Depth Image-Based Representation and Compression for Static and Animated 3-D Objects

Leonid Levkovich-Maslyuk, Alexey Ignatenko, Alexander Zhirkov, Anton Konushin, In Kyu Park, *Member, IEEE*, Mahnjin Han, *Member, IEEE*, and Yuri Bayakovski

***Abstract*—This paper describes a new family of three-dimensional (3-D) representations for computer graphics and animation, called depth image-based representations (DIBR), which have been adopted into MPEG-4 Part16: Animation Framework eXtension (AFX). Idea of the approach is to build a compact and photorealistic representation of a 3-D object or scene without using polygonal mesh. Instead, images accompanied by depth values for each pixel are used. This type of representation allows us to build and render novel views of objects and scene with an interactive rate. There are many different methods for the image-based rendering with depths, and the DIBR format is designed to efficiently represent the information necessary for such methods. The main formats of the DIBR family are *SimpleTexture* (an image together with depth array), *PointTexture* (an image with multiple pixels along each line of sight), and *OctreeImage* (octree-like data structure together with a set of images containing viewport parameters). In order to store and transmit the DIBR object, we develop a compression algorithm and bitstream format for *OctreeImage* representation.**

***Index Terms*—Compression, depth image, image-based rendering, image-based representation, MPEG-4 Animation Framework eXtension (AFX), octree, photorealistic representation.**

I. Introduction

THE dominant method of three-dimensional (3-D) objects representation in computer graphics is currently polygonal model. It allows us to approximate an arbitrary shape by colored polygonal mesh. Tremendous progress of software algorithms and development of graphics hardware made it possible to visualize highly realistic still and animated polygonal models of complex objects and scenes in real time.

However, attempts have been made to search for alternative 3-D representations during the last decade. The main reasons for this include the difficulty in constructing polygonal models for real-world objects and unsatisfactory quality in producing a truly photorealistic scene.

Manuscript received July 23, 2003; revised December 10, 2003, and March 17, 2004.

L. Levkovich-Maslyuk is with the Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Moscow 125047, Russian Federation (e-mail: levkovl@spp.keldysh.ru).

A. Ignatenko, A. Zhirkov, A. Konushin, and Y. Bayakovski are with the Graphics and Media Laboratory, Department of Computer Science, Moscow State University, Moscow 119899, Russian Federation (e-mail: ignatenko@graphics.cs.msu.su; azh@graphics.cs.msu.su; ktosh@graphics.cs.msu.su; yurib@graphics.cs.msu.su).

I. K. Park is with the School of Information and Communication Engineering, Inha University, Incheon 402-751, Korea (e-mail: pik@ieee.org)

M. Han is with the Multimedia Laboratory, Samsung Advanced Institute of Technology, Yongin 449-712, Korea (e-mail: manjini@sait.samsung.co.kr).

Digital Object Identifier 10.1109/TCSVT.2004.830676

Graphics applications require enormous amount of polygons; for example, a detailed model of a human body contains several million triangles, which are not easy to handle. Although recent progress in range-finding techniques, such as the laser range scanner, allows us to acquire dense range data with tolerable error, it is still not only very expensive but also very difficult to obtain a seamlessly complete polygonal model of the whole object. On the other hand, rendering algorithms to obtain photorealistic quality are computationally complex and thus far from real-time rendering. Furthermore, the achieved photorealism is rather artificial, making it easy to distinguish even a scene with global illumination from a real-world photo.

All of these have drawn a great deal of attention to the problem of developing 3-D representations based on images of an object and corresponding depth maps, which are arrays of distances from the camera focal plane to the object surface. Such an image-based representation looks algorithmically attractive since it encodes complete information of a colored 3-D object as a collection of two-dimensional (2-D) arrays—simple and regular structures to which popular methods of image processing and compression are readily applicable. Many of them can be hardware-supported. Besides, rendering time for image-based models is proportional to the number of pixels in the reference and output images, but in general not to the geometric complexity as in the polygonal case. In addition, when the image-based representation is applied to real-world objects and scene, photo-realistic rendering of a natural scene becomes possible without use of millions of polygons and expensive computation.

This paper describes a family of data structures, called depth image-based representation (DIBR), which provides effective and efficient methods based mostly on images and depth maps, fully utilizing the advantages described above. Let us briefly characterize the main DIBR formats: *SimpleTexture*, *PointTexture*, and *OctreeImage*.

The DIBR family has been developed for the new version of MPEG standard and adopted into MPEG-4 Part16: Animation Framework eXtension (AFX). AFX provides more enhanced features for synthetic MPEG-4 environments and includes a collection of interoperable tools that produce static and animated 3-D graphics contents. They are based on a virtual reality modeling language (VRML) scene graph as well as compatible with existing MPEG-4. Each AFX tool shows the compatibility with a MPEG-4 binary format for scenes (BIFS) node [1], a synthetic stream [2], and an audio-visual stream [2], [3].

SimpleTexture is a data structure that consists of an image, corresponding depth map, and camera description (its position, orientation, and projection type). Representation capability of a single SimpleTexture is restricted to objects like a facade of a building: a frontal image with a depth map allows us to reconstruct facade views at a substantial range of angles. However, a collection of SimpleTextures produced by properly positioned cameras makes it possible to represent the whole building, in case reference images cover all of the potentially visible parts of the building surface. Of course, the same applies to trees, human figures, cars, etc. Moreover, the union of SimpleTextures provides quite natural means for handling 3-D animated data. In this case, reference images are replaced with reference videostreams. Depth maps for each 3-D frame can be supplied in the alpha-channel of the videostreams or by separate grayscale videostreams. In this type of representation, images can be stored in lossy compressed format like JPEG. This significantly reduces the volume of the color information, especially in the animated case. However, geometry information (depth maps) should be compressed losslessly, avoiding unpleasant visual artifacts.

For objects with complex shape, it is sometimes difficult to cover the whole visible surface with a reasonable number of reference images. A preferable representation in this case might be *PointTexture*. This format also stores reference image and depth map, but in this case both are multivalued. For each line of sight, color and distance are stored for every intersection of the line and the object. The number of intersections might be different from line to line. The union of several PointTextures provides a very detailed representation even for complex objects. However, the format lacks most of the 2-D regularity of SimpleTexture and thus has no natural image-based compressed form. For the same reason, it is used only for still objects.

OctreeImage format occupies an intermediate position between "mostly 2-D" SimpleTexture and "mostly 3-D" PointTexture: it stores geometry of the object in the octree-structured volumetric representation (hierarchically organized voxels of usual octant subdivision of enclosing cube), while the color component is represented by a set of images. This format also contains an additional octree-like data structure, which stores, for each leaf voxel, the index of a reference image containing its color. In the process of rendering the OctreeImage, the color of the leaf voxel is determined by orthographically projecting it onto the corresponding reference image. We have developed a very efficient compression method for the geometry part of OctreeImage. It is a variant of adaptive context-based arithmetic coding in which the contexts are constructed with the explicit usage of geometric nature of the data. Usage of the compression together with lossy compressed reference images makes OctreeImage a very space-efficient representation. OctreeImage has an animated version, in which reference videostreams are used instead of reference images, plus additional stream of octrees representing geometry and voxel-to-image correspondence for each 3-D frame. A very useful feature of OctreeImage format is its built-in mipmapping capability.

DIBR formats have been designed so as to combine advantages of different approaches developed previously, providing a user with flexible tools best suited for a particular task. For example, static SimpleTexture and PointTexture are particular cases of the known formats (see Section II), while OctreeImage is an apparently new one. However, in MPEG-4 context, all three basic DIBR formats can be considered as building blocks, and their combinations by means of MPEG-4 constructs not only embrace many of the image-based representations suggested in the literature, but also give a great potential for constructing new such formats (for example, the *DepthImage* format in both still and animated version in Section III).

The paper is organized as follows. In Section II, we start with a brief review of the previous work in image-based representation and rendering of 3-D objects. Section III describes in detail the DIBRs, developed for MPEG-4 AFX. Formal specifications of MPEG-4 DIBR nodes are given in Section IV. In Section V, we concentrate on the compression method developed for OctreeImage DIBR format. In Section VI, we describe the methods of DIBR rendering. In Section VII, we demonstrate the rendered images of still and animated 3-D objects in DIBR formats and present the rendering speed and typical compression results. Finally, we conclude in Section VIII.

