CHAPTER 1

Introduction to mental ray

mental ray, a robust and independent rendering package, is well integrated within several 3D applications—Autodesk Maya, Autodesk 3ds Max, and Softimage|XSI, for example—as plug-in software. As a plug-in application for these host programs, mental ray specializes in generating photorealistic images, with an unsurpassed ability to re-create natural phenomena. As you'll see, it can also be used for creating nonphotorealistic contour renderings (NPRs). mental ray's photorealistic capabilities derive from an extensive set of tools that perform advanced camera, light, surface, and volume shading simulations. These simulations and the realistic renderings they allow lend themselves to various applications: architectural design, motion picture animation and visual effects, high-end television commercials, automotive and industrial design, and games. In a nutshell, mental ray provides outstanding render quality and an unsurpassed set of tools that cope with complex rendering challenges such as indirect illumination, volumetric lighting, memory handling and optimization, cross-platform network rendering, flicker reduction, and much more.

This chapter is an overview of mental ray, introducing the key terms and concepts that you'll examine in detail throughout this book. It provides a detailed introduction to what goes on in a mental images (.mi) file, not because I expect you to create such files by hand-coding but because understanding mental ray's functionality and settings lends to a better understanding of its features from within each host application. The chapter covers the following topics:

- **What Is mental ray?**
- **Why Use mental ray?**
- **The Structure of mental ray**
- **mental ray Integration**
- **Command-Line Rendering and the Stand-Alone Renderer**
- **mental ray Shaders and Shader Libraries**
- **Indirect Illumination**

What Is mental ray?

The main focus of the mental ray product technologies is the generation of photorealistic images, a process that requires complex computations using physics to simulate the way light behaves and interacts with surfaces. Light physicists have long been in pursuit of the definition of light, particularly for demystifying the *photoelectric effect*, which describes the reflection of light from surfaces and its physical characteristics and nature. Computer graphics (CG) software developers are particularly interested in applying the knowledge of lighting and its interaction within an environment, as well as its perception by the human eye, into shading and lighting models within their applications. This might explain why shading models developed by different scientists have inherited their names within our 3D applications. The Lambert shading model, for example, is the same in every application and provides the base model for all diffuse shading models; it is then extended to provide enhanced shading characteristics such as with a Blinn, Phong, Oren-Nayar, or any other shader. You will learn more about light and CG shading models in Chapter 10, "CG Shading Models and Light Primer."

mental ray is a product from a rich line of specialized tools developed by mental images GmbH, a company based in Germany. Most users, however, access mental ray primarily through OEM partners such as AutoDesk, Avid, and others. These partners offer mental ray both as an integrated render plug-in within their software with support for *satellite* rendering (also known as distributed rendering), and as a stand-alone renderer (sold separately).

mental ray Scene Description Language

At its core, mental ray is a fully functional 3D package that enables you to describe a scene consisting of geometric forms, surface materials, lights, environment shaders, and a camera using the mental images scene description language. For describing such scenes, mental ray does not provide its own user interface. Thus, you can simply describe a scene using plain text and a text editor, resulting in a rendered image upon execution. This sort of text file, known as a mental images file, is then executed from the command line simply by entering the default mental ray render command `ray` followed by a filename:

```
ray myfile.mi
```

Because of the efficiency and control it provides, this approach has many useful benefits for production houses that have the development resources, but not as many for the independent artist or smaller CG or CAD shop. Without a UI, mental ray is not very intuitive to use. These users normally would not consider purchasing a stand-alone render program; they are more likely to consider a package such as XSI, Maya, or 3ds Max, which provide a full set of tools and a user-friendly graphical interface. Artists with strong programming skills (and sometimes a computer science degree) can use the mental ray stand-alone

package to further customize rendering as well as design several additional features; for example, using the C or C++ programming language, they can design custom shaders that can then be added to the mental ray shader libraries. This obviously requires familiarity with the scene description language, C or C++, and mental ray shader libraries and their implementation.

Most of this book assumes you are using mental ray from within a host, but it also refers to using stand-alone mental ray where appropriate. mental ray options are labeled differently within each package. I will always refer to mental ray options as they are implemented (labeled) in mental ray and then demonstrate how to access those same options from within the different packages. With mental ray, render options are defined within a mental ray .mi file's *options block*. The options block options can then further be overridden on the command line with stand-alone rendering using similar syntax for options found in the options block. In most cases, syntax presented in this book refers to command-line commands that may be used to override option block options. When an option is unavailable as a command-line command, the options block syntax is presented. Non–options block options, such as internal options for lights and shadows, are presented in their relevant context. These options and their syntax and execution become clearer as you read through this chapter.

Host Translators

Figure 1.1
This feedback on the Maya command line indicates that the Maya scene is being translated for rendering with mental ray.

When using mental ray through a host application, it is not necessary to be familiar with the mental images scene description language. In fact, no knowledge of any mental ray programmable features is required in order to take full advantage of mental ray. During the render, the host application will automatically translate the scene into a mental images format that is then rendered with mental ray. (For example, when a render is executed in Maya, we can see the command line progress feedback, shown in Figure 1.1, indicate that the current frame is first being translated prior to indicating any render progress.) Hence, mental ray can render an image directly from within these host applications without any need for you to manually provide any programmable settings or help with the translation process.

This sort of mental ray integration from within host packages provides access to most mental ray features. It is what makes mental ray a practical tool for the CG artist, eliminating the need for advanced technical skills and allowing you to focus on your art. The integration of mental ray within OEM partner packages that will be discussed throughout this chapter is achieved through a structural component known as the host translator program deals with translation, supporting mental ray features within host applications. Thus, the translator program interfaces between the host application and mental ray while executing renders, exporting .mi files, or calculating mental ray specific maps such as photon maps, final gather maps, and light maps. For now, let's discuss some of the requirements and

goals of industry professionals from a wide variety of professions, as well as how mental ray caters to their needs.

Why Use mental ray?

Professionals in the various fields that use mental ray have different purposes in generating images, and those differences are reflected in the ways they work with the software and how they customize their production pipeline. The different approaches may range from a simple out-of-the-box rendering to advanced customized tools developed in-house, such as with Sony Image Works, Industrial Light & Magic, and other large-scale production houses.

Architectural and Industrial CAD

With architectural or industrial CAD rendering, usually a focus on establishing realism based on physically correct calculations is imperative. Architecture professionals are particularly interested in drawing a realistic image that represents an environment's appearance at a particular time of day or with a specific type of artificial lighting. This may require using *light profiles* (provided by light manufacturers) that specify the exact light intensity and falloff characteristics of a particular light source. mental ray then adds to these light models additional abilities to simulate light bounce within that environment; this is known as the *indirect illumination* of surfaces by reflected light. (You can learn more about indirect illumination later in this chapter and in Chapter 12.)

Industrial designers usually aren't concerned with simulating specific lighting conditions, so they have the creative freedom to seek a more aesthetic lighting scenario over a physically correct one. Their rendering focuses on generating realistic characteristics for surfaces and their interaction with light. They need to simulate realistically how surfaces reflect and transmit light. For example, chrome surfaces, aluminum, "heavy" metal, plastic, brushed metals, translucent surfaces, and glass of varying thickness and type all interact differently with light.

Chapters 10 tand 11 will guide you through several approaches for creating complex surface shaders and custom effects. In both architectural and industrial design, render times can be quite long, but this is not normally a serious obstacle. These fields in many cases may require rendering only a relatively small sequence of frames for print, a high-end commercial, or a video presentation.

Entertainment

mental ray's photorealistic capabilities are equally important in the entertainment industry, but the sheer number of frames to be generated means that another component needs to be considered: time. While beautifully rendered CG images may greatly increase a film's appeal, they must also be generated in a timely manner. For this reason, film productions usually prefer to avoid using mental ray's powerful *raytracing* abilities whenever possible (see Chapters 2 and 5, "Rendering Algorithms and Quality Control,"), and they expect a

fast turnaround in the production pipeline. In the entertainment industry, mental ray plays two different roles: one as a primary renderer for entire productions and the other as an additional renderer providing high-end realistic visual effects shots that emphasize realism.

