Parameterized Animation Compression

Ziyad Hakura, Jed Lengyel, John Snyder

Abstract

Synthetic images can be parameterized by more than time or viewpoint. We generalize image-based rendering by exploiting texture-mapping graphics hardware to decompress ray-traced animations. The animations are parameterized by two or more arbitrary variables allowing view/lighting changes or relative motion of objects. Starting with a field of ray-traced images and a description of the shading models, camera parameters, and scene geometry, we encode the parameterized animation as a set of per-object parameterized textures. We present a novel method to infer texture maps from the ray-tracer's segmented imagery that provide the best match when applied by graphics hardware. The parameterized textures are encoded as a multidimensional Laplacian pyramid on fixed size blocks of parameter space. This scheme captures the great 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 to take advantage of differences in coherence across different parameter dimensions and separate diffuse and specular lighting layers to further improve compression. We describe the run-time system and show high-quality results at compression ratios of 200 to 800 with interactive play back 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 far 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. Our imagery exhibits 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 [Ade91,McM95] can be generalized. We are free to parameterize the radiance field based on time, position of lights or viewpoint, surface reflectance properties, object positions, or any other degrees of freedom in the scene, resulting in an arbitrary-dimensional parameterized animation. Our goal is maximum compression of the 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 allowing 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. 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. In addition, the viewpoint can move from the original parameter samples without revealing geometric disocclusions.

Figure 1 illustrates our system. Ray-traced images at each point in the parameter space are fed to the compiler together with the scene geometry, lighting models, and viewing parameters. The compiler targets any desired type of graphics hardware and infers texture resolution, texture domain mapping, and texture samples for each object over the parameter space to produce as good a match as possible on that hardware to the “gold-standard” images. Per-object texture maps are then compressed using a novel, multi-dimensional compression scheme. The interactive runtime consists of two parts operating simultaneously: a texture decompression engine and a traditional hardware-accelerated rendering engine.

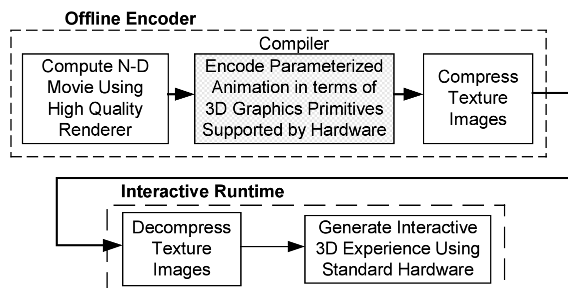


Figure 1: System Overview

Contributions:

- We introduce the problem of compressing multidimensional animations, not just radiance fields parameterized by viewpoint or animations through 1D time.
- We fully exploit cheap and ubiquitous graphics hardware by rendering texture maps on geometric objects rather than view-based images. We employ an automatic method to allocate storage over objects' texture maps and select texture map resolutions and domains based on the gold-standard images. We also separate diffuse and specular lighting layers to increase compression, using automatic storage allocation over these lighting layers.
- We present a novel strategy for texture inference from segmented images optimized for decoding on specific graphics hardware. It uses an optimization approach and introduces a pyramidal regularization term that ensures the entire texture is defined, with occluded regions filled smoothly.
- We present novel methods for general, multidimensional compression using an adaptive Laplacian pyramid that allows real-time decoding and high compression ratios.
- We describe a novel run-time system that caches to speed texture decoding and staggers block origins to distribute decompression load.
- We present realistic, highly specular examples with multiple objects containing thousands of polygons, using a PC equipped with a consumer graphics card. The quality and generality of our examples exceed previous work in image-based rendering. We demonstrate the superiority of our encoding over alternatives like MPEG4 and show high-quality results at compression ratios of 200-800 with near real-time (~2.4Hz) decoders capable of hardware implementation. Faster decoding (~9Hz) is also possible at reduced quality. Since the system's main bottleneck is texture decompression, our findings provide incentive for incorporating more sophisticated texture decompression functionality in future graphics pipelines.

Limitations:

- We assume that a list of the geometric objects and their texture parameterizations are given as input.
- Efficient encoding relies on parameter-independent geometry; that is, geometry that remains static or rigidly moving and thus represents a small fraction of the storage compared to the parameter-dependent textures. For each object, polygonal meshes with texture coordinates are encoded once as header information.
- The compiler must have access to an image at each point in parameter space, so compilation is exponential in dimension. We believe our compilation approach is good for spaces in which all but one or two dimensions are "secondary"; i.e., having relatively few samples. Examples include viewpoint movement along a 1D trajectory with limited side-to-side movement, viewpoint changes with limited, periodic motion of some scene components, or time or viewpoint changes coupled with limited changes to the lighting environment.

2 Previous Work

Image-Based Rendering (IBR) IBR has sought increasingly accurate approximations of the plenoptic function [Ade91,McM95], or spherical radiance field parameterized by 3D position, time, and wavelength. [Che93] pioneered this approach in computer graphics, using pixel flow to interpolate views. Levoy and Hanrahan [Lev96], and Gortler et al. [Gor96] reduced the plenoptic function to a 4D field, allowing view interpolation with view-dependent lighting. *Layered depth images* (LDI) [Sha98,Cha99] are another representation of the radiance field better able to handle disocclusions without unduly increasing the number of viewpoint samples. [Won99] and [Nis99] have extended this work to a 5D field that permits changes to the lighting environment. The challenge of such methods is efficient storage of the resulting high-dimensional image fields.

For spatially coherent scenes, Miller et al. [Mil98] and Nishino [Nis99] observed that geometry-based surface fields better capture coherence in the light field and achieve a more efficient encoding than view-based images like the LDI or lumigraph. In related work, [Hei99a] used a surface light field to encode reflected rays in glossy walkthroughs, [Stu97] and [Sta99] pre-computed a discrete sampling of the glossy reflection, and [Bas99] used an LDI to encode the reflected objects. Our work generalizes parameterizations based solely on viewpoint and automatically allocates texture storage per object. 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.

Another IBR hybrid uses *view-dependent textures* (VPT) [Deb96,Deb98,Coh99] 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 with no viewpoint dimensions. This is accomplished by generating per-object segmented images in the ray tracer and

inferring textures that match each segmented layer. In addition to our generalized parameterization, a major difference in our approach is that we use "intrinsic" texture parameterizations (i.e., viewpoint-independent *(u,v)* coordinates per vertex 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. Furthermore, disocclusions are handled without encoding which polygons are visible in which views or gathering polygons corresponding to different views in separate passes. To infer information corresponding to occluded regions of an object, we use a pyramidal regularization term in our texture inference (Section 3.2) that provides smooth "hole-filling" without a specialized post-processing pass.

Interactive Photorealism Another approach to interactive photorealism seeks to improve hardware shading models rather than fully tabulating incident or emitted radiance. Examples include the work of Diefenbach [Dief96], who used shadow volumes and recursive hardware rendering to compute approximations to global rendering, Ofek and Rappoport [Ofek98], who extended this work to curved reflectors, and Udeshi and Hansen [Ude99], who improved soft shadows and indirect illumination and added a separate compositing pass for specular reflections. Cabral et al. [Cab99] used image-based radiance distributions encoded in reflection maps for more photorealistic lighting. Kautz and McCool [Kau99] computed two texture maps to approximate a BRDF with hardware rendering. Heidrich and Seidel [Hei99b] encoded anisotropic lighting and specular reflections with Fresnel effects using hardware texturing. Even using many parallel graphics pipelines (8 for [Ude99]) 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 the gold-standard images is closely related to surface reflectance estimation in computer vision [Sat97,Mar98,Yu99]. Yu et al. [Yu99] recover diffuse albedo maps and a spatially invariant characterization of specularity in the presence of unknown, indirect lighting. 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 a particular graphics hardware target 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 [Wal97].

Compression Various strategies for compressing the dual-plane lumigraph parameterization have been proposed. [Lev96] used vector quantization and entropy coding to get compression ratios of up to 118:1 while [Lal99] used a wavelet basis with compression ratios of 20:1. [Mil98] compressed the 4D surface light field using a block-based DCT encoder with compression ratios of 20:1. [Nis99] used an eigenbasis (K-L transform) to encode surface textures achieving compression rations of 20:1 with eigenbases having 8-18 texture vectors. Such a representation requires an excessive number of "eigentextures" to faithfully encode highly specular objects. This prohibits real-time decoding, which involves computing a linear combination of the eigentextures. We use a Laplacian pyramid on blocks of the parameter space. This speeds run-time decoding (for 8×8 blocks of a 2D parameter space, only 4 images must be decompressed and added to decode a texture) and achieves good quality at compression ratios up to 800:1 in our experiments. Other work on texture compression in computer graphics includes Beers et al [Bee96], who used vector quantization on 2D textures for compression ratios of up to 35:1.

Another relevant area of work is animation compression. Standard video compression uses simple block-based transforms and image-based motion prediction [Leg91]. Guenter et al., [Gue93] observed that compression is greatly improved by exploiting information available in synthetic animations. In effect, the animation script provides perfect motion prediction, an idea also used in [Agr95]. Levoy [Lev95] 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 to the client. Cohen-Or et al. [Coh99] used view-dependent texture maps to progressively transmit diffusely-shaded, texture-intensive walkthroughs, finding factors of roughly 10 improvement over MPEG for scenes of simple geometric complexity. We extend this work to the matching of multidimensional animations containing non-diffuse, offline-rendered imagery by texture-mapping graphics hardware.

