utbm
université de technologie
Belfort-Montbéliard

Fabien Houlmann
Stéphane Metz

High Dynamic Range Rendering in OpenGL

✉ contacts: fabien.houlmann@gmail.com
stephane.metz@gmail.com

Table of contents

Introduction

Today, real-time 3d rendering applications are visually more impressive than before. Thanks to the hardware evolution which allows developers and artists to produce incredible effects.

High Dynamic Range Rendering is a set of techniques that emerge two years ago in some video games like *Half-Life 2: Lost Coast* or *Oblivion*. It was not possible to use real HDR rendering before due to GPU limitations.

In this document, we will expose what is HDR and how can it be used in real time rendering. The OpenGL project associated with this article will demonstrate the use of various HDR effects.

1. High Dynamic Range

1.1. Definition

Usual digital images are created in order to be displayed on monitors. Due to the limited human visual system, these monitors support up to 16.7 million colors (24bits). Thus, it's logical to store numeric image to match the color range of the display. By example, famous file formats like *bmp* or *jpeg* traditionally use 16, 24 or 32 bits for each pixel.

Each pixel is composed of 3 primary colors : red, green and blue (and eventually alpha). So if a pixel is stored as 24 bits, each component value can range from 0 to 255. This is sufficient in most cases but this image can only represent a 256:1 contrast ratio whereas a natural scene exposed in sunlight can expose a contrast of 50,000:1. Most computer monitors have a specified contrast ratio between 500:1 and 1000:1.

High Dynamic Range (HDR) involves the use of a wider dynamic range than usual. That means that every pixel can represent a larger contrast and a larger dynamic range. Usual range is called Low Dynamic Range (LDR).

HDR is typically employed in two applications : imaging and rendering.

- High Dynamic Range Imaging is used by photographers or by movie maker. It's focused on static images where you can have full control and unlimited processing time.

- On the contrary, High Dynamic Range Rendering focus on real-time applications like video games or simulations.

Illustration 1: Farcry without HDR

Illustration 2: Farcry with HDR

1.2. Exposure

When you take a photograph of a scene, you can control how long the sensor will be exposed to light and thus how much light hit the sensor. By controlling exposure time, photographers can create different effects.

An HDR image possess all necessary informations to simulate the exposure time. So a photograph can take a HDR photograph and re-work it on his computer by adjusting exposure after taking it. A few cameras are able to take HDR photographs. But you can create HDR image by composing images with different exposure time. Paul Debevec (http://www.debevec.org) made a great work on this field. One of his famous papers explain how to create HDR light probes from photographs.

Illustration 3: by Dean S. Pemberton, 6 exposures illustrating the Steps to take an HDR Image

Illustration 4: by Dean S. Pemberton, Tone Mapped High dynamic range image example

1.3. Tone mapping

Traditional display devices are limited and cannot render HDR colors. That's why tone-mapping exists. Tone-mapping allow an HDR color to be mapped to Low Dynamic Range in order to be displayed.

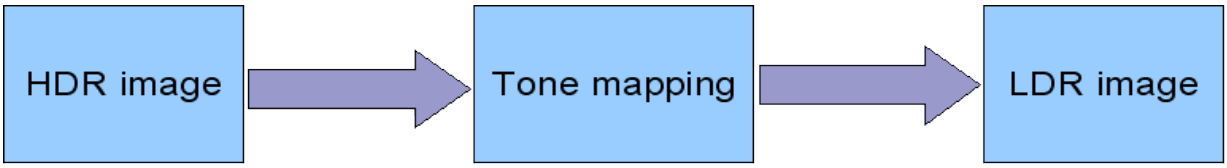
Various tone-mapping operators have been developed and give different visual results. They can be simple and efficient to be used in real time or very complex and customizable for image editing.

An easy tone-mapping operators is $L = \frac{Y}{(Y+1)}$.

This operator will map value from [0; inf) to [0;1).

Some tone-mapping operators use exposure or gamma as a parameter to control the final image. You can find various operators on this page: http://www.mpii.mpg.de/resources/tmo

The scene (or image) is rendered in HDR format and the tone-mapping operator is applied on the resultant image to convert it to Low Dynamic Range in order to be displayed on the monitor.



2. Real Time HDR Rendering

2.1. File format

HDR images contain HDR values contrary to classical digital images which store Low Dynamic Range values. There are mainly two popular formats to manipulate HDR images:

- Radiance:
  In 1985, Greg Ward created the Radiance file format. It's still being used today. These files used the RGBE format, 32bits per pixel. The concept is to use a share exponent for red, green and blue. It can handle very bright pixels without loss of precision for darker ones. Simple code is available to load .hdr files quickly.

- OpenEXR:
  OpenEXR was released in 2003 by Industrial Light and Magic (ILM) as an open standard under a free license. This HDR file format is robust and efficient and mainly used in movies production. OpenEXR support various color formats and is packaged with a set of tools (library, viewer, ...) to manipulate the HDR images.

We chose the Radiance file format in this project because it's easy to use and there are wonderful HDR environment map available made by Paul Debevec (http://www.debevec.org/Probes) in this format.

2.2. Principle

In order to render a scene with High Dynamic Range, you need to compute every calculations using HDR capable variables and buffers. All your GPU rendering pipeline must support HDR : lighting, textures, post processing effects, etc.

The normal rendering process deals with 32 bits colors and each component range in [0;1]. In HDR, values can be greater than 1. If your GPU doesn't support float textures and float render textures, all values will be clamped to [0;1].

HDR effects are more easily developed with vertex and fragment programs.

2.3. HDR in OpenGL

HDR rendering is supported in OpenGL 2.0 but some extensions are needed.

2.3.1. FrameBuffer Object

FBO is an extension (GL_EXT_framebuffer_object) to allow rendering to special framebuffers that can be used as texture. It's a render texture. You can use it as input or as output. You must specify the format of this buffer (like RGBA: 32 bits, RGBA16F: float 64 bits or RGBA32F: float 128 bits).

