AN OBJECT-ORIENTED ELECTROMAGNETIC PIC CODE

by

J. P. Verboncoeur, A. B. Langdon, and N. T. Gladd

Memorandum No. UCB/ERL M94/71

15 September 1994

AN OBJECT-ORIENTED ELECTROMAGNETIC PIC CODE

by

J. P. Verboncoeur, A. B. Langdon, and N. T. Gladd

Memorandum No. UCB/ERL M94/71

15 September 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

AN OBJECT-ORIENTED ELECTROMAGNETIC PIC CODE

by

J. P. Verboncoeur, A. B. Langdon, and N. T. Gladd

Memorandum No. UCB/ERL M94/71

15 September 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Submitted to Comp. Phys. Comm., June 15, 1994

AN OBJECT-ORIENTED ELECTROMAGNETIC PIC CODE

J. P. Verboncoeur, A. B. Langdon[1] and N. T. Gladd[2]
Electronics Research Laboratory
University of California, Berkeley, CA 94720

Abstract. The object-oriented paradigm provides an opportunity for advanced PIC modeling, increased flexibility, and extensibility. Particle-in-cell codes for simulating plasmas are traditionally written in structured FORTRAN or C. This has resulted in large legacy codes[3] which are difficult to maintain and extend with new models. In this ongoing research, we apply the object-oriented design technique to address these issues. The resulting code architecture, OOPIC (Object Oriented Particle-in-Cell), is a two-dimensional relativistic electromagnetic PIC code. The object-oriented implementation of the algorithms is described, including an integral-form field solve, and a piecewise current deposition and particle position update. The architecture encapsulates key PIC algorithms and data into objects, simplifying extensions such as new boundary conditions and field algorithms.

1 Introduction

Plasma simulation is the computational modeling of the interaction of charged particles with electric and magnetic fields. The plasma may exhibit complex nonlinear behavior, and the fields and particles may interact with time-dependent boundary conditions. Fluid codes model the plasma using moments of a distribution function at discrete grid points, while particle-in-cell (PIC)

---

[1] also at Lawrence Livermore National Laboratory, Livermore, CA 94550.

[2] Berkeley Research Associates, Berkeley, CA 94701.

[3] The term *legacy codes* refers to codes written using traditional structured techniques, and can often trace their ancestry to the 1960s or 1970s. Maintenance of legacy codes often consists of patching the existing code; such codes are difficult to rewrite because the close coupling of the structured technique may make modular upgrading impossible.

codes model the plasma using discrete particles, each representing many charged particles. Even a low density plasma has trillions of particles per cubic centimeter, so the modeler must be very clever to model the statistics of a physical system within the constraints of the computer.

PIC particles interact with fields defined at discrete locations in space (on a mesh, for example) using interpolation to compute the forces on the particles. The particles generate the current and charge density source terms for Maxwell's equations by interpolation from the particle locations to the mesh. The number of mesh cells or nodes required depends on the physics to be studied, ranging from 10^2 for simple one-dimensional problems to 10^9 or more for a complex three dimensional simulation.

A plasma physics code must contain a number of features to qualify for general distribution. We identify two classes of the target audience: the user, who will use the code by adjusting the predefined parameter set; and the developer, who will extend the functionality of the code by adding models and algorithms of various levels of complexity which must interact with the existing algorithms. The first class of user requires a code which is straightforward to operate and adjust parameters, with sufficient error trapping to prevent fatal errors due to improper input. The code must also protect the casual user from running a simulation with parameters which generate a nonphysical result. The second class, the developer, is more typical of the computational plasma physics research community. The developer must often add an algorithm to extend the existing code to model a new phenomenon or adjust an existing model because the problem at hand is sensitive to the accuracy of a particular parameter or method.

The requirements for a general plasma physics code can be summarized as follows:

- Capability to accurately model the physical phenomena of interest.
- Extensibility and reusability for adding new models or modifying existing models.
- Encapsulation of algorithms to localize the impact of code modifications.
- Efficiency of algorithms and architecture, including optimization of speed and storage.

- Fatal error trapping to prevent user-induced crashes, including unstable regimes of operation.

- Error trapping or warning for simulation regimes characterized by inaccuracy.

Historically, computational plasma physicists have written large codes in structured languages such as FORTRAN, and more recently, C. These structured programs can contain 10^4 to 10^6 lines of code, excluding the user interface. Initially, these codes were written to solve a specific, specialized problem, and required an expert to run. In many cases, the only qualified expert was the code developer. More recently, general codes applicable to a family of similar problems have become available, including MAGIC [2], ISIS[3] and PDx1 [4]. These codes provide limited parameterized flexibility in selecting algorithms and specifying the properties of the simulation. However, as the number of algorithms required to model complex devices increases, the interactions between algorithms becomes unwieldy.

The structured method, while providing limited modularity, is flawed in addressing many of the most pressing problems in the computational plasma physics area. For example, global variables can be (and usually are) accessed from many modules, making it difficult to track the evolution of data. The structured approach provides only weak encapsulation, resulting in close couplings of unrelated functions which depend on the internal implementation of other functions. Many side effects can occur when modifying code for maintenance or extension. The cost of debugging and maintaining a structured code can become prohibitive as the complexity of the code increases. The object-oriented technology provides a solution to this problem through data encapsulation.

Extending a structured code to include a new type of model is especially difficult, as many coupled functions must be rewritten to work with the algorithm. Languages such as C and FORTRAN provide no mechanism for software extension via reuse. For example, X-Lab is studying a new device which is using a new cesium-impregnated cathode. This cathode has properties very similar to a cathode presently modeled by the existing code in use at X-Lab, except the new cathode emits electrons with a distribution function dependent upon the local electric field strength. If the X-Lab code is written in a traditional structured language, the new model must be written as a new function, **newCathode()**. The **newCathode()** function will likely

consist of some code cut and pasted from **oldCathode()**, modified to model the new properties. The argument lists for the functions are now different, since **newCathode()** requires the local electric field vector. Furthermore, the caller must be changed to branch to the appropriate function based on a flag, and must pass the arguments required. Now errors introduced by the new function call and all its associated changes may propagate throughout the code, resulting in a large amount of time testing and debugging before X-Lab can have confidence in the updated software. Object-oriented technology can reduce this problem using polymorphism[4], as described below.

In this paper, we describe an object-oriented PIC architecture designed to address the issues of encapsulation[5], flexibility, extensibility and efficiency. In Section 2, we discuss several previous attempts to address these issues. In Section 3, an object-oriented technique for modeling device physics is described. In Section 4, we discuss the algorithms employed in the OOPIC (Object Oriented Particle-in-Cell) code, followed by the architecture and implementation of OOPIC. In Section 6, we summarize the advantages and disadvantages of the object-oriented approach, and discuss future directions of the ongoing research. Beyond the scope of this paper, parallel research is ongoing to develop embedded graphical analysis and configuration tools and an expert system tools for assistance in parametric input.

2 Background

Many computational physicists have realized the utility of the object-oriented concept and used structured programming languages to achieve a limited degree of object behavior in PIC codes. More recently, developers have adopted mainstream object oriented languages such as C++ and Object Pascal.

Several codes are implemented in structured languages with some elements of object-oriented technology. Eastwood [7] implemented MILO, a two-dimensional electromagnetic PIC code, in

[4] *Polymorphism* is the ability of an object to respond to a standard message with a specialized behavior. For example, the system may send a Paint() message to both circle and a square, without knowing the specific shape it is drawing.