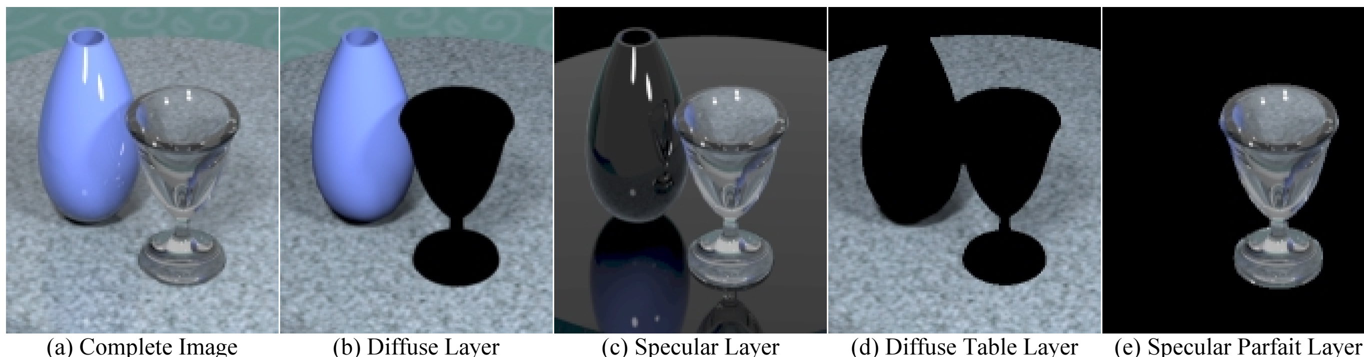
3 Parameterized Texture Inference

We infer texture maps using an optimization approach that models how the graphics hardware projects them to the screen. This is done by directly querying the target hardware using a series of test renderings of the actual geometry on that hardware (Section 3.2). Inferred texture maps can then be encoded (Section 4). To achieve a good encoding, it is important to determine an appropriate texture resolution and avoid encoding parts of the texture domain that are not visible (Section 3.4).

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. Bleeding decreases coherence and leads to misplaced silhouettes when the viewpoint moves away from the original samples. 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 then filter the relevant portion of the combined image as indicated by the mask and divide by the fractional coverage computed by applying the same filter to the object's mask. A gaussian filter kernel is used to avoid problems with negative coverages.

A second form of segmentation separates the view-dependent specular information from the view-independent diffuse information, in the common case that the parameter space includes at least one view dimension. This reduces the dimensionality of the parameter space for the diffuse layer, improving compression. As the image is rendered, the ray-tracer places information from the first diffuse intersection in a view-independent layer and all other information in a view-dependent one. Figure 2 illustrates segmentation for an example ray-traced image. We use a modified version of Eon, a Monte Carlo distribution ray-tracer [Coo84,Shi92,Shi96].



(a) Complete Image (b) Diffuse Layer (c) Specular Layer (d) Diffuse Table Layer (e) Specular Parfait Layer

Figure 2: 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 Least-Squares Method

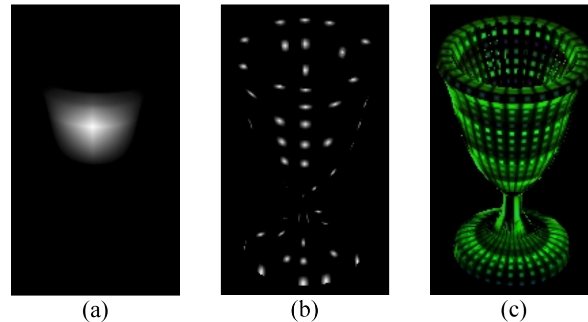
A simple algorithm for inferring an object's texture map from its segmented image maps each texel location to the image and then filters the neighboring region to reconstruct the texel's value [Mar98]. One problem with this approach is reconstruction of texels near arbitrarily-shaped object boundaries and occluded regions (Figure 2-d,e). It is also difficult to infer MIPMAPs when there are occluded regions whose corresponding regions in the texture image are undefined. 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:

$$\overbrace{\begin{bmatrix} \text{filter coefficients for } s_{0,0} \\ \text{filter coefficients for } s_{0,1} \\ \\ \vdots \\ \\ \text{filter coefficients for } s_{m-1,n-1} \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} \left.\begin{matrix} x^{0}_{0,0} \\ \vdots \\ x^{0}_{u-1,v-1} \end{matrix}\right\}\text{level}_0 \\ \left.\begin{matrix} x^{1}_{0,0} \\ \vdots \\ x^{1}_{\frac{u}{2}-1,\frac{v}{2}-1} \end{matrix}\right\}\text{level}_1 \\ \vdots \\ \left.\begin{matrix} x^{l-1}_{0,0} \\ \vdots \\ x^{l-1}_{\frac{u}{2^{l-1}}-1,\frac{v}{2^{l-1}}-1} \end{matrix}\right\}\text{level}_{l-1} \end{bmatrix}}^{x} = \overbrace{\begin{bmatrix} s_{0,0} \\ s_{0,1} \\ \\ \vdots \\ \\ s_{m-1,n-1} \end{bmatrix}}^{b} \qquad (1)$$

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. Note that this model ignores hardware nonlinearities in the form of rounding and quantization. While Equation 1 expresses the problem for just one color component, the matrix A is common across all color components.

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 the hardware z-buffer algorithm to determine object visibility on the screen, and need only consider rows (screen pixels) where the object is visible. Other rows are logically filled with zeroes but are actually deleted from the matrix, by using a table of visible pixel locations. Filter coefficients should sum to one in any row.[1]



(a) (b) (c)

Figure 3: Obtaining Matrix A. (a) Screen image with single texel in an 8x8 texture is set to full intensity value (b) Screen image when multiple texels in a 64×64 texture image are set to full intensity values, such that alternate 8×8 blocks do not overlap. (c) Screen image with 256×256 texture where two of the color components are used for encoding texels identifiers.

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 (Figure 3a), 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 (Figure 3b). 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 (Figure 3c).

For trilinear MIPMAP filtering, a given screen pixel accesses four texels in one MIPMAP level, as well as four texels either one level above or below having the same texture coordinates. To avoid corrupting the identifier, we must store the same texel identifier in the possible filtering neighborhood of a texel, as shown in Figure 4. By leaving sufficient spacing between texels computed in parallel, A can be inferred in a fixed number of renderings, P,

[1] In practice, row sums of inferred coefficients are often less than one due to truncation errors. A simple correction is to add an appropriate constant to all nonzero entries in the row. A more accurate method recognizes that each coefficient represents the slope of a straight line in a plot of screen pixel versus texel intensity. We can therefore test a variety of values and return the least squares line.

where P=6×6×3=108.[2] This assumes that the "extra" color components contain at least $\log_2(n/P)$ bits where n is the number of texels.

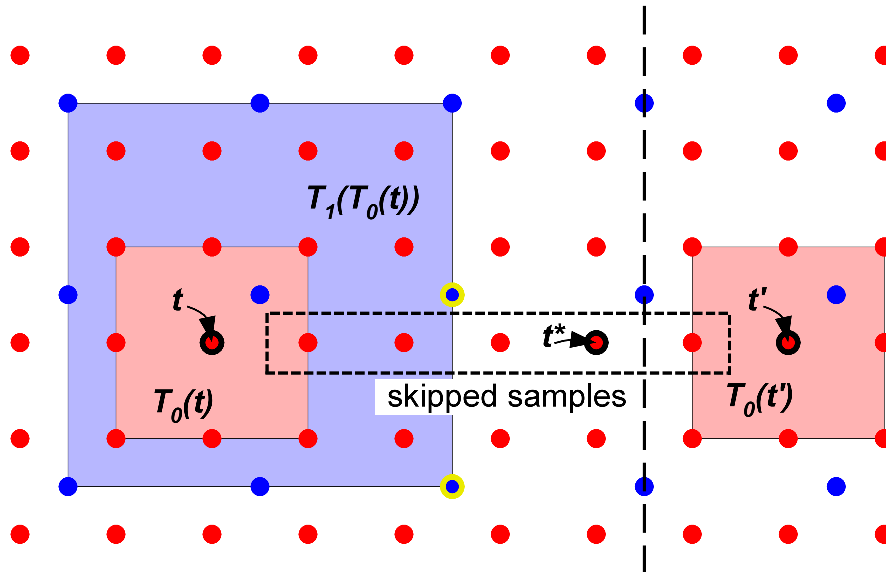


Figure 4: Trlinear Filtering Neighborhood for Parallel Matrix Inference: Red dots represent texel samples; blue dots are samples in the next higher level of the MIPMAP. To infer the filter coefficients at a sample t, we must ensure that all samples that could possibly be filtered with it to produce a screen pixel output have identical texel identifiers. The region $T_0(t)$ represents the region of texture space in the same MIPMAP level that could possibly access sample t with bilinear filtering, called its *level 0 neighborhood*. This region can possibly access samples from the next higher level of the MIPMAP shown in blue and labeled $T_1(T_0(t))$, the level 1 neighborhood of t's level 0 neighborhood. We must not solve in parallel a texel that shares any of these samples in its filtering neighborhood so only texels whose level 0 neighborhood are completely to the right of the dashed line are candidates. For example, the sample labeled t^* can not be solved in parallel with t since t^*'s level 1 neighborhood shares two samples with t, shown outlined in yellow. Even the sample to its right must be skipped since its level 0 neighborhood still includes shared samples at the next higher MIPMAP level. Sample t' is the closest to t that can be solved in parallel. Thus in each dimension, at least 5 samples must be skipped between texels that are solved in parallel.