For feature animated films, currently the norm is to use a RenderMan-compliant renderer, typically Pixar's PRMan, which provides a scalable and fast *scanline* renderer as well as a powerful raytracer when needed. However, some projects use more than one renderer in a production pipeline and divide the work among different studios that each assemble specific shots using their tools of choice. Typically, films that combine live action with 3D use mental ray more often than feature animations do. Some familiar feature films that have used mental ray in part or in full are *The Wild, The Matrix Revolutions*, *The Matrix Reloaded*, *Star Wars: Episode II*, *The Hulk*, *Terminator 3*, *Fight Club*, *Panic Room*, *Blade Trinity*, *The Cell*, *The Day After Tomorrow*, and *Walking with Dinosaurs*.

When simulated photorealism is used in films, it's is usually to create props or scenes that it would be too costly to build, such as spacecraft. But it also allows shots that otherwise would be impossible or too expensive to shoot. In some cases, mental ray is used to clone an environment or character into CG, enabling the director to obtain nonstandard camera shots. For example, in the Motion Pictures gallery on `www.mentalimages.com`, you'll find some images from *Panic Room*, a nonfuturistic film that at first glance does not appear to be loaded with special effects or 3D. It takes place in a New York City townhouse where everything appears to be real. However, it uses several shots that probably would have been impossible or at least very difficult to manually construct within a set. mental ray was used to render a replicate environment of the townhouse so that the nonstandard camera motions through the house could be shot. This sort of integration between real life and 3D requires a great deal of realism. Its goal is to prevent the viewer from distinguishing between real shots and CG-enhanced shots.

Games

In computer games, which are constantly evolving and offering more "realistic" experiences, the emphasis on complex, instantaneous interaction has always put high-end rendering out of reach. Enhanced CG requires complex shading models and lighting such as simulating indirect lighting. Games do not consist of images prerendered using mental ray or any other renderer; they run on a game engine that renders in real time. This real-time display is enabled using technologies that access and control hardware, through OpenGL or DirectX, and based on the hardware abilities such as with NVIDIA, 3DLabs, ATI, and other manufacturers' boards. The games industry bridges the technical gap by using mental ray's *light baking* options. Light baking is the process of converting surface shading and lighting from mental ray into texture maps that can then be applied to models. Thus, baked texture-map files may include surface-shading properties and their influence from direct and indirect lighting. In essence, textures can represent a *global illumination* (see the section "Indirect Illumination" later in this chapter) render that provides the indirect diffused

light bounce within an environment, providing for more appealing texture maps for game environments and characters. You will learn techniques and considerations for light baking in Chapter 14, "Light Maps (Baking) and Complex Shaders."

The Structure of mental ray

In the following sections, you will learn more about mental ray rendering procedures, as well as its integration with other applications. The goal is to provide a solid understanding of mental ray technologies and abilities so that when you're evaluating rendering technologies, you will be able to weigh one advantage against another. Being more familiar with the core technology and its algorithms will enable you to make a better decision for your rendering approach.

A mental ray file, regardless of whether it was generated within a host application or was custom-made, typically includes information on the spatial arrangement of objects, their physical characteristics within a given coordinate space (*object space*, *camera space*, *world space*), and their influences from a variety of shaders. When rendering, mental ray transfers data by sampling shader color values, typically from surface points within the file (the 3D scene). These color values are then passed into a 2D *frame buffer*, which acts as a storage container for the different rendered data. Frame buffers typically store the four standard color channels that represent an image, *RGBA*. The R, G, and B channels each represent a different additive primary color (red, green, and blue) and A represents the alpha masking channel. Image data is always stored within frame buffers until the render process has completed and the frame buffer is ready to be written to a file on disk.

> mental ray also supports several other custom channels that will be discussed within the book (in Chapter 3, "mental ray Output"), such as the Z-depth channel and motion vectors.

Photorealistic rendering requires a lot of processing, so mental ray is structured in a form that maximizes performance during rendering. In Chapter 2, you will learn more about the different rendering algorithms mental ray uses, which include different algorithms for scanline rendering, raytrace rendering, and hardware rendering.

Modularity

An important aspect of mental ray's structure is modularity. That is, mental ray is divided into several separate modules that act as software components. Each module is essentially plugged in to mental ray and is responsible for providing very specific tasks. For example, the image (IMG) module will load, write, or convert images to *memory mapped images* (see Chapter 11, "mental ray Shaders and Shader Trees") during the render or when prompted to do so with the mental ray *imf_copy* utility (described in the section "mental ray Components and Application Files" later in this chapter). For example, you may use the imf_copy

utility to convert image types, creating memory-mapped images by utilizing functions from the IMG module. Hence these modules, which are at the core structure of mental ray, are in fact "plug-in units," which, when combined, form a larger, more-flexible system with improved capabilities.

Another aspect of modularization is that all mental ray shaders are provided as external plug-in programs designed for very specific tasks. These external shader libraries provide a great deal of flexibility in developing custom tools and shaders, not just for programmers, but for artists as well. For example, it is fairly simple and straightforward for artists to append new shaders or shader libraries to mental ray, just as you can download and import shaders into XSI, Maya, or 3ds Max. With mental ray, a new shader can be implemented so that it is always available simply by adding its declaration file into the shader libraries and linking it to mental ray. Shader libraries can be found in mental ray's root directory for each package, as discussed in the section "mental ray Shaders and Shader Libraries" later in the chapter. With respect to modularity, shaders open the door to an ongoing development process, which not only adds new shaders through a shader library, but primarily leads to finding new and creative ways for blending several shaders into one complex shader tree, forming a more robust material shader for surfaces or any other special effects.

On-Demand Execution and the Geometry Cache

When a render is executed, mental ray constructs a *scene database,* which is stored within the *geometry cache* and contains all the relevant information mental ray currently requires for executing the render. In essence, mental ray manages the cache in a way that allows for information to be loaded and unloaded into the database while maintaining efficient memory handling. Beginning with mental ray 3.*x*, the scene information is loaded into the database on demand. Briefly, here's how mental ray's render management process works.

mental ray 3.*x* divides a render task into different *render jobs*, which are structured based on some form of dependency so that they may be executed in the most efficient way. In previous versions (2.*x*), mental ray would execute the render in consecutive phases. For example, all the geometry would first be loaded into the cache and then tessellated before a following phase could commence.

> *Tessellation* refers to all geometry—be it NURBS, polygons, or subdivision surfaces—in the scene that must be converted (tessellated) into polygonal triangles before rendering the geometry. This task has two primary phases: first loading the geometry into memory so the renderer is aware of its existence, and then tessellating it into triangles.

mental ray 3.*x* and later seek to optimize data flow with on-demand execution of jobs. Thus, jobs are executed when needed rather than in a predetermined order. A job can execute any type of task, such as tessellation, raytracing, calculating light maps, managing

texture data, rendering portions of the frame, and so forth, all based on the job status and data flow. This means that memory can be handled more effectively. Essentially, during rendering, most of the data that is being provided by the geometry cache to ongoing jobs is stored in memory and within the machine limits. Note that the cache can store all types of information, which may include spatial positioning, geometric tessellation data, texture maps, photon maps, Final Gather points, and any other data it may require for rendering. As the cache grows, the memory usage increases gradually until the machine limit or a specified limit is reached. If the limit is reached, mental ray will dump certain information from the cache to enable an ongoing render. This job-based model helps mental ray 3.*x* improve memory handling and optimization.

Enabling Message Logging and Verbosity Levels

mental ray provides message logs that are output into the console window when rendering so that you can track the rendering progress as well as retreive render statistics on the "quality" of the render, which helps troubleshoot or further optimize a render. You can control what information is displayed by enabling different levels of verbosity by using the mental ray `verbose` command either from within a host application or on the command line. When using a command-line renderer, you enable verbosity by specifying the `-v` (flag) and a verbosity level, as seen here for mental ray stand-alone rendering, Maya, XSI (`-verbose`), and 3ds Max command-line rendering. The topic of command-line rendering is discussed further in the section "Using the Host Application's Command Line" later in this chapter.

