Eurographics Symposium on Rendering 2003, pp. 1–13
Per Christensen and Daniel Cohen-Or (Editors)

Interactive Time-Dependent Tone Mapping Using Programmable Graphics Hardware

Nolan Goodnight, Rui Wang, Cliff Woolley, and Greg Humphreys

Department of Computer Science, University of Virginia

Abstract

Modern graphics architectures have replaced stages of the graphics pipeline with fully programmable modules. Therefore, it is now possible to perform fairly general computation on each vertex or fragment in a scene. In addition, the nature of the graphics pipeline makes substantial computational power available if the programs have a suitable structure. In this paper, we show that it is possible to cleanly map a state-of-the-art tone mapping algorithm to the pixel processor. This allows an interactive application to achieve higher levels of realism by rendering with physically based, unclamped lighting values and high dynamic range texture maps. We also show that the tone mapping operator can easily be extended to include a time-dependent model, which is crucial for interactive behavior. Finally, we describe the ways in which the graphics hardware limits our ability to compress dynamic range efficiently, and discuss modifications to the algorithm that could alleviate these problems.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms I.4.8 [Image Processing and Computer Vision]: Scene Analysis—Photometry I.4.1 [Image Processing and Computer Vision]: Enhancement—Digitization and Image Capture

1. Introduction

Dynamic range is defined as the range of light intensities present in a scene. In the real world, very large dynamic ranges are commonplace, sometimes exceeding ten orders of magnitude. It is quite easy to produce such an image on a computer by using either a physically-based rendering system or a combination of multiple photographs taken at different exposures[7]. However, displaying these images presents a challenge for computer graphics since most output devices have a relatively small displayable dynamic range; frequently only integer intensities between 0 and 255 are permitted. This disparity has given rise to the field of tone mapping, whose broad goal is to optimize the mapping from an image with a large dynamic range to a display with a small dynamic range. Although the algorithms used to achieve this goal are diverse, they all typically operate as an offline process rather than in real-time: a high dynamic range (HDR) image is either synthesized by a rendering system or recovered from multiple photographs, and then the tone mapping algorithm processes that image, eventually producing a low dynamic range version. Improvements in CPU processing power have recently led to impressive advancements in this field, producing nearly artifact-free images that closely mimic the local adaptation abilities of the human eye.

We have also seen a recent revolution in high-performance graphics architecture. Previously fixed stages of the graphics pipeline have been replaced with fully programmable ones, giving the user complete control over the processing of vertices or fragments. The primary purpose of this design change is to enable complex visual effects in interactive graphics applications. Because of the streaming nature of graphics hardware, the graphics processing unit (*GPU*) is able to achieve extremely high computational rates; the pixel pipeline on NVIDIA's GeForce FX card are capable of sustaining 51 GFLOPS[24], which is roughly 8 times the computational power of the fastest Pentium 4 available today. Very recently, these programmable graphics pipeline stages have become sufficiently general that non-traditional tasks have been implemented on the GPU, such as ray-tracing, sparse matrix solving, and motion planning[12]. These algo-

rithms perform extremely well on the GPU because they are able to take advantage of the parallelism available in the pixel-processing portion of the pipeline.

In this paper, we explore the potential for using this programmability to add real-time tone mapping to interactive graphics applications. This allows a substantial increase in flexibility of application design and brings considerable added realism to interactive visual simulation. Because of the enormous computational power present in the GPU, we have been able to implement a state-of-the-art tone mapping algorithm at interactive rates. In our experiments, we found that some algorithms are better suited to implementation on a GPU than others. In particular, the photography-inspired techniques of Reinhard et al.[27] are especially well-suited to an interactive GPU implementation; we will describe our implementation in detail. We will also describe an alternate algorithm for computing photographic zones that produces qualitatively similar results but is much more amenable to a fragment processor implementation.

Finally, we show that a GPU-based tone mapping algorithm can easily be extended to contain a time-dependent term using a technique very similar to the one described by Durand and Dorsey[9]. This is important in an interactive setting, because the average luminance in a scene can change quickly (for example, when a bright light source suddenly comes into view), and we would like to avoid temporal discontinuities in brightness. In practice, we have found that having some amount of temporal adaptation is more important than the details of the algorithm used to achieve it.

2. Background and Related Work

Scenes in the real world have a dynamic range that far exceeds the capabilities of 8-bit-per-channel output devices. This is especially true of scenes that contain a combination of indoor and outdoor elements, such as a room illuminated through a window. A variety of tone mapping (or "tone reproduction") algorithms exist to display these high dynamic range (HDR) images on a low dynamic range device. Devlin et al. give an excellent comprehensive review of research in this area[8].

2.1. Tone Mapping

Tone mapping operators are usually classified as either global (spatially uniform) or local (spatially varying). Global operators apply a single luminance transform function to every pixel in the image. The simplest global operator is a linear map between the HDR image and the range of the output device. Linear scaling preserves relative contrast but removes most details contained in the image due to uniform scaling. Tumblin and Rushmeier first proposed the idea of tone mapping based on human perception; their method preserves the overall impression of perceived brightness[34]. Ward then proposed preservation of perceived contrast rather than brightness[36]: his visibility-preserving operator maps the smallest perceptible luminance difference in the HDR image to the smallest perceptible luminance difference of the display device. In a later paper, Ward Larson et al. presented a histogram adjustment technique based on the distribution of local luminance adaptation in a scene[20]. This technique also improves image realism by incorporating models for human contrast sensitivity, glare, spatial acuity and color sensitivity. Tumblin et al. then introduced a new operator based on the human visual adaptation process. Their operator decomposed an image into illumination and reflectance layers, then compressed the illumination layer while preserving details contained in the reflectance[33].

Global operators are simple and computationally efficient, but they have difficulty effectively preserving local contrast in most HDR images. Local operators solve this problem by using a spatially varying mapping, so two identical input luminances may be mapped to different output values based on properties of their local neighborhood.

Chiu et al. and Schlick presented early experiments in local tone mapping[4, 30]. Jobson et al. and Pattanaik et al. later presented multi-resolution techniques (such as a retinex-based method) that attempted to mimic the behavior of the human visual system[16, 25]. Tumblin and Turk developed the Low Curvature Image Simplifier (LCIS) method, which uses a formula inspired by anisotropic diffusion to detect gradient discontinuities, thereby preserving much of the local detail[35]. Their method works well, but can overemphasize details and also requires the user to set many parameters.

