

Writing Mental Ray Shaders

The purpose of this tutorial is to

1. explain what types of shaders are there in *mental ray*, how to write them, and how to
2. integrate them into *Houdini*.

There are two books about *mental ray*. Both are part of a series called *mental ray Handbooks*. Volume 1 (see [1] for reference) is a more general introduction into rendering with *mental ray* and volume 2 (see [2] for reference) is kind of a reference manual for shader and translator writers. Newer editions of the first book come with a CDROM containing a demo version of *mental ray* (version 2.1). Unfortunately both books cover only *mental ray* versions up to version 2.1. For newer versions you have to refer to the HTML documentation coming with *mental ray*.

Shader Type Overview

The second book (see [2] for reference) gives an overview of all shader types of *mental ray* in chapter 3.3 and has sections for each shader type in chapter 3¹.

- **material shaders** are maybe the most important shader type in *mental ray*. They are called whenever a visible ray (eye ray, reflected ray, refracted ray, or transparency ray) hits an object.
- **volume shaders** may be attached to the camera or to a material. They accept an input color that they are expected to modify to account for the distance the ray traveled through a volume.
- **light shaders** are called from other shaders by sampling a light or directly if a ray hits a source.
- **shadow shaders** are called instead of material shaders when a shadow ray intersects with an object. Shadow rays are cast by light sources to determine visibility of an illuminated object.
- **environment shaders** provide a color for rays that leave the scene entirely, and for rays that would exceed the trace depth limit.

¹Using and Writing Shaders

- **photon shaders** are used in the photon tracing phase to compute the photon maps that are used to simulate caustics and global illumination.
- **photon volume shaders** are similar to photon shaders in the same way that volume shaders are similar to material shaders: they compute indirect light interactions in volumes, such as volume scattering.
- **photon emitter shaders** are used in the photon tracing phase to control the emission of photons from the light sources.
- **texture shaders** are called exclusively by other shaders to relieve other shaders, such as material or environment shaders, from performing color and other computations.
- **displacement shaders** are called during tessellation of polygonal or free-form surface geometry. Whenever the tessellator introduces or copies a vertex, the displacement shader is called and expected to return a scalar value that tells the tessellator to move the vertex by this distance along its normal vector.
- **geometry shaders** are functions that procedurally create geometric objects. It allows mental ray to store small and simple placeholders in the scene, and generate the objects only when they are needed, and remove them from the cache when they are no longer needed.
- **contour shaders** compute contour colors and widths. Any information about the geometry, illumination, and materials of the scene can be used.
- **lens shaders** are called when a primary ray is cast by the camera. They may modify the eye ray's origin and direction to implement cameras other than the standard pinhole camera, and may modify the result of the primary ray to implement effects such as lens flares.
- **output shaders** are called when the entire scene has been completely rendered. They modify the resulting image or images to implement special filtering or compositing operations.
- **lightmap shaders** sample the object that the material is attached to, and compute a map that contains information about the object. This can be used to bake illumination solutions or store vertex colors for a hardware game engine.

Integrate Mental Ray Shaders Into Houdini

Before we start writing our own shaders it's worth to invest some time to figure out how we can integrate them into Houdini and use them from there. Let's start with a simple scene in mind.

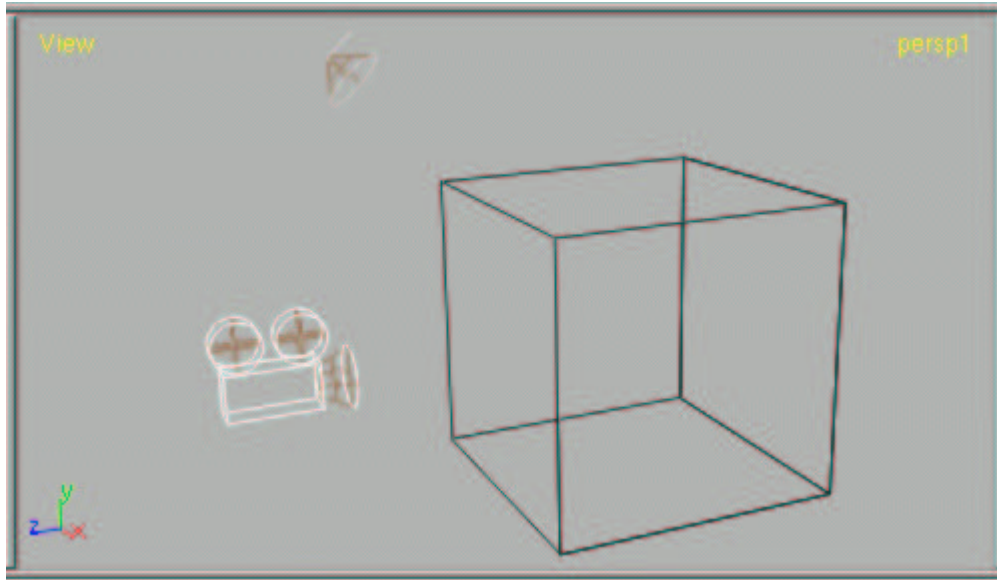


Figure 1: A simple scene.