II. Brief Overview of Image-Based Representations and Compression

Image-based representations have been developed in parallel with corresponding rendering methods. One of the earliest ideas in this field is the method of *generalized sprites* [5], [6], oriented mostly on virtual walkthroughs. Generalized sprite is a view of a part of a 3-D scene, for which changes caused by camera displacement are computed with the aid of affine transformation of the original view, instead of using 3-D geometry-based computation. This amounts to texture mapping for some proper planar regions in order to imitate actually nonplanar geometry. Distortion introduced by such a simplification is more noticeable when the camera displacement becomes larger. Also, the greater the *depth* (distance from the observer) variation is in the part of the scene represented by the sprite, the more severe the distortion is. On the other hand, the more distant the sprites are , the less visible the distortion is. This allows to obtain satisfactory quality for distant areas of such scenes as architectural environments.

Sprites with depth was introduced in [8] as an improvement over the planar sprites. This approach combines an explicit account for the depth variation in the sprite region with uniform (i.e., using single matrix) texture mapping. The position of each pixel is correctly computed by shifting according to its depth, which helps to achieve correct parallax by warping the image. When the viewing angle is large enough, direct rendering of such a model leads to holes in the image. This hole-filling problem arising in image-based techniques that use depth maps is commonly solved by a combination of filtering and *splatting*. Splatting is a technique of using small flat color patches (splats) with variable (in general) transparency as "rendering primitives."

Relief Texture (RT), introduced in [7], is a method of rendering a single image with a depth map. However, the model itself is also frequently called relief texture (see Fig. 1). The RT method features a fast "incremental" warping algorithm, which computes views of the object through the faces of the enclosing

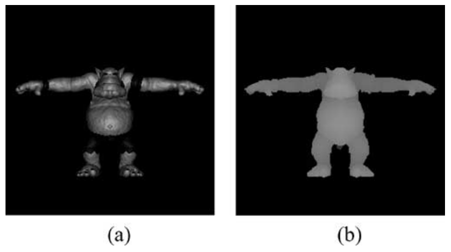


Fig. 1. Relief texture image and depth map.

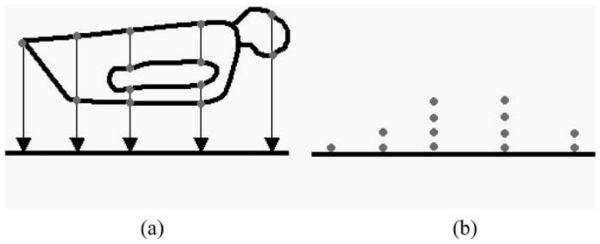


Fig. 2. Layered depth image (LDI). (a) Projection of the object. (b) Layered pixels.

cube. However, the RT algorithm substantially relies on the particular geometry of the cube. On the other hand, there is no way to render the parts of the object that are invisible from any of the six faces. This makes it impossible to use the RT together with fast warping algorithm for rendering noncubic, "adaptive" configurations of images with a depth that covers the visible surface.

Layered depth image (LDI) [8] was introduced to solve the problem of invisible parts of the surface. It was designed so as to use the fast warping algorithm [9], [10]. LDI is a data structure for multivalued depth map and image, corresponding to a single projection of a 3-D object (Fig. 2). This structure stores color and depth for each point of intersection of each projection ray with the object (a set of rays is properly discretized for either orthogonal or perspective projection). This allows to represent all parts of the surface, including those invisible from the particular viewpoint. A modification of this approach uses several LDIs constructed for the same object and different cameras [11], [12]. Rendering algorithms suggested for these representations provided better visual quality at a cost of slower rendering.

Depth image-based representations are closely connected with volumetric and point-based representations. An important example is *LDI tree* [13], which is an octree with an LDI attached to each cell (bounding box for the part of the scene (object) represented by this particular LDI). This representation is more like a hierarchical point structure than the original LDI format. Another important example of structures of this kind is *Qsplat* [14], a multiresolution point system based on hierarchical structure of spheres of variable size. An even more complex model is based on *surfels* [15], a hierarchically structured representation of a 3-D object with the aid of local surface elements whose attributes include color, normal vectors, and depth. The spatial cell of the corresponding tree structure contains projections of the object surface structured as three LDIs with the same bounding box. One of the main properties of the method is a sophisticated filtering algorithm applied to nonuniformly distributed screen samples.

To complete this brief review, we must mention the important class of image-based representations that use very little or no information on geometry. Such approaches as Lumigraph [16] and light field rendering [17] employ a large amount of reference views from many viewpoints, unified in a huge collection of rays. Views of the scene from arbitrary viewpoints are computed by interpolation of necessary rays from nearby rays belonging to the collection. A coarse depth structure is used to optimize nonuniform sampling of reference views.

It should be noted that few image-based methods have been suggested for 3-D animation so far, and most of them are for restricted classes of 3-D objects. For example, in [18], the idea of facial image modification for 3-D face geometry was developed. In [19], architectural scenes were animated with the aid of view-dependent texture mapping.

Compression of image-based representations has been given rather little attention in the literature. Compression is an essential part of such formats as Lumigraph and light fields. In order to reduce the huge size of necessary data, adaptive sampling and interpolation have been used in [16] and [17]. In [14], the point-based Qsplat model is itself considered a compressed representation with respect to the (hypothetical) polygonal model of the same object. A direct application of the most simple compression techniques like run-length coding and Huffman coding to the Qsplat model is mentioned as an option in [14], but is not investigated in detail. On the contrary, in this paper we give a great deal of attention to compression issues. We present a novel technique for efficient lossless compression of a linkless octree data structure.

A noticeable piece of work on voxel model compression is that by Kim and Lee [26], in which an efficient lossless method for surface voxel models compression has been developed. It uses a library of several hundred standard patterns of black voxels in $3 \times 3 \times 3$ subcubes and employs strong correlation of the neighbor patterns in voxel surfaces. It is more efficient on binary (black/white) surface voxel models than our approaches are. However, our representations are oriented on arbitrary objects (not only surfaces), allowing it to handle the color, and (in case of OctreeImage) possess additional multiresolution structure for 3-D mipmapping.

III. Depth Image-Based Representation

Considering the ideas outlined in the previous section as well as our previous development [20], we suggest the following set of image-based formats for use in MPEG-4 AFX: *SimpleTexture, PointTexture,* and *OctreeImage*. Note that *SimpleTexture* and *OctreeImage* have animated versions.

A. SimpleTexture and PointTexture

SimpleTexture is a single image combined with a depth image. It is equivalent to RT, while PointTexture is equivalent to LDI. Based on SimpleTexture and PointTexture as building blocks, we can construct a variety of representations using MPEG-4 constructs. Formal specification will be given in Section IV, and here we describe the result geometrically.

A *DepthImage* structure defines either SimpleTexture or PointTexture together with bounding box, position in space,

