Yet Another Rendering Framework

William Leeson Steven Collins
william.leeson@cs.tcd.ie steven.collins@cs.tcd.ie

Image synthesis Group isg@cs.tcd.ie
Trinity College, Dublin, Ireland

Abstract. This paper describes an efficient framework for implementing global illumination techniques, using object oriented and component object methods. This framework facilitates in the development of new techniques or the implementation of existing techniques. By providing a flexible but comprehensive geometric and numeric architecture. The framework abstracts programmers and researchers from implementing an entire system, enabling them to focus only on those areas they are interested in. To illustrate the application of the framework we implement and compare existing and new Monte Carlo methods for global illumination.

1 Introduction

There are many different methods and techniques involved in the production of a complete global illumination system. The production of a capable, extensible easy to use and flexible system takes a considerable amount of time, yet many researchers still tend to implement their own rendering system. Some people will modify an existing system that is freely available, such as Radiance [6], Rayshade [15], RenderPark[1] or the Vision System [22, 24]. These systems either focus on just one solution strategy and offer very little support for any other methods. Or in the case of the Vision and RenderPark approach offer a number of different algorithms that are fixed within the system. In all cases implementation of new algorithms require significant re-design and implementation.

These systems are usually incompatible with one another and hence prohibit sharing of code or data such as scene description files. Various attempts have been made to address these problems, however researchers still use other less generic packages or write their own requiring a more significant investment of their time. What is required is a rendering framework that is flexible, complete, well encapsulated and modular, but which does not prescribe the algorithm used. In short the ideal research framework will include flexible domain independent numerical methods for integration and comprehensive domain dependent geometry and rendering facilities.

2 Goals

The framework presented in this paper tries to build on previous frameworks by addressing the following goals

- as generic and flexible as possible.
- no fixed rendering algorithm.

- efficient.
- simple and easy to understand.
- user is not required to learn the entire kernel, just the parts of interest (i.e. modular).
- enable code sharing and reuse.
- not restricted to a single programming language.

3 Framework Design

The framework uses object oriented techniques to decompose the rendering process into simple manageable parts. To facilitate the use of shared libraries or DLLs[1] a standard method of sharing objects between these processes was needed. We also require a method that permits the use of different programming languages and facilitates the creation of a run time, as well as compile time, "plug-in" architecture. The *Component Object Model*[10, 23] allows us to achieve these goals. Other methods such as shared libraries (dynamic link libraries or DLLs) or CORBA may provide an alternative strategy. However these were found to be too specific to a given language, were too complex or required extra libraries to be linked in.

3.1 COM model

COM is basically a protocol for connecting software *components* or objects (classes in C++) together. The components communicate through a mechanism called an *interface*. In order to setup this communication a few simple functions must be implemented in every shared library or DLL that is associated with the program. These functions just register the components contained in the shared library so that the component manager can locate them. A COM interface is one of the sets of methods implemented by a component. For an example see Figures 1 and 2 these define an example of a COM interface. The *IUnknown* and *IClassFactory* interfaces are the only two standard interfaces defined by the COM specification. The *IClassFactory* interface allows the creation of objects and *IUnknown* the extraction of pointers to the interfaces each object provides. Every component must implement the *IUnknown* interface as this provides a standard means of obtaining any interface the component may provide. COM binds interfaces at runtime through *aggregation* and *containment*. Aggregation is a process where by a pointer to the real interface is given directly to the user so that they are communicating directly with the actual object. Containment however hides the interface behind a proxy interface that the user talks to which in turn communicates with the real interface. The COM model although originally native to Windows may be implemented easily on any platform and hence is portable. The framework has a simple implementation of COM built into it. To port this to other operating systems only the DLL loading code requires modification.

