Interactive Lighting Simulation for Theatrical Lighting Design

Alissa Feldman (alissaf@seas), Matt Gruskin (mgruskin@seas)
Faculty Advisor: Dr. Norman Badler
April 18th, 2008

1 Abstract

The goal of our project is to create lighting simulation software that is practical and useful for theatrical lighting designers. Our software allows the user to import 3D models and material data from external 3D modeling software, and then gives the user extensive control over how these 3D models are lit using simulations of theatrical lighting instruments. By using a variety of rendering techniques for different tasks, we achieve a more interactive user experience than is provided by other similar software.

The user interface uses standard OpenGL lighting to provide inaccurate but real-time feedback of the effects of light position, focus and color. In addition, we implemented a new relighting-optimized radiosity renderer of our own design, which provides fairly accurate and highly interactive feedback to changes in lighting instrument intensity and color. The software also provides very high quality non-real-time renders using Radiance to produce very accurate images of how a lighting design would appear in the real world.

2 Related Work

2.1 Radiance-based Visualization (LightLab)

One existing system which prompted us to begin investigating this project is the Virtual Light Lab (Performing Arts Shop, 2007) software developed and used by Peter Whinnery of the University of Pennsylvania Theatre Arts department. This software is used by students in the Concepts of Lighting Design class to simulate stage lighting by placing, focusing and coloring lighting instruments, and then generating rendered images by adjusting the intensities of these instruments. The software is currently useful as an educational tool, but certain limitations prevent it from being practical for more frequent real-world use.

A major goal of our project, as an improvement to the LightLab software, is to balance the need for rendering accuracy with the benefits of interactivity. LightLab uses the Radiance renderer (Larson & Shakespeare, 1998) exclusively, which is physically accurate and produces very realistic images, but is also quite slow. The accuracy of Radiance is excessive for tasks such as focusing, where features such as global illumination are unnecessary, but interactivity would be very beneficial. By providing a focusing interface using hardware-accelerated OpenGL instead of Radiance, the existing several-second turnaround time for each focusing change can be replaced with an interactive mouse-driven focusing interface, which would save the user a significant amount of time and provide a better user experience.

The process of writing cues is another task where the use of an interactive renderer would be beneficial. Each cue consists of a set of instrument intensities and colors designed to create a specific look on stage. During cueing, features such as global illumination and shadowing are important, which implies that a slower, accurate renderer such as Radiance should be used to achieve acceptable results. There are, however,
assumptions we can make during cueing that traditional renderers do not make - specifically, that the positions of scene objects and lighting instruments do not move. The rendering process can be greatly optimized by taking advantage of this fact. Our radiosity-based renderer, by parameterizing light intensities and colors during a preprocessing phase, is able to produce mostly accurate images at interactive rates (see Radiosity-based Rendering section below). Highly interactive and fairly accurate rendering is a useful educational tool for students as well as a practical tool for experienced lighting designers.

Other helpful feature additions could include external DMX-512 input (for controlling lighting with the console which will be used for the actual show), dynamic reconfiguration of set pieces for each scene and storing multiple scene configurations (in this software one scene is fixed), and easy import of theater and scene models from external 3D modeling software (right now a few theaters on campus are hard-coded into the software). Our software implements external model importing and a simple scene configuration interface, but unfortunately we do not support stored scenes, CAD importing, or DMX control at this time. Despite these shortcomings, we believe our software could be a useful and educational tool for students in the Concepts of Lighting Design class, which they could continue to use after completing the class to assist with their future lighting designs.

2.2 WYSIWYG

Another major influence on our project is the commercial software product WYSIWYG (Cast Group of Companies Inc, 2006), made by Cast Lighting. WYSIWYG is the standard lighting previsualization software for the theater and event industries, and has extremely powerful features for modeling a 3D world, keeping track of thousands of lighting instruments, simulating intelligent lighting, and much more. With all this power, though, comes a steep learning curve as well as a steep price tag.

Our project is not looking to compete with WYSIWYG, but rather to bring some of its benefits to a more amateur and educational environment. While WYSIWYG’s interface is designed for experienced experts to manage their highly complex designs, our interface is focused on quickly seeing the results of changes you make. Easy and fast feedback allows less experienced designers to make better decisions.

