mental ray Production Shader Library

Document version 3.8.0.0
November 20, 2008

Table of Contents

Chapter 1

Introduction

1.1 About the Library

The **production** shader library contains a set of shaders aimed at production users of mental ray.

The library contains a diverse collection of shaders, ranging from small utilities such as ray type switching and card opacity to large complex shaders such as the fast 2d motion blur shader.

All of the shaders exist in the `production.dll` on Windows or `production.so` on other platforms, and the shaders are declared in `production.mi`.

To use the shaders from standalone mental ray, include the following in your .mi files:

```
link     "production.so"
$include "production.mi"
```

For embedded versions of mental ray consult your application specific documentation.

Chapter 2

Motion Blur Shaders

Motion Blur

2.1 Introduction

Real world objects photographed with a real world camera exhibit *motion blur*. When rendering in mental ray, one has several choices for how to achieve these trade-offs. There are three principally different methods:

- Raytraced 3d motion blur
- Fast rasterizer (aka. "rapid scanline") 3d motion blur
- Post processing 2d motion blur

Each method has it's advantages and disadvantages:

2.1.1 Raytraced 3d Motion Blur

This is the most advanced and "full featured" type of motion blur. For every spatial sample within a pixel, a number of temporal rays are sent. The number of rays are defined by the "time contrast" set in the options block, the actual number being 1 / "time contrast" (unless the special mode "fast motion blur" is enabled, which sends a single temporal sample per spatial sample[1]).

Since every ray is shot at a different time, *everything* is motion blurred in a physically correct manner. A flat moving mirror moving in it's mirror plane will only show blur on it's edges - the mirror reflection itself will be stationary. Shadows, reflections, volumetric effects... everything is correctly motion blurred. Motion can be multi-segmented, meaning rotating object can have motion blur "streaks" that go in an arc rather than a line.

But the trade-off is rendering time, since full ray tracing is performed multiple times at temporal sampling locations, and every ray traced evaluates the full shading model of the sample.

With this type of motion blur, the render time for a motion blurred object increases roughly linearly with the number of temporal samples (i.e. 1 / "time contrast").

2.1.2 Fast Rasterizer (aka "Rapid Scanline") Motion Blur

The rasterizer is the new "scanline" renderer introduced in mental ray 3.4, and it performs a very advanced subpixel tessellation which decouples *shading* and *surface sampling*.

The rasterizer takes a set of shading samples and allows a low number of shading samples to be re-used as spatial samples. The practical advantage is that fast motion blur (as well as extremely high quality anti aliasing, which is why one should generally use the rasterizer when dealing with hair rendering) is possible.

The trade-off is that the shading samples are re-used. This means that the flat mirror in our earlier example will actually smear the reflection in the mirror together with the mirror itself. In most practical cases, this difference is not really visually significant.

[1] "Fast motion blur" mode is enabled by setting "time contrast" to zero and having the upper and lower image sampling rates the same, i.e. something like "samples 2 2". Read more about "Fast motion blur" mode at http://www.lamrug.org/resources/motiontips.html

The motion blur is still fully 3d, and the major advantage is that the rendering time does *not* increase linearly with the number of samples per pixel, i.e. using 64 samples per pixel is *not* four times slower than 16 samples. The render time is more driven by the number of shading samples per pixel.

2.1.3 Post Processing 2d Motion Blur

Finally we have motion blur as a post process. It works by using pixel motion vectors stored by the rendering phase and "smearing" these into a visual simulation of motion blur.

Like using the rasterizer, this means that features such as mirror images or even objects seen through foreground transparent object will "streak" together with the foreground object. Furthermore, since the motion frame buffer only stores one segment, the "streaks" are always straight, never curved.

The major advantage of this method is rendering speed. Scene or shader complexity has no impact. The blur is applied as a mental ray "output shader", which is executed after the main rendering pass is done. The execution time of the output shader depends on how many pixels need to be blurred, and how far each pixel needs to be "smeared".

2.2 Considerations when Using 2d Motion Blur

2.2.1 Rendering Motion Vectors

The scene must be rendered with the *motion vectors* frame buffer enabled and filled with proper motion vectors. This is accomplished by rendering with motion blur turned *on*, but with a shutter length of *zero*.

```
shutter 0 0
motion on
```

Note the order: "motion on" must come after "shutter 0 0".

In older versions of mental ray the following construct was necessary:

```
motion on
shutter 0 0.00001
time contrast 1 1 1 1
```

Which means:

- Setting a very very short but non-zero shutter length
- Using a time contrast of 1 1 1 1

If there is a problem and no motion blur is visible, try the above alternate settings.

2.2.2 Visual Differences - Opacity and Backgrounds

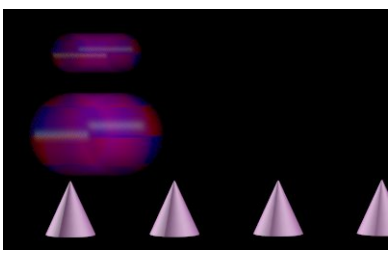
The 3d blur is a true rendering of the object as it moves along the time axis. The 2d blur is a *simulation* of this effect by taking a still image and *streaking* it along the 2d on-screen motion vector.

It is important to understand that this yields slightly different results, visually.

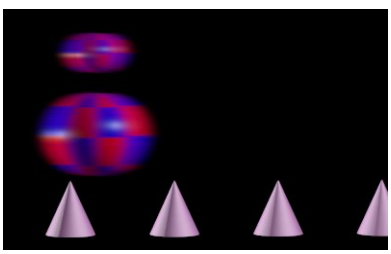
For example, an object that moves a distance equal to *it's own width* during the shutter interval will effectively occupy each point along it's trajectory 50 percent of the time. This means the motion blurred "streak" of the object will effectively be rendered 50 percent transparent, and any background behind it will show through accordingly.

In contrast, an object rendered with the 2d motion blur will be rendered in a stationary position, and then these pixels are later smeared into the motion blur streaks. This means that over the area the object originally occupied (before being smeared) it will still be completely opaque with no background showing through, and the "streaks" will fade out in both directions from this location, allowing the background to show through on each side.

The end result is that moving objects appear slightly more "opaque" when using 2d blur vs true 3d blur. In most cases and for moderate motion, this is not a problem and is never perceived as a problem. Only for extreme cases of motion blur will this cause any significant issues.



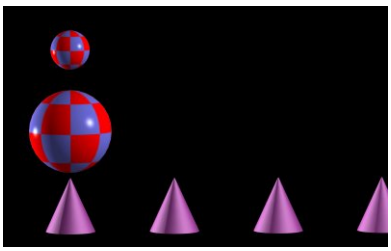
3D blur



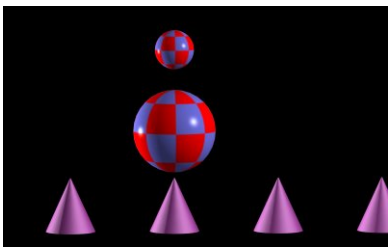
2D blur

2.2.3 Shutters and Shutter Offsets

To illustrate this behavior of the mental ray shutter interval we use these images of a set of still cones and two moving checkered balls that move such that on frame 0 they are over the first cone, on frame 1 they are over the second cone, etc:



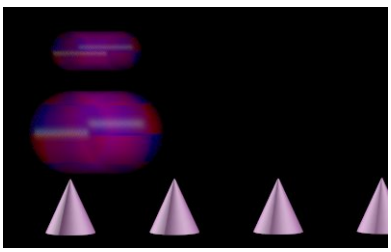
Object at t=0



Object at t=1

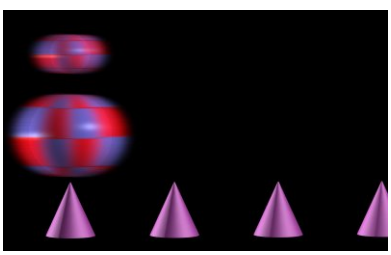
When using 3d motion blur, the mental ray virtual cameras shutter open at the time set by "shutter offset" and closes at the time set by "shutter".

Here is the result with a shutter offset of 0 and a shutter of 0.5 - the objects blur "begins" at t=0 and continues over to t=0.5:



3d blur, shutter open at t=0 and close at t=0.5

Now when using the **2d motion blur** it is important to understand how it works. The frame is rendered at the "shutter offset" time, and then those pixels are streaked *both forwards and backwards* to create the blur effect, hence, with the same settings, this will be the resulting blur:

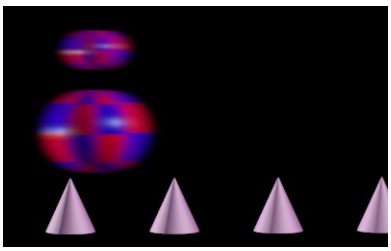


2d blur, shutter offset=0, mip_motionblur shutter=0.5

Note this behavior is *different* than the 3d blur case! If one need to mix renderings done with both methods, it is **important to use such settings to make the timing of the blur identical**.

