RenderToolbox3: MATLAB tools that facilitate physically based stimulus rendering for vision research

Benjamin S. Heasly Department of Psychology, University of Pennsylvania, Philadelphia, PA, USA

Nicolas P. Cottaris Department of Psychology, University of Pennsylvania, Philadelphia, PA, USA

Daniel P. Lichtman Department of Psychology, University of Pennsylvania, Philadelphia, PA, USA

Bei Xiao Department of Computer Science, American University, Washington, DC, USA

David H. Brainard Department of Psychology, University of Pennsylvania, Philadelphia, PA, USA

RenderToolbox3 provides MATLAB utilities and prescribes a workflow that should be useful to researchers who want to employ graphics in the study of vision and perhaps in other endeavors as well. In particular, RenderToolbox3 facilitates rendering scene families in which various scene attributes and renderer behaviors are manipulated parametrically, enables spectral specification of object reflectance and illuminant spectra, enables the use of physically based material specifications, helps validate renderer output, and converts renderer output to physical units of radiance. This paper describes the design and functionality of the toolbox and discusses several examples that demonstrate its use. We have designed RenderToolbox3 to be portable across computer hardware and operating systems and to be free and open source (except for MATLAB itself). RenderToolbox3 is available at https://github.com/DavidBrainard/RenderToolbox3.

Introduction

The use of physically based computer graphics is rapidly increasing in vision science and particularly in the study of color vision and the perception of material properties (e.g., Yang & Maloney, 2001; Fleming, Dror, & Adelson, 2003; Delahunt & Brainard, 2004; Xiao, Hurst, MacIntyre, & Brainard, 2012) as well as in the analysis of properties of the images formed from realistic visual scenes (e.g., Ruppertsberg & Bloj, 2007; Butler, Wulff, Stanley, & Black, 2012; Kim, Marlow, & Anderson, 2012). Using graphics to generate stimuli has the attractive feature that the experimenter can specify the scene in terms of the physical properties of shapes and light sources in the scene and then use software to produce the appropriate images for delivery to the subject. Doing so allows the independent variables in an experiment to be cast in terms of the distal stimulus. This is a critical advance for those interested in how visual processing transforms the retinal image into useful perceptual representations of the world around us.

For studying vision, a desirable property of a graphics system is physical accuracy. We would like the images viewed by a subject to be as close as possible to those that he or she would have experienced if viewing the actual scene specified. The goal of physical accuracy is, at present, met only in approximation: Models of surface reflectance approximate the actual reflectance of light from real objects, renderers embody simplifying assumptions to keep calculations tractable, and display technology limits the fidelity with which computed images can be delivered to the eye. Nonetheless, the quality of results that may be obtained with modern

Citation: Heasly, B. S., Cottaris, N. P., Lichtman, D. P., Xiao, B., & Brainard, D. H. (2014). RenderToolbox3: MATLAB tools that facilitate physically based stimulus rendering for vision research. *Journal of Vision*, *14*(2):6, 1–22, http://www.journalofvision.org/content/14/2/6, doi:10.1167/14.2.6.

doi: 10.1167/14.2.6 Received July 29, 2013; published February 7, 2014 ISSN 1534-7362 © 2014 ARVO

graphics software, and hardware is impressive and promises to keep getting better.

The basic tools to allow practicing vision scientists to incorporate graphics into their experiments are now widely available. Open-source (e.g., Blender, http://blender.org) and commercial (e.g., Maya, http://autodesk.com/products/autodesk-maya/overview) 3-D modeling programs provide graphical interfaces that allow users to specify the geometry of a scene, including the positions, sizes, and shapes of light sources and objects. Open-source (Radiance, Larson & Shakespeare, 1998; PBRT, Pharr & Humphreys, 2010; Mitsuba, Jakob, 2010) as well as commercial (e.g., Maxwell, http://maxwellrender.com; Arion, http://randomcontrol.com/arion-2) renderers have as a design goal good physical accuracy.

Although the basic graphics tools are available, we have found that their use in vision science is greatly facilitated by additional software that enables a smooth and flexible workflow between 3-D modeler, renderer, and experimental stimuli. Here we describe such software, which we call RenderToolbox3 and which is implemented as a set of MATLAB (http://mathworks.com) utilities. RenderToolbox3 is freely available under an open-source license (https://github.com/DavidBrainard/RenderToolbox3). It currently supports one open-source 3-D modeler (Blender) and two open-source renderers (PBRT and Mitsuba). Both renderers aim for physical accuracy, support multispectral rendering, and are well documented.

RenderToolbox3 is neither a 3-D modeler nor a renderer per se. Rather, it is designed to provide the following broad functionalities:

- *Stimulus family recipes*. It is often crucial in experiments on vision to vary aspects of the stimulus parametrically. For example, one might wish to study how perceived color varies as a function of object surface reflectance, object specularity, object size and pose, or the spectrum of the illumination. To facilitate creation of such parametrically varying stimuli, RenderToolbox3 provides a mechanism for producing a *family* of related stimuli based on one 3-D *parent scene* and a list of parametric manipulations to be applied to the scene. Together, the parent scene and specified manipulations comprise a rendering *recipe* that documents the family of stimuli and allows the stimuli to be regenerated automatically.
- *Multispectral rendering*. RenderToolbox3 supports the specification and rendering of scenes on a multispectral (wavelength-by-wavelength) basis. For studies of color vision, multispectral rendering is an important component of maximizing physical accuracy. In general, 3-D modelers do not allow multispectral reflectances to be assigned to materials nor multispectral power distributions to be assigned to illuminants. RenderToolbox3 adds this functionality to the rendering workflow in order to make use of the multispectral rendering capabilities of the supported renderers.
- *Physically based materials*. RenderToolbox3 supports arbitrary material specifications. As with multispectral reflectances and power distributions, 3-D modelers do not generally know about the various surface material models supported by physically based renderers. RenderToolbox3 adds the specification of arbitrary materials to the rendering workflow to increase the accuracy with which materials may be rendered.
- *Renderer validation*. RenderToolbox3 includes several rendering recipes that probe the behavior of supported renderers for conformance to known physical principles. As vision scientists, we are generally consumers of renderers in that we treat them as black boxes rather than reviewing their algorithms and source code in detail. Thus it is important to have end-to-end tests of renderer behavior. For example, the radiance of the image of a surface seen under a point light source should drop by a factor of four when the distance to the light source is doubled. Although testing such behaviors does not make the rendering workflow any easier, implementing and documenting such tests is an important part of understanding the stimuli produced by the renderers. RenderToolbox3 provides rendering recipes that use a simple calibration parent scene and probe renderer behavior by manipulating the scene in ways that should produce predictable results as derived from physical principles. These recipes facilitate documentation and validation of renderer behaviors. In turn, validation of renderer behaviors allows computation of renderer-specific constants that bring the rendering output into known radiometric units.
- *Renderer comparison*. RenderToolbox3 makes it easy to use the same rendering recipe with each of its supported renderers. Comparisons of renderer outputs from the same recipe can be useful to assess the degree to which independently implemented renderers agree. To the extent that multiple renderers do agree, one may be more confident that each is accurately simulating the flow of light through the scene. To the extent that multiple renderers disagree, one must assume a degree of uncertainty about the physical accuracy of the renderings, pending more detailed assessment of the reasons for the disagreement. Such comparisons are especially useful for complex scenes in which it is difficult to predict the correct results with simple analytical calculations.
- *Convenience*. Finally, by handling or standardizing many of the details of the rendering workflow, RenderToolbox3 aims to make it easier for the practicing scientist to employ graphics tools.

The rest of this paper is organized as follows. In the next section, we provide an overview of the design of RenderToolbox3 and describe a typical workflow. Then, we review a number of examples provided with the toolbox that illustrate its functionality. We conclude with a broader discussion.

Toolbox design and workflow

RenderToolbox3 contains a set of MATLAB utilities designed to facilitate the creation of graphical stimuli. It also prescribes a workflow for stimulus creation that incorporates existing graphics tools, including 3-D modelers and physically based renderers.

Utilities

RenderToolbox3 is intended to be portable across computer hardware and operating systems. Most of the utilities are implemented as m-functions in MATLAB's portable scripting language. One utility is written in C as a MATLAB mex-function, and RenderToolbox3 provides an m-function utility for building this mex-function. Collectively, these MATLAB utilities depend on three other software libraries:

1. MATLAB's built-in Java-based XML parsing tools allow RenderToolbox3 to read and manipulate parent scenes stored in Collada XML files.
2. The OpenEXR image library allows RenderToolbox3 to read multispectral renderer output data.
3. The Psychophysics Toolbox (http://psychtoolbox.org; Brainard, 1997) provides many colorimetric and image-processing functions.

Each of these libraries is itself portable across operating systems. It should be possible to use RenderToolbox3 on any system on which MATLAB, OpenEXR, and the Psychophysics Toolbox are available. We have used it both with Apple's Macintosh OS/X and with Linux.

In addition to being portable, RenderToolbox3 and its dependencies are almost entirely free and open source. MATLAB is the only nonfree, commercial component of RenderToolbox3.

Workflow

The RenderToolbox3 workflow for stimulus creation has four steps: (a) *3-D modeling* of a parent scene, (b) *batch processing* of the scene with parametric manipulations, (c) physically based *rendering* of the scene to produce a family of multispectral renderings, and (d) *image processing* of the renderings to produce stimuli for delivery to subjects and to perform other analyses.

To illustrate a typical RenderToolbox3 workflow, we consider these four steps in the context of a concrete example, which we call *DragonColorChecker*. The parent scene is modeled in Blender by importing an existing model of a dragon and creating other objects using Blender's modeling tools. The reflectance spectrum of the dragon is manipulated using multispectral colorimetric data imported from the Psychophysics Toolbox. The scene is rendered 24 times, producing a family of 24 multispectral renderings. Finally, the renderings are processed and combined into a single RGB montage that shows 24 dragons of various colors.

The fully functional DragonColorChecker example is included with the RenderToolbox3 distribution (see Appendix). For purposes of illustration below, we omit details.

3-D modeling

The RenderToolbox3 workflow begins with the Blender application. Blender is an open-source software tool for producing 3-D graphics and animations with a large community of open-source and commercial users. RenderToolbox3 users can use Blender to specify the geometry of the parent scene, including the position and orientation of the camera and the positions and shapes of the light sources and objects.

To create the DragonColorChecker parent scene, the user would import a model of a Chinese dragon statue from the Stanford 3-D Scanning Repository (http://graphics.stanford.edu/data/3Dscanrep/) using Blender's geometry-importing functionality. The user would then use Blender's basic modeling tools to create a box in which to place the dragon, light sources to illuminate the dragon, and a camera to view the dragon. Finally, the user would export the scene from Blender as a Collada file. In Figure 1, see the top-left box labeled "3-D Modeling."

Collada is an XML-based open standard for exchanging 3-D models between applications like modelers and renderers. So, although RenderToolbox3 nominates Blender as its preferred modeling tool, other modeling tools that support Collada should be able to produce parent scene files that are suitable for use with RenderToolbox3.

