EFFIGI
An Efficient Framework For Implementing Global Illumination

William Leeson
Carol O'Sullivan **Steven Collins**

Image Synthesis Group
Trinity College Dublin
Republic of Ireland

ABSTRACT

This paper presents a rendering framework called EFFIGI (Efficient Framework For Implementing Global Illumination) that uses interfaces which express both geometric concepts and mathematical ones, using object-oriented and component object methods. EFFIGI accelerates the development of new techniques and the implementation of existing ones, by providing a flexible but comprehensive geometric and mathematical architecture. The framework eliminates the need for users to implement an entire system, enabling them to focus only on those areas of particular interest.

Keywords: Rendering Framework, Object-Oriented Design, Global Illumination

1 Introduction

Global illumination deals with the propagation of light through an environment, and is described or modeled in mathematical form by a number of equations. Many rendering architectures have been proposed to support this. These frameworks tend to encapsulate mathematical concepts within physical ones. However it is sometimes desirable to separate these. Firstly the increased visibility of mathematical components make it easier to solve specific problems. Secondly separating the description from the solution contributes to smaller, reusable components, which can be combined and tested with greater ease.

This paper presents a rendering framework called EFFIGI that uses interfaces which express both geometric concepts and mathematical ones, using object-oriented and component object methods. It also handles other concerns that potential users might have, such as object description and scene acceleration. The framework differs from most other frameworks by mapping many global illumination concepts to mathematical ones rather than the reverse, making it far easier to impliment mathematical techniques.

In Section 2 some background information is provided, and Section 3 discusses the design of our architecture. The implementation of the framework is covered in Section 4 which also demonstrates the construction of various rendering algorithms. Conclusions and future work are presented in Section 5.

2 Previous Work

Many mathematical models are used in image synthesis, the rendering equation [Kajiy86] being one example. These mathematical models are solved by a variety of techniques. By simplifying these, easier and faster methods of solution may be found. For instance the radiosity equation [Cohen88] is a simplified form of the rendering equation that can be solved using finite element methods such as Galerkin [Zatz93] or point collocation [Cohen93] methods. However, this simplification limits the types of environment that can be described. More general descriptions like the rendering equation (see Figure 1) need more general methods of solution such as Monte Carlo integration. Some of the methods come directly from numerical techniques such as Metropolis [Veach97] or Galerkin methods. Others such as bi-directional path tracing [Veach94, Lafor93] are derived from the na-

$$
\begin{aligned}
L(\vec{\omega_i}, x) \quad = \quad & \underbrace{L_{\epsilon}(\vec{\omega_i}, x)}_{\textit{IFunction}} + \\
& \underbrace{\int_{\Omega} \overbrace{\overbrace{\rho(\vec{\omega_i}, x, \vec{\omega_o})}^{\textit{IFunction}} L(\vec{\omega_i}, x)}^{\textit{IFunction}} \cos\theta \, d\vec{\omega}}_{\textit{IIntegrator} \text{ and } \textit{IFunction}}
\end{aligned}
$$

Figure 1: The Rendering Equation

ture of ray paths. Most methods, however have the common property that they are described using mathematics.

Various architectures have been devised to handle this situation. Some have been tied to a specific technique [Cook87, Kirk88, Shirl91, Ward88, Walte97]. Others have been general enough to allow a variety of rendering techniques. The Cornel "testbed for image synthesis" [Trumb91] is a toolbox that can be used to construct a rendering package but is not object-oriented. Glassners "Spectrum" architecture [Glass91] is an object-oriented framework based on signal processing. The Vision system [Slusa95] is capable of most rendering methods as is the RenderPark system [Bekae]. In Section 5.1 of this paper we compare the last two frameworks with ours.

3 Design

The design of the EFFIGI framework is motivated by several key factors. It is fully object-oriented, thus enabling increased flexibility and code reuse. The abstractions are based not just on physical concepts such as radiance and radiosity but also on their mathematical foundations. In order to make development easy there is a means of testing components. The time taken to generate a picture is very important, so speed has been taken into consideration throughout all phases of the design process.