Inference with Antialiasing

To antialias images, we can perform supersampling and filtering in the graphics hardware. Unfortunately, this decreases the precision with which we can infer the matrix coefficients, since the final result is still an 8-bit quantity in the framebuffer. Higher precision is obtained by inferring based on the supersampled resolution (i.e., without antialiasing), and filtering the matrix A using a higher-precision software model of the hardware's antialiasing filter.

Sub-pixels (rows in the supersampled matrix) that are not covered by the object should not contribute to the solution. As in the segmentation technique of Section 3.1, we filter the matrix A and then divide by the fractional coverage at each pixel as determined by the hardware rendering. Small errors arise because of minor differences in pixel coverage between the ray-traced and hardware-generated images.

Solution Method

A is 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

$$\begin{aligned} f(x) &= \|Ax-b\|^2 \\ \nabla f(x) &= 2A^T(Ax-b) \end{aligned} \qquad (2)$$

[2] This number is obtained by solving in parallel every sixth sample in both dimensions of the same MIPMAP level, and every third MIPMAP level, thus ensuring that possible filtering neighborhoods of samples solved in parallel do not interfere. For hardware with the power of two constraint on texture resolution, there is an additional technical difficulty when the texture map has one or two periodic (wrapping) dimensions. In that case, since 6 does not evenly divide any power of 2, the last group of samples may wrap around to interfere with the first group. One solution simply solves in parallel only every eighth sample.

subject to the constraint $0 \le x_{i,j}^k \le 1$. Availability of the gradient, $\nabla f(x)$, allows use of the conjugate gradient method to minimize *f(x)* [Pre92]. Since *f(x)* and $\nabla f(x)$ are most often evaluated in pairs, we can factor out the computation of *Ax-b*. The main computation of the solution's inner loop multiplies *A* or A^T with a vector representing the current solution estimate. 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* with a vector is proportional to n_s.

Another way to express the same *f(x)* and $\nabla f(x)$ is as follows:

$$\begin{aligned} f(x) &= x A^T A x - 2x \cdot A^T b + b \cdot b \\ \nabla f(x) &= 2A^T A x - 2A^T b \end{aligned} \tag{3}$$

Again, since *f(x)* and $\nabla f(x)$ are often evaluated simultaneously, we can factor out the computation of $A^T A x$, and precompute the constants $A^T A$, $A^T b$, and $b \cdot b$. In this formulation, the inner loop's main computation multiplies $A^T A$, an $n_t \times n_t$ matrix, with a vector. Since $A^T A$ is also sparse, though likely less so than *A*, the cost of multiplying $A^T A$ with a vector is proportional to n_t. We use the following heuristic to decide which set of equations to use:

```
if (2n_s ≥ K n_t)
    Use AᵀA method: Equation (3)
else
    Use A method: Equation (2)
```

where *K* is a measure of relative sparsity of $A^T A$ compared to *A*. We use *K* =4. The factor 2 in the test arises because Equation (2) requires two matrix-vector multiplies while Equation (3) 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 [Lue94]. This multiresolution solver idea can also be extended to the other dimensions of the parameter space. 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. We find the second method to be more efficient.

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:

$$\overbrace{\begin{bmatrix} A_{v_0} \\ A_{v_1} \\ \vdots \\ A_{v_{n-1}} \end{bmatrix}}^{A'} \begin{bmatrix} x \end{bmatrix} = \overbrace{\begin{bmatrix} b_{v_0} \\ b_{v_1} \\ \vdots \\ b_{v_{n-1}} \end{bmatrix}}^{b'} \tag{4}$$

where matrix A' coalesces the *A* matrices for the individual viewpoints v_0 through v_{n-1}, vector b' coalesces the ray-traced images at the corresponding viewpoints, and vector *x* represents the diffuse texture to be solved. Since the number of rows in A' tends to be much larger than the number of columns, we use the $A^T A$ method described earlier in Equation (3). In addition to speeding up the solver, this method also reduces memory requirements.

Regularization

Samples in the texture solution should lie in the interval [0,1]. To ensure this we add a regularizing term to the objective function *f(x)*, a common technique for inverse problems in computer vision [Ter86,Lue94,Eng96]. The term, called the *range regularization*, is defined as follows:

$$\begin{aligned} g(x_{ij}^k) &= \frac{1}{(x_{ij}^k + \delta)(1 + \delta - x_{ij}^k)} \\ f_{\text{reg-01}}(x) &= f(x) + \varepsilon_b \overbrace{\left(\frac{n_s}{n_t}\right)\left(\frac{1}{g(0) - g(1/2)}\right)}^{\text{calibration constant}} \sum_{ijk} g(x_{ij}^k) \end{aligned} \tag{5}$$

where $\delta = 1/512$. The function *g* approaches infinity at $-\delta$ and $1+\delta$ and thus penalizes texels outside the range. The regularizing term consists of three parts: a summation over all texels in *x* of the function *g*, a calibration constant giving the regularizing term roughly equal magnitude with *f(x),* and a user-defined constant, ε_b, that adjusts the importance of constraint satisfaction. We compute $\nabla f_{\text{reg-01}}$ analytically for the conjugate gradient method.

One of the consequences of setting up the texture inference problem in the form of Equation (1) is that only texels actually used by the graphics hardware are solved, leaving the remaining texels undefined. To support

graceful degradation away from the original parameter samples and to improve spatial coherence, all texels should be defined. This can be achieved by adding a second term, called the *pyramidal regularization*, of the form:

$$f_{\text{reg-pyramid}}(x) = f_{reg-01}(x) + \varepsilon_f \left(\frac{n_s}{n_t} \right) \Gamma(x) \tag{6}$$

where $\Gamma(x)$ takes the difference between the texels at each level of the MIPMAP with an interpolated version of the next coarser level as illustrated in Figure 5. The factor n_s/n_t gives the regularization term magnitude roughly equal with f.[3] Again, we compute $\nabla f_{\text{reg-pyramid}}$ analytically. This regularizing term essentially imposes a filter constraint between levels of the MIPMAP, with user-defined strength ε_f. We currently use a simple bilinear filter to allow fast construction of the MIPMAP during texture decoding.[4] We find that the first regularizing term is not needed when this MIPMAP constraint is used.