2.3 3D Modeling Software

Our project has a very different goal than 3D modeling software such as Maya or 3D Studio Max, which also have facilities for simulated lighting. These programs are designed for creating rendered images as the final product, and as such, accurately reproducing the properties of real-world lighting instruments is not their intended use. Our project’s rendered images are intended to model real-world lighting as closely as possible, as the end goal of using our software should not be to produce a rendered image, but rather to assist a designer to light a real-world scene viewed in person or captured by a camera.

One difference between our project and 3D modeling software which helps us accomplish this is the use of photometrics data provided by the manufacturers of lighting instruments. While 3D modeling software has generic light sources, our light sources use photometric data to model the specific lighting instruments which will be used in the real world.

2.4 Radiosity-based Rendering

Radiosity rendering (Telea, 1996) is a rendering technique which is capable of producing more realistic and accurate images than traditional ray tracing renderers, especially when diffuse reflection is most prevalent (which is the case for theatre). Radiosity models light transfer in the direction of light flow, simulating the actual physical behavior of light - as opposed to ray tracing, which follows light backwards from the viewer’s eye. We have implemented a radiosity renderer optimized for the task of theatrical light previsualization.
Previous work on relighting using radiosity (Nielsen & Christensen, 2002) has demonstrated the impressive render times that can be achieved by pre-computing radiosity once, then applying actual light intensities and colors in a second reconstruction step. The radiosity form factor calculations can also be hardware accelerated to reduce processing time during the radiosity calculation phase (Elias, 2000). The radiosity calculations are also view independent, allowing the camera position to be changed in real time and allowing the user to interactively view their lighting design from different angles.

3 Technical Approach

There are many discrete components which together make up our project. The largest division is between the Python front-end, which implements the user interface, data model, and calls the various renderers, and parad, our radiosity renderer back-end implemented in C++ with Python bindings.

3.1 User Interface

The graphical user interface is the user’s primary method of interaction with our project. The two main goals of the user interface are to make it easy to use and to give the user nearly instantaneous feedback. This was accomplished by making a simple interface in which all of the features are easily accessible. There are no menus or other hidden controls. To show all of the controls in the given space, we created two tabs for two different stages of lighting design. This presents the user with the most efficient controls for the aspect of their design they are currently working on.

The focusing tab allows the user to add and remove lights and objects, as well as control their properties and position within the scene. Once the scene and lights are set up, the user moves to the cueing tab where the intensities of the lights can be changed. The benefits of this separation are twofold. First, once the scene is created, a lighting designer wants to see how changing lights affects the scene and does not need to change the scene itself. Second, by changing only the lights’ intensities and colors, we can take best advantage of our radiosity renderer to show changes to the scene both accurately and in real time, which is not possible with either the OpenGL renderer or the Radiance renderer.

3.1.1 Focusing Tab

The focusing tab (see Figure 1) contains several different views of the scene. One portion of the window contains an orthographic view from above. This view is used to select and place scene objects (such as scenery and actors) as well as lights. We have added many interactive mouse features and implemented picking to provide an intuitive user experience. The user can left-click on objects and drag them to translate them in the X and Y directions. The user can also middle-click anywhere on the stage and drag to view different parts of the stage. In order to zoom in and out of the stage, the user can right-click and move the mouse up (zoom in) or down (zoom out).

When the user left-clicks on an object, a form appears on the upper right, where they can adjust the following settings:

- **Translation** - This translates an object or light along the Z axis, moving it up or down. It is controlled by a spin control with a text box so that the user can either use the arrow buttons or enter a number into the box to set the height.

- **Rotation** - Both objects and spotlights can be rotated, but they are rotated in different ways. Spotlights are rotated using the spotlight view (discussed below) while objects are rotated in the settings panel. Rotations are performed by choosing an axis of rotation and then controlling the angle of rotation.
Figure 1: The focusing tab is used to position lights and mesh objects within the scene, as well as to control properties such as material colors, light instrument types and gel colors.
around that axis. The user first sets the axis and then uses a dial to rotate. Since no such dial existed in the wxWidgets library, we created a RotatorWidget, which allows the user to interactively change the angle of rotation and see the changes in the different scene views while dragging.

- Scale (objects only) - The text boxes allow the user to change the length, width and height of an object in the scene.

- Materials (objects only) - An imported object may have several different materials for its faces. The user can change the color of a material by right-clicking on an item in a list control and selecting the color from a dialog. Unfortunately, the names of the materials are not editable at this time, so the names of the old colors persist in the control - but the background of the material name in the list control will change to the new color.