[5] *Encapsulation* in this context refers to the isolation of data within objects; the object provides a defined interface to access and manipulate the data.

FORTRAN90. MILO decomposes the simulation region into localized blocks, connected by boundary conditions Eastwood terms *gluepatches*. The primary purpose of this approach is to facilitate running the code on massively parallel platforms.

The Plasma Device family of codes [4], PDP1, PDC1 and PDS1 are written in C with elements of object-oriented design on both the UNIX-X11 and DOS platforms. These codes implement polymorphic function calls using pointers to functions, and encapsulate behavior by storing function pointers within data structures. Since these codes are implemented in C, argument type checking can not be performed on the 'polymorphic' constructs. This deficiency can be remedied using an object-oriented language such as C++.

Early attempts at applying object-oriented technology to particle-in-cell simulation have resulted in codes with poor performance compared to traditional codes. Forslund [5] found C++ a powerful language to translate WAVE, but found disappointing performance compared to a similar FORTRAN code. One of Forslund's key objectives was to employ the object-oriented technique to simplify the parallelization of WAVE. Forslund compared timings for both optimized and unoptimized code, and found the FORTRAN version ran about twice as fast as the comparable C++ version. He attributes part of the difference to the finer granularity of the C++ representation and the difficulty of optimizing such code, and the balance of the difference to the greater maturity of FORTRAN compilers and libraries. Forslund also noted several deficiencies in the C++ specification for his application, particularly with regard to parallelization and linkage to C libraries when using templates.

Gisler [8] implemented a particle trajectory code using Object Pascal. This code demonstrated flexibility of the object method, choosing to represent particles with individual objects. The low-level representation did not prove a performance issue due to the nature of the code; only a small number of particle trajectories are needed in contrast to particle-in-cell codes which require a good statistical representation of particles as the source term for computing the electromagnetic fields.

Furnish [9] implemented a one-dimensional electrostatic PIC code with periodic boundaries in C++ on a distributed network of workstations. Although a relatively simple implementation, this code illustrates a number of object-oriented benefits, including operator overloading[6] and polymorphism. Furnish used multi-threaded function return futures, which postpone a stall until the return value of a function must be used in another expression, to improve the distributed performance compared to that of a vector processor.

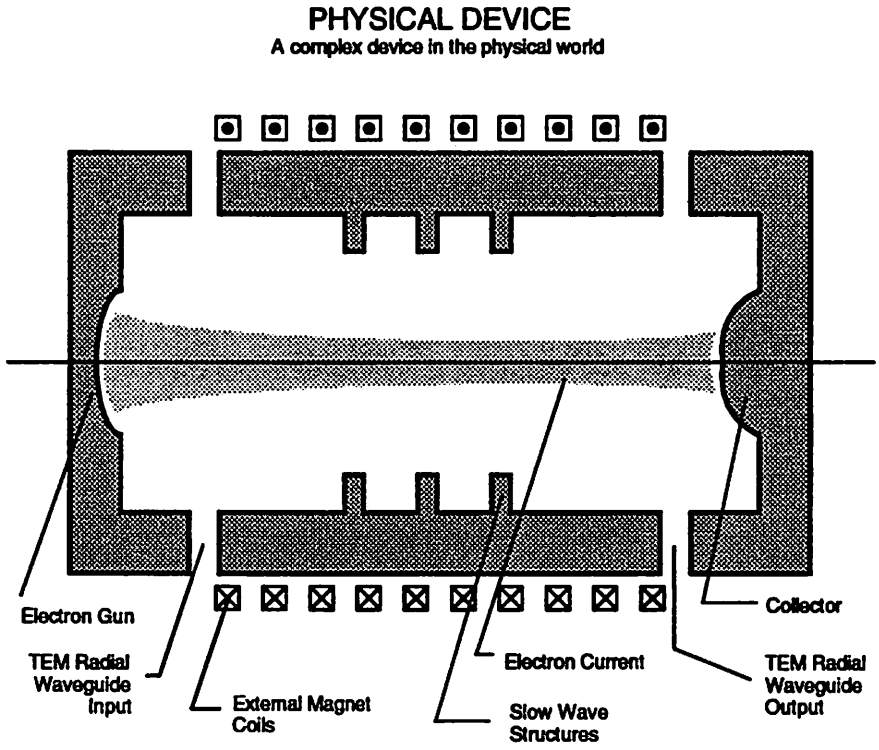

Figure 1. The disk loaded microwave tube is a simple example of a physical device.

3 Object Oriented Device Modeling

This section describes the methodology employed in modeling the physics of devices using object-oriented technology. Physical devices have a complicated relationship to the world outside the system under consideration, and exhibit complicated behavior internally as well. Modeling is

[6] *Overloading* refers to the replacement of a function with another of the same name. *Operator overloading* is polymorphism of an operator, such as '+', so that its behavior is operand-dependent. For example, the '*' operator may perform multiplication of real numbers, complex numbers, or compute the inner product of vectors.

an attempt to describe a physical device in mathematical terms. Simulation is the study of the behavior of a model to stimuli; note that this is different from studying the behavior of the physical device.

The object-oriented methodology is employed in going from the physical device to the mathematical model, and finally to the discrete model for simulation. The physical device, shown in Figure 1, includes the complete physical description of the a device in the physical world. This is the tool of the experimentalist, who must extract information from the device using a measurement and observation technique. Although the complete set of data is contained in the physical device, the act of observing the device can influence its behavior. Furthermore, any measurement involves compromises and limitations inherent in the diagnostic apparatus. Experimentation is often prohibitively expensive, and in many cases is not conducive to rapid variation of parameters or complete control of the external influences on the device.

For example, attempting to study the behavior of the disk-loaded microwave beam device shown in Figure 1 may involve constructing the device, applying a signal to the input waveguide, and measuring the signal at the output waveguide. Care must be taken in designing the device itself as well as the wave input generator, wave transmission system and output diagnostic. Each of these items can have a strong effect on the results of the experiment. If the experimentalist hypothesizes that the pitch or size of the slow wave structures controls the performance of the device, he must build a new device to test his theory. Thus, there exist many cases when experiment cannot provide sufficient detail on the behavior of a physical device, or characterize variations to the existing device without substantial time and expense.

The mathematical model describes the physical device in terms of the robust language of mathematics. An example of a mathematical model of a microwave beam device is shown in Figure 2. The model also includes a set of equations which are simplified from the full set of behaviors describing the physical device. This model can include such assumptions as azimuthal symmetry in cylindrical coordinates, for example. Even if virtually every aspect of a device is known and understood, it is seldom fruitful to describe the device fully. The mathematical model may also incorporate some effects with simplified descriptions, such as incorporating the quantum

mechanical affects of the electrons bound to surface aluminum atoms via a secondary emission model which provides an electron return current and velocity distribution for a given incident current and distribution.

