mental ray Architectural and Design Visualization Shader Library

Document version 1.7.5
June 07, 2007

Table of Contents

Chapter 1

Architectural and Design Visualization Shader Library

1.1 About the Library

The *mental ray* **architectural** library contains a set of shaders designed for architectural and design visualization.

The most important are the *mia_material(_x)*, an easy to use all-around general purpouse material, and the **Physical Sun and Sky** along with the portal light shaders.

The library also contains many other tools, like shaders for...

- ...creating render-time "rounded corners".
- ...physically based camera exposure and depth-of-field.
- ...light-surfaces and environment blurring.

In standalone *mental ray* the shaders are added by including the "mi" declaration file and linking to the library;

```
link "architectural.dll"
include "architectural.mi"
```

The library strictly requires *mental ray* version 3.6 or newer and will not function on earlier releases of *mental ray*.

1.2 Introduction

1.2.1 What is the *mia_material*?

The *mental ray mia_material* is a monolithic material shader that is designed to support most materials used by architectural and product design renderings. It supports most hard-surface materials such as metal, wood and glass. It is especially tuned for fast glossy reflections and refractions (replacing the DGS material) and high-quality glass (replacing the dielectric material).

The major features are:

- **Easy to use** - yet flexible. Controls arranged logically in a "most used first" fashion.
- **Templates** - for getting faster to reality.
- **Physically accurate** - the material is energy conserving, making shaders that breaks the laws of physics impossible.
- **Glossy performance** - advanced performance boosts including interpolation, emulated glossiness, and importance sampling.
- **Tweakable BRDF**[1] - user can define how reflectivity depends on angle.
- **Transparency - "Solid" or "thin" materials** - transparent objects such as glass can be treated as either "solid" (refracting, built out of multiple faces) or "thin" (non-refracting, can use single faces).
- **Round corners** - shader can simulate "fillets" to allow sharp edges to still catch the light in a realistic fashion.
- **Indirect Illumination control** - set the final gather accuracy or indirect illumination level on a per-material basis.
- **Oren-Nayar diffuse** - allows "powdery" surfaces such as clay.
- **Built in Ambient Occlusion** - for contact shadows and enhancing small details.
- **All-in-one shader** - photon and shadow shader built in.
- **Waxed floors, frosted glass and brushed metals...** - ...all fast and easy to set up.
- **Multiple outputs** - when using *mia_material_x*

[1] Bidirectional Reflectance Distribution Function

1.2.2 mental ray 3.6 enhancements and *mia_material_x*

The material comes in two variants, the mental ray 3.5 compatible *mia_material* and the new extended *mia_material_x*. These are just two different interfaces using the same underlying code, so the functionality is identical, except that *mia_material_x*...

- ...has some additional parameters relating to bump mapping described on page 38.
- ...supports setting **ao_do_details** to 2 for enabling "ambient occlusion with color bleed" (see page 30).
- ...returns multiple outputs in the form of a mental ray *struct* return. The various outputs are described in detail on page 41.

1.2.3 Structure of this Document

This document is divided into sections of *Fundamentals* (beginning on page 3) which explain the main features of the material , the *Parameters* section (page 15) that goes through all the parameters .

1.3 Fundamentals

1.3.1 Physics and the Display

The *mia_material* primarily attempts to be *physically accurate* hence it has an output with a high dynamic range. How visually pleasing the material looks depends on how the mapping of colors inside the renderer to colors displayed on the screen is done.

When working with the *mia_material* it is highly encouraged to make sure one is operating through a *tone mapper/exposure control* or at the very least are using gamma correction.

1.3.1.1 A Note on Gamma

Describing all the details about *gamma correction* is beyond the scope of this document and this is just a brief overview.

The color space of a normal off-the-shelf computer screen is not linear. The color with RGB value 200 200 200 is *not* twice as bright as a color with RGB value 100 100 100 as one would expect.

This is not a "bug" because due to the fact that our eyes see light in a *non linear* way, the former color is actually *perceived* to be about twice as bright as the latter. This makes the

color space of a normal computer screen roughly *perceptually uniform*. This is a good thing, and is actually the main reason 24 bit color (with only 8 bits - 256 discrete levels - for each of the red, green and blue components) looks as good as it does to our eyes.

The problem is that physically correct computer graphics operate in a *true linear* color space where a value represents actual light energy. If one simply maps the range of colors output to the renderer naively to the 0-255 range of each RGB color component it is incorrect.

The solution is to introduce a mapping of some sort. One of these methods is called *gamma correction*.

Most computer screens have a gamma of about 2.2[2], but most software default to a gamma of 1.0, which makes everything (especially mid-tones) look too dark, and light will not "add up" correctly.

Using gamma of 2.2 is the theoretically "correct" value, making the physically linear light inside the renderer appear in a correct linear manner on screen.

However, since the response of photographic film isn't linear either, users have found this "theoretically correct" value looks too "bright" and "washed out", and a very common compromise is to render to a gamma of 1.8, making things look more "photographic", i.e. as if the image had been shot on photographic film and then developed.

1.3.1.2 Tone Mapping

Another method to map the physical energies inside the renderer to visually pleasing pixel values is known as *tone mapping*. This can be done either by rendering to a floating point file format and using external software, or use some plugin to the renderer to do it on-the-fly.

Two tone mapping shaders are included in the library, the simple *mia_exposure_simple* and the more advanced *mia_exposure_photographic*, both of which are documented on page 79

Note: Take special care when using tone mapping together with gamma correction; some tone mapping shaders has their own gamma correction feature built in, and if one is not careful one can end up with washed out gamma due to it being applied twice. Make sure to keep an eye on the gamma workflow so it is applied in *one* place.

1.3.2 Use Final Gathering and Global Illumination

The material is designed to be used in a realistic lighting environment, i.e. using full direct and indirect illumination.

In mental ray there are two basic methods to generate indirect light: Final Gathering and Global Illumination. For best results at least one of these methods should be used.

[2] This is also known as the "sRGB" color space.

At the very least one should enable Final Gathering, or use Final Gathering combined with Global Illumination (photons) for quality results. Performance tips for using Final Gather and Global Illumination can be found on page 45 of this document.

If you are using an environment for your reflections, make sure the same environment (or a blurred copy of it) is used to light the scene through Final Gathering.

1.3.3 Use Physically Correct Lights

Traditional computer graphics light sources live in a cartoon universe where the intensity of the light doesn't change with the distance. The real world doesn't agree with that simplification. Light decays when leaving a light source due to the fact that light rays diverge from their source and the "density" of the light rays change over distance. This decay of a point light source is $1/d^2$, i.e. light intensity is proportional to the inverse of the square of the distance to the source.

One of the *reasons* for this traditional oversimplification is actually the fact that in the early days of computer graphics *tone mapping* was not used and problems of colors "blowing out" to white in the most undesirable ways[3] was rampant.

However, as long as only Final Gathering (FG) is used as indirect illumination method, such traditional simplifications *still work*. Even light sources with *no decay* still create reasonable renderings! This is because FG is only concerned with the transport of light from one surface to the next, not with the transport of light from the light source to the surface.

It's when working with Global Illumination (GI) (i.e. with *photons*) the troubles arise.

When GI is enabled, light sources shoot photons. It is *imperative* for the *mia_material* (or any other *mental ray* material) to work properly for the energy of these photons to *match the direct light cast by that same light!* And since photons model light in a physical manner, decay is "built in".

Hence, when using GI:

- Light sources must be emitting photons at the correct energy
- The direct light must decay in a physically correct way to match the decay of the photons.

Therefore it is important to make sure the *light shader* and the *photon emission shader* of the lights work well together.

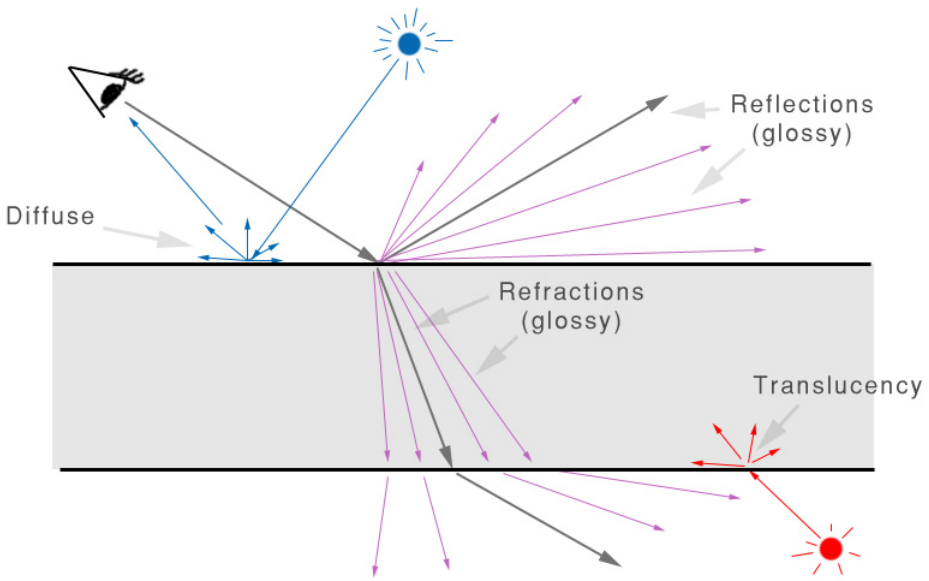
[3] Raw clipping in sRGB color space is very displeasing to the eye, especially if one color channel clips earlier than the others. Tone mapping generally solves this by "soft clipping" in a more suitable color space than sRGB.

1.4 Features

1.4.1 The Shading Model

From a usage perspective, the shading model consists of three components:

- **Diffuse** - diffuse channel (including Oren Nayar "roughness").
- **Reflections** - glossy anisotropic reflections (and highlights).
- **Refractions** - glossy anisotropic transparency (and translucency).



The mia_material shading model

Direct and indirect light from the scene both cause diffuse reflections as well as translucency effects. Direct light sources also cause traditional "highlights" (specular highlights).

Raytracing is used to create reflective and refractive effects, and advanced importance-driven multi-sampling is used to create glossy reflections and refractions.

The rendering speed of the glossy reflections/refractions can further be enhanced by interpolation as well as "emulated" reflections with the help of Final Gathering.

1.4.2 Conservation of Energy

One of the most important features of the material is that it is *automatically energy conserving*. This means that it makes sure that $diffuse + reflection + refraction <= 1$, i.e. that no energy is magically created and the incoming light energy is properly distributed to the diffuse, reflection and refraction components in a way that maintains the first law of thermodynamics[4].

In practice, this means for example that when adding more reflectivity, the energy must be taken from somewhere, and hence the diffuse level and the transparency will be automatically reduced accordingly. Similarly, when adding transparency, this will happen at the cost of the diffuse level.

[4] The first law of thermodynamics is that no one talks about thermodynamics ;)

The rules are as follows:

- Transparency takes energy from Diffuse, i.e. at 100% transparency, there will be no diffuse at all.
- Reflectivity takes energy from both Diffuse and Transparency, i.e. at 100% reflectivity there will neither be any diffuse nor any transparency.
- Translucency is a type of transparency, and **refr_trans_w** defines the percentage of transparency vs. translucency.

From left to right: Reflectivities 0.0, 0.4, 0.8 and 1.0

From left to right: Transparencies 0.0, 0.4, 0.8 and 1.0

It also means that the *level* of highlights is linked to the *glossiness* of a surface. A high **refl_gloss** value causes a narrower but very intense highlight, and a lower value causes a wider but less intense highlight. This is because the energy is now spread out and dissipated over a larger solid angle.

1.4.3 BRDF - how Reflectivity Depends on Angle

In the real world, the reflectivity of a surface is often *view angle dependent*. A fancy term for this is BRDF (*Bi-directional Reflectance Distribution Function*), i.e. a way to define how much a material reflects when seen from various angles.

The reflectivity of the wooden floor depends on the view angle

Many materials exhibit this behavior. Glass, water and other *dielectric* materials with *fresnel* effects (where the angular dependency is guided strictly by the *Index of Refraction*) are the most obvious examples, but other layered materials such as lacquered wood, plastic, etc. display similar characteristics.

The *mia_material* allows this effect both to be defined by the Index of Refraction, and also allows an explicit setting for the two reflectivity values for:

- 0 degree faces (surfaces directly facing the camera)
- 90 degree faces (surfaces 90 degrees to the camera)

See the BRFD section on page 27 for more details.

1.4.4 Reflectivity Features

The final surface reflectivity is in reality caused by the sum of three components:

- The Diffuse effect
- The actual reflections
- Specular highlights that simulate the reflection of light sources

Diffuse, Reflections and Highlights

In the real world "highlights" are just (glossy) reflections of the light sources. In computer graphics it's more efficient to treat these separately. However, to maintain physical accuracy the material automatically keeps "highlight" intensity, glossiness, anisotropy etc. in sync with the intensity, glossiness and anisotropy of reflections, hence there are no separate controls for these as both are driven by the reflectivity settings.

1.4.5 Transparency Features

The material supports full glossy anisotropic transparency, as well as includes a translucent component, described more in detail on page 24.

Translucency

1.4.5.1 Solid vs. Thin-Walled

The transparency/translucency can treat objects either as *solid* or *thin walled*.

If all objects were treated as solids at all times, every single window pane in an architectural model would have to be modeled as *two* faces; an entry surface (that refracts the light slightly in one direction), and immediately following it an exit surface (where the light would be refracted back into the original direction).

Not only is this additional modeling work, it is a waste of rendering power to model a refraction that has very little net effect on the image. Hence the material allows modeling the entire window pane as one single flat plane, foregoing any actual "'refraction" of light.

Solid vs. Thin-walled transparency and translucency

In the above image the helicopter canopy, the window pane, the translucent curtain and the right sphere all use "thin walled" transparency or translucency, whereas the glass goblet, the plastic horse and the left sphere all use "solid" transparency or translucency.

1.4.5.2 Cutout Opacity

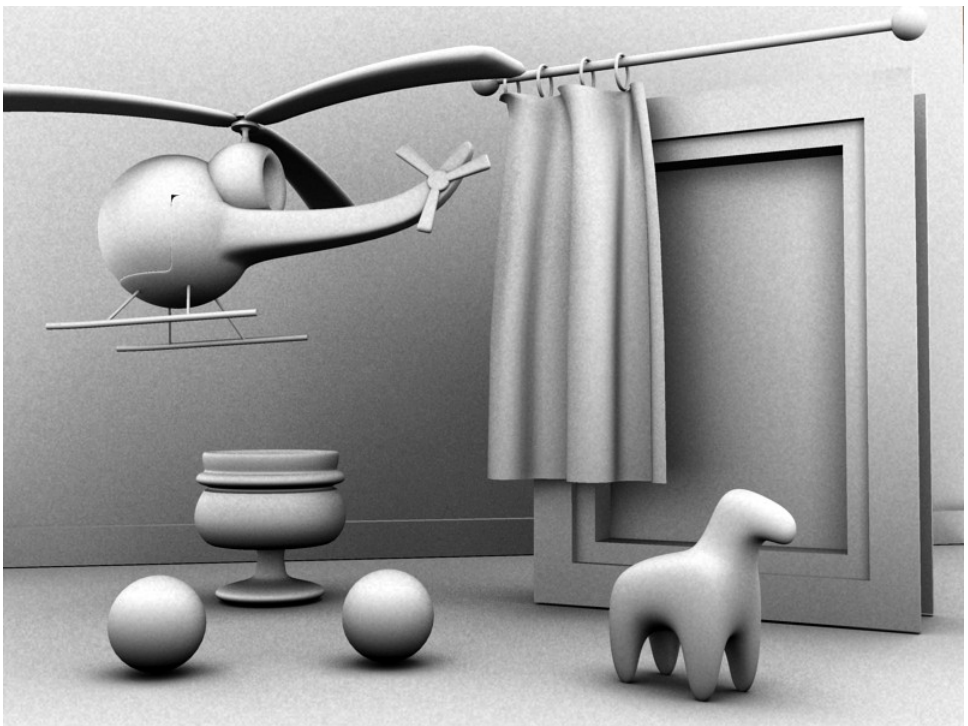
Beyond the "physical" transparency (which models an actual property of the material) there is a completely separate non-physical "cutout opacity" channel to allow "billboard" objects such as trees, or to cut out things like a chain-link fence with an opacity mask.

1.4.6 Special Effects

1.4.6.1 Built-in Ambient Occlusion

Ambient Occlusion (henceforth referred to as "AO") is a method spearheaded by the film industry to emulate the "look" of true global illumination by using shaders that calculate how *occluded* (i.e. blocked) an area is from receiving incoming light.

Used alone, an AO shader[5] creates a grayscale output that is "dark" in areas to which light cannot reach and "bright" in areas where it can:

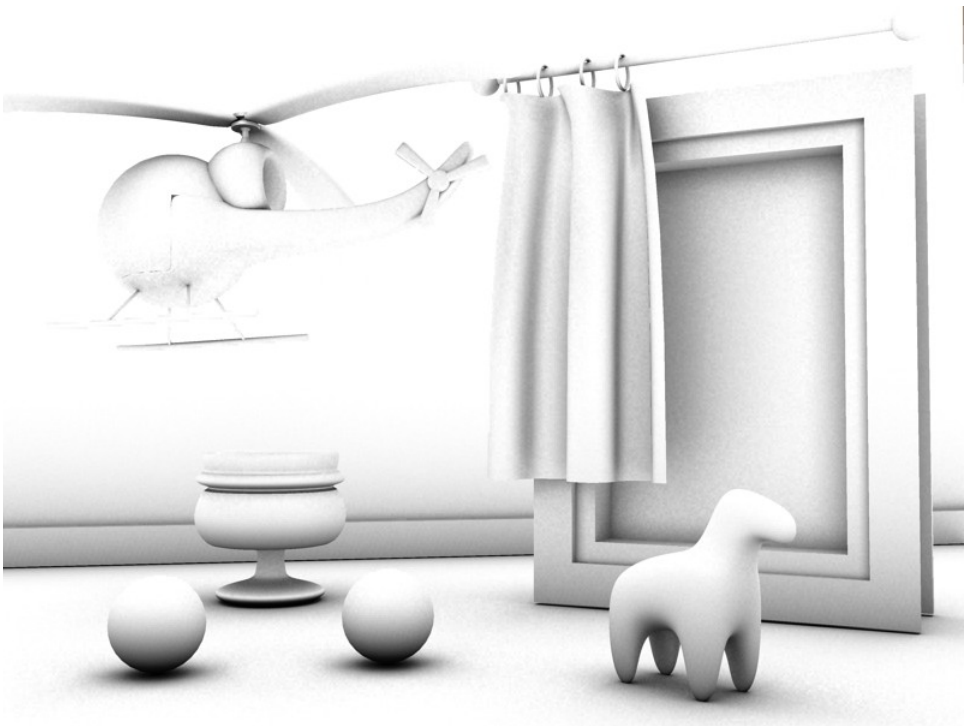


An example of AO applied to a scene

As seen in the above image, one of the main results of AO is dark in crevices and areas where light is blocked by other surfaces and it is bright in areas that are exposed to the environment.

One important aspect of AO is that one can tune the "distance" within which it looks for occluding geometry.

[5] Like the separate mental ray **mib_amb_occlusion** shader.



AO looked up within a shorter radius

Using a radius creates only a "localized" AO effect; only surfaces that are within the given radius are actually considered occluders (which is also *massively* faster to render). The practical result is that the AO gives us nice "contact shadow" effects and makes small crevices visible.

There are two ways to utilize the built in AO in the *mia_material*:

- "Traditional" AO for adding an omnipresent ambient light that is then attenuated by the AO to create details.
- Use AO for *detail enhancement* together with existing indirect lighting methods (such as Final Gathering or photons).

The latter method is especially interesting when using a highly "smoothed" indirect illumination solution (i.e. a very high photon radius, or an extremely low final gather density) which could otherwise lose small details. By applying the AO with short rays these details can be brought back.

1.4.7 Performance Features

Finally the *mia_material* contains a large set of built in functions for top performance, including but not limited to:

- Advanced importance sampling with ray rejection thresholds
- Adaptive glossy sample count
- Interpolated glossy reflection/refraction with detail enhancements
- Ultra fast emulated glossy reflections (**refl_hl_only** mode)
- Possibility to ignore internal reflections for glass objects
- Allowing a choice between traditional transparent shadows (suitable for e.g. a window pane) and refractive caustics (suitable for solid glass objects) on a per material basis.

1.5 Quick Guide to the Material Parameters

This section gives a quick overview of the parameters suitable as a memory refreshing tool for users already familiar with *mia_material*. A much more detailed run down is on page 17. Parameters labeled with [+] only exist in *mia_material_x*.

- **diffuse_weight** The amount of diffuse reflections.
- **diffuse** The diffuse color, i.e. the main color of the material.
- **diffuse_roughness** The Oren-Nayar "rougness".
- **reflectivity** Overall reflectivity level. Multiplied by the **brdf_xx_degree_refl** parameters.
- **refl_color** Overall reflectivity color. Normally white.
- **refl_gloss** Reflection glossiness. 1.0 = perfect mirror.
- **refl_gloss_samples** Number of samples (rays) for glossy reflections.
- **refl_interpolate** Interpolation (smoothing) of the glossy reflections. Speed at the price of accuracy.
- **refl_hl_only** Skip actual reflections, do only highlights and "emulated" reflections via FG.
- **refl_is_metal** Metal mode. Uses the diffuse color as reflection color.
- **transparency** The overall transparency level.
- **refr_color** The transparency (refraction) color.
- **refr_gloss** The transparency glossiness.
- **refr_ior** The Index of Refraction.
- **refr_gloss_samples** Number of samples (rays) for glossy refractions.
- **refr_interpolate** Interpolation (smoothing) of the glossy refractions.
- **refr_translucency** Enables translucency
- **refr_trans_color** The translucency color
- **refr_trans_weight** The translucency weight
- **anisotropy** Anisotropy. 1.0 = Isotropic.
- **anisotropy_rotation** The rotation of the anisotropy direction.
- **anisotropy_channel** The coordinate space to derive anistropy direction from.
- **brdf_fresnel** When on, uses the Fresnel equation (based on IOR) for the reflectivity curve, when off, uses the "user defined" settings below.

- **brdf_0_degree_refl** The user defined reflectivity curve value for surfaces facing the viewer.
- **brdf_90_degree_refl** The user defined reflectivity curve value for grazing surfaces.
- **brdf_curve** The user defined reflectivity curve shape.
- **brdf_conserve_energy** When on, makes sure that energy is conserved. Keep this on!
- **intr_grid_density** Interpolation grid density.
- **intr_refl_samples** Number of interpolation samples for reflections.
- **intr_refl_ddist_on** Enable "Detail Distance"
- **intr_refl_ddist** The detail distance.
- **intr_refr_samples** Number of interpolation samples for refraction.
- **single_env_sample** Do only a single environment sample even if multiple reflectivity rays are traced. Used together with *mia_envblur*.
- **refl_falloff_on** Enable distance falloff for reflections
- **refl_falloff_dist** The distance at which no reflections are seen.
- **refl_falloff_color_on** Enable the falloff color. When off, falls of to the environment color.
- **refl_falloff_color** The falloff color when above is on.
- **refl_depth** The trace depth for reflections.
- **refl_cutoff** The importance cutoff for reflections.
- **refr_falloff_on** Enable distance falloff for refractions (transparency).
- **refr_falloff_dist** The distance at where no transparency is seen *or* when the falloff "color" is reached.
- **refr_falloff_color_on** When off, reflections fall off to black (total absorption). When on, falls off to the given color tint.
- **refr_falloff_color** The color tintint for reflections per distance travelled in the medium.
- **refr_depth** The trace depth for refractions.
- **refr_cutoff** The importance cutoff for refractions.
- **indirect_multiplier** The weighting of indirect illumination (FG, GI, Caustics)
- **fg_quality** The quality of FG
- **fg_quality_w** The weighting of above parameter (used for texture mapping).
- **ao_on** Enable Ambient Occlusion (AO).
- **ao_samples** Number of AO probe rays.

- **ao_distance** The maximum distance to look for occlusion. Shorter is faster.
- **ao_dark** The "shadow color" of the AO
- **ao_ambient** The "additional light" for the AO.
- **ao_do_details** Indirect illumination detail enhancement mode. 1=using AO, 2=with color bleed.
- **thin_walled** Treat surfaces as thin wafers of material, rather than the boundaries of solids.
- **no_visible_area_hl** Disables traditional "highlights" for visible area lights.
- **skip_inside_refl** Skips weak reflections on inside of glass.
- **do_refractive_caustics** Do refractive caustics, rather than transparent shadows. Only when Caustics mode is on.
- **backface_cull** Make surfaces invisible (to the camera only) from their back side.
- **propagate_alpha** Transparent surfaces propagate the alpha channel value of what is behind them.
- **hl_vs_refl_balance** The relative intensity of "highlights" to reflections.
- **cutout_opacity** The overall opacity. Used for stencil/cut-out effects.
- **additional_color** An additional color simply added to the other shading.
- **bump** A shader used for perturbing the normal.
- **no_diffuse_bump** Disables bump mapping for the diffuse shading.
- **mode** The light list mode.
- **lights** The lights list itself.

