Parameterized Animation Compression

Ziyad S. Hakura[*], Jerome E. Lengyel[**], John M. Snyder[**]
[*]Stanford University, [**]Microsoft Research

Abstract. We generalize image-based rendering by exploiting texture-mapping graphics hardware to decompress ray-traced "animations". Rather than 1D time, our animations are parameterized by two or more arbitrary variables representing view/lighting changes and rigid object motions. To best match the graphics hardware rendering to the input ray-traced imagery, we describe a novel method to infer parameterized texture maps for each object by modeling the hardware as a linear system and then performing least-squares optimization. The parameterized textures are compressed as a multidimensional Laplacian pyramid on fixed size blocks of parameter space. This scheme captures the coherence in parameterized animations and, unlike previous work, decodes directly into texture maps that load into hardware with a few, simple image operations. We introduce adaptive dimension splitting in the Laplacian pyramid and separate diffuse and specular lighting layers to further improve compression. High-quality results are demonstrated at compression ratios up to 800:1 with interactive playback on current consumer graphics cards.

1 Introduction

The central problem of computer graphics is real-time rendering of physically illuminated, dynamic environments. Though the computation needed is beyond current capability, specialized graphics hardware that renders texture-mapped polygons continues to get cheaper and faster. We exploit this hardware to decompress animations computed and compiled offline. The decompressed imagery retains the full gamut of stochastic ray tracing effects, including indirect lighting with reflections, refractions, and shadows.

For synthetic scenes, the time and viewpoint parameters of the plenoptic function [1, 23] can be generalized to include position of lights, viewpoint, or objects, surface reflectance properties, or any other degrees of freedom in the scene. For example, we can construct a 2D space combining viewpoint movement along a 1D trajectory with independent 1D swinging of a light source. Our goal is maximum compression of the resulting arbitrary-dimensional *parameterized animation* that maintains satisfactory quality and decodes in real time. Once the encoding is downloaded over a network, the decoder can take advantage of specialized hardware and high bandwidth to the graphics system to allow a user to explore the parameter space. High compression reduces downloading time over the network and conserves server and client storage.

Our approach infers and compresses parameter dependent texture maps for individual objects rather than combined views of the entire scene, illustrated in Figure 1.

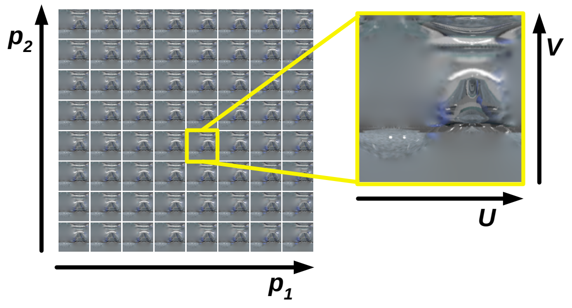


Fig. 1. An 8×8 block of parameterized textures for a glass parfait object is shown. In this example, dimension p_1 represents a 1D viewpoint trajectory while p_2 represents the swinging of a light source. Note the imagery's coherence.

To *infer* a texture map means to find one which when applied to a hardware-rendered geometric object matches the offline-rendered image. Encoding a separate texture map for each object better captures its coherence across the parameter space independently of where in the image it appears. Object silhouettes are correctly rendered from actual geometry and suffer fewer compression artifacts.

Figure 2 illustrates our system. Ray-traced images at each point in the parameter space are input to the compiler together with the scene geometry, lighting models, and viewing parameters. The compiler targets any desired type of graphics hardware by modeling the hardware as a linear system. It then infers texture resolution and texture samples for each object at each point in the parameter space to produce as good a match as possible on that hardware to the "gold-standard" images. We use pyramidal regularization [20] in our texture inference to provide smooth "hole-filling" for occluded regions without a specialized post-processing pass. Per-object texture maps are then compressed using a novel, multi-dimensional compression scheme that automatically allocates storage between different objects and their separated diffuse and specular lighting layers. The interactive runtime consists of a traditional hardware-accelerated rendering engine and a texture decompression engine that caches to speed decoding and staggers block origins to distribute decompression load.

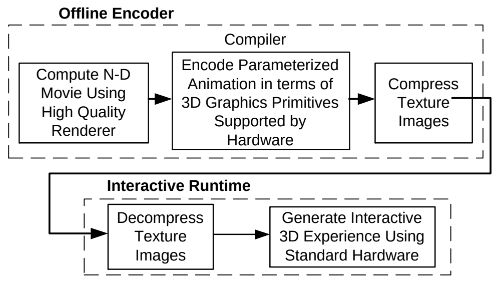

Offline Encoder
Compiler
Compute N-D Movie Using High Quality Renderer
Encode Parameterized Animation in terms of 3D Graphics Primitives Supported by Hardware
Compress Texture Images
Interactive Runtime
Decompress Texture Images
Generate Interactive 3D Experience Using Standard Hardware

Fig. 2. System Overview

2 Previous Work

Image-Based Rendering (IBR). IBR has sought increasingly accurate approximations of the plenoptic function [1, 23], including use of pixel flow [4], and tabulation of a 4D field, called a "light field" or "lumigraph" [18, 10] to interpolate views. Layered depth images (LDI) [30] are another representation of the radiance field better able to handle disocclusions and have found use in the rendering of glossy environments [2]. Extending to a 5D or higher field permits changes to the lighting environment [35, 27]. The challenge of such methods is efficient storage of the high-dimensional image fields.

For spatially coherent scenes, it has been observed that geometry-based surface fields better capture coherence in the light field, achieving a more efficient encoding than view-based images like the LDI or lumigraph [33, 25, 14, 32, 27]. Our work generalizes parameterizations based solely on viewpoint. We also encode an entire texture at each point in parameter space that can be accessed in constant time independent of the size of the whole representation. Other encoding strategies, such as Miller's [25], must visit an irregular scattering of samples over the entire 4D space to reconstruct the texture for a particular view and thus make suboptimal use of graphics hardware.

