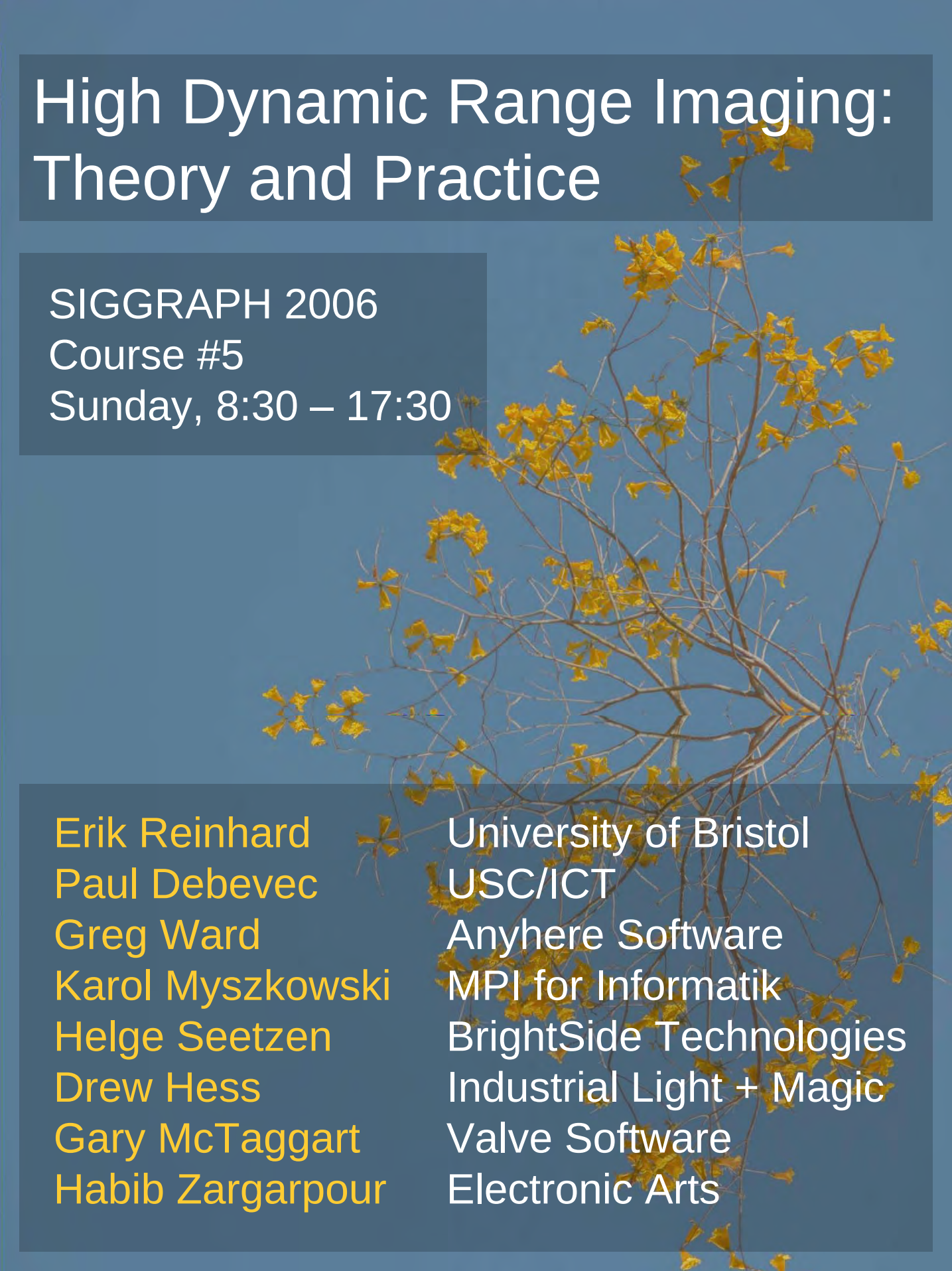
High Dynamic Range Imaging: Theory and Practice

SIGGRAPH 2006
Course #5
Sunday, 8:30 – 17:30



Erik Reinhard	University of Bristol
Paul Debevec	USC/ICT
Greg Ward	Anyhere Software
Karol Myszkowski	MPI for Informatik
Helge Seetzen	BrightSide Technologies
Drew Hess	Industrial Light + Magic
Gary McTaggart	Valve Software
Habib Zargarpour	Electronic Arts

Course Abstract

- Current display devices have a limited range of contrast and colors that can be displayed, which is one of the main reasons that most image and video acquisition, processing and display techniques use no more than eight bits per color channel. This course outlines recent advances in high dynamic range imaging (HDRI) - from capture to display - that lift this restriction thereby enabling images to represent the color gamut and dynamic range of the original scene rather than the limited subspace imposed by current monitor technology. In a hands-on approach, this course teaches how HDR images and video can be captured, the file formats available to store them, and the algorithms required to prepare them for display on low dynamic range display devices. The trade-offs at each step are assessed, allowing attendees to make informed choices about data capture techniques, file formats and tone reproduction operators. The course will cover the latest advances in Image-Based Lighting, in which HDR images can be used to illuminate CG objects and realistically integrate them into real-world scenes. In addition, we show the vast improvements in image fidelity afforded by HDRI through practical applications drawn from the film (ILM) and games industries (EA, Valve). Finally, recent advances in display hardware are demonstrated (BrightSide Technologies).

Prerequisites
Intended Audience

- Participants should be familiar with basic techniques in digital photography, traditional 8-bit image editing, and basic computer graphics modeling and rendering. Familiarity with a specific image-editing package or 3D modeling and rendering package would be helpful.

- This course is intended for students, researchers, and industrial developers in digital photography, computer graphics rendering, real-time photoreal graphics, game design and visual effects production (esp. rendering and compositing).

Summary

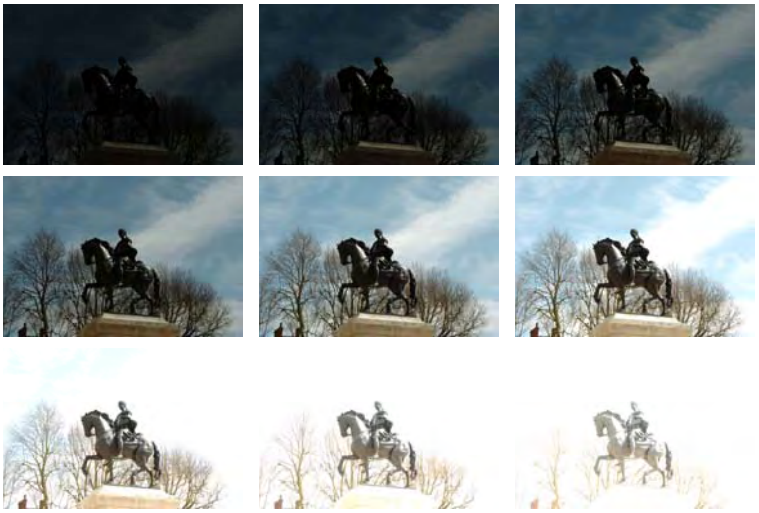
- This course teaches new techniques in capturing, representing, processing, and displaying High Dynamic Range Images and Video that cover the full range of light in the real world, and thereby enable marked improvements in visual fidelity and photorealism. Applications in lighting, compositing, game design and film, as well as advances in display hardware are covered.

Introduction

Images

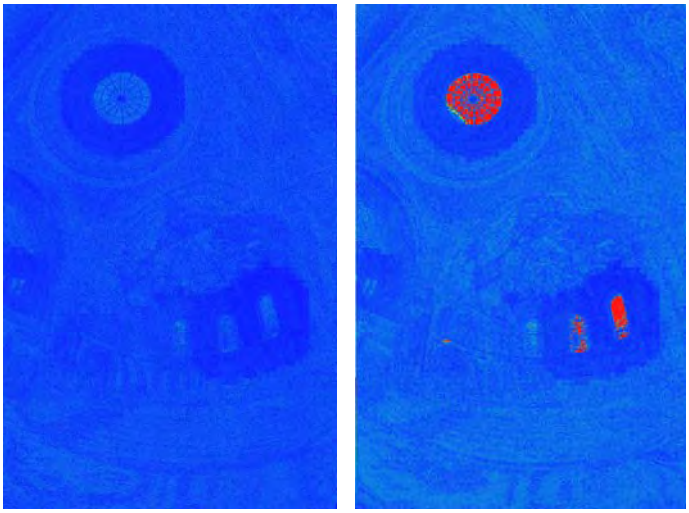
- Traditionally we use one byte per pixel per color channel
- What if we had floating point numbers to represent colored pixels?
- This may seem a small issue, but really isn't…

HDR Image Capture



HDR Image Capture

HDR File Formats



HDR Video

Tone Reproduction

Image-Based Lighting

SIGGRAPH2006

HDR Display Devices

SIGGRAPH2006

Industrial Light and Magic

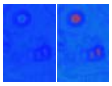
Valve Software

Electronic Arts

Course Schedule (Morning)

– Introduction – Erik Reinhard
(nearly finished)

– Taking HDR Images – Paul Debevec
8:40 – 9:15



– HDR Image Representation – Greg Ward
9:15 – 10:00

– Break
10:00 – 10:15

– HDR Video and Applications – Karol Myszkowski
10:15 – 10:55

Course Schedule (Morning)

- Tone Reproduction Operators – Erik Reinhard
  10:55 – 11:30

- HDR Image-Based Lighting – Paul Debevec
  11:30 – 12:15
- Lunch Break
  12:05 – 12:15

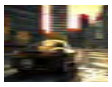
Course Schedule (Afternoon)

- HDR Display Devices – Helge Seetzen
  1:45 – 2:45

- HDR at ILM – Drew Hess
  2:45 – 3:30
- Break
  3:30 – 3:45

- HDR at Valve – Gary McTaggart
  3:45 – 4:30

Course Schedule (Afternoon)



- HDR at EA – Habib Zargarpour
  4:30 – 5:15

- Discussion - All
  5:15 – 5:30

Course Notes

Slides (clickable links):

Taking HDR Images – Paul Debevec

HDR Image Representation – Greg Ward

HDR Video and Applications – Karol Myszkowski

Tone Reproduction – Erik Reinhard

HDR Image-Based Lighting – Paul Debevec

Course Notes

Slides:

- HDR Display Devices – Helge Seetzen
- HDR at Industrial Light + Magic – Drew Hess
- HDR at Valve – Gary McTaggart
- HDR at Electronic Arts – Habib Zargarpour

- Contrast Processing – Rafal Mantiuk
- Lightness Perception – Grzegorz Krawczyk

Course Notes

Papers

Greg Ward and Maryann Simmons, 'Subband Encoding of High Dynamic Range Imagery', ACM Symposium on Applied Perception in Graphics and Visualization, pp 83-90, 2004

Greg Ward and Maryann Simmons, 'JPEG-HDR: A Backwards Compatible, High Dynamic Range Extension to JPEG', Proceedings of the 13th color imaging conference, 2005

Paul Debevec, 'Image-Based Lighting', IEEE Computer Graphics and Applications, 2002

Course Notes

White Papers:

- Technical Introduction to OpenEXR
- OpenEXR File Layout
- Reading and Writing OpenEXR Image Files with the IlmImf Library
- High Dynamic Range Rendering in Valve's Source Engine

Links to External Resources

- HDR mailing list (Greg Ward)

  http://www.radiance-online.org/mailman/listinfo/hdri
- Web site for book on High Dynamic Range Imaging

  http://www.hdrbook.com
- Light Probe Gallery (Paul Debevec)

  http://www.debevec.org/Probes
- Annotated List of HDR File Formats (Greg Ward)

  http://www.anyhere.com/gward/hdrenc/hdr_encodings.html
- OpenEXR website

  http://www.openexr.org

Links to External Resources

- HDRShop (Paul Debevec)

  http://www.hdrshop.com

- Photosphere and related Mac and Unix software (Greg Ward)

  http://www.anyhere.com

Acknowledgments

- Thanks to Rafal Mantiuk for providing slides on his tone reproduction operator
- Thanks to Grzegorz Krawczyk for providing extra slides on his model of lightness perception for tone reproduction
- Thanks to Yuanzhen Li for applying her tone reproduction algorithm to one of our images to enable a comparison

HDRI at SIGGRAPH 2006

- Papers Session: HDR and Systems
  - Tuesday 10:30 – 12:15
- BoF: High Dynamic Range Imagery
  - Boston Convention and Exhibition Center
  - Tuesday 17:00 – 18:00, Room 251

Taking HDR Images

Paul Debevec

Taking HDR Images

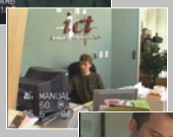
Paul Debevec
University of Southern California
Centers for Creative Technologies

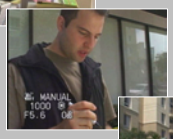
SIGGRAPH 2006 Course #5
High Dynamic Range Imaging: Theory and Applications
Sunday, July 30th, 2006

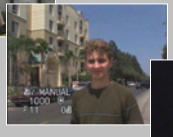
www.debevec.org

Dynamic Range in the Real World


1


1500

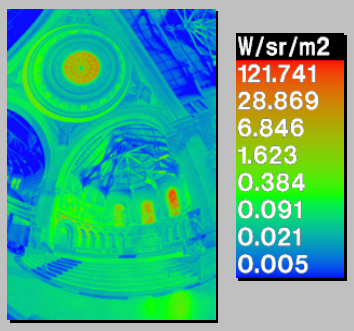

25,000


400,000


2,000,000,000

The real world is high dynamic range.

High-Dynamic Range Photography



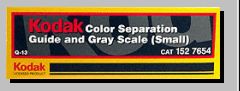
300,000 : 1

Visualization: Greg Ward

Allows one or more images of a scene to be converted to a high dynamic range image that is proportional to radiometric units.

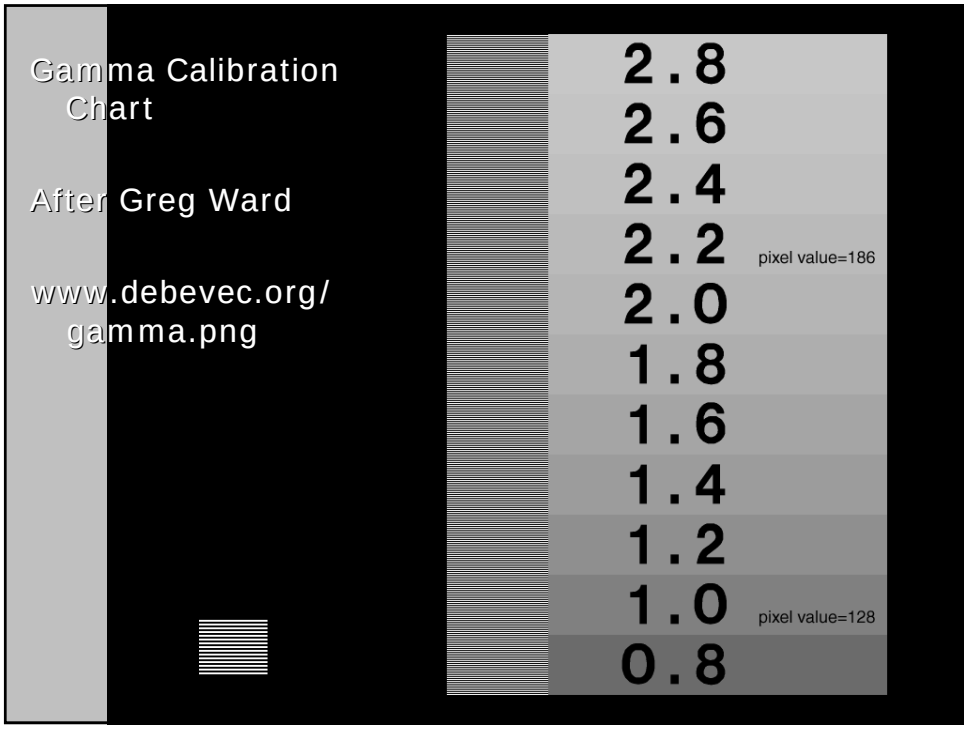
Instead of integer values 0-255, pixel values can range from 0 into the millions, with floating-point precision.

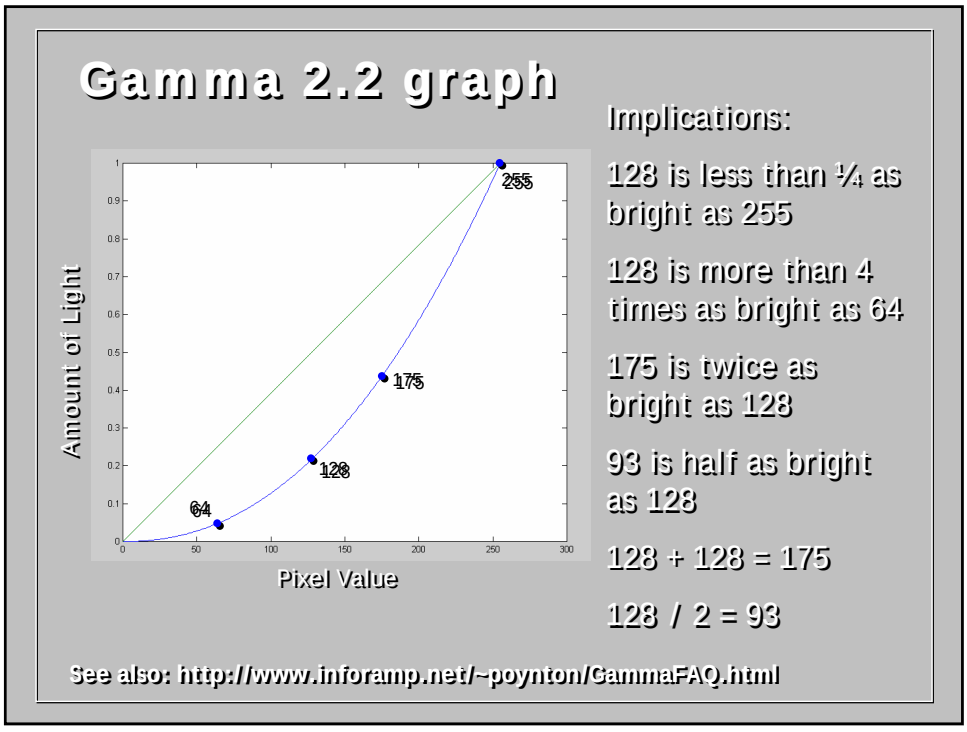
Gray Scale Calibration Charts



Twenty 0.1 density increments = 26% more reflective each step = 1/3 stop
Be careful to minimize specular reflections

<= Gretag MacBeth ColorChecker Chart bottom row reflectivity: 88%, 58%, 35%, 20%, 8%, 3%


Gamma Calibration Chart
After Greg Ward
www.debevec.org/gamma.png
2.8
2.6
2.4
2.2
pixel value=186
2.0
1.8
1.6
1.4
1.2
1.0
pixel value=128
0.8


Gamma 2.2 graph
Amount of Light
Pixel Value
255
175
128
64
Implications:
128 is less than ¼ as bright as 255
128 is more than 4 times as bright as 64
175 is twice as bright as 128
93 is half as bright as 128
128 + 128 = 175
128 / 2 = 93
See also: http://www.inforamp.net/~poynton/GammaFAQ.html

Response Curve Recovery

Debevec and Malik. Recovering High Dynamic Range Radiance Maps from Photographs. SIGGRAPH 97

Image series

Δt = 1/64 sec	Δt = 1/16 sec	Δt = 1/4 sec	Δt = 1 sec	Δt = 4 sec

$$\text{Exposure} = \text{Radiance} \times \Delta t$$

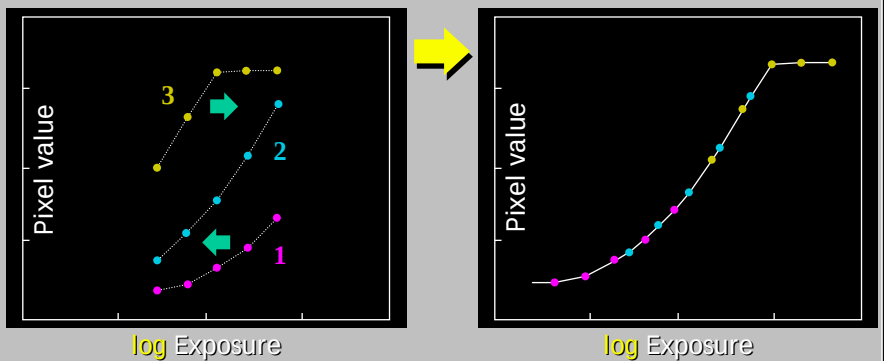
$$\log \text{Exposure} = \log \text{Radiance} + \log \Delta t$$

Recovering the Response Curve

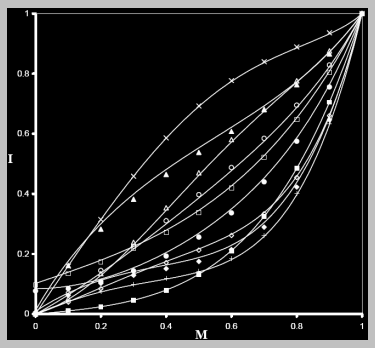
HDRShop www.hdrshop.com

Assuming unit radiance for each pixel

After adjusting radiances to obtain a smooth curve



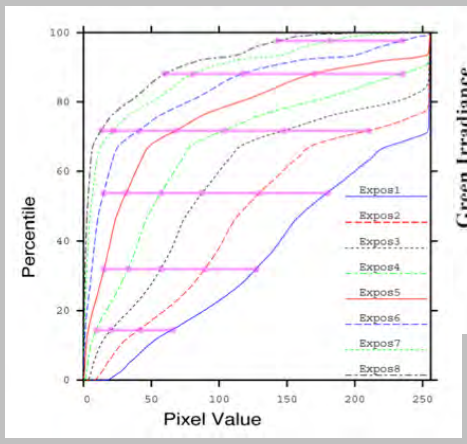
Mitsunaga-Nayar CVPR 99 Response Recovery

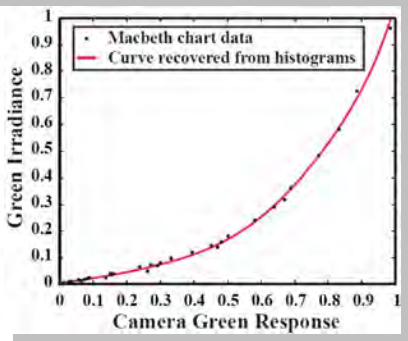


$$I = f(M) = \sum_{n=0}^{N} c_n M^n$$

Assumes polynomial form
Estimates unknown exposure times
Vignetting compensation
http://www.cs.columbia.edu/CAVE/

Grossberg and Nayar – Response recovery from image histograms

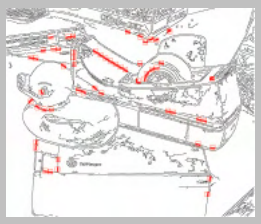
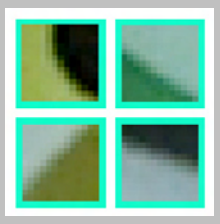




Robust to scene motion

What can be Known about the Radiometric Response from Images? ECCV 2002

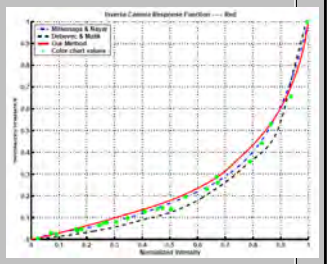
Lin, Gu, Yamazaki, Shum. ***Radiometric Calibration from a Single Image***. CVPR 2004

Assumes that edge regions are blends of two colors, with all intermediate values represented equally (in linear space)

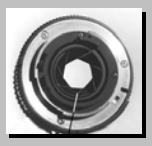
Locates edge regions in image

Estimates response based on intermediate value distributions



Ways to vary exposure

- Shutter Speed (preferred)
- F/stop (aperture, iris)




- Neutral Density (ND) Filters

Shutter Speed

Ranges: Canon D30: 30 to 1/4,000 sec.
Sony VX2000: ¼ to 1/10,000 sec.

Pros:

- Directly varies the exposure
- Usually accurate and repeatable

Issues:

- Digital: Noise in long exposures
- Film: Reciprocity failure at > ~ 5 sec.

Shutter Speed

Note: shutter times usually obey a power series – each "stop" is a factor of 2

¼, 1/8, 1/15, 1/30, 1/60, 1/125, 1/250, 1/500, 1/1000 sec

Usually really is:

¼, 1/8, 1/16, 1/32, 1/64, 1/128, 1/256, 1/512, 1/1024 sec

F/stop (aperture)


Ranges: Canon D30: f/2.8 to f/22

Sony VX2000: f/1.6 to f/11

Standard f-numbers: 2, 2.8, 4, 5.6, 8, 11, 16, 22

Exposure is proportional to the inverse square of the f-number:

f/22 is 1/64 the light of f/2.8 (6 stops)

Pros:

Can use aperture when you run out of shutter speed variation

F/stop (aperture)


Issues:

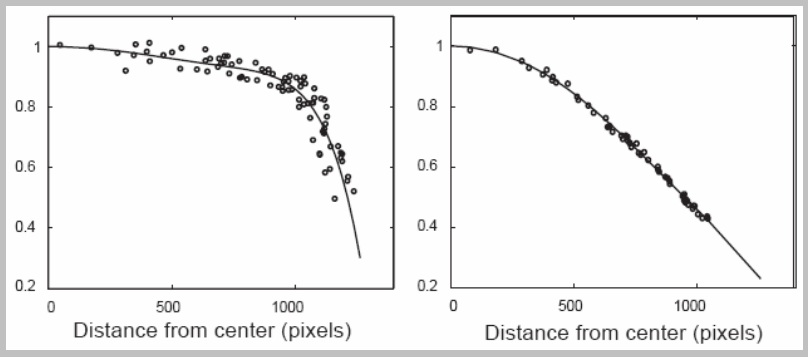
Changes depth of field

Less repeatable

Limited range of exposure variation

=> Not first choice for HDR

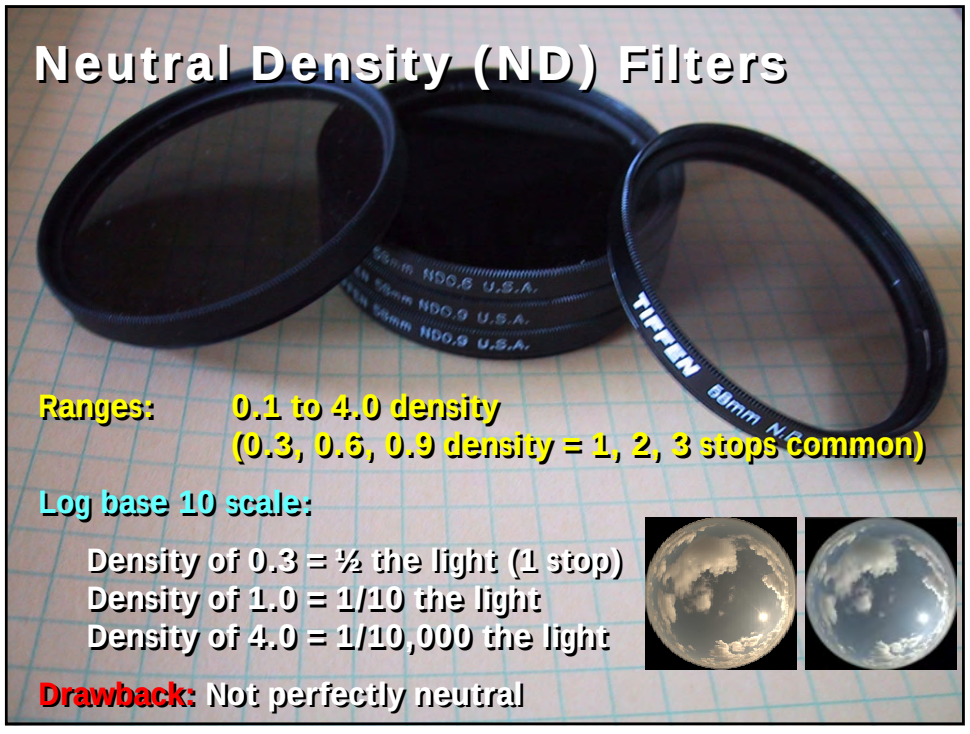
Radial Brightness Falloff (*Vignetting*)



f/16 Sigma 8mm fisheye lens f/8

From Stumpfel et al, *Direct HDR Capture of the Sun and Sky*, Afrigraph 2004

Neutral Density (ND) Filters



Ranges: 0.1 to 4.0 density
(0.3, 0.6, 0.9 density = 1, 2, 3 stops common)

Log base 10 scale:

Density of 0.3 = ½ the light (1 stop)
Density of 1.0 = 1/10 the light
Density of 4.0 = 1/10,000 the light

Drawback: Not perfectly neutral

Gain / ISO / Film Speed

Range: ISO 100 to 1600
0dB to 18dB (3dB = factor of 2)

Problem:
Usually just adds noise to your image

+18dB Gain on Sony VX2000

Storing High Dynamic Range Images

- Portable FloatMap .pfm
- Floating-Point TIFF .tif
- RADIANCE .hdr, .pic
- LogLuv TIFF .tif
- OpenEXR .exr
- ...

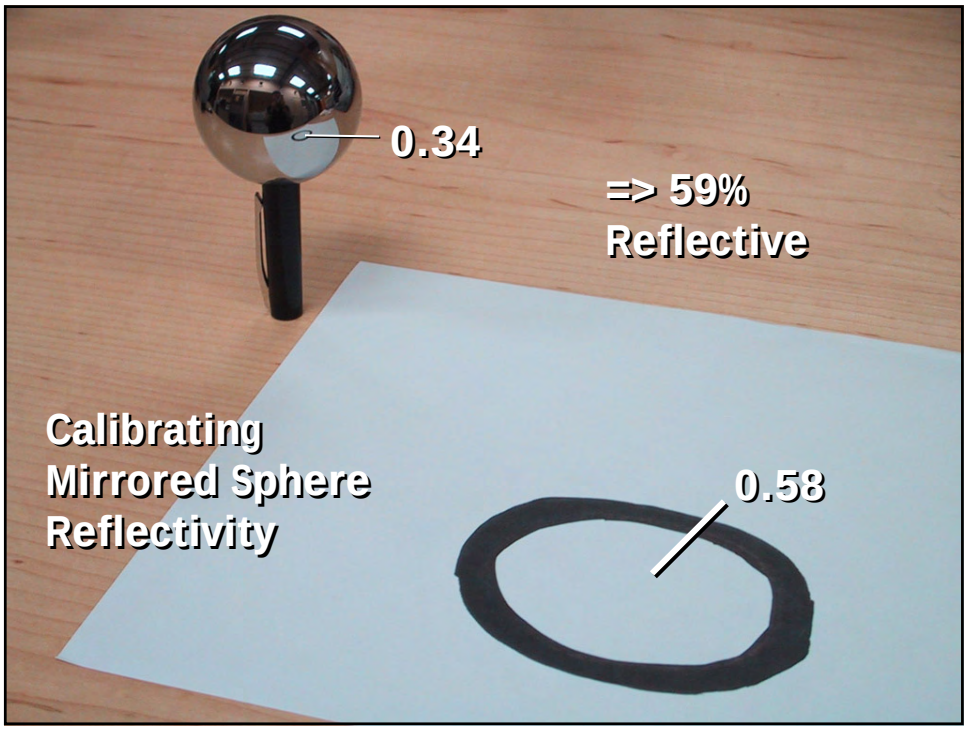
- See Greg's talk coming up...

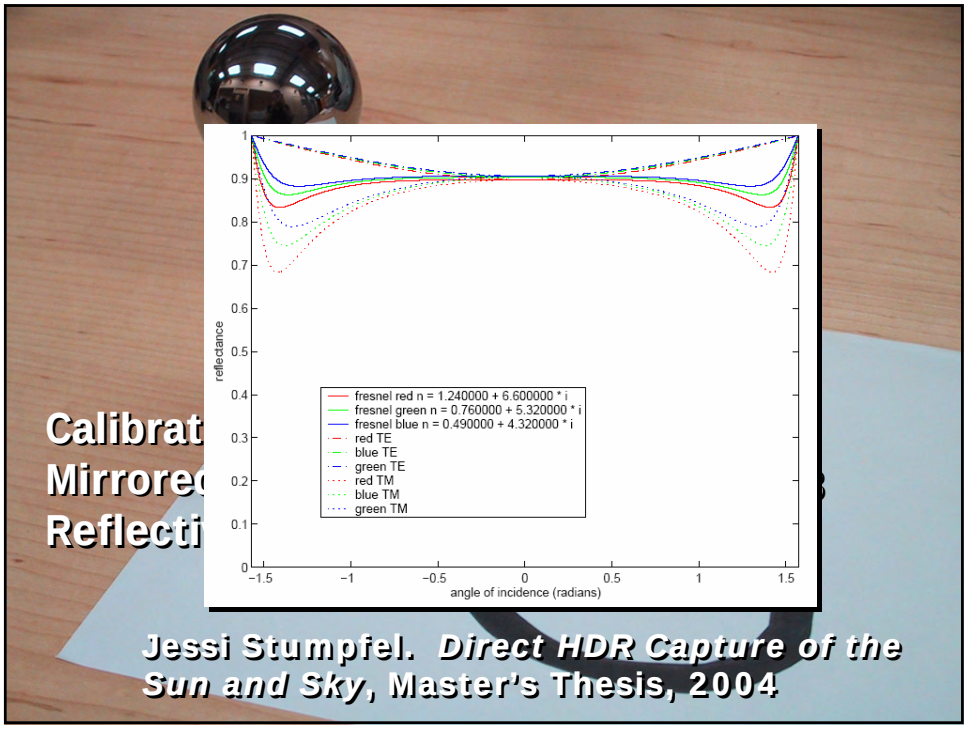
Methods for taking light probes: omnidirectional HDR images

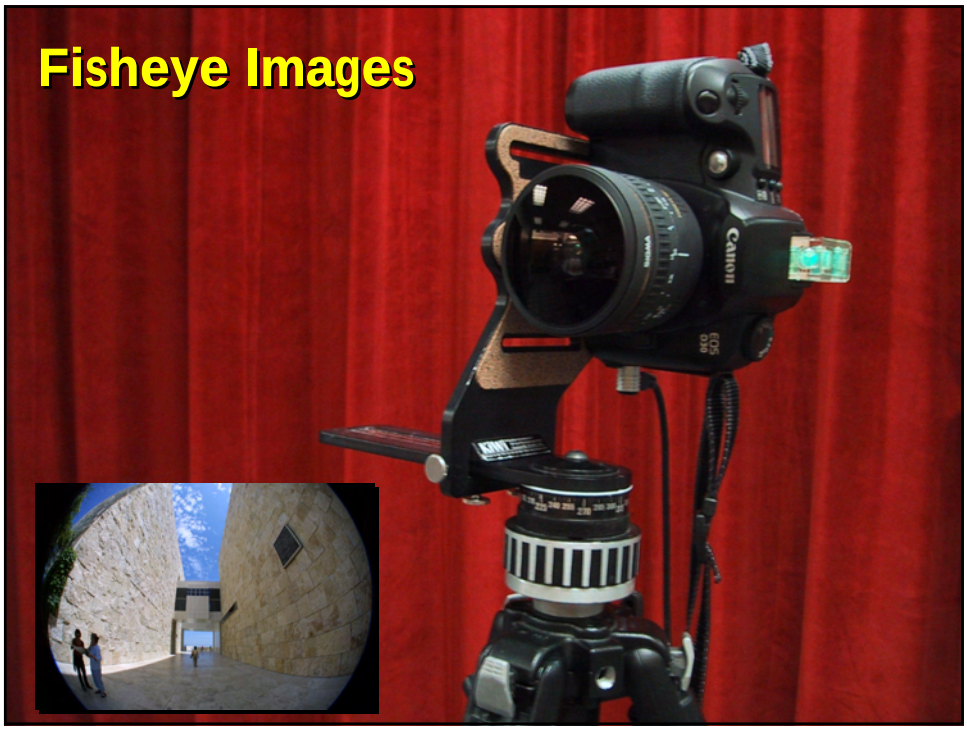
- Mirrored ball + camera
- Fisheye lens images
- Panoramic camera
- Image stitching

Sources of Mirrored Balls

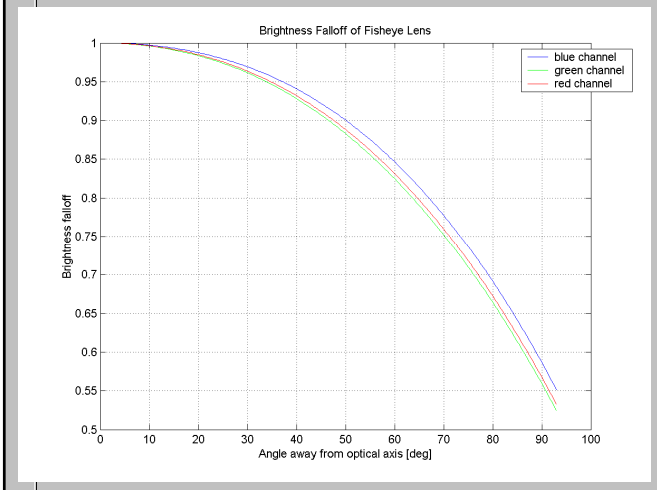
- 2-inch chrome balls ~ $20 ea.
  - McMaster-Carr Supply Company
    www.mcmaster.com
- 6-12 inch large gazing balls
  - Baker's Lawn Ornaments
    www.bakerslawnorn.com
- Hollow Spheres, 2in – 4in
  - Dube Juggling Equipment
    www.dube.com
- FAQ on www.hdrshop.com

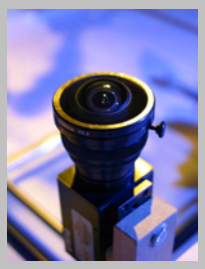



Calibrat
Mirrore
Reflecti
reflectance
angle of incidence (radians)
fresnel red n = 1.240000 + 6.600000 * i
fresnel green n = 0.760000 + 5.320000 * i
fresnel blue n = 0.490000 + 4.320000 * i
red TE
blue TE
green TE
red TM
blue TM
green TM
Jessi Stumpfel. Direct HDR Capture of the Sun and Sky, Master's Thesis, 2004


Fisheye Images

Fisheye Lens Radial Falloff





Scanning Panoramic Cameras (Panoscan, Spheron)

Pros:

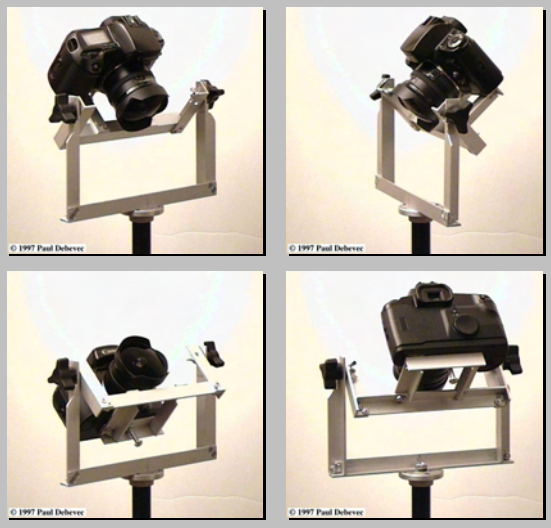
- *very* high res (10K x 7K+)
- Full sphere in one scan – no stitching
- Good dynamic range, some are HDR

Issues:

- More expensive
- Scans take a while

Tiled Photographs

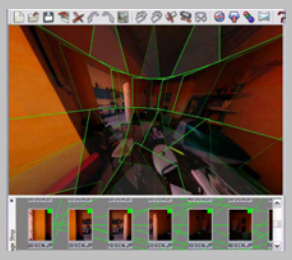
Tiled Photographs – Nodal Acquisition Rig



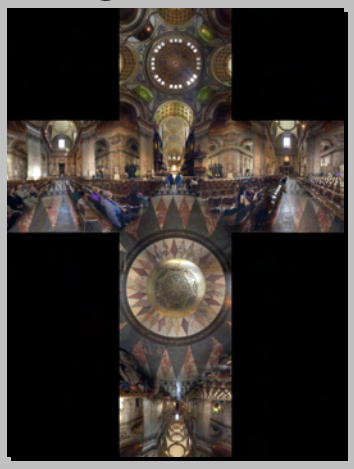
See also www.kaidan.com

Stitching HDRI with Realviz Stitcher

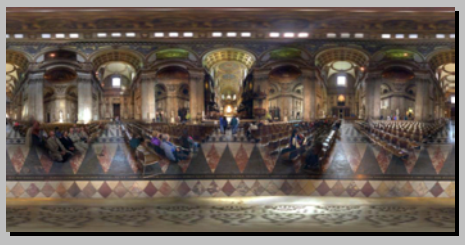
http://www.gregdowning.com/HDRI/stitched/



Types of Omnidirectional Images



Cube Map

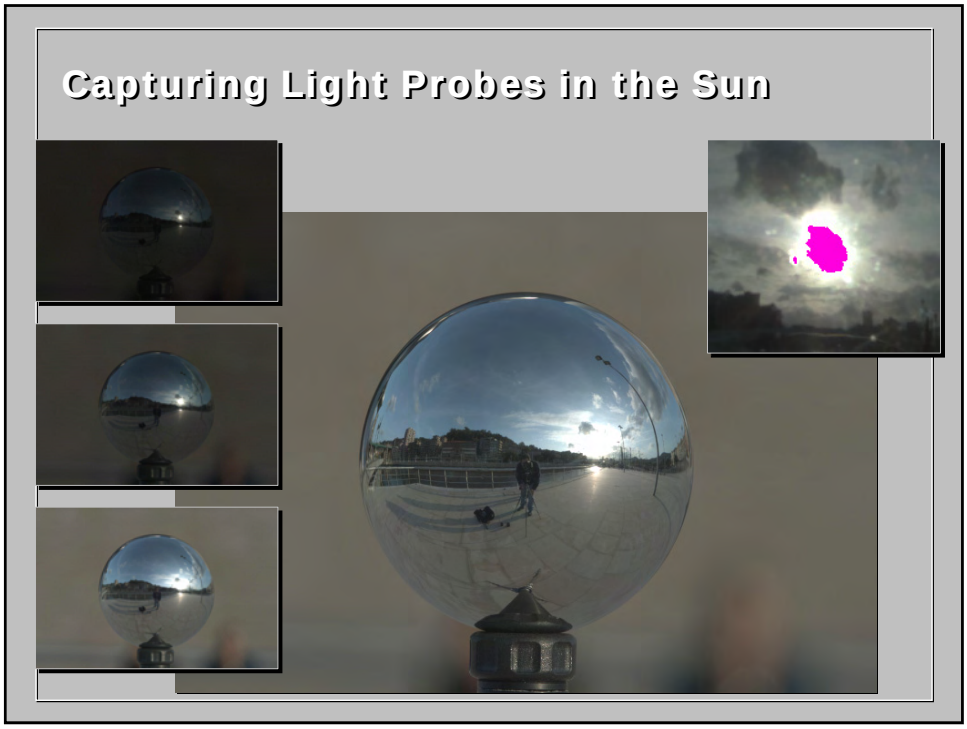


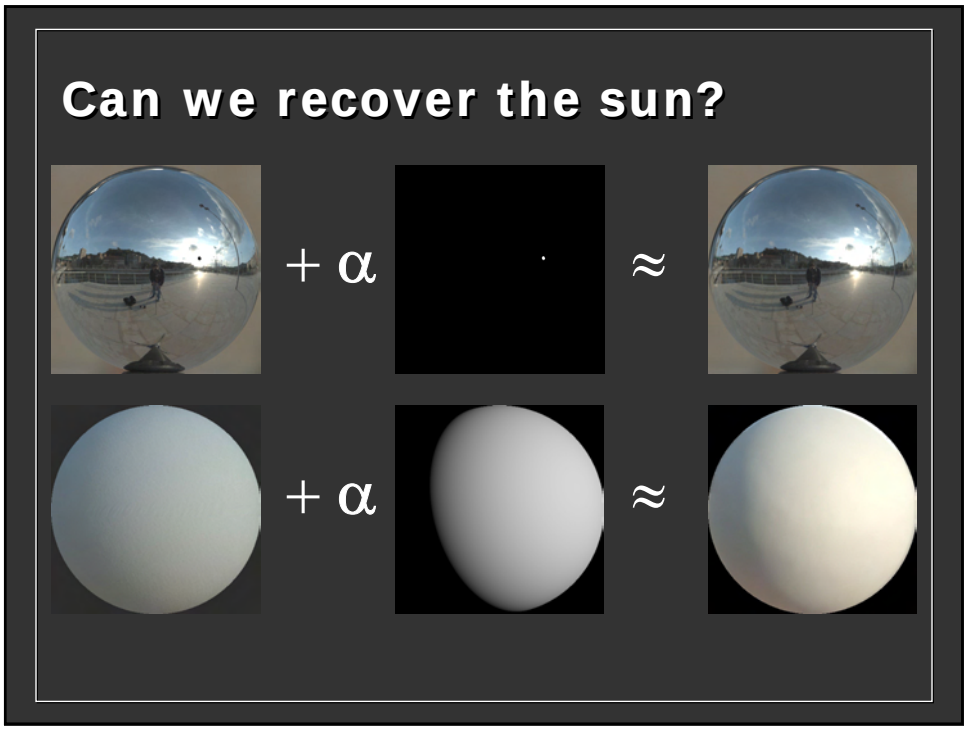
Latitude/Longitude

Types of Omnidirectional Images

Mirrored Ball

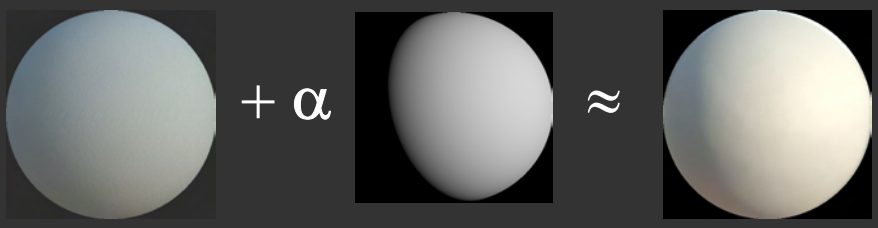
Angular Map


Capturing Light Probes in the Sun

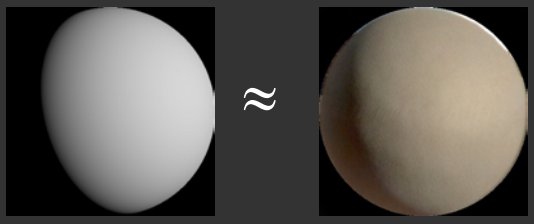

Can we recover the sun?
+ α
≈
+ α
≈

Photograph Diffuse Sphere

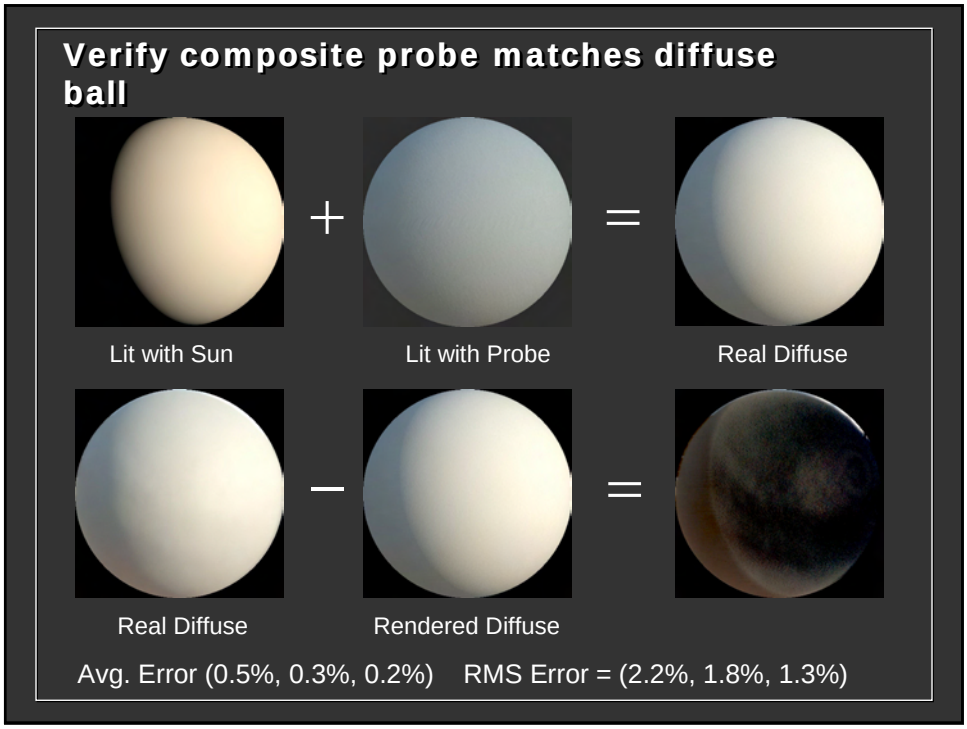
Solve for Sun Scaling Factor

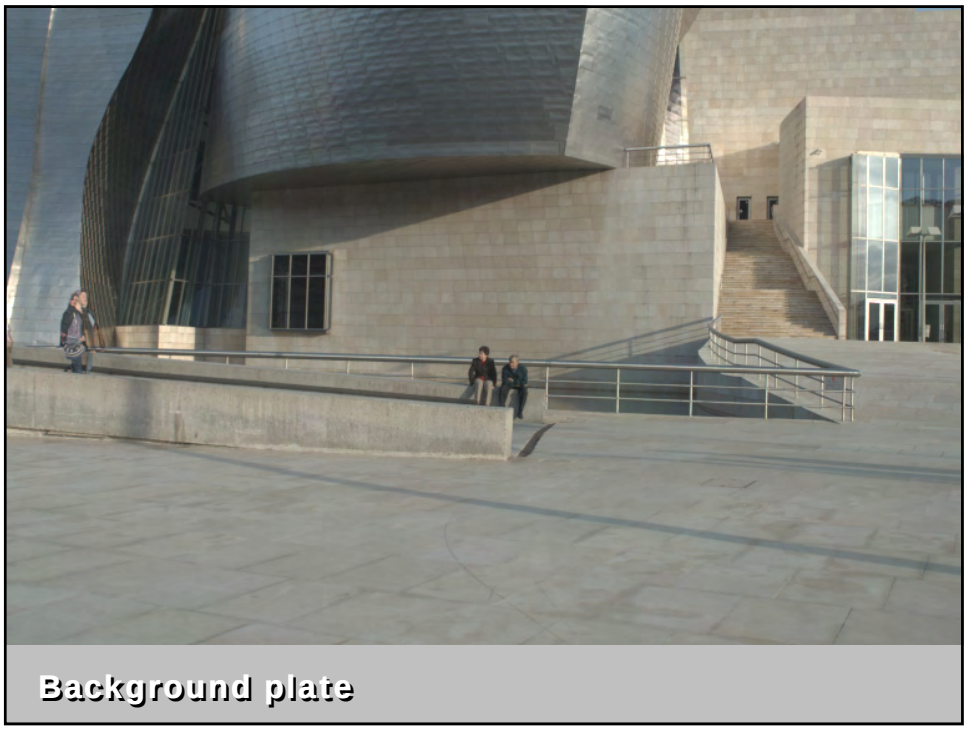


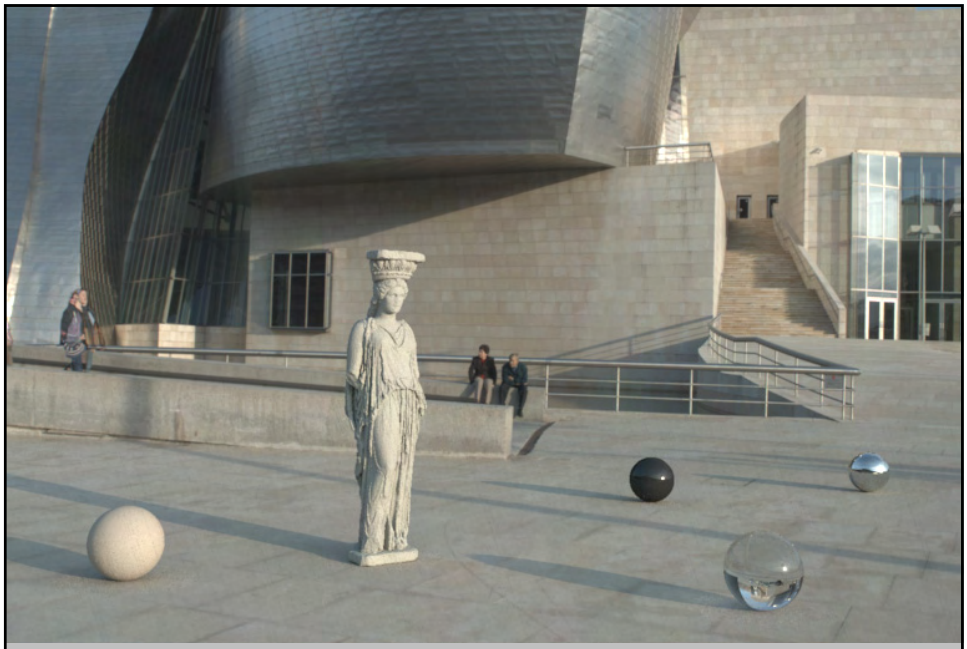
Diffuse Ball



α = (1.166, 0.973, 0.701)


Verify composite probe matches diffuse ball
Lit with Sun
Lit with Probe
Real Diffuse
Real Diffuse
Rendered Diffuse
Avg. Error (0.5%, 0.3%, 0.2%)
RMS Error = (2.2%, 1.8%, 1.3%)


Background plate



Synthetic objects added

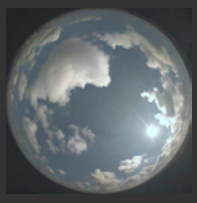
Direct HDR Capture of the Sun and Sky

Use Sigma 8mm fisheye lens and Canon EOS 1Ds to cover entire sky

Use 3.0 ND filter on lens back to cover full range of light

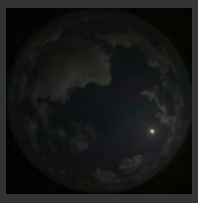
Stumpfel, Jones, Wenger, Tchou, Hawkins, and Debevec. "Direct HDR Capture of the Sun and Sky". To appear in Afrigraph 2004.

Extreme HDR Image Series

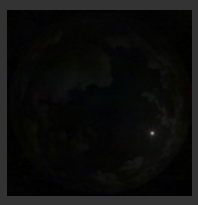


1 sec
f/4

1/4 sec
f/4



1/30 sec
f/4

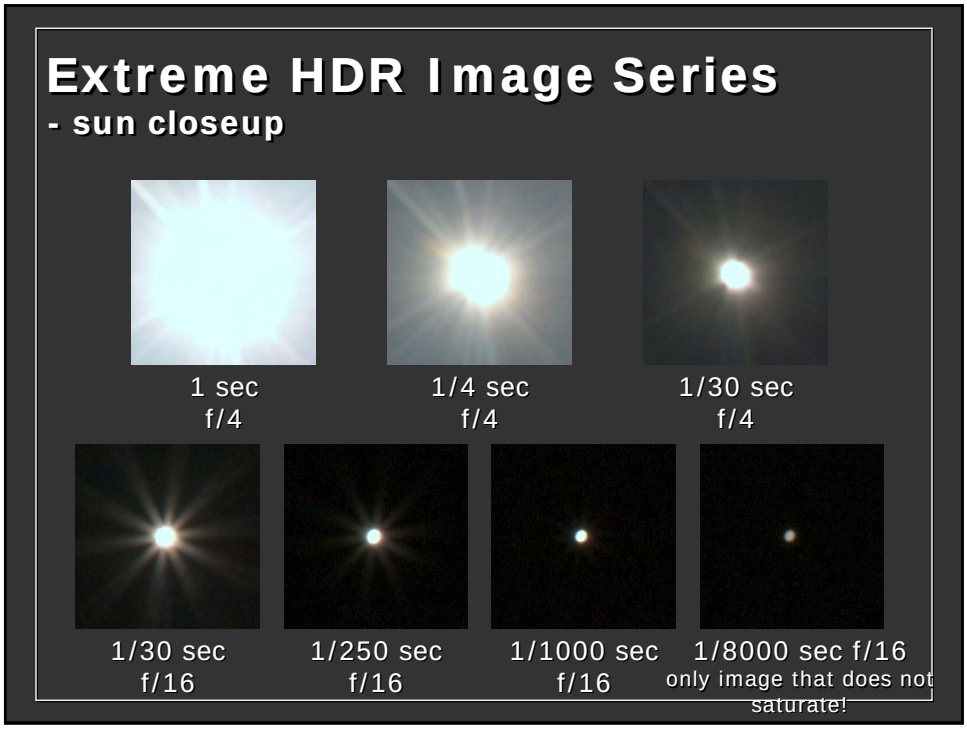


1/30 sec
f/16

1/250 sec
f/16

1/1000 sec
f/16

1/8000 sec
f/16


Extreme HDR Image Series
- sun closeup
1 sec
f/4
1/4 sec
f/4
1/30 sec
f/4
1/30 sec
f/16
1/250 sec
f/16
1/1000 sec
f/16
1/8000 sec f/16
only image that does not saturate!


Lit by sun and sky probe

HDR Image Representation

Greg Ward

SIGGRAPH2006

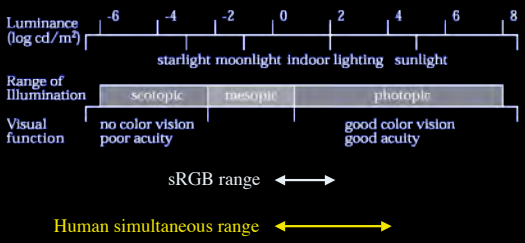
HDRI Representation

Greg Ward

Anyhere Software

Dynamic Range

From Ferwerda et al, SIGGRAPH '96



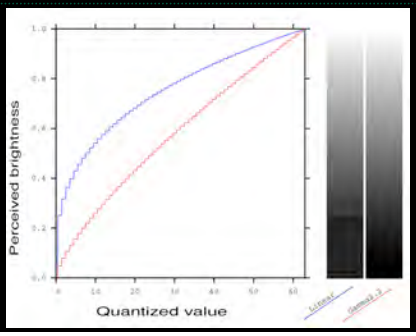
sRGB range ⟷

Human simultaneous range ⟷

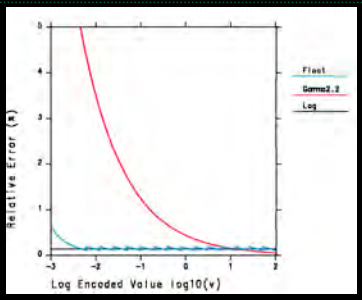
Value Encoding Methods

- Linear quantization
- Gamma function (e.g., CRT curve)
- Logarithmic encoding
- Floating point
- Perceptual

Linear vs. Gamma Encoding

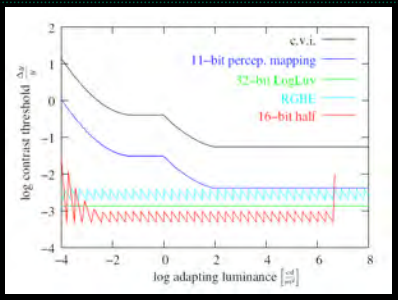


Logarithmic vs. Floating Point



Fictitious 12-bit encoding

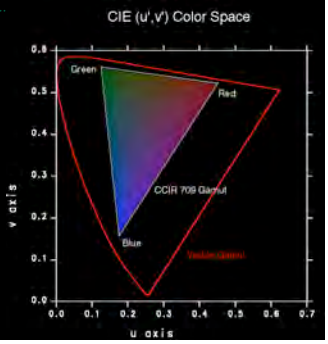
Perceptual Encoding



Mantiuk et al., SIGGRAPH 2004

CCIR-709 Color Space

- Human visible gamut is much larger than standard display's
- Saturated blues, greens, and purples are lost in sRGB
- Many HDR image formats also cover a larger color gamut



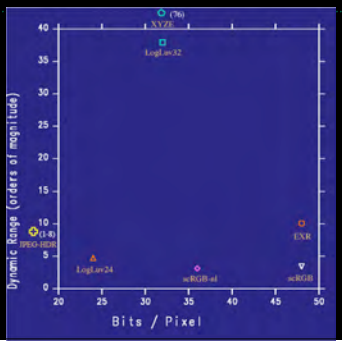
HDR Image & Video Formats

- Available high dynamic-range formats:
  - *Radiance* 32-bit RGBE and XYZE pictures
  - TIFF 48-bit integer and 96-bit float formats
  - SGI 24-bit and 32-bit LogLuv TIFF
  - ILM OpenEXR format
  - JPEG-HDR format
- Proposals and extensions:
  - HDR extensions to MPEG from MPI [Mantiuk et al. 2004]
  - HDR extensions to JPEG 2000 from UFL [Xu et al. 2005]
  - HDR texture compression (two papers at this conference)

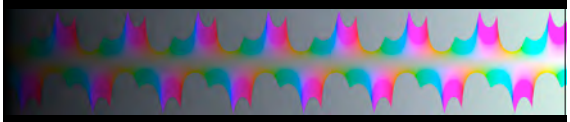
Encoding Comparison Chart

Encoding	Bits / pixel	Dynamic Range	Quant. Step	Covers Gamut
sRGB	24	1:10^1.6	Variable	No
Radiance RGBE	32	1:10^76	1%	No
Radiance XYZE	"	"	"	Yes
LogLuv 24	24	1:10^4.8	1.1%	Yes
LogLuv 32	32	1:10^38	0.3%	Yes
OpenEXR	48	1:10^10.7	0.1%	Yes
JPEG-HDR	1-7	1:10^9.5	Variable	Can

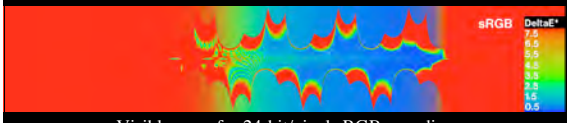
Full Gamut Encodings



HDR Acid Test Image

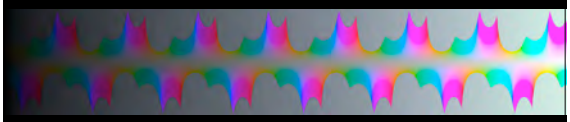


1:10^8 dynamic range, covering visible gamut

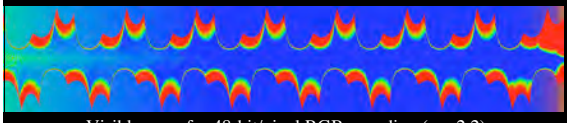


Visible error for 24-bit/pixel sRGB encoding

48-bit RGB TIFF Accuracy

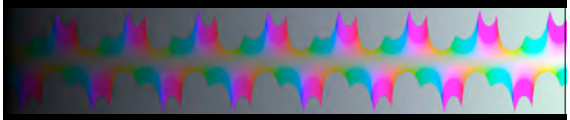


1:10^8 dynamic range, covering visible gamut

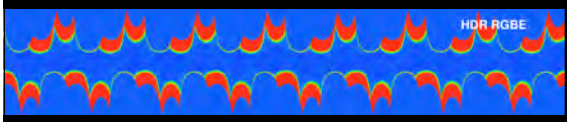


Visible error for 48-bit/pixel RGB encoding (γ = 2.2)

Radiance RGBE Accuracy

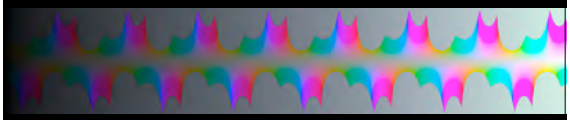


1:10^8 dynamic range, covering visible gamut

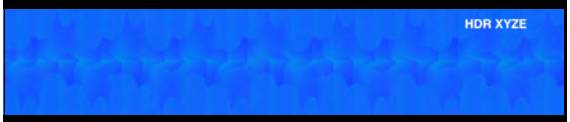


Visible error for 32-bit/pixel RGBE encoding

Radiance XYZE Accuracy

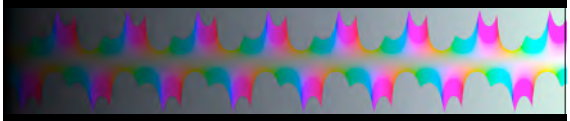


1:10^8 dynamic range, covering visible gamut

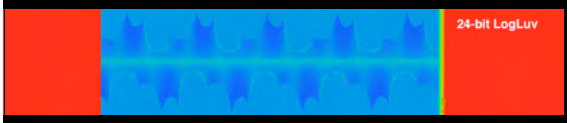


Visible error for 32-bit/pixel XYZE encoding

TIFF LogLuv24 Accuracy

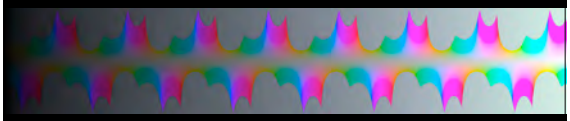


1:10^8 dynamic range, covering visible gamut

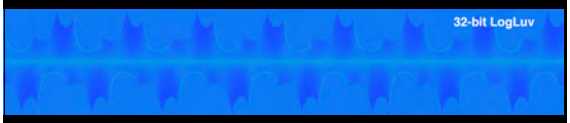


Visible error for 24-bit/pixel LogLuv encoding

TIFF LogLuv32 Accuracy

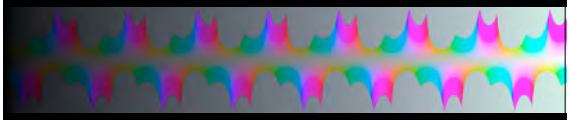


1:10^8 dynamic range, covering visible gamut



Visible error for 32-bit/pixel LogLuv encoding

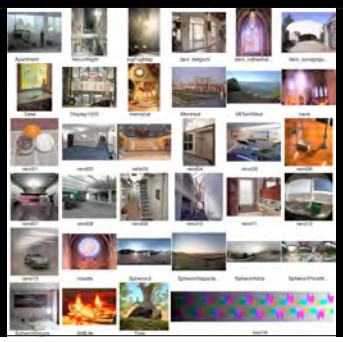
OpenEXR Accuracy



1:10^8 dynamic range, covering visible gamut

Visible error for 48-bit/pixel EXR `half` encoding

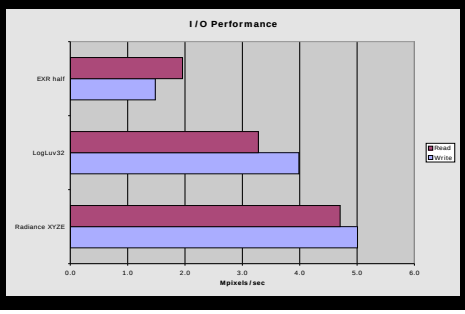
HDR Format Performance



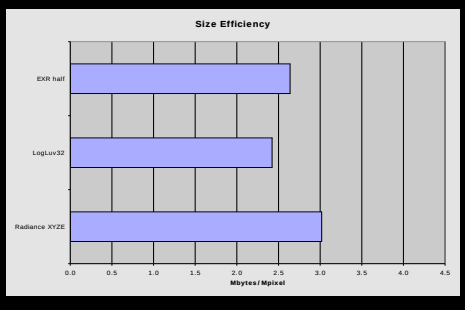
Test Set
(34 HDR images)

Measured i/o speed and image size

Read & Write Speed



Total Image Size



Radiance RGBE and XYZE

- Simple format with free source code
- 8 bits each for 3 mantissas and 1 exponent
- 76 orders of magnitude in 1% steps
- Run-length encoding (20% avg. compr.)
- RGBE format does not cover visible gamut
- Dynamic range at expense of accuracy
- Color quantization not perceptually uniform

Radiance Format (.pic, .hdr)

(145, 215, 87, 149) =
(145, 215, 87) * 2^(149-128) =
(1190000, 1760000, 713000)

(145, 215, 87, 103) =
(145, 215, 87) * 2^(103-128) =
(0.00000432, 0.00000641, 0.00000259)

Ward, Greg. "Real Pixels," in Graphics Gems IV, James Arvo ed., Academic Press, 1994

IEEE 96-bit TIFF

- Most accurate representation
- Support (with compression) in Photoshop CS2
- Uncompressed files are enormous
  - 32-bit IEEE floats look like random bits

16-bit/sample TIFF (RGB48)

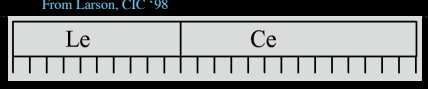
- Supported by Photoshop and TIFF library
- 16 bits each of log red, green, and blue
- 5.4 orders of magnitude in < 1% steps
- LZW lossless compression available
- Does not cover visible gamut
- Most applications think of max. value as “white”

SGI 24-bit LogLuv TIFF Codec

- Implemented in Leffler's TIFF library
- 10-bit LogL + 14-bit CIE (u',v') lookup
- 4.8 orders of magnitude in 1.1% steps
- Just covers visible gamut and range
- Amenable to tone-mapping as look-up
- Dynamic range is less than we would like
- No compression

24 -bit LogLuv Pixel

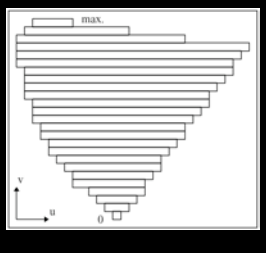
From Larson, CIC '98



$$L_e = \lfloor 64(\log_2 L + 12) \rfloor$$

$$u' = \frac{4x}{-2x+12y+3}$$

$$v' = \frac{9y}{-2x+12y+3}$$



SGI 32-bit LogLuv TIFF Codec

- Implemented in Leffler's TIFF library
- 16-bit LogL + 8 bits each for CIE (u',v')
- 38 orders of magnitude in 0.3% steps
- Run-length encoding (30% avg. compr.)
- Allows negative luminance value
- Amenable to tone-mapping as look-up

32-bit LogLuv Pixel

From Larson, JGT '98

$$L_e = \lfloor 256(\log_2 L + 64) \rfloor$$

$$u_e = \lfloor 410u' \rfloor$$

$$v_e = \lfloor 410v' \rfloor$$

$$u' = \frac{4x}{-2x+12y+3}$$

$$v' = \frac{9y}{-2x+12y+3}$$

ILM OpenEXR Format

- 16-bit/primary floating point (sign-e5-m10)
- 9.6 orders of magnitude in 0.1% steps
- Additional order of magnitude near black
- Wavelet compression of about 40%
- Negative colors and full gamut RGB
- Alpha and multichannel support
- Open Source I/O library released Fall 2002
- Slow to read and write

ILM's OpenEXR (.exr)

6 bytes per pixel, 2 for each channel, compressed

sign exponent mantissa

- Several lossless compression options, 2:1 typical
- Compatible with the "half" datatype in NVidia's Cg
- Supported natively on GeForce FX and Quadro FX

- Available at www.openexr.com

BrightSide Technology's JPEG-HDR Format

- Backwards-compatible JPEG extension for high dynamic range images
- Very compact: 1/10th size of other formats
- Naïve software displays tone-mapped sRGB
  - Different tone-mappings possible
- Desaturation can encompass visible gamut
- Lossy encoding so repeated read/write degrades
- Expensive (three pass) write process

File Size & HDR Adoption

- Compression can match size of JPEG images + 20%
- Rationale for "lossy" HDR:
  - Lossy encodings are all about perception
  - Lossy HDR supports display to the limits of human vision
  - Required for digital photography & web applications
  - Mantiuk et al.'s MPEG-4 extension (SIGGRAPH 2004)
  - Xu et al.'s JPEG-2000 extension (CG+A 2005)
  - Two 2006 papers on 8-bit/pixel HDR texture compression
- What if HDR format was backwards-compatible?
  - JPEG-HDR & new MPI technique (SIGGRAPH 2006)

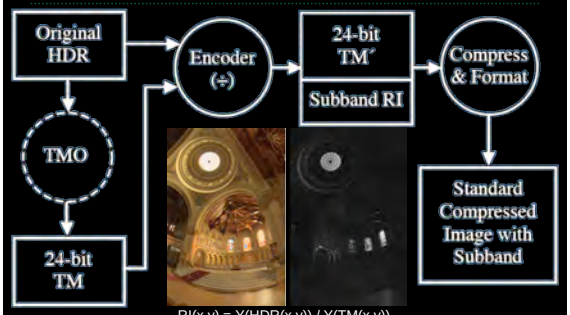
JPEG-HDR Format

Ward & Simmons 2004 & 2005

1. Tone-map HDR input into 24-bit sRGB
2. Write as *output-referred* JPEG
3. Record restorative information as metadata (supplement)