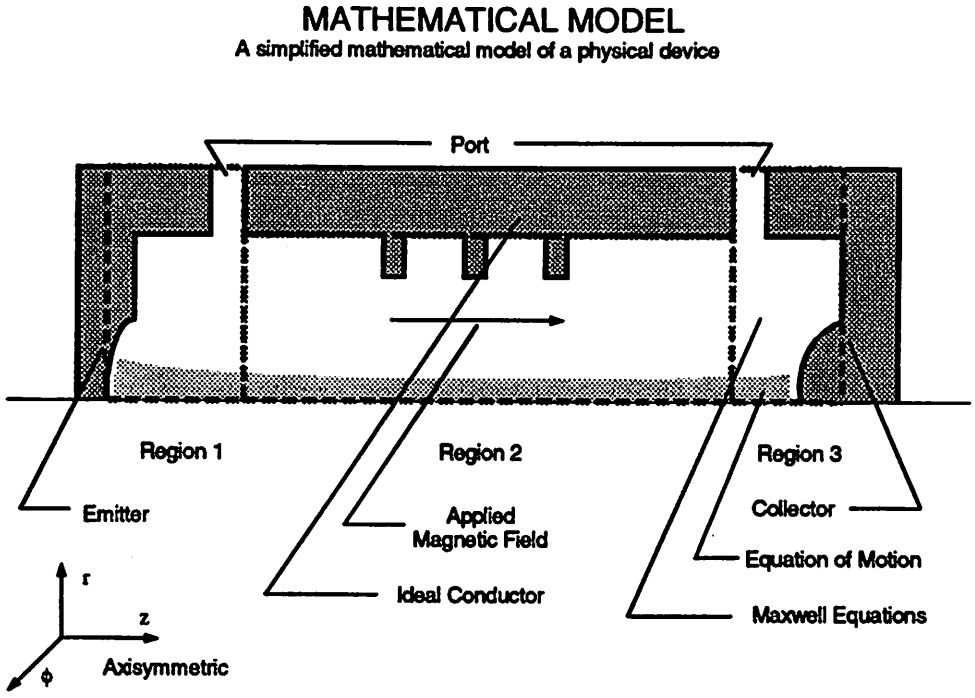

Figure 2 An example of a mathematical model of a microwave beam device.

The mathematical model may divide the device into various regimes with different physical properties which are best described with heterogeneous sets of equations. In Figure 2, the device is divided into three regions: the emitter (region 1), the beam circuit (region 2), and the collector (region 3). This division is useful since the emitter and collector regions are primarily governed by electrostatic processes, while the beam circuit region is governed by electromagnetic processes which vary on timescales several orders of magnitude faster than the electrostatic regions. The dashed lines in Figure 2 indicate the edges of each region, where boundary conditions are used to interface to other regions and the external world. Thus, each region in a mathematical model may be modeled with different sets of equations, each with its own simplifying assumptions, interacting through boundary conditions.

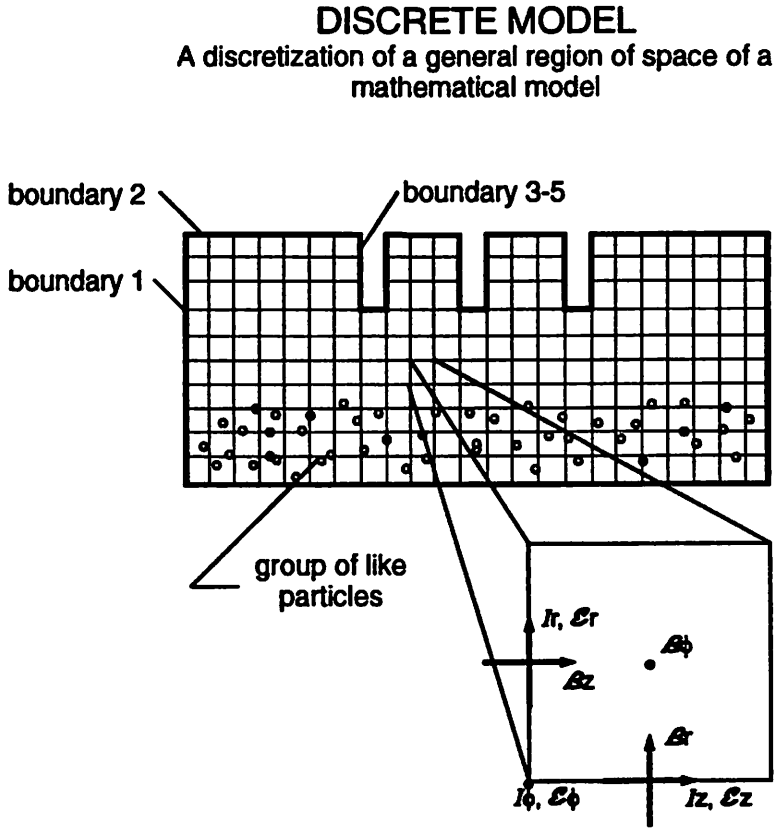

Figure 3. An example of the discretization of a mathematical model of a microwave beam device.

The mathematical model may be used to obtain an analytic solution, or may require a computational solution due to its complexity. With many physical devices, the smallest set of equations which sufficiently describe the device and can be solved analytically provide only a gross description of the device due to the simplifying assumptions made. A more complete description often requires a computational solution.

The discrete model, such as the discrete model of the beam-circuit region of the disk-loaded microwave beam device shown in Figure 3, provides an approximate description of the mathematical model. The discretization is necessary for simulation using a finite state, or digital, computer to solve the governing equations. The discrete model is independent of the implementation; both the structured and object-oriented approaches may share the same discrete model, and differ in the implementation. This model level describes the code representation of the

physical device, and in the case of an object-oriented paradigm the discrete model provides a natural guide for the choice of objects.

In the example shown in Figure 3, the space is gridded so that the electric and magnetic fields and current can be defined at discrete points. Values of gridded quantities can be approximated at intermediate points by interpolation. Particles are discrete representations of some *statistical* group of physical particles. Boundaries provide the boundary conditions necessary to complete the equations describing the particles and fields.

4 Algorithms

In this section, we describe the algorithms used in OOPIC to provide a basis for the following architecture discussion. The adaptation of these algorithms to the object-oriented model is described in the subsequent section, **Architecture and Implementation**. This discussion is not intended to provide a full analysis of the algorithms, stability and convergence properties, and extension to more general meshes and geometries; these topics will be the subject of a future paper.

For purposes of this discussion, we consider a device described in cylindrical z-r-ϕ coordinates, with azimuthal symmetry. Furthermore, the device is gridded by orthogonal quadrilaterals to maintain the simplicity of the implementation. The discussion will extend to Cartesian coordinates in a straightforward way.

4.1 Electromagnetic Fields

We consider the full set of electromagnetic fields with azimuthal symmetry. Many of the ideas for the electromagnetic field representation and solution can trace roots to Langdon [10] and Eastwood [7]. Maxwell's equations in integral form are written:

$$\int_S \mathbf{E} \cdot \mathbf{dS} = \int \rho dV,$$

$$\int_S \mathbf{B} \cdot \mathbf{dS} = 0, \tag{1}$$

$$\oint \mathbf{E}\cdot\mathbf{dl} = -\int \partial_t \mathbf{B}\cdot\mathbf{dS},$$

$$\oint \mathbf{H}\cdot\mathbf{dl} = \int \partial_t \mathbf{D}\cdot\mathbf{dS} + \int \mathbf{J}\cdot\mathbf{dS}.$$

We define a new set of field variables which encapsulate the mesh metrics:

$$\begin{aligned} \tilde{E} &= \int \mathbf{E}\cdot\mathbf{dl}, \\ \tilde{H} &= \int \mathbf{H}\cdot\mathbf{dl}, \\ \tilde{D} &= \int \mathbf{D}\cdot\mathbf{dS}, \\ \tilde{B} &= \int \mathbf{B}\cdot\mathbf{dS}, \\ I &= \int \mathbf{J}\cdot\mathbf{dS}. \end{aligned} \tag{2}$$

