On Refactoring and Modernizing the Radiance Lighting Simulation Software

Gregory Ward, Anyhere Software, gregoryjward@gmail.com
Taoning Wang, Lawrence Berkeley National Laboratory, taoningwang@lbl.gov

Executive Summary

This report summarizes a three-year effort (2022-2024) to modernize the Radiance lighting simulation software codebase with a C++ interface that enhances its utility. The scope of work supported development, validation, and dissemination of a new refactored version of Radiance to accelerate deployment of accurate and performant tools for modeling solar and lighting energy at building and community scale.

The completed work successfully updated Radiance's callable interface through object-oriented C++ classes, enhanced its capabilities with hyperspectral rendering, and established a Python interface (pyradiance) for direct access to the ray tracing engine. The refactoring maintains backward compatibility while enabling novel simulation workflow development and improved software maintainability.

The audience for this work included US Department of Energy/ Building Technology Office's Building Energy Modeling tools (OpenStudio, EnergyPlus, Spawn, WINDOW, COMFEN), third-party developers of modeling tools that use Radiance for its core engine (e.g., DIVA, Honeybee, IES-VE), manufacturers, regulators, engineers, and architects. The intended use by the audience is: building energy modeling of window heat gains, low-energy cooling systems, daylighting, thermal energy storage, visual comfort, thermal comfort, indoor daylight quality, non-visual health effects, model predictive controls, and systems integration. The software is open source and is incorporated into third-party software tools.

This work was synergistic to activities conducted within the International Energy Agency (IEA) SHC Task 70/ EBC Annex 90 Subtask C.

Introduction to Radiance

Radiance is a widely used, physically based lighting and daylighting simulation and rendering system primarily employed in architectural and lighting design. Originally developed for accurate daylight and electric lighting calculations, Radiance simulates light in three-dimensional environments to produce precise numerical lighting data. The software is particularly valued for its scientific accuracy and is used by architects, lighting designers, and researchers worldwide for energy-efficient building design and lighting analysis.

Project Background and Objectives

Challenge

Radiance was originally written in a pre-ANSI C compiler environment that did not support function prototypes, and some of this legacy code has been brought forward with minimal changes. Prototypes have been added for most, if not all library modules, but the C-call interface is still somewhat antiquated, making adoption in new programming environments problematic.

The first goal was to provide a more modern set of C++ classes to encapsulate the existing C calls and global variables comprising the core rendering system. This simplifies code integration in more modern C++ and Python programming environments. A more general and modern input/output subsystem and multi-process handling was to be provided, as requested by third party developers such as LadyBug LLC and Solemma.

The second goal was to enhance hyperspectral rendering capabilities. The original Radiance implementation was limited to three-channel simulations using RGB color representations. This limitation became increasingly problematic as lighting technology evolved to include:

- LED systems with variable spectral distributions
- Horticultural lighting requiring precise UV and IR wavelengths
- Circadian and other health-based lighting applications (mould, virus inactivation)
- Advanced materials with wavelength-specific optical properties
- Research applications requiring detailed spectral analysis

The third goal was to facilitate inclusion of Radiance in complex workflows involving co-simulation via Python, which enables communication and data exchange between disparate software environments.

IEA SHC Task 70/ EBC Annex 90 Objectives

This work was synergistic to International Energy Agency investigations related to energy, comfort, and health. This included extending Radiance's capabilities to support hyperspectral rendering across extended wavelength ranges while:

- Maintaining backward compatibility with existing workflows
- Preserving computational efficiency
- Enabling new scientific and commercial applications
- Supporting emerging lighting technologies

Technical Implementation

Core Architecture Modernization: C++ Class Refactoring

The most significant achievement of this task was the wrapping of Radiance's core rendering functionality of legacy C code with object-oriented C++ classes. This modernization effort introduced a hierarchical class structure that encapsulates the operations of core rendering tools including rtrace, rpict, rpiece, rcontrib, and rfluxmtx.

