Learning to write mental ray shaders

Andy Kopra
Senior Technical Writer
mental images, Inc.
Learning to write mental ray shaders

1. Shaders and the structure of mental ray
2. Strategy of the new shader book
3. Cross-referencing in the shader book
4. Website support for the book’s software
5. The CDROM included with the book
Shaders and the structure of mental ray
Shaders and the structure of mental ray

The mental ray renderer is a software library that is embedded in different interface applications.
mental ray
application

mental ray
Shaders and the structure of mental ray

Scene data in mental ray is implemented as a database.
mental ray

scene processing and rendering

scene database

shaders
Shaders and the structure of mental ray

Shaders are represented as elements in the scene database that can modify the behavior of many phases of the rendering pipeline.
scene database

- options
  - state
  - contour store
  - contour contrast
  - inheritance

- camera
  - output
  - volume
  - environment
  - lens
  - contour output

- light
  - light
  - emitter

- texture
  - light
  - texture

- material
  - material
  - displace
  - shadow
  - volume
  - environment
  - contour
  - photon
  - photonvol
  - lightmap

- instance
  - geometry
1. Create frame buffers in memory

options "opt"
  object space
  contrast .1 .1 .1 .1
  samples -1 2
  framebuffer 0 "rgba"
  framebuffer 1 "rgba"
end options

2. Get and put frame buffer pixels

```c
miBoolean shader( ... )
{
  miColor a, b, c, d;
  ...
  mi_fb_put(state, 0, &a);
  mi_fb_get(state, 0, &b);
  mi_fb_put(state, 1, &c);
  mi_fb_get(state, 1, &d);
  ...
}
```

3. Write frame buffers to file

```c
camera "cam"
  output "fb0" "tif"
    "buffer_0.tif"
  output "fb1" "tif"
    "buffer_1.tif"
  output "rgba" "tif"
    "standard_rgba.tif"
  ...
end camera
```
Strategy of the new shader book
Strategy of the new shader book

Beginning shader programmers needed a tutorial to complement the mental ray reference handbooks.
Programming mental ray
Third, completely revised edition

SpringerWienNewYork
Strategy of the new shader book

The order of shader presentation in the new book is based on how we see, not on how the underlying software is organized.
retinal image
experience with light

retinal image
isolation of object

identification of shape

experience with light

retinal image
understanding of scene

isolation of object

identification of shape

experience with light

retinal image
understanding of scene

isolation of object

identification of shape

experience with light

retinal image

color
understanding of scene

isolation of object

identification of shape

experience with light

retinal image

shape

light

color
understanding of scene

isolation of object

identification of shape

experience with light

retinal image

image

space

shape

light

color

color blocking
Perception

- understanding of scene
- isolation of object
- identification of shape
- experience with light
- retinal image

- image
- space
- shape
- light
- color

- varnish
- background relationships
- surface detail
- shading from light
- color blocking
Perception

- understanding of scene
  - isolation of object
    - identification of shape
      - experience with light
        - retinal image

Shaders

- image
  - space
    - shape
      - light
        - color

Painting

- varnish
  - background relationships
    - surface detail
      - shading from light
        - color blocking
Shaders
Part 2: Color

5. A single color
   5.1 The simplest shader model .......................... 49
   5.2 The shader function in C ................................ 50
   5.3 The shader source file .................................. 51
   5.4 Using a shader in a material ........................... 54
   5.5 Shader programming style ............................... 54
   5.6 Basic issues in writing shaders ......................... 55

6. Color from orientation
   6.1 The representation of rendering state: miState ....... 57
   6.2 A shader based on surface orientation ................. 58
   6.3 Passing parameters to a shader .......................... 59
   6.4 Other fields in miState ..................................... 59
   6.5 Calculation of surface orientation ....................... 61
   6.6 Shaders as analysis tools ................................. 61

7. Color from position
   7.1 Intensity based on distance ................................ 63
   7.2 Interpolating between colors based on distance ........ 64
   7.3 Clarifying the shader with auxiliary functions ......... 66

8. The transparency of a surface
   8.1 Using the API library to trace a ray ................. 69
   8.2 Functions as a description of a process .............. 73

9. Color from functions
   9.1 The texture coordinate system ......................... 78
   9.2 Quantizing the texture coordinates .................... 80
   9.3 Using the texture coordinates for texture mapping ... 82
   9.4 Manipulating the texture coordinates ................. 84
   9.5 Defining texture maps that tile well ................. 86
   9.6 Multiple texture spaces ............................... 87
10. The color of edges

10.1 The four contour shader types ............................................. 94
10.2 Location of the contour shaders in the scene file ............................................ 98
10.3 The Store shader: defining and saving contour data ............................................ 96
10.4 The Contrast shader: determining the location of contours ............................................ 98
10.5 The Contour shader: defining the properties of contours ............................................ 100
10.6 The Output shader: writing the contour image ............................................ 102
10.7 Using the four contour shaders ............................................ 104
10.8 Compositing contours with the color from the material shader ............................................ 104
10.9 Displaying tessellation with contour shaders ............................................ 106
10.9.1 The barycentric coordinates of a triangle ............................................ 109
10.10 Including color in the determination of contour lines ............................................ 112
10.10.1 Contour lines at edges ............................................ 112
10.10.2 Contour lines at color boundaries ............................................ 113
10.10.3 Converting illumination shading into regions of single color ............................................ 118

Part 3: Light

11. Lights ............................................ 125

11.1 Light from a single point ............................................ 127
11.2 A point light with shadows ............................................ 130
11.3 A simple spotlight ............................................ 131
11.3.1 Acquiring light parameter values from the scene database ............................................ 134
11.3.2 Using light parameter values in the shader ............................................ 137
11.4 A spotlight with a soft edge ............................................ 139
11.5 A spotlight with a better soft edge ............................................ 141
11.6 Soft shadows using area lights ............................................ 143
11.6.1 Area light primitives ............................................ 145
11.6.2 Using area lights in spotlights ............................................ 146
11.7 Modifying light intensity based on distance ............................................ 146
11.8 Defining good default values for parameters ............................................ 148

12. Light on a surface ............................................ 149

12.1 Diffuse reflection: Lambert shading ............................................ 149
12.2 Specular reflections ............................................ 154
12.2.1 The Phong specular model ............................................ 154
12.2.2 The Blinn specular model ............................................ 156
12.2.3 The Cook-Torrance specular model ............................................ 158
| 12.2.4 The Ward specular model | 160 |
| 12.3 Faking the specular component | 162 |
| 12.4 Adding arbitrary effects to the Lambert model | 164 |

| 13. Shadows | 167 |
| 13.1 Color shadows without full transparency | 168 |
| 13.2 Midrange transparency control | 171 |
| 13.3 Other parameters for the breakpoint function | 174 |
| 13.4 A shadow shader with continuous transparency change | 176 |
| 13.5 Combining the material and shadow shader | 179 |

| 14. Reflection | 183 |
| 14.1 Specular reflections | 183 |
| 14.2 Reflections and trace depth | 185 |
| 14.3 Glossy reflections | 187 |
| 14.4 Glossy reflection with multiple samples in the shader | 189 |
| 14.5 Glossy reflection with varying sample counts for reflections | 192 |
| 14.6 Glossy reflections and object geometry | 195 |

| 15. Refraction | 197 |
| 15.1 Specular refraction of non-intersecting objects | 198 |
| 15.2 Improving the determination of the indices of refraction | 201 |
| 15.3 Using different indices of refraction | 204 |
| 15.4 Glossy refraction | 206 |
| 15.4.1 Using a single glossy refraction ray | 206 |
| 15.4.2 Using multiple glossy refraction rays | 207 |
| 15.4.3 Controlling the number of rays per refraction | 209 |
| 15.5 Combining reflection and refraction with Phenomena | 212 |

| 16. Light from other surfaces | 219 |
| 16.1 Using the photon map for global illumination | 220 |
| 16.1.1 Adding global illumination to the Lambert shader | 221 |
| 16.1.2 The photon shader | 222 |
| 16.1.3 Global illumination statements in the options block | 224 |
| 16.2 Global illumination as the ambient parameter | 224 |
| 16.2.1 Overriding global illumination options in a shader | 226 |
| 16.3 Final gathering | 230 |
| 16.4 Caustics | 232 |
| 16.5 Visualizing the photon map | 235 |
| 16.6 Ambient occlusion | 238 |
| 16.6.1 Sampling the environment by tracing rays to detect occlusion | 239 |
Part 4: Shape

17. Modifying surface geometry
17.1 Surface position and the normal vector .................................................. 247
17.2 A simple displacement shader ................................................................ 248
17.2.1 Using texture coordinates .................................................................. 250
17.3 Shader lists and displacement mapping .................................................. 252
17.4 Using texture images to control displacement mapping ....................... 253
17.5 Combining displacement mapping with color .......................................... 255
17.6 Using noise functions with displacement mapping ............................... 257

18. Modifying surface orientation
18.1 Simulating surface orientation ................................................................. 259
18.2 Modifying the normal in a shader .......................................................... 260
18.3 Changing the normal based on texture coordinates .............................. 261
18.4 Evaluating a function in the sample neighborhood .................................. 263
18.5 Shader lists and bump mapping ............................................................. 266
18.6 Using images to control bump mapping ................................................ 266
18.7 Combining bump mapping with color ................................................... 269

19. Creating geometric objects
19.1 Creating a square polygon .................................................................... 271
19.2 Procedural use of the geometry API ....................................................... 277
19.3 Placeholders objects ............................................................................ 282
19.3.1 Creating an object from a file specified in the scene ......................... 282
19.3.2 Creating an object from a file within a shader .................................. 284
19.3.3 Creating an object instance from file within a shader ...................... 286

20. Modeling hair
20.1 Placeholder objects and callback functions ........................................... 289
20.2 Structure of the shader and its callback ................................................ 289
20.3 Basic principles of the hair geometric primitive .................................... 290
20.4 A geometry shader for a single hair ....................................................... 292
20.5 Visualizing the hair’s barycentric coordinates ...................................... 298
20.6 Higher order curves in the hair primitive ............................................. 298
20.7 Additional data attached to the hair primitive ...................................... 303
20.8 Multiple hairs ..................................................................................... 309
20.9 A basic lighting model for hair ............................................................. 315
20.10 The hair primitive as a general modeling tool .................................... 320
<table>
<thead>
<tr>
<th>Part 5: Space</th>
<th>337</th>
</tr>
</thead>
<tbody>
<tr>
<td>21. The environment of the scene</td>
<td>339</td>
</tr>
<tr>
<td>21.1 A single color for the environment</td>
<td>339</td>
</tr>
<tr>
<td>21.2 Defining a color ramp</td>
<td>340</td>
</tr>
<tr>
<td>21.3 Efficient shader access to a color ramp</td>
<td>342</td>
</tr>
<tr>
<td>21.4 Using parameter arrays for color channels in the ramp</td>
<td>345</td>
</tr>
<tr>
<td>21.4.1 Allocating memory in the init function</td>
<td>346</td>
</tr>
<tr>
<td>21.4.2 Releasing the memory allocated through the user pointer</td>
<td>349</td>
</tr>
<tr>
<td>21.4.3 Using channel, ramp in a named shader</td>
<td>350</td>
</tr>
<tr>
<td>21.5 A color array as a shader parameter</td>
<td>351</td>
</tr>
<tr>
<td>21.6 Environment shaders for cameras and objects</td>
<td>354</td>
</tr>
<tr>
<td>21.7 Encapsulating constant data in a Phenomenon</td>
<td>355</td>
</tr>
<tr>
<td>22. A visible atmosphere</td>
<td>357</td>
</tr>
<tr>
<td>22.1 Volume shaders and the camera</td>
<td>357</td>
</tr>
<tr>
<td>22.2 Changing the fog density</td>
<td>360</td>
</tr>
<tr>
<td>22.3 Adding a debugging mode to a shader</td>
<td>364</td>
</tr>
<tr>
<td>22.4 Varying the ground fog layer positions</td>
<td>367</td>
</tr>
<tr>
<td>23. Volumetric effects</td>
<td>369</td>
</tr>
<tr>
<td>23.1 Rays in volume shaders</td>
<td>369</td>
</tr>
<tr>
<td>23.2 Using a threshold value in the volume</td>
<td>371</td>
</tr>
<tr>
<td>23.3 Cumulative density for a ray in the volume</td>
<td>374</td>
</tr>
<tr>
<td>23.4 Illumination of samples in the volume</td>
<td>377</td>
</tr>
<tr>
<td>23.4.1 Fractional occlusion</td>
<td>379</td>
</tr>
<tr>
<td>23.4.2 Total light at a point in the volume</td>
<td>379</td>
</tr>
<tr>
<td>23.4.3 Accumulating and blending volume colors</td>
<td>380</td>
</tr>
<tr>
<td>23.4.4 The illuminated volume</td>
<td>381</td>
</tr>
<tr>
<td>23.5 Defining the density function with a shader</td>
<td>383</td>
</tr>
<tr>
<td>23.6 Using voxel data sets for the density function</td>
<td>389</td>
</tr>
<tr>
<td>23.7 Summary of the volume shaders</td>
<td>395</td>
</tr>
<tr>
<td>Part 6: Image</td>
<td>397</td>
</tr>
<tr>
<td>24. Changing the lens</td>
<td>399</td>
</tr>
<tr>
<td>24.1 A conceptual model of a camera</td>
<td>399</td>
</tr>
<tr>
<td>24.2 Shifting the lens position</td>
<td>401</td>
</tr>
</tbody>
</table>
24.3 Mapping the scene around the camera to a rectangle ........................................ 403
24.4 A fisheye lens ........................................................................................................ 407
24.5 Depth of field ........................................................................................................ 412
24.6 Multiple lens shaders .......................................................................................... 418

25. Rendering image components ............................................................................. 421
25.1 Definition and use of frame buffers ................................................................. 421
25.2 Separating illumination components into frame buffers ............................... 423
25.3 Compositing illumination components ............................................................... 426
25.4 Rendering shadows separately .......................................................................... 427
25.5 Saving components from standard shaders ....................................................... 430
25.6 Using a material Phenomenon with framebuffer components ..................... 434

26. Modifying the final image ..................................................................................... 439
26.1 An image processing vocabulary ....................................................................... 439
26.2 Adding a letterbox mask ..................................................................................... 440
26.3 A median filter in a shader ................................................................................ 443
26.4 Compositing text over a rendered image ........................................................ 447
26.5 Multiple output shaders ..................................................................................... 451
26.6 Late binding and beyond .................................................................................... 452

Appendices .................................................................................................................. 455

A. Rendered scene files .............................................................................................. 457
Chapter 5 — A single color ...................................................................................... 457
Chapter 6 — Color from orientation ........................................................................ 457
Chapter 7 — Color from position ............................................................................. 457
Chapter 8 — The transparency of a surface ............................................................ 458
Chapter 9 — Color from functions .......................................................................... 458
Chapter 10 — The color of edges ............................................................................ 459
Chapter 11 — Lights .................................................................................................... 460
Chapter 12 — Light on a surface .............................................................................. 461
Chapter 13 — Shadows ............................................................................................... 462
Chapter 14 — Reflection ............................................................................................ 463
Chapter 15 — Refraction ............................................................................................ 464
Chapter 16 — Light from other surfaces .................................................................. 465
Chapter 17 — Modifying surface geometry ............................................................ 466
Chapter 18 — Modifying surface orientation ......................................................... 467
Chapter 19 — Creating geometric objects .............................................................. 467
Chapter 20 — Modeling hair .................................................................................... 468
# Table of Contents

1. Introduction .......................... 1
   1.1 A visual hierarchy: color, light, shape, space, image .......................... 3
   1.2 What you need to know to write mental ray shaders .......................... 5
   1.3 How to read this book .......................... 5
   1.4 Web-based resources .......................... 7
   1.5 Acknowledgments .......................... 7