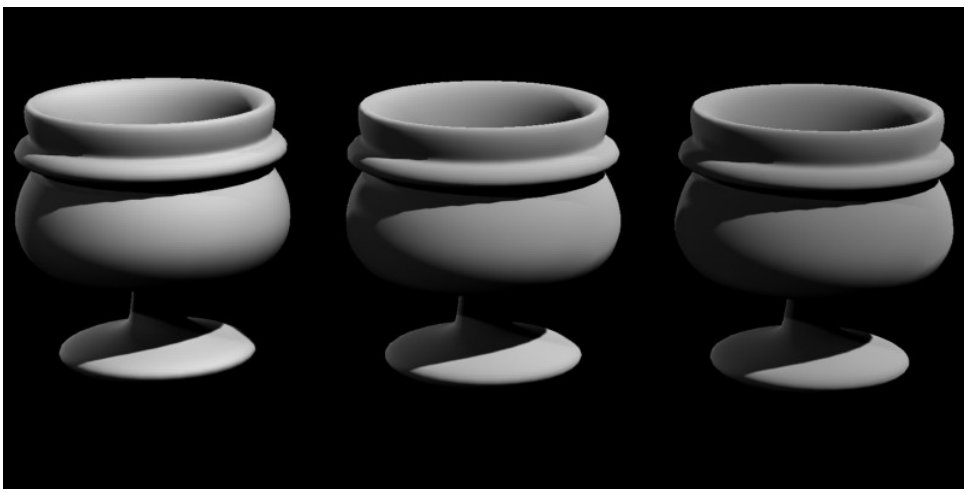
+ **bump_mode** Defines the mode of the new bump inputs. If zero, uses the compatible "bump" above.
+ **overall_bump** Bump that always affects everything.
+ **standard_bump** General bump. Does not affect diffuse when **no_diffuse_bump** is on.
+ **multiple_outputs** When on, activates the multiple outputs. When off, only writes to "result".

1.6 Detailed Description of Material Parameters

1.6.1 Diffuse

diffuse_weight sets the desired level (and **diffuse** the color) of the diffuse reflectivity. Since the material is energy conserving, the *actual* diffuse level used depends on the reflectivity and transparency as discussed above.

The diffuse component uses the Oren-Nayar shading model. When **diffuse_roughness** is 0.0 this is identical to classical Lambertian shading, but with higher values the surface gets a a more "powdery" look:



Roughness 0.0 (left), 0.5 (middle) and 1.0 (right)

1.6.2 Reflections

1.6.2.1 Basic Features

The **reflectivity** and **refl_color** together define level of reflections as well as the intensity of the traditional "highlight" (also known as "specular highlight").

This value is the *maximum* value - the *actual* value also depends on the angle of the surface and come from the BRDF curve. This curve (described in more detail on page 27) allows one to define a **brdf_0_degree_refl** (for surfaces facing the view) and **brdf_90_degree_refl** (for surfaces perpendicular to the view).

No reflectivity (left), angle dependent (center), constant (right)

- The left cup shows no reflectivity at all and a purely diffuse material.
- The center cup shows a **brdf_0_degree_refl** of 0.1 and a **brdf_90_degree_refl** of 1.0.
- The right cup has a both a **brdf_0_degree_refl** and **brdf_90_degree_refl** of 0.9, i.e. constant reflectivity across the surface.

Note how the high reflectivity automatically "subtracts" from the white diffuse color. If this didn't happen, the material would become unrealistically over-bright, and would break the laws of physics [6].

The **refl_gloss** parameter defines the surface "glossiness", ranging from 1.0 (a perfect mirror) to 0.0 (a diffusely reflective surface):

[6] See page 6.

Glossiness of 1.0 (left), 0.5 (center) and 0.25 (right)

The **refl_samples** parameters defines the maximum[7] number of samples (rays) are shot to create the glossy reflections. Higher values renders slower but create a smoother result. Lower values render faster but create a grainier result. Generally 32 is enough for most cases.

There are two special cases:

- Since a **refl_gloss** value of 1.0 equals a "perfect mirror" it is meaningless to shoot multiple rays for this case, hence only one reflection ray is shot.
- If the **refl_samples** value is set to 0, the reflections will be "perfect mirror" (and only one ray shot) *regardless* of the actual value of **refl_gloss** . This can be used to boost performance for surfaces with very weak reflections. The highlight still obeys the glossiness value.

Metallic objects actually influence the color of their reflection whereas other materials do not. For example, a gold bar will have gold colored reflections, whereas a red glass orb does not have red reflections. This is supported through the **refl_is_metal** option.

- When *off*, the **refl_color** parameter defines the color and **reflectivity** parameter (together with the BRDF settings) the intensity and colors of reflections.
- When *on*, the **diffuse** parameter defines the color of reflections, and **reflectivity** parameter sets the "weight" between diffuse reflections and glossy (metallic) reflections.

[7] The actual number is adaptive and depends on reflectivity, ray importance, and many other factors.

No metal reflections (left), Metal reflections (center), Metal mixed with diffuse (right)

The left image shows non-metallic reflections (**refl_is_metal** is off). One can see reflections clearly contain the color of the objects they reflect and are not influenced by the color of the materials.

The center image uses metallic reflections (**refl_is_metal** is on). Now the color of reflections are influenced by the color of the material. The right image shows a variant of this with the **reflectivity** at 0.5, creating a 50:50 mix between colored reflections and diffuse reflections.

1.6.2.2 Performance Features

Glossy reflections need to trace multiple rays to yield a smooth result, which can become a performance issue. For this reason there are a couple of special features designed to enhance their performance.

The first of those features is the *interpolation*. By turning **refl_interpolate** on, a smoothing algorithm allows rays to be re-used and smoothed[8]. The result is faster and smoother glossy reflections at the expense of accuracy. Interpolation is explained in more detail on page 35.

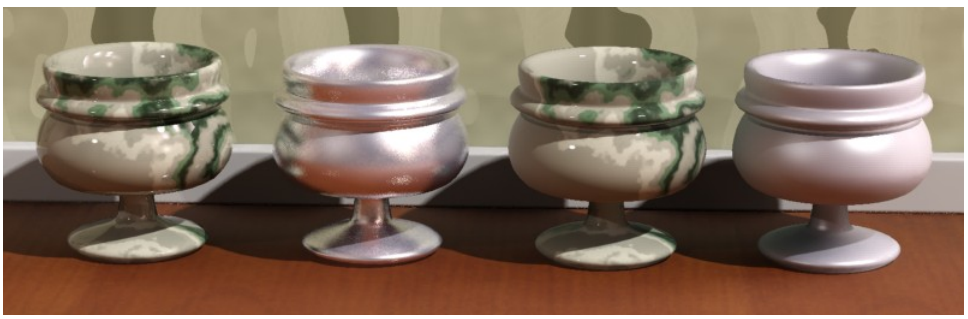
For highly reflective surfaces it is clear that true reflection rays are needed. However, for less reflective surfaces (where it is less "obvious" that the surface is really reflecting anything) there exists a performance-enhancing shortcut, the **refl_hl_only** switch.

When **refl_hl_only** is *on*, no actual reflection rays are traced. Instead only the "highlights" are shown, as well as soft reflections *emulated* with the help of using Final Gathering[9].

[8] The technique works best on flat surfaces.

[9] If Final Gathering is not enabled, this mode simply shows the highlights and attempts no emulation of reflections.

The **refl_hl_only** mode takes *no* additional render time compared to a non-glossy (diffuse) surface, yet can yield surprisingly convincing results. While it may not be completely convincing for "hero" objects in a scene it can work very well for less essential scene elements. It tends to work best on materials with weak reflections or *extremely* glossy (blurred) reflections:



The left two cups use real reflections, those on the right use ***refl_hl_only***

While the two cups on the left are undoubtedly more convincing than those on the right, the fact that the right hand cups have no additional render time compared to a completely non-reflective surface makes this mode very interesting. The emulated reflections still pull in a *directional color bleed* such that the bottom side of the cup is influenced by the color of the wooden floor just as if it was truly reflective.

1.6.3 Refractions

The **transparency** parameter defines the level of refractions and **refr_color** defines the color. While this color can be used to create "colored glass", there is a slightly more accurate method to do this described on page 46.

Due to the materials energy conserving nature (see page 6) the value set in the **transparency** parameter is the *maximum* value - the *actual* value depends on the reflectivity as well as the BRDF curve.

The **refr_ior** defines the Index of Refraction, which is a measurement of how much a ray of light "bends" when entering a material. Which direction light bends depends on if it is *entering* or *exiting* the object. The *mia_material* use the direction of the surface normal as the primary cue for figuring out whether it is entering or exiting. It is therefore *important* to model transparent refractive objects with the surface normal pointing in the proper direction.

The IOR can also be used to define the BRDF curve, which is what happens in the class of transparent materials known as "dielectric" materials, and is illustrated here:

Index of refraction 1.0 (left), 1.2 (center) and 1.5 (right)

Note how the leftmost cup looks completely unrealistic and is almost invisible. Because an IOR of 1.0 (which equals that of air) is impossible, we get no change in reflectivity across the material and hence perceive no "edges" or change of any kind. Whereas the middle and rightmost cups have a realistic change in reflectivity guided by the IOR.

One is however not forced to base the reflectivity on the IOR but can instead use the BRDF mode to set it manually:

Different types of transparency

The left cup again acquires it's curve from the index of refraction. The center cup has a manually defined curve, which has been set to a **brdf_90_degree_refl** of 1.0 and a **brdf_0_degree_refl** of 0.2, which looks a bit more like metallized glass. The rightmost cup uses the same BRDF curve, but instead is set to "thin walled" transparency (see page 10).

Clearly, this method is the better way to make "non-refractive" objects compared to simply setting **refr_ior** to 1.0 as we tried above.

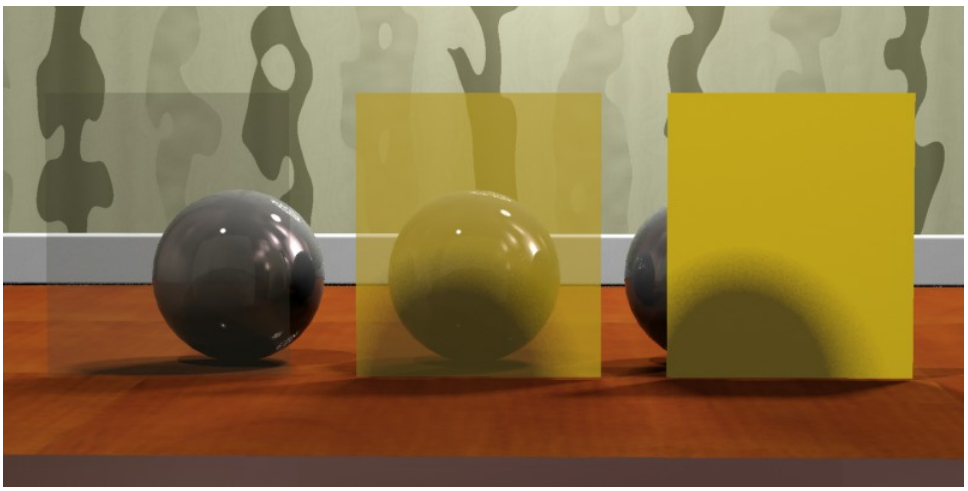
As with reflections, the **refr_gloss** parameter defines how sharp or blurry the refractions/transparency are, ranging from a 1.0 (a completely clear transparency) to 0.0 (an extremely diffuse transparency):

*A **refr_gloss** of 1.0 (left), 0.5 (center) and 0.25 (right)*

Just as with the glossy reflections, the glossy transparency has a **refr_interpolate** switch, allowing faster, smoother, but less accurate glossy transparency. Interpolation is described on page 35.

1.6.4 Translucency

Translucency is handled as a special case of transparency, i.e. to use translucency there must first exist some level of transparency, and the **refr_trans_w** parameter decides how much of this is used as transparency and how much is translucency:

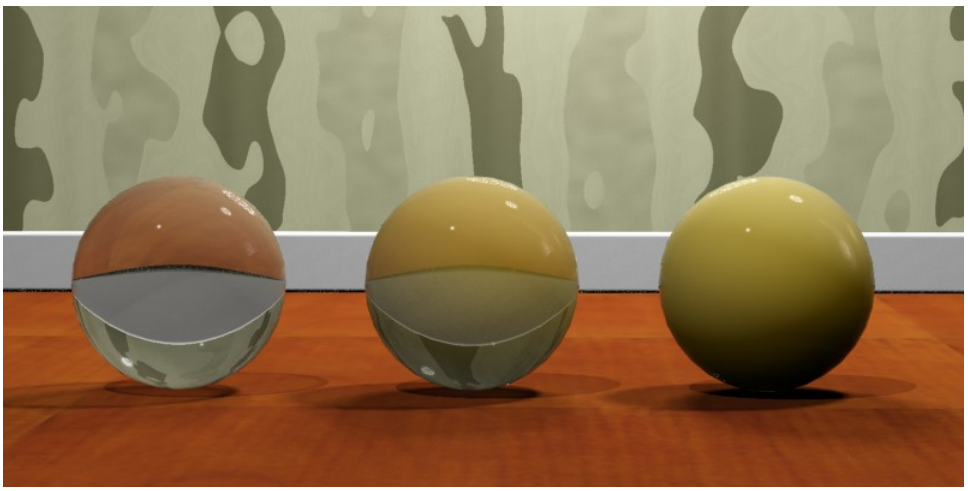


*A **transparency** of 0.75 and a **refr_trans_w** of 0.0 (left), 0.5 (center) and 1.0 (right)*

- If **refr_trans_w** is 0.0, all of the **transparency** is used for transparency.
- If **refr_trans_w** is 0.5, half of the **transparency** is used for transparency and half is used for translucency.
- If **refr_trans_w** is 1.0, all of the **transparency** is used for translucency and there is *no* actual transparency.

The translucency is primarily intended to be used in "thin walled" mode (as in the example above) to model things like curtains, rice paper, or such effects. In "thin walled" mode it simply allows the shading of the reverse side of the object to "bleed through".

The shader also operates in "Solid" mode, but the implementation of translucency in the *mia_material* is a simplification concerned solely with the transport of light from the back of an object to it's front faces and is not "true" SSS (sub surface scattering). An "SSS-like" effect can be generated by using glossy transparency coupled with translucency but it is neither as fast nor as powerful as the dedicated SSS shaders.



Solid translucency w. ***refr_trans_w*** *of 0.0 (left), 0.5 (center) and 1.0 (right)*

1.6.5 Anisotropy

Anisotropic reflections and refractions can be created using the **anisotropy** parameter. The parameter sets the ratio between the "width" and the "height" of the highlights, hence when **anisotropy**is 1.0 there is no anisotropy, i.e. the effect is disabled.

For other values of **anisotropy** (above and below 1.0 are both valid) the "shape" of the highlight (as well as the appearance of reflections) change.

anisotropy *values of 1.0 (left), 4.0 (center) and 8.0 (right)*

The anisotropy can be rotated by using the **anisotropy_rotation** parameter. The value 0.0 is un-rotated, and the value 1.0 is one full revolution (i.e. 360 degrees). This is to aid using a

texture map to steer the angle:

anisotropy_rotation *values of 0.0 (left), 0.25 (center) and textured (right)*

Note: When using a textured **anisotropy_rotation** it is important that this texture is *not* anti-aliased (filtered). Otherwise the anti-aliased pixels will cause local vortices in the anisotropy that appear as seam artifacts.

For values of 0 or above, the space which defines the "stretch directions" of the highlights are derived from the texture space set by **anisotropy_channel**[10].

anisotropy_channel can also have the following "special" values:

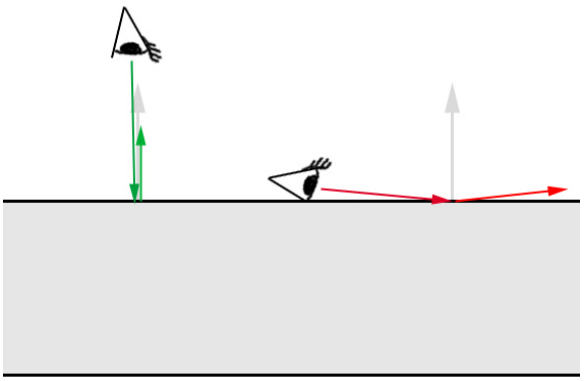
- -1: the base rotation follows the local object coordinate system.
- -2: the base rotation follows the bump basis vectors
- -3: the base rotation follows the surface derivatives
- -4: the base rotation follows a vector placed in `state>tex` prior to calling *mia_material*

See also "brushed metal" on page 55 in the tips section.

1.6.6 BRDF

As explained in the introduction on page 8 the materials reflectivity is ultimately guided by the *incident angle* from which it is viewed.

[10]Note that deriving the anisotropy from texture space only creates one space per triangle and may cause visible seams between triangles.



0 degree (green) and 90 degree (red) view angles

There are two modes to define this BRDF curve:

The first mode is "by IOR", i.e. when **brdf_fresnel** is *on*. How the reflectivity depends on the angle is then solely guided by the materials IOR. This is known as *Fresnel reflections* and is the behavior of most dielectric materials such as water, glass, etc.

The second mode is the manual mode, when **brdf_fresnel** is *off*. In this mode the **brdf_0_degree_refl** parameter defines the reflectivity for surfaces directly facing the viewer (or incident ray), and **brdf_90_degree_refl**defines the reflectivity of surfaces perpendicular to the viewer. The **brdf_curve** parameter defines the falloff of this curve.

This mode is used for most hybrid materials or for metals. Most material exhibit strong reflections at grazing angles and hence the **brdf_90_degree_refl** parameter can generally be kept at 1.0 (and using the **reflectivity** parameter to guide the overall reflectivity instead). Metals tend to be fairly uniformly reflective and the **brdf_0_degree_refl**value is high (0.8 to 1.0) but many other layered materials, such as linoleum, lacquered wood, etc. has lower **brdf_0_degree_refl** values (0.1 - 0.3).

See the tips on page 45 for some guidelines.

1.6.7 Special Effects

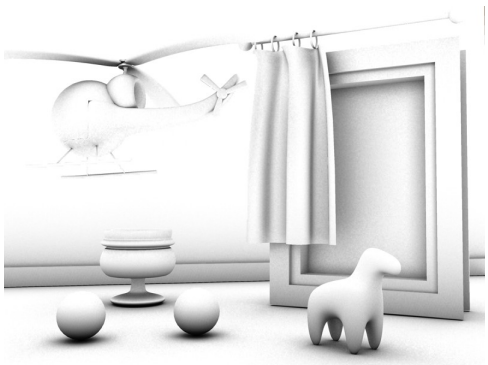
1.6.7.1 Built-in Ambient Occlusion

The built in Ambient Occlusion (henceforth shortened to "AO") can be used in two ways. Either it is used to enhance details and "contact shadows" in indirect illumination (in which case there must first *exist* some form of indirect illumination in the first place), or it is used together with a specified "ambient light" in a more traditional manner. Hence, if neither

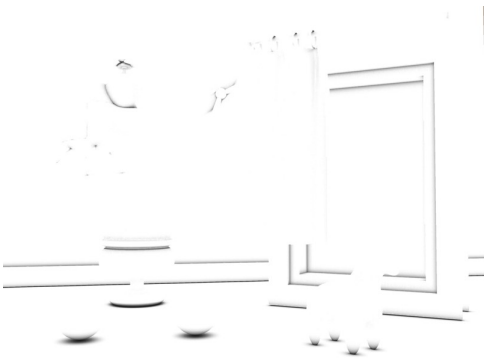
indirect light exists, nor any "ambient light" is specified, *the AO will have no effect* [11].

The **ao_samples** sets the number of samples (rays) shot for creating the AO. Higher value is smoother but slower, lower values faster but grainier. 16 is the default and 64 covers most situations.

The **ao_distance** parameter defines the radius within which occluding objects are found. Smaller values restrict the AO effect only to small crevices but are much faster to render. Larger values cover larger areas but render slower. The following images illustrate the raw AO contribution with two different distances:



Larger distance



Smaller distance

As mentioned in the introduction on page 11 the AO can be used for "detail enhancement" of indirect illumination. This mode is enabled by setting **ao_do_details** to 1.

This mode is used to apply short distance AO multiplying it with the existing *indirect illumination* (Final Gathering or GI/photons), bringing out small details.

Study this helicopter almost exclusively lit by indirect light:

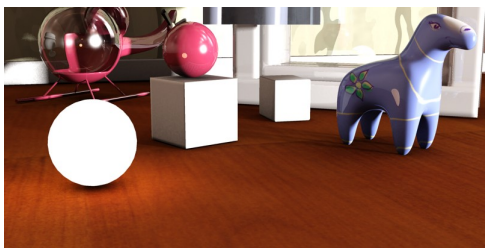
Without AO

With AO

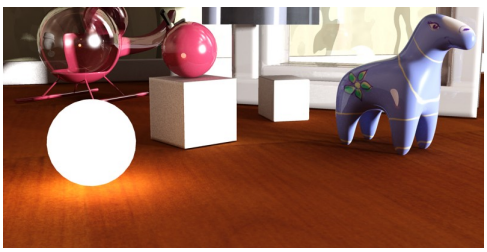
[11] Sometimes people use AO as a general multiplier to *all* diffuse light. This has the distinct drawback of affecting even brightly *directly* lit areas with "AO shadows", which can look wrong. This use is not covered by the built in AO shader because it is trivially achieved by simply applying the **mib_amb_occlusion** shader to the diffuse color of the material and putting the materials original color into it's **Bright** parameter.

Note how the helicopter does not feel "grounded" in the left image and the shadows under the landing skids are far too vague. The right image uses AO to "punch out" the details and the contact shadows.

One can also set **ao_do_details** to 2, which enables a more sophisticated AO mode **new in mental ray 3.6**. Instead of doing simple *occlusion*, which can only add "darkness" of varying degree, the shader will actually look at the color of the surrounding objects, and use that color rather than "darkness". Since this involves shading each of the points hit, this is not as fast as pure AO, but it has the additional effect of resolving both *bright* and *dark* details.



***ao_do_details** = 1*



***ao_do_details** = 2*

The image on the left illustrates the problem with the traditional AO; it applies to all indirect illumination and always makes it *darker*. It is most noticable on the glowing sphere (which has a dark spot under it) but can also be perceived on the floor in front of the cube which is suspiciously dark, even though the cube is strongly lit on the front, as well as between the legs of the horse and the underside of the red sphere.

In contrast, the image on the right is using **ao_do_details**=2 for all materials, and now the floor is correclty lit by the glowing ball, there is a hint of white bounce-light on the floor from the cube, there is light between the legs of the horse, and on the underside of the red ball.

If you find that using AO creates a "dirty" look with excessive darkening in corners, or dark rims around self-illuminated objects, try to set **ao_do_details** to 2 for a more accurate result.

The **ao_dark** parameter sets the "darkness" of the AO shadows. It is used as the multiplier value for completely occluded surfaces. In practice this means: A black color will make the AO effect very dark, a middle gray color will make the effect less noticeable (brighter) etc. When the new **ao_do_details** mode 2 is used, it instead sets the "blend" between the color picked up from nearby objects and "darkness". The blend is:

$(1 - \textbf{ao\_dark}) * (objectcolors) + black * \textbf{ao\_dark}$.

The **ao_ambient** parameter is used for doing more "traditional" AO, i.e. supplying the imagined "ever present ambient light" that is then attenuated by the AO effect to create shadows.

While "traditional AO" is generally used when rendering *without* other indirect light, it can also be combined with existing indirect light. One needs to keep in mind that this magical "ever present ambient light" is inherently non-physical, but may perhaps help lighten some troublesome dark corners.

1.6.8 Advanced Rendering Options

1.6.8.1 Reflection Optimization Settings

These parameters define some performance boosting options for reflections.

refl_falloff_dist allows limiting reflections to a certain distance, which both speeds up rendering as well as avoiding pulling in distant objects into extremely glossy reflections.

If **refl_falloff_color** is enabled and used, reflections will fade to this color. If it is not enabled, reflections will fade to the environment color. The former tends to be more useful for indoor scenes, the latter for outdoor scenes.