Batch processing

The workflow continues with batch processing, in which the user may specify a number of manipulations to be applied to the parent scene, including changes to scene geometry, reflectance and illuminant spectra,

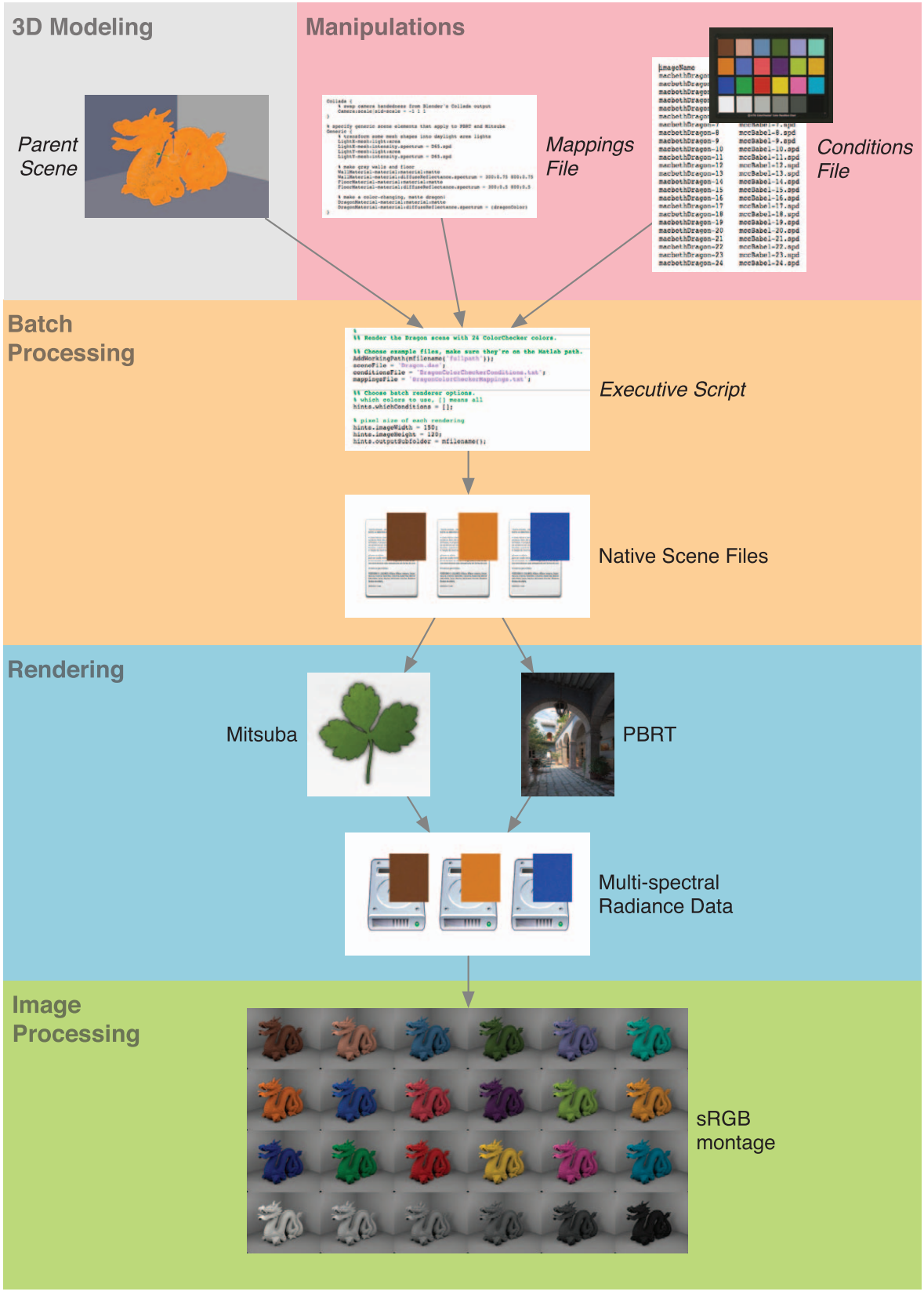

Figure 1. Overview of the DragonColorChecker workflow. Files that the user must supply are labeled with italics. These include the Parent Scene, which may be modeled using the Blender application and must be exported as a Collada file; the Mappings File, which maps reflectance spectra to the dragon model; the Conditions File, which lists 24 reflectance spectra measured from a ColorChecker

→

←
color rendition chart; and the Executive Script, which binds together the Parent Scene file, Mappings File, and Conditions File and invokes RenderToolbox3 utilities. Subsequent steps are performed automatically by RenderToolbox3 utilities. Twenty-four Native Scene Files are produced (only three are shown) that may drive one of the supported renderers, PBRT or Mitsuba, in order to produce 24 Multispectral Radiance Data files (only three are shown). All data files use a consistent MATLAB mat-file format and physical radiance units. Multispectral data files are combined into a single sRGB montage that resembles the original ColorChecker chart.

materials, types of lighting, and renderer behaviors. The user specifies these manipulations by writing two text files that use RenderToolbox3-specific syntax: the *conditions file* and the *mappings file*. In Figure 1, see the top-right box labeled "Manipulations." The user also writes an *executive script* in MATLAB that binds together the parent scene file, the conditions file, and the mappings file, and invokes RenderToolbox3 utilities. Together, the parent scene file, conditions file, mappings file, and executive script constitute a complete set of instructions for rendering that we call a RenderToolbox3 *recipe*.

RenderToolbox3 recipes are largely open-ended. The conditions file and mappings file may be written by hand, generated programmatically using RenderToolbox3 utilities, or omitted entirely (in which case the parent scene will be rendered only once). The executive script may invoke various RenderToolbox3 utilities and other MATLAB functions. The conditions file, mappings file, and executive script may refer to external resources, like multispectral reflectance and illuminant data files and texture image files. A specific recipe may be used to reproduce a family of renderings at any time. The recipe also serves as documentation of the renderings, and may be accessed during the rest of the rendering workflow and subsequent data analysis in order to avoid redundant data entry.

Conditions file

The conditions file uses a tabular format in which each column represents a recipe parameter and each row represents a condition and contains a set of parameter values for that condition. A general description of the conditions file syntax is available in the RenderToolbox3 online documentation (https://github.com/DavidBrainard/RenderToolbox3/wiki/Conditions-File-Format).

The DragonColorChecker conditions file specifies 24 conditions, one for each square on a Macbeth Color Checker Chart (McCamy, Marcus, & Davidson, 1976). Reflectance measurements of a Color Checker Rendition Chart are taken from the Psychophysics Toolbox collection of colorimetric data files and converted using a RenderToolbox3 utility to 24 multispectral reflectance data files that are compatible with PBRT and Mitsuba. The choice of 24 Color Checker reflectance spectra was arbitrary and made to illustrate how to specify multiple spectral reflectance spectra with RenderToolbox3. The conditions file declares two recipe parameters in two columns. Each column has a header row and 24 additional rows (see Listing 1). The first column provides a mnemonic name for each condition. The second column specifies the reflectance spectrum data file to use for the dragon color for each condition.

```
% Listing 1

imageName       dragonColor
brownDragon     reflectance-spectrum-1.spd
orangeDragon    reflectance-spectrum-2.spd
blueDragon      reflectance-spectrum-3.spd
…               …
```

Listing 1. Excerpt from a conditions file that has two column headers followed by 24 rows (only three rows are shown). The left-hand column has the header imageName and contains an arbitrary mnemonic for each of 24 conditions. The right-hand column has the header dragonColor and lists 24 reflectance spectrum data file names. Columns are delimited by tab characters, and rows are delimited by new lines. In general, there is no limit to the number of columns or rows in a conditions file.

Mappings file

The mappings file defines points of contact between the parent scene and the conditions file. It uses syntax inspired by the Collada XML schema and allows the user to map arbitrary values to elements of the parent scene, based on names specified during 3-D modeling. For example, the user might create a camera in Blender and assign to it the name "camera." Then in the mappings file, the user would be able to modify properties of the "camera" element, including its position and field of view. Likewise, the user could use the mappings file to modify named light sources, objects, and materials. Elements that are not referred to in the mappings file are rendered without manipulation. A general description of the mappings file syntax is available in the RenderToolbox3 online documentation (https://github.com/DavidBrainard/RenderToolbox3/wiki/Mappings-File-Format).

```
% Listing 2

…
Generic {
        % make a color-changing, matte dragon
        Dragon-material:material:matte
        Dragon-material:diffuseReflectance.spectrum = (dragonColor)

        % make gray walls and floor
        Wall-material:material:matte
        Wall-material:diffuseReflectance.spectrum = 300:0.75 800:0.75
        Floor-material:material:matte
        Floor-material:diffuseReflectance.spectrum = 300:0.5 800:0.5
}
…
```

Listing 2. Excerpt from a mappings file that maps user-defined values to scene elements. Here all mappings are contained in a Generic block, which is applied to any renderer and is delimited by curly braces {}. The Dragon-material scene element is declared to be a material of type matte. The user-defined spectrum (dragonColor) is mapped to the diffuseReflectance property of Dragon-material. Parentheses indicate that the value of dragonColor may change for each condition and should be taken from the "dragonColor" column of the conditions file. The Wall-material and Floor-material elements are also declared as materials of type matte. Constant strings are mapped to the diffuseReflectance property of Wall-material and Floor-material. These strings specify spectrally uniform reflectance spectra in the range 300 though 800 nm, using a syntax for specifying arbitrary sampled spectra that PBRT and Mitsuba can parse. The reflectance of the wall is specified as 0.75, and that of the floor is 0.5.

For the DragonColorChecker example, the values listed in the dragonColor column of the conditions file are mapped to the surface reflectance of the dragon model (see Listing 2). The mappings file also specifies that the objects in the scene are matte and that the floor and walls have spectrally flat reflectances.

The mappings file allows the user to introduce values into the workflow that would be difficult or impossible to specify using 3-D modeling programs like Blender or the Collada XML standard. Blender and Collada represent the reflectances of the dragon statue, the floor, and the walls using three-component RGB values. The mappings file allows the user to replace RGB values with multispectral reflectances and to specify the material properties of objects.

Executive script

The DragonColorChecker executive script invokes RenderToolbox3's batch-processing utilities, passing in the parent scene Collada file, the conditions file, and the mappings file, and producing a family of scene files in the native format of one of the supported renderers (see Listing 3). For each condition, the batch-processing utilities read the parent scene, manipulate scene elements using values from the conditions file and the mappings file, and write the manipulated scene to a renderer native scene file. Mitsuba provides its own utility for converting Collada files to Mitsuba native scene files. RenderToolbox3 provides a custom utility for converting Collada files to PBRT native scene files. The result is 24 new renderer native scene files, one for each of the reflectance spectrum data files specified in the conditions file. In Figure 1 see the upper-middle box labeled "Batch Processing."