3.1 Object Oriented Design

Object oriented techniques are used to decompose the rendering process into simple manageable parts. The interfaces that arise from this fit into four main groups:

- *mathematical* for sample generation, function evaluation and integration.

```
struct IFunction : IUnknown
{
 virtual integer GetDimension(void) const = 0;
 virtual real Evaluate(const real *p_parameters) = 0;
 virtual void GetDomain(real *p_min,real *p_max) const = 0;
 virtual void GetRange(real *p_min,real *p_max) const = 0;
};
```

Figure 2: IFunction interface

```
struct IIntegrator : IUnknown
{
 virtual void SetSamples(integer samples) = 0;
 virtual real Integrate(const real *p_min,
                        const real *p_max,IFunction *p_func) = 0;
};
```

Figure 3: IIntegrator interface

- *scene and object management* for ray queries, meshing and path generation.
- *data storage* to support data structures such as linked lists, k-d trees [Bentl75] and vectors.
- *setup* for object initialisation and setup.

Within each of these groups, various sets of interfaces are provided. For a full list of interfaces see Table 1.

Math	**Scene**	**Setup**	**Data**
ISampler	IIntersect	IRenderer	IContainer
IFunction	IShape	IInterface	IVector
IIntegrator	IMesh	IInitialise	ITree
IGenerator	IRay		IList
IRootFinder	IlluminationMap		
IWarp	IPathGenerator		
	IEvent		
	IEventStore		

Table 1: Interfaces in Framework

3.2 Mathematical Foundation

The abstraction of mathematical concepts provides generic access to various algorithms, allowing a "pick-and-mix" approach to choosing a suitable technique. This abstraction is important in image synthesis, because many models use mathematically-based methods. For example, the light propagation model used in rendering utilises sampling and integration. EFFIGI provides explicit support for these ideas, by translating concepts such as functions, integration and sampling into interfaces. The learning curve for the users of such a framework will not be steep, since such people are typically well-aware of the mathematical ideas involved.

The most common interfaces used are the *IIntegrator* and *IFunction* interfaces. These are usually used to represent the various parts of our

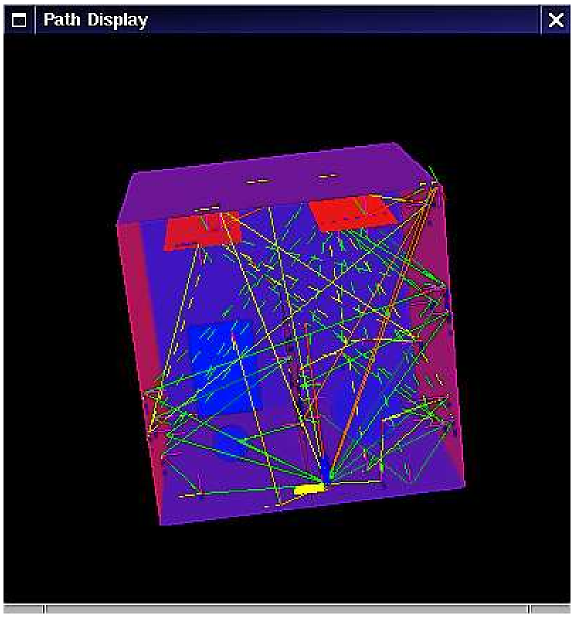

Figure 4: Ray Viewer

image synthesis model. The *IFunction* interface can be used to represent surface reflection models or to evaluate ray paths. When used by an *IIntegrator* the total light reaching a point can be estimated. Figure 1 shows how this can be encapsulated within another function interface to represent the rendering equation.

The parameters of the inner most *function* represent the direction (θ, ϕ) that the ray goes when it strikes a surface as in the case shown. They are equally likely to represent a surface and coordinates on it (s, u, v), if a surface form of the rendering equation is used. This configuration represents a distributed ray tracer since the function is recursive. A more efficient implementation would expand the inner function and use it to represent n bounces, where n is the first parameter, thus creating a path tracer. To allow this the function interface provides methods which disclose this information. The *IFunction* interface is given in figure 2. This interface enables functions to be evaluated, and provides information describing the limits of the domain of the function and the range of the result. The *IIntegrator* interface is shown in figure 3. It provides a means of evaluating an integral over a region in the functions domain, and setting the number of samples needed.