Recently, Fattal et al. presented a new method based on attenuating magnitudes of large luminance gradients[10]. Their method is conceptually simple and computationally efficient, although it does require the solution of a Poisson equation. Goodnight et al.[11] have implemented this algorithm on the GPU, though it does not run at interactive rates. Another recent paper by Reinhard et al. uses an approach inspired by Ansel Adams's photographic "zone system"[27]. A summary of their algorithm is presented in Section 2.3. This algorithm maps particularly well to programmable graphics hardware with a few modifications; our implementation of Reinhard's method is the primary focus of this paper.

Because of the computational complexity of tone mapping algorithms, interactive tone mapping techniques have received relatively little attention to date. Scheel introduced the application of tone reproduction in interactive walkthroughs[29]. This was done by modifying Ward Larson's operators[36, 20] and using textures to represent the luminance produced by global illumination rendering. However, Scheel's operator is still computationally demanding, and this approach does not incorporate any time-dependent adaptation. The interactive tone mapping framework presented by Durand and Dorsey[9] uses a multi-pass scheme that incorporates adaptation, glare, and loss of acuity. They use a global operator with time-dependent adaptation, incorporat-

ing a simple model for both light and chromatic adaptation. We adopt their time-dependent light adaptation to simulate the eye's adaptation in a dynamic setting, but apply it in the context of Reinhard's local tone mapping operator.

2.2. Programmable Graphics Hardware

Modern graphics hardware such as the NVIDIA GeForce FX[24] and the ATI Radeon 9700 and 9800[2] provides a flexible programming interface to the vertex and fragment portions of the graphics pipeline. Programs specified to these stages (known as *shaders*) execute in lockstep on a SIMD architecture and enjoy substantially higher peak floating-point performance than CPU programs. Although this computational horsepower has traditionally been used to enhance the visual appearance of interactive 3D rendering, GPUs have sufficient computational expressiveness to implement very different algorithms, as demonstrated by Purcell et al.'s raytracer that runs completely on graphics hardware[26].

Programmable vertex processing has limited applicability to general-purpose computation, because it cannot currently access memory (such as textures). Therefore, most general-purpose GPU computation work to date has concentrated exclusively on the programmable pixel pipeline. Even before programmability was added, special rasterization techniques were used to accelerate such diverse applications as motion planning[21], Voronoi diagrams[15], and radiosity[6, 17]. The addition of true fragment programmability has enabled the acceleration of myriad applications, including non-linear diffusion for solving partial differential equations[28], coupled-map lattices used to simulate boiling[13], and matrix multiplication[19, 32].

Many of these techniques had to work around or tolerate quirks of the programmable graphics hardware, such as limited precision or awkward programming models. Much less awkward programming techniques and interfaces are now possible, especially with the introduction of floating-point support in the GeForce FX and Radeon 9700 and of NVIDIA's high-level Cg programming language[22].

There has been little published research to date on integrating HDR images with a high-performance rendering pipeline. Cohen et al. represented and displayed high dynamic range texture maps (HDRTMs) using graphics hardware[5]. They first decompose 16-bit high dynamic range textures into two 8-bit texture maps and then perform dynamic exposure adjustment and gamma-correction of HDRTMs using programmable multitexturing. While this technique works well, it only uses a direct mapping from a slice of the HDRTM's range to the display, without performing any sophisticated tone mapping algorithms.

2.3. Review of Reinhard's Operator

We have chosen to implement the tone mapping operator of Reinhard et al.[27], although other operators could certainly be implemented as well. Reinhard's operator is based loosely on the "zone system" in photography[1]. First, a global scaling is applied that is analogous to setting an exposure level in a camera. Suppose $L_w(x,y)$ is world luminance of each pixel. The log average luminance is then given by

$$\overline{L}_w = \exp\left(\frac{1}{N}\sum_{x,y}\log\left(\delta + L_w(x,y)\right)\right) \tag{1}$$

where N is the number of pixels in the image, δ is a small constant used to avoid numerical underflow when taking the logarithm of black pixels. $\overline{L}_w$ is then mapped to the middle-grey zone by scaling pixel luminance with:

$$L(x,y) = \frac{a}{\overline{L}_w} L_w(x,y) \tag{2}$$

where a is a "key value" indicating whether a given image is subjectively light (high-key), normal, or dark (low-key). A normal-key image typically uses $a = 0.18$, which is the same value used by automatic exposure control in cameras.

Next, a simple global tone mapping operator is applied, obtaining display luminances $L_d(x,y)$:

$$L_d(x,y) = \frac{L(x,y)}{1 + L(x,y)} \tag{3}$$

This simple tone mapping operator appears to be sufficient to preserve details in low contrast areas, and it is guaranteed to bring all luminance within a displayable range of 0 to 1. However, Reinhard observes that details can be lost in images with very high dynamic range, especially in very bright regions. To counteract this effect, he uses a local contrast enhancement technique that is similar to photographic "dodging and burning".

First, the image is convolved with a set of Gaussian convolution kernels defined at multiple spatial scales, giving a set of responses V_i. Subtracting adjacent responses gives an estimate of the local contrast at multiple spatial scales. Reinhard uses a center-surround function given by

$$Activity(x,y,s_i) = \frac{V(x,y,s_i) - V(x,y,s_{i+1})}{2^{\phi} a / s_i^2 + V(x,y,s_i)} \tag{4}$$

to measure local contrast at a given scale s_i, using ϕ, which is a sharpening parameter controlling edge enhancement (set to 8.0 in the paper). Reinhard considers 8 scale levels; the smallest scale $s_1 = 0.35$ and $s_{i+1} = 1.6 \times s_i$.

For each pixel, the center surround function is computed from the lowest scale s_1, until the first scale s_m is found which satisfies $|Activity(x,y,s_m)| > \varepsilon$, where threshold ε is set to 0.05 by default. Essentially, s_m gives the largest area around a given pixel where no sudden contrast changes occur. Hence $V(x,y,s_m)$ can be used as local area luminance, replacing $L(x,y)$ in the denominator of Equation 3:

$$L_d(x,y) = \frac{L(x,y)}{1 + V(x,y,s_m)} \tag{5}$$

Because of the potential difference between $L_d(x,y)$ and

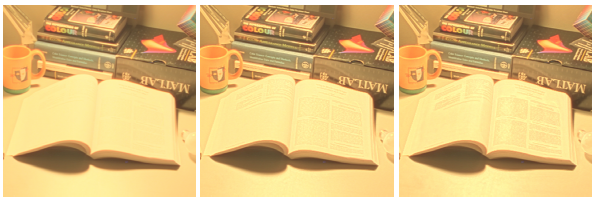


Figure 1: *Three images demonstrating different levels of local contrast preservation. The left image is compressed with the global transfer function, the middle with four adaptation zones, and the right with eight zones.*

$V(x,y,s_m)$, this new operator can retain substantial detail in very bright or dark regions.