Another IBR hybrid uses view-dependent textures (VPT) [6, 7, 5] in which geometric objects are texture-mapped using a projective mapping from view-based images. VPT methods depend on viewpoint movement for proper antialiasing - novel views are generated by reconstructing using nearby views that see each surface sufficiently "head-on". Such reconstruction is incorrect for highly specular surfaces. We instead infer texture maps that produce antialiased reconstructions independently at each parameter location, even for spaces without viewpoint change. We also use "intrinsic" texture parameterizations (i.e., viewpoint-independent (u, v) coordinates per vertex given as input on each mesh) rather than view-based ones. We can then capture the view-independent lighting in a single texture map rather than a collection of views to obtain better compression.

Interactive Photorealism. Another approach to interactive photorealism seeks to improve hardware shading models rather than fully tabulating radiance. Diefenbach [8] used shadow volumes and recursive hardware rendering to compute approximations to global rendering. Even using many parallel graphics pipelines (8 for [34]) these approaches can only handle simple scenes, and, because of limitations on the number of passes, do not capture all the effects of a full offline photorealistic rendering, including multiple bounce reflections and refractions and accurate shadows.

Texture Recovery/Model Matching. The recovery of texture maps from images is closely related to surface reflectance estimation in computer vision [29, 22, 39]. We greatly simplify the problem by using known geometry and separating diffuse and specular lighting layers during the offline rendering. We focus instead on the problem of inferring textures for particular graphics hardware that "undo" its undesirable properties, like poor-quality texture filtering. A related idea is to compute the best hardware lighting to match a gold standard [38]. Separating diffuse from specular shading to better exploit temporal and spatial coherence is a recurring theme in computer graphics [36, 26, 16, 19].

Compression. Various strategies for compressing the dual-plane lumigraph parameterization have been proposed. Levoy et al. [18] used vector quantization and entropy coding to get compression ratios up to 118:1 while Lalonde et al. [15] used a wavelet basis with compression ratios of 20:1. Miller et al. [25] compressed the 4D surface light field using a block-based DCT encoder with compression ratios of 20:1. Nishino et al. [27] used an eigenbasis to encode surface textures achieving compression ratios of 20:1 with eigenbases having 8-18 vectors.

Another relevant area of work is animation compression. Standard video compression uses simple block-based transforms and image-based motion prediction. Wallach et al. [37] used rendering hardware to accelerate standard MPEG encoding. Guenter et al. [11] observed that compression is greatly improved by exploiting information available in synthetic animations. Levoy [17] showed how simple graphics hardware could be used to match a synthetic image stream produced by a simultaneously-executing, high-quality server renderer by exploiting polygon rendering and transmitting a residual signal. We extend this work to the matching of multidimensional animations containing non-diffuse, offline-rendered imagery using texture-mapping graphics hardware.

3 Parameterized Texture Inference

To infer the texture maps that best match the input gold-standard rendered frames, we first model the graphics hardware as a large sparse linear system (Section 3.3), and then perform a least-squares optimization on the resulting system (Section 3.4). To achieve a good encoding, we first segment the input images (Section 3.1), and choose an appropriate texture domain and resolution (Section 3.2).

3.1 Segmenting Ray-Traced Images

Each geometric object has a parameterized texture that must be inferred from the ray-traced images. These images are first segmented into per-object pieces to prevent bleeding of information from different objects across silhouettes. To perform per-object segmentation, the ray tracer generates a per-object mask as well as a combined image, all at supersampled resolution. For each object, we filter the portion of the combined image indicated by the mask and divide by the fractional coverage computed by applying the same filter to the object's mask.

A second form of segmentation separates the view-dependent specular information from the view-independent diffuse information for the common case that the parameter space includes at least one view dimension. Figure 3 illustrates segmentation for an example ray-traced image. We use a modified version of Eon, a Monte Carlo distribution ray-tracer [31].

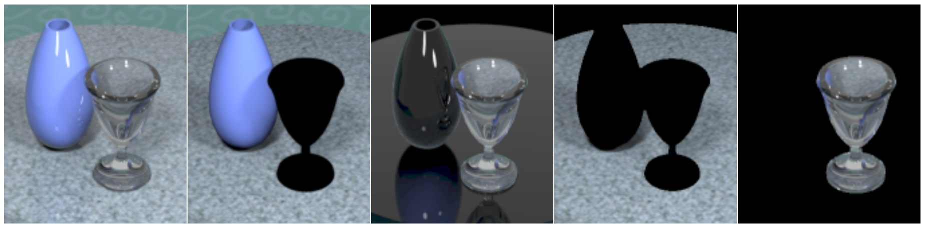


Fig. 3. Segmentation of Ray-Traced Images. (a) Complete Image, (b,c) Segmentation into diffuse and specular layers respectively, (d,e) Examples of further segmentation into per object layers.

3.2 Optimizing Texture Coordinates and Resolutions

Since parts of an object may be occluded or off-screen, only part of its texture domain is accessed. The original texture coordinates of the geometry are used as a starting point and then optimized so as to: 1) to ensure adequate sampling of the visible texture image with as few samples as possible, 2) to allow efficient computation of texture coordinates at run-time, and 3) to minimize encoding of the optimized texture coordinates. To satisfy the last two goals, we choose and encode a parameter-dependent affine transformation on the original texture coordinates rather than re-specify them at each vertex. One affine transformation is chosen per object per block of parameter space (see Section 4).

The first step of the algorithm finds the linear transformation, $R(u, v)$, minimizing the following objective function, inspired by [21]

$$R(u,v)=\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix},\ f(x)=\sum_{\text{edges } i} W_i\left(\frac{s_i-\|R(u_{i_0},v_{i_0})-R(u_{i_1},v_{i_1})\|}{\min\left(s_i,\|R(u_{i_0},v_{i_0})-R(u_{i_1},v_{i_1})\|\right)}\right)^2 \quad (1)$$

where s_i represents the length on the screen of a particular triangle edge, i_0 and i_1 represent the edge vertices, and W_i is a weighting term which sums screen areas of triangles on each side of the edge. At each point in the parameter block, the sum in f is taken over *visible* triangle edges determined by rasterizing triangle identifiers into a zbuffer after clipping to the view frustum.

This minimization choses a mapping that is as close to an isometry as possible by minimizing length difference between triangle edges in texture space and projected to the image. We divide by the minimum edge length so as to equally penalize edges that are an equal factor longer and shorter. $\nabla f(x)$ is calculated analytically for use in conjugate gradient minimization.