3.3 Testing Facilities

An important feature of the framework is its facility to test components for compliance with a given specification. For instance, a Bi-directional

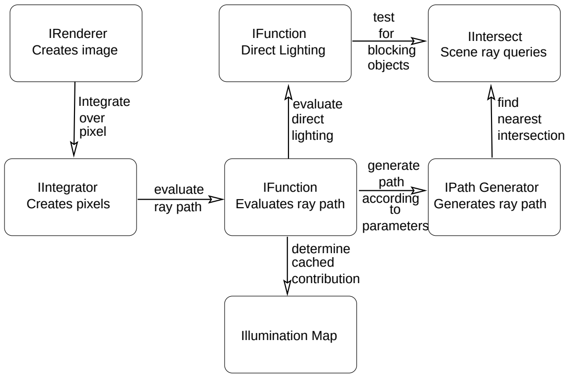

Figure 5: Path Tracer Configuration

Scattering Distribution Function(BSDF) should be physically plausible; a scene acceleration scheme should support all the required features, and provide the same intersection information as another scheme given the same data; an interface should produce valid responses to a specified set of inputs. Previously, it was necessary to run components with a complete setup, and then attempt to identify artefacts in the output. Now users can run the testing components with default setups, allowing rapid identification and location of errors. This speeds up the development process significantly. It is also possible for users to quickly ensure that the interfaces provided can support any new components they may wish to add to the framework. There are also visulisation tools for viewing the rendering process, for instance in figure 4 shows a component that allows you to see the rays as they are fired either from lights or the view plane.

4 Implementation

The *Component Object Model* [Krugl97, Roger97] was used in EFFIGI to provide a consistent interface for creation and destruction of components, and to provide Run Time Type Information (RTTI), for an in depth discussion of COM see [Krugl97] and [Roger97]. It allows the use of shared libraries or DLLs[1] which gives our framework the ability to have a "plug-in" style component architecture without any extra work by the programmer.

To date, EFFIGI has been used to implement many of the major rendering methods in use today. We now have a package, which can be configured to use a multitude of different techniques

[1] In UNIX they are called shared libraries in Windows dynamic link libraries or DLL's. These enable the sharing of compiled code by allowing libraries to be dynamically loaded at runtime.

Figure 6: Path Tracer

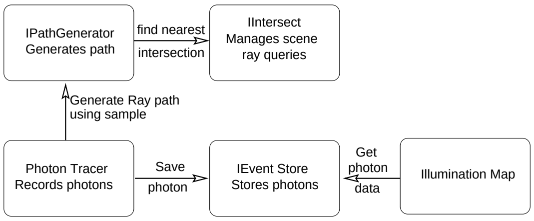

Figure 7: Photon Tracer Configuration

with ease. The framework can be used in two ways, via a simple scripting language that allows the user to create, destroy and set up links between objects or by using a programming language that supports COM such as C++. The scripting language is similar to C++. It can be used to modify any component, enabling scenes to be setup or scene files from other rendering packages can be loaded using a component that provides this. What follows are some example setups using the scripting language that illustrate some typical uses of the framework. Each configuration is presented with a diagram showing the communication links between the components that make up the method.

4.1 Path Tracer

The path tracer [Kajiy86, Dutre94] is constructed from a few basic components as shown in Figure 5. Direct Lighting is achieved using an *IFunction* interface which evaluates rays from the light to the point in question. This is then called from inside the path evaluator. Each box represents a component which can be substituted at run time, and thus a variety of configurations are possible.