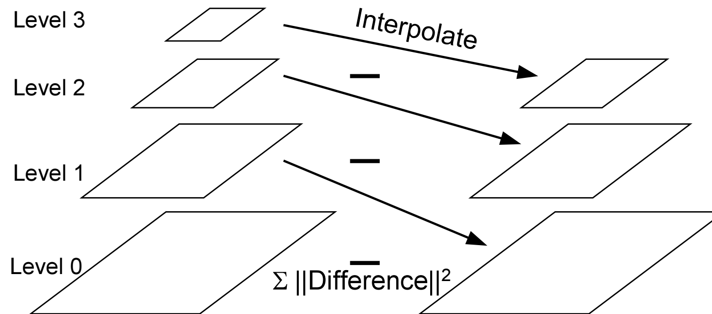
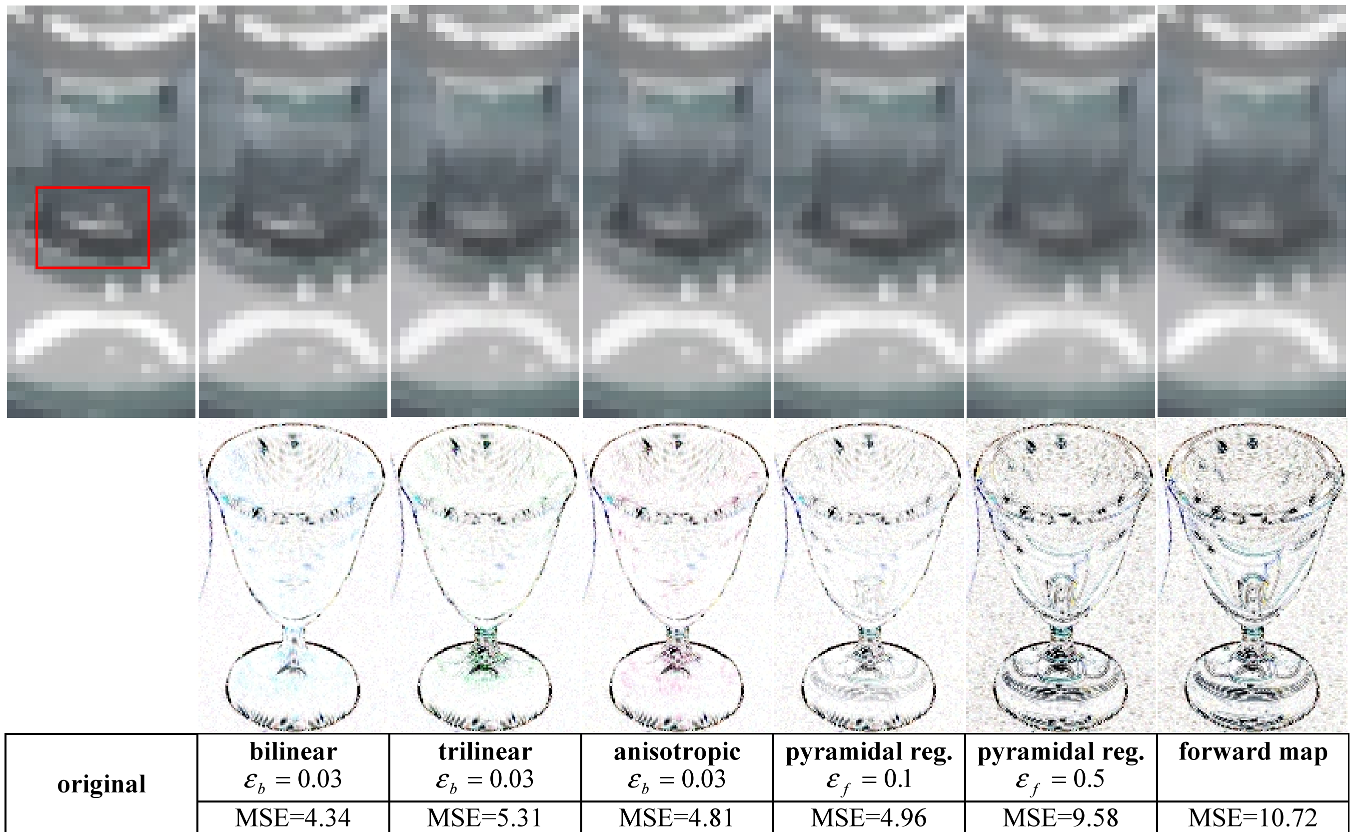


Figure 5: Pyramidal regularization is computed by taking the sum of squared differences between texels at each level of the MIPMAP with the interpolated image of the next higher level.

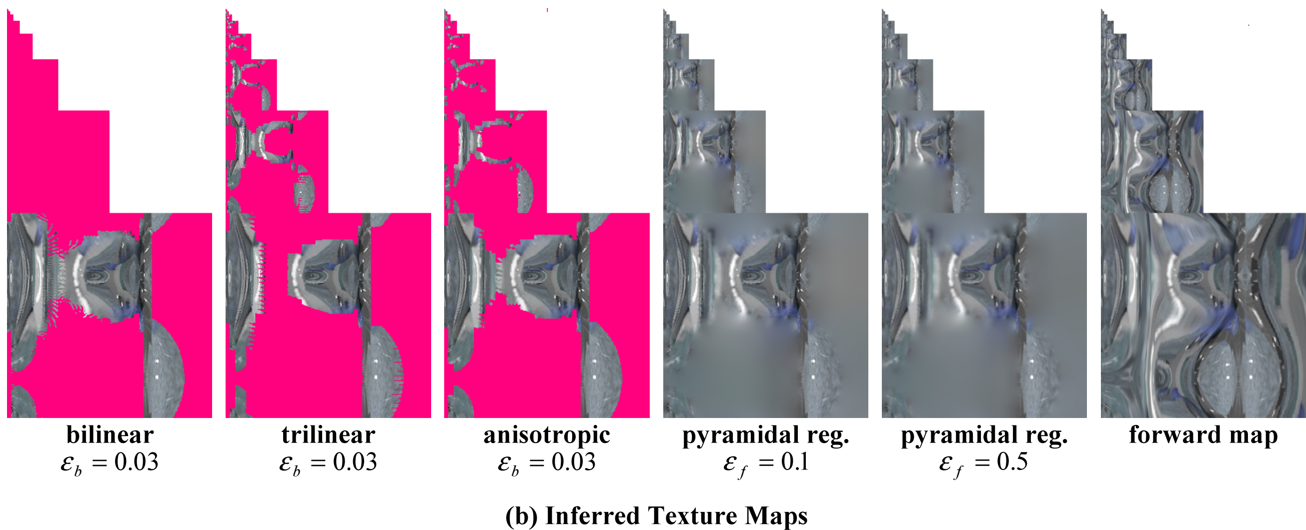
[3] The objective function f sums errors in screen space, while the two regularization terms sum errors in texture space. This requires a scale of the regularization terms by n_s/n_t.
[4] Note that while the solution step occurs during pre-preprocessing, it must account for whatever filter is actually used during the run-time processing to produce the best match.

3.3 Results with Least-Squares Method



original	bilinear $\varepsilon_b = 0.03$	trilinear $\varepsilon_b = 0.03$	anisotropic $\varepsilon_b = 0.03$	pyramidal reg. $\varepsilon_f = 0.1$	pyramidal reg. $\varepsilon_f = 0.5$	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

Figure 6: Texture Inference Results: (a) shows close-ups of the projected texture, compared to the original rendering on the far left. The white 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), or sum of squared pixel differences from the original image. (b) shows the corresponding texture maps. Pink regions represent undefined regions of the texture.

Figure 6 shows results of our least squares texture inference on a glass parfait object. The far left of the top row (a) is the image to be matched, labeled “original”. The next three images are hardware-rendered from inferred textures using three filtering modes on the Nvidia Geforce graphics system: bilinear, trilinear, and anisotropic. The

corresponding texture maps are shown in the first three columns of the next row (b). These three examples used only range regularization with $\varepsilon_b = 0.03$ and no pyramidal regularization. Most of the error in these examples is incurred on the parfait's silhouettes due to mismatch between hardware and ray-traced rendering. Also note that errors from texture inference can only be further degraded by lossy compression.

Bilinear filtering provides the sharpest and most accurate result because it uses only the finest level MIPMAP and thus has the highest frequency domain with which to match the original. Isotropic MIPMAP filtering produces a somewhat worse result, and anisotropic filtering is in between. Note the increase in texture area filled from the finest pyramid level for anisotropic filtering compared to trilinear, especially near the parfait stem. Better anisotropic filtering would decrease the difference between bilinear and anisotropic; the Nvidia chip supports only anisotropy factors up to 2. Note though that 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.* Even slight viewpoint changes away from those samples or decrease in image resolution during playback causes much larger errors.

The next two images show results of pyramidal regularization with anisotropic filtering. It can be seen that $\varepsilon_f = 0.1$ is almost identical to inference with no pyramidal regularization (labeled "anisotropic"), but $\varepsilon_f = 0.5$ causes noticeable blurring. The benefit of pyramidal regularization is that the entire texture is defined (i.e., the occlusion "holes" are all filled), allowing arbitrary movement away from the original viewpoint samples. Smooth hole filling also makes the texture easier to compress since there are no hard boundaries between defined and undefined samples. The regularization term makes MIPMAP levels tend toward filtered versions of each other; we exploit this fact by compressing only the finest level result of inference and creating the higher levels using on-the-fly decimation before the texture is loaded.

Finally, the far right image in (a) shows the "forward mapping" method in which texture samples are mapped forward to the object's image layer and interpolated using a high-quality filter (we used a separable Lanczos-windowed sinc function with 16 taps in both dimensions). To handle occlusions, we first filled undefined samples in the segmented layer using 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 this case, anisotropic hardware filtering was used). In addition, the reflections used to provide a good interpolation near occlusion boundaries fill up undefined texture regions with artificial, high-frequency information that is expensive to encode.

3.4 Optimizing Texture Coordinates and Resolutions

Since parts of an object may be occluded or off-screen, only part of its texture domain is useful. We therefore choose texture coordinates that minimize the texture area actually needed to render an object within a block of the parameter space (blocking will be discussed further in the next section). In performing this optimization we have three goals: to ensure there is adequate sampling of the visible texture image with as few samples as possible, to allow efficient computation of texture coordinates at run-time, and to minimize encoding of the optimized texture coordinates. To satisfy the second and third goals, we choose and encode a global affine transformation on the original texture coordinates rather than re-specify texture coordinates at individual vertices. Just six values are required for each object's parameter space block and texture coordinates can be computed with a simple, hardware-supported transformation. The algorithm follows:

1 Reposition branch cut in texture dimensions that have wrapping enabled
2 Find least-squares most isometric affine transformation
3 Compute maximum singular value of Jacobian of texture to screen space
   mapping and scale transformation along direction of maximal stretch
4 Repeat 3 until maximum singular value is below a given threshold
5 Identify bounding rectangle with minimum area
6 Determine texture resolution

We first attempt to reposition the branch cut in any texture dimensions that are periodic (i.e., have wrapping enabled). This adjustment realigns parts of the visible texture domain that have wrapped around to become discontiguous, for example, when the periodic seam of a cylinder becomes visible. A smaller portion of texture area can then be encoded. We consider each of the *u* and *v* dimensions independently, and compute the texture coordinate extents of visible triangle edges after clipping with the viewing frustum. If a gap in the visible extents exists, a branch cut is performed and texture wrapping disabled for that dimension.