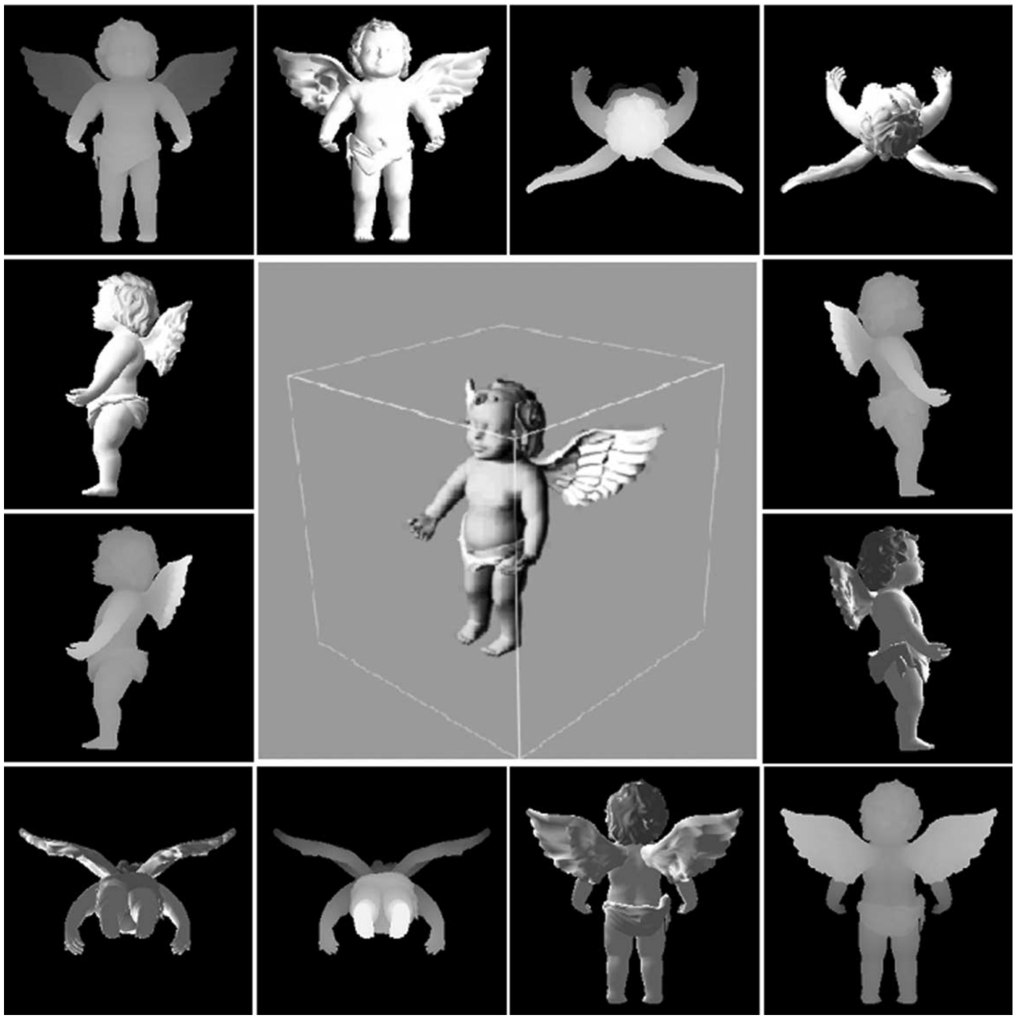


Fig. 3. Example of box texture. Six SimpleTextures (pairs of image and depth map) are used to render the model shown in the center.

and some other information. A set of DepthImages can be unified under a single structure called the Transform node, and this allows us to construct a variety of useful representations. Most commonly used are two that do not have a specific MPEG-4 name, but in our practice we called them Box Texture (BT) and Generalized Box Texture (GBT). BT is a union of six SimpleTextures corresponding to a bounding cube of an object or a scene, while GBT is an arbitrary union of any number of SimpleTextures that together provide a consistent 3-D representation. An example of BT is given in Fig. 3, where reference images, depth maps, and the resulting 3-D object are shown. BT can be rendered with the aid of incremental warping algorithm [6], but we use a different approach applicable to GBT as well (details of this rendering method are given in Section VI). An example of GBT representation is shown in Fig. 4, where 21 SimpleTextures are used to represent the complex object.

It should be noted that unification mechanism allows, for instance, to use several LDIs with different cameras to represent the same object or parts of the same object. Hence, data structures like image-based objects, cells of LDI tree, and cells of surfels-based tree structure are all particular cases of this format, which obviously offers much greater flexibility in adapting location and resolution of SimpleTextures and PointTextures to the structure of the scene.

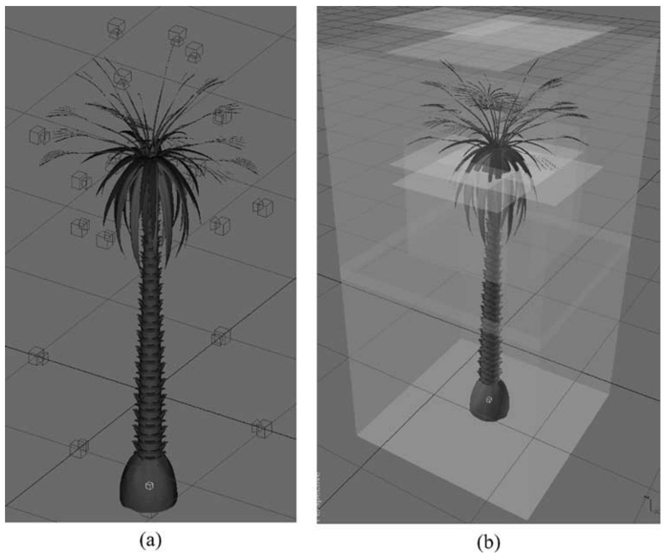


Fig. 4. Generalized box texture (GBT). (a) Camera locations. (b) Reference image planes for the same model (21 SimpleTextures are used).

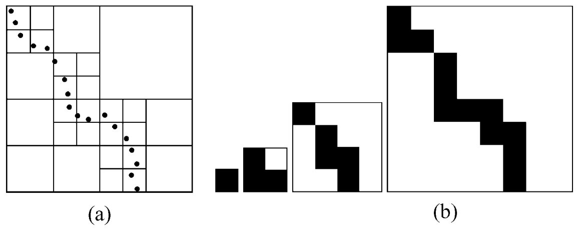


Fig. 5. Octree representation illustrated in 2-D. (a) Point Cloud. (b) Mipmaps.

B. OctreeImage: Textured Binary Volumetric Octree

In order to utilize multiresolution geometry and texture with more flexible representation and fast rendering, we develop OctreeImage representation, which is based on textured binary volumetric octree (TBVO). The objective of TBVO is to contrive a flexible representation/compression format with fast and high-quality visualization. TBVO consists of three main components: binary volumetric octree (BVO) which represents geometry, a set of reference images, and data structure of image indices corresponding to the octree nodes.

Geometric information in BVO form is a set of binary (occupied or empty) regularly spaced voxels combined in larger cells in usual octree manner. This representation can be easily obtained from DepthImage data through the intermediate "point cloud" form, since each pixel with depth defines a unique point in 3-D space. Conversion of the point cloud to BVO is illustrated in Fig. 5. An analogous process allows converting the polygonal model to BVO [20]. Texture information of the BVO can be retrieved from reference images. A reference image is texture of voxels at a given camera position and orientation. Hence, BVO itself, together with reference images, does already provide the model representation [20]. However, it turned out that the additional structure storing the reference image index for each BVO allows much faster visualization with better quality.

The main BVO visualization problem is that we must determine the corresponding camera index of each voxel during rendering. To this end, we must at least determine the existence of a camera, from which the voxel is visible. This procedure is very slow if we use a brute-force approach. In addition to this problem, there are still some troubles for voxels that are not visible from any camera, yielding undesirable artifacts in the rendered image.

A possible solution could be storing explicit color to each voxel. However, in this case, we have experienced some problems in compressing color information. That is, if we group voxel colors as an image format and compress it, the color correlation of neighboring voxels is destroyed such that the compression ratio would be unsatisfactory.

In TBVO, the problem is solved by storing a camera (image) index for every voxel. The index enables to determine the voxel color quickly at the rendering stage. Suppose the camera index of a voxel equals $\boldsymbol{i}$; then, in order to find the voxel color, we just project the voxel orthographically on the $\boldsymbol{i}$th reference image. The pixel it hits contains the necessary color.

Usually, the camera index is identical for relatively large groups of voxels, since many neighboring voxels are often visible from the same reference image. This allows the use of an octree structure for economic storage of the additional

```
if CurNode is not leaf node
   write current BVO-symbol corresponding to this node
if all the children have identical image index (texture-symbol)
   if parent of CurNode has '?' image index
         write image index equal for sub-nodes
else
   write '?' symbol
```

Fig. 6. Pseudocode for writing the TBVO bitstream.

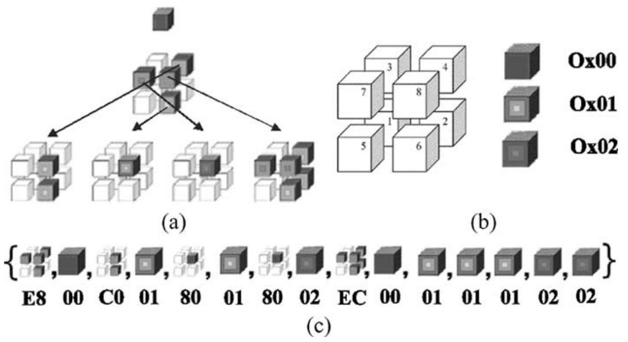


Fig. 7. Example of writing a TBVO stream. (a) TBVO tree structure. Uniform gray color is "undefined" texture symbol. Each color denotes the camera index. (b) Octree traversal order in a BVO node and camera indices. (c) TBVO stream. Filled cubes and octree cube denote the texture-bytes and BVO-bytes, respectively.

information, i.e., a set of camera indices. The representation we use is explained in detail in Section III-C. Note that it has been observed in the experiments that the amount of data increases only 15% on the average, compared with the representation using only BVO and reference images. Process of construction of a TBVO model of an object is a bit more complex in comparison with producing a BVO model, but the resulting model is visualized much faster, and it has better visual quality for the objects of complex geometry.

C. Streaming of TBVO

We suppose that 255 cameras are sufficient and assign up to 1 byte for the camera index. The TBVO stream is a stream of symbols. Every TBVO symbol is either a BVO-symbol or texture-symbol. Texture-symbol denotes camera index, which can be a specific number or an "undefined"code.

