Exploiting Spatial Redundancy with Adaptive Pyramidal Rendering

Orion Sky Lawlor
U. Alaska Fairbanks
2380 Steese Hwy
99712, Fairbanks, Alaska
lawlor@alaska.edu

Jon D. Genetti
U. Alaska Fairbanks
513 Ambler Lane
99775, Fairbanks, Alaska
jdgenetti@alaska.edu

ABSTRACT

Just as image data compression is designed to save space while preserving the essence of an image, we present an adaptive pyramidal rendering scheme designed to save rendering time while maintaining acceptable image quality. Our coarse-to-fine scheme predicts when and where it is safe to take less than one sample per output pixel, and exploits spatial redundancy to predict pixel colors in the resulting gaps, both of which can be performed at framerate in realtime on a modern GPU. As a lossy compression method, we present experimental data on the rendering time versus image quality tradeoff for several example renderers.

Keywords

GPU Raytracer, Data Compression, Time Compression, Sampling Theory

1. INTRODUCTION

Several current commercial trends are converging to require higher speed rendering in computer graphics. The transition to battery-powered mobile devices with modest onboard graphics processing ability, such as phones and tablets, means rendering efficiency is increasingly important. Graphics display pixel densities are also increasing, with 1080p mobile displays,[1] and laptop displays exceeding 5 megapixels.[2] Stereoscopic 3D output devices such as stereo headgear are becoming affordable, but require high resolution imagery to be generated at minimum latency, such as 1080p at 120Hz. Finally, there are well known rendering techniques such as path tracing illumination that are more general and higher quality than the current state of the art, but are only beginning to be affordable in real time [Otte13].

Modern lossy image data compression techniques such as JPEG can compress a still image by 50-fold, to less than one bit per color pixel, yet still reconstruct a high quality image nearly indistinguishable from the original. The reason this is possible is that most images have a high degree of redundancy, such as similar nearby pixel colors. Video compression schemes such as MPEG can produce even higher compression ratios by taking advantage of temporal redundancy, such as similarities between adjacent video frames.

This paper presents a general scheme called "rendering time compression" which aims to speed up rendering time, by exploiting redundancy in the rendered output pixels. The rendering time saved could be spent on higher resolution output, more detailed geometry, or more sophisticated rendering techniques such as global illumination; or used to deliver higher quality content on lower end devices such as cell phones. When a cloud-based renderer's mobile output device is only accessible via a slow network link, extensive data compression is already required, so rendering time compression could increase overall efficiency and make new graphics applications affordable.

Prior Work

Spatial coherence in rendering is well known and has long been exploited. The seminal raytracing work [Whitt80] used a spatial subdivision approach to perform per-pixel antialiasing, and Mitchell [Mitc87] adaptively placed randomized samples for antialiasing, but neither demonstrated interpolation of finished pixels. An incremental raycasting volume renderer [Levo90] cast a sparse grid of rays across the dense grid of pixels, starting at a sampling rate of one ray for every four pixels, then adaptively refined regions where colors differed by more than a fixed epsilon value, but did not demonstrate more than one level of refinement.

Compressive Rendering [Sen11] is a wavelet technique that can reconstruct a high-quality image from a sparse of image samples. This has delivered good results for real scenes, but the sparse linear algebra required for image reconstruction takes several minutes per frame, making the technique too slow for interactive rendering.

Temporal coherence has been exploited previously, such as an image caching raytracer [Deme98],

[1] For example, the Samsung Galaxy S4 has a 5 inch display at 1920x1080 resolution.
[2] For example, the Apple MacBook Pro has a 15 inch Retina display at 2880x1800 resolution.

 ISBN 978-80-86943-71-8

although in the pre-GPU era antialiasing was expensive, so aliasing limited the number of times a frame could be reused. Other ray tracers exploit both temporal and spatial coherence, such as radiance interpolants [Bala99] which can provide guaranteed radiance error bounds while making a per-pixel choice between interpolation and ray tracing. One difficulty with temporal coherence schemes is handling non-static geometry, such as character animation or simulated physics.

2. COMPRESSION THEORY

Mathematically, we can treat the true rendered image I as a function,

$$\mathrm{I} : \mathbb{R}^2 \rightarrow \mathbb{R}^n$$

The domain of the image function is the pixel coordinates (*x*,*y*), where $0 \leq x < w$ and $0 \leq y < h$ for an image with $w \times h$ pixels. The image also depends on the camera model, lighting, and scene geometry and shaders, but we will elide those here. The range of the image function is an *n* dimensional output "color" space, most commonly *n*=3 for conventional RGB color, but often *n*=4 to include an alpha channel or for CMYK print output, and in general *n* could be quite large for a sophisticated renderer that includes polarimetry and multispectral sampling, which we will nevertheless refer to as "color" here.