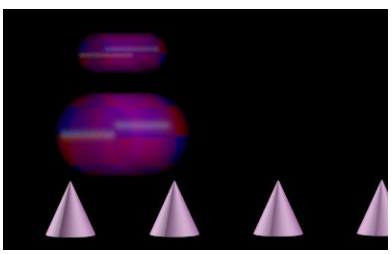
This is achieved by changing the "shutter offset" time to the desired *center time* of the blur (in our case t=0.25), like this:

```
shutter 0.25 0.25
motion on
```



2d blur, shutter offset=0.25, mip_motionblur shutter=0.5

Note how this matches the 3d blurs setting when shutter offset is 0. If, however, the 3d blur is given the shutter offset of 0.25 and shutter length 0.5 (i.e. a "shutter 0.25 0.75" statement), the result is this:



3d blur, shutter open at t=0.25 and close at t=0.75

Hence it is clear that when compositing renders done with different methods it is *very important* to keep this difference in timing in mind!

2.3 The mip_motionblur Shader

The *mip_motionblur* shader is a mental ray *output shader* for performing 2.5d[2] motion blur as a post process.

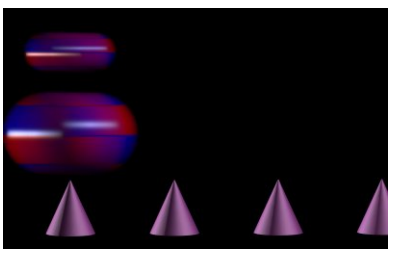
```
declare shader "mip_motionblur" (
        scalar  "shutter",
        scalar  "shutter_falloff",
        boolean "blur_environment",
        scalar  "calculation_gamma",
        scalar  "pixel_threshold",
        scalar  "background_depth",
        boolean "depth_weighting",
```

[2]Called "2.5d" since it also takes the Z-depth relationship between objects into account

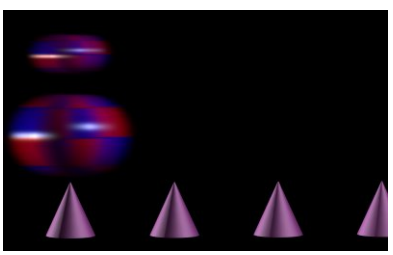
```
            string  "blur_fb",
            string  "depth_fb",
            string  "motion_fb",
            boolean "use_coverage"
        )
        version 1
        apply output
    end declare
```

shutter is the amount of time the shutter is "open". In practice this means that after the image has been rendered the pixels are smeared into streaks in both the forward and backward direction, each a distance equal to half the distance the object moves during the shutter time.

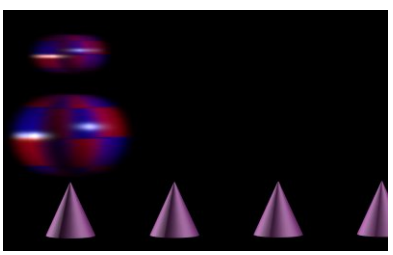
shutter_falloff sets the drop-off speed of the smear, i.e. how quickly it fades out to transparent. This tweaks the "softness" of the blur:



falloff = 1.0



falloff = 2.0



falloff = 4.0

Notice how the highlight is streaked into an almost uniform line in the first image, but tapers off much more gently in the last image.

It is notable that the *perceived length* of the motion blur diminishes with increased falloff, so one may need to compensate for it by increasing the shutter slightly.

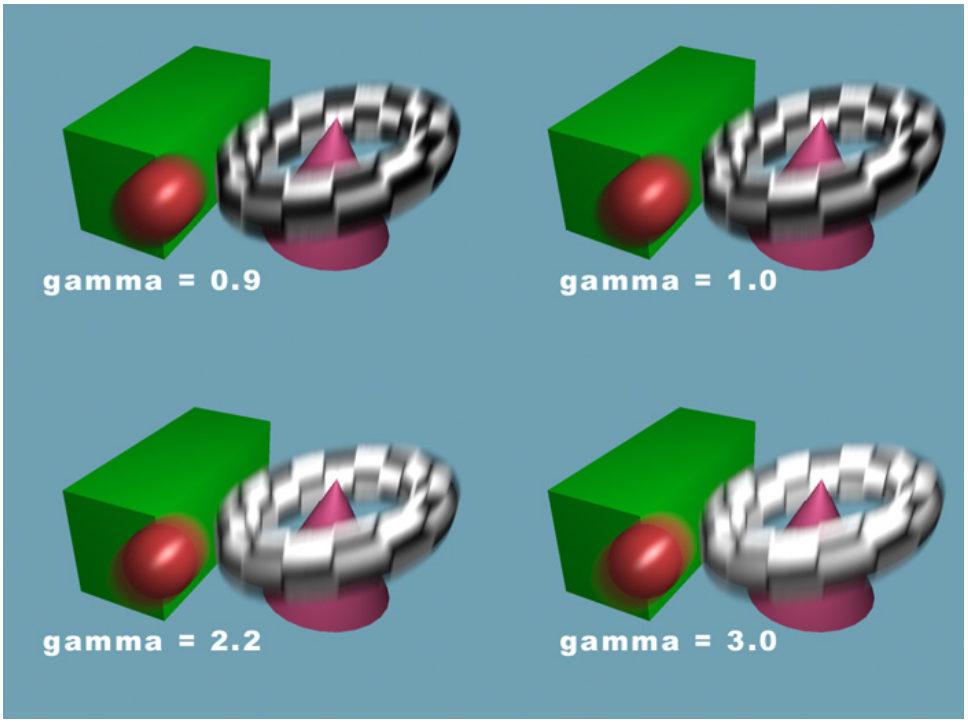
Therefore, falloff is especially useful when wanting the effect of over-bright highlights "streaking" convincingly: By using an inflated shutter length (above the cinematic default of 0.5) and a higher falloff, over-brights have the potential to smear in a pleasing fashion.

blur_environment defines if the camera environment (i.e. the background) should be blurred by the cameras movement or not. When *on*, pixels from the environment will be blurred, and when *off* it will not. Please note that camera driven environment blur *only* works if "scanline" is off in the options block.

If the background is created by *geometry* in the scene, this setting does not apply, and the backgrounds blur will be that of the motion of said geometry.

calculation_gamma defines in what gamma color space blur calculations take place. Since mental ray output shaders are performed on written frame buffers, and these buffers (unless floating point) already have any gamma correction applied, it is important that post effects are applied with the appropriate gamma.

If you render in linear floating point and plan to do the proper gamma correction at a later stage, set **calculation_gamma** to 1.0, otherwise set it to the appropriate value. The setting can also be used to artistically control the "look" of the motion blur, in which a higher gamma value favors lightness over darkness in the streaks.



Various gammas

Notice how the low gamma examples seem much "darker", and how the blur between the green box and red sphere looks questionable. The higher gamma values cause a smoother blend and more realistic motion blur. However, it cannot be stressed enough, that if gamma correction is applied *later in the pipeline* the calculation_gamma parameter should be kept at 1.0, unless one really desires the favoring of brightness in the blur as an artistic effect.

The **pixel_threshold** is a minimum motion vector length (measured in pixels) an object must move before blur is added. If set to 0.0 it has no effect, and every object even with sub-pixel movement will have a slight amount of blur. While this is technically *accurate*, it may cause the image to be perceived as overly blurry.

For example, a cockpit view from a low flying jet plane rushing away towards a tropical island on the horizon may still add a some motion blur to the island on the horizon itself even though its movement is very slight. Likewise, blur is added even for an extremely slow pan across an object. This can cause things to be perceived slightly "out of focus", which is undesirable. This can be solved by setting **pixel_threshold** to e.g. 1.0, which in effect subtracts one pixel from the length of all motion vectors and hence causing all objects moving only one pixel (or less) between frames not to have any motion blur at all. Naturally, this value should be kept very low (in the order of a pixel or less) to be anywhere near realistic, but it can be set to higher value for artistic effects.

The **background_depth** sets a distance to the background, which helps the algorithm figure out the depth layout of the scene. The value should be about as large as the scene depth, i.e. anything beyond this distance from the camera would be considered "far away" by the algorithm.

If **depth_weighting** is off, a heuristic algorithm is used to depth sort the blur of objects. Sometimes the default algorithm can cause blur of distant objects imposing on blurs of nearby objects. Therefore an alternate algorithm is available by turning **depth_weighting** on. This causes objects closer to the camera than **background_depth** to get an increasingly "opaque" blur than further objects, the closer objects blurs much more likely to "over-paint" further objects blurs. Since this may cause near-camera objects blurs to be unrealistically opaque, the option defaults to being off. This mode is most useful when there is clear separation between a moving foreground object against a (comparatively) static background.

blur_fb sets the ID[3] of the frame buffer to be blurred. An empty string ("") signifies the main color frame buffer. The frame buffer referenced must be a RGBA color buffer and must be written by an appropriate shader.

depth_fb sets the ID of the frame buffer from which to obtain depth information. An empty string ("") signifies the main mental ray z depth frame buffer. The frame buffer referenced must be a depth buffer and must be written by an appropriate shader.

motion_fb works identically to **depth_fb** but for the motion vector information, and the empty string here means the default mental ray motion vector frame buffer. The frame buffer

[3]This parameter is of type *string* to support the named frame buffers introduced in mental ray 3.6. If named frame buffers are not used, the string will need to contain a number, i.e. "3" for frame buffer number 3.

referenced must be a motion buffer and must be written by an appropriate shader.

use_coverage, when on, utilizes information in the "coverage" channel rather than the alpha channel when deciding how to utilize edge pixels that contain an anti-aliased mix between two moving objects.