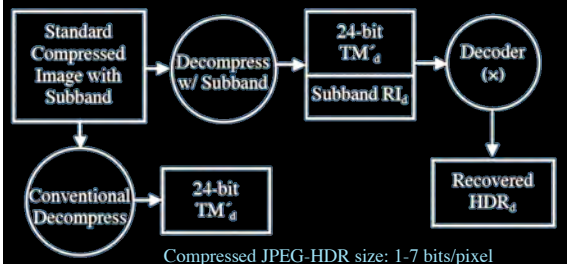
- Naïve applications see tone-mapped image
- HDR applications use supplement to recover *scene-referred* original
- Similar to Kodak's ERI (Spaulding et al. 2003)

JPEG-HDR Encoding Process



RI(x,y) = Y(HDR(x,y)) / Y(TM(x,y))

Decoding Process



Compressed JPEG-HDR size: 1-7 bits/pixel
(between 1/3 & 1/20 size of other HDR formats)

JPEG-HDR Software Availability

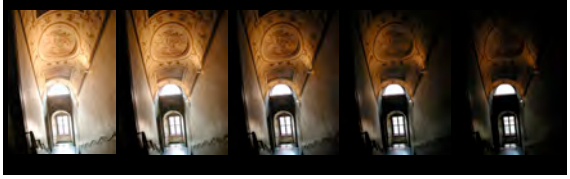
- Implemented as extension to Tom Lane's public JPEG library (www.ijg.org)
  - Tools & libraries included on HDRI book's DVD
  - Contact BrightSide Technologies for licensing info. at www.brightsidetech.com/software
- JPEG-HDR included in Photosphere
  - Handy export function for batch conversion and webpage creation
  - Download from www.anyhere.com

HDRI Encoding Conclusions

- Sufficient still formats to meet most needs:
  - Radiance RGBE for legacy systems
  - TIFF for greatest encoding variety
  - OpenEXR for good accuracy and support
  - JPEG-HDR for space efficiency
- HDR texture formats are in the works
- HDR video formats are coming soon

HDR Capture Refinements

- Automatic exposure alignment
- "Ghost" removal
- Lens flare removal
- Implementing HDR in still & video cameras



LDR Exposure Registration

[Ward 2003, *Journal of Graphics Tools*, 8(2)]

The *median threshold bitmap* (MTB) allows us to quickly compare and align different images, because it is constant with respect to exposure for any camera with a monotonic response function

The same is not true for an edge map, which changes with exposure even with careful normalization and approximate response curves

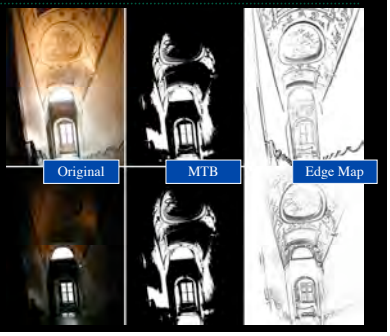
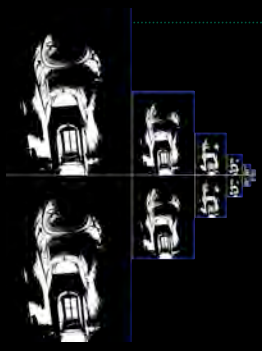


Image Pyramid Alignment



Grayscale images are scaled down repeatedly to create an image pyramid, which is then converted into MTBs for comparison

The smallest images are aligned first within a ±1 pixel distance, which corresponds to a ±32 pixel distance in the original

This becomes the MSB in the offset, which is shifted and used as the starting point for the next higher resolution alignment, and so on to the top

Alignment Results

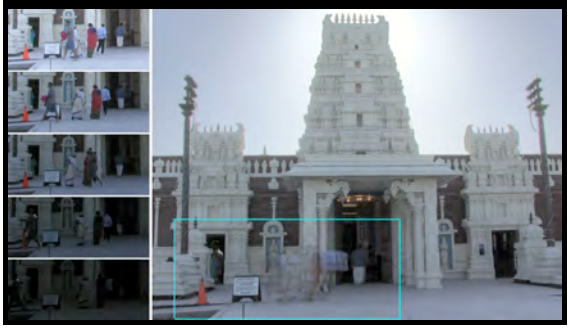
5 unaligned exposures | Close-up detail | MTB alignment

Time: About .2 second/exposure for 3 MPixel image

Automatic "Ghost" Removal

Before | After

Object Movement



Variance-based Detection

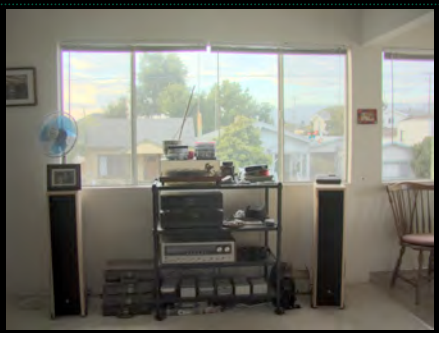
Region Masking

Best Exposure in Each Region

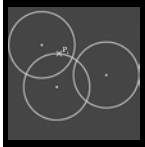
Lens Flare Removal

- From left image, we may directly measure the lens Point Spread Function (PSF)
- PSF is a function of focal length and aperture, so comprehensive measurement is impractical

More Usual Input



Estimate PSF from HDR Capture

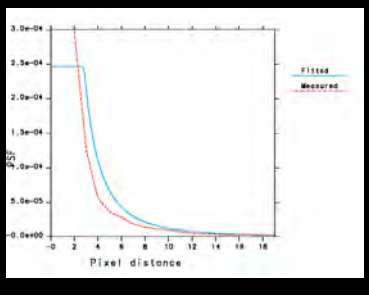


- For each "hot" pixel in input:
  - Find "coldest" relative pixel from at each radius
  - Consider overlapping hot pixel contributions
- PSF is minimal, monotonically decreasing function measured relative to cold pixels
  - Computed by fitting 3rd degree polynomial over all identified cold pixels:

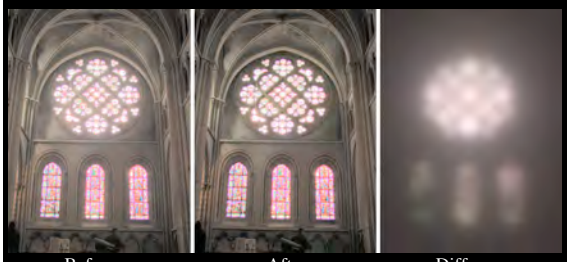
"Cold Pixel" $$P_i = \sum_j P_j \cdot \left(C_0 + \frac{C_1}{r_{ij}} + \frac{C_2}{r_{ij}^2} + \frac{C_3}{r_{ij}^3} \right)$$

Fitted vs. Measured PSF

PSF estimate (apt. capture fit vs. tin foil spot)



Apply: Simulate Flare & Subtract



Before After Difference

Photosphere Demo

- HDRI Browsing & Cataloging Application
  - Also builds HDRI's from bracketed exposures
- Available from www.anyhere.com
  - Mac OS X app., Linux command-line tool

Launch Photosphere

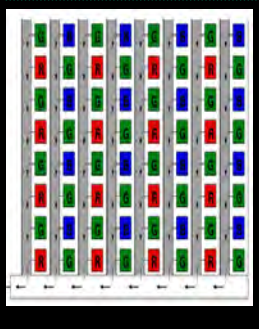
Fast Forward to HDR Cameras

- Leverage CMOS Sensor Technology
  - Fuji has sensor with dual-sensitivity pixel
  - SMaL Camera has log sensor
  - Pixim sensor has local pixel exposure
- Alter camera/sensor design
  - Multi-image capture using modified scanout
  - Multiple sensors
  - Spatially varying filters for video mosaicing
  - Sensors with assorted pixels
  - Adaptive dynamic range system

Two-Exposure HDR

- Compared Results to 5 exp. on 12 Scenes
  - Two exposures was usually sufficient
  - Less noise averaging but otherwise comparable
- Camera Implementation Reduces Artifacts
  - No alignment issues on short exposures
  - Longer exposure akin to "slow flash"
- Marginal Manufacturing Cost: $0.00

Interline CCD Scanout



Old Program:

Electronic shutter holds each exposure during scanout

Preview/movie uses electronic shutter, while still capture relies on mechanical shutter

New Program:

Instead, shift pixels under electronic shutter with 1/16th of mechanical exposure still remaining

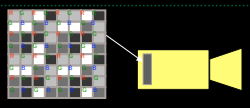
After scanning out long exposure, shift and scan out short exposure

Result: two exposures separated by 4 f-stops

Sample Results

Assorted Pixels

Nayar and Mitsunaga, IEEE CVPR 2000
Nayar and Narasimhan, ECCV 2002

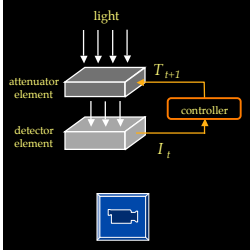


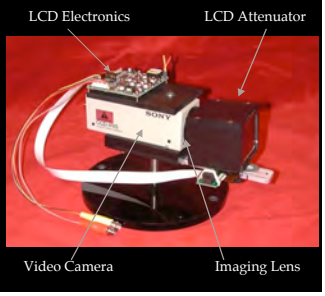
Digital Still Camera

Camera with Assorted Pixels

Adaptive Dynamic Range

Nayar and Branzoi, ICCV 2003





Conclusions

- HDR capture is still under active development
- HDR video is challenging but has many potential benefits
- Movie industry is an early adopter
- Home entertainment market will soon follow

HDR Video and Applications

Karol Myszkowski

max planck institut informatik

SIGGRAPH2006

High Dynamic Range Video

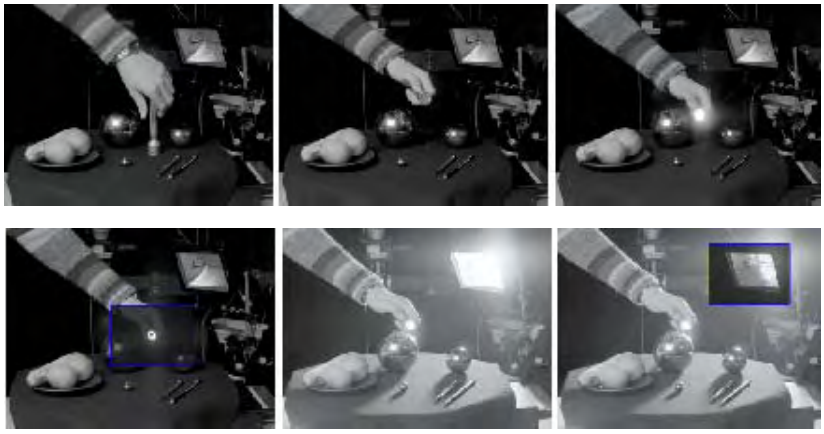
Rafal Mantiuk, Grzegorz Krawczyk, **Karol Myszkowski**
MPI Informatik, Germany

Overview

SIGGRAPH2006

- HDR video compression
- Real-time HDR playback with perceptual effects
- HDR Visible Differences Predictor
- HDR video environment maps

HDR Video Compression



http://www.mpi-inf.mpg.de/resources/hdr/compression

HDR Video Compression

- Color space for HDR pixel encoding
- 12-bit HDR MPEG
- 8-bit Backward Compatible HDR MPEG
- Demo

Standard *HDR-ready* Image and Video Formats

- Basic requirement
  - Capability of storing larger number of bits
- Formats capable of HDR
  - JPEG-2000
    - ➔Up to **16**-bit integers per component
    - ➔HDR extension [Xu et al. IEEE CG&A 2005]
  - MPEG-4 (IOS/IEC 14496-2 or ISO/IEC 14496-10)
    - ➔Up to **12**-bit integers per component
    - ➔HDR extension [Mantiuk et al. ACM Siggraph 2004]
- Needed special color space for HDR pixel encoding

Color Space for HDR pixels

- Full visible range of luminance and color gamut
- Perceptually uniform
- Only positive integer numbers
- Quantization below the detection threshold
- Minimum correlation between color channels
- Direct transformation: color space <--> XYZ

Luminance and Luma

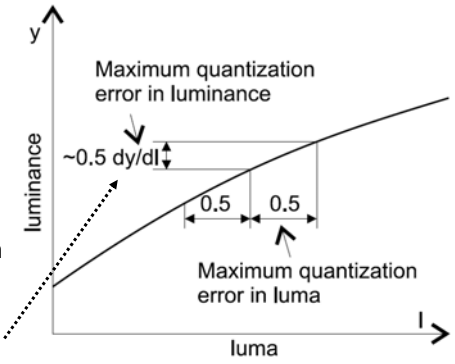
$$l \rightarrow y : y(l)$$

luma — luminance

From Wikipedia:

Luma is a new word proposed by the NTSC in 1953 (..) While luminance is the weighted sum of the linear RGB components of a color video signal, proportional to intensity, luma is the weighted sum of the non-linear R'G'B' components **after gamma correction** has been applied ...

$$y(l_o + 0.5) - y(l_o) \approx 0.5 \cdot \frac{dy}{dl}$$



Luminance ←→ Luma

Maximum quantization error < the detection threshold (JND)

$$0.5 \cdot \frac{dy(l)}{dl} < t(y_{\text{adapt}})$$

Adaptation to a single pixel: $y_{\text{adapt}} = y$

Inequality to equality:

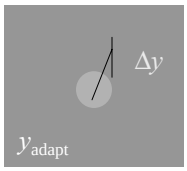
$$\frac{dy(l)}{dl} = 2 \cdot \frac{t(y)}{k} \qquad \text{JND scaled by } k > 1$$

Boundary conditions:

$$y(0) = 10^{-5}\left[\frac{\text{cd}}{\text{m}^2}\right], \quad y(l_{\max}) = 10^{10}\left[\frac{\text{cd}}{\text{m}^2}\right] \text{ for } l_{\max} = 2^{\text{bits}} - 1$$

Contrast Detection Models: $t(y_{adapt})$

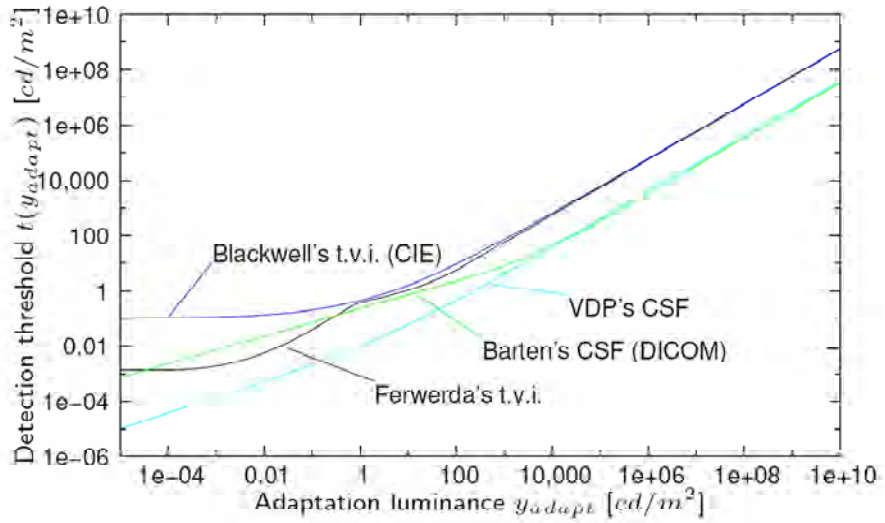
- Threshold versus intensity function (t.v.i.)
  - Measured for a fixed pattern on an uniform background
    - ➔Ferwerda's t.v.i. (computer graphics)
    - ➔Blackwell's t.v.i. (CIE 19/2.1 standard)



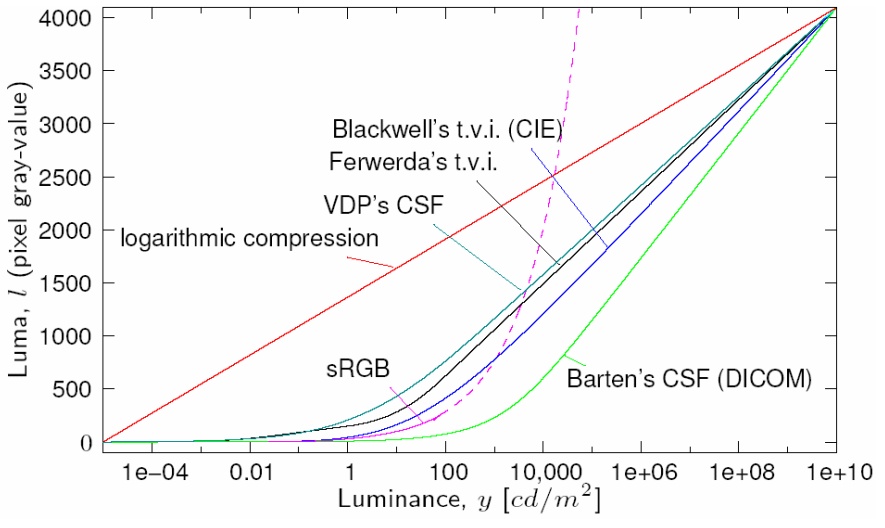
Contrast Detection Models: $t(y_{adapt})$

- Contrast Sensitivity Function (CSF)
  - Measured for sinusoidal patterns or Gabor patches of various spatial frequencies and orientations:
    - ➔Barten's CSF model (DICOM's grayscale standard display function)
    - ➔CSF model used in Daly's Visible Differences Predictor
  - for each y_{adapt} take the maximum sensitivity

Contrast Detection Models: $t(y_{adapt})$



Luminance to Luma Mapping



Conversion: Luminance → Luma

$$l(y) = \begin{cases} a \cdot y & \text{if } y < y_l \\ b \cdot y^c + d & \text{if } y_l \le y < y_h \\ e \cdot \log(y) + f & \text{if } y \ge y_h \end{cases}$$

Model	a	b	c	d	e	f	y_l	y_h
CIE *t.v.i.*	17.554	826.81	0.10013	-884.17	209.16	-731.3	5.6046	10469
VDP's CSF	769.18	449.12	0.16999	-232.25	181.70	-90.16	0.061843	164.10

Conversion: Luma → Luminance

$$y(l) = \begin{cases} a' \cdot l & \text{if } l < l_l \\ b' \cdot (l + d')^{c'} & \text{if } l_l \le l < l_h \\ e' \cdot \exp(f' \cdot l) & \text{if } l \ge l_h \end{cases}$$

Model	a'	b'	c'	d'	e'	f'	l_l	l_h
CIE *t.v.i.*	0.0570	7.3014e-30	9.987	884.17	32.994	0.00478	98.381	1204.7
VDP's CSF	0.0013	2.4969e-16	5.883	232.25	1.6425	0.00550	47.568	836.59

Chrominance and Chroma

- CIE 1976 Uniform Chromacity Scales ($u'v'$)

$$u' = \frac{4X}{X + 15Y + 3Z} \qquad v' = \frac{9Y}{X + 15Y + 3Z}$$

$$u_{8bit} = u' \cdot 410 \qquad v_{8bit} = v' \cdot 410$$

- Just noticeable differences (JND) [Hunt 1995, p. 154]

$$\Delta u', \Delta v' \approx 0.002 \quad (\text{or } \Delta u_{8bit}, \Delta v_{8bit} \approx 0.82)$$

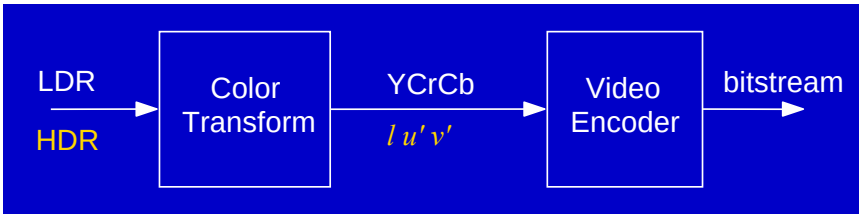
- 8-bit encoding gives sufficient precision for complex images
  - For smooth patterns we could see contouring artifacts for blue and purple colors for the highest luminance levels
- $L^*u^*v^*$ - not suitable for compression
  - too much correlation between luma and chroma

HDR Color Space: Summary

- Color space for HDR image and video encoding
  - 12-bit luma, 2 × 8-bit chroma
  - Derived from contrast detection data
  - Meets all requirements: full color gamut and luminance range; perceptually uniform; no countouring; positive integer numbers; channels decorrelated; conversion formulas from/to XYZ
- Simple extension for the existing formats

HDR Video Encoding [Mantiuk et al. Siggraph 2004]

- An extension to MPEG-4 (ISO/IEC 14496-2)
- $l\,u'v'$: 11 bits for luma, 2×8 bits for chroma

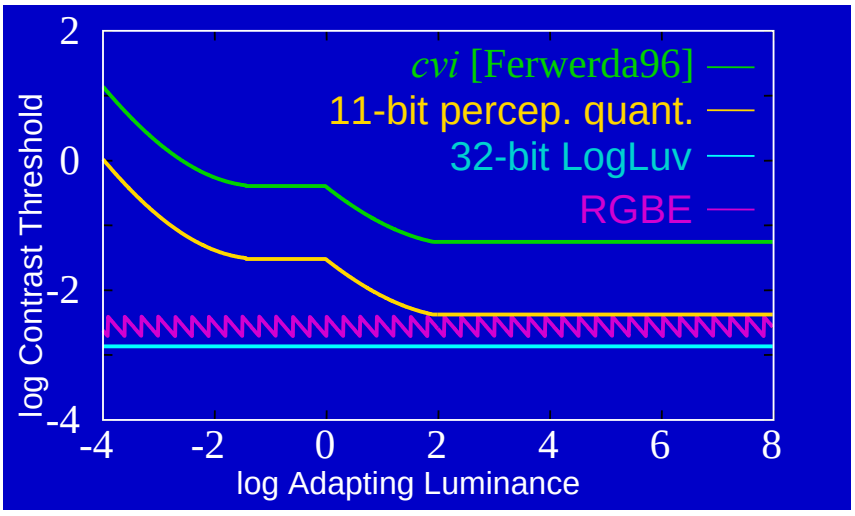


- White: MPEG
- Orange: HDR Encoder

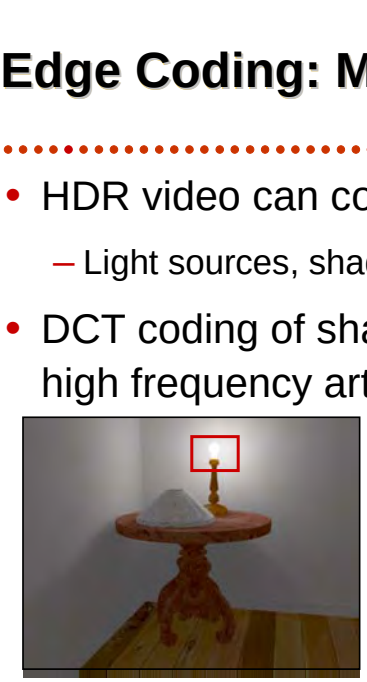
l: Ferwerda's t.v.i. for luma encoding

➔Similar to Capacity function [Ashikhmin, EGSR 2002]

Luminance Quantization Comparison



Edge Coding: Motivation

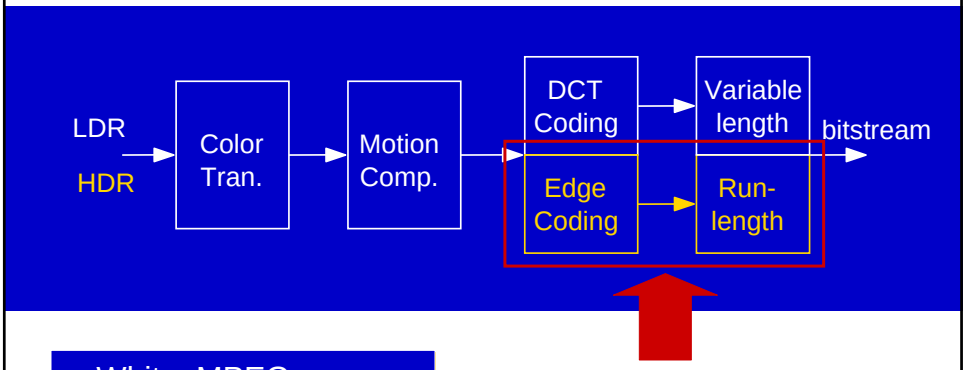


- HDR video can contain sharp contrast edges
  - Light sources, shadows
- DCT coding of sharp contrast may cause high frequency artifacts

DCT coding

Edge coding

Edge Coding

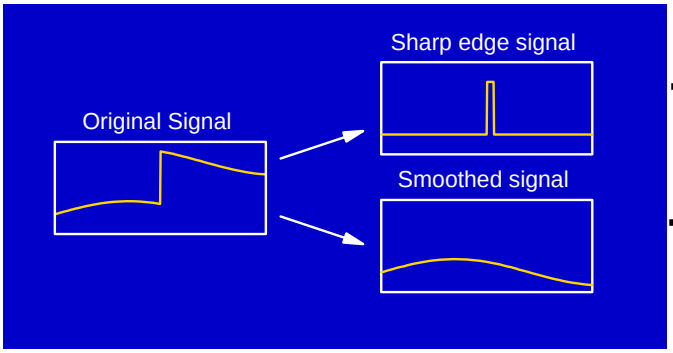


- White: MPEG
- Orange: HDR Encoder

Edge Coding: Solution

SIGGRAPH2006

- Solution: Encode sharp edges in spatial domain, the rest in frequency domain

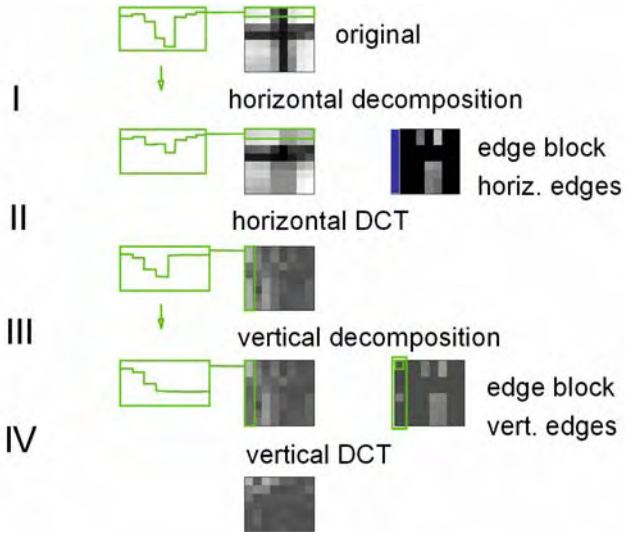


➔Run-length encoding

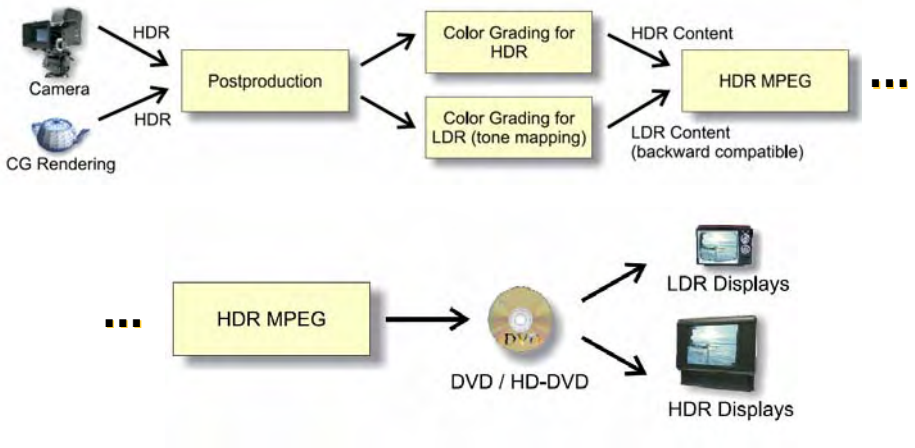
➔DCT encoding

Edge Coding: Algorithm

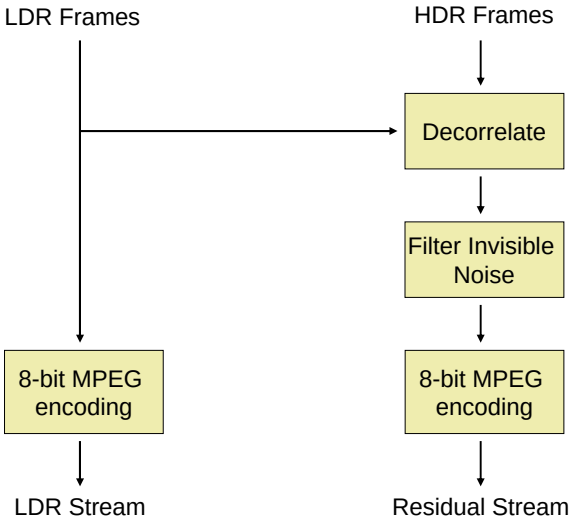
SIGGRAPH2006



Backward Compatible HDR MPEG [Mantiuk et al. Siggraph 2006]



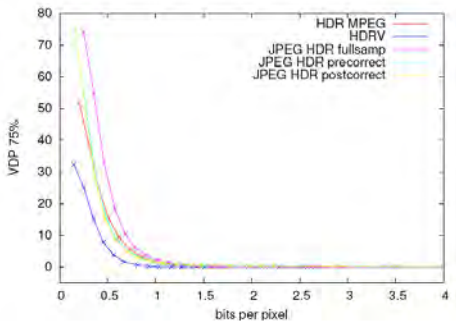
Backward Compatible HDR MPEG: Overview

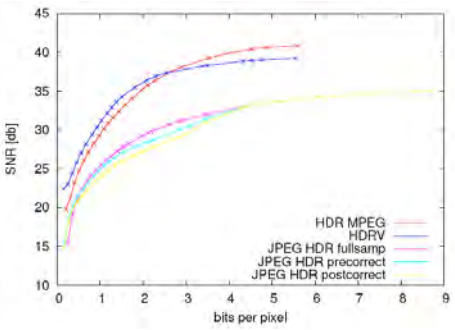


Backward Compatible HDR MPEG

- Standard chipset can be used for 8-bit MPEG-4 decoding
- HDR and LDR video appearance not affected by the compression scheme
  - Arbitrary tone mapping can be used
- Backward compatibility of LDR stream with existing DVD players
- Modest size of residual stream
  - Perception-based HDR-enabled filtering of invisible details

12-bit (HDRV) vs. 8-bit Backward Compatible HDR MPEG





Real-Time HDR Playback with Perceptual Effects

http://www.mpi-inf.mpg.de/resources/hdr/peffects

HDR Video Player

- Tone mapping adjusted to display device
- Inspecting various luminance ranges with a linear luminance mapping
- Physically based post-processing effects: blooming, motion blur, night vision
  - Require HDR information
  - Computed on-the-fly by graphics hardware
  - Can be scripted through annotations in video stream

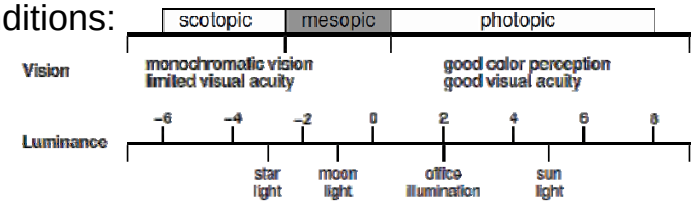
Real-time Tone Mapping for LDR/HDR Displays

tone mapping customized for a given display

Logarithmic mapping Photographic (local) mapping

Night Vision

- Human Vision operates in three distinct adaptation conditions:



Temporal Luminance Adaptation

SIGGRAPH2006

Light adaptation

- Compensates changes in illumination
- Simulated by smoothing adapting luminance in tone mapping equation
- Different speed of adaptation for photopic and scotopic vision

Tunnel entrance: dark adaptation

Visual Acuity

SIGGRAPH2006

- Perception of spatial details is limited with decreasing luminance level (Y)
- Details can be removed using convolution with Gaussian Kernel
- Highest resolvable spatial frequency

[Shaley 1937, Ward 1997]:

$$RF(Y) = 17.25 \cdot \arctan(1.4 \log_{10} Y + 0.35) + 25.72$$

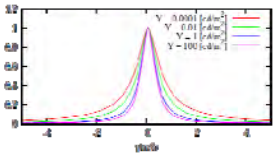
Veiling Luminance (Glare)

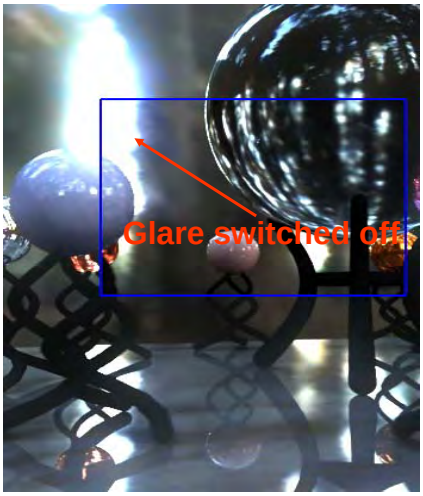
SIGGRAPH2006

- Decrease of contrast and visibility due to light scattering in the optical system of the eye
- Described by the optical transfer function [Deeley 1991]:

$$OTF(\rho, d(\bar{Y})) = \exp\left(-\frac{\rho}{20.9 - 2.1 \cdot d}^{1.3-0.07 \cdot d}\right)$$

ρ spatial frequency, d pupil aperture



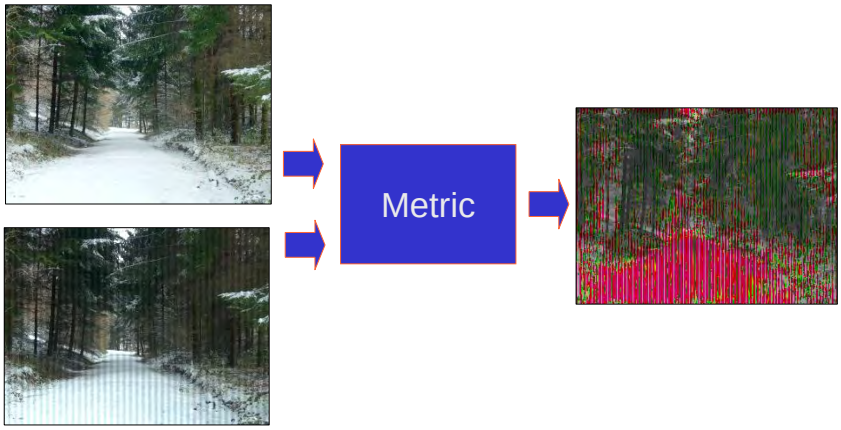


Motion Blur

SIGGRAPH2006

LDR HDR

HDR
Visible Difference Predictor



http://www.mpi-inf.mpg.de/resources/hdr/vdp

HDR VDP Overview

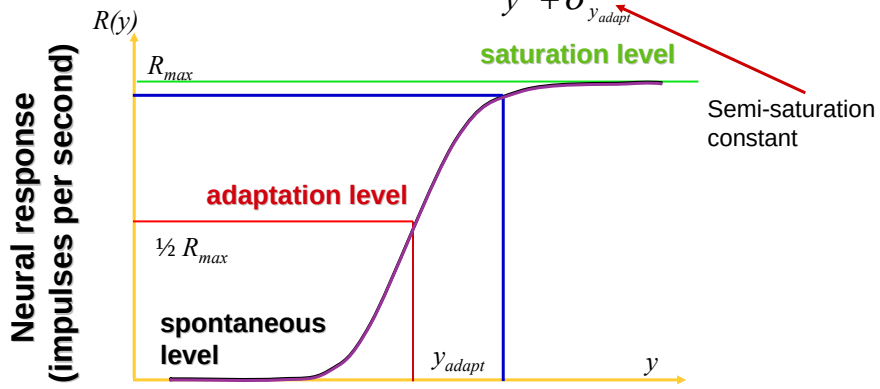
- Relevant characteristics of the Human Visual System
- LDR VDP
- HDR VDP extensions

Subjective vs. Objective Methods

- Subjective methods involving human subjects
  - Very costly
- Simple objective metrics e.g. RMS Error
  - Not reliable
- Basic characteristics of the Human Visual System (HVS) must be modeled to improve difference prediction reliability:
  + Luminance adaptation
  + Contrast sensitivity
  + Visual masking

Photoreceptor Response

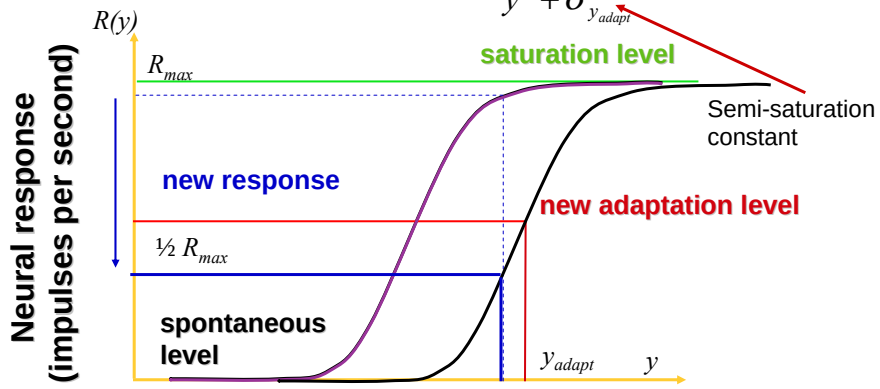
$$\text{Michelis} - \text{Menten Equation}: R(y) = \frac{y^n}{y^n + \sigma_{y_{adapt}}^n} R_{\max} \qquad n \approx 0.74$$



Photoreceptor Response

SIGGRAPH2006

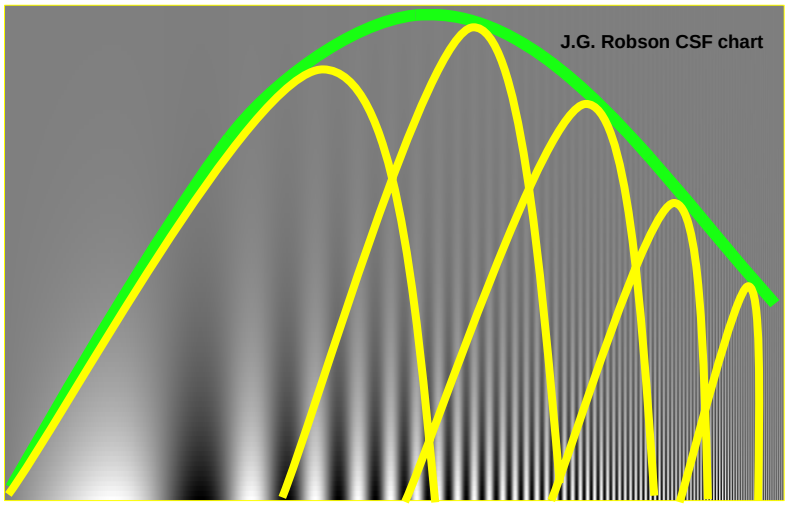
$$\text{Michelis} - \text{Menten Equation}: R(y) = \frac{y^n}{y^n + \sigma_{y_{adapt}}^n} R_{\max} \qquad n \approx 0.74$$



Luminance of light falling on receptive field's center

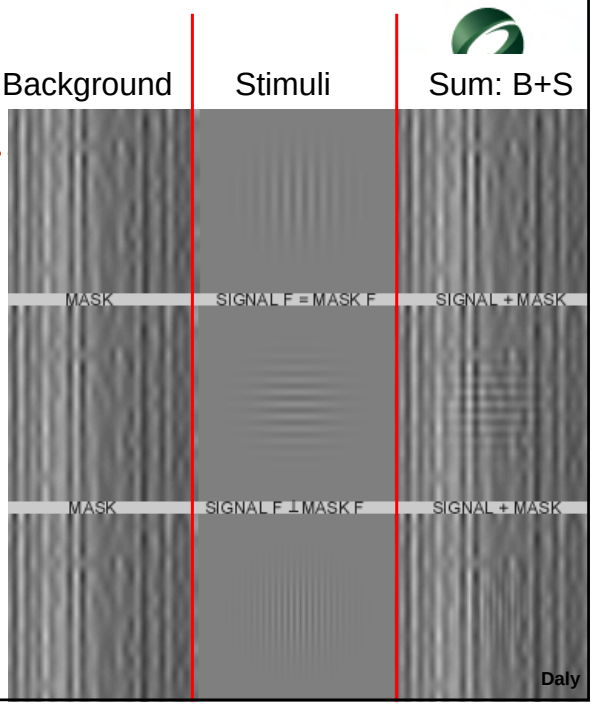
Contrast Sensitivity Function

SIGGRAPH2006



Visual Masking

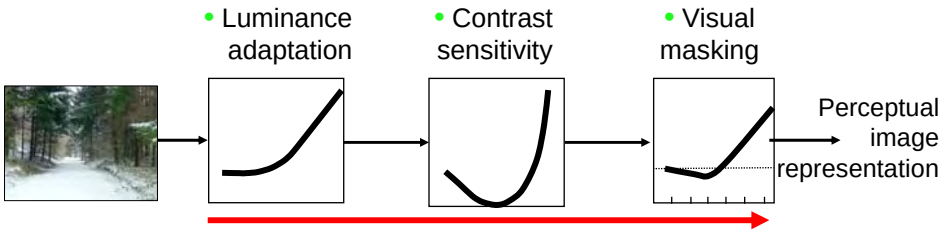
- Strong masking: similar spatial frequencies
- Weak masking: different orientations
- Weak masking: different spatial frequencies



Typical HVS Model

SIGGRAPH2006

Detection of perceivable differences between images depends on the following characteristics of the HVS:

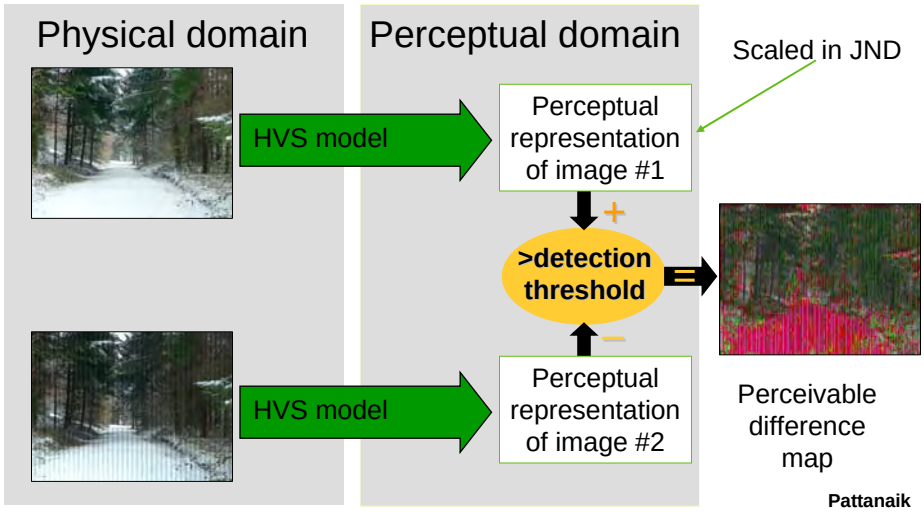


Increase of the detection threshold:

- With increase of adaptation luminance
- With increase of spatial frequencies
- With contrast increase

Pattanaik

Perceivable Differences Predictor [Daly SPIE 1992]



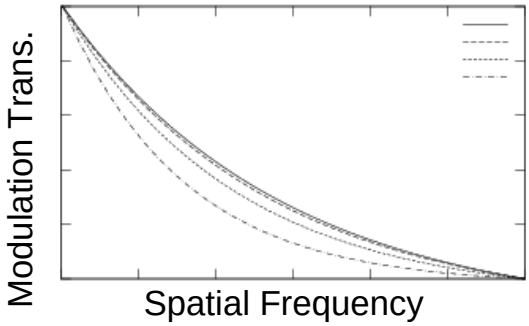
HDR VDP – Extensions [Mantiuk et al. SPIE 2005]

- Optical Transfer Function
  - Glare effect important for high contrasts
- JND scaled contrast
  - Contrast measure that does not depend on adaptation luminance
- Locally adaptive CSF
  - Instead of global adaptation for the whole image

➢ Assumption: The eye can adapt to luminance of very small patches (to a single pixel)

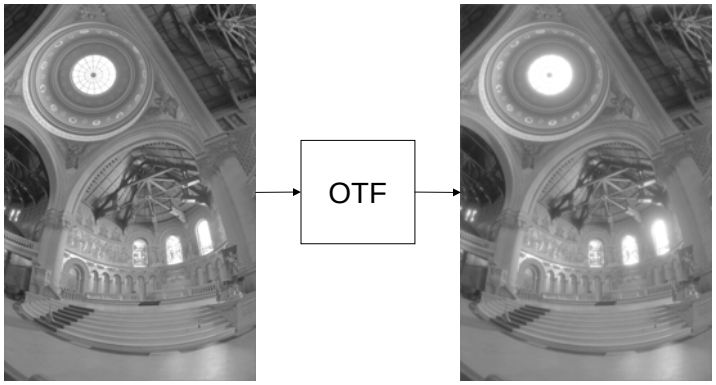
Optical Transfer Function

- Modulation Transfer function that simulates optics of the human eye [Deeley et al. 1991]

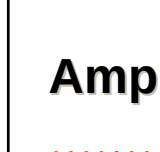


Optical Transfer Function

- OTF significantly reduces contrast in HDR scenes

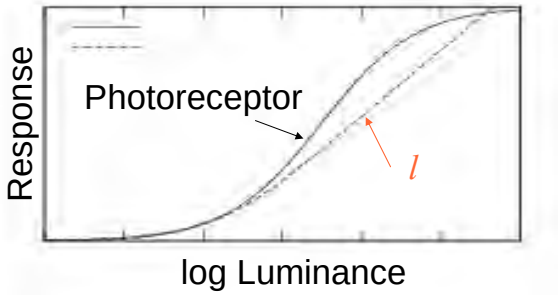


Amplitude Nonlinearity

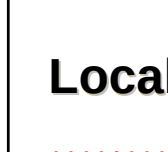


- Introduced contrast measure independent of adaptation luminance
- Scaled in JND units

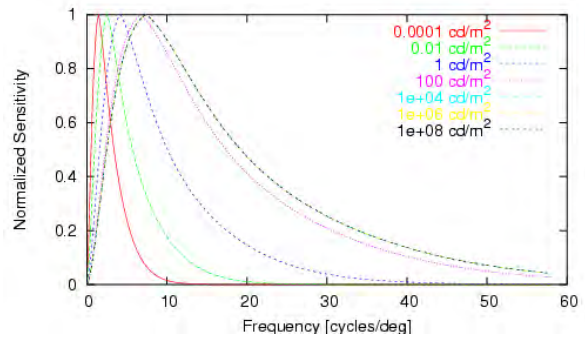
➔ our HDR luma space *l*



Local Adaptation for CSF



- The shape of CSF varies with adaptation luminance



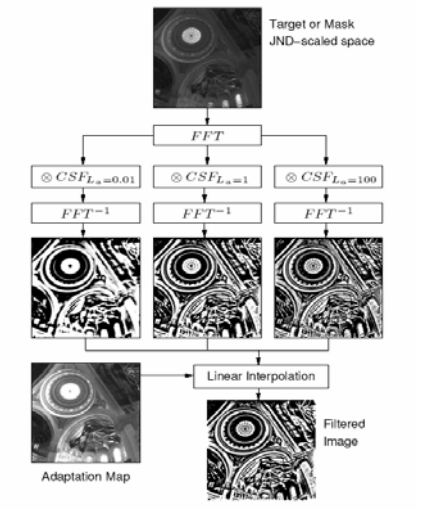
- Adaptation luminance can vary significantly across an image

Local Adaptation for CSF

SIGGRAPH2006

1. Use different CSF for filtering each part of an image

2. Interpolate filtered images depending on adaptation luminance [Durand03]



Calibration

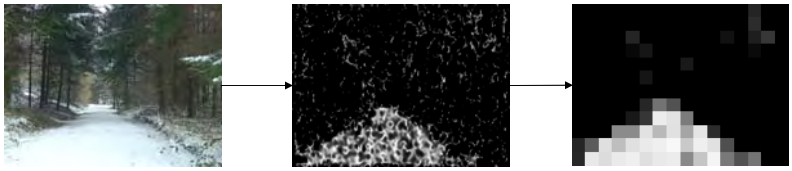
SIGGRAPH2006

- An experiment conducted on an HDR display to find subjective difference probability map
- HDR VDP parameters optimized to fit closely subjective data

Calibration: Experiment

- Subjects were to mark visible differences using rectangular blocks
- Results averaged across subjects
  - ➔fuzzy detection probability map

Calibration: Data Fitting



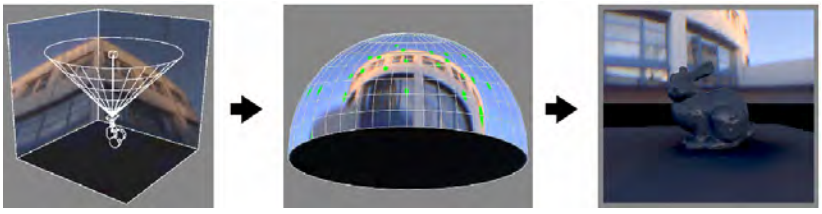
Distorted Image VDP Response Integrated Resp.

- HDR VDP response converted to format of the subjective data
- Found the best fit for peak threshold contrast and masking function slope

HDR VDP: Summary

- Objective visual difference metric
  - Not limited to existing display technology
  - Predicts changing sensitivity in bright and dark regions of an image
  - Small performance overhead
- Applications
  - HDR JPEG and MPEG compression
  - New display technology (HDR display)
  - Assessment of visibility for varying luminance

HDR Video Environment Maps



HDR Illumination Acquisition

Light Sources Computation

Rendering (GPU) (+Compositing)

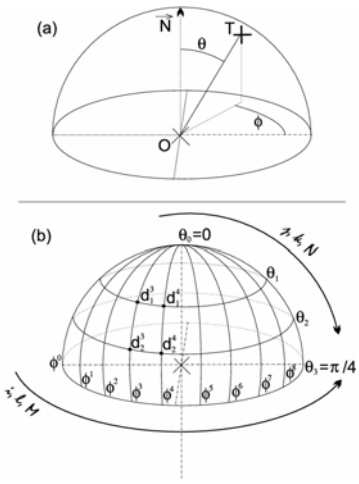
http://www.mpi-inf.mpg.de/resources/hdr/vem

Illumination Acquisition

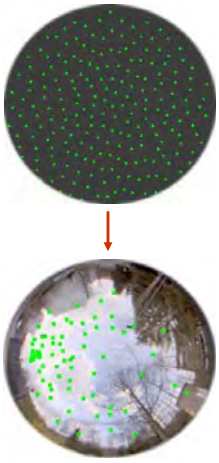
- Source of data: HDR camera
  - 12-bit logarithmic response
  - 8 orders of magnitude dynamic range
- Fish-eye lens
- Resolution 640x480
- Photometric calibration necessary

Video Environment Map (VEM) [Havran et al. EGSR 2005]

- HDR image on hemisphere defined by meridians and parallels
- Extension to time domain
- Luminance corresponds to the probability density function on the hemisphere (grid points)



VEM Decomposition into Directional Light Sources

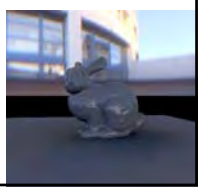


Importance Sampling Properties

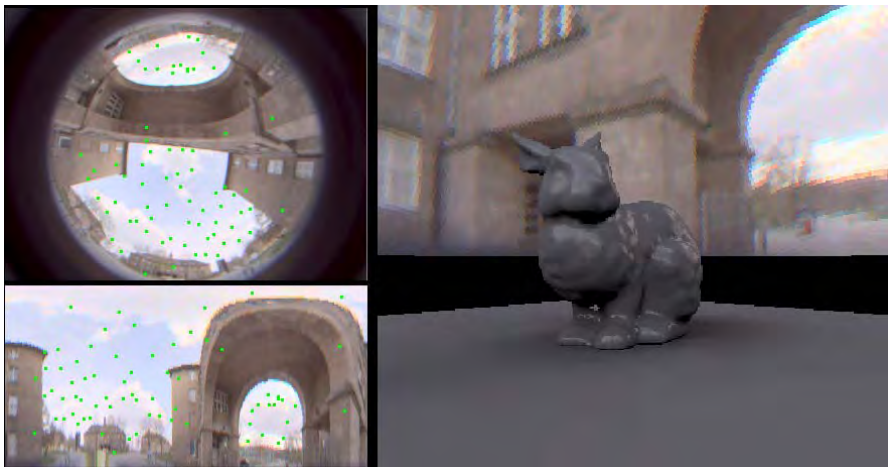
- Arbitrary number of directional light sources
- The same energy for each light source
- Progressive light source sequence
- Small memory requirements
- Real-time performance
- Dependence on the surface normal
- Energy and position of light sources processed with FIR filters

Rendering System

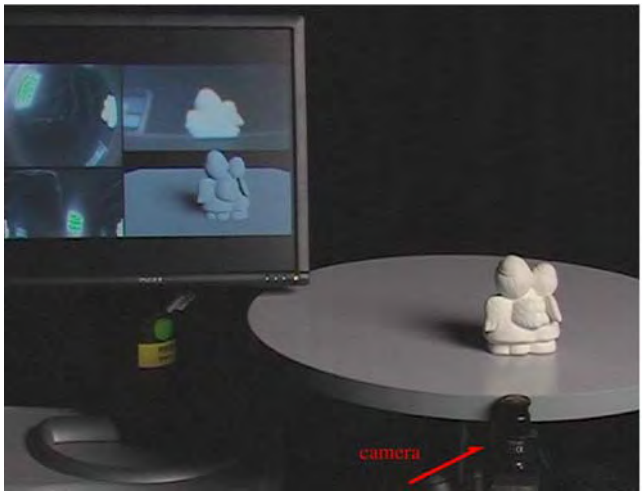
- Full pipeline from HDR acquisition to rendering
- Real-time computation of light sources with many properties useful for rendering
- Performance 5-10 FPS for 320×240 pixels on NVidia GeForce 6800GT
- Principles also useful for CPU rendering systems
- Dynamic geometry supported
- No interreflections



Snapshots & Video



Mixed Reality Applications



Special Case: Dynamic Lighting and Static Geometry

- CAVE system for car interior design and ergonomic studies
- Real-world driving scenarios
  - Tunnels, sunset, city driving, highways, etc
- Two HDR video cameras with fisheye lenses that capture:
  - Environment map
  - Windshield image

Precomputed Radiance Transfer (PRT) Rendering

- Car model contains approx. 500K triangles
- Visualization frame rate ~10fps

Acknowledgements

- We would like to thank Tom Annen, Scott Daly, Kirill Dmitriev, and Vlastimil Havran for their respective contributions to the presented work.
- We are grateful to Helge Seetzen and Greg Ward for their friendly advice and BrightSide Technologies for making the whole family of their HDR displays available to us.
- We are also grateful to Bernd Hoefflinger and IMS CHIPS for providing us with HDR cameras.

Tone Reproduction

Erik Reinhard

Tone reproduction

Match range of image intensities to those of the display device.

Possible approaches:

- Linear scaling
- Image formation
- Human visual system

Linear scaling

Linear

Linear + cropping

Image formation

Assumption: pixel intensities are the result of light reflecting off surfaces and hitting an image sensor.

A much simplified model: $L = i * r$

- L: pixel intensity
- i: illuminance
- r: surface reflectance

Surface reflectance

Surfaces reflect a fraction of between 0.005 and 1.0 of incident light (Stockham 1972)

Surface reflectance does not by itself create high dynamic range images.

Illuminance

The amount of light incident upon a surface could have any range of values (as long as these values are positive)

$L = i * r$: If L has a high dynamic range, then this is because i has a high dynamic range

Intensity vs. Density

In photography images are often represented as densities, i.e. in the logarithmic domain:

$$D = \log(L) = \log(i) + \log(r)$$

Note that D could be negative

The notation given here will be used throughout this section of the course. D is a ‘density image’

Frequency domain analysis

One more observation about reflectance and illuminance:

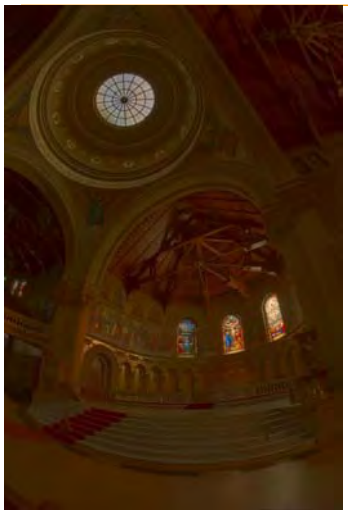
- Reflectance tends to exhibit high spatial frequencies
- Illuminance is slowly varying over surfaces

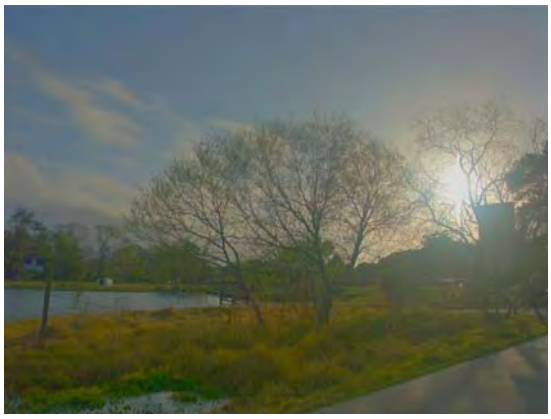
Oppenheim 1968

Given a set of pixel intensities, we wish to attenuate the illuminance component while keeping the reflectance component

Solution: Take the Fourier transform of a density image and attenuate the low frequencies. Then exponentiate to form a displayable image

Oppenheim 1968





Durand and Dorsey 2002

Split image into base- and detail layers:

- Use edge-preserving smoothing operator to filter density image and call result ‘base layer’
- Divide density image by base layer and call result ‘detail layer’

Durand and Dorsey 2002

- Note that base layer is high dynamic range and could be viewed as the illuminance component
- The detail layer is low dynamic range and could be viewed as the reflectance component

Durand and Dorsey 2002

Input image

Smoothed result

Durand and Dorsey 2002

Related techniques

- Choudhury and Tumblin (2003): Trilateral filter
  - A refinement of the edge-preserving smoothing operator used by Durand and Dorsey
- Pattanaik and Yee (2002):

  Gain control operator
  - Different kernel shapes

Alternatively…

Illuminance is slowly varying over a surface, whereas reflectance produces sharp discontinuities

In other words: large image gradients are caused by reflectance edges

Horn 1974

Model of lightness (i.e. a model of the perception of surface reflectance):

- Differentiate a density image
- Threshold density gradients (keep large gradients)
- Integrate image (numerical process)
- Exponentiate

Horn 1974

Original Processed

Image formation

This model of image formation breaks down

- if specular highlights are visible in the image
- if light sources are directly visible
- if image depicts fluorescent materials

In fact, we might want to attenuate large gradients rather than preserve them!

Fattal et al 2002

Gradient domain compression:

- Differentiate density image
- Attenuate large gradients
- Integrate
- Exponentiate

Fattal et al 2002

Human visual system

Sumanta already explained the principles by which the HVS deals with high dynamic range scenery

Here, we build on those insights to classify HVS-based tone reproduction operators

Global vs. Local

We may apply compressive functions directly in image space

- Global: use the same function for each image pixel (use a global adaptation value)
- Local: modulate compressive function by pixel neighborhood (use a local adaptation value)

Global operators

Since each pixel is treated independently:

- Efficient and fast
- Suitable for medium dynamic range imagery

Global operators

Only a small number of different functional shapes are known:

- Linear scaling
- Logarithmic scaling
- Histogram-based compression
- Sigmoidal functions (discussed by Sumant)

Linear scaling

- Already shown an ad-hoc linear scaling function at the start of this section
- Ward 1994 and Ferwerda et al 1996: Linear scaling based on TVI functions
  - Bring the log average luminance to a sensible display level
  - Better than ad-hoc scaling, not suitable for very high dynamic range images

Ward 1994 (linear scaling)

Ferwerda et al 1996


Applied different prescaling factors

Logarithmic scaling

Simple version:

$$L_d = \frac{\log(L_w + 1)}{\log(L_{w,\max} + 1)}$$

Drago et al 2003:

$$L_d = \frac{\log_b(L_w + 1)}{\log_b(L_{w,\max} + 1)}$$

b is a function of the pixel intensity and is modulated by a user parameter to steer overall appearance

Logarithmic compression

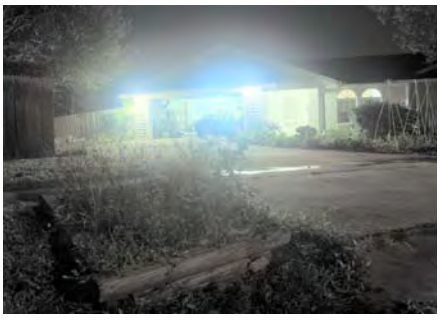
Drago et al 2003

Histogram adjustment

Ward et al (1997) use the shape of the image's histogram

- Compute histogram
- Compute cumulate histogram
- Result is a monotonically increasing function
- Reshape this function to avoid slopes greater than 1
- Remap luminances according to this function

Ward et al 1997



Sigmoids

S-shaped compression function used in a large number of tone reproduction operators.

Discussed in detail by Sumanta in previous section. Will omit discussion here.

Global operators

Recap:

- Easy to implement
- Fast
- Some provide very reasonable amounts of compression and plausible results are frequently obtained
- Useful for medium to high dynamic range images

Local operators

Sometimes the mismatch between HDR image and display device is very large

More compression afforded by local operators

Often at the cost of artifacts

Local operators

Two approaches:

- $L_d(x,y) = s(x,y)L_w(x,y)$
- Replace global adaptation value with a per-pixel local adaptation value

Local operators

- Chiu et al, Rahman et al, iCAM model:

$$L_d(x,y) = s(x,y)L_w(x,y)$$
$$s(x,y) = f\left(\frac{1}{L_w^{\mathrm{LPF}}(x,y)}\right)$$

i.e. divide the image by a low-pass filtered version of itself

Chiu et al 1993

Rahman et al

SIGGRAPH2006

Johnson and Fairchild (iCAM)

SIGGRAPH2006

Local operators

When dividing by a low-pass filtered image, we need two things to minimize artifacts:

- A very large kernel size
- Reduce the weight of the LPF image

Global to local

- Replace global average with local average within global operators
- Sigmoids! General shape:

$$L_d(x,y) = \frac{L_w(x,y)^n}{L_w^{\mathrm{LPF}}(x,y)^n + L_w(x,y)^n}$$

Local adaptation

There is a catch, though:

- The size of the low-pass filter kernel is important.
- Too large and haloing artifacts will occur
- Too small and compression will not be better than global operators

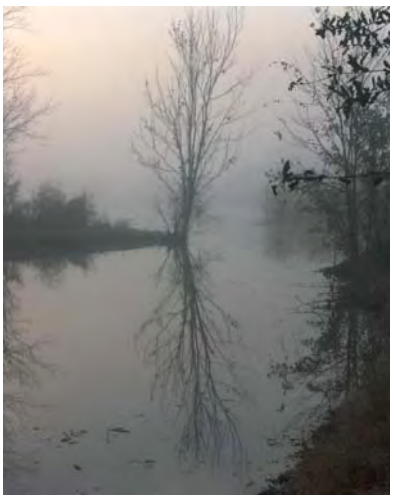
Low-pass filter kernel size

- For each pixel the size of the LPF kernel should be such that it does not overlap with sharp discontinuities
- But we still want to average over the largest spatial area for which the above is true (which may be different for each pixel)
- In practice often a small kernel size!

How to compute?