We then find the affine transformation,[5] $R(u,v)$, minimizing the following objective function, inspired by [Mai93]

$$R(u,v) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

$$f(x) = \sum_{\text{edges } i} W_i \left(\frac{s_i - \left\| R(u_{i_0}, v_{i_0}) - R(u_{i_1}, v_{i_1}) \right\|}{\min\left(s_i, \left\| R(u_{i_0}, v_{i_0}) - R(u_{i_1}, v_{i_1}) \right\|\right)} \right)^2 \quad (7)$$

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. This minimization choses a mapping from texture space to the screen that is as close to an isometry as possible. As noted in [Mai93], two triangles are isometric when their edges have the same lengths. Hence, our objective function minimizes difference in lengths between triangle edges in texture space and on the screen. We normalize by the minimum edge length so as to equally penalize edges that are an equal factor longer and shorter. Conjugate gradient performs the minimization with $\nabla f(x)$ calculated analytically. Note that a rotational degree of freedom remains in this optimization which is fixed in step 5.

To ensure adequate sampling of an object's texture, we check the greatest local stretch (singular value) across all screen pixels in the block where the object is visible, using the Jacobian of the mapping from texture to screen space. Since the Jacobian for the perspective mapping is spatially varying even within a single polygon, this computation is performed separately at each screen pixel. If the maximum singular value exceeds a user-specified threshold (such as 1.25), we scale the affine transformation by the maximum singular value, divided by this threshold, in the corresponding direction of maximal stretch. This essentially adds more samples to counteract the worst-case stretching. We then iterate until the maximum singular value is reduced below the threshold, usually in a very small number of iterations.

The next step identifies the minimum-area bounding rectangle on the affinely transformed texture coordinates. by searching over a set of discrete directions. The size of the bounding rectangle also determines the optimal texture resolution, which may need to be rounded to a power of 2 in current hardware. Finally, since texture resolution substantially impacts performance due to texture decompression and transfer between system and video memory, our compiler accepts user-specified resolution reduction factors that scale the optimal texture resolution on a per-object basis.

[5] Note that when a branch cut is not possible over a "wrapped" or periodic dimension, we reduce the affine transformation to a scale transformation by fixing the values of *b* and *c* to zero. This ensures that the texture's periodic boundary conditions are not disturbed. Note also that the translational components of the affine transformation cancel from the objective function.

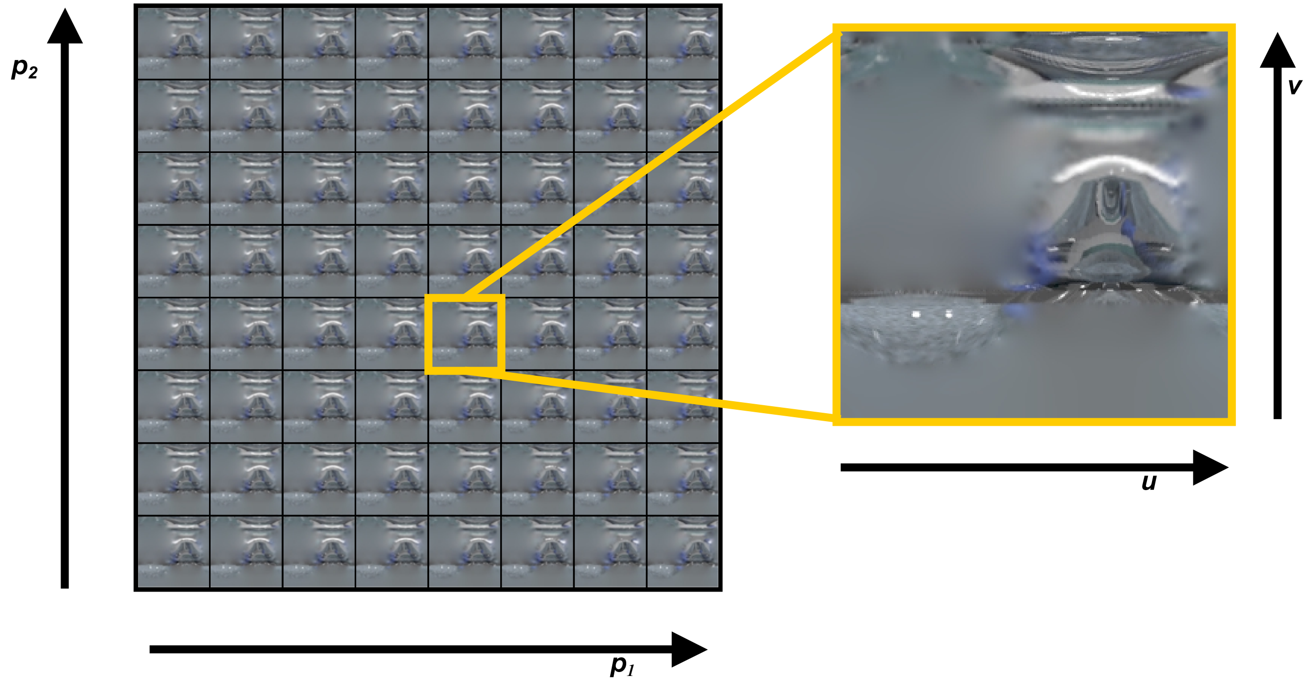


Figure 7: Parameter Block Compression. An 8×8 block of parameterized textures for a glass parfait object is shown. In this example, dimension p_1 represents a 1D viewpoint trajectory while p_2 represents the swinging of a light source. Note the high degree of coherence in the texture maps. One of the textures is shown enlarged, parameterized by the usual spatial parameters, denoted u and v. We use a Laplacian pyramid to encode the parameter space and standard 2D compression such as block-based DCT to further exploit spatial coherence within each texture (i.e., in u and v).

4 Parameterized Texture Compression

The multidimensional field of textures for each object is compressed by subdividing into parameter space blocks as shown in Figure 7. Larger blocks sizes better exploit coherence but are more costly to decode during playback; we use 8×8 blocks for our 2D examples.

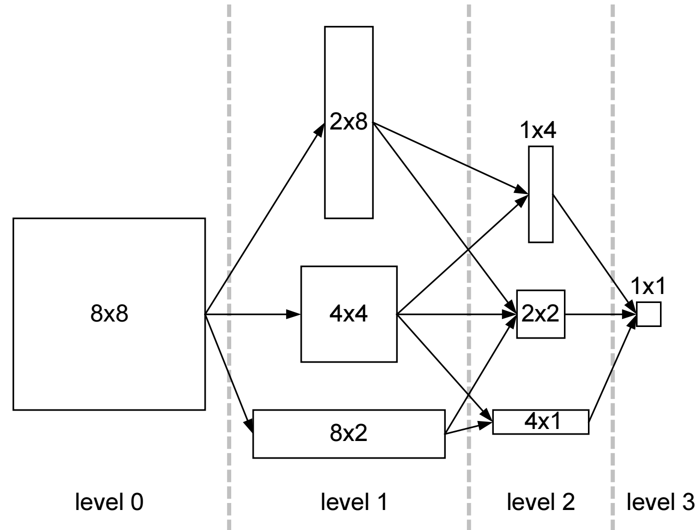


Figure 8: Adaptive Laplacian Pyramid

Adaptive Laplacian Pyramid

We encode parameterized texture blocks using a Laplacian pyramid [Bur83]. Consider a single (u,v) texture sample, parameterized by a *d*-dimensional space $\{p_1, p_2, \ldots, p_d\}$ with *n* samples in each dimension of the block. Starting from the finest (bottom) level with n^d samples, the parameter samples are filtered using a Gaussian kernel and subsampled to produce coarser versions, until the top of the pyramid is reached containing a single sample that averages across all of parameter space. Each level of the pyramid represents the detail that must be added to the sum of the higher levels in order to reconstruct the signal. Coherent signals have relatively little information at the lower levels of the pyramid, so this structure supports efficient encoding.

Though the Laplacian pyramid is not a critically sampled representation, it requires just $\log_2(n)$ simple image additions in order to reconstruct a leaf image. In comparison, a multidimensional Haar wavelet transform requires $(2^d - 1)\log_2(n)$ image additions and subtractions. Another advantage of the Laplacian Pyramid is that graphics hardware can perform the necessary image additions using multiple texture stages, thus enabling "on-the-fly" decompression[6]. For decoding speed, we reconstruct using the nearest-neighbor parameter sample; higher-order interpolation temporally smooths results but is much more expensive.

The "samples" at each pyramid level are entire 2D images rather than samples at a single (u,v) location. We use standard 2D compression (e.g., JPEG [Pen92,Xio96] and SPIHT [Sai96] 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 can not afford to process large fractions of the representation to decode a given parameter sample, a problem solved by using the Laplacian pyramid with fairly small block size.