In this scene there is a camera, one light, and a box. The steps to create this scene in *Houdini* are simple:

1. Start *Houdini*.
2. Create an empty scene by selecting **File** -> **New** or pressing **Alt+n**.
3. Create a **Camera** object and turn on the **Display Flag**.
4. Create a **Light** object and set the translation in y-direction to 1. You can scale the default geometry for the light down to 0.2 if you want to make it smaller in the viewport. It will not affect the rendering.
5. Create a **Geometry** object. Delete the default geometry and replace it by a **Box SOP**. Set the translation in z-direction to -1. If you want the light to look at the box you can set the **Look At** parameter of the light. It will not affect the rendering because the direction does not matter if the shader does not take the direction into account ².

²E.g. point lights

6. Save your work in a HIP file for later.

Each of these objects in the scene will get at least one *mental ray* shader. The camera first will have a lens shader but might have a volume shader later. The light shader gets a light shader and the box will have several shaders attached to it. We will build all these shaders from scratch by copying some lines from the shaders coming with *mental ray*.

Specifying The Type And The Parameters

We will create several debug shaders to see how the shaders cooperate in mental ray and how they affect the final image. The best thing to start with is to specify the type and the expected parameters in a MI file. Let's start with a material shader:

```
declare shader
    "dbg_material" (
        array light "lights"
    )
    version 1
    apply material
end declare
```

Save these few lines in a file called `dbg_material.mi`. As you can see the name of the shader is specified as `dbg_material` and it has only one parameter called `lights`. The type of this parameter is an array of lights. The type of this material shader is specified after the keyword `apply`. A MI file like this can be used for three different things:

1. It can be used in your MI file which will specify the scene to be rendered. You can either use *mental ray* directly from Houdini by simply creating a Mental Ray ROP and using its default settings or you can tell this ROP to write into a file by clicking on its parameter **Generate Script File** and specify a file name with the **Script File** parameter. If you press the **Render** button a file with this name will be created. If you look into the resulting file you will see two lines like this:

```
link "base.so"
$include <base.mi>
```

The first line tells *mental ray* to load a **shared library** called `base.so`. This library comes with *mental ray* and contains not only one shader but a collection of basic shaders. The second line tells *mental ray* to include the declarations of this shaders from a file called `base.mi`. If you want to use your shaders from within *Houdini* it's the best to have a MI file³ for each shader. You will find similar lines for your own shaders when you have created your shared libraries for them and have linked your shaders to *Houdini* objects.

2. An MI file can be used to create automatically a **skeleton** for your shader. With *mental ray* there comes an executable program called `mkmishader`. We will use it to create a C file from the declaration file:

```
mkmishader -v -r -i dbg_material.mi
```

The parameter `-v` tells the program to be **verbose**, `-i` will create init and exit shaders, and `-r` means that the `.rayrc` file should be read first⁴.

3. To **make Houdini aware** of the shader and his parameters use the following command:

```
mids -g 8 -d $HOME/houdini5/shop dbg_material.mi
```

Just type `mids` to get an explanation of this command and it's parameters. You will find a subdirectory called `fog`, `lens`, `light`, `surface`, or `surfaceshadow` in your `$HOME/houdini5/shop` directory⁵ and in one of them⁶ a file called `dbg_material.ds`. If you want you can edit this file to have a help text in Houdini. But before you can use this shader you have to copy some files from `$HIS/shop` to `$HIH/shop` and edit them. In this case you would have to copy the files `SHOPsurface` and `SHOPsurface.mi`. You should change the first two lines by replacing `HFS` through something else and insert the following line after the comments:

```
#include $HFS/houdini/shop/SHOPsurface
```

³and a corresponding shared library

⁴Type **mkmishader -h** to see the options.