This operator is particularly attractive for hardware implementation for two reasons. First, the global transfer function (Equation 3) is both simple to evaluate and highly effective at compressing HDR into a viewable range. If we use only the global operator, we can compress an image's dynamic range using a small number of rendering passes, simple fragment programs, and without any context switching. In addition, Equation 3 involves only one global computation: the log average luminance. Global computations are not particularly amenable to graphics hardware. For example, modern GPUs do not provide a mechanism for computing the average pixel value in a buffer. In Section 4 we discuss a straightforward reduction method that can be used to solve this problem. However, the technique is relatively expensive and so we would like to avoid operators that require even more global information about the image.

Second, while the local dodging and burning technique can be computationally expensive, the process lends itself to adaptive refinement. In other words, we can vary the number of adaptation zones depending on the level of detail we wish to preserve. This allows us to trade off efficiency and accuracy, which can be crucial for interactive applications. This idea is illustrated in Figure 1, which shows three images tone mapped using Reinhard et al.'s operator. As we increase the number of zones (from left to right) the computation time also increases. However, we are able to better preserve the detail in book text.

3. System Overview

Tone mapping algorithms require no high-level geometric or textural information from the application in order to compress the final output. We can therefore decouple our tone mapping system from any application that wants to use it.

3.1. Library API

Our system is implemented in a library that exports a small API (shown in Table 1). The API can be used by an application to compress its output prior to display. The application must first call `tmInit()` once during startup to initialize the tone mapping system. During the application's display routine, a call to `tmEnable()` causes all OpenGL calls to be redirected into an off-screen buffer in video memory that is local to the tone mapping system. Once all rendering is completed, the application uses `tmCompress()` to invoke the dynamic range compression algorithm. The results of this algorithm are placed in another buffer, which can either be bound to a texture unit using the `tmBind()` function for display by the application or sent to the display on behalf of the application by the library itself.

`tmInit`	Initializes the tone mapping system and allocates video memory for storing intermediate results.
`tmEnable`	Marks the start of rendering to be compressed. This function retargets all OpenGL calls to the floating-point buffer allocated by `tmInit`.
`tmDisable`	Turns off the tone mapping system and returns the application's rendering context to the exact OpenGL state that existed at the time of `tmEnable`.
`tmCompress`	Executes the actual tone mapping algorithm. The compressed image is held in a buffer local to the tone mapping system.
`tmBind`	Binds the output buffer of the tone mapping system to a specified texture unit. The application can then use that texture to display the result or read the data back for further processing or storage.

Table 1: *The interface between the application and our tone mapping system. The system exports a simple API that allows the application to control when its output is compressed.*

3.2. Data Layout

In many cases, it is not possible to represent high dynamic range imagery using only 8-bit color precision. Floating-point support is required to store the full range of the image as well as to resolve small differences between pixel values. Because many tone mapping algorithms involve multiple passes over the image, high precision is also necessary to avoid the visual artifacts of error propagation. Fortunately, graphics vendors have recently started to provide flexible pixel buffers (*pbuffers*) that support multiple pixel formats, including floating-point color representation. Pbuffers can be rendering targets as well as texture inputs. Additionally, each pbuffer can have several surfaces, which are exactly akin to the front and back surfaces used for double-buffered

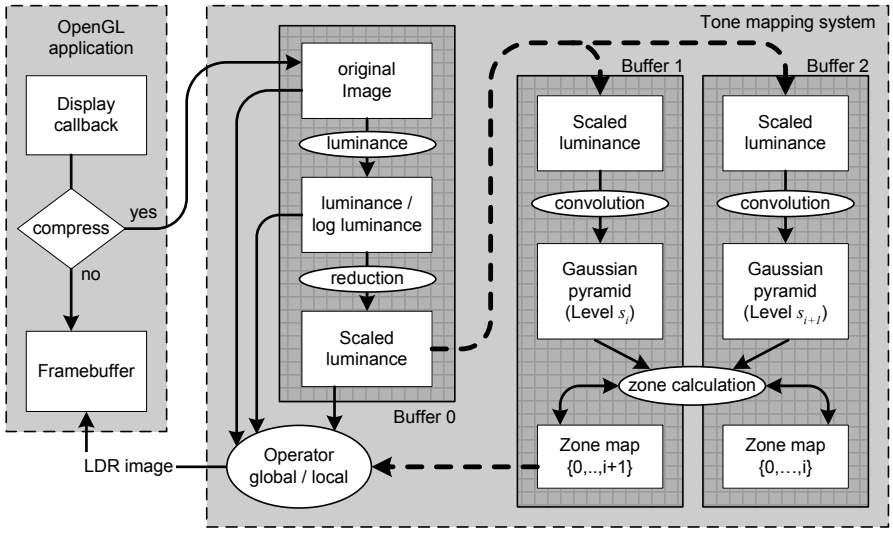


Figure 2: *A block diagram of our system for interactive tone mapping. Circular blocks represent shaders (or groups of shaders) that perform a particular part of the algorithm; rectangular blocks represent intermediate data storage. The global operator (Equation 3) is implemented using a single buffer (Buffer0) with multiple rendering surfaces. The local operator (Equation 5) requires two additional buffers (Buffer1 and Buffer2) to compute the Gaussian convolutions. Note that the local operator diagram illustrates one zone calculation (using level s_i and s_{i+1} in the Gaussian pyramid); this process is repeated for subsequent zones until all zones have been accumulated ($0,\ldots,i+1$ in the figure). In general, the arrows represent data flow as governed by the shaders. We only execute the dashed arrow paths (and all paths in Buffers 1 and 2) if we are running the local operator.*

rendering. In our tone mapping system, we store all image data in the surfaces of several floating-point pbuffers.

A high-level view of our shaders and the dataflow between them is presented in Figure 2. The system is divided into two conceptual components. The first uses a single pbuffer to store the initial HDR input from the application as well as several intermediate calculations needed to compute Reinhard's global transfer function (Equation 3). The second component requires two additional pbuffers, which are used to compute Gaussian convolutions and accumulate local adaptation zones for the local dodging-and-burning operations. Arrows in the figure represent render passes, where pixel data is manipulated and transferred between buffers.

4. Implementation

In this section we describe in detail how we compute Equation 3 (the global operator) and Equation 5 (the local operator) in graphics hardware. All of our algorithms are implemented using the OpenGL Architecture Review Board (ARB) fragment and vertex instruction sets. In the following text we frequently use the term *buffer* quite generally to denote any form of rendering target. These buffers, as described in the previous section, can be composed of multiple 4-channel surfaces. In many cases, implementation of the algorithms will require rendering back and forth among these surfaces to avoid reading from and writing to the same block of memory, but we omit these details in our explanation of the algorithms.