In contrast, texture maps are atomically decoded and loaded into the hardware memory and so provide more opportunity for a software codec that seeks maximum compression without regard for random access. We anticipate that texture map decoding functionality will soon be absorbed into graphics hardware [Bee96]. In that case, whatever compressed representation the hardware consumes is a good choice for the "leaf node" texture maps.

It is typically assumed in image coding that both image dimensions are equally coherent. This assumption is less true 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. Figure 8 illustrates all the possible permutations of a 2D adaptive pyramid with four levels, in which coarser levels still have 4 times fewer samples as in the standard Laplacian pyramid. Though not shown in the figure, it is also possible construct

[6] Present graphics hardware based on DirectX 6.0 supports additions between unsigned fragment values and signed texels. However, no graphics hardware currently supports more than two stages in the texture pipeline. We expect this number to increase in the future, as there is logical support for up to eight texture blending stages in the DirectX API. Our present prototype implements image operations using MMX instructions on the host processor.

pyramids with different numbers of levels, for example to “jump” directly from an 8×8 level to an 8×1. We pick the permutation that leads to the best compression using a greedy search.

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 [Ger92,p.606-610]. Curves of mean squared error 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. More precisely, let $\overline{E}_i(r_i)$ be the mean squared error (MSE) in the encoding of level i when using r_i bits. It can be shown that the minimum sum of MSE over all levels subject to a total storage constraint of R; i.e.,

$$\min \sum_i \overline{E}_i(r_i) \quad \ni \quad \sum_i r_i = R$$

occurs when the $\overline{E}_1{}' = \overline{E}_2{}' = \cdots = \overline{E}_m{}'$, where m is the total number of levels and $\overline{E}_i{}' = d\overline{E}/dr_i$. We minimize the sum of MSEs 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. A simple 1D root finder suffices to find $\overline{E}_i{}'$ from which the r_i can be derived by inverting the rate/distortion curve at level i.

There is also a need to perform storage allocation across objects; that is, to decide how much to spend in the encoding of object A’s texture vs. object B’s. We use the same method as for allocating between pyramid levels, except that the error measure is $E_i \equiv A_i \overline{E}_i$, where A_i is the screen area and $\overline{E}_i$ 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.[7]

A complication that arises is that there can be large variations in MSE among different objects, some of which can be perceptually important foreground elements. We therefore introduce a constraint that any object’s MSE satisfy $\overline{E}_i \le \alpha \overline{E}$ where $\overline{E}$ is the average MSE of all objects and $\alpha > 1$ is a user-specified constant. A two-pass algorithm is used in which we first minimize $\sum_i E_i$ over objects subject to an overall storage constraint. Using the resulting $\overline{E}$, we then eliminate the part of the rate distortion curves of any object that incurs more MSE than $\alpha \overline{E}$ and solve again. This reallocates storage from objects with low MSEs to objects with above-threshold MSEs in such a way as to minimize sum of squared error in the below-threshold objects.

The above algorithms can also be used as a starting point for manual allocation of storage across objects, so that more important objects can be more faithfully encoded.

For objects with both specular and diffuse reflectance, we encode separate lighting layers for which storage must be allocated. We use the method described above on the entire collection of textures across objects and lighting layers.

Compensation for Gamma Correction

Splitting an object’s lighting layers into the sum of two terms conflicts with gamma correction, since $\gamma(L_1 + L_2) \ne \gamma(L_1) + \gamma(L_2)$ where L_i are the lighting layers and $\gamma(x) = x^{1/g}$ is the (nonlinear) gamma correction function. Typically, $g \approx 2.2$. Without splitting, there is no problem since we can simply match texture maps to a gamma-corrected version of the gold standard. With splitting, we instead infer textures from the original, uncorrected layers so that sums are correctly performed in a linear space, and gamma correct as a final step in the hardware rendering. The problem arises because gamma correction magnifies compression errors in the dark regions.

To compensate, we instead encode based on the gamma corrected signals, $\gamma(L_i)$, effectively scaling up the penalty for compression errors in the dark regions. At run-time, we apply the inverse gamma correction function $\gamma^{-1}(x) = x^g$ to the decoded result before loading the texture into hardware memory, and, as before, sum using texture operations in a linear space and gamma correct the final result. We note that the inverse gamma function employed, as well as gamma correction at higher precision than the 8-bit framebuffer result, is a useful companion to hardware decompression.

5 Runtime System

The runtime system performs three functions: decompressing and caching texture images, applying encoded affine transformations to vertex texture coordinates, and generating calls to the graphics system for rendering.

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 based on the notion of

[7] To speed processing, we compute errors in texture space rather than rendering the textures and computing image errors. We find this provides an acceptable approximation.

lifetimes. Whenever a texture image is accessed, we reset a count of the number of frames since the image was last used. When the counter exceeds a given lifetime, L, the memory for the decompressed image is reclaimed. Different levels of the Laplacian pyramid have different levels of priority since images near the top are more likely to be reused. Lifetimes are therefore computed as $L = ab^l$ where a is a constant that represents the lifetime for leaf nodes (typically 20), b is the factor of lifetime increase for higher pyramid levels (typically 4) and l represents pyramid level. Note that the number of images cached at each pyramid level and parameter space block changes dynamically in response to user behavior.

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, to more evenly distribute decompression load.

6 Results

6.1 Demo1: Light × View

Compression Results The first example scene (Figure 10, top) consists of 6 static objects: a reflective vase, glass parfait, reflective table top, table stand, walls, and floor. It contains 4384 triangles and was rendered in about 5 hours/frame on a group of 400Mhz Pentium II PCs, producing gold standard images at 640×480 resolution. 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.[8] The image field was encoded using eight 8×8 parameter space blocks, each requiring storage 640×480×3×8×8= 56.25Mb/block.

Our least-squares texture inference method created parameterized textures for each object, assuming trilinear texture filtering. The resulting texture fields were compressed using a variety of methods, including adaptive 2D Laplacian pyramids of both DCT- and SPIHT-encoded levels. Storage allocation over objects was computed using the method of Section 4, with a max MSE variation constraint of $\alpha = 1.25$. The decoded textures were then applied in a hardware rendering on the Gullemot 3D Prophet SDR graphics card with Nvidia Geforce 256 chip, 32Mb local video memory, and 16Mb nonlocal AGB memory running on a Pentium II 400Mhz PC. To test the benfits of the Laplacian pyramid, we also tried encoding each block using MPEG on a 1D zig-zag path through the parameter space. A state-of-the-art MPEG4 encoder [MIC99] was used. Finally, we compared against direct compression of the original images (rather than renderings using compressed textures), again using MPEG 4 with one I-frame per block. This gives MPEG the greatest opportunity to exploit coherence with motion prediction.

Figure 10 shows the results at two targeted compression rates: 384:1 (middle row) and 768:1 (bottom row), representing target storage of 150k/block and 75k/block respectively. Due to encoding constraints, some compression ratios undershot the target and are highlighted in yellow. All texture-based images were generated on graphics hardware; their MSEs were computed from the framebuffer contents. MSEs were averaged over an entire block of parameter space.

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, labeled "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. The SPIHT-encoded Laplacian pyramid is slightly better than DCT, exhibiting blurriness rather than block artifacts (observe the left hand side of the vase for the 768:1 row). The differences in the pyramid schemes between the 384:1 and 768:1 targets are fairly subtle, but can be seen most clearly in the transmitted image of the table top through the parfait. Of course, artifacts visible in a still image are typically much more obvious temporally. [Note to reviewers: Result figures should be carefully observed under bright light.]

For MPEG encoding of textures we tried two schemes: one using a single I-frame per block (IBBPBBP…BBP) labeled "MPEG-texture 1I/block", and another using 10 I-frames (IBBPBBIBBPBBI…IBBP) labeled "MPEG-texture 10I/block". The zig-zag path was chosen so that the dimension of most coherence varies most rapidly, in this case the light position dimension. Though single I-frame/block maximizes compression, it increases decoding time. In the worst case, accessing a parameterized texture requires 23 inverse DCT operations, 22 forward predictions, 1 backward prediction and 1 interpolation prediction for the single I/block case.[9] We do not believe the 1I/block encoding is practical for real-time decoding, but include the result for quality comparison. For the 10I/block, 4 inverse DCT's, 2 forward predictions, 1 backward prediction, and 1 interpolative prediction are

[8] The specific parameters were light source radius of 8, pendulum length of 5, distance of pendulum center from table center of 71, and angular pendulum sweep of 22°/sample.

[9] This analysis is also true for the MPEG-view encoding. Note that decreasing the number of I-frames per block in MPEG is somewhat analogous to increasing the block size, and thus the number of levels, in our pyramid schemes – both trade-off decoding speed for better compression.

required in the worst case. This is roughly comparable to our DCT Laplacian pyramid decoding, which also requires 4 inverse DCT operation, though pyramid reconstruction involves only 3 image additions rather than more complicated motion predictions.