- Gel color (lights only) - The user can choose a gel from a drop down menu in order to change the color of the light. This setting simulates the situation in which the lighting designer will put a gel into a light to change its color. A library of common gel colors is included with the program, and new gels can easily be added by editing a simple python script.

- Intensity (lights only) - The user can change the intensity of the light (from zero to 100) by moving a slider. This slider is synchronized with the matching slider on the cueing tab. Furthermore, as the user moves the slider, they can see changes in both the spotlight and camera views.

- Instrument (lights only) - Each light has an instrument type that determines its beam angle, field angle, lumens, light distribution and color temperature. The user can choose the instrument type from a drop down menu. A library of common lighting instruments is included with the program, and new instruments can be added by editing a simple Python script - for the instrument to also work in Radiance, though, an IES instrument definition must exist as well.

- Delete - Clicking this button removes the object or light from the scene.

Since some settings are only relevant for lights and others only for objects, we create a new panel for each entity when it is selected. This also allows the settings panel to reflect any changes made since the object was last selected. We chose this approach rather than modifying a single settings panel because it gave us more flexibility in creating different types of entities. Any time a new entity is implemented, a matching settings panel can be implemented as well. The settings panels are inherited in the same structure as the entity types so that there is little code duplication.

The lower left portion of the window contains a 3D projection view from the point of view of the currently selected spotlight. By left-clicking in this window and dragging the mouse, the user can pan and tilt the light, just as one would if one were adjusting a spotlight in a theater.

A third display, on the lower right, shows the view from the perspective of any point in the scene. The user can change the camera’s viewpoint by left-clicking and dragging in this display. As with the orthographic view from above, the user can zoom in and out using the right-click button.

All of the focusing interface views are rendered using OpenGL so that the results of focusing can be seen in real time. The more accurate renderers are used only in the cueing panel so that once the user has placed all of their lights and objects, they can see a more realistic picture.

At the top of the tab is a tool bar with buttons for adding new objects and lights. The objects and lights are placed at the origin (the center) of the scene but can be moved with the mouse. The objects and lights are created with the default settings, which can then be changed using the settings panel. The decision to add objects on the focusing tab differed from our original idea, which was to have another tab to implement this feature. When we started to implement that design, we realized that this third tab’s functionality had
Figure 2: The cueing tab is used to view the effects of changing light intensities. The three 3D views (from left to right) are the unrealistic OpenGL-lit view, the radiosity rendered view, and the Radiance rendered view. The radiosity parameters can be found to the right of the radiosity view.

a lot in common with the focusing tab and it made more sense to be able to add objects and lights right into the focusing tab. We had also planned to implement a feature to save and load scenes, which would have been available through buttons on the tool bar, but we were not able to implement it in time. Despite this missing feature, the focusing tab is a quick and useful interface for a lighting designer, and a definite improvement over previous work.

3.1.2 Cueing Tab

The cueing tab (see Figure 2) provides intensity control sliders for each light, much like a conventional lighting console. It also provides displays for cue visualization, which show images generated from the OpenGL, radiosity and Radiance renderers. The main purpose of this tab is to allow the user to view and compare the images from the three renderers. On this tab, the user has the ability to change intensities and see the results accurately and in real time using the radiosity renderer (once the preprocessing has been completed).

The OpenGL rendered camera view, on the upper left, is still available and is synchronized with the scene in the focusing tab. This allows the user to roughly see intensity changes in real time, without preprocessing the radiosity renderer or rendering with Radiance. There is a button to render with Radiance and see the scene more accurately, but once Radiance is run, no intensity or view changes will be made in the Radiance
view until running Radiance again.

The radiosity renderer, on the upper right, gives the user the best of both worlds because they can see an accurate image but still change the light intensities. These intensities can be changed on the bottom of the tab with the sliders. Intensity changes will be reflected in the settings panel if the user moves back to the focusing tab.

3.1.3 Bringing it all together

Later in our paper, we talk about how we used wxWidgets with Python so that the program could be run on both Windows and Unix. However, since wxWidgets uses the native graphical user interface widgets, we still had some compatibility issues between platforms. For example, we originally used the mouse scroll wheel to zoom in and out on the various views on our interface. However, the mouse's coordinates on the screen are recorded differently on Windows and Unix. On Windows, the coordinates are based on the entire window but on our Linux machine, the coordinates were local to the inner panel, so it was difficult to tell which panel the user was scrolling. We solved this problem by implementing the zoom feature with the right-click button instead.