COMMAND-LINE CODE	SOURCE APPLICATION
`ray -v 5 myfile.mi`	mental ray stand-alone
`render -r mr -v 5 myfile.mb`	Maya command line
`xsi -r -verbose "prog" -scene fileName.scn`	XSI command line
`3dsmaxcmd -v:5 "scenes\anim.max"`	3ds Max command line

The message log can be viewed in different places depending on the host application that is executing the render. When using the mental ray stand-alone renderer or command-line rendering, the message log appears in the console window, which you'll see in Figure 1.5 in the next section. There are seven levels of verbosity; each level builds on the previous level, introducing more information into the output console. In general, default verbosity is set to level 2, and when enabled (verbosity specified without including a specific level) it defaults at level 5. For most troubleshooting, and as a general method to keep track of render progress, verbose levels 4 and 5 are useful. The following table describes the different levels of verbosity:

VERBOSITY LEVEL	MESSAGES LOGGED
0	No messages
1	Fatal errors
2	Non-fatal errors

VERBOSITY LEVEL	MESSAGES LOGGED
3	Warning messages
4	Informational messages
5	Progress messages
6	Debugging messages
7	Verbose debugging messages

Follow the steps presented for each host application to enable verbosity:

Maya

1. From the top menu bar navigate from Render → Batch Render or Render Current Frame and select the options box to reveal their attribute windows, as seen in Figure 1.2 for batch rendering. Both have similar settings with a difference in purpose. The Render Current Frame executes a render in the Render View, and the Batch Render is used for executing animation sequences.
2. From under Verbosity Level dropdown list you can specify the message level as seen in Figure 1.2, where Progress Messages are highlighted. Maya offers verbose levels 0 through 6.
3. When rendering using the Render Current Frame attribute (or icon shortcut), Windows users will see the output displayed in the Maya Output Window; OS X users will see it within a console window. (OS X users, note that you need to run Maya from the Maya console for this to function correctly.)
4. When rendering using the Batch Render attribute the verbosity output is saved in the Maya Render Log text file. The file is located under the user\My Documents\maya directory. Note that in this mode the verbosity is not visible during the render in the Maya Output window.

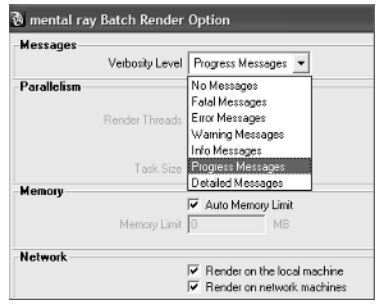


Figure 1.2

Enabling verbose message output from Maya. You'll see the resulting messages in the Maya Output window, the Maya console (OS X), or in the Script Editor.

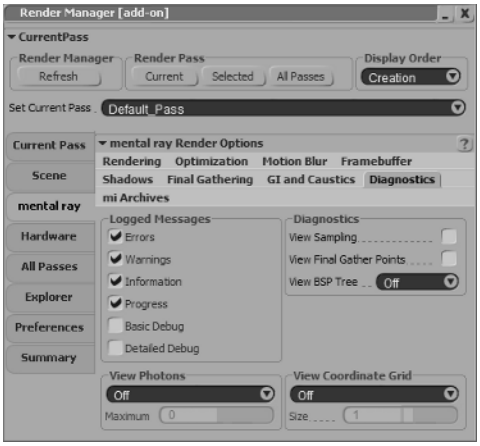
In previous versions of Maya (8.0 and lower), verbosity options are located under the Render Settings window → mental ray tab → Translation rollout.

XSI

1. Navigate from the top main menu or the Render toolbar (on the left side) to Render → Render Manager → mental ray tab (from the left column) → mental ray Render Options rollout → Diagnostics tab as seen in Figure 1.3. When per pass mental ray options are in effect use the Current Pass → mental ray Render Options, as further discussed in the sidebar "The Render manager in XSI 6.0".
2. You will see a list of verbosity levels under Logged Messages.

3. Enabling Progress messages as shown in Figure 1.3 will enable level 5 verbosity. XSI verbose messages offer levels 2 through 7.
4. When rendering within XSI, you can see the verbosity output within the Script Editor window.

Figure 1.3
Enabling verbose message output from XSI. You'll see the resulting messages in the Script Editor.



3ds Max

1. From the Main Menu bar, navigate to Rendering → mental ray Message Window.
2. This window, as shown in Figure 1.4, enables both specifying verbosity levels and viewing the output results (mental ray progress) while rendering.
3. By specifying Information, you enable verbosity level 4. 3ds Max offers levels 2 (Open on Error), 6 (Debug), 5 (Progress), and 4 (Information) at the bottom portion of the mental ray Messages window. The top portion of the window also specifies information regarding the number of CPUs and threads that are being used during the render.
4. From under the Main menu bar Customize, select Preferences… to reveal the Preference Settings window. Under the mental ray tab → Write Message to File parameter enables specifying an output log file that stores the verbosity messages as plain text, based on the parameters defined under the mental ray Messages window. The Append to File will enable adding these messages into an existing log, rather then overwriting the file.

With all these host applications, using the command-line renderer allows you to specify any verbosity level, even if it does not appear within the host UI.

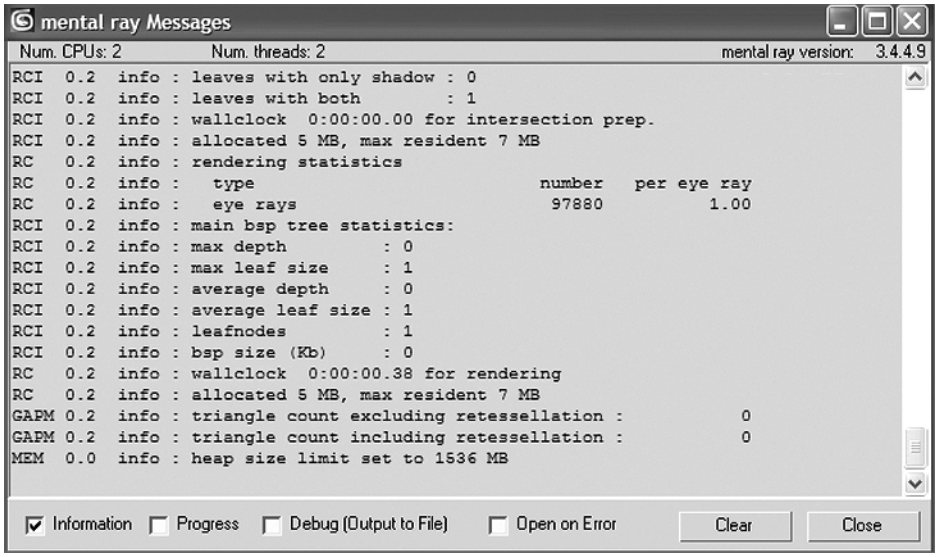


Figure 1.4

Enabling verbose message output from 3ds Max. The mental ray Messages window allows you to both set the verbosity level and see the output during rendering.

THE RENDER MANAGER IN XSI 6.0

The Render Manager window provides a more effective way to organize XSI passes and scene options. Essentially, the underlying concept is that settings are generally specified globally affecting all scene passes, however, you can then further apply independent per pass settings (overrides) when required. I will usually avoid any reference to scene (global) or per pass (local) options unless specifically required, as in both cases the mental ray options have the same relevance, just the context of pass or scene may differ. The concepts of passes and output are discussed in more detail in Chapter 3.

Consider that if you navigate in the Render Manger window to the Current Pass (in the left column) → Pass mental ray → mental ray Render Options rollout, the same mental ray options appear, which are tied with the global mental ray render options found under the mental ray tab (from the left column). Detaching this dependency is applied by pressing the Make Local to Current Pass option under Current Pass → mental ray Render Options → Rendering tab, which breaks the automatic linkage between global mental ray options (in the left column) and the current pass mental ray options. In both cases all the options are equivalent only with a difference in purpose, which is per pass, or globally for the scene. Once detached clearly the per pass options take effect.