In the second step, we ensure that the object's texture map contains enough samples by scaling the R found previously. We check the greatest local stretch (singular value) across all screen pixels in which the object is visible, using the Jacobian of the mapping from texture to screen space. If the maximum singular value exceeds a threshold, we scale R by the maximum singular value in the corresponding direction of maximal stretch, and iterate until the maximum singular value is reduced below the threshold. This essentially adds more samples to counteract the worst-case stretching of the projected texture.

Finally, the minimum-area bounding rectangle on the transformed texture coordinates determines the resulting texture resolution and affine texture transformation.

3.3 Modeling Hardware Rendering as a Linear System

A simple texture inference algorithm maps each texel to the image and then filters the neighboring region to reconstruct the texel's value [22]. One problem with this approach is reconstruction of texels near arbitrarily-shaped object boundaries and occluded regions (Figure 3-d,e). Such occluded regions produce undefined texture samples which complicates building of MIPMAPs. Finally, the simple algorithm does not take into account how texture filtering is performed on the target graphics hardware.

A more principled approach is to model the hardware texture mapping operation in the form of a linear system:

$$\overbrace{\begin{bmatrix} s_{0,0} \text{ filter coefficients} \\ s_{0,1} \text{ filter coefficients} \\ \\ \vdots \\ \\ s_{m-1,n-1} \text{ filter coefficients} \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} \left.\begin{matrix} x^{0}_{0,0} \\ \vdots \\ x^{0}_{u-1,v-1} \end{matrix}\right\} \begin{matrix}\text{level} \\ 0\end{matrix} \\ \left.\begin{matrix} x^{1}_{0,0} \\ \vdots \\ x^{1}_{\frac{u}{2}-1,\frac{v}{2}-1} \end{matrix}\right\} \begin{matrix}\text{level} \\ 1\end{matrix} \\ \vdots \\ \left.\begin{matrix} x^{l-1}_{0,0} \\ \vdots \\ x^{l-1}_{\frac{u}{2^{l-1}}-1,\frac{v}{2^{l-1}}-1} \end{matrix}\right\} \begin{matrix}\text{level} \\ l-1\end{matrix} \end{bmatrix}}^{x} = \overbrace{\begin{bmatrix} s_{0,0} \\ s_{0,1} \\ \\ \vdots \\ \\ s_{m-1,n-1} \end{bmatrix}}^{b} \tag{2}$$

where vector b contains the ray-traced image to be matched, matrix A contains the filter coefficients applied to individual texels by the hardware, and vector x represents the texels from all $l-1$ levels of the MIPMAP to be inferred. Superscripts in x entries represent MIPMAP level and subscripts represent spatial location. This model ignores hardware nonlinearities in the form of rounding and quantization. All three color components of the texture share the same matrix A.

Each row in matrix A corresponds to a particular screen pixel, while each column corresponds to a particular texel in the texture's MIPMAP pyramid. The entries in a given row of A represent the hardware filter coefficients that blend texels to produce the color at a given screen pixel. Hardware filtering requires only a small number of texel accesses per screen pixel, so the matrix A is very sparse. We use hardware z-buffering to determine object visibility on the screen, and need only consider rows (screen pixels) where the object is visible. Filter coefficients should sum to one in any row.

Obtaining Matrix A. A simple but impractical algorithm for obtaining A examines the screen output from a series of renderings, each setting only a single texel of interest to a nonzero value, as follows:

```
Initialize z-buffer with visibility information by rendering entire scene
For each texel in MIPMAP pyramid,
    Clear texture, and set individual texel to maximum intensity
    Clear framebuffer, and render all triangles that compose object
    For each non-zero pixel in framebuffer,
        Divide screen pixel value by maximum framebuffer intensity
        Place fractional value in $A$[screen pixel row][texel column]
```

Accuracy of inferred filter coefficients is limited by the color component resolution of the framebuffer, typically 8 bits.

To accelerate the simple algorithm, we observe that multiple columns in the matrix A can be filled in parallel as long as texel projections do not overlap on the screen and we can determine which pixels derive from which texels. An algorithm that subdivides texture space and checks that alternate texture block projections do not overlap can be devised based on this observation. A better algorithm recognizes that since just a single color component is required to infer the matrix coefficients, the other color components (typically 16 or 24 bits) can be used to store a unique texel identifier that indicates the destination column for storing the filtering coefficient. With this algorithm, described in-depth in [12], the matrix A can be inferred in 108 renderings, independent of texture resolution.

3.4 Least-Squares Solution

Removing irrelevant image pixels from Equation (2), A becomes an $n_s \times n_t$ matrix, where n_s is the number of screen pixels in which the object is visible, and n_t is the number of texels in the object's texture MIPMAP pyramid. Once we have obtained the matrix A, we solve for the texture represented by the vector x by minimizing a function $f(x)$ defined via

$$f(x) = \|Ax - b\|^2, \quad \nabla f(x) = 2A^T(Ax - b) \tag{3}$$

subject to the constraint $0 \leq x_{i,j}^k \leq 1$. Given the gradient, $\nabla f(x)$, the conjugate gradient method can be used to minimize $f(x)$. The main computation of the solution's inner loop multiplies A with a vector x representing the current solution estimate and, for the gradient, A^T with $Ax - b$. Since A is a sparse matrix with each row containing a small number of nonzero elements (exactly 8 with trilinear filtering), the cost of multiplying A or A^T with a vector is proportional to n_s. Another way to express $f(x)$ and $\nabla f(x)$ is:

$$f(x) = xA^TAx - 2x \cdot A^Tb + b \cdot b, \quad \nabla f(x) = 2A^TAx - 2A^Tb \tag{4}$$

In this formulation, the inner loop's main computation multiplies A^TA, an $n_t \times n_t$ matrix, with a vector. Since A^TA is also sparse, though less so than A, the cost of multiplying A^TA with a vector is proportional to n_t. We use the following heuristic to decide which set of equations to use:

if ($2n_s \geq Kn_t$**)** Use A^TA method: Equation (4) **else** Use A method: Equation (3)

where K is a measure of relative sparsity of A^TA compared to A. We use $K = 4$. The factor 2 in the test arises because Equation (3) requires two matrix-vector multiplies while Equation (4) only requires one.