## Part 1: Structure .......................... 9

2. The structure of the scene .......................... 11
   2.1 The scene in mental ray .......................... 11
   2.2 A block of statements with a name .......................... 11
   2.3 A block that includes another block .......................... 12
   2.4 A block containing a shader .......................... 13
   2.5 A block that refers to another block .......................... 14
   2.6 Grouping the elements in the scene .......................... 16
   2.7 Scene file commands .......................... 16
   2.8 Putting the parts together .......................... 17
   2.9 A complete scene file .......................... 18
   2.10 The scene file and 3D applications .......................... 19

3. The structure of a shader .......................... 21
   3.1 Defining a color with a shader .......................... 21
   3.2 Providing input to a shader .......................... 22
   3.3 Telling the shader about its context .......................... 23
   3.4 Accumulating the effect of a series of shaders .......................... 23
   3.5 Combining the previous and current results .......................... 24

4. Shaders in the scene .......................... 25
   4.1 Describing parameters: shader declarations .......................... 25
   4.2 Missing parameter values: defining defaults .......................... 27
   4.3 Defining the material: anonymous and named shaders .......................... 29
      4.3.1 Anonymous shaders .......................... 29
      4.3.2 Named shaders .......................... 30
   4.4 Chaining shaders of the same result type: shader lists .......................... 30
      4.4.1 Using the result of the previous shader .......................... 31
      4.4.2 Modifying the rendering state in a shader list .......................... 32
Scene

- Named shader

Scene file

```
shader "shader_depth"
  "depth_fade"
  (near 10,
   far -10
  )

shader "shader_front"
  "front_bright"
  (tint 1.05 1.05 1.05
  )

material "front_depth"
  "scaled_color"
  (base_color = "shader_front"
   scale_factor = "shader_depth"
  )
end material
```
declare phenomenon

**Result type**
color "front_depth"

**Interface parameters**
color "tint",
scalar "near",
scalar "far"

**Shader definition**
shader "front"
	"front_brightness" (  
		"tint" = interface "tint"
	)

shader "depth"
	"depth_fade" (  
		"near" = interface "near",
		"far" = interface "far"
	)

shader "combine"
	"scaled_color" (  
		"base_color" = "front",
		"scale_factor" = "depth"
	)

**Root shader**
root = "combine"

done declare
miBoolean one_color ( miColor *result, miState *state, struct one_color *params )
{
    miColor *color = mi_eval_color(&params->color);
    result->r = color->r;
    result->g = color->g;
    result->b = color->b;
    result->a = color->a;
    return miTRUE;
}
Cross-referencing in the shader book
Cross-referencing in the shader book

Marginal references point to the first two books.
2.6 Grouping the elements in the scene

We've only defined a single instance, but there can be any number in the scene file. Instances can be organized into hierarchies of any complexity using instance groups. Before we begin to render the scene, we will need to specify the top-level group that will contain all the instances to be rendered. In our simple scene, we've only instance a square, so the top-level instance group will only contain the square and camera instances. Defining the instgroup block follows the common pattern: a reserved word (instgroup), the name for the block, additional information appropriate for the type of block, and the final end statement.

```
instgroup "root"
    "main-camera" "yellow-square"
end instgroup
```

2.7 Scene file commands

All the previous examples showed the various elements that can be contained in the scene file. The scene file can also contain commands that do not define elements but tell mental ray to perform an action at the point in the scene file where the command occurs.

For example, to use a shader in a scene file we need to do two things:

1. Load the compiled shader into memory when rendering begins and
2. Declare the data types of the shader and its parameters so that its use later in the scene file can be correctly parsed.

In the scene file, shader loading and declaration are usually done using the link and sinclude commands, respectively. For example, to use the one_color shader in our simple scene, these commands are included at the beginning of the file:
2.6 Grouping the elements in the scene

We've only defined a single instance, but there can be any number in the scene file. Instances can be organized into hierarchies of any complexity using instance groups. Before we begin to render the scene, we will need to specify the top-level group that will contain all the instances to be rendered. In our simple scene, we've only instantiated a square, so the top-level instance group will only contain the square and camera instances. Defining the instgroup block follows the common pattern: a reserved word (instgroup), the name for the block, additional information appropriate for the type of block, and the final end statement.

```
instgroup "root"
  "main-camera" "yellow-square"
end instgroup
```

Figure 2.12: The root instance group for scene containing the elements to be used in rendering

To construct an object hierarchy, an instgroup block can contain other instance groups as well as other objects. In modeling applications, a hierarchy is often a natural way to represent large structures, and this organization in the application can be represented in the scene database with the instgroup element.

2.7 Scene file commands

All the previous examples showed the various elements that can be contained in the scene file. The scene file can also contain commands that do not define elements but tell mental ray to perform an action at the point in the scene file where the command occurs.

For example, to use a shader in a scene file we need to do two things:

1. Load the compiled shader into memory when rendering begins and
2. Declare the data types of the shader and its parameters so that its use later in the scene file can be correctly parsed.

In the scene file, shader loading and declaration are usually done using the link and include commands, respectively. For example, to use the one_color shader in our simple scene, these commands are included at the beginning of the file:
## 2.7 Scene file commands

All the previous examples showed that the scene file can also contain commands that perform an action at the point in the scene where they are declared.

For example, to use a shader in a scene,

1. *Load* the compiled shader into memory.
2. *Declare* the data types of the shader so they can be correctly parsed.

In the scene file, shader loading and declaration commands, respectively. For example, shader commands are included at the beginning of the application or script.
2.7 Scene file commands

All the previous examples showed that the scene file can also contain commands to perform an action at the point in the scene.

For example, to use a shader in a scene:

1. **Load** the compiled shader into memory.

2. **Declare** the data types of the shader, so it can be correctly parsed.

In the scene file, shader loading and declaration commands, respectively. For example, these commands are included at the beginning.
2.7 Scene file commands

All the previous examples showed the anatomy of a scene file. The scene file can also contain commands that perform an action at the point in the scene where the command is placed.

For example, to use a shader in a scene, you would:

1. **Load** the compiled shader into the scene.
2. **Declare** the data types of the shader so that it can be correctly parsed.

In the scene file, shader loading and declaration commands, respectively. For example, these commands are included at the beginning of the scene file.
Cross-referencing in the shader book

An index of topics in the first two books point to references in the new book.
References to Volumes I and II

Throughout this book, marginal references to the first two volumes of mental ray documentation, Volume I: Rendering with mental ray and Volume II: Programming mental ray point to further information about the current topic. This index lists the pages in this book that discuss the topics of Volumes I and II, sorted by section title.

Volume I: Rendering with mental ray
1.2. Scenes and Animation, 11
1.2.1. Geometric Objects, 12
1.2.2. Materials, 13
1.2.3. Light Sources, 127
1.2.4. Cameras, 6
1.3. Shaders, 11, 49
1.5.1. Transparency, Refractions, and Reflections, 69
1.9. Stages of Image Generation, 222
2.1. A Simple Scene, 11, 18
2.2. Anatomy of a Scene, 12-17
3. Cameras, 11
3.1. Pinhole Cameras, 399
3.2. Image Resolution, 399
3.3. Aspect Ratio, 399
3.8. Lenses: Depth of Field, 412
4. Surface Shading, 78
4.1. Color and Illumination, 34, 149, 154, 156, 158, 160
4.2. Texture Mapping, 77
4.2.1. Texture Projections, 84, 86
4.4. Bump Mapping, 32, 33, 259
4.5. Displacement Mapping, 247
4.6. Anisotropic Shading, 160
4.9. Reflection, 183
4.10. Transparency and Refraction, 197
5. Light and Shadows, 127, 128, 130
5.1. Point, Spot, and Infinite Lights, 32, 132
5.2. Raytraced Shadows, 130
5.2.2. Soft Shadows with Area Light Sources, 140
5.2.3. Shadow Modes: Regular, Sorted, Segmented, 167
5.3. Fast Shadows with Shadow Maps, 130
6.3. Ray Marching, 360, 370
7. Caustics and Global Illumination, 57, 329
7.1. Photon Mapping vs. Final Gathering, 6, 151, 220, 230
7.2. Local Illumination vs. Caustics vs. Global Illumination, 183
7.6. Caustics, 239, 232
7.7.4. Final Gathering, 220
10. Contours, 95
10.1. Outline Contours, 95
11. Shaders and Phenomena, 21, 46
11.1. Declarations, 16, 26, 46
11.2. Definitions, 26, 46, 224
11.3. Shader Lists, 24, 30, 46
11.4. Shader Graphs, 37, 41, 46, 225
11.5. Phenomena, 41, 46
11.5.1. Phenomenon Interface Assignments, 42
11.5.2. Shader and Phenomenon Options, 28
11.5.3. Phenomenon Roots, 43
12.1.1. Image Types, 421
13.7. Demand-loaded Placeholder Geometry, 271, 282
14.1. Instances, 14
14.2. Instance Groups, 16
16. Incremental Changes and Animations, 293
19. The Options Block, 12
19.2. Rendering Quality and Performance, 71, 184
19.3. Final Gathering, Global Illumination, and Caustics, 235
A. Command Line Options, 71
A.5. Image Copy: im exports 35
References to Volumes I and II

Throughout this book, marginal references to the first two volumes of mental ray documentation, Volume I: Rendering with mental ray and Volume II: Programming mental ray point to further information about the current topic. This index lists the pages in this book that discuss the topics of Volumes I and II, sorted by section title.

### Volume I: Rendering with mental ray

1. Scenes and Animation, 11
   1.2. Scenes and Animation, 11

2. Geometric Objects, 12
   1.2.1. Geometric Objects, 12

3. Materials, 13
   1.2.2. Materials, 13

4. Light Sources, 127
   1.2.3. Light Sources, 127

5. Cameras, 6
   1.2.4. Cameras, 6

6. Shaders, 49
   1.3. Shaders, 49

7. Transparency, Refractions, and Reflections, 69
   1.5.1. Transparency, Refractions, and Reflections, 69

8. Stages of Image Generation, 222
   1.9. Stages of Image Generation, 222

9. A Simple Scene, 11, 18
   2.1. A Simple Scene, 11, 18

10. Anatomy of a Scene, 15–17
    2.2. Anatomy of a Scene, 15–17

11. Cameras, 11
    2.3. Cameras, 11

12. Pinhole Cameras, 399
    2.1. Pinhole Cameras, 399

3.3. Aspect Ratio, 399

4.1. Color and Illumination, 154, 156, 158, 160

4.2. Texture Mapping, 77

4.2.1. Texture Projections, 84, 86

4.4. Bump Mapping, 29, 35, 297

4.5. Displacement Mapping, 247

4.6. Anisotropic Shading, 160

4.9. Reflection, 183

4.10. Transparency and Refraction, 197

5. Light and Shadow, 127, 128, 130

5.1. Point, Spec, and Infinite Lights, 32, 132

5.2. Raytraced Shadows, 130

5.2.2. Soft Shadows with Area Light Sources, 143

5.2.3. Shadow Modes: Regular, Sorted, Segmented, 150

5.3. Fast Shadows with Shadow Maps, 130

6.3. Ray Marching, 360, 370

7. Caustics and Global Illumination, 57, 219

7.1. Photon Mapping vs. Final Gathering, 6, 151, 220, 250

7.2. Local Illumination vs. Caustics vs. Global Illumination, 185

7.6. Caustics, 219, 225

7.7.4. Final Gathering, 220

7.7.5. Connect, 95

10.1. Outline Contours, 95

11. Shaders and Phong, 21, 46

11.1. Declarations, 26, 46

11.2. Definitions, 26, 46

11.3. Shader Lists, 24, 30, 46

14.6. Shader Graphs, 37, 41, 46, 225

14.7. Phong, 41, 46

15.5. Phong, 41, 46

15.5.1. Phenomenon Interface Assignments, 42

15.5.2. Shader and Phenomenon Options, 28

15.5.3. Phenomenon Roots, 43

17.1. Image Types, 42

17.7. Demand-loaded Placeholder Geometry, 271, 282

18.1. Instances, 14

18.2. Instance Groups, 16

18.3. Incremental Changes and Animations, 283

19. The Options Block, 12

19.2. Rendering Quality and Performance, 71, 184

19.2. Final Gathering, Global Illumination, and Caustics, 235

A. Command Line Options, 71
A.5. Image Copy: InfoLoopy, 83
C. Base Shaders, 28
Throughout this book, marginal references to the first two volumes of mention, *Volume I: Rendering with mental ray* and *Volume II: Programming*, further information about the current topic. This index lists the pages in the topics of Volumes I and II, sorted by section title.

*Volume I: Rendering with mental ray*

1.2. Scenes and Animations, 11
1.2.1. Geometric Objects, 12
1.2.2. Materials, 13
1.2.3. Light Sources, 127
1.2.4. Cameras, 6
1.3. Shaders, 11, 49
1.5.1. Transparency, Refractions, and Reflections, 69
1.9. Stages of Image Generation, 222

2.1. A Simple Scene, 11, 18
2.2. Anatomy of a Scene, 15–17

3. Cameras, 11
3.1. Pinhole Cameras, 399
3.2. Image Resolution, 399

---

7. Caustics and Global Illumination

7.1. Photon Mapping

7.2. Local Illumination

7.6. Caustics, 219, 232

7.7.4. Final Gathering

10. Contours, 93

10.1. Outline Contours

11. Shaders and Phenomena

11.1. Declarations, 16

11.2. Definitions, 26

11.3. Shader Lists, 24

11.4. Shader Graphs
Throughout this book, marginal references to the first two volumes of mention, *Volume I: Rendering with mental ray* and *Volume II: Programming*, provide further information about the current topic. This index lists the pages in the topics of Volumes I and II, sorted by section title.

**Volume I: Rendering with mental ray**

1.2. Scenes and Animations, 11
1.2.1. Geometric Objects, 12
1.2.2. Materials, 13
1.2.3. Light Sources, 127
1.2.4. Cameras, 6
1.3. Shaders, 11, 49
1.5.1. Transparency, Refractions, and Reflections, 69
1.9. Stages of Image Generation, 222
2.1. A Simple Scene, 11, 18
2.2. Anatomy of a Scene, 15–17
3. Cameras, 11
3.1. Pinhole Cameras, 399
3.2. Image Resolution, 399

**Volume II: Programming**

7. Caustics and Global Illumination
   7.1. Photon Mapping
       6, 151, 220, 230
   7.2. Local Illumination
   7.6. Caustics, 219, 232
   7.7.4. Final Gathering
   7.4. Final Gathering
   10. Contours, 93
   10.1. Outline Contours
   11. Shaders and Phenomena
   11.1. Declarations, 16
   11.2. Definitions, 26
   11.3. Shader Lists, 24
   11.4. Shader Graphs
Throughout this book, marginal references to the first two volumes of mention, Volume I: Rendering with mental ray and Volume II: Programming, provide further information about the current topic. This index lists the pages in which the topics of Volumes I and II, sorted by section title.

Volume I: Rendering with mental ray
1.2. Scenes and Animations, 11
  1.2.1. Geometric Objects, 12
  1.2.2. Materials, 13
  1.2.3. Light Sources, 127
  1.2.4. Cameras, 6
  1.3. Shaders, 11, 49
  1.5.1. Transparency, Refractions, and Reflections, 69
  1.9. Stages of Image Generation, 222
2.1. A Simple Scene, 11, 18
  2.2. Anatomy of a Scene, 15–17
3. Cameras, 11
  3.1. Pinhole Cameras, 399
  3.2. Image Resolution, 399

Volume II: Programming
7. Caustics and Global Illumination
  7.1. Photon Mapping
  6, 151, 220, 230
  7.2. Local Illumination
  7.6. Caustics, 219, 231
  7.7.4. Final Gathering
10. Contours, 93
  10.1. Outline Contours
11. Shaders and Phenomena
  11.1. Declarations, 16
  11.2. Definitions, 26
  11.3. Shader Lists, 24
  11.4. Shader Graphs
Cross-referencing in the shader book

Shader code examples in the text point to the full source code listing in Appendix B.
Once we've acquired the value of parameter \texttt{shiny} in line 9, the structure of main part of the shader in lines 10–13 is the same as \texttt{specular\_reflection}. In line 10 we determine the glossy reflection direction. Or rather, we should say that we determine one possible glossy direction. The direction values determined by \texttt{mi\_reflection\_dir\_glossy} are chosen somewhere within the cone determined by the \texttt{shiny} parameter. That “somewhere” is important; mental ray determines successive values of the glossy direction so that the rays are well distributed within the cone. (We’ll talk more about this in the next section.)