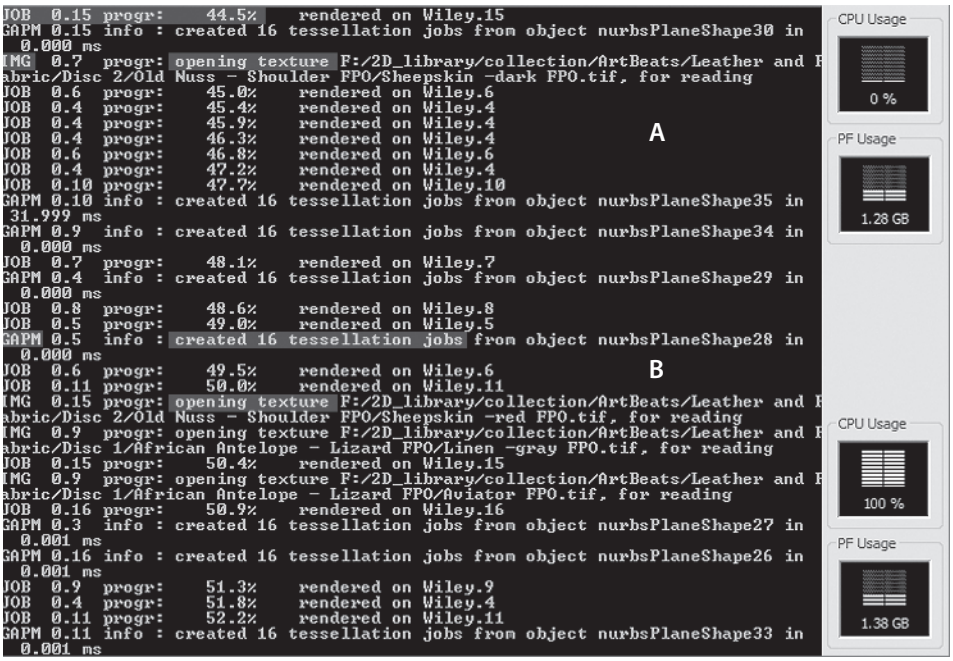
In addition, the Current Pass → Pass Output tab also has a global dependency driven by the Scene tab (on the left column). It too can be disconnected by specifying different options under the Current Pass → Pass Output → Output tab. For example, if you look under the Scene tab, notice that the Scene Globals→ Scene Renderer dropdown list is set to mental ray, defining mental ray as the current scene renderer. If you look under Current Pass → Output → Pass Renderer dropdown list, notice that Use Scene Render Options is specified as a default, deriving the scene renderer from the previous (global) scene parameter. Thus you can always override this local option specifying the hardware renderer, or specifically specifying mental ray (if it differs from the scene global option). The same is true for the remaining options found under the Pass Output tab. Notice that when you begin to enable per pass options their relevant properties appear, enabling you to specify per pass overrides.

An Example of On-Demand Execution

Now that you understand how to output messages during rendering, let's look at what they tell you about on-demand job execution. Figure 1.5 illustrates part of the message log for a render using verbosity level 5. I've added labels and highlighting so that you can follow the discussion.

Figure 1.5

mental ray's message log in the console window. Here you can identify the render progress as well as troubleshoot problematic renders.



The first highlighted line reads as follows:

```
JOB 0.15 progr: 44.5%     rendered on Wiley.15
```

All lines are formatted in a similar way, providing information from various mental ray modules and jobs. From left to right, the line tells us that the module JOB (a general indication of the render progress of a specific job, or render task, unique to mental ray 3.*x* and up) is currently operating on the following machines and threads. The machine reference is the first number (0) followed by a decimal point and the thread number (15). Machine 0 indicates the client machine that initiated the render, the machine that is currently being used to submit the render. The following message deals with the type of message this line is providing and its verbosity level, hence a progress (level 5) message, which is then followed by a plain-English description for the current information. The description confirms that mental ray has completed 44.5 percent of the rendering on this machine (0 - Wiley) using thread number 15, which is clearly a progress message on the status of a particular job, hence a level-5 verbosity message.

In section A of Figure 1.5, the highlighted text reads as follows:

```
IMG 0.7 progr: opening texture F:path…, for reading
```

This line tells us that the IMG module is providing the progress message and that this module is currently loading a texture file into memory. The messages in section B begin as follows:

```
GAPM 0.5 info: created 16 tesselation jobs from object...
```

This line indicates that the GAPM module, which deals with geometry approximation, is creating new tessellation jobs. Note that 50.0 percent of the render has already completed and mental ray is still initializing new tessellation jobs during the render. This should provide some insight into how mental ray actually utilizes the on-demand rendering process in practice. That is, geometry will be calculated only when needed rather than as prerequisite for rendering.

Because mental ray (3.x and up) executes only on-demand jobs, it does not need to tessellate an entire scene, only the elements that are required by the jobs. This essentially enables mental ray to ignore geometry that is not needed for a particular frame, even if it exists within the scene. Only geometry that is needed for a given frame will be tessellated. Further, if geometry is no longer needed, mental ray can clear it from cache, freeing up space for new geometry based on demand. Of course, this approach also has disadvantages: If the memory limit is exceeded, mental ray may dump the cache of geometry that will be needed in a subsequent frame, or even within that same frame (the later is unlikely). If geometry is removed from the cache, then that geometry will need to be recalculated the next time it needs to be used, such as in a subsequent frame. This sort of memory-dump behavior may be required for extremely "heavy" scenes or machines with a low memory capacity. On the other hand, if mental ray recognizes that the geometry will be needed in subsequent frames or by other jobs, it will try to keep it in the cache as long as there is sufficient memory to support completing other ongoing jobs. Essentially, any type of data may be loaded or removed from the geometry cache, based on mental ray's ability to determine the best workflow for rendering. Thus, mental ray efficiently divides the scene into small jobs that optimize render times and improve memory handling, trying to follow the most efficient path while increasing memory gradually in a stable manner.

Images loaded into memory, whether texture files, light maps, or any other image file data, have a significant impact on memory usage. Notice that Figure 1.5 displays two RAM PF Usage indicators showing how the memory usage increased after loading in additional images between 40 percent and 50 percent of the render completion. mental ray tries to improve image handling in different ways. For example, additional boosts in performance are provided by mental ray's ability to consider partial *Shadow maps* that can be quite large in file size. (You'll learn more about shadow maps in Chapter 7, "Shadow Algorithms.")

Dividing a rendering task into jobs also helps mental ray maximize performance by taking advantage of multiple processors on a single machine (*thread parallelism*). With multiple machines, *network parallelism* enables mental ray to use all the available processors over a network of multiprocessor machines to execute jobs. This significantly increases the ability to process large frames and data. For example, a traditional nonparallel render would render a frame on each processor, separate from the other frames. With parallelism, one frame can be computed over several processors and networked machines. Hence, network rendering can take advantage of mental ray's abilities to divide a render into jobs and effectively distribute them over the network, efficiently handling the flow of data. mental ray 3.2 and above also supports Intel's *Hyper-Threading*, when it's available.

mental ray Integration

This book focuses on using mental ray as it's integrated with three of the most widely used packages: Autodesk (Maya and 3ds Max) and Avid (XSI).

XSI has always incorporated mental ray, which is its default software renderer and currently the only software renderer that ships with it. Maya and 3ds Max both added integration with mental ray as their users came to need an alternative rendering solution. For users, this seamless integration appears simple and straightforward; behind the scenes, however, is a complex integration based on ongoing technical development. The functionality that bridges between packages is quite complex, and the integration methods may differ between packages.

Currently, the best integration is within XSI, which is designed to render solely with mental ray. One of my favorite features about XSI, which is absolutely invaluable, is its ability to continuously update a rendered region regardless of the components that are being changed. Thus, XSI allows every change within the rendered region—including raytracing, global render settings such as sampling or diagnostics, and even indirect illumination features—to be updated, all interactively and while you view the rendered result. On the other hand, Maya's Interactive Photorealistic Rendering (IPR) view and 3ds Max Active Shade view both have limited abilities to display updates while you tweak mental ray–specific features; for example, neither software package supports displaying render settings or raytracing features, such as reflections, refractions, ambient occlusion, and indirect illumination simulations, among other mental ray–specific features.