The solver can be sped up by using an initial guess vector x that interpolates the solution obtained at lower resolution. The problem size can then be gradually scaled up until it reaches the desired texture resolution. Alternatively, once a solution is found at one point in the parameter space, it can be used as an initial guess for neighboring points, which are immediately solved at the desired texture resolution.

Segmenting the ray-traced images into view-dependent and view-independent layers allows us to collapse the view-independent textures across multiple viewpoints. To compute a single diffuse texture, we solve the following problem:

$$\overbrace{\begin{bmatrix} A_{v_0} \\ A_{v_1} \\ \vdots \\ A_{v_{n-1}} \end{bmatrix}}^{A'} \begin{bmatrix} x \end{bmatrix} = \overbrace{\begin{bmatrix} b_{v_0} \\ b_{v_1} \\ \vdots \\ b_{v_{n-1}} \end{bmatrix}}^{b'} \qquad (5)$$

where matrix A' concatenates the A matrices for the individual viewpoints v_0 through v_{n-1}, vector b' concatenates the ray-traced images at the corresponding viewpoints, and vector x represents the single diffuse texture to be solved.

Regularization. One of the consequences of setting up the texture inference problem in the form of Equation (2) is that only texels actually used by the graphics hardware are solved, leaving the remaining texels undefined. To support movement away from the original viewpoint samples and to make the texture easier to compress, all texels should be defined. This can be achieved with *pyramidal regularization* of the form:

$$f_{reg}(x) = f(x) + \varepsilon \left(\frac{n_s}{n_t} \right) \Gamma(x) \qquad (6)$$

where $\Gamma(x)$ takes the difference between texels at each level of the MIPMAP with an interpolated version of the next coarser level as illustrated in Figure 4. The objective function f sums errors in screen space, while the regularization term sums errors in texture space. This requires a scale of the regularization term by n_s/n_t. We compute ∇f_{reg} analytically. This regularizing term essentially imposes a filter constraint between levels of the MIPMAP, with user-defined strength $\varepsilon \geq 0$. We currently define Γ using simple bilinear interpolation.

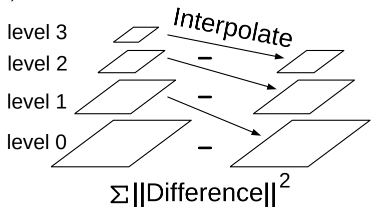


Fig. 4. Pyramidal regularization is computed by taking the sum of squared differences between texels at each level of the MIPMAP with the interpolated image of the next higher level.

3.5 Texture Inference Results

Figure 6 (see appendix) shows results of our least squares texture inference on a glass parfait object. The far left of the top row (a) is the original image to be matched. The next three columns are hardware-rendered from inferred textures using three filtering modes: bilinear, trilinear, and anisotropic.[1] The corresponding texture maps are shown in the first three columns of the next row (b). These three examples did not use pyramidal regularization.[2] Most of the error in these examples is incurred on the parfait's silhouettes due to mismatch between hardware and ray-traced rendering.

[1]Results were achieved with the NVidia Geforce chip supporting anisotropy factors up to 2.

[2]Without pyramidal regularization, we find that another regularization term is needed to ensure that the texture solution lies in the interval $[0, 1]$. Refer to [12] for details.

Bilinear filtering provides the sharpest, most accurate result because it uses only the finest level MIPMAP and thus has the highest frequency domain with which to match the original. Trilinear MIPMAP filtering produces a somewhat worse result, and anisotropic filtering is in between. Observe (Fig.6b) that more texture area is filled from the finest pyramid level for anisotropic filtering compared to trilinear, especially near the parfait stem, while bilinear filtering altogether ignores the higher MIPMAP levels. Bilinear filtering produces this highly accurate result *only at the exact parameter values (e.g., viewpoint locations) and image resolutions where the texture was inferred*. The other schemes are superior if viewpoint or image resolution are changed from those samples.

The next two columns show results of pyramidal regularization with anisotropic filtering. Inference with $\varepsilon = 0.1$ is almost identical to inference with no pyramidal regularization (labeled "anisotropic"), but $\varepsilon = 0.5$ causes noticeable blurring. Regularization makes MIPMAP levels tend toward filtered versions of each other; we exploit this by compressing only the finest level and re-creating higher levels by on-the-fly decimation.

Finally, the far right column in (a) shows the "forward mapping" method in which texture samples are mapped to the object's image layer and interpolated using a high-quality filter (we used a separable Lanczos-windowed sinc function). To handle occlusions, we first filled undefined samples with a simple boundary-reflection algorithm. Forward mapping produces a blurry and inaccurate result because it does not account for how graphics hardware filters the textures. In addition, reflection hole-filling produces artificial, high-frequency information in occluded regions that is expensive to encode.

4 Parameterized Texture Compression

The multidimensional field of textures for each object is compressed by subdividing into parameter space blocks as shown in Figure 1. Larger block sizes better exploit coherence but are more costly to decode during playback; we used 8×8 blocks in our 2D examples.

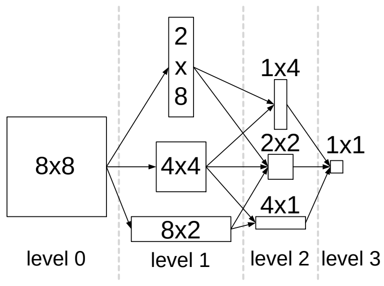


Fig. 5. Adaptive Laplacian Pyramid

Adaptive Laplacian Pyramid. We encode parameterized texture blocks using a Laplacian pyramid [3] where the "samples" at each pyramid level are entire 2D images. We use standard 2D compression (e.g., JPEG and SPIHT [28] encodings) to exploit spatial coherence over (u, v) space. Each level of the Laplacian pyramid thus consists of a series of encoded 2D images. Parameter and texture dimensions are treated asymmetrically because parameters are accessed along an unpredictable 1D subspace selected by the user at run-time. We avoid processing large fractions of the representation to decode a given parameter sample by using the Laplacian pyramid with small block size, requiring just $\log_2(n)$ simple image additions where n is the number of samples in each dimension of the block. Furthermore, graphics hardware can perform the necessary image additions using multiple texture stages, thus enabling on-the-fly decompression.