In Eq. (2), the line integrals are along cell sides, and the surface integrals are over surfaces formed by cell faces, where fields are as defined on the Yee mesh [11]. The constituency equations for the integral-form variables become

$$\begin{aligned} \tilde{\mathbf{E}} &= \mathbf{C}^{-1}\tilde{\mathbf{D}} \\ \tilde{\mathbf{H}} &= \mathbf{L}^{-1}\tilde{\mathbf{B}} \end{aligned}, \tag{3}$$

where $\mathbf{C}^{-1}$ and $\mathbf{L}^{-1}$ are coupling matrices with the dimensionality of capacitance and inductance, respectively. On a general mesh, the couplings can involve fields from one or more neighboring cells; the general mesh is beyond the scope of the present paper. On a non-uniform orthogonal Yee mesh in cylindrical *z-r* coordinates, the capacitances in OOPIC are written:

$$\begin{aligned} C^{-1}_{z,j+1/2,k} &= \frac{z_{j+1} - z_j}{\varepsilon\pi(r^2_{k+1/2} - r^2_{k-1/2})}, \\ C^{-1}_{r,j,k+1/2} &= \frac{r_{k+1} - r_k}{\varepsilon 2\pi r_{k+1/2}(z_{j+1/2} - z_{j-1/2})}, \\ C^{-1}_{\phi,j,k} &= \frac{2\pi r_k}{\varepsilon(r_{k+1/2} - r_{k-1/2})(z_{j+1/2} - z_{j-1/2})}. \end{aligned} \tag{4}$$

Similarly, the inductance elements for a non-uniform orthogonal Yee mesh in cylindrical *z-r* coordinates in OOPIC are:

$$L^{-1}_{z,j,k+1/2} = \frac{z_{j+1/2} - z_{j-1/2}}{\mu\pi(r^2_{k+1} - r^2_k)},$$

$$L^{-1}_{r,j+1/2,k} = \frac{r_{k+1/2} - r_{k-1/2}}{\mu 2\pi r_k (z_{j+1} - z_j)}, \tag{5}$$

$$L^{-1}_{\phi,j+1/2,k+1/2} = \frac{2\pi r_{k+1/2}}{\mu(r_{k+1} - r_k)(z_{j+1} - z_j)}.$$

The Maxwell curl equations can be written in terms of these variables and discretized. The transverse magnetic (TM) set becomes:

$$\begin{aligned}
\partial_t \tilde{B}_{\phi,j+1/2,k+1/2} &= \tilde{E}_{z,j+1/2,k+1} - \tilde{E}_{z,j+1/2,k} - \tilde{E}_{r,j+1,k+1/2} + \tilde{E}_{r,j,k+1/2} \\
\partial_t \tilde{D}_{z,j+1/2,k} &= \tilde{H}_{\phi,j+1/2,k+1/2} - \tilde{H}_{\phi,j+1/2,k-1/2} - I_{z,j+1/2,k} \\
\partial_t \tilde{D}_{r,j,k+1/2} &= \tilde{H}_{\phi,j+1/2,k+1/2} - \tilde{H}_{\phi,j-1/2,k+1/2} - I_{z,j,k+1/2} \\
Q_{j,k} &= \tilde{D}_{z,j+1/2,k} - \tilde{D}_{z,j-1/2,k} + \tilde{D}_{r,j,k+1/2} - \tilde{D}_{r,j,k-1/2}
\end{aligned} \tag{6}$$

where $Q = \int \rho dV$ over the cell volume. The indices refer to locations on the Yee mesh, and are ordered in z, r for a right-handed cylindrical coordinate system. The corresponding transverse electric (TE) set can be written:

$$\begin{aligned}
\partial_t \tilde{D}_{\phi,j,k} &= \tilde{H}_{r,j+1/2,k} - \tilde{H}_{r,j,k+1} - \tilde{H}_{z,j,k+1/2} + \tilde{H}_{z,j,k-1/2} - I_{\phi,j,k} \\
\partial_t \tilde{B}_{r,j+1/2,k} &= \tilde{E}_{\phi,j+1,k} - \tilde{E}_{\phi,j,k} \\
\partial_t \tilde{B}_{z,j,k+1/2} &= -\tilde{E}_{\phi,j,k+1} + \tilde{E}_{\phi,j,k} \\
0 &= \tilde{B}_{r,j+1/2,k} - \tilde{B}_{r,j-1/2,k} + \tilde{B}_{z,j,k+1/2} - \tilde{B}_{z,j,k-1/2}
\end{aligned} \tag{7}$$

The TM and TE field equations are advanced in time using a leap frog advance [1]. The source terms in Eqs. (6) and (7), I, are the currents resulting from charged particle motion. The field algorithms are completed by initial conditions and boundary conditions.

4.2 Particles

The particles follow the relativistic equations of motion in electric and magnetic fields, generating a source current for the field equations. The discretization of the relativistic equations of motion has been described extensively; OOPIC employs the relativistic time-centered Boris advance [1]. The position update occurs in a rotated Cartesian frame, which eliminates the problem of large angular displacements near the origin. Birdsall and Langdon discuss this issue in some detail [1].

We presently use a charge conserving current weighting algorithm, which ensures that Gauss' Law remains satisfied if it was initially satisfied. The current weighting is summarized in Figure 4 for a particle traversing multiple cells. The net particle motion is decomposed in segments, δx_1 and δx_2, separated by a cell crossing. The currents for each segment are deposited independently, so we describe only the first segment.

Let the initial particle location be x^n, and let the final location be x^{n^*} for this segment, so $\delta x_1 = x^{n^*} - x^n$. Let the particle be located in the mesh shown in Figure 4, where j and k are the cell indices such that $X_{j,k} \cdot (\hat{z} \text{ or } \hat{r}) \le (x_i \text{ or } x_f) \cdot (\hat{z} \text{ or } \hat{r}) \le X_{j+1,k+1} \cdot (\hat{z} \text{ or } \hat{r})$. Here, $X_{j,k}$ is the location of the j,kth node of the mesh.

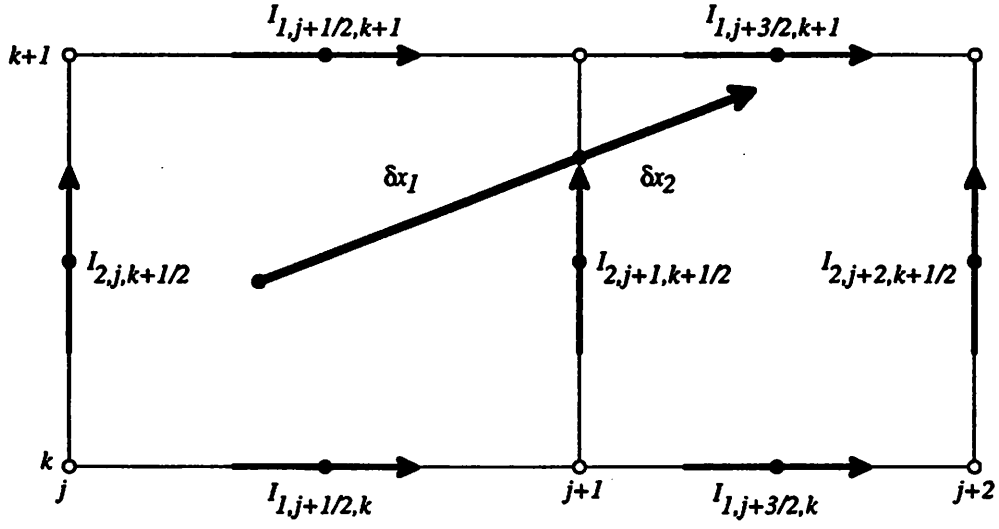

Figure 4. Current deposition for a multi-cell particle motion.