⁵dependent on your shader type

⁶In this case: `surface`

The files with extension `.ri` are used for *RenderMan* and the `.vm` extension for *Mantra* shaders written in VEX. You will tell *Houdini* with this entries that the search path should be extended to include your own *mental ray* shaders. The line in `SHOPsurface.mi` could look like this:

```
dbg_material shop/surface/dbg_material.ds \
    -label "Debug Material Shader"
```

Writing Debug Information

Before I will continue to explain how to use the different kinds of shaders from within *Houdini* I would like to introduce two simple concepts to debug the shaders or to visualize what's going on during rendering:

1. You can write simple lines of information to the standard output during rendering with the *mental ray* command `mi_info()`.
2. You can write information with standard C/C++ methods to a file. I will use this technique to write *Houdini* commands to a file to visualize rays.

A Debug Lens Shader

This debug information can result in a huge amount of data and slow down the rendering process. To limit the amount of information I will start with writing a **lens shader**. A lens shader is the first shader which will be called by *mental ray* and is attached to the camera. More than one lens shader may be attached to the camera and each one may modify the origin and direction of a ray calculated by the previous one. In this case I will use the lens shader for only one purpose: To limit the amount of rays coming from the camera to **exactly one ray**. Define the type and parameters for the shader in a file called `dbg_lens.mi`:

```
declare shader
    "dbg_lens" (
    )
    version 1
    apply lens
    scanline off
    trace on
end declare
```

Lens shaders that depend on modifying ray origin and direction should be declared with the `trace on` option and turn off `scanline` rendering.

I will not repeat each single line of code of the lens shader (or any other shader) in this tutorial. Most of the code was automatically produced by running the `mkmishader` command. I added the following lines after the `#include` statements and before the definition of the function `dbg_lens_version(void)`:

```
#include "dbg_flag.h"
#ifdef HOUDINI_CMD
FILE* dbg_fp = NULL;
int dbg_num_rays = 0;
#endif
static int initialized;
```

The first line will include a common header file I use to basically activate some lines in each shader or turn them off. The header file `dbg_flag.h` looks like this:

```
#ifndef DBG_FLAG_H
#define DBG_FLAG_H
#define HOUDINI_CMD
#endif
```

The line `#define HOUDINI_CMD` activates lines enclosed by the appropriate `#ifdef HOUDINI_CMD` and `#endif`. If you change this line in the header file to `#undef HOUDINI_CMD` all this lines will not be compiled into the resulting shared library for this shader. The header file is just a convenient way to change this behavior for all shaders simultaneously. Of course you can put a single line directly into the shader C/C++ code to change the behavior only for this shader in particular.

For the debug lens shader the two lines you can activate will be used to define a global file pointer and a counter which can be used by all the other shaders as well. The lens shader is responsible to open a file for writing and close the file afterwards because the lens shader is the first shader to be called by *mental ray* and therefore the last shader to be destroyed after rendering. The variable `initialized` is defined to be `static` to limit the visibility to the lens shader only. I use it to cast exactly one ray into the scene and ignore all other calls of the shader by doing nothing. You might think that setting the resolution to one pixel in each direction would result in the same effect but if you try this you will see that your shader is called several times for the corner points of your one by one pixel resolution.

In the function `dbg_lens_init()` I added the following lines after the `/* shader init */` comment line:

```

        mi_info("dbg_lens_init [shader init]");
        initialized = 0;
        /* Houdini stuff */
#ifdef HOUDINI_CMD
        dbg_fp = fopen("rays.cmd", "w");
        fprintf(dbg_fp, "cd /obj\n");
        fprintf(dbg_fp, "opadd -n geo rays\n");
        fprintf(dbg_fp, "cd rays\n");
#endif

```

I use the following **convention** for the `mi_info()` output lines. For the init and exit shader calls I use the name of the function and in brackets which part of the function is executed. There is a part for the shader itself which is called for the first usage of this shader and a part for each instance of this shader because every shader can be used several times in a scene.

In case you have set the appropriate flag in the header file the shader will open a file called `rays.cmd` for writing and writes several command lines to create a geometry object called `rays` in *Houdini*. You will have to execute this commands later from *Houdini*.

The shader init part for each instance simply writes a line to standard output:

```
mi_info("dbg_lens_init [shader instance init]");
```

The same is true for the shader exit part for each instance:

```
mi_info("dbg_lens_exit [shader instance exit]");
```

The common shader exit part is called after finishing all instances of the shader and closes the file after writing several *Houdini* commands:

```

        mi_info("dbg_lens_exit [shader exit]");
        initialized = 0;
        /* Houdini stuff */
#ifdef HOUDINI_CMD
        fprintf(dbg_fp, "opadd -n switch switch1\n");
        fprintf(dbg_fp, "opadd -n merge mergel\n");
        for (i = 0; i < dbg_num_rays; i++)
        {
            fprintf(dbg_fp, "opwire -n add%d -%d switch1\n", i+1, i);
            fprintf(dbg_fp, "opwire -n add%d -%d mergel\n", i+1, i);
        }
        fprintf(dbg_fp, "opset -d on switch1\n");
        fclose(dbg_fp);
#endif

```


The *Houdini* commands create for the geometry object called rays a switch and a merge node and connect both nodes to previously created add nodes. The switch node is used to step through single rays in *Houdini* to see in which order *mental ray* creates the different rays for different shaders. The merge node allows to see all rays at the same time.