Image coders often assume that both image dimensions are equally coherent. This is untrue of parameterized animations where, for example, the information content in a viewpoint change can greatly differ from that of a light source motion. To take advantage of differences in coherence across different dimensions, we use an *adaptive* Laplacian pyramid that subdivides more in dimensions with less coherence, illustrated in Figure 5. Coarser levels still have 4 times fewer samples.

Automatic Storage Allocation. To encode the Laplacian pyramid, storage must be assigned to its various levels. We apply standard bit allocation techniques from signal compression [9]. Curves of mean squared error (MSE) versus storage, called *rate/distortion curves*, are plotted for each pyramid level and points of equal slope on each curve selected subject to a total storage constraint. We effectively minimize the sum of MSEs across all levels of the pyramid, because a texture image at a given point in parameter space is reconstructed as a sum of images from each level, so an error in any level contributes equally to the resulting error.

There is also a need to perform storage allocation across objects; that is, to decide how much to spend in the encoding of object i's texture vs. object j's. We use the same method as for allocating between pyramid levels, except that the error measure is $A_i E_i$, where A_i is the screen area and E_i the MSE of object i. This minimizes the sum of squared errors on the screen no matter how the screen area is decomposed into objects. When objects have both specular and diffuse reflectance, our error measure sums across these lighting layers, each with an independent rate distortion curve.

5 Runtime System

The runtime system decompresses and caches texture images, applies affine transformations to vertex texture coordinates, and generates rendering calls to the graphics system. Movement off (or between) the original viewpoint samples is allowed by rendering from that viewpoint using the closest texture sample. Higher-order interpolation would improve smoothness at the expense of more texture map accesses.

The texture caching system decides which textures to keep in memory in decompressed form. Because the user's path through parameter space is unpredictable, we use an adaptive caching strategy that reclaims memory when the number of frames since last use exceeds a given lifetime. Images near the top of the pyramid are more likely to be reused and are thus assigned longer lifetimes. See [12] for more details.

If blocks of all objects are aligned, then many simultaneous cache misses occur whenever the user crosses a block boundary, creating a computational spike as multiple levels in the new blocks' Laplacian pyramids are decoded. We mitigate this problem by *staggering* the blocks – using different block origins for different objects.

6 Results

6.1 Demo1: Light $\times$ View

Compression Results. The first example scene (Figure 8 in appendix, top) consists of 6 static objects (4384 triangles): a reflective vase, glass parfait, reflective table top, table stand, walls, and floor. The 2D parameter space has 64 viewpoint samples circling around the table at 1.8°/sample and 8 different positions of a swinging, spherical light source. The image field was encoded using eight 8$\times$8 parameter space blocks, each requiring storage 640$\times$480$\times$3$\times$8$\times$8= 56.25MB/block.

Our least-squares texture inference method created parameterized textures for each object. The resulting texture fields were compressed using a variety of methods, including adaptive 2D Laplacian pyramids of both DCT- and SPIHT-encoded levels. To test the benefits of the Laplacian pyramid, we also encoded each block using MPEG on a 1D zig-zag path through the parameter space varying most rapidly along the dimension of most coherence. A state-of-the-art MPEG4 encoder [24] was used. Finally, we compared against direct compression of the original images (rather than renderings using compressed textures), again using MPEG4.

Figure 8 shows the results at two targeted compression rates: 384:1 (middle row) and 768:1 (bottom row). All texture-based images were generated on graphics hardware using 2×2 antialiasing; their MSEs were computed from the framebuffer contents, averaged over an entire block. Both Laplacian pyramid texture encodings (right two columns) achieve reasonable quality at 768:1, and quite good quality at 384:1. The view-based MPEG encoding, “MPEG-view”, is inferior with obvious block artifacts on object silhouettes, even though MPEG encoding constraints did not allow as much compression as the other examples.

For MPEG encoding of textures we tried two schemes: one using a single I-frame per block, and another using 10 I-frames. The decoding complexity for 10I/block is roughly comparable to our DCT Laplacian pyramid decoding. Single I-frame/block maximizes compression. The 10I/block MPEG-texture results have obvious block artifacts at both quality levels especially on the vase and background wallpaper. The 1I/block MPEG-texture results are better[3], but still inferior to the pyramid schemes at the 768:1 target as MPEG only exploit coherence in one dimension. Unlike the MPEG-view case, the MPEG-texture schemes use our novel features: hardware-targeted texture inference, separation of lighting layers, and optimal storage allocation across objects.

System Performance. Average compilation and preprocessing time per point in parameter space is shown in Table 1. It can be seen that total compilation time is a small fraction of the time to produce the ray-traced images.

To determine playback performance, we measured average and worst-case frame rates (fps) for a diagonal trajectory that visits a separate parameter sample at every frame, shown in Table 2.[4] The performance bottleneck is currently software decoding speed. Reducing texture resolution by an average of 91% using a manually specified reduction factor per object provides acceptable quality at about 31fps with DCT.

Table 1. Compilation time

texture coord. opt.	1 sec
solving for textures	4.83 min
compression	.58 min
total compilation	5.43 min
ray tracing	5 hours

Table 2. Runtime performance

Encoding	Texture	Worst fps	Average fps
Laplacian DCT	undecimated	2.46	4.76
	decimated	18.4	30.7
Laplacian SPIHT	undecimated	0.27	0.67
	decimated	2.50	5.48

6.2 Demo2: View $\times$ Object Rotation

In the second example, we added a rotating, reflective “gewgaw” on the table. The parameter space consists of a 1D circular viewpoint path, containing 24 samples at 1.5°/sample, and the rotation angle of the gewgaw, containing 48 samples at 7.5°/sample. Results are shown in Figure 7 (appendix) for encodings using MPEG-view and Laplacian SPIHT.