Some mental ray functionalities and shaders exist only in one package or are better implemented in one package than in another. Examples include Maya's unique phenomenon shaders and comprehensive export settings (for .mi files), XSI's shader wizard that supports implementing new mental ray shaders, and 3ds Max's great implementation for mental ray's *multipass* rendering feature, which deals with saving separate mental ray sampling files and then merging them together (see Chapter 3).

The complex integration between host applications and mental ray is accessed through the mental ray application programming interface (API). This not only enables software developers to access all of mental ray's features; it also enhances their ability to further customize tools that integrate mental ray with their own application. This way, they provide a graphical user interface based on the different mental ray functions (modules) with support for controlling mental ray settings from within their host application (XSI, Maya, 3ds Max). Maya and XSI also provide their own mental ray stand-alone versions with their packages. Each package provides extended access to the host's customized integration and any plug-in tools they may have customized, such as shader libraries.

Typically, when you render a scene using mental ray, a translator program (discussed earlier under "Host Translators"), is used to translate the host application's scene description into the mental images scene description for rendering. Each package has its own scene description language as its backbone, which means that scene files from Maya, XSI, and 3ds Max are all translated into equivalent mental images (ASCII or binary) files, which can then be saved as an .mi file (mental images file) and rendered with the stand-alone mental ray renderer. Alternatively, host applications can execute renders directly from within their applications through the use of the mental ray API and the translator application.

Exporting an .mi File

The following steps can be taken to export an .mi file from host applications; note that they all offer similar settings, with Maya offering the most-robust settings for customizing the translator export. Figures 1.6, 1.7, and 1.8 show the different export settings from within each application.

MAYA

1. Navigate from the top main menu to File → Export All (or Export Selection if you just want to export a specific element).
2. From within the Export All Options window, select mentalRay from the File Type drop-down menu, as shown in Figure 1.6.
3. Notice that when the Export Selected Items Only check box (not seen in the figure) is checked, several additional settings appear, which enable controlling and customizing the translation process from Maya to an .mi file. Maya offers options to export specific features such as geometry that can then be externally linked into an .mi file (on-demand geometry) during rendering, or exporting mental ray shading networks as *Phenomenon* shaders (see the section "mental ray Shaders and Shader Libraries" later in this chapter).
4. After selecting the features you would like to export, you simply execute the command by clicking the Apply button from the lower portion of the window.

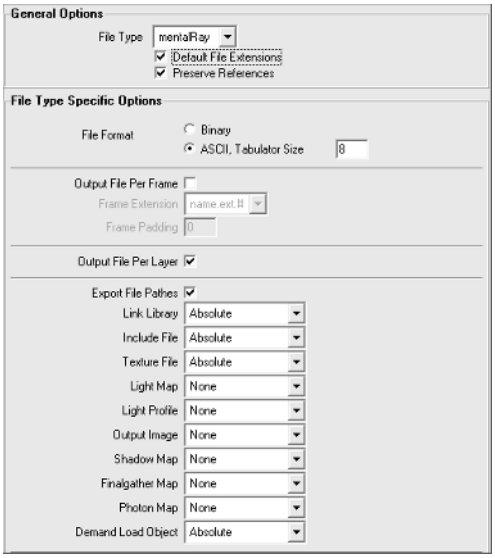


Figure 1.6

The Maya-to-mental-ray export dialog window (partial display). Here you control how Maya converts to mental ray .mi file formats.

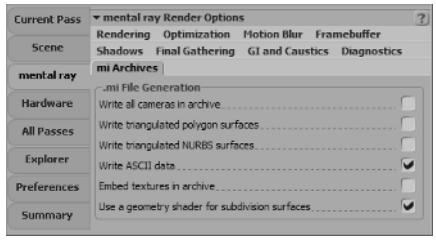


Figure 1.7

The XSI-to-mental-ray export options found under the mental ray Render Options → mi Archives for a given pass, or globally for a scene.

XSI

1. Navigate from the top main menu or under the Render toolbar (on the left side) to Render → Render Manager.
2. From the mental ray tab (or the Current Pass tab) reveal the mental ray Render Options → mi Archives tab. As with all host applications, a number of features appear for controlling how these settings are exported, as seen in Figure 1.7. Notice that with XSI, you have an option to override the scene globals using the equivalent Current Pass options, as discussed under "The Render Manger in XSI 6.0" sidebar.
3. In the Render Manager window under Current Pass → Pass Output tab → Archive, the Scene Archiving property (when enabled) exports frames incrementally within an .mi file, a topic further discussed in the following "Incremental Frames" section.
4. After specifying the required settings, navigate from the main menu bar or the Render toolbar to Export → Current Pass (or any of the other options) to initiate the export. The files will be saved into the current projects "Render_Pictures" directory.

3DS MAX

1. From the main toolbar, select the Rendering → Render… (F8) window, which opens the Render Scene: mental ray Renderer window.

2. In the Render Scene: mental ray Renderer window, reveal the Processing tab → Translator Options rollout options. Within the Translator Options is an Export to .mi File section with some relevant settings, as seen in Figure 1.8.
3. When the Export on Render check box is enabled, an .mi file rather than an image is created when you render. After, after enabling this check box, click Render (as you would for initiating any render) to export the .mi file. The file will appear in the Render Output folder in your 3ds Max directory.

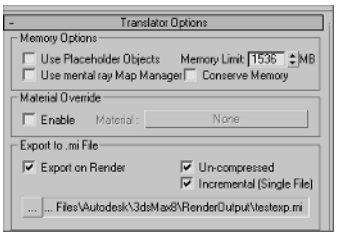


Figure 1.8
The 3ds Max Translator Options rollout for exporting .mi files from within 3ds Max

Commonalties between All Hosts

As with most mental ray features, there are several commonalities between these applications, and in this case they specify formatting options for exporting .mi files. All three applications derive most render options from their render settings as defined by the user. With Maya and 3ds Max, these would be the render settings you specify within the mental ray–specific render settings in each application and the Common render settings tab.

Some typical settings that are derived from the render settings into the exported .mi files relate to sequence frame length for animations as well as quality control settings. With Maya and XSI, these settings are also based on the current render pass (XSI) or the current render layer (Maya).

INCREMENTAL FRAMES

mental ray 3.*x* and up integrated a new approach with respect to describing animation within .mi files. This new approach refers to defining only changes that occur from one frame to the next, hence describing an *incremental* change. With this approach, rather than geometry or any other feature being described on a per-frame basis, only changes from the preceding frame need to be provided. Note that each package provides an option to export mental ray both on per-file basis (a file per frame) and as a single file that incorporates this new incremental approach. In 3ds Max, there is an Incremental (Single File) attribute, within XSI there is the same Scene Archiving property, and Maya outputs incrementally by excluding the Output File Per Frame option. Hence all three packages provide the same function in similar ways, and all describe incremental frames using the same mental images scene description structure.

> To see an example of this, export a sequence of about 10 frames from your application of choice and then examine the .mi file within a text editor. Try different settings from the export options and note how they influence the export.

The benefit of using incremental frames lends to optimizing rendering performance, as well as making .mi files more readable, by reducing the amount of clutter found within repeatedly redefined frames. For example, consider a camera traveling through an

environment that requires only the camera's spatial position for each frame to change, in which case only the camera would appear in the incremental statements within the .mi file.

ASCII VS. BINARY EXPORT

Each host application also offers ASCII or binary export options for .mi files. Thus mental ray files can either be ASCII (*plain text* files) or binary files. ASCII-encoded files are human-readable text files, and each character (of any type) is represented by 1 byte. Thus, with ASCII-encoded files, there is a one-to-one mapping between characters and bytes. Binary files support compression, particularly of vector data, in a form that provides more characters to be represented by fewer bytes; hence the file can be smaller and not as user-friendly. Typically for editing .mi files, you would take advantage of the ASCII export features. Additional export features from these host applications relate to geometry tessellation, file linking, and declarations, among other features, which change how the host applications export .mi files.

mental ray Components and Application Files

As integrated into host software, mental ray consists of three main components: application files, *shader libraries,* and *shader declaration* files. These files are always stored within the root directory of each application. They include most of the base files that ship with mental ray, as well as additional files provided by the different OEM partners.