4.1. Global Operator

The global operator (Equation 3) is a monotonic, per-pixel transfer function that maps world luminance to display luminance. It is therefore quite simple to implement on the GPU, requiring only a few render passes. This is illustrated in the left half of Figure 2. Starting with the output from the application, we transform into the luminance domain. In the same pass, we compute the log luminance for every pixel, storing the luminance in one output channel and the log luminance in another. Unfortunately, computing the global average of the log luminance values requires a multi-pass approach in current hardware, which lacks any sort of global accumulator. The most straightforward approach is to perform repeated downsamplings, averaging four neighboring values down to one in each pass with a fragment shader; this is exactly equivalent to building a *mipmap* for traditional textures and is the approach also used by Krüger et al.[18]. With this method, the $\log(n)^{\text{th}}$ pass (where n is width of the image) results in a single value which is the log average luminance. This value is then read back into system memory and bound as a parameter to the fragment processor. In a final render pass, we access the world luminance L_w at each pixel (as calculated in the first pass), scale it according to Equation 2, and then convert it to display luminance L_d accord-

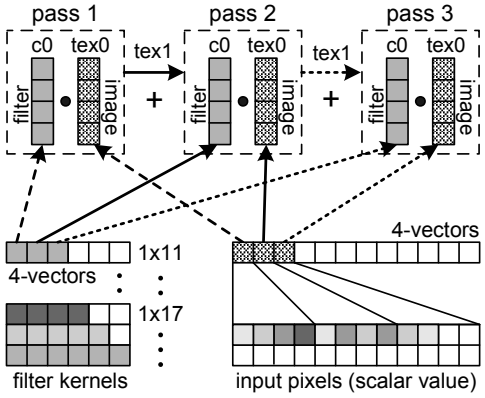


Figure 3: *A block diagram illustrating how we perform Gaussian convolutions on the GPU. We store each 4-vector element of a* $1 \times n$ *filter kernel in system memory and bind the values as parameters to the fragment pipeline. Likewise, we compress the image (scalar luminance) into a 4-channel texture map (shown in the bottom right). In this figure, the register used to store each element of the kernel is labeled* $c0$*. We bind the source image to texture unit 0 and accumulate previous results from texture unit 1. Arrows that correspond to the same render pass share the same drawing pattern.*

ing to Equation 3. To recover the compressed RGB display value, we scale the display luminance according to:

$$R_d = \left(\frac{R_w}{L_w}\right)^{\alpha} L_d, G_d = \left(\frac{G_w}{L_w}\right)^{\alpha} L_d, B_d = \left(\frac{B_w}{L_w}\right)^{\alpha} L_d \quad (6)$$

where α controls the saturation of the recovery; typical values of α are $0.4 \sim 0.8$.

4.2. Local Operator

Reinhard's local tone mapping operator (Equation 5) preserves detail by adding a local-area luminance term to Equation 3. In order to evaluate the center-surround function (Equation 4) used to determine local adaptation, we must first perform a series of Gaussian convolutions on the GPU. For large images or large kernels, it would be more efficient to perform this calculation in frequency space, where the convolutions can be replaced with a per-pixel multiplication, but this would require an implementation of FFT in hardware, and such a method has only recently been developed; see Moreland and Angel[23] for details. The OpenGL image subset extension provides methods for performing separable convolutions in the frame buffer[31] as an alternative, but support for this extension is missing on our target platform. We therefore found it necessary to implement an algorithm for performing arbitrary-sized convolutions on the GPU.

For example, given a $1 \times n$ filter kernel, we can express convolution at a point as a sum of 4-vector products. Since most GPU assembly languages provide a highly optimized 4-vector dot product instruction, we can perform convolutions efficiently by transforming a scalar-valued image into an array of 4-vectors. We start by binding four offsets as parameters to the vertex processor. These offsets correspond to the position of the filter kernel relative to the image. For example, in each dimension we define offsets that start at $-n/2$ and ultimately span the interval $-n/2$ to $n/2$. We then rasterize an image-sized quad to generate fragments. The offsets are used by the rasterizer to generate four sets of texture coordinates for every pixel in the source image, each corresponding to an adjacent pixel. We load those four adjacent pixel values into a single 4-vector floating-point register. By storing part of the filter kernel in another register we can compute a portion of the convolution with a simple dot product. In the next render pass, we repeat this process using the next 4-vector element of the kernel and corresponding vertex offsets, accumulating the results of each of these passes as we go along. The entire process is illustrated in Figure 3, which shows the three render passes required to convolve with a 1×11 Gaussian kernel. The process is identical for arbitrary-sized filters; for symmetry, we pad each kernel with zeros until it is a multiple of four.

Using the method just described, we can filter an image with an $n \times n$ separable kernel in $n/2+2$ render passes. We found that in some cases, however, it is more efficient to compute multiple 4-vector products per render pass. This approach reduces the number of passes required to compute a convolution, thus reducing any overhead associated with binding new shaders, parameters, or textures. As an example, consider a 49×49 Gaussian kernel. Using the method above, it would take 26 passes to convolve this kernel with any size image. By exploiting the rasterizer's capability of generating multiple texture coordinates per fragment and binding multiple 4-vector components of the kernel as fragment pipeline parameters, we can perform three dot products per pass instead of the single one described above. This reduces the total number of passes by more than half. Note, however, that the speedup in practice is only about 25%; we have not fundamentally reduced the number of computations required or the amount of memory accessed.

We also experimented with storing several kernels in a single 2D RGBA floating-point texture rather than in system memory. In this context, we can use the texture coordinate in one dimension to choose the appropriate kernel, while using the other dimension to access specific 4-vector elements of the kernel. While this eliminates the need to repeatedly transfer the kernel from system memory to GPU registers, it requires that we perform an extra texture lookup in the fragment program. This additional texture memory access actually caused the algorithm to run slower, suggesting that the system is memory-bandwidth limited. Further optimizations would therefore require that we reduce texture memory accesses in some way, perhaps by combining reads in a method similar to the one used by Bolz et al.[3]. We discuss this further in Section 6.1.

While performing any kind of convolution, it is important to properly deal with boundary conditions. In our current implementation, we use the common approach of replicating boundary pixels for all data access that falls outside the bounds of the image. In graphics hardware, all access to the image domain is through normalized (0 to 1) texture coordinates. We replicate boundary pixels by setting the `GL_TEXTURE_WRAP` parameter to `GL_CLAMP_TO_EDGE`, which guarantees that all texture coordinates outside the normal range return boundary values for a given texture.