Our goal in any rendering process is to computationally reconstruct the image function's shape throughout its domain, creating a rendering R(x,y) with the same domain and range as the true image, and ideally with the same colors. Hence we seek to minimize the **reconstructed image error** *E*, as scaled by a perceptual bias function *B*.

$$\mathrm{E}(\mathrm{I},\mathrm{R}) \equiv \int_{y=0}^{h} \int_{x=0}^{w} \mathrm{B}(\mathrm{I}(x,y),\mathrm{R}(x,y))\, dx\, dy$$

A simple perceptual bias function *B* might depend only on the *p*-norm difference between the true and rendered colors; for our experimental work we use $\mathrm{B} = |\mathrm{I} - \mathrm{R}|$, the L_1-norm color difference or sum of absolute color channel differences. A common choice is mean-squared-error (MSE), $\mathrm{B} = \mathrm{I} - \mathrm{R}^{\,2}/n$ for n pixels, although this ignores small differences. A more sophisticated function might also weight differences in image gradients, such as $\mathrm{B} = \alpha|\mathrm{I} - \mathrm{R}| + \beta|\nabla\mathrm{I} - \nabla\mathrm{R}|$ (for scalar weights α and β), or amplify differences in perceptually salient areas, such as the "structural similarity" metric. Mitchell [Mitc87] weighted differences in green more heavily, to match the human eye's color-dependent contrast sensitivity.

Sampling: Measure the Image

Our primary tool to construct a time-compressed rendering is point samples *I(x,y)* provided by ray tracing. Unlike conventional rasterizers, which naturally perform pixel writes for each piece of geometry in a raster scan order, ray tracers can sample the image at arbitrary locations in an arbitrary order, which gives rendering compression schemes much more freedom to efficiently skip sampling in smooth areas. Commercial GPU ray intersection libraries such as NVIDIA's OptiX [Park10] can trace over 100 million rays per second for general polygon meshes of approximately 100K triangles; the best research renderers [Bikk12] can approach a billion rays per second.

A conventional Whitted-style recursive raytracer [Whit80] produces a deterministic output color at a given screen location, which is convenient for rendering because only a single sample is needed per pixel. A distribution ray tracer [Cook86], by contrast, jitters ray samples in space and time to avoid aliasing and produce a correct average result, but each individual ray is merely a random estimate of this true average. Path tracing is a style of distribution ray tracing used to compute global illumination effects, and since the single traced path can be implemented with iteration instead of recursion, it avoids the incoherent memory accesses of a stack, making it more amenable to efficient GPU implementation and today nearly affordable in real time [Bikk13]. When rays vary like this, the true image represents an expected value, and our rendering may need to take several samples and estimate a sample mean.

Selection: Is Sampling Needed?

The selection phase determines if existing image samples adequately capture the appearance variation in the scene, or if additional samples are needed. One approach is to analytically bound the variation in the image, such as via radiance interpolants [Bala99], but the price for this predictability is restrictions on geometry, lighting, and shaders. In the more general case, the image is unknown, making selection a problem of spatial statistics.

If selection is based only on existing samples, a small isolated object such as a star that is missed by the initial sampling is unlikely to ever be recovered. If this is not tolerable, it would be possible to insert known information into the selection process to guarantee small features are sampled, such as the camera projection coordinates of small objects, or an estimate of specular highlight locations from an environment map approximating the scene lighting.

Selection need not depend only on the image samples so far—we could add a selection bias to render more detail in places we expect the viewer to examine closely, such as faces, text labels, or moving objects. Rendering selection bias based on eye tracking could

 ISBN 978-80-86943-71-8

deliver increased resolution to the user's fovea while minimizing rendering effort in peripheral vision.

Interpolation: Image Reconstruction

Given a sparse set of samples, we need to reconstruct a full dense grid of image pixels for final output.

For the general case of reconstructing a dense grid from arbitrary sparse samples, the geostatistics technique of kriging would be an ideal tool, except that it is too slow. Typical implementations scale at best quadratically with the number of sample points, and even recent CUDA GPU kriging [Srin10] is at best dozens of times too slow for realtime work.

A faster technique for sparse sample reconstruction might be to build a finite element triangulation using the image samples as vertices, then evaluate finite element shape functions to interpolate a continuous version of the image. High quality 2D Delaunay triangulations have historically been used for this, including edge constraints to match color discontinuities along object edges [Pigh97]. 2D Delaunay triangulation has recently been extended to the GPU [Qi13], with the latest algorithms and hardware running at framerate for approximately 1 million points, although this fully occupies a high-end desktop GPU, leaving little time for raytracing the underlying sample points.

We present an efficient pyramidal rendering scheme in the next section. A more sophisticated interpolation scheme might also include temporal information, such as using finished full-resolution pixels from previously rendered frames, similar to MPEG's motion vector based frame prediction.

Channel Demultiplexing

JPEG image compression separates color from image brightness, and can compress this luminance data using higher spatial resolution than color data, resulting in better compression than compressing all channels uniformly. Similarly, it can be advantageous to decouple various rendering channels for better overall performance.

The simplest channels to demultiplex are texture and illumination. Because texture changes rapidly, but illumination generally changes smoothly, much better results can be obtained by interpolating illumination across pixels, while sampling texture per pixel [Pigh97, Bala99]. Similarly, multi-bounce global illumination is expensive to compute via path tracing [Bikk12] but often varies predictably, while direct illumination is inexpensive to compute yet can vary rapidly due to sharp shadows. We can compute these two forms of illumination in separate passes, and use a higher render time compression rate on the expensive global illumination step, similar to the recent work on interpolating the global illumination light field [Leht12].