The idea is to render the scene in a float texture using HDR. This texture will contain HDR values. We then must render this texture on the whole screen to the normal 32bits framebuffer using tone-mapping to convert HDR to LDR.

FBO replace the deprecated pBuffer (pixel buffer) extension. FBO are way more easy and powerful than pBuffer !

2.3.2. GLSL

All lighting effects, blur and tone-mapping are performed in vertex and fragment programs using GL Shading Language. GLSL manipulate float vectors, and consequently is HDR capable.

2.3.3. Buffer formats and limitations

The FBO extension is available on latest GPUs but is not consistent on all cards. The two major vendors, ATI and NVIDIA, released GPUs with some limitations while using render textures.

First, the FBO format GL_RGBA32F (128 bits per color) is supported but is actually so slow than it can't be used for real time applications. If you want to use HDR in your program, you must then consider the GL_RGBA16F format which provided 16 bits floating value per component (called half-float).

Second, some cards don't support bilinear filtering while using render textures. Thus, if you want to realize good blur effect with these GPUs, you must filter the textures by yourself or use some tricks like storing render textures as 32 bits RGBE and custom shaders (more informations on the ATI sdk: HDR Texturing).

And third, OpenGL 2.0 supports non-power-of-two textures which are very useful to manage render textures of every size but some cards don't conform to the norm and don't allow NPOT textures. In this case, you can still use the texture rectangle extension (ARB_texture_rectangle).

You must be aware of these current limitations to implement HDR rendering.

3. The demo

3.1. Libraries

The demo project is a multi platform application (tested on Windows XP and Linux) based on OpenGL.

- **SDL :** We use SDL (Simple DirectMedia Layer) to manage the rendering window, mouse and key inputs.

- **GLee :** GLee (GL Easy Extension library) performs automatic loading of OpenGL extensions.

- **Boost :** In order to make asynchronous loading of resources, we implement threading with boost::thread.

- **MS3D :** Milkshape 3D is a simple modeling software which provide an easy to use file format : .ms3d. So all mesh files are stored as milkshake file and read during loading.

- **RGBE :** We use .hdr format for the cube map textures. Greg Ward release the code to easily load these files. The cube map image is then decomposed to extract each face. With these 6 images, we can upload them to OpenGL to create the cube map texture in the desired format.

  Simple code to read .hdr images:

```
// Read header
RGBE_ReadHeader(hdrfile, &sizeX, &sizeY, NULL);

// Read data
data = new float[3*sizeX*sizeY];
RGBE_ReadPixels_RLE(hdrfile, data, sizeX, sizeY);
```

- **Glintercept :** GLintercept is a must-have for every OpenGL developers. It allows you to intercept all OpenGL calls and gives visual debugging informations. By example you can catch your current frame: it will create images of all textures and of your framebuffer with the different rendering phases. You can also debug your shaders while your application is running and correct them in real-time.

3.2. Environment mapping and associated shaders

3.2.1. Presentation

Environment mapping simulates an object reflecting or refracting its surroundings. This technique is efficient and give very convincing visual results. The idea is to project the 3D environment on a shape and use it to calculate reflection or refraction.

The most famous environment mapping methods are sphere mapping and cube mapping. A sphere

map represent the environment from a fixed point. Sphere mapping has been gradually replaced by cube mapping which produce far better results.

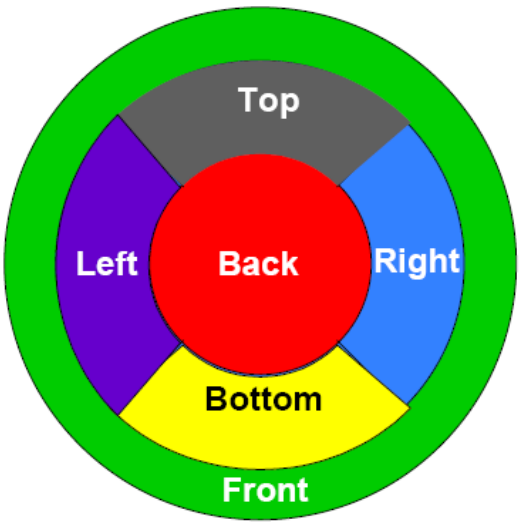


Illustration 5: Spherical Environment Map Shape (source : NVIDIA)

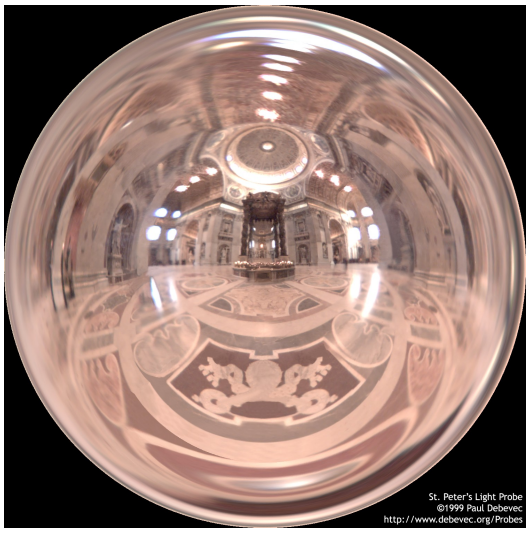


Illustration 6: Light probe from Paul Debevec

A cube map is composed of 6 textures : one for each face of the cube. These 6 images form an omnidirectionnal image and fit together like the faces of a cube.

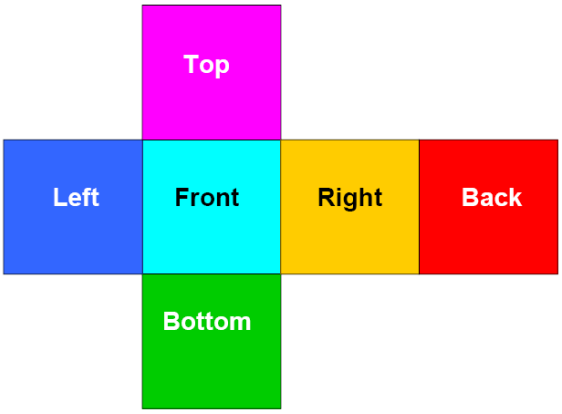


Illustration 7: Cube Environment Map Shape (source : NVIDIA)

Illustration 8: Cross probe from Paul Debevec

A cube map is viewpoint independent and can be dynamic. To access a cube map, you need a 3D vector and not two texture coordinates like 2D texture. This vector is the ray that originate from the center of the cube and intersect with one of the face. The face is then sampled at the right coordinates to get the color.