In order to accommodate both various platforms and different screen resolutions, we tried to make the interface as modular and flexible as possible. Instead of giving widgets fixed positions and sizes, we used layout sizers to place widgets on the screen. This prevented parts of widgets from being cut off and allowed the widgets to expand as the user expanded the window size.

One of the biggest challenges with the user interface was making sure that changes made in to an object or a light were reflected immediately in all areas of the interface. For example, if an intensity is changed in the focus panel, the matching slider in the cueing panel needs to be changed as well. Furthermore, if a setting is changed, all of the views need to be redrawn to reflect the change. It’s possible that a different design of our user interface code, perhaps using events to trigger updates, could have made managing these updates easier.

3.2 Data Model

A large component of our project is the object model which represents the meshes and lights within a scene. These classes encapsulate all of the properties of the meshes we are lighting (such as position and orientation, materials, and the object models themselves) as well as the properties of the lights (such as lighting instrument type, position, focus, color, beam properties, etc.) We had also planned to manage other configurable parameters in this layer, such as different scene object configurations and light intensity levels for stored cues - unfortunately these features are not yet implemented, but could be easily included in a future version.

We store the instrument type definitions and gel colors in two Python source files, using Python lists, tuples and dictionaries. These files can be easily edited to add new instruments and gels. We chose this technique over explicitly loading data files due to its simplicity and ease of implementation. In theory, the user would require a knowledge of Python to edit these files, but in actuality the syntax is so simple that anyone should be able to add and modify instruments and gels.

We have implemented importers for Wavefront OBJ models and their associated material files, which are then loaded into our mesh object classes. Each object stores its own materials, and these materials can be easily edited by the user to create simple variations of stock objects. We do not currently support the saving and loading of entire scenes, which would need to be implemented to allow users to save their work between uses of our program.

We succeeded in separating our data model and program logic from the user interface - the scene objects are loaded and exist separately from the GUI, and do not depend on the GUI except when they are drawing
themselves. This separation could allow for features such as command line or script-based access to our program to be added in the future, but we experience the benefits now as this helps to keep the source code clean and easily understandable.

3.3 OpenGL Renderer

We use an OpenGL-based renderer (Shreiner et al., 2005) for user interactions which benefit the most from real-time rendering, and do not require shadowing and global illumination. We include several different views into the scene, including a top-down orthographic view providing picking and transformation features, a perspective camera view for viewing the scene, and a perspective view designed especially for light instrument focusing.

The OpenGL lighting model has a limited number of lights, but by using ‘virtualized light sources’ (Kilgard, 2007) it is possible to produce real-time images using many lights, which could be useful for light focusing. For example, most OpenGL implementations only provide 8 lights, but there could be a system of 10 lights which it would be useful to focus simultaneously. The virtualization technique would help in this situation because not all of the lights are focused on the same areas of the scene - by enabling only the 8 lights which affect the current area the most, 10 simultaneous lights can be simulated.

Instead of implementing ‘virtualized light sources’, we used the simpler approach of only enabling the eight lights with the highest current intensity, but always making sure that the light the user is currently working with is included. This gives the user full control over the lights they are seeing at any given time, which is sufficient for the focusing task.

We explored implementing many different approaches for object picking in the top-down orthographic view, including using the OpenGL selection mode and manually projecting mouse clicks back into the scene as rays. These approaches were complicated and never worked quite right. We eventually settled on using a simpler technique called “Object Selection Using the Back Buffer” described in the OpenGL Red Book (Shreiner et al., 2005). The object tagging used by this technique is actually quite similar to how patches are tagged in our radiosity renderer. In the end, this technique is serving our purposes very well.

3.4 Radiance Renderer

We support Radiance as an external renderer for our scenes. The input to the Radiance tools consists of text files describing scene geometry, material properties and lighting properties. We implemented a module in Python which converts our internal representation of this data into Radiance’s nonstandard format, and then runs the appropriate Radiance command line tools to generate a final rendered image. This image is then loaded into our program and displayed to the user, integrating the external Radiance tools with our software.

