Efficient Histogram Generation Using Scattering on GPUs

Thorsten Scheuermann* Justin Hensley†

Graphics Product Group
Advanced Micro Devices, Inc.

Abstract

We present an efficient algorithm to compute image histograms entirely on the GPU. Unlike previous implementations that use a gather approach, we take advantage of scattering data through the vertex shader and of high-precision blending available on modern GPUs. This results in fewer operations executed per pixel and speeds up the computation.

Our approach allows us to create histograms with arbitrary numbers of buckets in a single rendering pass, and avoids the need for any communication from the GPU back to the CPU: The histogram stays in GPU memory and is immediately available for further processing.

We discuss solutions to dealing with the challenges of implementing our algorithm on GPUs that have limited computational and storage precision. Finally, we provide examples of the kinds of graphics algorithms that benefit from the high performance of our histogram generation approach.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading I.4.3 [Image Processing and Computer Vision]: Enhancement—Filtering I.4.10 [Image Processing and Computer Vision]: Image Representation—Statistical

Keywords: histogram, GPGPU, real-time rendering, image processing, tone mapping

1 Introduction and Related Work

The histogram of a gray-scale image consists of a discrete array of bins, each representing a certain gray-level range and storing the number of pixels in the image whose gray-level falls into that range. In other words, it defines a discrete function that maps a gray-level range to the frequency of occurrence in the image [Gonzalez and Woods 1992], and provides a measure of the statistical distribution of its contributing pixels. For the remainder of this paper we will denote N as the number of input pixels, and B as number of histogram bins.

Modern real-time graphics applications — such as games — commonly perform image processing on the rendered image in order to enhance the final output. Examples include color correction [Mitchell et al. 2006], image processing and enhancement [Hargreaves 2004], and high-dynamic range (HDR) tone mapping operators [Mitchell et al. 2006; Carucci 2006; Sousa 2005].

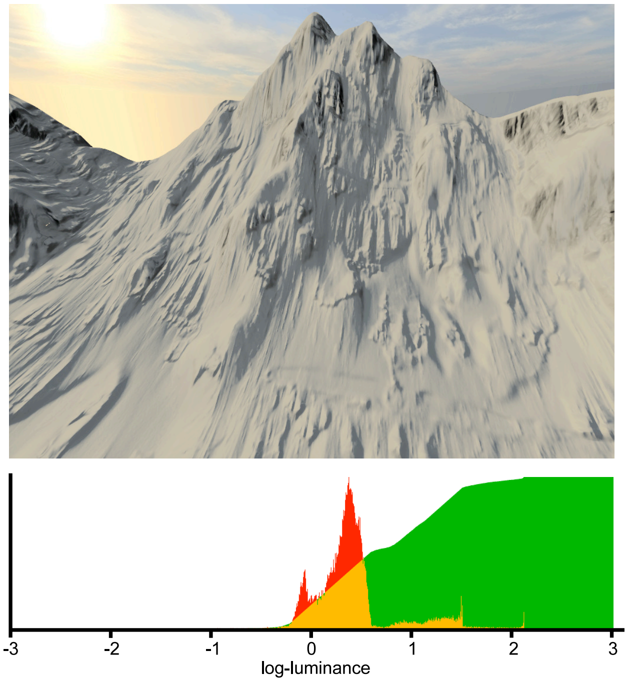

Figure 1: Efficient creation of histograms on a GPU enables new GPU-based post-processing algorithms. This screenshot shows our prototype implementation of Ward's Histogram Adjustment tone mapping operator applied to a real-time 3D scene rendered with high dynamic range. The graph below shows the histogram of the HDR frame buffer in red, and the tone mapping function derived from it in green (both in log-luminance space with base 10).

Histograms are an important building block in image processing algorithms such as tone mapping operators [Larson et al. 1997], which makes them applicable and useful for post-processing in real-time graphics. We explain some of these examples in more detail in section 3.

Carucci [2006] describes using histogram information for tone mapping and for enhancing the contrast in daylight and night scenes in the game Black & White 2. He proposes generating the histogram on the CPU after reading back a downsampled version of the framebuffer from the GPU.