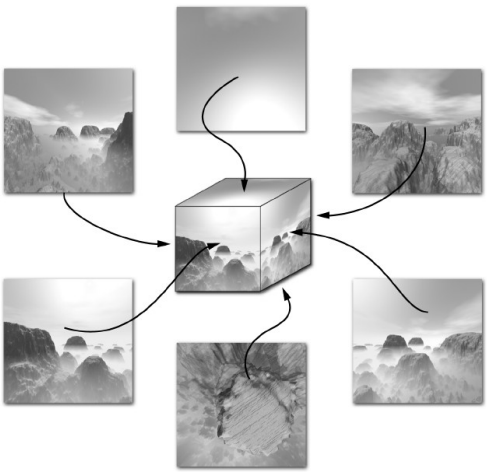


Illustration 9: Texture Images for a Cube Map (source : NVIDIA)

Next we'll explain in some words some effects that use an environment cube map as source.

3.2.2. Reflection

When a ray of light hit a surface of an object, it can be reflected if the surface is appropriate (metal, glass, mirror, ...).

The incident vector (I) goes from the eye to the object's surface. When it hit the surface, it is reflected in the direction R based on the surface normal N.

$$R = I - 2 \cdot N\,(N \cdot I)$$

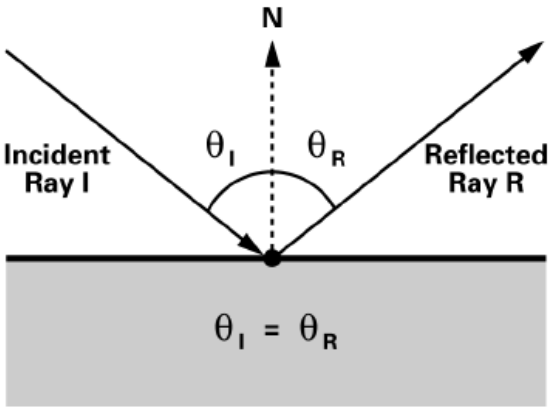


Illustration 10: Calculating the Reflected Ray (source : NVIDIA)

- GLSL reflection fragment program:

```
// Color of the material
uniform vec4 matColor;
// Cube map
uniform samplerCube env;
// Reflection factor
uniform float reflectionFactor;
// Reflected vector
varying vec3 R;

void main()
{
    gl_FragColor = mix(matColor, textureCube(env, R), reflectionFactor);
}
```

- GLSL reflection vertex program:

```
// Position of the view eye in world space
uniform vec3 eyePos;
// Reflected vector
varying vec3 R;

void main()
{
      // Create incident vector
      vec3 I = normalize(gl_Vertex.xyz - eyePos.xyz);
      // Calculate reflected vector
      R = reflect(I, gl_Normal);
      // Transform vertex
      gl_Position = ftransform();
}
```

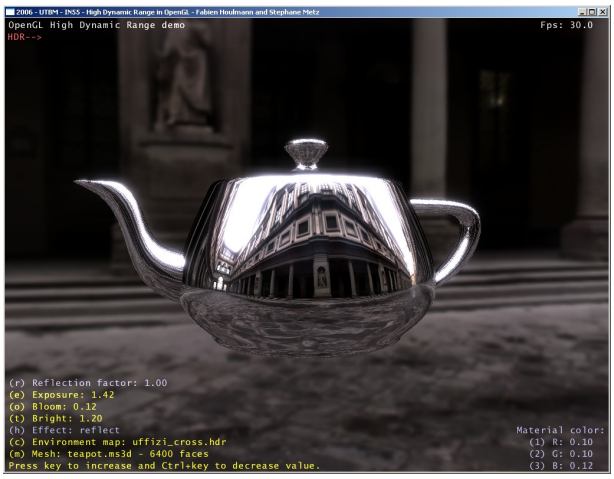


Illustration 11: HDR and reflection shader

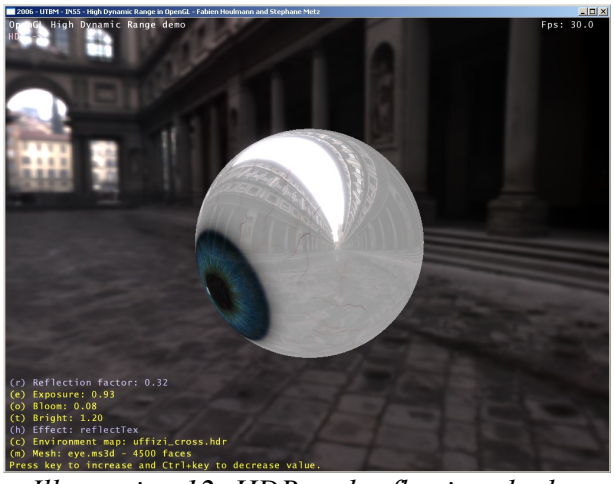


Illustration 12: HDR and reflection shader with decal texture

3.2.3. Refraction

When a ray of light goes through a boundary between two materials of different density, it's direction changes. This change is quantified by the ratio of indices of refraction of the two materials.