The 10I/block MPEG-texture results have obvious block artifacts at both quality levels especially on the vase and green wallpaper in the background. They are inferior to the pyramid encodings. This is true even though we were unable to encode the scene with higher compression than 418:1, significantly less than the other examples in the bottom row. This result is not surprising given that MPEG can only exploit coherence in one dimension. The 1I/block results are better, but still inferior to the pyramid schemes at the 384:1 target, where the vase exhibits noticeable block artifacts. For the 768:1 target, the quality of MPEG-texture 1I/block falls between the SPIHT and DCT pyramids. Note that the MPEG-texture schemes still use many of the novel features of our approach: hardware-targeted texture inference, separation of lighting layers, and optimal storage allocation across objects.

Figure 11 isolates the benefits of lighting separation and adaptive Laplacian subdivision. These results were achieved with the Laplacian SPIHT encoding at the 384:1 target. With combined lighting layers, adaptive subdivision increases fidelity especially noticeable in the table seen through the parfait (Figures 11a and b); MSE across the block is reduced 20%. This is because textures, especially the parfait's, change much less over the light position dimension than over the view dimension. In response, the first level of pyramid subdivision occurs entirely over the view dimension. We then separately encode the diffuse and specular lighting layers, still using adaptive subdivision (Figure 11c). While this increases MSE slightly because additional texture layers must be encoded[10], the result is perceptually better, producing sharper highlights on the vase.

System Performance Average compilation and preprocessing time per point in parameter space was as follows:

texture coordinate optimization	.93 sec
obtaining matrix A	2.15 min
solving for textures	2.68 min
storage allocation across pyramid levels	.5 min
storage allocation across objects	1 sec
compression	5 sec
total compilation	5.4 min
ray tracing	5 hours

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 (frames per second or fps) for a diagonal trajectory that visits a separate parameter sample at every frame. The results for both DCT- and SPIHT-Laplacian pyramid encodings are summarized in the following table, and used compression at the 384:1 target:

Encoding	Texture decimation	Worst fps	Average fps
Laplacian DCT	undecimated	1.30	2.38
	decimated	3.51	8.77
Laplacian SPIHT	undecimated	0.18	0.39
	decimated	1.03	2.62

The performance bottleneck is currently software decoding speed. When all necessary textures are cached in decompressed form, our system achieves an average frame rate of 34 frames/second. To improve performance, we tried encoding textures at reduced resolution. Reducing texture resolution by an average factor of 11 (91%) using a manually specified reduction factor per object provides acceptable quality at about 9fps with DCT. [Note to reviewers: These are the real-time results shown in the video.] Decoding speedup is not commensurate with resolution reduction because it partially depends on signal coherence and decimated signals are less coherent.

6.2 Demo2: View × 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 12 for encodings using MPEG-view and Laplacian SPIHT.

This is a challenging example for our method. There are many specular objects in the scene, reducing the effectiveness of lighting separation (the gewgaw and parfait have no diffuse layer). 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. On the other hand, the gewgaw itself is more coherent in the view dimension because it rotates faster than the view changes. MPEG can exploit this

[10] Only the table-top and vase objects had separately encoded diffuse and specular layers; they were the only objects with diffuse and reflective terms in their shading model. Thus a total of 8 parameterized textures were encoded for this scene.

coherence very effectively using motion compensation along the rotation dimension. Though our method is designed to exploit multidimensional coherence and lacks motion compensation, our adaptive pyramid also responds to the unbalanced coherence, producing a slightly better MSE and a perceptually better image.

To produce these results, we manually adjusted the storage allocation over objects. Shading on the background objects (walls, floor, and table stand) is static since they are diffuse and the gewgaw casts no shadows on them. Their storage can thus be amortized over all 18 blocks of the parameter space. Because they project to a significant fraction of the image and can be so efficiently compressed, our automatic method gives them more storage than their perceptual importance warrants. We reduced their allocation by 72% and devoted the remainder to an automatic allocation over the foreground objects. Even with this reduction, the texture-based encoding produces less error on the background objects, as can be seen in Figure 12.

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

7 Conclusions and Future Work

Synthetic imagery can be very generally parameterized using combinations of view, light, or object positions, among other parameters, to create a multidimensional animation. While real-time graphics hardware fails to capture all the shading effects of a ray tracer running offline, it does provide a useful operation for quickly decoding such an animation 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 can be achieved with good quality and real-time decoding. Unlike previous work in multidimensional IBR, we also show our methods to be superior to a state of the art image sequence coder applied to a sensible collapse of the space into 1D.

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 to encode reflections, hardware shadowing algorithms, and per-vertex shading models. We have also done preliminary work in using other texture blending operations such as multiplication. This is useful for separating a high-frequency but parameter-independent albedo map from a low-frequency, parameter-dependent incident irradiance field. In any case, we believe that the discipline of measuring compression ratios vs. error for encodings of photorealistic imagery is a useful benchmark for proposed hardware enhancements.

Extending this work to deforming geometry should be possible using parameter-dependent geometry compression [Len99]. Another extension is to match photorealistic camera models (e.g., imagery with depth of field effects) in addition to photorealistic shading. This may be possible with accumulation-buffer methods [Hae90] or with hardware post-processing on separately rendered sprites [Sny98]. Use of perceptual metrics to guide compression and storage allocation is another important extension [Lub95,Sub99]. Further work is also required to automatically generate contiguous, sampling-efficient texture parameterizations over arbitrary meshes using a minimum of maps.

Finally, we are interested in measuring storage required as the dimension of the parameterized space grows 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 [Hal98]. 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.

8 References

[Ade91] 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, 3-20.

[Agr95] Agrawala, M., A. Beers, and N. Chaddha, "Model-based Motion Estimation for Synthetic Animations," *Proc. ACM Multimedia '95*.

[Bas99] 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.

[Bee96] Beers, A. C., M. Agrawala, and N. Chaddha, "Rendering from Compressed Textures," *SIGGRAPH 96*, August 1996, pp. 373-378.

[Bur83] Burt, P., and E. Adelson, "The Laplacian Pyramid as a Compact Image Code," *IEEE Transactions on Communications*, Vol. Com-31, No. 4, April 1983, 532-540.

[Cab99] Cabral, B., M. Olano, and N. Philip, “Reflection Space Image Based Rendering,” *SIGGRAPH 99*, August 1999, 165-170.

[Che93] Chen, S.E., and L. Williams, “View Interpolation for Image Synthesis,” *SIGGRAPH 93*, August 1993, 279-288.

[Cha99] Chang, C., G. Bishop, and A. Lastra, “LDI Tree: A Hierarchical Representation for Image-Based Rendering,” *SIGGRAPH 99*, August 1999, 291-298.

[Coh99] Cohen-Or, D., Y. Mann, and S. Fleishman, “Deep Compression for Streaming Texture Intensive Animations,” *SIGGRAPH 99*, August 1999, 261-265.

[Coo84] Cook, R.L., T. Porter, and L. Carpenter, “Distributed Ray Tracing,” *SIGGRAPH 84*.

[Deb96] Debevec, P., C. Taylor, and J. Malik, “Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach,” *SIGGRAPH 96*, August 1996, 11-20.

[Deb98] Debevec, P., Y. Yu, and G. Borshukov, “Efficient View-Dependent Image-Based Rendering with Projective Texture Maps,” In *9th Eurographics Rendering Workshop*, Vienna, Austria, June 1998.

[Dief96] Diefenbach, P. J., *Pipeline Rendering: Interaction and Realism Through Hardware-Based Multi-Pass Rendering,* Ph.D. Thesis, University of Pennsylvania, 1996.

[Eng96] Engl E., M. Hanke, and A. Neubauer. *Regularization of Inverse Problems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[Ger92] Gersho, A., and R. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic, Boston, 1992.

[Gor96] Gortler, S., R. Grzeszczuk, R. Szeliski, and M. Cohen, “The Lumigraph,” *SIGGRAPH 96*, 43-54.

[Gue93] Guenter, B., H. Yun, and R. Mersereau, “Motion Compensated Compression of Computer Animation Frames,” *SIGGRAPH 93*, August 1993, 297-304.

[Hae90] Haeberli, P., and K. Akeley, “The Accumulation Buffer: Hardware Support for High-Quality Rendering, *SIGGRAPH 90*, August 1990, 309-318.

[Hal98] Halle, M., “Multiple Viewpoint Rendering,” *SIGGRAPH 98*, August 1998, 243-254.

[Hei99a] Heidrich, W., H. Lensch, and H. P. Seidel. “Light Field Techniques for Reflections and Refractions,” Eurographics Rendering Workshop 1999, June 1999, pp.195-204, p. 375.

[Hei99b] Heidrich, W. and H. P. Seidel. “Realistic, Hardware-Accelerated Shading and Lighting,” *SIGGRAPH 99*, August 1999, pp. 171-178.

[Hüt99] Hüttner, T., and W. Straßer, “Fast Footprint MIPmapping,” *in Proceedings 1999 Eurographics/Siggraph Workshop on Graphics Hardware*, Aug. 1999, 35-43.

[Kau99] Kautz, J. and M. D. McCool, “Interactive Rendering with Arbitrary BRDFs using Separable Approximations,” *Eurographics Rendering Workshop 1999*, June 1999, pp. 255-268 and p. 379.

[Lal99] Lalonde, P. and A. Fournier, “Interactive Rendering of Wavelet Projected Light Fields,” Graphics Interface '99, June 1999, pp. 107-114.