Rendering

RenderToolbox3 currently supports two open-source, physically based renderers: PBRT and Mitsuba. These renderers use ray and path-tracing algorithms to realistically simulate how light would flow through a scene. They emphasize correctness with respect to physical principles, like conservation of energy, as opposed to aesthetic or real-time considerations. Both renderers take advantage of modern computer hardware by dividing rendering computations among local CPU cores. In addition, RenderToolbox3's batch-rendering utilities may be configured to take advantage of computing clusters, via the MATLAB Distributed Computing Toolbox. The RenderToolbox3 online documentation provides instructions for compiling PBRT and Mitsuba for multispectral rendering using arbitrary spectrum sampling (https://github.com/DavidBrainard/RenderToolbox3/wiki/Building-Renderers).[1]

For the DragonColorChecker example, the executive script passes the names of 24 PBRT native scene files to the RenderToolbox3 batch-rendering utilities. These utilities invoke PBRT once per scene file, producing 24 multispectral renderings stored as data files with PBRT-specific units and formatting. RenderToolbox3 reads each PBRT data file, converts the data to physical radiance units, and saves the radiance data in a new MATLAB mat-file. Although each renderer may use its own output units and data file format, the RenderToolbox3 batch-rendering utilities always produce multispectral output data in physical radiance units stored in MATLAB mat-files. Uniform output data facilitates comparisons between renderers, swapping between renderers, image processing,

```
% Listing 3

...

%% choose recipe files
parentSceneColladaFile = 'Dragon.dae';
conditionsFile = 'DragonColorCheckerConditions.txt';
mappingsFile = 'DragonColorCheckerMappings.txt';

...

% batch processing: make 24 renderer-native scene files for PBRT
hints.renderer = 'PBRT';
rendererNativeSceneFiles = MakeSceneFiles( ...
    parentSceneColladaFile, conditionsFile, mappingsFile, hints);

% batch rendering: make 24 multi-spectral data files with PBRT
multispectralDataFiles = BatchRender(rendererNativeSceneFiles, hints);

% image processing: condense multi-spectral data files into one RGB montage
montageFile = 'DragonColorChecker-PBRT.png';
MakeMontage(multispectralDataFiles, montageFile);

...
```

Listing 3. A simplified DragonColorChecker MATLAB executive script that binds a parent scene Collada file with a conditions file and a mappings file and invokes RenderToolbox3 utilities to form a complete RenderToolbox3 recipe. The user chooses the PBRT renderer, using a structure of RenderToolbox3 hints. The MakeSceneFiles( ) utility uses the parent scene Collada file, conditions file, and mappings file to produce a family of PBRT native scene files that PBRT can render. The BatchRender( ) utility lets each PBRT native scene file drive PBRT in turn to produce a family of multispectral data files. The MakeMontage( ) utility condenses the family of multispectral data files into a single RGB image stored in a portable png file.

and other analyses. In Figure 1, see the lower-middle box labeled "Rendering."

Image processing

The workflow ends with processing the multispectral output data mat-files to form viewable images. RenderToolbox3 image-processing utilities can convert multispectral data into sRGB (http://www.w3.org/Graphics/Color/sRGB.html) images and montages as well *sensor images*, whose pixel values are computed by applying arbitrary color-matching functions or sensor spectral sensitivities to the multispectral data. The final product of the workflow may be a family of sRGB images or other sensor images, a single large sRGB montage, or analyses performed on the multispectral renderings themselves. Indeed, MATLAB provides many analysis tools that may be brought to bear on multispectral radiance data stored in the mat-files.

For the DragonColorChecker example, the executive script invokes RenderToolbox3's montage utility to combine the 24 multispectral renderings into a single sRGB image. In Figure 1, see the bottom box labeled "Image Processing."

Several other examples included in the RenderToolbox3 distribution use the same parent scene as the DragonColorChecker but perform different manipulations (see Appendix). The *Dragon* example renders a single matte dragon with a multispectral reflectance. The *DragonGraded* example renders a family of matte dragons with reflectances that are smoothly graded between two multispectral reflectances. The *DragonMaterials* example renders a family of four dragons with matte, plastic, metal, and glass materials.

Examples

The RenderToolbox3 distribution contains many fully functional examples, currently consisting of 16 parent scenes and 31 associated executive scripts. These examples demonstrate features of the toolbox and variations on the RenderToolbox3 workflow and include the dragon examples discussed in the previous section. In this section, we discuss five additional examples and their main themes. See the Appendix for a summary of all the examples.

RadianceTest

The *RadianceTest* example uses RenderToolbox3 to verify that renderers adhere to certain physical principles of radiometry (Wyszecki & Stiles, 1982) and to identify radiometric unit conventions used by the renderers. The RadianceTest recipe and resulting renderings document renderer behaviors and allow RenderToolbox3 to bring all renderer outputs into

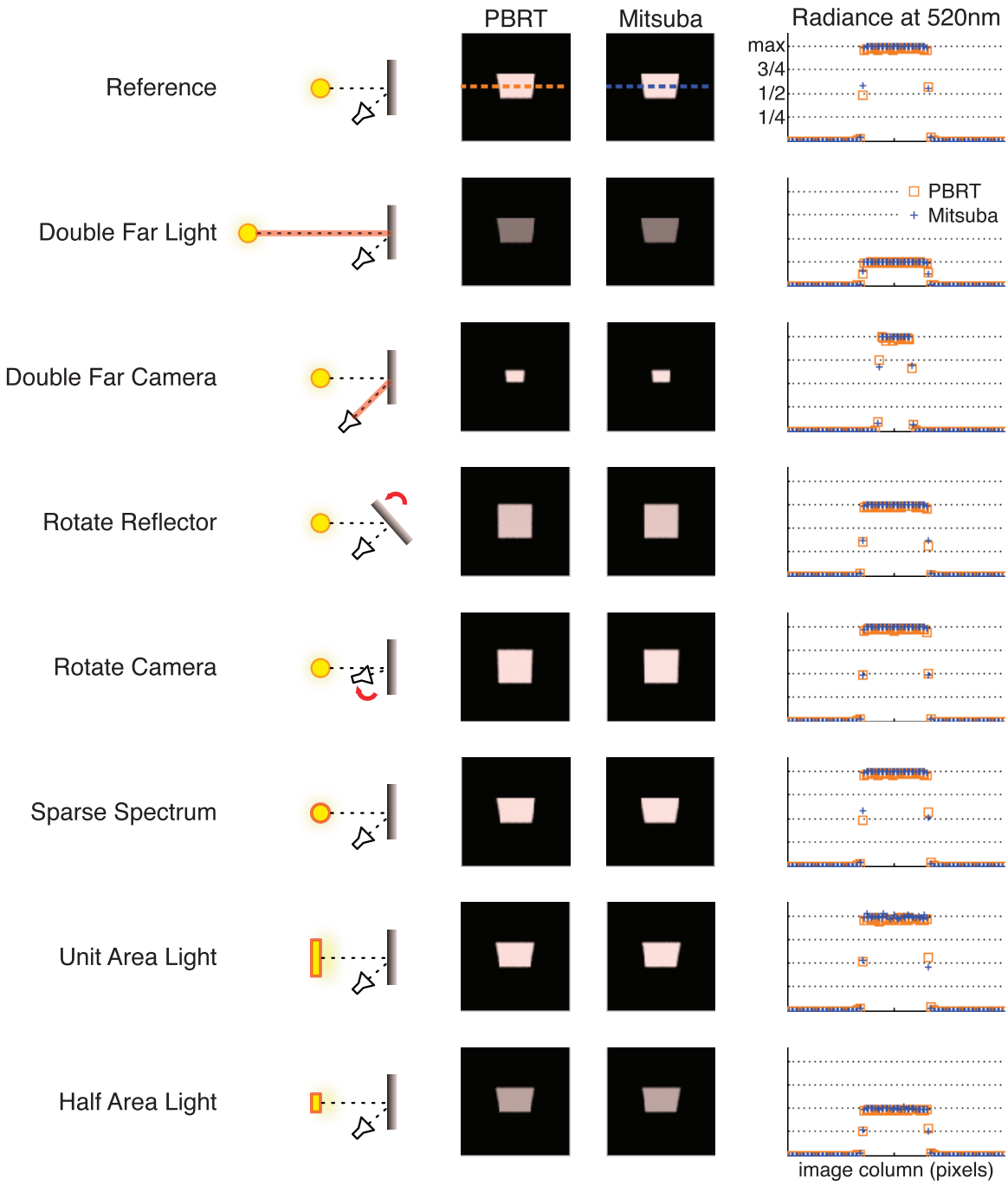

Figure 2. Summary of RadianceTest results. Each row shows results from one of the eight RadianceTest conditions. From left to right, each row contains (a) the name of the condition; (b) a schematic drawing of the scene geometry, including the light source (yellow circle or bar), reflector (gray bar), camera (black polygon), and manipulation (red highlight); (c) sRGB representation of the scene as rendered by PBRT; (d) sRGB representation of the scene as rendered by Mitsuba; and (e) reflected radiance profile taken at a single arbitrary wavelength (520 nm) at a horizontal slice through the multispectral renderings at the location of the dashed lines for PBRT (orange) and Mitsuba (blue). Radiance profiles vary in height and width with respect to the Reference condition in a manner consistent with physical principles and illuminant spectra that are treated in units of power per unit wavelength. A single common scale factor was applied to all sRGB images to facilitate visual comparisons.

physical radiance units. The recipe also demonstrates RenderToolbox3 facilities that are generally useful, such as manipulating object positions and orientations using the conditions file, creating area lights, and reusing the same recipe with both supported renderers.

The RadianceTest parent scene is simple, containing a distant point light, a flat diffuse reflecting surface

(referred to as the reflector in the following), and a camera. The point light emits unit intensity at all wavelengths, specified at evenly spaced wavelength intervals. The reflector is square, planar, and uses a matte material with unit reflectance at all wavelengths. The vector connecting the point light to the center of the reflector is normal to the reflector's surface. The camera uses perspective projection and views the reflector at an angle. The camera's line of sight passes through the center of the reflector and makes a 45° angle with the reflector's surface normal.

The RadianceTest conditions file specifies eight rendering conditions. Various parameters (e.g., distance to the light source, orientation of the reflector) are manipulated across the conditions. In Figure 2, see the condition names and schematic drawings on the left. Because the scene is simple, we can predict the magnitude of the effect that each manipulation should have on the camera image using basic physical principles and compare this to the output of the renderers.

The *Reference* condition establishes a baseline for comparison with its parameters defining *reference values*. These are the distance from the point light to the reflector, the orientation of the reflector relative to the point light, the distance from the reflector to the camera, the orientation of the reflector relative to the camera, and the emission spectrum of the point light.

The *Double Far Light* condition doubles the distance from the point light to the reflector and leaves other parameters at the reference values. As a consequence, the light reaching the reflector should have one-fourth intensity as compared to the reference condition. The light reflected toward the camera and the renderer output magnitude should also have one fourth of their reference values.

The *Double Far Camera* condition doubles the distance from the reflector to the camera and leaves other parameters at the reference values. The light reaching the reflector should be unchanged, and thus the reflector's radiance should be constant. This, in turn, means that the image of the reflector should have the same intensity values as in the reference condition. The image of the reflector, however, should be reduced in area by a factor of four and thus contain one fourth as many pixels as it does in the reference condition.

The *Rotate Reflector* condition rotates the reflector about its center by 41.4° so that the vector connecting the point light to the center of the reflector forms a 41.4° angle against the reflector's surface normal. This condition leaves other parameters at their reference values. The light reflected toward the camera should exhibit a cosine falloff from the reference condition, causing the image of the reflector to have cos(41.4) (approximately three fourths) of its reference intensity and pixel magnitudes.

The *Rotate Camera* condition rotates the camera about the reflector at constant distance so that the camera's line of sight makes a 10° angle with the reflector's surface normal. This condition leaves other parameters at their reference values, including the distance from the camera to the reflector. This should not affect the intensity values of the image of the reflector. However, because the camera views the reflector more nearly head-on than in the reference condition, the image of the reflector should have a larger area and thus contain more pixels than in the reference condition.