Let "undefined" code be "?" in any further description. The TBVO stream is traversed in breadth first order. Let us describe how to write a TBVO stream if we have BVO and every leaf voxel has an image index (the indices are assigned in process of the model creation). We should traverse all BVO nodes including leaf nodes (which do not have BVO-symbol) in breadth first order. In Fig. 6, the pseudocode which completes writing the stream is shown.

An example of writing the TBVO bitstream is shown in Fig. 7. For the TBVO tree shown in Fig. 7(a), a stream of symbols can be obtained as shown in Fig. 7(c), according to the procedure. In this example, the texture-symbols are represented in 1 byte. However, in the actual stream, each texture-symbol would only need 2 bits because we only need to represent three values (two cameras and the undefined code).

```
DepthImage {
   field     SFVec3f      position      0    0    10
   field     SFRotation   orientation   0    0    1    0
   field     SFVec2f      fieldOfView   0.785398   0.785398
   field     SFFloat      nearPlane     10
   field     SFFloat      farPlane      100
   field     SFBool       orthographic  TRUE
   field     SFDepthTextureNode    diTexture     NULL
}

SimpleTexture {
   field   SFTextureNode     texture      NULL
   field   SFTextureNode     depth        NULL
}

PointTexture {
   field   SFInt32   width             256
   field   SFInt32   height            256
   field   MFInt32   depth             []
   field   MFColor   color             []
   field   SFInt32   depthNbBits        7
}

OctreeImage {
   field   SFInt32           octreeResolution  256
   field   MFInt32           octree            []
   field   MFInt32           voxelImageIndex   []
   field   MFDepthImageNode  images            []
}
```

Fig. 8. Specification of the DIBR nodes.

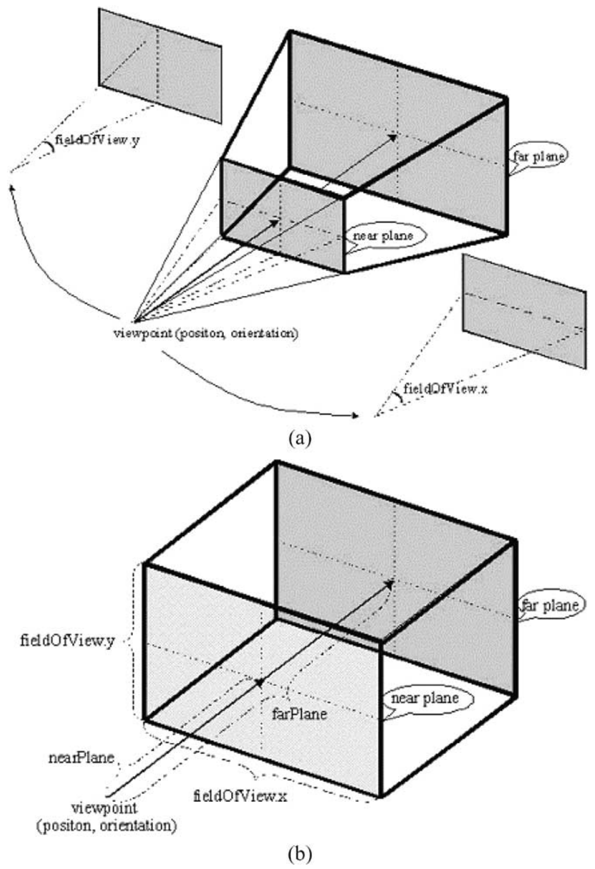


Fig. 9. View volume model for DepthImage. (a) Perspective view. (b) Orthographic view.

D. DIBR Animation

Animated versions are defined for two of the DIBR formats: DepthImage containing only SimpleTextures, and OctreeImage. Data volume is one of the crucial issues with 3-D animation. We have chosen these particular formats since video streams can be naturally incorporated in the animated versions, providing substantial data reduction.

For DepthImage, animation is performed by replacing reference images by MPEG-4 MovieTextures. High-quality lossy video compression does not seriously affect appearance of the resulting 3-D objects. Depth maps can be stored (in near lossless mode) in the alpha channels of reference video streams. Lossless or near-lossless compression for depth maps is used because distortions in geometry due to lossy compression are usually much more noticeable than distortions in reference images are when the model is viewed in three dimensions. At the rendering stage, a 3-D frame is rendered after all the reference images and depth frames are received and decompressed.

Animation of OctreeImage is similar—reference images are replaced by MPEG-4 MovieTextures, and a new stream of octree appears.

IV. MPEG-4 Node Specification

The DIBR formats are described in detail in MPEG-4 AFX nodes specifications [4]. DepthImage contains the fields determining the parameters of view frustum for either SimpleTexture or PointTexture. OctreeImage node represents an object in the form of TBVO-defined geometry and a set of reference images format. Scene-dependent information is stored in special fields of the DIBR data structures, allowing the correct interaction of DIBR objects with the rest of the scene. The definition of DIBR nodes is shown in Fig. 8.

Fig. 9 illustrates spatial layout of the DepthImage, in which the meaning of each field is shown. Note that the DepthImage node defines a single DIBR object. When multiple DepthImage nodes are related to each other, they are processed as a group, and thus, should be placed under the same Transform node. The diTexture field specifies the texture with depth (SimpleTexture or PointTexture), which shall be mapped into the region defined in the DepthImage node.

The OctreeImage node defines an octree structure and the projected textures. The octreeResolution field specifies maximum number of octree leaves along a side of the enclosing cube. The octree field specifies a set of octree internal nodes. Each internal node is represented by a byte. A “1” in ith bit of this byte means that the children nodes exist for the ith child of that internal node, while “0” means that they do not. The order of the octree internal nodes shall be the order of breadth first traversal of the octree. The order of eight children of an internal node is shown in Fig. 7(b). The voxelImageIndex field contains an array of image indices assigned to voxel. At the rendering stage, color attributed to an octree leaf is determined by orthographically projecting the leaf onto one of the images with a particular index. The indices are stored in an octree-like fashion: if a particular image can be used for all of the leaves contained

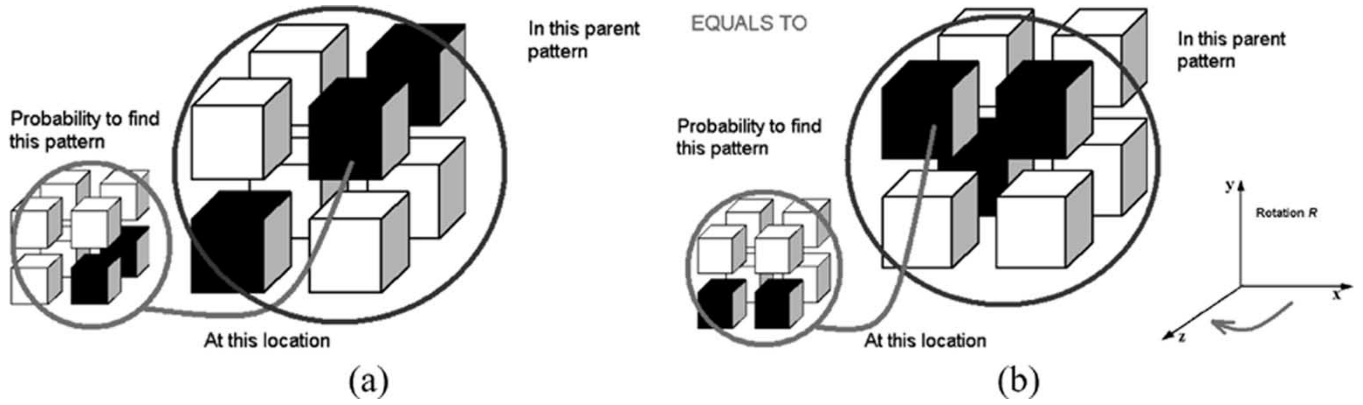


Fig. 10. Orthogonal invariance of node occurrence probability. (a) Original current and parent node. (b) Current and parent nodes, rotated around y axis by 90°.

in a specific voxel, the voxel containing index of the image is issued into the stream; otherwise, the voxel containing a fixed "further subdivision" code is issued, which means that the image index will be specified separately for each child of the current voxel (in the same recursive fashion). If the voxelImageIndex is empty, then the image indices are determined during the rendering stage. The images field specifies a set of DepthImage nodes with SimpleTexture in the diTexture field. In this case, the nearPlane and farPlane fields of the DepthImage node and the depth field in the SimpleTexture node are not used.