2.4 Using mip_motionblur

As mentioned before, the shader *mip_motionblur* requires the scene to be rendered with motion vectors:

```
shutter 0 0
motion on
```

The shader itself must also be added to the camera as an output shader:

```
shader "motion_blur" "mip_motionblur" (
   "shutter"          0.5,
   "shutter_falloff"  2.0,
   "blur_environment" on
)

camera "...."
   output "+rgba_fp,+z,+m" = "motion_blur"
   ...
end camera
```

The shader requires the depth ("z") and motion ("m") frame buffers and they should either both be interpolated ("+") or neither ("-"). The shader is optimized for using interpolated buffers but non-interpolated work as well.

If one wants to utilize the feature that the shader properly preserves and streaks over-brights (colors whiter than white) the color frame buffer must be floating point ("rgba_fp"), otherwise it can be plain "rgba".

2.4.1 Multiple Frame Buffers and Motion Blur

If one is working with shaders that writes to multiple frame buffers one can chain multiple copies of the *mip_motionblur* shader after one another, each referencing a different **blur_fb**, allowing one to blur several frame buffers in one rendering operation. Please note that only *color* frame buffers can be blurred!

Also please note that for compositing operations it is *not* recommended to use a **calculation_gamma** other than 1.0 because otherwise the compositing math may not work correctly. Instead, make sure to use proper gamma management in the compositing phase.

2.4.2 Good Defaults

Here are some suggested defaults:

For a fairly standard looking blur use **shutter** of 0.5 and a **shutter_falloff** of 2.0.

For a more "soft" looking blur turn the shutter up to **shutter** 1.0 but also increase the **shutter_falloff** to 4.0.

To match the blur of other mental ray renders, remember to set the mental ray shutter offset (in the options block) to half of that of the shutter length set in *mip_motionblur*. To match the motion blur to match-moved footage (where the key frames tend to lie in the center of the blur) use a shutter offset of 0.

2.5 The mip_motion_vector Shader

Sometimes one wishes to do compositing work before applying motion blur, or one wants to use some specific third-party motion blur shader. For this reason the *mip_motion_vector* shader exists, the purpose of which is to export motion in pixel space (mental ray's standard motion vector format is in world space) encoded as a color.

There are several different methods of encoding motion as a color and this shader supports the most common.

Most third party tools expect the motion vector encoded as colors where red is the X axis and green is the Y axis. To fit into the confines of a color (especially when not using floating point and a color only reaches from black to white) the motion is scaled by a factor (here called **max_displace**) and the resulting -1 to 1 range is mapped to the color channels 0 to 1 range.

The shader also support a couple of different floating point output modes.

The shader looks as follows:

```
declare shader "mip_motion_vector" (
        scalar  "max_displace"    default 50.0,
        boolean "blue_is_magnitude",
        integer "floating_point_format",

        boolean "blur_environment",
        scalar  "pixel_threshold",
        string  "result_fb",
```

```
            string  "depth_fb",
            string  "motion_fb",
            boolean "use_coverage"
        )
        version 2
        apply output
    end declare
```

The parameter **max_displace** sets the maximum encoded motion vector length, and motion vectors of this number of pixels (or above) will be encoded as the maximum value that is possible to express within the limit of the color (i.e. white or black).

To maximally utilize the resolution of the chosen image format, it is generally advised to use a **max_displace** of 50 for 8 bit images (which are not really recommended for this purpose) and a value of 2000 for 16 bit images. The shader outputs an informational statement of the maximum motion vector encountered in a frame to aid in tuning this parameter. Consult the documentation for your third party motion blur shader for more detail.

If the **max_displace** is zero, motion vectors are encoded relative to the image resolution, i.e. for an image 600 pixels wide and 400 pixels high, a movement of 600 pixels in positive X is encoded as 1.0 in the red channel, a movement 600 pixels in negative X is encoded as 0.0. A movement in positive Y of 400 pixels is encoded as 1.0 in the blue channel etc[4].

blue_is_magnitude, when on, makes the blue color channel represent the magnitude of the blur, and the red and green only encodes the 2d direction only. When it is off, the blue channel is unused and the red and green channels encode both direction and magnitude. Again, consult your third party motion blur shader documentation[5].

If **floating_point_format** is nonzero the shader will write real, floating point motion vectors into the red and green channels. They are **not** normalized to the **max_displace** length, not clipped and will contain *both positive and negative values*. When this option is turned on neither **max_displace** nor **blue_is_magnitude** has any effect.

Two different floating point output formats are currently supported:

- 1: The actual pixel count is written as-is in floating point.
- 2: The pixel aspect ratio is taken into account[6] such that the measurement of the distance the pixel moved is expressed in pixels in the Y direction, the X component will be scaled by the pixel aspect ratio.

More floating point formats may be added in the future.

[4] This mode will not work with 8 bit images since they do not have sufficient resolution.

[5] ReVisionFX "Smoothkit" expects vectors using **blue_is_magnitude** turned on, whereas their "ReelSmart Motion Blur" do not.

[6] Compatible with Autodesk Toxik.

When **blur_environment** is on, motion vectors are generated for the empty background area controlled by the camera movement. This option does not work if the scanline renderer is used.

The **pixel_threshold** is a minimum motion vector length (measured in pixels) an object must move before a nonzero vector is generated. In practice, this length is simply subtracted from the motion vectors before they are exported.

result_fb defines the frame buffer to which the result is written. If it is unspecified (the empty string) the result is written to the standard color buffer. However, it is more useful to define a separate frame buffer for the motion vectors and put its ID here. That way both the beauty render and the motion vector render are done in one pass. It should be a color buffer, and should not contain anything since it's contents will be overwritten by this shader anyway.

depth_fb sets the ID of the frame buffer from which to obtain depth information. An empty string ("") signifies the main mental ray z depth frame buffer. The frame buffer referenced must be a depth buffer and must be written by an appropriate shader.

motion_fb works identically to **depth_fb** but for the motion vector information, and the empty string here means the default mental ray motion vector frame buffer. The frame buffer referenced must be a motion buffer and must be written by an appropriate shader.

2.6 Using mip_motion_vector

The same considerations as when using *mip_motionblur* (page 12) about generating motion vectors, as well as the discussion on page 6 above about the timing difference between post processing motion blur vs. full 3d blur both apply.

Furthermore, one generally wants to create a separate frame buffer for the motion vectors, and save them to a file. Here is a piece of pseudo .mi syntax:

```
options ...
   ...
   # export motion vectors
   shutter 0 0
   motion on

   ...
   # Create a 16 bit frame buffer for the motion vectors
   frame buffer 0 "rgba_16"
end options

...

shader "motion_export" "mip_motion_vectors" (
   "max_displace"     2000,
   "blur_environment" on,
```

```
   # our frame buffer
   "result_fb"        "0"
)
...

camera "...."
   # The shader needs z, m
   output "+z,+m" = "motion_export"

   # Write buffers
   output "+rgba" "tif" "color_buffer.tif"
   output "fb0"   "tif" "motion_buffer.tif"
   ...

end camera
```

Chapter 3

Card/Opacity Shader

3.1 Introduction

When doing rendering that requires no form of post production, transparency requires no special consideration. One simply adds a transparency shader of some sort, and mental ray will render it correctly, and all is well.

However, as soon as one begins performing post production work on the image, and one is rendering to multiple frame buffers, even if simply using mental ray's built in frame buffers such as "z" (depth) or "m" (motion vectors), special thought must be put into how transparency is handled.

In general, mental ray collects it's frame buffer data from the *eye ray*, i.e. the ray shot by the camera that hits the first object. So the z-depth, motion vector etc. will come from this first object.

What if the first object hit is completely transparent? Or is transparent in parts, such as an image of a tree mapped to a flat plane and cut out with an opacity mask, standing in front of a house[1]?

When using most other transparency related shaders it is most likely that even though a tree is quite correctly visible in the final rendering and you can see the house between the branches, the "z" (depth) (and other frame buffers) will most likely contain the depth of the flat plane! For most post processing work, this is undesirable.

To solve this problem the mental ray API contains a function called *mi_trace_continue* which continues a ray "as if the intersection never happened". The shader *mip_card_opacity* utilizes this internally, and switches between "standard" transparency and using *mi_trace_continue* to create a "totally" transparent object at a given threshold.

[1] Using flat images to represent complex objects is known as putting things on "cards" in the rendering industry, hence the name of the shader

3.2 mip_card_opacity

```
declare shader "mip_card_opacity" (
        color   "input",
        boolean "opacity_in_alpha",
        scalar  "opacity",
        boolean "opacity_is_premultiplied",
        scalar  "opacity_threshold"
    )
    version 1
    apply material
end declare
```

The **input** parameter is the color of the object.