The *Sparse Spectrum* condition replaces the point light emission spectrum with a sparse spectrum sampled at half as many wavelengths and leaves other parameters at their reference values. If the renderers interpret spectra in units of power per unit wavelength, then this manipulation should have no effect on the image. If, on the other hand, the renderers interpret spectra in units of power per wavelength band, this manipulation should halve the intensity of the output values.

The *Unit Area Light* condition replaces the point light with a planar, circular, diffuse area light with unit area. The vector connecting the center of the area light with the center of the reflector is normal to both the area light and the reflector. This condition leaves all other parameters at their reference values, including the light's emission spectrum. The total light emitted toward the reflector should be approximately the same as in the reference condition if the renderers use consistent units between point lights (power) and area lights (power per unit area).

The *Half Area Light* condition is like the Unit Area Light condition in all ways except that the area light is reduced in size to one half unit area instead of unit area. As a consequence, the light reaching the reflector should have one-half intensity as compared to the reference condition. The reflector image should thus have one half of its reference intensity and pixel magnitudes.

For both PBRT and Mitsuba, each manipulation affects renderer output magnitude in the manner that would be predicted from physical principles as described above given a camera that records units of radiance and given that light emission spectra are interpreted in units of power per unit wavelength. In Figure 2, see the PBRT and Mitsuba renderings and radiance profiles. These results inform RenderToolbox3 utilities that convert colorimetric data from units of power per wavelength band, including Psychophysics Toolbox colorimetric data, into units of power per unit wavelength for use with renderers. They confirm basic agreement of the two renderers with each other and with physical principles. They also allow RenderToolbox3 to treat renderer numeric outputs in terms of useful radiometric units.[2]

```
% Listing 4


Mitsuba-path {
        Camera:transform|name=toWorld:scale.x = \
                                [Camera:optics:technique_common:orthographic:xmag]
        …
}
```

Listing 4. Excerpt from a mappings file that adjusts Mitsuba camera properties using values contained in the parent scene Collada file. All mappings are contained in a Mitsuba path block, which would apply low-level path syntax to Mitsuba, and the block is delimited by curly braces {}. The right-hand value in square braces [] refers to the x-magnification property of the Camera scene element of the Collada parent scene. This value is applied to the x-scale factor of the Camera element of the Mitsuba scene that will be generated during batch processing.

The renderers themselves do not specify particular units for radiance, and their raw outputs differ from one another by a scale factor. To make renderer outputs directly comparable, RenderToolbox3 scales the output from each renderer by a precomputed radiometric unit factor.

Computing renderer radiometric unit factors relies on the simplicity of this RadianceTest parent scene and a derivation from physical principles of how light should be emitted from the point light and reflected toward the camera in order to produce an expected source radiance. The radiometric unit factor for each renderer is chosen such that the actual renderer output from the reference condition times the radiometric unit factor equals this expected radiance. Once computed, RenderToolbox3 applies the same radiometric unit factor to all numeric outputs from the same renderer.[3] The RenderToolbox3 online documentation contains a detailed description and derivation of the expected radiance (https://github.com/DavidBrainard/RenderToolbox3/wiki/RadianceTest#radiometric-units).

The RenderToolbox3 online documentation contains further details about this RadianceTest example as well as the *ScalingTest*, which examines the effects of nonradiometric manipulations on renderer outputs. The ScalingTest shows that parameters such as rendering strategy and image reconstruction filter size can affect renderer output scaling. The same RenderToolbox3 utilities that apply renderer radiometric unit factors could potentially apply additional scale factors that normalize renderer outputs with respect to nonradiometric parameters. To that end, we have included functionality in these utilities for discovering many nonradiometric parameter values, but we have not yet implemented any additional scale factors.

SimpleSphere

Like the RadianceTest example, the SimpleSphere example also validates the renderers but for a scene in which simple analytic calculations cannot be used to predict the desired image. The example also introduces a feature of RenderToolbox3 mappings files that provides low-level access to the parent scene and the behavior of renderers.

The parent scene is again simple, containing a distant point light, a sphere, and a camera. The point light uses the CIE D65 (CIE, 2004) standard illuminant spectrum. The sphere uses a Ward model material (Ward, 1992) with orange-looking diffuse spectral reflectance and spectrally nonselective specular spectral reflectance. The camera uses orthographic projection and views the sphere head-on with the point light behind, above, and to the right of the camera.

The scene is rendered once each with PBRT, Mitsuba, and with a third renderer called the SphereRendererToolbox. The SphereRendererToolbox is a simple renderer implemented entirely in MATLAB by us and is available under an open-source license (https://github.com/DavidBrainard/SphereRendererToolbox/wiki). It can only render Ward model spheres under point light sources in orthographic projection, but for such scenes, it provides an independent rendering implementation based on the relevant physical principles.

The SimpleSphere mappings file chooses scene parameters in order to match the PBRT and Mitsuba scenes to the constraints of the SphereRendererToolbox as closely as possible. For example, the mappings file adjusts parameters of Mitsuba's orthographic camera that are not automatically set by Mitsuba's built-in Collada importer (see Listing 4). Because this adjustment is unusual and renderer-specific, it must be made using a special mapping syntax we refer to as *path syntax*. This syntax dispenses with convenience but allows arbitrary access to elements of the parent scene and elements of the renderer-specific scene files that will be generated during batch processing. The RenderToolbox3 online documentation contains more details about this path syntax (https://github.com/DavidBrainard/RenderToolbox3/wiki/Scene-DOM-Paths).

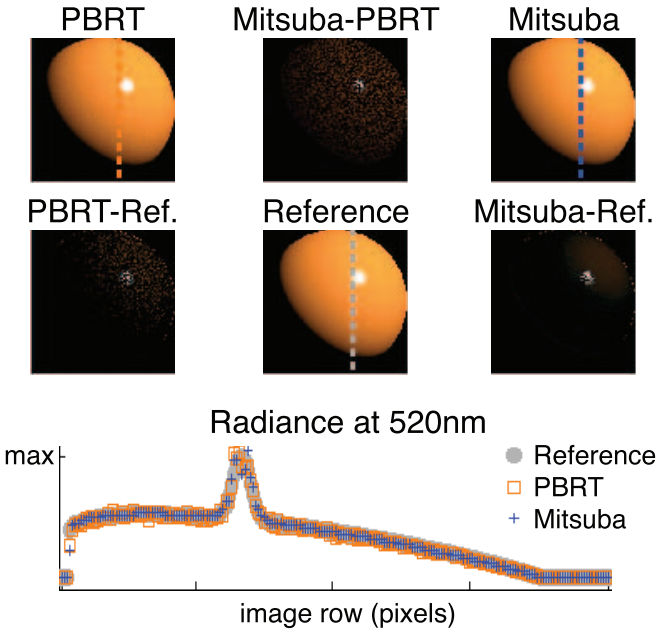

Figure 3. Comparison of SimpleSphere renderings. PBRT and Mitsuba rendered the SimpleSphere scene. The SphereRendererToolbox produced a third Reference rendering. The top and middle rows contain sRGB representations of SimpleSphere multispectral renderings and element-wise differences between multispectral renderings. Top row, from left to right: PBRT, Mitsuba minus PBRT, and Mitsuba. Middle row, from left to right: PBRT minus Reference, Reference, and Mitsuba minus Reference. A single common scale factor was applied to all sRGB images to facilitate visual comparisons. The bottom row contains reflected radiance profiles taken at a single arbitrary wavelength (520 nm) at a vertical slice through the multispectral renderings at the location of the dashed lines for PBRT (orange), Mitsuba (blue), and the SphereRendererToolbox Reference (gray).

Each renderer produces a multispectral rendering of the scene. If all three renderers essentially agree on how to render the scene, then any difference image, taken as the element-wise subtraction of one multispectral rendering from another, should contain only small values. Figure 3 shows sRGB representations of three such difference images as well as sRGB representations of the three original renderings produced by PBRT, Mitsuba, and the SphereRendererToolbox ("Reference").

The three renderers generally agree on how to render the SimpleSphere scene. The greatest differences appear at the location of the sphere's specular highlight (a white circle in the top right part of each rendering) and along the edge of the sphere. Differences at the edge may be the result of pixel filtering: PBRT and Mitsuba compute pixel values using filters that introduce slight blurring and gradual edges whereas the SphereRendererToolbox computes pixel values directly without filtering.

The RenderToolbox3 online documentation contains information about several examples that validate renderers against physical principles and against one another, including this SimpleSphere example. These include *SimpleSquare*, which tests multispectral reflections; *Interreflection* and *TableSphere*, which test reflections among objects; *CoordinatesTest*, which validates coordinate system handedness and spatial transformations; and *RGBPromotion*, which probes how renderers promote RGB values to multispectral representations.

MaterialSphereBumps

The MaterialSphereBumps scene renders a sphere using three different surface materials. This scene illustrates how RenderToolbox3 allows manipulation of the material properties of objects and also how it can be used to control the application of a bitmap texture to add bumps to the surface of the sphere. With this example, we also introduce RenderToolbox3's utilities for converting multispectral renderings to sensor images. In this case, the sensor images are calculated with respect to estimates of the spectral sensitivities of the human cones. The ability to create sensor images is useful in vision science as we often go on to take such images and use them together with monitor calibration data to produce images for display that have the same effect on the cone photoreceptors as the spectra rendered in the image plane would have had (Brainard, Pelli, & Robson, 2002; Brainard & Stockman, 2010).

The scene contains a polygon approximation of a sphere and a nearby point light. A perspective camera views the sphere head-on, such that the point light is directly above the camera.

Each rendering of the sphere shows bumpy, textured regions that outline the oceans and continents of the earth (see Figure 4). The MaterialSphereBumps mappings file declares the texture scene element, which stores pixel data from an image of the earth downloaded from the Planetary Pixel Emporium (http://planetpixelemporium.com) and used here with permission of the author. The earth's texture is bound to the surface material of the sphere as a *bump map*, which allows the renderer to interpret bitmap pixel values as variations in height of the surface of the sphere.

The scene's conditions file specifies three different surface materials for the sphere: matte material with red-looking reflectance, a Ward model material with green-looking diffuse reflectance and uniform specular reflectance, and a metal material with a multispectral index of refraction and absorption coefficient taken from measurements of gold that are provided with the

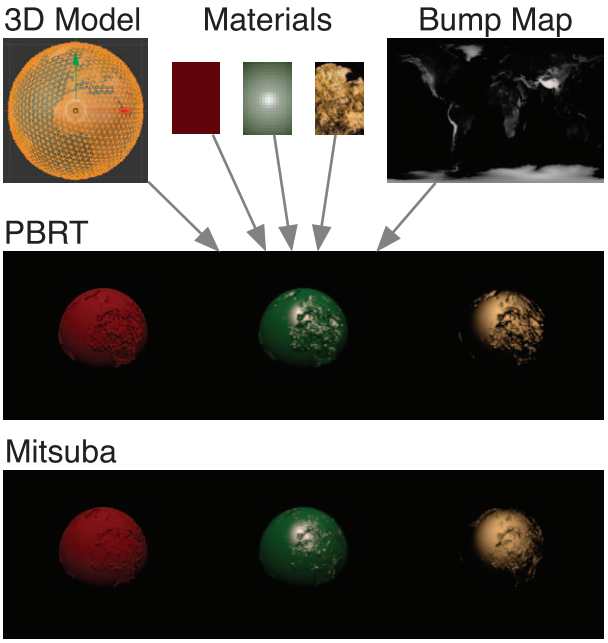

Figure 4. Summary of MaterialSphereBumps workflow. The top row shows key elements of the scene, from left to right: the parent scene 3-D model created in Blender; various sphere materials including red-looking matte, green-looking Ward material, and gold metal; and an image of the earth used as a bump map to alter the surface height of the sphere. The middle rows show a sRGB montage of renderings produced by PBRT. The bottom row shows a sRGB montage of renderings produced by Mitsuba. Both renderers support bump maps and produce renderings with surface height altered to resemble the earth image. However, the bumps appear taller in the PBRT rendering than they do in the Mitsuba rendering, indicating that the two renderers interpret bump maps differently.