Defining $w_1^n = (x^n - X_{jk}) \cdot \hat{z}$ and $w_2^n = (x^n - X_{jk}) \cdot \hat{r}$, we can write the relations for linearly interpolating the particle charge onto the mesh nodes:

$$\begin{aligned} Q_{j,k}^n &= q(1-w_1^n)(1-w_2^n) \\ Q_{j+1,k}^n &= qw_1^n(1-w_2^n) \\ Q_{j,k+1}^n &= q(1-w_1^n)w_2^n \\ Q_{j+1,k+1}^n &= qw_1^n w_2^n \end{aligned} \quad , \tag{8}$$

where q is the particle charge. Defining $\Delta \mathbf{w} = \mathbf{w}^{n^*} - \mathbf{w}^n$ and $\bar{\mathbf{w}} = (\mathbf{w}^{n^*} + \mathbf{w}^n)/2$, we can write the change in charge at each mesh node due to the motion from $\mathbf{x}^n$ to $\mathbf{x}^{n^*}$:

$$\begin{aligned}
\Delta Q_{j,k} &= Q^{n^*}_{j,k} - Q^n_{j,k} = q[-\Delta w_1(1-\bar{w}_2) - (1-\bar{w}_1)\Delta w_2] \\
\Delta Q_{j+1,k} &= Q^{n^*}_{j+1,k} - Q^n_{j+1,k} = q[\Delta w_1(1-\bar{w}_2) - \bar{w}_1\Delta w_2] \\
\Delta Q_{j,k+1} &= Q^{n^*}_{j,k+1} - Q^n_{j,k+1} = q[-\Delta w_1\bar{w}_2 + (1-\bar{w}_1)\Delta w_2] \\
\Delta Q_{j+1,k+1} &= Q^{n^*}_{j+1,k+1} - Q^n_{j+1,k+1} = q[\Delta w_1\bar{w}_2 + \bar{w}_1\Delta w_2]
\end{aligned} \tag{9}$$

One set of currents which satisfies conservation of charge for the changes given in Eq. (9) is:

$$\begin{aligned}
I_{1,j+1/2,k} &= q\Delta w_1(1-\bar{w}_2)/\Delta t \\
I_{1,j+1/2,k+1} &= q\Delta w_1\bar{w}_2/\Delta t \\
I_{2,j,k+1/2} &= q\Delta w_2(1-\bar{w}_1)/\Delta t \\
I_{2,j+1,k+1/2} &= q\Delta w_2\bar{w}_1/\Delta t
\end{aligned} \tag{10}$$

where Δt is the time for the motion. Note that these currents are not unique; adding a constant current around the loop formed by the cell edges also satisfies the continuity equation. One sees immediately that this current deposition is equivalent to the method of Morse and Nielson [11] for a single cell particle motion. For multiple cells, the method described here deposits current in each cell traversed, in a manner similar to Eastwood [13] and Villasenor and Buneman [14].

4.3 Boundary Conditions

The boundary conditions for fields and particles in an electromagnetic PIC code can be diverse and complex. Currently, OOPIC includes boundary conditions for the electric fields at the surface of ideal conductors, the cylindrical axis, and incoming and outgoing wave ports. We make no attempt here at a general discussion of electromagnetic boundary conditions.

For the ideal conductor, the component of the electric field tangential to the surface is zero. This can be achieved by setting the appropriate capacitance matrix element, C, to zero. This allows the field solve to proceed without testing for electric fields or explicitly setting the tangential fields to zero along conductors.

The field boundary conditions at the axis in cylindrical coordinates follow directly from Maxwell's equations. If $\partial_t B_z(z, r=0)$ is finite, then $E_\phi(z, r=0) = 0$. If $\partial_t D_z(z, r=0) + J_z(z, r=0)$ is finite, then $H_\phi(z, r=0) = 0$. Finally, $B_r(z, r=0)$ is a constant on axis and $\partial_t D_r(z, r=0) = -J_r(z, r=0)$.

The present incoming and outgoing wave boundary conditions in OOPIC are incomplete. The incoming wave boundary condition simply sets the fields to the desired electromagnetic mode at the boundary of interest, which does not address the outgoing wave at that surface properly. The outgoing wave boundary condition is a surface impedance method. It sets the ratio of the tangential components of the electric and magnetic fields to the local surface impedance of boundary medium. This method is amenable to attenuation of the high frequency components by ω-space reduction of historical field values.

Particle boundary conditions include absorption and transmission. Particles are transmitted by symmetry boundaries such as the axis of symmetry in cylindrical coordinates. All other boundaries absorb particles, and can optionally emit secondary electrons.

5 Architecture and Implementation

The code architecture is summarized in Figures 5 and 6. Figure 5 is a schematic representation of the flow control for a timestep. A similar flow control diagram can be drawn for the traditional structured PIC code. After initializing the simulation, the parameters are used to construct the simulation. Once constructed, the simulation is advanced in discrete units of time.

The fields are advanced using the field and source values from previous times, including the appropriate boundary conditions. The field advance consists of a half timestep advance for **B**, followed by an a full timestep advance for **E**, followed by application of the boundary conditions for **E**, and a final half timestep advance for **B**. The half timestep splitting of the magnetic field advance results in electric and magnetic fields at integral timesteps at the end of the field solve, while taking advantage of the accuracy of the time-centered difference equations. The updated fields are then interpolated to the node points of the mesh from the locations shown in Figure 3.