If **opacity_in_alpha** is on, the alpha component of the **input** color is used as the opacity.

If **opacity_in_alpha** is off, the **opacity** parameter is used as the opacity.

If **opacity_is_premultiplied** is on, the **input** color is assumed to already be premultiplied with the opacity value. If it is off, the **input** color will be attenuated (multiplied by) the opacity value before being used.

Finally, the **opacity_threshold** sets the opacity level where the shader switches from using standard transparency to becoming "completely transparent". Generally this should be kept at 0.0, i.e. only totally transparent pixels are indeed treated as "not even there", but if one raises this value more and more "opaque" pixels will be considered "not there" for frame buffers. Note that the actual visible rendered result is identical, only the contents of other frame buffers than the main color frame buffer is affected by this.

Chapter 4

Ray Type Switching Shaders

4.1 Generic Switchers

The *mip_rayswitch* and *mip_rayswitch_advanced* utility shaders allows different types of rays to return different results. There are many cases in which this can be useful, including but not limited to:

- Separating primary and secondary rays into calls to other shader.
- Returning a different environment to eye rays (i.e. a photographic background plate for screen background), reflection rays (i.e. a spherical high resolution environment to be seen in reflections) and final gather rays (a filtered environment suitable for lighting the scene).
- Limiting time consuming shaders where they are nearly invisible (avoiding a complicated secondary illumination or ambient occlusion shader in the refractions seen through frosted glass)

The *mip_rayswitch* shader is a simple shader that accepts a set of other colors (generally set to other subshaders) to apply for certain classes of rays.

```
declare shader "mip_rayswitch" (
        color  "eye",
        color  "transparent",
        color  "reflection",
        color  "refraction",
        color  "finalgather",
        color  "environment",
        color  "shadow",
        color  "photon",
```

```
        color  "default"
    )
    version 1
    apply material, texture, environment
end declare
```

For primary rays, **eye** sets the result for eye rays.

For secondary rays, **transparent** is the result for transparency rays, **reflection** for reflection rays and **environment** for environment rays.

The **finalgather** is the result for final gather rays as well as child rays to final gather rays.

Similarly, **shadow** is the result for shadow rays, **photon** catches all photon rays.

Finally, the **default** is the result for any other ray type. It is not a fall-through default, however, each of the above returns their respective result whether connected to a shader or not (i.e. generally 0 0 0 0 black).

If one wants fall-through defaults, one must use the advanced version of the shader:

```
declare shader "mip_rayswitch_advanced" (
        shader "eye",
        shader "transparent",
        shader "reflection",
        shader "refraction",
        shader "finalgather",
        shader "environment",
        shader "any_secondary",
        shader "shadow",
        shader "photon",
        shader "default"
    )
    version 1
    apply material, texture, environment
end declare
```

This shader works very similar to *mip_rayswitch*, but instead of accepting inputs of type "color", it accepts inputs of type "shader".

While this no longer allows assigning a fixed color directly, it instead allows *fall-through defaults*.

Each of the parameters works in a similar way: **eye** is the shader for eye rays, **transparent** for transparency rays, **reflection** for reflection rays, **refraction** for refraction rays, etc.

The difference is if one of these shaders are *not specified*, one of the fall-through cases take over. If either of the specific secondary ray type shaders are not specified, and a ray of that type

arrives, the **any_secondary** acts as a catch-all for all the secondary ray types not explicitly set.

Similarly, the **default** parameter works as a catch all for every unspecified shader above it.

4.2 Environment Switcher

A classical issue one runs into with mental ray is that there is simply a single concept of "the environment", whereas one often, in practical use, wants to separate the concept of a *background* to that of an *environment*.

This shader accomplishes exactly that:

```
declare shader "mip_rayswitch_environment" (
        color   "background"  default 0 0 0 0,
        color   "environment" default 0 0 0 0,
    )
    apply texture, environment
    version 1
end declare
```

The shader returns **background** for any eye ray, transparency ray that is a child of an eye ray, or any refracted ray that travels in the same direction as said transparency ray would (i.e. rays of type miRAY_REFRACT, but that was refracted by an IOR of 1.0 and is a direct child of an eye ray).

For any other type of rays (reflection, refraction, final gathering etc.) the shader returns the **environment** color.

The shader is intended to be used as the camera environment, but will function anywhere in a shading graph as a ray switching node, in many cases where one need to distinguish between "primary" and "secondary" rays.

For example, this is the ideal shader to switch between primary- and secondary rays to support the "best of both worlds" usage of *mip_matteshadow* described on page 51.

4.3 Render Stage Switcher

Sometimes one desires to use different colors or subshaders depending on where in the rendering pipeline mental ray is. For example, one may wish to make a certain material completely opaque to final gather rays, or for a light to have a different color while emitting photons, or similar.

```
declare shader "mip_rayswitch_stage" (
```

```
        color "unknown",
        color "main_render",
        color "finalgather_precomp",
        color "ao_precomp",
        color "caustic_photons",
        color "globillum_photons",
        color "importon_emit",
        color "lightmapping"
    )
    version 1
    apply material, texture, environment
end declare
```

The parameters all work in a similar way to all the other switcher shaders.

unknown is for any "unknown" stage. This should normally never be called, but exists as a safety precaution to safeguard against any new rendering stages introduced in future mental ray versions that are unknown at this time.

During the normal tile rendering pass, the shader returns the value of the **main_render** input.

During finalgather precomputation phase, the shader returns the value of **finalgather_precomp**.

During ambient occlusion precomputation phase, the shader returns the value of **ao_precomp**. Note that mental ray 3.7 does not call shaders at all during this phase, so this will never be called - but future versions of mental ray may act differently.

During caustic and global illumination photon tracing, the values for **caustic_photons** and **globillum_photons** inputs are used respectively.

During the importon emission phase, the value from **importon_emit** is used.

Finally, during the light mapping preprocessing phase (used, for example, by the subsurface scattering shaders) the value in **lightmapping** is used.

Chapter 5

Utility Shaders

5.1 Gamma/Gain Nodes

```
declare shader "mip_gamma_gain" (
        color   "input",
        scalar  "gamma"        default 1.0,
        scalar  "gain"         default 1.0,
        boolean "reverse"      default off,
    )
    apply texture, environment, lens
    version 1
end declare
```

This is a simple shader that applies a gamma and a gain (multiplication) if a color. Many similar shaders exists in various OEM integrations of mental ray, so this shader is primarily of interest for standalone mental ray and for cross platform phenomena development.

If **reverse** is *off*, the shader takes the **input**, multiplies it with the **gain** and then applies a gamma correction of **gamma** to the color.

If **reverse** is *on*, the shader takes the **input**, applies a reverse gamma correction of **gamma** to the color, and then divides it with the **gain**; i.e. the exact inverse of the operation for when **reverse** is off.

The shader can also be used as a simple gamma lens shader, in which case the **input** is not used, the eye ray color is used instead.

5.2 Render Subset of Scene

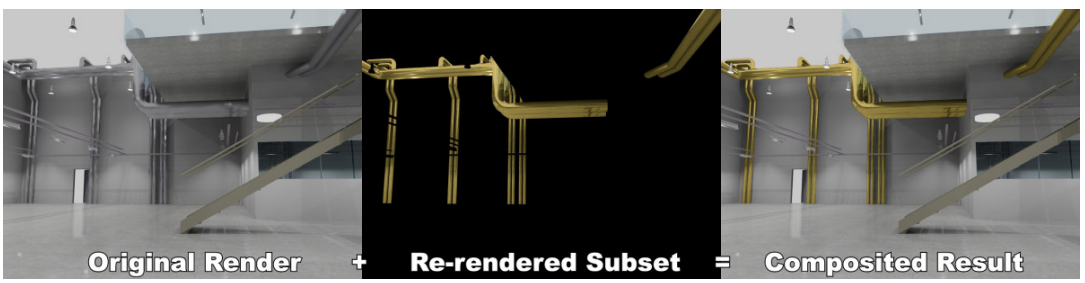
This shader allows re-rendering a subset of the objects in a scene, defined by material, geometric objects, or instance labels. It is the ideal "quick fix" solution when almost everything in a scene is perfect, and just this one little object or one material needs a small tweak[1].

It is applied as a lens shader and works by first testing which object an eye-ray hits, and *only if the object is part of the desired subset is it actually shaded at all.*

Pixels of the background and other objects by default return transparent black (0 0 0 0), making the final render image ideal for compositing directly on top of the original render.

So, for example, if a certain material in a scene did not turn out satisfactory, one can simply:

- Modify the material.
- Apply this lens shader and choosing that material.
- Render the image (at a fraction of the time of re-rendering the whole image).
- Composite the result on top of the original render!



An example of using mip_render_subset on one material

Naturally, only pixels which see the material "directly" are re-rendered, and not e.g. reflections in *other* objects that show the material.