3.2 Designing the Set of Interfaces

Since global illumination is all about solving integral equations it makes sense to base a substantial part of the set of interfaces around mathematical concepts. There are interfaces such as

[1] In UNIX they are called share 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.

```
#define interface struct
interface IUknown
   {
    virtual int QueryInterface(int nIid,void **ppvObj) = 0;
    virtual int AddRef(void) = 0;
    virtual int Release(void) = 0;
   };
```

Fig. 1. IUnknown COM interface

```
interface IClassFactory : public IUnknown
   {
    virtual int CreateInstance(int nIid,void **ppvObj) = 0;
   };
```

Fig. 2. IClassFactory COM interface

- `ISampler` sample set generator such as the Hammersly point set.
- `IFunction` generic mathematical function e.g. $sin(\theta)$.
- `IIntegrator` generic mathematical integrator e.g. Simpsons Rule
- `IGenerator` generates sequences of numbers e.g. a linear congruential random number generator.
- `IRootFinder` for finding the root of an equation e.g. Newton Raphson.
- `IWarp` for altering the distribution of a sample set e.g. Shirleys Non-Uniform Point Sets via warping[13].

No assumptions are made about the solution method. Integral equations and BRDFs are both functions and hence can be represented by the *IFunction* interface. This facilitates easy testing of the BRDFs to check if they are physically correct or to calculate the total reflectivity for use in Lafortune's control variate method[16]. Other interfaces however may be provided to these functions which allow alternative ways of using them (for example an interface has been created which allows ray and surface information to be passed to a BRDF to sample or evaluate it as shown in Figure 3.2).

Scene management and object representation do not fit well in this mathematical framework. Thus various COM interfaces were designed giving methods which provide non-specific access to these objects. Such as

- `IIntersect` intersects rays with shapes.
- `IShape` geometrical shape information (used for example in area determination).
- `ISampleSurfaceDirection` surface solid angle sampling e.g. sampling the solid angle of a sphere for direct lighting.

```
interface SurfaceModel : public IUnknown
{
 virtual value Evaluate(const ray *in, const intersection *hit, const ray *out) = 0;
 virtual value SampleDirection(const ray *in, const intersection *hit,
                               array1d *v,ray *out) = 0;
 virtual value SampleReflected(const ray *in, const intersection *hit,
                               array1d *v,ray *out) = 0;
 virtual value SampleRefracted(const ray *in, const intersection *hit,
                               array1d *v,ray *out) = 0;
};
```

Fig. 3. Alternative BRDF Interface

- `ISampleSurfaceArea` surface area sampling for e.g. sampling the area of a triangle for direct lighting
- `IAggregate` groups objects e.g. for bounding boxes
- `IRay` provides ray information such as direction and origin.

As the system expands further interfaces may be added to address the needs of the users. This is possible due to fact that the COM model allows the user to query for a more specific interface, giving the user the ability to query for access to more focused interfaces as is necessary. The user, on receiving the *ISurface* interface could specifically ask for the *IBoundingBox* interface, or a specific polygon interface if there is one.

The set of interfaces chosen in the design and specification of a rendering system must allow for speed and flexibility. These usually tend to be conflicting requirements. COM handles the flexibility issue by allowing later addition of interfaces without having to rewrite and recompile the existing code base. This allows for great flexibility in the code. These new components with the new interfaces will be able to be used by or use older components provided that the relevant interfaces have been implemented. Hence it is possible to combine new techniques with old ones reaping the benefits of both. Since COM is a small and efficient object model it not does not slow the program down.

4 Use of the Framework

To use the framework the implementor decides which interfaces their object will provide and which interfaces will be used by other objects. This allows links between their object and objects created by others. For instance an integrator object would use the *IFunction* interface to evaluate a function and provide the *IIntegrator* interface so that other objects that know about this interface can use the object.

The number of different interfaces supported by an object generally tends to indicate the number of different ways an object can be used. For instance a sphere object could provide a *IFunction* interface and a *IIntersect* thus enabling direct ray intersection via the *IIntesect* interface or the use of *IRootFinder* to find the intersection points. The implementor could also add his own interface to the set of interfaces thus providing a specific interface for the new object. The ability to add custom interfaces allows the implementor a great deal of flexibility and also provides for new techniques that don't "fit" into the current set of interfaces.