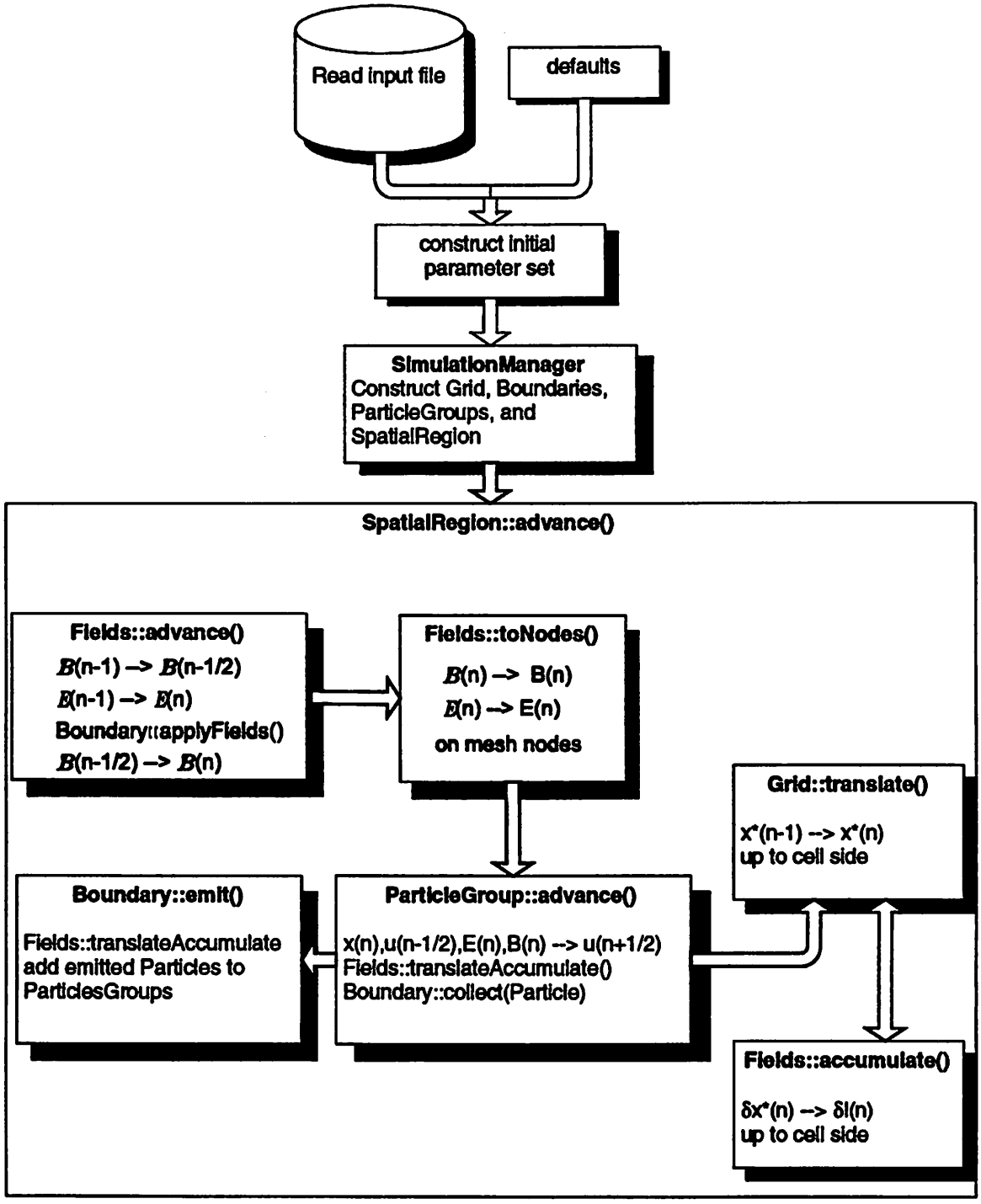

Figure 5. OOPIC flow control diagram for a single SpatialRegion.

Next, the forces on particles are computed by interpolating the fields to the particle position, and used to update the particle velocity, and subsequently the particle position. The position update

occurs in conjunction with the deposition of current. The relative displacement of the particle is computed in physical units (meters for example). The particle is advanced to successive cell side intersections, updating the position to the intersection point and depositing the current corresponding to that segment of the particle displacement to the mesh. At each cell edge, the appropriate boundary conditions are applied if the edge is a boundary.

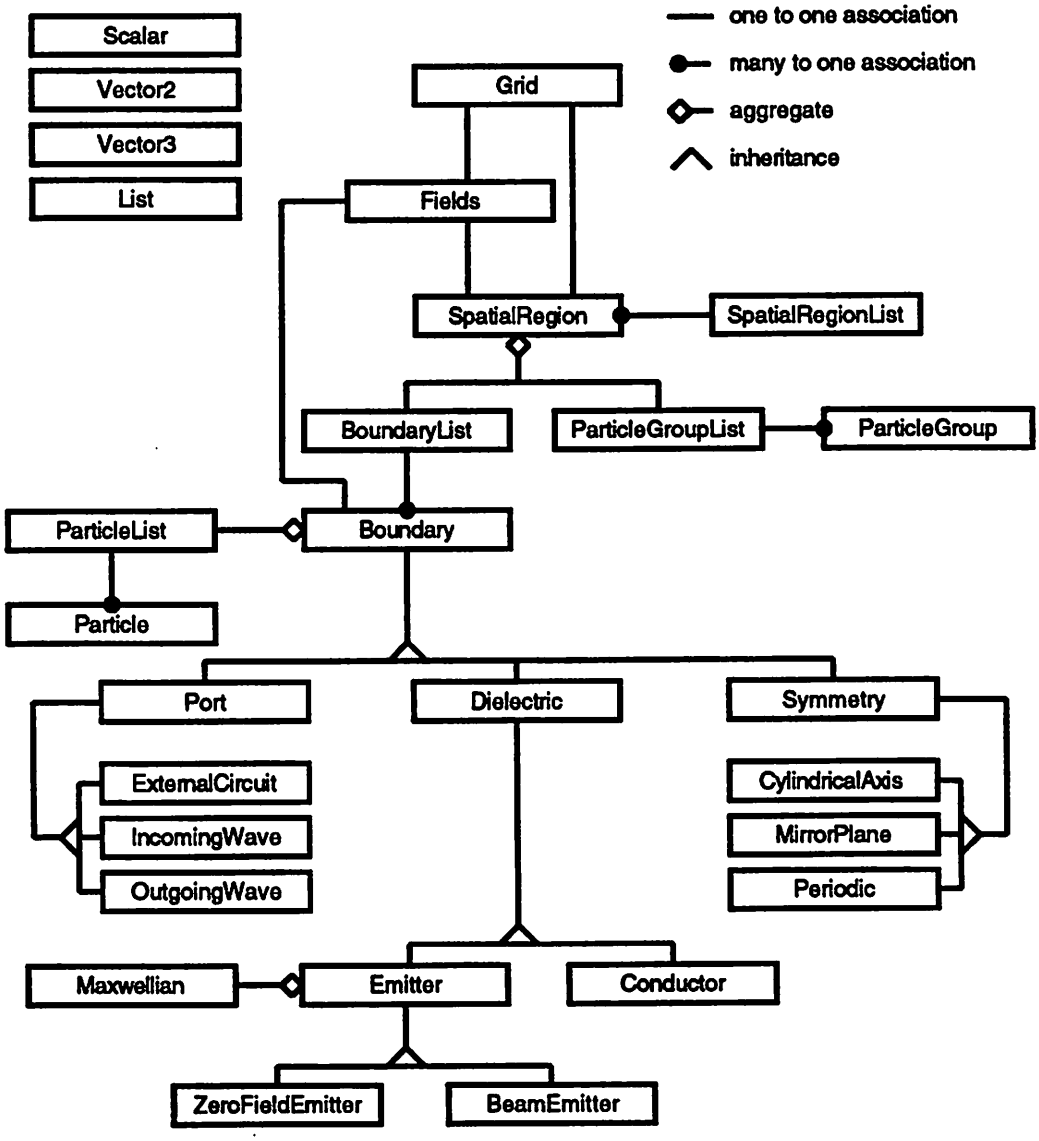

Figure 6. Class hierarchy for the OOPIC code using Rumbaugh [6] notation.

Finally, the boundary conditions for particle emission are applied. The emission includes transmission of particles through symmetry boundaries as well as emission due to particle current

injected into the system. The emission phase uses the same mechanisms as the particle push phase to advance particle position and deposit current. The sequence is repeated for each incremental advance of time.

Given the sequence of events comprising a timestep and the discrete model, we can now define the objects and their relationships. The class hierarchy, in Rumbaugh notation [6], is shown in Figure 6. An aggregate relationship indicates a class is comprised of one or more other classes, in the same way a chair is comprised of legs, a seat and a back. An association indicates an interaction or relation between classes, such as the relation between a chair and the floor.