Here is the source code for the shader function called `dbg_lens()`:

```
DLLEXPORT miBoolean dbg_lens(miColor *result,
                             miState *state,
                             void *param)
{
    miVector org;
    miVector dir;

    if (!initialized)
    {
        initialized = 1;
        org = state->org;
        dir.x = 0.0;
        dir.y = 0.0;
        dir.z = -1.0;
        mi_vector_from_camera(state, &dir, &dir);
        mi_vector_to_world(state, &dir, &dir);
        mi_vector_normalize(&dir);
        mi_point_to_world(state, &org, &org);
        mi_info("[dbg_lens] incolor:  %f, %f, %f, %f",
                result->r, result->g, result->b, result->a);
        mi_info("[dbg_lens] type:  %d", state->type);
        mi_info("[dbg_lens] org:   %f, %f, %f", org.x,   org.y,   org.z);
        mi_info("[dbg_lens] dir:   %f, %f, %f", dir.x,   dir.y,   dir.z);
        mi_vector_from_world(state, &dir, &dir);
        mi_point_from_world(state, &org, &org);
        mi_trace_eye(result, state, &org, &dir);
    }
    mi_info("[dbg_lens] outcolor:  %f, %f, %f, %f",
            result->r, result->g, result->b, result->a);

    return(miTRUE);
}
```

The global initialized variable will make sure that the part enclosed in brackets is called only once. Most of the shaders modify a color. Therefore I have included very similar lines to all debug shaders. They use `mi_info()` to print debug information about the color in the current state before and after the shader has modified it. Other useful information from the current state is the type, the origin, and the direction of the current ray. The only essential function call is `mi_trace_eye()` which is emitting an eye ray into the scene and therefore is responsible for other shaders being called. The ray is emitted in negative z-direction in camera space and translated into *mental ray*'s internal space by calling `mi_vector_from_camera()`. There are several functions to convert points, vectors, and normals from and to several spaces. Before you use information from the current state you should always convert to your preferred space (e.g. world or

object space) and before you write modified points or vectors back to the current state you should convert to internal space.

A Debug Material Shader

Let's define a material shader for the box in our simple scene. The material shader will write to the same file the lens shader has opened for writing. The following lines were added to the material shader called `dbg_material` after creating a skeleton with the program `mkmishader`:

```
#include "dbg_flag.h"
#ifdef HOUDINI_CMD
extern FILE* dbg_fp;
extern int dbg_num_rays;
#endif
```

In C you can tell the compiler that some global variables are defined in another file with the keyword `extern`. The variable type has to be the same as in `dbg_lens.c`. The init and exit functions for the material shader do nothing interesting beside giving debug info as explained above with the function `mi_info()`.

```
DLLEXPORT miBoolean dbg_material(miColor *result,
                                  miState *state,
                                  dbg_material_t *param)
{
    ...
    org = state->org;
    dir = state->dir;
    point = state->point;
    mi_vector_to_world(state, &org, &org);
    mi_vector_to_world(state, &dir, &dir);
    mi_vector_to_world(state, &point, &point);
    mi_info("[dbg_material] type: %d", state->type);
    mi_info("[dbg_material] org: %f, %f, %f", org.x, org.y, org.z);
    mi_info("[dbg_material] dir: %f, %f, %f", dir.x, dir.y, dir.z);
    mi_info("[dbg_material] point: %f, %f, %f", point.x, point.y, point.z);
#ifdef HOUDINI_CMD
    fprintf(dbg_fp, "opadd -n add add%d\n", dbg_num_rays+1);
    fprintf(dbg_fp, "opparm add%d usept0 ( on ) ", dbg_num_rays+1);
    fprintf(dbg_fp, "pt0 ( %f %f %f )\n", org.x, org.y, org.z);
    fprintf(dbg_fp, "opparm add%d usept1 ( on ) ", dbg_num_rays+1);
    fprintf(dbg_fp, "pt1 ( %f %f %f )\n", point.x, point.y, point.z);
    fprintf(dbg_fp, "opparm add%d prim0 ( '0 1' )\n", dbg_num_rays+1);
    dbg_num_rays++;
#endif
    mi_vector_from_world(state, &dir, &dir);
}
```

After defining the needed variables the shader prints some debug information with `mi_info()` and if the flag is set it writes *Houdini* commands to the file

opened by the lens shader. It uses the add operator of *Houdini* to add the origin and end point of the current ray to the object called `rays` and connects both points with a line primitive.