4.1 A Simple Example

Rather than the usual practice in global illumination of using classes to describe an object directly, the framework uses them to describe the properties of the object. These are then combined to form an object. As an example of developing a component in the framework we describe the implementation of a SEADS[5] scene. To create a scene object one needs to implement at the very minimum two interfaces. These are the ray intersection interface *IIntersect* and the aggregation interface *IAggregate*. The ray intersection interface shown in Figure 6 provides various means of intersecting a ray with an object such as a surface or scene. This is possible because the interface to an object

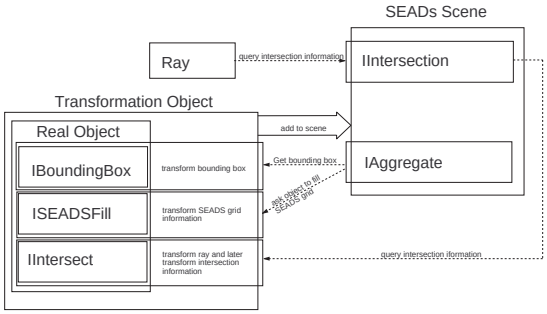


Fig. 4. SEAD Scene Interface Interaction

```
interface IAggregate : public IUnknown
   {
    virtual void Add(object) = 0;
    virtual void Remove(object) = 0;
   };
Add(IShape *p_ishape)
   {
    IBouningBox *p_iboundingbox;
    ISEADSfill *p_iSEADSfill;
    if((p_ishape->QueryInterface(SEADS_FILL,p_iSEADSfill)) != NULL)
      {
       get shape to fill in SEADS grid
      }
    else if((p_ishape->QueryInterface(BOUNDING_BOX,p_iboundingbox)) != NULL)
      {
       use bounding box to fill in SEADS grid
      }
    else
      {
       error cannot fill in SEADS grid not enough information
      }
   }
```

Fig. 5. IAggregate interface

is always recorded with the intersection information. The interface has methods to get a cross section through a scene or object, to find the nearest intersection of an object or scene and also to find if there are any intersections between two points on a ray. The *IAggregate* interface provides a means of adding objects into the scene (see Figure 5). There is no real difference between a scene and an object, such as a sphere for intersection purposes[2]. To assign properties to the object or scene the relevant interfaces are added to the object using the *IObject* interface (see Figure 13). For instance you could add a Phong[18] surface interface to the object or scene thus giving surface properties to that entity.

Implementing the intersection interface is just a matter of passing on the relevant ray parameters to the objects contained in the voxels of the SEADS grid, determining if the ray hits them or not and then passing that information back to the caller. The *IAggregate* interface takes care of adding objects to the scene. This interface is shown

[2]Since a pointer to the object intersected is always returned with the intersection information. Also any specifics are implemented through other interfaces.

```
interface IIntersect : public IUnknown
   {
    virtual void Intersect(ray *beam, intersection_info *hit) = 0;
    virtual void CrossSection(ray *beam, stack<intersection_info *> hit_list) = 0;
    virtual bool Blocking(ray *beam) = 0;
   };
```

Fig. 6. IIntersect Interface

in Figure 5. The scene must take the object interface provided by the object method and add it to its object database. At this stage the SEADS scene can make a number of choices

- it could use the information provided by the object interface to calculate where to put the object in the voxel grid.
- it could request an interface to help it do this.
  - it could ask for the bounding box interface and use that to fill the grid.
  - it could query the object to see if it supports the *ISEADSfill* interface. This is a custom interface created by the designer of the SEADS grid which gives the object the 3D array which represents the voxels of the SEADS grid and its bounds and allows it to fill the information into the array.

The custom interface allows for far more optimized use of the SEADS grid[3]. This object could then be contained in a transformation object, which transforms the ray and intersection information, so that the object can be placed anywhere (see Figure 4). Adding it to an object that transforms the rays before the intersection is performed does this. This object then passes the transformed rays on to the real object and then transforms the results. This transformation process can be used to create multiple instances of an object. Scenes can be imbedded in other scenes by adding them through the *IAggregate* interface. Thus the framework provides similar features and functionality to Arvo and Kirk's Ray tracing Kernel [14].