As another example, in aurora rendering, the foreground aurora is smooth and hence interpolates well but is computationally expensive to sample, while the background stars are computationally cheap but interpolate poorly. Hence it is better to separately render the aurora channel, using its strong spatial redundancy to speed up the process, and then composite in the background stars as a final pass.

PYRAMIDAL RENDERING

As an example of time compressed rendering, we implemented a simple adaptive pyramidal renderer. This renderer begins by sampling at each center of a coarse grid of "macropixels", which are blocks of 4x4 full-sized pixels—this is 1/16 the data (6.25%) of a full resolution image.

To create each finer image in the pyramid, for each finer grid pixel we first use an **error metric** to measure the spatial color variation in the coarser grid to determine if a new sample is required. If so, we **sample** the image at the fine grid pixel center; if not, we **interpolate** the color at that pixel from the coarser grid. This sample-or-interpolate process can be repeated to generate finer and finer grids until the desired resolution is reached—this could even exceed 1:1 pixel resolution, for a scene-adaptive version of multisample antialiasing. In the next section, we numerically evaluate various error metrics, and determine the best is a simple low-order polynomial fit to nearby colors, compared with a small stencil of neighboring coarse pixels. Currently, our error metrics only use pixel colors from the coarse grid, but could be extended to exploit temporal redundancy from the previous frame, or other information such as scene geometry.

Because each grid level is a regular 2D image, and grids are generated one level at a time, this technique matches even decade-old GPU hardware—it can be implemented using a simple OpenGL (or even WebGL) shader shown below using rendering passes at ¼ resolution, then ½, and finally full resolution. This technique also automatically generates a few coarser mipmap levels of the onscreen image, which could be useful for bloom effects, or postprocessed depth of field blurring.

```
// GLSL fragment shader for pyramidal rendering
varying vec2 pix; // fine target texture coordinates
uniform sampler2D coarser; // coarser grid texture
uniform bool coarsest; // true during first pass
uniform float threshold; // color error allowed
void main(void) {
  if(coarsest || errorMetric(coarser, pix)>threshold)
    gl_FragColor = sampleScene(pix);
  else // interpolate from coarser grid
    gl_FragColor = texture2D(coarser,pix);
}
```

 ISBN 978-80-86943-71-8

As shown in Figure 1, sampling the pixel centers results in the coarse and fine grids being offset, which means each fine pixel is the same distance from the nearest coarse pixel, but also means coarse pixel samples cannot be reused directly. In the worst case, where the sample selection scheme chooses to render every pixel at every level, we would render 1+¼ +1/16+1/64+... = 1⅓ times more pixels than a naive full resolution direct rendering. An alternative might be to render the corners of pixels, so ¼ of the fine pixels are coincident with a coarse pixel and can be copied directly, but we find this makes interpolation more difficult to perform well, while sampling pixel centers produces smooth[3] interpolated curves even using trivial bilinear interpolation. Bilinear interpolation is also very GPU friendly and is monotonicity-preserving, meaning it does not suffer from ringing artifacts near sharp edges.

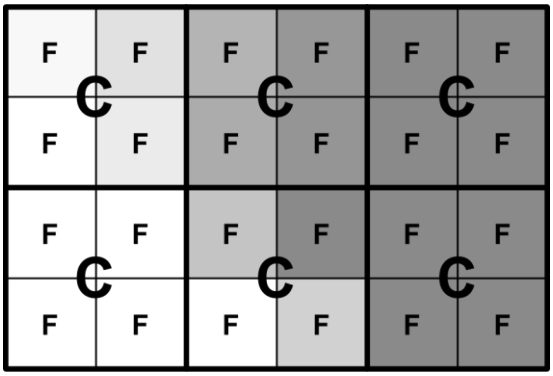


Figure 1: Interpolating a coarse (C) 3x2 pixel image to a finer (F) 6x4 pixel image.

3. PERFORMANCE RESULTS

We measured the performance of our pyramidal rendering algorithm for two interactive renderers and a variety of still images.

Pyramid Level Sensitivity

Starting with a coarse image pyramid level, such as 16x16 macroblocks, requires fewer samples at a given error threshold, but reconstructed image accuracy is poor because the coarse levels tends to skip over small features, which are then interpolated away. Starting with a finer grid, such as 4x4 pixel blocks, more reliably captures these features. Even if the selection threshold is adjusted so the coarse grid results in the same number of samples, a finer initial grid spreads the samples more evenly, resulting in lower reconstruction error. However, a finer initial grid requires more initial samples, leaving fewer remaining to allocate to the detected high-detail regions—see the numerical results averaged across our benchmark image library in Table 1.

[3] Repeated bilinear interpolation approaches gaussian impulse response, per the central limit theorem.