In this example, the parameter space is much more coherent in the rotation dimension than in the view dimension, because gewgaw rotation only changes the relatively small reflected or refracted image of the gewgaw in the other objects. MPEG can exploit this coherence very effectively using motion compensation along the rotation dimension, and so the difference between our approach and MPEG is less than in the previous example. Though our method is designed to exploit multidimensional coherence and lacks motion compensation, our adaptive pyramid produces a slightly better MSE and a perceptually better image.

Real-time performance for this demo is approximately the same as for demo1.

[3]MSE=25.9 at 768:1 target and MSE=10.1 at 384:1 target compression. See [12].

[4]Measured with Nvidia Geforce 256 chip, 32MB video/16MB AGP memory on Pentium III 733Mhz PC.

7 Conclusions and Future Work

Synthetic imagery can be very generally parameterized with combinations of view, light, or object positions, among other parameters, to create a multidimensional animation. While real-time graphics hardware fails to capture full ray-traced shading effects, it does provide a useful operation for decoding such animations compiled beforehand: texture-mapped polygon rendering. We encode a parameterized animation using parameterized texture maps, exploiting the great coherence in these animations better than view-based representations. This paper describes how to infer parameterized texture maps from segmented imagery to obtain a close match to the original and how to compress these maps efficiently, both in terms of storage and decoding time. Results show that compression factors up to 800:1 can be achieved with good quality and real-time decoding.

Our simple sum of diffuse and specular texture maps is but a first step toward more predictive graphics models supported by hardware to aid compression. Examples include parameterized environment maps, hardware shadowing algorithms, and per-vertex shading models. The discipline of measuring compression ratios vs. error for encoding photorealistic imagery is a useful benchmark for proposed hardware enhancements.

Other extensions include use of perceptual metrics for guiding compression and storage allocation, handling nonrigidly deforming geometry and photorealistic camera models, and automatic generation of texture parameterizations. Finally, we are interested in measuring storage requirements with growing dimension of the parameter space and hypothesize that such growth is quite small in many useful cases. There appear to be two main impediments to increasing the generality of the space that can be explored: slowness of offline rendering and decompression. The first obstacle may be addressed by better exploiting coherence across the parameter space in the offline renderer, using ideas similar to [13]. The second can be overcome by absorbing some of the decoding functionality into the graphics hardware. We expect the ability to load compressed textures directly to hardware in the near future. A further enhancement would be to load compressed parameter-dependent texture block pyramids.

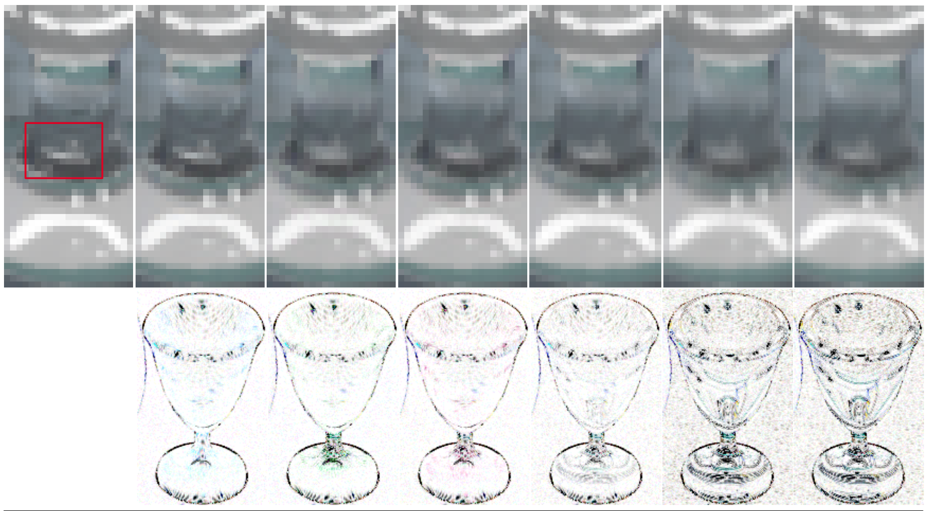
Acknowledgments

Extremely useful advice and discussions were provided by Henrique Malvar, Philip Chou, and Brian Guenter on compression, John Platt on optimization and regularization, and Steve Marschner on solving for texture maps and ray tracing. We thank Anoop Gupta and Turner Whitted for their guidance and support. Peter Shirley's Eon ray tracer was a valuable research tool.

References

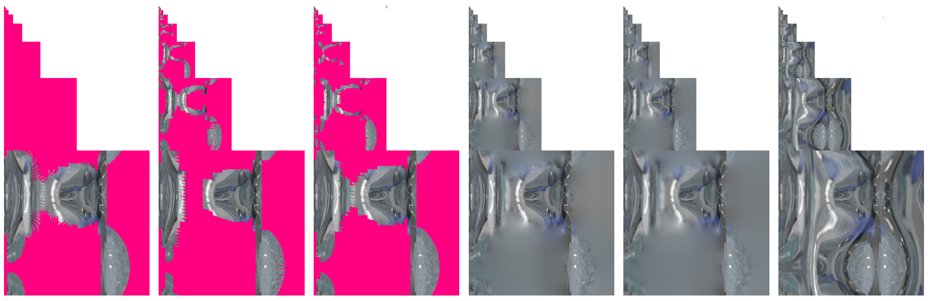
1. Adelson, E., and J. Bergen, "The Plenoptic Function and the Elements of Early Vision," In Computational Models of Visual Processing, MIT Press, Cambridge, MA, 1991, pp. 3-20.
2. Bastos, R., K. Hoff, W. Wynn, and A. Lastra. "Increased Photorealism for Interactive Architectural Walkthroughs," 1999 ACM Symposium on Interactive 3D Graphics, April 1999, pp. 183-190.
3. Burt, P., and E. Adelson, "The Laplacian Pyramid as a Compact Image Code," IEEE Transactions on Communications, Vol. Com-31, No. 4, April 1983, pp. 532-540.
4. Chen, S.E., and L. Williams, "View Interpolation for Image Synthesis," SIGGRAPH 93, August 1993, pp. 279-288.
5. Cohen-Or, D., Y. Mann, and S. Fleishman, "Deep Compression for Streaming Texture Intensive Animations," SIGGRAPH 99, August 1999, pp. 261-265.
6. Debevec, P., C. Taylor, and J. Malik, "Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach," SIGGRAPH 96, August 1996, pp. 11-20.
7. Debevec, P., Y. Yu, and G. Borshukov, "Efficient View-Dependent Image-Based Rendering with Projective Texture Maps," In 9th Eurographics Rendering Workshop, June 1998, pp. 105-116.