Some parts of the Radiance input files are not always generated directly from our software with each render, but were written and set up ahead of time and included with our software as data. These include files specifying the photometric properties of common theatrical lighting instruments and color gels. Chapter 8 of Rendering with Radiance (Larson & Shakespeare, 1998) provides guidelines that we used for configuring Radiance for theatrical lighting, and the existing LightLab software (Performing Arts Shop, 2007) also served as a working example of the input that should be provided to Radiance.

We faced a few challenges while developing a process to convert our program’s internal scene representation into Radiance files. Our lighting instruments are focused in the positive Y direction by default, while Radiance’s lighting instruments are focused in the negative Z direction by default, and the way Radiance handles rotations as a sequence of X, Y and Z rotations is different than our axis-angle rotation model. To handle this conversion, we implemented some tricky rotation conversion code in our Radiance spotlight
exporter. We also had to implement some path and file management code to handle the many temporary intermediate files generated by various Radiance tools in a platform-agnostic way.

### 3.5 Relighting Radiosity Renderer

A light cue design application has certain properties which allow us to create a renderer optimized for cueing. Specifically, while writing a cue, the scene geometry will not change at all, and the lighting fixtures will not change focus - only the intensities of the lights will change. It is wasteful of computer time and user time to render an image from scratch for each light intensity change. Instead, we can design a renderer that leverages the fact that many variables are fixed to achieve interactive frame rates for visualizing lighting cues.

We have implemented a radiosity renderer (also referred to as 'parad') with special optimizations for frequent changes in light intensity and color, but rare changes in scene geometry. To achieve interactive frame rates, we break the rendering down into two phases. Both phases of parad are implemented in C++, and exposed to our Python front-end using a SWIG interface. Input is provided to parad using a set of polygon and light creation functions which are called from our Python front-end. Parad’s output can be viewed by calling a provided OpenGL drawing function, which will draw the lit parad scene in the current OpenGL context.

The first rendering phase breaks the scene geometry down into small surface patches, with a maximum area which can be provided as a parameter. We then calculate the radiosity of each surface patch, in terms of light intensity and color variables which have not yet been specified. This phase uses OpenGL hardware acceleration for computing the form factors using hemicube approximations. Several passes of this calculation can be run over the scene to simulate multiple bounces of light.

The second phase generates an image using OpenGL, substituting in specific values for the intensity/color variables. This allows the slower first phase to be computed only once (as long as the scene geometry, light position and focus remain fixed), and the second phase to be quickly recomputed with every light intensity or color change, resulting in a slow preprocessing step but very fast render times for each intensity or color change. Also, because we are using OpenGL for display and the radiosity calculations are view-independent, the user’s view can be changed in real time, allowing the user to view their lighting design from different angles without re-rendering.

Internally, parad is a ‘shooting’ radiosity renderer - during every pass, each light source and surface patch ‘shoots’ light to every other patch that they can see. The light source patches are special for several reasons - during the first pass they are the only patches processed, and they do not gather and reflect light like surface patches. Light source patches also have a beam angle which controls the field of view angle used when drawing the hemicube view, and use a special cosine multiplier to simulate the non-flat light distribution of a lighting instrument.

We leverage the power of the GPU by using it to approximate the patch ‘form factors’, which describe how much light each patch projects onto every other patch. We assign each patch a unique ID, and map each patch ID to a unique 24-bit RGB color which we use when drawing that patch. This allows us to process scenes with over 16 million patches, well above the limit of what parad could process in a reasonable amount of time. We then draw the scene from the point of view of each patch, and read that image back from the GPU. We add up how much of each patch is visible, and apply a filter to this data to correct for the perspective transformation. Finally, we can add a specific amount of light contribution to each patch based on the shooting patch’s radiosity and the form factor of the two patches.

We can evaluate parad on both accuracy and performance. Parad-generated images clearly exhibit many of the desirable qualities of radiosity rendering, such as soft shadows, color bleeding, etc. Parad’s output does look quite different than Radiance’s output, but this is to be expected as the two renderers are using fundamentally different techniques. The output of both renderers should be useful to lighting designers in evaluating their work, and having both raytraced and radiosity-generated images at their disposal could help
users to look past the shortcomings of each technique.

Parad’s second phase has real-time performance and is ideal for interactive cueing. Parad’s first phase, though, is quite slow, easily taking several minutes for a high-quality render on a modern desktop or laptop computer. On the upside, parad’s parameters (such as subdivision amount, hemicube resolution, and number of passes) are very configurable, so it is easy for the user to choose their own tradeoffs between image quality and preprocessing speed.