Full reflections (left), fading over 100mm (center) or 25mm (right)

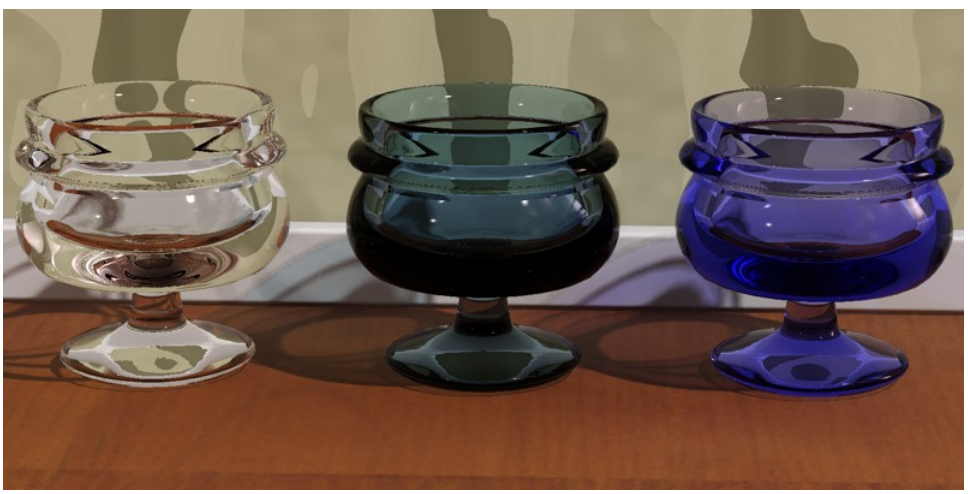
Each material can locally set a maximum trace depth using the **refl_depth** parameter. When this trace depth is reached the material will behave as if the **refl_hl_only** switch was enabled, i.e. only show highlights and "emulated" reflections. If **refl_depth** is zero, the global trace depth is used.

refl_cutoff is a threshold at which reflections are rejected (not traced). It's a relative value, i.e. the default of 0.01 means that rays that contribute less than 1% to the final pixel are ignored.

1.6.8.2 Refraction Optimization Settings

The optimization settings for refractions (transparency) are nearly identical to those for reflections. The exception is that of **refr_falloff_color** which behaves differently.

- When **refr_falloff_dist** is used, and **refr_falloff_color** is *not* used, transparency rays will fade to *black*. This is like smoked glass or highly absorbent materials. Transparency will just completely *stop* at a certain distance. This has the same performance advantage as using the **refl_falloff_dist** for reflections, i.e. tracing shorter rays are much faster.
- However, when **refr_falloff_color** *is* used, it works differently. The material will then make physically correct absorption. Exactly at the distance given by **refr_falloff_dist** will the refractions have the color given by **refr_falloff_color** - but the rays are *not* limited in reach. At twice the distance, the influence of **refr_falloff_color** is double, at half the distance half, etc.



No limit (left), fade to black (center), fade to blue (right)

The leftmost cup has no fading. The center cup has **refr_falloff_color** off, and hence fades to black, which also includes the same performance benefits of limiting the trace distance as when used for reflections.

The rightmost cup, however, fades to a *blue* color. This causes proper exponential attenuation in the material, such that the thicker the material, the deeper the color. See page 46 for a discussion about realistic colored glass.

Note: To render proper shadows when using **refr_falloff_dist** one must use ray traced shadows, and the shadow mode must be set to *segment*. See the mental ray manual on shadow modes.

Each material can locally set a maximum trace depth using the **refl_depth** parameter. When this trace depth is reached, the material returns a *black* refraction. Most other transparency/glass shaders return the environment, which can create very odd results when rendering an indoor rendering with an extremely bright outdoor environment, and bright areas appear in glass objects in dark cupboards that suddenly refract some sky. If **refl_depth** is zero, the global trace depth is used, and the environment *is* returned, rather than black.

refl_cutoff works identical to the reflection case described above.

1.6.8.3 Options

The options contain several on/off switches that control some of the deepest details of the material:

The **thin_walled** decides if a material causes refractions (i.e. behaves as if it is made of a solid transparent substance) or not (i.e. behaves as if made of wafer-thin sheets of a transparent material). This topic is discussed in more detail on page 10.

Solid (left) and Thin-walled (right)

The **do_refractive_caustics** parameter defines how glass behaves when *caustics* are enabled.

When not rendering caustics, the *mia_material* uses a shadow shader to create transparent shadows. For objects such as window panes this is perfectly adequate, and actually creates a better result than using caustics since the direct light is allowed to pass (more or less) undisturbed through the glass into e.g. a room.

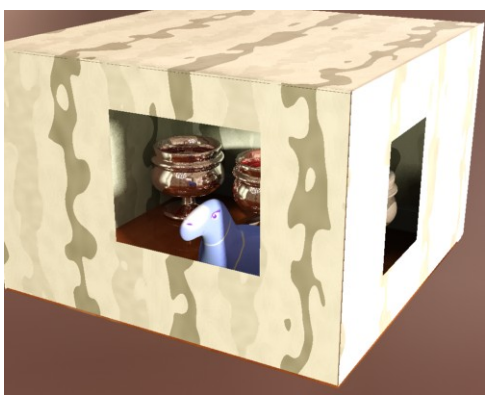
Traditionally, enabling caustics in *mental ray* cause all materials to stop casting transparent shadows and instead start to generate refractive caustics. In most architectural scenes this is undesirable; one may very well want a glass decoration on a table to generate caustic effect, but still want the windows of the room to let in quite normal direct light. This switch makes this possible on the material level.

Using transparent shadows

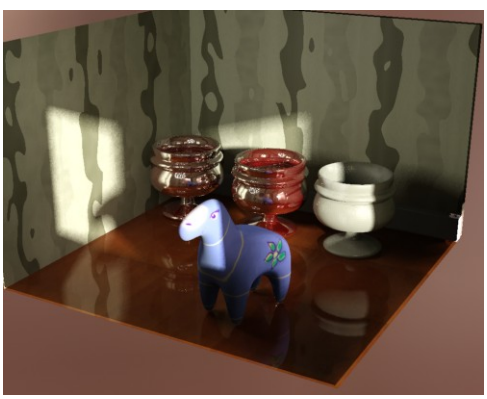
Using refractive caustics

The left image shows the result that happen when **do_refractive_caustics** is off, the right the result when it is on. Both modes can be freely mixed within the same rendering. Photons are automatically treated accordingly by the built in photon shader, shooting straight through as direct light in the former case, and being refracted as caustics in the latter.

The **backface_cull** switch enables a special mode which makes surfaces completely invisible to the camera when seen from the reverse side. This is useful to create "magic walls" in a room. If all walls are created as planes with the normal facing inwards, the **backface_cull** switch allows the room to be rendered from "outside". The camera will see into the room, but the walls will still "exist" and cast shadows, bounce photons, etc. while being magically "see through" when the camera steps outside.



No Back-face Culling



Back-face Culling on the walls

The **propagate_alpha** switch defines how transparent objects treats any alpha channel information in the background. When on, refractions and other transparency effects will propagate the alpha of the background "through" the transparent object. When off, transparent objects will have an opaque alpha.

The **no_visible_area_hl** parameter concerns the behavior of *visible* area lights.

Keep in mind that traditional "highlights" (i.e. specular effects) is a computer graphics "trick" in place of actually creating a glossy reflection of an actual visible light-emitting surface.

However, *mental ray* area lights can be visible, and when they are visible they will reflect in any (glossy) reflective objects. If both the reflection of the visible area light *and* the highlight is rendered, the light is added twice, causing an unrealistic brightening effect. This switch (which defaults to on) causes visible area lights to loose their "highlights" and instead only appear as reflections[12].

hl_vs_refl_balance modifies the balance between the intensity of the highlight and the intensity of reflections. The default value of 1.0 is the "as close to physically correct as possible" value. This parameter allows tweaking this default value where values above 1.0 makes the highlight stronger, and below 1.0 weaker.

A final optimization switch (also on by default) is the **skip_inside_refl** checkbox. Most reflections on the *insides* of transparent objects are very faint, except in the special case that occurs at certain angles known as "Total Internal Reflection" (TIR). This switch saves rendering time by ignoring the weak reflections completely but retaining the TIR's.

The **indirect_multiplier** allows tweaking of how strongly the material responds to indirect light, and **fg_quality** is a local multiplier for the number of final gather rays shot by the material. Both default to 1.0 which uses the global value.

To aid in mapping textures to **fg_quality** the additional **fg_quality_w**parameter exists. When zero, **fg_quality** is the raw quality setting, but for a nonzero **fg_quality_w** the actual quality used is the product of the two values, with a minimum of 1.0. This means that with a color texture mapped to **fg_quality** and **fg_quality_w** set to 5.0, black in the texture results in a quality of 1.0 (i.e. the number of final gather rays shot is the global default), and white in the texture in a quality of 5.0 (five times as many rays are shot).

1.6.9 Interpolation

Glossy reflections and refractions can be *interpolated*. This means they render faster and become smoother.

Interpolation works by pre-calculating glossy reflection in a grid across the image. The number of samples (rays) taken at each point is govern by the **refl_samples** or **refr_samples** parameters just as in the non-interpolated case. The resolution of this grid is set by the **intr_grid_density** parameter.

However, interpolation can cause *artifacts*. Since it is done on a low resolution grid, it can lose details. Since it blends neighbors of this low resolution grid it can cause over-smoothing. For this reason it is primarily useful on flat surfaces. Wavy, highly detailed surfaces, or surfaces using bump maps will not work well with interpolation.

Valid values for **intr_grid_density** parameter is:

- 0 = grid resolution is double that of the rendering

[12]Naturally this does not apply to the **refl_hl_only** mode, since it doesn't actually reflect anything.

- 1 = grid resolution is same as that of the rendering
- 2 = grid resolution is half of that of the rendering
- 3 = grid resolution is a third of that of the rendering.
- 4 = grid resolution is a fourth of that of the rendering.
- 5 = grid resolution is a fifth of that of the rendering.

Within the grid data is stored and shared across the points. Lower grid resolutions is faster but lose more detail information. Both reflection and refraction has an **intr_refl_samples** parameter which defines how many stored grid points (in an N by N group around the currently rendered point) is looked up to smooth out the glossiness. The default is 2, and higher values will "smear" the glossiness more, but are hence prone to more overmoothing artifacts.

No interpolation (left), looking up 2 points (center) and 4 points (right)

The reflection of the left cup in the floor is not using interpolation, and one can perceive some grain (here intentionally exaggerated). The floor tiles under the other two cup uses a half resolution interpolation with 2 (center) and 4 (right) point lookup respectively.

This image also illustrates one of the consequences of using interpolation: The foot of the left cup, which is near the floor, is reflected quite sharply, and only parts of the cup far from the floor are blurry. Whereas the interpolated reflections on the right cups have a certain "base level" of blurriness (due to the smoothing of interpolation) which makes even the closest parts somewhat blurry. In most scenes with weak glossy reflections this discrepancy will never be noticed, but in other cases this can make things like legs of tables and chairs feel "unconnected" with a glossy floor, if the reflectivity is high.

To solve this the **intr_refl_ddist** parameter exists. It allows a second set of detail rays to be traced to create a "clearer" version of objects within that radius.

No detail distance (left), 25mm detail distance (center) and 150mm detail distance (right)

All three floor tiles use interpolation but the rightmost two use different distances for the "detail distance".

This also allows an interesting "trick": Set the **refl_samples** to 0, which renders reflections as if they were mirror-perfect but use the interpolation to introduce blur into this "perfect" reflection (and perhaps use the **intr_refl_ddist** to make nearby parts less blurry). This is an extremely fast way to obtain a glossy reflection.

No detail distance (left), with detail distance (right)

The above floor tiles are rendered with mirror reflections, and the "blurriness" comes solely from the interpolation. This renders as fast (or faster!) than pure mirror reflections, yet gives a satisfying illusion of true glossy reflections, especially when utilizing the **intr_refl_ddist** as on the right.

1.6.10 Special Maps

The *mia_material* also supports the following special inputs:

1.6.10.1 Bump Mapping

The **bump** parameter accepts a shader that perturbs the normal for bump mapping. This parameter is only used it the new **bump_mode** parameter is *zero*.

When **no_diffuse_bump** is *off*, the bumps apply to all shading components (diffuse, highlights, reflections, refractions...). When it is *on*, bumps are applied to all component except the diffuse. This means bumps are seen in reflections, highlights, etc. but the diffuse shading shows no bumps. It is as if the materials diffuse surface is smooth, but covered by a bumpy lacquer coating.

***no_diffuse_bump** is off (left) and on (right)*

In *mia_material_x* there are also three new parameters related to bump mapping: two vector bump inputs, **overall_bump** and **standard_bump**, and a **bump_mode** parameter defining the coordinate-space of those vectors. The shaders put into **overall_bump**or **standard_bump** should return a *vector*, but it is also legal for those shaders to modify the normal vector themselves and return (0,0,0).

overall_bumpdefines an overall bump that *always applies* both to the diffuse and the specular component at all times, regardless of the setting of **no_diffuse_bump**˙**standard_bump** is a vector equivalent of the old **bump** parameter, in that it applies globally when **no_diffuse_bump** is *off*, and only to the specular/reflection "layer" when **no_diffuse_bump** is *on*. However, the **standard_bump** is added "on top of" the **overall_bump** result.

The intended use is to put the *mia_roundcorners* shader in **overall_bump**and your normal bump shader into **standard_bump**. This way, the "round corners" effect will apply both to the diffuse and specular component irregardless of the setting of **no_diffuse_bump**.

The **bump_mode** parameter defines the coordinate space of the vectors, and if they are additive or not. The following values are legal:

- 0: compatible mode. The old **bump** parameter is used in place of **overall_bump** and **standard_bump**.
- 1: "add" mode in "internal" space
- 2: "add" mode in world space
- 3: "add" mode in object space
- 4: "add" mode in camera space
- 5: "set" mode in "internal" space
- 6: "set" mode in world space
- 7: "set" mode in object space
- 8: "set" mode in camera space

The "add" modes mean that the vector should contain a *normal perturbation*, i.e. a modification that is "added" to the current normal. Whereas "set" mode means that the actual normal is *replaced* by the incoming vector, interpreted in the aforementioned coordinate space.

This new scheme makes the *mia_material_x* bump mapping compatible with more mental ray integrations, as well as allows the round corners to be applied even if **no_diffuse_bump** is on.

1.6.10.2 Cutout Opacity and Additional Color

The **cutout_opacity** is used to apply an opacity map to completely remove parts of objects. A classic example is to map an image of a tree to a flat plane and use opacity to cut away the parts of the tree that are not there.

Mapping the transparency (left) vs. ***cutout_opacity*** *(right)*

The **additional_color** is an input to which one can apply any shader. The output of this shader is simply added on top of the shading done by the *mia_material* and can be used both for "self illumination" type effects as well as adding whatever additional shading one may want.

The material also supports standard **displacement** and **environment** shaders. If no **environment** is supplied, the global camera environment is used.

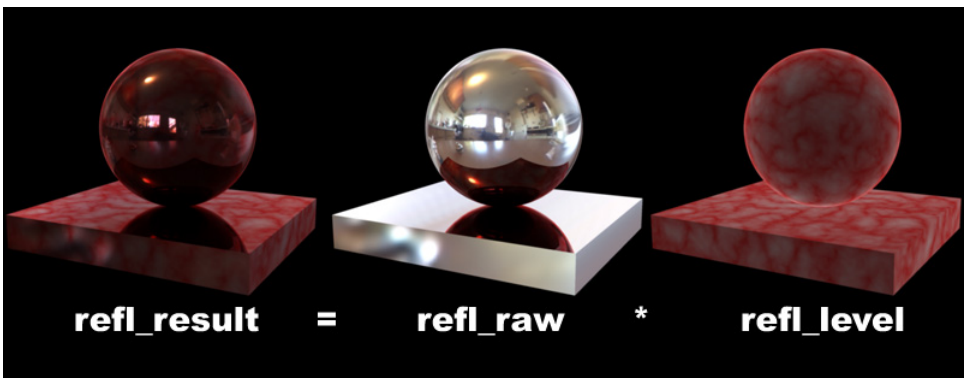
1.6.11 Multiple Outputs of *mia_material_x*

1.6.11.1 Introduction

Here follows a detailed listing of the available outputs of *mia_material_x*:

Most of the outputs follow the pattern of xxx_**result**, xxx_**raw** and xxx_**level**. The "result" is the final contribution, "raw" is the un-scaled contribution, and "level" is the scaling. The "level" is often related to an input parameter (or combinations thereof), and has been modified to abide by the energy conservation feature of the material.

Unless otherwise noted, it is true that $xxx\_result = xxx\_raw * xxx\_level$.



The different outputs and their relationship

Hence the outputs contain some redundancy; if one just wants the "current reflections" in a separate channel, use **refl_result**, but if one wants more control over the amount of reflections in post production, one can instead use **refl_raw** and **refl_level**, multiplying them in the compositing phase prior to adding them to the final color.

Be aware, though, that *mia_material_x* will *intentionally* sample reflections that has a very low level in the actual rendering phase at low quality (for performance), so doing *huge* modifications to reflection intensity in post should be avoided.

1.6.11.2 List of All Outputs

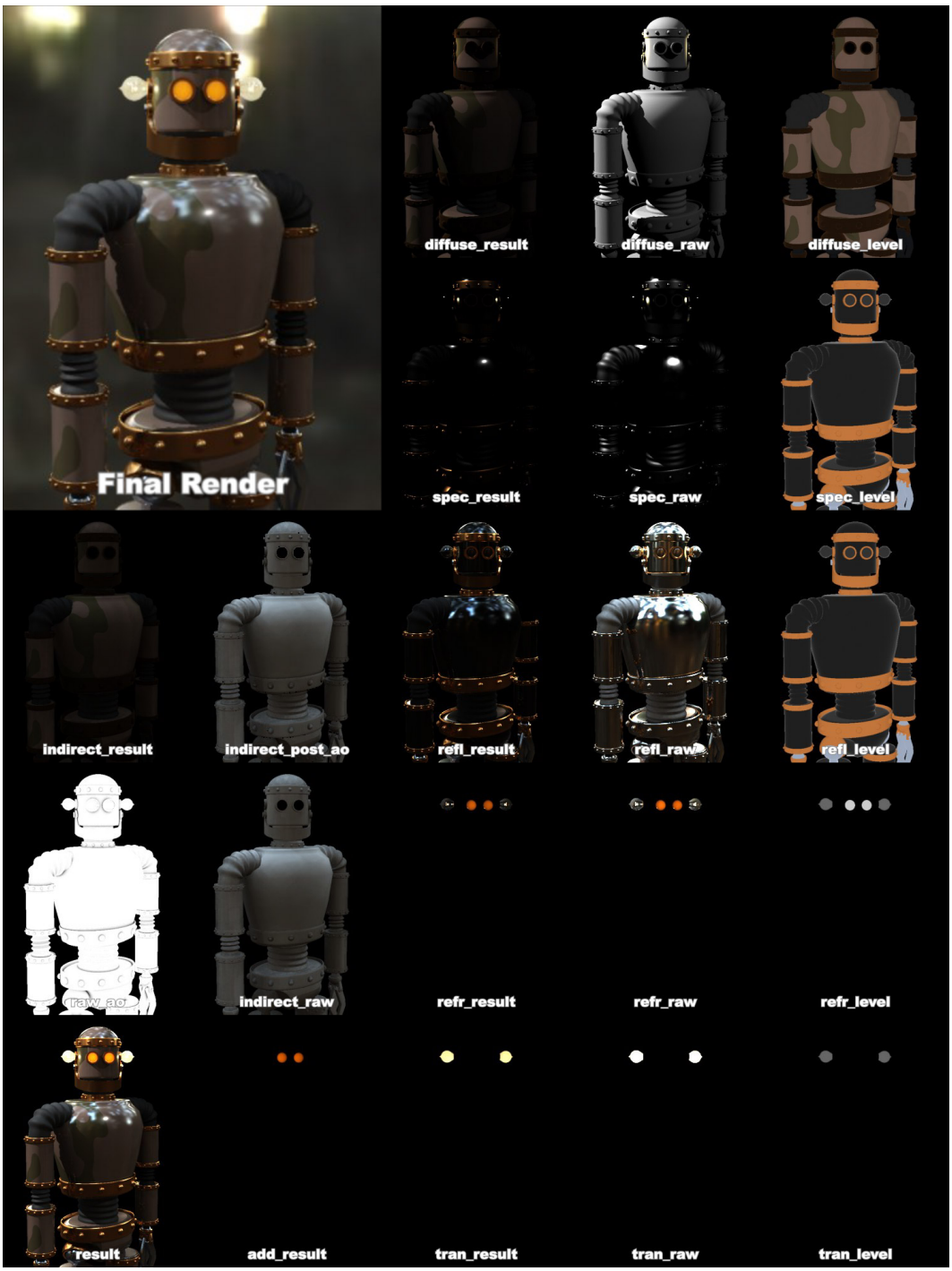
The following outputs exist:

- **result** is the main, blended output, i.e. the "beauty" pass. It is identical to the single output of *mia_material*. If the "safety" parameter **multiple_outputs** is off, *no other output except this one is ever written to* [13].

[13] To easier support using *mia_material_x* in phenomena and other places where formerly *mia_material* was

- **diffuse_result** is the resulting diffuse component after lighting, including textures, **diffuse_raw** is the diffuse lighting itself, without textures, and **diffuse_level** is the diffuse texture color *adjusted by the energy conservation*.
- **spec_result** is the resulting specular component, **spec_raw** is the raw un-attenuated highlight, and **spec_level** is the specular level, which is the same as **refl_level** if the input parameter **hl_vs_refl_balance** is 1.0.
- **refl_result** is the resulting reflection component, **refl_raw** is the raw (full intensity) reflection, and **refl_level** is the actual reflectivity, including reflection color and BRDF (or fresnel) curve attenuation effects.
- **refr_result** is the resulting refraction (transparency) component, **refr_raw** is the raw full intensity transparency, and **refr_level** is the actual transparency level, which has been adjusted by the energy conservation.
- **tran_result** is the resulting translucency component, **tran_raw** is the raw translucency, and **tran_level** the actual translucency level, adjusted by the energy conservation.
- **indirect_result** is the resulting indirect illumination including ambient occlusion effects and multiplying by the diffuse color, **indirect_raw** is the raw result from **mi_compute_avg_radiance()**, **indirect_post_ao** is the indirect illumination affected by AO but without being multiplied by the diffuse color, and **ao_raw** is the raw contribution of the AO.
- **add_result** is a straight passthrough of the **add_color** parameter.
- **opacity_result** contains the final contribution of any background of the object as a result of the input **cutout_opacity** being less than 1.0. **opacity_raw** contains the background 'without scaling by the opacity. These outputs will contain black if **cutout_opacity** is 1.0 since no actual transparency ray is ever traced in that case! The **opacity** output contains the actual opacity itself. Care must be taken if **opacity** equals zero, because this mean that *the material has performed no shading whatsoever* and *none* of the other outputs will contain any value at all!

used, the **multiple_outputs** parameter exists. If this is set to off, no other parameter than **result** is written to (leaving the others unmodified and hence undefined). This also makes it safe to supply *mia_material_x* to parameters of type "shader" which only expect a single color return value.



An example of outputs

1.6.11.3 Proper Compositing

Due to the redundancy available in the outputs, there are several ways to composite them together to yield the same result as the beauty render. Here we outline two compositing pipelines in equation form.

First we have the "simple" variant, which is simply a sum of the various **result** parameters. This version allows only minor post production changes to the overall balance between the materials.

But it has the advantage of not needing as many files, as well as working reasonably well in non-floating-point compositing.

```
Beauty = diffuse_result + indirect_result + spec_result +
            refl_result + refr_result + tran_result +
            add_result
```

Then we have the more "complex" variant which uses the various *raw* and *level* outputs, which allows much greater control in post production.

Note that the *raw* outputs needs to be stored and composited in floating point to maintain the dynamic range. The *level* outputs always stay in a 0.0-1.0 range and does not require floating point storage.

```
Beauty = diffuse_level * (diffuse_raw + (indirect_raw * ao_raw)) +
            spec_level * spec_raw +
            refl_level * refl_raw +
            refr_level * refr_raw +
            tran_level * tran_raw +
            add_result
```

1.7 Tips and Tricks

1.7.1 Final Gathering Performance

The Final Gathering algorithm in mental ray 3.5 is vastly improved from earlier versions, especially in it's *adaptivity*. This means one can often use much lower ray counts and much lower densities than in previous versions of mental ray.

Many stills can be rendered with such extreme settings as 50 rays and a density of 0.1 - if this causes "over-smoothing" artifacts, one can use the built in AO (see page 28) to solve those problems.

When using Final Gathering together with GI (photons), make sure the photon solution is fairly "smooth" by rendering with Final Gathering disabled first. If the photon solution is noisy, increase the photon search radius until it "calms down", and then re-enable Final Gathering.