New C++ Class Hierarchy

RadSimulManager (Base Class): The foundation class providing basic functionality for all Radiance operations
RtraceSimulManager (Core Rendering Class): Builds upon RadSimulManager to provide advanced ray tracing capabilities
RpictSimulManager (Picture Rendering Class): Specializes RtraceSimulManager for image generation
RcontribSimulManager (Contribution Mapping Class): Provides sophisticated contribution matrix calculations
RdataShare (Abstract I/O Class): Optimizes file and memory operations

Hyperspectral Rendering Enhancement

Extended Spectral Data Types

The refactoring introduced new fundamental data structures:

- **SCOLOR**: New spectral color type replacing traditional COLOR in ray evaluation
- **SCOLR**: Spectral color storage type replacing COLR for file output
- **Configurable sampling**: Up to MAXCSAMP wavelength samples, default 24 but expandable via build settings

New Scene Description Language Primitives

Five new primitives were implemented to specify spectral properties:

- spectrum: Direct wavelength specification
- specfile: File-based spectral data input
- specfunc: Functional spectral specification
- specdata: General data input for complex spectral datasets
- specpict: Hyperspectral image patterns (planned for future implementation)

Enhanced Command-Line Interface

New options provide flexible control over spectral rendering:

- -cs N: Enable spectral rendering with N wavelength samples
- -cw nmA nmB: Define spectral bandwidth from nmA to nmB nanometers
- -co: Output full spectral data instead of converting to RGB
- -p chromaticities: Specify custom color space primaries
- -pxyz: Output in XYZ color space format

Spectral-RGB Integration

The system maintains compatibility with existing RGB workflows through tuned spectral partitioning:

- **Red channel**: Longest wavelengths to red/green partition
- **Green channel**: Red/green partition to green/blue partition
- **Blue channel**: Green/blue partition to shortest wavelengths

This mapping preserves the familiar RGB workflow while enabling precise spectral control when needed.

File Format Enhancements

A new picture format "Hyperspectral Radiance Image" was implemented featuring:

- N spectral samples per pixel
- Wavelength range metadata in headers
- Component count specifications
- Backward compatibility with existing Radiance tools

Python Integration: pyradiance

A critical component of this subtask was the development of **pyradiance** (https://github.com/lbnl-eta/pyradiance), a Python interface that exposes the refactored Radiance C++ classes for direct access to the ray tracing engine. This integration represents a significant advancement in making Radiance accessible to modern computational workflows.

Key Features of pyradiance

Direct Ray Tracing Access: Python developers can now directly interface with Radiance's core ray tracing engine without shell command dependencies
Novel Simulation Workflows: The Python interface enables entirely new approaches to lighting simulation
Modern Development Practices: pyradiance brings Radiance into contemporary software development

Key Achievements

Software Architecture Modernization

- Complete refactoring from legacy C to object-oriented C++ classes

- Hierarchical class structure encapsulating core rendering operations
- Clean separation of concerns with specialized manager classes
- Thread-safe implementations with improved multiprocessing capabilities

Hyperspectral Rendering Capability

- Support for wavelengths extending beyond the visible spectrum
- Configurable spectral resolution (typically 24 20 samples)
- Maintains photometric accuracy across extended ranges
- Enhanced material systems with precise spectral definitions

Python Integration and Accessibility

- Development of pyradiance for direct Python access to ray tracing engine
- Elimination of shell command dependencies for Python workflows
- Integration with modern scientific computing ecosystems
- Support for novel simulation workflow development

Enhanced Performance and Scalability

- Near-linear scaling on multi-core systems through improved multiprocessing
- Memory-mapped file operations for efficient large-scale computations
- Generalized I/O operations through abstract RdataShare class
- Progressive calculation capabilities with recovery mechanisms

Backward Compatibility and Workflow Preservation

- Existing RGB workflows produce identical results
- No performance penalty for traditional rendering operations
- Maintained scientific accuracy and established photometric principles
- Seamless transition for existing user base

Applications and Impact

- **Enhanced Daylight Analysis**: More accurate simulation of natural light throughout the day
- **LED Lighting Design**: Precise modeling of modern LED systems with complex spectral outputs
- **Energy Optimization**: Better prediction of lighting energy consumption
- **Circadian Lighting**: Modeling of lighting systems designed to support human circadian rhythms

Conclusion

The Radiance refactoring work in collaboration with subtask C IEA Task 70 represents a comprehensive modernization and enhancement of the Radiance lighting simulation software. This refactoring effort has successfully transformed Radiance's architecture while maintaining its scientific integrity and user accessibility. The key accomplishments include:

1. **Modernized Software Architecture**: Wrapping of legacy C code with object-oriented C++ classes, improving extensibility and modern code compatibility
2. **Python Integration**: Development of pyradiance providing direct access to ray tracing engines, enabling novel simulation workflows and integration with modern computational ecosystems
3. **Enhanced Performance**: Improved multiprocessing capabilities with near-linear scaling on multi-core systems and optimized memory management
4. **Hyperspectral Capabilities**: Extended rendering beyond traditional RGB limitations while preserving backward compatibility

The completed work positions Radiance as both a preservation of established scientific computing excellence and a platform for future innovation. The C++ class refactoring provides a stable foundation for continued development, while pyradiance opens new possibilities for researchers and developers working in Python environments.

The work creates a pathway for Radiance to remain relevant and valuable as computational workflows increasingly move toward integrated, multi-disciplinary approaches that combine lighting simulation with optimization, machine learning, and data analysis.

Acknowledgments

This work was funded by the Assistant Secretary for Energy Efficiency and Renewable Energy, Building Technologies Office of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, with special thanks to Amir Roth, U.S. Department of Energy, for his dedicated support for this research.

Presentations/ Tutorials

Presentations at the 22nd International Radiance Workshop, Salt Lake City, Utah, August 26-28, 2024. https://www.radiance-online.org/community/workshops/2024-salt-lake-city-ut

- Ward, G., Wang, T., New hyperspectral rendering capabilities in Radiance: Workshop Tutorial.
- Wang, T., pyradiance and frads: Workshop Tutorial.
- Ward, G., Wang, T., What's new with Radiance.
- Ward, G., Wang, T., Radiance C++/ Python interface development and open discussion.

Appendix A: New C++ Radiance Classes

Greg Ward, Anyhere Software

We have introduced a new set of C++ classes that encapsulate the operations of the core rendering tools, including **rtrace**, **rpict**, **rpiece**, and **rcontrib**. In fact, we have functional "replacements" for each of these to demonstrate the new code, respectively named **rxtrace**, **rxpict**, **rxpiece**, and **rxcontrib**. These are included as optional builds in the "src/rt" directory available via **rmake** by naming them as targets.

We start by listing each of the relevant C++ header files and the classes declared in each, including a brief description. We follow this with an explanation of each new tool, their source files, and differences from their forebears.

C++ Header Files

These are listed in the order they were created, which matches the order of inheritance.

Classes in "RtraceSimulManager.h"

This header declares the basic classes used in all other Radiance classes.

RadSimulManager

This is the base class that provides basic functionality, such as loading an octree, manipulating the information header (metadata), starting and stopping threads, putting individual rays into the queue, waiting for results, and cleaning up. This class is not used directly by any tool, but provides a basic encapsulation of the raycalls.c and raypcalls.c library functionality. It forms the base class for *RtraceSimulManager*, which is parent to the other rendering classes.

RtraceSimulManager

This class builds on top of the public parent class above, and adds functionality for queuing a bundle of rays and FIFO (ordered) ray calculation. Methods are provided to assign call-back functions for traced and completed rays, which fit the *RayReportCall* prototype. Computation flags control whether FIFO is used, as well as immediate irradiance calculations (-I+ option) or ray distance limiting (-ld+), which happen to be mutually exclusive.

Classes in "RpictSimulManager.h"

This header really only defines one public class, which inherits from and protects the *RtraceSimulManager* class just described.

RpictSimulManager

Besides exposing select parent-class methods related to octree loading, header manipulation, threading, and so on, this class provides scanline- and tile-based rendering functionality for producing single and sequential picture output. All of the important operations needed by **rpict** as well as **rpiece** are provided, plus the ability to directly output hyperspectral pictures in common-exponent as well as binary float data.

Depth maps may be produced as either float data or 16-bit encoded values a la "src/common/depthcodec.h".

The class is written in such a way that memory-mapped output files are simple to support, and we use these in our implementation of **rxpiece**. Picture and depth map recovery is also supported for both scanline and tiled output.