The basic algorithm for creating a histogram on a general-purpose CPU is very simple:

```
set array $bins$ to 0
for each input pixel $p$
    compute bin index $i$ for $p$
    increment $bins[i]$
```

Green [2005] describes an approach to GPU histogram generation that works by rendering one quad per bucket covering the input image. The pixel shader for the quad fetches an input pixel and kills the fragment being processed if the luminance is outside the current bucket's brightness range. Occlusion queries issued for each

*e-mail: thorsten.scheuermann@amd.com
†e-mail: justin.hensley@amd.com

quad are used to read the count of pixels that pass the luminance threshold test back to the CPU.

Reading GPU occlusion query results back to the CPU introduces a synchronization point between the processors, which can hinder parallelism and can cause stalls in the GPU pipeline. In order to avoid stalling, occlusion query results should be read only several frames after they were issued, which increases the latency between rendering a frame and obtaining its histogram.

The SOURCE game engine [Valve Software 2006] uses a variant of Green's algorithm: Histogram generation is amortized over several frames by only updating a single bucket per rendered frame [Mitchell et al. 2006]. Because each bucket's pixel count is gathered from a different frame, the histogram representation available to the CPU generally never accurately represents any single rendered frame.

Fluck et al. [2006] generate histograms on the GPU by dividing the input image into tiles whose size is determined by the number of desired histogram bins. To compute a local histogram for each tile, every pixel in a tile must fetch all other pixels that belong to its tile and perform a count. A texture reduce operation is used to add the local histograms of neighboring tiles and yields the global histogram for the entire image after $\mathrm{O}(\log N)$ rendering passes.

Thanks to highly optimized texture fetching hardware, GPUs have traditionally been very performant for gather operations, and many GPGPU algorithms are tuned to exploit this strength. Both GPU-based histogram generation algorithms mentioned above use a gather approach, whereas the simple CPU algorithm uses a scatter approach: The output location in the *bins* array is dependent on each input pixel.

Buck [2005] points out that data scattering can be implemented on GPUs by rendering point primitives, with the scatter location computed in a vertex shader. Our algorithm uses this point primitive-based scatter approach by taking advantage of the render to vertex buffer (R2VB) [Scheuermann 2006] or vertex texture fetch (VTF) [Gerasimov et al. 2004] feature available in GPUs that support it to feed rendered image data back to the top of the graphics pipeline. This allows us to efficiently build a complete histogram of every rendered frame using an arbitrary number of buckets in a single pass. Peercy et al. [2006] describes a low-level GPU programming interface that provides access to scattering support in the pixel shader on some GPUs.

The remainder of this paper is structured as follows: Section 2 provides a detailed explanation of our algorithm. We discuss example applications of our GPU-based histogram in section 3. Section 4 covers experimental results, and section 5 concludes.

2 Histogram Generation

Scatter-based histogram generation consists of two sub-tasks: Bin selection for each input pixel and accumulation of bin contents. We represent the histogram buckets as texels in a one-dimensional renderable texture. Our algorithm renders one point primitive for each input pixel. We compute the bin index in the vertex shader and convert it to an output location that maps into our 1D bin texture. The fragment that is rasterized into our desired bin location in the histogram render target is accumulated by configuring the hardware blend units to add the incoming fragment to the contents of the render target. After scattering and accumulating all points in this manner, the output render target will contain the desired histogram.

2.1 Histogram Bin Selection

Performing bin selection in a vertex shader requires access to the input image pixels in this stage of the graphics pipeline. To accomplish this, we can take advantage of either vertex texture fetches [Gerasimov et al. 2004] — which are available on NVidia GeForce 6 and 7 series GPUs — or of rendering the input image pixels into a vertex buffer [Scheuermann 2006] — which is supported on ATI GPUs starting with the Radeon 9500 and up.

In the vertex texture fetch case, we store a texture coordinate addressing the corresponding pixel in the input image for each point, so that we can explicitly fetch the pixel color in the vertex shader. In the render to vertex buffer case, each input pixel will be directly available to the vertex shader as a vertex attribute.