V. Compression of OctreeImage Format

In this section, we consider a compression method for OctreeImage. Typical test results are presented and discussed in Section VII. Please note that the compression of PointTexture is not supported yet, and will be implemented in the next version of AFX.

The fields *octreeimages* and *octree* in OctreeImage are compressed separately. The proposed methods have been developed based on the notion that *octree* field must be compressed losslessly while some degree of visually acceptable distortion can be allowed for octreeimages.

A. OctreeImages Field Compression

The OctreeImages field is compressed by means of image compression (for static model) or video compression tools (for animated model) supported by MPEG-4. In our approach, we used the JPEG format for OctreeImages. Additional preprocessing of images by discarding irrelevant pixels and suppressing compression artifacts at the object/background boundary (see Section VII for details) increases both the compression rate and rendering quality simultaneously.

B. Octree Field Compression

Octree compression is the most important part of the OctreeImage compression, since it deals with compression of linkless binary tree representation which is already very compact. In our approach, the method explained below reduces the volume of this structure to about half of the original. In the animated OctreeImage version, Octree field is compressed separately frame by frame.

1) Context Model: Compression is performed by a variant of context-based adaptive arithmetic coding that makes *explicit* use of the geometric nature of the data. The *Octree* is a stream of bytes. Each byte represents a node (i.e., subcube) of the tree, in which its bits indicate the occupancy of the subcube after internal subdivision. The bit pattern is called *filling pattern* of the node. The proposed compression algorithm processes bytes one by one, in the following manner.

— a *context* for the current byte is determined
— "probability" (normalized frequency) of occurrence of the current byte in this context is retrieved from the "probability table" (*PT*) corresponding to the context.
— the probability value is fed to the arithmetic coder.
— current PT is updated by adding a specified step to the frequency of the current byte occurrence in the current context (and, if necessary, renormalized afterwards, see details below).

Thus, coding is the process of constructing and updating the PTs according to the context model. In the context-based adaptive arithmetic coding schemes (such as "Prediction with Partial Matching" [23], [24]), context of a symbol is usually a string of several preceding symbols. However, in our case, compression efficiency is increased by exploiting the octree structure and geometric nature of the data. The proposed approach is based on the two ideas that are apparently new in the problem of octree compression.

1) For the current node, the context is either its *parent node*, or the pair {*parent node, current node position in the parent node*};
2) It is assumed that the "probability" of the given node occurrence at the particular geometric location in the particular parent node is *invariant* with respect to a certain set of orthogonal (such as rotations or symmetries) transforms.

Assumption 1) is illustrated in Fig. 10 for the transform R, which is the rotation by $-90°$ on the x–z plane. The basic notion behind 2) is the observation that probability of occurrence of a particular type of child node in a particular type of parent node should depend only on their *relative* position. This assumption is confirmed in our experiments by analysis of probability tables. It allows us to use more complex context without having too many probability tables. This, in turn, helps to achieve quite good results in terms of data size and speed. Note that the more

TABLE I
ENUMERATION OF PROBABILITY TABLES

ID of PTs	0	1	…	255	Context description
0	$P_{0,0}$	$P_{0,1}$	…	$P_{0,255}$	0-CONTEXT : CONTEXT INDEPENDENT
1..22 (22)	$P_{I,0}$	$P_{I,1}$	…	$P_{I,255}$	1-CONTEXT : {*PARENT NODE CLASS*}
23...198 (176)	$P_{J,0}$	$P_{J,1}$	…	$P_{J,255}$	2-CONTEXT : {*PARENT NODE CLASS, CURRENT NODE POSITION*}

complex contexts are used, the sharper the estimated probability is, and thus the more compact the code is. A heuristic explanation of "transition probabilities" invariance lies in the fact that image-based representations deal with visible surfaces of the 3-D objects, i.e., with more or less smooth 2-D data. Suppose that a small 2-D patch is represented by a certain small subtree of fine level voxels (note that fine level voxels determine the overall statistics). Then one can expect that a version of this patch rotated by one of the orthogonal transforms defined below occurs elsewhere in the model surface. In this case, it will be represented by the rotated version of the same small subtree.

Let us introduce the set of transforms for which we will assume the invariance of probability distributions. In order to apply in our situation, such transforms should preserve the enclosing cube.

Consider a set G of the orthogonal transforms in Euclidean space, which are obtained by all compositions in any number and order of the three basis transforms (*generators*) m_1, m_2, and m_3, given by

$$\mathbf{m}_1 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{m}_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$$\mathbf{m}_3 = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{1}$$

In (1), $\mathbf{m}_1$ and $\mathbf{m}_2$ are reflections to the planes $x = y$ and $y = z$, respectively, and m_3 is reflection to the plane $x = 0$. One of the classical results of the theory of groups generated by reflections [25] states that G contains 48 distinct orthogonal transforms and is, in a sense, the maximal group of orthogonal transforms that take the cube into itself (so-called *Coxeter group* [25]). For example, rotation R in Fig. 10 is expressed through the generators as

$$\mathbf{R} = \mathbf{m}_3 \cdot \mathbf{m}_2 \cdot \mathbf{m}_1 \cdot \mathbf{m}_2 \tag{2}$$

where "$\cdot$" is matrix multiplication.

Transform from G, applied to an octree node, produces a node with a different filling pattern of subcubes. This allows to categorize the nodes according to the filling pattern of their subcubes. Using the group theory language [25], we say that *G acts on the set* of all filling patterns of the octree nodes. Computations show that there exist 22 distinct *classes* (also called *orbits* in group theory), in which, by definition, *two nodes belong to the same class, if and only if they are connected by a transform from G*. The number of elements in a class varies from 1 to 24 (and is always, in accordance with group theory, a divisor of 48).

The practical consequence of assumption 2) is that the probability table depends *not* on the parent node itself, but only on the class to which the parent node belongs. Note that there would be 256 tables for a parent-based context and additional $256 \times 8 = 2048$ tables for parent-and-child position-based context in the former case, while we need only 22 tables for parent-class-based context plus $22 \times 8 = 176$ tables in the latter case. Therefore, it is possible to use equivalently complex context with a relatively small number of probability tables. The constructed PT would have the form as shown in Table I.

2) Encoding Process: To make the statistics for probability tables more accurate, they are collected in different ways at three stages of encoding process.

At the first stage, we do not use contexts at all, accepting the "*0-context model*" and keep a single probability table with 256 entries, starting from the uniform distribution;

As soon as the first 512 nodes (it is an empirically found number) are encoded, we switch to the "*1-context model*" using parent node as a context. At the switching moment, the 0-context PT is copied to the PTs for all 22 contexts.

After 2048 nodes (another heuristic value) are encoded, we switch to "*2-context model*." At this moment, the 1-context PTs of the parent patterns are copied to the PTs for each position in the same parent pattern.

Key point of the algorithm is the determination of context and probability for the current byte. This is implemented as follows. In each class, we fix a single element, which is called "*standard element*." We store a class map table (CMT) indicating the class to which each of the possible 256 nodes belongs and the precomputed transform from G that takes this particular node into the standard element of its class. Thus, in order to determine the probability of the current node N, we perform the following steps.

Step 1) Look at the parent P of the current node.

Step 2) Retrieve the class from CMT, to which P belongs, and the transform T that takes P into the standard node of the class. Let the class number be c.

Step 3) Apply T to P and find the child position p in standard node to which current node N is mapped.

Step 4) Apply T to N. Then, newly obtained filling pattern TN is at the position p in the standard node of the class c.

Step 5) Retrieve the required probability from the entry TN of the probability table corresponding to the class-position combination (c, p).

Step 6) For the 1-context model, the above steps are modified in an obvious way. Needless to say, all the transforms are precomputed, and implemented in a lookup table.

Note that at the stage of *decoding* of the node N its parent P is already decoded, and hence transform T is known. All of the steps at the stage of decoding are absolutely similar to the corresponding encoding steps.

Finally, let us outline the probability update process. Let P be a probability table for some context. $P(N)$ is the entry of P corresponding to the probability of occurrence of the node N in this context. In our implementation, $P(N)$ is an integer, and after each occurrence of N, $P(N)$ is updated as

$$P(N) = P(N) + A \tag{3}$$

where A is an integer increment parameter varying typically from 1 to 4 for different context models. Let $S(P)$ be the sum of all entries in P. Then the "probability" of N that is fed to the arithmetic coder is computed as $P(N)/S(P)$. As soon as $S(P)$ reaches a threshold value 2^{16}, all the entries are renormalized: in order to avoid occurrence of zero values in P, entries equal to 1 are left intact, while the others are divided by 2.