Classes in "RcontribSimulManager.h"

There are two classes defined in this header, *RcontribOutput* and *RcontribSimulManager*. The first is just an adjunct to the second that provides external access to open output files.

RcontribOutput

Each object in this class is associated with a particular output file or channel, which provides random access via the abstract *RdataShare* class described below under "RdataShare.h". Only a single row is targeted at a time, though the output may be written in any order.

RcontribSimulManager

This class derives from *RtraceSimulManager*, but exposes just a few of the original member functions for loading the octree and manipulating the header. Multiprocessing is handled very differently in this class, even if the controlling calls are the same. The outOp and cdsF member variables determine how output channels are opened using an *RdataShare* object. The default is to open memory-mapped files as new, failing if any output exists. Forced overwrite may be selected instead, or data recovery mode. ASCII and stdout are not supported to facilitate multiprocessing and data indexing.

In general, a bundle of "accum" rays are sent to the ComputeRecord() method until all records have been sent, at which point the output will be complete, and FlushQueue() or Cleanup() may be called.

A novel feature is provided by ResetRow(), which rewinds the calculation to the indicated record. Combined with shared access to output data using *RdataShare*, this enables progressive calculation by copying the computed values before the rewind operation, optionally changing the "accum" variable to average more rays per record. Note that the RDSread flag should be OR'ed to the appropriate RSDOflags entry for read/write access.

Classes in "RdataShare.h"

The abstract *RdataShare* class provides methods for random access on an i/o channel, and is designed to optimize memory-mapped and regular file access in the derived *RdataShareMap* and *RdataShareFile* classes, respectively. Flags indicate whether the channel is opened read-only, read/write, or write-only, and determine if the contents are retained or clobbered. Some combinations are forbidden, and it is wise to set the expected file size at the outset, especially with the memory-mapped class. Operations are kept simple – the named channel is opened by the object constructor and closed by the destructor. Supported operations are GetMemory() to allocate/retrieve a buffer for the channel and ReleaseMemory() to free/write the buffer. A Resize() method is provided to change overall channel length, and this will synchronize the object with the actual channel length if passed a 0 argument (the default). Expanding the

file while one or more buffers are checked out may fail for memory-mapped files, and resizing is not allowed on read-only channels.

New C++ Rendering Tools

As mentioned in the introduction, new C++ implementations of **rtrace**, **rpict**, **rpiece**, and **rcontrib** have been written to test the classes just described. We note differences between our new tools and the originals below, highlighting class features that enable new functionality. Limitations are discussed as well.

rxtrace

The feature list for **rxtrace** is essentially the same as **rtrace**, minus support for persistent modes, which are rarely (if ever) used.

Implementation files include "RtraceSimulManager.cpp", "rxtrace.cpp", and "rxtmain.cpp".

rxpict

We lose the persistent and parallel-persistent modes in this tool as well, which *are* used by **rpiece** but not by the new **rxpiece** replacement, described next.

We add flexibility to the output types, including hyperspectral pictures and 16-bit encoded depth maps. The biggest addition is the –n multiprocessing option, which required the **rtpict** script to run **rtrace** in the standard C tools.

Besides "RtraceSimulManager.cpp", "RpictSimulManager.cpp" and "rxpmain.cpp" are sources for this tool.

rxpiece

This C++ tool calls the *RpictSimulManager* class methods to support tiled output to a memory-mapped destination, or pair of destinations if depth output is specified. A tally sheet is kept after the final output scanline, flagging which tiles have been started and which are finished in order to simplify recovery should the program die or be interrupted by the user.

In addition to supporting hyperspectral and encoded depth output, **rxpiece** provides a -n option directly, rather than requiring multiple invocations like **rpiece**. The downside is that distributed rendering on a shared network drive via a sync file is not supported, though this is not much used in today's computing environment.

This tool uses the same implementation files as **rxpict**, substituting "rxpmain.cpp" with "rxpiece.cpp".

rxcontrib

Of all the tools, this implementation is least similar to the original. All output files are memory-mapped, an approach that works well on Unix systems, but has not been tested under Windows. ASCII output is

not supported, nor is sending results to the standard output due to the abstract channel definition described using the RdataShare class. The -c 0 functionality is also dropped in the new tool.

