Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)
La Cité des Sciences et de l'Industrie, Paris, France, October 27-31, 2013

Geometric Templates for Improved Tracking Performance in Monte Carlo Codes

Brian R Nease[*], David L. Millman, David P. Griesheimer, Daniel F. Gill

Bettis Laboratory, West Mifflin, Pennsylvania, USA
[*] Corresponding Author, E-mail: brian.nease.contractor@unnpp.gov

One of the most fundamental parts of a Monte Carlo code is its geometry kernel. This kernel not only affects particle tracking (i.e., run-time performance), but also shapes how users will input models and collect results for later analyses. A new framework based on geometric templates is proposed that optimizes performance (in terms of tracking speed and memory usage) and simplifies user input for large scale models. While some aspects of this approach currently exist in different Monte Carlo codes, the optimization aspect has not been investigated or applied. If Monte Carlo codes are to be realistically used for full core analysis and design, this type of optimization will be necessary. This paper describes the new approach and the implementation of two template types in MC21: a repeated ellipse template and a box template. Several different models are tested to highlight the performance gains that can be achieved using these templates. Though the exact gains are naturally problem dependent, results show that runtime and memory usage can be significantly reduced when using templates, even as problems reach realistic model sizes.

KEYWORDS: Monte Carlo, template, overlay, efficiency, geometry, tracking

I. Introduction

Geometric tracking typically accounts for one third to one half of the run time for large-scale Monte Carlo transport calculations. Many production-level Monte Carlo codes use constructive solid geometry (CSG), where each geometric object is defined by the union, intersection, and/or complement of either 1) the boundary defined by a set of quadratic surface equations or 2) other solid body objects.(1),(2),(3) CSG is very flexible and allows for accurate representation of most realistic geometries, which is one of the strongest selling points of the Monte Carlo method. Typically, developers will invest significant time in the early stages of the code optimizing the tracking algorithms for memory usage and speed. However, even with optimization, this type of geometric representation is more expensive than a dedicated system that handles only one specific type of geometry (e.g., tracking over a structured mesh).

In this paper, a new geometric framework is developed, which is based on dedicated tracking systems called templates. These templates are specifically developed and tailored to known design applications, allowing the code to take advantage of simple, commonly repeated, geometric shapes. As a consequence, templates can greatly improve tracking speed and memory usage, even as model sizes increase.

II. Overview of Templates and Overlays

Most reactor core models use regularly repeated geometric shapes, such as the cylindrical pins in PWR / BWR fuel assemblies. While a model could be constructed with each of these pins represented using CSG, this is not necessarily efficient. This is because the representation cannot take advantage of known geometric characteristics, such as the cylindrical shape of each pin, the regular spacing between pins, the hierarchical relationship between the coolant, cladding, fuel, etc.

Many codes recognize the inherent problems with representing every object explicitly and have implemented various ways to improve efficiency. For instance, MCNP allows users to define universes, lattices, and repeated structures.(3) Lattices allow for a hexagonal prism or hexahedra subdivision of a CSG objects. Universes allow geometric hierarchy by defining a collection of objects in one coordinate system and then placing those objects inside another. Repeated structures allow users to geometrically replicate a collection of objects but assign them different properties. While these features improve memory usage and simplify user input, they do not improve tracking speed.(3)

Previous versions of the MC21 code also had options to improve tracking efficiency. In 2006, Donovan and Tyburski reported on the use of a dedicated system for tracking over collections of elliptical cylinders.(4) Their results demonstrated that the use of a dedicated system could significantly improve tracking speed. However, their original implementation was limited and could only be used to model a 2D lattice of repeated ellipses. Other geometries could not be modeled without significantly restructuring the code. Also, the original implementation did not allow for

