30 years specializing in volume visualization; graphics architectures, algorithms, and languages; virtual reality; user interfaces; and multimedia. For more information see http://www.cs.sunysb.edu/~ari.