- Multi-scale techniques using difference of Gaussians and scale selection (Reinhard 2002, Ashikhmin 2002)
- Edge-preserving smoothing operator (bilateral filter, mean shift algorithm, LCIS…)

Reinhard et al 2002



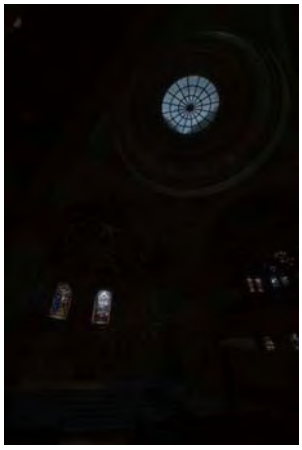
Ashikhmin 2002

Global vs. local

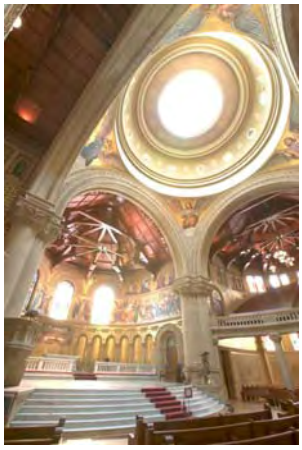
Sigmoid with global adaptation

Sigmoid with local adaptation

Informal comparison

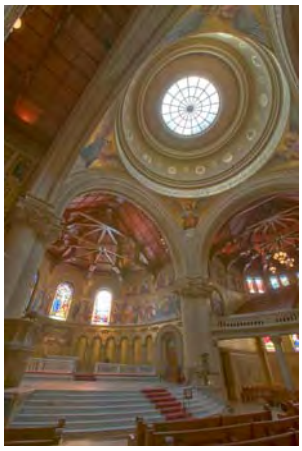


Linear scaling



Linear with clamping

Informal comparison



Bilateral filter

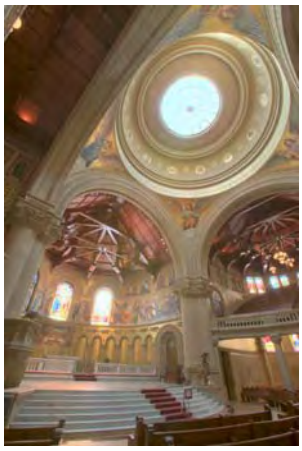
Gradient compression

Informal comparison

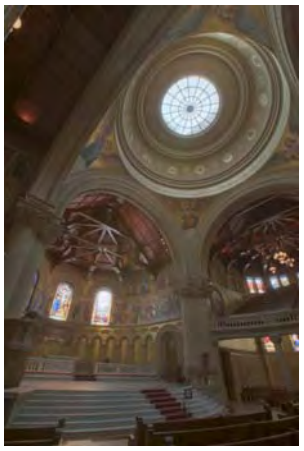
Histogram adjustment

Gradient compression

Informal comparison



Photographic operator



Photoreceptor-based

Informal comparison

Li et al, SIGGRAPH 2005

Conclusions

Only a few fundamentally different approaches to tone reproduction exist

- Based on image formation
  - Frequency domain
  - Gradient domain
- Based on the human visual system
  - Global operators
  - Local operators

Conclusions

Trade-offs exist between:

- Amount of compression
- Presence of artifacts
- Computation time

Validation studies are an important tool to figure out which operator is most suited to which task

HDR Image-Based Lighting

Paul Debevec

Image-Based Lighting

Paul Debevec
University of Southern California
Institute for Creative Technologies
Graphics Laboratory

SIGGRAPH 2005 Course #29
High-Dynamic-Range Imaging and Image-Based Lighting
Half-Day, Tuesday, 2 August, 8:30 am - 12:15 pm

www.debevec.org

"A McIntosh in the Kitchen" by Marc Jacquier (2004)


Nick Bertke
pogo767@gmail.com
Image courtesy of Nick Bertke

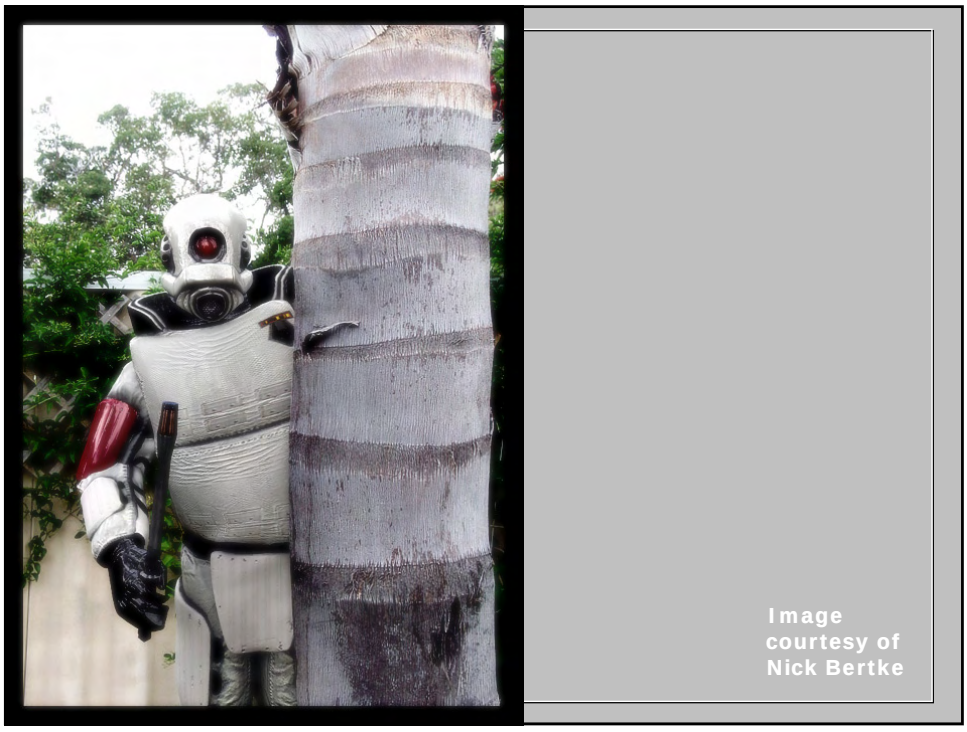


Image courtesy of Nick Bertke


Nick Bertke
pogo767@gmail.com


Canon Powershot G2
f/8, 1 sec


Canon Powershot G2
f/8, 1/2 sec


Canon Powershot G2
f/8, 1/5 sec

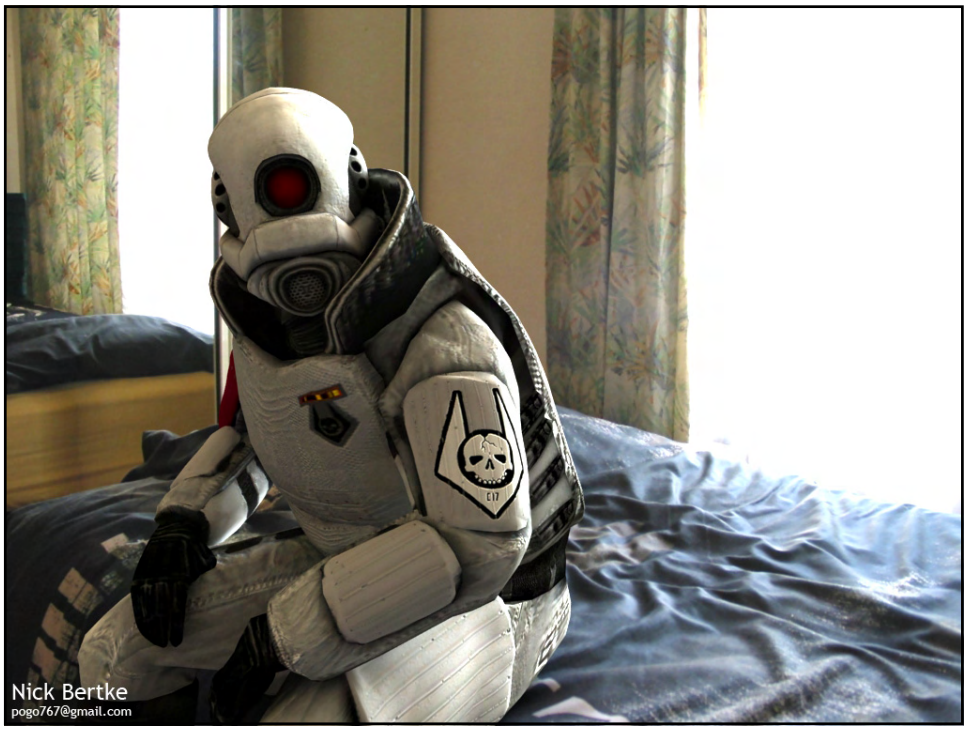

Canon Powershot G2
f/8, 1/13 sec


Canon Powershot G2
f/8, 1/30 sec


Canon Powershot G2
f/8, 1/80 sec


Canon Powershot G2
f/8, 1/200 sec

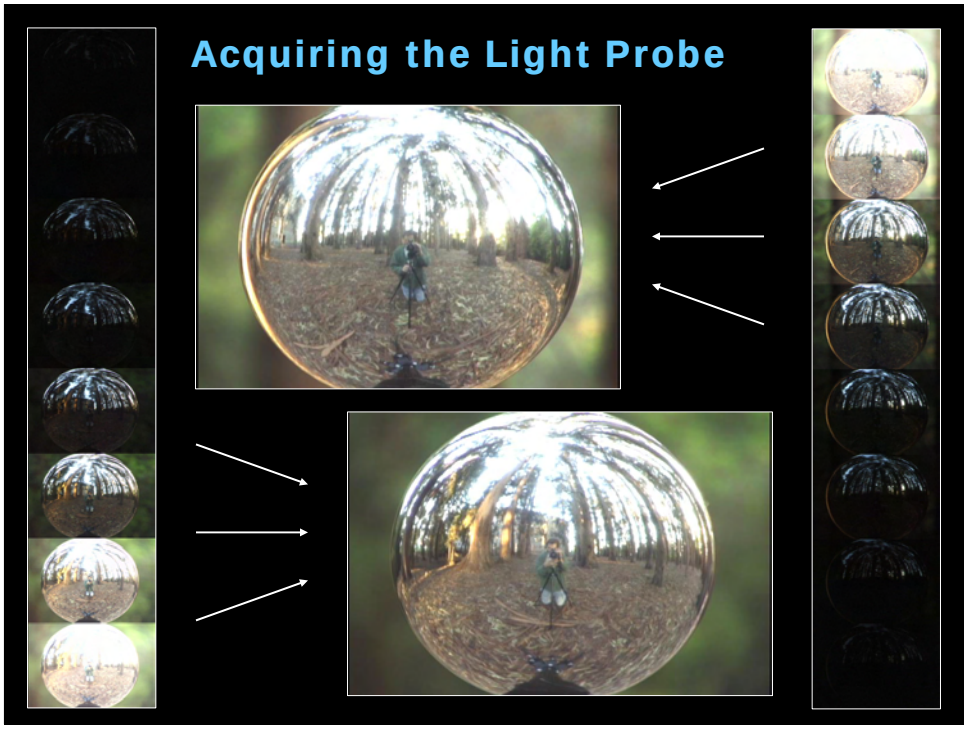

Nick Bertke


Nick Bertke
pogo767@gmail.com

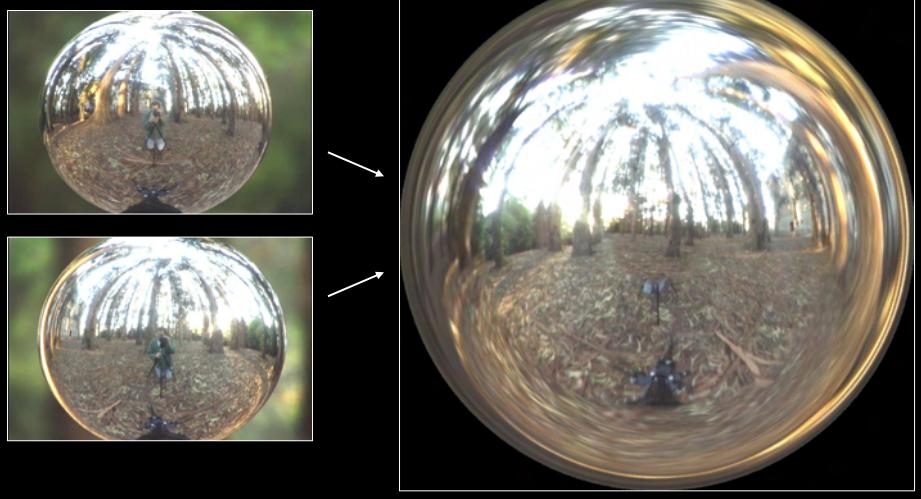

Nick Bertke
pogo767@gmail.com


Nick Bertke
pogo767@gmail.com

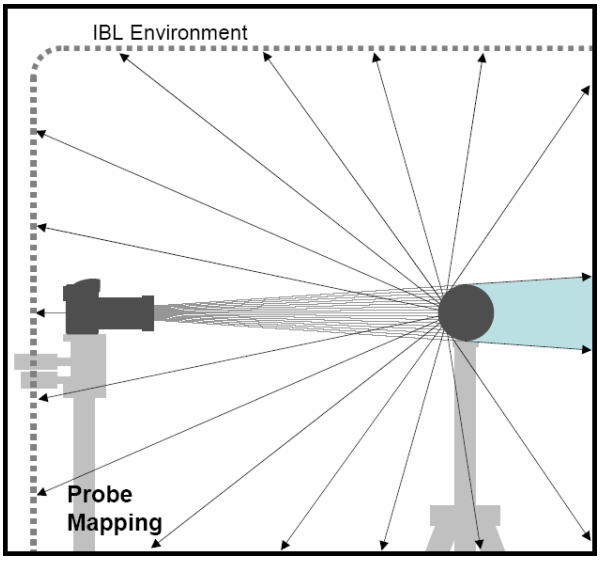

Rendering with Natural Light
SIGGRAPH 98 Electronic Theater

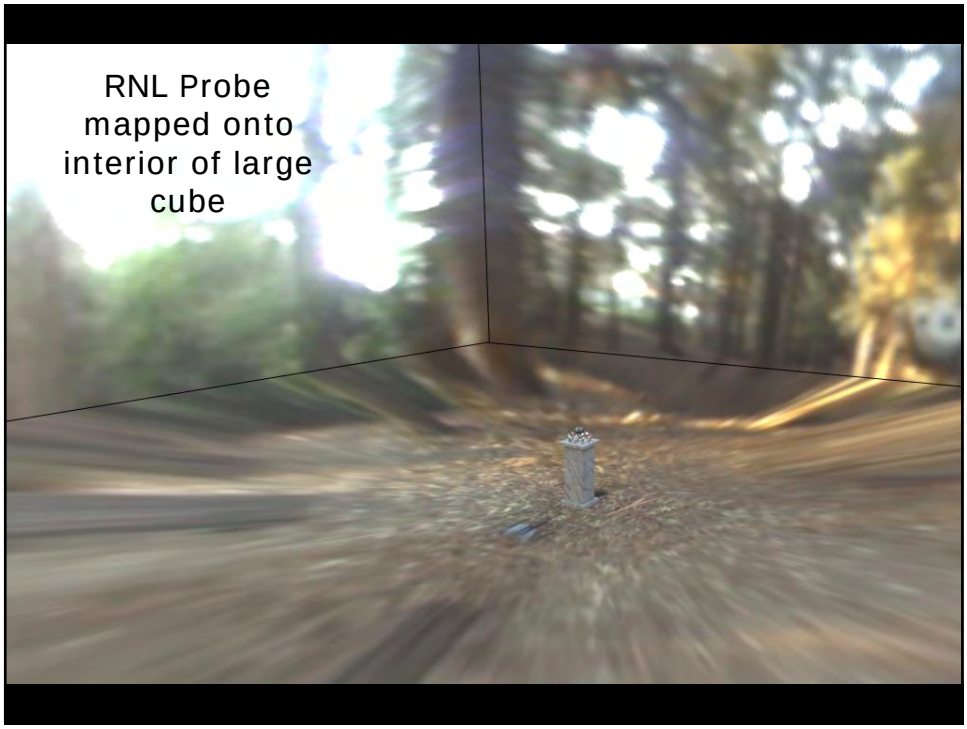

Acquiring the Light Probe

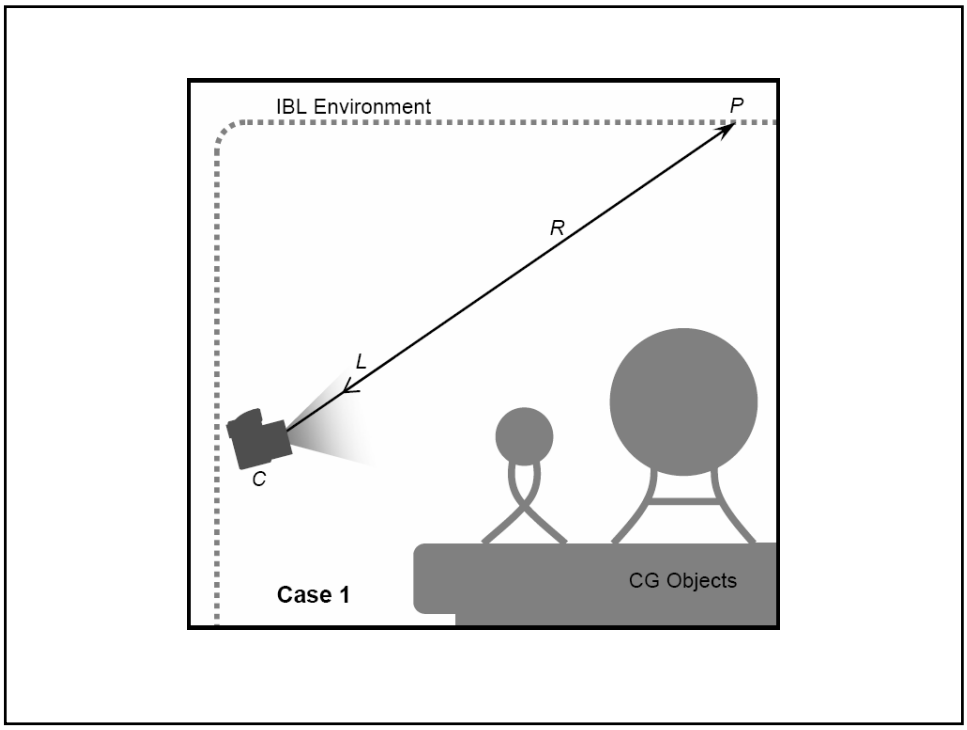
Assembling the Light Probe

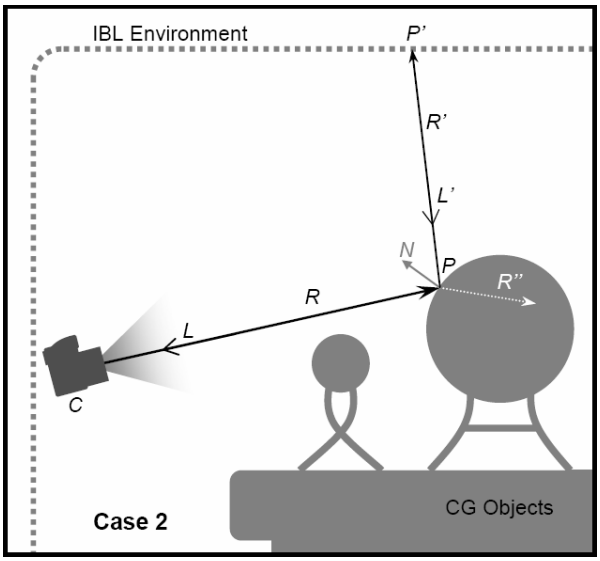


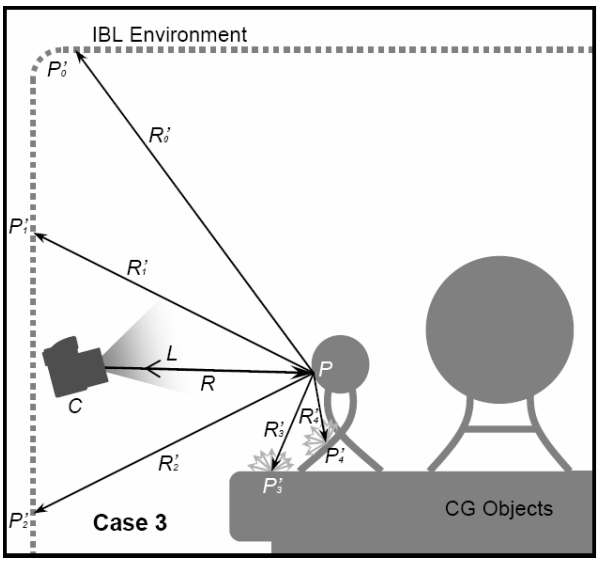
See HDRShop Tutorial #5, www.hdrshop.com



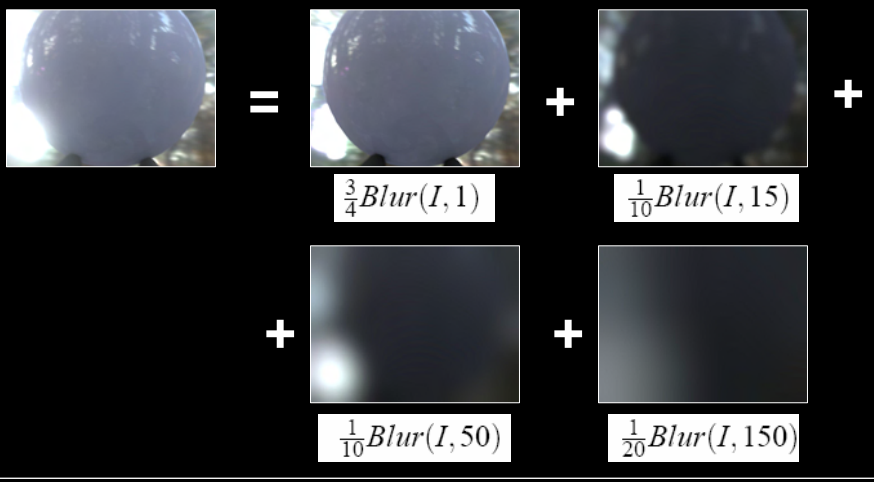

RNL Probe
mapped onto
interior of large
cube


IBL Environment
P
R
L
C
Case 1
CG Objects


IBL Environment
P'
R'
L'
N
P
R''
R
L
C
Case 2
CG Objects

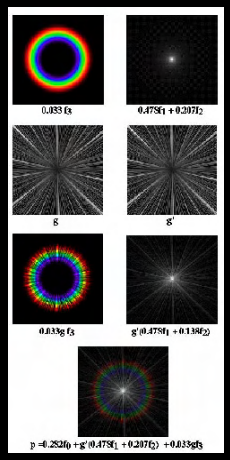

IBL Environment
P'0
R'0
P'1
R'1
L
R
C
P
R'3
R'4
P'4
R'2
P'3
P'2
Case 3
CG Objects

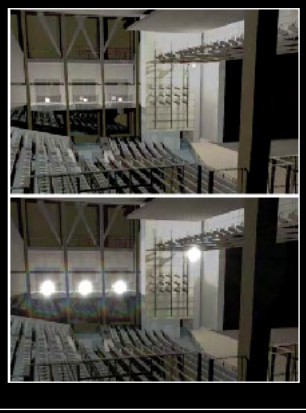
Summing different blur widths



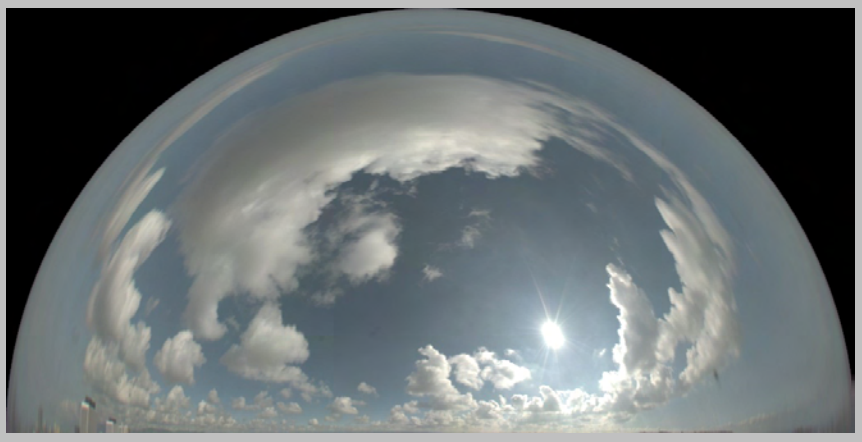
Simulating the Glare in the Human Eye

Greg Spencer, Peter Shirley, Kurt Zimmerman, and Donald Greenberg. Physically-based glare effects for digital images. SIGGRAPH 95.





The Sampling Problem



HDRI Sky Probe from http://www.ict.usc.edu/graphics/skyprobes/

Lit by sun and sky

Rendered in Marcos Fajardo's "Arnold" renderer

9 samples per pixel, 17 min.

Rendered in Marcos Fajardo's "Arnold" renderer

9 samples per pixel, 17 min.

Rendered in Marcos Fajardo's "Arnold" renderer

16 samples per pixel, 46 min.

Rendered in Marcos Fajardo's "Arnold" renderer

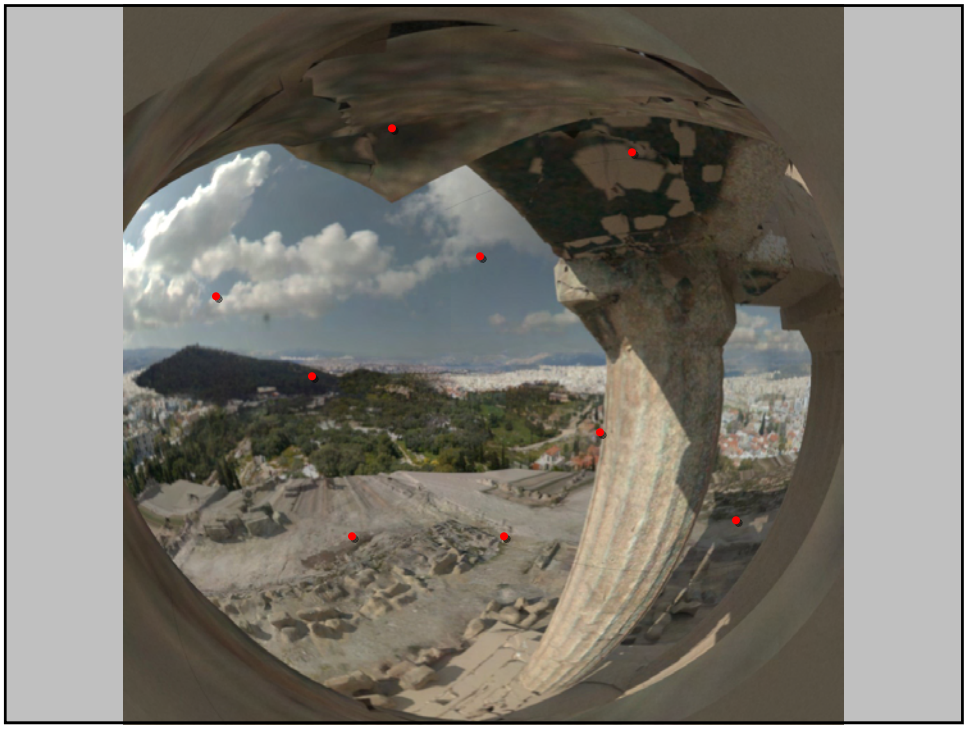
100 samples per pixel, 189 min.

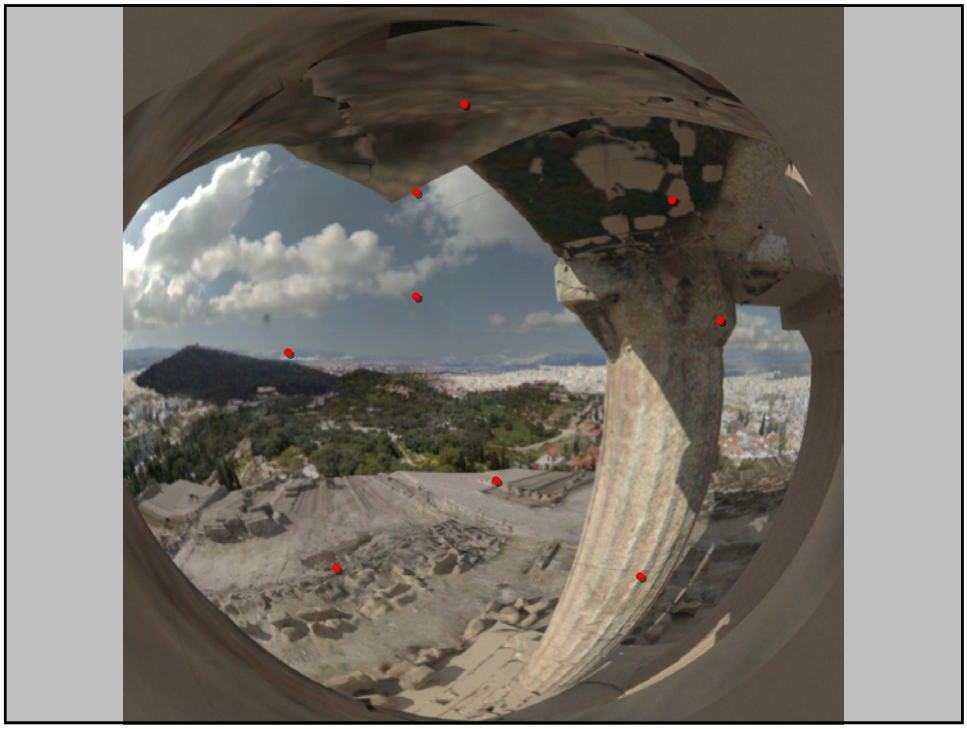
Rendered in Marcos Fajardo's "Arnold" renderer

A sunlit sample point

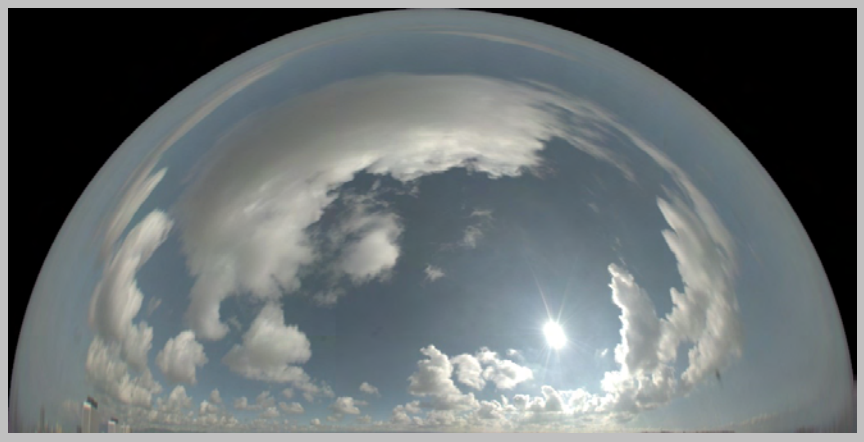
Rendered in Marcos Fajardo's "Arnold" renderer

A shadowed sample point

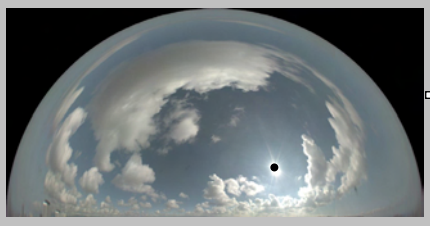


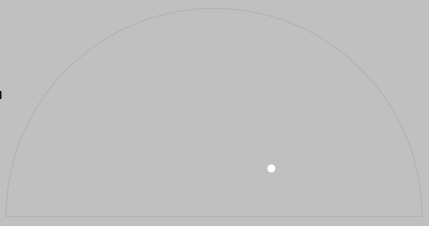


HDRI Sky Probe



Clipped Sky + Sun Source





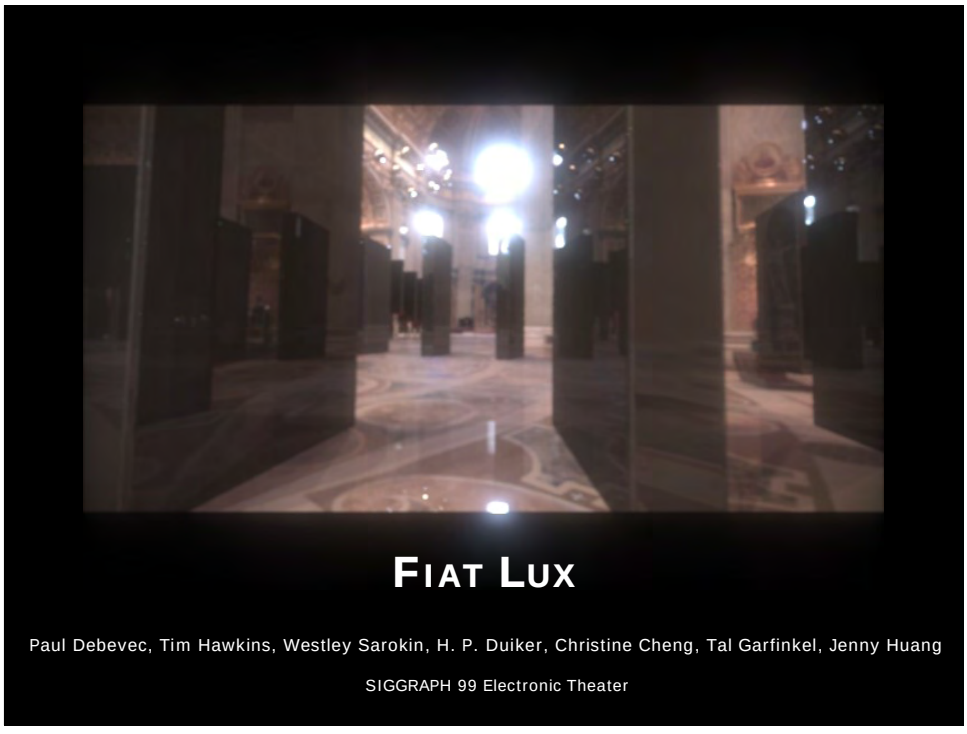
```
texture_map _HDRI_probe_map_clip_
"probe_09-55_clipped.hdr" {
swrap periodic
 filter bilinear
}

shader _HDRI_clip sky_HDRI {
 HDRI_map "_HDRI_probe_map_clip_"
 multiplier 1.000000
}
```

```
light sun directional 1 {
 direction -0.711743 -0.580805 -0.395078
 angle 0.532300
 color 10960000 10280000 866000
}
```


Lit by sky only, 17 min.

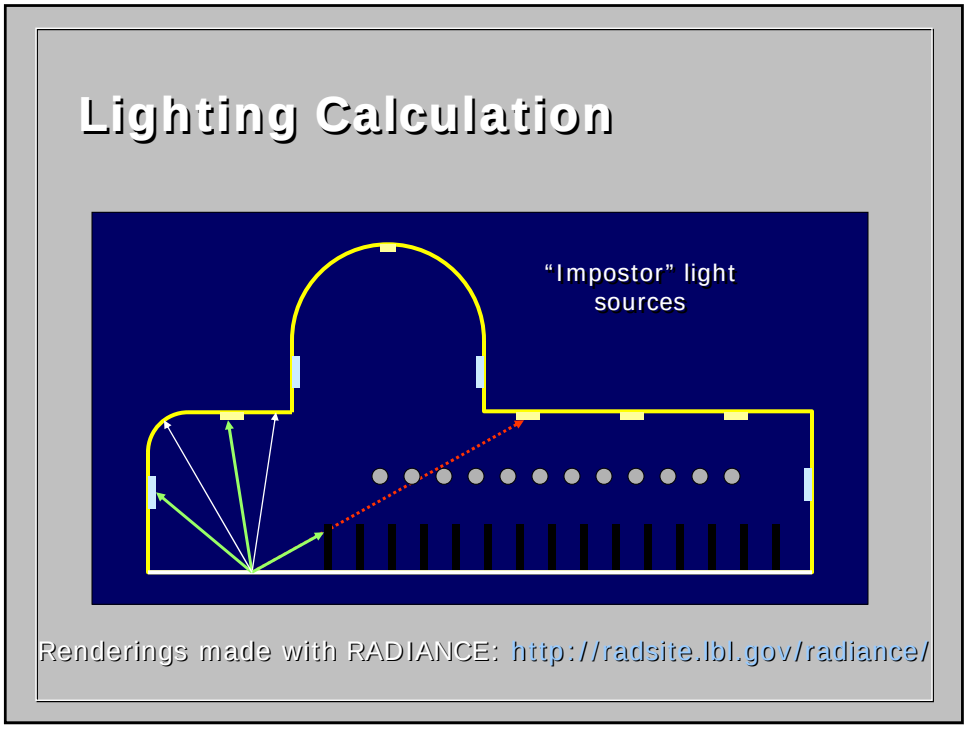

Lit by sun only, 21 min.


Lit by sun and sky, 25 min

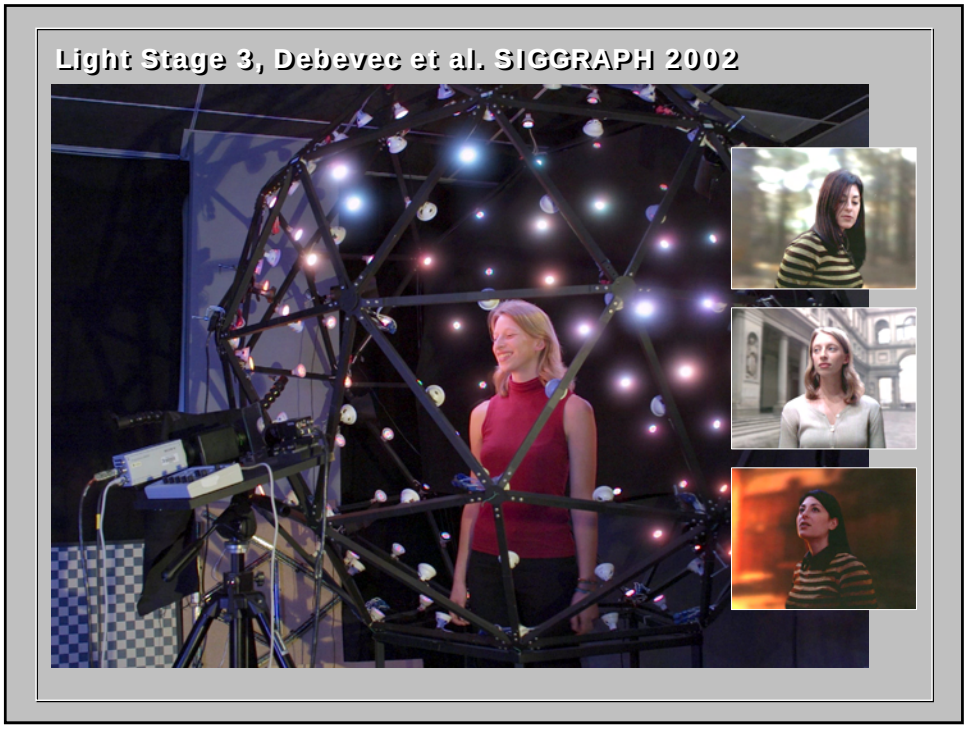

FIAT LUX
Paul Debevec, Tim Hawkins, Westley Sarokin, H. P. Duiker, Christine Cheng, Tal Garfinkel, Jenny Huang
SIGGRAPH 99 Electronic Theater

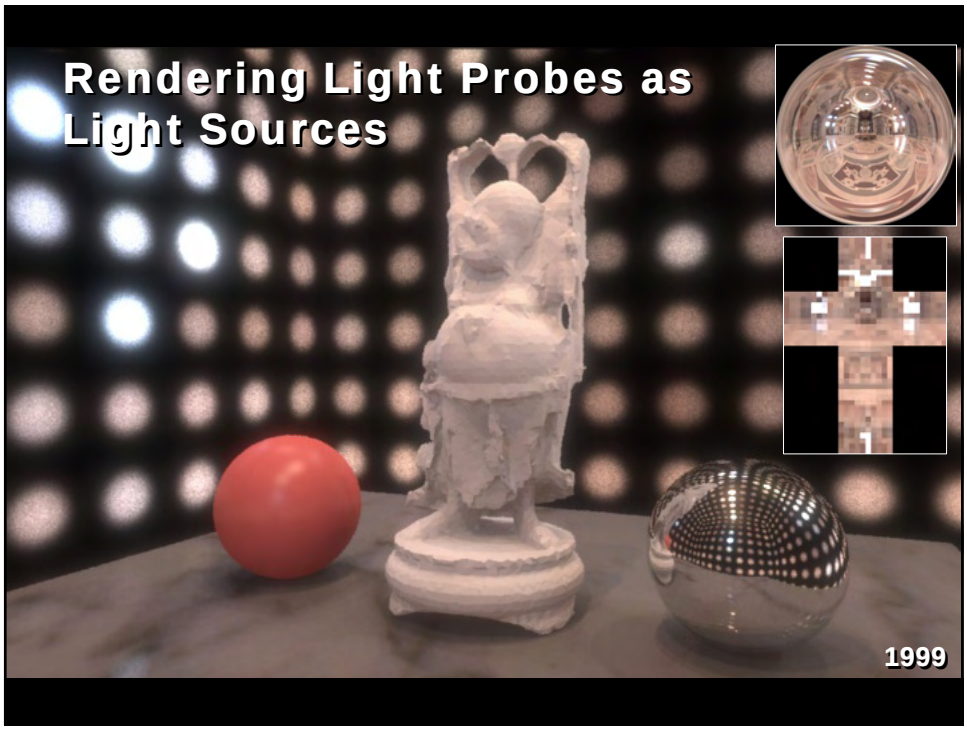
Assembled Panorama

Identified Light Sources

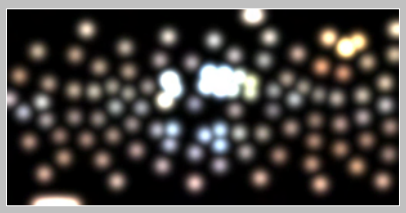

Lighting Calculation
"Impostor" light sources
Renderings made with RADIANCE: http://radsite.lbl.gov/radiance/


Rendering Light Probes as Light Sources
1999


Light Stage 3, Debevec et al. SIGGRAPH 2002

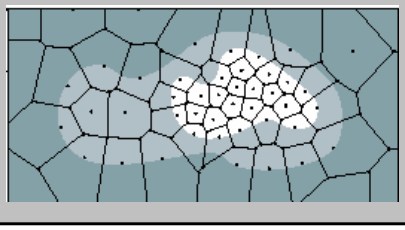

Rendering Light Probes as Light Sources
1999

Cohen and Debevec
"LightGen" HDR Shop Plugin, 2001
www.debevec.org/HDRShop/

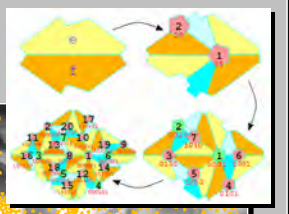


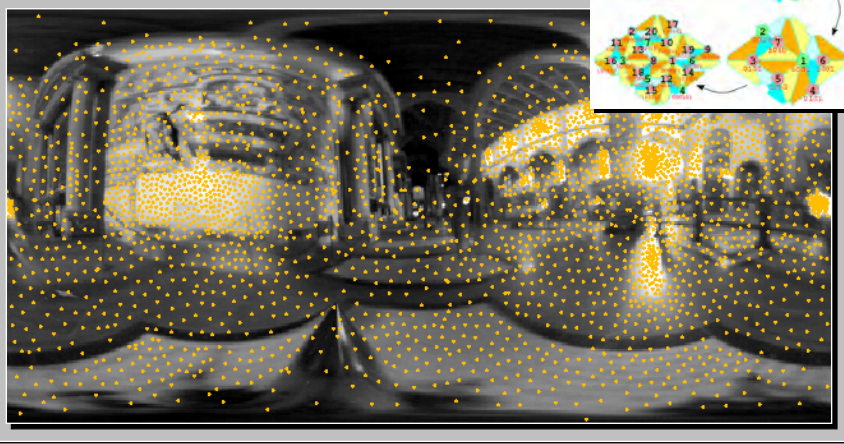
Kollig and Keller. *Efficient illumination by high dynamic range images*. Eurographics Symposium on Rendering 2003.

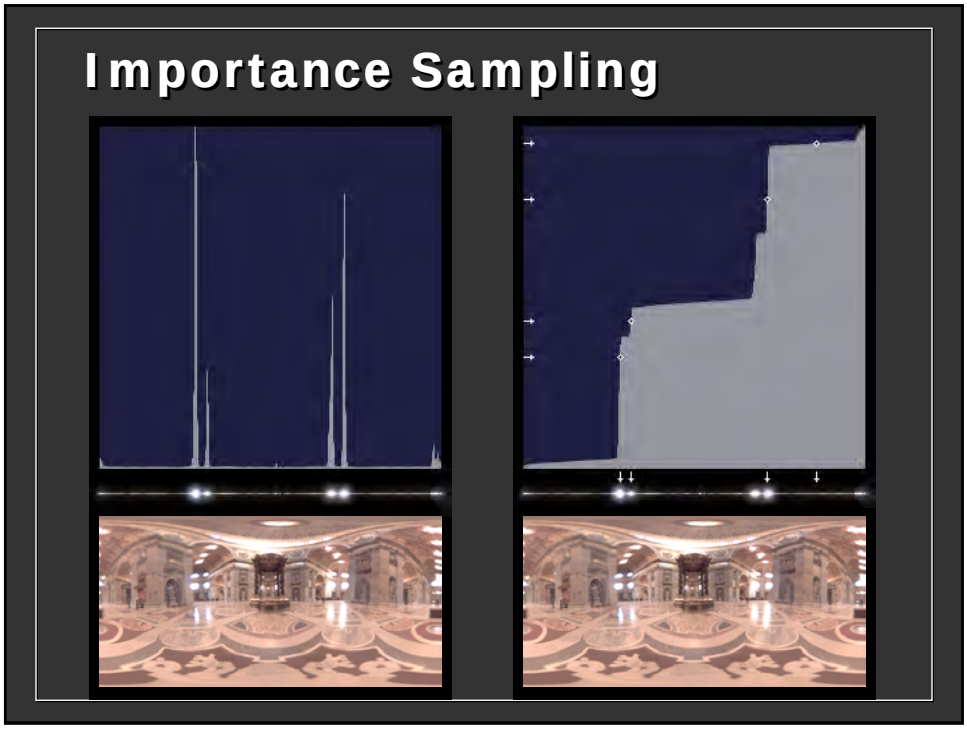
Agarwal, Ramamoorthi, Belongie, and Jensen, *Structured Importance Sampling of Environment Maps*, SIGGRAPH 2003

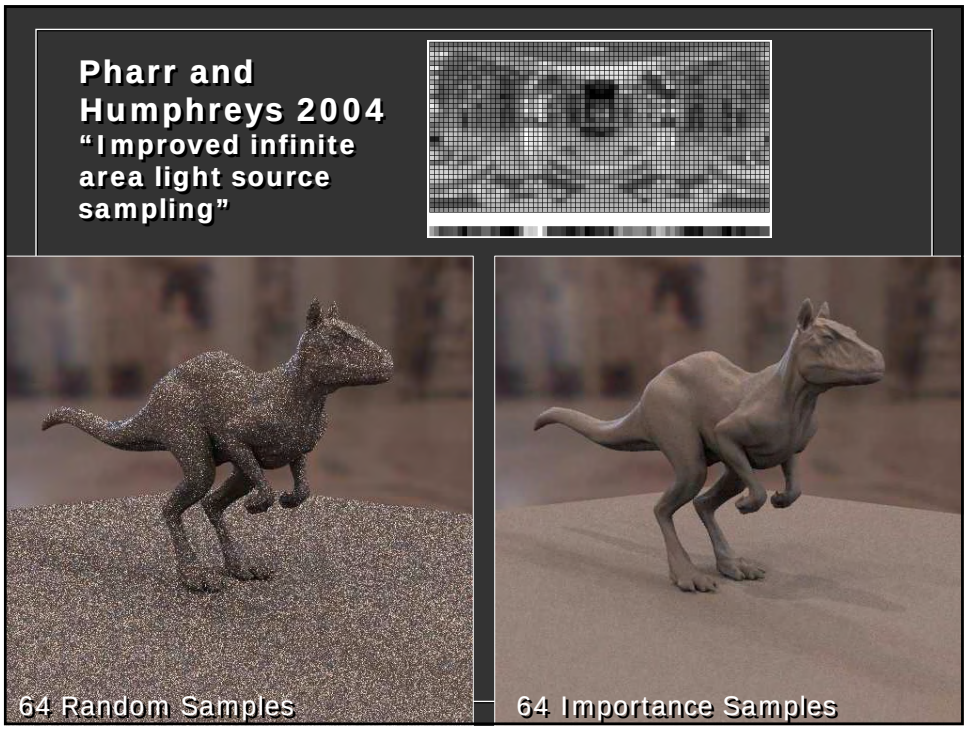


Ostromoukhov, Donohue, and Jodoin
Fast Hierarchical Importance Sampling with Blue Noise Properties
Proc. SIGGRAPH 2004






Importance Sampling


Pharr and Humphreys 2004
"Improved infinite area light source sampling"
64 Random Samples
64 Importance Samples

A Median Cut Algorithm for Light Source Placement

Inspired by Heckbert "Color Image Quantization for Frame Buffer Display", SIGGRAPH 82

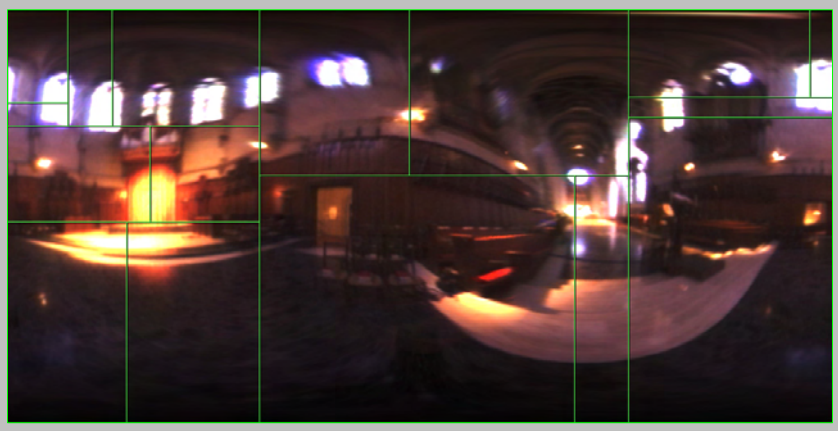
1. Add the entire light probe image to the region list as a single region
2. For each region, subdivide along the longest dimension such that its light energy is divided evenly
3. If the number of iterations is less than *n*, return to step 2.

Paul Debevec, SIGGRAPH 2005 Poster

Median Cut Algorithm

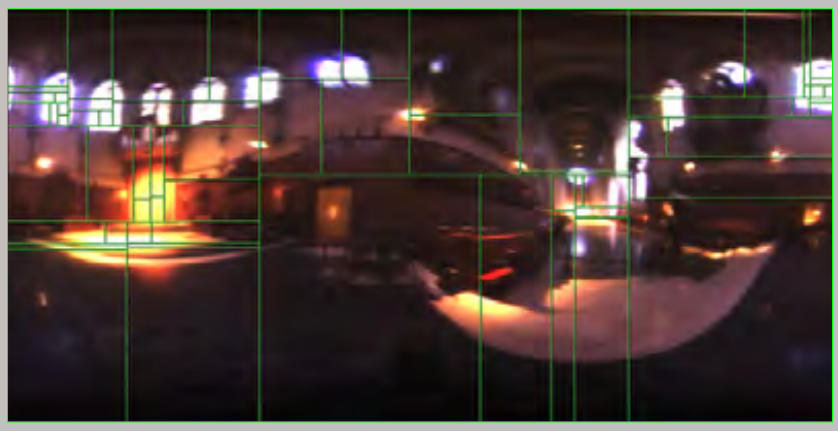
4 regions

Median Cut Algorithm



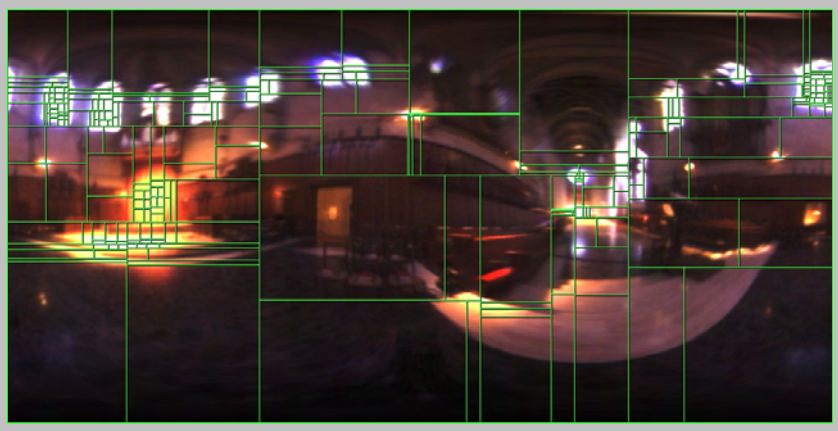
16 regions

Median Cut Algorithm



64 regions

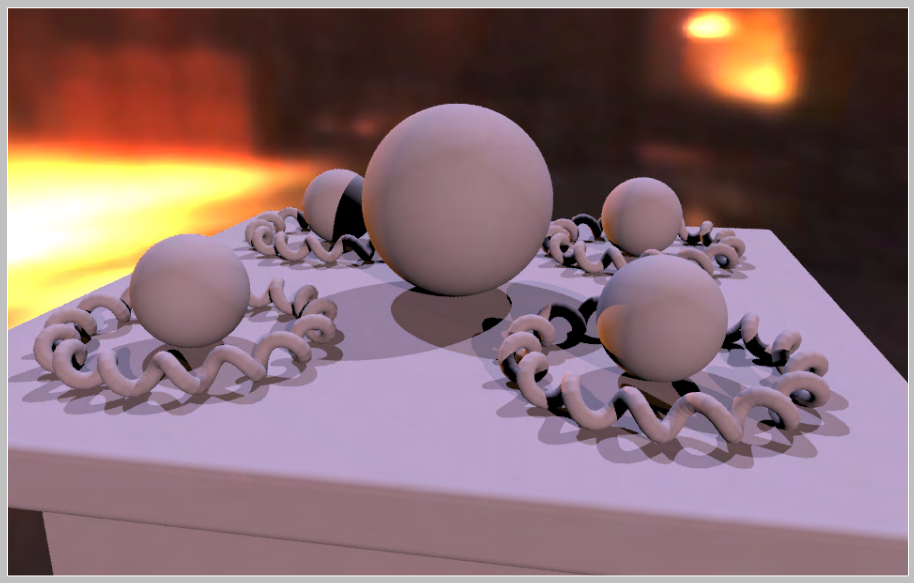
Median Cut Algorithm



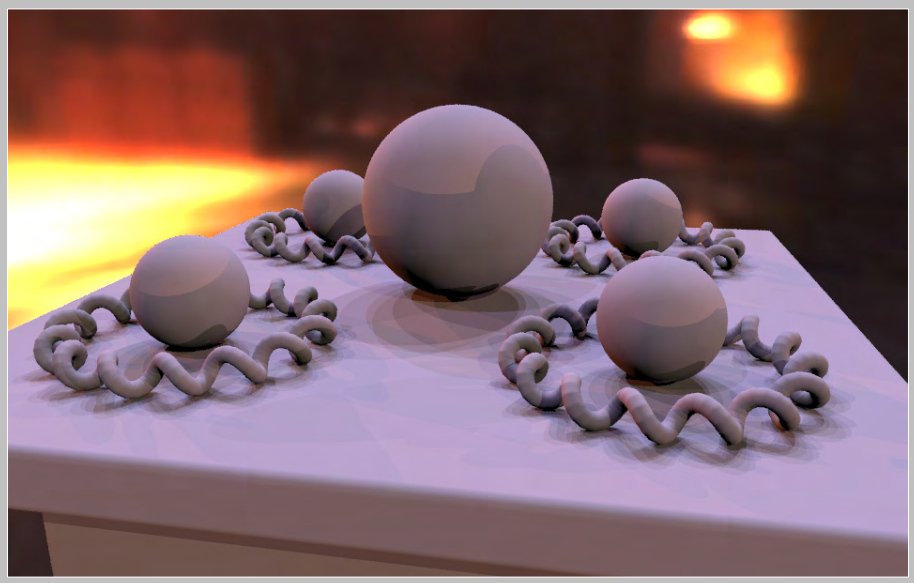
256 regions

Final Light Sources

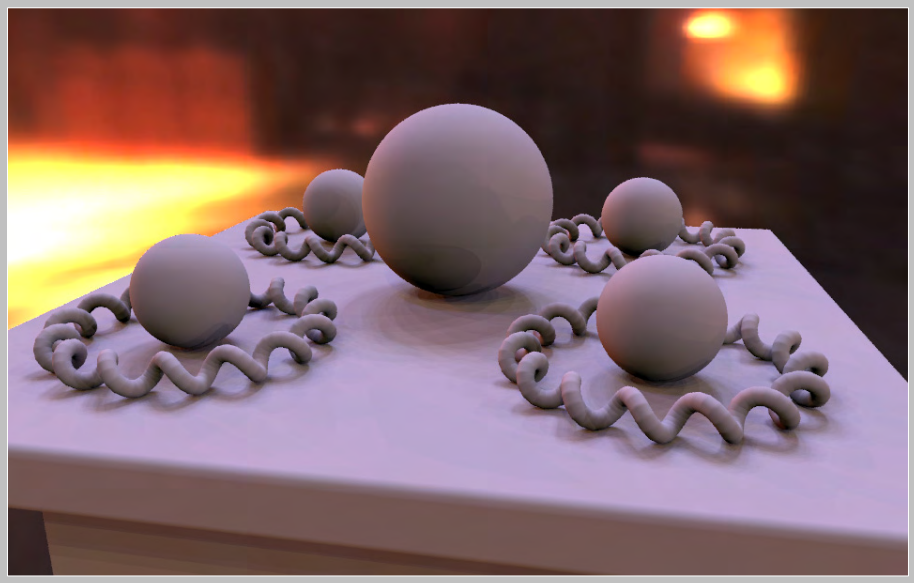
256 lights



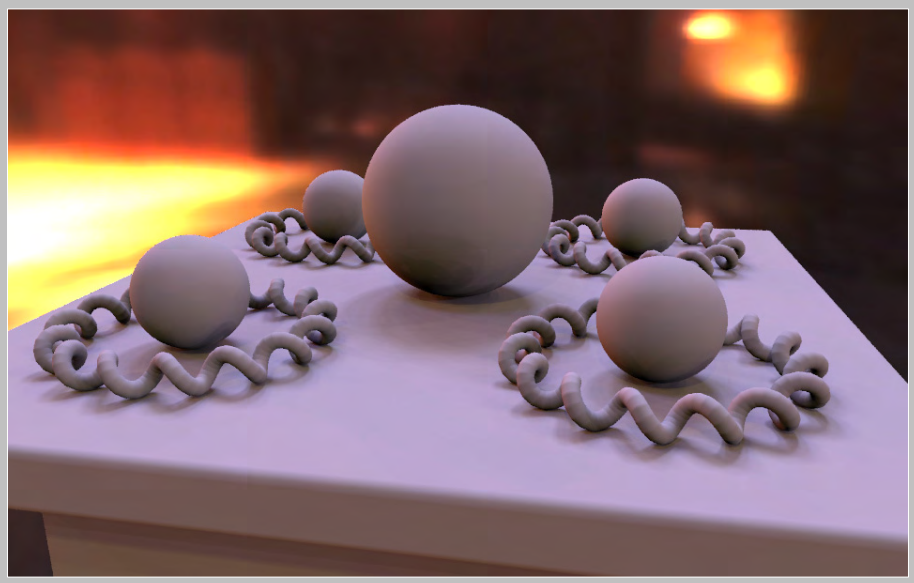
4 lights



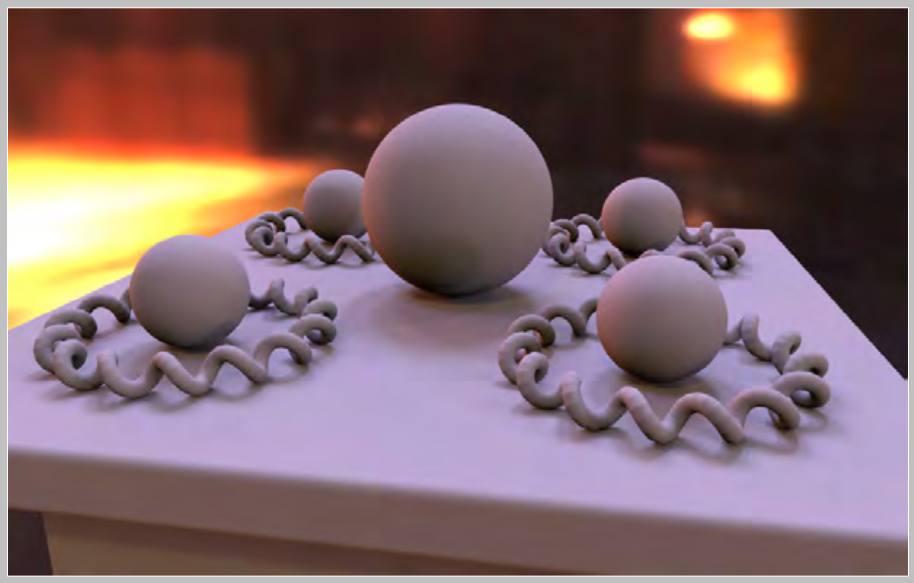
16 lights



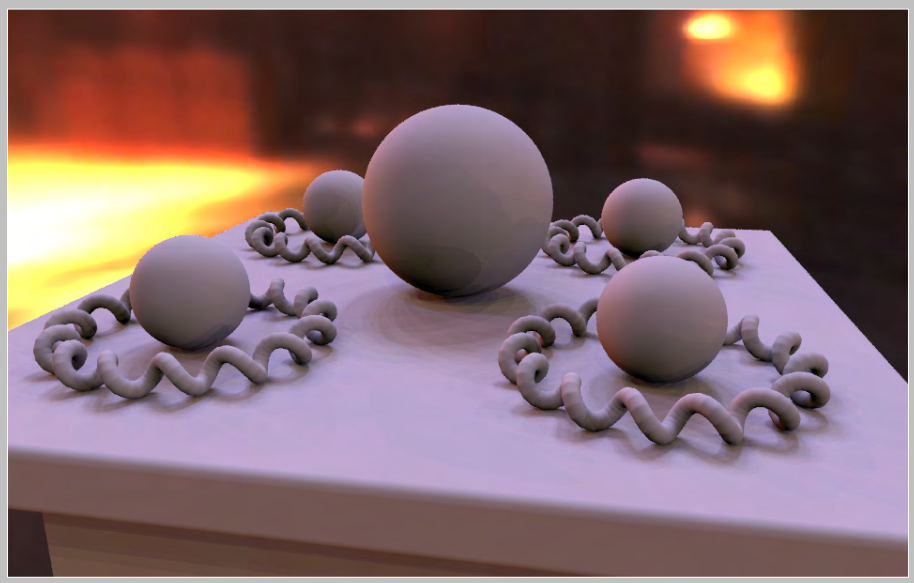
64 lights



256 lights



4096 random samples



64 lights

Implementation Details

- Perform energy split decisions on a grayscale version of the image
- Can speed up region sums using summed area tables (Frank Crow, *Summed Area Tables for Texture Mapping*, SIGGRAPH 84)
- Must take into account area stretching near poles => multiply intensity and area widths by cos ϕ

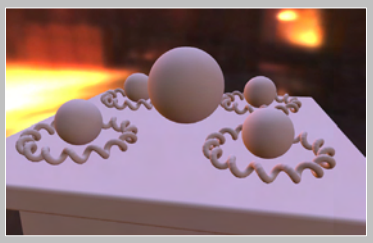
ϕ=–90°

ϕ=0

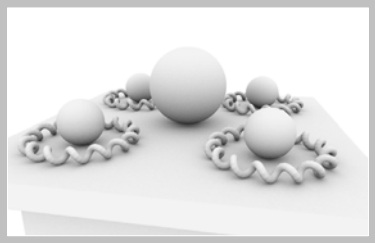
ϕ=90°

Ambient Occlusion

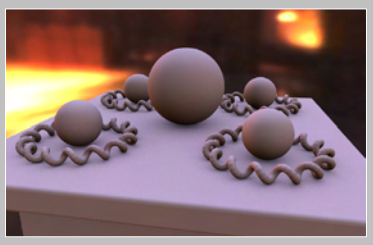
Landis, McGaugh, and Koch; Landis et al SIGGRAPH 2002



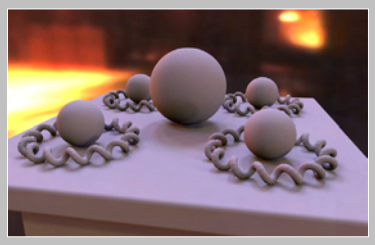
(a) diffuse env mapping



(b) ambient occlusion map

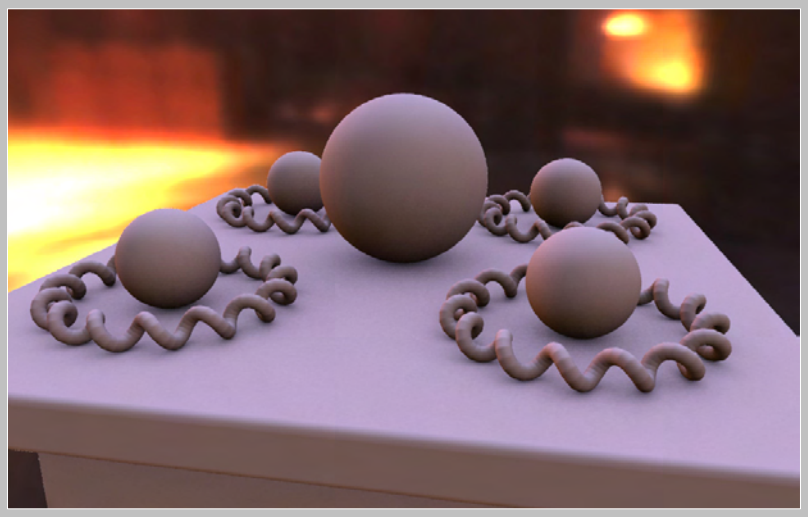


(a) * (b)



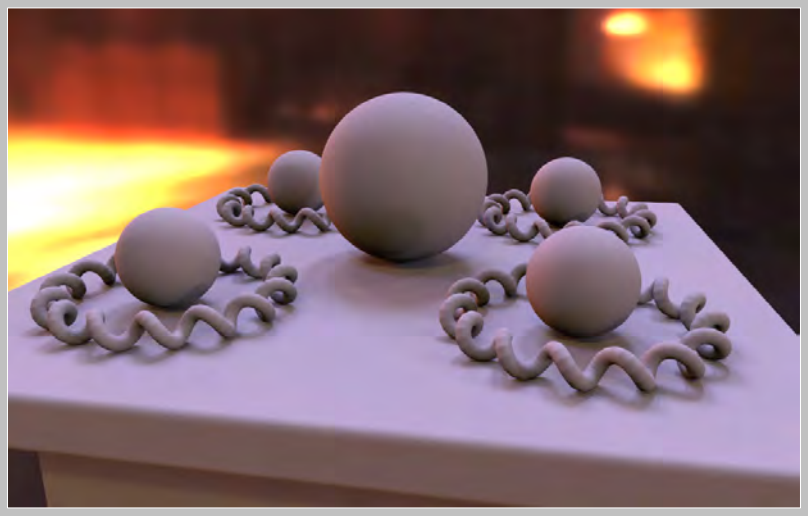
Full IBL

Ambient Occlusion

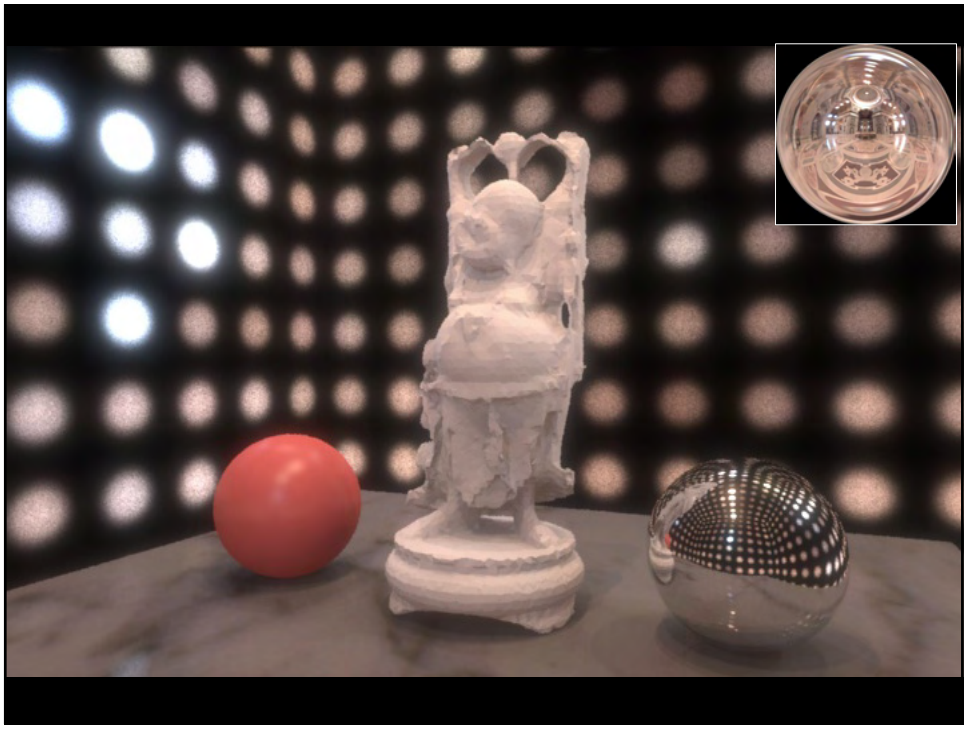


Ambient Occlusion Approximation

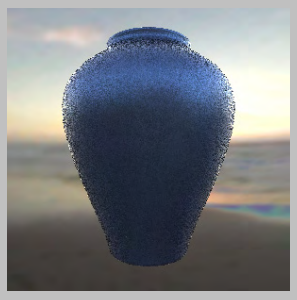
Ambient Occlusion



Full IBL



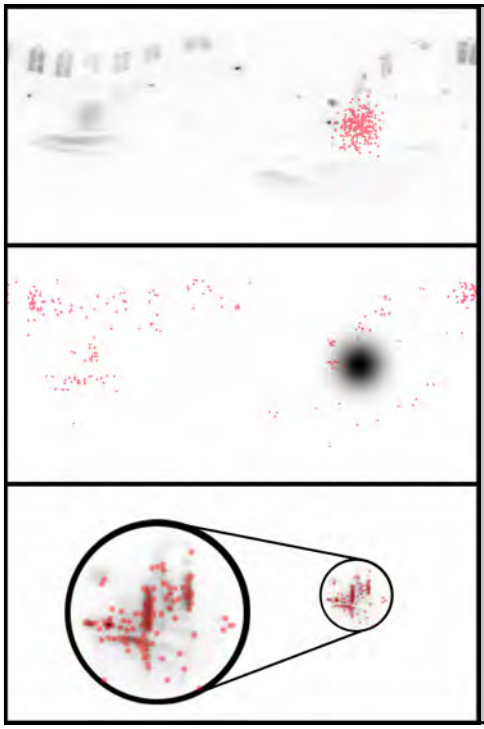
Sampling according to the BRDF



75 rays, uniform sampling

75 rays, factored BRDF sampling

From LAWRENCE, J., RUSINKIEWICZ, S., AND RAMAMOORTHI, R. 2004. Efficient BRDF importance sampling using a factored representation. *SIGGRAPH 2004*



David Burke, Abhijeet Ghosh and Wolfgang Heidrich. *Bidirectional Importance Sampling for Direct Illumination*. EGSR2005.

Uses *Structured Importance Resampling (SIR)*

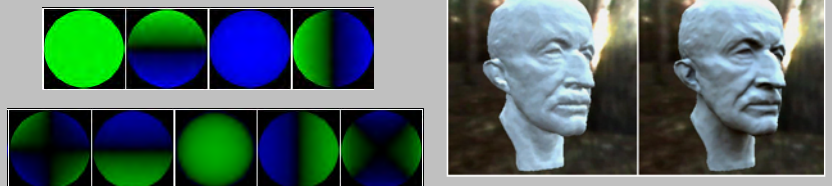
See Also:

Talbot et al. Importance Resampling for Global Illumination, EGSR2005

Lawrence et al, *Adaptive Numerical Cumulative Distribution Functions for Efficient Importance Sampling*. EGSR2005.

Ghosh and Heidrich. *Correlated Visibility Sampling for Direct Illumination*. SIGGRAPH 2005 Sketch.

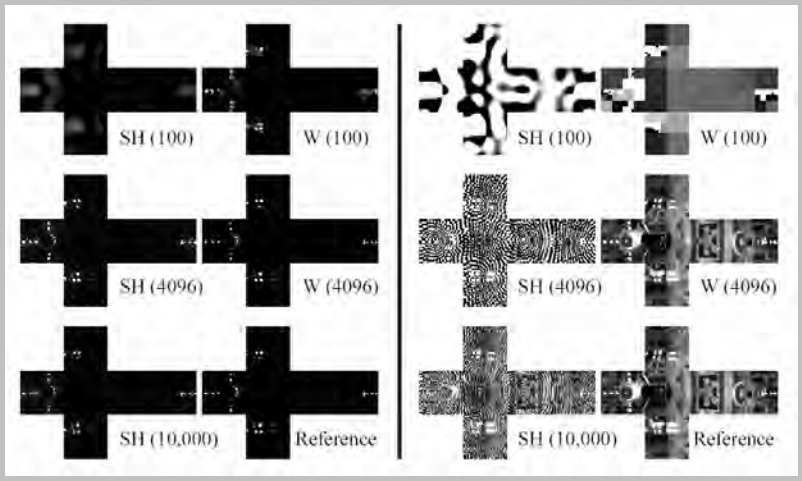
Real-Time IBL with Spherical Harmonics



Frequency Space Environment Map Rendering
Ravi Ramamoorthi, Pat Hanrahan, SIGGRAPH 2002

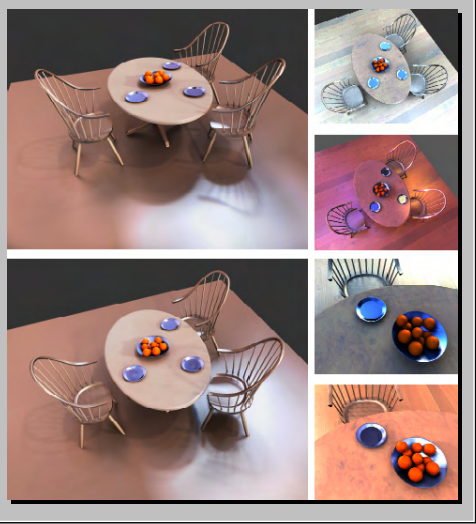
Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments
Peter-Pike Sloan, Jan Kautz, John Snyder, SIGGRAPH 2002

Ng, Ramamoorthi and Hanrahan
All-frequency shadows using non-linear wavelet lighting approximation
SIGGRAPH 2003

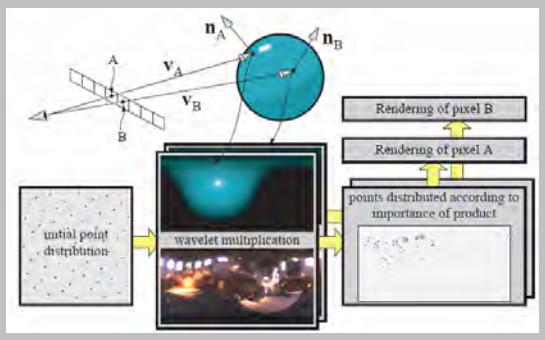


Ng, Ramamoorthi and Hanrahan
Triple Product Wavelet Integrals for All-Frequency Relighting
SIGGRAPH 2004

- **Lighting**
- **Visibility**
- **BRDF**



Wavelet Importance Sampling



Clarberg, Jarosz, Akenine-Möller, and Jensen. Wavelet Importance Sampling: Efficiently Evaluating Products of Complex Functions. SIGGRAPH 2005.

Summary

- IBL lights objects with images of light from the real world
- Techniques such as light source identification, light source constellations, ambient occlusion, and importance sampling allow for efficient and useful approximations
- Sampling according to both the environment and the reflectance properties is the most efficient

Thanks!

Nick Bertke, Marc Jacquier
Greg Ward, Marcos Fajardo
Henrik Wann Jensen, UCSD
Hiro Matsuguma, Naomi Dennis, Hadi Ogawa, Toppan Printing
UC Berkeley: Jitendra Malik, Larry Rowe, Westley Sarokin, HP Duiker
ICT Graphics Lab: Chris Tchou, Brian Emerson, Marc Brownlow, Tim Hawkins, Andreas Wenger, Andrew Gardner, Andrew Jones, Jonas Unger, Frederik Gorranson, John Lai, Tom Pereira, Laurie Swanson
ICT Sponsors: USC Office of the Provost, RDECOM, TOPPAN Printing Co, Ltd.
Bill Swartout, David Wertheimer, Dell Lunceford, USC ICT
Randy Hall, Max Nikias, USC
Course Co-Organizer and speaker: Erik Reinhard, Greg Ward

gl.ict.usc.edu www.debevec.org

HDR Display Devices

Helge Seetzen

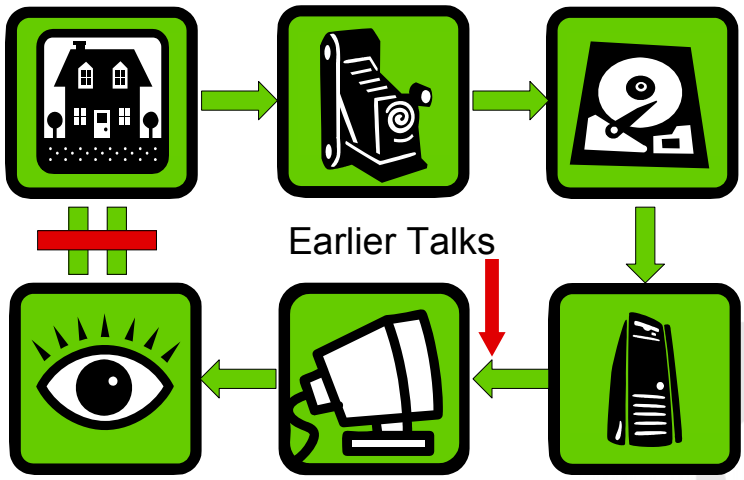
BRIGHTSIDE™

HDR Display Overview

August 2006

BRIGHTSIDE™

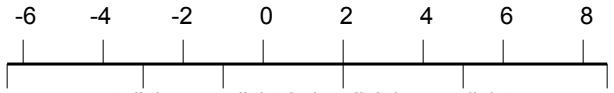
Imaging Pipeline – Low dynamic range rules!



BRIGHTSIDE™

What are we missing?