PBRT source code distribution. The textured bump map produces a different visual effect with each of the surface materials.

As with other example recipes, RenderToolbox3 makes it easy to render MaterialSphereBumps with PBRT or Mitsuba. Comparison of the outputs reveals that the two renderers treat bump maps differently, resulting in images that have visibly different textures (see Figure 4). Because the renderers disagree about how to render bump maps, care must be used in the physical interpretation of bump maps.

This MaterialSphereBumps recipe produces two kinds of output. First, the three separate multispectral renderings are combined into a single sRGB montage. Second, the same multispectral renderings are converted to sensor images, based on estimates of the spectral sensitivities of the human cones (see Listing 5). Sensor images produced this way could be delivered to research subjects via calibrated displays.

The RenderToolbox3 online documentation contains more information about this MaterialSphereBumps example as well as a *MaterialSphere* example that does not use bump maps and a *MaterialSphere-Remodeled* example that modifies the Collada parent scene on the fly in MATLAB. The documentation also contains information about the *Dice* and *CubanSphere-Textured* examples, which use bitmap textures to specify object reflectances instead of bumps.

Interior

The *Interior* example represents a unique RenderToolbox3 workflow that accommodates a 3-D scene created by an unknown author for purposes unrelated to RenderToolbox3. Many such scenes are available on the web and thus available for use with RenderTool-

```
% Listing 5

% batch processing and rendering
%   multispectralDataFiles

...

% image processing

% choose Psychophysics Toolbox data estimating human cones sensitivities
matchingFunction = {'T_cones_ss2.mat'};

% create sensor images estimating cone responses to multi-spectral renderings
sensorImages = MakeSensorImages(multispectralDataFiles, matchingFunction);
```

Listing 5. A simplified MaterialSphereBumps MATLAB executive script that produces sensor images based on estimates of the spectral sensitivities of the human cones. The user chooses a Psychophysics Toolbox colorimetric data file that contains estimates of human cone sensitivities. The cone data act as the color-matching function passed to the RenderToolbox3 MakeSensorImages( ) utility, which transforms multispectral renderings into sensor images that estimate the responses of human cones to the renderings. The sensor images are saved in MATLAB mat-file data files with file names automatically chosen based on the multispectral data file names and the color-matching function name.

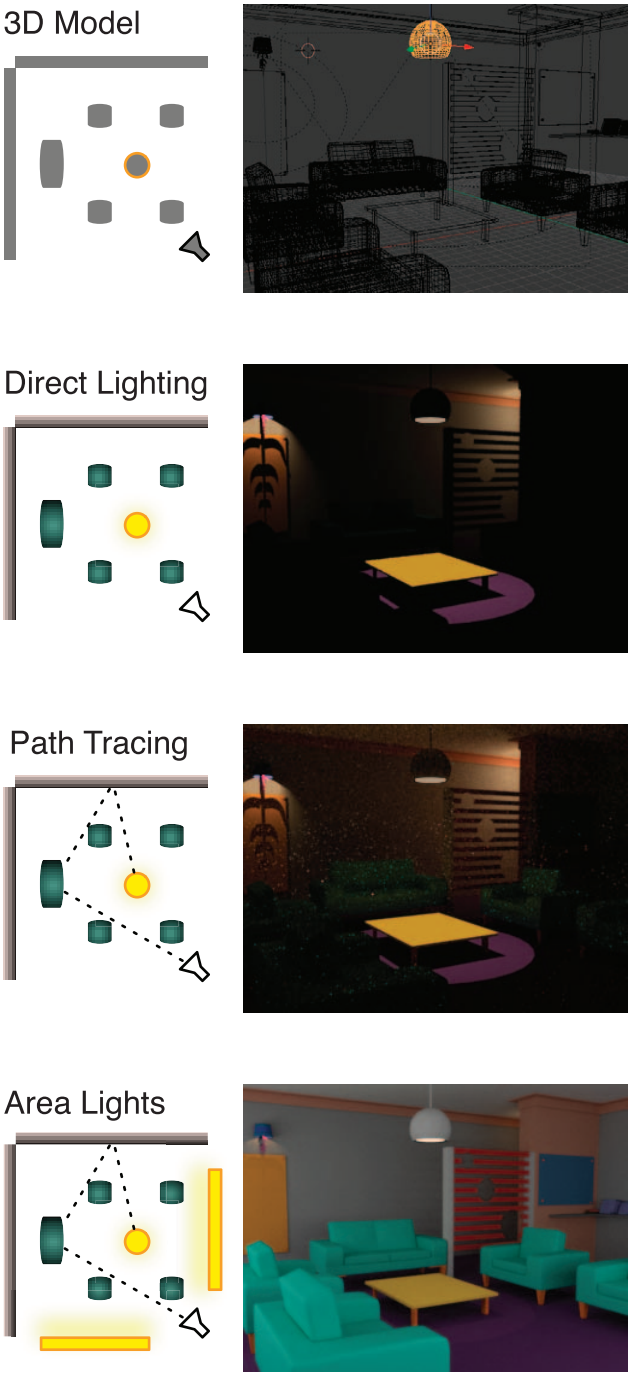

Figure 5. Summary of modifications to the "wild" Interior scene. Each row shows the scene at a particular step of modification and contains, from left to right, the name of the step, a schematic plan view of the Interior scene, and a Blender preview image or sRGB representation of the scene as rendered by Mitsuba. The original parent scene 3-D model was created by an unknown author for uses unrelated to RenderToolbox3. As viewed in Blender, many objects are visible, including two rear walls, five seats, and a hanging lamp. A perspective camera views the scene from a corner opposite the rear walls. When the original scene is rendered with the Direct Lighting strategy, →

box3. Some are provided under open-source licenses. In other cases, the scene creator may be contacted for permission to use the scene as was the case with this example. Because the Interior scene was not originally created for use with RenderToolbox3, we had to adapt it. This example introduces modeling adjustments to scenes in Blender to make them suitable for physically based rendering and RenderToolbox3 utilities that automatically help to accommodate "wild" scenes, that is, scenes produced by others.

Because the Interior scene contains complicated geometry, realistic rendering can take a long time. This example also demonstrates use of the mappings file and conditions file to control renderer behaviors, in this case to explore tradeoffs between rendering quality and rendering time.

3-D model

The original Interior parent scene was obtained from a repository of free Blender scenes produced by Nextwave Multimedia (http://nextwavemultimedia.com/html/3dblendermodel.html) and is used here with permission from Nextwave. As adapted for this example, the parent scene models a furnished interior room with a large number of shapes, three spotlights, and two large area lights. A perspective camera views the room from one corner.

The parent scene originally contained elements that RenderToolbox3, PBRT, and Mitsuba do not support, including unrealistic ambient lights, camera animation, a camera movement track, and a camera-point-of-view constraint. When exported from Blender to Collada, the scene also contained characters that MATLAB's Java-based XML parsing tools cannot process. RenderToolbox3 automatically filters out these unsupported scene elements and characters.

Direct lighting

Nevertheless, when rendered without modifications, the scene appears dark with only a few objects visible (see Figure 5). The darkness results in part from RenderToobox3's default rendering strategy, which is to consider *direct lighting* only. This strategy allows for fast rendering and previewing of a scene but ignores light that should have been reflected among objects in the scene.

← the room is dark and few objects are visible. When rendered with the Path Tracing strategy, a few more objects become visible, and some rendering noise artifacts appear as green and gray spots. After the parent scene model is adjusted to contain large Area Lights, the scene appears lighter and many more objects are visible.

Path tracing

To enable realistic interreflections, the Interior scene's mappings file specifies the *path tracing* rendering strategy instead of direct lighting. This has a modest effect and improves object visibility only slightly. In general, the effect of rendering strategy interacts with the scene specification, and changing to path tracing can have a large effect for some scenes.

Area lights

The greater cause of darkness is the original scene's reliance on unrealistic ambient lights. Because ambient light is not physically based, RenderToolbox3 filters out these lights, causing the scene to become poorly lighted. To compensate, we made modeling adjustments to the Interior scene in Blender to introduce two large rectangular area lights that resemble picture windows. These provide additional illumination, remove most of the remaining shadows, and drastically improve object visibility.

The Interior parent scene contains a large number of objects, such as chairs, cushions, walls, and wall hangings. Writing a mappings file to manipulate all of these objects would be error-prone and tedious for a user to do by hand. RenderToolbox3 provides a utility for automatically generating a mappings file based on a given Collada parent scene. For this Interior scene, a mappings file was automatically generated and then modified in order to specify the path tracing rendering strategy, the daylight spectrum for the new area lights, and other manipulations.

The Interior scene conditions file specifies several conditions that probe the effects of image size and rendering strategy on total rendering time. In this way, the Interior recipe differs from other recipes that use the conditions file to manipulate individual scene elements, not the overall behavior of renderers. We used this same recipe with PBRT and Mitsuba, yielding some general effects on rendering time. Small images render faster than large ones, and the direct lighting renderings are completed more quickly than path tracing renderings. Notably, Mitsuba seems to render more quickly than PBRT with comparable results. This may be the result of processor optimizations that Mitsuba includes (http://www.mitsuba-renderer.org/docs.html, Chapters 1 and 4.2) and PBRT does not (Pharr & Humphreys, 2010, Chapter 18.2).

The RenderToolbox3 online documentation contains further details about adapting the original Interior scene for use with RenderToolbox3 (https://github.com/DavidBrainard/RenderToolbox3/wiki/Interior-Example-Scene). The documentation also contains details about the *Dice* scene, which also uses complicated geometry.

SpectralIllusion

The SpectralIllusion example represents a RenderToolbox3 workflow that uses multispectral rendering analysis to produce a visual illusion. The executive script introduces utilities from the Psychophysics Toolbox for performing element-wise arithmetic on spectral data. Creation of the illusion builds on a physical principle and renderer behavior that we validated previously with another example called *SimpleSquare*. The SpectralIllusion example also introduces a tutorial from the Blender user community for modeling a soft-looking, rounded cube.

This scene produces a visual illusion similar to an illusion developed by Lotto and Purves (1999), in which illumination differences at different parts of a scene cause surprising perceptions of reflective object colors. The difference between this version and the original is that this version represents a realistically rendered image produced from a 3-D scene description rather than a drawn image. Thus this version may be studied using manipulations of the underlying scene variables (surface and illuminant properties) rather than manipulations made in the image plane.

The parent scene contains a cube with rounded edges modeled in Blender based on a video tutorial from the Blender user community (YouTube, accessed July 6, 2013: http://youtu.be/ssz9gcP0Ggg). Each face of the cube has 25 facets that use a matte material and have reflectances chosen so that no neighboring facets have the same reflectance. One facet on the top of the cube has a raised bump called the *target pip*. Two facets on the front right face of the cube have raised bumps, called the *destination pip* and the *reference pip*. The cube sits on a plane that uses a matte material and beige-looking reflectance.