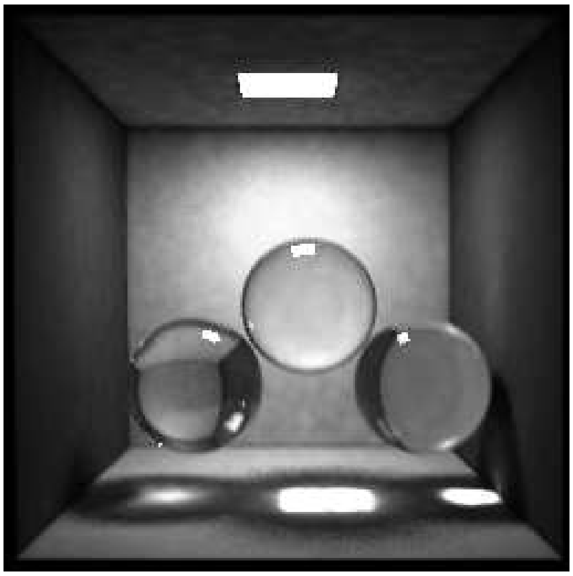

Figure 8: Photon Tracer

The picture shown in Figure 6 was rendered using a VEGAS [Lepag80] style integrator with a directional path generator and a surface to surface direct lighting function.

4.2 Photon Map

A Photon Map [Jense96] is a preprocess which is used to create illumination maps (see Figure 7). These can be used with the path tracer configuration just described by using a modified path evaluator. These path evaluators query the illumination maps at a given point, obtaining an estimate of the incoming radiance at that point. The photon generator can use all the samplers and warping schemes just like a path tracer, storing photons using an *IEventStore* interface. The component used to store the photons can use a k-d tree, quad tree or any other suitable structure, and might also directly support an *IIlluminationMap* interface. Alternatively, it can provide some facility to convert it to a suitable structure for some other component which would provide the illumination map during the rendering phase. Figure 8 shows a Photon Map used in conjunction with a path tracer, which in turn uses a Mean Sample Monte Carlo integrator [Kalos86] with a random sampler. The photons were generated from the light source using a Metropolis sampler which sampled the emission function $L_e(u, v, \lambda, \vec{\omega})$ of the light source to generate a sample set. A Metropolis sampler is a component based on the *ISampler* interface. It takes an *IFunction* interface and generates a set of samples, using the function as a basis for the pdf, this is done using the Metropolis [Kalos86, Veach97] method.

Figure 9: Irradiance Map

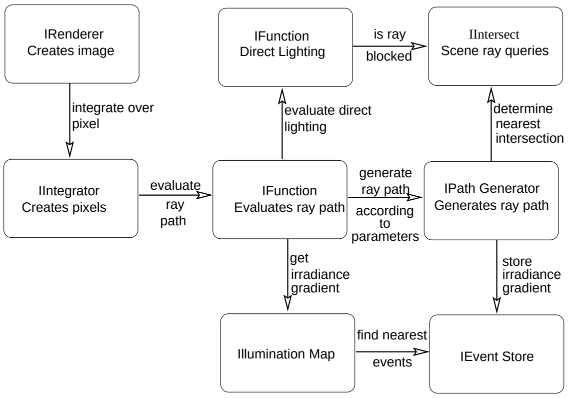

Figure 10: Irradiance Map Configuration

4.3 Irradiance Map

An Irradiance Map [Ward88] is a modified path evaluator which caches radiance information in an illumination map, also using an *IEventStore* interface. It is possible to re-use this interface for this, because the *IEvent* structure used with the event store dictates what is stored and what the information relates to. The *IEvent* interface provides a way to store information. By changing these events, different types of information can be stored, such as irradiance gradients or photons. Unlike the photon map, the irradiance map is not a preprocess but a modified path evaluator (see Figure 10). Shown in Figure 9 is a picture generated using an Irradiance Map style path evaluator and a Mean Sample Monte Carlo integrator using a Hammersly [Press96] quasi-random sample set.

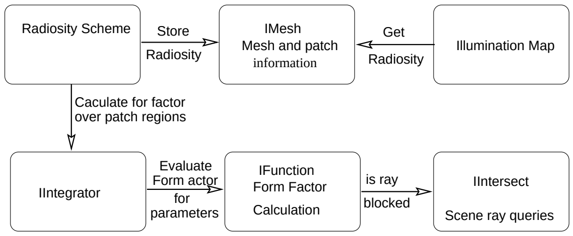

Figure 11: Radiosity Configuration

4.4 Radiosity

Radiosity [Cohen93] is implemented as a preprocess, as in the Photon Map method. Similarly radiosity is stored in a component, this time using the *IMesh* interface. This interface supports the ideas of patches from which form factors can be calculated and radiosity stored. The component may support an *IlluminationMap* interface, or it may provide the ability to output a texture, or something that can be used by an illumination map component. The form factor is calculated using a component that supports an *IFunction* interface that takes two surfaces as properties, the parameters of which are the (u, v) surface coordinates on each surface. This enables the use of various integration schemes (see Figure 11 for the configuration used). Figure 12 is a radiosity illumination map rendered using a Mean Sample integrator with random sampling. The radiosity structure used a Metropolis sampler to integrate the form factor function using the visibility function as a basis for the probability density function(pdf).