C. *VoxelImageIndex Field Compression*

The stream of symbols determining the image index for each voxel is compressed using its own probability table. In the terms used above, it has a single context. PT entries are updated with larger increment than entries for octree nodes: this allows to adapt the probabilities to high variability of the involved symbol frequencies; in the rest, there is no difference with node symbols coding.

VI. Rendering

Rendering methods for DIBR formats are not part of AFX, but it is necessary to explain the ideas used to achieve simplicity, speed, and quality of DIBR objects rendering. Our rendering methods are based on *splats*, small flat color patches used as "rendering primitives." In Section II, we mentioned this approach as widely used in image-based representations. Two approaches outlined below are oriented at two different representations: DepthImage and OctreeImage. In our implementation, OpenGL functions are employed for splatting to accelerate the rendering. Note that software rendering is also possible and enables to optimize computations using the simple structure of DepthImage or OctreeImage.

A. *Rendering DepthImage Objects*

The method we use for rendering DepthImage objects is extremely simple. It should be mentioned, however, that it depends on the OpenGL functions and works much faster with the aid of a hardware accelerator. In this method, we transform all of the pixels with depth from SimpleTextures and PointTextures that are to be rendered into 3-D points, then position small polygons (splats) at these points, and apply rendering functions of OpenGL. Pseudocode of this procedure for SimpleTexture case is given in Fig. 11. PointTexture case is treated exactly in the same way.

The size of the splat must be adapted to the distance between the point and the observer. We used the following simple approach. First, the enclosing cube of a given 3-D object is subdivided into a coarse uniform grid. The splat size is computed for each cell of the grid, and this value is used for the points inside the cell. The computation is performed as follows.

```
// Scan all points of the SimpleTexture
for y=0 y<image_height y++
    for x=0 x<image_width x++

    // Check if the point belongs to the object
    // projection
    if depth_map(x,y) != 0
    {
        // Compute 3D coordinates
        // up, right,dir,center - camera
        // orientation and location vectors
        point3d = up*y + right*x +
                  dir*depth_map(x,y) + center

        // Get colors
        color = color_map(x,y)

        // Visualize using OpenGL
        Splat point3d with a pre-computed radius
        using OpenGL functions.
    }
```

Fig. 11. Pseudocode of OpenGL-based rendering of SimpleTexture.

— Map the cell on the screen by means of OpenGL.
— "Calculate length L of the largest diagonal of projection (in pixels).
— Estimate D (splat diameter) as CL/N, where N is average number of points per cell side and C is a heuristic constant, approximately **1.3.**

This method could certainly be improved by sharper radius computations, more complex splats, and antialiasing, although even this simple approach provides good visual quality.

B. *Rendering OctreeImage Objects*

The same approach works for OctreeImage, where the nodes of the octree at one of coarser levels are used in the above computations of splat size. However, for the OctreeImage, color information should first be mapped on the set of voxels. This can be done very easily, because each voxel has its corresponding reference image index. The pixel position in a reference image is also known during the parsing of the octree stream. As soon as the colors of OctreeImage voxels are determined, splat sizes are estimated and the OpenGL-based rendering is used as described above.

VII. Experimental Results

DIBR formats have been implemented and tested on several 3-D models. One of the models ("Tower") was obtained by scanning actual physical object (Cyberware color 3-D scanner was used), and the others have been converted from the 3DS-MAX demo package. Tests have been performed on an Intel Pentium-IV 1.8-GHz PC with an OpenGL accelerator.

TABLE II
STATIC DIBR MODELS COMPRESSION. MODEL SIZES ARE IN KILOBYTES

Model		**Palm512**		**Angel256**		**Morton512**		**Tower256**	
Number of SimpleTextures		21		6		6		5	
Size of original 3DS-MAX Model (ZIP- archived)		4040		151		519		N/A	
DepthImage Size		3319		141		838		236	
Depth	Images	1903	1416	41	100	519	319	118	118
OctreeImage Size		267		75		171		83.4	
Compressed Octrees	Images	135	132	38.5	36.5	88	83	47.4	36

TABLE III
COMPRESSION RESULTS FOR OCTREE AND VOXELIMAGEINDEX FIELDS IN OCTREEIMAGE FORMAT. FILE SIZES ARE ROUNDED TO KILOBYTES

Model	**Number of Ref. Images**	**Size of Octree and voxelImageIndex Component**		**Compression Ratio**
		Uncompressed	**Compressed**	
ANGEL256	6	81.5	38.5	2.1
	12	86.2	41.7	2.1
MORTON512	6	262.2	103.9	2.5
	12	171.0	88.0	2.0
PALMS512	6	198.4	85.8	2.3
	12	185.1	83.1	2.2
ROBOT512	6	280.4	111.9	2.5
	12	287.5	121.2	2.4

In the following sections, we explain the methods of conversion from polygonal to DIBR formats and then present the modeling, representation, and compression results of the different DIBR formats. Most of the data is for DepthImage and OctreeImage models; these formats have animated versions and can be effectively compressed. All of the presented models have been constructed with the orthographic camera since it is, in general, a preferable way to represent "compact" objects. Note that the perspective camera is used mostly for economic DIBR representation of the distant environments.

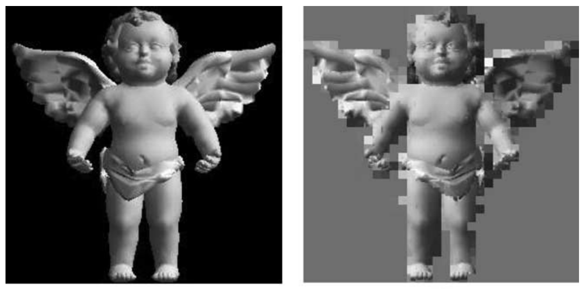

(a) (b)

Fig. 12. Compression of reference image in SimpleTexture. (a) Original reference image. (b) Modified reference image in JPEG format.

A. Generation of DIBR Models

DIBR model generation begins with obtaining a sufficient number of SimpleTextures. For a polygonal object, the SimpleTextures are computed, while for the real-world object the data are obtained from digital cameras and scanning devices. The next step depends on the DIBR format we want to use.

1) DepthImage: DepthImage is simply a union of the obtained SimpleTextures. Depth maps are stored in losslessly compressed form, because even small magnitude distortions in geometry, especially those unmatched in different but overlapping depth maps, usually lead to artifacts like holes and sharp peaks or edges which are clearly noticeable in three dimensions.

Reference images can be stored in lossy compressed form, but in this case a preprocessing is required. While it is generally tolerable to use popular methods like JPEG lossy compression, the boundary artifacts become more noticeable in the 3-D object views generated, especially due to the boundaries between object and background of the reference image, where the background color appears to "spill" into the object. The solution we have used to cope with the problem is to extend the image in the boundary blocks into the background using average color of the block and fast decay of intensity, and then apply the JPEG compression. The effect resembles "squeezing" the distortion into the background where it is harmless since background pixels are not used for rendering. Internal boundaries in lossy compressed reference images may also produce artifacts, but these are generally less visible.

2) OctreeImage: To generate OctreeImage models, we use an intermediate point-based representation (PBR). Set of points that constitute a PBR is a union of the colored points obtained

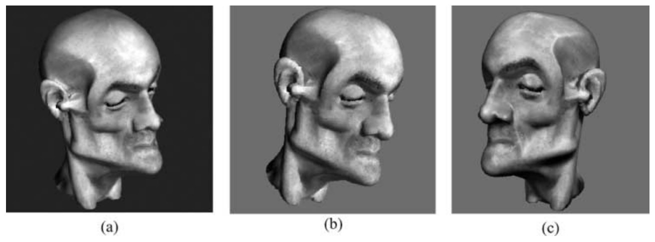


Fig. 13. "Morton" model in different formats. (a) Original polygonal format. (b) DepthImage model. (c) OctreeImage model.