4.2 A More Interesting Example

A far more interesting example of using the framework occurs when implementing a method for solving the rendering equation[11] via a ray based method. The two most common ray based approaches are

- distribution ray tracing
- path tracing

These two contrasting approaches can easily be represented in the framework. The first approach is implemented as a modified recursive descent distribution raytracer[25]. In the framework we evaluate the rendering equation explicitly by creating two function interfaces. The main function interface (the distribution raytracer) propagates a ray through the scene by using itself as a function parameter to an integrator. The integrator then calls this function thus setting up the recursive loop. The other function is used for evaluating the direct lighting calculation[4] for which the integrator is also used (see Figure 7).

[3] Since now the object can just fill in the voxels of the grid it occupies and not the voxels occupied by the bounding box

```
class Direct Lighting : public Ifunction , pubic IUnknown
{
 value Evaluate(Tarray1D<value> *parameters)
   {
    Generate surface point according to parameters
    Result = Calculate energy in ray specified by parameters
    return result;
   }
};
class Distribution Tracer : public Ifunction, public IUnknown
{
 value Evaluate(Tarray1D<value> *parameters)
   {
    result = integrate direct lighting function from point
    calculate ray direction and intersect with scene
    result = result  + integrate this function from new point
    return result;
   }
};
```

Fig. 7. Distribution Ray tracer

An alternative is to parameterize the ray path so that it becomes a function. This is done by mapping each variable that dictates the path of a ray into a set of function parameters. To map a ray tracer to a function simply requires that each bounce of the ray be mapped to a certain set of the parameters passed to the function (see Figure 8 and 12(a)). Thus to evaluate the function you just generate an n-dimensional sample in the parameter space defined by the function which is an n-dimensional array. This array describes a ray path in the parameter space of the function. The integrator uses this function by generating k n-dimensional samples of the parameter space. This turns out to be a path tracing method of solution using k ray paths.

Now that these two approaches are implemented the user is free to use whichever type of integration scheme they require. These are accessed using the *IIntegrator*, interface (see Figure 10). A Monte Carlo integration scheme such as the VEGAS[19, 20] algorithm or the mean sample method[12, 27] (see Figure 11) could be used. This separates some of the mathematics (integration method) from the physics (ray bouncing) allowing the programmer a great deal more flexibility since either part can be replaced with another scheme. This makes both parts easier to debug since either part can be tested separately with already working parts. The code for both sections has been spilt and so is much smaller and easier to manage than one large conglomerate. The processes are also easier to reuse since any integration scheme may be used with any function.

It may be desirable that the rendering equation is broken up into simpler more manageable functions. This enables the rendering equation to be split into the camera model (also known as the pixel equation) and the scene propagation model. The camera model may be further split into the film and lens models. If this is done a generic method of creating one larger function is needed. Using a combining function that maps the parameters of each function into a single function enables this. For example the camera function may take parameters $(u, v), (x, y)$ which are surface points on the film and lens but the ray propagation function requires the start position and direction for the ray $(x, y, z), (dx, dy, dz)$. To use these parameters in the propagation function the camera model must convert its parameters (u, v) and (x, y) into the (x, y, z) and (dx, dy, dz)

```
class Path Tracer : public Ifunction ,public IUnknown
{
 value Evaluate(Tarray1D<value> *parameters)
   {
    determine ray direction and origin from parameters
    do{
       intersect ray with scene
       // Direct Lighting
       sample surface using variables from parameters
       result = result + weight * energy from surface
                to light along ray * throughput of shadow
                ray
       // Ray bouncing
       if(ray path not terminated)
         {
          determine new ray direction from parameters
          weight = weight * throughput of new ray direction
         }
      }while((parameters remain) && (ray path not finished));
    return result;
   }
};
```

Fig. 8. Path tracer

```
interface Ifunction : public IUnknown
   {
    virtual value Evaluate(Tarray1d<value> *parameters) = 0;
   };
```

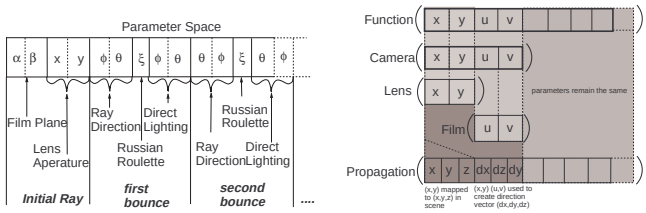
Fig. 9. IFunction interface

```
interface IIntegrator : public IUnknown , public IIntegrator
   {
    virtual value Integrate(const Tarray1d<value> *min, const Tarray1d<value> *max,
                            function *func) = 0;
   };
```