Begin pyramid at ½ resolution (fine)	2.18%
Begin pyramid at ¼ resolution	2.31%
Begin pyramid at ⅛ resolution	2.60%
Begin pyramid at 1/16 resolution	2.84%

Table 1: At a fixed rendering rate of ⅓ sample per pixel, average reconstruction error rates improve with finer starting grid level, even though the coarser starting grids require fewer initial samples.

Interpolation Error Metric

When creating increasingly finer pyramid levels from a coarser level, our rendering algorithm needs to decide between sampling the underlying scene or interpolating the pixel. Generally, we want to interpolate in smooth featureless regions, and sample where things are changing, which we must distinguish according to an **error metric**. We found changing the error metric used during image expansion had a surprisingly weak effect—generally, an area that will interpolate well is smooth enough to have a low error under nearly any reasonable metric. Table 2 summarizes average reconstruction errors for our test scenes under a variety of error metrics, using our usual ⅓ sample per pixel rate, and beginning the image pyramid expansion at ¼ resolution.

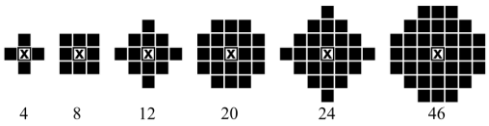
We empirically determined the best error metric is a low-order polynomial fit to the nearby colors, compared against a compact stencil of neighboring pixels. That is, we take a sample if

$$\Sigma_{i,j \text{ in stencil}} |\mathrm{R}(x+i, y+j) - \mathrm{P}_{ij}^{t}| \geq \text{error threshold}$$

Here R is the coarse image reconstructed so far, we examine the colors around a coarse pixel R(x,y), fit a 2D polynomial P_{ij}^{t} with t terms, and compare the polynomial to each neighboring pixel R(x+i,y+j). For example, P^1 is a constant color equal to R(x,y), P^3 is a three-term 2D linear polynomial color fit P_{ij}^{3}=A+Bi+Cj, while $\mathrm{P}^9$is a general 2D quadratic.

Table 2 summarizes reconstructed image error rate for various polynomial orders and stencil sizes. First, smaller stencils work better. Expanding the neighbor list beyond a few pixels causes false positives, as the longer reach causes unnecessary sampling far from real features. Using higher order polynomials causes false negatives, as the polynomial infers smooth higher-order curves in irregular areas that should instead be sampled. But the difference between plausible metrics is small, a few tenths of a percent in average color error. Using an implausible metric such as random pixel refinement produces over twice as much error—and only manages that well due to the dense sampling on the initial coarse grid.

 ISBN 978-80-86943-71-8



	4	8	12	20	24	46
P^1	2.31%	2.32%	2.36%	2.48%	2.53%	2.63%
P^3	2.31%	2.26%	2.29%	2.34%	2.35%	2.41%
P^5	*[4]	2.54%	2.25%	2.31%	2.31%	2.34%
P^9	*	*	2.28%	2.30%	2.29%	2.30%

Table 2: Varying the error metric's polynomial order (vertical) and testing stencil (horizontal) during sample selection affects reconstruction accuracy.

If we compare these metrics against the "contrast" metric max-min/(max+min) [Mitc87], we find using the contrast metric on a pixel and its 8 neighbors as a pyramidal error metric for adaptive refinement produces an average color error of 3.06%, worse than any of the other metrics we tested. This is because the contrast metric produces a relative color difference, amplifying absolute differences with low intensity, such as shadows.

Interactive Aurora Renderer

To demonstrate rendering time compression in an interactive renderer, we applied the technique to an aurora borealis GPU volume renderer [Lawl11], which is also in use by other researchers [Ishi11]. For each pixel, this renderer steps along the 3D camera ray through an auroral curtain, accumulating emitted light. A distance field acceleration structure allows the renderer to take much longer steps in the empty volumes between curtains, and it uses a closed form analytic approximation for the ray's integral through an exponential atmosphere, allowing interactive performance on modern GPU hardware. At 720p output resolution on a modest embedded Intel Ivy Bridge Mobile graphics chip, this renderer gives a tolerable 8-15 frames per second (fps).

Adding pyramidal rendering time compression was surprisingly straightforward: a new GLSL shader function was added to perform pixel selection and interpolation, and the old renderer shader main became the sampling function, thus maintaining the original renderer's single-shader design. We modified our pyramidal code to locate its pixels using the built-in onscreen location gl_FragCoord, which allowed the renderer to keep its existing texture coordinates and geometry coordinate system.

[4] * Indicates the polynomial fits the stencil exactly, so we must use a larger stencil to measure fit error.

We used three passes (at ¼, ½, and full resolution), and our error metric was P^3 with an 8-neighbor stencil. Finally, we applied the unpredictable background star field and planet city lights textures only during the final compositing pass, rather than at each pyramid level, so the pyramidal renderer was only working with the smooth aurora and atmosphere layers—the renderer works even with all channels multiplexed, but then stars blink in and out of the rendering.