The additional files that each application provides are primarily the host's custom shader libraries, which describe host specific shaders found within that application. With XSI, as it is solely based on mental ray, these are actually custom shaders developed specifically for rendering with mental ray. In general, shader libraries provide for three main functions: converting application-based shaders to mental ray, integrating new custom shaders within the application, and loading the mental ray base shader libraries into the application. (You'll learn more about shaders in mental ray later in this chapter and in Chapters 10 and 11.) In Maya or 3ds Max, for example, their software native (not mental ray–specific) shaders can be found within the extended mental ray shader libraries (`mayabase.mi` and `3dsmax8.mi`, respectively) and are used to translate already existing "native" shading models into models that mental ray can support and render, hence integrating these shaders with mental ray rendering. The Paint phenomenon shader library, however, is a collection of mental ray–specific custom shaders that deal specifically with vehicle shading and have been integrated into Maya, providing new mental ray custom shaders that are not part of the mental ray base shader libraries or XSI, and 3ds Max, native (host specific) shaders.

Tables 1.1–1.5 summarize the different types of files typically included with host applications, and Table 1.6 shows their directory locations for mental ray source files.

Table 1.1
Base Application Files Typically Included with Maya and XSI

APPLICATIONS	DEFINITION
ray	mental ray renderer (the mental ray executable file is labeled differently in host specific stand-alone versions.)
imf_disp	Image display utility. Type **imf_disp** into a command line console window to open this utility. Note that you may need to specify the utility's directory location; you can then use it to view mental ray–supported image formats.
imf_copy	Image copying and conversion utility. Used to convert to different image formats and to mental ray memory-mapped images (.map). Type **imf_copy** into the command line and then execute to see a list of supported flags and help.
imf_info	Provides image-related info.
imf_diff	A comparison utility for comparing images.
mkmishader	Used for writing shaders; creates C-based shader skeletons.
fg_copy	A utility that handles merging several Final Gather maps and is extremely useful at reducing Final Gather flickering. More on this in Chapter 13, "Final Gathering and Ambient Occlusion."

Note that XSI and Maya use different versions of the ray *render command with their respective stand-alone versions. XSI uses the* ray3 *command and Maya uses the* mentalrayrender *command.*

Table 1.2
Shader Declaration and Shader Library Files Typically Included with Host Applications

DECLARATION FILE	SHADER LIBRARY
base.mi	base.dll or base.so
physics.mi	physics.dll or physics.so
contour.mi	contour.dll or contour.so
subsurface.mi	Subsurface.dll or subsurface.so
architectural.mi	architectural.dll or architectural.so
paint.mi	paint.dll or paint.so

With respect to the OS platform, shaders are Dynamic Shared Object (DSO) files on Unix-based systems and Dynamic Link Libraries (DLLs) on Windows-based systems.

Table 1.3
Custom Shader Declaration and Library Files Included with Maya

DECLARATION FILE	SHADER LIBRARY
mayabase.mi	mayabase.dll
mayahair.mi	mayahair.dll
surfaceSampler.mi	surfaceSampler.dll

Table 1.4
Custom Shader Declaration and Library Files Included with XSI

DECLARATION FILE	SHADER LIBRARY
sibase.mi	Sibase.dll
motionblur.mi	motionblur.dll
softimage.mi	Softimage.dll
softimage.mi2	Softimage.dll
legacy.mi	Legacy.dll

In XSI an some base and custom shaders exist, which are not ecessarily exposed in the UI. You can find the SPDL files used to declare these (unexposed) shaders within the different directories found in the installation path under; Softimage\XSI_6.0\Application\phenolib\spdl, for example, the mibase folder. The SPDL files can then be used to install unexposed shaders using the Plug-in Manager window, a topic further discussed in Chapter 10.

Table 1.5
Custom Shader Declaration and Library Files Included with 3ds Max

DECLARATION FILE	SHADER LIBRARY
3dsmax8.mi	3dsmax8.dll
3dsmaxhair.mi	3dsmaxhair.dll
physics_phen.mi	Physics_phen.dll
lume.mi2	lume.dll

Table 1.6
Directory Locations for mental ray Source Files

APPLICATION	WINDOWS DIRECTORY
Maya	C:\Program Files\Alias\Maya7.0\mentalray
XSI	C:\Softimage\XSI_5.0\Application\rsrc
3ds Max	C:\Program Files\Autodesk\3ds Max 9\mentalray\shaders_standard

Typically, an include folder contains all the shader declaration files and an additional lib folder includes the `.dll` *or* `.so` *shader libraries. Also note that C:\ simply represents the root drive; your actual drive may be different.*

The mental ray Initialization File (.rayrc)

Each application uses an additional file, named `rayrc` (or some variant), that defines and links mental ray shader libraries and also sets mental ray environment variables. The rayrc file is essential for mental ray's integration with these applications. It can be found within the same mental ray directories that include shader declaration files and libraries:

APPLICATION	RAYRC DIRECTORY	RAYRC FILE
Maya	Alias\Maya7.0\mentalray\	maya.rayrc
XSI	Softimage\XSI_5.0\Application\rsrc\	ray3rc
3ds Max	Autodesk\3ds Max 9\mentalray\	rayrc

The rayrc file is loaded when your application loads. It provides your application and mental ray with shader declarations and links to shader libraries so that you may render mental ray shaders from within these applications. Any additional shader or shader library that you would like to add to your application must first be declared and linked through this rayrc file. The following is a portion of the maya.rayrc file that deals with linking shader libraries and shader declaration files when Maya is started:

```
# Copyright 1986-2003 by mental images GmbH & Co.KG, Fasanenstr. 81,
D-10623
# Berlin, Germany. All rights reserved.

registry "{MRMAYA_START}"
     link      "{MAYABASE}/lib/base.{DSO}"
     link     "{MAYABASE}/lib/physics.{DSO}"
     link     "{MAYABASE}/lib/mayabase.{DSO}"
     link     "{MAYABASE}/lib/contour.{DSO}"
     link     "{MAYABASE}/lib/subsurface.{DSO}"
     link     "{MAYABASE}/lib/paint.{DSO}"
```

```
    link     "{MAYABASE}/lib/mi_openexr.{DSO}"
    link     "{MAYABASE}/lib/mayahair.{DSO}"
    mi     "{MAYABASE}/include/mayabase.mi"
    mi      "{MAYABASE}/include/base.mi"
    mi      "{MAYABASE}/include/physics.mi"
    mi      "{MAYABASE}/include/contour.mi"
    mi      "{MAYABASE}/include/subsurface.mi"
    mi      "{MAYABASE}/include/paint.mi"
    mi      "{MAYABASE}/include/mayahair.mi"
    echo     "mental ray for Maya - startup done"
end registry

$lookup     "{MRMAYA_START}"
```

Each .mi file listed within the rayrc file consists of shader declarations in plain text, using the mental images scene description language. The link statements are used to connect these declarations with the compiled shaders from their respective shader libraries. You can learn more about this integration in the section "mental ray Shaders and Shader Libraries" later in this chapter.

> With Maya 8, Maya 8 and 3ds Max 9 it will suffice to place new shader DLLs and .mi declaration files within the correct directories. There is no longer a need to add them to the rayrc file because those directoriesare searched for any available shader libraries and appended automatically. In Maya use the mentalray\include and lib folders, and with 3ds Max use the shaders_autoload\include and shaders folders. With XSI shaders are typically installed using .xsiaddon files that are unpacked into the user add-on paths. The topic of custom shaders and installation is discussed in detail in Chapter 11.