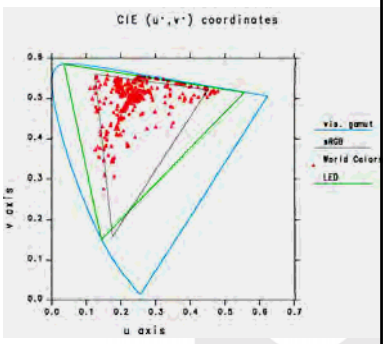
Human Overall Luminance Vision Range
(14 orders of magnitude, scale in log cd/m2)



Human Simultaneous Luminance Vision Range — 5 orders of magnitude

Today's Devices — 2-3 orders

BrightSide Technologies — 5 orders of magnitude



4/29/2006 ©2005 Brightside Technologies Inc. Proprietary & Confidential 3

BRIGHTSIDE™

Designing Commercial HDR Devices

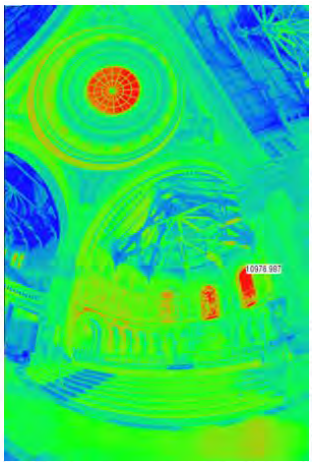
Requirements:

1. High Dynamic Range
2. Compatibility
   - New devices need to function in 8-bit environment and still deliver significant benefit
   - New devices need to be usable in stand-alone mode
3. Cost
   - Ideally no extra cost compared to 8-bit devices
   - If extra cost is necessary then in line with benefit

4/29/2006 ©2005 Brightside Technologies Inc. Proprietary & Confidential 4

BRIGHTSIDE™

Limits of Conventional Output Devices



Stanford Memorial Church
Courtesy Paul Debevec

BRIGHTSIDE™

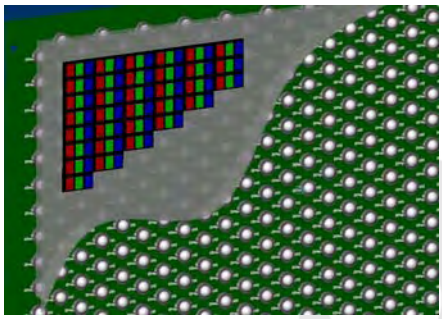
Display Technology – Conventional LCD's

- LCD backlight is provided by CCFL tubes (fluorescent light)
- Light is spread evenly behind LCD panel and does not vary with image content
- Image control is limited to 8 bit single to Red, Green and Blue colour channels (255 steps of control)

BRIGHTSIDE™

Display Technology – HDR Display

- LCD backlight is provided by an array of LED's
- Each LED is controlled with 8 bit (255 step) signal
  - Brightness is adjusted to level demanded by source image
- LCD panel provides additional 8 bits of brightness control
- LED and LCD panel combine optically to deliver 16 bit performance
- LED's provide greater brightness



BRIGHTSIDE™

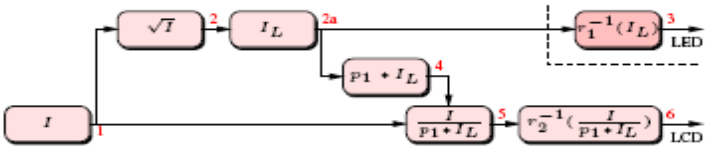
Display Technology – HDR Display

High resolution colour LCD | High Dynamic Range Display | Low resolution Individually Modulated LED array

- Dual modulation
- Low / high resolution and correction
- Veiling luminance
- Implementation in display and projection

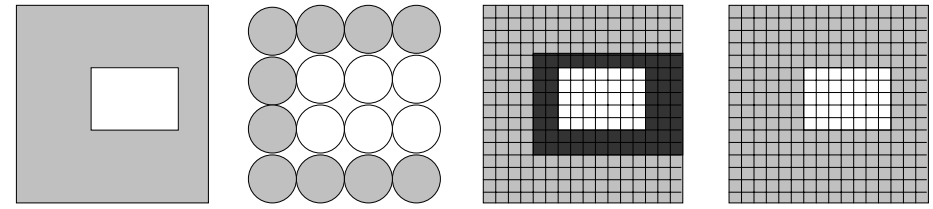
BRIGHTSIDE™

Image Processing Algorithm

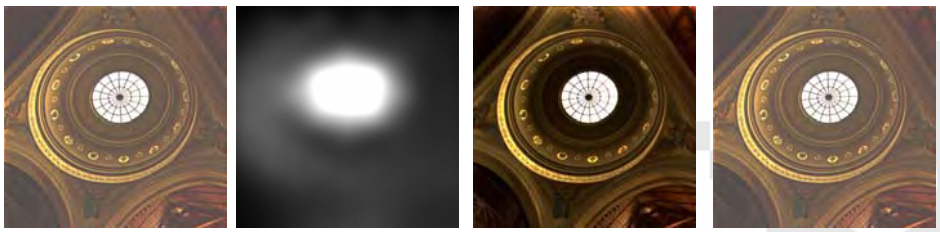


BRIGHTSIDE™

Image Processing Algorithm

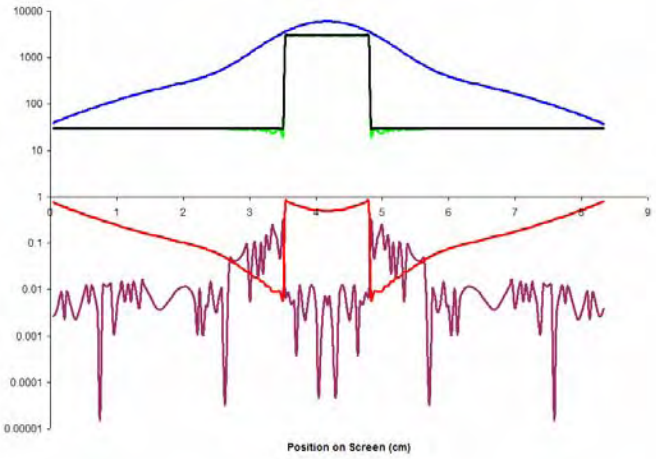


HDR Image | LED array | LCD with correction | Output image



BRIGHTSIDE™

Image Processing Algorithm - Easy



Receive Image

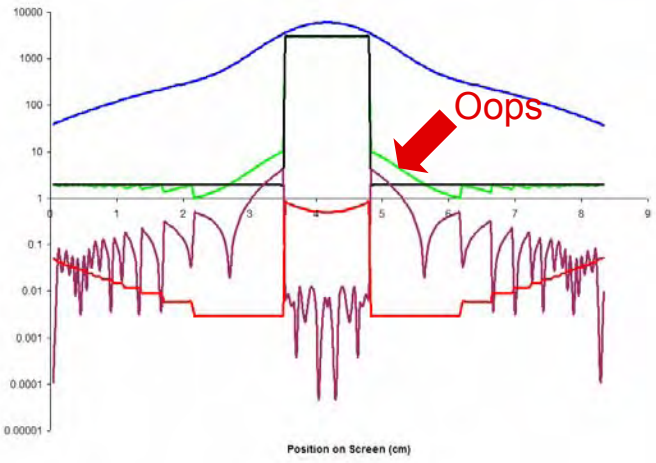
Drive LED

Divide Image by LED light field to obtain LCD values

Output Luminance is the product of LED light field and LCD transmission (modest error)

BRIGHTSIDE™

Image Processing Algorithm - Hard



Receive Image

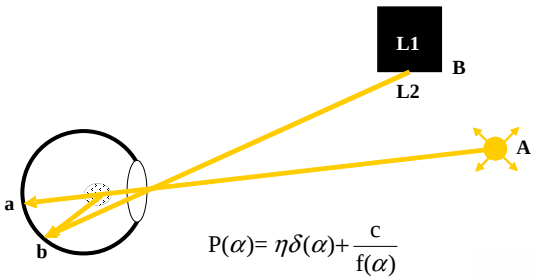
Drive LED

Divide Image by LED light field to obtain LCD values

Output Luminance is the product of LED light field and LCD transmission (Problematic error)

BRIGHTSIDE™

Veiling Luminance

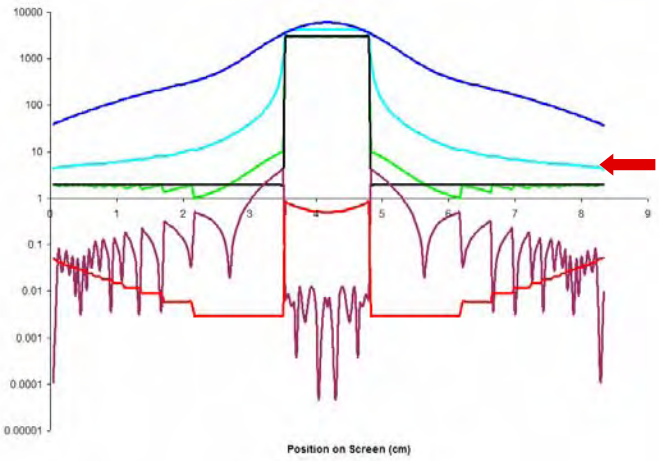


$$P(\alpha) = \eta\delta(\alpha) + \frac{C}{f(\alpha)}$$

4/29/2006 ©2005 Brightside Technologies Inc. Proprietary & Confidential 13

BRIGHTSIDE™

Image Processing Algorithm - Hard



Veiling Luminance masks imperfection

4/29/2006 ©2005 Brightside Technologies Inc. Proprietary & Confidential 14

BRIGHTSIDE™

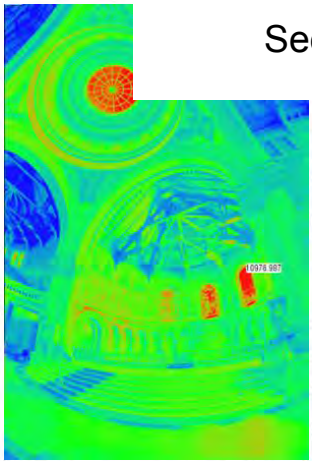
Display Technology – Review

- Compatibility
  - Based on commercially available components (LCD, LED)
  - Legacy support through Reverse Tone Mapping and Saturation Extension
  - Small number of LEDs allows encoding of LED data in conventional video signal
- Cost
  - LED cost money (less every day)
  - Significant power reduction (~25% of comparable constant backlight LCD on average)

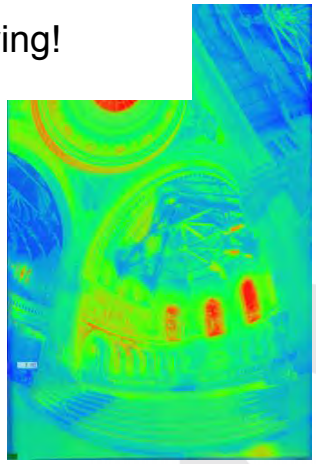
BRIGHTSIDE™

Display Technology – Results

Seeing is Believing!

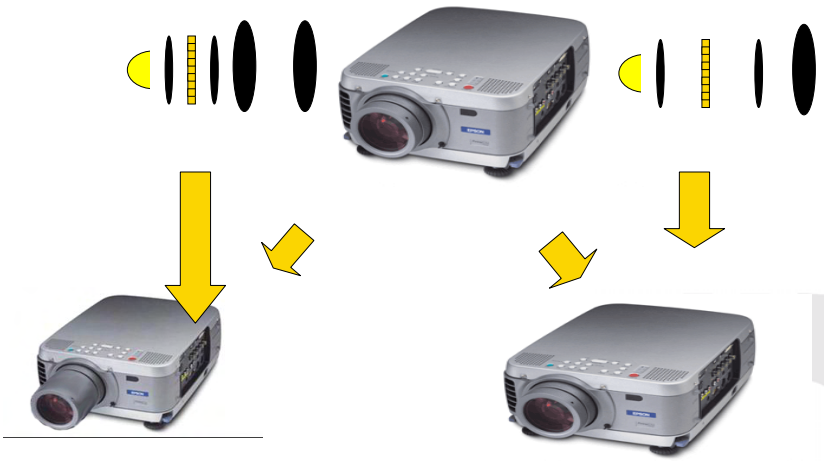


Stanford Memorial Church
Courtesy Paul Debevec



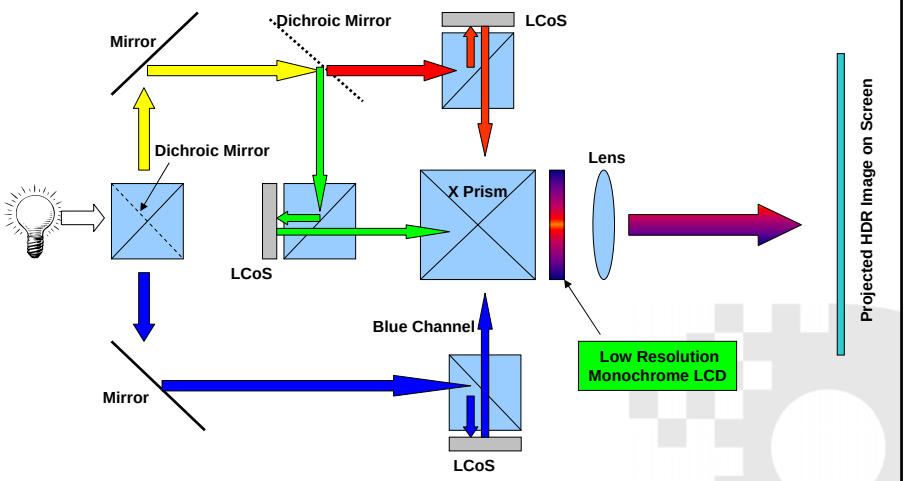
BRIGHTSIDE™

Projection Technology – Concept



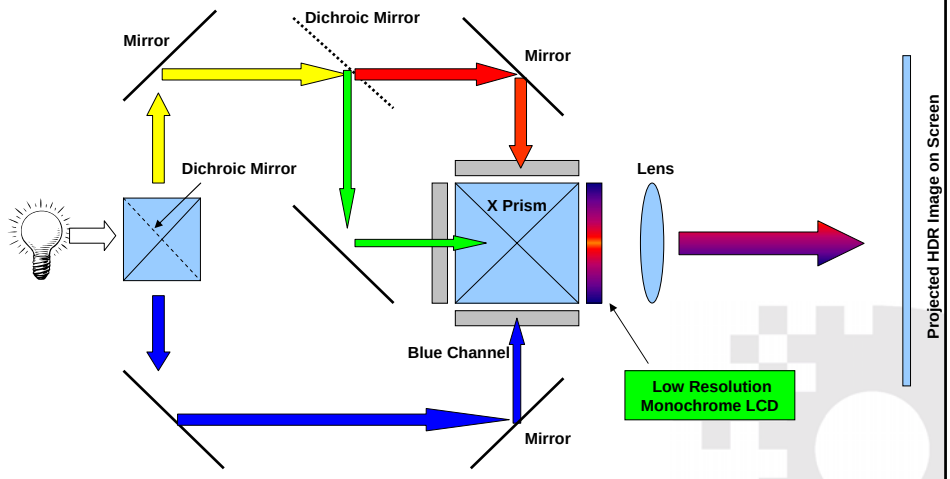
BRIGHTSIDE™

HDR Projection - LCoS Implementation



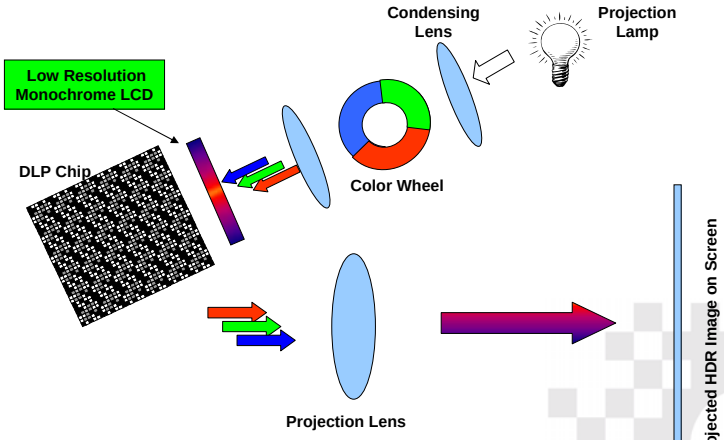
BRIGHTSIDE™

HDR Projection – 3 LCD Implementation



BRIGHTSIDE™

HDR Projection – DLP Implementation



BRIGHTSIDE™

Technology Overview – A word on Contrast

- High Dynamic Range = Contrast + Amplitude Resolution


≠


- Contrast depends on
  - Measurement technique
  - Ambient environment
  - Device Limitations

BRIGHTSIDE™

Technology Overview – A word on Contrast

Contrast Measurement Techniques

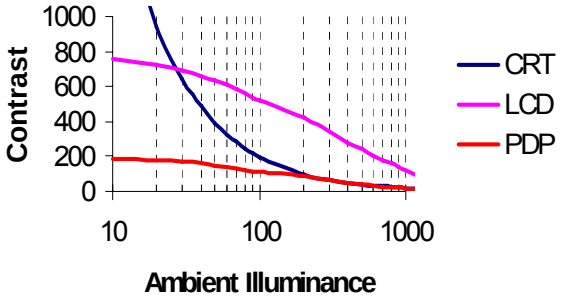
ANSI Contrast

On/Off Contrast

BRIGHTSIDE™

Technology Overview – A word on Contrast

Ambient Environment



Source:
Sharp Corp

BRIGHTSIDE™

Technology Overview – A word on Contrast

Device Limitations

- CRT: high front surface reflection
- LCD: Overdrive might prevent black
- Plasma: Size limit on white area

BRIGHTSIDE™

Technology Overview – Novel Concepts

Organic Light Emitting Diodes

- Good contrast
- Limited brightness
- Lifetime issues
- Optimists predict ~5years to compete with current LCD

Laser Projection

- Very high contrast
- Green & blue laser not available
- Optimists predict ~10-15 years

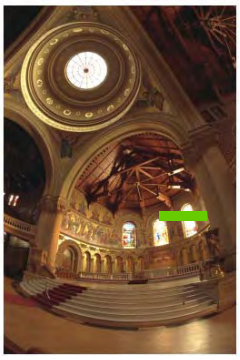
BRIGHTSIDE™

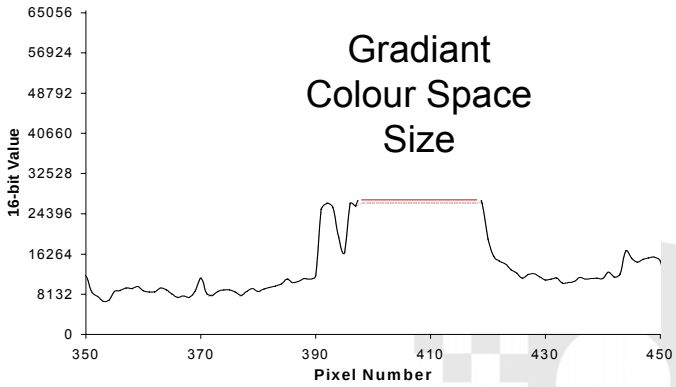
We have HDR Display – How to feed them?

- Floating point GPUs are becoming standard
- Most CG and Games done in HDR today
- TV remains low dynamic range and is unlikely to change soon
- Reverse Tone Mapping and Saturation Extension provide quality gain even for legacy content

BRIGHTSIDE™

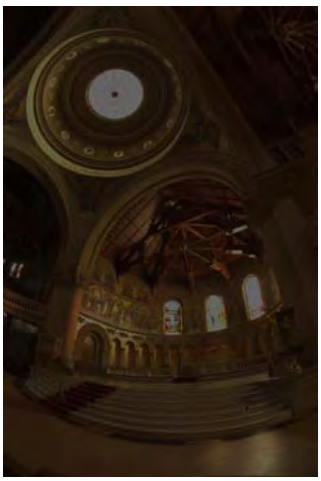
LDR to HDR Signal Extension





BRIGHTSIDE™

LDR to HDR Signal Extension



Compatibility
- only needed for legacy content
- calculation supported in 8-bit processor

Cost
- very fast (1-2 operations per pixel)

BRIGHTSIDE™

LDR to HDR Signal Extension

BRIGHTSIDE™

Summary

- HDR Displays are available
- HDR Projectors are coming
- HDR Output Devices can provide benefit to legacy content
- HDR Software / Input Devices are leading the way

HDR at Industrial Light + Magic

Drew Hess

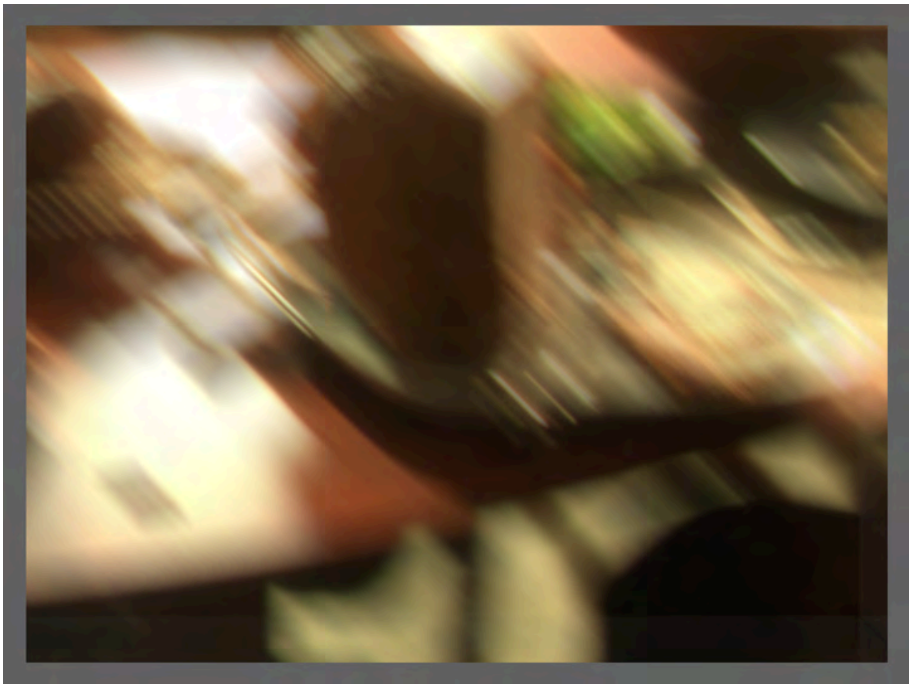
HDR in Film Production

Drew Hess
<dhess@ilm.com>
Industrial Light + Magic

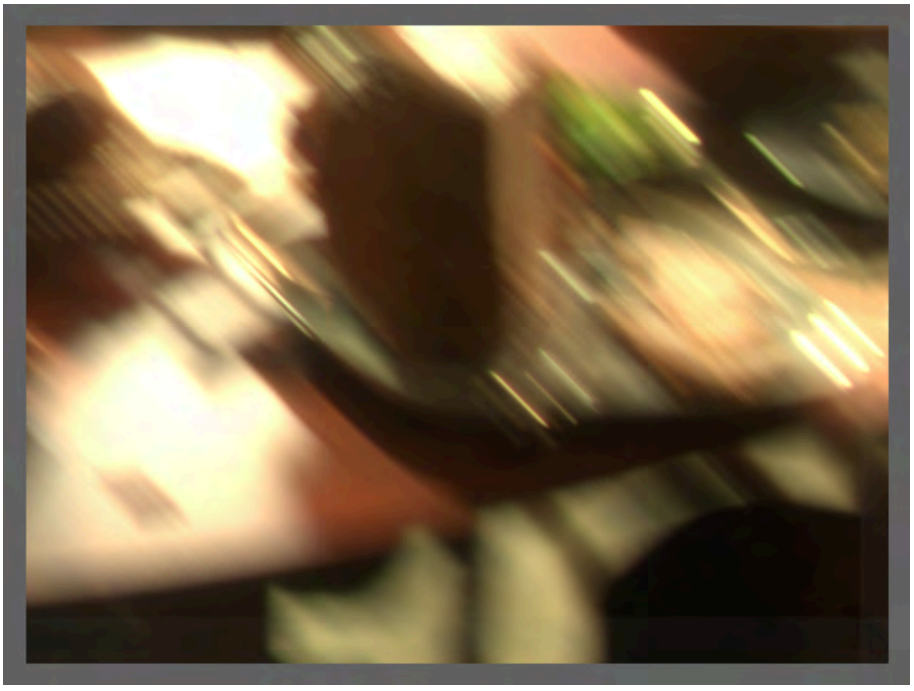
SIGGRAPH 2006 Course #5
High Dynamic Range Imaging: Theory and Applications
Full Day, Sunday, 30 July, 8:30AM - 5:30PM

Why is HDR important?

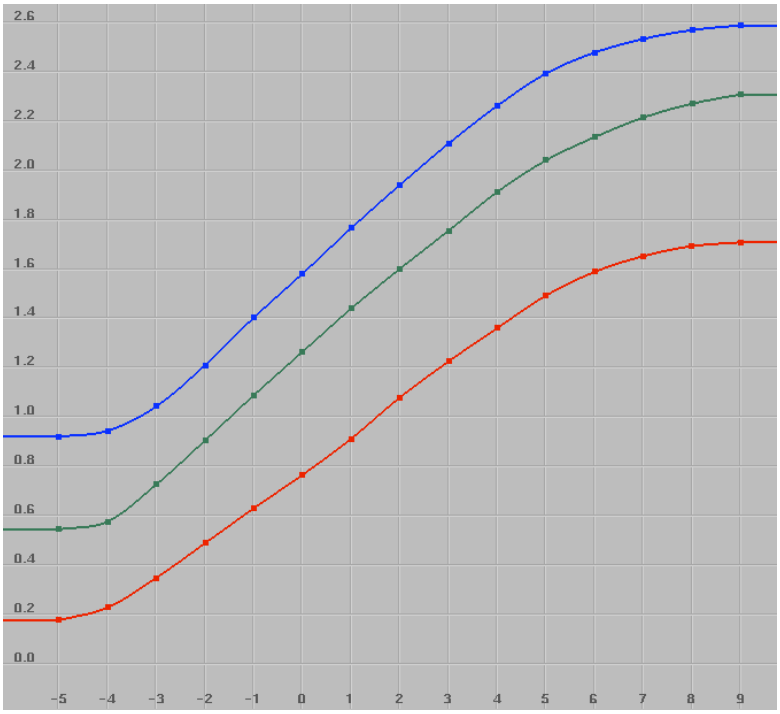
Motion Blur (bad)



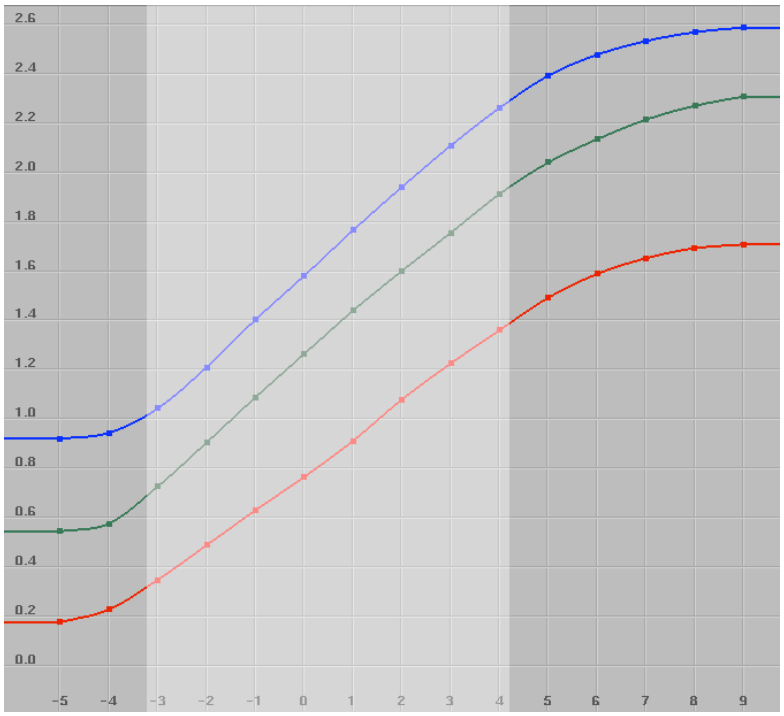
Motion Blur (good)



Film Characteristic Curve



Film Characteristic Curve



SpheroCamHDR

-8 stops

SpheroCamHDR

-4 stops

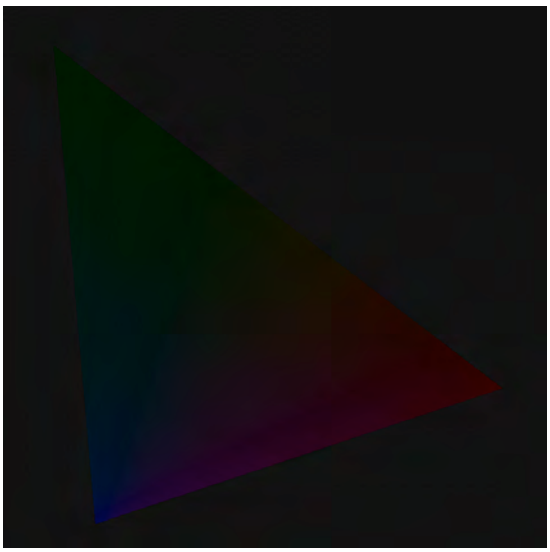
SpheroCamHDR

neutral

SpheroCamHDR

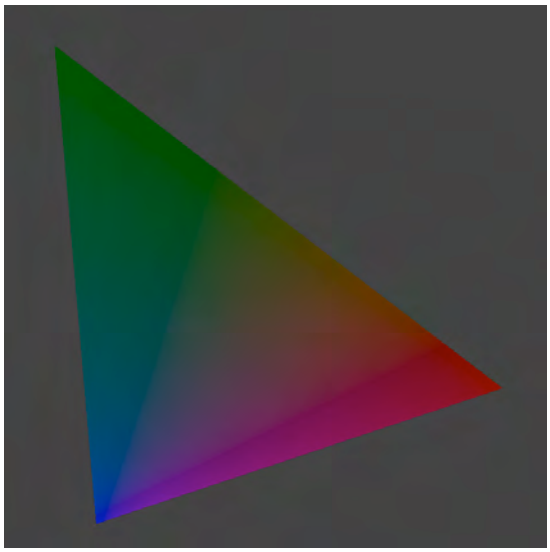
+4 stops

CG image



-8 stops

CG image

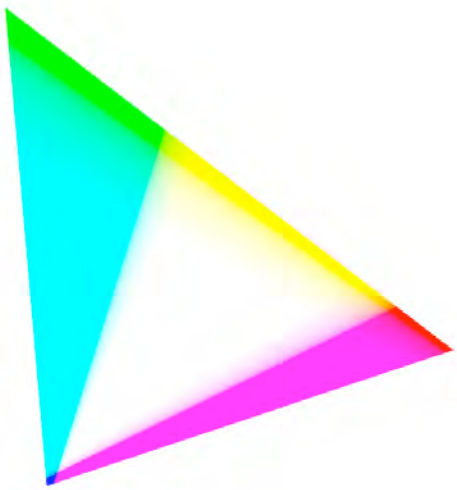


-4 stops

CG image

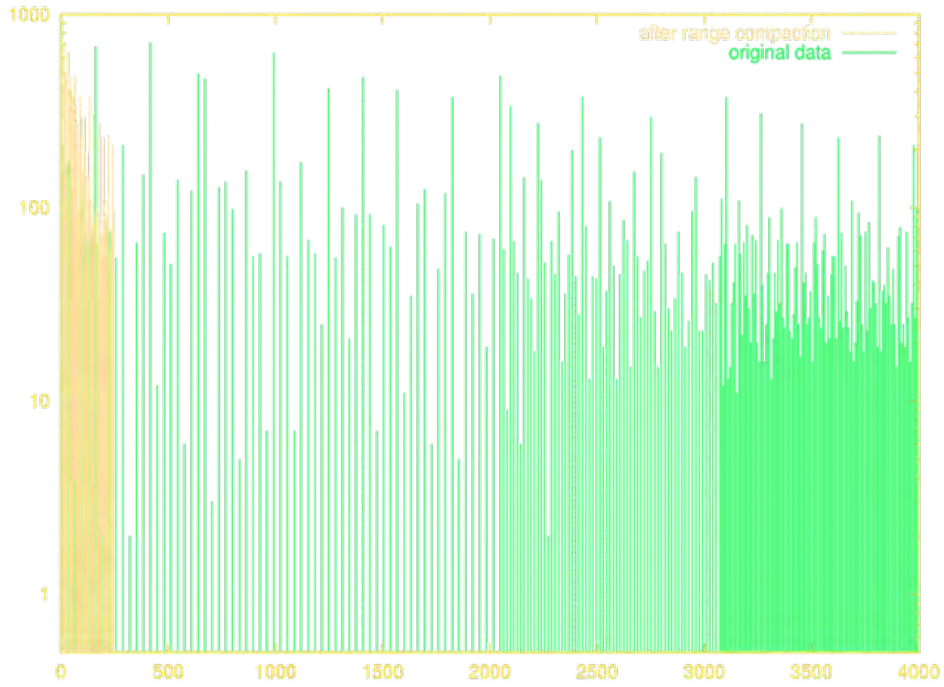
neutral

CG image



+4 stops


OpenEXR

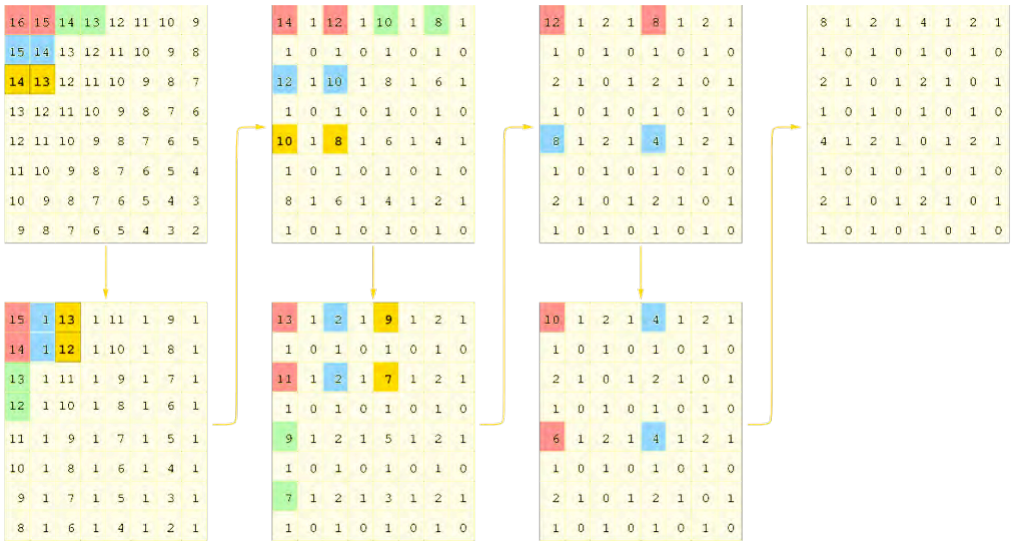


Wavelet Transform - Haar Wavelets

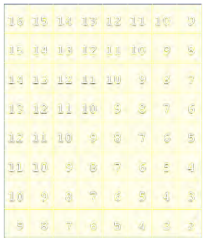
forward: **L = (A+B)/2**
H = A-B

reverse: **A = L+H/2+** { **1, if H is odd**
0, if H is even
B = A-H

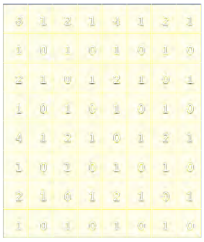
Original Data **Transformed Data**



Original Data

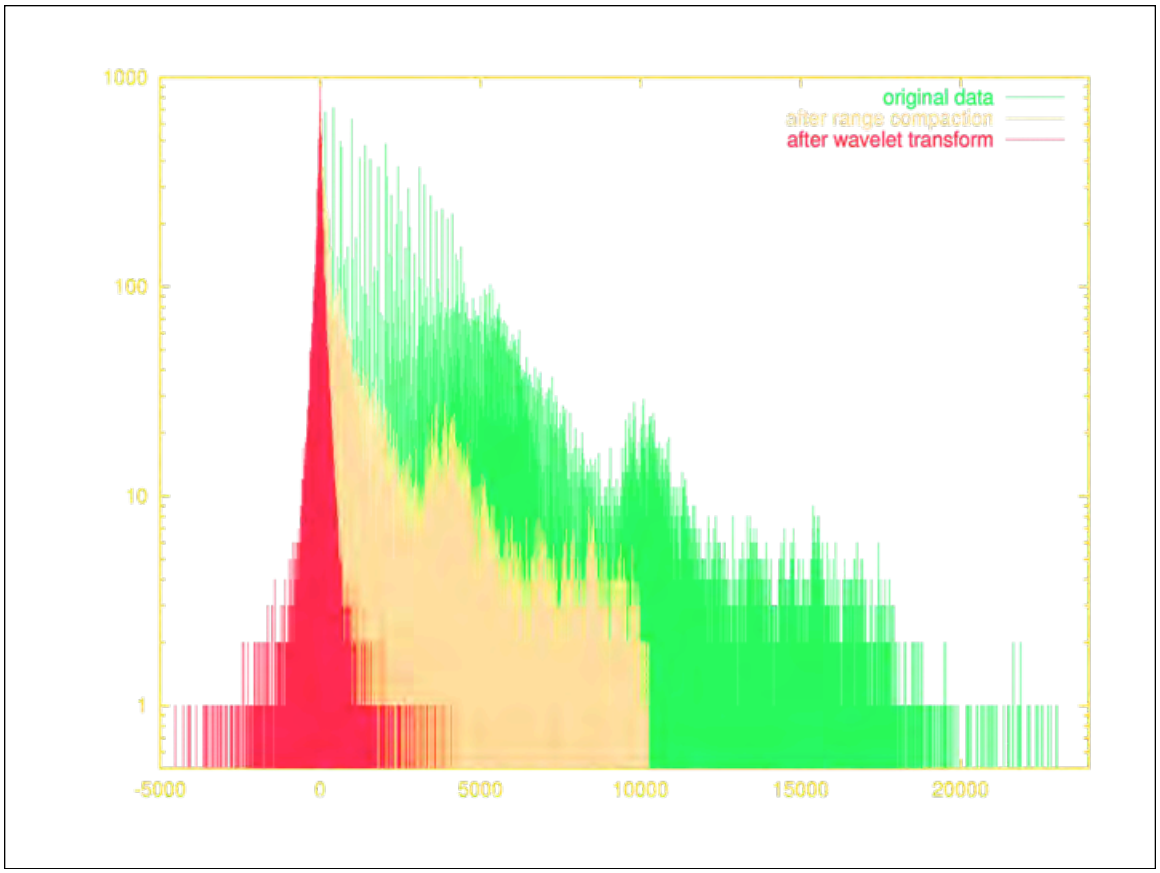


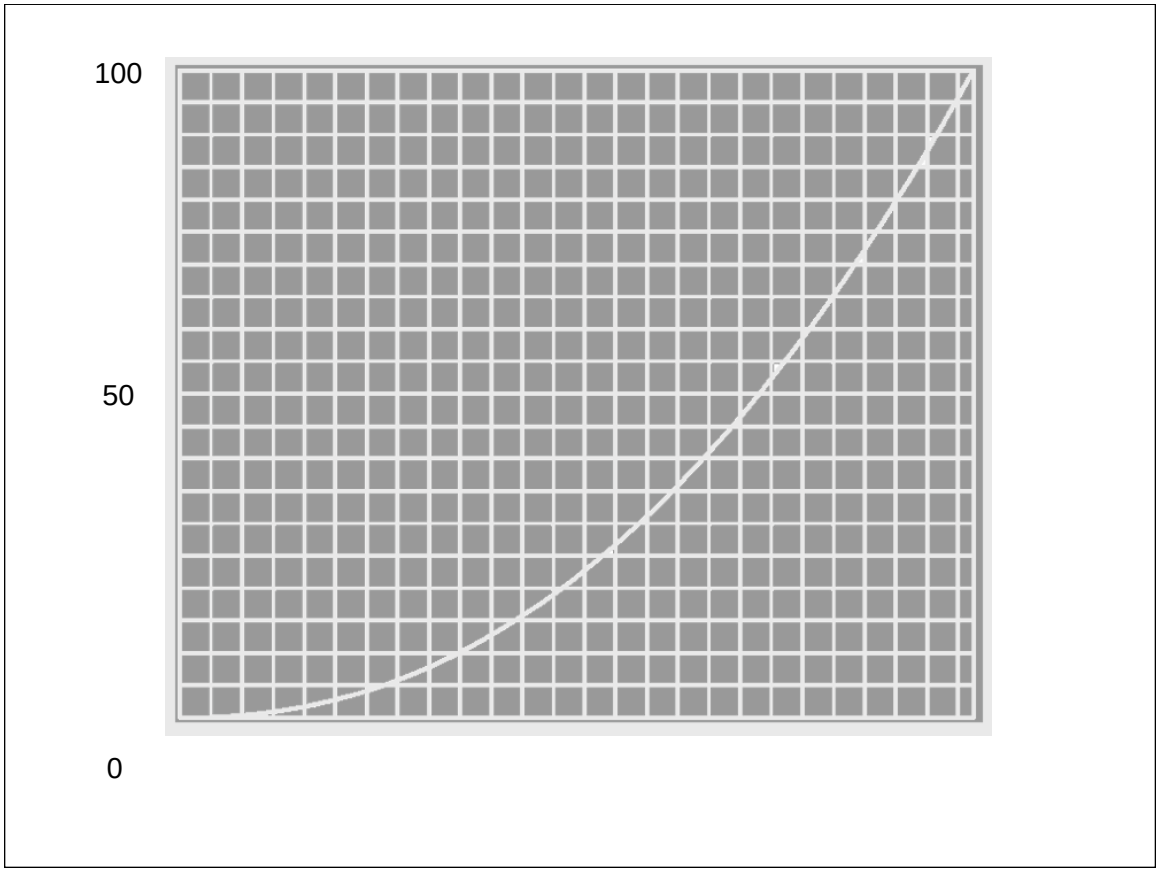
Transformed Data

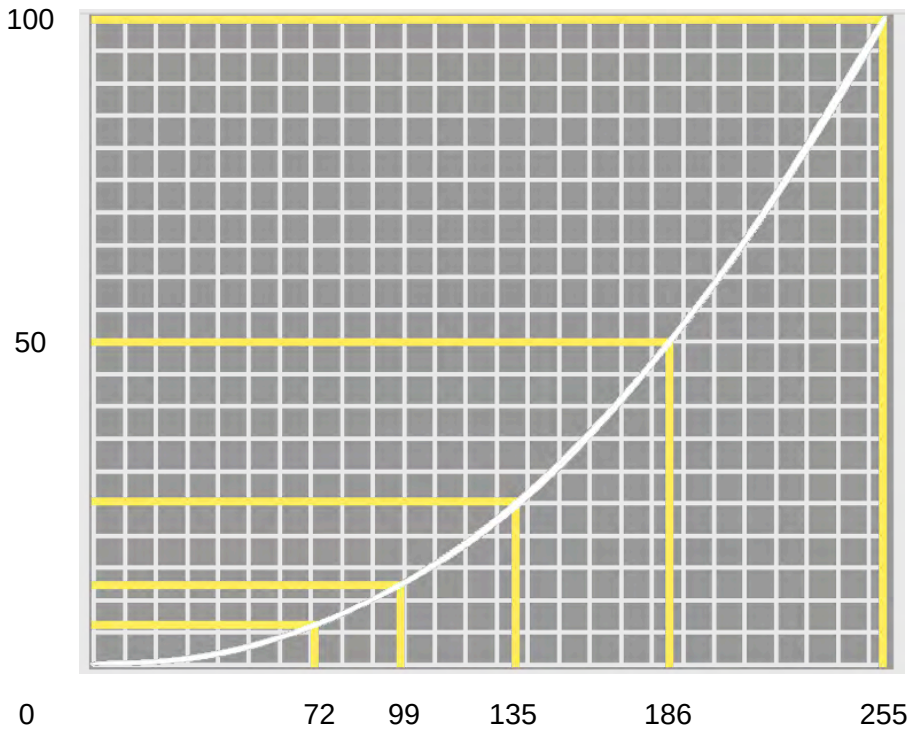


value	# occurrences	bits
2	1	6.0
3	2	5.0
4	3	4.4
5	4	4.0
6	5	3.7
7	6	3.4
8	7	3.2
9	8	3.0
10	7	3.2
11	6	3.4
12	5	3.7
13	4	4.0
14	3	4.4
15	2	5.0
16	1	6.0

value	# occurrences	bits
0	21	1.6
1	32	1.0
2	8	3.0
4	2	5.0
8	1	6.0


1000
100
10
1
-5000
0
5000
10000
15000
20000
original data
after range compaction
after wavelet transform


100
50
0



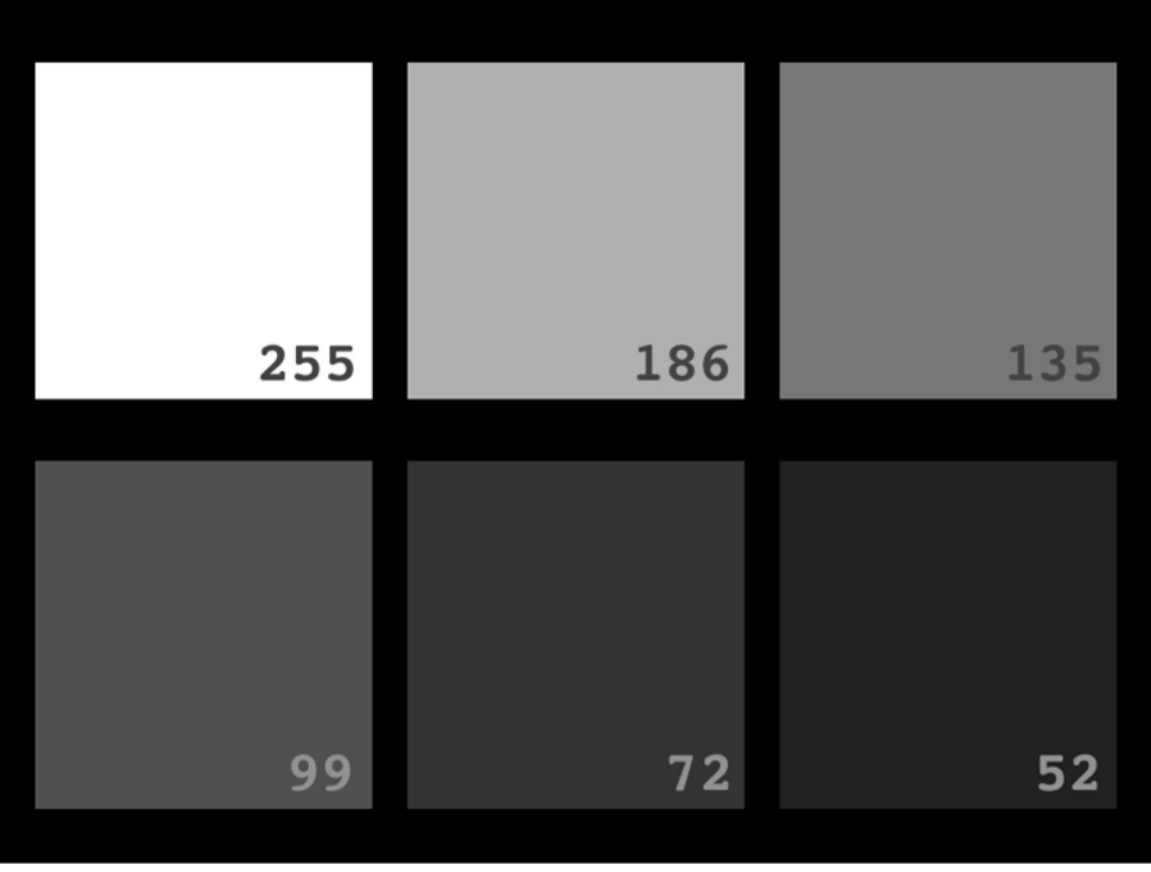
Implications of non-linear pixels

$$99 \times 2 = 198$$

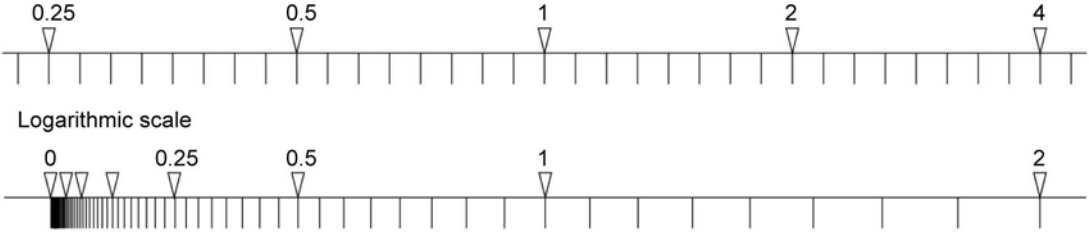
$$99 \times 2 = 135$$

$$(99/255)^{2.2} = 0.1247$$

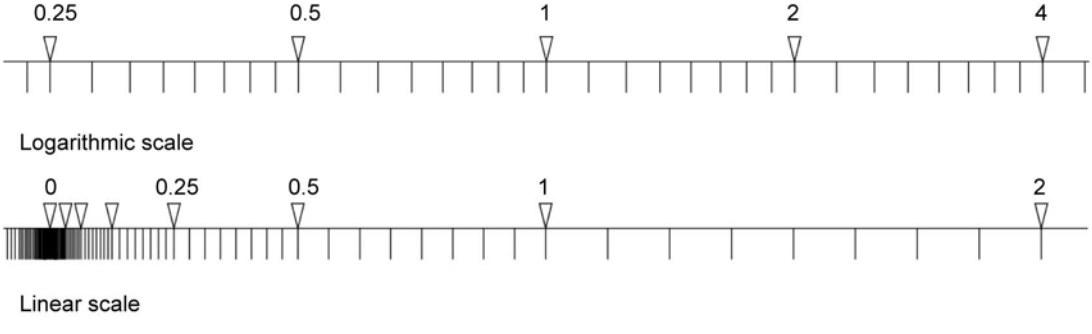
$$(135/255)^{2.2} = 0.2468$$



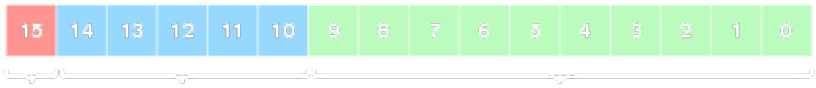
Logarithmic distribution:



Floating-point distribution:



half



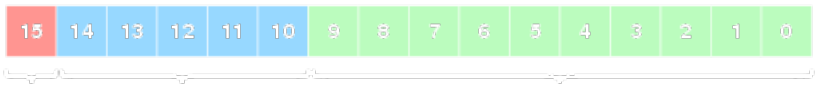
S E M

S	**1 bit**	**sign bit**
E	**5 bits**	**exponent**
M	**10 bits**	**mantissa**

Value (normalized case):

$$v = (-1)^S * 2^{E-15} * (1 + M/1024)$$

half



S E M

Range:

30 f-stops

maximum **65504.0**

minimum { **0.00061 (normalized)**; **0.00000006 (denormalized)**

Precision:

1024 increments per stop
0.1% relative error

Tone mapping

“neutral”

Tone mapping

Deluxe Vision

Tone mapping

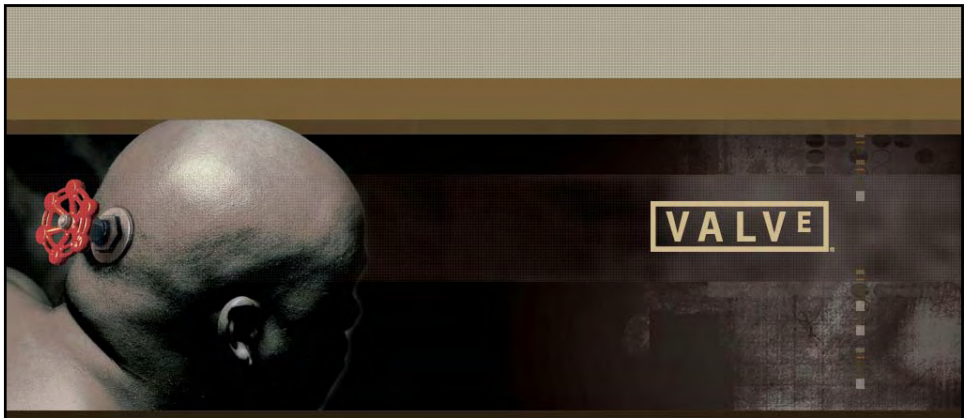
Bleach bypass

Tone mapping

DLP

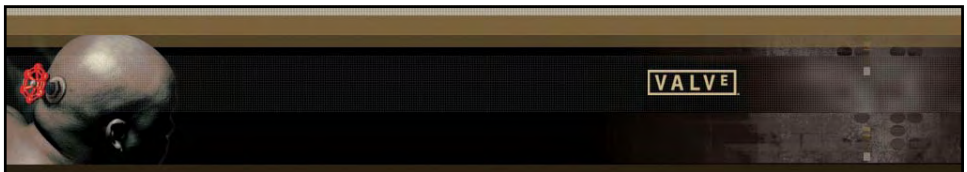
HDR at Valve

Gary McTaggart



High-Dynamic Range Rendering

Valve Design Team

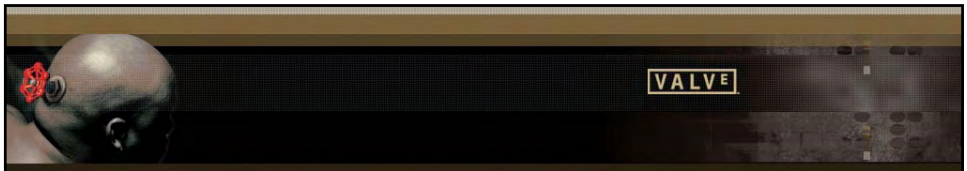


Agenda

- What is HDR?
- Technology – HDR in Source
- Production – Creating HDR Content
- Challenges
- Benchmarking and visualization
- Lost Coast demo
- Playtest Day of Defeat
- Q&A and giving you a build to take away

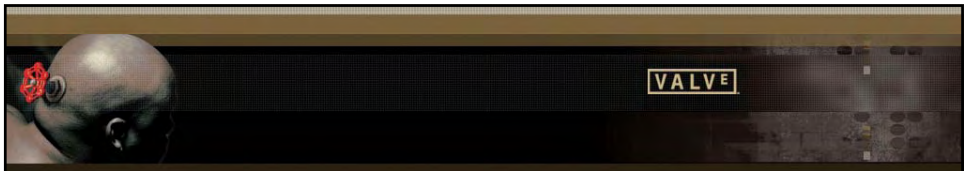
"The 'dynamic range' of a scene is the contrast ratio between its brightest and darkest parts. A plate of evenly-lit mashed potatoes outside on a cloudy day is low-dynamic range. The interior of an ornate cathedral with light streaming in through its stained-glass windows is **high dynamic range**. In fact, any scene in which the light sources can be seen directly is high dynamic range."

– Paul Debevec



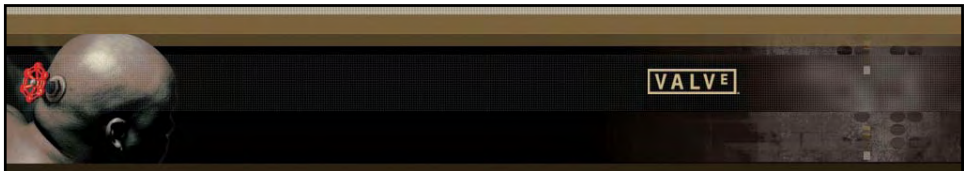
HDR

- A High-Dynamic Range image is an image that has a greater contrast range than can be shown on a standard display device, or that can be captured with a standard camera with just a single exposure.



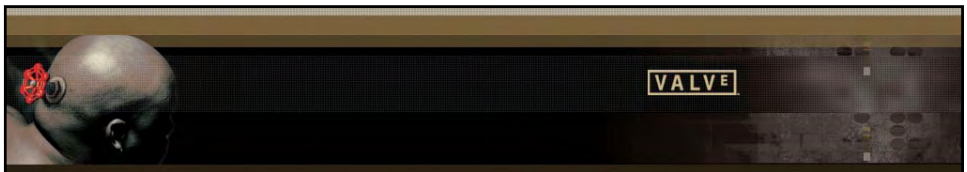
HDR

- High-Dynamic Range rendering attempts to take an HDR image and produce a more realistic representation on a limited-range computer monitor.



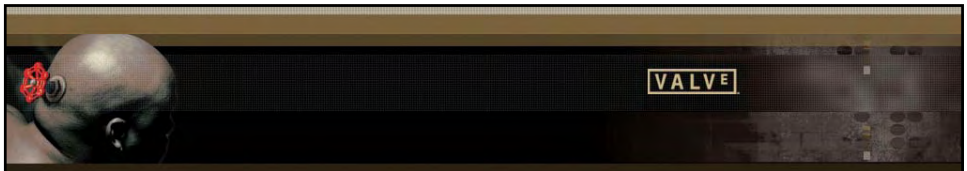
Lost Coast and HDR

- HDR as incremental technology
  - More efficient way to develop technology
  - Delivers value to customers and licensees
  - In all future games (DoD, Aftermath, ...)
- Part of the graphics roadmap
- Shorten the delay between hardware availability and game support and then target the delivery
- Give the GPU developers production code with which to evaluate their decisions



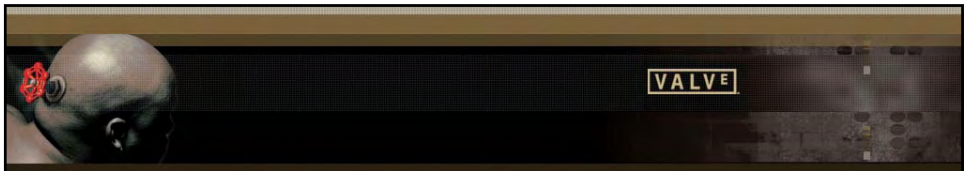
HDR in games

- So far, most video game “HDR” implementations are really just blooming.
- Content not authored for HDR since HDR is more of an afterthought.
- Since the definition of HDR varies within the game industry, we have a list of what we think is important for HDR.



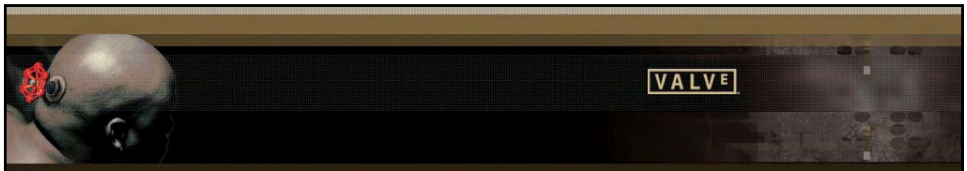
HDR and Source

- Blooming – adding a blurred version of the bright parts of the scene to emulate a camera's overexposure



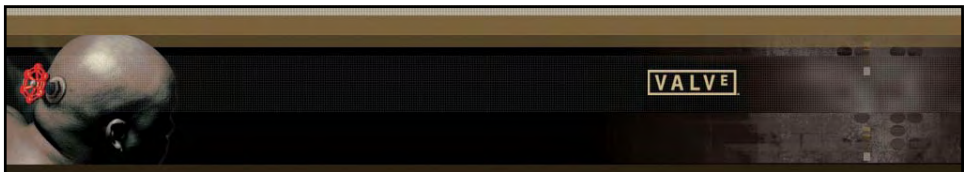
HDR and Source

- HDR skybox – authored by painting multiple exposures of the sky to allow for real-time exposure adjustment.



HDR and Source

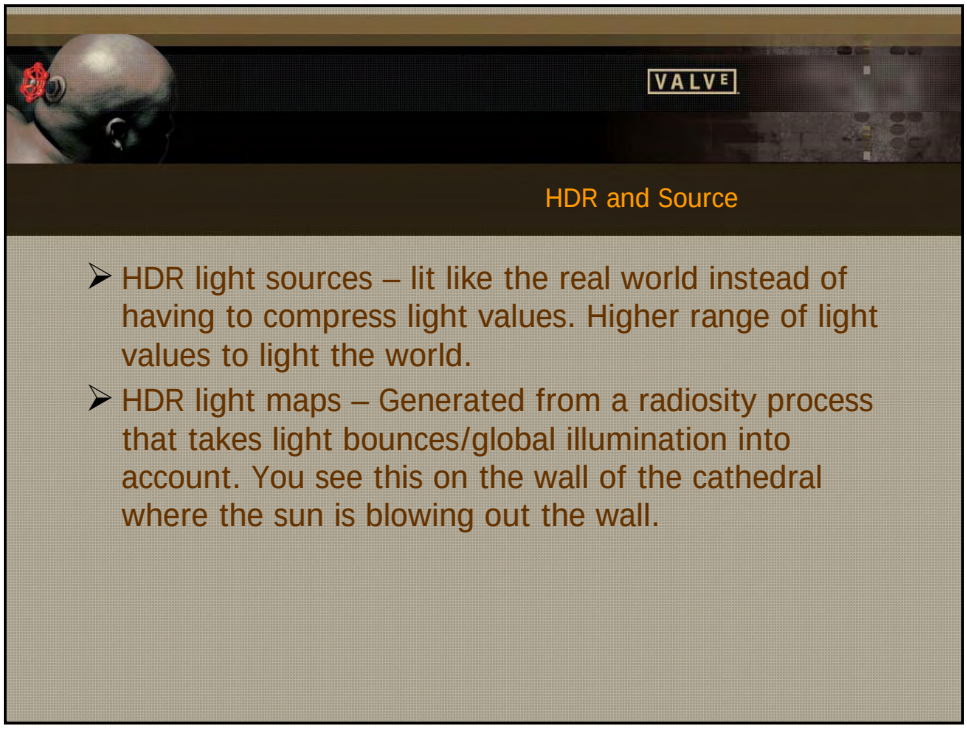
- HDR cube maps – generated by the engine using the HDR skybox as well as the HDR light sources and the HDR light maps. This means that if an objects is reflecting the sun or some other bright part of the scene, you will see this in the full affect of the brightness in the reflection.

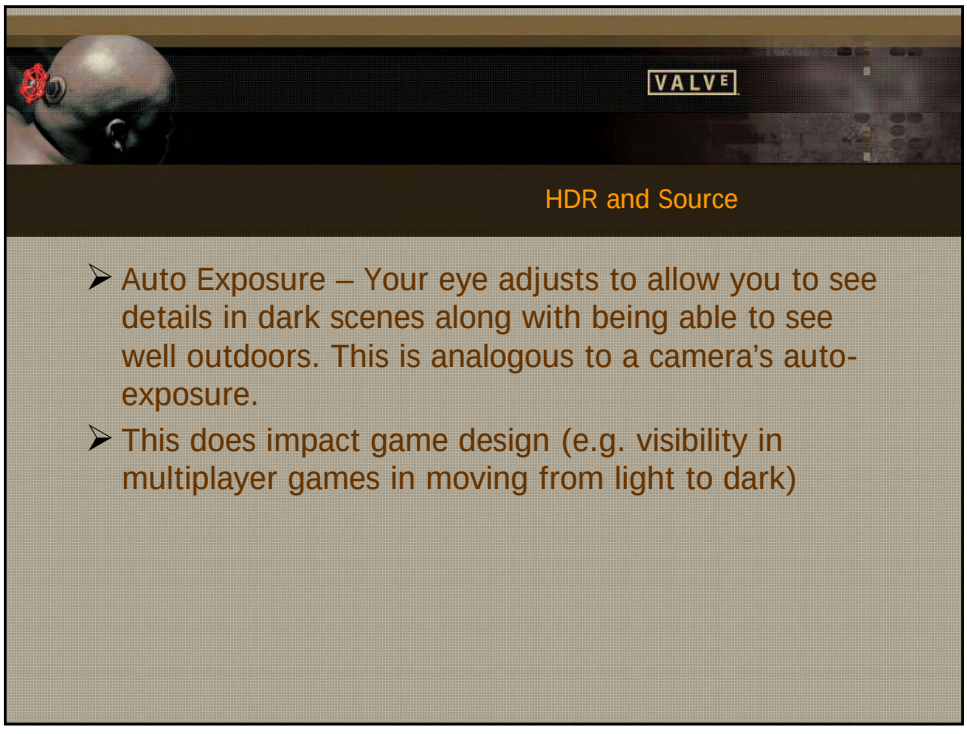


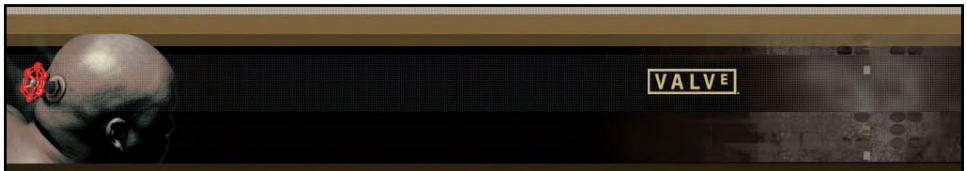
HDR and Source

- HDR water reflection/refraction – Where the reflection of the sky is really really bright, you get white hot spots along with blooming. If you are under water looking out, you'll get a similar effect.


LDR
HDR

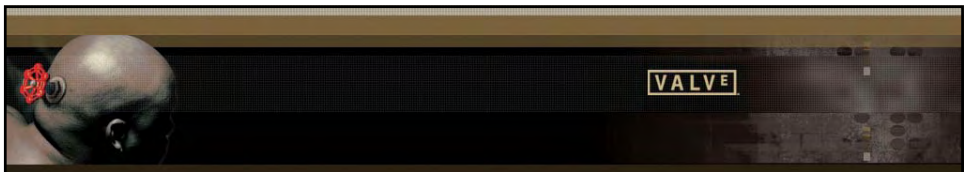

VALVE
HDR and Source
HDR light sources – lit like the real world instead of having to compress light values. Higher range of light values to light the world.
HDR light maps – Generated from a radiosity process that takes light bounces/global illumination into account. You see this on the wall of the cathedral where the sun is blowing out the wall.


VALVE
HDR and Source
Auto Exposure – Your eye adjusts to allow you to see details in dark scenes along with being able to see well outdoors. This is analogous to a camera's auto-exposure.
This does impact game design (e.g. visibility in multiplayer games in moving from light to dark)



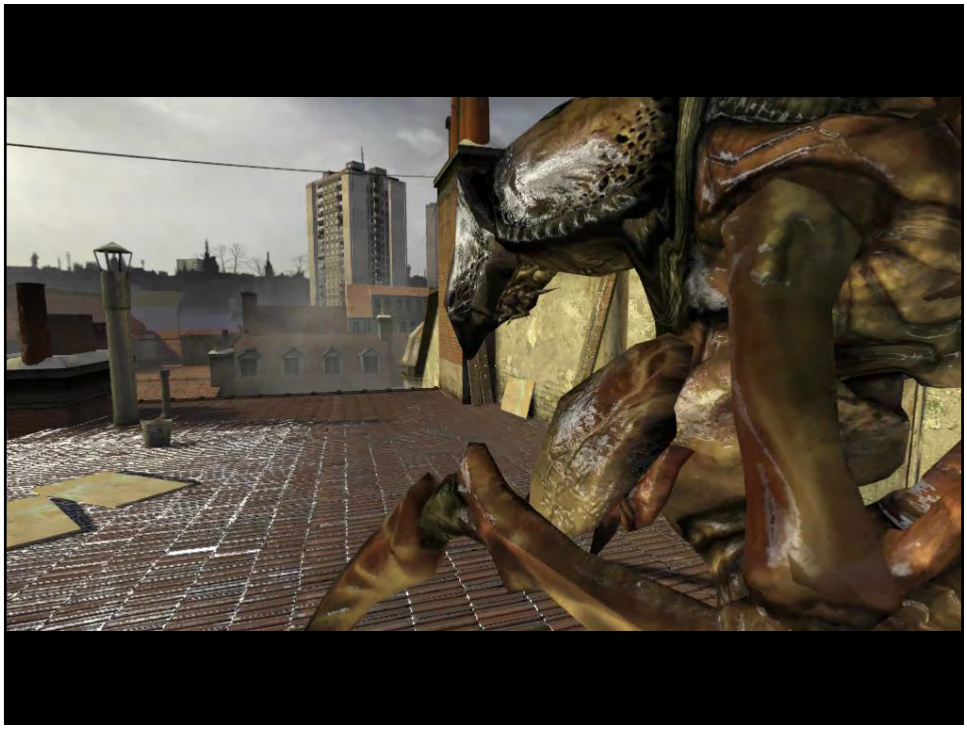
HDR Implementations

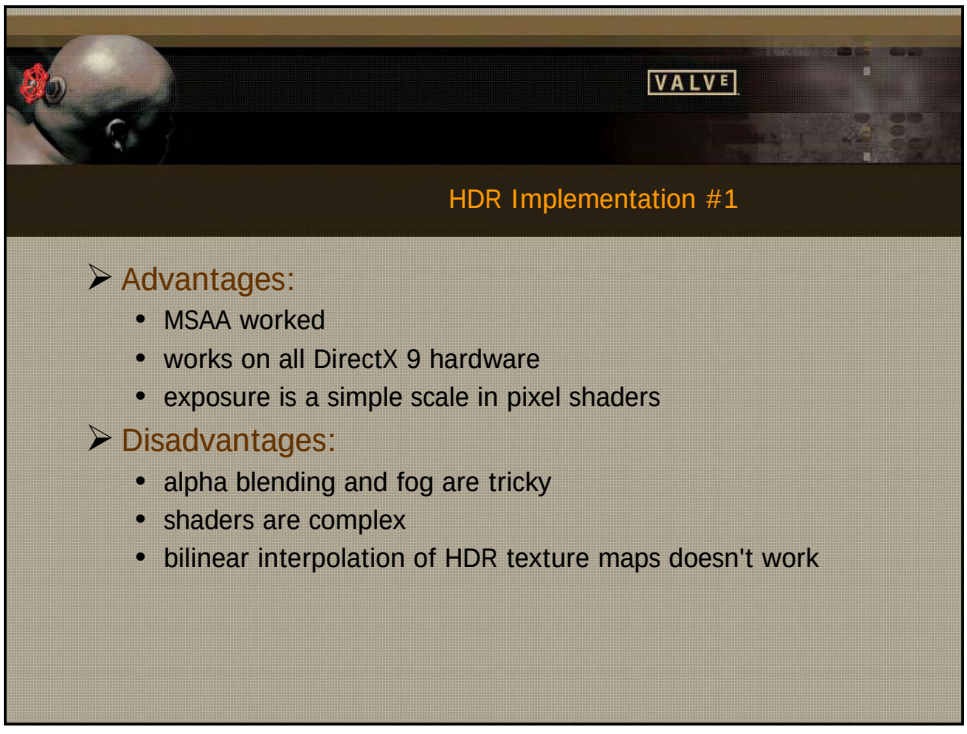
- HDR in real-time with hardware acceleration is cutting edge technology
- Lots of side effects and consequences were unknown when we started
- As a result, we implemented 4 different methods
- We will give an overview of the advantages and disadvantages of each method



HDR Implementation #1

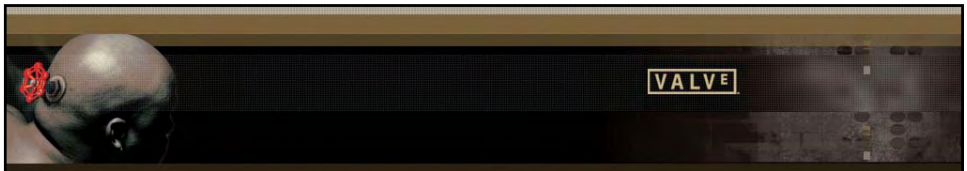
- Demonstrated in Fall of 2003 at Shader Day
- Stored HDR textures as RGBA textures where Alpha contained a value between 0 and 16 to multiply RGB by.
- We learned from this demo that doing a full HDR implementation for HL2 would require a huge content and technology investment. We postponed HDR for HL2 in early 2004.





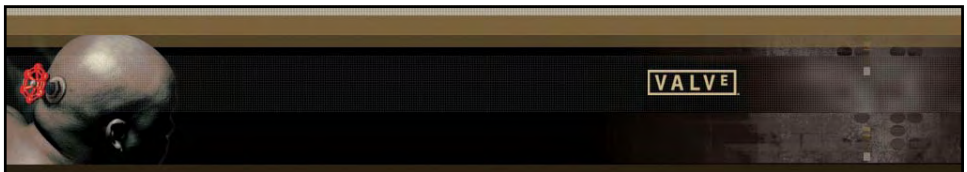
HDR Implementation #1

- Advantages:
  - MSAA worked
  - works on all DirectX 9 hardware
  - exposure is a simple scale in pixel shaders
- Disadvantages:
  - alpha blending and fog are tricky
  - shaders are complex
  - bilinear interpolation of HDR texture maps doesn't work



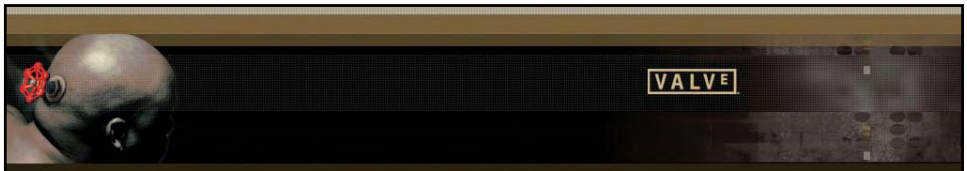
HDR Implementation #2

- This is the first post-Half-Life 2 HDR experiment and was used for the first Lost Coast demos.
- This method uses two MRT buffers/textures for all HDR resources.
  - The first buffer stores visible data.
  - The second buffer stores overbright data.