The shader relies on the calling order used in ray tracing and does not work correctly (and yields *no* benefit in render time) when using the rasterizer, because the rasterizer calls lens shaders *after* already shading the surface(s).

```
declare shader "mip_render_subset" (
        boolean "enabled"        default on,
        array geometry "objects",
        array integer  "instance_label",
        material       "material",
```

[1] And the client is on his way up the stairs.

```
        boolean "mask_only"      default off,
        color   "mask_color"     default 1 1 1 1,
        color   "background"     default 0 0 0 0,
        color   "other_objects"  default 0 0 0 0,
        boolean "full_screen_fg" default on
    )
    apply lens
    version 5
end declare
```

enabled turns the shader on or off. When off, it does nothing, and does not affect the rendering in any way.

objects, **instance_label** and **material** are the constraints one can apply to find the subset of objects to shade. If more than one constraint is present, all must be fulfilled, i.e. if both a material and three objects are chosen, only the object that actually have that material will be shaded.

If one do not want to shade the subset, but only find where it is on screen, one can turn on **mask_only**. Instead of shading the objects in the subset, the **mask_color** is returned, and no shading whatsoever is performed, which is *very fast*.

Rays not hitting *any* objects return the **background** color, and rays hitting any object *not* in the subset return the **other_objects** color.

Finally, the **full_screen_fg** decides if the FG preprocess should apply to all objects, or only those in the subset. Since FG blends neighboring FG samples, it is probable that a given object may use information in FG points coming from nearby objects not in the subset. This is especially true if the objects are coplanar. Therefore it is advised to let the FG pre-pass "see" the entire scene.

Naturally, turning off this option and creating FG points only for the subset of objects is **faster**, but there is a certain risk of boundary artifacts, especially in animations. If the scene uses a saved FG map, this option can be left *off*.

5.3 Binary Proxy

This shaders allows a very fast way to implement demand loaded geometry. It's main goal is performance, since it leapfrogs any form of translation or parsing by directly writing to a binary file format which can be sucked directly into RAM at render time. There are many other methods to perform demand loading in mental ray (assemblies, file objects, geometry shaders, etc.) but this may require specific support in the host application, and generally involves parsing or translation steps that can impact performance.

To use it, the shader is applied as a geometry shader in the scene. See the mental ray manual about geometry shaders.

```
declare shader
    geometry "mip_binaryproxy" (
        string "object_filename",

        boolean "write_geometry",
        geometry "geometry",
        scalar   "meter_scale",
        integer  "flags"
    )
    version 4
    apply geometry
end declare
```

object_filename is the filename to read (or write). By convention, the file extension is ".mib" for "mental images binary".

The shader has a "read" mode and a "write" mode:

- When the boolean **write_geometry** is on and the **geometry** parameter points to an instance of an existing scene object, this object will be written to the .mib file named by **object_filename**.
- When **write_geometry** is off, the **geometry** parameter is ignored (not used). Instead, a mental ray placeholder object is created by the shader which contains a callback that will load the actual geometry from the file on demand (generally when a ray hits it's bounding box, although mental ray may choose to load it for other reasons at other times).

The **meter_scale** allows the object to be interpreted in a unit independent way. If this is 0.0, the feature is not used. When used, the value should be the number of scene units that represent one meter, i.e. if scene units are millimeters this would be 1000.0 etc.

When writing (**write_geometry** is on), this value is simply stored as meta data in the .mib file. When reading (**write_geometry** is off) the object is scaled by the ratio of the value

stored in the file and the value passed at "read time", to account for the difference in unit, if any.

The **flags** parameter is for algorithm control and should in most cases be left to 0. It is a bit flag, with each bit having a specific meaning.

Currently used values are:

1 Force use of "assemblies" rather than "placeholders". These are two slightly different internal techniques that mental ray uses to demand-load objects. See the mental ray manual for more information. Note that assemblies only work with BSP2 acceleration, and that multiple instances of the same assembly cannot have different materials or object flags applied. This limitation does not exist for placeholders.

2 "Auto-assemblies" mode. The shader uses assemblies if BSP2 acceleration is used, placeholders if not.

4 Do not tessellate the object before writing it. The object is written in it's raw format. The object must already be a mental ray primlist (miBox) for this to work. When this bit is set, displacement will not be baked to the file. When it is not set (the default) displacement *will* be baked[2]

All other bits should be kept zero, since they may become meaningful in future versions.

5.4 FG shooters

This shader is used to "shoot" finalgather (FG) rays into the scene in a pre-defined way, hence it's name. Normally, without this shader, during the final gather precomputation stage, mental ray shoots eye rays through the camera into the scene, and FG points are calculated in the scene based on the location caused by how these eye rays are shot.

The advantage of this is that the density of FG points are related to image space, and hence is automatically adaptive to exist in areas where they are necessary. On top of this comes an adaptive calculation of the FG point density, creating the optimal solution for the image.

However, during animation *in which the camera moves* this means that the location of where FG points are calculated will actually move along with the camera. In an animation, this can potentially cause artifacts in certain situations - especially if the camera moves *slowly*, or only moves a little (e.g. in a small pan, dolly, tilt, truck or crane move).

The "FG Shooter" shader exists to safeguard for this eventuality; it allows using a fixed (or a set of fixed) transformation matrices as the root location from which to "shoot" the eye rays

[2]Note that when baking displacement, a view-dependent approximation can not be used. This is because there is no view set up at the time when this shader executes, so the resulting tessellation will turn out very poor.

that are used *during the final gather precomputation phase only.* This guarantees that even if the camera moves

```
declare shader "mip_fgshooter" (
        integer "mode",
        array transform "trans"
    )
    version 1
    apply lens
end declare
```

The **trans** parameter contains an array of transformation matrices, defining how the eye rays are shot during the final gather prepass. Instead of using the camera to calculate the "eye" rays, they are shot from the 0,0,0 point, mapping the screen plane to the unit square surrounding the point 0,0,-1 and then transformed by the passed matrix (or matrices) into world space.

The **mode** parameter defines how the matrix (or matrices) passed are displayed during the final gather prepass. Since how it is *displayed* impacts the adaptivity, this has some functional implications.

- 0 breaks the render into "subframes, each containing the image as seen from a given position. This requires a higher final gather *density* to resolve the same number of details, but gives the best adaptivity.
- 1 stacks the different results on top of each other. This does not require any additional density, but the final gather adaptivity does not work as well.
- 2 works like 1, but only visibly displays one pass (but all the others are calculated exactly the same)

Chapter 6

Mirror/Gray Ball Shaders

6.1 Introduction

In the visual effects industry it is common to shoot a picture of a mirror ball (aka a "light probe") on set, as well as a gray ball for lighting reference.

Ideally, one shoots these at multiple exposures and uses a tool [1] to combine these into a single high dynamic range image and/or unwrap the mirrored/gray ball into a spherical environment map.

However, it is often difficult to regain the proper orientation of spherical map so it matches the camera used to render the CG scene. Furthermore, a single photo of a mirror/gray ball contains poor data for certain angles that one want to avoid seeing in the final render.

These shaders are intended to simplify a special case: When the mirror/gray ball is already shot from the *exact camera angle* that the final image will be rendered from.

It simply utilizes the mental ray camera coordinate space and applies the mirror/gray ball in this space, hence the orientation of the reflections will always 'stick' to the rendering camera.

6.2 Mirror Ball

This shader is intended as an *environment* shader, since it looks up based on the ray direction. It will map the proper direction to a point on the mirrored ball and retrieve its color.

```
declare shader "mip_mirrorball" (
        color  texture "texture",
```

[1] For example Photosphere (Mac) or HDRShop (PC)

```
        scalar "multiplier"   default 1.0,
        scalar "degamma"      default 1.0,
        scalar "blur"         default 0.0,
    )
    apply texture, environment
    version 3
end declare
```

The **texture** parameter should point to an image of a mirrored ball that is cropped so the ball exactly touches the edges of the image. Ideally it should point directly to a mental ray color texture but it *can* also be another shader.

If necessary, the shader can apply an inverse gamma correction to the texture by the **degamma** parameter. If gamma is handled by other shaders in the chain, or if the global mental ray gamma is used, use a value of 1.0, which means "unchanged".

The color is multiplied by the **multiplier** and *if the **texture** parameter points to a literal mental ray texture*, the **blur** parameter can be used to blur it.

6.3 Gray Ball

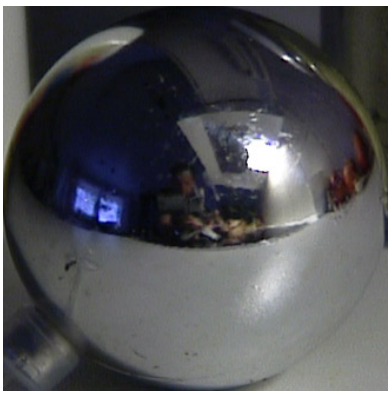
This shader is can be used either as an *environment* shader or a texture shader, since it looks up based on the direction of the surface normal. It will map the normal vector direction to a point on the gray ball and retrieve its color.

```
declare shader "mip_grayball" (
        color  texture "texture",
        scalar "multiplier"   default 1.0,
        scalar "degamma"      default 1.0,
        scalar "blur"         default 0.0,
    )
    apply texture, environment
    version 3
end declare
```

The parameters are identical to **mip_mirrorball**.