3.6 Additional Technical Considerations

Another goal of our project is cross-platform support. The LightLab software used by the Concepts of Lighting Design class is very Unix specific, which is often a roadblock for students trying to use the software on their home computers for their own designs. We have achieved cross-platform support by choosing technologies which are well supported on both Windows and Linux operating systems, such as Python, wxWidgets and OpenGL.

One of our earliest technical considerations was choosing a language for implementing our user interface and data model. Both Java and Python are cross-platform, and have widget toolkits providing the features we needed. We decided to use Python because it would make it easy for us or others to extend our project with new features and data. In addition, Python integrates well with C and C++, which gave us the flexibility to implement parts of our software in lower-level languages for performance reasons. This proved to be the right decision as we implemented our radiosity renderer in C++, and provided an interface to our Python front-end using SWIG.

We chose to build our user interface using the wxWidgets toolkit because it is one of the most featureful and complete toolkits available for Python. It is cross-platform, uses the host system’s native widgets through
an abstraction layer to maintain a native look and feel, and integrates well with OpenGL.

We faced two major challenges in implementing our cross-platform support. The first was supporting the Radiance renderer, which uses many Unix-specific features, on Windows. We succeeded in compiling Radiance using Cygwin (a Unix emulation layer for Windows), and we include these Cygwin Radiance binaries with our program distribution. We also had to translate between Unix and Windows paths when using Radiance on Windows, and wrote some path management code in Python to resolve these issues.

The second challenge was developing our radiosity renderer, which was implemented in C++, so that it could compile and run on both Windows and Linux. We were careful to use only cross-platform libraries such as GLUT and GLEW, and tested on both platforms as development continued. We tried at first to use MinGW to build the renderer on Windows, but eventually we settled with Visual Studio 2005, because it has better support for building Python modules implemented in C++ using SWIG. Our software provides a Makefile for building on Unix and a Visual Studio project for building on Windows.

4 Conclusion

This project succeeded in developing a tool capable of providing useful interactive feedback to lighting designers. Our implementation of a parameterized radiosity renderer sets this program apart from other existing software, and demonstrates the practicality of using this type of rendering in a lighting visualization context. The user interface we developed allows the user to quickly configure scenes and lights, and instantly observe the results of their changes.

There are still some feature improvements we feel would be necessary for this to become a practical system for general use. The ability to load and save scene configurations, light instrument placements and cues would be necessary for a user to save their work between sessions. More optimization should also be done to improve performance on complex scenes with tens or hundreds of thousands of polygons - moving more code from Python to C++ might help with some scene complexity performance issues. Radiosity preprocessing is also a performance bottleneck to which we have already given much attention, but it would need even more work to become practical for complex scenes.

Our attention and goals shifted throughout the course of implementing this project. In the end, we wound up spending more time developing the radiosity renderer and improving its output, at the expense of some of the features and polish we intended for the front-end program. We feel this was a justified tradeoff - perhaps the most interesting and novel part of this project was the application of a parameterized radiosity algorithm to lighting design visualization, and by focusing on this aspect of the problem we were able to produce a software package that has many practical improvements over past work.

References


The CIS 460 course notes contain valuable reference material for many aspects of computer graphics programming. These slides are our main reference for 3D geometry vector and matrix calculations. We also use these slides as a reference for viewing and camera transformations. The slides contain information on many rendering techniques, and we plan to refer to the section about radiosity rendering when implementing our radiosity renderer.


Cast Lighting WYSIWYG is a commercial lighting design and previsualization product used professionally in the theatrical lighting and event industries. It contains powerful features for
creating and generating the paperwork for a design, producing renderings of a design, and even
previsualizing the cues for an entire show. A free demo/viewer version is available, but the fully
featured versions can cost hundreds or even thousands of dollars.

http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm

This website provides a high-level description of how radiosity rendering works, as well as pseudocode and images demonstrating the necessary algorithms. It begins by highlighting the differences between traditional ray tracing renderers and radiosity rendering. Following this introduction, each part of the radiosity algorithm is thoroughly described. Breaking down the scene into patches, computing the light incident on each patch using hemicubes, and various optimization techniques are all presented. The high quality of the resulting images demonstrates the success of Elias’ techniques, and we found this to be an extremely useful resource when implementing our own radiosity renderer. The website has not been updated in many years but the information is still highly relevant.