1.7.2 Quick Guide to some Common Materials

Here are some quick rules-of-thumb for creating various materials. They each assume basic default settings as a starting point.

1.7.2.1 General Rules of Thumb for Glossy Wood, Flooring, etc.

This is the kind of "hybrid" materials one run into in many architectural renderings; lacquered wood, linoleum, etc.

For these materials **brdf_fresnel** should be *off* (i.e. we define a custom BRDF curve). Start out with **brdf_0_degree_refl** of 0.2 amd **brdf_90_degree_refl** of 1.0 and apply some suitable texture map to the **diffuse**. Set **reflectivity** around 0.5 to 0.8.

How glossy is the material? Is reflections very clear or very blurry? Are they Strong or Weak?

- For clear, fairly strong reflections, keep **refl_gloss** at 1.0
- For slightly blurry but strong reflections, set a lower **refl_gloss** value. If performance becomes an issue try using **refl_interpolate**.
- For slightly blurry but also very *weak* reflections one can often "cheat" by setting a lower **refl_gloss** value (to get the broader highlights) but set **refl_samples** value to 0. This shoots only one mirror ray for reflections - but if they are very weak, one can often not really tell.
- For *medium* blurry surfaces set an even lower **refl_gloss** and maybe increase the **refl_samples**. Again, for performance try **refl_interpolate**.

- For *extremely* blurry surfaces or surfaces with very weak reflections, try using the **refl_hl_only** mode.

A typical wooden floor could use **refl_gloss** of 0.5, **refl_samples** of 16, **reflectivity** of 0.75, a nice wood texture for **diffuse**, perhaps a slight bump map (try the **no_diffuse_bump** checkbox if bumpiness should appear only in the lacquer layer).

A linoleum carpet could use the same but with a different texture and bump map, and probably with a slightly lower **reflectivity**. and **refl_gloss**.

1.7.2.2 Ceramics

Ceramic materials are *glazed*, i.e. covered in a thin layer of transparent material. They follow similar rules to the general materials mentioned above, but one should have **brdf_fresnel** *on* and the **refr_ior** set at about 1.4 and **reflectivity** at 1.0.

The **diffuse** should be set to a suitable texture or color, i.e. white for white bathroom tiles.

1.7.2.3 Stone Materials

Stone is usually fairly matte, or has reflections that are so blurry they are nearly diffuse. The "powdery" character of stone is simulated with the **diffuse_roughness** parameter - try 0.5 as a starting point. Porous stone such as bricks would have a higher value.

Stone would have a very low **refl_gloss** (lower than 0.25) and one can most likely use **refl_hl_only** to good effect for very good performance. Use a nice stone texture for **diffuse**, some kind of bump map, and perhaps a map that varies the **refl_gloss** value.

The **reflectivity** would be around 0.5-0.6 with **brdf_fresnel** off and **brdf_0_degree_refl** at 0.2 and **brdf_90_degree_refl** at 1.0

1.7.2.4 Glass

Glass is a dielectric, so **brdf_fresnel** should definitely be *on*. The IOR of glass is around 1.5. Set **diffuse_weight** to 0.0, **reflectivity** to 1.0 and **transparency** to 1.0. This is enough to create basic, completely clear refractive glass.

If this glass is for a window pane, set **thin_walled** to *on*. If this is a solid glass block, set **thin_walled** to *off* and consider if caustics are necessary or not, and set **do_refractive_caustics** accordingly.

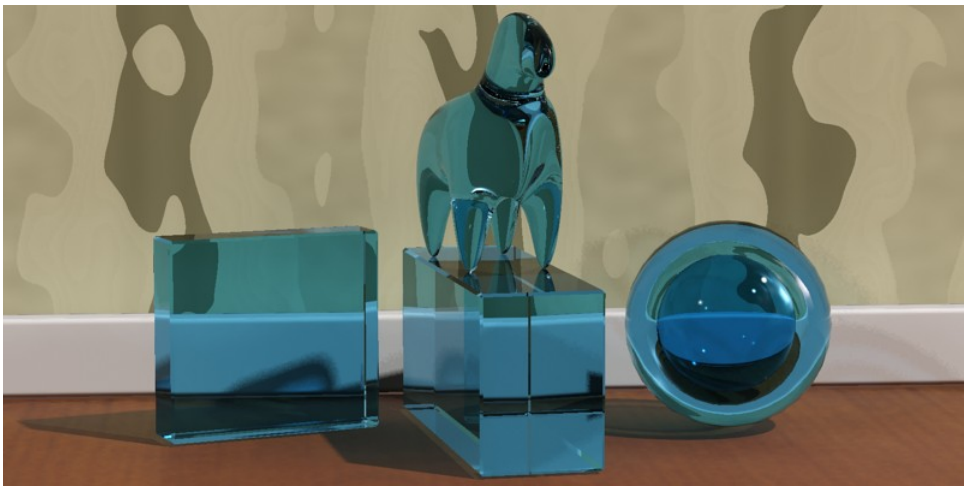
Is the glass frosted? Set **refr_gloss** to a suitable value. Tune the **refr_samples**for good quality or use **refr_interpolate** for performance.

1.7.2.5 Colored Glass

For clear glass the tips in the previous section work. But *colored* glass is a slightly different story.

Many shaders set the transparency at the *surface* of the glass. And indeed this is what happens if one simply sets a **refr_color** to some value, e.g. blue. For glass done with **thin_walled** turned *on* this works perfectly. But for solid glass objects this is not an accurate representation of reality.

Study the following example. It contains two glass blocks of very different size and a sphere with a spherical hole inside of it[14] plus a glass horse.



*With a blue **refr_color**: Glass with color changes at the surface*

The problems are evident:

- The two glass blocks are of completely different thickness, yet they are exactly the same level of blue.
- The inner sphere is *darker* than the outer.

Why does this happen?

Consider a light ray that enters a glass object. If the color is "at the surface", the ray will be colored somewhat as it enters the object, retain this color through the object, and receive a second coloration (attenuation) when it exits the object:

[14] Created by inserting a second sphere with the normals flipped inside the outer sphere. Don't forget to flip normals of such surfaces or they will not render correctly!

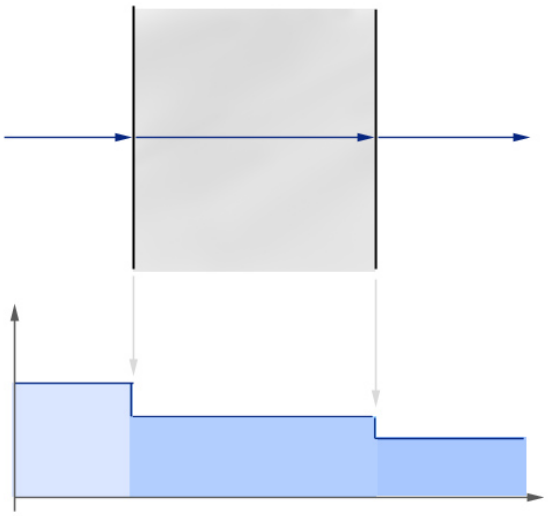
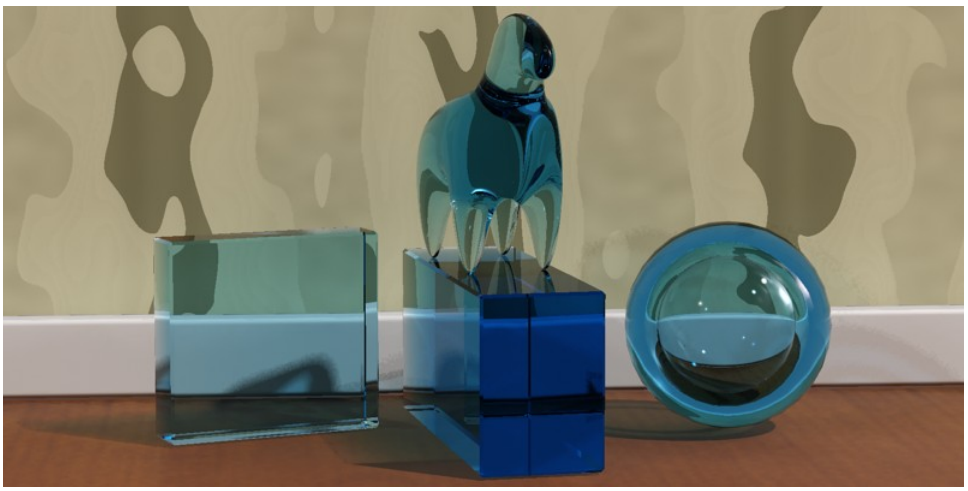


Diagram for glass with color changes at the surface

In the illustration above the ray enters from the left, and at the entry surface it drops in level and gets slightly darker (bottom of graph schematically illustrates the level). It retains this color throughout the travel through the medium and drops in level again at the exit surface.

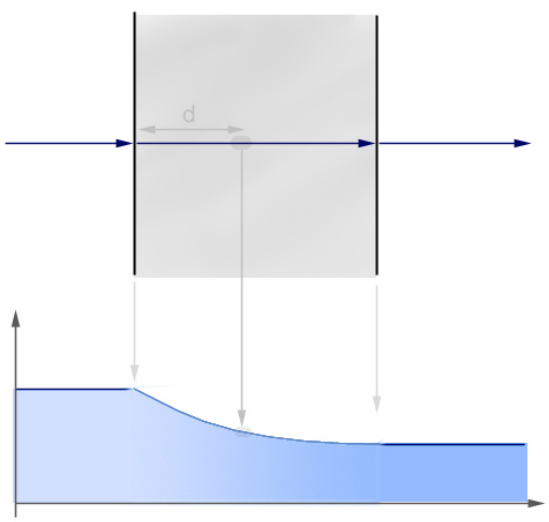
For simple glass objects this is quite sufficient. For any glass using **thin_walled** it is by definition the correct thing to do, but for any complex solid it is not. It is *especially* wrong for negative spaces inside the glass (like the sphere in our example) because the light rays have to travel through four surfaces instead of two (getting two extra steps of "attenuation at the surface")

In real colored glass, light travels *through* the medium and is attenuated "as it goes". In the *mia_material*this is accomplished by enabling the **refr_falloff_dist** and use the **refr_falloff_color** and setting the **refr_color** to white. This is the result:



Glass with color changes within the medium

The above result is clearly much more satisfactory; the thick glass block is much deeper blue than the thin one, and the hollow sphere now looks correct. In diagram form it looks as follows:



*d=**refr_falloff_dist** where attenuation is **refr_falloff_color***

The ray enters the medium and during it's entire travel it is attenuated. The strength of the attenuation is such that precisely at the **refr_falloff_dist** (d in the figure) the attenuation will match that of **refr_falloff_color** (i.e. at this depth the attenuation is the same as was received immediately at the surface with the previous model). The falloff is exponential such that at double **refr_falloff_dist** the effect is that of **refr_falloff_color** squared, and so on.

There is one minor trade off:

To correctly render the *shadows* of a material using this method one must either use caustics *or* make sure *mental ray* is rendering shadows in "segment" shadow mode.

Using caustics naturally gives the most correct looking shadows (the above image was not rendered with caustics), but will require that one has caustic photons enabled and a physical light source that shoots caustic photons.

On the other hand, the *mental ray* "segment" shadows have a slightly lower performance than the more widely used "simple" shadow mode. But if it is not used, there shadow intensity will not take the attenuation through the media into account properly[15].

1.7.2.6 Water and Liquids

Water, like glass, is a *dielectric* with the IOR of 1.33. Hence, the same principles as for glass (above) applies for solid bodies of water which truly need to refract things... for example water running out of a tap. Colored beverages use the same principles as colored glass, etc.

Water into Wine

To create a beverage in a container as in the image above, it is important to understand how the *mia_material* handles refraction through multiple surfaces vs. how the "real world" tackles the same issue.

What is important for refraction is how the transition from one medium to another with a different IOR. Such a transition is known as an *interface*.

[15] But it could potentially still look "nice".

For lemonade in a glass, imagine a ray of light travelling through the air (IOR = 1.0) enter the glass, and is refracted by the IOR of the glass (1.5). After travelling through the glass the ray leaves the glass and enters the liquid, i.e. it passes an *interface* from one medium of IOR 1.5 to another medium of IOR 1.33.

One way to model this in computer graphics is to make the glass one separate closed surface, with the normals pointing towards the inside of the glass and an IOR of 1.5, and a second, closed surface for the beverage, with the normals pointing inwards and an IOR of 1.33, and leaving a small "air gap" between the container and the liquid.

While this "works", there is one problem with this approach: When light goes from a higher IOR to a lower there is a chance of an effect known as "Total Internal Reflection" (TIR). This is the effect one sees when diving in a swimming pool and looking up - the objects above the surface can only be seen in a small circle straight above, anything below a certain angle only shows a reflection of the pool and things below the surface. The larger the difference in the IOR of the two media, the larger is the chance of TIR.

So in our example, as the ray goes from glass (IOR=1.5) to air, there is a large chance of TIR. But in *reality* the ray would move from a medium of IOR=1.5 to one of IOR=1.33, which is a much smaller step with a much smaller chance of TIR. This will look different:

Correct refraction (left) vs. the "air gap" method (right)

The result on the left is the correct result, but how it is obtained?

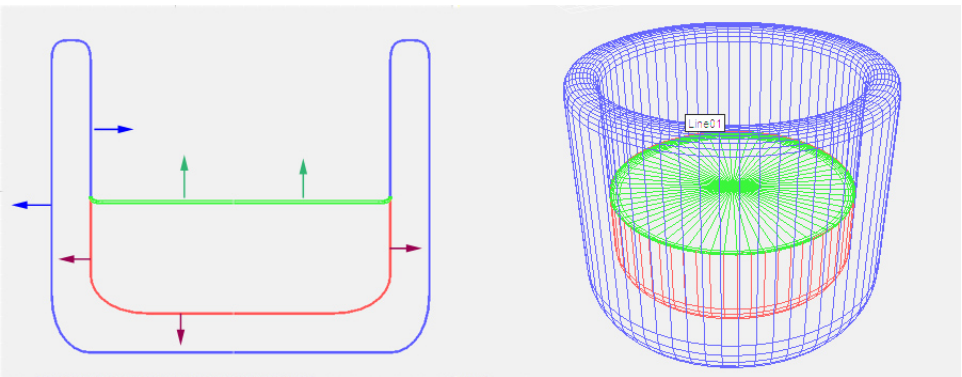
The solution to the problem is to rethink the modeling, and not think in terms of media, but in terms of *interfaces*. In our example, we have three different interfaces, where we can consider the IOR as the ratio between the IOR's of the outside and inside media:

- Air-Glass interface (IOR = 1.5/1.0 = 1.5)
- Air-Liquid interface (IOR = 1.33/1.0 = 1.33)

- Glass-Liquid interface (IOR=1.33/1.5=0.8)

It is evident that in the most common case of an interface with air, the IOR to use is the IOR of the media (since the IOR of air is 1.0), whereas in an *interface* between two different media, the situation is different.

To correctly model this scenario, we then need *three* surfaces, each with a separate *mia_material* applied:



The three interfaces *for a liquid in a glass*

- The Air-glass surface (blue), with normals pointing *out of* the glass, covering the area where air directly touches the glass, having an IOR of 1.5
- The Air-liquid surface (green), with normals pointing *out of* the liquid, covering the area where air directly touches the liquid, having an IOR of 1.33
- The Glass-liquid surface (red), with normals pointing *out of* the liquid, covering the area where the glass touches the liquid, having an IOR of 0.8

By setting a suitable **refr_falloff_dist** and **refr_falloff_color** for the two liquid materials (to get a colored liquid), the image on the left in the comparison above is the result.

1.7.2.7 The Ocean and Water Surfaces

A water *surface* is a slightly different matter than a visibly transparent liquid.

The ocean isn't blue - it is *reflective*. Not much of the light that goes down under the surface of the ocean gets anywhere of interest. A little bit of it is scattered back up again doing a little bit of very literal "sub surface scattering".

To make an ocean surface with the *mia_material* do the following steps:

Set **diffuse_weight** to 0.0, **reflectivity** to 1.0 and **transparency** to 0.0 (yes, we do not use refraction at all!).

Set the **refr_ior** to 1.33 and **brdf_fresnel** to *on*. Apply some interesting wobbly shader to **bump** and our ocean is basically done!

This ocean has *only* reflections guided by the IOR. But this might work fine - try it. Just make sure there is something there for it to reflect! Add a sky map, objects, or a just a blue gradient background. There must be *something* or it will be completely black.

The Ocean isn't blue - the sky is

For a more "tropical" look, try setting **diffuse** to some slight greenish/blueish color, set the **diffuse_weight** to some fairly low number (0.1) and check the **no_diffuse_bump** checkbox.

Now we have a "'base color" in the water which emulates the little bit of scattering occurring in the top level of the ocean.

Enjoy the tropics

1.7.2.8 Metals

Metals are very reflective, which means they need something to reflect. The best looking metals come from having a true HDRI environment, either from a spherically mapped HDRI photo[16], or something like the *mental ray* physical sky.

To set up classic chrome, turn **brdf_fresnel** *off*, set **reflectivity** to 1.0, **brdf_0_degree_refl** to 0.9 and **brdf_90_degree_refl** to 1.0. Set **diffuse** to white and check the **refl_is_metal** checkbox.

This creates an almost completely reflective material. Tweak the **refl_gloss** parameter for various levels of blurry reflections to taste. Also consider using the "round corners" effect, which tend to work very well on metallic objects.

Metals also influence the *color* of their reflections. Since we enabled **refl_is_metal** this is already happening; try setting the **diffuse** to a "gold" color to create gold.

Try various levels of **refl_gloss** (with the help of **refl_interpolate** for performance, when necessary).

One can also change the **reflectivity** which has a slightly different meaning when **refl_is_metal** is enabled; it blends between the reflections (colored by the **diffuse**) and normal diffuse shading. This allows a "blend" between the glossy reflections and the diffuse shading, both driven by the same color. For example, an aluminum material would need a bit of diffuse blended in, whereas chrome would not.

[16] Many HDRI images are available online.

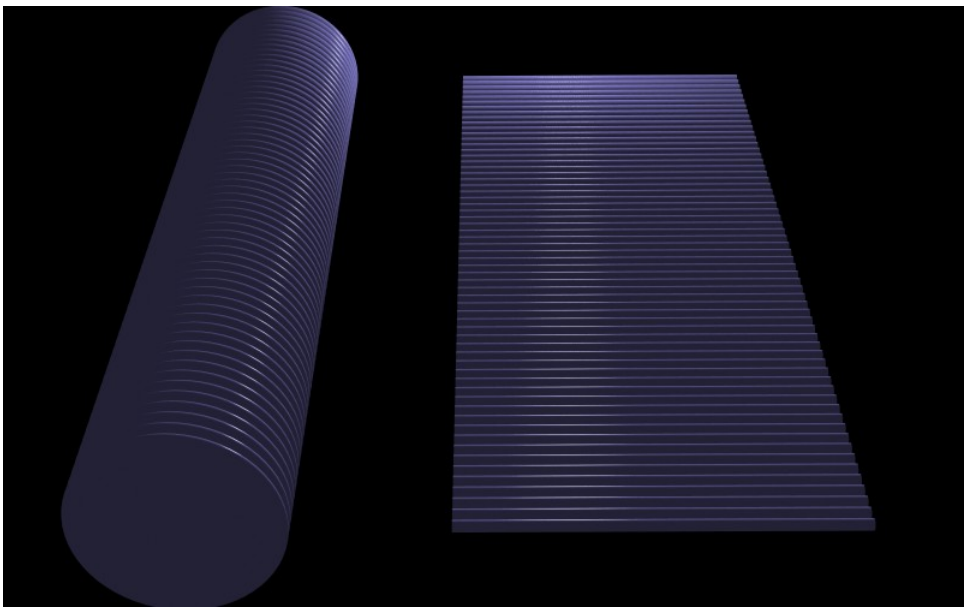
Gold, silver and copper, perhaps?

1.7.2.9 Brushed Metals

Brushed metal is an interesting special case of metals. In some cases, creating a brushed metal only takes turning down the **refl_gloss** to a level where one receives a "very blurred" reflection. This is sufficient when the brushing direction is random or the brushes are too small to be visible even as an aggregate effect.

For materials that have a clear *brushing direction* and/or where the actual brush strokes are *visible*, creating a convincing look is a slightly more involved process.

The tiny grooves in a brushed metal all work together to cause *anisotropic* reflections. This can be illustrated by the following schematic, which simulates the brush grooves by actually modeling many tiny adjacent cylinders, shaded with a simple Phong shader:



Many small adjacent cylinders

As one can see, the specular highlight in each of the cylinders work together to create an aggregate effect which is the *anisotropic highlight*.

Also note that the highlight isn't continuous, it is actually broken up in small adjacent segments. I.e. the main visual cues that a material is "brushed metal' are:

- Anisotropic highlights that stretch out in a direction *perpendicular* to the brushing direction.
- A discontinuous highlight with "breaks" in the brushing direction.

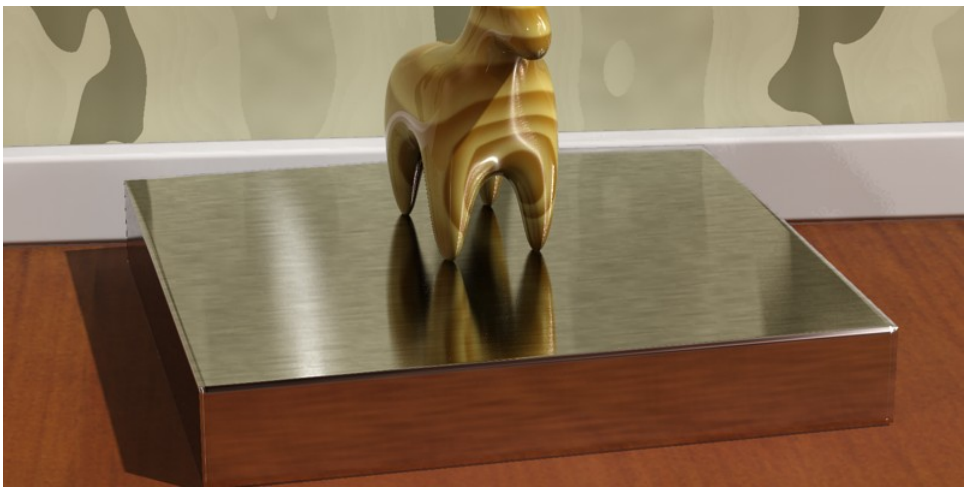
Many attempts to simulate brushed metals have only looked at the first effect, the anisotropy. Another common mistake is to think that the highlight stretches *in* the brushing direction. Neither is true.

Hence, to simulate brushed metals, we need to simulate these two visual cues. The first one is simple; use **anisotropy** and **anisotropy_rotation**to make anisotropic highlights. The second can be done in several ways:

- With a **bump** map
- With a map that varies the **anisotropy** or **refl_gloss**
- With a map that varies the **refl_color**

Each have advantages and disadvantages, but the one we will try here is the last one. The reason for choosing this method is that it works well together with interpolation.

1. Create a map for the "brush streaks". There are many ways to do this, either by painting a map in a paint program, or by using a "Noise" map that has been stretched heavily in one direction.
2. The map should vary between middle-gray and white. Apply this map to the **refl_color** in a scale suitable for the brushing.
3. Set **diffuse** to *white* (or the color of the metal) but set **diffuse_weight** to 0.0 (or a small value).
4. Make sure **refl_is_metal** is enabled.
5. Set **refl_gloss** to 0.75.
6. Set **anisotropy** to 0.1 or similar. Use **anisotropy_rotation** to align the highlight properly with the map. If necessary use **anisotropy_channel** to base it on the same texture space as the map.



Brushed Metal

Chapter 2

Sun and Sky

2.1 Introduction

The *mental ray* physical sun & sky shaders are designed to enable physically plausible daylight simulations and very accurate renderings of daylight scenarios.

The *mia_physicalsun* and *mia_physicalsky* are intended to be used *together*, with the *mia_physicalsun* shader applied to a *directional light* that represents the sun light, and the *mia_physicalsky* shader used as the scenes camera environment shader. The environment shader should be used to illuminate the scene with the help of Final Gathering (which must be enabled) and bounced light from the sun can be handled either by final gather diffuse bounces, or via GI (photons).

For improving quality in indoor shots, the sky can be combined with the *mia_portal_light* shader described on page 67.

2.2 Units

The sun and sky work in true photometric units, but the output can be converted to something else with the **rgb_unit_conversion** parameter.

If it is set to 1 1 1, both the values returned by the *mental ray* shader API functions *mi_sample_light* (for the sunlight) and *mi_compute_avg_radiance* (for the skylight), when sent through the *mi_luminance* function, can be considered as photometric *illuminance* values in *lux*.