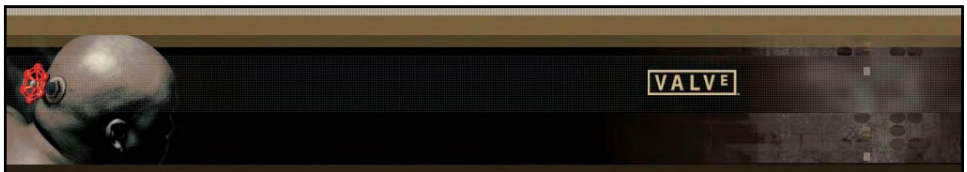
HDR Implementation #2

- Pluses:
  - Main motivation: alpha blending works.
  - works on all DirectX 9 hardware
  - bilinear interpolation works
- Minuses:
  - MSAA doesn't work: MSAA and MRT don’t work together on current hardware
  - can't use the hardware fog
  - HDR textures, render targets, etc take twice as much space.



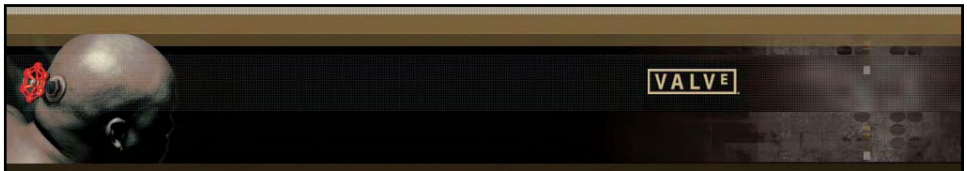
HDR implementation #3

- Used for E3 2005 Lost Coast demo
- HDR textures and render targets are all full floating-point RGB.
- this is the future
- Improved tone mapping
- HDR refraction



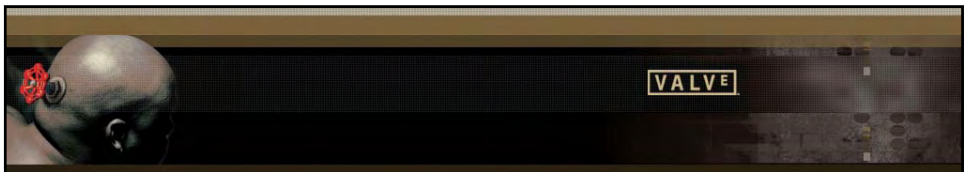
HDR Implementation #3

- Pluses:
  - generates the best data, we use this mode to generate our procedural HDR data
  - you get actual overbright data to bloom with
- Minuses:
  - Requires floating-point alpha blending
  - Performance not that good for supported hardware.
  - Tons of memory used for all resources.
  - no MSAA
  - Current hardware lacks precision to do fog



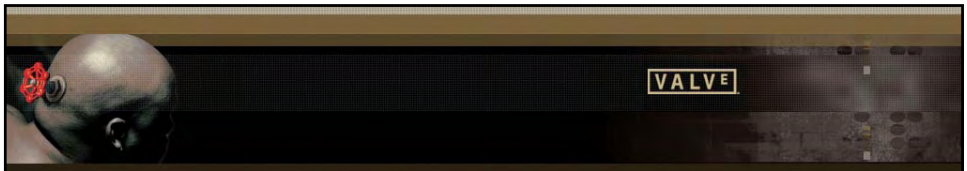
HDR Implementation #4

- Lost Coast and Day of Defeat will ship with this!
- Similar to 2 buffer version, but without over-bright data
- Store HDR textures in fp16 (linear color space) when filtered fp16 textures are supported.
- Store HDR textures in 4.12 linear color space otherwise.
- Had to implement this version along with previous version to support more hardware.



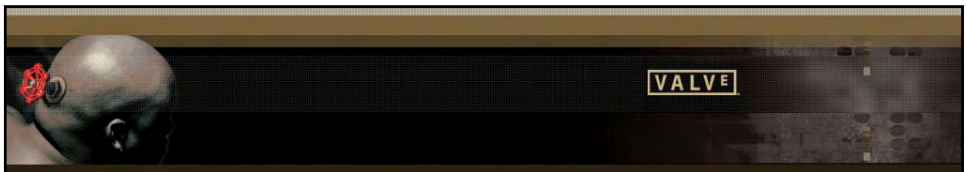
HDR Implementation #4

- Pluses:
  - MSAA works!!!
  - works on all dx9 hardware (runs well even on 9600)
  - best performance out of all of our methods
    - is a small performance hit over HL2's shipped LDR solution
  - doesn't take much memory (only extra memory is for HDR textures and bloom buffers)
  - fog interpolators work
- Minuses:
  - Refraction doesn't carry HDR data through



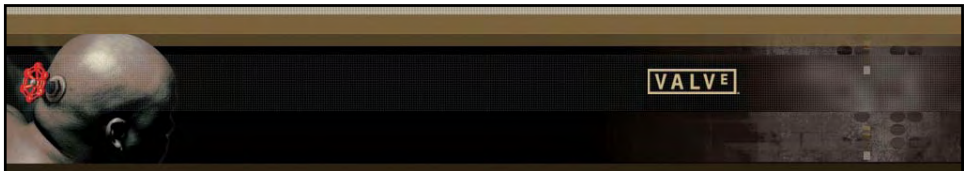
HDR and Authoring Content

- Only hand-authored HDR textures are skybox.
- Techniques for creating HDR skyboxes
  - Use a 3D rendering program
    - Many 3D packages read and write HDR formats
  - Use multiple exposures photo-reference
    - Use HDR Shop to build HDR images.
  - Hand-authored
    - HDR Shop + Photoshop
    - Currently HDR paint programs are inadequate. (including Photoshop CS2)



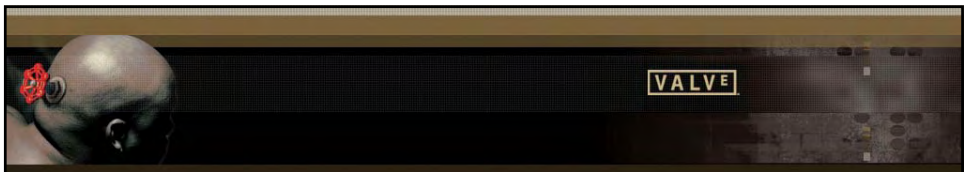
HDR and Authoring Content

- Lighting for HDR is different
- LDR: light for a single exposure
  - use supplemental fill lights to brighten dark areas
- HDR: light more like the real world, with dynamic exposure
  - Lost Coast uses one primary light, the sun.
  - Would not work without radiosity to bounce the light around.
- We actually light levels separately for LDR and HDR



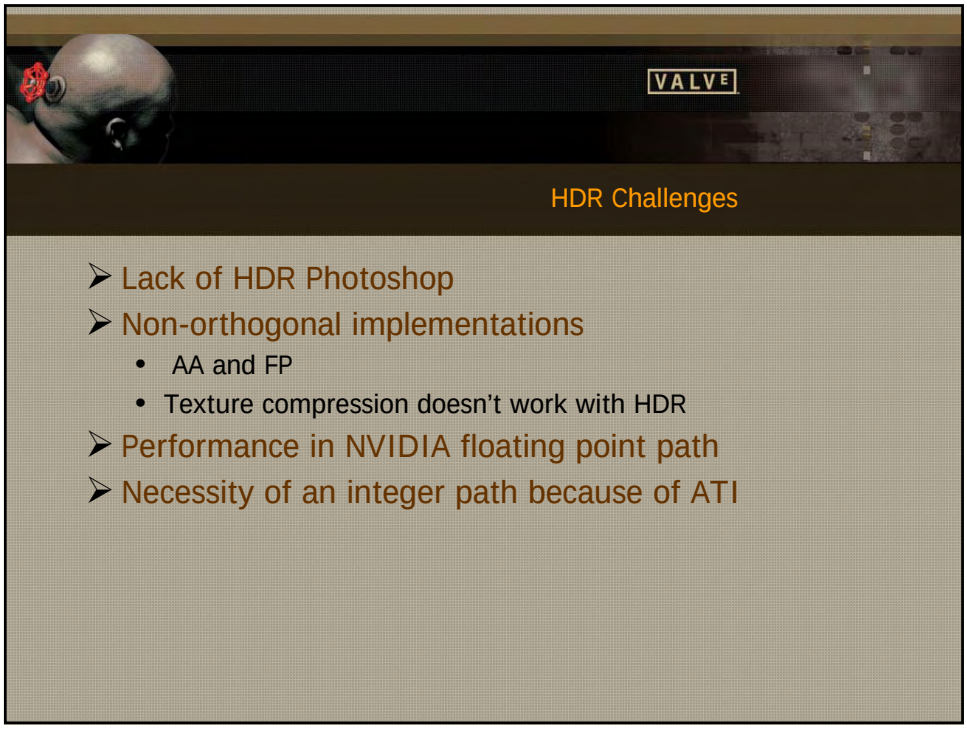
HDR and Authoring Content

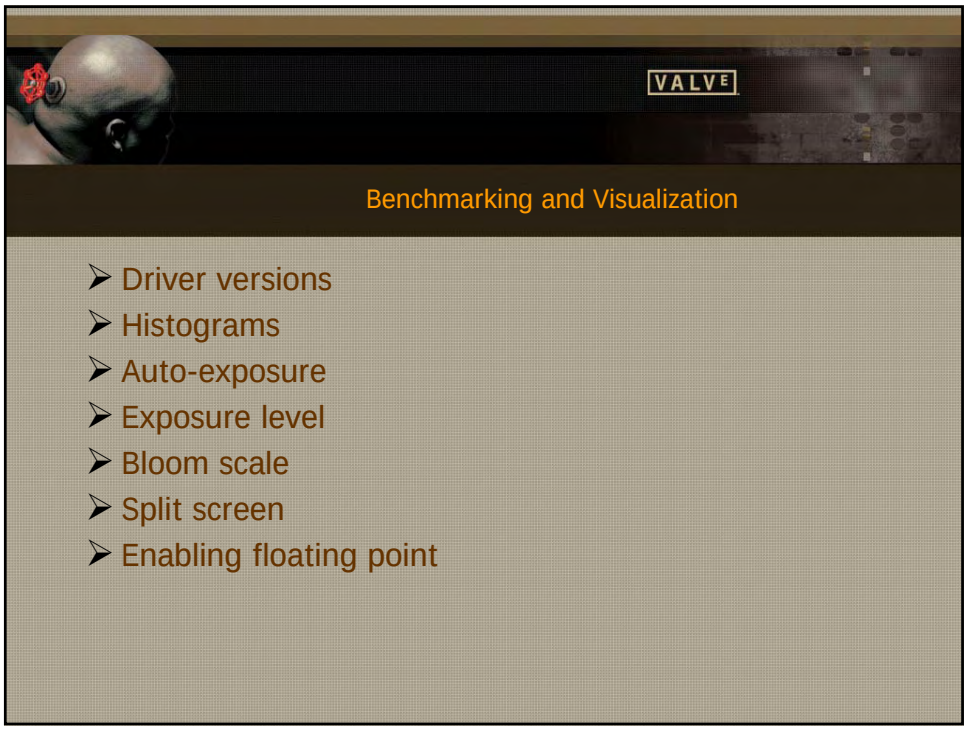
- Bloom amount and exposure range
  - Are settable on a per map or per map area basis by level designers to override the automatic settings.
- Within this set range we use auto-exposure to adjust the exposure level.



HDR and MODs

- When Lost Coast ships, HDR support in SDK will also be released
- MODs will need to author LDR and HDR lighting for maps, and create HDR sky boxes (or use ours)
- LDR and HDR lighting contained in the same BSP
- There's gameplay to explore for stealth and in exposure control (e.g. flash bangs)

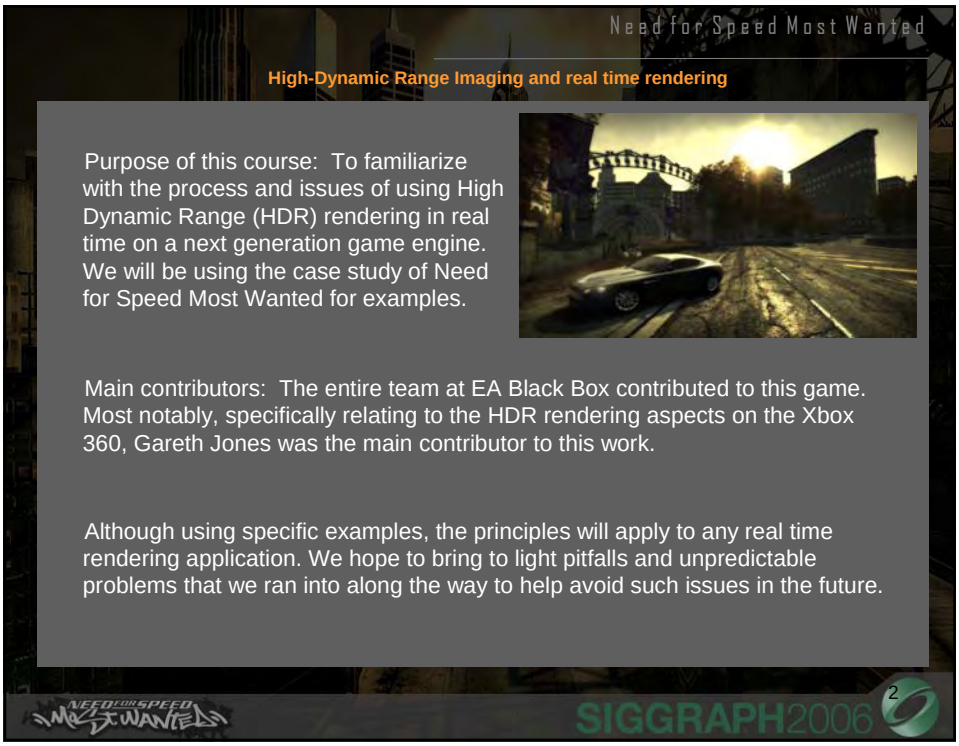

VALVE
HDR Challenges
Lack of HDR Photoshop
Non-orthogonal implementations
AA and FP
Texture compression doesn't work with HDR
Performance in NVIDIA floating point path
Necessity of an integer path because of ATI


VALVE
Benchmarking and Visualization
Driver versions
Histograms
Auto-exposure
Exposure level
Bloom scale
Split screen
Enabling floating point

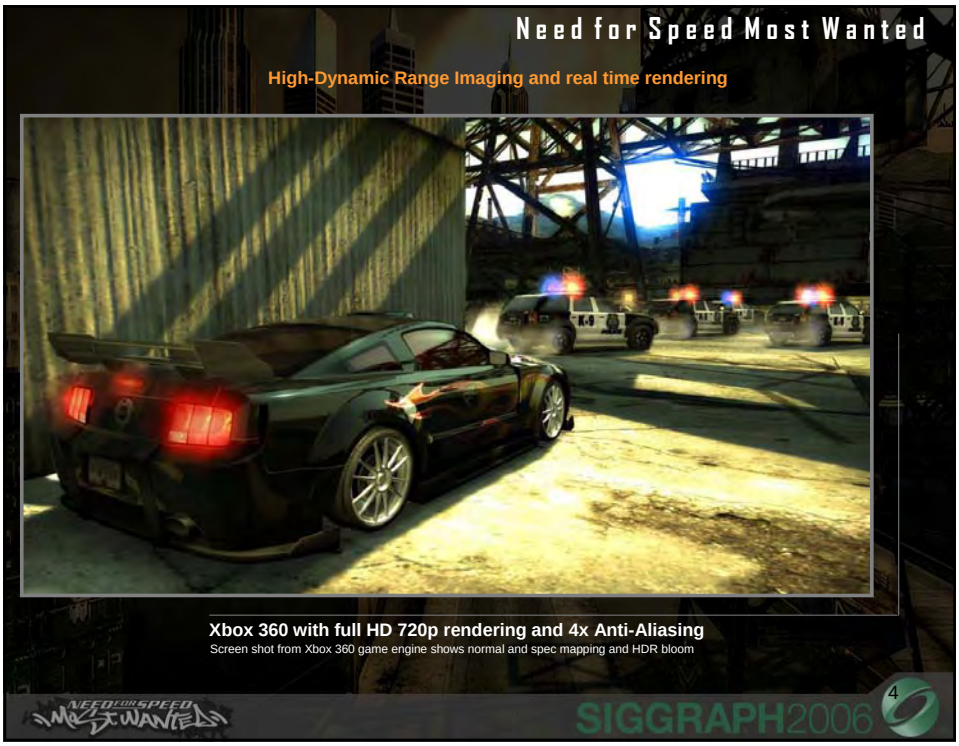
HDR at Electronic Arts

Habib Zargarpour


High-Dynamic Range Imaging
SIGGRAPH2006
Need for Speed Most Wanted
Habib Zargarpour
http://www.zargarpour.net
Senior Art Director, Electronic Arts
Need for Speed Team


Need for Speed Most Wanted
High-Dynamic Range Imaging and real time rendering
Purpose of this course: To familiarize with the process and issues of using High Dynamic Range (HDR) rendering in real time on a next generation game engine. We will be using the case study of Need for Speed Most Wanted for examples.
Main contributors: The entire team at EA Black Box contributed to this game. Most notably, specifically relating to the HDR rendering aspects on the Xbox 360, Gareth Jones was the main contributor to this work.
Although using specific examples, the principles will apply to any real time rendering application. We hope to bring to light pitfalls and unpredictable problems that we ran into along the way to help avoid such issues in the future.
2
SIGGRAPH2006


Need for Speed Most Wanted
High-Dynamic Range Imaging and real time rendering
Xbox 360 with full HD 720p rendering and 4x Anti-Aliasing
Screen shot from Xbox 360 game engine shows normal and spec mapping and HDR bloom
3
SIGGRAPH2006


Need for Speed Most Wanted
High-Dynamic Range Imaging and real time rendering
Xbox 360 with full HD 720p rendering and 4x Anti-Aliasing
Screen shot from Xbox 360 game engine shows normal and spec mapping and HDR bloom
4
SIGGRAPH2006

Lighting of sky is done in conjunction with the atmosphere and fog with dynamic scattering

Screen shot showing Good sky exposure and bloom

- Sky texture in HDR buffer
- Histogram used to determine exposure
- Resulting bright areas are isolated for blooming.
  - Blooming depends on exposure amount
  - Exposure is adjusted in real time according to the brightness on screen

Bloom can also affect bright reflections on the surface of the car

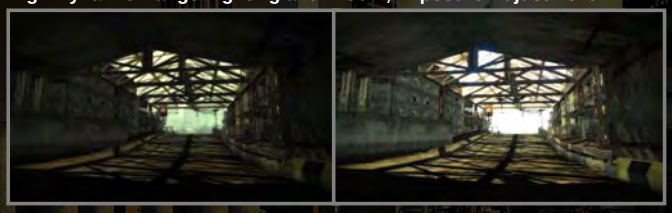
Cast shadows also affect cars

Dynamic real time lighting Sun rises and sets in 30 minute cycles

All objects are lit in real time and cast shadows move with the sun

- Weather Effects and rain

High Dynamic Range Lighting and Bloom, Exposure Adjustment



- HDR lighting, Bloom, and Dynamic Iris **OFF** -

- HDR lighting, Bloom, and Dynamic Iris **ON** -

All screen shots from Xbox 360 game engine

Sun and Volumetric Particle Lighting

Visual Filter - real time compositing and post processing

- Visual Filter and Motion Blur **OFF** -

- Visual Filter and Motion Blur **ON** -

Exposure adjustment in the shade makes sky naturally blown out

Tree Lighting

Real time lighting on trees softly distributes the sunlight according to the sun position

Visual Filter - real time compositing and post processing

- Visual Filter OFF -

- Visual Filter ON -

High Dynamic Range Lighting and Bloom, Exposure Adjustment

Visual Filter - real time compositing and post processing

- HDR lighting, Bloom, and Dynamic Iris **OFF** -

All screen shots from Xbox 360 game engine

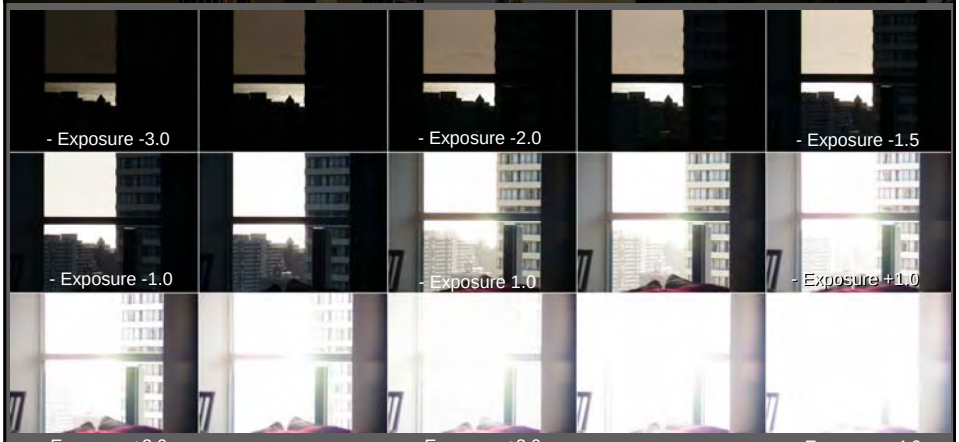
High Dynamic Range Lighting and Bloom, Exposure Adjustment

Visual Filter - real time compositing and post processing

- HDR lighting, Bloom, and Dynamic Iris **ON** -

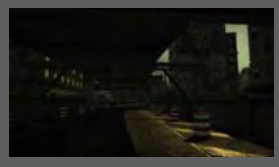
All screen shots from Xbox 360 game engine

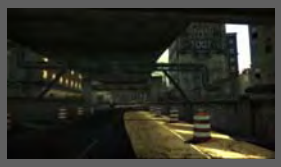
Exposure Adjustment: Real Photo Reference



- **Using a digital camera with aperture and shutter speed control we vary the exposure to get a wide range of stops to form a high dynamic range image.**
- **Notice the increased bloom effect at higher exposures.**

Exposure Adjustment: Real time within the game engine


- Exposure 0.4


- Exposure 1.0


- Exposure 1.5


- Exposure 2.0


- Exposure 3.0


- Exposure 4.0


- Exposure 5.0


- Exposure 9.0

Increasing the exposure will automatically cause areas that are over-exposed to contaminate the area around them.

You can see the similar behavior compared to the real photo reference

Radiosity Lighting

- Lighting levels need to be balanced between exterior and interior environments
- The relative ratio is critical for achieving exposure changes when going from one to the other

- Example Ambient occlusion lighting pass for an exterior environment

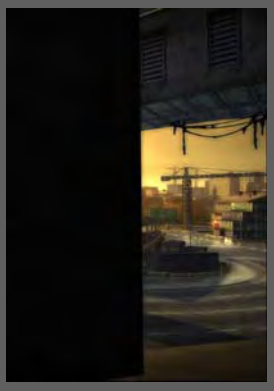
High Dynamic Range Lighting and Bloom, Exposure Adjustment
Lessons: Tunnel Lighting Exposure

- HDR tunnel lights reflect onto the car

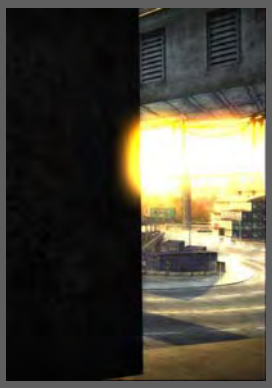
- Exposure begins to auto-adjust

- **Interior lighting exposure has to be correct relative to the exterior lighting, otherwise no bloom would occur when exiting a tunnel.**

Using HDRI to select Exposure and Bloom



- HDRI and Bloom **OFF**



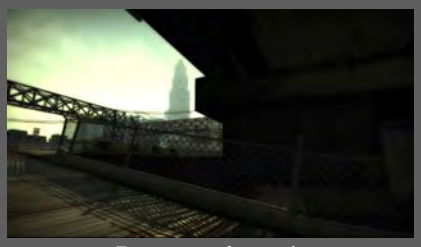
- HDRI and Bloom **ON**

- Because Blooming affects it's surrounding pixels it needs to be calculated after the exposure range has been determined and the overexposed area isolated.
- The dark part of the image above causes the virtual iris to open up causing the sky to be overexposed and bloom

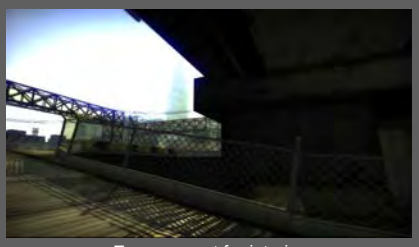
Using HDRI to select Exposure and Bloom

- Exposure set for exterior

- Exposure set for interior

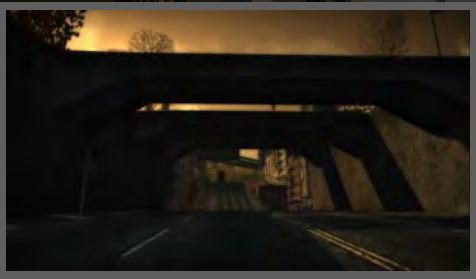


- Exposure set for exterior



- Exposure set for interior

Using HDRI to select Exposure



- Sunset with exposure adjustment **OFF**

- Sunset with exposure adjustment **ON**

- Dynamic Time of Day changes the level of light in an exterior environment
- With Auto-iris adjustment there is less concern about certain times of day or weather conditions becoming too dark or too bright
- The relative lighting between shade and sunlit areas, however, does need to be tuned accurately

Auto-Iris Exposure adjustment

- Auto-Iris adjustment **OFF**

- Auto-Iris adjustment **ON**

- Bloom **OFF**

- When the exposure is set close to mid-range levels, toggling Auto-iris adjustment will only slightly change the exposure
- Lens Bloom effect can be turned off separately

HDRI and Interior lighting

- Exposure set for exterior

- Exposure Auto-adjusted for interior

- Example on the right shows dangers of letting the exposure get too bright
- Compression artifacts and texture imperfections that are normally too dark to notice become glaringly visible

- Exposure set too high

High Dynamic Range Lighting and Bloom, Exposure Adjustment
Pitfalls:

What is wrong with the car? Is the car too dark?

- HDR lighting, Bloom, and Dynamic Iris **ON** -

High Dynamic Range Lighting and Bloom, Exposure Adjustment

Pitfalls:

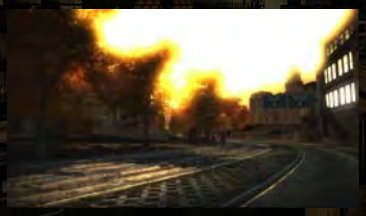
...No, the background was too bright, dynamic Iris was compensating, making the car appear too dark.

- HDR lighting, Bloom, and Dynamic Iris **ON** -

Exposure Adjustment and Bloom Threshold

- Bloom Threshold 0.5

- Bloom Threshold 0.2



- Bloom Threshold 0.05

- Bloom Threshold 0.01

Reduction of the Bloom Threshold will cause more contamination of the dark areas from the bright sky

Special Thanks

EA Blackbox Team

XBox 360 Rendering:

Greg D'Esposito
Gareth Jones
Colin O'Connor
David Lam

Producers:

Michael Mann
Larry LaPierre

Associate Art Director:

Eduardo Agostini

Contrast Processing

Rafal Mantiuk

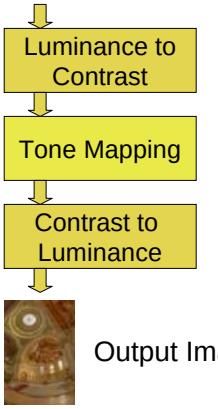
Tone Mapping in Contrast Domain 1/3

- Operate on image contrast instead of luminance
- Similar to the gradient methods, e.g. [Fattal'02]

Mantiuk et al., A Perceptual Framework for Contrast Processing of HDR Images. TAP 2006.

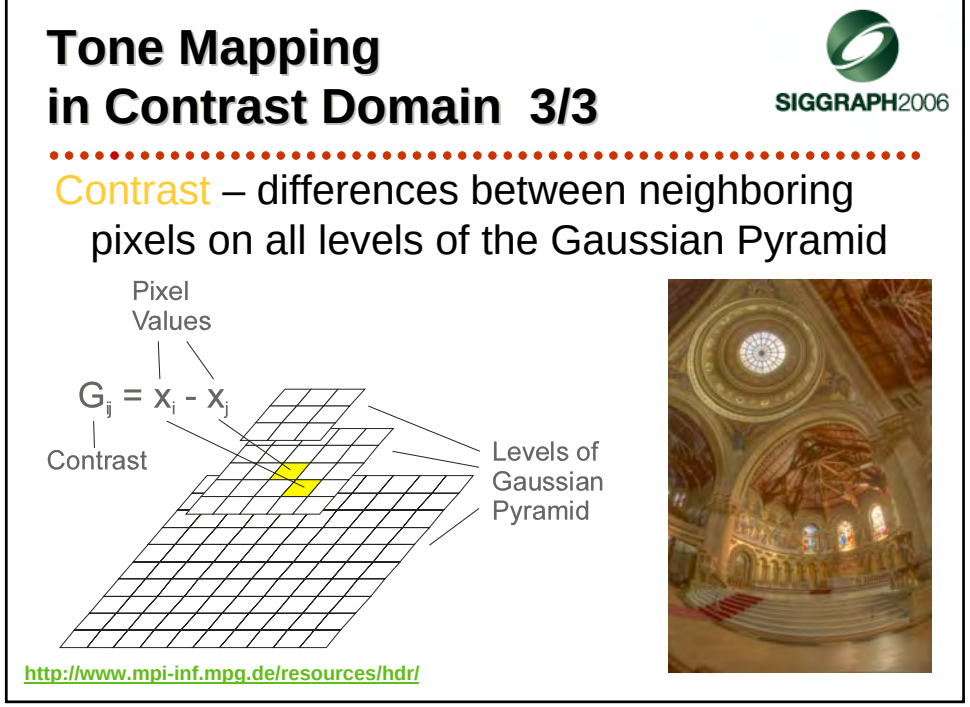
Tone Mapping in Contrast Domain 2/3

Input Image



Output Image

Tone Mapping in Contrast Domain 3/3



Contrast – differences between neighboring pixels on all levels of the Gaussian Pyramid

http://www.mpi-inf.mpg.de/resources/hdr/

Lightness Perception

Grzegorz Krawczyk

Lightness Perception in Tone Reproduction

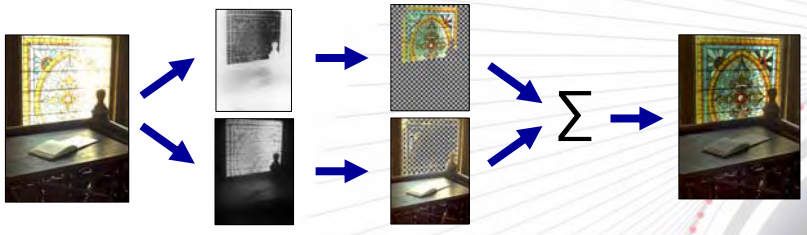
Model the lightness perception corresponding to conditions in the real world.

The theory:

- An Anchoring Theory of Lightness Perception [Gilchrist et al. 1999]

Key concepts:

- Frameworks – areas of common illumination
- Anchoring – luminance → lightness mapping



Lightness Perception in Tone Reproduction for HDR Images
Krawczyk et al., Computer Graphics Forum vol.24, 2005.

Frameworks

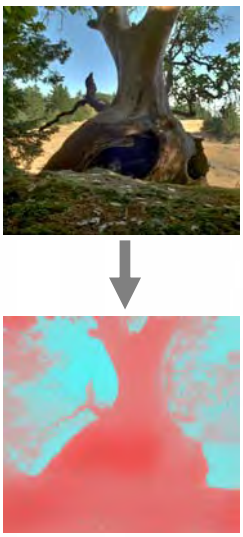


Figure: Frameworks of common illumination.

Frameworks allow for lightness estimation in complex images.

Perceptual organization:

- semantic grouping
- good continuation
- grouping of illumination
- proximity

Frameworks:

- defined by probability maps
- each pixel may belong to several frameworks

Anchoring

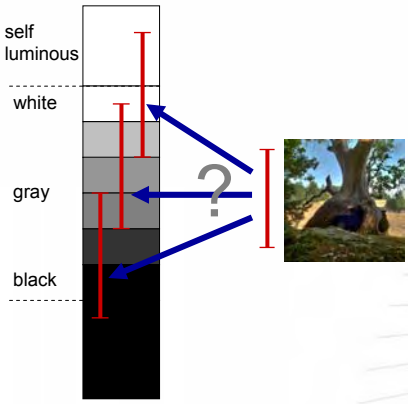


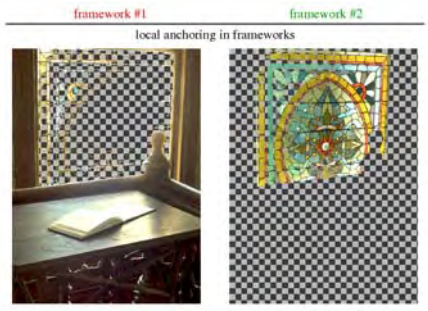
Figure: Perceived gray shades.

Mapping between
luminance value and
value on a scale of perceived gray shades.

Two possibilities:
- Anchoring to middle-gray
- Anchoring to white

Experimental evidence favors anchoring to white.

Estimation of Anchor



Anchoring to white rule:
- tendency of the highest luminance to appear white
- tendency of the largest area to appear white

Self-luminosity:
- small white disc surrounded by a large dark area appears luminous

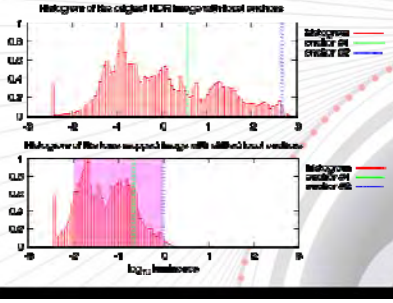
Area related approach to anchor estimation:
- Remove the top 5% of pixels of highest luminance
- Measure the highest luminance of the rest of the pixels

Net Lightness Calculation

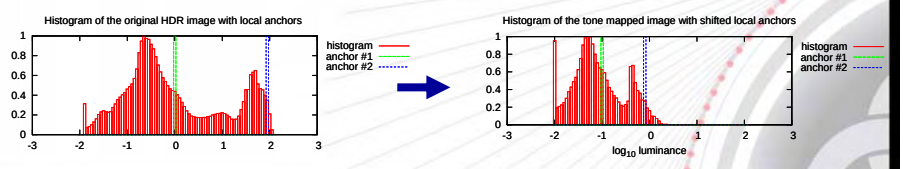
Shift original luminance (Y)

- according to local anchors (W)
- proportionally to belongingness (P)

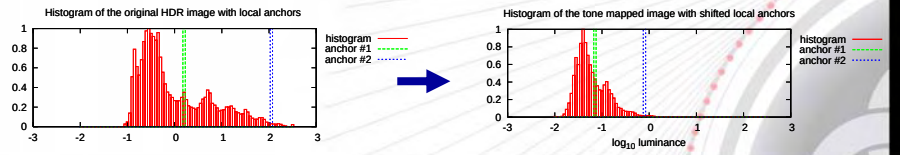
$$L(x,y) = Y(x,y) - \sum_i W_i \cdot P_i(x,y)$$



Tone Mapping: Tree



Tone Mapping: Desk



Conclusions

- Computational model of lightness perception theory
- Formalize method for extracting frameworks
- Application of the anchoring theory to tone reproduction
- Simulation of lightness constancy failure
- Lightness calculation in tone mapping

Subband Encoding of High Dynamic Range Imagery

Greg Ward
Sunnybrook Technologies

Maryann Simmons
Walt Disney Feature Animation

Abstract

The transition from traditional 24-bit RGB to high dynamic range (HDR) images is hindered by excessively large file formats with no backwards compatibility. In this paper, we propose a simple approach to HDR encoding that parallels the evolution of color television from its grayscale beginnings. A tone-mapped version of each HDR original is accompanied by restorative information carried in a subband of a standard 24-bit RGB format. This subband contains a compressed *ratio image*, which when multiplied by the tone-mapped foreground, recovers the HDR original. The tone-mapped image data may be compressed, permitting the composite to be delivered in a standard JPEG wrapper. To naïve software, the image looks like any other, and displays as a tone-mapped version of the original. To HDR-enabled software, the foreground image is merely a tone-mapping suggestion, as the original pixel data are available by decoding the information in the subband. We present specifics of the method and the results of encoding a series of synthetic and natural HDR images, using various published global and local tone-mapping operators to generate the foreground images. Errors are visible in only a very small percentage of the pixels after decoding, and the technique requires only a modest amount of additional space for the subband data, independent of image size.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image generation – Display Algorithms
I.4 [Image Processing and Computer Vision]: I.4.10 Image Representation

Keywords: high dynamic range image formats, lossy compression, image processing

1. Introduction

Visible light in the real world covers a vast dynamic range. Humans are capable of simultaneously perceiving over 4 orders of magnitude (1:10000 contrast ratio), and can adapt their sensitivity up and down another 6 orders. Conventional digital images, such as 24-bit *sRGB* [Stokes et al. 1996], hold less than 2 useful orders of magnitude. Such formats are called "output-referred" standards because they are tailored to what can be displayed on a common CRT monitor – colors outside this limited gamut are not represented. A "scene-referred" standard is designed instead to represent colors and intensities that are visible in the real world, and though they may not be rendered faithfully on today's output devices, they will be visible on displays in the near future [Seetzen et al. 2003]. Such image representations are referred to as extended gamut or *high dynamic range* (HDR) formats, and a few alternatives have been introduced over the past 15 years, mostly by the graphics research and special effects communities.

Unfortunately, none of the existing or proposed HDR formats supports lossy compression, and only one comes in a conventional image wrapper. These formats yield prohibitively large images that can only be viewed and manipulated by specialized software. Commercial hardware and software developers have been slow to embrace scene-referred standards due to their demands on image capture, storage, and use. Some digital camera manufacturers attempt to address the desire for greater exposure control during processing with their own proprietary RAW formats, but these camera-specific encodings fail in terms of image archiving and third-party support. They are convenient for the manufacturers, but for no one else.

What we really need is a compact image that looks and displays like an output-referred JPEG, but holds the extra information needed to enable it as a scene-referred standard. Future HDR cameras will then be able to write to this format without fear that the software on the receiving end won't know what to do with it. Conventional image manipulation and display software will see only the tone-mapped version of the image, gaining some benefit from the HDR capture due to its better exposure. HDR-enabled software will have full access to the original dynamic range recorded by the camera, permitting large exposure shifts and contrast manipulation during image editing. Establishing such a standard will provide a smooth upgrade path for manufacturers and consumers alike.

1.1. Background

High dynamic range imaging goes back many years. A few early researchers in image processing advocated the use of logarithmic encodings of intensity (e.g., [Jourlin & Pinoli 1988]), though it was global illumination that brought us the first standard. A space-efficient format for HDR images was introduced in 1989 as part of the *Radiance* rendering system [Ward 1991; Ward 1994]. However, the *Radiance* RGBE format was not widely used until HDR photography and environment mapping were developed by Debevec [Debevec & Malik 1997; Debevec 1998]. About the same time, Ward Larson [1998] introduced the LogLuv alternative to RGBE and distributed it as part of Leffler's public TIFF library [Leffler et al. 1999]. The LogLuv format has the advantage of covering the full visible gamut in a more compact representation, whereas RGBE is restricted to positive primary values. A few graphics researchers adopted the LogLuv format, but most continued to use RGBE (or its extended gamut variant XYZE), until Industrial Light and Magic made their EXR format available in 2002 [Kainz et al. 2002]. The OpenEXR library uses the same basic 16-bit floating point data type as modern graphics cards, and is poised to be the new favorite in the special effects industry. Other standards have also been proposed or are in the works, but they all have limited dynamic range relative to their size (e.g., [IEC 2003]). None of these standards is backwards compatible with existing software.

The current state of affairs in HDR imaging parallels the development of color television after the adoption of black and white broadcast. A large installed base must be accommodated as

well as an existing standard for transmission. The NTSC solution introduced a subband to the signal that encoded the additional chroma information without interfering with the original black and white signal [Jolliffe 1950]. We propose a similar solution in the context of HDR imagery, with similar benefits.

As in the case of black and white television, we have an existing, de facto standard for digital images: output-referred JPEG. JPEG has a number of advantages that are difficult to beat. The standard is unambiguous and universally supported. Software libraries are free and widely available. Encoding and decoding is fast and efficient, and display is straightforward. Compression performance for average quality images is competitive with more recent advances, and every digital camera writes to this format. Clearly, we will be living with JPEG images for many years to come. Our chances of large scale adoption increase dramatically if we can maintain backward compatibility with this standard.

Our general approach is to introduce a subband that accompanies a tone-mapped version of the original HDR image. This subband is compressed to fit in a metadata tag that will be ignored by naïve applications, but can be used to extract the HDR original in enabled software. We utilize JPEG/JFIF as our standard wrapper in this implementation, but our technique is compatible with any format that provides client-defined tags (e.g., TIFF, PNG, etc.).

The Method section of our paper describes the basic idea behind HDR subband encoding, followed by a more detailed description of the steps involved and our trial implementation. The Results and Discussion section presents a set of example images and examines sources of error in our encoding. We end with our conclusions and future directions.

2. Method

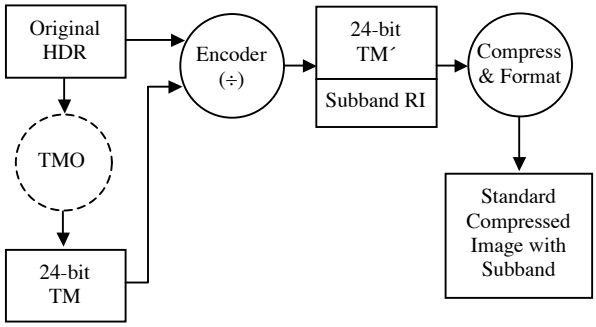


Figure 1. High level HDR subband encoding pipeline.

Figure 1 shows an overview of our HDR encoding pipeline. We start with two versions of our image: a scene-referred HDR image, and an output-referred, tone-mapped version. If we like, we can generate this tone-mapped version ourselves, but in general this is a separable problem, because our technique is designed to work with multiple operators. The encoding stage takes these two inputs and produces a composite, consisting of a potentially modified version of the original tone-mapped image, and a subband ratio image that contains enough information to reproduce a close facsimile of the HDR original. The next stage compresses this information, offering the tone-mapped version as the JPEG base image, and storing the subband as metadata in a standard JFIF wrapper.

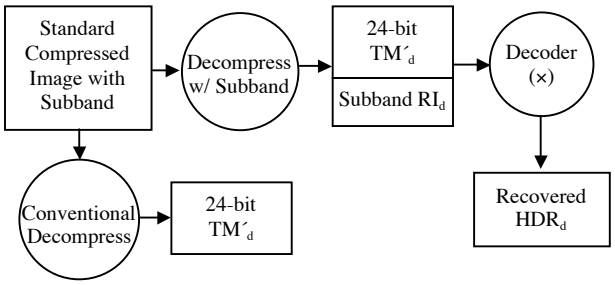


Figure 2. Alternate decoding paths for compressed composite.

Figure 2 shows the two possible decoding paths. The high road decompresses both the tone-mapped foreground image and the subband, delivering them to the decoder to recover the HDR pixels. Naïve applications follow the low road, ignoring the subband in the metadata and reproducing only the tone-mapped foreground image.

In the simplest incarnation of this method, the encoder divides the HDR pixels by the tone-mapped luminances, producing an 8-bit ratio image that is stored as metadata in a standard JPEG compressed image. Decoding follows decompression with a multiplication step to recover the original HDR pixels. Unfortunately, the simplest approach fails for pixels that are mapped to black or white by the tone-mapping operator (TMO), and the limited size of metadata in JPEG makes subband compression a challenge. These are two important issues we must address with our technique.

2.1. The Ratio Image

JPEG/JFIF offers us a limited subband channel, called the "application markers." Sixteen application markers exist in the JFIF specification, three of which are spoken for. A marker may hold up to 64 Kbytes of data, regardless of the image dimensions. This is an important limitation in our method – we want to keep our subband data under 64K, independent of image size. The foreground signal in our application is a tone-mapped version of the original HDR image. This output-referred image is stored in the usual 8×8, 8-bit blocks using JPEG's lossy DCT compression [Wallace 1991]. The subband encodes the information needed to restore the HDR original from this compressed version.

Let us assume that our selected tone-mapping operator possesses the following properties:

A. The original HDR input is mapped smoothly into a 24-bit output domain, with no components being rudely clamped at 0 or 255.

B. Hue is maintained at each pixel, and mild saturation changes can be described by an invertible function of input and output value.

Most tone-mapping operators for HDR images have the first property as their goal, so this should not be a problem. If it is, we can override the operator by locally replacing each clamped pixel with a lighter or darker value that fits the range. Similarly, we can enforce the second property by performing our own color desaturation, using the tone-mapping operator as a guide only for

luminance changes. Most tone-mapping operators address color in a very simple way, if they consider it at all.[1] Exceptions are found in operators that simulate human vision (e.g., [Pattanaik et al. 1998]), whose support we leave as future work.

Given these restrictions, a ratio image may be computed by dividing the HDR original luminance at each pixel by the tone-mapped output luminance:

$$RI(x,y) = \frac{L(HDR(x,y))}{L(TM(x,y))} \quad (1)$$

The ratio image is compressed and sent in our subband along with the saturation formula. The tone-mapped version is then encoded as the foreground image, modified as necessary to avoid clamping. During reconstruction, the ratio image is multiplied by the foreground image to recover the original HDR luminance values. Color is then restored using the recorded saturation formula. (See Appendix for details.)

2.2. Subband Compression

Obviously, we will not meet our goal of fitting the subband into 64 Kbytes if we send the ratio image along uncompressed. Since our goal is to encapsulate the subband in a JPEG wrapper, it would be most convenient if we could compress the ratio image using JPEG as well.[2] If we could also fix the maximum size of the ratio image, we could avoid the problem of needing more space or greater compression for larger input images.

Figure 3 shows the Memorial Church image, tone-mapped using the global zone operator of Reinhard et al. [2002]. Ratio values are mapped into an 8-bit range by a log encoding that captures the extrema, as we show on the right. Because the original image is only 512x768 pixels, we can compress our ratio image into 48 Kbytes with a JPEG quality setting of 90, without resorting to downsampling. We see a before and after close-up of the recovered HDR image near the window, where luminance compression is greatest, in Figure 4. A comparison of the dark ceiling is shown on the left. Qualitatively, we are able to reproduce the original HDR pixels using this method, although JPEG artifacts are beginning to appear. Clearly, we cannot push this approach to much larger images and hope to stay within the 64 Kbyte limit.

Figure 5 shows a 2048x1536 image of the Dyrham Church next to its reduced ratio image. In this composite, we have downsampled the subband image to 768x576 and compressed it to 41 Kbytes using the same JPEG quality setting of 90. Recovery is quite good in darker regions, as shown on the right of Figure 6, but we start to lose focus on bright boundaries, such as the window panes shown on the left. This is due to blur in the ratio image introduced when we upsample prior to multiplication. We need some method of recovering the high frequencies we lost when we downsampled the ratio image. In the following sections, we present two alternate recovery methods: precorrecting the foreground image, and postcorrecting the ratio image.

[1] We can use a fitting function to match desaturation of the exemplar tone-mapped image if it is unknown.

[2] The Exif files produced by digital cameras use this trick for storing thumbnail images.

Figure 3. Memorial Church shown tone-mapped with Reinhard's global operator (left) along with the log-encoded ratio image needed for HDR recovery (right).[3]

Figure 4. Linear displays of original (top) and recovered (bottom) HDR images. On the left is a close-up of the ceiling, and on the right is a close-up of the rightmost windows.[3]

Figure 5. The Dyrham Church image rendered with Reinhard's global operator, shown next to the corresponding downsampled ratio image.

[3] Figures 3, 4, 6, 7, 8, 9, 10, 12, and 13 are repeated on the color plate.

Figure 6. Details of the image recovered using the downsampled subband.[3]

2.3. Precorrection of Foreground Image

One way to retain the high frequency information is to precorrect the foreground image based on the compressed subband. After computing the subband as above, we redivide the HDR original by the decompressed and upsampled ratio image to get a modified foreground image:

$$TM' = \frac{HDR}{RI_d} \tag{2}$$

Substituting this modified foreground image for the original effectively undoes the damage of lossy compression and downsampling. The left image in Figure 7 shows the lattice window after applying Reinhard's global tone-mapping operator to the Dyrham Church image. On the right, we see the result of dividing the HDR original by the computed ratio image. By construction, the result of multiplying this modified foreground image by our downsampled ratio image will be very close to the original, even after JPEG transmission. Figure 8 shows a linear rendition of the HDR original next to the recovered result. The precorrection method is a good complement to Reinhard's operator because it restores some of the contrast lost at the high end. However, the sharpening produced by this technique may be undesirable when the TMO has already produced an optimal foreground image. In such cases, we may prefer not to modify the foreground image during encoding, choosing instead to recover high frequencies in a post-process.

Figure 7. Reinhard's tone-mapping operator applied to the Dyrham Church image, before and after modification by the downsampled ratio image in our precorrection method.[3]

Figure 8. Linear comparison of HDR image recovery improved by precorrecting the tone-mapped foreground image.[3]

2.4. Postcorrection of Ratio Image

If we invest the time to generate a high quality foreground image with a sophisticated TMO, we may be unwilling to accept the effects of precorrection as described in the previous section. For a sufficiently high resolution original, recomputing TM´ using Eq. (2) can result in small halos that are visible in close-ups, as shown in the sunset of Figure 9.

Figure 9. Reverse gradients visible on a modified high-resolution (3000x1700) foreground image. The original bilateral filter tone-mapping is shown on the left inset.[3]

Ideally, we would like to preserve the tone-mapped representation in the foreground image without losing resolution in the recovered HDR result. One way to achieve this without exceeding our 64 Kbyte subband limit is to synthesize the missing high frequencies in our resampled ratio image. Since we have a full-resolution foreground image, we can use this as a guide for where edges belong in the ratio image. If the frequency content of the foreground image and the ratio image before downsampling were the same, one could recover the high frequencies in the resampled ratio image with the following simple formula:

$$RI_{synth} = RI_d \frac{L(TM)}{L(TM_r)}$$

In this equation, $L(TM_r)$ is the luminance of the tone-mapped foreground image, resampled in the same way as the encoded ratio image. Of course, there is no guarantee that the frequency content of the foreground and ratio images are the same, especially using a TMO like the bilateral filter, which attempts to

preserve local detail [Durand & Dorsey 2002]. A better approximation therefore attenuates the effect by the ratio between the local variance of RI_d and TM_r. If the ratio image has little or no variance after resampling, then it probably had little in the way of high frequencies beforehand. This improved frequency correction may be written as follows:

$$RI_{synth} = RI_d \cdot \left[\frac{L(TM)}{L(TM_r)}\right]^{\sigma} \qquad (3)$$

$$where: \quad \sigma = \frac{\mathrm{var}(RI_d)}{\mathrm{var}(L(TM_r))}$$

The relative variance is computed over a small neighborhood, equal to the resampling radius, and σ is set to 0 if var($L(TM_r)$) is less than our compression-decompression error. In practice, we do not allow the exponent σ to exceed 1, as this can cause overshooting in the output. We define relative variance as the difference between the maximum and minimum in a neighborhood, divided by the central value. This is a simple application of resolution enhancement via example learning. Much more sophisticated techniques have been developed by the vision community [Baker & Kanade 2002].

Figure 10 shows the results of applying high frequency synthesis to the Napa Valley image. The foreground image for this encoding was computed using a bilateral filter, hence there are significant discrepancies between the tone-mapped version and the ratio image. As we can see, the approximation in Eq. (3) does a reasonable job of restoring the high frequency information that has been lost during the resampling process, without introducing objectionable artifacts. However, the errors in the recovered HDR image are greater with this method than those of Eq. (2), so the application should decide whether the benefits of a cleaner foreground image are worth the costs. A flag must be passed in the subband, indicating whether the image was precorrected by Eq. (2), or should be postcorrected by Eq. (3).

Figure 10. The center is the original HDR image with a linear tone-mapping. On the left is our uncorrected ratio image multiplied by the foreground image. On the right, we postcorrected the ratio image with synthetic high frequencies.[3]

3. Results and Discussion

We tested our encoding on the fifteen HDR images shown in Figure 11. Table 1 lists each image size and dynamic range in base 10 units. (E.g., 4.8 is 1:$10^{4.8}$ or 1:63,000 dynamic range.)

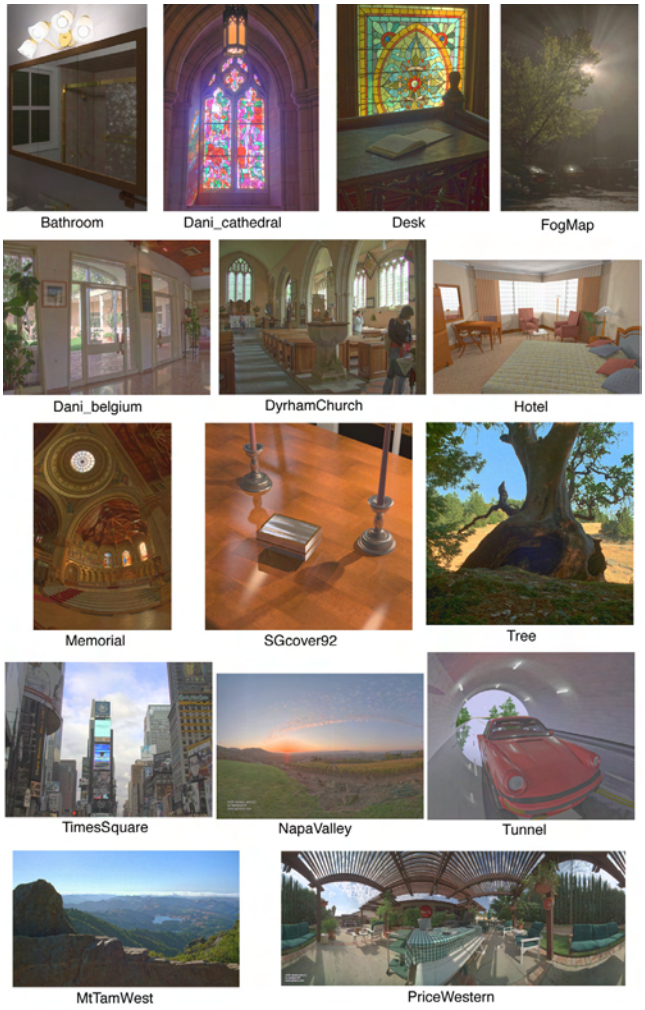


Figure 11. Test image set.

Image	Size	Dynamic Range	Source
Bathroom	346×512	4.8	*Radiance*
Dani_belgium	1025×769	5.8	digital
Dani_cathedral	767×1023	4.8	digital
Desk	644×874	5.2	film
DyrhamChurch	2048×1536	4.0	digital
FogMap	751×1130	4.1	film
Hotel	3000×1950	4.7	*Radiance*
MtTamWest	1214×732	4.1	film
Memorial	512×768	5.5	film
NapaValley	3025×2129	5.3	Spheron
PriceWestern	3272×1280	3.7	Spheron
SGcover92	1024×1024	4.7	*Radiance*
TimesSquare	2272×1704	3.6	digital
Tree	928×906	4.1	film
Tunnel	5462×4436	9.2	*Radiance*

Table 1. Test image sizes. Images were either synthetic (*Radiance* renderings), or captured (multiple film or digital exposures, or panoramic SpheronVR scans).

Eleven images are captures of natural scenes, and four are synthetic. The dynamic ranges of the images are between 3.6 and 9.2 orders of magnitude, with 4.9 being average. The smallest

image is 346×512; the largest is 5462×4436, and the median is 0.9 megapixels.

We tested four different tone-mapping operators to produce the foreground image: Ward Larson et al.'s histogram method [1997], Reinhard et al.'s global zone method [2002], Fattal et al.'s gradient operator [2002], and Durand & Dorsey's bilateral filter method [2002].

Our test procedure was simple: encode then decode each image using the selected TMO and JPEG compression parameters, then compare the recovered HDR image to the original. We tested two JPEG compression levels (quality settings) on the foreground image: 90 and 100, using Tom Lane's public JPEG implementation. The ratio image was always compressed with the highest JPEG quality setting that kept the result under 60 Kbytes, leaving ample room for other subband data. If the original image was greater than 400,000 pixels, the ratio image was downsampled to this size before compression.

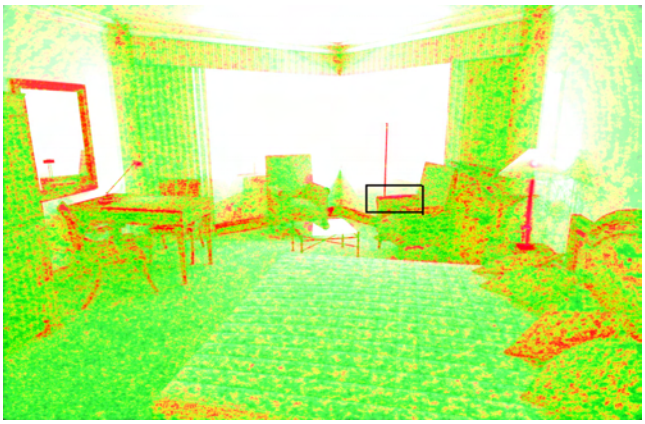


Figure 12. Sample VDP output for Hotel image. Red shows threshold where difference detection probability exceeds 0.75.[3] (Green $p \le 0.5$, Yellow $p \le 0.63$, Purple $p > 0.95$)

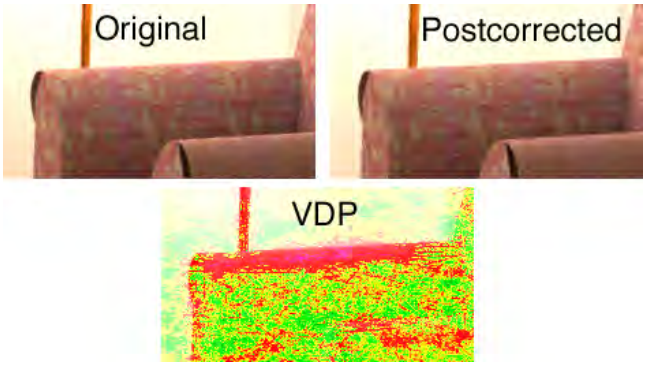


Figure 13. Close-up of Hotel chair's arm from box in Figure 12 with VDP visualization.[3]

To compare our decoded HDR images to their corresponding originals, we employed Daly's Visible Differences Predictor (VDP) [1993]. This metric tells us what percentage of our decoded image pixels are likely (probability $p > 0.75$) to be perceived as different from the originals under standard viewing conditions. Figure 12 visualizes the VDP output for the postcorrected Hotel encoding using the bilateral filter with JPEG quality set to 100. Figure 13 shows a close-up of the indicated region of this image, verifying the correlation between actual perceptible differences and VDP on the arm of the chair. Empirically, we found VDP to be an excellent predictor of where we could see differences in our images.

Our results are summarized in Table 2, where we have averaged the VDP percentages over all images except "Tunnel," which we considered an outlier. We see a sizeable discrepancy in the VDP results for different tone-mapping operators, and for precorrection versus postcorrection. The JPEG quality setting also made a difference, as one would expect.

TMO	Quality	VDP (pre)	VDP (post)
Bilateral Filter	90 100	0.93% 0.02%	5.4% 1.8%
Reinhard Global	90 100	2.5% 0.09%	4.7% 2.8%
Histogram Adj.	90 100	5.9% 0.63%	21% 17%
Gradient	90 100	7.5% 3.0%	36% 34%

Table 2. Percentage of perceptibly different pixels summed over all images for four tone mapping operators using VDP metric on precorrected and postcorrected encodings. Two JPEG quality settings were tested for each foreground image.

Image	Qual.	CR	VDP (pre)	VDP (post)
Bathroom	90 100	9.6 5.7	1.3% 0.00%	0.00% 0.00%
Dani_belgium	90 100	14.4 5.3	0.32% 0.01%	6.2% 3.0%
Dani_cathedral	90 100	12.3 4.5	0.24% 0.00%	2.7% 1.1%
Desk	90 100	10.7 4.6	0.06% 0.02%	2.7% 1.8%
DyrhamChurch	90 100	21.3 6.4	0.03% 0.00%	3.2% 1.5%
FogMap	90 100	16.6 5.9	0.90% 0.00%	1.2% 0.22%
Hotel	90 100	26.0 7.9	0.14% 0.04%	7.1% 2.8%
Memorial	90 100	10.3 4.4	5.3% 0.01%	12.3% 0.76%
MtTamWest	90 100	14.2 4.9	2.4% 0.00%	4.5% 0.17%
NapaValley	90 100	26.3 6.9	0.01% 0.01%	1.9% 1.6%
PriceWestern	90 100	14.5 5.0	2.1% 0.16%	27.3% 9.8%
SGcover92	90 100	18.3 6.2	0.01% 0.00%	0.50% 0.47%
TimesSquare	90 100	22.4 6.7	0.08% 0.01%	4.0% 1.4%
Tree	90 100	9.6 3.9	0.09% 0.00%	1.9% 0.42%
Tunnel	90 100	23.5 7.3	5.3% 3.0%	53% 45%

Table 3. Compression ratios and VDP percentages for each of the test images, mapped with the bilateral filter TMO.

The bilateral filter proved to be an excellent fit to our encoding scheme, with lower associated errors than the other techniques. The Reinhard TMO also fared reasonably well, but the histogram and gradient methods generated many visible errors, especially when coupled with our postcorrection method. Most of these errors occurred in darker regions, where pixels were mapped to very small values. This is not a general indictment of these operators – it simply means they are less suited for splitting information between foreground and ratio images as our method requires.

Our results using the bilateral filter are broken out for each image in Table 3. When using a precorrected foreground image with low JPEG compression, only the Tunnel image had more than a small percentage of pixels where differences were discernable. These percentages got larger for the lower quality setting, reaching 5.3% for the Memorial image, but remained acceptable for most of the others. (We found 2% to be a reasonable cut-off for a good side-by-side match.) In all but a few cases, the postcorrected results were acceptable with the higher JPEG quality, but VDP reached a few percent for about half our images on the lower setting, and showed a real failure on the Tunnel image.

We measured our compression performance relative to an uncompressed RGBE original (i.e., 32 bits/pixel). The compression ratios varied so little between precorrected and postcorrected encodings that we averaged the two in Table 3. At a foreground quality setting of 90, we saw compression ratios between 9.6 and 26.3, with an average performance of 17.0. Taking a typical example, the Fog Map image compressed from its original 3.2 Mbytes down to 200 Kbytes. At a foreground quality setting of 100, we saw compression ratios ranging from 3.9 to 7.9, with an average performance of 5.7. Unsurprisingly, the smallest compression ratios were associated with the smallest original images. The highest compression ratios were associated with the Napa Valley image. For comparison, the average compression ratio achieved by the most sophisticated, lossless HDR image format is 1.6 on this data set, using OpenEXR's "PIZ" wavelet encoding [Kains et al. 2002].

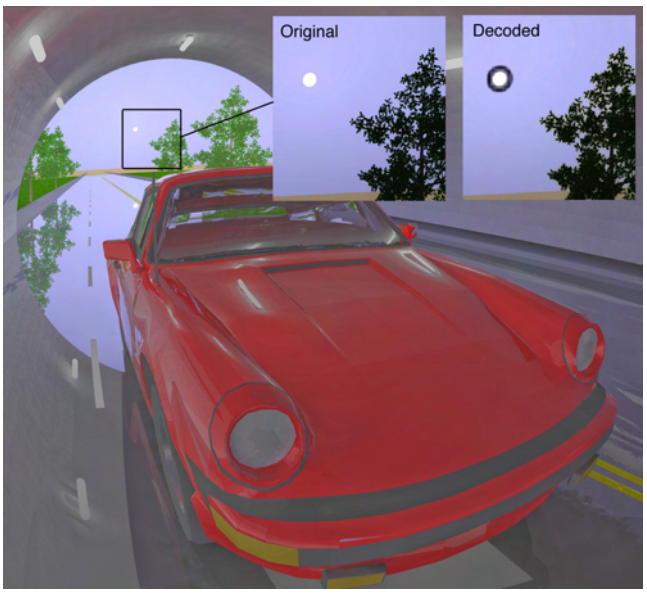


Figure 14. Close-up showing details lost in small regions of the Tunnel due to downsampling of the ratio image.

We would expect our errors to increase somewhat for larger images with higher dynamic range, since the ratio image must be scaled to fit both these input parameters. Examining our bilateral filter results, we searched for correlations between VDP and image size, and VDP and dynamic range, but found no significant trends in our data, with the exception of a single outlier: the Tunnel. This is really the worst case image for our algorithm – not only is it the largest (23 Mpixel), it also has the greatest dynamic range (a billion to 1), and since it came unfiltered out of a stochastic ray-tracing program, it contains abnormal amounts of high frequency, HDR pixel noise. Our postcorrection algorithm had visible errors over half the image, and even precorrection could not cope where sampling discontinuities exceeded the 8-bit carrying capacity of the foreground image. For example, Figure 14 shows a halo around the sun that derives from a huge luminance discontinuity – almost 5 orders of magnitude between neighboring pixels. This is an extreme jump that could only be generated synthetically, and demands an encoding that carries 16 bits at every pixel. Therefore, we may wish to consider ways to bypass the restriction on the ratio image resolution for such extreme images, possibly working around the 64 Kbyte limit for JPEG markers by stringing multiple markers in series, which is permitted by the JFIF standard.

4. Conclusion and Future Directions

By providing a lossy, high dynamic range image format that is backwards compatible with existing JPEG software, we remove an important barrier to the adoption of HDR imaging technology by digital camera manufacturers and web content providers. The subband encoding method we presented couples a high quality, tone-mapped (i.e., output-referred) foreground image with metadata that enables HDR software to recover the original, scene-referred luminances at 16-bit resolution. Naïve applications see only the tone-mapped version, which still encompasses the larger dynamic range, albeit with 8-bit precision. Color is encoded in the foreground image as well, which HDR-enabled applications may resaturate to access a wider gamut than standard RGB. With the current prototype implementation, we obtain predicted visible differences at only a very small percentage of the pixels over a wide range of HDR image sizes, adding just 64 Kbytes of subband data to each tone-mapped JPEG.

Although it will not affect the critical standardization of the subband format and decoding method, there is more work to do in finding an optimal tone-mapping operator for the encoding. We found Durand and Dorsey's bilateral filter [2002] to behave quite well, and Reinhard et al.'s global operator [2002] to perform adequately, but further testing is needed. We would like to incorporate an operator that is fast, robust, and completely automatic. Further, we would like to explore tuning of this operator to minimize problems in the ratio image that could show up as artifacts in the decoded HDR result.

Although our initial implementation is tied to the standard JPEG encoding, there is no reason the same separation of tone-mapped and ratio image could not be applied within other existing and emerging image standards. The advantage to this approach over a direct extension to incorporate an HDR color space is two-fold. Firstly, an output-referred image is immediately available to all applications, avoiding the need for a potentially time-consuming tone-mapping step prior to viewing. Secondly, separate control is possible for the fidelity/bitrate of the output-referred and scene-referred versions, permitting application-tuned encodings. As an

added benefit, web-savvy formats such as PNG and JPEG 2000 can send the ratio image as a separate bundle only when requested by the client, eliminating the associated cost of this additional information where it is not needed. In other words, there may be benefits to backwards compatibility looking forward as well.

5. Acknowledgements

The VDP implementation was provided by Karol Myszkowski, and Dave Shreiner helped process all the data. Thanks to Paul Debevec, Dani Lischinski, Erik Reinhard, Jack Tumblin, Spheron Corporation, and ILM for lending their HDR images. Thanks also to Scott Daly for useful insights into his VDP metric.

6. References

ASHIKHMIN, M. 2002. A Tone Mapping Algorithm for High Contrast Images. In *Proceedings of 13th Eurographics Workshop on Rendering*, 145-156.

BAKER, S & KANADE, T. 2002. Limits on super-resolution and how to break them. IEEE Transactions on Pattern Analysis and Machine Intelligence., 24(9):1167-1183, September. 2002.

CHIU, K. HERF, M., SHIRLEY, P., SWAMY, M., WANG, C., and ZIMMERMAN, K., 2002. A Tone Mapping Algorithm for High Contrast Images. In *Proceedings of 13th Eurographics Workshop on Rendering*, 245-253.

DALY, S. 1993. The Visible Differences Predictor: An Algorithm for the Assessment of Image Fidelity. In *Digital Images and Human Vision*, A.B. Watson, editor, MIT Press, Cambridge, Massachusetts.

DEBEVEC, P., and MALIK, J. 1997. Recovering High Dynamic Range Radiance Maps from Photographs. In *Proceedings of ACM SIGGRAPH 1997*, 369-378.

DEBEVEC, P. 1998. Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography. In *Proceedings of ACM SIGGRAPH 1998*, 189-198.

DURAND, F., and DORSEY, J. 2002. Fast Bilateral Filtering for the Display of High-Dynamic Range Images. *ACM Transactions on Graphics*, 21, 3, 249-256.

FATTAL, R., LISCHINSKI, D., and WERMAN, M. 2002. Gradient Domain High Dynamic Range Compression. *ACM Transactions on Graphics*, 21, 3, 257-266.

IEC. 2003. 61966-2-2. Extended RGB colour space – scRGB, *Multimedia systems and equipment – Colour measurement and management* – Part 2-2: Colour management.

JOLLIFFE, C.B., 1950. Answers to Questions about Color Television. members.aol.com/ajaynejr/rca2.htm.

JOURLIN, M., PINOLI, J-C., 1988. A model for logarithmic image processing, *Journal of Microscopy*, 149(1), pp. 21-35.

KAINS, F., BOGART, R., HESS, D., SCHNEIDER, P., ANDERSON, B., 2002. *OpenEXR*. www.openexr.org/.

LEFFLER, S., WARMERDAM, F., KISELEV, A., 1999. *libTIFF*. remotesensing.org/libtiff.

MOON P., and SPENCER, D. 1945. The Visual Effect of Non-Uniform Surrounds. *Journal of the Optical Society of America*, 35, 3, 233-248.

PATTANAIK, S., FERWERDA, J., FAIRCHILD, M., and GREENBERG, D. 1998. A Multiscale Model of Adaptation and Spatial Vision for Realistic Image Display, In *Proceedings of ACM SIGGRAPH 1998*, 287-298.

REINHARD, E., STARK, M., SHIRLEY, P., and FERWERDA, J. 2002. Photographic Tone Reproduction for Digital Images. *ACM Transactions on Graphics*, 21,3, 267-276.

SEETZEN, H., WHITEHEAD, L., and WARD, G. 2003. A High Dynamic Range Display Using Low and High Resolution Modulators. In *Proceedings of the Society for Information Display International Symposium*, Baltimore, MD.

STOKES, M., ANDERSON, M., CHANDRASEKAR, S., and MOTTA, R. A 1996. Standard Default Color Space for the Internet. www.w3.org/Graphics/Color/sRGB.

TUMBLIN, J., and TURK, G. 1999. LCIS: A Boundary Hierarchy for Detail-Preserving Contrast Reduction. *ACM Trans. on Graphics*, 21, 3, 83-90.

WALLACE, G. 1991. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34, 4, 30-44.

WARD LARSON, G., RUSHMEIER, H., and PIATKO, C. 1997. A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes. *IEEE Trans. on Visualization and Computer Graphics*, 3, 4.

WARD LARSON, G. 1998. Overcoming Gamut and Dynamic Range Limitations in Digital Images. *Proc. of IS&T 6th Color Imaging Conf.*

WARD, G. 1991. Real Pixels. In *Graphics Gems II*, edited by James Arvo, Academic Press, 80-83.

WARD, G. 1994. The RADIANCE Lighting Simulation and Rendering System. , In *Proceedings of ACM SIGGRAPH 1994*, 459-472.

7. Appendix: Color Saturation and Gamut

Our treatment of color is straightforward. Any color with a primary component outside the "safe" range for JPEG encoding is scaled back into range, and the ratio image is adjusted for correct recovery. This works only for positive primary values. Negative primary values are legal in some HDR formats, and are indeed necessary for certain real chromaticities outside the standard, triangular RGB gamut.