4.3. Calculating adaptation zones on the GPU

With a GPU-based method for performing large kernel convolutions, we can easily compute $V(x,y,s_i)$ for any level s_i in a Gaussian pyramid. We could start by pre-allocating the entire pyramid $s_0, s_1, ..., s_i$, storing each level in a separate buffer, but this could result in the use of large amounts of video memory, especially if the image is screen-size. All that is necessary to determine a pixel's zone is the difference between neighboring levels in the pyramid, so we can perform all filtering computation using just two buffers. In addition to being memory-efficient, this approach makes it easy to dynamically decide how many zones to calculate without having to transfer pixel data among several rendering contexts.

Although we can compute all the adaptation zones using a single pass given sufficient hardware resources, this requires extensive use of conditionals in the fragment shader. We would need to evaluate the center-surround function at every resolution in the Gaussian pyramid, using conditionals at each level. Because fragment programs execute in lock-step on a SIMD architecture, conditionals are very expensive; all execution paths are evaluated on all fragments. Furthermore, such a fragment shader would have to have simultaneous access to all levels in the Gaussian pyramid, meaning we would have to precompute every filtered image and bind them as input textures before calculating zones. In order to avoid these complications, we build the adaptation zone map in multiple passes using a cumulative process.

In a given render pass, we mark all pixels corresponding to a single zone. This process involves four buffers: two buffers used to store adjacent levels in the Gaussian pyramid and two buffers for accumulating adaptation zones. For example, if we have already calculated zone i, we can calculate zone $i+1$ using the following steps:

1. Filter the scaled luminance according to level s_{i+2} in the Gaussian pyramid.
2. Set the zone buffer used to store zones $0, ..., i-1$ as the render target.
3. Bind Gaussian pyramid level s_{i+1} (filtered in the previous pass) and s_{i+2} as well as the remaining zone buffer as input textures.
4. Render an image-sized quad with the zone computation fragment shader activated.

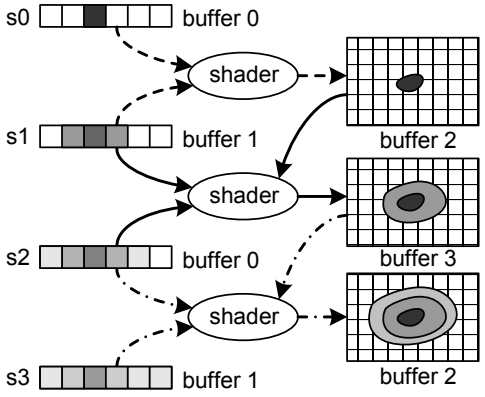


Figure 4: *An illustration of how we accumulate adaptation zones in graphics hardware. We use two buffers to store adjacent levels in a Gaussian pyramid, which is labeled s0 through s3. The zone information is accumulated using another two buffers which we use as alternating render targets. The arrows in the figure are drawn with different patterns to distinguish each render pass. Arrows entering the shader block represent input textures for that particular pass.*

The shader we use to calculate zones is a straightforward implementation of a center-surround threshold. For every pixel where $|Activity(x,y,s_i)| > \varepsilon$, we output the luminance from level s_i to the target buffer. A texture lookup on the input zone buffer allows us to determine whether each pixel has already been assigned a zone; those that have are copied through to the target buffer. Pixels for which no zone has previously been selected and which are not chosen for the current zone by the above inequality are left unmodified and empty in the target buffer using the *fragment kill* operation of the graphics hardware. This process is illustrated in Figure 5 for the calculation of three zones using a total of four levels in the Gaussian pyramid.

Figure 5 shows false-color visualizations of a zone map; one image is computed using our software implementation of Reinhard's local operator (Equation 5), and the other is computed in hardware. A total of eight zones is shown; darker region represent larger discontinuities in the luminance. The images are nearly identical; the small disparity is due to floating-point imprecision on the GPU (see Section 5.2 for an error analysis).

4.4. Time-Dependent Model

Interactive applications can often suffer from large temporal discontinuities in dynamic range (when, for example, a light source comes into view). We would like our tone mapping algorithm to smooth those discontinuities over time in order to create a more natural and plausible-looking animation. To do this, we have incorporated a model of time-dependent adaptation proposed by Durand and Dorsey[9]. The details of

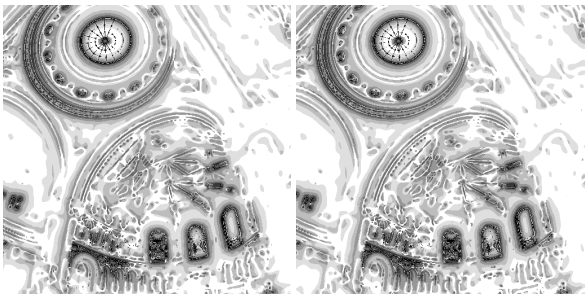


Figure 5: *False-color visualizations of the adaptation zones map as generated using Reinhard's local operator (Equation 5). Each of the eight zone values is normalized between 0 and 1, where darker regions represent lower levels in the Gaussian pyramid. We use 0.05 for the activity threshold. The image on the left is generated in software, and the image on the right is from our GPU implementation. The two images are nearly identical; differences arise due to the GPUs limited floating-point precision.*

this model can be found in their paper; we summarize the key points below.

This model simulates both multiplicative and subtractive light adaptation by applying a global multiplicative scale factor m during the mapping from world luminance L_w to display luminance L_d ($L_d = mL_w$). When the dynamic range changes suddenly, sensitivity recovery is simulated by changing m with an exponential filter: $\frac{dm}{dt} = \frac{m^* - m}{\tau}$, where m^* is the unmodified scale factor that would be used without time-dependent adaptation, and τ is a parameter controlling how fast the viewer will adapt to changes in light intensity.

To use this model in our GPU-based system, we apply the exponential filter to the log average luminance ($\overline{L}_w$) for each frame. Recall that Reinhard's operator first scales world luminance by $\frac{a}{\overline{L}_w}$ (see Equation 2), mapping the overall brightness of the image to a subjective key value a for display. Modulating the log average luminance by the exponential filter is therefore equivalent to controlling the gain m in Durand's case. We therefore use $\frac{d\overline{L}_w}{dt} = \frac{\overline{L}_w^* - \overline{L}_w}{\tau}$ to simulate light adaptation, where $\overline{L}_w^*$ is the target log average luminance.