Snell's Law: $\eta_1 \cdot \sin(\Theta_I) = \eta_2 \cdot \sin(\Theta_T)$

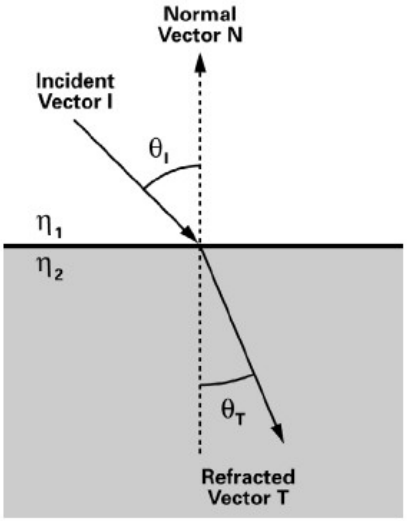


Illustration 13: Snell's Law
(source : NVIDIA)

- GLSL refraction fragment program:

```
// Cube map
uniform samplerCube env;
// Refracted vector
varying vec3 R;

void main()
{
      gl_FragColor  = textureCube(env, R);
}
```

- GLSL refraction vertex program:

```
// Position of the view eye in world space
uniform vec3 eyePos;

// Ratio of indices of refraction
uniform float etaRatio;

// Refracted vector
varying vec3 R;

void main()
{
      // Create incident vector
      vec3 I = normalize(gl_Vertex.xyz - eyePos.xyz);

      // Calculate refracted vector
      R = refract(I, gl_Normal, etaRatio);

      // Transform vertex
      gl_TexCoord[0] = gl_MultiTexCoord0;
      gl_Position = ftransform();
}
```

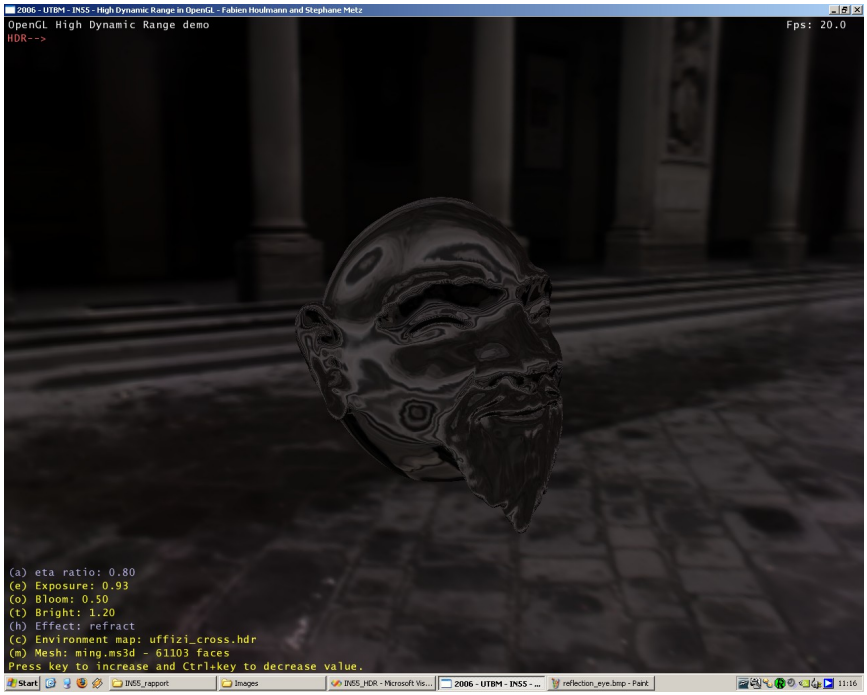


Illustration 14: HDR and refraction shader

3.2.4. Fresnel Effect

In real scene, when light hit a boundary between two materials, some light reflects off the surface and some refracts through the surface. The Fresnel equation describes this phenomenon precisely. This equation is complex, so it's common to use a simplified version.

Approximation of the Fresnel Equation:

$$reflectionCoefficient = max(0, min(1, bias + scale \times (1 + I \cdot N)^{power}))$$

Illustration 15: HDR and fresnel shader

- GLSL fresnel effect fragment program:

```
// Cube map
uniform samplerCube env;
// Reflected and refracted vectors
varying vec3 reflectedVector, refractedVector;
// Reflection factor based on fresnel equation
varying float refFactor;

void main()
{
      // Read cube map
      vec4 reflectedColor = textureCube(env, reflectedVector);
      vec4 refractedColor = textureCube(env, refractedVector);
      // Mix reflected and refracted colors
      gl_FragColor = mix(refractedColor, reflectedColor, refFactor);
}
```

- GLSL fresnel effect vertex program:

```
// Position of the view eye in world space
uniform vec3 eyePos;
// Fresnel parameters
uniform float fresnelBias, fresnelScale, fresnelPower;
// Ratio of indices of refraction
uniform float etaRatio;
// Reflected and refracted vectors
varying vec3 reflectedVector, refractedVector;
// Reflection factor based on fresnel equation
varying float refFactor;

void main()
{
      // Create incident and normal vectors
      vec3 I = normalize(gl_Vertex.xyz - eyePos.xyz);
      vec3 N = normalize(gl_Normal);
      // Calculate reflected and refracted vectors
      reflectedVector = reflect(I, N);
      refractedVector = refract(I, N, etaRatio);
      // Appoximation of the fresnel equation
      refFactor = fresnelBias+fresnelScale*pow(1+dot(I,N), fresnelPower);

      // Transform vertex
      gl_TexCoord[0] = gl_MultiTexCoord0;
      gl_Position = ftransform();
}
```

3.2.5. Chromatic dispersion

In fact, when a ray of light is refracted, it's direction is not only based on the ration of indices of refraction but also depends on the wavelength of the incident light. This phenomenon is known as chromatic dispersion.