Two lights illuminate the scene. A yellow "sun" point light is located above, behind, and to the left of the cube and emits a CIE daylight spectrum correlated to 4000 K. A blue "sky" area light occupies a large circular plane above the entire scene and emits a CIE daylight spectrum correlated to 10,000 K. The emission spectra are scaled so that neither light dominates the scene. A perspective camera views the cube at an oblique angle so as to view three faces of the cube, including one face that is shaded from the "sun" light.

The visual illusion involves the perceived colors of the target pip on the top face of the cube and the lower destination pip on the front right face (see Figure 6). Although the destination pip appears brighter/pinker than the target pip, the two pips have essentially identical RGB values (destination: [176 49 61] vs. target: [175 48 60]). The upper reference pip on the front right face, on the other hand, has the same reflectance as the target pip and appears more similar in color to the target pip even though its RGB values

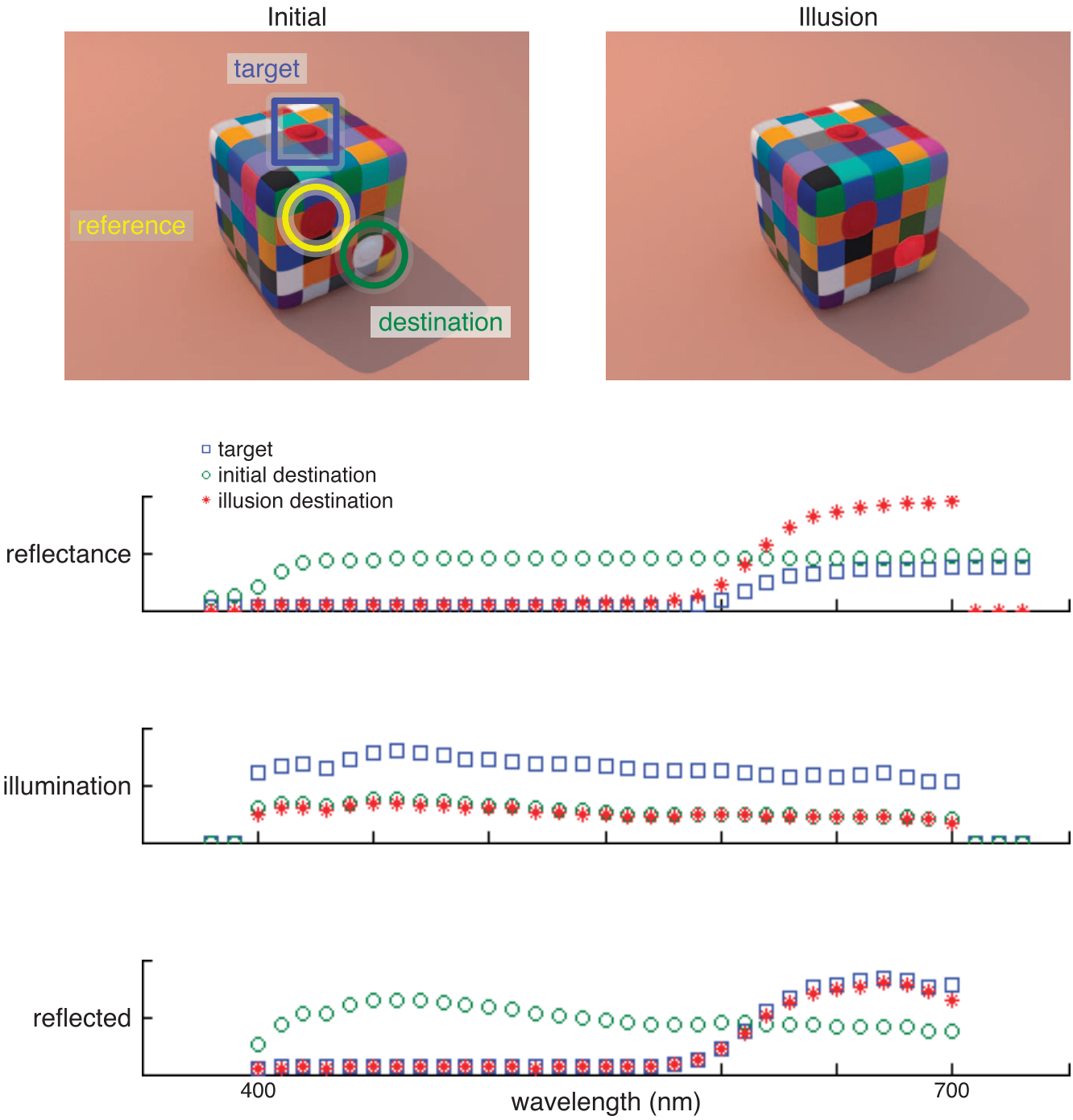

Figure 6. Summary and analysis of the SpectralIllusion. The SpectralIllusion was produced from two renderings. The top left image shows a sRGB representation of the Initial Mitsuba rendering. This initial rendering allowed estimation of the illumination arriving at the destination pip (inside the green circle). The top right image shows a sRGB representation of the Illusion Mitsuba rendering, which used a new reflectance for the destination pip that was calculated using the estimated illumination. In the Illusion image, the target pip (inside the blue square) and the destination pip have essentially equal RGB values even though the destination pip appears brighter/pinker (destination: [176 49 61] vs. target: [175 48 60]). The reference pip (inside the yellow circle) appears to have a color more similar to the target pip even though its RGB values differ substantially from the target's (reference: [115 28 40] vs. target: [175 48 60]). The second row plots the multispectral reflectance specified for the target pip (blue squares), the reflectance of the destination pip in the Initial rendering (green circles), and the calculated reflectance used for the destination pip in the Illusion rendering (red stars). The third row plots the estimated spectrum of illumination arriving at each pip. The illumination arriving at the destination pip has essentially the same spectrum in the Initial and Illusion renderings and is generally different from the illumination arriving at the target pip. The bottom row plots the final reflected radiance for each pip as read from multispectral renderings. The final reflected radiance of the target pip has essentially the same spectrum as the final reflected radiance of the destination pip in the Illusion rendering.

differ substantially (reference: [115 28 40] vs. target: [175 48 60]).

To create the illusion rendering, the reflectance spectrum for the destination pip was chosen to compensate for the difference in illumination arriving at the target and destination pips. To calculate the reflectance of the destination pip, we relied on the physical principle that the final reflected spectrum of a matte reflector should equal the element-wise product of the reflector's own reflectance spectrum times the spectrum of illumination arriving at the reflector (Wandell, 1987).[4]

Because we wish the final reflected spectra of the target and destination pips to be equal, it follows that we should choose the reflectance spectrum of the destination pip to equal the element-wise ratio of the target pip's final reflected spectrum, divided by the spectrum of illumination arriving at the destination pip.

The spectrum of illumination arriving at the destination pip is not apparent in the scene recipe, however, nor can we estimate this spectrum until the scene is already rendered. Therefore the SpectralIllusion executive script renders the scene twice. The first rendering uses an arbitrary reflectance spectrum for the destination pip and allows the executive script to estimate the spectrum of illumination arriving at that pip as the element-wise ratio of the pip's final reflected spectrum divided by the pip's reflectance spectrum. The executive script then calculates a new reflectance for the destination pip as described above and renders the scene a second time to produce the visual illusion (see Listing 6).

The RenderToolbox3 online documentation contains additional details about this SpectralIllusion example and the SimpleSquare example that we used to validate renderer multispectral reflection behavior.

Discussion

RenderToolbox3 aims to facilitate the creation of visual stimuli using computer graphics tools. To that end, it prescribes a workflow with four key steps: (a) 3-D modeling of scenes, (b) batch processing of scenes with manipulations, (c) scene rendering with physically based renderers, and (d) image processing of multispectral renderings. The toolbox contains MATLAB utilities that facilitate batch processing, rendering, and image processing. RenderToolbox3 also provides example scenes that demonstrate the functionality available at all four steps as well as that validate and document the performance of the renderers.

In this discussion, we review the RenderToolbox3 workflow and highlight toolbox features at each step that were introduced in the examples above. We also suggest potential uses for RenderToolbox3 that we have not tried, some current limitations of the toolbox, and potential extensions to the toolbox and its interactions with other software tools.

3-D modeling

RenderToolbox3 nominates the open-source application Blender as its preferred 3-D modeler. Blender facilitates the creation of simple scenes using primitive shapes, such as the SimpleSphere and RadianceTest, as well as complex scenes, such as the Interior, and scenes that import arbitrary 3-D models from external sources, such as the DragonColorChecker. Blender has an active community of open-source and professional users and enjoys many online tutorials for these and other modeling tasks as well as example projects that are free to download.

The Blender Foundation has supported the creation of open-source animated films using Blender (http://www.blender.org/features-gallery/movies/). Many scenes in these films approximate complex natural scenes and therefore may be of interest to vision researchers. Indeed, the Blender project files for these films are free to download and can allow researchers to benefit from months of film design and modeling effort (Butler et al., 2012). By incorporating Blender into its workflow, RenderToolbox3 makes projects like these films more accessible to researchers. As with the Interior example, these scenes may require modeling adjustments in Blender and other modifications in order to produce useful stimuli. We have not yet rendered any complete film scenes with RenderToolbox3.

The parent scenes included with the RenderToolbox3 distribution, as well as many of the online Blender community tutorials, are based on Blender's graphical user interface (GUI). That is, these scenes and tutorials use Blender by hand. Blender also supports a programming interface, allowing scenes to be modeled programmatically with the Python language (http://wiki.blender.org/index.php/Doc:2.4/Manual/Extensions/Python). Python is similar in many ways to MATLAB's scripting language: It is portable across computer hardware and operating systems, it uses high-level syntax with flexible variable typing, and it runs in an interactive interpreter that performs automatic memory management (http://www.python.org/). RenderToolbox3 parent scenes can be modeled programmatically, and their Python modeling scripts can be included as part of the RenderToolbox3 recipes. These modeling scripts document the process of 3-D modeling in greater detail than the parent scene Collada file itself and complement the documentation provided by the conditions file, the mappings file, and the executive script. In

```
% Listing 6

% batch processing and first rendering
%   multispectralDataFile
%   S_renderer
%   S_reflectance
%   R_destination

...

% multi-spectral reflectance calculation

% locate pixel X and Y coordinates within the target and destination pips
data = load(multispectralDataFile{1});
rendering =  data.multispectralImage;
height = size(rendering, 1);
width = size(rendering, 2);
X_target = round(width * (165/320));
Y_target = round(height * (68/240));
X_destination = round(width * (211/320));
Y_destination = round(height * (151/240));

% read final reflected spectra F for target and destination pixels
F_target = squeeze(rendering(Y_target, X_target, :));
F_destination = squeeze(rendering(Y_destination, X_destination, :));

% compute apparent illumination I at the destination pixel
F_destination_resample = SplineRaw(S_renderer, F_destination, S_reflectance);
I_destination = F_destination_resample ./ R_destination;

% compute the new reflectance R for the destination pixel
F_target_resample = SplineRaw(S_renderer, F_target, S_reflectance);
R_destination_new = F_target_resample ./ I_destination;

...

% second rendering and image processing
```

Listing 6. Excerpt from the SpectralIllusion executive script that calculates the reflectance for the destination pip. Previously, the executive script would have performed batch processing and an initial rendering, resulting in a multispectral data file. The spectral sampling used internally by the renderer S_renderer; the spectral sampling used to specify reflectances S_reflectance; and the initial, arbitrary reflectance of the destination pip R_destination would have been specified previously. The executive script loads rendering data from the initial rendering, locates pixels in the rendering that fall within the target and destination pips, and reads final reflected spectra F_target and F_destination from those pixels. The script can then estimate the spectrum of illumination I_destination at the destination pip and calculate a new reflectance for the destination pip R_destination. Note that the spectral samplings S_renderer and S_reflectance are not generally equal. Before performing element-wise arithmetic on spectra, the executive script uses the SplineRaw( ) utility from the Psychophysics Toolbox to resample final reflected spectra to match the sampling of the reflectance spectra.

addition, procedural modeling of scene geometry enables the user to generate arbitrarily complex scene components, such as stochastic elevation maps, which would be difficult and perhaps impossible using Blender's GUI. An example of using Python to model a scene is provided in the BlenderPython example (see Appendix).