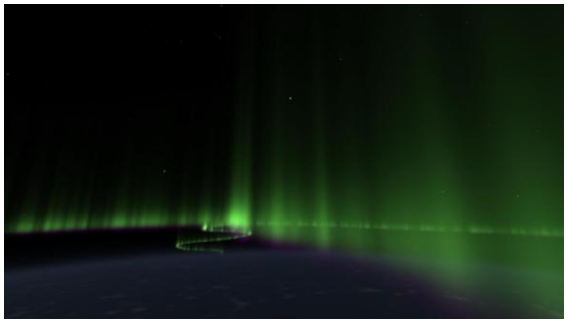


Figure 2: Screenshot from pyramidal aurora renderer, using a 1% average color error threshold.

The result, shown in Figure 2, is pyramidal rendering increases performance about twofold, to 16-28 fps, using a 0.6% average color error threshold which is virtually indistinguishable from the original rendering. We can increase performance about threefold, to 22-37 fps, using a 1% error threshold, although small blurry patches are just perceptible on distant curtains. Using a higher error threshold gives even better framerates, but compression artifacts begin to be more noticable. Framerates for a benchmark camera path are shown in Figure 3.

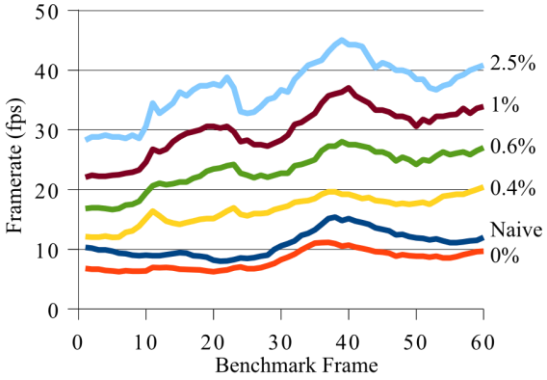


Figure 3: Framerate for pyramidal aurora renderer using different error thresholds, compared to the original naive single sample per pixel renderer.

Pyramidal Fractal Renderer

Since our rendering time compression scheme is content dependent, the most challenging scenes have detail at all scales. Hence for a more difficult test of our pyramidal rendering scheme, we implemented a

 ISBN 978-80-86943-71-8

pyramidal Mandelbrot set renderer on the GPU. To allow for greater zoom factors before numerical issues arise, but still use GPU-friendly single precision floating point, we used the "double single" technique [Bail05] to emulate double precision floating point using single precision operations. Our benchmark is a zoom into the spiral, shown in Figure 4, centered at -0.7451580638+*i* 0.1125749162, scaling from unit field of view to 10^{-6}, iteration count limited to 255. We used four pyramid levels, starting at ⅛ resolution, and got slightly better performance by storing the iteration count in the pyramid pixels, and applying the color table only at the final full resolution pass.

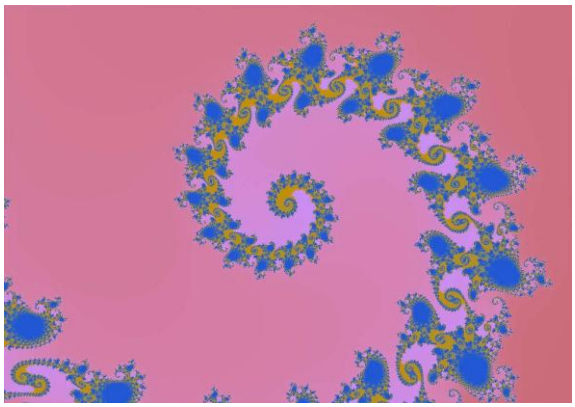


Figure 4: A spiral in the Mandelbrot set, as reconstructed by our pyramidal renderer at a 6% error threshold.

Figure 5 shows the performance of our pyramidal renderer, compared to a naive single sample per pixel renderer, both on an NVIDIA GeForce 650M. Unlike the smooth curves of the aurora, which slowly degrade with increasing error threshold, richly textured fractal surfaces reconstruct nearly independent of the refinement error threshold. This is because there is so much detail near the set that any reasonable error threshold will take further samples there; and there is so little detail in smooth regions even a zero error threshold—sample unless binary identical—will still not refine them. The resulting image only begins to noticeably degrade at an enormous 12% average neighborhood error refinement threshold.

Pyramidal rendering provides a huge fourfold performance improvement early on, while zooming past large flat regions of Mandelbrot set points. These points all require the maximum number of iterations, so each sample is slow to compute, but the colors are identical, so adaptive interpolation saves an enormous amount of work. Approaching the detailed area near the set boundary, nearly the entire image is full of detail, and adaptivity provides negligible speedup, and even a slight slowdown for a zero error threshold. After entering the spiral, only the smooth regions between the spiral arms can be interpolated. Figure 4 shows the area of these smooth regions exceeds 50%, but the iteration trip count is lower in the smooth areas, so the speedup from interpolating through these smooth regions averages only 30%. As the zoom factor increases, the average non-set iteration trip count increases, so adaptive pyramidal rendering provides increasing speedup.