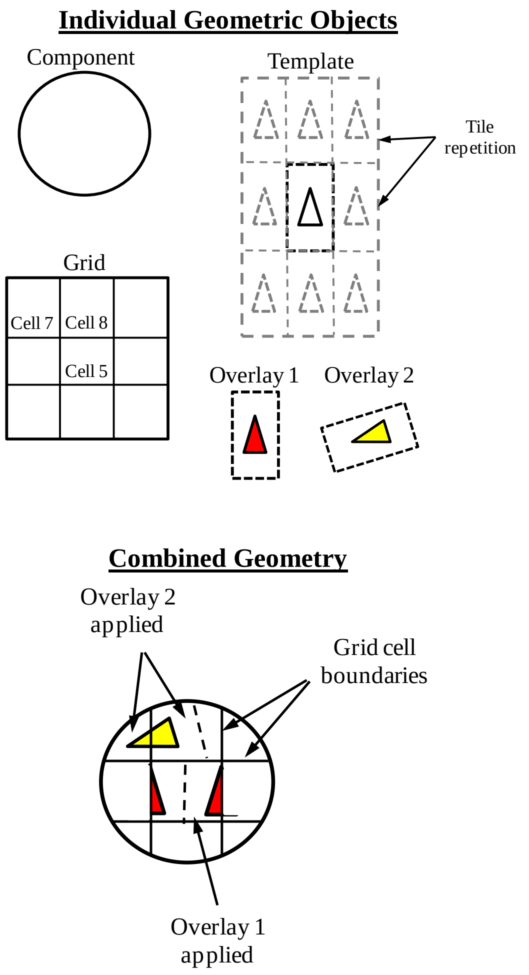

Figure 1: The top is an illustration of the individual geometric objects, including a component, grid, template, and overlay assignments. The bottom is an illustration of those objects combined into a complete geometry. The grid is assigned to the component. Overlay 1 is assigned to grid cell 5, and overlay 2 is assigned to grid cells 7 and 8. Color indicates material assignment.

the reuse of the underlying ellipse definition. Every definition (even those that were geometrically identical) had to be repeated on input and in memory, making it cumbersome and memory intensive.

In this paper, a new framework is introduced based on templates and overlays. Though this framework can be modified to work in most Monte Carlo codes, this paper will focus on its implementation in MC21. In MC21, there are three levels of representation: components, grids, and overlays. A *component* is a 3D solid-geometry object constructed from the unions and intersections of primitives. Every model must contain at least one component, which defines the maximum extent of the solution space. Additional geometric detail can be added by assigning the component a *grid*, which subdivides the interior of the component into a collection of grid cells. Any part of a grid that extends beyond the bounds of its component is truncated. Additional detail can be added by assigning an *overlay* to a grid cell. An overlay is a collection of cells (referred to as *overlay cells*) that represent simple geometric objects. In practice, the shapes and relative arrangements of these overlay cells are often repeated throughout the model. However, the overlay cells are usually assigned different properties (such as material or temperature) and have a different overall placement within the model. For this reason, an *overlay* is used to define the properties and placements of the overlay cells, and a *template* is used to define the shapes and relative arrangements of the overlay cells. For convenience, the geometric objects defined by a template are referred to as *template cells*. An overlay assigns the template cells properties and places them within the model, making them overlay cells. Any part of an overlay cell that extends beyond the bounds of its grid cell is truncated. Multiple overlays can use the same template, allowing for a given arrangement of geometric objects to be defined once and reused. This results in improved memory usage, since multiple copies of the same geometry do not need to be stored. When a template is first created, the template cells are defined on a *tile*, which is a unique coordinate frame that can be either finite or infinite in dimension. If a tile is finite, then the entire tile (and all of the template cells contained within it) is repeated infinitely in all directions.