The vertex shader used to process the point primitives converts the input color to a suitable grayscale representation (for example linear or logarithmic luminance), maps it to the representative histogram bin, and converts the bin index to the appropriate output coordinate. If the input pixel's luminance is outside the range represented by the histogram bins, we clamp it so that it maps to either the minimum or maximum bin.

2.2 Precision Considerations for Bin Content Accumulation

Many GPUs pose limitations on which render target formats support hardware blending. Lack of blendable high-precision render targets can limit the accuracy of the resulting histogram: If we use an 8-bit render target to store the histogram bins, a bin is saturated after accumulating only 256 points into it. Saturated bins distort the true luminance distribution of the input image, and a histogram with saturated bins violates the equality $\sum_i bins[i] = N$. The severity of saturation depends on the storage precision of the histogram bins and the statistical distribution of the input data. Whether histogram saturation poses a serious problem depends on the context of the application in which the histogram data is used.

There are several approaches for mitigating the issues associated with limited-precision render target formats: First, by increasing the histogram render target size and thus using more bins, each bin represents a smaller luminance range. This can help spread points to more bins, reducing the total number of points that map to each bin.

A generalization of this workaround is to generate local histograms and combine them to the final histogram, similar to [Fluck et al. 2006]. If we create L local histograms, the point primitive with index i should be added to the local histogram with index $i \bmod L$. Local histograms can be stored as additional texels rows in the histogram render target. As an optimization each individual color channel in the output render target should be used to store a local histogram. This way, up to four local histograms can be stored per texel row. To get the final histogram all local histograms are added, either in a single rendering pass, or using multiple texture reduction passes. Because these render passes use a gather approach to sum the corresponding bins of all local histograms and do not rely on hardware blending, we have more flexibility in choosing an output render target format with suitably high precision to avoid saturating bins, such as 16-bit fixed point, or 32-bit floating point.

GPUs that support the shader model 3 profile specified in the Direct3D 9 API provide support for blendable 16-bit floating point (fp16) render targets. Even with this format, we can only take advantage of 11 bits of precision available in the mantissa (10 bits + 1 hidden bit). The reason for this are precision issues inherent in floating point representations when trying to add numbers with large differences in magnitude [Goldberg 1991]. If we keep incrementing a histogram bin with fp16 precision by 1, we will start introducing errors as the accumulated value exceeds 2048.

Modern GPUs that implement the Direct3D 10 specification [Blythe 2006] support blending into fixed-point (16 or 32 bit) and 32-bit floating point render targets. Using these data formats for the histogram render target is preferable, because they provide ample precision to avoid overflowing the bins during accumulation.

2.3 Efficiency

As outlined in section 1, Green's histogram generation algorithm performs one render pass per bucket, so its asymptotic complexity is $O(NB)$. That of Fluck et al.'s algorithm is also $O(NB)$, because it requires $B/4$ texture fetches for each input pixel. Assuming we have enough precision in the histogram render target to be able to ignore the workarounds described in section 2.2, the asymptotic complexity of our histogram generation algorithm is $O(N)$. The high degree of parallelism in GPUs diminishes the significance of the factor N — which represents the number of input pixels — for the overall running time.

Our algorithm has the disadvantage that the majority of its operations happen in the vertex shading units, which generally have a lower level of parallelism than the pixel processing pipeline on non-unified graphics architectures. On a unified GPU architecture such as the GPU in Microsoft's XBox 360 game console — where generic shader ALU resources can be allocated to either pixel or vertex processing depending on the relative load — this disadvantage disappears: Since the pixel shader is trivial — it just returns a constant — most shading resources can be allocated to executing the vertex shader that scatters points into the histogram render target.

Compared to Green's algorithm, our approach avoids the synchronization issues of occlusion query results passed from the GPU back to the CPU. Moreover, if the histogram is needed for subsequent operations on the GPU, it avoids passing the assembled histogram back to the GPU.

3 Applications

To verify the applicability of our algorithm to real-world image processing tasks that could be used in the post-processing stage of a real-time 3D applications, we implemented two types of applications: Histogram equalization and tone mapping operators for high-dynamic range images.