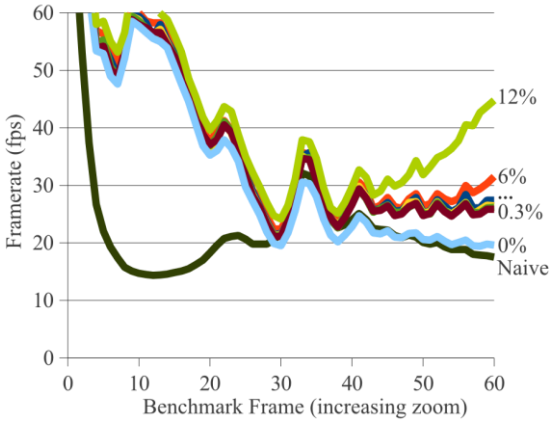


Figure 5: Framerate for pyramidal fractal renderer using different error thresholds, compared to naive renderer.

Still Image Reconstruction

The accuracy of our rendering time compression technique depends strongly on the scene being rendered—a flat blue sky could be reconstructed perfectly using a single sample per frame, while a high contrast unpredictable black and white pattern such as a QR code will require dense sampling. Thus while the renderers described above work well, it would be useful to evaluate this technique for more realistic general scenes.

For an unbiased benchmark set of comparison scenes, we have chosen to reconstruct the raytraced images from the final two years (2005 and 2006) of the Internet Ray Tracing Competition [irtc06]. Since this was a still image competition, we can assume scenes were designed and judged purely for aesthetics, not for renderer performance. We included all the winning and honorable mention images submitted at a resolution over 720 pixels in portrait or landscape aspect ratio, a total of 32 images, and includes the natural, artificial, and artistic scenes shown in Figure 6.

Because the original 3D raytraced scenes are largely unavailable, to test our reconstruction algorithm, when taking a sample instead of tracing a ray as we would for an interactive application, we look up the location in the raytraced image. Since textures and lighting effects are combined, this represents a worst case for a time compression renderer. The image

also acts as the reference, so we can measure the accuracy of our reconstructions. This is clearly not an efficient way to copy a texture, but it allows us to experimentally test different error metrics and stencils, and measure reconstruction accuracy for a variety of scenes.

Figure 6 shows each scene sorted by reconstruction accuracy at a sampling rate of ⅓ sample per pixel. Highly textured and outdoor scenes are near the top, as they are difficult to reconstruct accurately at this rate, but the average color error per scene for these images is still under 5.5%. Smooth or abstract scenes near the bottom reconstruct very easily; the average color error of the bottom two rows is 1%.

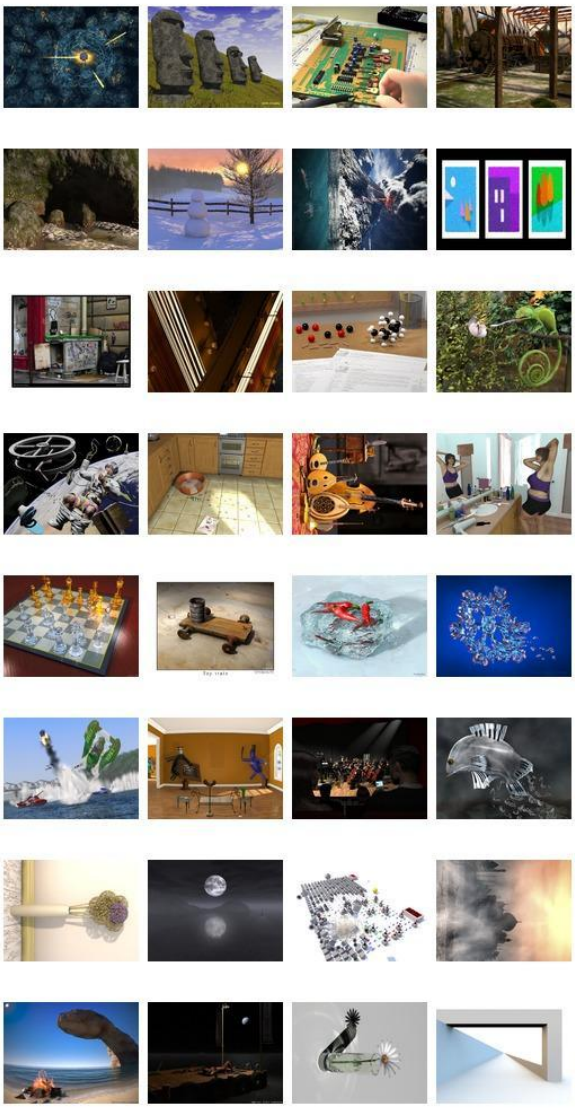


Figure 6: Reconstructed benchmark images from the Internet Ray Tracing Competition, sorted top-to-bottom in raster order by increasing reconstruction accuracy, at a sample rate of ⅓ samples per pixel.