As in the \texttt{specular\_reflection} shader, if the reflected ray did not strike another object or the trace depth would be exceeded, we use the color of the environment in line 13 for the result of our shader.

A single ray sent for each glossy reflection produces the grainy look in Figure 14.12. Our contrast value in the options block is \texttt{0.1 .1 .1}. Will increasing the quality of the render by lowering the contrast value help with grainy look of our glossy reflection? We'll set the contrast to \texttt{.01 .01 .01} to find out.
Once we've acquired the value of parameter shiny in line 9, the structure of main part of the shader in lines 10-15 is the same as specular.reflection. In line 10 we determine the glossy reflection direction. Of rather, we should say that we determine one possible glossy direction. The direction values determined by mi_reflection_dir.glossy are chosen somewhere within the cone determined by the shiny parameter. That "somewhere" is important; mental ray determines successive values of the glossy direction so that the rays are well distributed within the cone. (We'll talk more about this in the next section.)

As in the specular.reflection shader, if the reflected ray did not strike another object or the trace depth would be exceeded, we use the color of the environment in line 13 for the result of our shader.

A single ray sent for each glossy reflection produces the grainy look in Figure 14.12. Our contrast value in the options block is 0.1 0.1. Will increasing the quality of the render by lowering the contrast value help with grainy look of our glossy reflection? We'll set the contrast to 0.01 0.01 to find out.
1  struct glossy_reflection {
2       miScalar shiny;
3  };
4
5  miBoolean glossy_reflection (  
6     miColor *result, miState *state, struct glossy_reflection *params )
7  {
8     miVector reflection_dir;
9     miScalar shiny = *mi_eval_scalar(&params->shiny);
10    mi_reflection_dir_glossy(&reflection_dir, state, shiny);
11
12    if (!mi_trace_reflection(result, state, &reflection_dir))
13       mi_trace_environment(result, state, &reflection_dir);
14
15    return miTRUE;
16  }

Figure 14.11: Shader source of glossy_reflection (p.513)

Once we’ve acquired the value of parameter shiny in line 9, the structure of main part of the shader in lines 10–13 is the same as specular_reflection. In line 10 we determine the glossy
Figure 14.11: Shader source of glossy_reflection

Once we've acquired the value of parameter shiny in line 9, the structure of main part of the shader in lines 10–13 is the same as specular_reflection. In line 10 we determine the glossy
**specular.reflection**

```c
declare shader
    color "specular.reflection" ()
version 1
apply material
end declare

#include "shader.h"

int specular.reflection.version(void) { return 1; }

niBoolean specular.reflection(
    niColor *result, niState *state, void *params
)
{
    miVector reflection_dir;
    miVector *reflection_dir(state->reflection_dir);
    ReflectionDir( state, &reflection_dir);
    if (mi.trace.reflection(result, state, &reflection_dir))
        mi.trace.environment(result, state, &reflection_dir);
    return miTRUE;
}
```

**glossy.reflection**

```c
declare shader
    color "glossy.reflection" (,
        scalar "shiny" default 5 )
version 1
apply material
end declare

#include "shader.h"

struct glossy.reflection
{
    miScalar shiny;
};

int glossy.reflection.version(void) { return 1; }

niBoolean glossy.reflection(
    niColor *result, niState *state, struct glossy.reflection *params
)
{
    miVector reflection_dir;
    miScalar shiny = mi_eval.scalar(&params->shiny);
    miVector *reflection_dir_glossy = (mi_trace_dir_glossy(result, state, shiny);
    if (mi.trace.reflection(result, state, &reflection_dir))
        mi.trace.environment(result, state, &reflection_dir);
    return miTRUE;
}
```
specular_reflection

```c
declare shader
    color "specular_reflection" ()
    version 1
    apply material
end declare

#include "shader.h"

int specular_reflection_version(void) { return 1; }

miBoolean specular_reflection (miColor *result, miState *state, void *params)
{
    miVector reflection_direction,
    mi_reflection_dir(reflection_direction, state);

    if (!mi_trace_reflection(result, state, &reflection_direction))
        mi_trace_environment(result, state, &reflection_direction);

    return miTRUE;
}
```

glossy_reflection

```c
declare shader
    color "glossy_reflection"
    scalar "shiny" default 5
    version 1
    apply material
end declare

#include "shader.h"

struct glossy_reflection {
    miScalar shiny;
};

int glossy_reflection_version(void) { return 1; }

miBoolean glossy_reflection (miColor *result, miState *state, struct glossy_reflection *params)
{
    miVector reflection_dir,
    mi_reflection_dir, glossy(reflection_dir, state, shiny);

    if (!mi_trace_reflection(result, state, &reflection_dir))
        mi_trace_environment(result, state, &reflection_dir);

    return miTRUE;
}
specular_reflection

```c
// declare shader
color "specular_reflection" ()
version 1
apply material
end declare

#include "shader.h"

int specular_reflection_version(void) { return 1; }

miBoolean specular_reflection
(miColor *result, miState *state, void *params)
{
    miVector reflection_dir;
    mi_reflection_dir(reflection_dir, state);
    if (!ni_trace_reflection(result, state, &reflection_dir))
        ni_trace_environment(result, state, &reflection_dir);
    return miTRUE;
}
```

glossy_reflection

```c
// declare shader
color "glossy_reflection" ()
scalar "shiny" default 5
version 1
apply material
end declare

#include "shader.h"

struct glossy_reflection {
    miScalar shiny;
};

int glossy_reflection_version(void) { return 1; }

miBoolean glossy_reflection
(miColor *result, miState *state, glossy_reflection *params)
{
    miVector reflection_dir;
    mi_reflection_dir_glossy(reflection_dir, state, shiny);
    if (!ni_trace_reflection(result, state, &reflection_dir))
        ni_trace_environment(result, state, &reflection_dir);
    return miTRUE;
}
```
glossy_reflection

declare shader
    color "glossy_reflection" (    
        scalar "shiny" default 5 )
    version 1
    apply material
end declare

#include "shader.h"

struct glossy_reflection {
    miScalar shiny;
};

int glossy_reflection_version(void) { return 1; }

miBoolean glossy_reflection (    
    miColor *result, miState *state, struct glossy_reflection *params )
{
    miVector reflection_dir;
    miScalar shiny = *mi_eval_scalar(&params->shiny);
    mi_reflection_dir_glossy(&reflection_dir, state, shiny);

    if (!mi_trace_reflection(result, state, &reflection_dir))
        mi_trace_environment(result, state, &reflection_dir);

    return miTRUE;
}
glossy_reflection

```c
# declare shader
  color "glossy_reflection" (  
    scalar "shiny" default 5 )  
  version 1
  apply material
end declare

#include "shader.h"

struct glossy_reflection {  
  miScalar shiny;
};

// int glossy_reflection_version(void) { return 1; }

miBoolean glossy_reflection (  
  miColor *result, miState *state, struct glossy_reflection *params )  
{  
  miVector reflection_dir;
  miScalar shiny = *mi_eval_scalar(&params->shiny);
  mi_reflection_dir_glossy(&reflection_dir, state, shiny);

  if (!mi_trace_reflection(result, state, &reflection_dir))
    mi_trace_environment(result, state, &reflection_dir);

  return miTRUE;
}
```
# declare shader
    color "glossy_reflection" (  
        scalar "shiny" default 5 )  
version 1  
apply material  
end declare

#include "shader.h"

struct glossy_reflection {  
    miScalar shiny;  
};

int glossy_reflection_version(void) { return 1; }

miBoolean glossy_reflection (  
    miColor *result, miState *state, struct glossy_reflection *params  
) {
    miVector reflection_dir;  
    miScalar shiny = *mi_eval_scalar(&params->shiny);  
    mi_reflection_dir_glossy(&reflection_dir, state, shiny);

    if (!mi_trace_reflection(result, state, &reflection_dir))
        mi_trace_environment(result, state, &reflection_dir);

    return miTRUE;
}
Cross-referencing in the shader book

Source code listings in Appendix B point back to the code’s description in the text.
specular_reflection

```c
declare shader
    color "specular_reflection" ()
    version 1
    apply material
end declare

#include "shader.h"

int specular_reflection_version(void) { return 1; }

miBoolean specular_reflection(
    miColor *result, miState *state, void *params)
{
    miVector reflection_dir;
    mi_reflection_dir(result, state, reflection_dir);
    if (!mi_trace_reflection(result, state, reflection_dir))
        mi_trace_environment(result, state, reflection_dir);
    return miTRUE;
}
```

glossy_reflection

```c
declare shader
    color "glossy_reflection" (
        scalar "shiny" default 5)
    version 1
    apply material
end declare

#include "shader.h"

struct glossy_reflection {
    miScalar shiny;
};

int glossy_reflection_version(void) { return 1; }

miBoolean glossy_reflection(
    miColor *result, miState *state, struct glossy_reflection *params)
{
    miVector reflection_dir;
    mi_reflection_dir(result, state, reflection_dir);
    miScalar shiny = mi_eval_scalar(params->shiny);
    mi_reflection_dir_glossy(result, state, shiny);
    if (!mi_trace_reflection(result, state, reflection_dir))
        mi_trace_environment(result, state, reflection_dir);
    return miTRUE;
}
```
specular.reflection

declare shader
    color "specular.reflection" ()
    version 1
    apply material
end declare

#include "shader.h"

int specular.reflection.version(void) { return 1; }

miBoolean specular.reflection (
    miColor *result, miState *state, void *params )
{
    miVector reflection_direction;
    mi_reflection_dir(reflection_direction, state);
    if (!mi_trace_reflection(result, state, &reflection_direction))
        mi_trace_environment(result, state, &reflection_direction);

    return miFALSE;
}

glossy.reflection

declare shader
    color "glossy.reflection" (  
        scalar "shiny" default 5 )
    version 1
    apply material
end declare

#include "shader.h"

struct glossy.reflection {
    miScalar shiny;
};

int glossy.reflection.version(void) { return 1; }

miBoolean glossy.reflection (
    miColor *result, miState *state, struct glossy.reflection *params )
{
    miVector reflection_dir;
    mi_reflection_dir(result, state, &reflection_dir, shiny);
    mi_trace_reflection(result, state, &reflection_dir);
    mi_trace_environment(result, state, &reflection_dir);

    return miFALSE;
}
Once we've acquired the value of parameter shiny in line 9, the structure of main part of the shader in lines 10-13 is the same as specular_reflection. In line 10 we determine the glossy reflection direction. Of rather, we should say that we determine one point of glossy direction. The direction values determined by mi反思_direction_glossy are chosen somewhere within the cone determined by the shiny parameter. That "somewhere" is important; mental ray determines successive values of the glossy direction so that the rays are well distributed within the cone. (We'll talk more about this in the next section.)

As in the specular_reflection shader, if the reflected ray did not strike another object or the trace depth would be exceeded, we use the color of the environment in line 13 for the result of our shader.

A single ray sent for each glossy reflection produces the grainy look in Figure 14.12. Our contrast value in the options block is 1.1.1. Will increasing the quality of the render by lowering the contrast value help with grainy look of our glossy reflection? We'll set the contrast to .01 .01 .01 to find out.
Cross-referencing in the shader book

Utility functions encapsulate lower-level details and make shader code clearer.
7.3 Clarifying the shader with auxiliary functions

Defining auxiliary functions can clarify the method being used in the shader. In this chapter we'll begin to develop a library of auxiliary functions that will make the strategies of the shaders clearer to their code. All of these functions will be named with a prefix of *mvsu* ("mental images auxiliary," in the same spirit as the *m1u* functions in the mental ray library), and will be consolidated in a library.

First we'll define a function to rescale a value from one range to another, so that the proportional relationships of the new values match those of the original:

![Figure 7.8: Converting from one scale to another](image)

This basic function will be useful when we want to define a relationship between scales of arbitrary type, like the mapping from the near and far parameter values to a normalized range of 0.0 to 1.0.

```cpp
1 double mvsu_fit(double v, double oldmin, double oldmax, double newmin, double newmax)
2 {
3  return newmin + ((v - oldmin) / (oldmax - oldmin)) * (newmax - newmin);
4 }
```

![Figure 7.9: Function mvsu_fit](image)

Since we're blending two colors based on a weighting factor, we'll define a function to perform this color blending:
7.3 Clarifying the shader with auxiliary functions

Defining auxiliary functions can clarify the method being used in the shader. In this chapter we'll begin to develop a library of auxiliary functions that will make the strategies of the shaders clearer from their code. All of these functions will be named with a prefix of `miaux` ("mental images auxiliary," in the same spirit as the `m1_` functions in the mental ray library), and will be consolidated in a library.

First we'll define a function to rescale a value from one range to another, so that the proportional relationships of the new values match those of the original:

![Diagram](image)

Figure 7.8: Converting from one range to another

This basic function will be useful when we want to define a relationship between scales of arbitrary type, like the mapping from the `near` and `far` parameter values to a normalized range of 0.0 to 1.0.

```
double miaux_fit(double oldmin, double oldmax, double newmin, double newmax)
{
    return newmin + ((v - oldmin) / (oldmax - oldmin)) * (newmax - newmin);
}
```

Figure 7.9: Function miaux_fit

Since we're blending two colors based on a weighting factor, we'll define a function to perform this color blending:
First we’ll define a function to rescale a value from one range to another, so that the proportional relationships of the new values match those of the original:

![Diagram](image)

Figure 7.8: Converting from one scale to another

This basic function will be useful when we want define a relationship between scales of arbitrary type, like the mapping from the `near` and `far` parameter values to a normalized range of 0.0 to 1.0.

```c
1  double miaux_fit(
2      double v, double oldmin, double oldmax, double newmin, double newmax)
3  {
4      return newmin + ((v - oldmin) / (oldmax - oldmin)) * (newmax - newmin);
5  }
```

Figure 7.9: Function `miaux_fit`
First we’ll define a function to rescale a value from one range to another, so that the proportional relationships of the new values match those of the original:

![Diagram showing rescaling from original range to proportional fit to new range]

Figure 7.8: Converting from one scale to another

This basic function will be useful when we want define a relationship between scales of arbitrary type, like the mapping from the near and far parameter values to a normalized range of 0.0 to 1.0.