The rest of the shader is very similar to the examples you can find in the source code for the base shaders coming with mental ray or the example described in chapter 3.8 of the book (see [2] for reference):

```
i_lights = *mi_eval_integer(&param->i_lights);
n_lights = *mi_eval_integer(&param->n_lights);
lights = mi_eval_tag ( param->lights);
for (i = 0; i < n_lights; i++)
{
    sum.r = sum.g = sum.b = 0;
    samples = 0;
    while (mi_sample_light(&color, &dir, &dot_nl, state,
                          lights[i_lights + i], &samples))
    {
        sum.r += dot_nl * color.r;
        sum.g += dot_nl * color.g;
        sum.b += dot_nl * color.b;
    }
    if (samples)
    {
        result->r += sum.r / samples;
        result->g += sum.g / samples;
        result->b += sum.b / samples;
    }
}
result->a = 1.0;

if (mi_trace_transparent(&color, state) ||
    mi_trace_environment(&color, state, &dir))
{
    result->r += color.r;
    result->g += color.g;
    result->b += color.b;
    result->a += color.a;
}

return(miTRUE);
}
```

The shader basically steps through a list of lights and adds the contribution of each light to the color given as input (`result`). Each light might be an area light. That's why you have to use a loop for each light. The function `mi_sample_light()` must be called in a loop until it returns `miFALSE`. The variable `samples` will then contain the total number of light samples taken. It may be larger than 1 for area light sources.

There are three important function calls which might result in calling another shader:

1. **`mi_sample_light()`** casts a light ray from the light source to the intersection point, causing the light source's light shader to be called. The light shader

may then calculate shadows by casting a shadow ray to the intersection point. This may cause shadow shaders of occluding objects to be called, and will also cause the volume shader of the state to be called, if there is one.

2. **mi_trace_transparent()** casts a transparency ray. It returns `miFALSE` if the trace depth has been exhausted or if the ray has left the scene without hitting any other object. If the ray direction does change you should call **mi_trace_refraction()** instead.
3. **mi_trace_environment()** casts a ray into the environment. The environment shader in the state is called to compute the color to be returned.

A Debug Light Shader

The debug light shader works pretty much the same as the material shader for the init and exit functions. It prints out debug information and adds a ray for *Houdini* in the main shader part. The most interesting part is:

```
*result = *mi_eval_color(&param->color);
if (state->type != miRAY_LIGHT) return(miFALSE); /* visible area light */
d = *mi_eval_scalar(&param->factor);
if (*mi_eval_boolean(&param->shadow) && d < 1)
{
    filter.r = filter.g = filter.b = filter.a = 1;
    if (!mi_trace_shadow(&filter, state) || BLACK(filter))
    {
        /* opaque */
        result->r *= d;
        result->g *= d;
        result->b *= d;
        if (d == 0) return(miFALSE);
    }
    else
    {
        /* transparent */
        omf = 1 - d;
        result->r *= d + omf * filter.r;
        result->g *= d + omf * filter.g;
        result->b *= d + omf * filter.b;
    }
}
mi_info("[dbg_light] outcolor:  %f, %f, %f, %f",
        result->r, result->g, result->b, result->a);
return(miTRUE);
```

The first line gets the light color from one of the shader parameters. The second line checks if the light is an area light with geometry which was hit by another ray (no light ray). The rest of the shader is basically taken from the `baselight.c` source code which can be downloaded from *mental image's* ftp

server and implements the shader called `mib_light_point`. The most important call is `mi_trace_shadow()` which computes shadows for the given light ray by casting shadow rays.

A Debug Shadow Shader

As described in the previous section, light shaders may trace shadow rays between the light source and the point to be illuminated. When this ray hits an occluding object, that object's shadow shader is called, if present⁷. Shadow shaders accept an input color that is dimmed according to the transparency and color of the occluding object. If the resulting color is black, that is, if the object is fully opaque, the shadow shader should return `miFALSE`.