8. Diefenbach, P. J., Pipeline Rendering: Interaction and Realism Through Hardware-Based Multi-Pass Rendering, Ph.D. Thesis, University of Pennsylvania, 1996.
9. Gersho, A., and R. Gray, Vector Quantization and Signal Compression, Kluwer Academic, Boston, 1992, pp. 606-610.
10. Gortler, S., R. Grzeszczuk, R. Szeliski, and M. Cohen, "The Lumigraph," SIGGRAPH 96, pp. 43-54.
11. Guenter, B., H. Yun, and R. Mersereau, "Motion Compensated Compression of Computer Animation Frames," SIGGRAPH 93, August 1993, pp. 297-304.
12. Hakura, Z., J. Lengyel, and J. Snyder, "Parameterized Animation Compression", MSR-TR-2000-50, June 2000.
13. Halle, M., "Multiple Viewpoint Rendering," SIGGRAPH 98, August 1998, 243-254.
14. Heidrich, W., H. Lensch, and H. P. Seidel. "Light Field Techniques for Reflections and Refractions," In 10th Eurographics Rendering Workshop, June 1999, pp. 195-204, p. 375.
15. Lalonde, P. and A. Fournier, "Interactive Rendering of Wavelet Projected Light Fields," Graphics Interface '99, June 1999, pp. 107-114.
16. Lengyel, J., and J. Snyder, "Rendering with Coherent Layers," SIGGRAPH 97, August 1997, pp. 233-242.
17. Levoy, M., "Polygon-Assisted JPEG and MPEG compression of Synthetic Images," SIGGRAPH 95, August 1995, pp. 21-28.
18. Levoy, M., and P. Hanrahan, "Light Field Rendering," SIGGRAPH 96, August 1996, pp. 31-41.
19. Lischinski, D., and A. Rappoport, "Image-Based Rendering for Non-Diffuse Synthetic Scenes," In 9th Eurographics Workshop on Rendering, June 1998.
20. Luettgen, M., W. Karl, and A Willsky, "Efficient Multiscale Regularization with Applications to the Computation of Optical Flow," IEEE Transactions on Image Processing, 3(1), 1994, pp. 41-64.
21. Maillot, J., H. Yahia, A. Verroust, "Interactive Texture Mapping," SIGGRAPH 93, pp. 27-34.
22. Marschner, S. R., Inverse Rendering for Computer Graphics, Ph.D. Thesis, Cornell University, August 1998.
23. McMillan, L., and G. Bishop, "Plenoptic Modeling," SIGGRAPH 95, pp. 39-46.
24. Microsoft MPEG-4 Visual Codec FDIS 1.02, ISO/IEC 14496-5 FDIS1, August 1999.
25. Miller, G., S. Rubin, and D. Poncelen, "Lazy Decompression of Surface Light Fields for Pre-computed Global Illumination," In 9th Eurographics Rendering Workshop, June 1998, pp. 281-292.
26. Nimeroff, J., J. Dorsey, and H. Rushmeier, "Implementation and Analysis of an Image-Based GLobal Illumination Framework for Animated Environments," IEEE Transactions on Visualization and Computer Graphics, 2(4), Dec. 1996, pp. 283-298.
27. Nishino, K., Y. Sato, and K. Ikeuchi, "Eigen-Texture Method: Appearance Compression based on 3D Model," Proceedings of 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. Fort Collins, CO, June, 1999, pp. 618-24 Vol. 1.
28. Said, A., and W. Pearlman, "A New, Fast, and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees," IEEE Transactions on Circuits and Systems for Video Technology, Vol. 6, June 1996, pp. 243-250.
29. Sato, Y., M. Wheeler, and K. Ikeuchi, "Object Shape and Reflectance Modeling from Observation," SIGGRAPH 97, August 1997, pp. 379-387.
30. Shade, J., S. Gortler, L. He, and R. Szeliski, "Layered Depth Images," SIGGRAPH 98, August 1998, pp. 75-82.
31. Shirley, Wang and Zimmerman, "Monte Carlo Methods for Direct Lighting Calculations," ACM Transactions on Graphics, January 1996, pp. 1-36.
32. Stamminger, M., A. Scheel, et al., "Efficient Glossy Global Illumination with Interactive Viewing," Graphics Interface '99, June 1999, pp. 50-57.
33. Stürzlinger, W. and R. Bastos, "Interactive Rendering of Globally Illuminated Glossy Scenes," Eurographics Rendering Workshop 1997, June 1997, pp. 93-102.
34. Udeshi, T. and C. Hansen, "Towards interactive, photorealistic rendering of indoor scenes: A hybrid approach," Eurographics Rendering Workshop 1999, June 1999, pp. 71-84 and pp. 367-368.
35. Wong, T. T., P. A. Heng, S. H. Or, and W. Y. Ng, "Image-based Rendering with Controllable Illumination," Eurographics Rendering Workshop 1997, June 1997, pp. 13-22.
36. Wallace, J., M. Cohen, and D. Greenberg, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray-Tracing and Radiosity Methods," SIGGRAPH 87, July 1987, pp. 311-320.
37. Wallach, D., S. Kunapalli, and M. Cohen, "Accelerated MPEG Compression of Dynamic Polygonal Scenes," SIGGRAPH 94, July 1997, pp. 193-196.
38. Walter, B., G. Alppay, E. Lafortune, S. Fernandez, and D. Greenberg, "Fitting Virtual Lights for Non-Diffuse Walkthroughs," SIGGRAPH 97, August 1997, pp. 45-48.
39. Yu, Y., P. Debevec, J. Malik, and T. Hawkins, "Inverse Global Illumination: Recovering Reflectance Models of Real Scenes From Photographs," SIGGRAPH 99, August 1999, pp. 215-224.



original	bilinear	trilinear	anisotropic	pyr. $\varepsilon = 0.1$	pyr. $\varepsilon = 0.5$	forward map
	MSE=4.34	MSE=5.31	MSE=4.81	MSE=4.96	MSE=9.58	MSE=10.72

(a) Images (Close-up of Parfait Stem and Error Signal)



(b) Inferred Texture Maps

Fig. 6. Texture Inference Results: (a) shows close-ups of the projected texture, compared to the original rendering on the far left. The highlight within the red box is a good place to observe differences. The next row shows the inverted error signal, scaled by a factor of 20, over the parfait. The bottom row contains the mean-squared error (MSE) from the original image. (b) shows corresponding texture maps. Pink regions represent undefined regions of the texture.