RenderToolbox3 reads all parent scene data using the Collada XML schema. Collada is an open standard for exchanging 3-D models between applications like modelers and renderers. Blender's built-in Collada exporter allows users to export any Blender scene as a Collada file. Other modeling applications, such as Maya (http://collada.org/mediawiki/index.php/ColladaMaya) and Google SketchUp (http://collada.org/mediawiki/index.php/SketchUp) can also export Collada scenes although we have not tested these with RenderToolbox3. A downside of Collada is that text can be an inefficient way to specify the properties of large scenes. We view this as a cost of the portability provided by Collada.

Batch processing

The Collada XML-schema allows RenderToolbox3 to perform flexible batch processing of scenes with the help of MATLAB's built-in XML-parsing utilities. Batch processing allows the user to render the same parent scene multiple times, under various conditions, in order to produce a family of related renderings. For each condition, users may manipulate elements of the

parent scene, for example, by translating and rotating objects (see RadianceTest), by introducing multispectral reflectances and illuminant power distributions (see DragonColorChecker and SpectralIllusion), and by introducing new types of materials and light sources (see MaterialSphereBumps and Interior). RenderToolbox3 can import multispectral colorimetric data from the Psychophysics Toolbox and use utilities from the Psychophysics Toolbox to manipulate spectra. RenderToolbox 3 also provides utilities for adding textures and bump maps to the surfaces of reflective objects as in MaterialSphereBumps.

RenderToolbox3 users must write an executive script in MATLAB that binds together a parent scene Collada file with manipulations that are specified in a conditions file and a mappings file and invokes RenderToolbox3 utilities. Executive scripts are often short (fewer than 100 lines) and follow a typical pattern although they may be extended arbitrarily (as for the SpectralIllusion). Together, the parent scene Collada file, conditions file, mappings file, and executive script constitute a complete recipe that RenderToolbox3 may use to reproduce a family of renderings. The recipe also serves as documentation of the renderings, which may be accessed during the rendering workflow and during later data analysis.

Specifying object reflectances is an important manipulation for physically accurate rendering. RenderToolbox3 allows users to specify multispectral object reflectances that use arbitrary spectral sampling. Users may also specify spatially varying reflectances using bitmap textures that are based on RGB image files and wrapped around reflective objects. Unfortunately, PBRT and Mitsuba do not currently allow for multispectral textures. However, it is enticing to note that RenderToolbox3, PBRT, and Mitsuba support the OpenEXR image format, which can represent images with arbitrary spectral channels. In principle, and with some development effort, multichannel OpenEXR images could be used to specify object reflectances that are both multispectral and spatially varying.

As of version 1.1, RenderToolbox3 provides a utility that exposes how renderers promote RGB reflectances to internal multispectral sampling. This utility allows users to predict how RGB reflectances and image textures will behave during rendering even though it does not allow users to specify arbitrary multispectral textures. The RenderToolbox3 distribution includes the RGBPromotion example, which demonstrates this utility for PBRT and Mitsuba with a variety of RGB reflectance values.

The Collada XML schema can represent shapes using polygon meshes of arbitrary size and detail. But Collada files, including those exported by Blender, typically do not contain parametric descriptions of curved shapes like disks or spheres. In order to achieve smooth-looking curves, it is often necessary to use meshes that contain many vertices, which can lead to large, unwieldy Collada files. With some development effort, it would be possible for RenderToolbox3 to convert large meshes to specific parametric forms that are defined by the supported renderers. As with multispectral reflectances and illuminant power distributions and physically based materials, converting meshes to parametric forms would allow users to improve the physical accuracy of scenes during batch processing.

In addition to the batch-processing utilities described above, RenderToolbox3 defines a *Remodeler Plugin API*, which allows custom functions to modify the Collada parent scene XML document on the fly during batch processing. Remodeler functions may facilitate modifications to Collada parent scenes that would be difficult to perform otherwise. For example, a remodeler function might perform stochastic modifications or add vertices to a mesh object by performing interpolation. Modifications like these would be difficult to accomplish using a modeling application like Blender or by manually editing the Collada document.

Each remodeler plugin must comprise a remodeler name and one to three remodeling functions:

1. *BeforeAll*. A plugin's BeforeAll function may modify the Collada XML document once at the start of batch processing and may affect all renderer native scene files that are produced.
2. *BeforeCondition*. A plugin's BeforeCondition function may modify the Collada XML document once for each condition just before mappings are applied and may affect the renderer native scene file for just one condition.
3. *AfterCondition*. A plugin's AfterCondition function may modify the Collada XML document once for each condition just after mappings are applied and just before the parent scene is converted to a renderer native scene file and may affect the renderer native scene file for just one condition.

By default, RenderToolbox3 does not use any remodeler. An executive script may specify one remodeler at a time, using its remodeler name. The RenderToolbox3 online documentation describes the MaterialSphereRemodeled example, which demonstrates how to define and use a custom remodeler.

Rendering

As of version 1.1, RenderToolbox3 supports two physically based renderers, PBRT and Mitsuba, and provides MATLAB utilities that handle details of

invoking them. These renderers have largely overlapping functionality, so often the same recipe may drive each renderer. The toolbox automatically scales the output from either renderer into physical radiance units and stores the data in a consistent mat-file format. This facilitates comparison and validation of renderer behaviors and swapping between renderers.

The same batch-processing utilities that allow users to manipulate scene elements also allow users to manipulate and probe renderer behaviors. Many behaviors may be manipulated equivalently for both renderers, such as path tracing versus direct lighting rendering strategies.

But users are not restricted to the subset of behaviors that are supported by both renderers. Using renderer-specific mappings and low-level path mappings, users may specify and manipulate arbitrary renderer behaviors, including behaviors that only one renderer supports. An example of the use of renderer-specific features is provided in the *CubanSphere* example, in which different renderer-specific material types and their renderer-specific parameters are used for Mitsuba and PBRT (see Appendix).

RenderToolbox3 works with PBRT and Mitsuba by way of renderer plugins. Each plugin is a set of five MATLAB m-functions that adapt key renderer behaviors to make them compatible with RenderToolbox3. Additional plugins could be written to allow RenderToolbox3 to work with additional renderers. The five required m-functions and associated renderer behaviors are the following:

1. *VersionInfo*. A plugin's VersionInfo function must gather version information about the renderer to be stored along with each rendering produced by the renderer.
2. *ApplyMappings*. A plugin's ApplyMappings function must receive data from the mappings file and collect the data in a renderer native format.
3. *ImportCollada*. A plugin's ImportCollada function must combine a Collada parent scene file with data that was collected by the ApplyMappings function and produce a new renderer native scene description.
4. *Render*. A plugin's Render function must accept a renderer native scene description as produced by the ImportCollada function and invoke the renderer in order to produce a multispectral rendering.
5. *DataToRadiance*. A plugin's DataToRadiance function must accept a multispectral rendering produced by the Render function and convert the data to units of physical radiance.

Renderer plugin functions must conform to the RenderToolbox3 *Renderer Plugin API*. That is, they must obey specific conventions for function naming, expected inputs, and expected outputs. But the inner details of each plugin function are unspecified and may be tailored to each renderer. For example, a plugin might invoke a renderer like PBRT or Mitsuba that is an external application separate from MATLAB. Another plugin might invoke a renderer that is implemented entirely within MATLAB. Yet another plugin might invoke a renderer that is implemented as a remote web service. These implementation choices would be circumscribed by the respective renderer plugins, so the overall RenderToolbox3 workflow would be unaffected.

The Mitsuba renderer plugin almost entirely uses built-in Mitsuba functionality. The Mitsuba VersionInfo function returns a description Mitsuba's executable file, including the file creation date. The Mitsuba ApplyMappings function collects mapping data in an adjustments file that has the same format as a Mitsuba native scene file. The Mistuba ImportCollada function invokes Mitsuba's built-in Collada importer. The Mitsuba Render function invokes the Mitsuba application itself, which performs multispectral rendering internally and produces outputs as multichannel OpenEXR files with wavelength-by-wavelength granularity. Finally, the Mitsuba DataToRadiance function scales renderings by a precomputed constant (see RadianceTest).

The PBRT renderer plugin supplements built-in PBRT functionality with custom functionality. The PBRT VersionInfo function returns a description of PBRT's executable file, including the file creation date. The PBRT ApplyMappings function collects mappings data in an adjustments file that has a custom XML format. The PBRT ImportCollada function uses custom m-functions to combine a Collada parent scene file with the custom adjustments file and produce a PBRT native scene file. The PBRT Render function invokes a modified version of PBRT called pbrt-v2-spectral, which adds support for multispectral outputs with wavelength-by-wavelength granularity (https://github.com/ydnality/pbrt-v2-spectral). Finally, the PBRT DataToRadiance function scales renderings by a precomputed constant that is valid for a particular set of rendering strategies (see RadianceTest and ScalingTest in Appendix). The RenderToolbox3 online documentation provides instructions for obtaining pbrt-v2-spectral and compiling it to use arbitrary spectral sampling.

The Renderer Plugin API also allows users to extend RenderToolbox3 by developing custom plugins that are not part of the official RenderToolbox3 distribution. For example, a user might develop a modified Mitsuba or PBRT plugin for performing monochrome rendering or RGB rendering rather than multispectral rendering. Or a user might develop a plugin for an entirely new renderer. These users would not have to wait for RenderToolbox3 to officially support the same modi-

fications or renderers. Another potential use of the Renderer Plugin API might be to incorporate a binary mesh specification during the conversion of Collada to a renderer native format for a renderer that supported use of binary meshes. If such a mesh were substituted for a text mesh placeholder produced by the modeler, this could increase efficiency for handling very large meshes, albeit at a cost in portability.

A previous version of the RenderToolbox (version 2) supported Radiance, another open-source, physically based renderer. In principle, a RenderToolbox3 renderer plugin could be written for Radiance, but doing so would require some development effort. Because Radiance does not perform multispectral rendering internally, the Radiance Render function would have to invoke Radiance multiple times, once for each wavelength. This approach would incur a performance penalty because it would require Radiance to repeat time-consuming geometric calculations at each wavelength. A Radiance ApplyMappings function would need to be developed to collect RenderToolbox3 mappings data in a Radiance native format. A Radiance ImportCollada function would need to be developed to combine a Collada parent scene file with mappings data and produce a Radiance native scene file. Finally, a Radiance DataToRadiance function would need to be developed to scale renderings to units of physical radiance. It would require some investigation to determine the correct radiometric unit factor for Radiance and to determine which nonradiometric rendering parameters affect the scaling of Radiance outputs.