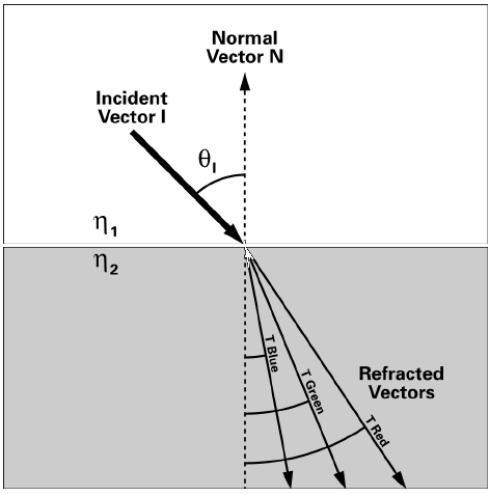


Illustration 16: Understanding Chromatic Dispersion (source : NVIDIA)

- GLSL chromatic dispersion fragment program:

```
// Cube map
uniform samplerCube env;
// Reflected and refracted vectors
varying vec3 R, TRed, TGreen, TBlue;
// Reflection factor based on fresnel equation
varying float refFactor;

void main()
{
      // Read cube map for reflected color
      vec4 reflectedColor = textureCube(env, R);
      // Read cube map for refracted color (3 times)
      vec4 refractedColor;
      refractedColor.r = textureCube(env, TRed).r;
      refractedColor.g = textureCube(env, TGreen).g;
      refractedColor.b = textureCube(env, TBlue).b;
      // Mix reflected and refracted colors
      gl_FragColor = mix(refractedColor, reflectedColor, refFactor);
}
```

- GLSL chromatic dispersion vertex program:

```
// Position of the view eye in world space
uniform vec3 eyePos;
// Ratio of indices of refraction for the 3 colors
uniform vec3 etaRatioRGB;
// Fresnel parameters
uniform float fresnelBias, fresnelScale, fresnelPower;
// Reflection vector;
varying vec3 R;
// Separate refraction vectors
varying vec3 TRed, TGreen, TBlue;
// Reflection factor based on fresnel equation
varying float refFactor;
```

```
void main()
{
	// Create incident and normal vectors
	vec3 I = normalize(gl_Vertex.xyz - eyePos.xyz);
	vec3 N = normalize(gl_Normal);
	// Reflection vector
	R = reflect(I, N);
	// Separate refraction vectors
	TRed = refract(I, N, etaRatioRGB.r);
	TGreen = refract(I, N, etaRatioRGB.g);
	TBlue = refract(I, N, etaRatioRGB.b);
	// Appoximation of the fresnel equation
	refFactor=fresnelBias+fresnelScale*pow(1,0+dot(I,N),fresnelPower);
	// Transform vertex
	gl_TexCoord[0] = gl_MultiTexCoord0;
	gl_Position = ftransform();
}
```

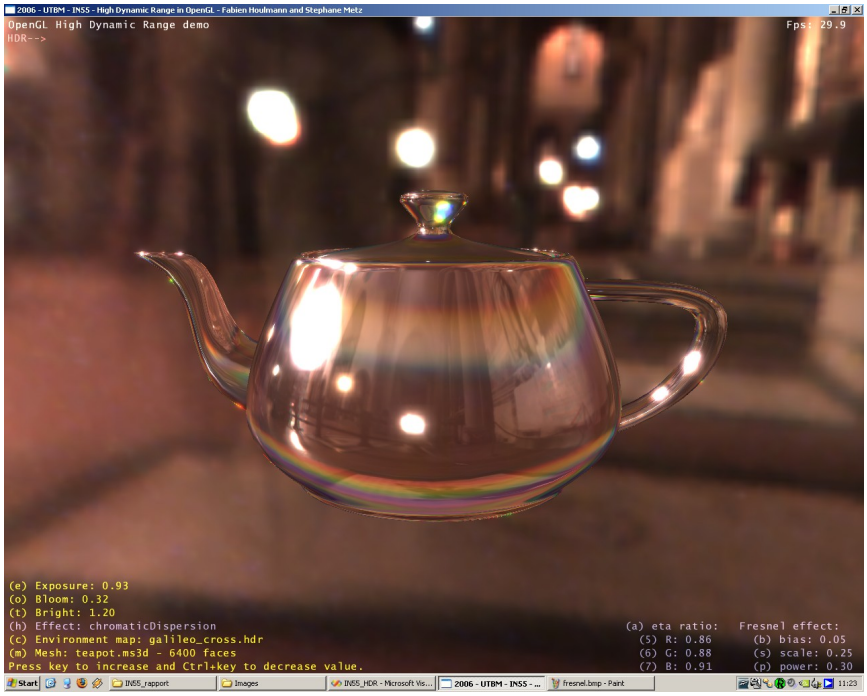


Illustration 17: HDR and chromatic dispersion shader

3.3. HDR rendering pipeline

The demo scene is quite simple: a mesh with one of the precedent effect and a skybox representing the cube map.

- First we activate a float FBO to render the scene on it.
  ➔ *Check source code : OpenGLRenderer, OpenGLFBO, HdrApp.*

- Second, the sky box is rendered using the selected cube map. This cube map is read from .hdr file and stored in the GL_RGBA16F format so it contains HDR values.
  ➔ *Check source code : Image, ImageManager, OpenGLTextureCube.*

- Third, we draw the mesh on the center of the scene with one effect on it (reflection, refraction, Fresnel effect or chromatic dispersion). The mesh reflect and refract the environment in HDR so the mesh is colored with HDR values. The current framebuffer contains an image in HDR.
  ➔ *Check source code : TriMesh, TriMeshManager, OpenGLShader.*

- Next, we create the bloom effect on the whole scene. To do this, very bright areas of the image are extracted in a HDR FBO. This texture is then down-sampled. That means that we reduce it's size a desired number of times with bilinear filtering active. The down-sampled textures are blurred with a median filter shader.
  ➔ *Check source code : DownSampler.*

- Then, we composed the original image by adding all down-sampled blurred textures with additive blending. The resulting effect is called bloom.

- Finally we get an HDR framebuffer with the desired scene and effects. We must now convert HDR image to LDR by applying a tone-mapping operator on the texture. We try a lots of operators but only a few give us good results.

Tone map operator: $$Y = exposure \times \left(\frac{\left(\frac{exposure}{maxLuminance} + 1 \right)}{(exposure + 1)} \right)$$

This operator allow the user to control exposure. With a very small exposure, the scene is very dark but some hidden details appear. On the other hand, with a high exposure, the scene is very bright.