Since the intensity of the sun outside the atmosphere is calibrated as a 5900 degrees Kelvin blackbody radiator providing an illuminance of 127500 lux, this is *very bright* when seen compared to a more "classical" rendering where light intensities generally range from 0 to 1.

The **rgb_unit_conversion** parameter is applied as a multiplier and could be set to a value below 1.0 (e.g. 0.001 0.001 0.001) to convert the raw lux value to something more "manageable".

For convenience, the special **rgb_unit_conversion** value of 0 0 0 is internally set so that 80000 lux (approximately the amount of light on a sunny day) equals the classical light level of 1.0.

An interesting alternative is to set **rgb_unit_conversion** to 0.318 0.318 0.318. Then the final rendered pixels in the image (when rendered with *mia_material* or shaders following that shading convention and when sent through the *mi_luminance* function) are true photometric *luminance* values in *candela per square meter*[1].

These true luminance values then fit perfectly as input to the photographic tone mapper described on page 79.

2.3 Important Note on Fast SSS and Sun&Sky

To use the *mental ray* fast SSS shaders together with the high dynamic range indirect sun and skylight, it is very important to turn *on* the **Indirect** parameter so the SSS shader can scatter the skylight (which is considered indirect).

It is equally important to turn *off* the **Screen composit** parameter (otherwise the output of the SSS shaders are clamped to a low dynamic range and will appear to render black).

2.4 Common Parameters

Some parameters exist both in the *mia_physicalsun* and *mia_physicalsky* and all do the same thing. For physical correctness, it is necessary to keep these parameter *in sync* with each other in both the sun and sky. For example, a sun with a different *haze* value than the sky cannot be guaranteed to be physically plausible.

[1] The value 0.318 (1/pi) originates from the illuminance/luminance ratio of a theoretically perfect Lambertian reflector.

The most important common parameters are those that drive the entire shading- and colorization model:

- **haze** sets the amount of haze in the air. The range is from 0 (a completely clear day) to 15 (extremely overcast, or sandstorm in sahara). The haze influences the intensity and color of the sky and horizon, intensity and color of sunlight, softness of the suns shadows, softness of the glow around the sun, and the strength of the aerial perspective.

Haze=0

Haze=3

Haze=8

Haze=15

- **redblueshift** gives artistic control over the "redness" of the light. The default value of 0.0 is the physically correct value[2], but can be changed with this parameter which ranges from -1.0 (extremely blue) to 1.0 (extremely red).

Redness=-0.3

Redness=+0.3

[2]Calculated for a 6500K whitepoint.

- **saturation** is also an artistic control, where 1.0 is the physically calculated saturation level. The parameter ranges from 0.0 (black and white) to 2.0 (extremely boosted saturation)

2.5 Sun Parameters

The *mia_physicalsun* is responsible for the color and intensity of the sunlight, as well as emitting photons from the sun. The shader should be applied as *light shader* and *photon emission shader* on a *directional light source* (it does not work on any other light type).

```
declare shader "mia_physicalsun" (
        boolean "on"                    default on,
        scalar  "multiplier"            default 1.0,
        color   "rgb_unit_conversion"   default 0.0001 0.0001 0.0001,
        scalar  "haze"                  default 0.0,
        scalar  "redblueshift"          default 0.0,
        scalar  "saturation"            default 1.0,
        scalar  "horizon_height"        default 0.0,
        scalar  "shadow_softness"       default 1.0,
        integer "samples"               default 8,
        vector  "photon_bbox_min",
        vector  "photon_bbox_max",
        boolean "automatic_photon_energy",
        boolean "y_is_up"
    )

    version 5
    apply light

end declare
```

As mentioned above, the *mia_physicalsun* contains several of the common parameters that are exposed in the *mia_physicalsky* as well (**haze**, **redblueshift** etc.). The value of these parameters for the *mia_physicalsun* should match those in the *mia_physicalsky*.

The parameters specific to the *mia_physicalsun* are as follows:

- **samples** is the number of shadow samples for the soft shadows. If it is set to 0, no soft shadows are generated.
- **shadow_softness** is the softness for the soft shadows. A value of 1.0 is the value which matches the softness of real solar shadow most accurately. Lower values makes the shadows *sharper* and higher *softer*.

  When **photon_bbox_min** and **photon_bbox_max** are left set to 0,0,0 the photon bounding box will be calculated automatically by the shader. If these are set, they define

a bounding box *in the coordinate system of the* ***light*** within which photons are aimed. This can be used to "focus" GI photons on a particular area-of-interest. For example, if one has modelled a huge city as a backdrop, but is only rendering the interior of a room, *mental ray* will by default shoot photons over the entire city and maybe only a few will find their way into the room. With the **photon_bbox_max** and **photon_bbox_min** parameters one can focus the photon emission of the *mia_physicalsun*to only aim at the window in question, greatly speeding up and enhancing the quality of the interior rendering.

automatic_photon_energy enables automatic photon energy calculation. When this is *on*, the light source does not need to have a valid energy value that matches that of the sun (it does, however, need a nonzero energy value or photon emission is disabled by *mental ray*). The correct energy and color of the photons will be automatically calculated. If this parameter is *off*, the photons will have the energy defined by the lights energy value.

2.6 Sky Parameters

The *mia_physicalsky* shader is responsible for creating the color gradient that represent the atmospheric skydome, which is then used to light the scene with the help of Final Gathering and/or sky portals (page 67). The *mia_physicalsky*, when used as an environment shader, also show the sky to the camera and in reflections.

mia_physicalsky also creates a *virtual ground plane* that exists "below" the model. This makes it unnecessary to actually model geometry all the way to the horizon line - the *virtual ground plane* provides both the visuals and the bounce-light from such ground.

```
declare shader "mia_physicalsky" (
        boolean "on"                  default on,
        scalar  "multiplier"          default 1.0,
        color   "rgb_unit_conversion" default 0.0001 0.0001 0.0001,
        scalar  "haze"                default 0.0,
        scalar  "redblueshift"        default 0.0,
        scalar  "saturation"          default 1.0,

        scalar  "horizon_height"      default 0.0,
        scalar  "horizon_blur"        default 0.1,
        color   "ground_color"        default 0.2 0.2 0.2,
        color   "night_color"         default 0 0 0,

        vector  "sun_direction",
        light   "sun",

        # The following parameters are only useful
        # when the shader is used as environment
        scalar  "sun_disk_intensity"  default 1.0,
        scalar  "sun_disk_scale"      default 4.0,
        scalar  "sun_glow_intensity"  default 1.0,
```

```
        boolean "use_background",
        shader  "background",

        # For the lens/volume shader mode
        scalar  "visibility_distance",

        boolean "y_is_up",
        integer "flags"
    )

    version 4
    apply environment, texture, lens, volume

end declare
```

- **on** turns the shader on or off. The default is *on*.
- **multiplier** is a scalar multiplier for the light output. The default is 1.0.
- **rgb_unit_conversion** allows setting the units, described in more detail above. The special value of 0 0 0 matches 80000 lux (light level on a sunny day) to the output value 1, suitable for low dynamic range rendering.
- **horizon_height** sets the "level" of the horizon. The default value of 0.0 puts the horizon at standard "height". But since the horizon is *infinitely* far away this can cause trouble joining up with any *finite* geometry that is supposed to represent the ground. It can also cause issues rendering locations that are supposed to be at a high altitude, like mountain tops or the top of New York skyscrapers where the horizon *really is* visibly "below" the viewer.

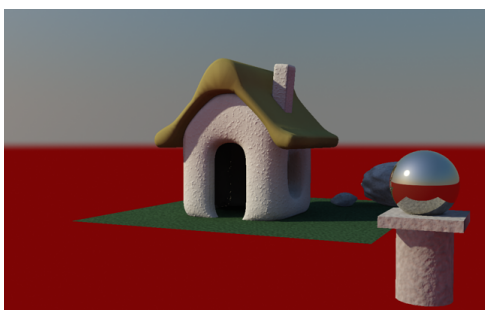
  This parameter allows tuning the position of the horizon. Note that the horizon doesn't actually exist at any specific "height" in 3D space - it is a shading effect for rays that go below a certain *angle*. This parameter tweaks this angle. The total range available range is somewhat extreme, reaching from -10.0 (the horizon is "straight down") to 10.0 (the horizon is at the zenith)! In practice, only much smaller values are actually useful, like for example -0.2 to push the horizon down just below the edge of a finite visible ground plane.

  Note: The **horizon_height** affects not only the visual representation of the horizon in the *mia_physicalsky* shader, but also the color of the *mia_physicalsun* itself, i.e. the point where the sun "sets" will indeed change for a nonzero **horizon_height**.
- **horizon_blur** sets the "blurriness" with which the horizon is rendered. At 0.0 the horizon is completely sharp. Generally low values (lower than 0.5) are used, but the full range is up to 10.0 for a horizon which only consists of blur and no actual horizon at all.

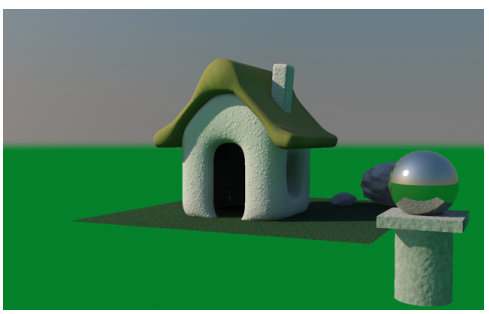
***horizon_height**=0.0, **horizon_blur**=0.0*

***horizon_height**=-0.3, **horizon_blur**=0.2*

- **ground_color** is the color of the *virtual ground plane*. Note that this is a diffuse reflectance value (i.e. albedo). The ground will appear as if it was a Lambertian reflector with this diffuse color, lit by the sun and sky *only*, does *not* receive any shadows.



Red ground



Green ground

Note in the above images how bounce-light from the ground tints the walls of the house. Also note that the virtual ground plane does *not* receive shadows.

Many sky models neglect the influence from bounce light from the ground, assuming only the sky is illuminating the scene. To compare the output if the *mia_physicalsky* with e.g. the IES sky model one must therefore set the **ground_color** to black.

- **night_color** is the minimum color of the sky - the sky will never become darker than this value. It can be useful for adding things like moon, stars, high altitude cirrus clouds that are lit long after sunset etc. As the sun sets and the sky darkens, the contribution from **night_color** is unaffected and remains as the "base light level".

- **sun_direction** is the direction of the sun disk when specified manually. If the **sun** parameter is used, this parameter is ignored.

- **sun** is the way to automatically set the sun direction. It should be the tag of the light instance that contains the directional light that represents the sun - i.e. the same light that has the *mia_physicalsun*shader. This will make the visible sun disk automatically follow the direction of the actual sunlight.

- *Aerial Perspective* is a term used by painters to convey how distant objects are perceived as hazier and tinted towards the blue end of the spectrum. *mia_physicalsky* emulates this with the **visibility_distance** parameter. When nonzero, it defines the "10% distance", i.e. the distance at which approximately 10% of haze is visible at a **haze** level of 0.0.

  To use this effect, the shader must be applied as either a *lens* or *camera volume* shader.

- **y_is_up** defines what constitutes "up". Some OEM integrations of *mental ray* considers the Z axis "up" and hence this parameter should be *off* - others consider the Y axis "up" and in that case this parameter should be *on*.
- **flags** is for future expansion, testing and internal algorithm control. Should be set to zero.

It is important to note that the *mia_physicalsky* shader treats rays differently. Direct rays from the camera, as well as reflection and refraction rays see the "entire" effect, including the "sun disk" described below. But since the lighting already has a direct light that represents the sun (using the *mia_physicalsun* shader) the sun disk is not visible to the finalgather rays [3].

These parameters do not affect the Final Gathering result, only the "visible" result, i.e. what the camera sees and what is seen in reflection and refraction:

- **sun_intensity** and **glow_intensity** is the intensity of the visible sun disk and it's glow, which can be used to tune the "look" of the sun.

***glow_intensity**=5*

***glow_intensity**=0.1*

- **sun_scale** sets the size of the visible sun disk. The value 1.0 is the "physically correct" size, but due to the fact that people tend to misjudge the proper size of the sun in photographs, the default is the slightly more visually pleasing 4.0

***sun_scale**=1*

***sun_scale**=4*

[3] This would otherwise cause noise in the Final Gathering solution and too much light added to the scene.

- When **use_background** is enabled but no **background** has been set, the background of the rendering will be transparent black, i.e. suitable for external compositing. If a **background** shader is supplied, the background of the rendering will come from *that* shader (for example a texture shader that looks up a background photograph of a real location or similar). In either case the *mia_physicalsky* will still be visible in refractions and reflections.

2.7 Sky- and Environment Portals

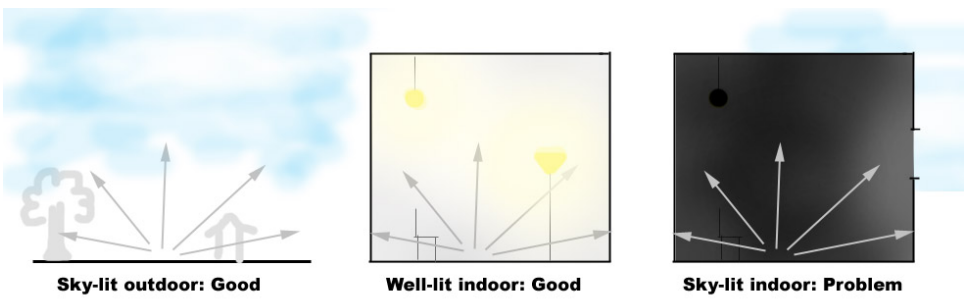
2.7.1 The Problem

A classic problem in computer graphics is lighting a scene solely through indirect light, like from a sky, or other "environment" light from an acquired HDRI or similar.

This is accomplished in mental ray using Final Gathering (henceforth abbreviated as FG), and is done by tracing a large number of "FG rays" to see which hit the environment (or other lit surfaces). Since this is a *large* number of rays, the results are cached (for performance) at *FG points* and the result is interpolated, "smoothing" the result.

This all works very well when there is a lot of fairly uniform light that is "seen" by the FG rays. In general, FG gives the best result when the light levels in a scene is fairly uniform; it works well to illuminate an outdoors scene by the sky (most FG rays "see" the sky), and it works well to bounce secondary light in a room in which most surfaces are lit by direct lights (most FG rays "see" some already-lit surface).

However, a scene of a dark room with *no* lights, and a single window *only* exposed to the sky is more problematic:



In the image on the right, almost all FG rays will "see" blackness and only a select few will be able to "escape" through the narrow window to hit the sky. To resolve this accurately one need to shoot very many FG rays, which has a negative impact on performance.

A further difference is that in the first two scenes (the outdoor scene and the well-lit interior) there are already direct lights which cause high-quality direct shadows, which resolves the details in the scene; FG is only used for additional bounce- or sky-lighting, and doesn't provide the bulk of the lighting. Therefore, any "over-smoothing" caused by the interpolation of FG points is drowned out by other lights (or can be resolved with AO in *mia_material*).

But the rightmost scene does not have that luxury, *all* light is indirect skylight. Any over-smoothing due to interpolation will be clearly visible, which means that one needs *both* a high FG ray count *and* a high FG point density to create a pleasing render, which gives longer render times.

A common technique used by many people doing interior renderings is to put an area light in the window, to provide the sky-lighting rather than rely on FG to "find" the sky. But this gives rise to the question "how bright and what color should this light be?".

2.7.2 The Solution

To solve *all* these issues the concept of a *portal light* is introduced. The portal light is a (rectangular) area light which is placed in the window, which obtains it's proper intensity and color from the sky *outside* the window (i.e. an environment shader, like *mia_physicalsky* or similar) and how much of that sky that is "seen"[4].

Practically, this makes the portal light behave as a "FG concentrator" so instead of having to send thousands of FG rays around the scene to "find" the window, the portal light actually *blocks* FG rays, and instead converts light from beyond the window to direct light, with high-quality area shadows with no interpolation related issues possible.

FG will now see a *well lit* room rather than a *black* room, and can be performed at much lower FG ray counts. Furthermore, since the light from the window is now *direct*, we gain one extra light bounce "for free".

2.7.2.1 mia_portal_light

The *mia_portal_light* shader should be applied both as light- and photon emitter shader on a *rectangular area light*. The mental ray light instance must be set to be *visible* (this is a technical requirement for the portal light to be able to "block" final gather rays. If the light actually *is* visible or not in the rendering is instead handled by the shader).

Furthermore, the mental ray light instance must be set up such that the rectangular area light is extended in the X/Y plane of the lights own coordinate space, and any transformation of the light must be handled with the light instance's transform[5].

The following parameters exist:

[4] The subtended solid angle of the window as seen from the shading point.

[5] This is how most OEM application already sets up the mental ray area light instances.

```
declare shader "mia_portal_light" (
        boolean "on"                      default on,
        scalar  "multiplier"              default 1.0,
        color   "tint_color"              default 1 1 1,
        boolean "reverse"                 default off,
        scalar  "cutoff_threshold"        default 0.005,
        boolean "shadows"                 default true,
        boolean "use_custom_environment" default off,
        shader  "custom_environment",
        boolean "visible"                 default off,
        boolean "lookup_using_fg_rays"    default on,
        scalar  "shadow_ray_extension"    default 0.0,
        boolean "emit_direct_photons"     default off,
        color   "transparency"            default 1 1 1
    )
    version 9
    apply light, emitter
end declare
```

on enables or disables the light.

multiplier sets the intensity and **tint_color** modifies the light color. When it is white, and the **multiplier** is 1.0, the light emitted is equal in intensity (and color) to the environment light that FG would have seen if allowed to send many thousands of FG rays[6].

The light normally shines in the positive Z direction of the light instance's coordinate space. If **reverse** is on, it shines in the negative Z direction.

cutoff_threshold is a performance optimization option. Any light below this level is ignored, and no shadow rays are traced (which is what consumes the bulk of the render time of an area light). Of course, this makes the scene *slightly* darker since light is ignored, but can save a lot of excess render time.

shadows can turn shadows on and off.

If **use_custom_environment** is *off*, the shader looks in the global camera environment for the color of the light. If it is *on*, it calls the shader passed as **custom_environment** to find the color.

Tip: While the shader is intended to be a portal to an environment, one can also treat it as a "light card" shader by putting a shader returning a solid color as the **custom_environment**, for example using **mib_blackbody** creates a light card with a given color temperature.

If **use_custom_environment** is off and no **custom_environment** is actually passed, the shader behaves as a white light card.

visible defines if the light emitting surface is visible or not. When *off*, eye rays, reflection rays etc. go straight through so the portal light itself remains unseen (and we still "see"

[6]For FG filter 0, the unbiased mode.

out the window). When *on*, the actual light emitting surface becomes visible to eye rays, reflection rays etc (and one do not "see" out the window any more, although one still "see" the environment shaders result). The *on* mode is useful when using *mia_portal_light* as a light card shader.

When **lookup_using_fg_rays** is *off*, the environment shader is looked up with a normal call to *mi_trace_environment()*. However, some shaders behave differently if they are called by an FG ray or by another ray (the *mia_physicalsky* shader, for example, does not show the "image" of the visible sun to FG rays). Since the idea of *mia_portal_light* is to act as an "FG concentrator" it should therefore follow that behaviour. So when **lookup_using_fg_rays** is *on*, it calls the environment with the ray type set to miRAY_FINALGATHER, so that shaders that switch behaviour based on this can return the color appropriate for a FG ray.

If **shadow_ray_extension** is zero, the shader begins tracing shadow rays "at" the light. When positive, the shadow rays actually start that distance "outdoors". So if there is a large object just *outside* the window, it's shadow will be taken into account. Conversely, a negative value allows the shadow rays to begin that distance *inside* the window, which can allow them to "skip" over troublesome geometry near the window (flowers, curtains) that would otherwise just introduce noise into the shadows.

If **emit_direct_photons** is on, the light only shoots direct photons, and does not actually emit any direct light at all.

The **transparency** parameters has two functions.

- When **visible** is *on*, it is a multiplier to the "visible color" of the area light. When this is *white*, the directly "visible" color is the one dictated by the laws of physics for a surface that emits that amount of light.

  Changing the parameter away from *white* allows one to artificially change the balance between the visible result (which is changed by changing this parameter) and the intensity of the emitted light (which is *not* affected by this parameter). This can be useful to avoid noise

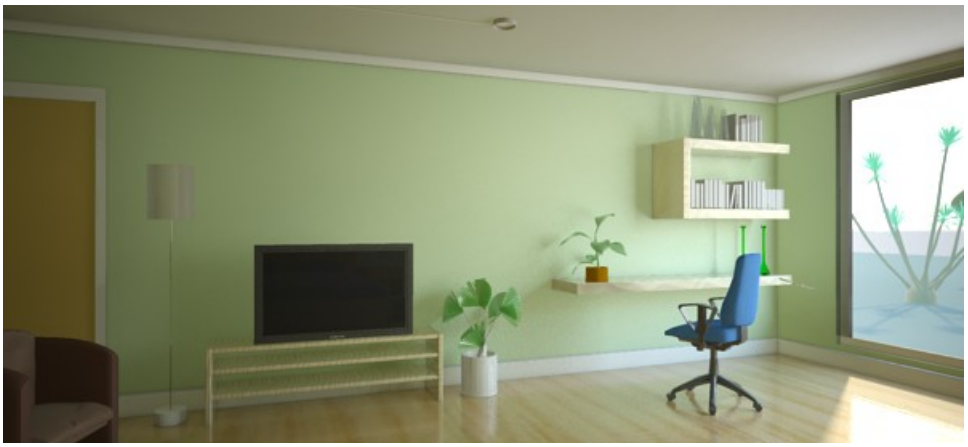
- If **visible** is *off* it defines the transparency of the area light.

  This allows the *mia_portal_light* shader to double as a "gel" on the window, to subdue the intensity of what is seen outdoors, which otherwise tends to appear overexposed and blown out. The actual emitted light intensity is not affected by this, nor does this affect the intensity of other light rays travelling through the window, it only affects what is visible to the eye, in refractions or reflections.

2.7.3 Examples

In this section we will examine the benefits of using the portal lights compared to what has been possible in previous versions of *mental ray*.

We are using the following scene[7]:



Our scene, using the portal lights, GI and FG.

The scene is only lit by the sun and sky, there are no light sources inside the room of any kind.

2.7.3.1 Without Portal Lights

To clearly demonstrate what is direct and indirect light in the scene, we here show the scene with portals, GI and FG turned off:

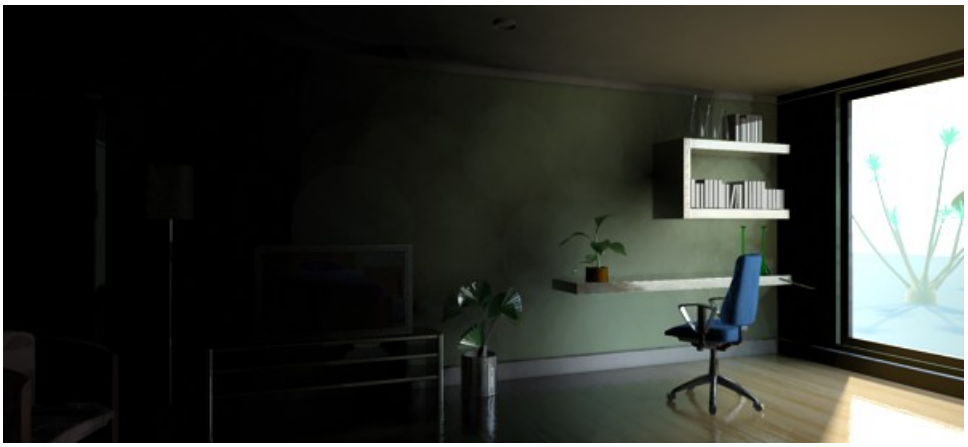
The direct lighting of our scene

The above image shows the isolated direct light. This means that this result is what FG will "see" - an extremely high-contrast scene consisting of complete blackness, the hotspot of the

[7]Large parts of the example scene geometry is provided courtesy of Giorgio Adolfo Krenkel.

direct sunlight on the floor, and the very bright sky, and sun-lit ground outside - a sub-optimal input to the FG algorithm.