5 Conclusions and Future Work

EFFIGI has been successfully used in the implementation of many of the latest techniques in image synthesis. It is extremely flexible and provides a means of easy augmentation using both source code and binary DLL's. The use of interfaces in the framework allows easy extension of many older methods without having to reimplement them. Since all the modules can be provided as a DLL, only the modules actually referenced are loaded thus reducing the memory footprint of the system.

Implementers are also allowed to experiment with various ideas (for example sampling schemes) without having to alter any code. If the code is written in a generic way it can be used with many of the other components, thus extending the entire system. This encourages the user to break up any objects they create into many reusable parts

Figure 12: Radiosity

so that they can be used with the other elements of the system. This has resulted in greater flexibility, more code reuse and a powerful rendering environment.

5.1 Comparisons

Although comparing frameworks is difficult, the following attempts to highlight some of the differences between our framework and two others introduced in Section 2 which share similar objectives.

RenderPark is an object oriented rendering framework which is freely available to the public. Both it and EFFIGI divide the set of rendering techniques into view dependent or view independent, and this dictates how they are implemented. The strength of RenderPark is its extensive suite of Radiosity algorithms. However, it lacks the mathematical basis of EFFIGI, meaning that integration and other mathematical methods cannot be changed easily. In addition, scene acceleration has been implemented using utility functions that do not support any common form of interface.

Vision is another object-oriented rendering framework which, like EFFIGI, derives its geometry subsystem from the Ray Tracing Kernel [Kirk88]. Vision uses a physically based object-oriented abstraction of the rendering process, and can render physically correct as well as non-physical systems. However, it specifies fixed mathematical models for some interfaces that cannot be changed by the programmer, for instance the VolumeLighting subsystem. In addition, the sampling interfaces have been fragmented for use with different techniques.

Table 2 presents a summary of the features in the frameworks: Effigi(E), Vision(V) and RenderPark(RP).

features	*E*	*V*	*RP*
Physically based	yes	yes	yes
Radiosity	PR	many	many
Object Oriented	yes	yes	mostly
Plug-ins	yes	yes	no
Mathematical basis	yes	no	no
Object Model	COM	CORBA	no

Table 2: Framework Features

5.2 Speed

Although the code in the framework is quite general it does not suffer a significant performance degradation, but it is not as fast as less general instances of the algorithms. For instance the picture in figure 6 was rendered at a resolution of 512x512 pixels with 256 samples per pixel, and this took approximately 4 hours. The scene was made up of about 50,000 polygons of which the couch, chair and T.V. contributed most. The picture of the balls using the photon map (see figure 8)was rendered at a resolution of 256x256 with 64 samples per pixel and 60,000 ray paths were generated to create the photon map, taking about 15 minutes in total. The radiosity picture in figure 11 also took approximately 20 minutes where the radiosity pass was let run for about 19 minutes and the final render took less than 1 minute. The however depend strongly on the suitability of the scene acceleration method. This can vary quite significantly if an inappropriate choice is taken, as it is this calculation which dictates the overall performance of the system. All the scenes were generated on a Pentium II 350Mhz with 64Mb RAM running FreeBSD. The ray query engine can determine all the intersections for approximately 30,000 rays in 1 second. This estimate was determined using the living room scene in figure 6.

5.3 Future Work

To date the framework supports an extensive set of ray-based techniques, but only one Radiosity method. In the future this will be extended to many of the other radiosity methods. This will probably add more interfaces as more common ground between each technique is exposed. Another extension is the support of parallel implementations such as data- distributed methods for large data sets. Hopefully this can be hidden in scene type components. As yet we are not sure if the parallel nature of various algorithms needs to be made explicit, or if it can be hidden behind the current set of interfaces. If this is the case all the current methods would work seamlessly with parallel methods. Finally, it is proposed to provide support for more numerical methods, especially for the common tasks of integration and sampling which are useful in all areas of rendering.