- GLSL tone-mapping fragment program:

```
// render texture and bloom map
uniform sampler2D tex, bloom;
// Control exposure with this value
uniform float exposure;
// How much bloom to add
uniform float bloomFactor;
// Max bright
uniform float brightMax;

void main()
{
      vec2 st = gl_TexCoord[0].st;
      vec4 color = texture2D(tex, st);
      vec4 colorBloom = texture2D(bloom, st);

      // Add bloom to the image
      color += colorBloom * bloomFactor;

      // Perform tone-mapping
      float Y = dot(vec4(0.30, 0.59, 0.11, 0.0), color);
      float YD = exposure * (exposure/brightMax + 1.0) / (exposure + 1.0);
      color *= YD;

      gl_FragColor = color;
}
```

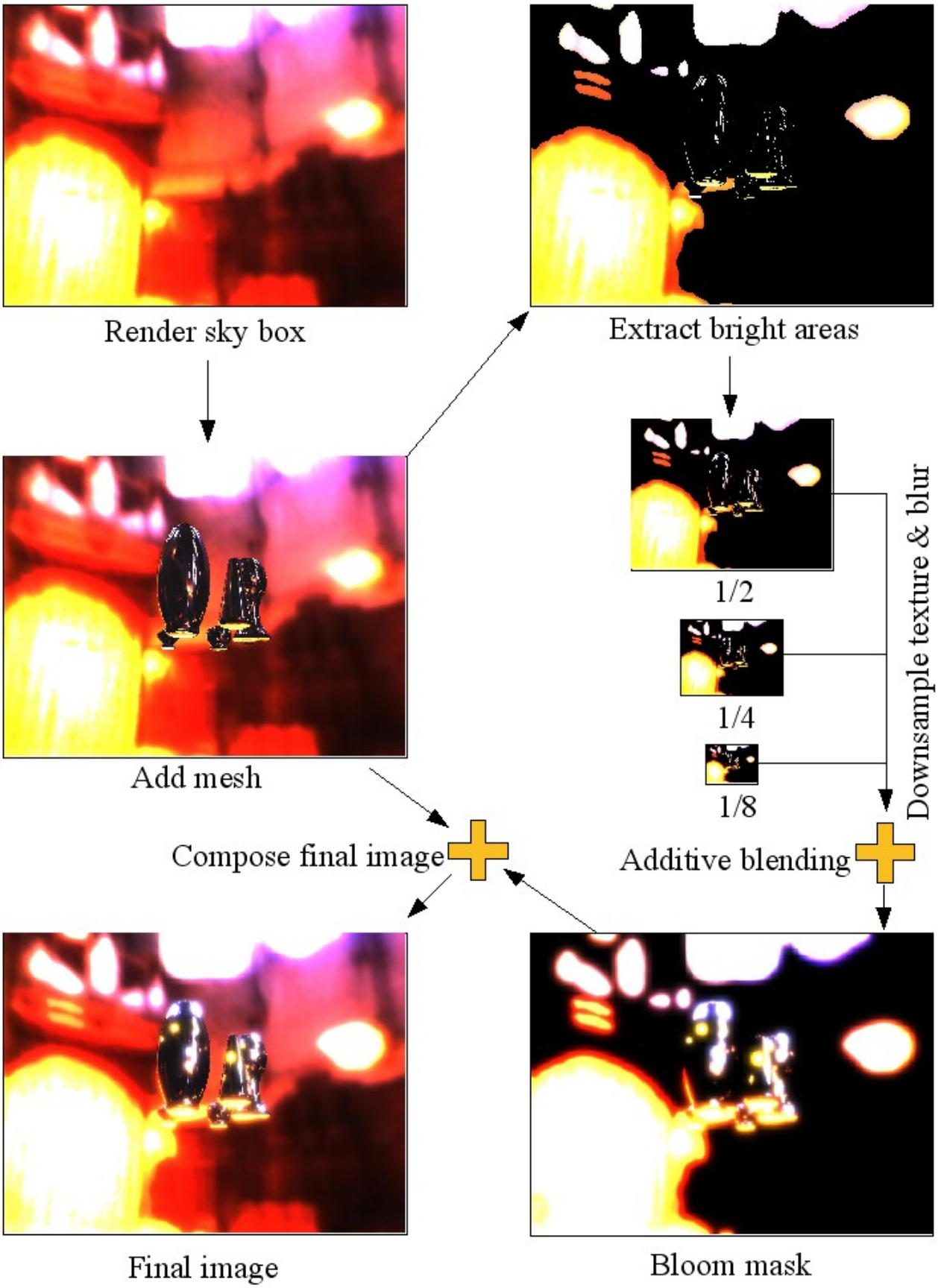


Illustration 18: HDR rendering process

3.4. Manual

The demo offers a high level of customization.
Each press on a key increase the associated value
Each press on key with control key down decrease the associated value.

3.4.1. Basic features

left-click and move	➔	rotate around the mesh.
right-click and move	➔	zoom in or out.
M	➔	change mesh
C	➔	change cube map
H	➔	change shader effect
Space	➔	reset effects

3.4.2. HDR & LDR

The rendering window is cut to show the difference between HDR and LDR rendering. You can control the ratio of the screen by pressing LEFT and RIGHT keys.

3.4.3. Advanced features

E	➔	allow the user to control exposure.
O	➔	change the intensity of the bloom effect
T	➔	modify the brightness threshold used to extract bright areas of the scene.

3.4.4. Effects parameters

R	➔	change the reflection factor
A	➔	change the η ratio of materials indice
B, S, P	➔	change Fresnel parameters
1, 2, 3	➔	vary the color of the material
5, 6, 7	➔	vary the η ratio for each individual color

3.5. Improvements

This demo was realized in a limited time for a school project and thus doesn't contain all HDR effects. For example, HDR rendering can achieve very good results with depth of field (A good demo using DirectX can be found here: rthdribl).

We also just implement bloom effect, but you can add some streaks or lens flare effects which are very similar (Shaders for game programmers and artits)

Exposure can be controlled by the application to reproduce the same perception as the human eye. That means that the eye adapt himself to the current environment luminance. This technique called auto-exposure calculate the mean luminance of the scene and determine the correct exposure based on transition time and desired luminance.