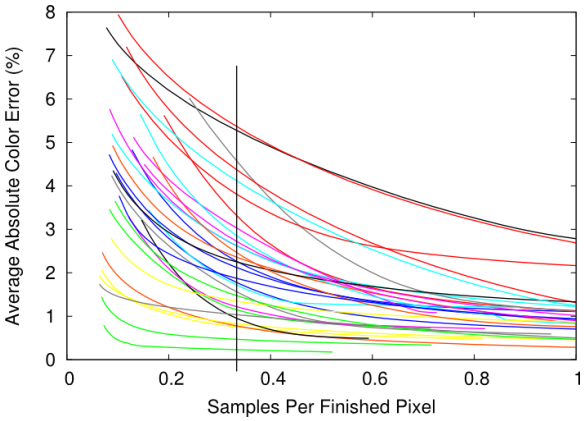


Figure 7: Accuracy of images reconstructed with our rendering time compression algorithm, when using different numbers of samples per pixel. The lines represent different source images. Figure 6 shows a vertical transect at ⅓ samples per pixel, shown here by the vertical line.

Figure 7 varies the number of rays sampled per pixel, and shows the absolute color error in the resulting rendering with our technique, illustrating the quality speed tradeoff. Reconstructed image accuracy increases with more samples, but only asymptotically approaches zero. In particular, note that taking one point sample per pixel does not yield zero error for most images, due to the need to area sample sub-pixel detail near sharp edges.

Figure 8 shows a 1024x768 pixel reconstruction of a 12 megapixel photograph using our technique.

4. CONCLUSIONS

We have presented a scheme called *rendering time compression*, which carefully selects regions of the scene that need more detail, takes raytraced samples there, and interpolates the remaining areas of the image. The net result is to cast less than one ray per pixel, but still derive an accurate approximation of the rendered scene.

One key difficulty in both illumination and antialiasing is estimating area integrals from the point samples of classic ray tracing. Feature film-quality renderers may use thousands of rays per pixel to reduce per-ray noise, taking hours per frame. An old technique known as cone tracing effectively thickens rays into cones, allowing it to evaluate at least box-filtered integrals directly, but the difficulty has always been how to evaluate the cone-geometry integral efficiently for general scenes with occlusion. A technique using a mipmap-friendly voxel geometry approximation has recently been used to compute global illumination on the GPU using cone tracing

 ISBN 978-80-86943-71-8

[Cras11]. A cone tracer could allow much higher rendering time compression rates, by providing smoother estimates of broad regions, and could even be extended to output a brightness variance estimate for sample selection, or directly convolve portions of the scene with a spectral basis function.

With a careful implementation, it is possible our technique could be extended beyond raytracers and other point-sample renderers. For example, in a conventional rasterizer such as DirectX or OpenGL, for a shader-limited program our interpolation step could skip over predictable pixels, reducing the average per-fragment time enough to outweigh the cost to re-traverse the scene geometry at each pyramid level.

Rendering time compression is a promising technique for accelerating a variety of rendering problems. We have shown a simple and GPU-friendly adaptive pyramidal rendering technique that can choose where to interpolate two out of every three pixels, resulting in a several-fold speedup for interactive renderers, while only affecting colors by a few percent. But the much higher image compression rates achieved by existing still and motion image compression algorithms indicate that there is still more unexploited redundancy in rendered imagery. It is possible that even better results could be achieved by more closely following an existing compression scheme, such as designing a sample selection and interpolation scheme that directly estimates the rendered image's discrete cosine transform (DCT) frequency coefficients, for example by using the DCT analog of a sparse Fourier transform, which we look forward to exploring. Other promising areas for future work involve motion estimation to exploiting frame coherence via our knowledge of the motion of the scene geometry and camera, directly outputting compressed MPEG bitstreams from the renderer, and decoupling illumination from texturing for faster global illumination effects.

5. REFERENCES

[Bail05] Bailey, D., Hida, Y., Li, X., Thompson, B., Jeyabalan, K., and Kaiser, A. High-Precision Software: DSFUN90. Lawrence Berkeley Lab, http://crd-legacy.lbl.gov/~dhbailey/mpdist/, 2005.

[Bala99] Bala, K., Dorsey, J., and Teller, S. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics*, 18(3), pp. 213-256, 1999.

[Bikk12] Bikker, J. Ray Tracing for Real-Time Games, Technische Universiteit Delft PhD thesis, 2012.

[Bikk13] Bikker, J., and van Schijndel, J. The Brigade Renderer: A Path Tracer for Real-Time Games, *International Journal of Computer Games Technology*, vol. 2013, Article ID 578269, 14 pages, 2013.

[Cook86] Cook, R.L. Stochastic sampling in computer graphics. *ACM Transactions on Graphics* 5(1), 51-72, 1986.

[Cras11] Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum* 30(7), pp. 1921-1930, 2011.

[Deme98] Demers, J., Yoon, I., Kim, T.Y., and Neumann, U. Accelerating Ray Tracing by Exploiting Frame-to-Frame Coherence. University of Southern California Computer Science Dept, USC-TR-98-668, 1998.