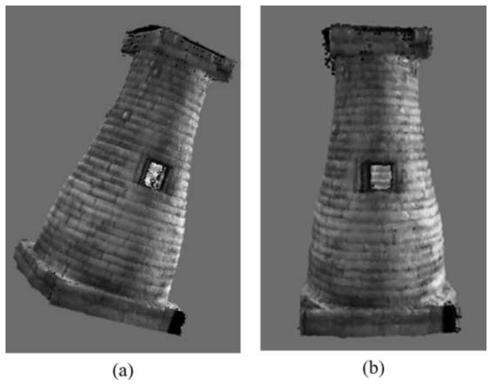


Fig. 14. Rendering examples. (a) Scanned "Tower" model, DepthImage. (b) The same model, OctreeImage (scanner data have been used without noise removal, hence the black dots are observed in the upper part of the model).

by shifting pixels in reference images by distances specified in the corresponding depth maps. Original SimpleTextures should be constructed so that the resulting PBR would provide sufficiently accurate approximation of the object surface. After that, PBR is converted into OctreeImage as outlined in Fig. 5 and is used to generate a new complete set of reference images that satisfy restrictions imposed by this format (see Section IV). At the same time, additional data structure voxelImageIndex (see Sections III-B and III-C) representing reference image indices for octree voxels, is generated. In case the reference images must be stored in a lossy format, they are first preprocessed as explained in the previous subsection. Besides, since a TBVO structure explicitly specifies the pixel containing its color of each voxel, redundant pixels are discarded, which further reduces the volume of voxelImageIndex. Examples of the original and modified reference image in JPEG format are shown in Fig. 12.

Note that quality degradation due to lossy compression is negligible for OctreeImages, but sometimes still noticeable for DepthImage objects.

3) PointTexture: PointTexture models are constructed using projection of the object onto a reference plane, as explained in Section III-A. If this does not produce enough samples (which may be the case for the surface parts nearly tangent to vector of projection), additional SimpleTextures are constructed to provide more samples. The obtained set of points is then reorganized into the PointTexture structure.

B. Compression and Rendering

1) Compression: In Table II, we compare data sizes of the several polygonal models and their DIBR versions. Numbers in the model names denote the resolution (in pixels) of their reference images.

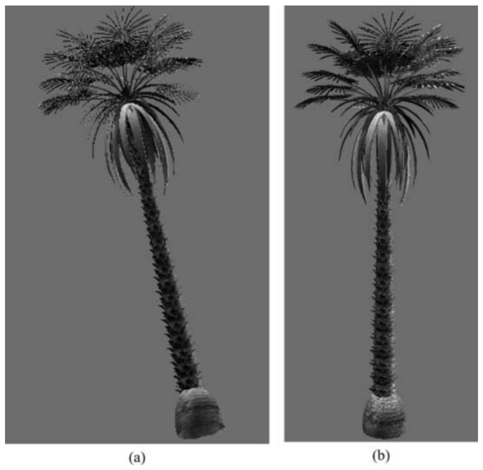


Fig. 15. Rendering examples. (a) "Palm" model, original polygonal format. (b) Same model, DepthImage.

Fig. 16. Rendering examples. A frame from "Dragon512" animation in OctreeImage.

Fig. 17. Rendering examples. "Angel512" in PointTexture.

Depth maps in DepthImages have been stored in PNG format, while reference images are stored in high-quality JPEG. The data in Table II indicate that DepthImage model size is not always smaller than that of the archived polygonal model. However, compression provided by OctreeImage is usually much higher. This is a consequence of unification of depth maps into a single efficiently compressed octree data structure, as well as

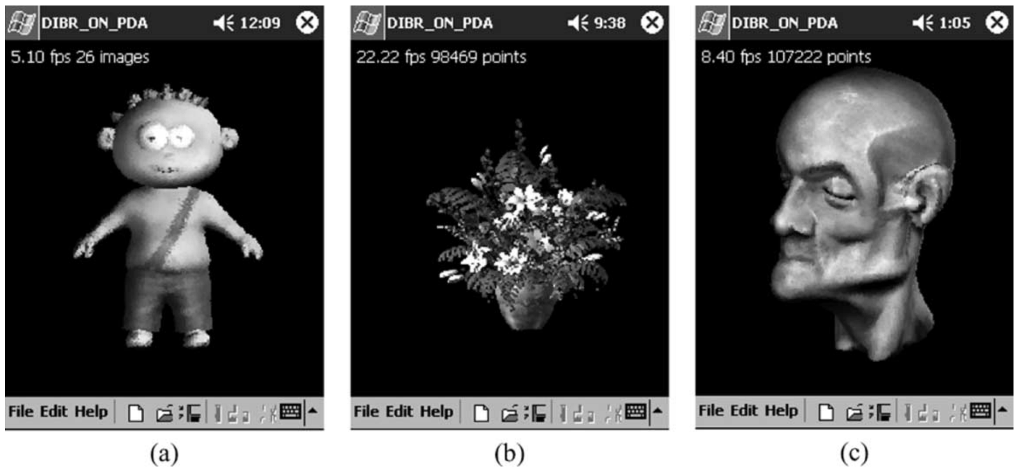


Fig. 18. Rendering of DIBR objects on a PDA. (a) "Boo" model with 26 SimpleTextures. (b) "Flower" model in PointTexture with 98 469 points. (c) "Morton" model in OctreeImage.

of sophisticated preprocessing which removes redundant pixels from reference images. On the other hand, DepthImage structure provides simple and universal means for representing complex objects like "Palm" without difficult preprocessing.

Table III presents OctreeImage-specific data, giving the idea of efficiency of the compression developed for this format. Table entries are data sizes of compressed and uncompressed parts of the models comprising octree and voxelImageIndex components. It is shown that the compression ratio varies from 2 to 2.5. Note that "Palms" model in Table III is not the same one as "Palm" in Table II.

We conclude this subsection with data on rendering speed and animated model size. Rendering speed of DepthImage "Palm512" is about 2 frames per second (fps), while other static models we tested with reference image side 512 are rendered at 5–6 fps. Note that rendering speed depends mostly on the number and resolution of the reference images, but not on the complexity of the scene. This is an important advantage over the polygonal representations, especially in the animated case. Animated OctreeImage "Dragon512" is visualized at 24 FPS in which the compression results are as follows.

a) Compressed size of octree plus voxelImageIndex component: 910 KB (696 KB and 214 KB, respectively);
b) Six reference videostreams in compressed AVI format: 1370 KB;
c) Total data volume: 2280 KB.

2) Rendering Examples: "Angel256" DepthImage model is shown in Fig. 3. Figs. 13–17 show several other DIBR and polygonal models. Fig. 13 compares the appearance of the polygonal and DepthImage "Morton" model. The DepthImage model uses reference images in JPEG format and rendering is performed by the simplest splatting described in Section VI, but the image quality is quite acceptable. Fig. 14 compares two versions of the scanned "Tower" model. Black dots in the upper part of the model are due to noisy input data. Fig. 15 demonstrates more complex "Palm" model, composed of 21 SimpleTextures. It also shows good quality, although the leaves are, in general, wider than those in the 3DS-MAX original—which is a consequence of simplified splatting.

Finally, Fig. 16 presents a 3-D frame from "Dragon512" OctreeImage animation. Fig. 17 demonstrates the ability of PointTexture format to provide models of excellent quality.

3) Rendering on a Mobile Device: In order to show the rendering performance on a device with limited ability, we have developed a renderer on a personal digital assistant (PDA) (iPAQ 3970). The PDA is equipped with an Intel PXA250 400 MHz processor and PocketPC 2002 operating system. Note that it does not support floating point operation and there is no 3-D graphic library, which are the barriers in implementing 3-D graphics on PDAs. Although it is not fully optimized yet, it is observed that we have achieved high frame rates as shown in Fig. 18. It is almost impossible to render same detailed mesh models on a PDA VRML browser. In addition, it is not easy even on a PC to model and render a complex object like the "Flower" model shown in Fig. 18(b). This confirms the efficient modeling and rendering capability of DIBR.

VIII. Conclusion

We have described the family of DIBR, adopted into MPEG-4 AFX as an alternative to conventional polygonal 3-D representations. The main formats of the family are DepthImage, OctreeImage and PointTexture. DepthImage represents an object by a union of its reference images and corresponding depth maps. OctreeImage converts the same data into hierarchical octree-structured voxel model, set of compact reference images, and a tree of voxel-image correspondence indices. PointTexture represents the object as a set of colored points parameterized by projection onto a regular 2-D grid. DepthImage and OctreeImage have animated versions, where reference images are replaced by videostreams. DIBR formats are very convenient for 3-D model construction from 3-D range-scanning and multiple-source video data. MPEG-4 framework allows us to construct a wide variety of representations from the main DIBR formats, providing flexible tools for effective work with 3-D models.

Compression of the DIBR formats is achieved by application of image (video) compression techniques to depth maps and reference images (videostreams). In addition, we have developed a very efficient method for OctreeImage compression, combining context-based adaptive arithmetic coding with the context tables making use of group invariance assumptions.

Our future work includes developing a compression algorithm of the PointTexture model and the OctreeImage animation model. In addition, we plan to develop adaptive representation and rendering of DIBR, in which level-of-detail and view-dependent rendering methods will be investigated.