A number of utility classes are defined. These include **Scalar**, **Vector2**, **Vector3**, **List**, and **Maxwellian**. **Scalar** provides a representation of a scalar quantity such as the mass or charge of a particle. **Vector2** and **Vector3** represent two and three component vector objects, respectively. The **List** is a simple yet versatile container class, which is used for storing singly linked lists as well as stacks of objects. The **Maxwellian** class represents a maxwellian distribution function, useful for describing the particle emission properties of a cathode, for example.

The vector classes provide component storage as well as overloaded operators and common vector operations such as dot and cross products. Although other authors, including Furnish [9], concluded there was little benefit in a vector class, we find the notation compact and powerful, resulting in a more readable and maintainable code. Care must be exercised in the design of fundamental classes such as **Vector2** and **Vector3** for performance critical situations, particularly with regard to creation of temporary intermediate objects for function return values [15].

The **SpatialRegion** class represents a defined region of the discrete model. A device can be divided into **SpatialRegions** based on time and space scales as well as for parallel processing or for reduction of the number of empty cells for non-rectangular devices. For example, a klystron cavity may force a rectangular mesh to waste many cells inside metal. The **SpatialRegion** provides a single interface to the objects it 'contains'. Each **SpatialRegion** has its own **Fields** and **Grid** objects. In addition, each has its own list of boundaries which border the region, **boundaryList**, and its own list of particles stored in **particleGroupList**. It stores the local

simulation time, and provides a single public interface for advancing all its constituents in time, **advance()**. The **advance()** message initiates the sequence of events shown in Figure 5.

The **Fields** class represents the electromagnetic fields on the mesh for a given SpatialRegion. The fields include electric, magnetic, and current. It can also store the charge density and the scalar potential in space. **Fields** provides a public interface to the fields at arbitrary locations using the interpolation capability of **Grid**. The **advance()** message, sent by **SpatialRegion**, updates the field equations in time. The **translateAccumulate()** method advances the particle position using **Grid::differentialMove()** and simultaneously deposits the particle current while checking particle boundary conditions as it crosses cell edges. Some may argue that this is a less object-oriented way of updating particle position and current, but the performance gain which results from the simultaneous operations is substantial since the accumulation of source current is a performance bottleneck.

The **Grid** class represents the discretization mesh for a **SpatialRegion**. This class stores the positions of the mesh nodes in physical units for both non-uniform and non-orthogonal general quadrilaterals. The field algorithms for general non-orthogonal quadrilaterals are not yet implemented in OOPIC. It also stores an array of cell edges with boundary conditions. **Grid** provides the coordinate transformations between logical, or code, coordinates and physical coordinates using the methods **getMKS()** and **getGridCoords()**. The **interpolateBilinear()** method interpolates a discrete vector field onto a specified location in the mesh. One could add additional interpolation methods to change the order of the splines used for particle-grid interactions without affecting code outside **Grid**. A number of methods provide integrals along grid lines and surface integrals used for normalized variables in the code, enabling the developer to effect a coordinate system change from within the **Grid** object with minimal impact on external code. The **translate()** method computes logical coordinates for a relative displacement in physical units using a step-wise cell-intersection method, eliminating the need for a cell search algorithm [16 and 17] when the grid is non-uniform or non-orthogonal. The **differentialMove()** method computes a relative physical displacement for a given velocity, and performs a coordinate rotation on the velocity vector in cylindrical coordinates [1].

The **ParticleGroup** class represents a group of similar particles. This class stores the particles in an array format, and stores the common mass, charge, and numerical particle weighting. This class provides a number of overloaded **add()** methods for adding new particles to a group, a **gamma()** method to compute the relativistic mass, and an **advance()** method to advance the equations of motion for this **ParticleGroup**.

The **Particle** class describes an individual particle, including its position, momentum, mass, charge, and numerical weight. **Particle** objects provide a flexible mechanism for passing particles between **Boundary** and **ParticleGroup** objects. This construct is only used when flexibility outweighs performance considerations for a small number of particles.

The choice of grouping particles based on a common charge to mass ratio is critical to performance; this ratio must be computed for each particle to advance the equations of motion each timestep. Both the expense of the division and the cost of accessing the data scale linearly with the number of particles if grouping is not used. If stored independently, additional redundant storage for the mass, charge, and numerical weight is required, and the non-contiguous memory locations will neither cache nor vectorize well. Furthermore, if particles are taken as true objects, each must have its own polymorphic **advance()** method, with the consequence of a virtual function call per particle per timestep. Note that there are two apparent grouping mechanisms which provide the best performance: arrays for the position, **x**, and velocity, **v**; or arrays of a particle object which includes only particle data, without functions. For the latter, it is critical that the particle data is incorporated directly into the object, rather than pointers to the data. A good compiler can optimize references to the data included directly to an offset from the base address of the object, increasing cache hits and eliminating further dereferencing. Both grouping techniques can provide performance equivalent to that of a structured representation such as multi-dimensional arrays.

The **Boundary** class represents both physical and logical boundary conditions for fields and particles. Physical boundary conditions include conductive walls, waveguide ports, and emitting surfaces. Logical boundaries refer to boundaries placed between **SpatialRegions** for purposes of

separating the regions and localizing the algorithms in each region. **Boundary** objects store the location of the boundary condition on the grid, the location of the associated **Fields** object, and a linked list of the **Particle** objects associated with the **Boundary**. A **Particle** is attached to a **Boundary** when it is collected, or when it is queued for emission. Presently, **Boundary** objects can apply three boundary conditions. The **applyFields()** method operates on the electromagnetic fields, applying all field-related boundary conditions except passive boundary conditions. Passive boundary conditions are only applied at initialization, and include setting C^{-1} to zero for conductors. The **collect()** method collects particles which pass through the boundary surface, placing them in a list. The collected particles may be used for diagnostics, transmission to another **SpatialRegion**, etc. The **emit()** method emits particles from the boundary, including emitting, transmitting and symmetry boundaries. This method uses the same mechanism to update the particle position as **ParticleGroup::advance()**, ensuring consistent treatment of particle equations of motion and current deposition.

The **Boundary** class is an abstract class which provides a polymorphic base for its derivatives. The **Boundary** subclasses are shown in Figure 6. The use of inheritance[7] and polymorphism provides a significant benefit when implementing new boundary conditions. We have often modeled a new boundary condition in this scheme by adding only a few lines of code, while taking advantage of the existing well-tested functionality. To add a new boundary condition in OOPIC, one may overload one or more of the existing methods described above and place the boundary object in the boundary list for the appropriate **SpatialRegion**; the **applyFields()**, **collect()**, and **emit()** messages are automatically passed to the new boundary object at the proper time.

This architecture is implemented in OOPIC using Borland C++ on the 80x86 PC platform, with a GUI operating under Windows 3.1 [18]. In addition, XOOPIC is the X11-based Unix version written in GNU G++, using the XGrafix user interface [19]. The physics source code is identical

---

[7] *Inheritance* enables a derived class (also called subclass or child class) to inherit properties (data) and behaviors (methods or functions) from a base class (also called superclass or parent class). The derived class may then specialize by adding or modifying properties and behaviors. Inheritance can occur over many levels forming a hierarchy analogous to the classification hierarchy of biology, with kingdom, phylum, subphylum, ..., and species. Inherits from more than one base class is called multiple inheritance.

in both cases, making use of inline[8] and virtual[9] functions, inheritance, and templates[10]. There is presently neither exception handling nor run-time type support in G++, so these features have not yet been implemented in OOPIC.