3.1 Histogram Equalization

Histogram equalization is an image processing technique for enhancing the local contrast of an image by remapping the pixel values so that each pixel value is represented with the same frequency [Gonzalez and Woods 1992]. This remapping increases the contrast of luminance ranges that are more frequently represented in the input image. Carucci [2006] mentions histogram equalization in the context of post-processing for a computer game.

A good approximation for a discretized histogram equalization remapping function is $T[i] = \frac{1}{N}\sum_{j=1}^{i} bins[j]$, which is just the numerical integration of the histogram [Gonzalez and Woods 1992].

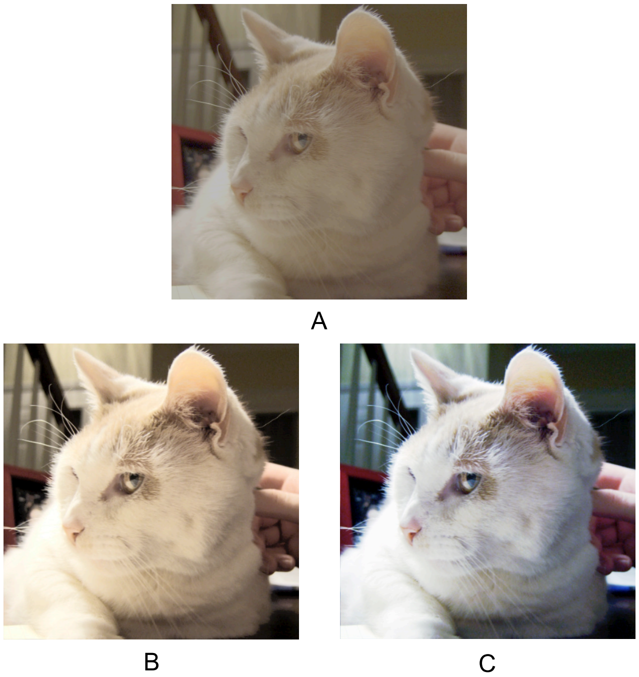

Figure 2: Results of histogram equalization performed on the GPU using our histogram generation and integration algorithm. Image A shows the unprocessed image, which exhibits low contrast. Image B is histogram-equalized using only a luminance histogram. Image C shows the result after performing histogram equalization independently on the R, G, and B color channels.

$T[i]$ can be computed efficiently from the histogram in $O(\log B)$ rendering passes using the GPU-based algorithm for creating summed-area tables outlined in [Hensley et al. 2005]. Just like the histogram, $T[i]$ is stored in a 1D texture. This way, applying the luminance remapping function to the input image reduces to a simple texture lookup based on the input pixel value.

For a color image, histogram equalization can be applied either to the gray-scale luminance, or independently to each color channel. Figure 2 shows an example photograph before and after histogram equalization processing on the GPU using both approaches.

3.2 Tone Mapping

High-dynamic range (HDR) imaging and rendering tries to capture the dynamic range of illumination observed in real-world scenes. In this context, tone mapping operators are used to map the measured or simulated scene to the representation on a display device with limited dynamic range, while maintaining a good visual match between them [Reinhard et al. 2006].

3.2.1 Auto-Exposure Driven by Histogram Percentiles

The histogram equalization remapping table T is the logical equivalent of a cumulative distribution function in the field of statistics. This makes it easy to find the median and – more generally – the nth percentile of the histogram by using a shader to perform a binary search on the texture used to store T. Carucci [2006] explains how these percentiles can be used to drive a simple auto-exposure tone mapping operator. Using our histogram generation algorithm, Carucci's tone mapping approach is feasible to use on current hardware.