Although this is a simple model of adaptation and is not physically based (Durand and Dorsey give more involved models that more closely mimic the human eye), it produces qualitatively reasonable results. Our experience has been that the presence of a time adaptation model is much more important than the details of the model itself, since the visual content often changes much more quickly than the dynamic range.

5. Results

All of our experiments have been conducted using the ATI Radeon 9800 Pro graphics card. This architecture supports medium-precision (24-bit) floating-point computations and texture maps. All example images and timing reports were recorded on a dual-processor AMD Athlon 1800+ MP system with 512MB of memory running Windows XP.

To test our implementation, we developed an OpenGL application that uses high dynamic range textures. The application first creates a window of the same size as the HDR image to be compressed. The application then renders a full-window quad that is textured with the HDR image. Without tone mapping, the output is clamped to the range of 0 to 1. Our application allows the user to pan a "tone map window" over the image to run our algorithm on a subset of the full HDR image. This allows us to test our time-dependent model easily by panning between dark and bright regions. Our algorithms are invoked simply by drawing the tone map window quad with our pixel shaders. Figure 6 shows a screenshot of this test application. The background image is a direct output to the application's frame buffer; the smaller viewport shows the tone mapped portion of the image.

5.1. Performance

In this section we discuss the performance of our GPU-based tone mapping system. The graph in Figure 7 gives frame

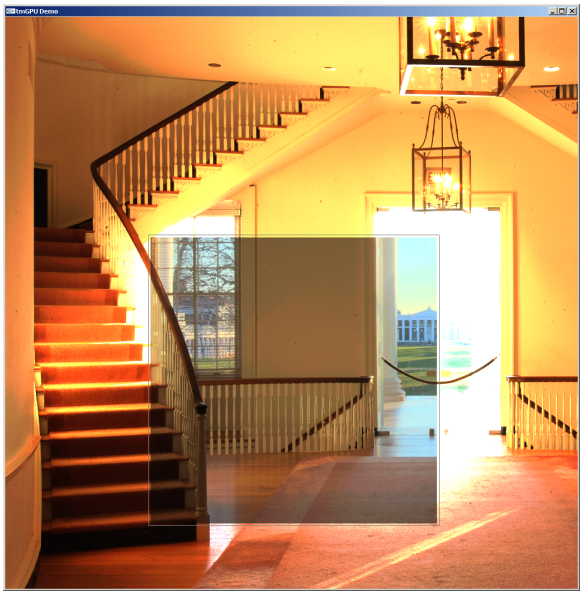


Figure 6: *A* 1024×1024 *HDR image from inside the Rotunda at U.Va. The dynamic range is roughly 150,000:1. The background is clamped to the range of 0 to 1; the smaller image* (512×512) *is compressed using our hardware implementation of Equation 3.*

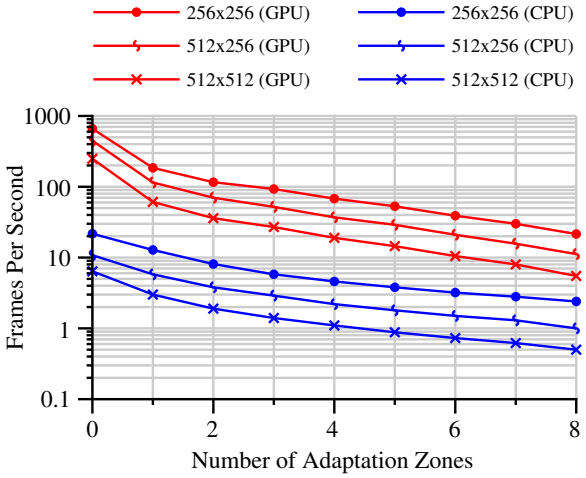


Figure 7: *A log graph of frame rate achieved by our system compared to frame rate of a CPU implementation of the same algorithms. The curves show data for three different image resolutions and zones ranging from zero to eight. For the* 256×256 *case, we can maintain >20 fps in all cases. However, for* 512×512 *the frame rate is not quite interactive for a large number of zones.*

rates achieved by our system as well as frame rates for the same algorithms implemented in software. We should note that our CPU implementation is by no means highly optimized, but it is reasonably efficient. As with the GPU, we build the Gaussian pyramid using spatial convolutions instead of more efficient frequency space techniques (as used by Reinhard et al.). In addition to this we use a relatively expensive recovery function (Equation 6). However, the two implementations are consistent with regard to complexity.

In the case of the global operator (zero zones), we are able to achieve extremely high frame rates for all of the listed image resolutions. This is not really surprising considering the simplicity of the transfer function. In fact, we found that for the global operator a substantial portion of the computation is spent building the mipmap. For example, disabling this step resulted in roughly a 60% speedup in some cases. However, as the number of zones increases, adding local adaptations, the Gaussian convolutions quickly become the bottleneck. To compute eight zones we must convolve with filter kernels ranging from 3×3 pixels to 49×49 pixels. Even with an efficient GPU-based convolution algorithm, such large kernels prevent us from maintaining real-time frame rates on today's hardware. With eight adaptation zones, our system runs at around 5 Hz for a 512×512 image and 20 Hz at 256×256. While 5 frames per second is not interactive, it does showcase the sheer computational power of the fragment hardware, and real-time frame rates (>30 Hz) are easily achievable if we limit ourselves to smaller image resolutions or a smaller number of zones. A gallery of results at 512×512 generated at roughly 30 Hz each is shown in Plate 1.

The given frame rates are calculated from the time taken to run the tone mapping algorithm in hardware. The overall frame rate would obviously be lower when we factor in the time taken by the application itself. The sudden falloff in frame rate when we move from the global operator to the local one is simply due to additional overhead incurred in the local case (significantly more memory reads and writes must occur to prepare for the zone calculations).

5.2. Accuracy

Tone mapping algorithms can be quite susceptible to numeric imprecision, and this is especially true for local operators because of their computational complexity. For example, without sufficient precision when evaluating the center-surround function, it is possible that some pixels will be assigned incorrect adaptation zones. This can introduce unpleasant visual artifacts such as halos in the compressed image. Because local operators tend to require significantly more computation that global operators, any shortcomings in precision can bring about compounded error. In order to avoid these problems, software implementations of tone mapping algorithms typically store data and perform all calculations using IEEE (32-bit) single-precision floats. However, our target GPU architecture, ATI's Radeon 9800, only supports 24-bit floating-point computations. In an effort to quantify the effects of this limited precision, we have run a series of experiments comparing output from the GPU to a software implementation of the same algorithm. To compare the results, we evaluate Root Mean Squared (RMS) percent error between the CPU and GPU implementations as:

$$\text{error}_{\text{RMS}\,\%} = \sqrt{\frac{1}{n}\sum_{x,y}\left[\frac{p_{\text{cpu}}(x,y) - p_{\text{gpu}}(x,y)}{p_{\text{cpu}}(x,y)}\right]^2} \qquad (7)$$

where n is the number of pixels in the image and $p(x,y)$ is the pixel value. We also evaluate the mean percent error as:

$$\text{error}_{\text{Mean}\,\%} = \frac{1}{n}\sum_{x,y}\left|\frac{p_{\text{cpu}}(x,y) - p_{\text{gpu}}(x,y)}{p_{\text{cpu}}(x,y)}\right| \qquad (8)$$

Table 2 gives error calculations for images resulting from several stages in the algorithm.