```c
1 double miaux_fit
2     double v, double oldmin, double oldmax, double newmin, double newmax)
3 {
4     return newmin + ((v - oldmin) / (oldmax - oldmin)) * (newmax - newmin);
5 }
```

Figure 7.9: Function miaux_fit
p->v0 = mi_eval_vector(p_params->v0);
p->approximation = mi_eval_integer(p_params->approximation);
p->degree = mi_eval_integer(p_params->degree);

miaux_define_hair_object(
    p->name, hair_geo_4v_bbox, p, result, hair_geo_4v_callback);

    return mTRU;
}

mIBOOLEAN hair_geo_4v_callback(mITag tag, void *params)
{
    mIBHair_list *hair_list;
    mIScalar *hair_scalars;
    mIIndex *hair_indices;
    hair_geo_4v_i p = (Hair_geo_4v_i *)params;
    int hair_count = 4, hair_scalar_count = 6 * 3 + 1;

    mI_api_incremental(mTRUE);
    miaux_define_hair_object(p->name, hair_geo_4v_bbox, p, NULL, NULL);
    hair_list = mI_api_hair_begin();
    hair_list->approx = p->approximation;
    hair_list->degree = p->degree;
    mI_api_hair_info(0, 'v', 1);

    hair_scalars = mI_api_hair_scalars_begin(hair_scalar_count);
    hair_scalars->p-radius = p->radius;
    miaux_append_hair_vertex(hair_scalars, p->v0);
    miaux_append_hair_vertex(hair_scalars, p->v1);
    miaux_append_hair_vertex(hair_scalars, p->v2);
    miaux_append_hair_vertex(hair_scalars, p->v3);
    miaux_hair_scalars_end(hair_scalar_count);

    hair_indices = mI_api_hair_hairs_begin(hair_count + 1);
    hair_indices[0] = 0;
    hair_indices[1] = hair_scalar_count;
    mI_api_hair_hairs_end();

    mI_api_hair_end();
    mI_api_object_end();

    return mTRUE;
}

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>miaux_init_bbox</td>
<td>998</td>
</tr>
<tr>
<td>miaux_adjust_bbox</td>
<td>998</td>
</tr>
<tr>
<td>miaux_set_factor</td>
<td>998</td>
</tr>
<tr>
<td>miaux_describe_bbox</td>
<td>998</td>
</tr>
<tr>
<td>miaux_define_hair_object</td>
<td>998</td>
</tr>
<tr>
<td>miaux_rag_to_string</td>
<td>998</td>
</tr>
<tr>
<td>miaux_append_hair_vertex</td>
<td>998</td>
</tr>
</tbody>
</table>
p->v0 = msi_eval_vector(pParams->v4);
p->approximation = msi_eval_integer(pParams->approximation);
p->degree = msi_eval_integer(pParams->degree);

miavax_define_hair_object(
    p->name, hair_geo_4v_bbox, p, result, hair_geo_4v_callback);

return miTRUE;

}

miBoolean hair_geo_4v_callback(miTag tag, void *params)
{
  miHair_list *hair_list;
  miScalar *hair_scalars;
  miIndex *hair_indices;
  hair_geo_4v_t *p = (hair_geo_4v_t *)params;
  int hair_count = 2, hair_scalar_count = 4 * 3 + 7;

  mi_api_incremental(miTRUE);
  miavax_define_hair_object(p->name, hair_geo_4v_bbox, p, NULL, NULL);
  hair_list = mi_api_hair_begin();
  hair_list->approx = p->approximation;
  hair_list->degree = p->degree;
  mi_api_hair_info(0, 'v', 1);

  hair_scalars = mi_api_hair_scalars_begin(hair_scalar_count);
  hair_scalars->p = p->radius;
  miavax_append_hair_vertex(hair_scalars, p->v0);
  miavax_append_hair_vertex(hair_scalars, p->v1);
  miavax_append_hair_vertex(hair_scalars, p->v2);
  miavax_append_hair_vertex(hair_scalars, p->v3);
  mi_api_hair_scalars_end(hair_scalar_count);

  hair_indices = mi_api_hair_hairs_begin(hair_count + 1);
  hair_indices[0] = 0;
  hair_indices[1] = hair_count;
  mi_api_hair_hairs_end();

  mi_api_hair_end();
  mi_api_object_end();

  return miTRUE;

}
miBoolean hair_geo_4v_callback(miTag tag, void *params)
{
    miHair_list *hair_list;
    miScalar   *hair_scalars;
    miGeoIndex *hair_indices;
    hair_geo_4v_t *p = (hair_geo_4v_t *)params;
    int hair_count = 1, hair_scalar_count = 4 * 3 + 1;

    mi_api_incremental(miTRUE);
    miaux_define_hair_object(p->name, hair_geo_4v_bbox, p, NULL, NULL);
    hair_list = mi_api_hair_begin();
    hair_list->approx = p->approximation;
    hair_list->degree = p->degree;
    mi_api_hair_info(0, 'r', 1);

    hair_scalars = mi_api_hair_scalars_begin(hair_scalar_count);
    *hair_scalars++ = p->radius;
    miaux_append_hair_vertex(&hair_scalars, &p->v1);
    miaux_append_hair_vertex(&hair_scalars, &p->v2);
    miaux_append_hair_vertex(&hair_scalars, &p->v3);
    miaux_append_hair_vertex(&hair_scalars, &p->v4);
    mi_api_hair_scalars_end(hair_scalar_count);

    hair_indices = mi_api_hair_hairs_begin(hair_count + 1);
    hair_indices[0] = 0;
    hair_indices[1] = hair_scalar_count;
    mi_api_hair_hairs_end();

    mi_api_hair_end();
    mi_api_object_end();

    return miTRUE;
}
miBoolean hair_geo_4v_callback(miTag tag, void *params)
{
    miHair_list *hair_list;
    miScalar    *hair_scalars;
    miGeoIndex  *hair_indices;
hair_geo_4v_t *p = (hair_geo_4v_t *)params;
int hair_count = 1, hair_scalar_count = 4 * 3 + 1;

    mi_api_incremental(miTRUE);
    miaux_define_hair_object(p->name, hair_geo_4v_bbox, p, NULL, NULL);
    hair_list = mi_api_hair_begin();
    hair_list->approx = p->approximation;
    hair_list->degree = p->degree;
    mi_api_hair_info(0, 'r', 1);

    hair_scalars = mi_api_hair_scalars_begin(hair_scalar_count);
    *hair_scalars++ = p->radius;
    miaux_append_hair_vertex(&hair_scalars, &p->v1);
    miaux_append_hair_vertex(&hair_scalars, &p->v2);
    miaux_append_hair_vertex(&hair_scalars, &p->v3);
    miaux_append_hair_vertex(&hair_scalars, &p->v4);
    mi_api_hair_scalars_end(hair_scalar_count);

    hair_indices = mi_api_hair_hairs_begin(hair_count + 1);
    hair_indices[0] = 0;
    hair_indices[1] = hair_scalar_count;
    mi_api_hair_hairs_end();

    mi_api_hair_end();
    mi_api_object_end();

    return miTRUE;
}
p->v0 = msi_eval_vector(&params->v0);
p->approximation = msi_eval_integer(&params->approximation);
p->degree = msi_eval_integer(&params->degree);

msiu_def_hair_object(
    p->name, hair_geo_4v_bbox, p, result, hair_geo_4v_callback);

return miTRUE;
}

miBoolean hair_geo_4v_callback(miTag tag, void *params)
{
    miHair_list *hair_list;
    miScalar  *hair_scalars;
    miIndex *hair_indices;
    hair_geo_4v_t p = (hair_geo_4v_t *)params;
    int hair_count = 1, hair_scalar_count = 4 * 3 + 1;
    int len = incremental(miTRUE);

    miDef_hair_object(p->name, hair_geo_4v_bbox, p, MiNIL, MiNIL);
    hair_list = miApi_hair_begin();
    hair_list->approx = p->approximation;
    hair_list->degree = p->degree;
    miApi_hair_info(0, 'v', 1);

    hair_scalars = miApi_hair_scalars_begin(hair_scalar_count);
    hair_scalars = miApi_hair_scalarsܫ(&hair_scalars);
    msiApp_hair_vertex(hair_scalars, &p->v0);
    msiApp_hair_vertex(hair_scalars, &p->v1);
    msiApp_hair_vertex(hair_scalars, &p->v2);
    msiApp_hair_vertex(hair_scalars, &p->v3);
    msiApi_hair_scalars_end(hair_scalar_count);

    hair_indices = miApi_hair_indices_begin(hair_count + 1);
    hair_indices[0] = 0;
    hair_indices[1] = hair_scalar_count;
    miApi_hair_indices_end();
    miApi_hair_end();
    miApi_object_end();

    return miTRUE;
}

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msiuiInitBox</td>
<td>998</td>
</tr>
<tr>
<td>msiuiAddBox</td>
<td>998</td>
</tr>
<tr>
<td>msiuiSetVector</td>
<td>996</td>
</tr>
<tr>
<td>msiuiDescribeBox</td>
<td>998</td>
</tr>
<tr>
<td>msiuiDefHairObject</td>
<td>998</td>
</tr>
<tr>
<td>msiuiTagToString</td>
<td>990</td>
</tr>
<tr>
<td>msiuiAppendHairVertex</td>
<td>998</td>
</tr>
</tbody>
</table>
p->v0 = msi_eval_vector(&params->v0);
p->approximation = msi_eval_integer(&params->approximation);
p->degree = msi_eval_integer(&params->degree);

miux_define_hair_object(
    p->name, hair_geo_4v_bbox, p, result, hair_geo_4v_callback);

return mTRUE;
}

miBoolean hair_geo_4v_callback(miTag tag, void *params)
{
    miHairList *hair_list;
    miScalar *hair_scalars;
    miIndex *hair_indices;
    hair_geo_4v_t *p = (hair_geo_4v_t *)params;
    int hair_count = 1, hair_scalar_count = 6 * 3 + 1;

    mapi_incremental(mTRUE);
    miux_define_hair_object(p->name, hair_geo_4v_bbox, p, NULL, NULL);
    hair_list = mapi_hair_begin();
    hair_list->approx = p->approximation;
    hair_list->degree = p->degree;
    mapi_hair_info(0, "v", 1);
    hair_scalars = mapi_hair_scalars_begin(hair_scalar_count);
    *hair_scalars++ = p->radius;
    miapi_append_hair_vertex(hair_scalars, &p->v1);
    miapi_append_hair_vertex(hair_scalars, &p->v2);
    miapi_append_hair_vertex(hair_scalars, &p->v3);
    miapi_append_hair_vertex(hair_scalars, &p->v4);
    miapi_hair_scalars_end(hair_scalar_count);

    hair_indices = miapi_hair_hairs_begin(hair_count + 1);
    hair_indices[0] = 0;
    hair_indices[1] = hair_scalar_count;
    mapi_hair_hairs_end();
    mapi_hair_end();
    mapi_object_end();

    return mTRUE;
}

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>miux_init_bbox</td>
<td>598</td>
</tr>
<tr>
<td>miux_adjust_bbox</td>
<td>598</td>
</tr>
<tr>
<td>miux_set_factor</td>
<td>598</td>
</tr>
<tr>
<td>miux_describe_bbox</td>
<td>598</td>
</tr>
<tr>
<td>miux_def_hair_obj</td>
<td>598</td>
</tr>
<tr>
<td>miux_tag_to_string</td>
<td>598</td>
</tr>
<tr>
<td>miux_append_hair_v</td>
<td>598</td>
</tr>
</tbody>
</table>
```c
hair_indices = mi_api_hair_hairs_begin(hair_count + 1);
hair_indices[0] = 0;
hair_indices[1] = hair_scalar_count;
mi_api_hair_hairs_end();

mi_api_hair_end();
mi_api_object_end();

return miTRUE;
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>miaux_init_bbox</td>
<td>598</td>
</tr>
<tr>
<td>miaux_adjust_bbox</td>
<td>598</td>
</tr>
<tr>
<td>miaux_set_vector</td>
<td>590</td>
</tr>
<tr>
<td>miaux_describe_bbox</td>
<td>598</td>
</tr>
<tr>
<td>miaux_define_hair_object</td>
<td>598</td>
</tr>
<tr>
<td>miaux_tag_to_string</td>
<td>590</td>
</tr>
<tr>
<td>miaux_append_hair_vertex</td>
<td>598</td>
</tr>
</tbody>
</table>
hair_indices = mi_api_hair_hairs_begin(hair_count + 1);
hair_indices[0] = 0;
hair_indices[1] = hair_scalar_count;
mi_api_hair_hairs_end();

mi_api_hair_end();
mi_api_object_end();

return miTRUE;

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>miaux_init_bbox</td>
<td>598</td>
</tr>
<tr>
<td>miaux_adjust_bbox</td>
<td>598</td>
</tr>
<tr>
<td>miaux_set_vector</td>
<td>590</td>
</tr>
<tr>
<td>miaux_describe_bbox</td>
<td>598</td>
</tr>
<tr>
<td>miaux_define_hair_object</td>
<td>598</td>
</tr>
<tr>
<td>miaux_tag_to_string</td>
<td>590</td>
</tr>
<tr>
<td>miaux_append_hair_vertex</td>
<td>598</td>
</tr>
</tbody>
</table>
Chapter 20 — Modeling hair

void miaux_define_hair_object(
    miTag name_tag, miucion_base_function bbase_function, void *params,
    miaux_base_header_result, miapi_object_callback callback)
{
    miTag tag;
    miObject obj;
    char *name = miucion_tag_to_string(name_tag, "\t\t\t\t\t\t\n");
    obj = miapi_object_begin(mi_bin_strdup(name));
    obj->visible = miTRUE;
    obj->shadow = obj->reflection = obj->refraction = 3;
    bbase_function(obj, params);
    if (gheckader_result == NULL || callback == NULL) {
        miapi_object_callback(callback, params);
        tag = miapi_object_end(obj);
        miaux_defobj_result(gheckader_result, tag);
        obj = miaux_object_time_name_edit(tag);
        obj->type.placeholder_list.type = miOBJ_HAIR;
        mi_socket_edit_end(tag);
    }
}

void miaux_discribe_box(miObject obj)
{
    mi_progress("Object box: Xf,Xf,Xa Yf,Yf,Ya Zf,Zf,Za",
        obj->bbox_min.x, obj->bbox_min.y, obj->bbox_min.z,
        obj->bbox_max.x, obj->bbox_max.y, obj->bbox_max.z);
}

void miaux_adjust_box(miObject obj, mivec4 v, miscalor extra)
{
    mivec4 extra;
    miaux_set_vector(v.x, v.y, v.z, extra);
    mi_vector_sub(min, v, v.extra);
    mi_vector_add(min, v, v.extra);
    obj->bbox_min.x = obj->bbox_min.x + extra.x;
    obj->bbox_min.y = obj->bbox_min.y + extra.y;
    obj->bbox_min.z = obj->bbox_min.z + extra.z;
}

void miaux_init_box(miObject obj)
{
    obj->bbox_min.x = miONE SCALAR;
    obj->bbox_min.y = miONE SCALAR;
    obj->bbox_min.z = miONE SCALAR;
    obj->bbox_max.x = miONE SCALAR;
    obj->bbox_max.y = miONE SCALAR;
    obj->bbox_max.z = miONE SCALAR;
}

void miaux_append_hair_vertex(miscalor **scalar_array, mivec4 *v)
{
    (**scalar_array)[] = v->x;
    (**scalar_array)[1] = v->y;
    (**scalar_array)[2] = v->z;
    (**scalar_array)[3] = 3;
}

void miaux_append_hair_data(
    miscalor **scalar_array, mivec4 *, miscalor *miscalor_position,
    miscalor root_radius, micolor *micolor root, miscalor tip_radius, micolor *tip)
{
    (**scalar_array)[] = v->x;
    (**scalar_array)[1] = v->y;
    (**scalar_array)[2] = v->z;
    (**scalar_array)[3] = 3;
return mi_api.object_end();
}