The most interesting part of the `dbg_shadow` shader are the following lines:

```
if (state->options->shadow == 's')
{
    /* next segment */
    mi_trace_shadow_seg(result, state);
}
```

The function `mi_trace_shadow_seg()` recursively calls the shadow shader for the next shadow segment and returns its result, or the result of the light shader if there is no more shadow intersection. It does nothing if shadow segments are turned off. We will talk about this later but you should realize that the shader checks first if the option `shadow segments` is used before it calls the function.

A Debug Volume Shader

Volume shaders are normally called in three situations.

1. When a material shader returns.
2. Unless the `shadow segment mode` is in effect, volume shaders are also called when a light shader has returned. In `shadow segment mode`, volume shaders are not called for light rays but for every shadow ray segment from the illuminated point towards the light source.
3. Finally, volume shaders are called after an environment shader was called.

A Debug Fog Shader

The fog shader called `dbg_fog` differs not much from the previous volume shader called `dbg_volume`. There is nothing special for this two shaders except that the `dbg_volume` will be attached to geometry like a cube and the `dbg_fog` shader will be attached to the camera.

⁷If the object has no shadow shader, the object is assumed to block all light.

Houdini Experiments

Let's start with some experiments in *Houdini*. Start *Houdini* in the directory where you have compiled the debug *mental ray* shaders. Load the scene we created before and create an output ROP for *mental ray*. Activate the **Generate Script File** option. As **Script File** type `test.mi`. If you go to the shaders section (SHOPs) you should be able to create a **Debug Lens Shader**. If you can't find an entry for this shader you should read the section called "Integrate Mental Ray Shaders Into Houdini" again. You can activate the **Override Default Res** option for the *mental ray* ROP and specify a resolution of 1 in each direction. Go to the **Specific** tab and select the **Debug Lens Shader** you have created before for the **SHOP lens** parameter.

If you hit the **Render** button for the *mental ray* ROP you should find a file called `test.mi` in the directory where you started *Houdini* from. If you look into this file you will find two lines where for the `dbg_lens` the shared library is linked and the corresponding MI file is included. In the camera definition you will find a line where the shader is attached to the camera.

It might happen that you run into problems during rendering the `test.mi` file with `ray -verbose on test.mi`. If so look into the `test.mi` and change the two lines containing the word `socket`. One of this lines should be found in the camera definition and you should change the string containing the `socket` word into `"test.rgb"` or something similar. The second line will be a system call to `imdisplay` and you can put a hash (#) character in front of this line to deactivate this command.

If you render again you should find a 1x1 image with the name `"test.rgb"`. This is not very interesting but you should read the output of the render. If you want to store the output of the renderer in a file you should use the command `ray -verbose on test.mi >& log.txt` for rendering. It will redirect the output to the file called `log.txt`. You will find several lines concerning the debug shader:

```
...
LINK 0.0  progr: module #2 (./dbg_lens.so) loaded
...
PHEN 0.0  info : shader "dbg_lens" sets scanline off
...
PHEN 0.0  info : dbg_lens_init [shader init]
PHEN 0.0  info : dbg_lens_init [shader instance init]
PHEN 0.0  info : [dbg_lens] incolor: 0.000000, 0.000000, 0.000000, 0.000000
PHEN 0.0  info : [dbg_lens] type: 0
PHEN 0.0  info : [dbg_lens] org: 0.000000, 0.000000, 0.000000
PHEN 0.0  info : [dbg_lens] dir: 0.000000, 0.000000, -1.000000
PHEN 0.0  info : [dbg_lens] outcolor: 1.000000, 1.000000, 1.000000, 1.000000
PHEN 0.0  info : [dbg_lens] outcolor: 0.000000, 0.000000, 0.000000, 0.000000
PHEN 0.0  info : [dbg_lens] outcolor: 0.000000, 0.000000, 0.000000, 0.000000
PHEN 0.0  info : [dbg_lens] outcolor: 0.000000, 0.000000, 0.000000, 0.000000
...
```

```
PHEN 0.0 info : dbg_lens_exit [shader instance exit]
PHEN 0.0 info : dbg_lens_exit [shader exit]
```

The shared library for the shader is loaded, scanline rendering is turned off because the MI file of the shader contains a line for this, the shader is created and an instance is used for rendering. You will see only one line for the `incolor` but 4 lines for the `outcolor`. This happens because the shader is called 4 times even if the resolution is set to 1x1. The `initialized` variable prevents that the main part of the shader containing the `mi_trace_eye()` call is executed more than once.

Another effect of the rendering was that a file called `rays.cmd` was created and it contains commands for *Houdini*. Try to load this file into *Houdini* with source `rays.cmd` in a **Textport** window. A new geometry object will be created with a merge and a switch node. Delete this geometry node before you start the next experiment.