Input Size	Algorithm	Histogram Size 64	256	1024
256^2	Scatter (R2VB)	0.5 ms	0.25 ms	0.13 ms
	Scatter (VTF)	1.09 ms	1.09 ms	1.09 ms
	Fluck et al.	0.51 ms	1.06 ms	4.05 ms
	Green	2.34 ms	7.02 ms	25.95 ms
512^2	Scatter (R2VB)	2.55 ms	1.37 ms	0.82 ms
	Scatter (VTF)	4.43 ms	4.38 ms	4.38 ms
	Fluck et al.	1.34 ms	4.36 ms	16.27 ms
	Green	4.34 ms	12.86 ms	44.94 ms
1024^2	Scatter (R2VB)	10.97 ms	5.87 ms	3.52 ms
	Scatter (VTF)	17.71 ms	17.56 ms	17.56 ms
	Fluck et al.	5.11 ms	17.21 ms	64.67 ms
	Green	12.1 ms	43.27 ms	166.72 ms

Table 1: Histogram generation benchmark results comparing different algorithms. (See also figure 3)

3.2.2 Ward Histogram Adjustment

Ward's histogram adjustment tone mapping operator — described in [Larson et al. 1997] — uses histogram information in logarithmic luminance space to derive a tone mapping curve for HDR images. We only consider the simpler histogram adjustment method with a linear ceiling, described in section 4.4 of the original paper. The histogram adjustment method works similarly to naive histogram equalization, but attempts to avoid the problem of expanding and exaggerating contrast in dense regions of the histogram. In order to limit contrast, the authors suggest creating a modified histogram from the input histogram, so that no histogram bin exceeds a constant ceiling. The value of this ceiling is defined by the dynamic range of the input data and that of the display device. The tone mapping curve is then obtained by numerically integrating the modified histogram and used to remap input luminance to display luminance, just like in the histogram equalization case.

Our prototype implementation of this tone mapping operator creates a log-luminance histogram with 1024 bins from a downsampled version of the HDR back buffer. Our next step is to compute the limiting ceiling value and apply it while numerically integrating the histogram. Because the ceiling depends on the sum of all histogram bins, and clamping the histogram bins changes this sum, the ceiling changes as well. Larson et al. [1997] suggest iterating the clamping and integration step until the change in the ceiling value falls below a threshold. For simplicity, our implementation performs a fixed number of iterations. The final step consists of converting the full-resolution back buffer to the displayable range by applying the tone mapping curve as a lookup texture for the input luminance. Our implementation for computing the tone mapping curve has negligible impact on the performance of our test scene, which renders at about 40 to 50 frames per second.

4 Results

We compared the performance of our scatter-based histogram generation algorithm to our own implementations of the algorithms of Fluck et al. and Green. Our test system used a Pentium 4 CPU (3 GHz), 1 GB of RAM, and an ATI Radeon X1900XT GPU. Because this GPU does not support vertex texture fetch, we benchmarked the version of our algorithm using vertex texture fetch on an NVidia Geforce 7800GTX GPU. Therefore the timing results for the vertex texture fetch implementation cannot be compared directly to the other results. Our implementation of Fluck et al.'s algorithm stores the histogram data in 16-bit fixed-point render targets, while our scatter algorithm uses 16-bit floating point render targets to store the histogram.