6.4 Examples

Here are the photos used for the example renderings:



The mirror ball photo

The gray ball photo

Here are a few objects with an environment made using **mip_mirrorball**. The objects are also lit by final gathering based on same environment:

One angle

Another angle

What is noteworthy in the above images is how the orientation of reflections stay constant parallel to the camera. Therefore, this shader is only intended to be used when the rendering camera direction (roughly) matches the direction from which the mirror ball was photographed.

When using the gray ball shader:

Raw mip_grayball shader

Using ambient occlusion

The left image is the raw output of the **mip_grayball** shader, but in the right the shader is put into the **bright** slot of the **mib_amb_occlusion** shader, where it is occluded and looked up with bent normals[2].

More usage tips can be found on page 39.

[2]If one do not desire the bent normal lookup of **mib_amb_occlusion**, set its **mode** parameter to -1. Then the **bright** parameter is looked up using the original (un-bent) normal, which can yield a smoother result.

Chapter 7

Matte/Shadow Objects and Camera Maps

7.1 Introduction

Often one wants to include synthetic[1] objects into an existing photographic background plate filled with real world objects, for example, adding a yet-to-be-constructed building into an empty lot, adding a virtual car onto a road, or having a virtual character walk through a scene and realistically interact with objects in the real world scene.

Two main shaders exists to facilitate this, *mip_cameramap*, which "projects" an image from the camera onto geometry, and *mip_matteshadow* which takes care of generating hold-out mattes, as well as allowing the real world objects in the photographic plate both cast and receive shadows, as well as receive reflections and indirect light.

7.2 mip_cameramap

This shader is used to look up a color texture based on the surface points on-screen pixel coordinate. The shader is similar in function to **mib_lookup_background** from the `base` library, but with the following important differences:

- The coordinate being looked up is the true "back transformation" of the shading point into raster space, rather than just the current rendered pixels raster position. This means that the background is correctly seen in reflections and refractions.
- It can perform a "per pixel" match, avoiding blurring of the background due to interpolation.

[1] When we use the term "synthetic" we mean additional objects to be inserted in the scene, and "real world" for objects that are already there.

```
declare shader "mip_cameramap" (
        color texture "map",
        scalar  "multiplier"               default 1.0,
        scalar  "degamma"                  default 1.0,
        boolean "per_pixel_match"          default off,
        boolean "transparent_alpha"        default off,
        boolean "offscreen_is_environment" default on,
        color   "offscreen_color"
    )
    version 4
    apply material, texture, environment
end declare
```

The **map** parameter is a color texture to be looked up, and **multiplier** a multiplier for that map.

If necessary, the shader can apply an inverse gamma correction to the texture by the **degamma** parameter. If gamma is handled by other shaders in the chain, or if the global mental ray gamma is used, use a value of 1.0, which means "unchanged".

When **per_pixel_match** is off, the texture is simply stretched to fill the scene exactly. When **per_pixel_match** is on, the lower left pixel of the map is exactly matched to the lower left rendered pixel. If the pixel size of the map and the pixel size of the rendering is different a warning is printed, but the image is still rendered although the image will be cropped/padded as needed.

Sometimes one wants the shader applied as a background, but still want to extract information from the alpha channel. If **transparent_alpha** is on, the alpha values is always zero. If it is off, the alpha value from the texture is used.

The shader performs a true "reverse transform" of the shading point into raster space. While this by necessity results in an on-screen location for eye rays, this is not so for reflections, refractions etc. These rays may hit a point on the object which is off screen. If the **offscreen_is_environment** is on, these rays will return the environment. If it is off, the **offscreen_color** is returned. In either case, this color is *not* affected by neither the **multiplier** nor **gamma**.

7.3 mip_matteshadow

This shader is used to create "matte objects", i.e. objects who is used to "represent" existing real world objects in an existing photographic plate, for the purposes of...

- ...blocking another synthetic object from the cameras view (to allow the synthetic objects to go behind the real world object).
- ...allowing synthetic objects to cast shadows and occlusion on and receive shadows from the real world objects.

- ...adding reflections of synthetic objects onto real world objects.
- ...allow the interplay of indirect light between synthetic and real world objects.

In all above cases the *mip_matteshadow* is applied to an object *representing* the real world object, and the synthetic object is using a traditional material.

The shader can also function as a "shadows only" shader, i.e. a shader which only shows how much in shadow a point is compared to the incoming light, but ignoring the actual amount of incoming light itself (only the occluded percentage of it).

```
declare shader
        struct {
            color "result",
            color "shadows_raw",
            color "ao_raw",
            color "refl_raw",
            color "indirect_raw",
            color "illumination_raw"
        } "mip_matteshadow" (
        color   "background"        default 0 0 0 0,
        # Shadows
        boolean "catch_shadows"     default on,
        color   "shadows"           default 0 0 0 1,
        color   "ambient"           default 0.2 0.2 0.2,
        boolean "no_self_shadow"    default on,
        boolean "use_dot_nl"        default on,
        scalar  "colored_shadows"   default 1.0,
        # AO
        boolean "ao_on"             default on,
        color   "ao_dark"           default 0.0 0.0 0.0,
        integer "ao_samples"        default 16,
        scalar  "ao_distance"       default 0.0,
        # Reflections
        boolean "catch_reflections" default off,
        color   "refl_color"        default 0.2 0.2 0.2 0.2,
        color   "refl_subtractive"  default 0.2 0.2 0.2 0.2,
        integer "refl_samples"      default 0,
        scalar  "refl_glossiness"   default 10.0,
        scalar  "refl_max_dist"     default 0.0,
        scalar  "refl_falloff"      default 2.0,
        # Indirect
        boolean "catch_indirect"    default off,
        color   "indirect",
        # System
        boolean "multiple_outputs"  default off,
        # Additional illumination
        boolean "catch_illuminators" default off,
        array light "illuminators",
        # Extra input
        color   "additional_color"  default 0 0 0,
```

```
        # Light linking
        integer     "mode",
        array light "lights"
    )
    version 6
    apply material, texture
end declare
```

NOTE: This section will only **briefly** lists the parameters and the section "usage tips" on page 39 gives an in depth explanation of different use cases.

The shader has multiple outputs (it returns a struct). However, by default (for compatibility reasons) it only fills in the first item of the struct, the compound "result". Only if the parameter **multiple_outputs** is enabled are the separate results written to the other outputs.

The **background** parameter is the background color. If neither of the **catch_...** options are on, this result is simply returned, including its alpha, and the shader does nothing. Otherwise it is the base color upon which all the other operations occur. When using external compositing this is generally transparent black (0 0 0 0), otherwise one would use the real world background plate mapped with the *mip_cameramap* shader.

The **catch_shadows** option enables other objects to cast shadows on this object.

The **shadows** parameter is the color of the shadows. When shadows are detected, a blend between **background** and **shadows** depending on how much "in shadow" the point is.

The **ambient** parameter sets a "base light level". It raises the lowest "in shadow" level. For example, if this is 0.2 0.2 0.2 the darkest shadow produced will be an 20 percent blend of **background** to a 80 percent blend of **shadow** (unless ambient occlusion is enabled).

Turning on the **no_self_shadow** option (and also using an instance of the shader as shadow shader) causes any object using *mip_matteshadow* not to receive shadows on any other such object.

The **use_dot_nl** option defines if the angle to the light is considered when calculating the incoming amount or not.

If **color_shadows** is 0.0, all shadows are cast in gray scale. If it is 1.0, the shadows have full color. For example, if the surface is lit by one red and one green light, the red light will have a green shadow, and the green light will have a red shadow.

If **ao_on** is enabled, a built in *Ambient Occlusion* (henceforth simply called "AO") is applied based on the color in the **ambient** parameter. The AO respects the **no_self_shadow** switch and will not cause AO from objects with the same material instance as itself.

The **ao_dark** parameter decides how dark shadows the AO will cause. The default black is generally adequate, but a lighter color will cause a less pronounced shading effect.

ao_samples number of AO rays are shot. One can limit the reach of the AO rays with the **ao_distance** parameter. If it is zero, the rays reach infinitely far. Short rays increase performance dramatically but localizes the AO effect.

The **catch_reflections** turns on reflections.

The **refl_color** is the multiplier for reflections. Note that the alpha value of this color is important in that the reflections will influence the alpha by this amount. In some cases the reflections may need to be subtractive, in which case **refl_subtractive** is used - see page 39 for details.

The **refl_samples** parameters sets the number of glossy reflection samples. If it is zero, mirror reflections are used. Otherwise the **refl_glossiness** sets the ward glossiness for reflections.

Reflections are seen at most **refl_max_dist** and the shape of the falloff is **refl_falloff**, very similar like the `base` library's shader **mib_glossy_reflection**.

If the option **catch_indirect** is enabled, indirect light is gathered and scaled by the **indirect** color (which one generally sets to the same as the **background** color, thereby treating the background color as a reflectance value for the indirect light).

The **multiple_outputs** causes the shader to output more than a single value in the returned struct.

If the **catch_illuminators** switch is on, the lights listed as **illuminators** are tested and allowed to actually light the scene. This is a simple Lambertian illumination treating the **background** parameter as the diffuse color[2].