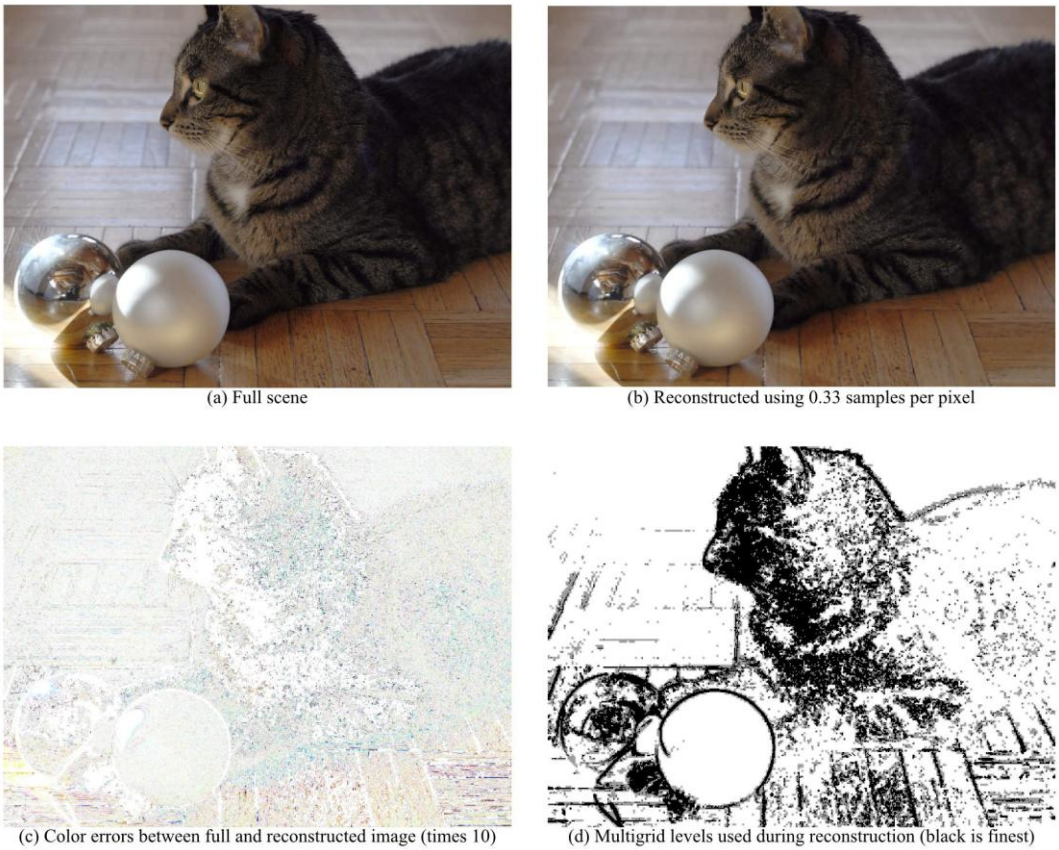


Figure 8: Reconstructing a photograph using ⅓ sample per pixel with our pyramidal technique. The largest reconstruction errors are unpredictable dots in the cat's coloring, and small cracks in the wood floor.

[irtc06] Internet Ray Tracing Competition, http://www.irtc.org/stills/, 2006.

[Ishi11] Ishikawa, T., Yue, Y., Iwasaki, K., Dobashi, Y., and Nishita, T. Modeling of aurora borealis using the observed data. *Proc. 27th Spring Conference on Computer Graphics (SCCG)*, pp. 13-16, 2011.

[Lawl11] Lawlor, O. S., & Genetti, J. Interactive volume rendering aurora on the GPU. *Journal of WSCG*, 2011.

[Leht12] Lehtinen, J., Aila, T., Laine, S., and Durand, F. Reconstructing the indirect light field for global illumination. *ACM Trans. Graph. TOG*, 31(4), p. 51, 2012.

[Levo90] Levoy, M. Volume rendering by adaptive refinement. *Visual Computer*, 6(1), pp. 2-7, 1990.

[Mitc87] Mitchell, D.P., Generating antialiased images at low sampling densities. *ACM SIGGRAPH Computer Graphics*, 1987, vol. 21, pp. 65–72.

[Otte13] Otte, V. Path tracing on GPU. Bachelor's Thesis, Masaryk Univerzity Faculty of Informatics, 2013.

[Park10] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(4), article 66, 2010.

[Pigh97] Pighin, F.P., Lischinski, D., and Salesin, D. Progressive previewing of ray-traced images using image-plane discontinuity meshing. *Rendering Techniques 97*, pp. 115-125, 1997.

[Qi13] Qi, M., Cao, T.T., and Tan, T.S. Computing 2D Constrained Delaunay Triangulation Using the GPU, *IEEE Transactions on Visualization and Computer Graphics*, 19(5), pp.736-748, May 2013.

[Sen11] Sen, P., and Darabi, S. Compressive rendering: A rendering application of compressed sensing. *IEEE Vis. Comput. Graph.*, 17(4), pp. 487–499, 2011.

[Srin10] Srinivasan, B.V., Duraiswami, R., and Murtugudde, R. Efficient kriging for real-time spatio-temporal interpolation. *Conf. on Probability and Statistics in the Atmospheric Sciences*, pp. 228-235, 2010.

[Whit80] Whitted, T. An improved illumination model for shaded display. *Communications of the ACM*, 23(6), pp. 343-349, 1980.