Renderer plugins might also be written for some commercially available renderers. Arion (http://www.randomcontrol.com/documentation?section=what-is-light) and Maxwell (http://support.nextlimit.com/display/maxwelldocs/Introduction+to+Maxwell) are two commercial renderers that aim for physical accuracy and perform multispectral rendering internally. We have not investigated these renderers in detail, including whether or not they have built-in Collada importers, whether they can produce multispectral outputs with wavelength-by-wavelength granularity, or which nonradiometric rendering parameters might affect their output scaling.

Blender itself supports two renderers, which are integrated with Blender's 3-D modeling environment. We have used these renderers to produce quick previews of scenes during modeling. We have not investigated the feasibility of a RenderToolbox3 renderer plugin for the Blender renderers. For example, we have not investigated whether the Blender renderers aim for physical accuracy, whether they can import Collada inputs from an external application like RenderToolbox3, whether they can perform multispectral rendering, or whether they can produce multispectral outputs with wavelength-by-wavelength granularity.

Image processing

The Rendering step produces multispectral data files, which are rich in information but difficult to visualize. RenderToolbox3 provides utilities to convert multispectral data to sRGB images and montages and sensor images based on arbitrary color-matching functions. These representations are better suited for review and delivery to subjects.

RenderToolbox3 also facilitates analysis of multispectral renderings themselves. The toolbox produces renderings in a consistent mat-file format, using physical radiance units. Multispectral data in this form are available for analysis with any of the various MATLAB analysis tools.

The toolbox includes specific examples of multispectral analyses that validate renderers with respect to physical principles and to each other. Such analyses can use parameters that were specified as part of the original rendering recipe. For example, analyses performed for the SimpleSquare, RadianceTest, and SpectralIllusion examples make use of the ParseConditions() utility, which reads RenderToolbox3 conditions files into MATLAB. A similar ParseMappings() utility reads mappings files into MATLAB. Providing direct access to recipe parameters facilitates detailed analysis and reduces the need for redundant data entry and the potential for data entry errors. Analyses like these allow users to build on assumptions about correct rendering behavior to create stimuli like the Interior and the SpectralIllusion that are both intricate and accurate.

Summary

In summary, RenderToolbox3 provides utilities and prescribes a workflow that should be useful to researchers who want to employ graphics in the study of vision and perhaps in other endeavors as well. This paper describes the design and functionality of the toolbox and discusses several examples that demonstrate rendering utilities and a workflow involving 3-D modeling, batch processing of a 3-D parent scene, rendering to produce a family of related renderings, and image processing to produce viewable images and do other analyses. We have designed RenderToolbox3 to be portable across computer hardware and operating systems and to be free and open source (except MATLAB). Using open-source software has allowed us to extend the toolbox, for example, by incorporating a modified version of the renderer PBRT, and invites further extensions.

This paper describes version 1.1 of RenderToolbox3. For future versions, we would welcome contributions that improve or extend RenderToolbox3 and its documentation or that enhance its interactions with other software tools. Interested users, please visit the RenderToolbox3 site. In particular, please see our list of known issues and features in progress (https://github.com/DavidBrainard/RenderToolbox3/issues) and consider joining the RenderToolbox3 team (https://github.com/DavidBrainard/RenderToolbox3/wiki/Join-Us).

Keywords: vision science, stimuli, graphics rendering, color, material perception

Acknowledgments

We thank Paul Kanyuk and Ana Radonjić for testing and suggestions. Andy Lai Lin and Brian Wandell developed and shared the spectral version of PBRT that we use. Supported by NIH R01 EY10016.

Commercial relationships: none.
Corresponding author: David H. Brainard.
Email: brainard@psych.upenn.edu.
Address: Department of Psychology, University of Pennsylvania, Philadelphia, PA, USA.

Footnotes

[1]The internal spectral sampling used by the renderers is determined when they are compiled. Spectra specified at other samplings are converted by the renderers to their internal sampling. RGB values are promoted to the same internal sampling. We compile renderers to sample the spectrum using 31 evenly spaced bins, each 10 mn wide, spanning the range 395 nm through 705 nm. This choice of spectral sampling is arbitrary and represents a tradeoff between spectral precision and rendering time.

[2]The results of the RadianceTest example do not distinguish whether renderer output is in units of reflector radiance, in units of irradiance at the camera sensor, or in units representing the linear response of camera sensor elements as these quantities are linearly related to each other for fixed camera aperture, exposure duration, and sensor properties. A second set of tests implemented in the ScalingTest example provided with RenderToolbox3 (see Appendix), however, suggests that the renderers are providing images whose intensive quantity is best thought of as source radiance.

[3]During the RadianceTest itself, the radiometric unit factor for each renderer is set to unity.

[4]The SimpleSquare example provided with RenderToolbox3 verifies this principle for both PBRT and Mitsuba with a variety of reflectances (see Appendix).

References

Brainard, D. H. (1997). The Psychophysics Toolbox. *Spatial Vision, 10*(4), 433–436.

Brainard, D. H., Pelli, D. G., & Robson, T. (2002). Display characterization. In *Encyclopedia of Imaging science and technology* (pp. 72–188). New York: Wiley.

Brainard, D. H., & Stockman, A. (2010). Colorimetry. In M. Bass, C. DeCusatis, J. Enoch, V. Lakshminarayanan, G. Li, C. Macdonald, V. Mahajan, & E. van Stryland (Eds.), *The Optical Society of America handbook of optics, 3rd edition, Volume III: Vision and vision optics* (pp. 10.11–10.56). New York: McGraw Hill.

Butler, D. J., Wulff, J., Stanley, G. B., & Black, M. J. (2012). *A naturalistic open source movie for optical flow evaluation*. (pp. 611–625). Paper presented at European Conference on Computer Vision, Florence, Italy, October 2012.

CIE. (2004). *Colorimetry, third edition (No. 15.2004)*. Vienna: Bureau Central de la CIE.

Delahunt, P. B., & Brainard, D. H. (2004). Does human color constancy incorporate the statistical regularity of natural daylight? *Journal of Vision, 4*(2):1, 57–81, http://www.journalofvision.org/content/4/2/1/, doi:10.1167/4.2.1. [PubMed] [Article]

Fleming, R. W., Dror, R. O., & Adelson, E. H. (2003). Real-world illumination and the perception of surface reflectance properties. *Journal of Vision, 3*(5):3, 347–368, http://www.journalofvision.org/content/3/5/3, doi:10.1167/3.5.3. [PubMed] [Article]

Jakob, W. (2010). Mitsuba renderer. http://www.mitsuba-renderer.org.

Kim, J., Marlow, P. J., & Anderson, B. L. (2012). The dark side of gloss. *Nature Neuroscience, 15,* 1590–1595.

Larson, G. W., & Shakespeare, R. (1998). *Rendering with radiance: The art and science of lighting visualization*. San Francisco: Morgan Kaufman Publishers.

Lotto, R. B., & Purves, D. (1999). The effects of color on brightness. *Nature Neuroscience, 2*(11), 1010–1014.

McCamy, C. S., Marcus, H., & Davidson, J. G. (1976).

A color-rendition chart. *Journal of Applied Photographic Engineering, 2*(3), 95–99.

Pharr, M., & Humphreys, G. (2010). *Physically based rendering: From theory to implementation* (2nd ed.). San Francisco: Morgan Kaufmann Publishers.

Ruppertsberg, A. I., & Bloj, M. (2007). Reflecting on a room of one reflectance. *Journal of Vision, 7*(13):12, 1–13, http://www.journalofvision.org/content/7/13/12, doi:10.1167/7.13.12. [PubMed] [Article]

Wandell, B. A. (1987). The synthesis and analysis of color images. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 9,* 2–13.

Ward, G. J. (1992, July). Measuring and modeling anisotropic reflection. *ACM SIGGRAPH Computer Graphics*, *26*(2), 265–272.

Wyszecki, G., & Stiles, W. S. (1982). *Color science – Concepts and methods, quantitative data and formulae* (2nd ed.). New York: John Wiley & Sons.

Xiao, B., Hurst, B., MacIntyre, L., & Brainard, D. H. (2012). The color constancy of three-dimensional objects. *Journal of Vision, 12*(4):6, 1–15, http://www.journalofvision.org/content/12/4/6, doi:10.1167/12.4.6. [PubMed] [Article]

Yang, J. N., & Maloney, L. T. (2001). Illuminant cues in surface color perception: Tests of three candidate cues. *Vision Research, 41,* 2581–2600.

Appendix

The RenderToolbox3 source code distribution contains many fully functional examples, which demonstrate toolbox features and workflows. As of version 1.1, these include 16 parent scenes and 31 executive scripts. This appendix lists the name of each example scene, the name or names of executive scripts that may be executed in MATLAB to generate a family of related renderings, and a short description of the parent scene and any manipulations. Online documentation about each example may be found at https://github.com/DavidBrainard/RenderToolbox3/wiki.

Parent scene name	Executive script name(s)	Scene description
BlenderPython	MakeCheckerShadowScene	Modeled procedurally using Blender's Python API
Checkerboard	MakeCheckerboard	Binocular checkerboard test pattern
CoordinatesTest	MakeCoordinatesTest	Test pattern for coordinate systems and geometric transformations
CubanSphere	MakeCubanSphere, MakeCubanSphereTextured	Camera motion about a cube and a sphere with variant that uses a textured sphere
Dice	MakeDice	textured dice depicted in mid-roll
Dragon	MakeDragon, **MakeDragonColorChecker**, MakeDragonMaterials, MakeDragonGraded	Stanford dragon model with variants that use Color Checker reflectances, various materials, and graded reflectances
Interior	**MakeInterior**, MakeInteriorDragon	Furnished room from NextWave multimedia with Stanford dragon variant
Interreflection	MakeInterreflection, MakeInterreflectionFigure	Test of reflections between objects with results summary figure
MaterialSphere	MakeMaterialSphere, **MakeMaterialSphereBumps**, MakeMaterialSpherePortable, MakeMaterialSphereRemodeled	Sphere rendered in various materials with portable and bump map variants
RadianceTest	**MakeRadianceTest, MakeRadianceTestFigure**	Test of radiometric principles with results summary figure
RGBPromotion	MakeRGBPromotionFigure	Exposition of renderer promotion of RGB to internal multispectral sampling
ScalingTest	MakeScalingTest, MakeScalingTestFigure	Test of nonradiometric renderer output scaling with results summary figure
SimpleSphere	**MakeSimpleSphere, MakeMatlabSimpleSphere, MakeSimpleSphereFigure**	Test of rendering Ward model sphere under point light and orthogonal projection with results summary figure
SimpleSquare	MakeSimpleSquare, MakeSimpleSquareFigure	Test of multispectral reflectance principle with results summary figure
SpectralIllusion	**MakeSpectralIllusion**	Visual illusion by iterative rendering and analysis
TableSphere	MakeTableSphere, MakeTableSphereFigure	Test of illuminants and reflections between objects with results summary figure

Table A1.