3.6. Screenshots

3.6.1. Exposure examples

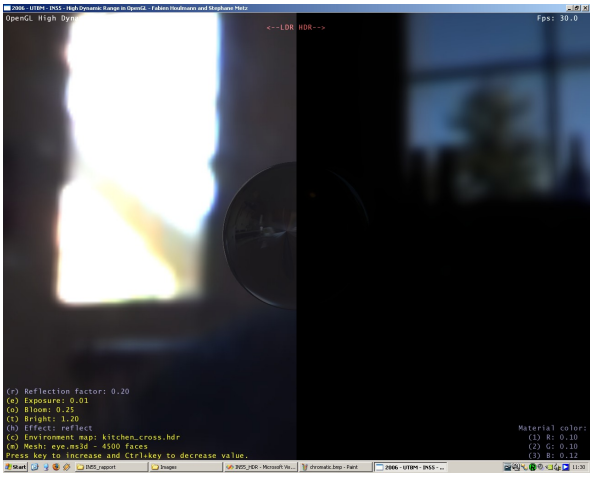


Illustration 19: Minimum exposure

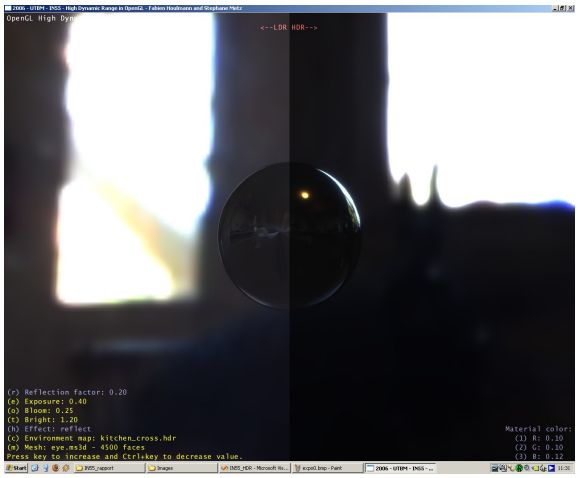


Illustration 20: Low exposure

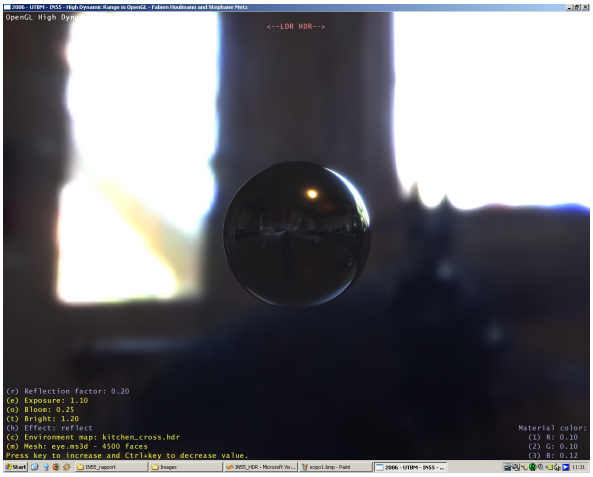


Illustration 21: Normal exposure

Illustration 22: High exposure

3.6.2. HDR & LDR comparison

Illustration 23: Car with Fresnel effect

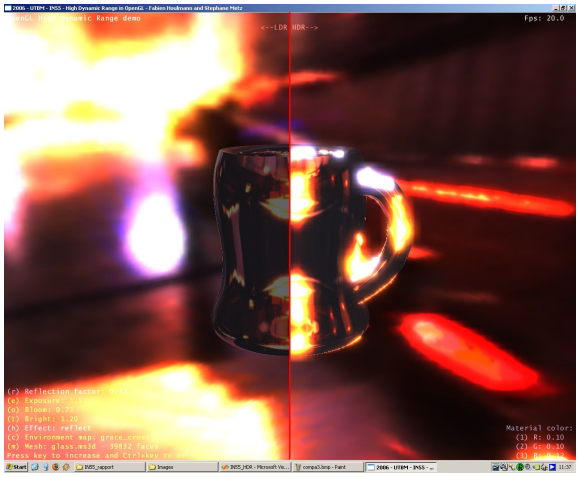


Illustration 24: Glass with Fresnel effect

3.6.3. Various examples

Illustration 25: Reflecting car

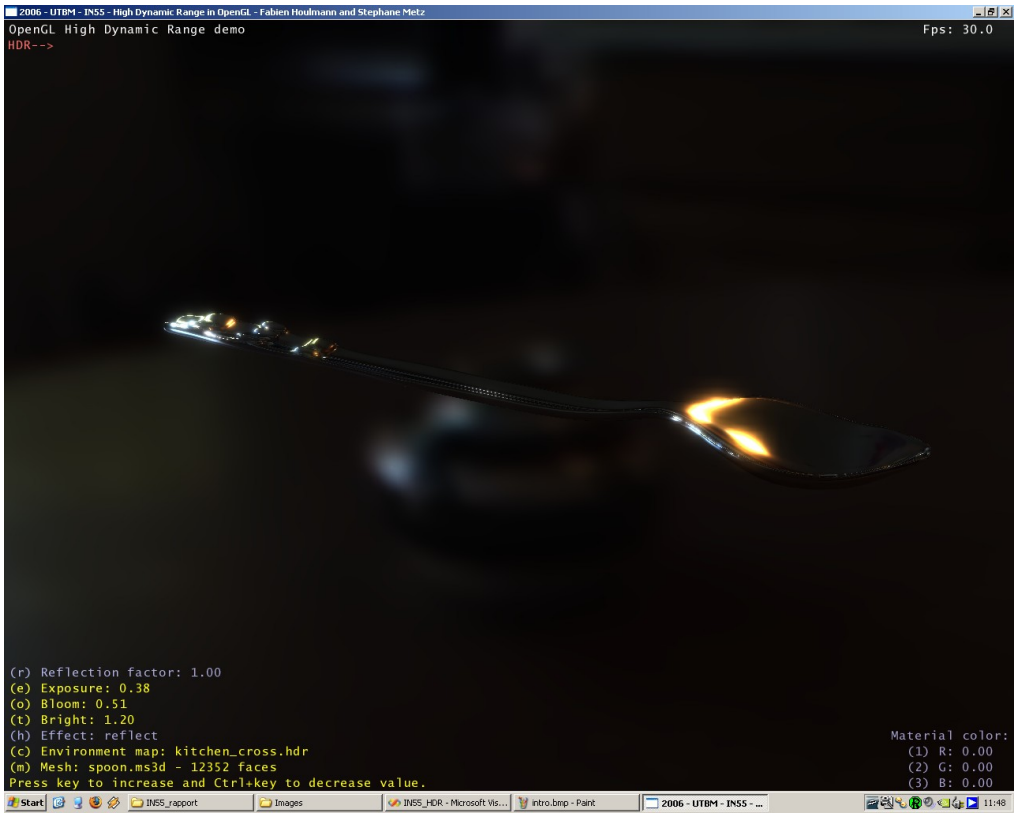


Illustration 26: Reflecting spoon

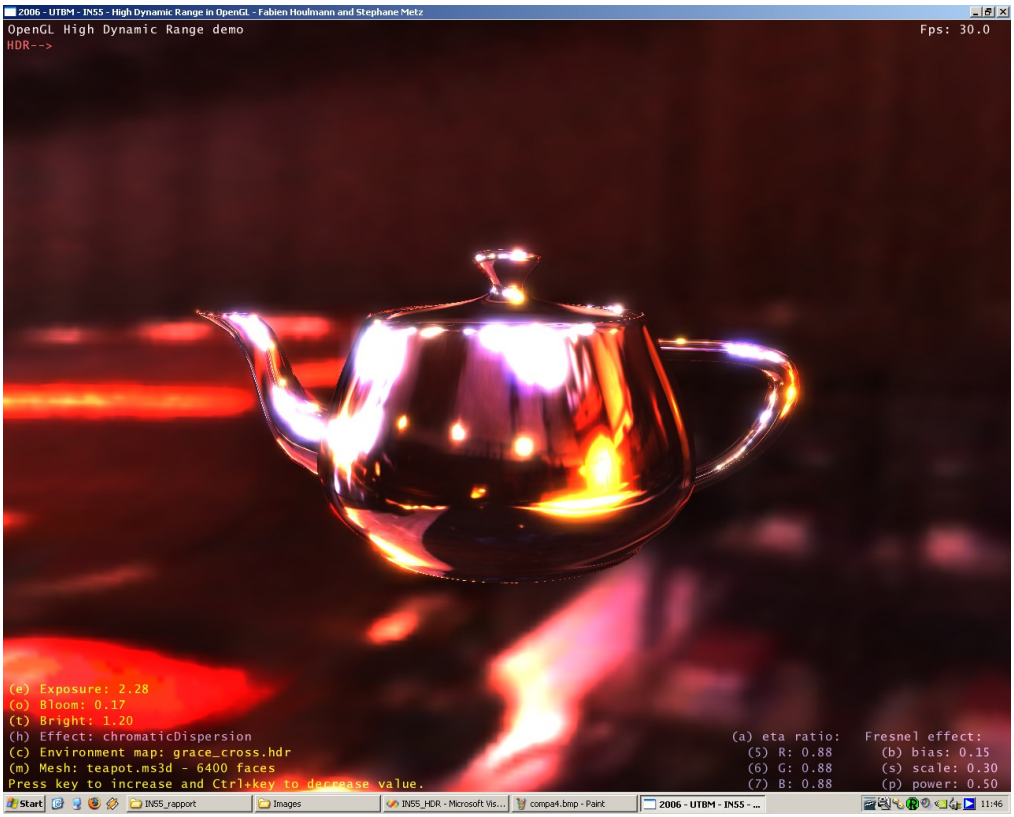


Illustration 27: Reflecting teapot

Conclusion

High Dynamic Range Rendering and Imaging make the virtual scene more real by controlling the final image with exposure, bloom or other effects. HDR tends to mimic natural effects because it eliminates clamping and can represent wide range of intensity levels as in real scenes.

HDR is still a young technique and consequently was not used often in the industry. Recent video cards with shaders and floating point buffer support allow HDR rendering to emerge in commercial products like video games.

Even though current hardware can manage High Dynamic Range image, our screens can only render 16 million colors. That's why tone-mapping must be applied in order to map HDR value to screen limited value. HDR screens are still being experimented but we can hope to be equipped with full HDR hardware in the future.