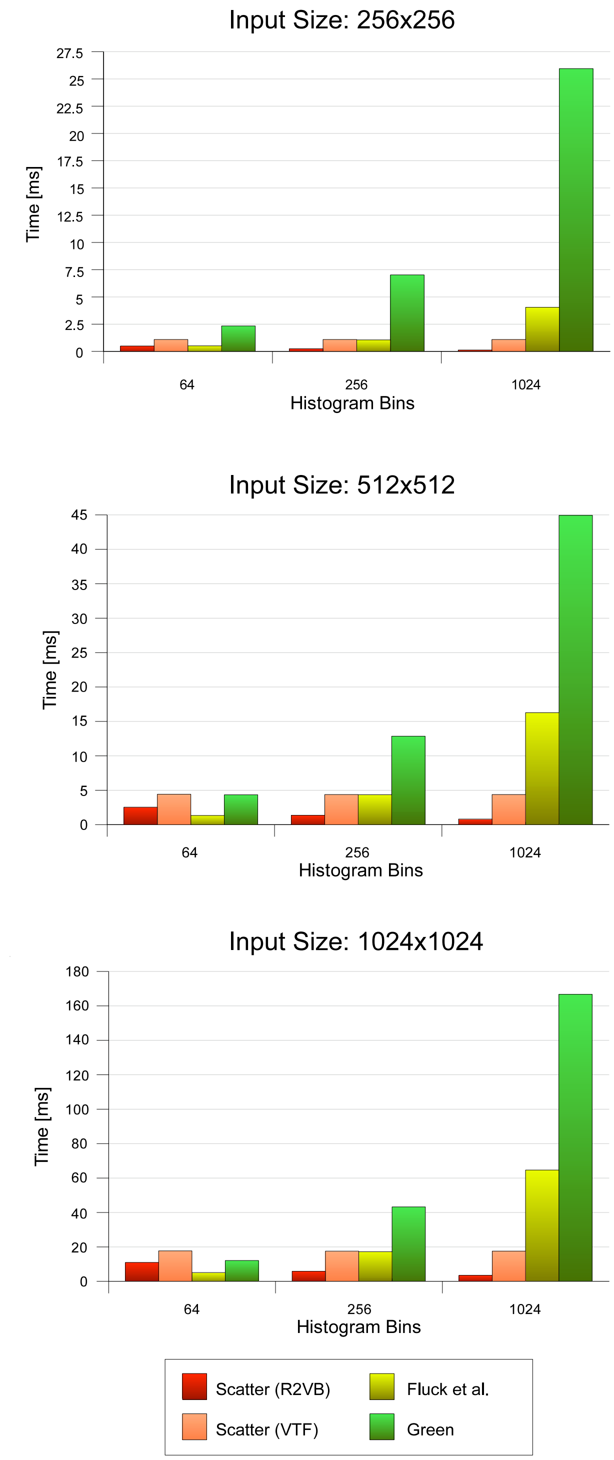

Figure 3: Comparison of histogram generation performance for the tested algorithms. Our algorithm's execution time scaling is better than constant with regards to the histogram size, giving it a clear advantage over the other algorithms, which scale linearly with histogram size.

Our benchmark test creates a luminance histogram of an 8-bit RGBA source render target that contains a static image. Table 1 lists the execution measured times for different input image sizes and histogram sizes. Figure 4 shows the histogram of the image we

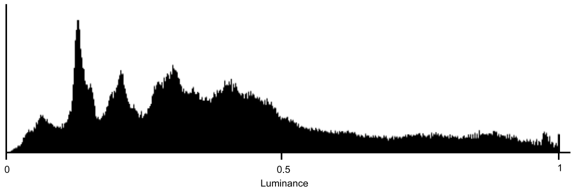

Figure 4: Histogram of the image used for performance tests.

used for our performance measurements.

The results reveal that Fluck's outperforms Green's algorithm by a factor of 2.4 to 6.6. Thanks to the high degree of parallelism in the pixel processing pipeline Fluck's algorithm also outperforms our scatter-based algorithm for histograms with 64 bins, but for larger histogram sizes it is bound by its large number of texture fetches. For a histogram size of 512 buckets, the render to vertex buffer version of our algorithm is roughly 2.9 to 4.2 times faster than Fluck's. For a 1024-bucket histogram, the execution time difference increases to a factor of 18.4 to 31.2.

When implemented using vertex texture fetches, our algorithm exhibits the expected constant running time with regards to the histogram size: Performance is bound by the input image size. However, when using the render to vertex buffer approach to recycle image data to the vertex shader, our algorithm exhibits sub-constant performance with regards to histogram size on the test hardware: The time to generate the histogram decreases as the number of histogram buckets increases. We hypothesize that this is due to the algorithm being limited by the performance of the blending units, which have to serialize a large number of fragments covering the same sample location in a small histogram render target. This bottleneck makes the processing time of our algorithm dependent on the distribution of the input data, which is not the case for the gather-based algorithms.