Acknowledgment

The authors would like to thank Dr. Y.-S. Seo and Dr. S. Y. Jung in Samsung Advanced Institute of Technology for the consistent support to the project. They are also deeply grateful to D. Timasov for his dedicated work of incorporation and testing the DIBR formats in the MPEG-4 AFX reference software.

References

[1] *Information Technology – Coding of Audio-Visual Objects – Part 1: Systems*, ISO/IEC Standard JTC1/SC29/WG11 14 496–1.

[2] *Information Technology – Coding of Audio-Visual Objects – Part 2: Visual*, ISO/IEC Standard JTC1/SC29/WG11 14 496–2.

[3] *Information Technology – Coding of Audio-Visual Objects – Part 3: Audio*, ISO/IEC Standard JTC1/SC29/WG11 14 496–3.

[4] *Information Technology – Coding of Audio-Visual Objects – Part 16: Animation Framework eXtension (AFX)*, ISO/IEC Standard JTC1/SC29/WG11 14 496–16:2003.

[5] G. Schaufler and W. Stürzlinger, "A three-dimensional image cache for virtual reality," in *Proc. Eurographics*, 1996, pp. 227–236.

[6] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder, "Hierarchical image caching for accelerated walk-throughs of complex environments," in *Proc. SIGGRAPH*, 1996, pp. 75–82.

[7] M. Oliveira, G. Bishop, and D. McAllister, "Relief textures mapping," in *Proc. SIGGRAPH*, July 2000, pp. 359–368.

[8] J. Shade, S. Gortler, L. He, and R. Szeliski, "Layered depth images," in *Proc. SIGGRAPH*, July 1998, pp. 231–242.

[9] L. McMillan, "A List-Priority Rendering Algorithm for Redisplaying Projected Surfaces," Univ. of North Carolina, Chapel Hill, 95–005, 1995.

[10] L. McMillan and G. Bishop, "Plenoptic modeling: An image-based rendering system," in *Proc. SIGGRAPH*, August 1995, pp. 39–46.

[11] D. Lischinski and A. Rappoport, "Image-based rendering for nondiffuse synthetic scenes," in *Proc. 9th Eurographics Workshop Rendering*, 1998, pp. 301–314.

[12] M. Oliveira and G. Bishop, "Image-based objects," in *Proc. ACM Symp. Interactive 3D Graphics*, Apr. 1999, pp. 191–198.

[13] C. Chang, G. Bishop, and A. Lastra, "LDI tree: A hierarchical representation for image-based rendering," in *Proc. SIGGRAPH*, Aug. 1999, pp. 291–298.

[14] S. Rusinkiewicz and M. Levoy, "QSplat: A multiresolution point rendering system for large meshes," in *Proc. SIGGRAPH*, July 2000, pp. 343–352.

[15] H. Pfister, M. Zwicker, J. Baar, and M. Gross, "Surfels: Surface elements as rendering primitives," in *Proc. SIGGRAPH*, July 2000, pp. 335–342.

[16] S. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen, "The lumigraph," in *Proc. SIGGRAPH*, Aug. 1996, pp. 43–54.

[17] M. Levoy and P. Hanrahan, "Light field rendering," in *Proc. SIGGRAPH*, Aug. 1996, pp. 31–42.

[18] C. Bregler, "Video based animation techniques for human motion," in *SIGGRAPH'00 Course 39: Image-Based Modeling and Rendering*, July 2000.

[19] P. Debevec, C. Taylor, and J. Malik, "Modeling and rendering architecture from photographs: A hybrid geometry and image-based approach," in *Proc. SIGGRAPH*, 1996, pp. 11–20.

[20] A. Zhirkov, "Binary volumetric octree representation for image based rendering," in *Proc. GRAPHICON*, Sept. 2001, pp. 195–198.

[21] M. Levoy and T. Whitted, "The Use of Points as a Display Primitive," Univ. of North Carolina, Chapel Hill, 85–022, 1985.

[22] S. Kang, "A survey of image-based rendering techniques," *Proc. SPIE*, vol. 3641, pp. 2–16, 1999.

[23] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun.*, vol. COM-32, pp. 396–402, Apr. 1984.

[24] J. Rissanen and G. Langdon, "Universal modeling and coding," *IEEE Trans. Inform. Theory*, vol. 27, pp. 12–23, Jan. 1981.

[25] H. Coxeter and W. Moser, *Generators and Relations for Discrete Groups*, 3rd ed. Berlin, Germany: Springer-Verlag, 1972.

[26] C.-S. Kim and S.-U. Lee, "Compact encoding of 3D voxel surfaces based on pattern code representation," *IEEE Trans. Image Processing*, vol. 11, pp. 932–943, Aug. 2002.

Leonid Levkovich-Maslyuk received the M.S. degree in mathematics from Moscow Lomonosov State University, Moscow, Russia, in 1976.

In 1983, he joined the Keldysh Institute of Applied Mathematics, Moscow, where he currently is a Senior Scientist. His research interests include data compression, computer graphics, and fractal and wavelet analysis.

Alexey Ignatenko graduated from Moscow Lomonosov State University, Moscow, Russia, in 2002, where he is currently working toward the Ph.D. degree.

His research interests are in computer graphics, image-based rendering and lighting, point sample rendering, multiresolution rendering, photorealistic rendering, virtual and augmented reality, and software engineering.

Alexander Zhirkov graduated from Moscow Lomonosov State University (MSU), Moscow, Russia, in 2001, where he is currently working toward the Ph.D. degree.

His research interests are in image and video compression, real-time photorealistic rendering, fractal and multiscale analysis, object and speech recognition, artificial neural networks, synergetics, and human-centered interfaces.

Anton Konushin graduated from Moscow Lomonosov State University (MSU), Moscow, Russia, in 2002, where he is currently working toward the Ph.D. degree.

His research interests are in computer graphics, image-based modeling, virtual and augmented reality, dynamic knowledge-based systems, and software engineering.

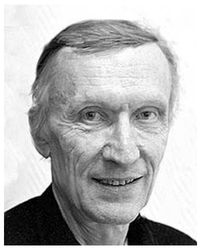
In Kyu Park (S'96–M'01) received the B.S., M.S., and Ph.D. degrees from Seoul National University, Seoul, Korea, in 1995, 1997, and 2001, respectively, all in electrical engineering and computer science.

From September 2001 to March 2004, he was a Member of Technical Staff with the Multimedia Laboratory, Samsung Advanced Institute of Technology, Yongin, Korea, where he was involved in several projects on computer graphics and multimedia applications. Also, he has been actively involved in MPEG-4 and MPEG-7 standardization activities. Since March 2004, he has been with the School of Information and Communication Engineering, Inha University, Incheon, Korea, where he is currently a Full-time Lecturer. His research interests include the joint area of three-dimensional (3-D) computer graphics, computer vision, and multimedia application, especially image-based modeling and rendering, 3-D shape reconstruction, range data processing, and 3-D face modeling.

Dr. Park is a member of the IEEE Computer Society and the Association for Computing Machinery.

Mahnjin Han (M'02) received the B.Sc. degree in computer science and the M.Sc. degree in multimedia from Yonsei University, Seoul, Korea, in 1994 and 1996, respectively.

Currently, he is a Member of Technical Staff and a Project Manager with the Multimedia Laboratory, Samsung Advanced Institute of Technology, Yongin, Korea, focusing on the SNHC research of MPEG-4. He has been involved in the standardization of 3-D Model Coding in the Synthetic Natural Hybrid Coding Group and now is focusing on Animation Framework eXtension (AFX) effort. He is the main contributor to the depth image-based representation in AFX and is also actively involved in research of interpolator compression. His research interests include computer graphics, data compression, image-based rendering, and scientific visualization.



Yuri Bayakovski received the M.S. degree (with honors) in electrical and computer engineering from Moscow Power Engineering Institute, Moscow, Russia, in 1960 and the Ph.D degree in mathematics and computer science from Keldysh Institute of Applied Mathematics (KIAM), Moscow, in 1974.

Since 1977, he has been the head of the Computer Graphics Department, KIAM. Since 1983, he has been a part-time Professor with Moscow Lomonosov State University (MSU), where he established the Graphics and Media Laboratory. He gives lectures on computer graphics and computer vision. His research interests are in scientific visualization, geometry modeling, virtual environments, and computer science education. He has translated and edited more than 20 technical books from English and French into Russian. He served as a Program Committee Chair for several international conferences on computer graphics held in Russia.

Dr. Bayakovski was the recipient of an award given by the U.S.S.R. Government in 1986.