Now create a **Debug Light Shader** and a **Debug Material Shader** and attach it to the light and the box. This time you will attach the shaders in the **Shading** tab of the light (parameter **SHOP Light**) and the geometry object (parameter **SHOP surface**). Set the light color of the shader (not the light object) to white (`rgba = 1.0`) but leave the **shadow** parameter turned off.

If you render again and load the resulting *Houdini* command file in the **Textport** (with source `rays.cmd`) you should get several add nodes connected to the merge and switch nodes. If you turn on the **Display** flag for the merge node you will see all the resulting rays in *Houdini*⁸, if you turn on the **Display** flag for the switch node you can step through the rays and see the order of them by dragging the **Select Input** slider.

I wrote a little *Python* script called `sortRays.py` to sort the rays by reading the output log file `log.txt` of the rendering. If you run it you will get only the output writing `material.cmd` This is a result of having only messages from the material and the lens shader in the log file. Let's turn on the **shadow** parameter of the light shader and render again. After running the *Python* script you should get several *Houdini* command files:

```
writing material.cmd ...
writing light.cmd ...
```

If you execute the commands from within *Houdini* you get separate rays for each type of shader. You can use this to give the different rays different colors for each shader type (e.g. light rays could be yellow and material rays blue).

Now let's try some variations. Move the light source to (0, 1, -1). If you render again and visualize the resulting rays in *Houdini* you will notice that the

⁸see figure 2

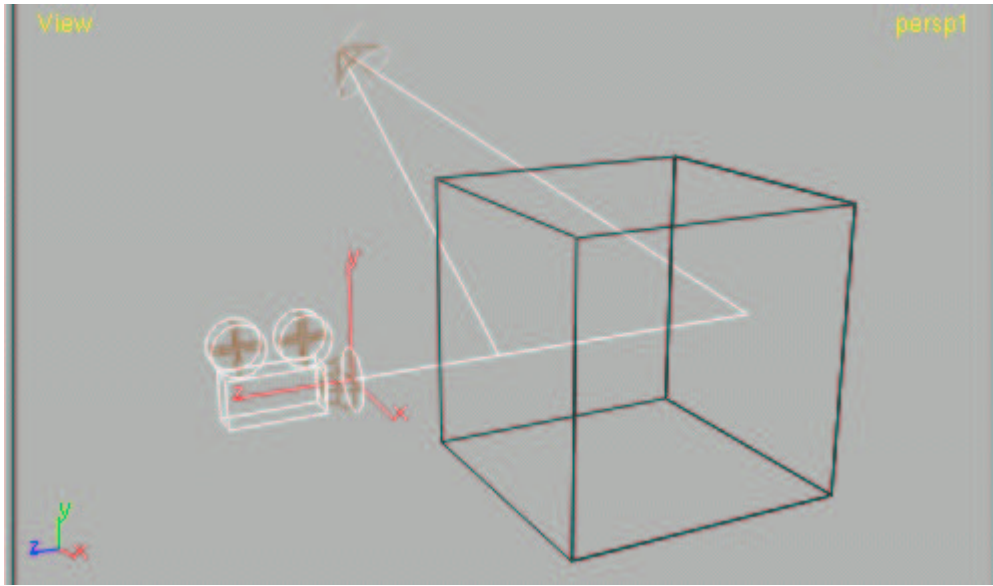


Figure 2: The merged rays.

first light ray for the front face of the cube is now missing. The back face still has a light ray. If you want to change this behavior of mental ray you have to modify the internal state. In chapter 3.10 of the book [2] you will find some information about volume shaders and how you can turn off certain optimizations such as backface elimination.

Now add a volume shader to the box geometry by specifying the volume shader `dbg_volume` with the parameter **SHOP shader**⁹. You can also add a shadow shader with **SHOP shadow**. After that you can copy the geometry and light objects several times and create a more interesting scene with 3 boxes and two lights. The camera will get a fog shader. Maybe you would expect to attach the volume shader for the fog to the camera or to the *mental ray* ROP. But the way to do this in *Houdini* is to create an atmosphere object and specifying the shader in the Shading tab of this object. After rendering this scene the sorting of the rays should create more files for Houdini:

```
writing material.cmd ...
writing light.cmd ...
writing fog.cmd ...
writing shadow.cmd ...
writing volume.cmd ...
```

⁹Maybe **SHOP volume** would have been a better name.