6 Acknowledgements

I would like to thank Dave Gargan, Gareth Bradshaw, Leo Talbot and Hugh McCabe for their help. This project has been supported by Enterprise Ireland strategic research grants ST/96/104 and ST/98/001 and Hitachi Dublin Laboratory.

REFERENCES

[Bekae] Philippe Bekaert. http://www.cs.kuleuven.ac.be/cwis/research/graphics/renderpark/.

[Bentl75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. In *Communications of ACM*, volume 18, pages 509–517, 1975.

[Cohen88] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):75–84, August 1988. Held in Atlanta, Georgia.

[Cohen93] Michael F. Cohen and John R. Wallace. Radiosity and realistic image synthesis. 1993. Held in San Diego, CA.

[Cook87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102, July 1987. Held in Anaheim, California.

[Dutre94] Philip Dutre and Yves D. Willems. Importance-driven monte carlo light tracing. *Fifth Eurographics Workshop on Rendering*, pages 185–194, June 1994. Held in Darmstadt, Germany.

[Glass91] Andrew Glassner. Spectrum: a proposed image synthesis architecture. *SIGGRAPH '91 Frontiers in Rendering course notes*, July 1991.

[Jense96] Henrik Wann Jensen. Global illumination using photon maps. *Eurographics Rendering Workshop 1996*, pages 21–30, June 1996. ISBN 3-211-82883-4. Held in New York City, NY.

[Kajiy86] James T. Kajiya. The rendering equation. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):143–150, August 1986. Held in Dallas, Texas.

[Kalos86] Malvin H. Kalos and Paula A. Whitlock. *Basics*, volume 1 of *Monte Carlo Methods*. John Wiley and Sons, New York, Chichester, Brisbane, Toronto and Singapore, 1986.

[Kirk88] David Kirk and James Arvo. The ray tracing kernel. *Proceedings of Ausgraph '88*, pages 75–82, 1988.

[Krugl97] David J. Kruglinski. *Inside Visual C++*. Microsoft Press, Redmond, Washington, fourth edition, 1997.

[Lafor93] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Compu-Graphics*, pages 145–153, 1993.

[Lepag80] G. P. Lepage. Vegas:an adaptive multidimensional integration program. In *CLNS-80/447*, volume 4, pages 190–195, 1980.

[Press96] William H. Press, Saul A. Teukolsky, William T.Vetterling, and Brain P. Flannery. *Numerical Recipes in C The Art of Scientific Computing*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, second edition, 1996.

[Roger97] Dale Rogerson. *Inside COM*. Microsfot Press, New York, 1997.

[Shirl91] Peter Shirley, Kelvin Sung, and William Brown. A ray tracing framework for global illumination systems. *Graphics Interface '91*, pages 117–128, June 1991.

[Slusa95] P. Slusallek and Hans-Peter Siedel. Vision - an architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):77–96, March 1995. ISSN 1077-2626.

[Trumb91] B. Trumbore, W. Lytle, and D. P. Greenberg. A testbed for image synthesis. *Eurographics '91*, pages 467—480, September 1991.

[Veach94] Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. *Fifth Eurographics Workshop on Rendering*, pages 147–162, June 1994. Held in Darmstadt, Germany.

[Veach97] Eric Veach and Leonidas J. Guibas. Metropolis light transport. *Proceedings of SIGGRAPH 97*, pages 65–76, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[Walte97] Bruce Walter, Philip M. Hubbard, Peter Shirley, and Donald F. Greenberg. Global illumination using local linear density estimation. *ACM Transactions on Graphics*, 16(3):217–259, July 1997. ISSN 0730-0301.

[Ward88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):85–92, August 1988. Held in Atlanta, Georgia.

[Zatz93] Harold R. Zatz. Galerkin radiosity: A higher order solution method for global illumination. *Proceedings of SIGGRAPH 93*, pages 213–220, August 1993. ISBN 0-201-58889-7. Held in Anaheim, California.