Fig. 10. IIntegrator interface

```
class MeanSampleMC : public Iunknown, public IFunction
   {
    value Integrate(const Tarray1d<value> *min, const Tarray1d<value> *max,
                    function *func)
       {
        while(Samples left)
             {
              pd = GetSample(sample);
              result = result + func->Evaluate(sample)/pd;
              n++;
             }
     return result/value(n);
    }
   };
```

Fig. 11. Mean Sample Integrator



(a) Function parameter space as seen by the integrator

(b) Mapping of parameter spaces between functions

Fig. 12. Parameter Space and mapping scheme

```
interface IObject : public IUnknown
   {
    virtaul int Add(interface *,int iid) = 0;
    virtaul int Delete(int iid) = 0;
    virtaul int Container(IUnknown *) = 0;
   };
```

Fig. 13. IObject interface

required by the propagation function to describe the initial ray. A standard set of functions can be used to do this or the camera function itself could convert these parameters (see Figure 12(b)).

As an illustration of the ease with which rendering methods can be created the VEGAS algorithm from particle physics has been implemented.

4.2.1 VEGAS algorithm. The VEGAS algorithm is a multi-pass adaptive sampling algorithm that also has limited support for stratified sampling. It is normally used in particle physics simulations and hence may be useful in image synthesis. The algorithm adaptively constructs a multidimensional weight function that is separable.

$$p \propto w(x, y, z, \dots) = w_x(x) w_y(y) w_z(z) \dots \tag{1}$$

It is a multi-pass method because in order to construct the weight function a coarse estimate of the function needs to be found. This then creates the initial weighting function. Further passes are made to refine the weighting functions. During each pass the real estimate as well as the weighting functions are being improved. From Equation 1 the optimal separable weight function can be shown to be

$$w_x(x) \propto \left[\int dy \int dz \dots \frac{f^2(x, y, z, \dots)}{w_y(y) w_z(z) \dots} \right]^{\frac{1}{2}} \tag{2}$$

Equation 2 presents a way of creating the weighting function using Monte Carlo estimation methods. The estimates for this function are then stored in k bins for each dimension d thus giving a total of kd bins where d is the dimension of the function. These bins can later be sampled as a pdf by sampling one set of bins per dimension.

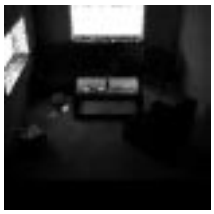
5 Application

To date the framework has been used to implement many of the major rendering methods in use to day. It has made the creation of these methods relatively easy. We now have a rendering package, which can be configured to use a multitude of different rendering techniques. The following methods have been implemented with the framework

- classical ray tracer [25]
- distribution ray tracer[3]
- eye path tracer [11]
- light path tracer [21]
- bi-directional path tracer [17]
- photon map[9, 7, 8]
- irradiance maps [6]

A progressive refinement radioisty implementation is currently being developed. What follows are some example pictures that use a variety of the rendering methods implemented with the system each of which took approximately the same amount of time. The framework may not only be applied to realistic image synthesis but also to any application which uses numerical integration.

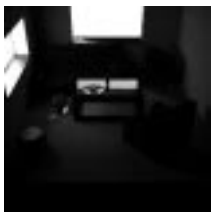
Photon Map



Random MC

Quasi-Random MC

VEGAS MC



6 Summary

The rendering framework has been successful 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 re-implement 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.