Bibliography

OpenGL:
http://opengl.org
http://opengl.org/documentation/specs/version2.0/glspec20.pdf
http://nehe.gamedev.net

OpenGL Shading Language:
http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf
http://www.lighthouse3d.com/opengl/glsl
http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html

OpenGL Cube Map Texturing:
http://developer.nvidia.com/object/cube_map_ogl_tutorial.html

The CG tutorial:
http://developer.nvidia.com/object/cg_tutorial_home.html

Shaders for game programmers and artists (Chapter 8):
http://www.courseptr.com/downloads/chapterpreview/00924-ch8prev.pdf

High Dynamic Range Rendering:
http://www.gamedev.net/reference/articles/article2208.asp

High Dynamic Range with DirectX:
http://www.daionet.gr.jp/~masa/rthdribl
http://www.gamedev.net/columns/hardcore/hdrrendering
msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/HDRLighting_Sample.asp

Dynamic Range In Digital Photography:
http://www.cambridgeincolour.com/tutorials/dynamic-range.htm

Tone Mapping:
http://en.wikipedia.org/wiki/Tone_mapping
http://graphics.cs.uni-sb.de/Courses/ss02/CG2/folien/12ToneMapping.pdf

Paul Debevec home page:
http://www.debevec.org

GPU sdk:
http://www.ati.com/developer
http://developer.nvidia.com

Free 3d models:
http://www.turbosquid.com