In *Houdini* you can load all this different rays and give them different colors. The merge and switch nodes allow you to select and visualize the rays you want to see. The best approach is to step through the log file to see the timing¹⁰ and to read the chapters of the book to understand why a shader is called at the current time. Maybe as a last test, before I leave you alone to think about your own tests you want to make, change the **Shadows** parameter in the **Specific** tab of the *mental ray* ROP to see the different ways how the material, the light, and the shadow shaders are cooperating and how the different settings affect the timing of the shader calls.

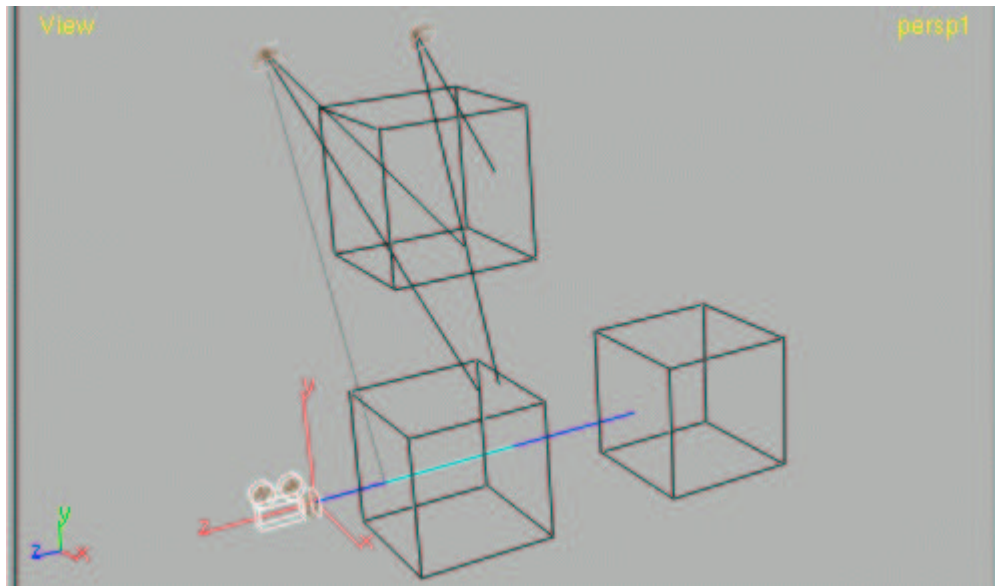


Figure 3: The sorted rays in different colors.

There are four options in *Houdini* which result in a similar entry in the *mental ray* options section of the MI file:

1. **shadow off**: This statement disables all shadows.
2. **shadow on**: Simple shadows are enabled. This is the most efficient and least flexible of the three shadow modes. If shadows overlap because multiple objects obscure the light source, the order in which these objects are considered (and their shadow shaders are called) is undefined. If one object is found to completely obscure the light, no other obscuring objects are considered. This statement turns off **shadow sorting** and **shadow segments**.

¹⁰In which order are the rays created?

3. **shadow sort**: This shadow mode enables shadow sorting. It is similar to the preceding shadow mode, but ensures that the shadow shaders of obscuring objects are called in the correct order, object closest to the illuminated point first. This mode is slightly slower but allows specialized shaders to record information about obscuring objects. If no such special shader is used, this mode offers no advantage over simple **shadow on**.
4. **shadow segments**: Like with **shadow sort**, the shadow shaders are called in order. Additionally, shadow rays are traced much like regular rays, passing from one obscuring object to the next, from the light source to the illuminated point. Each such ray is called a shadow segment. This slows down rendering, but is required if volume effects should cast shadows (such as certain complex shaders like fur and smoke volume shaders). This mode requires support from the shadow shader, which must use the function call `mi_trace_shadow_seg()` to cast the next shadow ray segment.

My last comment concerns the volume shaders. Maybe you will have noticed during your experiments that the fog shader attached to the camera and the volume shader attached to the boxes are not alternating in the expected way. I leave this for you as an exercise because there are two ways how to deal with that:

1. Either you do the inside/outside calculation for your volume (e.g. the box) yourself. Then you have to consider the fact that the camera can be inside the volume or that other objects might be in your volume.
2. Or you use the new feature of *mental ray* version 3.0 and later called **auto-volume**. This feature is **not** explained in the books [1] or [2] but you can read in the the online documentation coming with *mental ray* about it.

I guess this was a lot of information even if the examples were really very basic. I hope you liked this tutorial and it would be nice to get some feedback from you, the reader. Please send your questions or comments via email to either wahn@acm.org or to jan@millfilm.co.uk.

References

- [1] Thomas Driemeyer. *Rendering With Mental Ray*. Springer–Verlag, 2000.
- [2] Thomas Driemeyer. *Programming Mental Ray*. Springer–Verlag, 2000.