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

7 Future

To date, importance based Monte Carlo schemes have been used extensively in image synthesis. However stratified sampling of the entire solution or scene have not really been examined in relation to image synthesis (except in [11]) (this is not the same as stratified sampling of surfaces for direct lighting). Here Kajiya describes some stratified sampling schemes such as *sequential uniform sampling, hierarchical integration* and *adaptive hierarchical integration*. These methods if used in conjunction with importance sampling could considerably improve rendering speed and accuracy. This is because they could be used to divide the scene in to regions that have similar properties such as the same lighting setting. Thus when a ray enters that area a rough approximation to the light in that area is known and this could be used as a *control variate*(see [16]) for a Monte Carlo integration scheme. Dark areas could use fewer samples, as less accuracy is needed because of the low contrast between shapes (see [2]). Since the integral has been broken into regions this reduces the variance of each specific integral thus improving the accuracy of the result. Other adaptive stratification schemes such as Press and Farrars *recursive stratified sampling*[26] scheme could also be used.

8 Acknowledgements

I would like to thank Dave Gargan, Gareth Bradshaw, Leo Talbot and Hugh McCabe for helping me write this paper among other things. This project has been supported by Enterprise Ireland strategic research grants ST/96/104 and ST/98/001 and Hitachi Dublin Laboratory.

References

1. Philippe Bekaert. http://www.cs.kuleuven.ac.be/cwis/research/graphics/renderpark/.
2. Mark R. Bolin and Gary W. Meyer. A frequency based raytracer. In *Computer Graphics*, volume 29, pages 85–92, 1995.

3. Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Computer Graphics*, volume 18, pages 137–145. ACM Press, 1984.
4. Monte Carlo Techniques for Direct Lighting Calculations. In *Transactions on Graphics*, 1990.
5. A. Fujimoto, T. Tanaka, and K. Iwata. Arts:accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6, 1986.
6. Francis M. Rubinstein Gregory J. Ward and Robert F. Clear. A ray tracing solution for diffuse interreflection. In *Computer Graphics*, volume 22, pages 85–92, 1988.
7. Henrik Van Jensen. Rendering caustics on non-lambertian surfaces, 1995.
8. Henrik Van Jensen and Neils Jorgen Christensen. Efficiently rendering shadows using the photon map.
9. Henrik Van Jensen and Neils Jorgen Christensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. In *Computers and Graphics*, volume 19, pages 215–224, 1995.
10. David J.Kruglinski. *Inside Visual C++*. Microsoft Press, Redmond, Washington, fourth edition, 1997.
11. James T. Kajiya. The rendering equation. In *Computer Graphics*, volume 20, pages 143–150. ACM Press, 1986.
12. 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.
13. David Kirk. *Graphic Gems III*. Academic Press Inc., London, 1992.
14. David Kirk and James Arvo. The ray tracing kernel. In *Ausgraph*, pages 75–82, 1988.
15. Kraig Kolb. http://graphics.stanford.edu/ cek/rayshade/.
16. Eric P. Lafortune and Yves D. Willems. The ambient term as a variance technique for monte carlo ray tracing.
17. Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *CompuGraphics*, pages 145–153, 1993.
18. Eric P. Lafortune and Yves D. Willems. Using the modified phong reflectance model for physically based rendering. Report CW 197, Department of Computing Science K.U. Leuven, November 1994.
19. G.P. Lepage. In *Journal of Computational Physics*, volume 27, pages 192–203, 1978.
20. G.P. Lepage. Vegas:an adaptive multidimensional integration program. In *CLNS-80/447*, volume 4, pages 190–195, 1980.
21. Eric P. Lafortune Phillip Dutre and Yves D. Willems. Monte carlo light tracing with direct computation of pixel intensities.
22. P.Slusallek and H.P. Seidel. Vision:an architecture for global illumination calculations. *Transactions on Visulistaion and Computer Graphics*, 1:77–96, 1995.
23. Dale Rogerson. *Inside COM*. Microsfot Press, New York, 1997.
24. P. Slusallek. *Vision - An Architecture For Physically based rendering*. PhD thesis, University of Enlangen, 1995.
25. T. Whitted. An imporved illumination model for shaded display. In *Computer Graphics*, volume 23, pages 343–349. ACM Press, 1980.
26. W.H.Press and G.R. Farrar. In *Computers in Physics*, volume 4, pages 190–195, 1990.
27. Reuven Y.Rubinstein. *Simulation and the Monte Carlo Method*. John Wiley and Sons, New York, 1981.