The **additional_color** is a color input simply added to the result, for easy shader graph construction. It can be used for anything[3].

Finally, **mode** sets the light inclusion/exclusion mode and **lights** is the light list used for casting shadows, just like in many other shaders.

The shader also has *multiple outputs*. For compatibility, the shader does not actually write any values to anything but the main output if the **multiple_outputs** parameter is not on. But if this parameter is enabled, the shader outputs the following values:

- **result** - the compound result.
- **shadows_raw** - the raw full-color shadow pass on white background, suitable for compositing on top of a background in "multiply" mode.

[2]The difference between **lights** and **illuminators** is that the **lights** are only used to cause shadows on the background, whereas the **illuminators** are used to throw actual *light* on it. This means that the **lights** array should contain lights which are already present in the background plate, and the **illuminators** array contains any additional lights introduced by a CG element, for example the headlights of a CG car.

[3]For example to add a specular highlight to an illuminator by plugging in a **mib_illum_phong** with it's diffuse color set to black.

- **ao_raw** - the raw ambient occlusion.
- **refl_raw** - the raw reflections
- **indirect_raw** - the indirect light arriving
- **illumination_raw** - light gathered from any lights in the **illuminators** list.

All outputs are as "raw" as possible to be maximally useful as layers in post production, e.g. the reflections have not been multiplied with the reflection color, etc.

7.4 mip_matteshadow_mtl

This is a *material phenomena* that embeds *mip_matteshadow*, and applies it as surface, shadow and photon shader for a material. It has the same parameters as *mip_matteshadow* itself, plus a scalar **opacity**.

The opacity does not apply to the shadows or photons, but only to the visible surface shading itself. By using the opacity, one can use painted masks for irregular edges. For example, use a simple tapered cylinder to roughly match a forearm, and use a camera-projected mask (applied with the help of *mip_cameramap*) for the actual contours of the arm. This also allows matching motion blur that may be present in the background plate with the help of masks.

7.5 Usage Tips

The shaders in this section are concerned with combining a real world photographic plate with synthetic objects. For the sake of our examples, we are basing them on the following *photograph* of a kitchen counter, with an old beat up plastic salt shaker, a spice jar, a piece of paper, and some other things on it:

This is our background photograph. This is not a rendering... yet.

7.5.1 Camera Mapping Explained

What is the difference between *mip_cameramap* and the *mib_lookup_background* from the `base` library?

The main difference is that the former only uses the current raster position (i.e. x and y coordinate of the pixel currently being rendered) whereas the latter actually calculates the image space position of the shaded point.

Why does that matter? Lets try an example. See the following example of reflective sphere on top of a background photograph mapped to a flat plane using the two different shaders:

mib_lookup_background

mip_cameramap

Notice how the left image, which uses *mib_lookup_background* looks incorrect. It looks transparent rather than reflective. What is the cause of this?

Imagine we are rendering pixel coordinate 200,200 which lies on the sphere. An eye ray is sent that hits the sphere. This hit point is indeed on the 200,200 pixel coordinate. But the ray continues as a reflection ray to hit the ground plane.

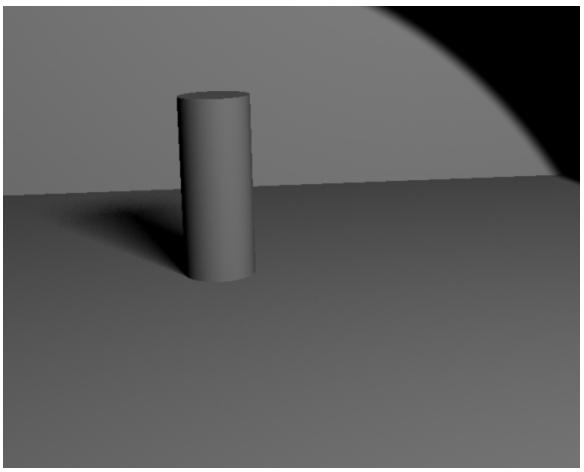
This new point (on the ground plane) is not the same as the pixel being rendered, it is somewhere else in the model. Yet *mib_lookup_background* only concerns itself with the current pixel (200,200) and will return the color at that point from the map.

Conversely, *mip_cameramap* actually converts this *new* point to a new set of raster coordinates (e.g. 129,145) and will look up the map at that *new* position, creating the correct appearance in the reflection.

However, there is a snag; what happens if the reflected point lies *outside the screen*? What if the calculated raster coordinate is (-45, 39)? The answer is that *mip_cameramap* either returns the specified color, or, if **offscreen_is_environment** is on, returns the environment color for that ray direction.

7.5.2 Matte Objects and Catching Shadows

In this section we will use *mip_matteshadow* to create stand-in objects (also known as "matte objects") for real world geometry. For the example we only concern ourself with the old plastic salt shaker on the left of the image, and the kitchen counter itself. The following simple 3D model has been constructed:



The simple 3D model of our background

If we are *not* using *mip_matteshadow* and for example simply attempt to map our background to a standard Lambertian surface we get the following result:

Mapping the background onto a Lambert shader

Clearly unsatisfactory, since Lambertian shading is applied on top of shading already present in the real world image. What we need is to separate *shadows* from the *shading*.

This can be done using use *mip_matteshadow*. Lets show the result by applying the shader with a white background and black shadows:

Raw shadow information

This image contains the raw shadow information, and is otherwise white. This mode of operation is very useful for external compositing usage, where a completely separate *shadow pass* can be generated by making the synthetic objects invisible to the camera (the torus in our case) and using *mip_shadowmatte* with white background and black shadows.

NOTE: This usage supports colored shadows! With a red and green light present, the green light will show red shadows and vice versa.

There is one problem with the above image: It contains a shadow of the salt shaker. But our real-world picture already contains a shadow of the salt shaker. This is solved by the following steps:

- Applying *mip_matteshadow* as the shadow shader for the material.
- Turning its **no_self_shadow** option on.

The mip_matteshadow material does not shadow itself

When the shader is used as the materials shadow shader and the **no_self_shadow** is on, no object with *mip_matteshadow* will shadow another object with *mip_matteshadow*, but will still cast shadows on other objects, and other objects will cast shadows upon it.

Notice however that the shadows are very dark and the torus shadow thrown *onto* the salt shaker has a harsh left edge. To solve this, one can set the **ambient** color to a low value and turn on **use_dot_nl** which will make the shadows "fade out" as they become perpendicular to the light, avoiding the harsh edge:

Lighter shadow and use_dot_nl on

At this point we can replace the white color with our background using *mip_cameramap*:

Background used instead of white

7.5.3 Advanced Catching of Shadows

Shadows can be handled in several ways with *mip_matteshadow*. We already mentioned the method above of creating a shadow pass using a white **background** and black **shadows** setting.

We also showed the method of using a picture for **background** and black for **shadows**.

There are two other options, however:

One is to use a picture of the *lit* scene for **background** and *another* picture of the scene in shadow for the **shadow** parameter.

The scene fully lit

The scene in shadow

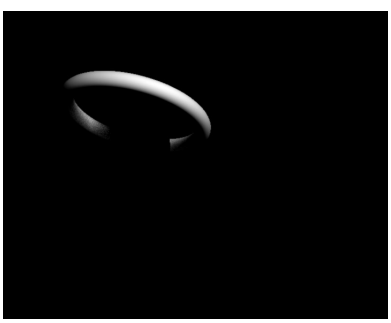
This technique is especially useful since it solves *any* self-shadowing issues perfectly, it is even advised to have **no_self_shadow** off in this mode. Here is the result:

Shadows come from unlit version of photo

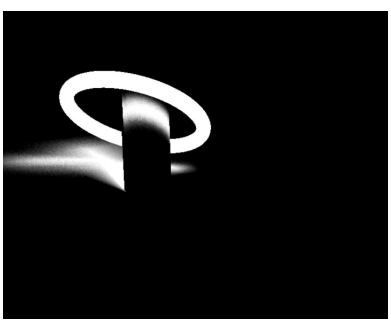
This gives amazing detail and automatically gives the "correct" color and strength of shadows without any tuning being necessary.

A final way is to handle shadows completely in external compositing. This is accomplished by setting **background** to 0 0 0 0 (i.e. transparent black, alpha is zero) and **shadow** to 0 0 0 1 (i.e. opaque black, alpha is one).