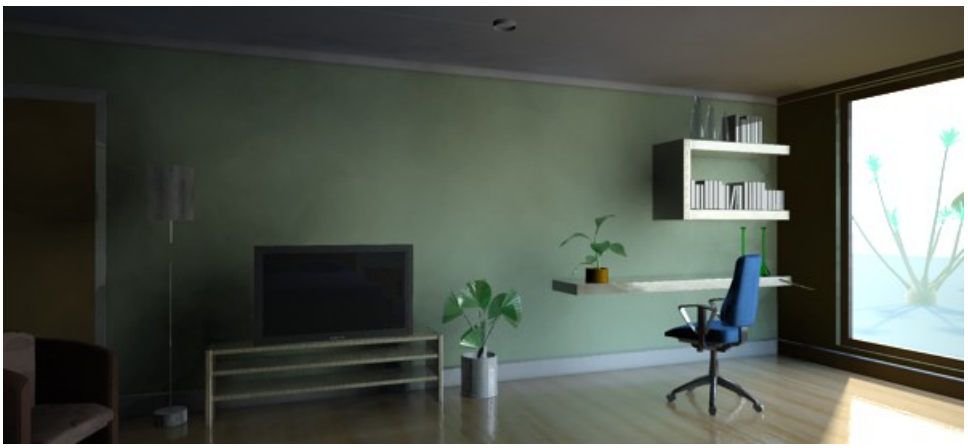
If we turn on FG on relatively low settings we get:



FG with 50 rays and density 0.1

This image wins no beauty awards. It is splotchy, the shelves seem to "float" away from the wall, but most surprisingly, it's very dark. Why is that?

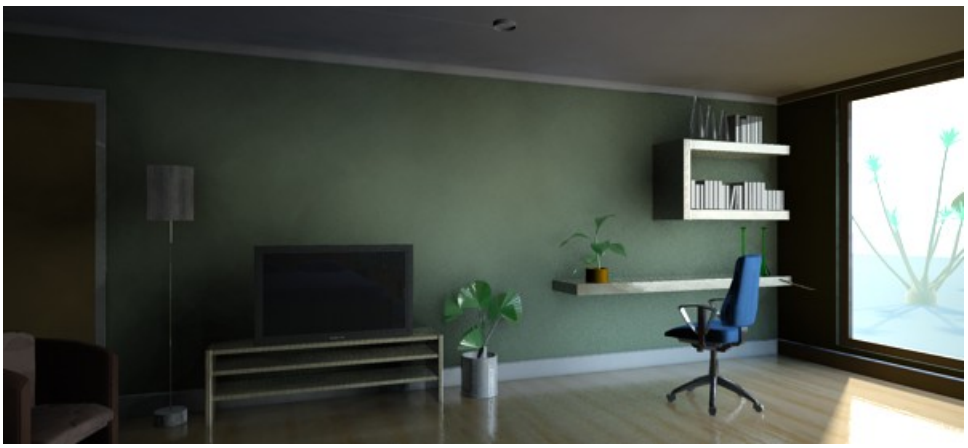
The reason is the high contrast input. FG contains a filter that is intended to avoid a speckled result if some stray FG rays hit a single extremely bright object, so the filter removes the brightest rays. But our scene really *is* high contrast, and we actually *expect* some rays to be much brighter than others.



Using FG filter = 0

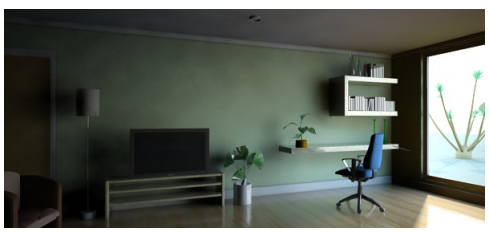
In our case the filter actually is fighting what we are trying to do. We can turn off the filter, as in the image above, which helps our light distribution some, but not the splotches, nor the "disconnect" of the shelves to the wall.

Since we are using *mia_material*, we have a built in ambient occlusion to help in exactly these situations. However, turning that up only helps partially:

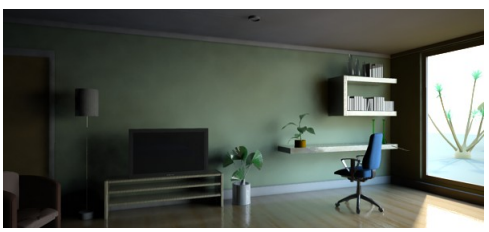


Adding AO - helps a bit

The lone solution available in the past was to simply increase the quality of the FG settings. And naturally, we are using *very low* settings, so the bad quality is not surprising at this stage. Lets turn up the knobs:



FG with 250 rays and density 0.8

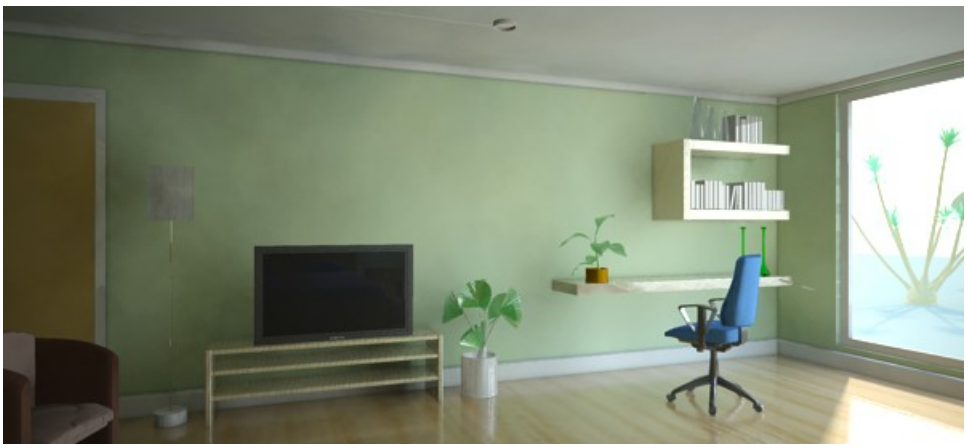


FG with 500 rays and density 1.5

Yes, increasing the number of rays and the density *helps* but it hurts our render time a lot. Even at the high setting we are still not near the optimal result. We would have to go even higher to resolve all the deatil!

Lets back down from the high FG settings for a moment and concentrate on light transport. The scene is still very dark, because we are only getting a single FG bounce, and since the lighting from the sky is *indirect* we get *no* bounce of that light! Meaning: In this scene, the sunlight is bouncing once, but the skylight is not bouncing at all!

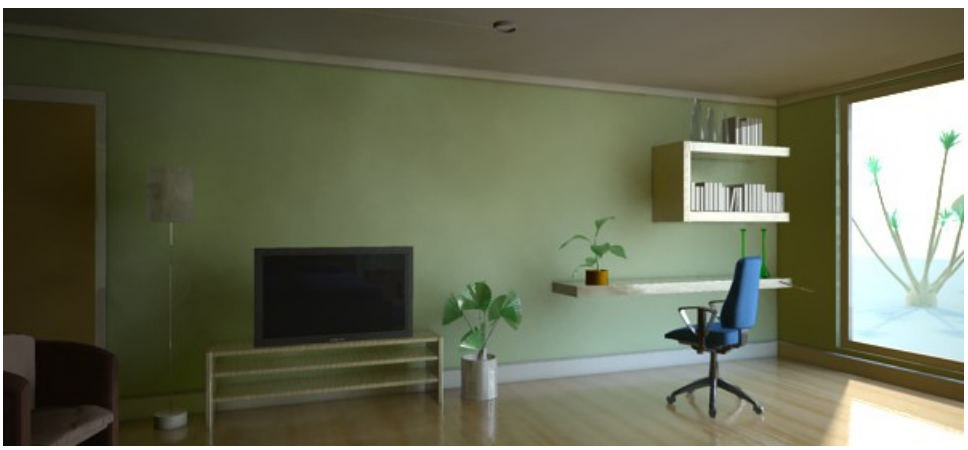
Lets turn up the number of bounces to 3:



FG with 3 bounces

This gave us more light. Due to the fact we stepped back from our "high" FG settings, the lighting isn't smooth and lacks any detail.

Using FG multi-bounce is just one of the ways *mental ray* can transport light in a scene. The alternative is to use Photons (GI). But keep in mind that when Photons are enabled, FG goes back to a single bounce, letting photons handle all the remaining bounces:

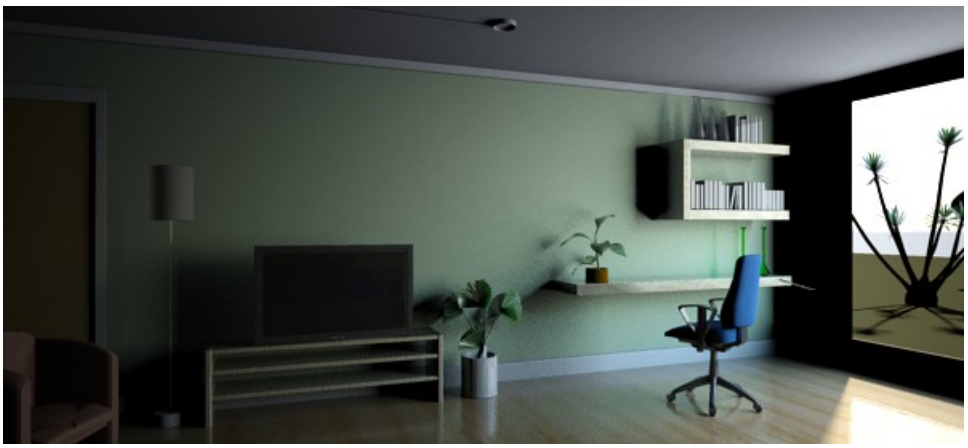


FG with GI (Photons)

Now something odd happened; the color shifted towards yellow. Why is that? This is because the sky does not generate any photons. So we now have multiple good-quality bounces of the *sun*, but we are back to *zero* bounces of the sky!

2.7.3.2 With Portal Lights

Now lets turn to the portal lights. First lets turn all FG and GI off, and simply add a portal light to the window with it's default settings. The resulting image is this:

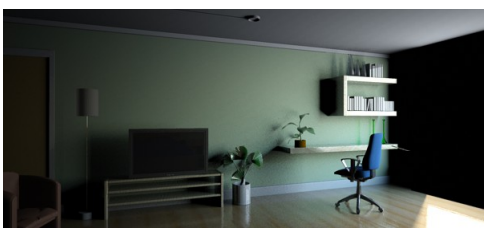


The portal light only, no FG or Photons

Color wise this looks very similar to our early FG results above, but the *level of detail* is much higher. Since no interpolation is going on, the shelves sit securely on the walls. All shadows contain the subtlest of details.

Since this is now *direct* light it will be picked up by FG when that is turned on. Let us compare what FG would "see" with, and without the portal lights:

Without portal lights

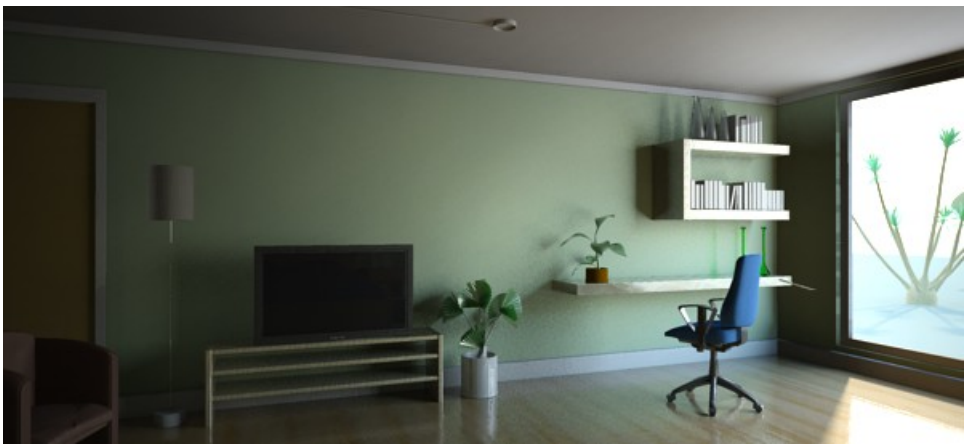


With portal lights

The left is the super-high contrast result FG would "see" without portals, but the right is a fairly well balanced scene. And not only is it already filled with subtle direct light - the sky itself is actually *invisible* to FG, so it never has to carry the burden of hitting a high contrast area[8] and the problem with the final gather filter pretty much disappears.

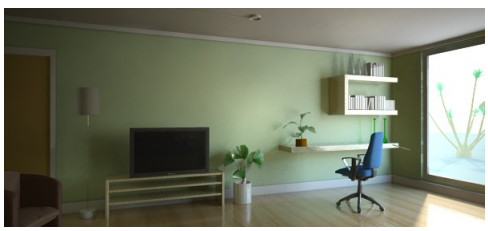
The second feature of the portal lights being *direct* is that if we turn FG back on, we now get once bounce of light "for free":

[8] Although FG rays that directly hit the pool of light on the floor caused by the sun will indeed see a high contrast.

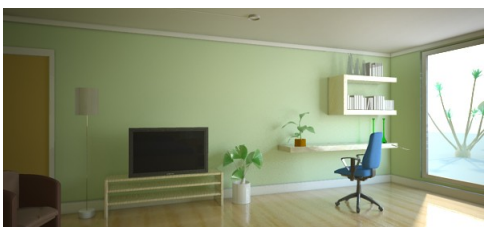


One free bounce of sky lighting, courtesy of the portal lights

Notice how the wall inside the window now has some bounce light on it from the sky light, even though we are using FG with a single bounce!



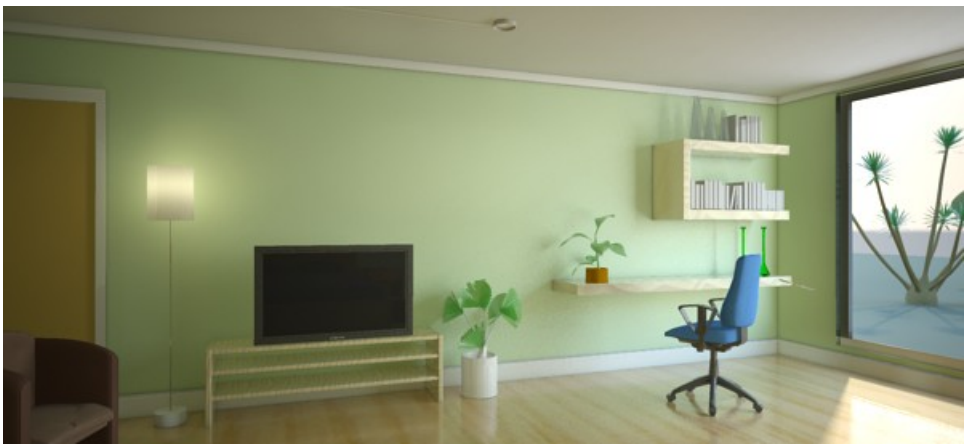
Using portal lights and 3 diffuse bounces



Using GI (Photons)

Turning on multiple diffuse bounces (on the left) makes the image come alive. Turning on GI (Photons) now yields a much more balanced image *because the portal lights actually shoot photons*! No longer is there the big skewing in favor of bounced sun light, the sky light will bounce equally well in the form of photons.

Finally, turning on one of the indoor lights, and utilizing the portal lights **transparency** parameter to combat the overexposure of the outdoor view, we get a final image:



A final image

In conclusion: The portal lights help...

- ...reducing render time (by reducing the need for using extremely high FG settings).
- ...maintain the light balance in the scene (by converting environment light to photons, which can bounce around indoors just as well as sun light).
- ...increase the *quality* of the sky lighting, making smooth areas smooth and blotch-free, as well as revealing the most minute detail with full fidelity.
- ...*dramatically* reduce scene set-up time (by requiring little to no tweaking, always generating the "optimal" image out-of-the-box).

The subtle interplay of skylight, courtesy of the portal lights

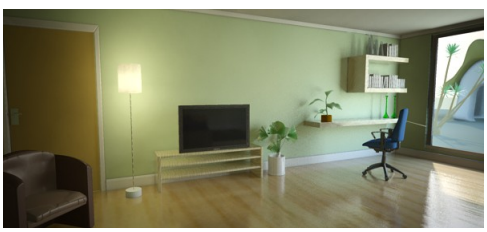
Chapter 3

Camera- and Exposure Effects

3.1 Tone Mapping

When rendering physical light levels one runs into the problem of managing the HDRI output of the real physics vs. the limited dynamic range of computer displays. This was discussed in more detail on page 3.

Going from this...



...to this.

There are numerous shaders and algorithms for doing "tone mapping" (this is a very active area of research within the CG industry), and the architectural library provides two: One very simple shader that simply adds a knee compression to "squash" over-brights into a manageable range, and a more complex "photographic" version that converts real photometric luminances with the help of parameters found on a normal camera into an image.

These shaders can be applied either as a lens shaders (which will tone map the image "on the fly" as it is being rendered) or as output shaders (will tone map the image as a post process).

Since both of these tone mappers affects each pixel individually[1] , the former method (as lens shader) is encouraged, since it applies on the sample level rather than the pixel level.

[1]Many advanced tone mappers weight different areas of the image against other areas, to mimic the way the human visual system operates. These tone mappers need the entire image before they can "do their job".

Also, both of the supplied tone mappers include a *gamma* term. It is important to know if one is already applying a gamma correction elsewhere in the imaging pipeline (in an image viewer, in compositing, etc.), and if so, set it to 1.0 in these shaders. If one is *not* applying gamma correction anywhere but simply displaying the image directly on screen with a viewer that does not apply it's own gamma, one should most likely use the gamma in these shaders, set to a value between 1.8 and 2.4.

3.1.1 The "Simple" Tone Mapper

3.1.1.1 mia_exposure_simple

```
declare shader  "mia_exposure_simple" (
        scalar  "pedestal"       default 0.0,
        scalar  "gain"           default 1.0,
        scalar  "knee"           default 0.5,
        scalar  "compression"    default 2.0,
        scalar  "gamma"          default 2.2,
        color   texture "preview",
        boolean "use_preview"
    )
    version 1
    apply lens, output
end declare
```

The operation of this tone mapper is very simple. It does not refer to real physical luminance values in any way. It simply takes the high dynamic range color and perform these operations in order:

- **pedestal** is *added* to the color.
- The color components are then multiplied with **gain**.
- The resulting colors are checked if they are above the **knee** value.
- If they are, they are "squashed" by the compression ratio **compression**
- Finally, gamma correction with **gamma** is performed.

That's the theory. What is the practical use of these parameters?

Changing **pedestal** equates to tweaking the "black level". A positive value will add some light so even the blackest black will become slightly gray. A negative value will subtract some light and allows "crushing the blacks" for a more contrasty artistic effect.

gain is the "brightness knob". This is the main point where the high dynamic range values are converted to low dynamic range values. For example: if one knows the approximate range

of color intensities goes between 0 and 10, this value should then be approximately 0.1 to get this range into the desired 0-1 range.

However, the whole point of tone mapping is *not* to blindly linearly scale the range down. Simply setting it to 0.1 most likely yields a dark and boring image. A much more likely value is 0.15 or even 0.2. But a value of 0.2 will map our 0-to-10 range to 0 to 2.... what to do about that stuff above 1.0?

That's where the compression comes in. The *knee* level is the point where the over-brights begin to be "squashed". Since this is applied *after* the gain, it should be in the range of 0.0 to 1.0. A good useful range is 0.5 to 0.75.

Assume we set it to 0.75. This means any color that (after having **pedestal** added and multiplied by **gain**) that comes out above 0.75 will be "compressed". If **compression** is 0.0 there is no compression. At a **compression** value of 5.0 the squashing is fairly strong.

Finally, the resulting "squashed" color is gamma-corrected for the output device (computer screen etc.)

The **use_preview** and **preview** parameters are used to make the process of tweaking the tone mapper a little bit more "interactive".

The intended use is the following, for when the shader is applied as a *lens shader*:

- Disable the *mia_exposure_simple* shader.
- Render the image to a file in some form of HDR capable format (like .exr, .hdr or similar), for example `preview.exr`.
- Enable *mia_exposure_simple* shader again.
- Set the **preview** parameter to the file saved above, e.g. `preview.exr`.
- Enable the **use_preview** parameter.
- Disable any photon mapping or final gathering.
- Re-render. The rendering will be near instant, because no actual rendering occurs at all; the image is read from `preview.exr` and immediately tone mapped to screen.
- Tweak parameters and re-render again, until satisfied.
- Re-enable any photons or final gathering.
- Turn off **use_preview**.
- Voila - the tone mapper is now tuned.

3.1.2 The “Photographic” Tone Mapper

3.1.2.1 mia_exposure_photographic

The photographic tonemapper converts actual pixel luminances (in candela per square meter) into image pixels as seen by a camera, applying camera-related parmeters (like f-stops and shutter times) for the exposure, as well as applying tonemapping that emulates film- and camera-like effects.

It has two basic modes:

- “Photographic” - in which it assumes input values are (or can be converted to) candela per square meter.
- “Arbitrary” - in which scene pixels are not considered to be in any particular physical unit, but are simply scaled by a factor to fit in the display range of the screen.

If the **film_iso** parameter is nonzero, the “‘Photographic” mode is used, and if it is zero, the “Arbitrary” mode is chosen.

```
declare shader "mia_exposure_photographic" (
        scalar "cm2_factor"         default 1.0,
        color  "whitepoint"         default 1 1 1,
        scalar "film_iso"           default 100,
        scalar "camera_shutter"     default 100.0,
        scalar "f_number"           default 16.0,
        scalar "vignetting"         default  1.0,

        scalar "burn_highlights"    default 0.0,
        scalar "crush_blacks"       default 0.25,
        scalar "saturation"         default 1.0,

        scalar "gamma"              default 2.2,

        integer "side_channel_mode" default 0,
        string  "side_channel",

        color   texture "preview",
        boolean "use_preview"
    )
    version 4
    apply lens, output
end declare
```

In “Photographic mode” (nonzero **film_iso**) **cm2_factor** is the conversion factor between pixel values and candela per square meter. This is discussed more in detail below.

In "Arbitrary" mode, **cm2_factor** is simply the multiplier applied to scale rendered pixel values to screen pixels. This is analogous to the *gain* parameter of *mia_exposure_simple*.

whitepoint is a color that will be mapped to "white" on output, i.e. an incoming color of this hue/saturation will be mapped to grayscale, but its intensity will remain unchanged.

film_iso should be the ISO number of the film, also known as "film speed". As mentioned above, if this is zero, the "Arbitrary" mode is enabled, and all color scaling is then strictly defined by the value of **cm2_factor**.

camera_shutter is the camera shutter time expressed as fractional seconds, i.e. the value 100 means a camera shutter of 1/100. This value has no effect in "Arbitrary" mode.

f_number is the fractional aperture number, i.e. 11 means aperture "f/11". Aperture numbers on cameras go in specific standard series, i.e. "f/8", "f/11", "f/16", "f/22" etc. Each of these are refered to as a "stop" (from the fact that aperture rings on real lenses tend to have physical "clicks" for these values) and each such "stop" represents halving the amount of light hitting the film per increased stop[2]. It is important to note that this shader doesn't count "stops", but actually wants the f-number *for* that stop. This value has no effect in "Arbitrary" mode.

In a real camera the angle with which the light hits the film impacts the exposure, causing the image to go darker around the edges. The **vignetting** parameter simulates this. When 0.0, it is off, and higher values cause stronger and stronger darkening around the edges. Note that this effect is based on the cosine of the angle with which the light ray would hit the film plane, and is hence affected by the field-of-view of the camera, and will not work at all for orthographic renderings. A good default is 3.0, which is similar to what a compact camera would generate[3].

The parameters **burn_highlights** and **crush_blacks** guide the actual "tone mapping" of the image, i.e. exactly *how* the high dynamic range imagery is adapted to fit into the black-to-white range of a display device.

If **burn_highlights** is 1 and **crush_blacks** is zero, the transfer is linear, i.e. the shader behaves like a simple linear intensity scaler only.

burn_highlights can be considered the parameter defining how much "over exposure" is allowed. As it is decreased from 1 towards 0, high intensities will be more and more "compressed" to lower intensities. When it is 0, the compression curve is asymptotic, i.e. an *infinite* input value maps to *white* output value, i.e. over-exposure is no longer possible. A good default value is 0.5.

When the upper part of the dynamic range becomes compressed it naturally loses some of it's former contrast, and one often desire to regain some "punch" in the image by using the **crush_blacks** parameter. When 0, the lower intensity range is linear, but when raised towards

[2]Sometimes the f-number can be found labeled "f-stop". Since this is ambiguous, we have chosen the term f-number for clarity.

[3]Technically, the pixel intensity is multiplied by the cosine of the angle of the ray to the film plane raised to the power of the **vignetting** parameter.

1, a strong "toe" region is added to the transfer curve so that low intensities gets pushed more towards black, but in a gentle (soft) fashion.

Compressing bright color components inherently moves them towards a less saturated color. Sometimes, very strong compressions can make the image in an unappealingly de-saturated state. The **saturation** parameter allows an artistic control over the final image saturation. 1.0 is the standard "unmodified" saturation, higher increases and lower decreases saturation.