The main advantage to **rxcontrib** is a better multiprocessing model, which extends to more cores with near-linear speedup. The new output model means the parent no longer manages file writes, which are taken over by the children via shared maps. Since file size is determined at the outset, **rxcontrib** uses the "NROWS=" header setting to track results progress rather than relying on a truncated file during recovery.

The "RtraceSimulManager.cpp" module is used as with the other tools, and "RcontribSimulManager.cpp" builds on it, with "rxcmain.cpp" implementing the main() function to handle options and run the simulation.

Conclusion

Our hope is that developers will adapt and build tools, GUIs, and APIs using these new C++ classes, providing us feedback on what works and what doesn't, so we may improve this interface over time. We are happy to assist with any queries, bug reports, and requests along these lines. While we may add more C++ code "under the hood" in the near future, the functionality provided by this initial set of classes should serve as a stable basis for development.

The C++ class implementations described do not fully encapsulate our rendering library inasmuch as global variables still control many aspects of the calculation, and only a single octree may be loaded at a time, associated with a single rendering object. This is not a limitation we expect to disappear anytime soon, unfortunately.

Appendix B Spectral Rendering Plan

Proposal for Spectral Rendering in Radiance 6.0

Greg Ward, Anyhere Software

Goal

Add ability to render over extended (hypervisual) spectra up to some maximum number of wavelenth samples (nominally 24 but may be overridden with compiler "-DMAXCSAMP=" setting)

Plan

Add new primitives to scene description language that confer hyperspectral color to modified materials. This will include a simple in-line primitive ("spectrum") that gives starting and ending wavelengths and regularly-spaced spectral samples. There will be an equivalent primitive ("specfile") for taking spectrum from a data file, as well as a functional specification primitive ("specfunc"), and a more general data input primitive ("specdata"). Finally, we provide a spectral pattern primitive ("specpict"). These last two will probably be added at a later date.

Spectral rendering is enabled with a command-line option "-cs N" where *N* is the number of spectral samples (N>3). A second option, "-cw nmA nmB" specifies the spectral bandwidth limits in nanometers. The RGB primitives corresponding to "radiance" are in watts/sr/meter^2/(133nm) to correspond to the original defined spectral range that also gives us the standard luminous efficacy of 179 lumens/watt. Increasing the covered spectral range will effectively increase the energy stored in the R and/or B components to maintain a consistent simulation. Output pictures and values will be converted to standard RGB primaries unless the "-co" option is also specified, which triggers spectral output in *N* wavelength bands. Output color space may be set instead using a "-p" option with primary chromaticities, or "-pxyz" for Radiance_rle_xyze format.

Code Changes

New SCOLOR and SCOLR types are introduced in "common/color.h", and these are used in place of COLOR during ray evaluation and COLR for storage and output. A new picture format, "Radiance_rle_spectra" will be created with N spectral samples, starting and ending wavelengths ("WAVELENGTH_SPLITS") and NCOMP also specified in the header.

When a spectral color is multiplied by a tristimulus (RGB) color, the red, green, and blue channels apply to spectra as partitioned by WLPART[]. Red goes from the longest wavelength held in the global variable WLPART[0] to a red/green paritition wavelength at WLPART[1]. Green will continue to the green/blue partition wavelength at WLPART[2], and blue goes from there to the shortest wavelenth given by WLPART[3]. (To maximize accuracy, all 3 RGB channels should therefore be equal in all spectrally defined materials, and only patterns will specify spectra.) The supporting routines will be added mostly

to "common/color.c" and "common/spec_rgb.c" The actual units for radiance in an RGB color correspond to watts/sr/meter^2/(133 nm) corresponding to the original (default) visible range of 780 to 380 nm.

Ideally, the spectra specified in the Radiance scene file will match range and resolution with the "-cs" and "-cw" options on the command line, which will optimize accuracy and calculation time, especially for the "specfunc" and "specdata" primitive types. Spectral definitions that fall short of the extrema will be zero-filled.

Hyperspectral pictures may be used as patterns via the new "specpict" primitive, which will require additions and changes to the "rt/data.c" module. Obviously, we will need new modules for the other primitives, also.

[Footnote: I abandoned plan to redesign input language using extensions to MGF due to entailing requirement to rewrite almost the entirety of Radiance.]