This will create a rendering as follows:



The color channels

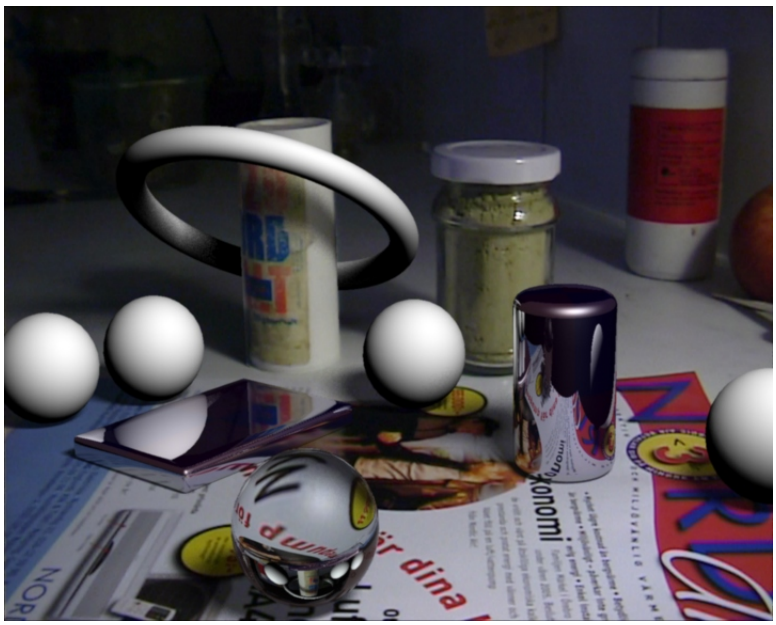


The alpha channel

This image can be composited on top of the background externally since the shadow information is present in the alpha channel.

7.5.4 Reflections

Since we are using *mip_cameramap* reflections and refractions of the background objects work correctly:



Reflection and refraction

Note how even the reflection of the salt shaker is correct.

Note that this also means final gathering will pick up the background as indirect illumination correctly:

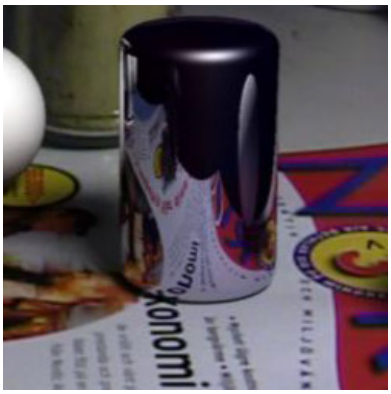
Final gathering lights up our objects

The shader can also show (glossy) reflections of other objects:

Glossy paper reflects our objects

Please note how the reflection only involves the actual synthetic objects added, the shader already assumes that the real world photo contains reflections for the other real world objects, i.e. a *mip_matteshadow* object never reflects another, nor does it reflect the environment.

One subtle detail is still missing, though. Notice how the cylinder is reflecting the paper, but nothing else. This is because the **mip_matteshadow** does not know what to do about rays that hit the ground plane somewhere "off screen".

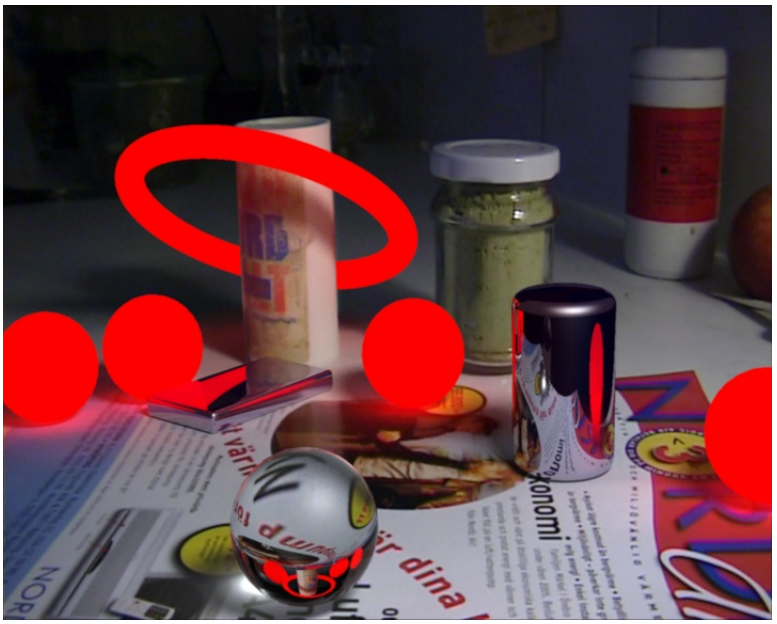


No reflections

This is solved by using the **offscreen_is_environment** parameter of **mip_cameramap**, and then utilize an environment map (for example **mip_mirrorball**) to supply an environment map:

Much better

The shader can also receive indirect light such as final gathering and (if also applied as photon shader) photons. To demonstrate this we enable **catch_indirect** and turn our white objects luminous red and made sure our glass sphere generates caustics:



Receiving indirect light

As above, the shader makes sure no indirect light is bounced from one matte object to another, and no indirect light is received from the environment (since both these effects are assumed to be present in the original real-world photo).

A final special case for reflection is when the background photo already contains massive amounts of reflections. Here is a new background photo on top of a highly reflective surface:

Photograph of our salt shaker on a very reflective surface

If we add our synthetic objects to this scene and use the same settings we used for the glossy paper, we get the following rather odd looking results:

Additive reflections - doesn't look right

But this looks very incorrect! One can see the reflection of the background through the added reflections, and the added reflections appear much too bright.

To fix this one uses the **refl_subtract** to set the amount of attenuation of the background caused by the reflection. Naturally this can be a painted map to match reflective areas in the image, i.e. a water puddle in front of a house etc.

Subtractive reflections - that's much better!

The correct result. The synthetic reflections override the existing reflections and replace them! Also note that even reflections of objects *behind a real world object inside a real world reflection* works correctly (the white ball behind the salt shaker is behind it even in the reflection).

Finally, all objects pick up indirect illumination based on the background plate itself. Only one rectangular area light source exists in the rendering.

7.5.5 Reflections and Alpha

When preparing for external compositing and using transparent black for **background** and opaque black for **shadows** it is important that reflections also get an alpha channel.

The alpha component of the **refl_color** parameter defines how much the reflection is present in the alpha channel.

Likewise the alpha component of the **indirect** parameter defines how much indirect light is seen in the alpha channel.

Neither of those have any function if the alpha of the background is 1.0, i.e. when using a background photo.

7.5.6 Best-of-Both-Worlds Mode

If one need the synthetic objects to reflect the real objects but still want the flexibility of external compositing for shadows (i.e. one does not want to add in the background already when rendering) one can create a hybrid approach.

This is accomplished with the help of the *mip_rayswitch* shader. Use the shader to switch between using a transparent black (0 0 0 0) background for eye rays and using the camera mapped photo for other rays (e.g. reflection, refraction and final gathering).

7.5.7 Conclusion and Workflow Tips

Here is a simple step-by-step workflow to render synthetic objects into a background plate. It assumes one has

Prerequisite stuff

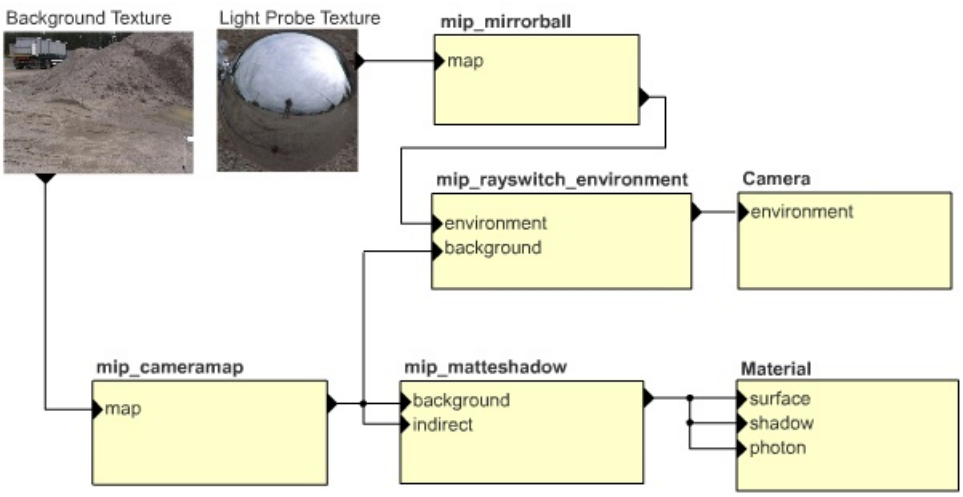
- A background plate
- A mirror ball photo taken from the same camera angle [4]

Then perform as follows:

- Put **mip_rayswitch_environment** in the mental ray camera environment.
- In its **background** slot, use a **mip_cameramap** with the background plate in its **map** slot.
- In its **environment** slot, use **mip_mirrorball** with the mirror ball photo in its **map** slot.

[4] One can often get away with a low dynamic range mirror ball photo if one keeps it slightly underexposed.

- Create a ground plane, and use **mip_matteshadow** as the surface, shadow and photon shader on that ground plane.
- Use the *same* instance of **mip_cameramap** in its **background** parameter.
- Add lights to mimic the "real world" lighting, or use final gathering based on the environment, or a combination thereof to achieve the proper shading.
- Tune any ambient occlusion by setting the **ambient** parameter of **mip_matteshadow** and adjusting the **ao_distance**.
- If the "ground" should catch general indirect lighting, enable **catch_indirect** and connect the map in **background** to **indirect** as well.



The shade tree

Add any CG objects with physically plausible shaders (such as the **mia_material** from the Architectural library). Render. Smile.

Render. Smile.