To accommodate negative primaries, we apply a global desaturation to the image, pulling all colors in towards gray by an amount that is guaranteed to contain the entire visible gamut in the legal JPEG range. This may in fact be beneficial to the appearance of the foreground image, as dynamic range compression tends to leave the colors with an oversaturated appearance. The desaturation process is then reversed during the decoding, rendering the colors back into their original, full gamut glory. We use the following definition of input color saturation:

$$S \equiv 1 - \min(R,G,B)/Y \tag{4}$$

Y in the formula above is the luminance of red, green and blue taken together. We expect saturation to be greater than 1 for negative primaries. Zero saturation implies a neutral value, which is passed. Non-neutral values are desaturated with a two-parameter formula:

$$S' = \alpha \cdot S^{\beta} \tag{5}$$

The α parameter controls how much saturation we wish to keep in the encoded colors, and is generally ≤ 1. The β parameter controls the color "contrast," and is usually ≥ 1. This modified saturation is used with the original saturation from Eq. (4) to determine the encoded primary values. Below is the formula for the desaturated red channel:

$$R' = \left(1 - \frac{S'}{S}\right) \cdot Y + \frac{S'}{S} \cdot R \tag{6}$$

And similarly for the green and blue channels. Note that Y does not change under this transformation, and the primary that was smallest before is the smallest after. Resaturating the encoded color to get back the original pixel is done by inverting these equations. If the smallest primary value were blue for example, this inverse transformation would yield:

$$B = Y - Y \cdot \left(\frac{Y - B'}{\alpha Y}\right)^{1/\beta} \tag{7}$$

The red and green channels would then be determined by:

$$R = Y - \frac{(Y - R')}{\alpha}\left(1 - \frac{B}{Y}\right)^{1-\beta}, \quad G = Y - \frac{(Y - G')}{\alpha}\left(1 - \frac{B}{Y}\right)^{1-\beta} \tag{8}$$

If either red or green were the minimum primaries, these equations would be switched around accordingly.

JPEG-HDR: A Backwards-Compatible, High Dynamic Range Extension to JPEG

Greg Ward ***Maryann Simmons***
BrightSide Technologies ***Walt Disney Feature Animation***

Abstract

The transition from traditional 24-bit RGB to high dynamic range (HDR) images is hindered by excessively large file formats with no backwards compatibility. In this paper, we demonstrate a simple approach to HDR encoding that parallels the evolution of color television from its grayscale beginnings. A tone-mapped version of each HDR original is accompanied by restorative information carried in a subband of a standard output-referred image. This subband contains a compressed ratio image*, which when multiplied by the tone-mapped foreground, recovers the HDR original. The tone-mapped image data is also compressed, and the composite is delivered in a standard JPEG wrapper. To naïve software, the image looks like any other, and displays as a tone-mapped version of the original. To HDR-enabled software, the foreground image is merely a tone-mapping suggestion, as the original pixel data are available by decoding the information in the subband. Our method further extends the color range to encompass the visible gamut, enabling a new generation of display devices that are just beginning to enter the market.*

Introduction

Visible light in the real world covers a vast dynamic range. Humans are capable of simultaneously perceiving over 4 orders of magnitude (1:10000 contrast ratio), and can adapt their sensitivity up and down another 6 orders. Conventional digital images, such as 24-bit *sRGB* [Stokes et al. 1996], hold less than 2 useful orders of magnitude. Such formats are called *output-referred* encodings because they are tailored to what can be displayed on a common CRT monitor – colors outside this limited gamut are not represented. A *scene-referred* standard is designed instead to represent colors and intensities that are visible in the real world, and though they may not be rendered faithfully on today's output devices, they may be visible on displays in the near future [Seetzen et al. 2004]. Such image representations are referred to as *high dynamic range* (HDR) formats, and a few alternatives have been introduced over the past 15 years, mostly by the graphics research and special effects communities.

Unfortunately, none of the existing HDR formats supports lossy compression, and only one comes in a conventional image wrapper. These formats yield prohibitively large images that can only be viewed and manipulated by specialized software. Commercial hardware and software developers have been slow to embrace scene-referred standards due to their demands on image capture, storage, and use. Some digital camera manufacturers attempt to address the desire for greater exposure control during processing with their own proprietary RAW formats, but these camera-specific encodings fail in terms of image archiving and third-party support.

What we really need for HDR digital imaging is a compact representation that looks and displays like an output-referred JPEG, but holds the extra information needed to enable it as a scene-referred standard. The next generation of HDR cameras will then be able to write to this format without fear that the software on the receiving end won't know what to do with it. Conventional image manipulation and display software will see only the tone-mapped version of the image, gaining some benefit from the HDR capture due to its better exposure. HDR-enabled software will have full access to the original dynamic range recorded by the camera, permitting large exposure shifts and contrast manipulation during image editing in an extended color gamut. The availability of an efficient, backwards-compatible HDR image standard will provide a smooth upgrade path for manufacturers and consumers alike.

Background

High dynamic range imaging goes back many years. A few early researchers in image processing advocated the use of logarithmic encodings of intensity (e.g., [Jourlin & Pinoli 1988]), but it was global illumination that brought us the first standard. A space-efficient format for HDR images was introduced in 1989 as part of the *Radiance* rendering system [Ward 1991; Ward 1994]. However, the *Radiance* RGBE format was not widely used until HDR photography and image-based lighting were developed by Debevec [Debevec & Malik 1997; Debevec 1998]. About the same time, Ward Larson [1998] introduced the LogLuv alternative to RGBE and distributed it as part of Leffler's public TIFF library [Leffler et al. 1999]. The LogLuv format has the advantage of covering the full visible gamut in a more compact representation, whereas RGBE is restricted to positive primary values. A few graphics researchers adopted the LogLuv format, but most continued to use RGBE (or its extended gamut variant XYZE), until Industrial Light and Magic made their EXR format available to the community in 2002 [Kainz et al. 2002]. The OpenEXR library uses the same basic 16-bit/channel floating-point data type as modern graphics cards, and is poised to be the new favorite in the special effects industry. Other standards have also been proposed or are in the works, but they all have limited dynamic range relative to their size (e.g., [IEC 2003]). None of these standards is backwards compatible with existing software.

The current state of affairs in HDR imaging parallels the development of color television after the adoption of black and white broadcast. A large installed base must be accommodated as well as an existing standard for transmission. The NTSC solution introduced a subband to the signal that encoded the additional chroma information without interfering with the original black and white signal [Jolliffe 1950]. We propose a similar solution in the context of HDR imagery, with similar benefits.

As in the case of black and white television, we have an existing, de facto standard for digital images: output-referred JPEG. JPEG has a number of advantages that are difficult to beat. The standard is unambiguous and universally supported. Software libraries are free and widely available. Encoding and decoding is fast and efficient, and display is straightforward. Compression performance for average quality images is competitive with more recent advances, and every digital camera writes to this format. Clearly, we will be living with JPEG images for many years to come. Our chances of large scale adoption increase dramatically if we can maintain backward compatibility to this standard.

Our general approach is to introduce a subband that accompanies a tone-mapped version of the original HDR image. This subband is compressed to fit in a metadata tag that will be ignored by naïve applications, but can be used to extract the HDR original in enabled software. We utilize JPEG/JFIF as our standard wrapper in this implementation, but our technique is compatible with any format that provides client-defined tags (e.g., TIFF, PNG, etc.).

The Method section of our paper describes the ideas behind HDR subband encoding, followed by specific details in the Implementation section. The Results section presents some examples and compression statistics. We end with our conclusions and future directions.

Method

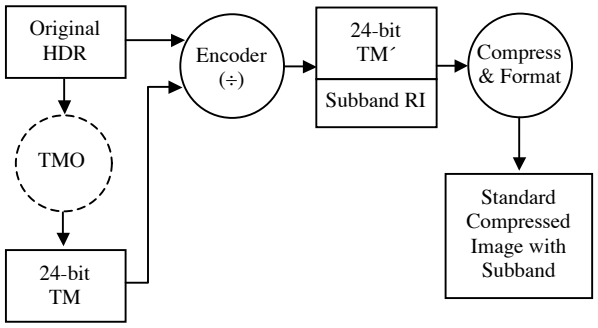


Figure 1. High level HDR subband encoding pipeline.

Figure 1 shows an overview of our HDR encoding pipeline. We start with two versions of our image: a scene-referred HDR image, and an output-referred, tone-mapped version. If we like, we can generate this tone-mapped version ourselves, but in general this is a separable problem, and our implementation is compatible with multiple operators. The encoding stage takes these two inputs and produces a composite, consisting of a potentially modified version of the original tone-mapped image, and a subband ratio image that contains enough information to reproduce a close facsimile of the HDR original. The next stage compresses this information, offering the tone-mapped version as the JPEG base image, and storing the subband as metadata in a standard JFIF wrapper.

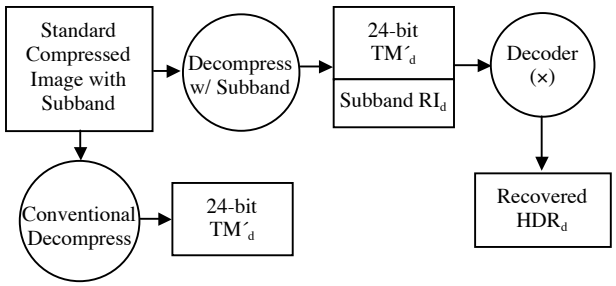


Figure 2. Alternate decoding paths for compressed composite.

Figure 2 shows the two possible decoding paths. The high road decompresses both the tone-mapped foreground image and the subband, delivering them to the decoder to recover the HDR pixels. Naïve applications follow the low road, ignoring the subband in the metadata and reproducing only the tone-mapped foreground image.

In the simplest incarnation of this method, the encoder divides the HDR pixels by the tone-mapped luminances, producing an 8-bit ratio image that is stored as metadata in a standard JPEG compressed image. Decoding follows decompression with a multiplication step to recover the original HDR pixels. Unfortunately, the simplest approach fails for pixels that are mapped to black or white by the tone-mapping operator (TMO), and the standard clamping of RGB values can result in a substantial loss of color saturation at the top end. These are two important issues we must address with our technique, and these are discussed in the following subsections.

A similar approach to JPEG gamut extension has been implemented in Kodak's ERI system [Spaulding et al. 2003]. In the ERI format, a *residual image* rather than a ratio image is recorded in a subband. This residual is defined as the arithmetic difference between an input ERIMM RGB color space and the recorded sRGB foreground image, reduced to an 8-bit range. Unfortunately, the 12 bits present in the original data cannot be recovered by adding two 8-bit channels together. Kodak's reconstructed image gains at most two f-stops in dynamic range over sRGB (about 2.2 orders of magnitude total), rather than the four orders of magnitude we need for true HDR imagery.

The Ratio Image

JPEG/JFIF provides a limited set of metadata storage channels, called the "application markers." Sixteen application markers exist in the JFIF specification, three of which are spoken for. A single marker holds a maximum of 64 Kbytes of data, regardless of the image dimensions. This 64K limit can be circumvented by storing multiple markers reusing the same identifier, and this is what we have done in our implementation. The foreground signal in our encoding is a tone-mapped version of the original HDR image. This output-referred image is stored in the usual 8×8, 8-bit blocks using JPEG's lossy DCT compression [Wallace 1991]. The subband encodes the information needed to restore the HDR original from this compressed version.

Let us assume that our selected tone-mapping operator possesses the following properties:

- The original HDR input is mapped smoothly into a 24-bit output domain, with no components being rudely clamped at 0 or 255.
- Hue is maintained at each pixel, and mild saturation changes can be described by an invertible function of input and output value.

Most tone-mapping operators for HDR images have the first property as their goal, so this should not be a problem. If it is, we can override the operator by locally replacing each clamped pixel with a lighter or darker value that fits the range. Similarly, we can enforce the second property by performing our own color desaturation, using the tone-mapping operator as a guide only for luminance changes. Most tone-mapping operators address color in a very simple way, if they consider it at all. Exceptions are found in operators that simulate human vision (e.g., [Pattanaik et al. 1998]), whose support is outside the scope of this encoding.

Given these restrictions, a ratio image may be computed by dividing the HDR original luminance at each pixel by the tone-mapped output luminance:

$$RI(x,y) = \frac{L(HDR(x,y))}{L(TM(x,y))} \tag{1}$$

The ratio image is log-encoded and compressed as an 8-bit greyscale image stored in our subband along with the saturation formula described in the following subsection. The range of the ratio image is determined beforehand in order to optimize the subband encoding's precision. The tone-mapped (and desaturated) version is then encoded as the foreground image. During reconstruction, color is restored using the recorded saturation parameters. The ratio image is then remultiplied pixel by pixel to recover the original HDR image. An example tone-mapped foreground image and its associated subband is shown in Figure 3.

Figure 3. Memorial Church shown tone-mapped with Reinhard's global operator (left) along with the log-encoded ratio image needed for HDR recovery (right).

Color Saturation and Gamut

To preserve colors during transcoding through the output-referred space of our foreground image, we apply two separate but complimentary techniques. The first technique makes use of normally unused YCC code values to extend the working gamut. The second technique desaturates colors prior to encoding, then applies resaturation during decoding (i.e., gamut companding).

In the first technique, we take advantage of the fact that the standard 24-bit YCbCr space employed in JPEG is larger than the sRGB space it encodes. As sRGB nears its white value (255,255,255), its gamut triangle gets vanishingly small. In comparison, the YCC space has a generous gamut as it approaches its white (255,128,128), because the chroma values can take on any value from 0-255, 128 being neutral. By ascribing meaning to these otherwise unused YCC code values, it is possible to extend the effective gamut of a YCC image to include bright, saturated colors that would otherwise be lost. This strategy has been employed successfully by Canon and Sony to preserve otherwise out-of-gamut colors, and there is currently an effort underway to standardize such an extension by IEC TC100. By itself, such an extension to YCC does not provide additional dynamic range because the Y value is still clamped at 255, but it is a perfect complement to our greyscale subband, since it provides the extra saturation we need near the top of the range. Otherwise, we would have to introduce an unnatural darkening to regions of the foreground image where the bright values happened to be colorful, as in a sunset.

The YCC extension we have implemented precisely matches the standard sRGB to YCbCr mapping where it is defined, then logically extends past the boundaries to encompass additional colors. Because we expect linear floating point RGB input values, we define our mapping to and from linear RGB, bypassing sRGB while maintaining consistency with it. We assume that the RGB color space uses the standard primaries and white point given in Table 1. However, we permit our input channels to take on values outside the 0-1 range, mirroring our gamma lookup to include negative values as defined below:

$$R' = \begin{cases} 1.055R^{0.42} - 0.055 & if\ R > 0.0031308 \\ 12.92R & if\ |R| \le 0.0031308 \\ -1.055(-R)^{0.42} + 0.055 & if\ R < -0.0031308 \end{cases} \tag{2}$$

This equation is then repeated for G' and B', which are combined with the following standard transform to yield our mapping from RGB to YCC:

$$\begin{aligned} Y &= (0.299R' + 0.587G' + 0.114B') \times 256 \\ Cb &= (-0.169R' - 0.331G' + 0.5B' + 0.5) \times 256 \\ Cr &= (0.5R' - 0.419G' - 0.813B' + 0.5) \times 256 \end{aligned} \tag{3}$$

Values outside the allowed 8-bit range (0-255) must of course be truncated, but even so this mapping provides a much larger gamut near white. To extend the gamut even further, we apply a second technique, *gamut companding*.

Table 1. sRGB Primary Chromaticities

	R	*G*	*B*	*W*
x	0.640	0.300	0.150	0.3127
y	0.330	0.600	0.060	0.3290

To compress the gamut during encoding, we can apply a global desaturation to the image, pulling all colors in towards gray by an amount that is guaranteed to contain the entire visible gamut in our extended YCC range. This may also be beneficial to the appearance of the foreground image, as dynamic range compression tends to leave the colors with an oversaturated appearance. The gamut is then expanded during HDR decoding, rendering the colors back into their original saturation. We employ the following definition of color saturation based on the linear values:

$$S \equiv 1 - \min(R,G,B)/Y \tag{4}$$

Y in the formula above is the luminance of red, green and blue taken together (0.213*R + 0.715*G + 0.072*B). Note that we expect saturation to be greater than 1 for negative primaries. Zero saturation implies a neutral value, which is passed unchanged. Non-neutral values are desaturated with a two-parameter formula to obtain our compressed color saturation:

$$S_c = \alpha \cdot S^{\beta} \tag{5}$$

The α parameter controls how much saturation we wish to keep in the encoded colors, and is generally ≤ 1. The β parameter controls the color "contrast," and may be greater or less than 1 depending on the desired effect. This modified saturation is used with the original saturation from Eq. (4) to determine the encoded primary values. Below is the formula for the desaturated red channel:

$$R_c = \left(1 - \frac{S_c}{S}\right) \cdot Y + \frac{S_c}{S} \cdot R \tag{6}$$

This equation is repeated for the green and blue channels. Note that *Y* does not change under this transformation, and the primary that was smallest before is the smallest after. The R_c in Eq. (6) then becomes the input *R* in Eq. (2) for the YCC mapping, and similarly for G_c and B_c.

Resaturating the encoded color to get back the original pixel is done by inverting these equations. If the smallest primary value were blue for example, this inverse transformation would yield:

$$B = Y - Y \cdot \left(\frac{Y - B_c}{\alpha Y}\right)^{1/\beta} \tag{7}$$

The red and green channels would then be determined by:

$$R = Y - \frac{(Y - R_c)}{\alpha}\left(1 - \frac{B}{Y}\right)^{1-\beta}$$
$$G = Y - \frac{(Y - G_c)}{\alpha}\left(1 - \frac{B}{Y}\right)^{1-\beta} \tag{8}$$

If either red or green were the minimum primaries, these same equations hold, with the color values substituted accordingly.

Gamut companding applies to our linear RGB values, prior to YCC encoding during compression and following YCC decoding during expansion. The subband contains no color information, but it, too, is compressed using the same JPEG DCT method employed for the color foreground image. The difference is that one log-encoded channel is present instead of three gamma-encoded channels.

Subband Compression

Downsampling the ratio image can greatly reduce the size of the subband without serious consequences to the recovered HDR image's appearance. As in the perception of color, our eye has a limited ability to register large, high frequency changes in luminance. This is due primarily to optical scattering in the eye, which has also been exploited to make efficient HDR displays with different resolution modulators [Seetzen et al. 2004]. Note that we can and do see small, high frequency luminance changes, but these can be adequately represented in the foreground image. By trading resolution with the foreground, we may obtain much better compression using a downsampled subband. Details of how this is done and the trade-offs involved are described in [Ward & Simmons 2004].

To summarize, we have implemented two alternative methods for subband downsampling: a *precorrection* method and a *postcorrection* method. The precorrection method starts by downsampling and then upsampling the ratio image in order to predict what the decompressor will see on the receiving end. Equation (1) is then rearranged so that the resampled ratio image, RI_d, is divided into the original HDR luminance values to obtain a pre-corrected, tone-mapped *Y* channel:

$$TM' = \frac{HDR}{RI_d} \tag{9}$$

By construction, this pre-corrected foreground image will then reproduce the HDR original when it is multiplied by the upsampled ratio image during decompression. The only visible effect is an increase in sharpness at high contrast boundaries in the foreground image. This sharpening will then cancel out in the recovered HDR image. Precorrection is therefore, the most appropriate method when the fidelity of the decoded HDR image is paramount.

The alternative postcorrection method is more appropriate when the appearance of the output-referred image must not be compromised. In this method, we synthesize the high frequencies that we lost to downsampling of the ratio image using our

foreground image as a guide. This is a simple application of resolution enhancement via example learning. Much more sophisticated techniques have been developed by the vision community [e.g., Baker & Kanade 2002]. We refer the reader to our previous paper for details and examples of this method [Ward & Simmons 2004].

Implementation

We have implemented a fully functional library for reading and writing our compressed, high dynamic range format, which we call JPEG-HDR. As described above, it employs a standard JPEG/JFIF wrapper to hold the foreground image, and encodes a subband containing a compressed ratio image in one or more application 11 markers [Spinal Tap 1984, "This one goes to 11"]. In addition to this JPEG-compressed greyscale image, the subband contains the lower and upper ratios corresponding to 0 and 255, the correction method used if the image was downsampled, the saturation parameters $\alpha$and β, and a calibration factor for recovering absolute luminance values where available.

Since one of the key goals of this format is backwards compatibility, we based our implementation on the existing, widely-used and public IJG (Independent JPEG group) library written by Thomas Lane [IJG 1998]. Most imaging applications either use this library or some modified version of it, so it serves as a common reference for code development. By matching the *libjpeg* call structure and leveraging its routines, we hope to minimize the coding effort required to support our JPEG-HDR extension. In fact, the code we have written links to an unmodified version of the IJG library, and works interchangeably on Windows, Linux, and Apple platforms.

The code changes required to support the reading and writing of HDR data are minimally invasive, requiring in most cases nothing more than the substitution of a few calls and data structures. For example, in place of the standard *libjpeg* jpeg_create_decompress() call to initialize a reader structure, an HDR-enabled application calls jpeghdr_create_decompress(), which initializes both a standard JPEG reader structure and the extended structure containing it. The subsequent call to jpeghdr_read_header() then returns either JPEG_HEADER_OK in the case of a standard JPEG, or JPEG_HEADER_HDR in the case of a JPEG-HDR image. If the image is a standard JPEG, then the application proceeds as before, calling the normal *libjpeg* routines. If the image is a JPEG-HDR, however, the application uses our replacement routines to recover floating-point RGB scanlines rather than the more usual sRGB data. (For convenience, we also provide a means to access the tone-mapped YCC or sRGB data for applications that want it.)

As a further convenience for new applications, we also provide a simplified pair of calls for reading and writing JPEG-HDR images to and from memory. The jpeghdr_load_memory() routine takes a named JPEG image file and allocates either a floating-point or 8-bit frame buffer depending on its type, and reads the image into memory. Similarly, the jpeghdr_write_memory() routine takes a floating-point frame buffer and writes it out to a JPEG-HDR image. Parameters are provided for setting the quality, correction method, and gamut compression.

Bear in mind that it is not necessary to use our extended library to read a JPEG-HDR image, as the format itself is backwards-compatible with the JPEG/JFIF standard. Any application that reads standard JPEG images will be able to read and display JPEG-HDR with no changes or recompilation. It will only see the tone-mapped foreground image in this case, but if the application is not HDR-enabled, this is the best visual representation available to it.

Tone-Mapping

As mentioned in the previous section, our framework is designed to function with multiple tone-mapping operators. We have experimented with a selection of global and local TMOs, including Reinhard's global photographic tone operator [Reinhard et al. 2002], Ward Larson's global histogram adjustment [Ward Larson et al. 1997], Fattal's gradient operator [Fattal et al. 2002], and Durand & Dorsey's bilateral filter [Durand & Dorsey 2002]. Of these, we found that the bilateral filter performed best with our method, followed closely by Reinhard's photographic TMO [Ward & Simmons 2004].

In our library implementation, we provide a default mapping based on Reinhard's TMO, along with hooks so the calling application can substitute its own luminance mapping if desired, similar to the way applications can provide their own JPEG quantization tables. These hooks include a method for computing the log histogram of luminance, as required by most global TMOs, and a call to check whether or not a tone-mapped RGB color is inside the compressed YCC gamut of our foreground image. The latter is convenient for adjusting the tone-mapping to minimize color saturation losses in the encoded image, and is used for this purpose in the default TMO provided.

We fully expect that custom tone-mappings will be a key "value added" area for software and hardware vendors using HDR in the future, and the separation of the TMO from the decoding method is one of the key strengths of this format.

Results

JPEG-HDR has been fully integrated into Photosphere 1.3, a freeware application for building and cataloging HDR images under Mac OS X. We have converted hundreds of HDR images to the JPEG-HDR format and studied compression performance, looking for artifacts and errors, and we consider the library to be well-tested at this stage.

Figure 4 shows compression performance statistics for different quality settings on a collection of 217 natural scene captures. The JPEG-HDR image size, in bits per pixel, is plotted as a function of the quality setting. Although the quality may be set anywhere from 0-100, the appearance degrades rapidly below a setting of 60, so we consider the range here to be representative. There was only a small difference in total file size between the precorrection and postcorrection methods, precorrection being larger because it introduces additional high frequency information into the

foreground image. (There is no subband downsampling at quality levels above 95, which is why the Q=99 points are the same.)

The JPEG-HDR format achieved average bits/pixel rates from 0.6 to 3.75 for corresponding quality settings from 57-99%. For comparison, the Radiance RGBE format [Ward 1991] uses on average 26 bits/pixel for this image set, and OpenEXR [Kains 2002] uses 28 bits/pixel. The best compression performance we measured for a lossless format was LogLuv TIFF [Ward Larson 1998], which averaged 21.5 bits/pixel for this data set. Thus, relative to these lossless formats, JPEG-HDR offers additional compression of between 6:1 (84%) and 40:1 (97%) over lossless methods, depending on quality.

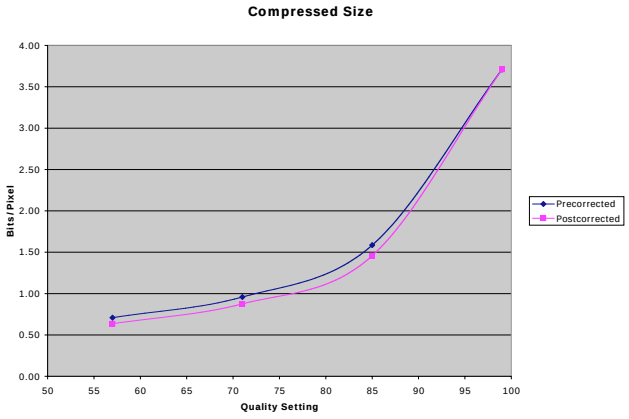


Figure 4. JPEG-HDR compressed image size.

Figure 5 shows the fraction of the saved file taken up by the ratio image subband. At lower quality levels, the ratio image is downsampled at a higher rate, which leads to a smaller percentage of the total file size. On average, the ratio image occupies less than 1/4 of the total file size, though this percentage can vary quite a bit from one image to the next. In our image captures, the compressed subband took between 16% and 27% of the file at Q=71, and between 24% and 37% at Q=85. The actual space taken up by the subband is about the same for the precorrection and postcorrection cases. It ends up being a larger fraction for the postcorrection case simply because the foreground image ends up being smaller, as we just explained.

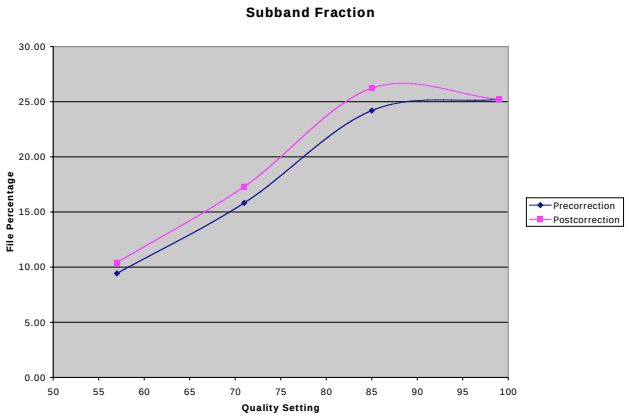


Figure 5. JPEG-HDR subband size.

Figure 6 shows another view of the Memorial church in false color, indicating a luminance range that spans over 5 orders of magnitude. Because the artifacts are most visible in the darkest regions of our image, we have magnified the right-hand alcove ceiling for the our visual quality comparison and normalized the exposure with a linear scale that is a factor of 8000 (13 stops) above the saturation level for the brightest region. The results shown in Figure 7 at are nearly perfect at Q=99, even in the deepest shadows. As we decrease our quality setting (and our bitrate), we see that the image degrades in a controlled way. At the lowest quality we tested, Q=57, block artifacts are visible at the ceiling's apex, and ringing is evident near the light fixtures.

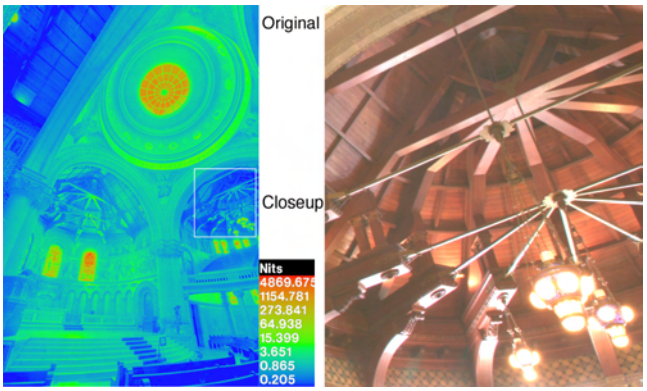


Figure 6. A close-up of one of the darker sections of the Memorial church, shown in its original, uncompressed form.

Figure 7. Quality of recovered image at different compression settings using subband precorrection method and Reinhard's global TMO (bits/pixel in parentheses).

To test our algorithm more systematically, we used Daly's Visible Differences Predictor [Daly 1993] to evaluate image fidelity before and after JPEG-HDR compression. At a quality setting of 90, we found fewer than 2.5% of the HDR image pixels (on average) were visibly different after decompression using Reinhard's global TMO and our subband precorrection method. This increased to 4.7% with the postcorrection method. At the maximum quality setting, fewer than 0.1% of the pixels were visibly different in each image [Ward & Simmons 2004].

Conclusions and Future Directions

Although one could implement a lossy high dynamic-range format as an extension to JPEG 2000, it would not have the benefit of backwards compatibility with existing hardware and software. Does that really matter? Certainly, for HDR-specific applications, backwards compatibility is of little import. However, more traditional imaging applications are bound to be slow in adopting HDR because of the additional burden it places on tone-mapping for viewing and output. Consider how long it has taken for ICC profiles to see widespread use. The ICC is just now coming to terms with scene-referred standards and what they mean to the imaging pipeline. The more difficult it is for software vendors to support a new color technology, the longer it will take to reach the mainstream. Backwards compatibility offers us a shortcut, a way to bring HDR to the masses before they come to us, demanding it. We will all get there, eventually. In the meantime, JPEG-HDR allows the advance guard to share scene-referred images without bringing down the network.

High dynamic-range imaging has enormous potential for digital photography, which has been suffering since its inception from the mistaken but convenient notion that a camera is somehow like a scanner, and the same color strategy that works for prepress can work for photography. It cannot and it does not. There is no white in the real world. There is only brighter and darker, redder and bluer. "Scene-referred color" is unbounded and unreferenced. We may choose a white point, but in practice our choice is based on scene statistics and heuristics rather than any known or measurable illuminant. Similarly, we should never attempt to define reflectance values in photography. There is no such thing. If we set a gray card in the scene and call that 20%, it is relative to the light falling on it in that position and orientation at that moment. If someone's shadow passes over our reference, its brightness may drop by a factor of 100, and yet we still call that 20%? Our notions and our imaging techniques must change to fit with the reality, not the other way around. Photography measures light, not reflectance. Digital images should record light, and for this we need an HDR format. A backwards-compatible, scene-referred extension to JPEG allows us to live with our mistake *while* we correct it.

The most obvious and important future work in this area is an HDR extension to MPEG. Mantiuk et al. suggested one such extension, though their solution is not backwards-compatible [Mantiuk et al. 2004]. Backwards compatibility is even more critical to video applications, as the installed base of DVD players is not going away anytime soon. A version of MPEG that does not play on existing hardware is likely to be confined to the laboratory until the next major video format revolution, which is at least 10 years away by most predictions.

Fortunately, the extension of the subband method to MPEG is reasonably straightforward. To achieve low bitrates, a subband needs to be compressed across multiple frames, and this is most easily accomplished as a sub-video within the MPEG main stream. Just as we were able to smuggle our ratio image in a JPEG marker, we can smuggle a *ratio video* as MPEG metadata. In specially equipped players, this subband will be decoded simultaneously with the main MPEG stream and recombined in a multiplication step for HDR viewing on an appropriate display [Seetzen et al. 2004]. If no HDR player or display is available, the foreground video's tone-mapping will already provide an appropriate color space for low dynamic-range output. As in still imagery, this approach allows the gradual introduction of HDR to the video market, a market we know will not accept any other type of introduction.

Software Availability

Photosphere, an HDR image builder and browser for Mac OS X, reads, writes, and converts JPEG-HDR and is available for free download from www.anyhere.com. Additional software, including format converters and a non-commercial JPEG-HDR library for Windows, Apple, and Linux platforms are available on the DVD-ROM that accompanies [Reinhard et al. 2005]. For

commercial inquiries, please visit the BrightSide Technologies website at www.brightsidetech.com/products/process.php

References

ASHIKHMIN, M. 2002. A Tone Mapping Algorithm for High Contrast Images. In *Proceedings of 13th Eurographics Workshop on Rendering*, 145-156.

BAKER, S & KANADE, T. 2002. Limits on super-resolution and how to break them. IEEE Transactions on Pattern Analysis and Machine Intelligence., 24(9):1167-1183, September. 2002.

CHIU, K. HERF, M., SHIRLEY, P., SWAMY, M., WANG, C., and ZIMMERMAN, K., 2002. A Tone Mapping Algorithm for High Contrast Images. In *Proceedings of 13th Eurographics Workshop on Rendering*, 245-253.

DALY, S. 1993. The Visible Differences Predictor: An Algorithm for the Assessment of Image Fidelity. In *Digital Images and Human Vision*, A.B. Watson, editor, MIT Press, Cambridge, Massachusetts.

DEBEVEC, P., and MALIK, J. 1997. Recovering High Dynamic Range Radiance Maps from Photographs. In *Proceedings of ACM SIGGRAPH 1997*, 369-378.

DEBEVEC, P. 1998. Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography. In *Proceedings of ACM SIGGRAPH 1998*, 189-198.

DURAND, F., and DORSEY, J. 2002. Fast Bilateral Filtering for the Display of High-Dynamic Range Images. *ACM Transactions on Graphics*, 21, 3, 249-256.

FATTAL, R., LISCHINSKI, D., and WERMAN, M. 2002. Gradient Domain High Dynamic Range Compression. *ACM Transactions on Graphics*, 21, 3, 257-266.

IEC. 2003. 61966-2-2. Extended RGB colour space – scRGB, *Multimedia systems and equipment – Colour measurement and management* – Part 2-2: Colour management.

INDEPENDENT JPEG GROUP. 1998. www.ijg.org/

JOLLIFFE, C.B., 1950. Answers to Questions about Color Television. members.aol.com/ajaynejr/rca2.htm.

JOURLIN, M., PINOLI, J-C., 1988. A model for logarithmic image processing, *Journal of Microscopy*, 149(1), pp. 21-35.

KAINS, F., BOGART, R., HESS, D., SCHNEIDER, P., ANDERSON, B., 2002. *OpenEXR*. www.openexr.org/.

LEFFLER, S., WARMERDAM, F., KISELEV, A., 1999. *libTIFF*. remotesensing.org/libtiff.

LI, Y., SHARAN, L., ADELSON, E. 2005. "Compressing and Companding High Dynamic Range Images with Subband Architectures," ACM Trans. on Graphics (Proceedings of ACM SIGGRAPH), pp. 836-44.

MANTIUK, R., KRAWCZYK, G., MYSZKOWSKI, K., SEIDEL, H-P. 2004. "Perception-motivated High Dynamic Range Video Encoding," *SIGGRAPH 2004*.

MOON P., and SPENCER, D. 1945. The Visual Effect of Non-Uniform Surrounds. *Journal of the Optical Society of America*, 35, 3, 233-248.

PATTANAIK, S., FERWERDA, J., FAIRCHILD, M., and GREENBERG, D. 1998. A Multiscale Model of Adaptation and Spatial Vision for Realistic Image Display, In *Proceedings of ACM SIGGRAPH 1998*, 287-298.

REINHARD, E., WARD, G., PATTANAIK, S., DEBEVEC, P. 2005. *High Dynamic Range Imaging: Acquisition, Display, and Image-based Lighting*, Morgan Kaufmann Publishers, San Francisco.

REINHARD, E., STARK, M., SHIRLEY, P., and FERWERDA, J. 2002. Photographic Tone Reproduction for Digital Images. *ACM Transactions on Graphics*, 21,3, 267-276.

SEETZEN, H., HEIDRICH, W., STUEZLINGER, W., WARD, G., WHITEHEAD, L., TRENTACOSTE, M., GHOSH, A., VOROZCOVS, A. 2004. "High Dynamic Range Display Systems," *ACM Trans. Graph.* (special issue SIGGRAPH 2004).

SPAULDING, K., WOOLFE, G., & JOSHI, R. 2003. "Using a Residual Image to Extend the Color Gamut and Dynamic Range of an sRGB Image," white paper posted on the Kodak website.

SPINAL TAP, THIS IS. 1984. Dir. Rob Reiner. MGM Home Entertainment.

STOKES, M., ANDERSON, M., CHANDRASEKAR, S., and MOTTA, R. 1996. Standard Default Color Space for the Internet. www.w3.org/Graphics/Color/sRGB.

TUMBLIN, J., and TURK, G. 1999. LCIS: A Boundary Hierarchy for Detail-Preserving Contrast Reduction. *ACM Trans. on Graphics*, 21, 3, 83-90.

WALLACE, G. 1991. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34, 4, 30-44.

WARD LARSON, G., RUSHMEIER, H., and PIATKO, C. 1997. A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes. *IEEE Trans. on Visualization and Computer Graphics*, 3, 4.

WARD LARSON, G. 1998. Overcoming Gamut and Dynamic Range Limitations in Digital Images. *Proc. of IS&T 6th Color Imaging Conf.*

WARD, G. 1991. Real Pixels. In *Graphics Gems II*, edited by James Arvo, Academic Press, 80-83.

WARD, G. 1994. The RADIANCE Lighting Simulation and Rendering System. , In *Proceedings of ACM SIGGRAPH 1994*, 459-472.

WARD, G. & SIMMONS, M. 2004. "Subband Encoding of High Dynamic Range Imagery," *First Symposium on Applied Perception in Graphics and Visualization* (APGV).

Image-Based Lighting

Paul Debevec
USC Institute for Creative Technologies

This tutorial shows how image-based lighting can illuminate synthetic objects with measurements of real light, making objects appear as if they're actually in a real-world scene.

Image-based lighting (IBL) is the process of illuminating scenes and objects (real or synthetic) with images of light from the real world. It evolved from the reflection-mapping technique[1,2] in which we use panoramic images as texture maps on computer graphics models to show shiny objects reflecting real and synthetic environments. IBL is analogous to image-based modeling, in which we derive a 3D scene's geometric structure from images, and to image-based rendering, in which we produce the rendered appearance of a scene from its appearance in images. When used effectively, IBL can produce realistic rendered appearances of objects and can be an effective tool for integrating computer graphics objects into real scenes.

The basic steps in IBL are

1. capturing real-world illumination as an omnidirectional, high dynamic range image;
2. mapping the illumination onto a representation of the environment;
3. placing the 3D object inside the environment; and
4. simulating the light from the environment illuminating the computer graphics object.

1 A microscope, modeled by Gary Butcher in 3D Studio Max, rendered using Marcos Fajardo's Arnold rendering system as illuminated by light captured in a kitchen.

Figure 1 shows an example of an object illuminated entirely using IBL. Gary Butcher created the models in 3D Studio Max, and the renderer used was the Arnold global illumination system written by Marcos Fajardo. I captured the light in a kitchen so it includes light from a ceiling fixture; the blue sky from the windows; and the indirect light from the room's walls, ceiling, and cabinets. Gary mapped the light from this room onto a large sphere and placed the model of the microscope on the table in the middle of the sphere. Then, he used Arnold to simulate the object's appearance as illuminated by the light coming from the sphere of incident illumination.

In theory, the image in Figure 1 should look about how a real microscope would appear in that environment. It simulates not just the direct illumination from the ceiling light and windows but also the indirect illumination from the rest of the room's surfaces. The reflections in the smooth curved bottles reveal the kitchen's appearance, and the shadows on the table reveal the colors and spread of the area light sources. The objects also successfully reflect each other, owing to the ray-tracing-based global-illumination techniques we used.

This tutorial gives a basic IBL example using the freely available Radiance global illumination renderer to illuminate a simple scene with several different lighting environments.

Capturing light

The first step in IBL is obtaining a measurement of real-world illumination, also called a light probe image.[3] The easiest way to do this is to download one. There are several available in the Light Probe Image Gallery at http://www.debevec.org/Probes. The Web site includes the kitchen environment Gary used to render the microscope as well as lighting captured in various other interior and outdoor environments. Figure 2 shows a few of these environments.

Light probe images are photographically acquired images of the real world, with two important properties. First, they're omnidirectional—for every direction in the world, there's a pixel in the image that corresponds to that direction. Second, their pixel values are

0272-1716/02/$17.00 © 2002 IEEE

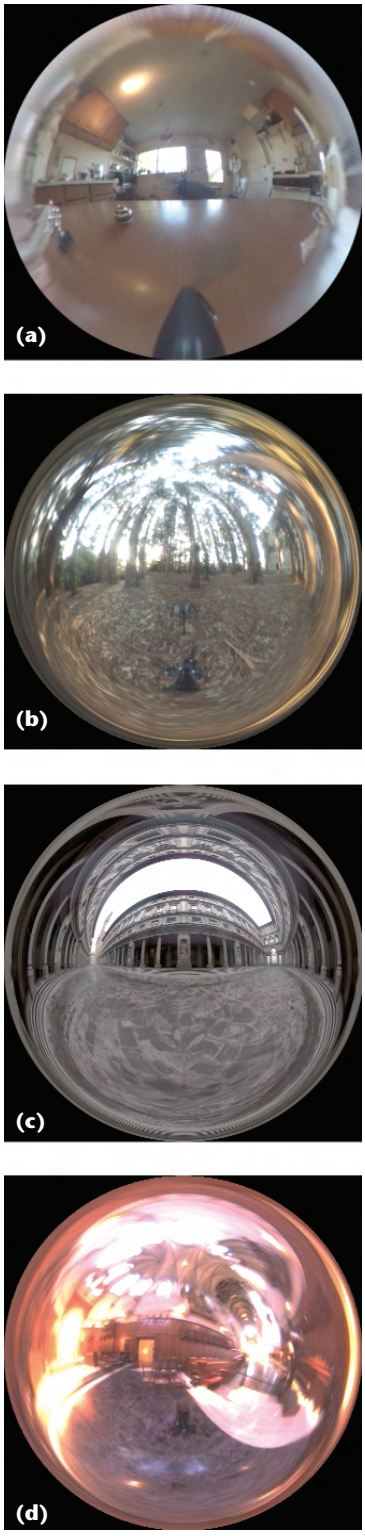
linearly proportional to the amount of light in the real world. In the rest of this section, we'll see how to take images satisfying both of these properties.

We can take omnidirectional images in a number of ways. The simplest way is to use a regular camera to take a photograph of a mirrored ball placed in a scene. A mirrored ball actually reflects the entire world around it, not just the hemisphere toward the camera. Light rays reflecting off the outer circumference of the ball glance toward the camera from the back half of the environment. Another method of obtaining omnidirectional images using a regular camera is to shoot a mosaic of many pictures looking in different directions and combine them using an image stitching program such as RealViz's Stitcher. A good way to cover a particularly large area in each shot is to use a fisheye lens,[4] which lets us cover the full field in as few as two images. A final technique is to use a special scanning panoramic camera (such as the ones Panoscan and Sphereon make), which uses a vertical row of image sensors on a rotating camera head to scan across a 360-degree field of view.

In most digital images, pixel values aren't proportional to the light levels in the scene. Usually, light levels are encoded nonlinearly so they appear either more correctly or more pleasingly on nonlinear display devices such as cathode ray tubes. Furthermore, standard digital images typically represent only a small fraction of the dynamic range—the ratio between the dimmest and brightest regions accurately represented—present in most real-world lighting environments. When part of a scene is too bright, the pixels saturate to their maximum value (usually 255) no matter how bright they really are.

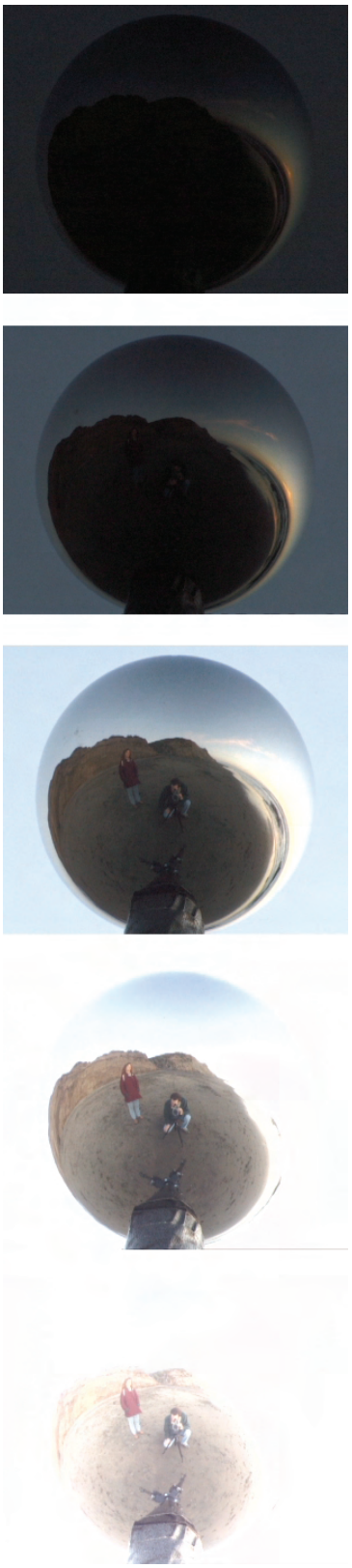
We can ensure that the pixel values in the omnidirectional images are truly proportional to quantities of light using high dynamic range (HDR) photography techniques.[5] The process typically involves taking a series of pictures of the scene with varying exposure levels and then using these images to solve for the imaging system's response curve and to form a linear-response composite image covering the entire range of illumination values in the scene. Software for assembling images in this way includes the command-line mkhdr program at http://www.debevec.org/Research/HDR and the Windows-based HDR Shop program at http://www.debevec.org/HDRShop.

HDR images typically use a single-precision floating-point number for red, green, and blue, allowing the full range of light from thousandths to billions to be represented. We can store HDR image data in a various file formats, including the floating-point version of the TIFF file format or the Portable Floatmap variant of Jef Postsanzer's Portable Pixmap format. Several other representations that use less storage are available, including Greg Ward's Red-Green-Blue Exponent (RGBE) format[6] (which uses one byte each for red, green, blue and a common 8-bit exponent) and his new 24-bit and 32-bit LogLuv formats recently included in the TIFF standard. The light probe images in the light probe image gallery are in the RGBE format, which lets us easily use them in Ward's Radiance global illumination renderer. (We'll see how to do precisely that in the next section.)



2 Several light probe images from the Light Probe Image Gallery at http://www.debevec.org/Probes. The light is from (a) a residential kitchen, (b) the eucalyptus grove at UC Berkeley, (c) the Uffizi gallery in Florence, Italy, and (d) Grace Cathedral in San Francisco.

Figure 3 (next page) shows a series of images used in creating a light probe image. To acquire these images, we placed a three-inch polished ball bearing on top of a tripod at Funston Beach near San Francisco and used a digital camera with a telephoto zoom lens to take a series of exposures of the ball. Being careful not to disturb the camera, we took pictures at shutter speeds ranging from 1/4 second to 1/10000 second, allowing



3 A series of differently exposed images of a mirrored ball photographed at Funston Beach near San Francisco. I merged the exposures, ranging in shutter speed from 1/4 second to 1/1000 second, into a high dynamic range image so we can use it as an IBL environment.

the camera to properly image everything from the dark cliffs to the bright sky and the setting sun. We assembled the images using code similar to that now found in mkhdr and HDR Shop, yielding a high dynamic range, linear-response image.

Illuminating synthetic objects with real light

IBL is now supported by several commercial renderers, including LightWave 3D, Entropy, and Blender. For this tutorial, we'll use the freely downloadable Radiance lighting simulation package written by Greg Ward at Lawrence Berkeley Laboratories. Radiance is a Unix package, which means that to use it you'll need to use a computer running Linux or an SGI or Sun workstation. In this tutorial, we'll show how to perform IBL to illuminate synthetic objects in Radiance in just seven steps.

1. Download and install Radiance

First, test to see if you already have Radiance installed by typing `which rpict` at a Unix command prompt. If the shell returns "Command not found," you'll need to install Radiance. To do this, visit the Radiance Web site at http://radsite.lbl.gov/radiance and click on the download option. As of this writing, the current version is 3.1.8, and it's precompiled for SGI and Sun workstations. For other operating systems, such as Linux, you can download the source files and then compile the executable programs using the `makeall` script. Once installed, make sure that the Radiance binary directory is in your `$PATH` and that your `$RAYPATH` environment variable includes the Radiance library directory. Your system administrator should be able to help you if you're not familiar with installing software packages on Unix.

2. Create the scene

The first thing we'll do is create a Radiance scene file. Radiance scene files contain the specifications for your scene's geometry, reflectance properties, and lighting. We'll create a simple scene with a few spheres sitting on a platform. First, let's specify the material properties we'll use for the spheres. Create a new directory and then call up your favorite text editor to type in the following material specifications to the file scene.rad:

```
# Materials

void plastic red_plastic
0
0
5 .7 .1 .1 .06 .1

void metal steel
0
0
5 0.6 0.62 0.68 1 0

void metal gold
0
0
5 0.75 0.55 0.25 0.85 0.2

void plastic white_matte
0
0
5 .8 .8 .8 0 0
```

```
rview -vtv -vp 8 2.5 -1.5 -vd -8 -2.5 1.5 -vu 0 1 0 -vh 60 -vv 40
```

4 Use your text editor to create the file camera.vp with the camera parameters as the file's first and only line.

```
void dielectric crystal
0
0
5 .5 .5 .5 1.5 0

void plastic black_matte
0
0
5 .02 .02 .02 .00 .00

void plastic gray_plastic
0
0
5 0.25 0.25 0.25 0.06 0.0
```

These lines specify the diffuse and specular characteristics of the materials we'll use in our scene, including crystal, steel, and red plastic. In the case of the red plastic, the diffuse RGB color is (.7, .1, .1), the proportion of light reflected specularly is .06, and the specular roughness is .1. The two zeros and the five on the second through fourth lines are there to tell Radiance how many alphanumeric, integer, and floating-point parameters to expect.

Now let's add some objects with these material properties to our scene. The objects we'll choose will be some spheres sitting on a pedestal. Add the following lines to the end of scene.rad:

```
# Objects

red_plastic sphere ball0
0
0
4 2 0.5 2 0.5

steel sphere ball1
0
0
4 2 0.5 -2 0.5

gold sphere ball2
0
0
4 -2 0.5 -2 0.5

white_matte sphere ball3
0
0
4 -2 0.5 2 0.5

crystal sphere ball4
0
0
4 0 1 0 1

!genworm black_matte twist \
 "cos(2*PI*t)*(1+0.1*cos(30*PI*t))" \
 "0.06+0.1+0.1*sin(30*PI*t)" \
 "sin(2*PI*t)*(1+0.1*cos(30*PI*t))" \
 "0.06" 200 | xform -s 1.1 -t 2 0 2 \
 -a 4 -ry 90 -i 1

!genbox gray_plastic pedestal_top 8 \
 0.5 8 -r 0.08 | xform -t -4 -0.5 \
 -4
!genbox gray_plastic pedestal_shaft \
 6 16 6 | xform -t -3 -16.5 -3
```

These lines specify five spheres made from various materials sitting in an arrangement on the pedestal. The first sphere, **ball0**, is made of the **red_plastic** material and located in the scene at (2,0.5,2) with a radius of 0.5. The pedestal itself is composed of two beveled boxes made with the Radiance **genbox** generator program. In addition, we invoke the **genworm** program to create some curly iron rings around the spheres. (You can leave the **genworm** line out if you want to skip some typing; also, the backslashes indicate line continuations which you can omit if you type everything on one line.)

3. Add a traditional light source

Next, let's add a traditional light source to the scene to get our first illuminated glimpse —without IBL—of what the scene looks like. Add the following lines to scene.rad to specify a traditional light source:

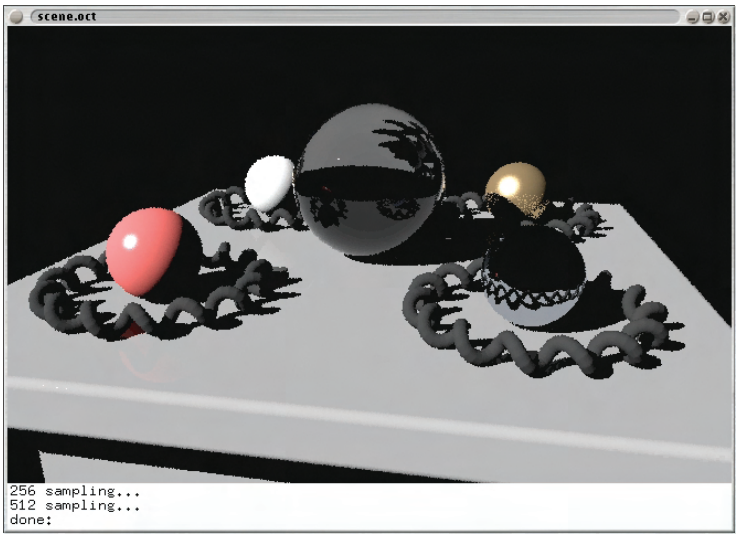
```
# Traditional Light Source

void light lightcolor
0
0
3 10000 10000 10000

lightcolor source lightsource
0
0
4 1 1 1 2
```

4. Render the scene with traditional lighting

In this step, we'll create an image of the scene. First, we need to use the **oconv** program to process the scene file into an octree file for Radiance to render. Type the following command at the Unix command prompt:

```
# oconv scene.rad > scene.oct
```

The **#** indicates the prompt, so you don't need to type it. This will create an octree file scene.oct that can be rendered in Radiance's interactive renderer **rview**. Next, we need to specify a camera position. This can be done as command arguments to **rview**, but to make things



5 The Radiance rview interactive renderer viewing the scene as illuminated by a traditional light source.

simpler, let's store our camera parameters in a file. Use your text editor to create the file camera.vp with the camera parameters as the file's first and only line (see Figure 4). In the file, this should be typed as a single line.

These parameters specify a perspective camera (`-vtv`) with a given viewing position (`-vp`), direction (`-vd`), and up vector (`-vu`) and with horizontal (`-vh`) and vertical (`-vv`) fields of view of 60 and 40 degrees, respectively. (The `rview` text at the beginning of the line is a standard placeholder in Radiance camera files, not an invocation of the `rview` executable.)

Now let's render the scene in `rview`. Type:

```
# rview -vf camera.vp scene.oct
```

In a few seconds, you should get an image window similar to the one in Figure 5. The image shows the spheres on the platform, surrounded by the curly rings, and illuminated by the traditional light source. The image might or might not be pleasing, but it certainly looks computer generated. Now let's see if we can make it more realistic by lighting the scene with IBL.

5. Download a light probe image

Visit the Light Probe Image Gallery at http://www.debevec.org/Probes and choose a light probe image to download. The light probe images without concentrated light sources tend to produce good-quality renders more quickly, so I'd recommend starting with the beach, uffizi, or kitchen probes. Here we'll choose the beach probe for the first example. Download the beach_probe.hdr file by shift-clicking or right-clicking "Save Target As..." or "Save Link As..." and then view it using the Radiance image viewer `ximage`:

```
# ximage beach_probe.hdr
```

If the probe downloaded properly, a window should pop up displaying the beach probe image. While the window is up, you can click and drag the mouse pointer over a region of the image and then press "=" to re-expose the image to properly display the region of interest. If the image didn't download properly, try downloading and expanding the all_probes.zip or all_probes.tar.gz archive from the same Web page, which will download all the light probe images and preserve their binary format. When you're done examining the light probe image, press the "q" key in the `ximage` window to dismiss the window.

6. Map the light probe image onto the environment

Let's now add the light probe image to our scene by mapping it onto an environment surrounding our objects. First, we need to create a new file that will specify the mathematical formula for mapping the light probe image onto the environment. Use your text editor to create the file angmap.cal with the following content (the text between the curly braces is a comment that you can skip typing if you wish):

```
{
angmap.cal

Convert from directions in the world \
 (Dx, Dy, Dz) into (u,v) \
 coordinates on the light probe \
 image

-z is forward (outer edge of sphere)
+z is backward (center of sphere)
+y is up (toward top of sphere)
}

d = sqrt(Dx*Dx + Dy*Dy);

r = if(d, 0.159154943*acos(Dz)/d,0);

u = 0.5 + Dx * r;
v = 0.5 + Dy * r;
```

This file will tell Radiance how to map direction vectors in the world (Dx, Dy, Dz) into light probe image coordinates (u, v). Fortunately, Radiance accepts these coordinates in the range of zero to one (for square images) no matter the image size, making it easy to try out light probe images of different resolutions. The formula converts from the angular map version of the light probe images in the light probe image gallery, which differs from the mapping a mirrored ball produces. If you need to convert a mirrored-ball image to this format, HDR Shop has a Panoramic Transformations function for this purpose.

Next, comment out (by adding #'s at the beginning of the lines) the traditional light source in scene.rad that we added in step 3:

```
#lightcolor source lightsource
#0
#0
#4 1 1 1 2
```

Note that these aren't new lines to add to the file but lines to modify from what you've already entered. Now, add the following to the end of scene.rad to include the IBL environment:

```
# Image-Based Lighting Environment

void colorpict hdr_probe_image
7 red green blue beach_probe.hdr
 angmap.cal u v
0
0

hdr_probe_image glow light_probe
0
0
4 1 1 1 0

light_probe source ibl_environment
0
0
4 0 1 0 360
```

The **colorpict** sequence indicates the name of the light probe image and the calculations file to use to map directions to image coordinates. The **glow** sequence specifies a material property comprising the light probe image treated as an emissive glow. Finally, the **source** specifies the geometry of an infinite sphere mapped with the emissive glow of the light probe. When Radiance's rays hit this surface, their illumination contribution will be taken to be the light specified for the corresponding direction in the light probe image.

Finally, because we changed the scene file, we need to update the octree file. Run **oconv** once more to do this:

```
# oconv scene.rad > scene.oct
```

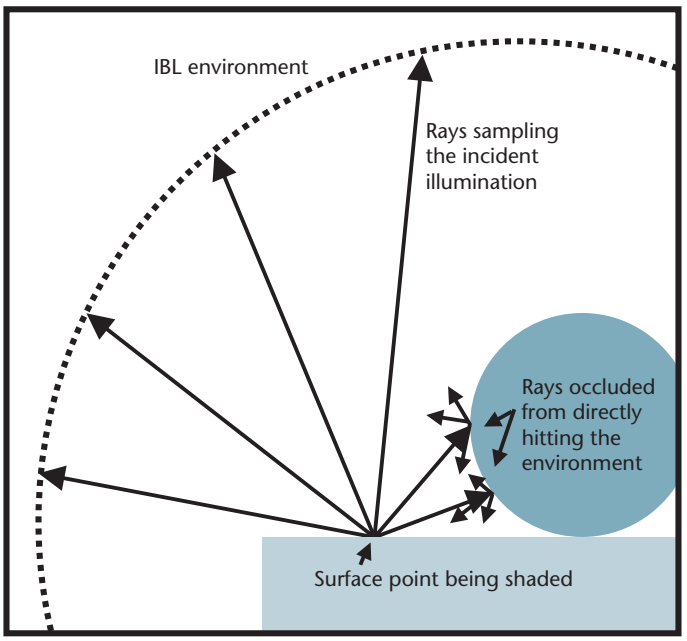
7. Render the scene with IBL

Let's now render the scene using IBL. Enter the following command to bring up a rendering in **rview**:

```
# rview -ab 1 -ar 5000 -aa 0.08 -ad \
 128 -as 64 -st 0 -sj 1 -lw 0 -lr \
 8 -vf camera.vp scene.oct
```

Again, you can omit the backslashes if you type the whole command as one line. In a few moments, you should see the image in Figure 6 begin to take shape. Radiance is tracing rays from the camera into the scene, as Figure 7 illustrates. When a ray hits the environment, it takes as its pixel value the corresponding value in the light probe image. When a ray hits a particular point on an object, Radiance calculates the color and intensity of the incident illumination (also known as irradiance) on that point by sending out a multitude of rays (in this case 192 of them) in random directions to quantify the light arriving at that point on the object. Some of these rays will hit the environment, and others will hit other parts of the object, causing Radiance to recurse into computing the light coming from this new part of the object. After Radiance computes the illumination on the object point, it calculates the light reflected toward the camera based on the object's material properties and this becomes the pixel value of that point of the object. Images calculated in this way can take a while to render, but they produce a faithful rendition of how the captured illumination would illuminate the objects.

6 The spheres on the pedestal illuminated by the beach light probe image from Figure 3.



7 How Radiance traces rays to determine the incident illumination on a surface from an IBL environment.

The command-line arguments to **rview** tell Radiance how to perform the lighting calculations. The **-ab 1** indicates that Radiance should produce only one ambient-bounce recursion in computing the object's illumination—more accurate simulations could be produced with a value of 2 or higher. The **-ar** and **-aa** set the resolution and accuracy of the surface illumination calculations, and the **-ad** and **-as** set the number of rays traced out from a surface point to compute its illumina-

8 The objects illuminated by the kitchen, eucalyptus grove, Uffizi Gallery, and Grace Cathedral light probe images in Figure 2.

tion. The `-st`, `-sj`, `-lw`, and `-lr` specify how the rays should be traced for glossy and shiny reflections. For more information on these and more Radiance parameters, see the reference guide on the Radiance Web site.

When the render completes, you should see an image of the objects as illuminated by the beach lighting environment. The synthetic steel ball reflects the environment and the other objects directly. The glass ball both reflects and refracts the environment, and the diffuse white ball shows subtle shading, which is lighter toward the sunset and darkest where the ball contacts the pedestal. The rough specular reflections in the red and gold balls appear somewhat speckled in this medium-resolution rendering; the reason is that Radiance sends out just one ray for each specular sample (regardless of surface roughness) rather than the much greater number it sends out to compute the diffuse illumination. Rendering at a higher resolution and filtering the image down can alleviate this effect.

We might want to create a particularly high-quality rendering using the command-line `rpict` renderer, which outputs the rendered image to a file. Run the following `rpict` command:

```
# rpict -x 800 -y 800 -t 30 -ab 1 - \
 ar 5000 -aa 0.08 -ad 128 -as 64 - \
 st 0 -sj 1 -lw 0 -lr 8 -vf \
 camera.vp scene.oct > render.hdr
```

The command-line arguments to `rpict` are identical to `rview` except that one also specifies the maximum x and y resolutions for the image (here, 800×800 pixels) as well as how often to report back on the rendering progress (here, every 30 seconds.) On an 800-MHz computer, this should take approximately 10 minutes. When it completes, we can only view the rendered output image with the ximage program. To produce high-quality renderings, you can increase the x and y resolutions to high numbers, such as $3{,}000 \times 3{,}000$ pixels and then filter the image down to produce an antialiased rendering. We can perform this filtering down by using either Radiance's `pfilt` command or the HDR Shop. To filter a $3{,}000 \times 3{,}000$ pixel image down to $1{,}000 \times 1{,}000$ pixels using `pfilt`, enter:

```
# pfilt -1 -x /3 -y /3 -r 1 \
 render.hdr > filtered.hdr
```

I used this method for the high-quality renderings in this article. To render the scene with different lighting environments, download a new probe image, change the beach_probe.hdr reference in the scene.rad file, and call `rview` or `rpict` once again. Light probe images with concentrated light sources such as grace and stpeters will require increasing the `-ad` and `-as` sampling parameters to the renderer to avoid mottled renderings. Figure 8 shows renderings of the objects illuminated by the light probes in Figure 2. Each rendering shows different effects of the lighting, from the particularly soft shadows under the spheres in the overcast Uffizi environment to the focused pools of light from the stained glass windows under the glass ball in the Grace Cathedral environment.

Advanced IBL

This tutorial has shown how to illuminate synthetic objects with measurements of real light, which can help the objects appear as if they're actually in a real-world scene. We can also use the technique to light large-scale environments with captured illumination from real-world skies. Figure 9 shows a computer

9 A computer model of the ruins of the Parthenon as illuminated just after sunset by a sky captured in Marina del Rey, California. Modeled by Brian Emerson and Yikuong Chen and rendered using the Arnold global illumination system.

10 A rendering from the Siggraph 99 film *Fiat Lux*, which combined image-based modeling, rendering, and lighting to place monoliths and spheres into a photorealistic reconstruction of St. Peter's Basilica.

model of a virtual environment of the Parthenon illuminated by a real-world sky captured with high dynamic range photography.

We can use extensions of the basic IBL technique in this article to model illumination emanating from a geometric model of the environment rather than from an infinite sphere of illumination and to have the objects cast shadows and appear in reflections in the environment. We used these techniques[3] to render various animated synthetic objects into an image-based model of St. Peter's Basilica for the Siggraph 99 film *Fiat Lux*, (see Figure 10). (You can view the full animation at http://www.debevec.org/FiatLux/.)

Some more recent work[7] has shown how to use IBL to illuminate real-world objects with captured illumination. The key to doing this is to acquire a large set of images of the object as illuminated by all possible lighting directions. Then, by taking linear combinations of the color channels of these images, images can be produced showing the objects under arbitrary colors and intensities of illumination coming simultaneously from all possible directions. By choosing the colors and intensities of the incident illumination to correspond to those in a light probe image, we can show the objects as they would be illuminated by the captured lighting environment, with no need to model the objects' geometry or reflectance properties. Figure 11 shows a collection of real objects illuminated by two of the light probe images from Figure 2. In these renderings, we used the additional image-based technique of environment matting[8] to compute high-resolution refractions and reflections of the background image through the objects.

11 Real objects illuminated by the Eucalyptus grove and Grace Cathedral lighting environments from Figure 2.

Conclusion

IBL lets us integrate computer-generated models into real-world environments according to the principles of global illumination. It requires a few special practices for us to apply it, including taking omnidirectional photographs, recording images in high dynamic range, and including measurements of incident illumination as sources of illumination in computer-generated scenes. After some experimentation and consulting the Radiance reference manual, you should be able to adapt these examples to your own scenes and applications. With a mirrored ball and a digital camera, you should be able to acquire your own lighting environments as well. For more information, please explore the course notes for the Siggraph 2001 IBL course at http://www.debevec.org/IBL2001. Source files and more image-based lighting examples are available at http://www.debevec.org/CGAIBL. ■

References

1. J.F. Blinn, "Texture and Reflection in Computer Generated Images," *Comm. ACM*, vol. 19, no. 10, Oct. 1976, pp. 542-547.
2. G.S. Miller and C.R. Hoffman, "Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments," *Proc. Siggraph 84*, Course Notes for Advanced Computer Graphics Animation, ACM Press, New York, 1984.
3. P. Debevec, "Rendering Synthetic Objects Into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography," *Computer Graphics* (Proc. Siggraph 98), ACM Press, New York, 1998, pp. 189-198.
4. N. Greene, "Environment Mapping and Other Applications of World Projections," *IEEE Computer Graphics and Applications*, vol. 6, no. 11, Nov. 1986, pp. 21-29.
5. P.E. Debevec and J. Malik, "Recovering High Dynamic Range Radiance Maps from Photographs," *Computer Graphics* (Proc. Siggraph 97), ACM Press, New York, 1997, pp. 369-378.
6. G. Ward, "Real Pixels," *Graphics Gems II*, J. Arvo, ed., Academic Press, Boston, 1991, pp. 80-83.
7. P. Debevec et. al, "Acquiring the Reflectance Field of a Human Face," *Computer Graphics* (Proc. Siggraph 2000), ACM Press, New York, 2000, pp. 145-156.
8. D.E. Zongker et. al, "Environment Matting and Compositing," *Computer Graphics* (Proc. Siggraph 99), ACM Press, New York, 1999, pp. 205-214.

Paul Debevec *is an executive producer at the University of Southern California's Institute for Creative Technologies, where he directs research in virtual actors, virtual environments, and applying computer graphics to creative projects. For the past five years, he has worked on techniques for capturing real-world illumination and illuminating synthetic objects with real light, facilitating the realistic integration of real and computer-generated imagery. He has a BS in math and a BSE in computer engineering from the University of Michigan and a PhD in computer science from University of California, Berkeley. In August 2001, he received the Significant New Researcher Award from ACM Siggraph for his innovative work in the image-based modeling and rendering field. He is a member of ACM Siggraph and the Visual Effects Society, and is a program cochair for the 2002 Eurographics Workshop on Rendering.*

Readers may contact Paul Debevec at USC/ICT, 13274 Fiji Way, 5th Floor, Marina Del Rey, CA 90292, email paul@debevec.org.

Direct HDR Capture of the Sun and Sky

Jessi Stumpfel Andrew Jones Andreas Wenger
Chris Tchou Tim Hawkins Paul Debevec *

University of Southern California Institute for Creative Technologies†

ABSTRACT

We present a technique for capturing the extreme dynamic range of natural illumination environments that include the sun and sky, which has presented a challenge for traditional high dynamic range photography processes. We find that through careful selection of exposure times, aperture, and neutral density filters that this full range can be covered in seven exposures with a standard digital camera. We discuss the particular calibration issues such as lens vignetting, infrared sensitivity, and spectral transmission of neutral density filters which must be addressed. We present an adaptive exposure range adjustment technique for minimizing the number of exposures necessary. We demonstrate our results by showing time-lapse renderings of a complex scene illuminated by high-resolution, high dynamic range natural illumination environments.

Categories and Subject Descriptors

I.4.1 [**Image Processing and Computer Vision**]: Digitization and Image Capture; I.3.7 [**Computing Methodologies**]: Computer Graphics Three-Dimensional Graphics and Realism

General Terms

Measurement, Lighting, Photometry

Keywords

Image-based lighting, Real-world capture, High dynamic range photography

1. INTRODUCTION

A significant area for recent research has been in digitizing the geometry, reflectance properties, and illumination of real-world scenes to create more realistic computer renderings. Recent techniques employing omnidirectional high dynamic range photography have produced useful datasets of real-world illumination environments [5], which have been used as sources of illumination for both offline [4] and real-time [18, 19] rendering techniques. While datasets seen so far include both indoor and outdoor lighting environments, current techniques have not been able to record an outdoor environment that includes a directly visible sun, which is a common and important type of environment to capture.

Figure 1: The camera with 180° fisheye lens and laptop on the roof of our building (left). This is a typical assembled probe (right) taken with the fisheye lens and neutral density filter spanning the 17 stops of the sky and sun.

Capturing the upper hemisphere of the sky is easily done using a fisheye lens, but there are two principal challenges in capturing the full dynamic range of outdoor illumination. The first is the breadth of the range - the sun can be well over five orders of magnitude (or seventeen stops) brighter than the sky and clouds, which is a greater range than can be covered with typically available shutter speeds. The second is the absolute intensity of the sun, which is much brighter than what cameras are designed to capture. In this paper, we overcome these problems by using a combination of varying shutter speed, aperture, and a properly chosen neutral density filter, and we describe the calibration procedures necessary to make use of all three of these light attenuation mechanisms. With optimized settings, we are able to capture the full dynamic range of the sky using at most seven exposures using a camera with a 12-bit linear-response sensor. In this work we also discuss an adaptive capture technique in which fewer exposures are acquired when the sun is not visible.

To demonstrate these techniques we captured several full days of natural outdoor illumination which we use to render an outdoor scene under time-lapse illumination conditions.

*{stumpfel,jones,wenger,tchou,timh,debevec}@ict.usc.edu
†13274 Fiji Way, Marina Del Rey, CA 90292

.

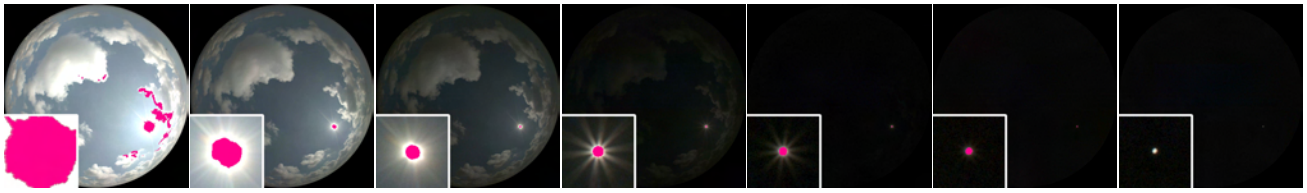


Figure 2: HDR sequence and camera settings to span the 17 stops of the sky and sun in 7 exposures. A detailed view of the sun is shown in the bottom left of each image; pink regions indicate saturated pixels. The darkest image is the only image that does not saturate for the sun.