6 Conclusions and Future Work

We have found one of the principal advantages of the object-oriented method is the potential for rapid extension and enhancement of the code. After a design period of about six months, followed by six months of initial development and design revision, we have found that adding new code requires far less effort than testing and verifying the new algorithms. It is important to note that OOPIC is also serving as a platform for testing novel electromagnetic PIC algorithms and techniques, so the effort in testing the algorithms is usually unrelated to the object-oriented paradigm.

We are now seeing the payoff of the object oriented technology in the ease of extending the present set of models. For example, the **ZeroFieldEmitter** class, shown in the OOPIC class hierarchy (Figure 6), describes a surface which emits electrons due to physics quite different from the **BeamEmitter** class, which simply injects a given current. The difference is analogous to a voltage source versus a current source in an electrical circuit. Both **ZeroFieldEmitter** and **BeamEmitter** are subclasses of the **Emitter** class, and inherit the basic emitter behavior, as well as sharing a common model for collection of particles and a common boundary condition for the electromagnetic fields. Thus, only the **emit()** function must be overloaded, and appropriate constructor and destructor code provided. The balance of the code does not need to distinguish

---

[8] *Inline* functions are functions whose code is placed at the origin of the call. This eliminates the overhead of a function call, improving speed. Note that inline functions often increase the executable image size, although short inline functions can reduce the size when the function code is smaller than the overhead to set up the stack and call a non-inline function.

[9] *Virtual* functions are a C++ mechanism for providing polymorphism. Virtual functions are resolved dynamically at run time by searching a hash table, potentially impacting performance.

[10] *Templates* are a C++ mechanism for providing type safety for generic classes. For example, OOPIC uses templates to implement linked lists and stacks of such objects as Particles and Boundaries. Although all lists have the same base code, templates enforce type checking on list elements to ensure only the proper data type is used.

between specific **Boundary** subclasses, using the polymorphic nature of the design to hide the implementation details so all boundaries are treated alike.

The notation and operator overloading of C++ enables the developer to write a code in a notation very similar to the mathematical model. OOPIC was written using vector operations where possible to reduce the lines of code, with the consequence of reducing the probability of error and increasing code readability. The convenient encapsulation of objects makes OOPIC resemble the physical device much more closely, leading to more intuitive extension and use of the code.

We have not yet formally optimized OOPIC, so performance is only addressed in a qualitative manner here. However, we believe the code can achieve the same level of performance as a structured code. Some object-oriented constructs reduce performance, most notably virtual functions and the construction of temporaries. Conversely, standard C++ provides several performance-enhancing features, such as inline functions, templates and reference types for parameter passing. We have avoided using virtual functions in loops, and we make liberal use of inline functions, particularly within looping constructs. For example, the vector classes are comprised entirely of inline functions, since they have a strong impact on performance.

It is clear to us that object-oriented design is superior for this type of computational code. We have developed many PIC codes using the structured programming paradigm, and have found the object oriented approach results in increased productivity and code quality. Another important facet of a PIC code is the extensibility. We have found OOPIC to be far easier to extend than any of our previous codes due to the reduction of interactions and elegance of notation. For example, adding a slow wave structure consisting of three azimuthally symmetric vanes to a cylindrical beam-cavity simulation required only *three lines of code* to describe the location of each vane, and the vanes automatically collected particles and shorted tangential fields.

The primary purpose of this architecture is to provide a testbed for new plasma simulation algorithms, making modeling extensions effortless. Ongoing research includes building the capability to simulate multiple SpatialRegions, consideration of parallel processing issues, and addition of models for boundary conditions, field solution, fluid representations and collisions.

Extension of the present model to Cartesian and r-ϕ cylindrical coordinates on non-uniform, non-orthogonal body fitted meshes continues; the next logical step is extension to three dimensions. A graphical expert system is under development to aid the user in configuring and running a simulation, and both real time and postprocess graphics development continue. In addition, we continue to refine and optimize the architecture.

7 Acknowledgements

This research was supported by the Air Force Office of Scientific Research under grant F49620-92-J0487. We gratefully acknowledge the contributions of C. K. Birdsall, W. Peter, K. Cartwright, and P. Mardahl, as well as the contributions of our coworkers at George Mason University and FM Technologies.

8 References

1. C. K. Birdsall and A. B. Langdon, *Plasma Physics Via Computer Simulation*, Adam-Hilger (1991).

2. B. Goplen, L. Ludeking, D. Smithe and G. Warren, *MAGIC Users Manual*, Mission Research Corp. Report no. MRC/WDC-R-282, Newington, VA (1991).

3. K. Birman, R. Cooper, T. Joseph, K. Kane and F. Schmuck, *The ISIS System Manual*, Version 1.2 (1989).

4. PDx1 (Plasma Device 1 Dimensional, x = {P = planar, C = cylindrical, S = spherical)}, PDC1, and PDS1 are available from the Industrial Liason Program, University of California, Berkeley, CA 94720, software@eecs.Berkeley.EDU (1993).

5. D. W. Forslund, C. Wingate, P. Ford, J. Jackson, and S. C. Pope, "Experiences in Writing a Distributed Particle Simulation Code in C++," *Proc. 1990 USENIX C++ Conference* (1990).

6. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall (1991).

7. J. W. Eastwood, "Finite Element EM PIC," *Unpublished Proc. 1st OOPIC Workshop*, Fairfax, VA (1992).

8. G. Gisler, "Simulations of Charged Particles in Time Dependent Electromagnetic Fields Using Object Programming," *Unpublished Proc. 1st OOPIC Workshop*, Fairfax, VA (1992).

9. G. Furnish and M. G. Gray, "Distributed Plasma Simulation Using ESP," *Unpublished Proc. 1st OOPIC Workshop*, Fairfax, VA (1992).

10. A. B. Langdon, "On Enforcing Conservation Laws in Electromagnetic Particle-in-Cell Codes," *Proc. 14th Int. Conf. Num. Sim. Plasmas*, Annapolis, MD (1991).

11. K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Ewuations in Isotropic Media," *IEEE Trans. Antennas Prop.* **14** (1966).

12. R. L. Morse and C. W. Nielson, "Numerical Simualtion of the Weibel Instability in One and Two Dimensions," *Phys. Fluids* **14** (1971).

13. J. W. Eastwood, "The Virtual Particle Electromagnetic Particle-Mesh Method," *Comp. Phys. Comm.* **64** (1991).

14. J. Villasenor and O. Buneman, "Rigorous Charge Conservation for Local Electromagnetic Field Solvers," *Comp. Phys. Comm.* **69** (1992).

15. R. M. Adams, "Temporary Object Management Through Dual Classes," *C Users J.* **12**:5 (1994).

16. D. Seldner and T. Westermann, "Algorithms for Interpolation and Localization in Irregular 2D Meshes," *J. Comp. Phys.* **79** (1988).

17. T. Westermann, "Localization Schemes in 2D Boundary-Fitted Grids," *J. Comp. Phys.* **101** (1992).

18. J. Acquah, "TheOOPIC Visualization & GUI Framework," *Unpublished Proc. 4th OOPIC Workshop*, Los Gatos, CA (1994).

19. V. Vahedi and J. P. Verboncoeur, "XGrafix: An X-Windows Environment for Real-Time Interactive Simulations," *Proc. 14th Int. Conf. Num. Sim. Plasmas*, Annapolis, MD (1991).