Command-Line Rendering and the Stand-Alone Renderer

Another powerful feature mental ray offers is a stand-alone renderer. In general, large productions or smaller high-end specialist production houses can get more from mental ray by using the stand-alone renderer for troubleshooting. They use it primarily for taking advantage of additional mental ray features that are not fully incorporated in host software packages and instead of, or as a means to develop render farms without a need to install host applications on render nodes (making it more cost-effective). mental ray has several features for fine-tuning renders and dealing with problems such as memory or flickering that can be improved while using the full power of mental ray, the stand-alone version. With the stand-alone renderer, most mental ray rendering options can be enabled, changed, or disabled directly by using override command-line commands while executing renders. For example, suppose you are rendering a Maya scene with the mental ray stand-alone

renderer. After the host scene has been converted to an .mi file, you recognize that you need higher sampling values or more raytracing rays. You can just type in the appropriate flags and resend the render; you won't have to load the Maya UI, apply the changes, then resave and render the scene. In general for shader writers and technical directors, it is easier to write custom code that controls mental ray than to develop a plug-in for a host. For some, creating custom tools for rendering specific tasks may be easier with the stand-alone version rather than a host application.

Currently, customized stand-alone packages are provided for Maya and XSI. 3ds Max requires you use the mental images stand-alone package.

Using the Host Application's Command Line

We have already covered the two main options for rendering with mental ray, one from within the host application and the other externally with the stand-alone renderer, in previous sections in this chapter. The stand-alone renderer obviously requires an .mi file and cannot render the host's native file format without translation. Another option for rendering is using the host application's command-line utilities. This means you can still use command-line rendering, but through the host's application and not with the mental ray stand-alone renderer. This sort of command-line rendering does not support using an .mi file, since it works exactly the same as within the host application. Thus, a Maya, XSI, or 3ds Max command-line render will use either an .mb or .ma (Maya), .scn (XSI), or .max (3ds Max) binary or ASCII file as a source file. Thus, rendering through these host command-line utilities, still requires that the host application utilize its translator to provide a mental images renderable file.

Some advantages of using command-line rendering are reducing memory the full host application normally requires when the UI is enabled, specifying batch render scripts that perform several render operations consecutively, and quickly specifying different render setting overrides. With respect to command-line overrides, you can use these overrides (*flags*) only within the limits of the host's supported flags. With Maya and 3ds Max, the available mental ray flags are very limited; hence their command-line utilities don't support the entire range of flags that exist with the stand-alone renderer. Each application provides an extensive set of flags for its native renderer, including common settings such as frame range, resolution, aspect ratios, and so forth, as well as some extended flags specifically for mental ray. XSI, as mental ray is its native renderer, supports the widest range of mental ray–specific command-line flags. Another point for consideration is that some shops develop in-house tools for their pipeline using Java, Perl, Python, or another programming language; these tools can then automatically construct and execute command-line (or shell) renders on a network by piping code for execution, provided the command-line utility they access supports the settings they wish to override.

In general, it is always better to render from a command-line utility rather than from within a package. Aside from reducing the amount of memory used on your machine, you gain the ability to list several renders within a command-line render script file known as a *batch render script*, discussed in the section "Batch Rendering." Command-line rendering can be executed directly in a command prompt window in Windows (or a shell in Unix-based systems) by typing the commands discussed in the next sections for each host application. Note that the path to the render utility with XSI and 3ds Max must be specified as part of the syntax; alternatively, if you navigate to that directory in advance, you can then execute the render without specifying the path. Let's look at an example of a command-line render using each host.

Command-Line Render Execution

On Windows systems, to open the command prompt, simply choose Run from the Start menu. You are then prompted for a program to execute. Type **`cmd`** into the text line and click OK to execute. The command-prompt window will open, typically in your default directory. With XSI and 3ds Max, you then need to either navigate to the directory where the render utility is located or specify that directory with the `render` command (see examples later). In addition, with all hosts you must specify the directory for the scene file that you wish to render or navigate to that directory and execute the command there, which then does not require you to specify a full path.

Note that you can change a directory in the command prompt by copying and pasting the directory using the `chdir` command-prompt command. For example, for the 3ds Max directory, enter the following:

```
chdir "C:\Program Files\Autodesk\3ds Max 9\"
```

Once the directory is set, in most cases you can then specify relative paths for the scene and image files from the current location. You will see the `render` command in the XSI and 3ds Max batch render script examples on the book CD; the following sections show how to use it with each host.

MAYA

With Maya, the command can be entered in any command prompt directory without specifying a path for the Maya render utility; however, you must specify a path for the scene file or navigate to that directory in advance. The following line can then be used to execute a render:

```
render -r mr -v 5 -s 1 -e 10 -b 1 "…path\fileName.mb"
```

This line, read from left to right, has the following meaning: `render` starts a Maya render, and the `-r mr` flag specifies that mental ray should be used (`mr` for mental ray, `sw` for software, etc.). Verbosity is specified with the `-v 5` flag, equivalent to level-5 progress messages. `-s 1`, `-e 10`, and `-b 1` specify start frame, end frame, and step increment frame, respectively. The path and filename are indicated at the end. If you type `render -r mr`

`-help` the -help flag provides a list of mental ray command line options you can review. Later, we'll look at using this command within a batch script.

XSI

With XSI, the command can be entered in any command prompt directory as long as you specify the full path, as in this example:

```
C:\Softimage\XSI_6.0\Application\bin\xsi -r
```

You can also use just the `xsi` command if you navigate to that directory before executing the command, or alternatively you can use the XSI-specific command prompt. You can find the XSI command-prompt under the Softimage program folder through your Start menu. In any case, you must specify a path for the scene file or navigate to that directory in advance. The following line can then be used to execute a render:

```
xsi -r -s 1,5,1 -verbose "prog" -scene "…path\fileName.scn"
```

This line reads from left to right as follows: `xsi` means start an XSI render, and the `-r` flag specifies rendering. Note that if you just type `xsi` and execute, the XSI application will launch. The `-s` flag is an abbreviated flag specifying the start, end, and step frames with comma-separated values. Verbosity `-verbose "prog"` specifies an output of level-5 progress messages, and the filename is indicated at the end. In general, the filename should be specified with a full or relative path, especially when using a script. If you type `xsi -r -h` the -h (help) flag provides a list of command line options you can review.

3DS MAX

With 3ds Max, the command entered in the command prompt must specify the full path to the render command; for example, on most Windows machines the path would be as follows:

```
C:\Program Files\Autodesk\3ds Max 9\3dsmaxcmd -?
```

The -? flag is a help flag that will list all the options for command line rendering. You can also use just the `3dsmaxcmd` command if you navigate to that directory before executing the command. You can find that command in the the root 3dsMax directory. The following line is then used to execute a render from the 3ds Max 9 directory:

```
3dsmaxcmd -frames:0-10 -v:5 "scenes\filename.max"
```

This line reads from left to right as follows: `3dsmaxcmd` means start a 3ds Max render. The `-sframes:0-10` flag specifies the render frame range. Verbosity `-v:5` specifies an output of level-5 progress messages, and the filename is indicated at the end. In general, the filename should be specified with a full or relative path, especially when using a script.

Batch Rendering

You can open a simple text file and list several lines (as seen in the following code), which will then enable batch rendering. As one render job completes, the next one can be executed. This sort of render list can be saved as an executable file. Within such a render script file,

you may specify mental ray stand-alone renders or host command-line utility rendering. The following examples demonstrate an XSI command-line render script that renders separate files with different frames for each file.

Note that with XSI, you can only batch render continuously when using the XSI Batch render utility. I omitted the full paths, hence the three dots in the path directory. Note that with XSI (and 3ds Max), I first indicate a change directory (`chdir`) command so that the command prompt initiates from the correct directory, where the `xsibatch.exe` utility exists; without that, it will not find the utility:

```
chdir "C:\Softimage\XSI_6.0\Application\bin\"
xsibatch -r -s 1,10,1 -scene C:\...\fileName1.scn
xsibatch -r -s 10,20,1 -scene C:\...\fileName2.scn
xsibatch -r -s 20,30,1 -scene C:\...\fileName3.scn
pause
```

The `pause` command is another command-prompt command that keeps the command prompt open after rendering has completed so you may review render statistics; otherwise, once the render completes, the command prompt closes automatically. If you specify within the host to save verbosity output to a file, as demonstrated earlier under "Enabling Message Logging and Verbosity Levels", then you don't really need to use the pause command

If you type such a script into a simple text document, such as a Notepad document in Windows, you can save the script as a BAT (`.bat`) executable file simply by typing the name in quotes when prompted to save, as seen in Figure 1.9. For now, whether you are using Maya, XSI, or 3ds Max, I have provided batch-render scripts (for Windows systems) for each application in the Chapter 1 directory on the companion CD. Open these files and examine them; they should help you quickly and easily set up your own batch-render scripts. Note that you must adjust directories and filenames to match your system and files.