The scaled luminance error takes into account all numeric inaccuracies accumulated by the repeated averaging technique as well as the transform to luminance space. Given the simplicity of these computations, we would expect this error to be small, and our experiments verified that this is in fact the case. The convolution examples show the effects of 24-bit floating-point precision over the course of many rendering passes. Naturally, the image that was filtered with the larger kernel contains more error. The local area luminance image has a much higher percent error than the previous examples. This is due to the fact that even small errors

Image (computation)	RMS % error	mean % error
Scaled luminance	0.022 %	0.022 %
Convolution (5×5)	0.026 %	0.026 %
Convolution (49×49)	0.032 %	0.032 %
Local area luminance	4.552 %	0.764 %
Final image	1.051 %	0.177 %

Table 2: *RMS percent error and mean percent error for different stages in our GPU implementation of Reinhard et al's local tone mapping operator. These values are calculated by treating the output from our CPU implementation as the accepted value. The relatively large RMS percent error for the local area luminance image can be attributed to slight variation in the number of pixels in each zone.*

in the threshold comparison can cause slight variations in the boundaries between zones. In other words, if the width of the Gaussian kernel increases significantly between levels in the pyramid (as it does in this algorithm), small finite difference errors can result in large errors in the local area luminance image. Fortunately, we have found that the visual impact of this is more or less negligible, and the error in the final tone mapped image is significantly smaller.

6. Discussion

We have shown that the graphics pipeline has sufficiently evolved to support sophisticated tone mapping algorithms and compress images at interactive rates. This is not a panacea, however. Many questions remain, including when interactive tone mapping is most effective, and which tone mapping algorithms are best suited to interactive applications.

6.1. Optimizations

A troublesome aspect of GPU programming is that it requires exceedingly careful optimization in order to extract the performance we would expect. A number of factors contribute to this problem, such as memory bandwidth, driver overhead (especially context-switching overhead), etc. Approaches to these problems have been explored at length in several recent papers, including Bolz et al.[3] and Goodnight et al.[11].

Of particular note is a caveat to the use of pbuffers, which is that they cannot share a rendering context with the application. Context switching can become a serious bottleneck for an algorithm that must transfer data among a large number of buffers. We have minimized context switching by allocating pbuffers with multiple rendering surfaces (`GL_FRONT`, `GL_BACK`, `GL_AUX`*i*, etc.), all of which share the same rendering context. While we have been able to implement large portions of our tone mapping algorithm using only a small number of buffers for data storage, some amount of context switching is unavoidable.

The remaining major bottleneck is certainly memory bandwidth, as we would expect in these types of algorithms. Memory accesses can be reduced somewhat by more tightly packing the data[3], a technique that works well for a number of general-purpose GPU algorithms. In the case of interactive tone mapping, however, this data packing step (and the associated unpacking afterward) would have to occur once per frame, making it unclear that such a technique would give a significant speedup.

6.2. The Effect of Varying Frame Rate on Our Time-Dependent Model

Obviously, we only have an opportunity to apply our tone mapping algorithm once per frame. When we apply the exponential filter described in Section 4.4, we need an estimate of the elapsed time Δt in order to determine how much to change the gain m. If the frame rate is very high, then Δt is low and m changes smoothly over time, giving a very convincing impression of adaptation.

If, however, the application's frame rate is low, then using elapsed wall-clock time in our time-dependent model gives rise to large luminous discontinuities. Although applications with low frame rates tend to have large spatial discontinuities which severely detract from the user experience, compounding that problem with visual adaptation discontinuities seems to make the experience quite unpleasant.

It is therefore advisable to establish some maximum estimate of elapsed time when applying a time-dependent model. If the frame rate drops below some threshold, the estimated elapsed time will then remain fixed. This has the effect of slowing down the adaptation with respect to wall-clock time, but the effect appears much less upsetting to the user. Other heuristics such as boosting the value of τ if the frame rate gets too low might be fruitful as well, but the key is to avoid severe adaptation discontinuities.

7. Conclusion and Future Work

We have described our implementation of a state-of-the-art tone mapping algorithm using programmable graphics hardware. Our time-dependent version of Reinhard's photographic tone reproduction algorithm achieves high refresh rates. In addition, our ability to add a time-dependent term to the tone mapping algorithm makes it quite suitable for interactive simulation.

There are a number of directions for future work. First, we would like to implement our algorithm as a non-invasive add-on for unmodified OpenGL applications using the Chromium framework[14]. It should be straightforward to allow an unmodified application to render directly into a floating-point texture, and we can then apply our algorithm

whenever the application swaps buffers. This would allow anyone to experiment with the use of high dynamic range textures in an interactive application. We could thus allow existing games like Id Software's *Quake III: Arena* to use special floating-point texture maps to draw very bright regions such as explosions or the sun. In addition, our OpenGL replacement could enable a vertex program to compute a standard OpenGL lighting model without clamping, allowing an ordinary OpenGL program to benefit from HDR rendering without requiring selective texture replacement. Because the tone mapping algorithms require no high-level information from the application, any application could immediately benefit from a real-time tone adaptation model.

Second, it would be useful to design an extended API so that HDR-aware applications could control the tone mapping subsystem. For example, the API could allow users to control tone mapping parameters such as the "key level" a and threshold ε in Reinhard's algorithm. Applications might also desire to damp the HDR compression level near the extremes of the dynamic range to let more or less of the image wash out. More generally, we would like to explore the extent to which rapid interactive change affects the perceptual utility of precise tone mapping. Providing a feedback mechanism for the application to control performance by specifying how aggressively to preserve detail would be would be necessary to conduct such experiments.

Acknowledgments

We would like to thank Mark Segal and James Percy at ATI and David Kirk, Pat Brown, Matt Papakipos, Nick Triantos, and Matt Pharr at NVIDIA for providing early cards and excellent driver support; Mark Harris, Aaron Lefohn, and Ian Buck for productive discussions on general-purpose GPU computation; and the anonymous reviewers for their thorough and constructive comments.