Chapter 20 – Modeling hair

void miemx_def_hair_obj(
  miTag name_tag, miemx_bbox_function bbox_function, void *params,
  miTag geomshader_result, miapi_object_callback callback)
{
  miTag tag;
  miObject obj;
  char *name = miemx_tag_to_string(name_tag, "hair");
  obj = mi_api.object_begin(mi_emx_startup(name));
  obj->parent = miEMX;
  obj->bbox = bbox_function(obj, params);
  if (geomshader_result != NULL && callback != NULL) {
    mi_api.object_callback(callback, params);
    tag = mi_api_object_end();
    mi_geomshader_add_result(geomshader_result, tag);
    obj = miObject_type_memo_edit(tag);
    obj->geom.placeholder_list_type = miOBJ_HAIR;
    mi_memo_edit_end(tag);
  }
}

void miemx_desc_bbox(miObject *obj)
{
  mi_progress("Object bbox: %f,%f,%f %f,%f,%f",
    obj->bbox_min.x, obj->bbox_min.y, obj->bbox_min.z,
    obj->bbox_max.x, obj->bbox_max.y, obj->bbox_max.z);
}

void miemx_adj_bbox(miObject *obj, miVector *v, miScalar extra)
{
  miVector v_extra, min, max;
  miemx_set_vector(&v_extra, extra, extra, extra);
  mi_vector_sub(min, v, v_extra);
  mi_vector_add(max, v, extra);
  mi_vector_min(obj, &bbox_min, &bbox_min, &bbox_min, &bbox_min, &bbox_min);
  mi_vector_max(obj, &bbox_max, &bbox_max, &bbox_max, &bbox_max, &bbox_max);
}

void miemx_init_bbox(miObject *obj)
{
  obj->bbox_min.x = miEMX_SCALED;
  obj->bbox_min.y = miEMX_SCALED;
  obj->bbox_min.z = miEMX_SCALED;
  obj->bbox_max.x = miEMX_SCALED;
  obj->bbox_max.y = miEMX_SCALED;
  obj->bbox_max.z = miEMX_SCALED;
}

void miemx_append_hair_vertex(miScalar **scalar_array, miVector *v)
{
  (*scalar_array)[0] = v->x;
  (*scalar_array)[1] = v->y;
  (*scalar_array)[2] = v->z;
  (*scalar_array) += 3;
}

void miemx_append_hair_data(
  miScalar **scalar_array, miVector *v, miScalar position,
  miScalar root_radius, miScalar root, miScalar tip_radius, miScalar *tip)
{
  (*scalar_array)[0] = v->x;
  (*scalar_array)[1] = v->y;
  (*scalar_array)[2] = v->z;
  (*scalar_array) += 3;

  (*scalar_array)[0] = position;
  (*scalar_array)[1] = position;
  (*scalar_array)[2] = position;
  (*scalar_array) += 3;

  (*scalar_array)[0] = root_radius;
  (*scalar_array)[1] = root_radius;
  (*scalar_array)[2] = root_radius;
  (*scalar_array) += 3;

  (*scalar_array)[0] = root;
  (*scalar_array)[1] = root;
  (*scalar_array)[2] = root;
  (*scalar_array) += 3;

  (*scalar_array)[0] = tip_radius;
  (*scalar_array)[1] = tip_radius;
  (*scalar_array)[2] = tip_radius;
  (*scalar_array) += 3;

  (*scalar_array)[0] = *tip;
  (*scalar_array)[1] = *tip;
  (*scalar_array)[2] = *tip;
  (*scalar_array) += 3;
}
void miaux_adjust_bbox(miObject *obj, miVector *v, miScalar extra)
{
    miVector v_extra, vmin, vmax;
    miaux_set_vector(&v_extra, extra, extra, extra, extra);
    mi_vector_sub(&vmin, v, &v_extra);
    mi_vector_add(&vmax, v, &v_extra);
    mi_vector_min(&obj->bbox_min, &obj->bbox_min, &vmin);
    mi_vector_max(&obj->bbox_max, &obj->bbox_max, &vmax);
}

void miaux_init_bbox(miObject *obj)
{
    obj->bbox_min.x = miHUGE_SCALAR;
    obj->bbox_min.y = miHUGE_SCALAR;
    obj->bbox_min.z = miHUGE_SCALAR;
    obj->bbox_max.x = -miHUGE_SCALAR;
    obj->bbox_max.y = -miHUGE_SCALAR;
    obj->bbox_max.z = -miHUGE_SCALAR;
}
Cross-referencing in the shader book

Example renderings are displayed with the relevant portion of the scene file that produced them.
5.4 Using a shader in a material

To use shader one_color to render a scene, we supply values for its parameters and include it in a material. In Figure 5.8, shader one_color is included anonymously in three different materials for the three object instances.

![Image of shader code and rendered objects]

Figure 5.8: A single color for the entire object defined by shader one_color in the material.

Throughout the book, we’ll be rendering images like Figure 5.8 using the shaders developed in each chapter. Accompanying the image will be a fragment of the scene file that describes how the shader is used, or includes other parts of the scene file that will affect the final image. All the rendered images are collected in Appendix A beginning on page 457 and serve as a visual index to the various techniques we’ll develop.

5.5 Shader programming style

To simplify the structural diagram of shader one_color, the three arguments to the shader function were placed on different lines. To shorten the source code examples throughout the book, most shaders will be shown with their arguments all on the same line.

```plaintext
mIColor one_color (mIColor *Result, mIColor *state, struct one_color *params) {
    mIColor *color = mi_eval_color(*params->color);
    result->r = color->r;
    result->g = color->g;
    result->b = color->b;
    result->a = color->a;
    return miTRUE;
}
```

Figure 5.9: Putting the standard shader arguments on a single line.

But we can go further than just rearranging the code to shorten it. The size of a variable of type mIColor is known by the compiler in advance, a C struct containing a field for each of the red,
Store contour shader returns multiple values. (See Figure 10.7.)

5.4 Using a shader in a material

To use shader one.color to render a scene, we supply values for its parameters and include in it a material. In Figure 5.8, shader one.color is included anonymously in three different materials for the three object instances.

Figure 5.8: A single color for the entire object defined by shader one.color in the material

Throughout the book, we'll be rendering images like Figure 5.8 using the shaders developed in each chapter. Accompanying the image will be a fragment of the scene file that describes how the shader is used, or includes other parts of the scene file that will affect the final image. All the rendered images are collected in Appendix A beginning on page 457 and serve as a visual index to the various techniques we'll develop.

5.5 Shader programming style

To simplify the structural diagram of shader one.color, the three arguments to the shader function were placed on different lines. To shorten the source code examples throughout the book, most shaders will be shown with their arguments all on the same line.

```
mbool32 one_color ( mIColor *Result, mistate *state, struct one_color *params )
{
  miColor <color = mi_float_color (params->color);
  result.r = color->r;
  result.g = color->g;
  result.b = color->b;
  result.a = color->a;
  return mTRUE;
}
```

Figure 5.9: Putting the standard shader arguments on a single line

But we can go further than just rearranging the code to shorten it. The size of a variable of type miColor is known by the compiler in advance (a C struct containing a field for each of the red,
a material. In Figure 5.8, shader one_color is included anonymously in three different materials for the three object instances.

```
material "yellow"
  "one_color" (  
    "color" 1 1 .4 )
end material

material "blue"
  "one_color" (  
    "color" .4 .4 1 )
end material

material "red"
  "one_color" (  
    "color" 1 .4 .4 )
end material
```

Figure 5.8: A single color for the entire object defined by shader one_color in the material

Throughout the book, we’ll be rendering images like Figure 5.8 using the shaders developed in each chapter. Accompanying the image will be a fragment of the scene file that describes how
21.6 Environment shaders for cameras and objects

Different environment shaders can be used for the camera and object instances. For any object instance without an environment shader, the camera's environment shader will be used as the default.

```plaintext
shader "red_sunset"
  "color_ramp"
  "colors" [ 0.3 0.2 0.1 0.0, 0.1 0.2 0.1 0.3, 0.3 0.2 0.1 0.1, 0.3 0.2 0.1 0.0, 0.1 0.2 0.1 0.3, 0.3 0.2 0.1 0.1, 0.3 0.2 0.1 0.0, 0.1 0.2 0.1 0.3, 0.3 0.2 0.1 0.1, 0.3 0.2 0.1 0.0, 0.1 0.2 0.1 0.3, 0.3 0.2 0.1 0.1, 0.3 0.2 0.1 0.0, 0.1 0.2 0.1 0.3, 0.3 0.2 0.1 0.1, 0.3 0.2 0.1 0.0, 0.1 0.2 0.1 0.3, 0.3 0.2 0.1 0.1, 0.3 0.2 0.1 0.0, 0.1 0.2 0.1 0.3, 0.3 0.2 0.1 0.1, 0.3 0.2 0.1 0.0, 0.1 0.2 0.1 0.3, 0.3 0.2 0.1 0.1, 0.3 0.2 0.1 0.0, 0.1 0.2 0.1 0.3, 0.3 0.2 0.1 0.1, 0.3 0.2 0.1 0.0 ];

shader "pink_sunset"
  "color_ramp"
  "colors" [ 0.7 0.2 0.1 0.0, 0.1 0.1 0.1 0.0, 0.3 0.2 0.1 0.0, 0.5 0.2 0.1 0.0, 0.7 0.2 0.1 0.0, 0.1 0.1 0.1 0.0, 0.3 0.2 0.1 0.0, 0.5 0.2 0.1 0.0, 0.7 0.2 0.1 0.0, 0.1 0.1 0.1 0.0, 0.3 0.2 0.1 0.0, 0.5 0.2 0.1 0.0, 0.7 0.2 0.1 0.0, 0.1 0.1 0.1 0.0, 0.3 0.2 0.1 0.0, 0.5 0.2 0.1 0.0, 0.7 0.2 0.1 0.0 ];

camera "red"
  output "rgb" "tif" "environment_6.tif"
  focal 1.5
  aperture 1.5
  aspect 1
  resolution 300 300
  environment = "red_sunset"
  end camera

material "corinthian_material"
  "specular_reflection" 1
  environment = "pink_sunset"
  end material
```

Figure 21.33: Color curves specified in the scene file for shader color_ramp

Figure 21.34: Different environment shaders used for the camera and object
21.6 Environment shaders for cameras and objects

Different environment shaders can be used for the camera and object instances. For any object instance without an environment shader, the camera's environment shader will be used as the default.

```plaintext
shader "red_sunset"
  "color_ramp"
  "colors" { 0.3 0.2 0.1 0.0, 0.1 0.05 0.0 0.49, 0.4 0.3 0.5 0.5, 1.0 1.0 0.0 0.51, 1.0 0.2 0.0 0.55, 0.1 0.2 0.6 0.7, 0.12 0.58 0.2 1.0 } }

shader "pink_sunset"
  "color_ramp"
  "colors" { 0.3 0.2 0.1 0.0, 0.1 0.05 0.0 0.49, 0.4 0.3 0.5 0.5, 1.0 1.0 0.0 0.51, 1.0 0.2 0.0 0.55, 0.1 0.2 0.6 0.7, 0.05 0.1 0.5 1.0 } }

camera "cam"
  output "rgb" "tif" "environment_6.tif"
  focal 1.5
  aperture 1.5
  aspect 1
  resolution 300 300
  environment = "red_sunset"
end camera

material "corinthian_material"
  "specular_reflection" 0
  environment = "pink_sunset"
end material
```

Figure 21.34: Different environment shaders used for the camera and object
shader "red_sunset"
  "color_ramp" (  
  "colors" [ 0.3 0.2 0.1 0.0,  
            0.1 0.05 0.0 0.49,  
            0.4 0.3 0.0 0.5,  
            1.0 1.0 0.0 0.51,  
            1.0 0.2 0.0 0.55,  
            0.1 0.2 0.6 0.7,  
            0.12 0.05 0.2 1.0 ] )

shader "pink_sunset"
  "color_ramp" (  
  "colors" [ 0.3 0.2 0.1 0.0,  
            0.1 0.05 0.0 0.49,  
            0.4 0.3 0.1 0.50,  
            1.0 1.0 0.8 0.51,  
            1.0 0.6 0.4 0.55,  
            0.1 0.2 0.6 0.7,  
            0.05 0.1 0.2 1.0 ] )

camera "cam"
  output "rgba" "tif" "environment_6.tif"
  focal 1.5
  aperture 1.5
  aspect 1
  resolution 300 300
  environment  
    = "red_sunset"
end camera

material "corinthian_material"
  "specular_reflection" ()
  environment = "pink_sunset"
end material

Figure 21.34: Different environment shaders used for the camera and object
Figure 21.34: Different environment shaders used for the camera and object
Cross-referencing in the shader book

The picture index in Appendix A provides pointers back to the section in which the rendered picture was discussed.
Appendix A

Rendered scene files

This appendix contains rendered images from the scene files used as examples throughout the book labeled with the page number on which the image appears.

Chapter 5 – A single color

Chapter 6 – Color from orientation
is transmitted in line 16, the incoming energy is attenuated by the transparency parameter in lines 12–13. The refraction direction in lines 14–15 is calculated by `ni_refraction_dir` in the same manner as the refraction shaders of Chapter 15. Here, however, we’re not sampling the scene with a new ray, but sending a photon in the refraction direction to represent the transmission of light energy.