The **gamma** parameter applies a display gamma correction. Be careful not to apply gamma twice in the image pipeline. This is discussed in more detail on page 3.

The **side_channel** and **side_channel_mode** is intended for OEM integrations of the shader, to support "interactive" tweaking as well as for the case where one wants to insert an output shader prior to the conversion to "display pixel values".

This is accomplished by applying two copies of the shader, one as lens shader, the other as output shader. The two shaders communicate via the "side channel", which is a separate floating point frame buffer that needs to be set up prior to rendering.

Valid values for **side_channel_mode** are:

- 0 - the shader is run normally as either lens- or output-shader.
- 1 - the lens shader will save the un-tonemapped value in the **side_channel** frame buffer. The output shader will re-execute the tone mapping based on the data in the side channel, *not* the pixels in the main frame buffer.
- 2 - the lens shader works as for option 1, the output shader will read the pixels from the **side_channel** frame buffer back into the main frame buffer. This is useful when one want to run third party output shaders on it that only support working on the main frame buffer.

One may wonder, if one wants to apply the tone mapping as a post process, why not skip applying the lens shader completely? The answer is twofold: First, by applying the lens shader (even though it's output is never used), one can *see* something while rendering, which is always helpful. Second, the mental ray *over-sampling* is guided by the pixels in the main frame buffer. If these are left in full dynamic range, mental ray may needlessly shoot thousands of extra samples in areas of "high contrast" that will all be tone mapped down to white in the final stage.

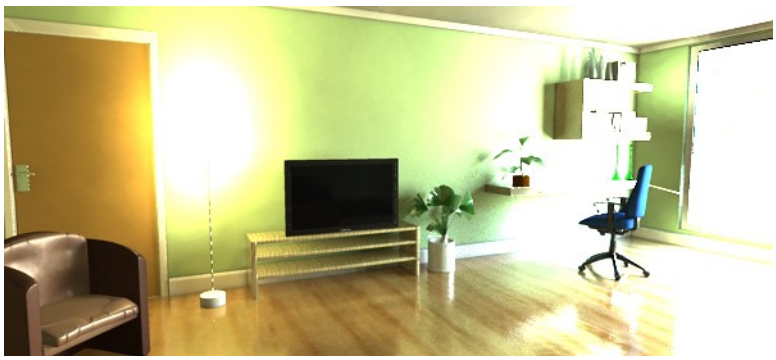
The **use_preview** and **preview** work exactly like in the *mia_exposure_simple* shader described above.

3.1.2.2 Examples

Lets walk through a practical example of tuning the photographic tone mapper. We have a scene showing both an indoor and outdoor area, using the sun and sky, *mia_material*, and a

portal light in the window. The units are set up such that the raw pixels are in candela per square meter.

The raw render, with no tone mapping, looks something like this:



No tone mapping

This is the typical look of a Gamma=1 un-tonemapped image. Bright areas blow out in a very unpleasing way, shadows are unrealistically harsh and dark, and the colors are extremely over-saturated.

Applying *mia_exposure_photographic* with the following settings:

```
"cm2_factor"          1.0,
"whitepoint"          1 1 1,
"film_iso"            100,
"camera_shutter"      100.0,
"f_number"            16.0,
"vignetting"          0.0,
"burn_highlights"     1.0,
"crush_blacks"        0.0,
"saturation"          1.0,
"gamma"               2.2,
```

...gives the following image as a result[4]:

[4]For speedy results in trying out settings for the tone mapping shaders, try the workflow with the "preview" parameter described on page 81.

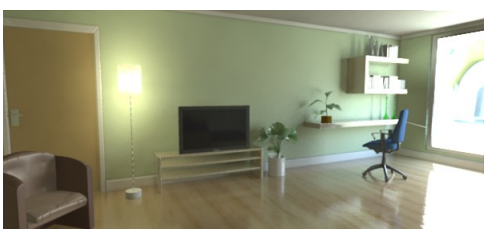
The "Sunny 16" result

The settings we used abide by the "Sunny 16" rule in photography; for an aperture of f/16, setting the shutter speed (in fractional seconds) equal to the film speed (as an ISO number) generates a "good" exposure for an outdoor sunny scene. As we can see, indeed, the outdoor area looks fine, but the indoor area is clearly underexposed.

In photography, the "film speed" (the ISO value), the aperture (f_number) and shutter time all interact to define the actual exposure of the camera. Hence, to modify the exposure you could modify either for the "same" result. For example, to make the image half as bright, we could halve the shutter time, halve the ISO of our film, or change the aperture one "stop" (for example from f/16 to f/22, see page 83 for more details).

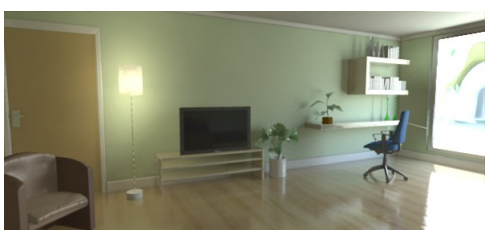
In a real world camera there would be subtle differences between these different methods, but with this shader they are mathematically equivalent.

f/8



f/4

Clearly the f/4 image is the best choice for the indoor part, but now the outdoor area is extremely overexposed. This is because we have the **burn_highlights** parameter at 1.0, which does not perform *any* compression of highlights.

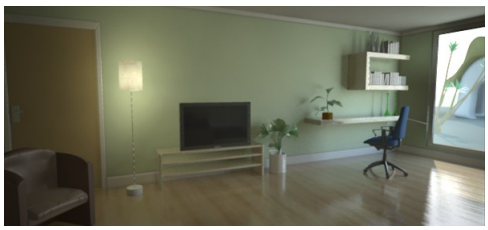


***burn_highlights** = 0.5*

***burn_highlights** = 0.0*

The left image subdues the overexposure some. The right image is using a **burn_highlights** of zero, which actually removes *any* over- exposure completely. However, this has the drawback of killing most contrast in the image, and while real film indeed performs a compression of the over-brights, no film exists that magically removes *all* overexposure. Hence, it is suggested to keep **burn_highlights** small, yet non-zero. In our example we pick the 0.5 value.

Real cameras have a falloff near the edges of the image known as "vignetting", which is supported in this shader by using the parameter of the same name:



***vignetting** = 5*

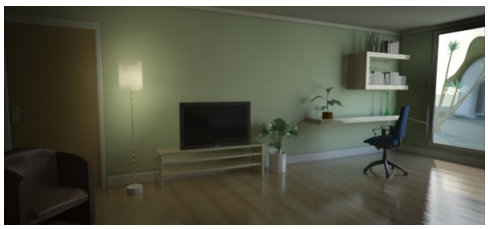
***vignetting** = 11*

The image on the right is interesting in that it "helps" our overexposed outdoors by the fact that it happens to be on the edge of the image and is hence attenuated by the vignetting. However, the left edge turns out too dark.

We will try a middle-of-the road version using a **vignetting** of 6 and we modify the **burn_highlights** to 0.25 and get this image:

***vignetting** = 6 **burn_highlights** = 0.25*

This image is nice, but it really lacks that feel of "contrast". To help this we play with the **crush_shadows** parameter:

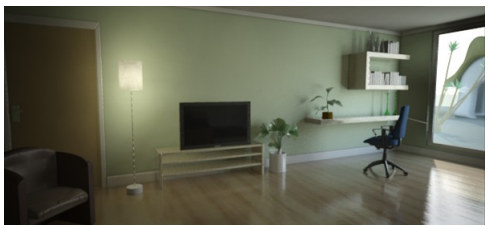


***crush_shadows** = 0.2*

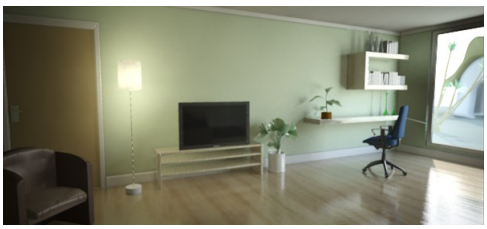
***crush_shadows** = 0.6*

The **crush_shadows** parameter deepens the lower end of the intensity curve, and we get some very nice contrast. However, since this image is already near the "too dark" end of the spectrum, it tends to show an effect of darkening the entire image a bit.

This can be compensated by changing the exposure. Lets try a couple of different **shutter** values, and prevent overexposure by further lowering our **burn_highlights** value:

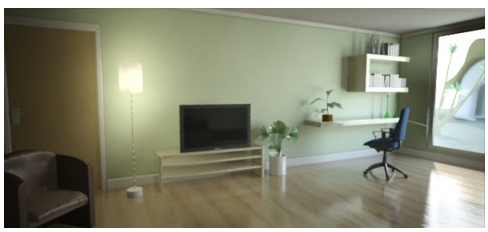


***shutter** = 50, **burn_highlights** = 0.1*

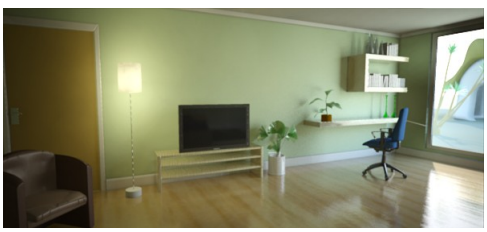


***shutter** = 30, **burn_highlights** = 0.1*

The final image is pretty good. However, since so much highlight compression is going on, we have lost a lot of color saturation by now. Lets try to finalize this by compensating:



***saturation** = 1.0*



***saturation** = 1.4*

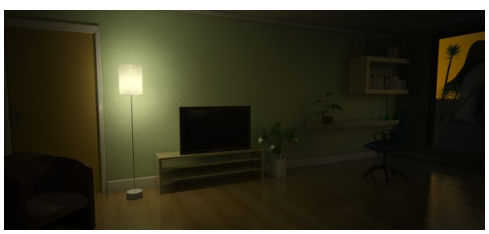
Now we have some color back, the image is fairly well balanced. We still see something exists outdoors. This is probably the best we can do that is still *physically correct*.

However, remember we were using the portal lights for the window? These have a non-physical "transparency" mode. This mode is intended to solve exactly this issue. Even though the result we have so far is "correct", many people intuitively expect to see the outdoor scene much more clearly. Since our eyes does such a magnificent job at compensating for the huge dynamic range difference between the sunlit outdoors and the much darker indoors, we expect our computers to magically do the same. Using the transparency feature of the portal lights, this can be achieved visually, without actually changing the *actual* intensities of any light going into the room:

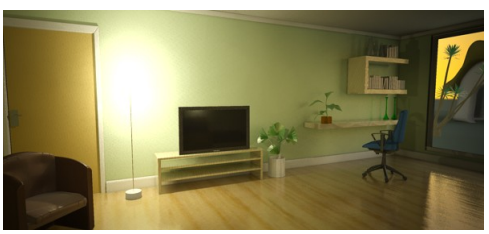
*A **transparency** of 0.5 in the portal light*

Now the objects outdoors are visible, while maintaining the contrast indoors. The light level indoors hasn't changed and still follows the real world values (unlike if we had put actual dark glass into the window, which would have attenuated the incoming light as well).

Now this is a mid day scene. What if we change the time of day and put the sun lower on the horizon? The scene will be much darker, and we can compensate by changing the exposure:



7 PM



With shutter 1/2 second

Keeping the same settings (left) with the new sun angle makes a very dark image. On the right, all we changed was the shutter time to 2 (half a second). This image has a bit of a yellow cast due to the reddish sunlight, as well as the yellowish incandescent lighting. To compensate, we set the white-point to a yellowish color:

Evening shot with adjusted white-point

The remaining issue is the overexposure around the lamp (although it would be there in a real photograph), so we make the final image with **burn_highlights** set to a very low value, yet keep it nonzero to maintain a little bit of "punch":

***burn_highlights** = 0.01*

In conclusion: Photography-related parameters help users with experience in photography make good judgements on suitable values. The intuitive "look" parameters allow further adjustment of the image for a visually pleasing result.

3.2 Depth of Field / Bokeh

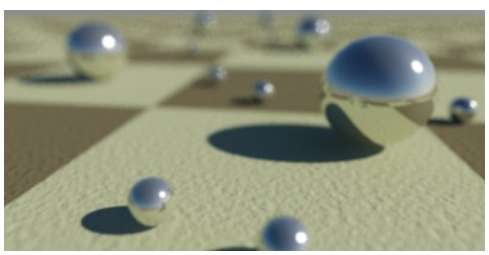
"Bokeh" is a Japanese term meaning "blur", that is often used to refer to the perceived "look" of out-of-focus regions in a photograph. The term "Depth of Field" (henceforth abbreviated DOF) does in actuality not describe the blur itself, but the "depth" of the region that *is in focus*. However, it is common parlance to talk about "DOF" while refering to the blur itself.

This shader is very similar to the **physical_lens_dof** in the physics library, but with more control on the actual *appearance* and quality of the blur.

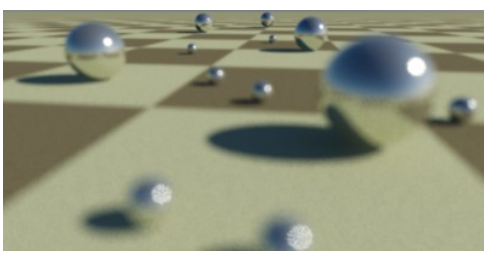
```
declare shader "mia_lens_bokeh" (
        boolean "on"          default on,
        scalar  "plane"       default 100.0,
        scalar  "radius"      default 1.0,
        integer "samples"     default 4,
        scalar  "bias"        default 1.0,
        integer "blade_count" default 0,
        scalar  "blade_angle" default 0,
        boolean "use_bokeh"   default off,
        color texture "bokeh"
    )
    version 4
    apply lens
    scanline off
end declare
```

on enables the shader.

plane is the distance to the focal plane from the camera, i.e. a point at this distance from the camera is completely in focus.



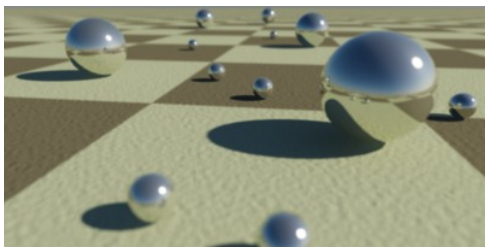
Focal point near camera



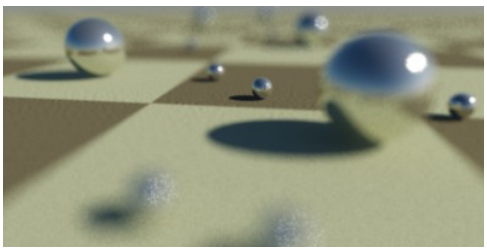
Focal point far away

radius is the radius of confusion. This is an actual measurement in scene units, and for a real-world camera this is approximately the radius of the *iris*, i.e. it depends on the cameras f-stop. But it is a good rule of thumb to keep it on the order of a couple of centimeters in size (expressed in the current scene units), otherwise the scene may come out unrealistic and be perceived as a minature[5].

[5]The diameter of the iris is the focal length of the lens (in scene units) divided by the f-number. Since this parameter is a radius (not a diameter) it is half of that, i.e. (focal_length / f_number) / 2.

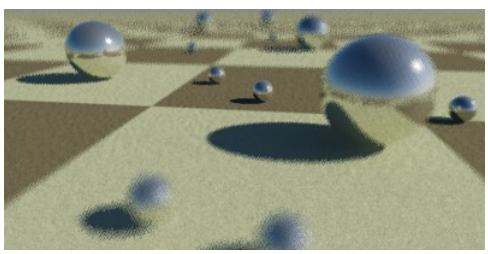


Small radius - deep DOF

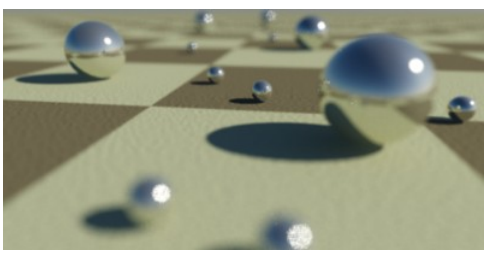


Large radius - shallow DOF

samples defines how many rays are shot. Fewer is faster but grainier, more is slower but smoother:

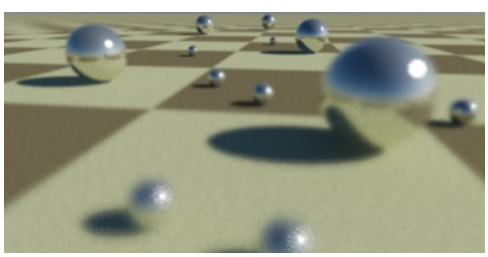


Few samples

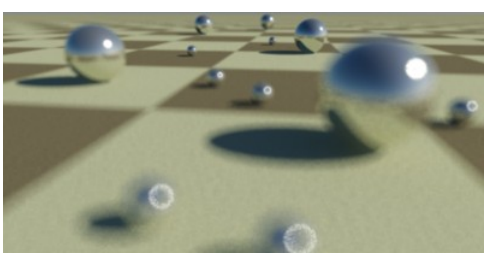


Many samples

When **bias** is 1.0, the circle of confusion is sampled as a uniform disk. Lower values push the sample probability towards the center, creating a "softer" looking DOF effect with a more "misty" look. Higher values push the sample probability towards the edge, creating a "harder" looking DOF where bright spots actually resolve as small circles.

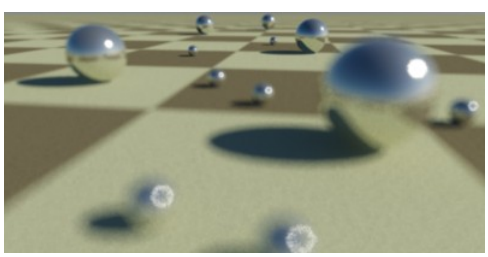


bias = 0.5

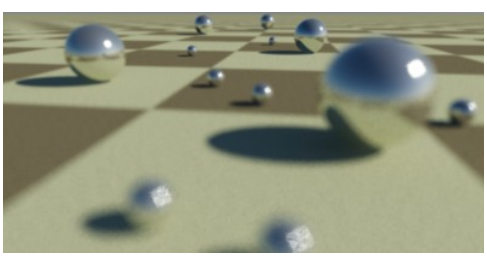


bias = 2.0

The **blade_count** defines how many "edges" the "circle" of confusion has. A zero value makes it a perfect circle. One can also set the angle with the **blade_angle** parameter, which is expressed such that 0.0 is zero degrees and 1.0 is 360 degrees.

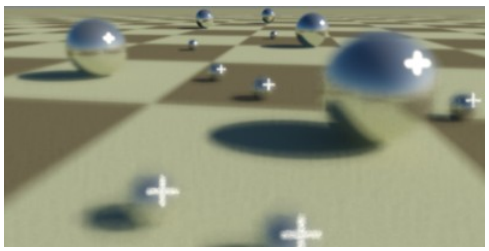


blade_count=6, angle=0.0, bias=2.0

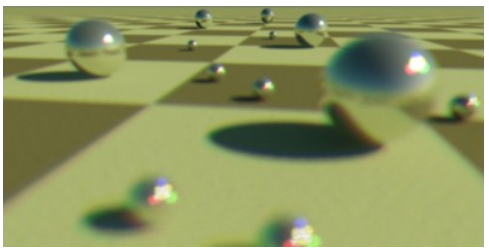


blade_count=4, angle=0.1, bias=2.0

The **use_bokeh** parameter enables the user of a specific **bokeh** map. When this parameter is used, the parameters **bias**, **blade_count** and **blade_angle** have no effect. The map defines the shape of the DOF filter kernel, so a filled white circle on a black background is equivalent to the standard blur. Generally, one need more samples to accurately "resolve" a custom bokeh map than the built-in bokeh shape, which has an optimal sampling distribution.



A cross shaped bokeh map



Chromatic Aberation via colored map

With **bias** at 1.0, **samples** at 4, **blade_count** at 0 and **use_bokeh** off, this shader renders an identical image to the old **physical_lens_dof** shader.

Chapter 4

General Utility Shaders

4.1 Round Corners

CG has a tendency to look "unrealistic" because edges of objects are geometrically sharp, whereas all edges in the real world are slightly rounded, chamfered, worn or filleted in some manner. This rounded edge tends to "catch the light" and create highlights that make edges more visually appealing.

The *mia_roundcorners* shader can create an illusion of "rounded edges" at render time. This feature is primarily intended to speed up modeling, where things like a table top need not be created with actual filleted or chamfered edges.

No round corners

Round corners

The shader perturbs the normal vector, and should be applied where bump maps are normally used, e.g. in the **bump** parameter if the *mia_material*.

The function is not a displacement, it is merely a shading effect (like bump mapping) and is best suited for straight edges and simple geometry, not advanced highly curved geometry.

```
declare shader vector "mia_roundcorners" (
        scalar "radius",
        boolean "allow_different_materials",
        shader  "bump",
        integer "bump_mode",
        vector  "bump_vector",
    )
    version 3
    apply texture
end declare
```

The **radius** parameter defines the radius of the rounding effect, in world space units.

When **allow_different_materials** is *off*, the rounding effect happens only against faces with the same material. If it is *on* the rounding effect happens against any face of any material.

The **bump** parameter is a passthrough to any other bump shader that handles additional bumping of the surface, for example **mib_bump_map2** or similar. *This parameter is only used if **bump_mode** is 0.*

To better support OEM integration, the new parameters **bump_mode** and **bump_vector** was introduced.

bump_mode defines the coordinate space of the **bump_vector**, as well as that of the return value of the shader itself (which is also a vector), and if it is interpreted as a "normal vector perturbation" or a whole new "normal vector"‘.

The following values are legal:

- 0: compatible mode. The old **bump** parameter is used in place of **bump_vector**. The return value is 0,0,0, and the shader actually perturbs the normal vector itself.
- 1: "add" mode in "internal" space
- 2: "add" mode in world space
- 3: "add" mode in object space
- 4: "add" mode in camera space
- 5: "set" mode in "internal" space
- 6: "set" mode in world space
- 7: "set" mode in object space
- 8: "set" mode in camera space

The "add" modes mean that the vector contains a *normal perturbation*, i.e. a modification that is "added" to the current normal. The "set" mode means that the actual normal is

replaced by the incoming vector, interpreted in the aforementioned coordinate space. Equally for output, an "add" mode implies that the shader returns a perturbation vector intended to be added to the current normal, and "set" mode implies that it returns a whole normal vector. In neither case does the shader actually modify the current normal by itself.

4.2 Environment Blur

4.2.1 Shader Functionality and Parameters

The *mia_envblur* shader works by accepting some *other* environment shader as input, which would usually be a shader that performs an environment lookup in an HDRI environment map. When the render starts, it performs a one-time setup and rasterizes the result of this environment shader in a special format into an internal pyramidal filter structure.

Then, when rendering proceeds, the shader can perform an extremely efficient blurring operation in this environment map in way that looks very similar to shooting an *extremely large* amount of glossy reflection rays into it; i.e. it yields a *perfectly smooth* result - quickly.

```
declare shader "mia_envblur" (
        shader  "environment",
        scalar  "blur"               default 0.0,
        boolean "mia_material_blur"  default on,
        integer "resolution"         default 200
    )
    version 1
    apply environment, texture
end declare
```

environment is the actual environment shader looked up by this shader. If this is not specified, the global camera environment is used.

blur is the amount of blur (range 0.0 to 1.0) applied on the image. If this is 0.0, the internal bitmap is bypassed and the *environment* shader is looked up directly, as if the *mia_envblur* shader was not there.

The blur amount can be automatically calculated setting *mia_material_blur* to *on*. Any reflective environment lookup performed by *mia_material* will cause the appropriate blur in *mia_envblur*. Leave **blur** at 0.0 in this case. This feature is described in more detail on page 100.

resolution is the resolution of the internal pyramidal data structure used for the filtering. The default value of 200 means that a map of 200 x 200 samples are taken and stored, for subsequent filtering. This should be set high enough to resolve the smallest feature in the environment map. 200 is generally enough for any still image - animations need higher resolutions (1000).

It is important to remember that *mia_envblur* does a *one time* rasterization of the **environment** shader at start up time. This means that that shader must be constant across the scene, and cannot be a complicated position-dependent blend of environment shaders. The environment can stil change *over time*, since the rasterization step is performed anew each frame.

4.2.2 Use Cases

The environment blur shader *mia_envblur* is intended to increase quality and performance of renderings that are largely reflecting the mental ray *environment* (i.e. that do *not* primarily reflect other *objects*).

The shader is primarily useful in product visualization renderings that are surrounded by an HDRI environment map for reflections, and also for visual effects work where one wants to help integrate CG objects in a real scene with the help of HDRI reflections, and want a smooth yet fast lookup.