This website contains an explanation of radiosity rendering. Particular emphasis is placed on the mathematical foundations of the algorithm. The website also explains the hemicube approximation for form factor calculation. There is a discussion of the problems the authors encountered when implementing their radiosity renderer, and some suggestions for avoiding these problems. The information and advice on this website was useful for our radiosity renderer implementation. Their presentation is relatively recent (2002), and the information presented makes sense and is accurate when compared to our other sources of radiosity information.


This website describes the technique of using virtualized light sources to simulate the effect of more lights than OpenGL natively supports. Most OpenGL implementations support only a maximum of 8 lights, which is insufficient for many applications. By using virtualized light sources, the 8 OpenGL lights are dynamically reconfigured to be whichever 8 lights have the greatest effect on the geometry currently being drawn. This technique is especially useful when many of the lights are spotlights that are not all focused on the same areas in the scene (as is very common in theatrical lighting design). This could have been a useful technique in our OpenGL renderer used for focusing lights. This technique appears to be somewhat old, but it is still quite useful as OpenGL has still not moved beyond its 8 light limitation.


Rendering with Radiance is a guide to using the Radiance rendering tools, and also contains information about how Radiance works and the rendering techniques it uses. It is an excellent reference for creating the geometry and material description files for Radiance (which our program generates). There is also a chapter describing how to render scenes specifically for theatre, taking into account effects that occur when dimming lighting instruments and filtering them
with color gels, which was especially valuable for this project. We also used the book as a reference for importing real-world photometric data - this book describes how photometric data is captured and specified and how it can be imported into Radiance. Larson is the authority on Radiance as he developed the system, confirming that this book is trustworthy.


This journal article presents a method for interactively changing the intensities and colors of light sources in a radiosity-lit environment - assuming that the scene geometry and light positions remain fixed. By keeping track of the relative contribution of each light source to the scene, the actual intensities and colors are 'pulled' outside of the radiosity solution, and can be applied in a final reconstruction step. This methods results in radiosity solution times on the order of several seconds, and relighting 'reconstruction' times of many frames per second. The task of writing cues for theater has the property that light intensity and color will vary much more often than scene geometry, therefore we hope to take advantage of the optimizations presented in this article to achieve interactive framerates in our radiosity renderer. This article's presence in a peer-reviewed journal confirms its trustworthiness.


The Virtual Lighting Design Lab is a suite of programs used for lighting visualization by performing arts groups at Penn. It relies on some off the shelf components such as the Radiance renderer, but also contains custom GUI tools for hanging a virtual light plot, focusing instruments, and setting light intensities and colors. The software only runs on Linux, but there is a Knoppix-based bootable live CD that contains all of the light lab software.


The OpenGL Red Book is the de facto guide to programming with OpenGL. It describes the basic concepts of OpenGL, and explains in depth how to use OpenGL’s commands for drawing geometry, setting up viewing projections, and configuring lighting, texturing and material properties. OpenGL provides our software with hardware accelerated (but inaccurate) rendering capabilities which are useful for real-time user interactions. We also accelerate our radiosity renderer by using OpenGL for our hemicube computations. This reference is up to date, at least for the portions of OpenGL which we are using, and it is also trustworthy as it is written by the experts and accurately reflects the OpenGL API.


This thesis provides an excellent overview of the mathematics behind radiosity rendering, and discusses many different algorithms and techniques which can be used to implement a radiosity renderer. Different approaches to tasks such as mesh subdivision and form factor determination are compared on their strengths and weaknesses. Design considerations such as accuracy, speed, and memory usage are also thoroughly discussed, and the author explains the design decisions he made while implementing his renderer. Our renderer is actually implemented quite differently than the one discussed in this paper, mostly because we are able to take advantage of the GPU to perform our form factor computations, although there are also several advanced techniques such
as adaptive subdivision and progressive refinement which our renderer does not implement. Despite these differences, this paper proved to be a useful resource while implementing our renderer.


This book describes the science behind stage lighting color media, and provides a guide to gel colors ordered by their physical behavior. The gel color guide is quite out of date, but the chapter on colorimetry is still relevant, containing useful information about how color filters and lamps with different color temperatures interact with each other to create a specific color. We found this material useful when implementing our lamp and gel color computations.