[Leg91] Legall, D., “MPEG: A video compression standard for multimedia applications,” in *Communications of the ACM*, 34(4), April 1991, 46-58.

[Len99] Lengyel, J., “Compression of Time-Dependent Geometry”, *1999 ACM Symposium on Interactive 3D Graphics*, April 1999, 89-96 (April 1999).

[Lev95] Levoy, M., “Polygon-Assisted JPEG and MPEG compression of Synthetic Images,” *SIGGRAPH 95*, August 1995, 21-28.

[Lev96] Levoy, M., and P. Hanrahan, “Light Field Rendering,” *SIGGRAPH 96*, August 1996, 31-41.

[Lis98] Lischinski, D., and A. Rappoport, “Image-Based Rendering for Non-Diffuse Synthetic Scenes,” *Rendering Techniques ’98, Proc. 9th Eurographics Workshop on Rendering*.

[Lub95] Lubin, J., “A Visual Discrimination Model for Imaging System Design and Evaluation,” In E. Peli, editor, *Vision Models for Target Detection and Recognition*, World Scientific, 1995, 245-283.

[Lue94] 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, 41-64.

[Mai93] Maillot, J., H. Yahia, A. Verroust, “Interactive Texture Mapping,” SIGGRAPH ’93, 27-34.

[Mar97] Mark, W., L. McMillan, and G. Bishop, “Post-rendering 3D Warping,” *1997 Symposium on Interactive 3D Graphics*, 7-16.

[Mar98] Marschner, S. R., *Inverse Rendering for Computer Graphics*, Ph.D. Thesis, Cornell University, August 1998.

[McM95] McMillan, L., and G. Bishop, “Plenoptic Modeling,” *SIGGRAPH 95*, 39-46.

[Mic99] Microsoft MPEG-4 Visual Codec FDIS 1.02, ISO/IEC 14496-5 FDIS1, August 1999.

[Mil98] Miller, G., M. Halstead, and M. Clifton, "On-the-fly Texture Computation for Real-Time Surface Shading," *IEEE Computer Graphics and Applications*, March 1998, 44-58.

[Mil98] Miller, G., S. Rubin, and D. Poncelen, "Lazy Decompression of Surface Light Fields for Pre-computed Global Illumination," *Rendering Techniques '98 (Proc. Of Eurographics Rendering Workshop '98)* 1998, 281-292.

[Nis99] 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, p. 618-24 Vol. 1.

[Pen92] Pennebaker, W., and J. Mitchell, "JPEG Still Image Compression Standard," New York, Van Nostrand Reihold, 1992.

[Pre92] Press, W.H., B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, "Numerical Recipes in C: The Art of Scientific Computing," Cambridge University Press, 1992.

[Ofek98] Ofek, E. and A. Rappoport, "Interactive Reflections on Curved Objects," *SIGGRAPH 98*, July 1998, pp. 333-342.

[Ram99] Ramasubramanian, M., S. Pattanaik, and D. Greenberg, "A Perceptually Based Physical Error Metric for Realistic Image Synthesis," *SIGGRAPH 99*, August 1999, 73-82.

[Sat97] Sato, Y., M. Wheeler, and K. Ikeuchi, "Object Shape and Reflectance Modeling from Observation," *SIGGRAPH 97*, August 1997, 379-387.

[Sai96] Said, A., and W. Pearlman, "A New, Fast, and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees," In IEEE Transactions on Circuits and Systems for Video Technology, Vol. 6, June 1996.

[Sha98] Shade, J., S. Gortler, L. He, and R. Szeliski, "Layered Depth Images," *SIGGRAPH 98*, August 1998, 75-82.

[Shi92] Shirley, and Wang, "Distribution Ray Tracing: Theory and Practice," *Proceedings of the 3rd Eurographics Rendering Workshop*, Bristol, 1992.

[Shi96] Shirley, Wang and Zimmerman, "Monte Carlo Methods for Direct Lighting Calculations," *ACM Transactions on Graphics*, January 1996.

[Sny98] Snyder, J., and J. Lengyel, "Visibility Sorting and Compositing without Splitting for Image Layer Decompositions," *SIGGRAPH 98*, August 1998, 219-230.

[Sta99] Stamminger, M., A. Scheel, X. Granier, F. Perez-Cazorla, G. Drettakis, and F. X. Sillion, "Efficient Glossy Global Illumination with Interactive Viewing," *Graphics Interface '99*, June 1999, pp. 50-57.

[Stu97] Stürzlinger, W. and R. Bastos. "Interactive Rendering of Globally Illuminated Glossy Scenes," *Eurographics Rendering Workshop 1997*, June 1997, pp. 93-102.

[Ter86] Terzopoulos D., "Regularization of Inverse Visual Problems Involving Discontinuities," *IEEE Transactions on PAMI*, 8(4), July 1986, 413-424.

[Ude99] 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.

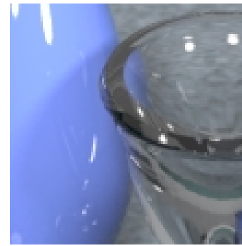
[Won97] Wong, T. T., P. A. Heng, S. H. Or, and W. Y. Ng, "Image-based Rendering with Controllable Illumination," *Eurographics Rendering Workshop 1997*, June 1997, pp. 13-22.

[Wal97] Walter, B., G. Alppay, E. Lafortune, S. Fernandez, and D. Greenberg, "Fitting Virtual Lights for Non-Diffuse Walkthroughs," *SIGGRAPH 97*, August 1997, 45-48.

[Xio96] Xiong, Z., O. Guleryuz, and M. Orchard, "A DCT-based Image Coder," *IEEE Signal Processing Letters*, November 1996.

[Yu99] 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



Original

MPEG-view, 355:1

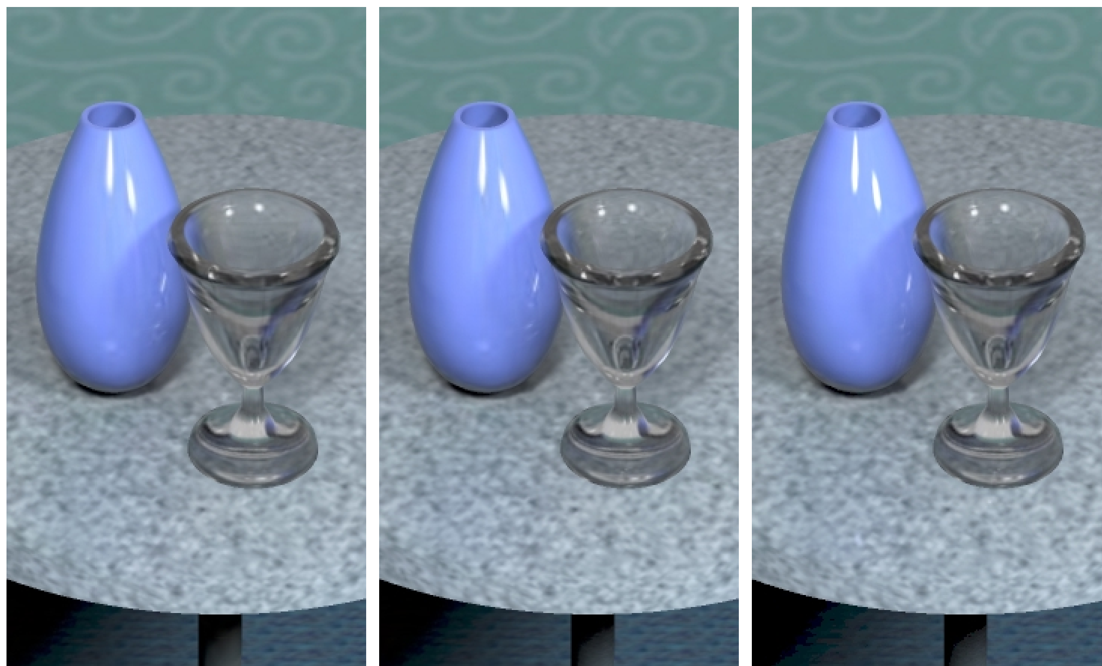
Laplacian DCT, 366:1

Laplacian SPIHT, 379:1

Zoomed-in, Target=384:1

View-based	**Texture-based**			
MPEG-view 1I/block	**MPEG-texture 1I/block**	**MPEG-texture 10I/block**	**Laplacian SPIHT**	**Laplacian DCT**
355:1, MSE=13.5	384:1, MSE=10.1	375:1, MSE=20.3	379:1, MSE=8.66	366:1, MSE=11.8
Target compression = 384:1				
570:1, MSE=29.8	760:1, MSE=25.9	418:1, MSE=25.0	751:1, MSE=16.5	732:1, MSE=18.1
Target compression = 768:1				

Figure 10: Demo1 Compression Results



(a) Combined, non-adaptive MSE=11.52 **(b) Combined, adaptive** MSE=9.64 **(c) Separated, adaptive** MSE=10.3

Figure 11: Benefits of Adaptive Subdivision and Lighting Separation

Original

MPEG-view, 361:1, MSE=10.9

Laplacian SPIHT, 361:1, MSE=10.3.

Figure 12: Demo2 Results