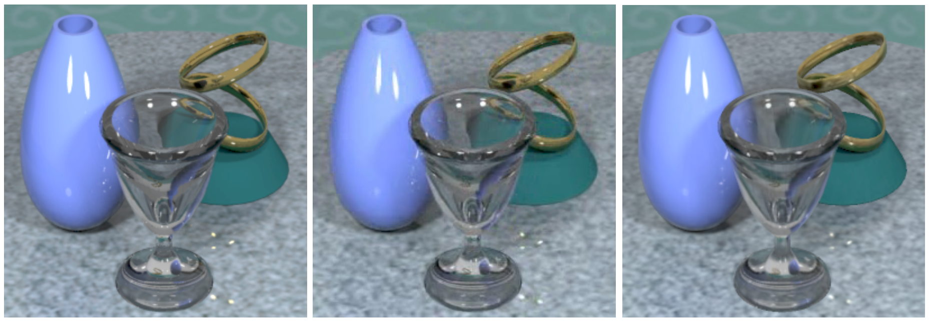
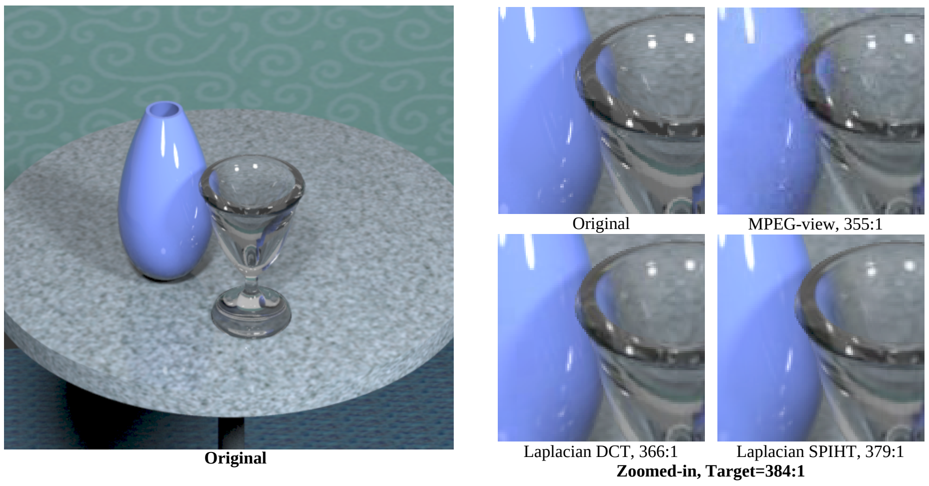


Fig. 7. Demo2 Compression Results



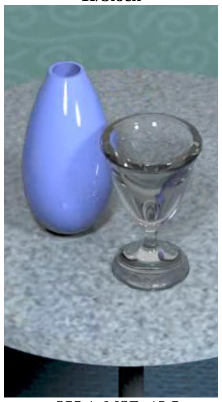
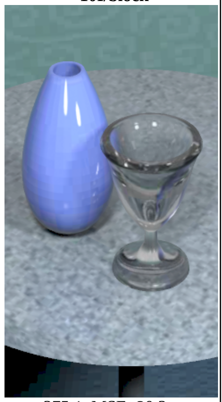
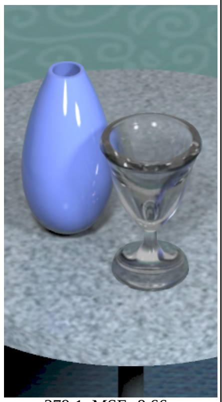
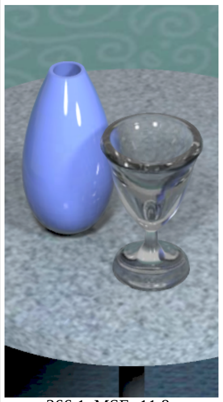
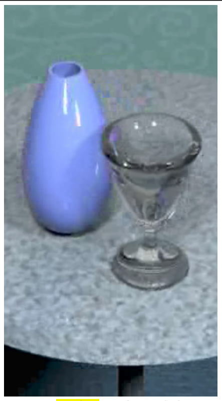
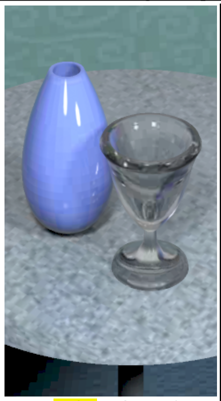
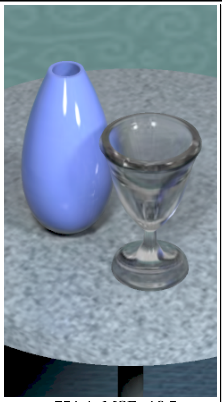
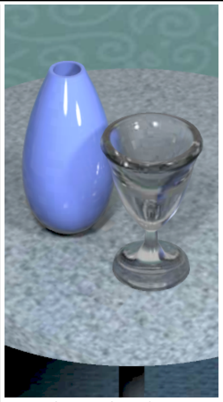
View-based	Texture-based		
MPEG-view 1I/block	**MPEG-texture 10I/block**	**Laplacian SPIHT**	**Laplacian DCT**
			
355:1, MSE=13.5	375:1, MSE=20.3	379:1, MSE=8.66	366:1, MSE=11.8
	Target compression = 384:1		
			
570:1, MSE=29.8	418:1, MSE=25.0	751:1, MSE=16.5	732:1, MSE=18.1
	Target compression = 768:1		

Fig. 8. Demo1 Compression Results