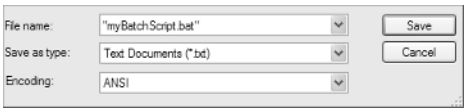


Figure 1.9

Saving this plain text file as a BAT file creates an executable file that can be used to submit a render list to either a host command-line render utility or the stand-alone renderer.

If you're using a Unix-based system (OSX or Linux), you can use the same syntax in a standard text file, but you must convert the file into an executable file through the terminal by executing the `chmod a+x` command.

mental ray Shaders and Shader Libraries

Shaders are the fundamental building blocks of rendering software. As you probably know from other applications, a shader is a program that determines the surface characteristics of an object in a 3D drawing. But mental ray shaders are far more than the typical surface shaders we commonly think of. There are material shaders, light shaders, geometric shaders, texture shaders, camera lenses, and more. You will learn a great deal about mental ray and its shader capabilities throughout this book. The mental ray shader libraries include an extensive collection of base and custom shaders. They include common shaders, like the familiar Blinn, Phong, Lambertian, and Anisotropic shaders, as well as the typical texture-placement shaders and light shaders that are commonly found within 3D packages.

Keeping these functions in external libraries enables software developers and mental ray users to easily integrate new custom shaders and shader libraries.

Shader libraries are collections of C- or C++-based shaders that have been compiled for mental ray and can be described as plug-in programs for mental ray. These libraries may include a cluster of shaders that handle numerous specific tasks. The declaration files describe these shaders and their options using the mental images scene description language. In essence, to use a shader you must effectively declare it within the mental ray .mi file. The mental images declaration files (.mi) essentially transfer shader information from these declaration files into the mental ray file. Once declared, they know how to interact with their counterparts from within the shader libraries. So you may think of these declaration files as your interface into the shader libraries.

Most mental ray shaders perform very specific functions. This approach lends itself to modularization and custom shader development. Because each shader is designed to handle a very particular task, shaders are not interdependent and can be used in various ways. For example, if you use a base *illumination* shader such as a Blinn, you can then connect it to a *sample compositing* reflection shader that provides reflections. Because the mental ray Blinn shader does not include a reflection shader, it can take advantage of a new and improved reflection, as you will see in Chapter 11. In contrast, if you used a host's Blinn shader, a reflection shader would already be part of its functionality and you could use only that built-in feature. Stripping down shaders to their base functionalities gives the developer more control in creating complex custom effects and reduces unnecessary duplication. In our example, the developer would need to write only the new reflection shader rather than a whole new Blinn shader, which obviously requires more work and provides less flexibility. The key concept behind this modular approach is to enable a flexible procedural approach for designing custom effects, combining multiple shaders in a way that provides for more-complex effects and more flexibility as well as a speedier development process.

Shaders provided by the 3D host applications are usually far more robust than a simple shader. Hence, some of these applications have already provided some sort of procedural shader tree based on several "simple" mental ray shaders. These shader trees are hidden from the user and typically combined by using mental ray's Phenomenon technology.

Phenomena

Shaders can be combined and interact with any other type of shader. For example, you may use a geometric shader to define a volume in the scene and then, using a variety of other shaders, apply a complex volumetric effect, or you could use several shaders to create a complex surface-shading effect such as subsurface scattering for skin. These shader graphs may be combined using several base mental ray shaders from the mental ray libraries (`base.mi`, `physics.mi`, `subsurface.mi`) that may include illumination shaders, light maps, sample compositing shaders, photonic shaders, environmental shaders, and

essentially any type of shader. The process of compiling individual shaders into one of these compounds may be tedious and redundant. To spare you this effort, mental ray allows you to create a Phenomenon shader.

Phenomena are shader trees compiled from several other shaders, forming a complex effect. Once you have named and exported the Phenomenon, this new shader can be linked through the rayrc file (see " The mental ray Initialization File (.rayrc)" earlier in this chapter). The shader will then become available as a single node within each application. The entire shader tree remains hidden from the user so that the user has access to a single shader interface that essentially controls several embedded shaders from within the Phenomenon shader tree. The developer can create an interface for the new Phenomenon shader either within these applications or by editing the .mi file. Essentially, the creator decides what settings should become available from within the tree and manually links them to the shader interface.

Note that a phenomenon shader does not require a compiled shader library, as it is based on existing shaders. Thus only a declaration file is required that provides access to the shaders settings as well as the shader libraries that were used to derive this new shader. In production this tool can become very useful to streamline redundant shader trees and simplify the general process.

Indirect Illumination

mental ray is packed with tools to simulate the realistic interaction of indirect light with surfaces. As light reflects, it hits and "bounces" from one surface to the next. This explains why, although you most likely wouldn't have a light fixture under your desk, light that hits the floor would bounce and illuminat the entire region beneath the desk. This light interaction carries light energy and color from one surface to the next—a phenomenon known as *color bleeding*. To generate indirect illumination, mental ray uses a *Photon Map* that describes the contribution of indirect light on surfaces. Raytracing, in this case, is the process of emitting light photons from a light source and tracking its behavior throughout the scene. This sort of indirect illumination is used with the following mental ray features:

Global illumination is mental ray's primary indirect lighting feature; it calculates the indirect light bounce of diffused light. Diffused light in CG refers to the diffused color contribution from material shaders.

Caustic light, a subset of global illumination, represents the light behavior for surface reflections and refractions. As light reflects or refracts through surfaces, it typically magnifies in intensity and appears to focus or spread out based on surface characteristics. By using global illumination and caustics, you can simulate a wide range of light characteristics, which include diffuse, glossy, and specular light reflections. These light characteristics are discussed in detail in Chapter 10 "The Fundamentals of Light and Shading Models."

Participating Media refers to light scattering from particles suspended in air. This term is used to describe particles that participate in the illumination within a defined region. This is yet another powerful feature mental ray offers to simulate non-geometric effects that influence lighting. Typically, suspended particles of dust or smoke contribute by reflecting and absorbing light within a scene. This also has an effect on shadowing and direct lighting in the scene, as would any geometry that blocks or reflects light. You'll learn about participating media in Chapter 12, "Indirect Illumination."

Sub-surface scattering refers to the transmission of photons through translucent surfaces. Typically, this refers to skin, jade, wax, plastic, and several other types of surface where scattered light within the surface may become visible. mental ray enables calculating this sort of effect with a physical shader from the `subsurface.mi` shader library, which scatters photons within a surface. A second approach to simulating sub-surface scattering does not use photons to calculate sub-surface scattering but instead utilizes a complex shader that simulates the light influence across a surface based on *lightmaps* as well as certain parameters defined within the shader.

Final Gather is an additional feature that calculates indirect illumination, but unlike those just listed, it does not use photons to calculate its effect. The name refers to the "final gathering" of light influence in a scene. Final Gather is evaluated after global illumination (if enabled) has been calculated and before the render commences. This feature is based on casting rays into the scene from a hemispherical point on a surface and evaluating the total influence of light on that point, from the surrounding objects. This enables you to simulate the effect of light being occluded between surfaces in close proximity as well as simulate the influence from different light intensities derived from a high dynamic range (HDR) image.

Chapters 10 through 15 demonstrate and explain all of these features in great detail. You will first learn how light interacts in real life and then how that is translated and re-created using these indirect lighting algorithms. A solid understanding of light and surface behaviors will enhance your ability to control and predict the results of such simulations.