The shader is *not* as useful for interior architectural renderings, since in those (enclosed) scenes, most reflection rays are bound to hit other objects; the purpose of this shader is to help reflection rays that do *not* hit other objects, i.e. the largest benefit is in an "open" scene[1].

Object reflecting an environment map

Here an example object is reflecting an environment map[2] with no glossiness (i.e. perfect mirror reflection). This looks fine, because there is no quasi-random sampling performed.

[1] It is also very useful combined with using **refl_falloff_dist** in *mia_material*.

[2] The "Gallileo's tomb" probe from www.debevec.org, which contains many small bright areas that tend to be troublesome in this context.

But what is we want to make a glossy reflection? If one simply uses the glossy reflection of *mia_material* one receives the following result:

Glossy reflection with 8 samples

It is obvious that the default 8 glossy reflection samples are nowhere near enough, especially with an environment map with such high contrasts in it. Trying with 100 glossy samples (at a large performance hit) the result is:

Glossy reflection with 100 samples

This is better, but still nowhere near a "smooth" glossy reflection. The 100 samples made the rendering an order of magnitude slower, and it is still not enough! What can we do?

What we want to is *not* look up the environment multiple times. Not 8 times, nor 100 times, but *once*, except we want that lookup to *already contain the desired blur*!

This is accomplished by enabling the **single_env_sample** parameter of *mia_material*and then apply *mia_envblur* as our environment shader and our "original" environment map as the **environment** parameter of *mia_envblur*.

Going back to our original 8 glossy samples, the following result can be rendered, very quickly:

Glossy reflections using mia_envblur

This looks much better, and renders much faster, but the level of blur is constant. A much more advanced way is to let *mia_envblur* derive the blur to apply by enabling *mia_material_blur*.

Assume we have applied the following map to the **refl_gloss** parameter of the *mia_material*:

Glossiness map

The resulting render, with the help of *mia_envblur* will be this:

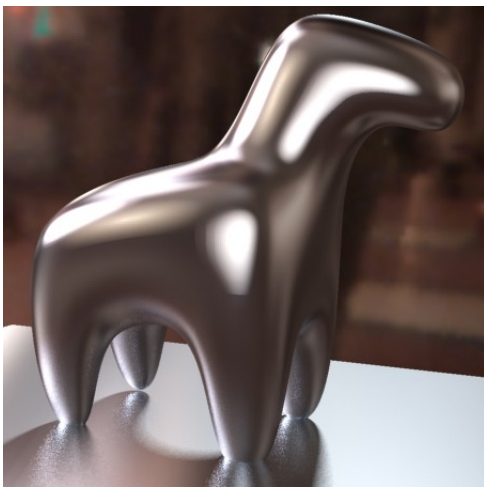
Mapped glossiness via mia_material_blur

Please note that *mia_envblur* shader only supports *isotropic* lookups, and will ignore any anisotropy parameters of *mia_material* when using it like this.

Also, do not forget to use the **single_env_sample** feature; just blurring the environment map is often not enough to combat the noise.

Keep in mind that *other objects* will still reflect in the traditional manner with multiple samples, and this feature only applies to environment lookups. Therefore it can be very advantageous to use **refl_falloff_dist** to limit "actual" reflections to nearby objects only, and let the environment take over for distant objects[3]:

*No **refl_falloff_dist***



***refl_falloff_dist** used*

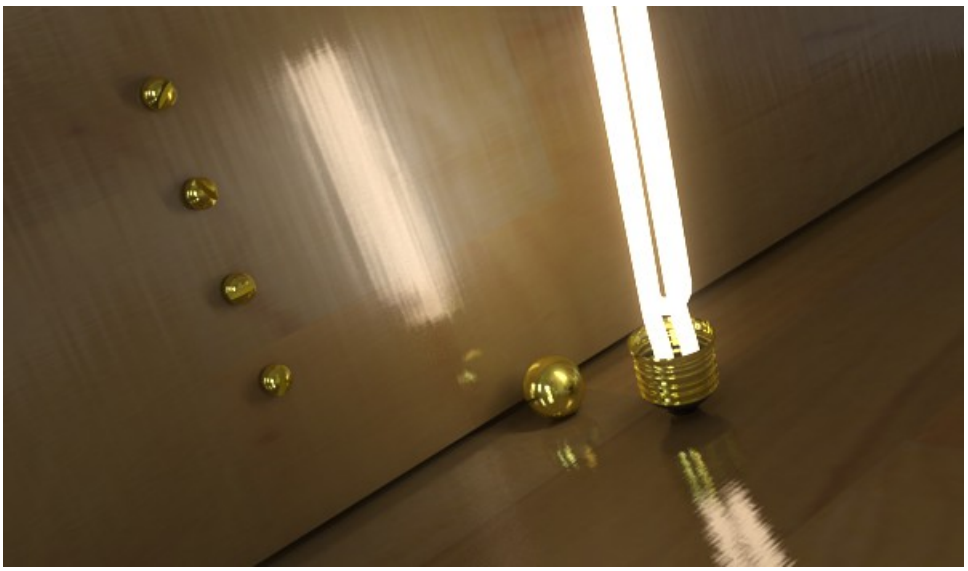
In the right image the legs of the horse only reflect very near to the floor, and conversely the horse only reflects the very nearest parts of the floor - the rest is environment reflections. This yields a faster result than actually tracing those reflections, and cuts down on the noise in the image.

4.3 Light Surface

4.3.1 Shader Functionality and Parameters

The *mia_light_surface* shader is primarily intended to help creating physically plausible renders of the "visible" portion of a light source - the actual tube in a fluorescent tube, the actual bulb in a light bulb, etc. while still using a traditional CG "light" to create the illumination of the scene (see the use cases described below).

[3]This is particularly true in a visual effects context.



An example of using mia_light_surface

In the image above, actual illumination comes from a long thin rectangular area light, which is set not to cause any specular highlights. The visible "glow" of the fluorescent tube is set be visible in reflections (and hence become our much more accurate "highlight") but still be invisible to FG rays, so it will not be incorrectly picked up as additional light.

The *mia_light_surface* shader can either provide a color all by itself, or derive the color from an existing (set of) light(s) in the scene.

The shader itself only returns a color and does not do any *other shading* per se. A tip is to use it plugged into the **additional_color** of *mia_material*.

These are the parameters of the *mia_light_surface* shader:

```
declare shader "mia_light_surface" (
        color "color"        default 1 1 1,
        scalar      "intensity"    default 1.0,
        scalar      "fg_contrib"   default 0.0,
        scalar      "refl_contrib" default 0.0,
        boolean     "use_lights",
        scalar      "lights_multiplier" default 1.0,
        vector      "lights_eval_point" default 0.0 0.0 0.0,
        array light "lights"
    )

    version 3
    apply texture
end declare
```

color is the overall color, and applies both to built in light or light derived from light sources.

intensity is the intensity of the "built in" light, i.e. the surface will appear to the camera to have an intensity of **color** multiplied by **intensity** (assuming **use_lights** is off - see below).

fg_contrib is how much of the light that is visible to FG rays, and **refl_contrib** how much that is visible to reflection rays.

When **use_lights** is on, the lights listed in the **lights** array are polled and their intensity (multiplied by the **lights_multiplier**) is *added* to the output dictated by the *intensity* parameter; i.e. if L is the output of all lights in the *lights* list, the final output color of the shader is:

$color * (L * lights\_multiplier + intensity)$

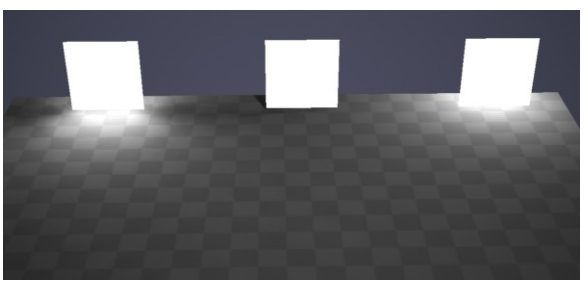
When **lights_eval_point** is 0,0,0 the intensity of the light is evaluated (with shadows disabled) at the point in 3D space that is being shaded. Since this may vary in an undesirable way for a light that has an IES profile, one can specify an explicit point at which the light color is evaluated. This point is in the coordinate space of *the light*.

4.3.2 Use Cases

4.3.2.1 Illumination in General

In the real world every light emitting object is visible, has an area, and emits light from that area. When using FG, *mental ray* will treat any surface that adds light energy into the scene as if it was a light source. However, to get the best possible quality out of this, one need very high FG settings, with long render times.

Light sources in computer graphics can be either point sources or area lights, and the area lights themselves may, or may not, be visible in the rendering. In most cases It is simply more efficient to use actual light sources:



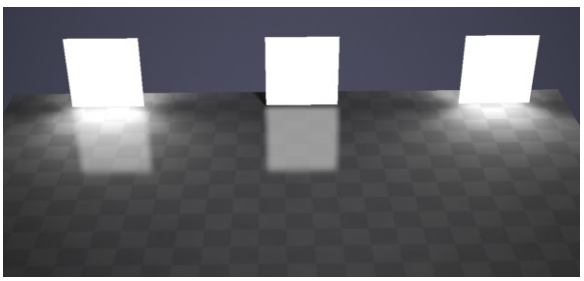
Emitting FG, Not emitting FG, And using a point light

The image above shows 3 patches that are all using the *mia_light_surface* above a checkered plane. The leftmost patch has it's **fg_contrib** set to 1.0 (and is hence illuminating the floor),

the other two has it set to 0.0. But the rightmost patch has a point light hidden behind it.

At close distances (or with very large light sources, like the entire outdoor sky) FG can illuminate objects just fine with very good quality. At long distances or with small sources, using an explicit light source is much more efficient.

mia_light_surface gives separate control if the object should be "seen" by reflection rays or FG rays (and how much) via the **fg_contrib** and **refl_contrib** parameters. This image uses a slightly reflective checkered plane to illustrate this. No light sources are used behind these patches:



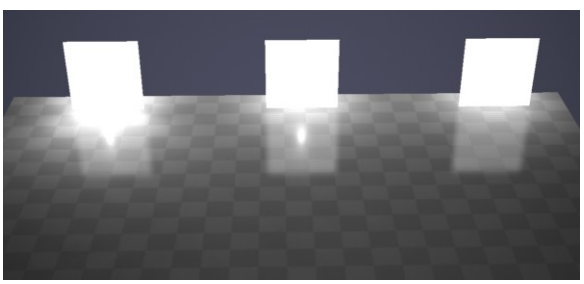
Visibility to FG and reflections

The leftmost patch is visible both to FG (illuminates the plane) and reflection (reflects in the floor). The center patch does not illuminate (**fg_contrib** is zero) and the rightmost does not reflect (**refl_contrib** is zero).

4.3.2.2 Highlights vs. Reflections

In the real world, the visible light emitting objects are both visible to any camera photographing the scene, as well as reflect (glossily) in other objects. In computer graphics, light sources tend to be invisible to the camera, and their reflections are "cheated" with the help of "highlights" by material shaders.

The *mia_material* supports a protocol for light sources to tell it if they should generate highlights or not. This is implemented by most OEM applications light sources, look for flags like "Affect Specular" or "Emit Specular" or similar on the lights.



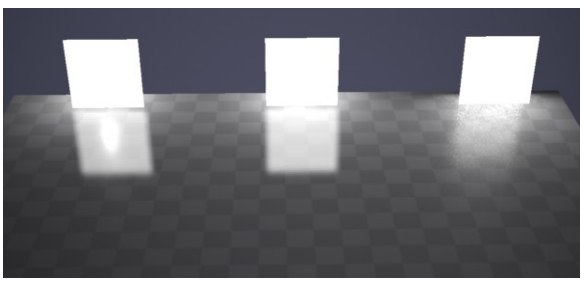
Light hidden behind patch with various flags.

There is a light hidden behind each of the patches in the image above. The leftmost is emitting both specular- and diffuse light. The middle one is only emitting specular light, and the rightmost is only emitting diffuse light. Furthermore, the rightmost patch has it's **fg_contrib** set to zero.

As we can see, the leftmost patch creates too much illumination, since we get illumination both from the light and from the patch. The center patch generates the strange effect of a small "highlight" (our light source is just a point light) inside the reflection of what we are trying to sell as "the visible light".

In this case this effect is distracting, so we want it disabled - like we have done on the light on the right. That is the way to go; let the surface handle the reflection, not the illumination, and let the light handle the illumination, not the reflection (i.e. no traditional "specular highlight").

An alternative, when using area lights, is to allow the area light to create the specular highlights:



Area light specular highlights can also "work"

This illustrates the alternative. The leftmost patch again uses a point light with specular on. This is what we do *not* want, since it looks strange. The other two patches use an area light. The middle patch has the area lights specularity turned off and the patch has **refl_contrib** set to 1.0, the right has the lights specular enabled, and a **refl_contrib** of zero.

As we can see, both variants on the right side "work", so it is a matter of choice which method is used for area lights.

However, the middle method tends to be preferred, because often one is using a geometrically simple area light (sphere, rectangle, etc.) in place of an *object* that is actually geometrically complex (a neon sign, for example). In those cases one *definitely* want to use the method in which the object is reflected, and the light does *not* emit specular light:

Complex geometry require real reflections

Both neon signs use a rectangular area light for their illumination, and both cast nice area shadows (as witnessed by the small spheres). But the sign on the left has a blatantly rectangular "reflection", which does not make any sense. For this use, definitely use the method on the right side, with the lights specularity off, and the surface's **refl_contrib** nonzero.

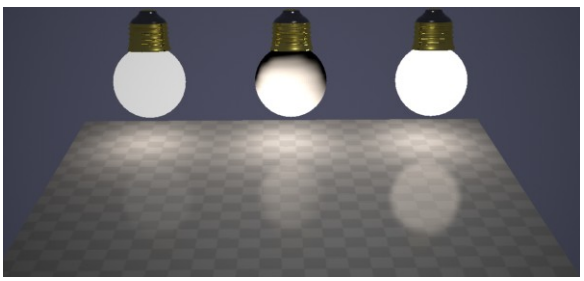
4.3.2.3 Complex Light Distribution

In a real world luminaire, complex interactions between the bulb, the reflector, and various occluders determine the final distribution of light flying off into the scene.

Naturally, one do not want to consider all those minute details just to make a rendering of a desk lamp, and indeed, the problem comes pre-solved in the form of measured data (i.e. IES or ELUMDAT files).

The *mia_light_surface* shader can get its intensity from a light. This saves the user the time of keeping the intensity and color of the "visible surface" in sync with the intensity and color of the light source. If the **use_lights** is on and some light is passed in the **lights** parameter, this happens automatically.

This works fine for isotropic point lights and similar, but when using measured light data, one can run into an issue:



Picking up intensity/color from the lights

The above scene contains three spheres. Lets assume they are the models of the light bulbs of some desk lamp. They each contain a light that has an IES profile fitting that *luminaire* applied.

The leftmost sphere uses *mia_light_surface* with a manual setting, i.e. **use_lights** is off. The user is responsible for making sure the intensity and color of the surface is "correct", as well as manually keeping it in sync with any animated intensity changes etc.

The center sphere uses **use_lights**, but we get a very strange effect from it; half our lightbulb is dark. This is because the IES profile we chose only emits light in the down direction. Since the light source is in the center of the sphere, only the lower half of the sphere will gather any intensity values off the light!

Clearly this is not desirable; the IES profile is a compound effect applying to the *entire luminaire*, and should not be "painted" onto the light bulb itself!

The solution is to use **lights_eval_point** as is done on the rightmost sphere. Here a point just in front of the light is picked (in the lights own coordinate space) as the "evaluation point". Each point on the sphere (regardless of its location) will have the intensity "measured" at that point in space.

This gives the sphere a color that automatically follows any intensity- and color-changes of the light, yet is uniform across the surface of the sphere. This sphere will reflect correctly in a more plausible way than a traditional "highlight" would.

In conclusion: The *mia_light_surface* shader allows...

- ...creating a surface as a "stand in" for an existing light, that "looks" bright, without emitting additional light into the scene.
- ...controlling the amount of reflection and light actually introduced into the scene.
- ...driving the appearance of the surface from an existing light source.
- ...more physically correct "highlights" by replacing them with glossy reflections of actual high-intensity objects.

Another example using mia_light_surface

Chapter 5

Advanced Topics

This section is mainly of interest to OEM integrators of *mental ray* shaders into applications.

5.1 mia_material API

The *mia_material* exposes features that allows a much deeper integration into OEM applications than ever before. Most notably it exports:

- an interface for obtaining sub-components of the rendered result (diffuse, reflections, etc.).
- an interface for obtaining separate "diffuse" and "specular" lighting.

This is implemented as a C API for shader developers, utilizing *shader states*[1] for passing the information in and out of the *mia_material*.

The keys and structs used are available in the public file `mia_material_api.h` which is listed in its entirety below.

5.1.1 Obtaining Sub-Components of the Rendering

The new *mia_material_x* already has multiple outputs. Hence, the interface described here is an alternate method to get to the outputs.

To obtain independent results from the various shading components with this method one must *wrap* the *mia_material* shader in some other shader that sets up the appropriate shader state, calls *mia_material* and then reads the info stored in the shader state for further processing.

[1] See the book *Programming mental ray* for more details.

Setting up the shader state structure is the responsibility of this *wrapping shader*. The structure member `struct_size` must be initialized to the size of the struct (for version control) and `in_use` initialized to `miFALSE`.

Two different structs are supported, the compatible `mia_material_api_storage` and the new `mia_material_x_api_storage`. Since these structs are different size, the shader detects which is used by the `struct_size` value.

Then *mia_material* is called, for example with the help of `mi_call_shader_x()`.

After the call the `in_use` parameter is inspected. If true, the rest of the structure contains valid values.

A sample shader saving into separate *mental ray* frame buffers is listed on page 115.

5.1.2 Defining Characteristics of Light Sources

This interface allows light shaders to inform the *mia_material* if they are emitting specular or diffuse light (or both, which is the default).

This shader state is set up *by* the *mia_material* and filled with defaults. Light shaders should only test for the presence of the shader state. If it is missing, the light shader needs to take no further action. But If the shader state *exists*, the light shader can modify the structures values, i.e. set the `affects_diffuse` or `affects_specular` scalars to suitable values.

See listing below for the meaning of each parameter:

5.2 mia_material_api.h File Listing

Here follows a complete listing of the interface header file:

```
/**************************************************************************
* Copyright 1986-2007 by mental images GmbH, Fasanenstr. 81, D-10623 Berlin,
* Germany. All rights reserved.
**************************************************************************
* Created:  06.04.12
* Module:   architectural
* Purpose:  the architecture & design material PUBLIC API
*
* Exports:
*
*      mia_material_api_*()
*
* History:
*
* Description:
```

```
*****************************************************************************/

#define miA_MATERIAL_API_STORAGE   "miXMST"

/*
    Protocol for extracting arbitrary outputs from the mia_material:

    1. Create a shader state named after the macro miA_MATERIAL_API_STORAGE
       pointing to a struct mia_material_api_outputs

    2. Set it's struct_size to sizeof(mia_material_api_outputs);

    3. Set in_use to miFALSE; No further initialization is needed.

    4. Call the mia_material shader.

    5. Upon return of mia_material, see if in_use is miTRUE.
       If so, the structure in the shaderstate will be filled in with the
       topmost shaders values.
*/

typedef struct  {
    /* Set before calling mia_material shader: */
    miUint     struct_size;  /* Set to sizeof() struct */
    miBoolean  in_use;       /* Set to miFALSE */
    /* Return values after calling mia_material shader */
    miScalar   opacity;      /* scalar opacity */
    miColor    indir_result; /* Indirect shading (FG and GI) */
    miColor    diff_result;  /* Diffuse shading */
    miColor    spec_result;  /* Specular/Highlights */
    miColor    tran_result;  /* Translucency */
    miColor    refl_result;  /* Reflections */
    miColor    refr_result;  /* Refractions */
    miColor    add_result;   /* "Additional color" */
    miColor    ao_result;    /* AO contribution only */
    miColor    diff_level;   /* Actually used diffuse color/level */
    miColor    refl_level;   /* Actually used reflection color/level */
    miColor    refr_level;   /* Actually used refraction color/level */
    miColor    tran_level;   /* Actually used translucency color/level */
    miRay_type type;       /* Ray type for the stored data */
} mia_material_api_storage;

/* New: the return structure of mia_material_x, interface version 15 (or higher) */

typedef struct {
    miColor   result;

    miColor   diffuse_result;
    miColor   diffuse_raw;
    miColor   diffuse_level;

    miColor   spec_result;
    miColor   spec_raw;
```

```
    miColor   spec_level;

    miColor   refl_result;
    miColor   refl_raw;
    miColor   refl_level;

    miColor   refr_result;
    miColor   refr_raw;
    miColor   refr_level;

    miColor   tran_result;
    miColor   tran_raw;
    miColor   tran_level;

    miColor   indirect_result;
    miColor   indirect_raw;
    miColor   indirect_cooked;
    miColor   ao_raw;

    miColor   add_result;

    miColor   opacity_result;
    miColor   opacity_raw;
    miScalar  opacity;

    /* Extra space in the struct for padding, never accessed */
    miColor   spare[2];
} mia_material_x_return;

/* New mia_material_x_return struct */
typedef struct  {
    /* Set before calling mia_material shader: */
    miUint     struct_size;  /* Set to sizeof() struct */
    miBoolean  in_use;       /* Set to miFALSE */
    miRay_type type;       /* Ray type for the stored data */

    /* Return values after calling mia_material shader */
    mia_material_x_return    output;
} mia_material_x_api_storage;

#define miA_MATERIAL_API_LIGHTDATA "miXMLD"

typedef struct {
    /* All below set up by mia_material prior to calling each light */
    miUint     struct_size;  /* Set to sizeof() struct, for versioning */

    /* Read only's: DO NOT change in light shader */

    miScalar   glossiness;   /* Light shader can make decisions based on glossiness */
    miScalar   importance;   /* Light shader can make decisions based on importance */

    /* Modify as needed in the light shader */
```

```
    /* How much does this light affect diffuse and specular? */
    miScalar   affect_diffuse;  /* defaults to 1.0 */
    miScalar   affect_specular; /* defaults to 1.0 */

    /* Is this the mr Sun? (should only be set by the mr sun) */
    miBoolean  is_mr_sun;       /* defaults to miFALSE */
    /* If this is a visible area light, but it still desires to
       get a highlight, set force_specular to true */
    miBoolean  force_specular;  /* defaults to miFALSE */
} mia_material_api_lightdata;
```

5.2.1 Sample Shader Source

Sample shader (code snippet) for saving the output of a *mia_material* into frame buffers:

```
#include <shader.h>

/* Parameter struct */
typedef struct [
    miTag mtl;  /* Tag of mia_material shader instance */
} mia_material_out_wrapper;

DLLEXPORT miBoolean mia_material_out_wrapper(
    miColor *result,
    miState *state,
    mia_material_out_wrapper *param)
{
    /* Tag of actual mia_material (sent in as parameter to wrapper shader) */
    miTag mtl = *mi_eval_tag(&param->mtl);
    /* Struct to store the data in */
    mia_material_x_api_storage mtldata; /* NEW: Use _x variant */
    /* We also need a pointer */
    mia_material_x_api_storage *dp;
    /* Need the key len */
    static int klen = sizeof(miA_MATERIAL_API_STORAGE);

    /* Initialize the struct */
    mtldata.struct_size = sizeof(mtldata);
    mtldata.in_use      = miFALSE;

    /* Create/set the shader state */
    mi_shaderstate_set(state, miA_MATERIAL_API_STORAGE, &mtldata, sizeof(mtldata), 0);

    /* Call shader */
    mi_call_shader_x(result, miSHADER_MATERIAL, state, mtl, NULL);


    /* Check if the data is there */
    dp = mi_shaderstate_get(state, miA_MATERIAL_API_STORAGE, &klen);
```

```
    /* So, was valid data written? */
    if (dp && dp->in_use)
    {
        /* Diffuse to fb #10 */
        mi_fb_put(state, 10, &dp->output.diffuse_result);

        /* Reflection to fb #11 */
        mi_fb_put(state, 11, &dp->output.refl_result);

        /* Refraction to fb #12 */
        mi_fb_put(state, 12, &dp->output.refr_result);

        /* etc. */
    }

    return miTRUE;
}
```

Sample light shader (code snippet only) to set a light to “specular only”

```
    static int klen = sizeof(miA_MATERIAL_API_LIGHTDATA);

    mia_material_api_lightdata* ld = (mia_material_api_lightdata*)
             mi_shaderstate_get(state, miA_MATERIAL_API_LIGHTDATA, &klen);

    /* Is there a shader state? */
    if (ld)
    {
        /* Our light is specular-only (no diffuse) */
        ld->affect_diffuse  = 0.0f;
        ld->affect_specular = 1.0f;
     }
```