The blending bottleneck can be mitigated by always creating a large histogram. If a smaller number of histogram buckets is necessary for further processing, the large histogram can be reduced by adding neighboring buckets with a cheap render pass. We measured 0.026 ms as the additional overhead of this reduction step (reducing from 1024 to 64 bins), which makes this two-step approach the fastest implementation in all cases covered by our tests.

In the context of using histograms to aid in post-processing the rendered image in a real-time application, the particularly high performance of our algorithm for small input data sizes is encouraging: Because the histogram of an image and that of a scaled version of the same image is roughly the same — except for a scale factor — it is often preferable to decrease the resolution of the rendered image and then create a histogram from the smaller version instead of building the histogram directly from the full-resolution image. Moreover, for applications that benefit from it, the high performance of our algorithm makes it feasible to run it three times on the input data in order to obtain separate histograms for the R, G, and B color channels.

5 Conclusions and Future Work

In this paper, we have presented a new GPU-based algorithm for generating histograms that outperforms previous algorithms by over an order of magnitude for large histograms. We have outlined solutions to the issue of limited precision in the accumulation stage, which are necessary for a robust implementation of our algorithm on GPUs that don't provide support for hardware blending into a high-precision render target format. Finally, we provided examples of how scene post-processing in real-time 3D applications can benefit from histogram information.

In the future, we would like to explore additional uses for histogram information in real-time graphics. It would also be interesting to investigate if point primitive scattering is applicable to other GPGPU algorithms, and whether it would provide a performance benefit in these cases.

References

BLYTHE, D. 2006. The Direct3D 10 system. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006) 25*, 3, 724–734.

BUCK, I. 2005. GPU computation strategies & tricks. In *SIGGRAPH Course 39. GPGPU: General-Purpose Computation on Graphics Hardware*.

CARUCCI, F., 2006. HDR meets Black & White 2. Game Developer's Conference 2006: D3D Tutorial Day, March.

FLUCK, O., AHARON, S., CREMES, D., AND ROUSSON, M., 2006. GPU histogram computation. Poster at SIGGRAPH 2006.

GERASIMOV, P., FERNANDO, R., AND GREEN, S. 2004. Using vertex textures. Tech. rep., NVidia.

GOLDBERG, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23*, 1, 5–48.

GONZALEZ, R. C., AND WOODS, R. E. 1992. *Digital Image Processing*. Addison Wesley, Boston, MA, USA.

GREEN, S., 2005. Image processing tricks in OpenGL. Game Developer's Conference 2005: OpenGL Tutorial Day, March.

HARGREAVES, S. 2004. Non-photorealistic post-processing filters in MotoGP 2. In *ShaderX 2*, W. Engel, Ed. Wordware, 469–480.

HENSLEY, J., SCHEUERMANN, T., COOMBE, G., SINGH, M., AND LASTRA, A. 2005. Fast summed-area table generation and its applications. In *Computer Graphics Forum (Proceedings of Eurographics 2005)*, Blackwell Publishing, vol. 24, 3.

LARSON, G. W., RUSHMEIER, H., AND PIATKO, C. 1997. A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Transactions on Visualization and Computer Graphics 3*, 4, 291–306.

MITCHELL, J., MCTAGGART, G., AND GREEN, C. 2006. Shading in Valve's Source engine. In *SIGGRAPH Course 26: Advanced Real-Time Rendering in 3D Graphics and Games*. August.

PEERCY, M., SEGAL, M., AND GERSTMANN, D. 2006. A performance-oriented data parallel virtual machine for GPUs. In *ACM SIGGRAPH Sketches*, ACM Press.

REINHARD, E., WARD, G., PATTANAIK, S., AND DEBEVEC, P. 2006. *High Dynamic Range Imaging*. Morgan Kaufmann, San Francisco, CA.

SCHEUERMANN, T. 2006. Render to vertex buffer with D3D9. In *SIGGRAPH 2006 Course 3: GPU Shading and Rendering*. August.

SOUSA, T. 2005. Adaptive glare. In *ShaderX 3*, W. Engel, Ed. Charles River Media, Hingham, MA, USA.

VALVE SOFTWARE, 2006. Source engine SDK documentation. http://developer.valvesoftware.com/wiki/SDK_Docs.