Adding global illumination brightens the lower parts of the objects as light from the caustics are now included in the total light calculation.

```plaintext
options 'opt'
  object space
  contrast .1 .1 .1
  samples 4
  shadow on
display on
  max_displace .2
globallum on
globallum accuracy 500 1
  caustics on
causitic accuracy 500 .1
end options

material 'caustic'
  'specular_refraction'
  'index_of_refraction' 1.5
  'displace'
    'displace_ripple'
      'center' .2 .5 0
      'frequency' 20
      'amplitude' .01
    'displace_ripple'
      'center' 0 0 0
      'frequency' 20
      'amplitude' .01
    'displace_ripple'
      'center' .8 .8 0
      'frequency' 20
      'amplitude' .01
photon
  'transmit_specular_photons'
  'index_of_refraction' 1.5
end material

Figure 16.28: Caustics and global illumination used together

16.5 Visualizing the photon map

In Chapter 6, "Color from orientation," we made a shader that converted the surface normal into a color. A vector isn’t a color, after all, but by treating it as one we are able to visualize the orientation of the surface. This could be a very useful technique when we are checking for possible problems in the construction of our geometric models.

In a similar vein, we can set the global illumination options to values that would not be useful for producing final imagery, but can help us understand the way that the photon tracing process distributes photons to construct the photon map for its use in rendering.

In the photon tracing phase, the energy of a light emitting photons is divided up equally among
Cross-referencing in the shader book

The textual index includes references to shader and function names.
Index

Symbols
+ (pixel interpolation), 422
= 1 (shader reference), 50
# (comment character), 29

A
altitude, 140
ambient light, 151
ambient occlusion, 28, 220, 238
angle of incidence, 183
of reflection, 183
of view, 199
anisotropic, 160
API library functions, 51, 52, 61
ni_api.hair_begin, 294–296, 300, 301, 305–
307, 312, 313, 321, 322, 323, 328, 331
ni_api.hair_end, 294–296, 300, 301, 305–
307, 312, 313, 321, 322, 323, 328, 331
ni_api.hair.hairs.begin, 294–296, 300,
301, 305–307, 312, 313, 321, 322, 325
ni_api.hair.hairs.end, 294–296, 300,
301, 305–307, 312, 313, 321, 322, 325
ni_api.hair.info, 294–296, 300, 301,
305–307, 312, 313, 321, 322, 325
ni_api.hair.scale.begin, 294–296, 300,
301, 305–307, 312, 313, 321, 322, 325
ni_api.hair.scale.end, 294–296, 300,
301, 305–307, 312, 313, 321, 322, 325
ni_api.incremental, 294–296, 300, 301,
305–307, 312, 313, 321, 322, 325, 328, 331
ni_api.instance_begin, 286
ni_api.instance_end, 286
ni_api.object_begin, 274, 281
ni_api.object_end, 274, 294–296, 300, 301,
305–307, 312, 313, 321, 322, 325, 328, 331
ni_api.object_group_begin, 274, 281
ni_api.object_group_end, 274, 281
ni_api.poly.begin.rag, 274
ni_api.poly.end, 274
ni_api.poly_index.add, 274
ni_api.poly_index.add.rg, 274
ni_api.poly.project, 274
ni_api.poly.normal.add, 274
ni_api.poly.text.add, 274
ni_api_render_pass, 386
ni_api_render_pass, 386
ni_composite_add_radiance, 221, 225,
229
ni_cpu_info, 50, 51, 68
ni_debug, 320, 321, 347–349
ni_fbo, 424, 428, 430
ni_flush_cache, 267
ni_getShaderAddResult, 274, 281, 284,
286
ni_getContourLine, 102
ni_image_get.color, 441, 450
ni_image_get.color, 441, 445, 450
ni_lookup.color.texture, 84, 85, 254,
267
ni_matrix_copy, 286
ni_matrix_identity, 286
ni_matrix_invert, 286
ni_matrix_rotate, 285
ni_mem_allocate, 294–296, 300, 301, 305–
307, 312, 313, 321, 322, 323, 328, 331,
343, 344, 445
ni_mem_release, 321, 322, 325, 343, 344,
445
ni_opacity.set, 70, 74, 179, 308
ni_output_image.clone, 441, 445, 450
ni_output_image.open, 441, 445, 450
ni_par.abort, 441, 445, 450
ni_phasic.clone, 286
shader source code
add_colors, 431, 580
ambient occlusion, 240, 526
ambient occlusion_cutoff, 242, 527
annotate, 450, 583
average_radiance, 225, 523
average_radiance_options, 229, 524
blinn, 158, 502
bump_ripple, 264, 532
bump_texture, 267, 534
bump_wave, 261, 531
bump_wave_uv, 262, 532
c_contour, 101, 488
c_contrast, 99, 487
c_output, 102, 488
c_store, 97, 487
c_tessellate, 107, 490
c_toon, 115, 493
channel_ramp, 347–349, 558
chrome_ramp, 343, 344, 557
color_ramp, 352, 353, 560
cook_torrance, 159, 503
<table>
<thead>
<tr>
<th>Index references to shader source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>shader source code</td>
</tr>
<tr>
<td>add_colors, 431, 580</td>
</tr>
<tr>
<td>ambient_occlusion, 240, 526</td>
</tr>
<tr>
<td>ambient_occlusion_cutoff, 242, 527</td>
</tr>
<tr>
<td>annotate, 450, 583</td>
</tr>
<tr>
<td>average_radiance, 225, 523</td>
</tr>
<tr>
<td>average_radiance_options, 229, 524</td>
</tr>
<tr>
<td>blinn, 158, 502</td>
</tr>
<tr>
<td>bump_ripple, 264, 532</td>
</tr>
<tr>
<td>bump_texture, 267, 534</td>
</tr>
<tr>
<td>bump_wave, 261, 531</td>
</tr>
<tr>
<td>bump_wave_uv, 262, 532</td>
</tr>
<tr>
<td>c_contour, 101, 488</td>
</tr>
<tr>
<td>c_contrast, 99, 487</td>
</tr>
<tr>
<td>c_output, 102, 488</td>
</tr>
<tr>
<td>c_store, 97, 487</td>
</tr>
<tr>
<td>c_tessellate, 107, 490</td>
</tr>
<tr>
<td>c_toon, 115, 493</td>
</tr>
<tr>
<td>channel_ramp, 347–349, 558</td>
</tr>
<tr>
<td>chrome_ramp, 343, 344, 557</td>
</tr>
<tr>
<td>color_ramp, 352, 353, 560</td>
</tr>
<tr>
<td>cook_torrance, 159, 503</td>
</tr>
</tbody>
</table>
shader source code
  add_colors, 431, 580
  ambient_occlusion, 240, 526
  ambient_occlusion_cutoff, 242, 527
  annotate, 450, 583
  average_radiance, 225, 523
  average_radiance_options, 229, 524
  blinn, 158, 502
  bump_ripple, 264, 532
  bump_texture, 267, 534
  bump_wave, 261, 531
  bump_wave_uv, 262, 532
  c_contour, 101, 488
  c_contrast, 99, 487
  c_output, 102, 488
  c_store, 97, 487
  c_tessellate, 107, 490
  c_toon, 115, 493
  channel_ramp, 347–349, 558
  chrome_ramp, 343, 344, 557
  color_ramp, 352, 353, 560
  cook_torrance, 159, 503
state
commonly used fields, 57
conceptual categories, 57
fields in mState
  state->bary, 60, 111, 292
  state->bary[1], 292, 315
  state->camera, 57
  state->camera->x_resolution,
    406
  state->camera->y_resolution,
    406
  state->child, 60
  state->derivs[0], 316
  state->dir, 60, 61, 133, 136, 155, 340, 358,
    362, 370, 400, 402, 407, 416, 418
  state->dist, 60, 146, 147, 358, 359, 366,
    370, 374
  state->dot_nd, 58–61, 114, 164
  state->environment, 60
  state->face, 60
  state->instance, 60
  state->inv_normal, 60
  state->ior, 60, 202
  state->ior_in, 60, 202
  state->label, 60
  state->light_instance, 60, 135
  state->material, 60
  state->normal, 60, 61, 155, 248, 260, 261,
    263, 265, 266, 319
API library functions, 51, 52, 61
  mi_api_hair_begin, 294–296, 300, 301, 305–307, 312, 313, 321, 322, 325, 328, 331
  mi_api_hair_end, 294–296, 300, 301, 305–307, 312, 313, 321, 322, 325, 328, 331
  mi_api_hair_hairs_begin, 294–296, 300, 301, 305–307, 312, 313, 321, 322, 325
  mi_api_hair_hairs_end, 294–296, 300, 301, 305–307, 312, 313, 321, 322, 325
  mi_api_hair_info, 294–296, 300, 301, 305–307, 312, 313, 321, 322, 325
  mi_api_hair_scalars_begin, 294–296, 300, 301, 305–307, 312, 313, 321, 322, 325
  mi_api_hair_scalars_end, 294–296, 300, 301, 305–307, 312, 313, 321, 322, 325
  mi_api_incremental, 294–296, 300, 301, 305–307, 312, 313, 321, 322, 325, 328, 331
  mi_api_instance_begin, 286
  mi_api_instance_end, 286
  mi_api_object_begin, 274, 281
U
unit density, 361
unit square, 12
utility functions
miaux_add_blinn_specular_component, 157, 592
miaux_add_color, 593
Website support for the book’s software
Website support for the book’s software

http://www.writingshaders.com/
Writing mental ray shaders

A perceptual introduction

This website provides additional resources for the book *Writing mental ray shaders: A perceptual introduction*. It includes all the shader source code and scene files described in the book, available for you to download. These pages will also help you understand where and how the shaders are used in the example scenes throughout the book. In the page for a shader, the scenes that use it are linked; conversely, in the page for a scene, the shaders used by that scene are linked. The book’s bibliography is also duplicated here, but with the addition of links to the original papers and books that it references. I am also adding material beyond what is covered in the book to this site, including teaching materials as well as additional shaders that extend ideas presented in the book or based on new shader ideas suggested by readers. The book includes a CDROM that contains a complete version of mental ray.

Background
- Getting started
- The book’s CDROM
- Downloading examples from the book

Shaders
- Shader compilation on various platforms
- Shader source files
- Additional shaders
- A few notes on programming style

Scenes
- Accessing custom shaders during rendering
- Scene files

Reference
- Teaching materials
- Bibliography
Website support for the book’s software

The “Background” section is an overview of the resources provided by the website and how you can use them.
Getting started

The good news is that mental ray shaders are written in the C or C++ programming languages. Unfortunately, the bad news is that … mental ray shaders are written in C or C++. But let’s focus on the good part: Since the shaders are in C or C++, everything you know about programming in those languages, and all the libraries that you’ve written or have acquired, can potentially be useful in mental ray shader programming. However, the way that you compile C/C++ code varies in its details across different operating systems, and getting this right can be a frustrating hurdle right at the beginning. The installation of mental ray and your custom shaders are also dependent upon the structure of the file system. I’ve included a lot of information in this website to deal with these issues. I hope this page will help clarify the big picture so that you don’t get lost in the details right off the bat.

Compiling shaders

The page “Shader compilation on various platforms” describes the various pieces required to compile the shaders in the book.

The book organizes the shader code in a simple way: each shader is almost always defined in its own file. (I say “almost always” because contour shading is divided into four processes, each represented by a separate shader. I’ve organized one set of related contours shaders in a single file in Chapter 10 of the c_tessellate shader.)

Many shaders also use utility functions from the “miaux” library, written for the book and designed as an auxiliary set to complement the “mi” library supplied with mental ray.

I also defined a few simple shapes as geometry shaders in file “newblocks.cpp”. The “newblocks” shaders depend up a library of C++ classes called “mrpoly.” These classes are an initial sketch of how you can approach geometry shaders at a higher level through class design. The low-level
**Downloading examples from the book**

The shaders and scenes in *Writing mental ray shaders* can be examined and downloaded individually in the shaders and scenes pages. For convenience, you can download all the files in a single zip file.

| Shader code and scene files | WMRS_source_april_2008.zip | Directory contents |

The zip file expands to a directory that contains two subdirectories, shaders and scenes. This directory structure is also used in the training classes offered by mental images.

For information on shader compilation, see “Shader compilation on various platforms.” For information on using custom shaders, see “Accessing custom shaders during rendering.”

25 April 2008 00:21:11
Downloading examples from the book

The shaders and scenes in *Writing mental ray shaders* can be examined and downloaded individually in the shaders and scenes pages. For convenience, you can download all the files in a single zip file.

- Shader code and scene files
- WMRS_source_april_2008.zip
- Directory contents

The zip file expands to a directory that contains two subdirectories, shaders and scenes. This directory structure is also used in the training classes offered by mental images.

For information on shader compilation, see “Shader compilation on various platforms.” For information on using custom shaders, see “Accessing custom shaders during rendering.”

25 April 2008 09:21:11
<table>
<thead>
<tr>
<th>Length</th>
<th>Date</th>
<th>Time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>04-22-08</td>
<td>23:52</td>
<td>WMRS_source_april_2008/</td>
</tr>
<tr>
<td>0</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/</td>
</tr>
<tr>
<td>2012</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/ambocclude_1.mi</td>
</tr>
<tr>
<td>2013</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/ambocclude_2.mi</td>
</tr>
<tr>
<td>2063</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/ambocclude_3.mi</td>
</tr>
<tr>
<td>5135</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/atmosphere_1.mi</td>
</tr>
<tr>
<td>5273</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/atmosphere_2.mi</td>
</tr>
<tr>
<td>5272</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/atmosphere_3.mi</td>
</tr>
<tr>
<td>5620</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/atmosphere_4.mi</td>
</tr>
<tr>
<td>5672</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/atmosphere_5.mi</td>
</tr>
<tr>
<td>5679</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/atmosphere_6.mi</td>
</tr>
<tr>
<td>5682</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/atmosphere_7.mi</td>
</tr>
<tr>
<td>12591274</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/bubbles.tif</td>
</tr>
<tr>
<td>2901</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/buffer_1.mi</td>
</tr>
<tr>
<td>3175</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/buffer_2.mi</td>
</tr>
<tr>
<td>3710</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/buffer_3.mi</td>
</tr>
<tr>
<td>4344</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/buffer_4.mi</td>
</tr>
<tr>
<td>6080</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/buffer_5.mi</td>
</tr>
<tr>
<td>1310</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/bump_1.mi</td>
</tr>
<tr>
<td>1367</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/bump_2.mi</td>
</tr>
<tr>
<td>1334</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/bump_3.mi</td>
</tr>
<tr>
<td>1450</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/bump_4.mi</td>
</tr>
<tr>
<td>1374</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/bump_5.mi</td>
</tr>
<tr>
<td>1582</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/bump_7.mi</td>
</tr>
<tr>
<td>3284</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/caustics_1.mi</td>
</tr>
<tr>
<td>3882</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/caustics_2.mi</td>
</tr>
<tr>
<td>3928</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/caustics_3.mi</td>
</tr>
<tr>
<td>135464</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/colorstrip.tif</td>
</tr>
<tr>
<td>19376</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_10.mi</td>
</tr>
<tr>
<td>20316</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_11.mi</td>
</tr>
<tr>
<td>19324</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_12.mi</td>
</tr>
<tr>
<td>19750</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_13.mi</td>
</tr>
<tr>
<td>20559</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_14.mi</td>
</tr>
<tr>
<td>20580</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_15.mi</td>
</tr>
<tr>
<td>20586</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_16.mi</td>
</tr>
<tr>
<td>46560</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_2.mi</td>
</tr>
<tr>
<td>46766</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_3.mi</td>
</tr>
<tr>
<td>47048</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_4.mi</td>
</tr>
<tr>
<td>50416</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_5.mi</td>
</tr>
<tr>
<td>50410</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_6.mi</td>
</tr>
<tr>
<td>50565</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_7.mi</td>
</tr>
<tr>
<td>18942</td>
<td>04-22-08</td>
<td>22:45</td>
<td>WMRS_source_april_2008/scenes/contours_8.mi</td>
</tr>
</tbody>
</table>
Website support for the book’s software

The “Shaders” section contains a catalog of all the shaders in the book, along with instructions for compilation on various platforms.
Shader source files

This page contains links to the the source code for all the shaders in *Writing mental ray shaders*, organized by chapter. The source code can also be downloaded from a single ZIP file described in the “Downloading examples from the book” page.

Besides the source code for all the shaders, the ZIP file also contains a shader library called “newblocks” that contains geometry shaders used in many of the scenes in the book. (These shaders are used for the construction of scene objects and are not part of the book’s discussion of geometry shaders. I plan to include techniques for procedural object construction in the “Additional shaders” page.)

The shader catalog

All the shaders of the book are listed below by chapter, divided by the five major parts in the book. For each shader page, the declaration in .mi syntax is listed first, followed by the full C source code. The declaration and C code are themselves links to an unformatted file that can be downloaded individually. You can also copy and paste individual sections of those pages.

If the shader contains miaux auxiliary functions (as introduced in Chapter 7 with shader `depth_fade_tint_2`), they are listed after the shader source code. For convenience, the miaux functions used in the shader are contained in a separate file that for which the page also provides a link. However, all miaux utility functions are declared together in `miaux.h` and defined in `miaux.c`.

To see all the scene files in which a shader is used, click on the “Scenes” link in the upper right corner of the shader source code page. For example, shader `one_color` is used in many scenes in the book, as you can see in this page.

A description of shader compilation is contained in the page “Shader compilation on various platforms.” The `DLLEXPORT` macro is declared in the mental ray header file `shader.h`. This macro is is required for compilation on Microsoft “Windows,” but is ignored during compilation under other operating systems.

**Part 2: Color**

Chapter 5: A single color

   one_color

Chapter 6: Color from orientation

   front_bright
   front_bright_dot
   normals_as_colors

Chapter 7: Color from position
Chapter 8: The transparency of a surface

transparent
transparent_modularized

Chapter 9: Color from functions

show_uv
show_uv_steps
texture_uv_simple
texture_uv
vertex_color
summed_noise_color

Chapter 10: The color of edges

c_store
c_contrast
c_contour
c_output
c_tessellate
show_barycentric
front_bright_steps
c_toon
lambert_steps

Part 3: Light

Chapter 11: Lights

point_light
point_light_shadow
spotlight
soft_spotlight
sinusoid_soft_spotlight
point_light_falloff

Chapter 12: Light on a surface
Chapter 8: The transparency of a surface

transparent
transparent_modularized

Chapter 9: Color from functions

show_uv
show_uv_steps
texture_uv_simple
texture_uv
vertex_color
summed_noise_color

Chapter 10: The color of edges

c_store
c_contrast
c_contour
c_output
c_tessellate
show_barycentric
front_bright_steps
c_toon
lambert_steps

Part 3: Light

Chapter 11: Lights

point_light
point_light_shadow
spotlight
soft_spotlight
sinusoid_soft_spotlight
point_light_falloff

Chapter 12: Light on a surface
Shader point_light_shadow

Click on the filename to display or download the file.

point_light_shadow.mi

declare shader
    color "point_light_shadow" (   
        color "light_color" default 1 1 1 )
version 1
apply light
end declare

point_light_shadow.c

#include "shader.h"

struct point_light_shadow {
    miColor light_color;
};

DLLEXPORT
int point_light_shadow_version(void) { return 1; } 

DLLEXPORT
miBoolean point_light_shadow (  
    miColor *result, miState *state, struct point_light_shadow *params )
{
    *result = *mi_eval_color(&params->light_color);
    return mi_trace_shadow(result, state);
}
Shader point_light_shadow

Click on the filename to display or download the file.

point_light_shadow.mi

declare shader
    color "point_light_shadow" (    
        color "light_color" default 1 1 1 )
version 1
apply light
end declare

point_light_shadow.c

#include "shader.h"

struct point_light_shadow {
    miColor light_color;
};

DLLEXPORT
int point_light_shadow_version(void) { return 1; }

DLLEXPORT
miBoolean point_light_shadow (   
    miColor *result, miState *state, struct point_light_shadow *params )
{
    *result = *mi_eval_color(&params->light_color);
    return mi_trace_shadow(result, state);
}
Scenes using shader point_light_shadow

Click on the chapter title to see all the scenes in that chapter. Click on a filename to display or download that scene file.

Chapter 10: The color of edges
    contours_4.mi
    contours_15.mi
    contours_16.mi

Chapter 11: Lights
    lights_4.mi
    lights_8.mi

Chapter 12: Light on a surface
    shading_1.mi
    shading_2.mi
    shading_3.mi
    shading_4.mi
    shading_5.mi
    shading_6.mi
    shading_7.mi

Chapter 13: Shadows
    shadows_1.mi
    shadows_2.mi
    shadows_A.mi
    shadows_B.mi
    shadows_C.mi
    shadows_C1.mi
    shadows_D.mi
    shadows_E.mi
    shadows_F.mi
    shadows_G.mi

Chapter 19: Creating geometric objects
Shader lambert

Click on the filename to display or download the file.

lambert.mi

declare shader
  color "lambert" {
    color "ambient" default 0 0 0,
    color "diffuse" default 1 1 1,
    array light "lights" )
  version 1
  apply material
end declare

lambert.c

#include "shader.h"
#include "miaux.h"

struct lambert {
  miColor ambient;
  miColor diffuse;
  int i_light;
  int n_light;
  miTag light[1];
};

DLLEXPORT
int lambert_version(void) { return 1; }

DLLEXPORT
miBoolean lambert (
  miColor *result, miState *state, struct lambert *params )
{
  int i, light_count, light_sample_count;
  miColor sum, light_color;
  miScalar dot_n_l;
  miTag *light;

  miColor *diffuse = mi_eval_color(&params->diffuse);
  miaux_light_array(&light, &light_count, state,
    &params->i_light, &params->n_light, params->light);
  *result = *mi_eval_color(&params->ambient);

  for (i = 0; i < light_count; i++, light++) {
  ...
Shader lambert

Click on the filename to display or download the file.

---
lambert.mi

declare shader
    color "lambert" {
        color "ambient" default 0 0 0,
        color "diffuse" default 1 1 1,
        array light "lights" )
    version 1
    apply material
end declare

---
lambert.c

#include "shader.h"
#include "miaux.h"

struct lambert {
    miColor ambient;
    miColor diffuse;
    int i_light;
    int n_light;
    miTag light[1];
};

DLLEXPORT
int lambert_version(void) { return 1; }

DLLEXPORT
miBoolean lambert (miColor *result, miState *state, struct lambert *params)
{
    int i, light_count, light_sample_count;
    miColor sum, light_color;
    miScalar dot_n_l;
    miTag *light;

    miColor *diffuse = mi_eval_color(&params->diffuse);
    miaux_light_array(&light, &light_count, state,
        &params->i_light, &params->n_light, params->light);
    *result = *mi_eval_color(&params->ambient);

    for (i = 0; i < light_count; i++, light++)
    {

#include "shader.h"
#include "miaux.h"

struct lambert {
  miColor ambient;
  miColor diffuse;
  int i_light;
  int n_light;
  miTag light[1];
};

DLLEXPORT
int lambert_version(void) { return 1; }

DLLEXPORT
miBoolean lambert(
  miColor *result, miState *state, struct lambert *params)
{
  int i, light_count, light_sample_count;
  miColor sum, light_color;
  miScalar dot_nl;
  miTag *light;

  miColor *diffuse = mi_eval_color(&params->diffuse);
  miaux_light_array(&light, &light_count, state, &params->i_light, &params->n_light, params->light);
  *result = *mi_eval_color(&params->ambient);

  for (i = 0; i < light_count; i++, light++) {
    miaux_set_channels(&sum, 0);
    light_sample_count = 0;
    while (mi_sample_light(&light_color, NULL, &dot_nl, state, *light, &light_sample_count))
      miaux_add_diffuse_component(&sum, dot_nl, diffuse, &light_color);
    if (light_sample_count)
      miaux_add_scaled_color(result, &sum, 1.0/light_sample_count);
  }
  return miTRUE;
}
Shader lambert

Click on the filename to display or download the file.

lambert.mi

declare shader
    color "lambert" {
        color "ambient" default 0 0 0,
        color "diffuse" default 1 1 1,
        array light "lights" )
    version 1
    apply material
end declare

lambert.c

#include "shader.h"
#include "miaux.h"

struct lambert {
    miColor ambient;
    miColor diffuse;
    int    i_light;
    int    n_light;
    miTag   light[l];
};

DLLEXPORT
int lambert_version(void) { return 1; } 

DLLEXPORT
miBoolean lambert ( 
    miColor *result, miState *state, struct lambert *params )
{
    int i, light_count, light_sample_count;
    miColor sum, light_color;
    miScalar dot_n_l;
    miTag *light;

    miColor *diffuse = mi_eval_color(&params->diffuse);
    miaux_light_array(&light, &light_count, state,
        &params->i_light, &params->n_light, params->light);
    *result = *mi_eval_color(&params->ambient);

    for (i = 0; i < light_count; i++, light++){

Shader lambert

Click on the filename to display or download the file.