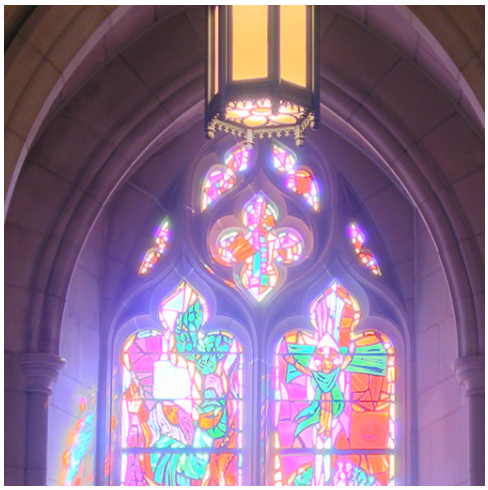
References

1. Ansel Adams. *The Print*. Little, Brown and Company, 1983.
2. ATI. Radeon 9700 Pro, 2002. `http://mirror.ati.com/products/pc/radeon9700pro/`.
3. Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3), July 2003.
4. Ken Chiu, Michael Herf, Peter Shirley, S. Swamy, Changyaw Wang, and Kurt Zimmerman. Spatially nonuniform scaling functions for high contrast images. In *Proceedings of Graphics Interface 1993*, pages 245–253, May 1993.
5. Jonathan Cohen, Chris Tchou, Tim Hawkens, and Paul Debevec. Real-time high-dynamic range texture mapping. In *Proceedings of Eurographics Workshop on Rendering*, pages 313–320, June 2001.
6. Michael F. Cohen, Donald P. Greenberg, David S. Immel, and Philip J. Brock. An progressive refinement approach to fast radiosity image generation. In *Proceedings of SIGGRAPH 1988*, pages 75–84, August 1988.
7. Paul Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. In *Proceedings of SIGGRAPH 1997*, pages 369–378, August 1997.
8. Kate Devlin, Alan Chalmers, Alexander Wilkie, and Werner Purgathofer. STAR: Tone reproduction and physically based spectral rendering. In *Proceedings of Eurographics 2002*, pages 101–123, September 2002.
9. Frédo Durand and Julie Dorsey. Interactive tone mapping. In *Eurographics Workshop on Rendering*, pages 219–230, June 2000.
10. Raanan Fattal, Dani Lischinski, and Michael Werman. Gradient domain high dynamic range compression. *ACM Transactions on Graphics*, 21(3):249–256, July 2002.
11. Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, July 2003.
12. Mark Harris. GPGPU: General-purpose computation using graphics hardware, 2003. `http://www.cs.unc.edu/~harrism/gpgpu`.
13. Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 109–118, August 2002.
14. Greg Humphreys, Mike Houston, Ren Ng, Sean Ahern, Randall Frank, Peter Kirchner, and James T. Klosowski. Chromium: A stream processing framework for interactive graphics on clusters of workstations. *ACM Transactions on Graphics*, 21(3):693–702, July 2002.
15. Kenneth E. Hoff III, John Keyser, Ming C. Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 1999*, pages 277–286, August 1999.
16. Daniel J. Jobson, Zia ur Rahman, and Glenn A. Woodell. A multiscale retinex for bridging the gap between color images and the human observation of scenes.

IEEE Transactions on Image Processing, 6(7):965–976, July 1997.

17. Alexander Keller. Instant radiosity. In *Proceedings of SIGGRAPH 1997*, pages 49–56, August 1997.
18. Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3), July 2003.
19. E. Scott Larsen and David K. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of IEEE Supercomputing 2001*, November 2001.
20. Greg Ward Larson, Holly Rushmeier, and Chistine Piatko. A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):291–306, October-December 1997.
21. Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics. In *Proceedings of SIGGRAPH 1990*, pages 327–335, July 1990.
22. William R. Mark, Steve Glanville, and Kurt Akeley. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, August 2003.
23. Kenneth Moreland and Edward Angel. The FFT on a GPU. In *Proceedings of Graphics Hardware 2003*, July 2003.
24. NVIDIA. GeForceFX, 2003. `http://www.nvidia.com/view.asp?PAGE=fx_desktop`.
25. Sumanta N. Pattanaik, James A. Ferwerda, Mark D. Fairchild, and Donald P. Greenberg. A multiscale model of adaptation and spatial vision for realistic image display. In *Proceedings of SIGGRAPH 1998*, pages 287–298, July 1998.
26. Tim Purcell, Ian Buck, William Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.
27. Erik Reinhard, Michael Stark, Peter Shirley, and Jim Ferwerda. Photographic tone reproduction for digital images. *ACM Transactions on Graphics*, 21(3):267–276, July 2002.
28. Martin Rumpf and Robert Strzodka. Nonlinear diffusion in graphics hardware. In *Proceedings of Eurographics/IEEE TCVG Symposium on Visualization*, pages 75–84, May 2001.
29. Annette Scheel, Marc Stamminger, and Hans-Peter Seidel. Tone reproduction for interactive walkthroughs. *Computer Graphics Forum*, 19(3):301–312, August 2000.
30. Christophe Schlick. Quantization techniques for visualization of high dynamic range pictures. In *Proceedings of Eurographics Workshop on Rendering*, pages 7–20, June 1994.
31. Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. 1999. `ftp://ftp.sgi.com/opengl/doc/opengl1.2/`.
32. Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pages 306–317, November 2002.
33. Jack Tumblin, Jessica K. Hodgins, and Brian K. Guenter. Two methods for display of high contrast images. *ACM Transactions on Graphics*, 18(1):56–94, January 1999.
34. Jack Tumblin and Holly E. Rushmeier. Tone reproduction for realistic images. *IEEE Computer Graphics and Applications*, 13(6):42–48, November 1993.
35. Jack Tumblin and Greg Turk. LCIS: A boundary hierarchy for detail-preserving contrast reduction. In *Proceedings of SIGGRAPH 1999*, pages 83–90, August 1999.
36. Greg Ward. *A Contrast-based Scalefactor for Luminance Display. In Graphics Gems IV*, chapter VII.2, pages 415–421. Academic Press, 1994.

4 : 1



53 : 1

56 : 1

140 : 1

621 : 1

905 : 1

Plate 1: *A series of* 512×512 *HDR images that have been tone mapped on the GPU using Equation 5. Underneath each image is the compression ratio achieved by our algorithm using two adaptation zones. All images were generated at nearly 30 Hz.*