2. RELATED WORK

To date, most computer graphics renderings illuminated by sun and sky light have been done using analytic sky models as in [16, 21] and the "gensky" function in [22]. A model specifically designed for realistic rendering of large-scale environments was presented in [17]; this model used an atmospheric scattering model to compute not only the colors in the sky but also physically-based aerial perspective of surfaces near the ground. A night sky model was developed by [11] where they model the effect of the sun, moon, atmosphere, orbiting dust, stars, and the Milky Way. [23] adapted an analytic sky model so that it could be fit to a small set of sky intensity measurements for an inverse rendering application. Work has also been done to efficiently compute the lighting of outdoor natural illumination on synthetic scenes such as [3]. These models can produce synthetic skies that match ideal conditions well, but they do not model the more visually interesting aspects of cloud formation and motion. Several techniques have been proposed to simulate cloud formation, illumination, and evolution [9, 7, 15]. These techniques produce synthetic 3D clouds that are realistic in appearance, but do not necessarily represent the full range of atmospheric effects encountered in nature, or the specific types of cloud formation found in particular geographic locations.

In the meteorological community, instruments and cameras monitor weather year round at locations worldwide. The goal in this case is to compile weather statistics, such as average cloud cover and precipitation; one such study is presented by [14]. These statistics do not require capturing the full dynamic range of the sun and sky. Instead they use a moving arm to block the view of the sun from the sensor.

In our work, we use an image-based technique to directly image the sky, clouds, and sun based on the illumination capture process of [4], adapted to capture the full dynamic range of the sky up to and including the sun. A technique proposed in [6] reconstructs the intensity of the sun by observing a diffuse sphere in addition to a mirrored sphere used to capture the sky; however, this fails to capture brightness detail in the circumsolar region and is less useful for rendering direct views of the sky. Finally, we use our technique to capture time-lapse natural illumination measurements throughout the day in a manner similar to the high dynamic range video project of [12].

3. CAPTURING THE FULL HDR RANGE OF THE SUN AND SKY

The roof of our laboratory provided an unobstructed view of the horizon in all directions. The entire sky could be imaged from this vantage point by placing a Canon EOS 1DS camera pointing up, equipped with a 8mm Sigma fisheye lens [Figure 1]. While the Canon camera provides automatic exposure bracketing, using this to adjust the exposure between 2 seconds and 1/8000th of a second yields only 14 stops. Shutter speed times longer than 2 seconds are impractical because of the motion of the clouds. Since each stop changes the light level by a factor of 2, that is a dynamic range of 2^{14} (or 16384). This is insufficient to capture the dynamic range of a typical sunny sky.

A library provided by Canon [1] allowed us to control the camera from a laptop, while downloading the images to the laptop's disk. This allowed us to programmatically adjust the aperture and exposure, acquiring an image sequence detailed in Figure 2. This sequence spanned the needed 17 stops in a manner sufficient for reconstructing the HDR image, and required approximately 50 seconds to acquire and download each series. We consider each individual image to have a dynamic range of 5 stops. This 5 stop range in addition to the 17 stop variance in the sequence, provides a 22 stop range, or over 4 million times the light. Our images are taken at most 3 stops apart, allowing sufficient dynamic range overlap between images to assemble them into an HDR image.

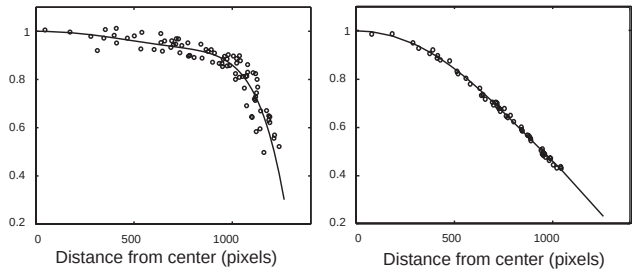


Figure 3: Graphs representing relative pixel brightness for given distances from the image center. Circles represent measurements of a reference illuminant at image locations. Curved lines indicate the fitted polynomial function. Falloff curve for Sigma 8mm lens at f/16 (left) and falloff curve for the same lens at f/4 (right).

3.1 Calibration

To achieve accurate results, it was necessary to calibrate the fisheye lens geometrically and photometrically. Geometric fisheye calibration was accomplished by marking scene

correspondences while rotating the camera around the average nodal point. We assume an ideal fisheye projection [10], and solve for image center and radius of the lens. A more sophisticated geometric model could be based on a more accurate measurement of the lens construction [13], or by fitting a lens distortion model. Also, there is significant radial intensity falloff and vignetting associated with fisheye lens. This has two causes: first, exposure falloff across the image sensor increases with wider apertures, and secondly at larger angles occlusion due to other lens elements introduces vignetting [13]. As seen in Figure 3, we fit a separate radial polynomial function to the data for both f/4 and f/16 apertures [6].

Given the sensitivity of our camera's image sensor, we found that imaging the sun without saturation required at least a 3.0 Neutral Density (ND) filter, allowing just 0.1% of the light to pass through. ND filters are usually placed in front of the lens, but no standard filters fit the curved front surface of the Sigma 8mm fisheye lens. Instead we placed a piece of a Kodak WRATTEN 3.0 Neutral Density gelatin filter between the lens and the camera.

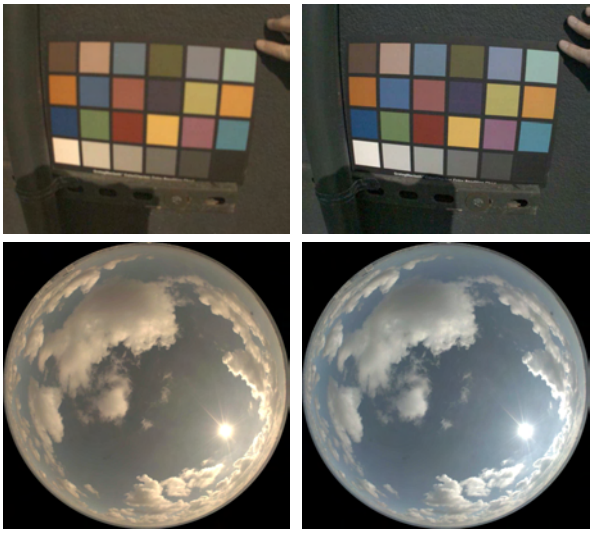


Figure 4: The neutral density filter while reducing the amount of light introduces a chromatic change as well. To correct for this a color chart was photographed with and without the ND filter. These images were then used to solve for a color correction matrix.

Figure 4 shows an uncorrected image of the sky with reddish-brown clouds taken with the ND filter. Contrary to their name, neutral density filters are typically not chromatically neutral. We estimated a correction for the chromatic shift in the visible portion of the ND filter spectrum. We photographed a Macbeth color chart in sunlight with and without the ND filter [Figure 4], then solved for a 3x3 linear color transform that most closely matched the color samples between the images. This transforms the images into the raw color space of the Canon EOS 1DS camera, which could then be mapped into a standardized color space.

To further investigate the effect of the ND filters, we used a Photo Research PR-650 spectroradiometer to measure the ND filter's spectral transmissivity [Figure 5]. Not only is the

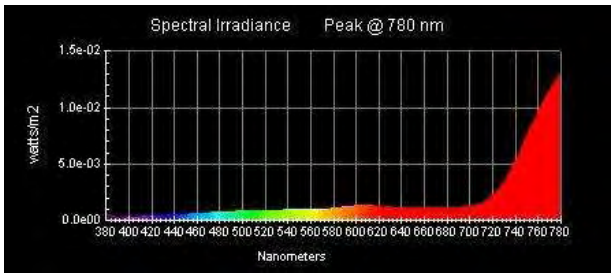


Figure 5: The spectral calibration of the 3.0 ND Kodak WRATTEN Filter, showing a gentle increase in the visible range from blue to red. A significant amount of IR light is transmitted.

filter more transmissive in the red region than the blue, but it also displays a large amount of light transmitted in the near infrared region above 720nm, which not all ND filters are designed to block.

This can be potentially problematic, as many digital cameras have a significant sensitivity to near IR light. Attenuating the visible light can only increase the error resulting from this sensitivity. To determine the influence of IR light we took multiple photographs under an incandescent illuminant. By taking measurements with and without an IR-pass filter, that removes visible wavelengths, we determined the contribution of IR light to the recorded pixel values [Table 1]. Fortunately, our experiments showed that our Canon E0S 1DS does not have a significant near IR response. However other cameras (eg. the Sony VX1000) exhibit strong IR response. As sunlight has a very strong near IR component, any camera that exhibits IR response should use an IR-cutoff filter.

IR Contribution - Sony VX1000

Filter	R	G	B
none	0.74%	0.03%	0.08%
3.0 ND	6.02%	1.00%	3.25%

IR Contribution - Canon D0S 1DS

Filter	R	G	B
none	0.007%	0.006%	0.010%
3.0 ND	0.23%	0.29%	0.57%

Table 1: Percentage of sensor response caused by incandescent infrared light for the Canon EOS 1DS and the Sony VX1000.

4. ADAPTIVE HDR IMAGING

The f/4 1 second exposure was not long enough to capture dark clouds before and during dawn and after dusk. In general, over the course of the day the sky brightness can change drastically with sun position and weather.

To address this problem, as well as to increase the speed of our capture and reduce wear on the camera shutter, we created a process to adaptively select the exposures and apertures when capturing an HDR sequence. This program analyzes each image as it is downloaded to determine when pixels in the sky are underexposed or saturated. The following pseudo-code describes this adaptive capture technique.

```
SetLightMetering(CenterWt_Avg);
while (captureHDRSequences) {
   imageA = TakePhoto(F4, Av);

   tooDark = imageA.IsUnderExposed();
   tooBright = imageA.IsSaturated();

   exposeTime = imageA.GetTv() * 8;
   while (tooDark) {
        image = TakePhoto(F4, exposeTime);
        exposeTime = exposeTime * 8;
        tooDark = image.IsUnderExposed();
    }

    exposeTime = imageA.GetTv() * 2;
    while (tooBright) {
        image = TakePhoto(F16, exposeTime);
        exposeTime = exposeTime / 8;
        tooBright = image.IsSaturated();
    }
}
```

The program captures an HDR sequence by first capturing an image in aperture priority mode (Av). The light metering on the camera is set to "Center-weighted Average Metering". This allows the camera to choose a shutter speed that properly exposes the majority of the image. This image is then analyzed to determine if shorter and/or longer exposures are required. Taking into account both aperture and exposure time, the images in our adaptive HDR sequences are always three stops apart.

We classify an image as underexposed and/or saturated by comparing the raw pixel values against an upper and lower threshold. This method was makes the analysis as fast as possible in order to maximize capture speed. The raw pixel values on Canon EOS class cameras provide 12 bits of data, exhibiting a range from 0 to 16380 in steps of 4. We have measured the Canon sensors to be linearly responsive to light up to a pixel value of 12000. At this point the pixels become less sensitive, assumably due to saturation. To avoid problems of non-linearity, we set our saturation threshold at 9000 and our underexposure threshold at 500.

Using the adaptive capture program we were able to capture an HDR image every 40 seconds with sequences consisting of three to seven pictures. The adaptive capture also allowed us to handle the highly increasing and decreasing light levels during the entire day from pre-dawn to post-dusk. The primary speed bottleneck we encountered was file transfer from the camera to the laptop, which will hopefully improve in new camera systems.

Motion artifacts appear primarily where bright fast moving clouds span multiple exposures. Away from the sun, clouds are easily captured in fewer than three exposures. Chromatic abberations from cloud movement can be reduced during HDR assembly by color-interpolating each photograph in advance.

5. RENDERING AND RESULTS

The captured lighting presented here could be used to render any outdoor scene providing realistic illumination as well as a sky backdrop. Due to the extreme contrast between the sun and sky, the standard global illumination algorithms will produce a large amount of noise, as they will have difficulty sampling the sun sufficiently.

Algorithms have been developed for intelligent sampling of the environment with priority given to high intensity regions such as in [2]. We utilize a simpler solution, approximating the sun as a 0.53 degree diameter directional light, while using traditional sampling techniques for the remainder of the sky. Pixels in the lighting environment that are above a high threshold are set to black and represented by a directional light; this directional light is placed at the weighted centroid of the blackened pixels and set to their total energy. By manually adjusting the threshold, we select a balance between accuracy and noise.

With this method we rendered a synthetic scene, illuminated by an entire day of captured light, using the Arnold global illumination system [8]. Several frames are shown in Figure 7. For these frames, the threshold was set to the peak cloud intensity in the sequence. These datasets have allowed us to visualize the full dynamic range of natural illumination over the course of the day, as seen in Figure 6.

5.1 Future Work

In the future, we would like to extend optical flow algorithms to interpolate between extreme-high dynamic range frames and create smoother time-lapse animations [12]. Motion compensation is difficult as the clouds are not visible in several photographs. We would also like to look into possibilities for removing lens flare artifacts observed in the fisheye images. Because skylight is partially polarized we are exploring techniques for capturing this polarization [20]. Methods of spectral recovery in natural illumination environments also merit investigations.

6. CONCLUSION

Using these techniques we have captured several days of light with variable weather conditions. We believe these HDR images are the first calibrated recordings of natural illumination in high- resolution including a visible sun. Exploring the practical issues of capturing the full high dynamic range of the sky using a fisheye lens with a neutral density filter allows the natural dynamics of the sky and sun to be represented faithfully in our renderings. Several captured HDR days of light can provide flexibility for choosing lighting for novel outdoor renderings. Datasets taken for this paper are available at: http://www.ict.usc.edu/graphics/skyprobes/.

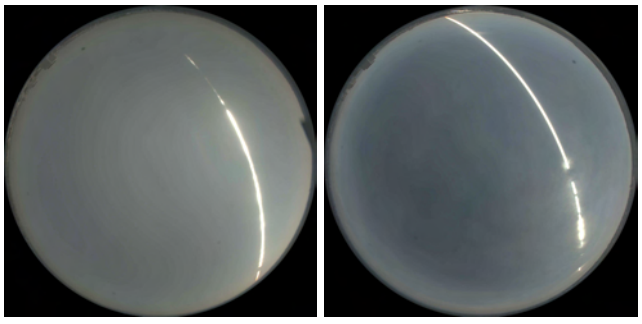


Figure 6: These images are formed as the average of 700 HDR images of the sky taken on February 19, 2004 (left) and February 23, 2004 (right) at one minute intervals. The sun streak is occluded at times by clouds over the day, and individual clouds are averaged to a constant color.

Figure 7: Rendering of a virtual model of the Parthenon with lighting from 7:04am (top left), 10:35am (top right), 4:11pm (bottom left), and 5:37pm (bottom right). Capturing high-resolution outdoor lighting environments with over 17 stops of dynamic range with time lapse photography allows for realistic lighting and effects such as simulated lens post-processed sun flare, as seen in the top left.

Acknowledgments

We would like to thank Lora Chen, David Wertheimer, Richard Lindheim, and Neil Sullivan. Also special thanks to Richard DiNinni for disarming the roof alarm when necessary. This work has been sponsored by the University of Southern California and U.S. Army contract number DAAD19-99-D-0046. Any opinions, findings, and conclusions or recommendations expressed in this paper do not necessarily reflect the views of the sponsors.

7. REFERENCES

[1] Canon digital camera software developers kit, 2004. http://consumer.usa.canon.com/.

[2] S. Agarwal, R. Ramamoorthi, S. Belongie, and H. W. Jensen. Structured importance sampling of environment maps.

[3] K. Daubert, H. Schirmacher, F. X. Sillion, and G. Drettakis. Hierarchical lighting simulation for outdoor scenes. In *Eurographics Rendering Workshop 1997*.

[4] P. Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proc. SIGGRAPH'98*.

[5] P. Debevec. Light probe image gallery, 1999. http://www.debevec.org/Probes/.

[6] P. Debevec et al. Estimating surface reflectance properties of a complex scene under captured natural illumination. *Conditionally Accepted to ACM Transactions on Graphics*, 2004.

[7] Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita, and T. Nishita. A simple, efficient method for realistic animation of clouds. In *Proc. SIGGRAPH 2000*.

[8] M. Fajardo. Monte carlo ray tracing in action. *ACM SIGGRAPH 2001 Course 29*.

[9] G. Y. Gardner. Visual simulation of clouds. In *Proc. SIGGRAPH'85*.

[10] N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 1986.

[11] H. W. Jensen, F. Durand, M. M. Stark, S. Premoze, J. Dorsey, and P. Shirley. A physically-based night sky model. In *Proc. SIGGRAPH 2001*.

[12] S. B. Kang, M. Uyttendaele, S. Winder, and R. Szeliski. High dynamic range video. In *Proc. SIGGRAPH 2003*.

[13] C. Kolb, D. Mitchell, and P. Hanrahan. A realistic camera model for computer graphics. In *Proc. SIGGRAPH'95*.

[14] C. Long and J. Deluishi. Development of an automated hemispheric sky imager for cloud fraction retrievals. In *10th Symposium on Meteorological Observations and Instrumentation*, Boston, MA, 1998. American Meteorological Society.

[15] R. Miyazaki, S. Yoshida, Y. Dobashi, and T. Nishita. A method for modeling clouds based on atmospheric fluid dynamics. In *9th Pacific Conference on Computer Graphics and Applications*, 2001.

[16] T. Nishita and E. Nakamae. Continuous tone representation of three-dimensional objects illuminated by sky light. In *Proc. SIGGRAPH'86*.

[17] A. J. Preetham, P. S. Shirley, and B. E. Smits. A practical analytic model for daylight. In *Proc. SIGGRAPH'99*.

[18] R. Ramamoorthi and P. Hanrahan. Frequency space environment map rendering. In *Proc. SIGGRAPH 2002*.

[19] P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proc. SIGGRAPH 2002*.

[20] J. Stumpfel. Outdoor lighting capture for rendering and inverse global illumination. Master's thesis, California Institute of Technology, 2004.

[21] K. Tadamura, E. Nakamae, K. Kaneda, M. Baba, H. Yamashita, and T. Nishita. Modeling of skylight and rendering of outdoor scenes. *Comput. Graph. Forum*, 12(3), 1993.

[22] G. J. Ward. The RADIANCE lighting simulation and rendering system. In *SIGGRAPH'94*.

[23] Y. Yu and J. Malik. Recovering photometric properties of architectural scenes from photographs. In *Proc. SIGGRAPH'98*.

A Median Cut Algorithm for Light Probe Sampling

Paul Debevec* USC Institute for Creative Technologies

ABSTRACT We present a technique for approximating a light probe image as a constellation of light sources based on a median cut algorithm. The algorithm is efficient, simple to implement, and can realistically represent a complex lighting environment with as few as 64 point light sources.

Introduction The quality of approximating an image-based lighting (IBL) environment as a finite number of point lights is increased if the light positions are chosen to follow the distribution of the incident illumination; this has been a goal of previous stratified sampling approaches [Cohen and Debevec 2001; Kollig and Keller 2003; Agarwal et al. 2003; Ostromoukhov et al. 2004]. In this work, we show that subdividing the image into regions of equal energy achieves this property and yields a well-conditioned and easy to implement static sampling algorithm.

Figure 1: The Grace Cathedral light probe subdivided into 64 regions of equal light energy using the median cut algorithm. The small circles are the 64 light sources chosen as the energy centroids of each region; the lights are all approximately equal in energy.

Algorithm Taking inspiration from Paul Heckbert's median-cut color quantization algorithm [Heckbert 1982], we can partition a light probe image in the rectangular latitude-longitude format into 2^n regions of similar light energy as follows:

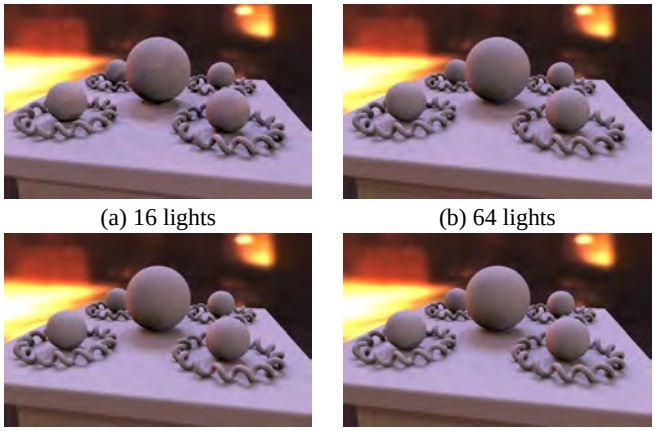
1. Add the entire light probe image to the region list as a single region.
2. For each region in the list, subdivide along the longest dimension such that its light energy is divided evenly.
3. If the number of iterations is less than n, return to step 2.
4. Place a light source at the center or centroid of each region, and set the light source color to the sum of pixel values within the region.

Implementation Calculating the total energy within regions of the image can be accelerated using a summed area table [Crow 1984]. Computing the total light energy is most naturally performed on a monochrome version of the lighting environment rather than the RGB pixel colors; such an image can be formed as a weighted average of the color channels of the light probe image, e.g. $Y = 0.2125R + 0.7154G + 0.0721B$ following ITU-R Recommendation BT.709. While the partitioning decisions are made on the monochrome image, the light source colors are computed using the corresponding regions in the original RGB image.

*Email: debevec@ict.usc.edu Web: www.debevec.org/MedianCut/

The latitude-longitude mapping over-represents regions near the poles. To compensate, the pixels of the probe image should first be scaled by $\cos\phi$ where ϕ is the pixel's angle of inclination. Determining the longest dimension of a region should also take the over-representation into account; this can be accomplished by weighting a regions width by $\cos\phi$ for an inclination ϕ at center of the region.

Results Fig. 1 shows the Grace Cathedral lighting environment partitioned into 64 light source regions, and Fig. 2 shows a small diffuse scene rendered with 16, 64, and 256 light sources chosen in this manner. Using 64 lights produces a close approximation to a computationally expensive Monte Carlo solution, and the 256-light approximation is nearly indistinguishable.



(a) 16 lights (b) 64 lights

(c) 256 lights (d) 4096 ray samples

Figure 2: (a-c) Noise-free renderings in the Grace Cathedral environment approximated by 16, 64, and 256 light sources. (d) A not quite noise-free Monte Carlo rendering using 4096 randomly chosen rays per pixel.

Conclusion The median cut technique is extremely fast compared to most other sampling techniques and produces noise-free renderings at the expense of bias inversely proportional to the number of light sources used. In future work we will investigate the stability of the technique for animated lighting environments and explore adaptations for scenes with general BRDFs.

References

AGARWAL, S., RAMAMOORTHI, R., BELONGIE, S., AND JENSEN, H. W. 2003. Structured importance sampling of environment maps. *ACM Transactions on Graphics 22*, 3 (July), 605–612.

COHEN, J. M., AND DEBEVEC, P. 2001. The LightGen HDRShop plugin. http://www.hdrshop.com/main-pages/plugins.html.

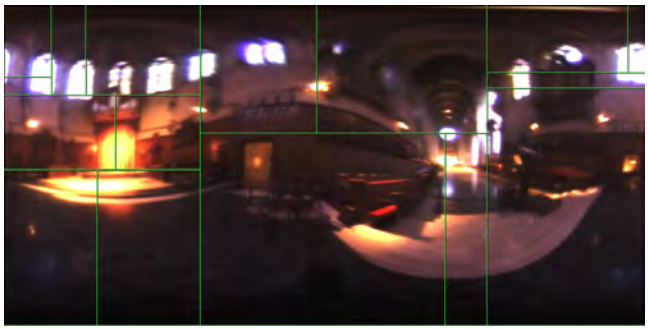
CROW, F. C. 1984. Summed-area tables for texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, vol. 18, 207–212.

HECKBERT, P. 1982. Color image quantization for frame buffer display. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 297–307.

KOLLIG, T., AND KELLER, A. 2003. Efficient illumination by high dynamic range images. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, 45–51.

OSTROMOUKHOV, V., DONOHUE, C., AND JODOIN, P.-M. 2004. Fast hierarchical importance sampling with blue noise properties. *ACM Transactions on Graphics 23*, 3 (Aug.), 488–495.

(a) 4 regions

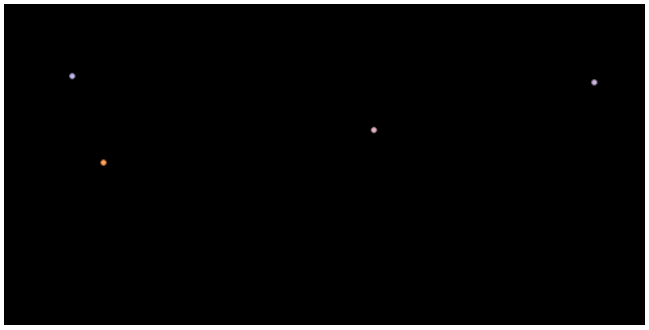


(b) 16 regions

(c) 64 regions

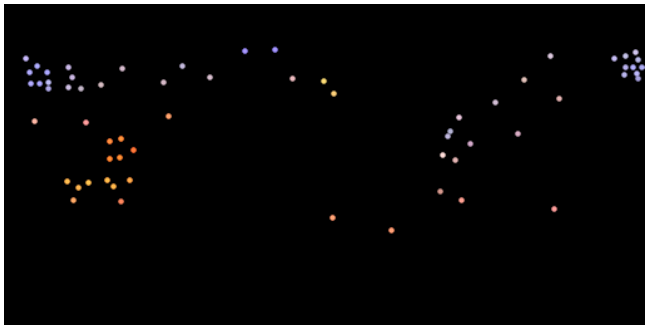
(d) 256 regions

Figure 1: The Grace Cathedral light probe subdivided into 4, 16, 64, and 256 regions of equal light energy using the median cut algorithm.



(a) 4 lights

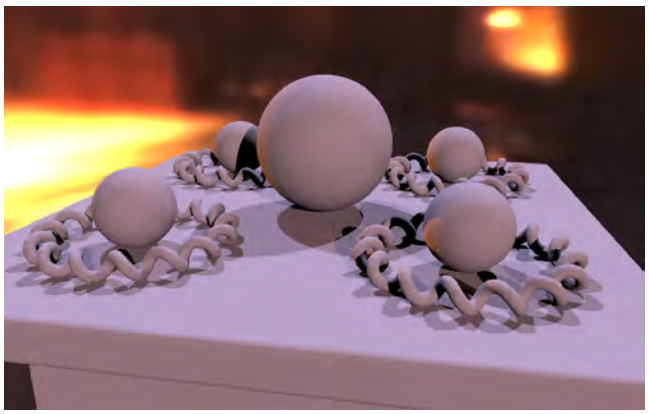
(b) 16 lights



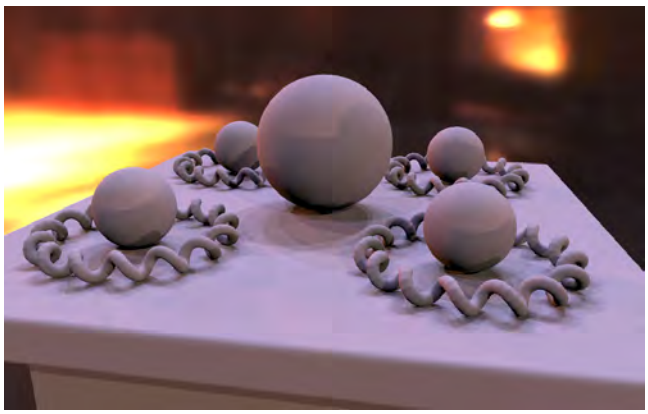
(c) 64 lights

(d) 256 lights

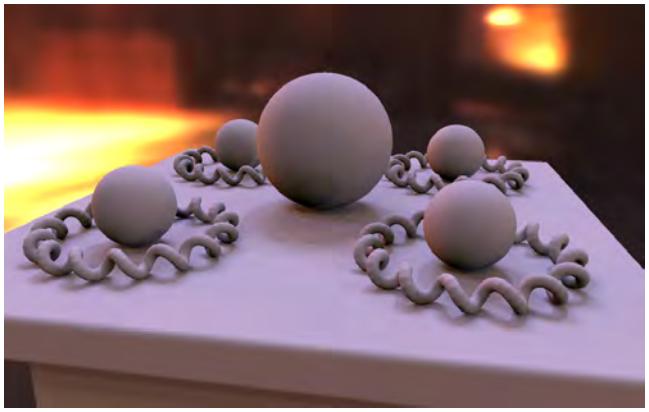
Figure 2: The Grace Cathedral light probe represented as 4, 16, 64, and 256 light sources chosen as the energy centroids of each region; each light is approximately equal energy.



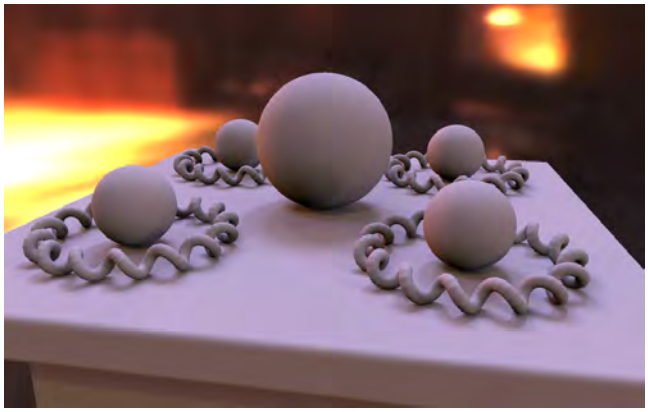
(a) 4 lights



(b) 16 lights

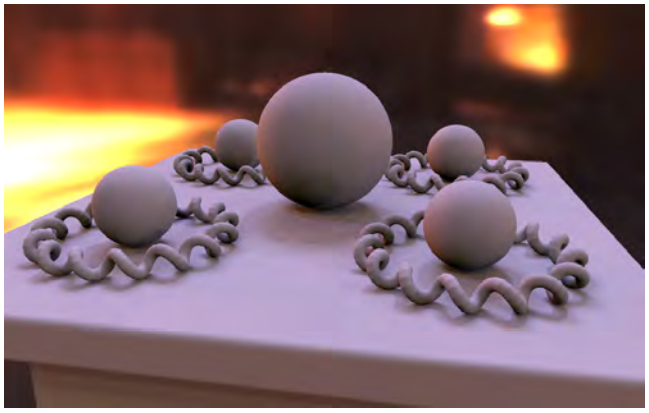


(c) 64 lights

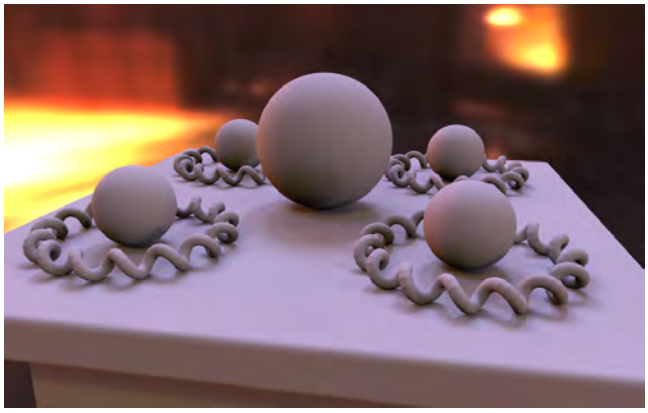


(d) 256 lights

Figure 3: Noise-free renderings in the Grace Cathedral environment approximated by 4, 16, 64, and 256 light sources.



(a) 64 lights



(b) 4096 ray samples

Figure 4: (a) Noise-free rendering in the Grace Cathedral approximated by 64 light sources, compared to (b) a not quite noise-free Monte Carlo rendering using 4096 randomly chosen rays per pixel.

Technical Introduction to OpenEXR

Last Update: 02/03/05

OpenEXR is an open-source high-dynamic-range image file format that was developed by Industrial Light & Magic. This document presents a brief overview of OpenEXR and explains concepts that are specific to this format.

Table of Contents

Features of OpenEXR

Starting in 1999, Industrial Light & Magic developed OpenEXR, a high-dynamic-range image file format for use in digital visual effects production. In early 2003, after using and refining the file format for two years, ILM released OpenEXR as an open-source C++ library. A unique combination of features makes OpenEXR a good fit for high-quality image processing and storage applications:

- high dynamic range

  Pixel data are stored as 16-bit or 32-bit floating-point numbers. With 16 bits, the representable dynamic range is significantly higher than the range of most image capture devices: 10^9 or 30 f-stops without loss of precision, and an additional 10 f-stops at the low end with some loss of precision. Most 8-bit file formats have around 7 to 10 stops.

- good color resolution

  with 16-bit floating-point numbers, color resolution is 1024 steps per f-stop, as opposed to somewhere around 20 to 70 steps per f-stop for most 8-bit file formats. Even after significant processing (e.g., extensive color correction) images tend to show no noticeable color banding.

- compatible with graphics hardware

  The 16-bit floating-point data format is fully compatible with the 16-bit frame-buffer data format used in some new graphics hardware. Images can be transferred back and forth between an OpenEXR file and a 16-bit floating-point frame buffer without losing data.

- lossless and lossy data compression

  Most of the data compression methods currently implemented in OpenEXR are lossless; repeatedly compressing and uncompressing an image does not change the image data. With the lossless compression methods, photographic images with significant amounts of film grain tend to shrink to somewhere between 35 and 55 percent of their uncompressed size. OpenEXR also supports lossy compression, which tends to shrink image files more than lossless compression, but doesn't preserve the image data exactly. New lossless and lossy compression schemes can be added in the future.

- arbitrary image channels

  OpenEXR images can contain an arbitrary number and combination of image channels, for example red, green, blue, and alpha; luminance and sub-sampled chroma channels; depth, surface normal directions, or motion vectors.

- scan-line and tiled images, multiresolution images

  Pixels in an OpenEXR file can be stored either as scan lines or as tiles. Tiled image files allow random-access to rectangular sub-regions of an image. Multiple versions of a tiled image, each with a different resolution, can be stored in a single multiresolution OpenEXR file.

  Multiresolution images, often called "mipmaps" or "ripmaps", are commonly used as texture maps in 3D rendering programs to accelerate filtering during texture lookup, or for operations like stereo image matching. Tiled multiresultion images are also useful for implementing fast zooming and panning in programs that interactively display very large images.

- ability to store additional data

  Often it is necessary to annotate images with additional data; for example, color timing information, process tracking data, or camera position and view direction. OpenEXR allows storing of an arbitrary number of extra attributes, of arbitrary type, in an image file. Software that reads OpenEXR files ignores attributes it does not understand.

- easy-to-use C++ and C programming interfaces

  In order to make writing and reading OpenEXR files easy, the file format was designed together with a C++ programming interface. Two levels of access to image files are provided: a fully general interface for writing and reading files with arbitrary sets of image channels, and a specialized interface for the most common case (red, green, blue, and alpha channels, or some subset of those). Additionally, a C-callable version of the programming interface supports reading and writing OpenEXR files from programs written in C.

  Many application programs expect image files to be scan-line based. With the OpenEXR programming interface, applications that cannot handle tiled images can treat all OpenEXR files as if they were scan-line based; the interface automatically converts tiles to scan lines.

  The C++ and C interfaces are implemented in the open-source IlmImf library.

- portability

  The OpenEXR file format is hardware and operating system independent. While implementing the C and C++ programming interfaces, an effort was made to use only language features and library functions that comply with the C and C++ ISO standards.

Overview of the OpenEXR File Format

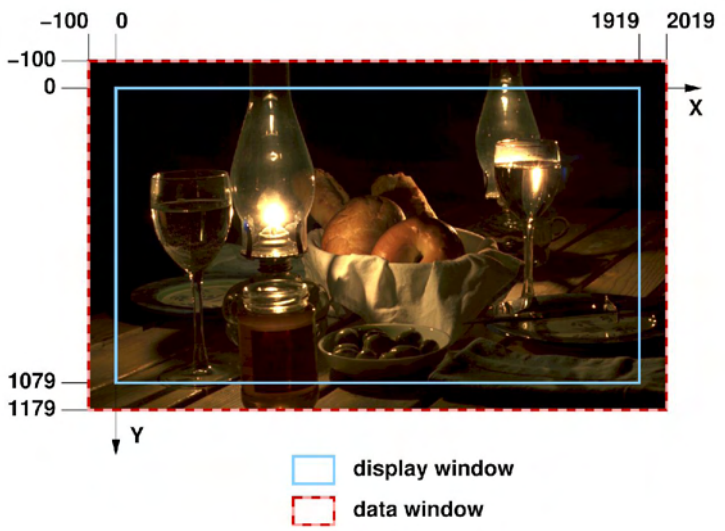
Definitions and Terminology

- *Pixel space* is a 2D coordinate system with x increasing from left to right and y increasing from top to bottom. *Pixels* are data samples, taken at integer coordinate locations in pixel space.
- The boundaries of an OpenEXR image are given as an axis-parallel rectangular region in pixel space, the *display window*. The display window is defined by the positions of the pixels in the upper left and lower right corners, (x_{min}, y_{min}) and (x_{max}, y_{max}).
- An OpenEXR file may not have pixel data for all the pixels in the display window, or the file may have pixel data beyond the boundaries of the display window. The region for which pixel data are available is defined by a second axis-parallel rectangle in pixel space, the *data window*.

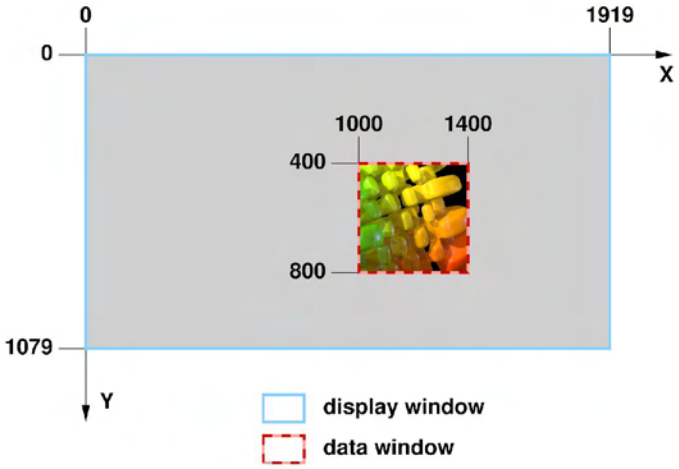
  Examples:

  - Assume that we are producing a movie with a resolution of 1920 by 1080 pixels. The display window for all frames of the movie is (0, 0) - (1919, 1079). For most images, in particular finished frames that will be recorded on film, the data window is the same as the display window, but for some images that are used in producing the finished frames, the data window differs from the display window.

  - For a background plate that will be heavily post-processed, extra pixels, beyond the edge of the film frame, are recorded and the data window is set to (-100, -100) - (2019, 1179). The extra pixels are not normally displayed. Their existence allows operations such as large-kernel blurs or simulated camera shake to avoid edge artifacts.



- While tweaking a computer-generated element, an artist repeatedly renders the same frame. To save time, the artist renders only a small region of interest close to the center of the image. The data window of the image is set to (1000, 400) - (1400, 800). When the image is displayed, the display program fills the area outside of the data window with some default color.



- Every OpenEXR image contains one or more *image channels*. Each channel has a name, a data type, and x and y *sampling rates*.

  The channel's name is a text string, for example "R", "Z" or "yVelocity". The name tells programs that read the image file how to interpret the data in the channel.

For a few channel names, interpretation of the data is predefined:

name	interpretation
R	red intensity
G	green intensity
B	blue intensity
A	alpha/opacity: 0.0 means the pixel is transparent; 1.0 means the pixel is opaque. By convention, all color channels are premultiplied by alpha, so that "foreground + (1-alpha) × background" performs a correct "over" operation.

Three channel data types are currently supported:

type name	description
HALF	16-bit floating-point numbers; for regular image data. (see The HALF Data Type, on page 13)
FLOAT	32-bit IEEE-754 floating-point numbers; used where the range or precision of 16-bit number is not sufficient, for example, depth channels.
UINT	32-bit unsigned integers; for discrete per-pixel data such as object identifiers.

The channel's x and y sampling rates, s_x and s_y, determine for which of the pixels in the image's data window data are stored in the file: Data for a pixel at pixel space coordinates (x, y) are stored only if

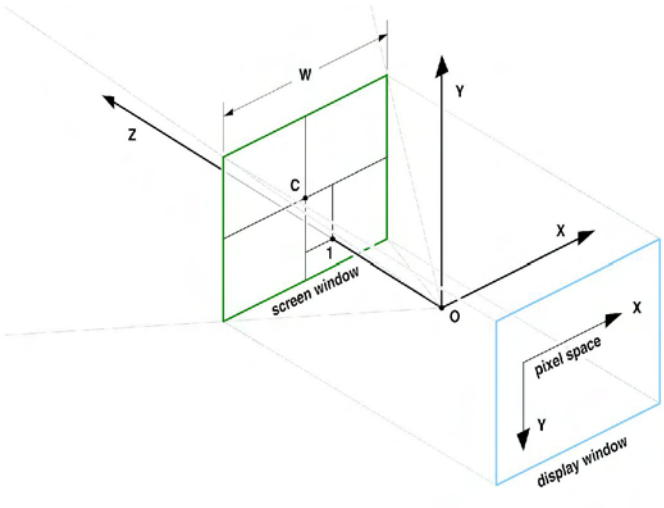
$$x \bmod s_x = 0$$

and

$$y \bmod s_y = 0.$$

For RGBA (red, green, blue, alpha) images, s_x and s_y are 1 for all channels, and each channel contains data for every pixel. For other types of images, some channels may be sub-sampled. For example, in images with one luminance channel, Y, and two croma channels, RY and BY, s_x and s_y would be 1 for the Y channel, but for the RY and BY channels, s_x and s_y might be set to 2, indicating that chroma data are only given for one out of every four pixels. (See also the Luminance/Chroma Images section, on page 11.)

- Many images are generated by a perspective *projection*. We assume that a camera is located at the origin, O, of a 3D *camera coordinate system*. The camera looks along the positive z axis. The positive x and y axes correspond to the camera's "left" and "up" directions. The 3D scene is projected onto the z = 1 plane. The image recorded by the camera is bounded by a rectangle, the *screen window*. In pixel space, the screen window corresponds to the file's display window. In the file, the size and position of the screen window are specified by the x and y coordinates of the window's center, C, and by the window's width, W. The screen window's height can be derived from C, W, the display window and the pixel aspect ratio.



- In scan-line-based files, the image's pixels are stored in horizontal rows, or *scan lines*. A file whose data window is $(x_{min}, y_{min}) - (x_{max}, y_{max})$ contains $y_{max} - y_{min} + 1$ scan lines. Each scan line contains $x_{max} - x_{min} + 1$ pixels.

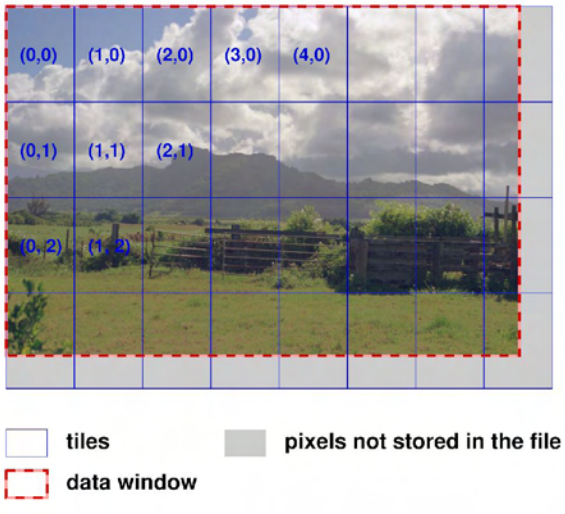
  Scan-line-based files cannot contain multiresolution images.

- In tiled files, the image is subdivided into an array of smaller rectangles, called *tiles*. Each tile contains p_x by p_y pixels. An image whose data window is $(x_{min}, y_{min}) - (x_{max}, y_{max})$ contains $ceil(w/p_x)$ by $ceil(h/p_y)$ tiles, where w and h are the width and height of the data window:

  $w = x_{max} - x_{min} + 1$

  $h = y_{max} - y_{min} + 1$

  The upper left corner of the upper left tile is aligned with the upper left corner of the data window, at (x_{min}, y_{min}). The rightmost column and the bottom row of tiles may extend outside the data window. If a tile contains pixels that are outside the data window, then those extra pixels are discarded when the tile is stored in the file.



- A single tiled OpenEXR files may contain multiple versions of the same image, each with a different resolution. Each version is called a *level*. The number of levels in a file and their resolutions depend on the file's *level mode*. Currently, OpenEXR supports three level modes:

mode name	description
ONE_LEVEL	The file contains only a single full-resolution level. A tiled ONE_LEVEL file is equivalent to a scan-line-based file; the only difference is that pixels are accessed by tile rather than by scan line.
MIPMAP_LEVELS	The file contains multiple versions of the image. Each successive level is half the resolution of the previous level in both dimensions. The lowest-resolution level contains only a single pixel. For example, if the first level, with full resolution, contains 16×8 pixels, then the file contains four more levels with 8×4, 4×2, 2×1, and 1×1 pixels respectively.
RIPMAP_LEVELS	Like MIPMAP_LEVELS, but with more levels. The levels include all combinations of reducing the resolution of the first level by powers of two independently in both dimensions. For example, if the first level contains 4×4 pixels, then the file contains eight more levels, with the following resolutions: 2×4 1×4 4×2 2×2 1×2 4×1 2×1 1×1

Levels are identified by *level numbers*. A level number is a pair of integers, (l_x, l_y). Level (0,0) is the highest-resolution level, with w by h pixels. Level (l_x, l_y) contains

$$rf(\frac{w}{2^{l_x}})$$

by

$$rf(\frac{h}{2^{l_y}})$$

pixels, where rf(x) is a rounding function, either floor(x) or ceil(x), depending on the file's *level size rounding mode* (ROUND_DOWN or ROUND_UP).

MIPMAP_LEVELS files contain only levels where $l_x = l_y$. ONE_LEVEL files contain only level (0,0).

Examples:

- The levels in a RIPMAP_LEVELS file whose highest-resolution level contains 4 by 4 pixels have the following level numbers:

		width		
		4	**2**	**1**
	4	(0,0)	(1,0)	(2,0)
height	**2**	(0,1)	(1,1)	(2,1)
	1	(0,2)	(1,2)	(2,2)

In an equivalent MIPMAP_LEVELS file, only levels (0,0), (1,1), and (2,2) are present.

- In a MIPMAP_LEVELS file with a highest-resolution level of 15 by 17 pixels, the resolutions of the remaining levels depend on the level size rounding mode:

rounding mode	**level resolutions**
ROUND_DOWN	15×17, 7×8, 3×4, 1×2, 1×1
ROUND_UP	15×17, 8×9, 4×5, 2×3, 1×2, 1×1

- In a file with multiple levels, tiles have the same size, regardless of their level. Lower-resolution levels contain fewer, rather than smaller, tiles. Within a level, a tile is identified by a pair of integer *tile coordinates*, which specify the tile's column and row. The upper left tile has coordinates (0,0). In order to identify a tile uniquely in a multiresolution file, both the tile coordinates and the level number are needed.

File Structure

An OpenEXR file has two main parts: the *header* and the *pixels*.

The header is a list of *attributes* that describe the pixels. An attribute is a named data item of an arbitrary type. To ensure that OpenEXR files written by one program can be read by other programs, certain required attributes must be present in all OpenEXR file headers:

name	**description**
displayWindow, dataWindow	The image's display and data window.
pixelAspectRatio	Width divided by height of a pixel when the image is displayed with the correct aspect ratio. A pixel's width (height) is the distance between the centers of two horizontally (vertically) adjacent pixels on the display.
channels	Description of the image channels stored in the file.
compression	Specifies the compression method applied to the pixel data of all channels in the file.
lineOrder	Specifies in what order the scan lines in the file are stored in the file (increasing Y, decreasing Y, or, for tiled images, also random Y).
screenWindowWidth, screenWindowCenter	Describe the perspective projection that produced the image (see page 5). Programs that deal with images as purely two-dimensional objects may not be able so generate a description of a perspective projection. Those programs should set screenWindowWidth to 1, and screenWindowCenter to (0, 0).
tileDescription	This attribute is required only for tiled files. It specifies the size of the tiles, and the file's level mode.

In addition to the required attributes, a program may place any number of additional attributes in the file's header. Often it is necessary to annotate images with additional data, for example color timing information, process tracking data, or camera position and view direction. Those data can be packaged as extra attributes in the image file's header.

When a scan-line-based image file is written, the scan lines must be written either in increasing Y order (top scan line first) or in decreasing Y order (bottom scan line first). When a scan-line-based file is read, random access to the scan lines is possible; the scan lines can be read in any order. Reading the scan lines in the same order as they were written causes the file to be read sequentially, without "seek" operations, and as fast as possible.

When a tiled image file is written or read, the tiles can be accessed in any order. When a tiled file is written, the IlmImf library may buffer and sort the tiles, depending on the file's line order. If the tiles in a file have been sorted into a predictable sequence, application programs reading the file can avoid slow "seek" operations by reading the tiles sequentially, in the order as they appear in the file.

For tiled files, line order is interpreted as follows:

line order	**description**
INCREASING_Y	The tiles for each level are stored in a contiguous block. The levels are ordered like this: $(0, 0)$ $(1, 0)$ ... $(n_x-1, 0)$ $(0, 1)$ $(1, 1)$ ... $(n_x-1, 1)$ ... $(0, n_y-1)$ $(1, n_y-1)$ ... (n_x-1, n_y-1), where $n_x = rf(\log_2(w)) + 1$, $n_y = rf(\log_2(h)) + 1$ if the file's level mode is RIPMAP_LEVELS, or $n_x = n_y = rf(\log_2(max(w,h)) + 1$ if the level mode is MIPMAP_LEVELS, or $n_x = n_y = 1$ if the level mode is ONE_LEVEL. In each level, the tiles are stored in the following order: $(0, 0)$ $(1, 0)$ ... $(t_x-1, 0)$ $(0, 1)$ $(1, 1)$ ... $(t_x-1, 1)$ ... $(0, t_y-1)$ $(1, t_y-1)$ ... (t_x-1, t_y-1), where t_x and t_y are the number of tiles in the x and y direction respectively, for that particular level.
DECREASING_Y	Levels are ordered as for INCREASING_Y, but within each level, the tiles are stored in this order: $(0, t_y-1)$ $(1, t_y-1)$ ... (t_x-1, t_y-1), ... $(0, 1)$ $(1, 1)$ ... $(t_x-1, 1)$ $(0, 0)$ $(1, 0)$ ... $(t_x-1, 0)$

RANDOM_Y	When a file is written, tiles are not sorted; they are stored in the file in the order they are produced by the application program. If an application program produces tiles in an essentially random order, selecting INCREASSING_Y or DECREASING_Y line order may force the IlmImf library to allocate significant amounts of memory to buffer tiles until they can be stored in the file in the proper order. If memory is scarce, allocating this extra memory can be avoided by setting the file's line order to RANDOM_Y. In this case the library doesn't buffer and sort tiles; each tile is immediately stored in the file.

Data Compression

OpenEXR currently offers four different data compression methods, with various speed versus compression ratio tradeoffs. Optionally, the pixels can be stored in uncompressed form. With fast filesystems, uncompressed files can be written and read significantly faster than compressed files.

Compressing an image with a lossless method preserves the image exactly; the pixel data are not altered. Compressing an image with a lossy method preserves the image only approximately; the compressed image looks like the original, but the data in the pixels may have changed slightly.

Supported compression schemes:

name	**description**
PIZ (lossless)	A wavelet transform is applied to the pixel data, and the result is Huffman-encoded. This scheme tends to provide the best compression ratio for the types of images that are typically processed at Industrial Light & Magic. Files are compressed and decompressed at roughly the same speed. For photographic images with film grain, the files are reduced to between 35 and 55 percent of their uncompressed size. PIZ compression works well for scan-line-based files, and also for tiled files with large tiles, but small tiles do not shrink much. (PIZ-compressed data start with a relatively long header; if the input to the compressor is short, adding the header tends to offset any size reduction of the input.)
ZIP (lossless)	Differences between horizontally adjacent pixels are compressed using the open source zlib library. ZIP decompression is faster than PIZ decompression, but ZIP compression is significantly slower. Photographic images tend to shrink to between 45 and 55 percent of their uncompressed size. Multiresolution files are often used as texture maps for 3D renderers. For this application, fast read accesses are usually more important than fast writes, or maximum compression. For texture maps, ZIP is probably the best compression method.
RLE (lossless)	Differences between horizontally adjacent pixels are run-length encoded. This method is fast, and works well for images with large flat areas, but for photographic images, the compressed file size is usually between 60 and 75 percent of the uncompressed size.

name	description
PXR24 (lossy)	After reducing 32-bit floating-point data to 24 bits by rounding, differences between horizontally adjacent pixels are compressed with zlib, similar to ZIP. PXR24 compression preserves image channels of type HALF and UINT exactly, but the relative error of FLOAT data increases to about 3×10^{-5}. This compression method works well for depth buffers and similar images, where the possible range of values is very large, but where full 32-bit floating-point accuracy is not necessary. Rounding improves compression significantly by eliminating the pixels' 8 least significant bits, which tend to be very noisy, and difficult to compress.

Luminance/Chroma Images

Encoding images with one luminance and two chroma channels, rather than as RGB data, allows a simple but effective form of lossy data compression that is independent of the compression methods listed above. The chroma channels can be stored at lower resolution than the luminance channel. This leads to significantly smaller files, with only a small reduction in image quality. The specialized RGBA interface in the IlmImf library directly supports reading and writing luminance/chroma images. When an application program writes an image file, it can choose either RGB or luminance/chroma format. When an image file with luminance/chroma data is read, the library automatically converts the pixels back to RGB.

Given linear RGB data, luminance, Y, is computed as a weighted sum of R, G, and B:

$$Y = R \times w_R + G \times w_G + B \times w_B$$

The values of the weighting factors, w_R, w_G, and w_B, are derived from the chromaticities of the image's primaries and white point. (See the RGB Color section on page 12.)

Chroma information is stored in two channels, RY and BY, which are computed like this:

$$RY = \frac{R - Y}{Y}$$

$$BY = \frac{B - Y}{Y}$$

The RY and BY channels can be low-pass filtered and subsampled without degrading the original image very much. The RGBA interface in IlmImf uses vertical and horizontal sampling rates of 2. Even though the resulting luminance/chroma images contain only half as much data, they usually do not look noticeably different from the original RGB images.

Converting RGB data to luminance/chroma format also allows space-efficient storage of gray-scale images. Only the Y channel needs to be stored in the file. The RY and BY channels can be discarded. If the original is already a gray-scale image, that is, every pixel's red, green, and blue are equal, then storing only Y preserves the image exactly; the Y channel is not subsampled, and the RY and BY channels contain only zeroes.

The HALF Data Type

Image channels of type HALF are stored as 16-bit floating-point numbers. The 16-bit floating-point data type is implemented as a C++ class, `half`, which was designed to behave as much as possible like the standard floating-point data types built into the C++ language. In arithmetic expressions, numbers of type half can be mixed freely with `float` and `double` numbers; in most cases, conversions to and from `half` happen automatically.

`half` numbers have 1 sign bit, 5 exponent bits, and 10 mantissa bits. The interpretation of the sign, exponent and mantissa is analogous to IEEE-754 floating-point numbers. `half` supports normalized and

denormalized numbers, infinities and NANs (Not A Number). The range of representable numbers is roughly 6.0×10^{-8} - 6.5×10^{4}; numbers smaller than $6.1\times10^{-5}$are denormalized. Conversions from `float` to `half` round the mantissa to 10 bits; the 13 least significant bits are lost. Conversions from `half` to `float` are lossless; all `half` numbers are exactly representable as `float` values.

The data type implemented by class half is identical to Nvidia's 16-bit floating-point format ("`fp16` / `half`"). 16-bit data, including infinities and NANs, can be transferred between OpenEXR files and Nvidia 16-bit floating-point frame buffers without losing any bits.

What's in the Numbers?

We store linear values in the RGB 16-bit floating-point numbers. By this we mean that each value is linear relative to the amount of light it represents. This implies that display of images requires some processing to account for the non-linear response of a typical display. In its simplest form, this is a power function to perform gamma correction. There are many recent papers on the subject of tone mapping to represent the high dynamic range of light values on a display. By storing linear data in the file (double the number, double the light), we have the best starting point for these downstream algorithms. Also, most commercial renderers produce linear values (before gamma is applied to output to lower precision formats).

With this linear relationship established, the question remains, What number is white? The convention we employ is to determine a middle gray object, and assign it the photographic 18% gray value, or .18 in the floating point scheme. Other pixel values can be easily determined from there (a stop brighter is .36, another stop is .72). The value 1.0 has no special significance (it is not a clamping limit, as in other formats); it roughly represents light coming from a 100% reflector (slightly brighter than paper white). But there are many brighter pixel values available to represent objects such as fire and highlights.

The range of normalized 16-bit floats can represent thirty stops of information with 1024 steps per stop. We have eighteen and a half stops over middle gray, and eleven and a half below. The denormalized numbers provide an additional ten stops with decreasing precision per stop.

Recommendations

RGB Color

Simply calling the R channel red is not sufficient information to determine accurately the color that should be displayed for a given pixel value. The IlmImf library defines a "chromaticities" attribute, which specifies the CIE x,y coordinates for red, green, blue, and white; that is, for the RGB triples (1, 0, 0), (0, 1, 0), (0, 0, 1), and (1, 1, 1). The x,y coordinates of all possible RGB triples can be derived from the chromaticities attribute. If the primaries and white point for a given display are known, a file-to-display color transform can correctly be done. The IlmImf library does not perform this transformation; it is left to the display software. The chromaticities attribute is optional, and many programs that write OpenEXR omit it. In a file doesn't have a chromaticities attribute, display software should assume that the file's primaries and the white point match the display.

Channel Names

An OpenEXR image can have any number of channels with arbitrary names. The specialized RGBA image interface assumes that channels with the names "R", "G", "B" and "A" mean red, green, blue and alpha. No predefined meaning has been assigned to any other channels. However, for a few channel names we recommend the interpretations given in the table below. We expect this table to grow over time as users employ OpenEXR for data such as shadow maps, motion-vector fields or images with more than three color channels.

name	interpretation
Y	luminance, used either alone, for gray-scale images, or in combination with RY and BY for color images.
RY, BY	chroma for luminance/chroma images, see above.
AR, AG, AB	red, green and blue alpha/opacity, for colored mattes (required to composite images of objects like colored glass correctly).

Standard Attributes

By adding attributes to an OpenEXR file, application programs can store arbitrary auxiliary data along with the image. In order to make it easier to exchange data between programs written by different people, the IlmImf library defines a set of standard attributes for commonly used data, such as colorimetric data (see RGB Color, above), time and place where an image was recorded, or the owner of an image file's content. Whenever possible, application programs should store data in standard attributes, instead of defining their own. For a current list of all standard attributes, see the IlmImf library's source code. The list grows over time, as OpenEXR users identify new types of data they would like to represent in a standard way.

Credits

The ILM OpenEXR file format was designed and implemented by Florian Kainz, Wojciech Jarosz, and Rod Bogart. The PIZ compression scheme is based on an algorithm by Christian Rouet. Josh Pines helped extend the PIZ algorithm for 16-bit and found optimizations for the float-to-half conversions. Drew Hess packaged and adapted ILM's internal source code for public release and maintains the OpenEXR software distribution. The PXR24 compression method is based on an algorithm written by Loren Carpenter at Pixar Animation Studios.

OpenEXR was developed at Industrial Light & Magic, a division of Lucas Digital Ltd. LLC, Marin County, California.

OpenEXR File Layout

Last Update: 04/17/06

This document gives an overview of the layout of OpenEXR image files as byte sequences. The text assumes that the reader is familiar with OpenEXR terms such as "channel", "attribute" or "data window". For an explanation of those terms see the Technical Introduction to OpenEXR.

This document does not define the OpenEXR file format. OpenEXR is defined as the file format that is read and written by the IlmImf open-source C++ library. If this document and the IlmImf library disagree, then the library takes precedence.

Table of Contents

Basic Data Types

An OpenEXR file is a sequence of 8-bit bytes. Groups of bytes represent basic objects such as integral numbers, floating-point numbers and text. Those objects are grouped together to form compound objects such as attributes or scan lines.

Integers

Binary integral numbers with 8, 16, 32 or 64 bits are stored as 1, 2, 4 or 8 bytes. Integral numbers can be signed or unsigned. Signed numbers are represented using two's complement. Integral numbers are little-endian, that is, the least significant byte is closest to the start of the file.

OpenEXR uses the following six integer data types:

name	signed	size in bytes
`unsigned char`	no	1
`short`	yes	2
`unsigned short`	no	2
`int`	yes	4
`unsigned int`	no	4
`unsigned long`	no	8

Floating-point numbers

Binary floating-point numbers with 16, 32 or 64 bits are stored as 2, 4 or 8 bytes. The representation of 32-bit and 64-bit floating-point numbers conforms to the IEEE 754 standard. The representation of 16-bit floating-point numbers is analogous to IEEE 754, but with 5 exponent bits and 10 bits for the fraction. The exponent bias is 15. Floating-point numbers are little-endian: the least significant bits of the fraction are in the byte closest to the beginning of the file, while the sign bit and the most significant bits of the exponent are in the byte closest to the end of the file.

The following table lists the names and sizes of OpenEXR's floating-point data types:

name	Size in bytes
`half`	2
`float`	4
`double`	8

Text

Text strings are represented as sequences of 1-byte characters of type `char`. Depending on the context, either the end of a string is indicated by a null character (0x00), or the length of the string is indicated by an `int` that precedes the string.

Packing

Data in an OpenEXR file are densely packed; the file contains no "padding". For example, consider the following C struct:

```
struct SI
{
    short s;
    int i;
};
```

on most computers, the in-memory representation an `SI` object occupies eight bytes: 2 bytes for `s`, 2 padding bytes to ensure four-byte alignment of `i`, and 4 bytes for `i`. In an OpenEXR file the same same object would consume only six bytes: 2 bytes for `s` and 4 bytes for `i`. The two padding bytes are not stored in the file.

File Layout

High-Level Layout

Depending on whether the pixels in an OpenEXR file are stored as scan lines or as tiles, the file consists of the following components:

file with scan lines:	file with tiles:
magic number	magic number
version field	version field
header	header
line offset table	tile offset table
scan line blocks	tiles

Magic Number

The magic number, of type `int`, is always 20000630 (decimal). It allows file readers to distinguish OpenEXR files from other files, since the first four bytes of an OpenEXR file are always 0x76, 0x2f, 0x31 and 0x01.

Version Field

The version field, of type `int`, is treated as two separate bit fields. The 8 least significant bits (bits 0 through 7) contain the file format version number. The 24 most significant bits (8 through 31) are treated as a set of boolean flags.

The current OpenEXR version number is 2. (Version 1 was used internally by ILM before OpenEXR was released as open source. The IlmImf library can no longer read or write version 1 files.)

Bit number 9 of the version field (bit mask 0x200) indicates how the pixels in the file are stored. If the bit is zero, the pixels are stored as scan lines; if the bit is one, the pixels are stored as tiles. The remaining 23 flags in the version field are currently unused and should be set to zero.

This means that currently there are only two valid settings for the version number field: 0x2 for scan-line based images, and 0x202 for tiled images.

Header

The header is a sequence of attributes, followed by a single null byte (0x00). The layout of an attribute is as follows:

attribute name

attribute type

attribute size

attribute value

The attribute name and the attribute type are null-terminated text strings. Excluding the null byte, the name and type must each be as least 1 byte and at most 31 bytes long.

The attribute size, of type `int`, indicates the size, in bytes, of the attribute value.

The layout of the attribute value depends on the attribute type. The IlmImf library predefines several different attribute types (see page 6). Application programs can define and store additional attribute types.

The header of every OpenEXR file must contain at least the following attributes:

attribute name	attribute type
`channels`	`chlist`
`compression`	`compression`
`dataWindow`	`box2i`
`displayWindow`	`box2i`
`lineOrder`	`lineOrder`
`pixelAspectRatio`	`float`
`screenWindowCenter`	`v2f`
`screenWindowWidth`	`float`

In addition, every tiled file must contain a tile description attribute, with name "tiles" and type "tiledesc". The tile description attribute determines the size of the tiles and the number of resolution levels in the file.

The IlmImf library ignores tile description attributes in scan-line based files. The decision whether the file contains scan lines or tiles is based on the file's version field, not on the presence of a tile description attribute.

Scan Line Blocks

One or more scan lines are stored together as a scan-line block. The number of scan lines per block depends on how the pixel data are compressed:

compression method	number of scan lines per block
`NO_COMPRESSION`	1
`RLE_COMPRESSION`	1
`ZIPS_COMPRESSION`	1
`ZIP_COMPRESSION`	16
`PIZ_COMPRESSION`	32
`PXR24_COMPRESSION`	16

Each scan line block has a y coordinate of type `int`. The block's y coordinate is equal to the pixel space y coordinate of the top scan line in the block. The top scan line block in the image is aligned with the top edge of the data window, that is, the y coordinate of the top scan line block is equal to the data window's minimum y.

If the height of the image's data window is not a multiple of the number of scan lines per block, then the block that contains the bottom scan line contains fewer scan lines than the other blocks.

The layout of a scan line block is as follows:

y coordinate

pixel data size

pixel data

The pixel data size, of type `int`, indicates the number of bytes occupied by the actual pixel data.

Within the pixel data, scan lines are stored top to bottom. Each scan line is contiguous, and within a scan line the data for each channel are contiguous. Channels are stored in alphabetical order, according to channel names. Within a channel, pixels are stored left to right.

If the file's compression method is `NO_COMPRESSION`, then the original, uncompressed pixel data are stored directly in the file. Otherwise, the uncompressed pixels are fed to the appropriate compressor, and either the compressed or the uncompressed data are stored in the file, whichever is smaller.

The layout of the compressed data depends on which compression method was applied. The compressed formats are not described here. For information on the compressed data formats, see the source code for the IlmImf library.

Line Offset Table

The line offset table allows random access to scan line blocks. The table is a sequence of scan line offsets, with one offset per scan line block. A scan line offset, of type `unsigned long`, indicates the distance, in bytes, between the start of the file and the start of the scan line block. In the table, scan line offsets are ordered according to increasing scan line y coordinates.

Tiles

The layout of a tile is as follows:

tile coordinates

pixel data size

pixel data

The tile coordinates, a sequence of four `int`s, tileX, tileY, levelX, levelY indicate the tile's position and resolution level. The pixel data size, of type `int`, indicates the number of bytes occupied by the pixel data.

The pixel in a tile data are laid out in the same way as in a scan line block, but the length of the scan lines is equal to the width of the tile, and the number of scan lines is equal to the height of the tile.

If the width of a resolution level is not a multiple of the file's tile width, then the tiles at the right edge of that resolution level have shorter scan lines. Similarly, if the height of a resolution level is not a multiple of the file's tile height, then tiles at the bottom edge of the resolution level have fewer scan lines.

Tile Offset Table

The tile offset table allows random access to tiles. The table is a sequence of tile offsets, one offset per tile. A tile offset, of type `unsigned long`, indicates the distance, in bytes, between the start of the file and the start of the tile. In the table scan line offsets are sorted the same way as tiles in `INCREASING_Y` order.

Predefined Attribute Types

The IlmImf library predefines the following attribute types:

type name	data
box2i	Four `int`s: xMin, yMin, xMax, yMax
box2f	Four `float`s: xMin, yMin, xMax, yMax
chlist	A sequence of channels followed by a null byte (0x00). Channel layout: name — zero-terminated string, from 1 to 31 bytes long pixel type — `int`, possible values are `UINT` = 0 `HALF` = 1 `FLOAT` = 2 reserved — `int`, should be zero xSampling — `int` ySampling — `int`
chromaticities	Eight `float`s: redX, redY, greenX, greenY, blueX, blueY, whiteX, whiteY
compression	`unsigned char`, possible values are `NO_COMPRESSION` = 0 `RLE_COMPRESSION` = 1 `ZIPS_COMPRESSION` = 2 `ZIP_COMPRESSION` = 3 `PIZ_COMPRESSION` = 4 `PXR24_COMPRESSION` = 5
double	`double`
envmap	`unsigned char`, possible values are `ENVMAP_LATLONG` = 0 `ENVMAP_CUBE` = 1
float	`float`
int	`int`
keycode	Seven `int`s: filmMfcCode, filmType, prefix, count, perfOffset, perfsPerFrame, perfsPerCount
lineOrder	`unsigned char`, possible values are `INCREASING_Y` = 0 `DECREASING_Y` = 1 `RANDOM_Y` = 2
m33f	9 `float`s
m44f	16 `float`s

preview	two `unsigned int`s: width, height, followed by 4×width×height `unsigned char`s of pixel data Scan lines are stored top to bottom, within a scan line pixels are stored from left to right. A pixel consists of four `unsigned char`s, R, G, B, A.
string	String length, of type `int`, followed by a sequence of `char`s.
tiledesc	Two `unsigned int`s: xSize, ySize, followed by mode, of type `unsigned char`, where mode = levelMode + roundingMode×16 Possible values for levelMode: `ONE_LEVEL` = 0 `MIPMAP_LEVELS` = 1 `RIPMAP_LEVELS` = 2 Possible values for roundingMode: `ROUND_DOWN` = 0 `ROUND_UP` = 1
timecode	Two `unsigned int`s: timeAndFlags, userData
v2i	Two `int`s
v2f	Two `float`s
v3i	Three `int`s.
v3f	Three `float`s.

Sample File

The following is an annotated byte-by-byte listing of a complete OpenEXR file. The file contains a scan-line based image with four by three pixels. The image has two channels: G, of type `HALF`, and Z, of type `FLOAT`. The pixel data are not compressed. The entire file is 415 bytes long.