```
# lambert.mi

declare shader
  color "lambert" {
    color "ambient" default 0 0 0,
    color "diffuse" default 1 1 1,
    array light "lights"
  } version 1
end declare

# lambert.c

#include "shader.h"
#include "miaux.h"

struct lambert {
  miColor ambient;
  miColor diffuse;
  int i_light;
  int n_light;
  miTag light[1];
};

DLLEXPORT
int lambert_version(void) { return 1; }

DLLEXPORT
miBoolean lambert (miColor *result, miState *state, struct lambert *params )
{
  int i, light_count, light_sample_count;
  miColor sum, light_color;
  miScalar dot_nl;
  miTag *light;

  miColor *diffuse = mi_eval_color(&params->diffuse);
  miaux_light_array(&light, &light_count, state, &params->i_light, &params->n_light, params->light);
  result = mi_eval_color(&params->ambient);
  for (i = 0; i < light_count; i++, light++) {
```

Shader source for lambert

```c

*result = *mi_eval_color(&params->ambient);

for (i = 0; i < light_count; i++, light++) {
    miaux_set_channels(&sum, 0);
    light_sample_count = 0;
    while (mi_sample_light(light_color, NULL, &dot_nl,
                            state, *light, &light_sample_count))
        miaux_add_diffuse_component(&sum, dot_nl, diffuse, &light_color);
    if (light_sample_count)
        miaux_add_scaled_color(result, &sum, 1.0/light_sample_count);
}
return miTRUE;
}

lambert_util.c

void miaux_light_array(miTag **lights, int *light_count, miState *state,
                        int *offset_param, int *count_param, miTag *lights_param)
{
    int array_offset = *mi_eval_integer(offset_param);
    *light_count = *mi_eval_integer(count_param);
    *lights = mi_eval_tag(lights_param) + array_offset;
}

void miaux_set_channels(miColor *c, miScalar new_value)
{
    c->r = c->g = c->b = c->a = new_value;
}

void miaux_add_diffuse_component(
    miColorT *result,
    miScalar light_and_surface_cosine,
    miColor *diffuse, miColor *light_color)
{
    result->r += light_and_surface_cosine * diffuse->r * light_color->r;
    result->g += light_and_surface_cosine * diffuse->g * light_color->g;
    result->b += light_and_surface_cosine * diffuse->b * light_color->b;
}

void miaux_add_scaled_color(miColor *result, miColor *color, miScalar scale)
{
    result->r += color->r * scale;
    result->g += color->g * scale;
    result->b += color->b * scale;
}
```
void miaux_light_array(miTag **lights, int *light_count, miState *state, 
int *offset_param, int *count_param, miTag *lights_param)
{
    int array_offset = *mi_eval_integer(offset_param);
    *light_count = *mi_eval_integer(count_param);
    *lights = *mi_eval_tag(lights_param) + array_offset;
}

void miaux_set_channels(miColor *c, miScalar new_value)
{
    c->r = c->g = c->b = c->a = new_value;
}

void miaux_add_diffuse_component(
    miColorT *result,
    miScalar light_and_surface_cosine,
    miColor *diffuse, miColor *light_color)
{
    result->r += light_and_surface_cosine * diffuse->r * light_color->r;
    result->g += light_and_surface_cosine * diffuse->g * light_color->g;
    result->b += light_and_surface_cosine * diffuse->b * light_color->b;
}

void miaux_add_scaled_color(miColor *result, miColor *color, miScalar scale)
{
    result->r += color->r * scale;
    result->g += color->g * scale;
    result->b += color->b * scale;
}
Writing mental ray shaders

A perceptual introduction

This website provides additional resources for the book *Writing mental ray shaders: A perceptual introduction*. It includes all the shader source code and scene files described in the book, available for you to download. These pages will also help you understand where and how the shaders are used in the example scenes throughout the book. In the page for a shader, the scenes that use it are linked; conversely, in the page for a scene, the shaders used by that scene are linked. The book’s bibliography is also duplicated here, but with the addition of links to the original papers and books that it references. I am also adding material beyond what is covered in the book to this site, including teaching materials as well as additional shaders that extend ideas presented in the book or based on new shader ideas suggested by readers. The book includes a CDROM that contains a complete version of mental ray.

Background

- Getting started
- The book’s CDROM
- Downloading examples from the book

Shaders

- Shader compilation on various platforms
- Shader source files
- Additional shaders
- A few notes on programming style

Scenes

- Accessing custom shaders during rendering
- Scene files

Reference

- Teaching materials
- Bibliography
Writing mental ray shaders

A perceptual introduction

This website provides additional resources for the book *Writing mental ray shaders: A perceptual introduction*. It includes all the shader source code and scene files described in the book, available for you to download. These pages will also help you understand where and how the shaders are used in the example scenes throughout the book. In the page for a shader, the scenes that use it are linked; conversely, in the page for a scene, the shaders used by that scene are linked. The book’s bibliography is also duplicated here, but with the addition of links to the original papers and books that it references. I am also adding material beyond what is covered in the book to this site, including teaching materials as well as additional shaders that extend ideas presented in the book or based on new shader ideas suggested by readers. The book includes a CDROM that contains a complete version of mental ray.

Background
- Getting started
- The book’s CDROM
- Downloading examples from the book

Shaders
- Shader compilation on various platforms
- Shader source files
- Additional shaders
- A few notes on programming style

Scenes
- Accessing custom shaders during rendering
- Scene files

Reference
- Teaching materials
- Bibliography
Chapter 25 demonstrates how rendering components can be saved into separate files using framebuffers. This page describes the simplified scene file syntax for the definition of framebuffers introduced in mental ray version 3.6 as well as the use of geometry shaders in framebuffer definition.

A review of framebuffer use prior to version 3.6

In releases of mental ray prior to 3.6, framebuffers defined by the user are identified by integer indices, and named simply by preceding the framebuffer index with the string “fb”. These names are defined in the options block of the scene, for example, in lines 7-9 in this set of options statements from scene file buffer_5.mi from Chapter 25.

```
1 options "opt"
2 object space
3 contrast .1 .1 .1 1
4 samples 0 2
5 finalgather on
6 finalgather accuracy 50 2 .5
7 frame buffer 0 "rgba"
8 frame buffer 1 "rgba"
9 frame buffer 2 "rgba"
10 end options
```

During rendering, shaders can access framebuffers using the API functions mi_fb_put() and mi_fb_get(). For example, Chapter 25 defines the shader `framebuffer_put` that simply passes along the color in the `result` pointer, but with the side effect of storing the color in a framebuffer with `mi_fb_put()` in line 7:

```
1 miBoolean framebuffer_put(
2 miColor *result, miState *state, struct framebuffer_put *params)
3 {
4   *result = *mi_eval_color(&params->color);
5   if (state->type == miRAY_EYE)
6     mi_fb_put(state, *mi_eval_integer(&params->index), result);
7   return miTRUE;
8 }
```

When rendering is done, framebuffers are written to the files defined by “output statements” in the camera:
Website support for the book’s software

The “Scenes” section contains a rendered scene for all the examples in the book.
Scene files

All the scene files for *Writing mental ray shaders* may be downloaded from the links below, organized by chapter. You can also download the entire set of scene files and the other files they reference (textures, objects, particle datasets, and voxel datasets) through the “Downloading examples from the book” page.

The zip-compressed file contains within it other zip files; the fully uncompressed set of files is quite large (due primarily to the voxel data files from Chapter 23). The filenames of the compressed files all end with a .zip extension.

You can also download individual scenes from the links below. The additional files referenced by the scenes are listed at the beginning of each chapter. Several of the files (the textures, for example) are used throughout the book, but they are listed for each chapter in which they are used for reference.

**Part 1: Structure**

Chapter 4: Shaders in the scene

**Part 2: Color**

Chapter 5: A single color
Chapter 6: Color from orientation
Chapter 7: Color from position
Chapter 8: The transparency of a surface
Chapter 9: Color from functions
Chapter 10: The color of edges

**Part 3: Light**

Chapter 11: Lights
Chapter 12: Light on a surface
Chapter 13: Shadows
Chapter 14: Reflection
Chapter 15: Refraction
Chapter 16: Light from other surfaces

**Part 4: Shape**
Scene files

All the scene files for *Writing mental ray shaders* may be downloaded from the links below, organized by chapter. You can also download the entire set of scene files and the other files they reference (textures, objects, particle datasets, and voxel datasets) through the “Downloading examples from the book” page.

The zip-compressed file contains within it other zip files; the fully uncompressed set of files is quite large (due primarily to the voxel data files from Chapter 23). The filenames of the compressed files all end with a .zip extension.

You can also download individual scenes from the links below. The additional files referenced by the scenes are listed at the beginning of each chapter. Several of the files (the textures, for example) are used throughout the book, but they are listed for each chapter in which they are used for reference.

**Part 1: Structure**

Chapter 4: Shaders in the scene

**Part 2: Color**

Chapter 5: A single color
Chapter 6: Color from orientation
Chapter 7: Color from position
Chapter 8: The transparency of a surface
Chapter 9: Color from functions
Chapter 10: The color of edges

**Part 3: Light**

Chapter 11: Lights
Chapter 12: Light on a surface
Chapter 13: Shadows
Chapter 14: Reflection
Chapter 15: Refraction
Chapter 16: Light from other surfaces

**Part 4: Shape**
Chapter 11: Lights

- **point_light**
  - `lambert`
  - `point_light`

- **point_light**
  - `lambert`
  - `point_light`

- **one_color**
  - `lambert`
  - `one_color`

- **point_light_shadow**
  - `lambert`
  - `point_light_shadow`
# mental ray scene file from "Writing mental ray shaders"
# http://www.writingshaders.com/scene_catalog.html

verbose on

link "lambert.so"
$include "lambert.mi"
link "point_light.so"
$include "point_light.mi"

options "opt"
    object space
    contrast 1.1 1.1 1
    samples 0 2
end options

light "white_light"
    "point_light" ()
    origin 2 2 5
end light

instance "light_instance"
    "white_light"
end instance

material "brown" opaque
    "lambert" {
        "diffuse" 1 .95 .7,
        "lights" ["light_instance"]
    }
end material

material "red" opaque
    "lambert" {
        "diffuse" 1 .4 .4,
        "lights" ["light_instance"]
    }
end material

material "yellow" opaque
    "lambert" {
        "diffuse" 1 1 .4,
        "lights" ["light_instance"]
    }
end material

material "blue" opaque
    "lambert" {
        "diffuse" .4 .4 1,
        "lights" ["light_instance"]
    }
end material
Chapter 11: Lights

lights_1.mi

lights_2.mi

lights_3.mi
Shader lambert

Click on the filename to display or download the file.

lambert.mi

declare shader
  color "lambert" {
    color "ambient" default 0 0 0,
    color "diffuse" default 1 1 1,
    array light "lights"
  }
version 1
apply material
end declare

lambert.c

#include "shader.h"
#include "miaux.h"

struct lambert {
  miColor ambient;
  miColor diffuse;
  int  i_light;
  int  n_light;
  miTag  light[1];
};

DLLEXPORT
int lambert_version(void) { return 1; }

DLLEXPORT
miBoolean lambert (  
  miColor *result, miState *state, struct lambert *params  )
{
  int i, light_count, light_sample_count;
  miColor sum, light_color;
  miScalar dot_n_l;
  miTag *light;

  miColor *diffuse = mi_eval_color(&params->diffuse);
  miaux_light_array(&light, &light_count, state,
                    &params->i_light, &params->n_light, params->light);
  *result = *mi_eval_color(&params->ambient);

  for (i = 0; i < light_count; i++, light++) {
    // Code to render the light
  }

// End of lambert function
Chapter 11: Lights

lambert
point_light

lights_1.mi

lambert
point_light

lights_2.mi

lambert
one_color

lights_3.mi

lambert
point_light_shadow
Chapter 11: Lights

lambert
point_light

lights_1.mi

lambert
point_light

lights_2.mi

lambert
one_color

lights_3.mi

lambert
point_light_shadow
Shader point_light

Click on the filename to display or download the file.

---

point_light.mi

declare shader
    color "point_light" (
        color "light_color" default 1 1 1
    )
version 1
apply light
end declare

---

point_light.c

#include "shader.h"

struct point_light {
    miColor light_color;
};

DLLEXPORT
int point_light_version(void) { return 1; }

DLLEXPORT
miBoolean point_light (
    miColor *result, miState *state, struct point_light *params)
{
    *result = *mi_eval_color(&params->light_color);
    return miTRUE;
}
Chapter 14: Reflection

Texture: octatile.tif

reflection_1.mi
- lambert
- one_color
- specular_reflection
- texture_uv
- point_light

reflection_2.mi
- lambert
- one_color
- specular_reflection
- texture_uv
- point_light

reflection_3.mi
- lambert
- one_color
- specular_reflection
- texture_uv
- point_light
Chapter 14: Reflection

Texture: octatile.tif

lambert
one_color
specular_reflection
texture_uv
point_light

reflection_1.mi

lambert
one_color
specular_reflection
texture_uv
point_light

reflection_2.mi

lambert
one_color
specular_reflection
texture_uv
point_light

reflection_3.mi
Chapter 14: Reflection

Texture: octatile.tif

reflection_1.mi
lambert
one_color
specular_reflection
texture_uv
point_light

reflection_2.mi
lambert
one_color
specular_reflection
texture_uv
point_light

reflection_3.mi
lambert
one_color
specular_reflection
texture_uv
point_light
Website support for the book’s software

The “Reference” section contains slides for use in teaching and links for the book’s bibliography.
Teaching materials

Slides for lectures

If you are a teacher and would like to use Writing mental ray shaders in the classroom, I have reformatted the book’s diagrams and code as a set of PDF files that can be used in classroom lectures.

<table>
<thead>
<tr>
<th></th>
<th>WMRS_part0_introduction_april_2008.pdf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>WMRS_part1_structure_april_2008.pdf</td>
</tr>
<tr>
<td>Color</td>
<td>WMRS_part2_color_april_2008.pdf</td>
</tr>
<tr>
<td>Light</td>
<td>WMRS_part3_light_april_2008.pdf</td>
</tr>
<tr>
<td>Shape</td>
<td>WMRS_part4_shape_april_2008.pdf</td>
</tr>
<tr>
<td>Space</td>
<td>WMRS_part5_space_april_2008.pdf</td>
</tr>
<tr>
<td>Image</td>
<td>WMRS_part6_image_april_2008.pdf</td>
</tr>
</tbody>
</table>

The lectures use the shaders and scene files from the book as well as some additional material you can download.

<table>
<thead>
<tr>
<th>Additional lecture resources</th>
<th>WMRS_lectures_extra_april_2008.zip</th>
</tr>
</thead>
</table>

Examples of the lecture slides

Using shaders in the scene file

*Anonymous shaders*
  Shader called directly in the material

*Named shaders*
  Shader defined for later reference

*Shader lists*
  Accumulation of shader results

*Shader graphs*
  Shaders used as input to other shaders

*Phenomena*
  Examples include light and shadow...
Examples of the lecture slides

Using shaders in the scene file
- **Anonymous shaders**
  - Shader called directly in the material
- **Named shaders**
  - Shader defined for later reference
- **Shader lists**
  - Accumulation of shader results
- **Shader graphs**
  - Shaders used as input to other shaders
- **Phenomena**
  - Formalized shader graphs

The environment of the scene

Environment shaders for cameras and objects

```plaintext
shader "end_moonset"
  "color_ramp" { 0.3 0.2 0.1 0.0, 0.1 0.05 0.0 0.45, 0.05 0.0 0.0 0.5, 0.0 0.0 0.0 0.5, 0.0 0.0 0.0 0.5, 1.0 0.2 0.0 0.55, 0.1 0.2 0.0 0.52, 0.1 0.2 0.0 0.5, 0.12 0.05 0.2 1.0 } }

shader "pink_moonset"
  "color_ramp" { 0.3 0.2 0.1 0.0, 0.1 0.05 0.0 0.45, 0.05 0.0 0.0 0.5, 0.0 0.0 0.0 0.5, 0.0 0.0 0.0 0.5, 1.0 0.2 0.0 0.55, 0.1 0.2 0.0 0.52, 0.1 0.2 0.0 0.5, 0.12 0.05 0.2 1.0 } }

camera "cam"
  output "rgba" "tif" "environment_4.tif"
  focal 1.5
  aperture 1.5
  aspect 1
  resolution 300 300
  environment "red_moonset"
end camera

material "corinthian_material"
  "specular_reflection" ()
  environment = "pink_moonset"
end material

Different environment shaders used for the camera and object

Rendering image components

Definition and use of frame buffers
Formalized shader graphs

The environment of the scene

Environment shaders for cameras and objects

The environment shaders used for the camera and object

Rendering image components

Definition and use of frame buffers

Interaction of the scene file and shaders in the use of frame buffers
Formalized shader graphs

**The environment of the scene**

Environment shaders for cameras and objects

```plaintext
shader "red_mood" {
  "color ramp" {
    colors { 0.3 0.2 0.1 0.9,
    0.1 0.05 0.6 0.49,
    0.4 0.3 0.9 0.9,
    1.0 1.0 0.0 0.51,
    1.0 0.2 0.0 0.55,
    0.1 0.2 0.4 0.7,
    0.12 0.05 0.2 1.0 } }
}

shader "pink_mood" {
  "color ramp" {
    colors { 0.3 0.2 0.1 0.9,
    0.1 0.05 0.6 0.49,
    0.4 0.3 0.9 0.9,
    1.0 1.0 0.0 0.51,
    1.0 0.2 0.0 0.55,
    0.1 0.2 0.4 0.7,
    0.12 0.05 0.2 1.0 } }
}

camera "cam"
  output "rgba" "lif" "environment_s.lif"
  focal 1.5
  aspect 1.1
  resolution 384 384
  environment = "red_mood"
end camera

material "corithian_material"
  "specular_reflection" ()
  environment = "pink_mood"
end material
```

Different environment shaders used for the camera and object

**Rendering image components**

Definition and use of frame buffers

1. Create frame buffers in memory
2. Get and put frame buffer pixels
3. Write frame buffers to file

Interaction of the scene file and shaders in the use of frame buffers
Formalized shader graphs

The environment of the scene

Environment shaders for cameras and objects

Different environment shaders used for the camera and object

Rendering image components

Definition and use of frame buffers

Interaction of the scene file and shaders in the use of frame buffers
Bibliography

The following bibliographical entries are duplicated from *Writing mental ray shaders*, but also include a link for the source of each article or book. Some books (Pharr and Humphrey's *Physically Based Rendering*, for example) are supplemented by websites from which software and additional information may be acquired. Many of the original research articles in the field of computer graphics (like Bui Tuong Phong's lighting model paper from 1975) are available through the ACM Digital Library Portal.

**Blinn 77**  

**Cook 81**  

**Driemeyer 05a**  

**Driemeyer 05b**  

**Jensen 98**  

**Jensen 01**  

**Marschner 03**  

**Perlin 85**  

**Perlin 02**  

**Pharr 04**  

**Phong 75**  

**Shirley 03**  

**Torrance 67**  

**Vince 05**  
Illumination for computer generated pictures

Full text  Pdf (1.65 MB)
Source  Communications of the ACM archive
Volume 18, Issue 6 (June 1975) table of contents
Pages: 311 - 317
Year of Publication: 1975
ISSN:0001-0782

Author  Bui Tuong Phong Stanford Univ., Stanford, CA
Publisher  ACM New York, NY, USA

Bibliometrics Downloads (6 Weeks): 43, Downloads (12 Months): 442, Citation Count: 179

ABSTRACT

The quality of computer generated images of three-dimensional scenes depends on the shading technique used to paint the objects on the cathode-ray tube screen. The shading algorithm itself depends in part on the method for modeling the object, which also determines the hidden surface algorithm. The various methods of object modeling, shading, and hidden surface removal are thus strongly interconnected. Several shading techniques corresponding to different methods of object modeling and the related hidden surface algorithms are presented here. Human visual perception and the fundamental laws of optics are considered in the development of a shading rule that provides better quality and increased realism in generated images.

REFERENCES

Note: OCR errors may be found in this Reference List extracted from the full text article. ACM has opted to expose the complete list rather than only correct and linked references.
REALISTIC RAY TRACING
SECOND EDITION

by Peter Shirley, R. Keith Morley

Price: $49.00
Availability: Temporarily out of stock. You may backorder this item. We will ship it when it becomes available.

Add to Shopping Cart

Summary

Concentrating on the “nuts and bolts” of writing ray tracing programs, this new and revised edition emphasizes practical and implementation issues and takes the reader through all the details needed to write a modern rendering system.

Most importantly, the book adds many C++ code segments, and adds new details to provide the reader with a better intuitive understanding of ray tracing algorithms.

Details

ISBN: 978-1-56881-998-7
Year: 2003
Format: Hardover
Pages: 235
Web Site: http://www.cs.utah.edu/~shirley/

All rights reserved.
From movies to video games, computer-rendered images are pervasive today. Physically Based Rendering introduces the concepts and theory of photorealistic rendering hand in hand with the source code for a sophisticated renderer. By coupling the discussion of rendering algorithms with their implementations, Matt Pharr and Greg Humphreys are able to reveal many of the details and subtleties of these algorithms. But this book goes further; it also describes the design strategies involved with building real systems—there is much more to writing a good renderer than stringing together a set of fast algorithms. For example, techniques for high-quality antialiasing must be considered from the start, as they have implications throughout the system. The rendering system described in this book is itself highly readable, written in a style called literate programming that mixes text describing the system with the code that implements it. Literate programming gives a gentle introduction to working with programs of this size. This lucid pairing of text and code offers the most complete and in-depth book available for understanding, designing, and building physically realistic rendering systems.

For a preview, download Chapter 7, "Sampling and Reconstruction".

News

June 28, 2008
luxrender, a GPL Open Source fork of pbrt, has released version 0.5 with many new features, including full spectral rendering, bidirectional path tracing, a hierarchical material and texture system, displacement mapping, and much more.

November 18, 2007
Check out luxrender, a GPL Open Source fork of pbrt. The project seems to be off to a strong start and there are some amazing images in the gallery.

July 4, 2007
The long-awaited 1.03 patch release of pbrt has been released. See the downloads page.

August 12, 2006
A number of cool new renderings have been added to the gallery page.

January 9, 2006
Mark Colbert has updated his Maya plugin that exports Maya scenes to pbrt and renders them inside Maya to both support Maya under Windows and Maya under OS X.

May 17, 2005
A Mathematica-to-PBRT exporter has been released. See the downloads page.

April 25, 2005
The long-delayed first patch release of the pbrt source code has been released. This release fixes a number of bugs found by readers and the authors in the first 6 months after the book's publication. See the downloads page for links to the source code and information about the fixes in this release.

April 25, 2005
A new photon mapping plugin, implementing many improvements to the photon mapping implementation described in the book, has been added to the plugins page.

February 14, 2005
Physically Based Rendering has won an Honorable Mention in in the Computer and Information Science category from the The Professional and Scholarly Publishing Division of the Association of American Publishers Awards. The award winners in each category were selected for their unique contribution to scholarly publishing and are considered by the panel of judges to be the best of the best for 2004.
The CDROM included with the book
The CDROM included with the book

A fully functional version of mental ray is included on the CDROM.
Writing mental ray Shaders

Proof of Purchase

The enclosed CD-ROM contains a fully functional version of mental ray®. To claim your license, please enter the following information into the SPM® license manager form after software installation.

For further information please visit http://www.mentalimages.com/licenses/
Writing mental ray shaders

A perceptual introduction

Install the fully functional mental ray® demo version

This CDROM contains a fully functional version of mental ray and a license that will allow you to run it on your computer. The installation process will copy mental ray and the SPM licensing software to your file system. If you are currently attached to the Web, a license will be requested from mental images and sent back to your computer. You will then be able to render with mental ray.

Enter the CDROM version of the book’s website

The shader source code, scene files, other data, and additional explanations are available on the book’s website at http://www.writingshaders.com/. This CDROM also contains a copy of the on-line website made when the book was published. Once you’ve installed the software and license, you can compile the example shaders and start rendering the scene files with mental ray.

http://www.writingshaders.com/
Writing mental ray shaders

A perceptual introduction

Install the fully functional mental ray® demo version

This CDROM contains a fully functional version of mental ray and a license that will allow you to run it on your computer. The installation process will copy mental ray and the SPM licensing software to your file system. If you are currently attached to the Web, a license will be requested from mental images and sent back to your computer. You will then be able to render with mental ray.

Enter the CDROM version of the book’s website

The shader source code, scene files, other data, and additional explanations are available on the book’s website at http://www.writingshaders.com/. This CDROM also contains a copy of the on-line website made when the book was published. Once you’ve installed the software and license, you can compile the example shaders and start rendering the scene files with mental ray.
Writing mental ray shaders

A perceptual introduction

If you want to install mental ray, please read the software license first:

Software End User License

Please select your operating system version from the following list to install mental ray on your computer:

Linux x86
Linux x86-64
Mac OS X (x86, PPC)
Mac OS X 64bit (x86, PPC)
Windows x86 (XP, Vista)
Windows x86-64 (XP64, Vista64)

Return to main page

mental images®, mental ray®, mental mill®, MentalSLI™, Phenomenon™, and Phenomenon™ are trademarks or, in some countries, registered trademarks of mental images GmbH, Berlin, Germany.

http://www.writingshaders.com/
#3?
Writing mental ray® Shaders

A perceptual introduction

This website provides additional resources for the book Writing mental ray shaders: A perceptual introduction. It includes all the shader source code and scene files described in the book, available for you to download. These pages will also help you understand where and how the shaders are used in the example scenes throughout the book. In the pages for a shader, the scenes that use it are linked; conversely, in the pages for a scene, the shaders used by that scene are linked. The book's bibliography is also duplicated here, but with the addition of links to the original papers and books that it references. I am also adding material beyond what is covered in the book to this site, including teaching materials as well as additional shaders that extend ideas presented in the book or based on new shader ideas suggested by readers. The book includes a CD-ROM that contains a complete version of mental ray.

Andreas Kopra
mental images

- Background
- Getting started
- The book's CDROM
- Downloading examples from the book
- Shaders
  - Shader compilation on various platforms
  - Shader source files
  - Additional shaders
- A few notes on programming style
- Scenes
  - Accessing custom shaders during rendering
  - Scene files
- Reference
  - Teaching materials
  - Bibliography