The first line of text in each of the gray boxes below lists up to 16 bytes of the file in hexadecimal notation. The second line in each box shows how the bytes are grouped into integers, floating-point numbers and text strings. The third and fourth lines indicate how those basic objects form compound objects such as attributes or the line offset table.

```
76   2f   31   01   02   00   00   00   63   68   61   6e   6e   65   6c   73
     20000630     |         2         |  c    h    a    n    n    e    l    s
   magic number   |  version, flags   | attribute name
                  |                   | start of header
```

```
00   63   68   6c   69   73   74   00   25   00   00   00   47   00   01   00
\0 |  c    h    l    i    s    t   \0 |        37         |  G   \0 | HALF
   | attribute type                   | attribute size    | attribute value
```

```
00   00   00   00   00   00   01   00   00   00   01   00   00   00   5a   00
        |         0         |         1         |         1         |  Z   \0 |
```

```
02   00   00   00   00   00   00   00   01   00   00   00   01   00   00   00
FLOAT             |         0         |         1         |         1         |
                                                                              |
```

```
00   63   6f   6d   70   72   65   73   73   69   6f   6e   00   63   6f   6d
\0 |  c    o    m    p    r    e    s    s    i    o    n   \0 |  c    o    m
   | attribute name                                          | attribute type
```

```
70   72   65   73   73   69   6f   6e   00   01   00   00   00   00   64   61
 p    r    e    s    s    i    o    n   \0 |         1         | NONE| d    a
                                          | attribute size    |value|
```

```
74   61   57   69   6e   64   6f   77   00   62   6f   78   32   69   00   10
 t    a    W    i    n    d    o    w   \0 |  b    o    x    2    i   \0 |
attribute name                             | attribute type              |
```

```
00   00   00   00   00   00   00   00   00   00   00   03   00   00   00   02
   16        |         0         |         0         |         3         |
attribute size| attribute value
```

```
00   00   00   64   69   73   70   6c   61   79   57   69   6e   64   6f   77
   2         |  d    i    s    p    l    a    y    W    i    n    d    o    w
             | attribute name
```

```
00   62   6f   78   32   69   00   10   00   00   00   00   00   00   00   00
\0 |  b    o    x    2    i   \0 |        16         |         0         |
   | attribute type              | attribute size    | attribute value
```

```
 00   00   00   03   00   00   00   02   00   00   00   6c   69   6e   65   4f
    0         |         3         |         2         | l    i    n    e    O
                                                     | attribute name
```

```
 72   64   65   72   00   6c   69   6e   65   4f   72   64   65   72   00   01
  r    d    e    r   \0 |  l    i    n    e    O    r    d    e    r   \0 |
                        | attribute type                                  |
```

```
 00   00   00   00   70   69   78   65   6c   41   73   70   65   63   74   52
    1         |INCY | p    i    x    e    l    A    s    p    e    c    t    R
attribute size|value| attribute name
```

```
 61   74   69   6f   00   66   6c   6f   61   74   00   04   00   00   00   00
  a    t    i    o   \0  | f    l    o    a    t   \0 |         4         |
                         | attribute type             | attribute size    |
```

```
 00   80   3f   73   63   72   65   65   6e   57   69   6e   64   6f   77   43
   1.0         | s    c    r    e    e    n    W    i    n    d    o    w    C
attribute value| attribute name
```

```
 65   6e   74   65   72   00   76   32   66   00   08   00   00   00   00   00
  e    n    t    e    r   \0 |  v    2    f   \0 |         8         |
                             | attribute type    | attribute size    |
```

```
 00   00   00   00   00   00   73   63   72   65   65   6e   57   69   6e   64
0.0       |         0.0       |  s    c    r    e    e    n    W    i    n    d
attribute value               | attribute name
```

```
 6f   77   57   69   64   74   68   00   66   6c   6f   61   74   00   04   00
  o    w    W    i    d    t    h   \0 |  f    l    o    a    t   \0 |
                                       | attribute type              |
```

```
 00   00   00   00   80   3f   00   3f   01   00   00   00   00   00   00   5f
4        |        1.0       | \0  |                 319                 |
size     | attribute value  |     |        offset of scan line 0          |
                    end of header | start of scan line offset table
```

```
 01   00   00   00   00   00   00   7f   01   00   00   00   00   00   00   00
            351                  |                383                 |
   offset of scan line 1          |       offset of scan line 2          |
                                        end of scan line offset table |
```

```
 00   00   00   18   00   00   00   00   00   54   29   d5   35   e8   2d   5c
    0         |         24        |  0.000  |  0.042  |  0.365  |  0.092  |
    y         |  pixel data size  |  pixel data for G channel             |
  scan line 0
```

```
 28   81   3a   cf   e1   34   3e   8b   0b   bb   3d   89   74   f9   3e   01
0.000985395   |   0.176643        |   0.0913306       |   0.487217        |
pixel data for Z channel                                                  |
                                                                          |
```

```
 00   00   00   18   00   00   00   37   38   76   33   74   3b   73   38   7f
    1         |         24        |  0.527  |  0.233  |  0.932  |  0.556  |
    y         |  pixel data size  |  pixel data for G channel             |
  scan line 1
```

```
 ab   e8   3e   8a   cf   54   3f   5b   6c   11   3f   20   35   50   3d   02
0.454433      |   0.831292        |   0.56806         |   0.0508319       |
pixel data for Z channel                                                  |
                                                                          |
```

```
 00   00   00   18   00   00   00   23   3a   0a   34   02   3b   5d   3b   38
    2         |         24        |  0.767  |  0.252  |  0.876  |  0.920  |
    y         |  pixel data size  |  pixel data for G channel             |
  scan line 2
```

```
 f3   9a   3c   4d   ad   98   3e   1c   14   08   3f   4c   f3   03   3f
0.0189148     |   0.298197        |   0.531557        |   0.515431
pixel data for Z channel
                                                          end of file
```

Reading and Writing OpenEXR Image Files with the IlmImf Library

Last Update: 02/03/05

This document shows how to write C++ code that reads and writes OpenEXR image files. The text assumes that the reader is familiar with OpenEXR terms like "channel", "attribute", or "data window". For an explanation of those terms see the Technical Introduction to OpenEXR document. The OpenEXR source distribution contains a subdirectory, IlmImfExamples, with most of the code examples below. A Makefile is also provided, so that the examples can easily be compiled and run.

Table of Contents

1 Scan-line-based and Tiled OpenEXR files

In an OpenEXR file, pixel data can be stored either as scan lines or as tiles. Files that store pixels as tiles can also store multiresolution images. For each of the two storage formats (scan line or tile-based), the IlmImf library supports two reading and writing interfaces: the first, fully general, interface allows access to arbitrary channels, and supports many different in-memory pixel data layouts. The second interface is easier to use, but limits access to 16-bit (`HALF`) RGBA (red, green, blue, alpha) channels, and provides fewer options for laying out pixels in memory.

The interfaces for reading and writing OpenEXR files are implemented in the following eight C++ classes:

	tiles	scan lines	scan lines and tiles
arbitrary channels	`TiledInputFile`		`InputFile`
	`TiledOutputFile`	`OutputFile`	
RGBA only	`TiledRgbaInputFile`		`RgbaInputFile`
	`TiledRgbaOutputFile`	`RgbaOutputFile`	

The classes for reading scan-line-based images (`InputFile` and `RgbaInputFile`) can also be used to read tiled image files. This way, programs that do not need support for tiled or multiresolution images can always use the rather straightforward scan-line interfaces, without worrying about complications related to tiling and multiple resolutions. When a multiresolution file is read via a scan-line interface, only the highest-resolution version of the image is accessible.

2 Using the RGBA-only Interface for Scan-line-based Files

2.1 Writing an RGBA Image File

Writing a simple RGBA image file is fairly straightforward:

```
void
writeRgba1 (const char fileName[],
            const Rgba *pixels,
            int width,
            int height)
{
    RgbaOutputFile file (fileName, width, height, WRITE_RGBA);      // 1
    file.setFrameBuffer (pixels, 1, width);                         // 2
    file.writePixels (height);                                      // 3
}
```

Construction of an RgbaOutputFile object, in line 1, creates an OpenEXR header, sets the header's attributes, opens the file with the specified name, and stores the header in the file. The header's display window and data window are both set to `(0, 0) - (width-1, height-1)`. The channel list contains four channels, R, G, B, and A, of type `HALF`.

Line 2 specifies how the pixel data are laid out in memory. In our example, the `pixels` pointer is assumed to point to the beginning of an array of `width*height` pixels. The pixels are represented as `Rgba` structs, which are defined like this:

```
struct Rgba
{
    half r;    // red
    half g;    // green
    half b;    // blue
    half a;    // alpha (opacity)
};
```

The elements of our array are arranged so that the pixels of each scan line are contiguous in memory. The `setFrameBuffer()` function takes three arguments, `base`, `xStride`, and `ystride`. To find the address of pixel `(x,y)`, the `RgbaOutputFile` object computes

```
base + x * xStride + y * yStride.
```

In this case, `base`, `xStride` and `yStride` are set to `pixels`, `1`, and `width`, respectively, indicating that pixel `(x,y)` can be found at memory address

```
pixels + 1 * x + width * y.
```

The call to `writePixels(),` in line 3, copies the image's pixels from memory to the file. The argument to `writePixels()`, `height`, specifies how many scan lines worth of data are copied.

Finally, returning from function `writeRgba1()` destroys the local `RgbaOutputFile` object, thereby closing the file.

Why do we have to tell the `writePixels()` function how many scan lines we want to write? Shouldn't the `RgbaOutputFile` object be able to derive the number of scan lines from the data window? The IlmImf library doesn't require writing all scan lines with a single `writePixels()` call. Many programs want to write scan lines individually, or in small blocks. For example, rendering computer-generated images can take a significant amount of time, and many rendering programs want to store each scan line in the image file as soon as all of the pixels for that scan line are available. This way, users can look at a partial image before rendering is finished. The IlmImf library allows writing the scan lines in top-to-bottom or bottom-to-top direction. The direction is defined by the file header's line order attribute (`INCREASING_Y` or `DECREASING_Y`). By default, scan lines are written top to bottom (`INCREASING_Y`).

You may have noticed that in the example above, there are no explicit checks to verify that writing the file actually succeeded. If the IlmImf library detects an error, it throws a C++ exception instead of returning a C-style error code. With exceptions, error handling tends to be easier to get right than with error return

values. For instance, a program that calls our `writeRgba1()` function can handle all possible error conditions with a single try/catch block:

```
try
{
    writeRgba1 (fileName, pixels, width, height);
}
catch (const std::exception &exc)
{
    std::cerr << exc.what() << std::endl;
}
```

2.2 Writing a Cropped Image

Now we are going to store a cropped image in a file. For this example, we assume that we have a frame buffer that is large enough to hold an image with `width` by `height` pixels, but only part of the frame buffer contains valid data. In the file's header, the size of the whole image is indicated by the display window, `(0, 0) - (width-1, height-1)`, and the data window specifies the region for which valid pixel data exist. Only the pixels in the data window are stored in the file.

```
void
writeRgba2 (const char fileName[],
            const Rgba *pixels,
            int width,
            int height,
            const Box2i &dataWindow)
{
    Box2i displayWindow (V2i (0, 0), V2i (width - 1, height - 1));
    RgbaOutputFile file (fileName, displayWindow, dataWindow, WRITE_RGBA);
    file.setFrameBuffer (pixels, 1, width);
    file.writePixels (dataWindow.max.y - dataWindow.min.y + 1);
}
```

The code above is similar to that in section 2.1, where the whole image was stored in the file. Two things are different, however: When the `RgbaOutputFile` object is created, the data window and the display window are explicitly specified rather than being derived from the image's width and height. The number of scan lines stored in the file by `writePixels()` is equal to the height of the data window instead of the height of the whole image. Since we are using the default `INCREASING_Y` direction for storing the scan lines in the file, `writePixels()` starts at the top of the data window, at y coordinate `dataWindow.min.y`, and proceeds toward the bottom, at y coordinate `dataWindow.max.y`.

Even though we are storing only part of the image in the file, the frame buffer is still large enough to hold the whole image. In order to save memory, a smaller frame buffer could have been allocated, just big enough to hold the contents of the data window. Assuming that the pixels were still stored in contiguous scan lines, with the `pixels` pointer pointing to the pixel at the upper left corner of the data window, at coordinates `(dataWindow.min.x, dataWindow.min.y)`, the arguments to the `setFrameBuffer()` call would have to be to be changed as follows:

```
int dwWidth = dataWindow.max.x - dataWindow.min.x + 1;

file.setFrameBuffer
    (pixels - dataWindow.min.x - dataWindow.min.y * dwWidth, 1, dwWidth);
```

With these settings, evaluation of

```
base + x * xStride + y * yStride
```

for pixel `(dataWindow.min.x, dataWindow.min.y)` produces

```
  pixels - dataWindow.min.x - dataWindow.min.y * dwWidth
    + dataWindow.min.x * 1
    + dataWindow.min.y * dwWidth

= pixels -
    - dataWindow.min.x
    - dataWindow.min.y * (dataWindow.max.x - dataWindow.min.x + 1)
    + dataWindow.min.x
    + dataWindow.min.y * (dataWindow.max.x - dataWindow.min.x + 1)

= pixels,
```

which is exactly what we want. Similarly, calculating the addresses for pixels `(dataWindow.min.x+1, dataWindow.min.y)` and `(dataWindow.min.x, dataWindow.min.y+1)` yields `pixels+1` and `pixels+dwWidth`, respectively.

2.3 Storing Custom Attributes

We will now to store an image in a file, and we will add two extra attributes to the image file header: a string, called "comments", and a 4×4 matrix, called "cameraTransform".

```
void
writeRgba3 (const char fileName[],
            const Rgba *pixels,
            int width,
            int height,
            const char comments[],
            const M44f &cameraTransform)
{
    Header header (width, height);
    header.insert ("comments", StringAttribute (comments));
    header.insert ("cameraTransform", M44fAttribute (cameraTransform));

    RgbaOutputFile file (fileName, header, WRITE_RGBA);
    file.setFrameBuffer (pixels, 1, width);
    file.writePixels (height);
}
```

The `setFrameBuffer()` and `writePixels()` calls are the same as in the previous examples, but construction of the `RgbaOutputFile` object is different. The constructors in the previous examples automatically created a header on the fly, and immediately stored it in the file. Here we explicitly create a header and add our own attributes to it. When we create the `RgbaOutputFile` object, we tell the constructor to use our header instead of creating its own.

In order to make it easier to exchange data between programs written by different people, the IlmImf library defines a set of standard attributes for commonly used data, such as colorimetric information, time and place where an image was recorded, or the owner of an image file's content. For the current list of standard attributes, see the header file `ImfStandardAttributes.h`. The list is expected to grow over time as OpenEXR users identify new types of data they would like to represent in a standard format. If you need to store some piece of information in an OpenEXR file header, it is probably a good idea to check if a suitable standard attribute exists, before you define a new attribute.

2.4 Reading an RGBA Image File

Reading an RGBA image is almost as easy as writing one:

```
void
readRgba1 (const char fileName[],
           Array2D<Rgba> &pixels,
           int &width,
           int &height)
{
    RgbaInputFile file (fileName);
    Box2i dw = file.dataWindow();

    width  = dw.max.x - dw.min.x + 1;
    height = dw.max.y - dw.min.y + 1;
```

```
    pixels.resizeErase (height, width);

    file.setFrameBuffer (&pixels[0][0] - dw.min.x - dw.min.y * width, 1, width);
    file.readPixels (dw.min.y, dw.max.y);
}
```

Constructing an `RgbaInputFile` object, passing the name of the file to the constructor, opens the file and reads the file's header.

After asking the `RgbaInputFile` object for the file's data window, we allocate a buffer for the pixels. For convenience, we use the IlmImf library's `Array2D` class template (the call to `resizeErase()` does the actual allocation). The number of scan lines in the buffer is equal to the height of the data window, and the number of pixels per scan line is equal to the width of the data window. The pixels are represented as `Rgba` structs.

Note that we ignore the display window in this example; in a program that wanted to place the pixels in the data window correctly in an overall image, the display window would have to be taken into account.

Just as for writing a file, calling `setFrameBuffer()` tells the `RgbaInputFile` object how to access individual pixels in the buffer (see also section 2.2, Writing a Cropped Image, on page 4).

Calling `readPixels()` copies the pixel data from the file into the buffer. If one or more of the R, G, B, and A channels are missing in the file, the corresponding field in the pixels is filled with an appropriate default value. The default value for R, G and B is 0.0, or black; the default value for A is 1.0, or opaque.

Finally, returning from function `readRgba1()` destroys the local `RgbaInputFile` object, thereby closing the file.

Unlike the `RgbaOutputFile`'s `writePixels()` method, `readPixels()` has two arguments. Calling `readPixels(y1,y2)` copies the pixels for all scan lines with y coordinates from `y1` to `y2` into the frame buffer. This allows access to the the scan lines in any order. The image can be read all at once, one scan line at a time, or in small blocks of a few scan lines. It is also possible to skip parts of the image.

Note that even though random access is possible, reading the scan lines in the same order as they were written, is more efficient. Random access to the file requires seek operations, which tend to be slow. Calling the RgbaInputFile's `lineOrder()` method returns the order in which the scan lines in the file were written (`INCREASING_Y` or `DECREASING_Y`). If successive calls to `readPixels()` access the scan lines in the right order, the IlmImf library reads the file as fast as possible, without seek operations.

2.5 Reading an RGBA Image File in Chunks

The following shows how to read an RGBA image in blocks of a few scan lines. This is useful for programs that want to process high-resolution images without allocating enough memory to hold the complete image. These programs typically read a few scan lines worth of pixels into a memory buffer, process the pixels, and store them in another file. The buffer is then re-used for the next set of scan lines. Image operations like color-correction or compositing ("A over B") are very easy to do incrementally this way. With clever buffering of a few extra scan lines, incremental versions of operations that require access to neighboring pixels, like blurring or sharpening, are also possible.

```
void
readRgba2 (const char fileName[])
{
    RgbaInputFile file (fileName);
    Box2i dw = file.dataWindow();

    int width  = dw.max.x - dw.min.x + 1;
    int height = dw.max.y - dw.min.y + 1;
    Array2D<Rgba> pixels (10, width);

    while (dw.min.y <= dw.max.y)
    {
        file.setFrameBuffer (&pixels[0][0] - dw.min.x - dw.min.y * width,
                             1, width);

        file.readPixels (dw.min.y, min (dw.min.y + 9, dw.max.y));
```

```
            // processPixels (pixels)

            dw.min.y += 10;
        }
    }
```

Again, we open the file and read the file header by constructing an `RgbaInputFile` object. Then we allocate a memory buffer that is just large enough to hold ten complete scan lines. We call `readPixels()` to copy the pixels from the file into our buffer, ten scan lines at a time. Since we want to re-use the buffer for every block of ten scan lines, we have to call `setFramebuffer()` before each `readPixels()` call, in order to associate memory address `&pixels[0][0]` first with pixel coordinates `(dw.min.x, dw.min.y)`, then with `(dw.min.x, dw.min.y+10)`, `(dw.min.x, dw.min.y+20)` and so on.

2.6 Reading Custom Attributes

In section 2.3, we showed how to store custom attributes in the image file header. Here we show how to test whether a given file's header contains particular attributes, and how to read those attributes' values.

```
void
readHeader (const char fileName[])
{
    RgbaInputFile file (fileName);

    const StringAttribute *comments =
        file.header().findTypedAttribute <StringAttribute> ("comments");

    const M44fAttribute *cameraTransform =
        file.header().findTypedAttribute <M44fAttribute> ("cameraTransform");

    if (comments)
        cout << "comments\n   " << comments->value() << endl;

    if (cameraTransform)
        cout << "cameraTransform\n" << cameraTransform->value() << flush;
}
```

As usual, we open the file by constructing an RgbaInputFile object. Calling `findTypedAttribute<T>(n)` searches the header for an attribute with type `T` and name `n`. If a matching attribute is found, `findTypedAttribute()` returns a pointer to the attribute. If the header contains no attribute with name `n`, or if the header contains an attribute with name `n`, but the attribute's type is not `T`, `findAttribute()` returns `0`. Once we have pointers to the attributes we were looking for, we can access their values by calling the attributes' `value()` methods.

In this example, we handle the possibility that the attributes we want may not exist by explicitly checking for `0` pointers. Sometimes it is more convenient to rely on exceptions instead. Function `typedAttribute()`, a variation of `findTypedAttribute()`, also searches the header for an attribute with a given name and type, but if the attribute in question does not exist, `typedAttribute()` throws an exception rather than returning `0`.

Note that the pointers returned by `findTypedAttribute()` point to data that are part of the `RgbaInputFile` object. The pointers become invalid as soon as the `RgbaInputFile` object is destroyed. Therefore, the following will not work:

```
void
readComments (const char fileName[], StringAttribute *&comments)
{
    // error: comments pointer is invalid after this function returns
    RgbaInputFile file (fileName);
    comments = file.header().findTypedAttribute <StringAttribute> ("comments");
}
```

`readComments()` must copy the attribute's value before it returns; for example, like this:

```
void
readComments (const char fileName[], string &comments)
{

    RgbaInputFile file (fileName);
    comments = file.header().typedAttribute<StringAttribute>("comments").value();
}
```

2.7 Luminance/Chroma and Gray-Scale Images

Writing an RGBA image file usually preserves the pixels without losing any data; saving an image file and reading it back does not alter the pixels' R, G, B and A values. Most of the time, lossless data storage is exactly what we want, but sometimes file space or transmission bandwidth are limited, and we would like to reduce the size of our image files. It is often acceptable if the numbers in the pixels change slightly as long as the image still looks just like the original.

The RGBA interface in the IlmImf library supports storing RGB data in luminance/chroma format. The R, G, and B channels are converted into a luminance channel, Y, and two chroma channels, RY and BY. The Y channel represents a pixel's brightness, and the two chroma channels represent its color. The human visual system's spatial resolution for color is much lower than the spatial resolution for brightness. This allows us to reduce the horizontal and vertical resolution of the RY and BY channels by a factor of two. The visual appearance of the image doesn't change, but the image occupies only half as much space, even before data compression is applied. (For every four pixels, we store four Y values, one RY value, and one BY value, instead of four R, four G, and four B values.)

When opening a file for writing, a program can select how it wants the pixels to be stored. The constructors for class `RgbaOutputFile` have an `rgbaChannels` argument, which determines the set of channels in the file:

`WRITE_RGB`	red, green, blue
`WRITE_RGBA`	red, green, blue, alpha
`WRITE_YC`	luminance, chroma
`WRITE_YCA`	luminance, chroma, alpha
`WRITE_Y`	luminance only
`WRITE_YA`	luminance, alpha

`WRITE_Y` and `WRITE_YA` provide an efficient way to store gray-scale images. The chroma channels for a gray-scale image contain only zeroes, so they can be omitted from the file.

When an image file is opened for reading, class `RgbaInputFile` automatically detects luminance/chroma images and converts the pixels back to RGB format.

3 Using the General Interface for Scan-line-based Files

3.1 Writing an Image File

This example demonstrates how to write an OpenEXR image file with two channels: one channel, of type `HALF`, is called G, and the other, of type `FLOAT`, is called Z. The size of the image is `width` by `height` pixels. The data for the two channels are supplied in two separate buffers, `gPixels` and `zPixels`. Within each buffer, the pixels of each scan line are contiguous in memory.

```
    void
    writeGZ1 (const char fileName[],
              const half *gPixels,
              const float *zPixels,
              int width,
              int height)
    {
        Header header (width, height);                                  // 1
        header.channels().insert ("G", Channel (HALF));                 // 2
        header.channels().insert ("Z", Channel (FLOAT));                // 3

        OutputFile file (fileName, header);                             // 4

        FrameBuffer frameBuffer;                                        // 5

        frameBuffer.insert ("G",                                // name  // 6
                            Slice (HALF,                        // type  // 7
                                   (char *) gPixels,            // base  // 8
                                   sizeof (*gPixels) * 1,       // xStride// 9
                                   sizeof (*gPixels) * width)); // yStride// 10

        frameBuffer.insert ("Z",                                // name  // 11
                            Slice (FLOAT,                       // type  // 12
                                   (char *) zPixels,            // base  // 13
                                   sizeof (*zPixels) * 1,       // xStride// 14
                                   sizeof (*zPixels) * width)); // yStride// 15

        file.setFrameBuffer (frameBuffer);                              // 16
        file.writePixels (height);                                      // 17
    }
```

In line 1, an OpenEXR header is created, and the header's display window and data window are both set to `(0, 0) - (width-1, height-1)`.

Lines 2 and 3 specify the names and types of the image channels that will be stored in the file.

Constructing an `OutputFile` object in line 4 opens the file with the specified name, and stores the header in the file.

Lines 5 through 16 tell the `OutputFile` object how the pixel data for the image channels are laid out in memory. After constructing a `FrameBuffer` object, a `Slice` is added for each of the image file's channels. A `Slice` describes the memory layout of one channel. The constructor for the `Slice` object takes four arguments, `type`, `base`, `xStride`, and `yStride`. `type` specifies the pixel data type (`HALF`, `FLOAT`, or `UINT`); the other three arguments define the memory address of pixel `(x,y)` as

```
base + x * xStride + y * yStride.
```

Note that `base` is of type `char*`, and that offsets from `base` are not implicitly multiplied by the size of an individual pixel, as in the RGBA-only interface. `xStride` and `yStride` must explictly take the size of the pixels into account.

With the values specified in our example, the IlmImf library computes the address of the G channel of pixel `(x,y)` like this:

```
    (half*)((char*)gPixels + x * sizeof(half) * 1 + y * sizeof(half) * width)
  = (half*)((char*)gPixels + x * 2 + y * 2 * width),
```

The address of the Z channel of pixel `(x,y)` is

```
    (float*)((char*)zPixels + x * sizeof(float) * 1 + y * sizeof(float) * width)
  = (float*)((char*)zPixels + x * 4 + y * 4 * width).
```

The `writePixels()` call in line 9 copies the image's pixels from memory into the file. As in the RGBA-only interface, the argument to `writePixels()` specifies how many scan lines are copied into the file (see section 2.1, Writing an RGBA Image File, on page 3).

If the image file contains a channel for which the `FrameBuffer` object has no corresponding `Slice`, then the pixels for that channel in the file are filled with zeroes. If the `FrameBuffer` object contains a `Slice` for which the file has no channel, then the `Slice` is ignored.

Returning from function `writeGZ1()` destroys the local `OutputFile` object and closes the file.

3.2 Writing a Cropped Image

Writing a cropped image using the general interface is analogous to writing a cropped image using the RGBA-only interface, as shown in section 2.2, on page 4: In the file's header the data window is set explicitly instead of being generated automatically from the image's width and height. The number of scan lines that are stored in the file is equal to the height of the data window, instead of the height of the entire image. As in section 2.2, the example code below assumes that the memory buffers for the pixels are large enough to hold `width` by `height` pixels, but only the region that corresponds to the data window will be stored in the file. For smaller memory buffers with room only for the pixels in the data window, the `base`, `xStride` and `yStride` arguments for the `FrameBuffer` object's slices would have to be adjusted accordingly (again, see section 2.2).

```
void
writeGZ2 (const char fileName[],
          const half *gPixels,
          const float *zPixels,
          int width,
          int height,
          const Box2i &dataWindow)
{
    Header header (width, height);
    header.dataWindow() = dataWindow;
    header.channels().insert ("G", Channel (HALF));
    header.channels().insert ("Z", Channel (FLOAT));

    OutputFile file (fileName, header);

    FrameBuffer frameBuffer;

    frameBuffer.insert ("G",                                // name
                        Slice (HALF,                        // type
                               (char *) gPixels,            // base
                               sizeof (*gPixels) * 1,       // xStride
                               sizeof (*gPixels) * width)); // yStride

    frameBuffer.insert ("Z",                                // name
                        Slice (FLOAT,                       // type
                               (char *) zPixels,            // base
                               sizeof (*zPixels) * 1,       // xStride
                               sizeof (*zPixels) * width)); // yStride

    file.setFrameBuffer (frameBuffer);
    file.writePixels (dataWindow.max.y - dataWindow.min.y + 1);
}
```

3.3 Reading an Image File

In this example, we read an OpenEXR image file using the IlmImf library's general interface. We assume that the file contains two channels, R, and G, of type `HALF`, and one channel, Z, of type `FLOAT`. If one of

those channels is not present in the image file, the corresponding memory buffer for the pixels will be filled with an appropriate default value.

```
void
readGZ1 (const char fileName[],
         Array2D<half> &rPixels,
         Array2D<half> &gPixels,
         Array2D<float> &zPixels,
         int &width, int &height)
{
    InputFile file (fileName);

    Box2i dw = file.header().dataWindow();
    width  = dw.max.x - dw.min.x + 1;
    height = dw.max.y - dw.min.y + 1;

    rPixels.resizeErase (height, width);
    gPixels.resizeErase (height, width);
    zPixels.resizeErase (height, width);

    FrameBuffer frameBuffer;

    frameBuffer.insert ("R",                                    // name
                        Slice (HALF,                            // type
                               (char *) (&rPixels[0][0] -       // base
                                         dw.min.x -
                                         dw.min.y * width),
                               sizeof (rPixels[0][0]) * 1,      // xStride
                               sizeof (rPixels[0][0]) * width,  // yStride
                               1, 1,                            // x/y sampling
                               0.0));                           // fillValue

    frameBuffer.insert ("G",                                    // name
                        Slice (HALF,                            // type
                               (char *) (&gPixels[0][0] -       // base
                                         dw.min.x -
                                         dw.min.y * width),
                               sizeof (gPixels[0][0]) * 1,      // xStride
                               sizeof (gPixels[0][0]) * width,  // yStride
                               1, 1,                            // x/y sampling
                               0.0));                           // fillValue

    frameBuffer.insert ("Z",                                    // name
                        Slice (FLOAT,                           // type
                               (char *) (&zPixels[0][0] -       // base
                                         dw.min.x -
                                         dw.min.y * width),
                               sizeof (zPixels[0][0]) * 1,      // xStride
                               sizeof (zPixels[0][0]) * width,  // yStride
                               1, 1,                            // x/y sampling
                               FLT_MAX));                       // fillValue

    file.setFrameBuffer (frameBuffer);
    file.readPixels (dw.min.y, dw.max.y);
}
```

First, we open the file with the specified name, by constructing an `InputFile` object.

Using the `Array2D` class template, we allocate memory buffers for the image's R, G and Z channels. The buffers are big enough to hold all pixels in the file's data window.

Next, we create a `FrameBuffer` object, which describes our buffers to the IlmImf library. For each image channel, we add a slice to the `FrameBuffer`.

As usual, the slice's `type`, `xStride`, and `yStride` describe the corresponding buffer's layout. For the R channel, pixel `(dw.min.x, dw.min.y)` is at address `&rPixels[0][0]`. By setting the `type`, `xStride` and `yStride` of the corresponding `Slice` object as shown above, evaluating

```
base + x * xStride + y * yStride
```

for pixel `(dw.min.x, dw.min.y)` produces

```
    (char*)(&rPixels[0][0] - dw.min.x - dw.min.y * width)
     + dw.min.x * sizeof (rPixels[0][0]) * 1
     + dw.min.y * sizeof (rPixels[0][0]) * width

  = (char*)&rPixels[0][0]
     - dw.min.x * sizeof (rPixels[0][0])
     - dw.min.y * sizeof (rPixels[0][0]) * width
     + dw.min.x * sizeof (rPixels[0][0])
     + dw.min.y * sizeof (rPixels[0][0]) * width

  = &rPixels[0][0].
```

The address calculations for pixels `(dw.min.x+1, dw.min.y)` and `(dw.min.x, dw.min.y+1)` produce `&rPixels[0][0]+1` and `&rPixels[0][0]+width`, which is equivalent to `&rPixels[0][1]` and `&rPixels[1][0]`.

Each `Slice` has a `fillValue`. If the image file does not contain an image channel for the `Slice`, then the corresponding memory buffer will be filled with the `fillValue`.

The `Slice's` remaining two parameters, `xSampling` and `ySampling` are used for images where some of the channels are subsampled, for instance, the RY and BY channels in luminance/chroma images. (see section 2.7, Luminance/Chroma and Gray-scale Images, on page 8). Unless an image contains subsampled channels, `xSampling` and `ySampling` should always be set to 1. For details see header files `ImfFrameBuffer.h` and `ImfChannelList.h`.

After describing our memory buffers' layout, we call `readPixels()` to copy the pixel data from the file into the buffers. Just as with the RGBA-only interface, `readPixels()` allows random-access to the scan lines in the file (see section 2.5 Reading an RGBA Image File, on page 6).

3.4 Interleaving Image Channels in the Frame Buffer

Here is a variation of the previous example. We are reading an image file, but instead of storing each image channel in a separate memory buffer, we interleave the channels in a single buffer. The buffer is an array of structs, which are defined like this:

```
typedef struct GZ
{
    half  g;
    float z;
};
```

The code to read the file is almost the same as before; aside from reading only two instead of three channels, the only difference is how `base`, `xStride` and `yStride` for the `Slices` in the `FrameBuffer` object are computed:

```
void
readGZ2 (const char fileName[],
         Array2D<GZ> &pixels,
         int &width, int &height)
{
    InputFile file (fileName);

    Box2i dw = file.header().dataWindow();
    width  = dw.max.x - dw.min.x + 1;
    height = dw.max.y - dw.min.y + 1;
    int dx = dw.min.x;
    int dy = dw.min.y;

    pixels.resizeErase (height, width);

    FrameBuffer frameBuffer;

    frameBuffer.insert ("G",
                        Slice (HALF,
                               (char *) &pixels[-dy][-dx].g,
                                sizeof (pixels[0][0]) * 1,
                                sizeof (pixels[0][0]) * width));
```

```
    frameBuffer.insert ("Z",
                        Slice (FLOAT,
                               (char *) &pixels[-dy][-dx].z,
                                sizeof (pixels[0][0]) * 1,
                                sizeof (pixels[0][0]) * width));

    file.setFrameBuffer (frameBuffer);
    file.readPixels (dw.min.y, dw.max.y);
}
```

3.5 Which Channels are in a File?

In functions `readGZ1()` and `readGZ2()`, above, we simply assumed that the files we were trying to read contained a certain set of channels. We relied on the IlmImf library to do "something reasonable" in case our assumption was not true. Sometimes we want to know exactly what channels are in an image file before reading any pixels, so that we can do what we think is appropriate.

The file's header contains the file's channel list. Using iterators similar to those in the C++ Standard Template Library, we can iterate over the channels:

```
const ChannelList &channels = file.header().channels();

for (ChannelList::ConstIterator i = channels.begin(); i != channels.end(); ++i)
{
    const Channel &channel = i->second;
    // ...
}
```

Channels can also be accessed by name, either with the `[]` operator, or with the `findChannel()` function:

```
const ChannelList &channels = file.header().channels();
const Channel &channel = channelList["G"];
const Channel *channelPtr = channelList.findChannel("G");
```

The difference between the `[]` operator and `findChannel()` function is how errors are handled: If the channel in question is not present, `findChannel()` returns `0`; the `[]` operator throws an exception.

4 Tiles, Levels and Level Modes

A single tiled OpenEXR file can hold multiple versions of an image, each with a different resolution. Each version is called a *level*. A tiled file's *level mode* defines how many levels are stored in the file. There are three different level modes:

`ONE_LEVEL`	The file contains only a single, full-resolution level. A `ONE_LEVEL` image file is equivalent to a scan-line-based file; the only difference is that the pixels are accessed by tile instead of by scan line.
`MIPMAP_LEVELS`	The file contains multiple levels. The first level holds the image at full resolution. Each successive level is half the resolution of the previous level in x and y direction. The last level contains only a single pixel. `MIPMAP_LEVELS` files are used for texture-mapping and similar applications.
`RIPMAP_LEVELS`	Like `MIPMAP_LEVELS`, but with more levels. The levels include all combinations of reducing the resolution of the image by powers of two independently in x and y direction. Used for texture mapping, like `MIPMAP_LEVELS`. The additional levels in a `RIPMAP_LEVELS` file can help to accelerate anisotropic filtering during texture lookups.

In `MIPMAP_LEVELS` and `RIPMAP_LEVELS` mode, the size (width or height) of each level is computed by halving the size of the level with the next higher resolution. If the size of the higher-resolution level is odd, then the size of the lower-resolution level must be rounded up or down in order to avoid arriving at a non-integer width or height. The rounding direction is determined by the file's *level size rounding mode*.

Within each level, the pixels of the image are stored in a two-dimensional array of tiles. The tiles in an OpenEXR file can be any rectangular shape, but all tiles in a file have the same size. This means that lower-resolution levels contain fewer, rather than smaller, tiles.

An OpenEXR file's level mode and rounding mode, and the size of the tiles are stored in an attribute in the file header. The value of this attribute is a `TileDescription` object:

```
enum LevelMode
{
    ONE_LEVEL,
    MIPMAP_LEVELS,
    RIPMAP_LEVELS
};

enum LevelRoundingMode
{
    ROUND_DOWN,
    ROUND_UP
};

class TileDescription
{
  public:

    unsigned int      xSize;        // size of a tile in the x dimension
    unsigned int      ySize;        // size of a tile in the y dimension
    LevelMode         mode;
    LevelRoundingMode roundingMode;

    ...                             // (methods omitted)
};
```

5 Using the RGBA-only Interface for Tiled Files

5.1 Writing a Tiled RGBA Image File with One Resolution Level

Writing a tiled RGBA image with a single level is easy:

```
void
writeTiledRgbaONE1 (const char fileName[],
                    const Rgba *pixels,
                    int width, int height,
                    int tileWidth, int tileHeight)
{
    TiledRgbaOutputFile out (fileName,
                             width, height,         // image size
                             tileWidth, tileHeight, // tile size
                             ONE_LEVEL,             // level mode
                             ROUND_DOWN,            // rounding mode
                             WRITE_RGBA);           // channels in file // 1

    out.setFrameBuffer (pixels, 1, width);                              // 2

    for (int tileY = 0; tileY < out.numYTiles (); ++tileY)              // 3
        for (int tileX = 0; tileX < out.numXTiles (); ++tileX)          // 4
            out.writeTile (tileX, tileY);                               // 5
}
```

Opening the file and defining the pixel data layout in memory are done in almost the same way as for scan-line-based files:

Construction of the `TiledRgbaOutputFile` object, in line 1, creates an OpenEXR header, sets the header's attributes, opens the file with the specified name, and stores the header in the file. The header's display window and data window are both set to `(0, 0) - (width-1, height-1)`. The size of each tile in the file will be `tileWidth` by `tileHeight` pixels. The channel list contains four channels, R, G, B, and A, of type `HALF`.

Line 2 specifies how the pixel data are laid out in memory. The arithmetic involved in calculating the memory address of a specific pixel is the same as for the scan-line-based interface (see section 2.1). We assume that the `pixels` pointer points to an array of `width*height` pixels, which contains the entire image.

Lines 3 and 4 loop over all tiles within the image. The `TiledRgbaOutputFile`'s `numXTiles()` method returns the number of tiles in the x direction, and similarly, the `numYTiles()` method returns the number of tiles in the y dimension. During these loops, line 5 writes out each tile in the image.

This simple method works well when enough memory is available to allocate a frame buffer for the entire image. When allocating a frame buffer for the whole image is not desirable, for example because the image is very large, a smaller frame buffer can be used. Even a frame buffer that can hold only a single tile is sufficient, as demonstrated in the following example:

```
void
writeTiledRgbaONE2 (const char fileName[],
                    int width, int height,
                    int tileWidth, int tileHeight)
{
    TiledRgbaOutputFile out (fileName,
                             width, height,         // image size
                             tileWidth, tileHeight, // tile size
                             ONE_LEVEL,             // level mode
                             ROUND_DOWN,            // rounding mode
                             WRITE_RGBA);           // channels in file // 1

    Array2D<Rgba> pixels (tileHeight, tileWidth);                       // 2

    for (int tileY = 0; tileY < out.numYTiles (); ++tileY)              // 3
    {
        for (int tileX = 0; tileX < out.numXTiles (); ++tileX)          // 4
        {
            Box2i range = out.dataWindowForTile (tileX, tileY);         // 5
```

```
            generatePixels (pixels, width, height, range);              // 6

            out.setFrameBuffer (&pixels[-range.min.y][-range.min.x],
                                1,          // xStride
                                tileWidth); // yStride                  // 7

            out.writeTile (tileX, tileY);                               // 8
        }
    }
}
```

In line 2 we allocate a `pixels` array with `tileWidth*tileHeight` elements, which is just enough for one tile. Line 5 computes the data window range for each tile, that is, the set of pixel coordinates covered by the tile. The `generatePixels()` function, in line 6, fills the `pixels` array with one tile's worth of image data. The same `pixels` array is reused for all tiles. We must call `setFrameBuffer()`, in line 7, before writing each tile so that the pixels in the array are accessed properly in the `writeTile()` call in line 8. Again, the address arithmetic to access the pixels is the same as for scan-line-based files. The values for the `base`, `xStride`, and `yStride` arguments to the `setFrameBuffer()` call must be chosen so that evaluating the expression

```
base + x * xStride + y * yStride
```

produces the address of the pixel with coordinates `(x,y)`.

5.2 Writing a Tiled RGBA Image File with Mipmap Levels

In order to store a multiresolution image in a file, we can allocate a frame buffer large enough for the highest-resolution level, `(0,0)`, and reuse it for all levels:

```
void
writeTiledRgbaMIP1 (const char fileName[],
                    int width, int height,
                    int tileWidth, int tileHeight)
{
    TiledRgbaOutputFile out (fileName,
                             width, height,
                             tileWidth, tileHeight,
                             MIPMAP_LEVELS,
                             ROUND_DOWN,
                             WRITE_RGBA);                               // 1

    Array2D<Rgba> pixels (height, width);                               // 2
    out.setFrameBuffer (&pixels[0][0], 1, width);                       // 3

    for (int level = 0; level < out.numLevels (); ++level)             // 4
    {
        generatePixels (pixels, width, height, level);                  // 5

        for (int tileY = 0; tileY < out.numYTiles (level); ++tileY)     // 6
            for (int tileX = 0; tileX < out.numXTiles (level); ++tileX) // 7
                out.writeTile (tileX, tileY, level);                    // 8
    }
}
```

The main difference here is the use of `MIPMAP_LEVELS` in line 1 for the `TiledRgbaOutputFile` constructor. This signifies that the file will contain multiple levels, each level being a factor of 2 smaller in both dimensions than the previous level. Mipmap images contain `n` levels, with level numbers

```
(0,0), (1,1), ... (n-1,n-1),
```

where

```
n = floor (log (max (width, height)) / log (2)) + 1
```

if the level size rounding mode is `ROUND_DOWN`, or

```
n = ceil (log (max (width, height)) / log (2)) + 1
```

if the level size rounding mode is `ROUND_UP`. Note that even though level numbers are pairs of integers, `(lx,ly)`, only levels where `lx` equals `ly` are used in `MIPMAP_LEVELS` files.

Line 2 allocates a `pixels` array with `width` by `height` pixels, big enough to hold the highest-resolution level.

In addition to looping over all tiles (lines 6 and 7), we must loop over all levels in the image (line 4). `numLevels()` returns the number of levels, `n`, in our mipmapped image. Since the tile sizes remain the same in all levels, the number of tiles in both dimensions varies between levels. `numXTiles()` and `numYTiles()` take a level number as an optional argument, and return the number of tiles in the x or y direction for the corresponding level. Line 5 fills the `pixels` array with appropriate data for each level.

As with `ONE_LEVEL` images, we can choose to only allocate a frame buffer for a single tile and reuse it for all tiles in the image:

```
void
writeTiledRgbaMIP2 (const char fileName[],
                    int width, int height,
                    int tileWidth, int tileHeight)
{
    TiledRgbaOutputFile out (fileName,
                             width, height,
                             tileWidth, tileHeight,
                             MIPMAP_LEVELS,
                             ROUND_DOWN,
                             WRITE_RGBA);

    Array2D<Rgba> pixels (tileHeight, tileWidth);

    for (int level = 0; level < out.numLevels (); ++level)
    {
        for (int tileY = 0; tileY < out.numYTiles (level); ++tileY)
        {
            for (int tileX = 0; tileX < out.numXTiles (level); ++tileX)
            {
                Box2i range = out.dataWindowForTile (tileX, tileY, level);

                generatePixels (pixels, width, height, range, level);

                out.setFrameBuffer (&pixels[-range.min.y][-range.min.x],
                                    1,              // xStride
                                    tileWidth);     // yStride

                out.writeTile (tileX, tileY, level);
            }
        }
    }
}
```

The structure of this code is the same as for writing a `ONE_LEVEL` image using a tile-sized frame buffer, but we have to loop over more tiles. Also, `dataWindowForTile()` takes an additional level argument to determine the pixel range for the tile at the specified level.

5.3 Writing a Tiled RGBA Image File with Ripmap Levels

The ripmap level mode allows for storing all combinations of reducing the resolution of the image by powers of two independently in both dimensions. Ripmap files contains `nx*ny` levels, with level numbers:

```
(0, 0),   (1, 0),   ... (nx-1, 0),
(0, 1),   (1, 1),   ... (nx-1, 1),
 ...
(0,ny-1), (1,ny-1), ... (nx-1,ny-1)
```

where

```
nx = floor (log (width) / log (2)) + 1
ny = floor (log (height) / log (2)) + 1
```

if the level size rounding mode is `ROUND_DOWN`, or

```
nx = ceil (log (width) / log (2)) + 1
ny = ceil (log (height) / log (2)) + 1
```

if the level size rounding mode is `ROUND_UP`.

With a frame buffer that is large enough to hold level `(0,0)`, we can write a ripmap file like this:

```
void
writeTiledRgbaRIP1 (const char fileName[],
                    int width, int height,
                    int tileWidth, int tileHeight)
{
    TiledRgbaOutputFile out (fileName,
                             width, height,
                             tileWidth, tileHeight,
                             RIPMAP_LEVELS,
                             ROUND_DOWN,
                             WRITE_RGBA);

    Array2D<Rgba> pixels (height, width);
    out.setFrameBuffer (&pixels[0][0], 1, width);

    for (int yLevel = 0; yLevel < out.numYLevels (); ++yLevel)
    {
        for (int xLevel = 0; xLevel < out.numXLevels (); ++xLevel)
        {
            generatePixels (pixels, width, height, xLevel, yLevel);

            for (int tileY = 0; tileY < out.numYTiles (yLevel); ++tileY)
                for (int tileX = 0; tileX < out.numXTiles (xLevel); ++tileX)
                    out.writeTile (tileX, tileY, xLevel, yLevel);
        }
    }
}
```

As for `ONE_LEVEL` and `MIPMAP_LEVELS` files, the frame buffer doesn't have to be large enough to hold a whole level. Any frame buffer big enough to hold at least a single tile will work.

5.4 Reading a Tiled RGBA Image File

Reading a tiled RGBA image file is done similarly to writing one:

```
void
readTiledRgba1 (const char fileName[],
                Array2D<Rgba> &pixels,
                int &width,
                int &height)
{
    TiledRgbaInputFile in (fileName);
    Box2i dw = in.dataWindow();

    width  = dw.max.x - dw.min.x + 1;
    height = dw.max.y - dw.min.y + 1;
    int dx = dw.min.x;
    int dy = dw.min.y;

    pixels.resizeErase (height, width);

    in.setFrameBuffer (&pixels[-dy][-dx], 1, width);

    for (int tileY = 0; tileY < in.numYTiles(); ++tileY)
        for (int tileX = 0; tileX < in.numXTiles(); ++tileX)
            in.readTile (tileX, tileY);
}
```

First we need to create a `TiledRgbaInputFile` object for the given file name. We then retrieve information about the data window in order to create an appropriately sized frame buffer, in this case large enough to hold the whole image at level `(0,0)`. After we set the frame buffer, we iterate over the tiles we are interested in, and read them from the file.

This example only reads the highest-resolution level of the image. It can be extended to read all levels, for multiresolution images, by also iterating over all levels within the image, analogous to the examples in sections section 5.2 and 5.3.

6 Using the General Interface for Tiled Files

6.1 Writing a Tiled Image File

This example is a variation of the one in section 3.1, on page 9. We are writing a `ONE_LEVEL` image file with two channels, G, and Z, of type `HALF`, and `FLOAT` respectively, but here the file is tiled instead of scan-line-based:

```
    void
    writeTiled1 (const char fileName[],
                 Array2D<GZ> &pixels,
                 int width, int height,
                 int tileWidth, int tileHeight)
    {
        Header header (width, height);                                          // 1
        header.channels().insert ("G", Channel (HALF));                         // 2
        header.channels().insert ("Z", Channel (FLOAT));                        // 3

        header.setTileDescription
            (TileDescription (tileWidth, tileHeight, ONE_LEVEL));               // 4

        TiledOutputFile out (fileName, header);                                 // 5

        FrameBuffer frameBuffer;                                                // 6

        frameBuffer.insert ("G",                                    // name     // 7
                            Slice (HALF,                            // type     // 8
                                   (char *) &pixels[0][0].g,        // base     // 9
                                    sizeof (pixels[0][0]) * 1,      // xStride // 10
                                    sizeof (pixels[0][0]) * width)); // yStride // 11

        frameBuffer.insert ("Z",                                    // name     // 12
                            Slice (FLOAT,                           // type     // 13
                                   (char *) &pixels[0][0].z,        // base     // 14
                                    sizeof (pixels[0][0]) * 1,      // xStride // 15
                                    sizeof (pixels[0][0]) * width)); // yStride // 16

        out.setFrameBuffer (frameBuffer);                                       // 17

        for (int tileY = 0; tileY < out.numYTiles (); ++tileY)                  // 18
            for (int tileX = 0; tileX < out.numXTiles (); ++tileX)              // 19
                out.writeTile (tileX, tileY);                                   // 20
    }
```

As one would expect, the code here is very similar to the code in section 3.1. The file's header is created in line 1, while lines 2 and 3 specify the names and types of the image channels that will be stored in the file. An important addition is line 4, where we define the size of the tiles and the level mode. In this example we use `ONE_LEVEL` for simplicity. Line 5 opens the file and writes the header. Lines 6 through 17 tell the `TiledOutputFile` object the location and layout of the pixel data for each channel. Finally, lines 18 through 20 loop over all tiles in the image and write out each tile.

6.2 Reading a Tiled Image File

Reading a tiled file with the general interface is virtually identical to reading a scan-line-based file, as shown in section 3.4, on page 12; only the last three lines are different. Instead of reading all scan lines at once with a single function call, here we must iterate over all tiles we want to read.

```
    void
    readTiled1 (const char fileName[],
                Array2D<GZ> &pixels,
                int &width, int &height)
    {
        TiledInputFile in (fileName);

        Box2i dw = in.header().dataWindow();
        width  = dw.max.x - dw.min.x + 1;
        height = dw.max.y - dw.min.y + 1;
        int dx = dw.min.x;
```

```
    int dy = dw.min.y;

    pixels.resizeErase (height, width);

    FrameBuffer frameBuffer;

    frameBuffer.insert ("G",
                        Slice (HALF,
                               (char *) &pixels[-dy][-dx].g,
                                sizeof (pixels[0][0]) * 1,
                                sizeof (pixels[0][0]) * width));

    frameBuffer.insert ("Z",
                        Slice (FLOAT,
                               (char *) &pixels[-dy][-dx].z,
                                sizeof (pixels[0][0]) * 1,
                                sizeof (pixels[0][0]) * width));

    in.setFrameBuffer (frameBuffer);

    for (int tileY = 0; tileY < in.numYTiles(); ++tileY)
        for (int tileX = 0; tileX < in.numXTiles(); ++tileX)
            in.readTile (tileX, tileY);
}
```

In this example we assume that the file we want to read contains two channels, G and Z, of type `HALF` and `FLOAT` respectively. If the file contains other channels, we ignore them. We only read the highest-resolution level of the image. If the input file contains more levels (`MIPMAP_LEVELS` or `MIPMAP_LEVELS`), we can access the extra levels by calling a four-argument version of the `readTile()` function:

```
in.readTile (tileX, tileY, levelX, levelY);
```

7 Miscellaneous

7.1 Is this an OpenEXR File?

Sometimes we want to test quickly if a given file is an OpenEXR file. This can be done by looking at the beginning of the file: The first four bytes of every OpenEXR file contain the 32-bit integer "magic number" 20000630 in little-endian byte order. After reading a file's first four bytes via any of the operating system's standard file I/O mechanisms, we can compare them with the magic number, either by calling function `isImfMagic()` or by explicitly testing if the bytes contain the values `0x76`, `0x2f`, `0x31`, and `0x01`.

Given a file name, the following function returns true if the corresponding file exists, is readable, and contains an OpenEXR image:

```
bool
isOpenExrFile (const char fileName[])
{
    std::ifstream f (fileName, std::ios_base::binary);

    char bytes[4];
    f.read (bytes, sizeof (bytes));

    return !!f && Imf::isImfMagic (bytes);
}
```

7.2 Custom Low-Level File I/O

In all of the previous file reading and writing examples, we were given a file name, and we relied on the constructors for our input file or output file objects to open the file. In some contexts, for example in a plugin for an existing application program, we may have to read from or write to a file that has already been opened. The representation of the open file as a C or C++ data type depends on the application program and on the operating system.

At its lowest level, the IlmImf library performs file I/O via objects of type `IStream` and `OStream`. `IStream` and `OStream` are abstract base classes. The IlmImf library contains two derived classes, `StdIFStream` and `StdOFStream`, that implement reading from `std::ifstream` and writing to `std::ofstream` objects. An application program can implement alternative file I/O mechanisms by deriving its own classes from `Istream` and `Ostream`. This way, OpenEXR images can be stored in arbitrary file-like objects, as long as it is possible to support read, write, seek and tell operations with semantics similar to the corresponding `std::ifstream` and `std::ofstream` methods.

For example, assume that we want to read an OpenEXR image from a C stdio file (of type `FILE *`) that has already been opened. To do this, we derive a new class, `C_IStream`, from `IStream`. The declaration of class `IStream` looks like this:

```
class IStream
{
  public:

    virtual ~IStream ();

    virtual bool    read (char c[], int n) = 0;
    virtual Int64   tellg () = 0;
    virtual void    seekg (Int64 pos) = 0;
    virtual void    clear ();
    const char *    fileName () const;

  protected:

    IStream (const char fileName[]);

  private:

    ...
```

```
};
```

Our derived class needs a public constructor, and it must override four methods:

```
class C_IStream: public IStream
{
  public:

    C_IStream (FILE *file, const char fileName[]):
        IStream (fileName), _file (file) {}

    virtual bool    read (char c[], int n);
    virtual Int64   tellg ();
    virtual void    seekg (Int64 pos);
    virtual void    clear ();

  private:

    FILE *          _file;
};
```

`read(c,n)` reads `n` bytes from the file, and stores them in array `c`. If reading hits the end of the file before `n` bytes have been read, or if an I/O error occurs, `read(c,n)` throws an exception. If `read(c,n)` hits the end of the file after reading `n` bytes, it returns `false`, otherwise it returns `true`:

```
bool
C_IStream::read (char c[], int n)
{
    if (n != fread (c, 1, n, _file))
    {
        // fread() failed, but the return value does not distinguish
        // between I/O errors and end of file, so we call ferror() to
        // determine what happened.

        if (ferror (_file))
            Iex::throwErrnoExc();
        else
            throw Iex::InputExc ("Unexpected end of file.");
    }

    return feof (_file);
}
```

`tellg()` returns the current reading position, in bytes, from the beginning of the file. The next call to `read()` will begin reading at the indicated position:

```
Int64
C_IStream::tellg ()
{
    return ftell (_file);
}
```

`seekg(pos)` sets the current reading position to `pos` bytes from the beginning of the file:

```
void
C_IStream::seekg (Int64 pos)
{
    clearerr (_file);
    fseek (_file, pos, SEEK_SET);
}
```

`clear()` clears any error flags that may be set on the file after a `read()` or `seekg()` operation has failed:

```
void
C_IStream::clear ()
{
    clearerr (_file);
}
```

In order to read an RGBA image from an open C stdio file, we first make a `C_IStream` object. Then we create an `RgbaInputFile`, passing the `C_IStream` instead of a file name to the constructor. After that, we read the image as usual (see section 2.4, Reading an RGBA Image File, on page 5):

```
void
readRgbaFILE (FILE *cfile,
              const char fileName[],
              Array2D<Rgba> &pixels,
              int &width,
              int &height)
{
    C_IStream istr (cfile, fileName);
    RgbaInputFile file (istr);

    Box2i dw = file.dataWindow();
    width  = dw.max.x - dw.min.x + 1;
    height = dw.max.y - dw.min.y + 1;
    pixels.resizeErase (height, width);
    file.setFrameBuffer (&pixels[0][0] - dw.min.x - dw.min.y * width, 1, width);
    file.readPixels (dw.min.y, dw.max.y);
}
```

7.3 Preview Images

Graphical user interfaces for selecting image files often represent files as small *preview* or *thumbnail* images. In order to make loading and displaying the preview images fast, OpenEXR files support storing preview images in the file headers.

A preview image is an attribute whose value is of type `PreviewImage`. A `PreviewImage` object is an array of pixels of type `PreviewRgba`. A pixel has four components, `r`, `g`, `b` and `a`, of type `unsigned char`, where `r`, `g` and `b` are the pixel's red, green and blue components, encoded with a gamma of 2.2. `a` is the pixel's alpha channel; `r`, `g` and `b` should be premultiplied by `a`. On a typical display with 8-bits per component, the preview image can be shown by simply loading the `r`, `g` and `b` components into the display's frame buffer. (No gamma correction or tone mapping is required.)

The code fragment below shows how to test if an OpenEXR file has a preview image, and how to access a preview image's pixels:

```
RgbaInputFile file (fileName);

if (file.header().hasPreviewImage())
{
    const PreviewImage &preview = file.header().previewImage();

    for (int y = 0; y < preview.height(); ++y)
        for (int x = 0; x < preview.width(); ++x)
        {
            const PreviewRgba &pixel = preview.pixel (x, y);
            ...
        }
}
```

Writing an OpenEXR file with a preview image is shown in the following example. Since the preview image is an attribute in the file's header, it is entirely separate from the main image. Here the preview image is a smaller version of the main image, but this is not required; in some cases storing an easily recognizable icon may be more appropriate. This example uses the RGBA-only interface to write a scan-line based file, but preview images are also supported for files that are written using the general interface, and for tiled files.

```
void
writeRgbaWithPreview1 (const char fileName[],
                       const Array2D<Rgba> &pixels,
                       int width,
                       int height)
{
    Array2D <PreviewRgba> previewPixels;                                    // 1
    int previewWidth;                                                       // 2
    int previewHeight;                                                      // 3

    makePreviewImage (pixels, width, height,                                // 4
                      previewPixels, previewWidth, previewHeight);
```

```
    Header header (width, height);                                          // 5

    header.setPreviewImage                                                  // 6
        (PreviewImage (previewWidth, previewHeight, &previewPixels[0][0]));

    RgbaOutputFile file (fileName, header, WRITE_RGBA);                     // 7
    file.setFrameBuffer (&pixels[0][0], 1, width);                          // 8
    file.writePixels (height);                                              // 9
}
```

Lines 1 through 4 generate the preview image. Line 5 creates a header for the image file. Line 6 converts the preview image into a `PreviewImage` attribute, and adds the attribute to the header. Lines 7 through 9 store the header (with the preview image) and the main image in a file.

Function `makePreviewImage()`, called in line 4, generates the preview image by scaling the main image down to one eighth of its original width and height:

```
void
makePreviewImage (const Array2D<Rgba> &pixels,
                  int width,
                  int height,
                  Array2D<PreviewRgba> &previewPixels,
                  int &previewWidth,
                  int &previewHeight)
{
    const int N = 8;

    previewWidth  = width / N;
    previewHeight = height / N;
    previewPixels.resizeErase (previewHeight, previewWidth);

    for (int y = 0; y < previewHeight; ++y)
    {
        for (int x = 0; x < previewWidth; ++x)
        {
            const Rgba  &inPixel = pixels[y * N][x * N];
            PreviewRgba &outPixel = previewPixels[y][x];

            outPixel.r = gamma (inPixel.r);
            outPixel.g = gamma (inPixel.g);
            outPixel.b = gamma (inPixel.b);
            outPixel.a = int (clamp (inPixel.a * 255.f, 0.f, 255.f) + 0.5f);
        }
    }
}
```

To make this example easier to read, scaling the image is done by just sampling every eighth pixel of every eighth scan line. This can lead to aliasing artifacts in the preview image; for a higher-quality preview image, the main image should be lowpass-filtered before it is subsampled.

Function `makePreviewImage()` calls `gamma()` to convert the floating-point red, green, and blue components of the sampled main image pixels to `unsigned char` values. `gamma()` is a simplified version of what the exrdisplay program does in order to show an OpenEXR image's floating-point pixels on the screen (for details, see exrdisplay's source code):

```
unsigned char
gamma (float x)
{
    x = pow (5.5555f * max (0.f, x), 0.4545f) * 84.66f;
    return (unsigned char) clamp (x, 0.f, 255.f);
}
```

`makePreviewImage()` converts the pixels' alpha component to unsigned char by by linearly mapping the range [0.0, 1.0] to [0, 255].

Some programs write image files one scan line or tile at a time, while the image is being generated. Since the image does not yet exist when the file is opened for writing, it is not possible to store a preview image in the file's header at this time (unless the preview image is an icon that has nothing to do with the main image). However, it is possible to store a blank preview image in the header when the file is opened. The

preview image can then be updated as the pixels become available. This is demonstrated in the following example:

```
void
writeRgbaWithPreview2 (const char fileName[],
                       int width,
                       int height)
{
    Array <Rgba> pixels (width);

    const int N = 8;

    int previewWidth = width / N;
    int previewHeight = height / N;
    Array2D <PreviewRgba> previewPixels (previewHeight, previewWidth);

    Header header (width, height);
    header.setPreviewImage (PreviewImage (previewWidth, previewHeight));

    RgbaOutputFile file (fileName, header, WRITE_RGBA);
    file.setFrameBuffer (pixels, 1, 0);

    for (int y = 0; y < height; ++y)
    {
        generatePixels (pixels, width, height, y);
        file.writePixels (1);

        if (y % N == 0)
        {
            for (int x = 0; x < width; x += N)
            {
                const Rgba  &inPixel = pixels[x];
                PreviewRgba &outPixel = previewPixels[y / N][x / N];

                outPixel.r = gamma (inPixel.r);
                outPixel.g = gamma (inPixel.g);
                outPixel.b = gamma (inPixel.b);
                outPixel.a = int (clamp (inPixel.a * 255.f, 0.f, 255.f) + 0.5f);
            }
        }
    }

    file.updatePreviewImage (&previewPixels[0][0]);
}
```

7.4 Environment Maps

An environment map is an image that represents an omnidirectional view of a three-dimensional scene as seen from a particular 3D location. Every pixel in the image corresponds to a 3D direction, and the data stored in the pixel represent the amount of light arriving from this direction. In 3D rendering applications, environment maps are often used for image-based lighting techniques that appoximate how objects are illuminated by their surroundings. Environment maps with enough dynamic range to represent even the brightest light sources in the environment are sometimes called "light probe images."

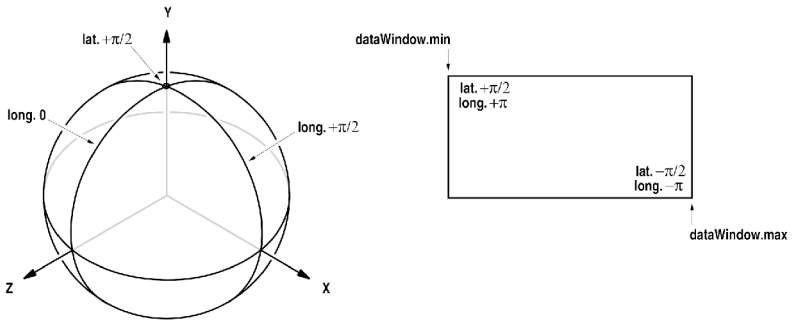
In an OpenEXR file, an environment map is stored as a rectangular pixel array, just like any other image, but an attribute in the file header indicates that the image is an environment map. The attribute's value, which is of type `Envmap`, specifies the relation between 2D pixel locations and 3D directions. `Envmap` is an enumeration type. Two values are possible:

`ENVMAP_LATLONG`

Latitude-Longitude Map: The environment is projected onto the image using polar coordinates (latitude and longitude). A pixel's x coordinate corresponds to its longitude, and the y coordinate corresponds to its latitude. The pixel in the upper left corner of the data window has latitude $+\pi/2$ and longitude $+\pi$; the pixel in the lower right corner has latitude $-\pi/2$ and longitude $-\pi$.

In 3D space, latitudes -π/2 and +π/2 correspond to the negative and positive y direction. Latitude 0, longitude 0 points in the positive z direction; latitude 0, longitude π/2 points in the positive x direction.

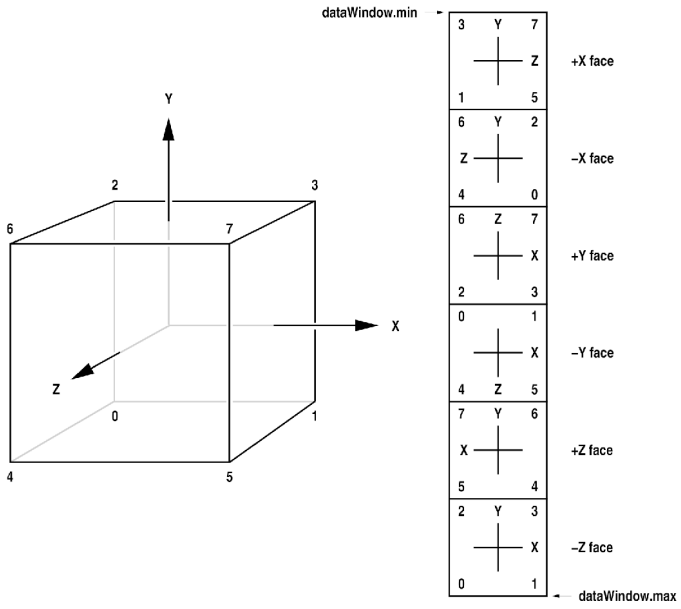
For a latitude-longitude map, the size of the data window should be 2×N by N pixels (width by height), where N can be any integer greater than 0.



ENVMAP_CUBE

Cube Map: The environment is projected onto the six faces of an axis-aligned cube. The cube's faces are then arranged in a 2D image as shown below.

For a cube map, the size of the data window should be N by 6×N pixels (width by height), where N can be any integer greater than 0.



The following code fragment tests if an OpenEXR file contains an environment map, and if it does, which kind:

```
RgbaInputFile file (fileName);

if (hasEnvmap (file.header()))
{
    Envmap type = envmap (file.header());
    ...
}
```

For each kind of environment map, the IlmImf library provides a set of routines that convert from 3D directions to 2D floating-point pixel locations and back. Those routines are useful in application programs that create environment maps and in programs that perform map lookups. For details, see header file `ImfEnvmap.h`.

7.5 Thread-Safety

Except for inititialization, the IlmImf library is thread-safe in the following sense: In a multithreaded application program, multiple threads can concurrently read and write distinct OpenEXR files, but multithreaded reading or writing of a single file requires mutual exclusion. In other words, each thread can independently create, use and destroy its own input and output file objects, but if multiple threads share a single input or output file object, then the application program must ensure mutual exclusion between the threads during accesses to the object.

Before any OpenEXR files can be read or written, the IlmImf library must initialize some internal data structures that are shared between threads. In order to ensure that initialization happens in a thread-safe manner, a multithreaded application program must call the `Imf::staticInitialize()` function before accessing any other functions or classes in the IlmImf library. In a single-threaded program initialization happens automatically; it is not necessary to call `Imf::staticInitialize()`.

High Dynamic Range Rendering in Valve's Source Engine

Gary McTaggart, Chris Green and Jason Mitchell

VALVE

Figure 1 - Scene from Lost Coast

Introduction

After shipping Half-Life 2, we implemented High Dynamic Range (HDR) rendering in the Source engine using a novel method which runs on graphics cards which support 2.0 pixel shaders and 16-bit per channel source textures [Green06]. This means that a whopping 60% of our users get to experience our games in HDR today. This technology was used to create the *Lost Coast* demo and has been used in all subsequent Valve games including *Day of Defeat: Source* and the recent episodic release *Half-Life 2: Episode 1*.

Many different methods for performing HDR rendering on modern GPUs were implemented and evaluated before finding the set of tradeoffs most appropriate for rendering our HDR content on current GPUs. Our implementation makes use of primarily 8-bit per channel textures with 16-bit per channel (either floating point or fixed point depending upon hardware support) light maps and cubic environment maps plus a few 16-bit per channel hand-authored textures for the sky, clouds, and sun.

Figure 2 - Scene from Lost Coast at Varying Exposure Levels

Since all of the shaders in the Source engine's DirectX9 code path are single pass, it is possible to perform HDR lighting and tone mapping operations during normal rendering. This required the addition of a tone mapping subroutine to all pixel shaders but enabled us to render to ordinary 8-bit per channel render targets. By using 8-bit per channel render targets instead of floating point buffers, there is very little performance difference with the low dynamic range path, memory usage is the same and, most importantly, multisample anti-aliasing is supported.

Figure 3 - Scene from Lost Coast at Varying Exposure Levels

Figure 4 – MSAA Supported with HDR Rendering

Using 8-bit per channel render buffers also allows us to avoid an extra copy and conversion of the rendering output to the final display format. When intermediate renderings are used as the inputs for subsequent graphics operations, the dynamic range of these intermediate results is carefully managed so as to still provide high quality outputs from 8-bit data. For instance, renderings done for dynamic reflection in the water have their exposure adjusted by the water's reflectivity coefficient so that proper highlights from the sun are still produced.

HDR Reflections

When actual full-range HDR rendering outputs are needed (such as when rendering the cube maps used for reflection using the engine), the engine is used to produce a series of renderings at varying exposure levels, which are subsequently combined to produce one floating point HDR output image.

(a) Low exposure (b) High exposure

Figure 5 – HDR Environment maps on world geometry

Auto Exposure

Auto exposure calculations needed for tone mapping are performed in a unique fashion in the Source engine. A pixel shader is applied to the output image to determine which output pixels fall within a specified luminance range and writes a flag into the stencil buffer indicating whether each pixel passed the test. An asynchronous occlusion query is then used to count how many pixels were within the specified range.

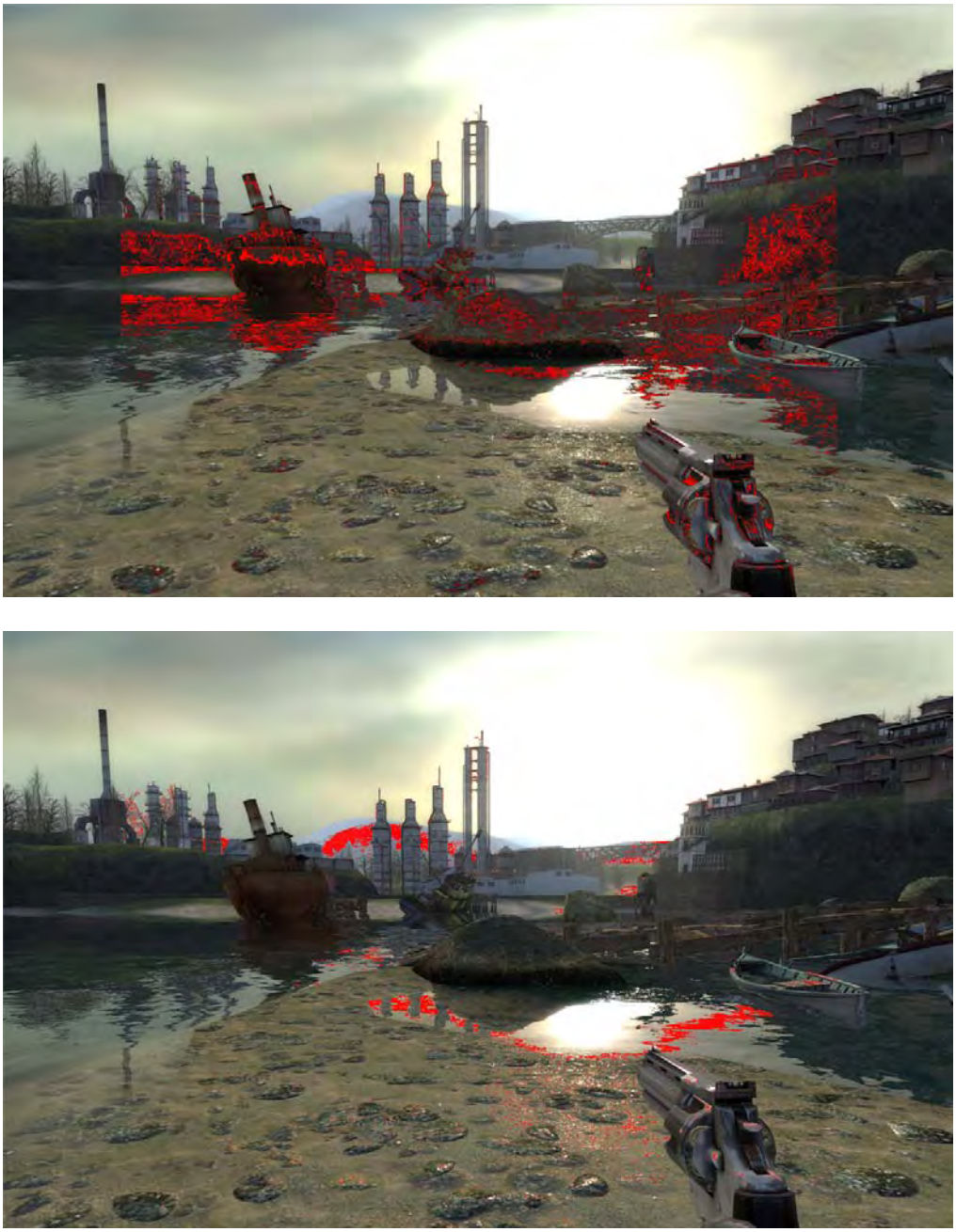


Figure 6 – Two different luminance histogram buckets (shown in red)

When the occlusion query's results become available, they are used to update a running luminance histogram. One luminance range test is done per frame, amortizing the cost of generating the histogram over multiple frames. Time averaging is done to smooth out the results of this incremental histogram. This technique has a number of advantages over other methods, including the fact that it computes a full histogram instead of just an average luminance value, and that this histogram is available to the CPU without stalling the graphics pipeline or performing texture reads from GPU memory. In addition to the running histogram, other auto exposure and bloom filter parameters are controllable by level designers and can be authored to vary spatially with

our game worlds or in response to game events. In addition to adjusting our tone mapping operations as a function of the running luminance of our scenes, we perform an image-space bloom in order to further stylize our scenes and increase the perception of brightness for really bright direct or reflected light. In the offline world, such bloom filtering is typically performed in HDR space prior to tone mapping but we find that tone mapping before blooming is not an objectionable property of our approach, particularly considering the large number of users who get to experience HDR in our games due to this tradeoff.

For those interested in implementation details, the source code for the auto exposure calculation is provided in the Source SDK.