An example of this geometry system is illustrated in Figure 1. At the top of the figure is each of the individual objects: a component, grid, template, and two overlays. The template is defined on a finite tile, so it repeats in each direction. Both overlays use the same template, but each has a different rotation. The bottom of the figure shows the result of combining these objects into a complete geometry. The grid is assigned to the component, and the component boundaries truncate the grid boundaries. Overlay 1 is assigned to grid cell 5. The overlay is placed such that the tile is repeated within the grid cell, and the grid cell boundaries truncate overlay cell boundaries on each tile. Overlay 2 is assigned to both grid cells 7 and 8. By assigning overlay 2 in this way, the entire overlay cell can be represented (instead of it being truncated by the grid cell boundaries). However, even though the same overlay is applied to two different grid cells, there is only one set of properties, since both grid cells were assigned overlay 2. Thus, if this model were depleted, the overlays assigned to grid cells 7 and 8 (colored in yellow) would deplete together, while the overlay assigned to grid cell 5 (colored in red) would deplete separately.

The shapes and arrangements of template cells are limited by design. This allows for tracking routines to be specifically tailored to each type of template. Developers can easily create new templates based on the specific needs of their users. Such an approach is highly amenable to object-oriented programming. The next section of this paper will discuss the implementation of this framework into the MC21 code, which is written using Fortran 2008 standards.

III. Template/Overlay Methods and Data Structures

In the MC21 code, each instance of the overlay data structure has data members that include an overlay identifier, an assigned template identifier, a translation vector, a rotation matrix, and properties for every overlay cell. The overlay has a single method:

`transform(p)`*: Given a particle p in the grid coordinate frame, return the coordinates of p in the template coordinate frame.*

The shared data and methods of templates are represented with an abstract base class. All data and methods that are unique to a specific template type are represented as extensions of that base class. The shared template data consists of a template identifier. It has a single method that is shared by all templates:

`handleScatter(p,d)`*: Given a particle p, handle a scatter event at a distance d along the trajectory of p.*

The base class also has four virtual functions, which must be overwritten by all specific templates:

`inside(T,p)`*: Given a particle p located in template T, return the template cell containing p.*

`distance(p,c)`*: Given a particle p that is inside template cell c, at coordinate q, and moving in direction d, return the distance from q to the boundary of c along d.*

`moveToBoundary(p,c)`*: Given a particle p inside template cell c, move particle along its direction vector to the boundary of c.*

`crossBoundary(p,c)`*: Given a particle p inside of template cell c, cross the boundary from c into the neighboring cell.*

To make these functions as efficient as possible, it is important to identify what information can be reused among operations. For instance, during particle transport, much of the information computed in the `inside` routine can be reused in calls to the `distance` routine (calls to the `inside` routine are almost always followed by calls to the `distance` routine). In order to reduce the overall number of operations performed (and thereby increase tracking speed), the templates are designed to take advantage of a generic geometry caching scheme. Each template type maintains its own cache that cannot be modified by other template types. These caches are designed to store any information that is useful for repetitive calculations. Some examples include the last template cell found in an `inside` call, local coordinate and direction vector, etc. Template caches also store information that indicates whether or not the cache is stale, in which case it must be recomputed. It should be emphasized that there only needs to be one cache per template type (not per template or per overlay), since a particle can only be in one template at a time. Thus, templates can store any useful information, without creating excessive memory requirements.

The template framework also works naturally with delta scattering. (5) At a minimum, each template must have an `inside` routine. The `distance`, `moveToBoundary`, and `crossBoundary` routines are only necessary for performing full particle tracking. This aspect makes the template framework very appealing because new template types can be quickly prototyped by only developing the `inside` routine. If it is determined that a given template is particularly useful, then developers can later write the other routines to offer more robust tracking and tally options.

Currently, MC21 offers three types of templates: a repeated ellipse, a repeated ellipsoid, and a box. The repeated ellipse and box templates will be discussed in detail in the following sections. The repeated ellipsoid template is a generalization of the repeated ellipse template into three dimensions. Since the algorithm and data structures are nearly identical for the two templates, the repeated ellipsoid template will not be discussed.

IV. Repeated Ellipse Template

The repeated ellipse template is a two-dimensional template used to model an infinitely repeating pattern of parallel, extruded elliptical cylinders. This is especially useful for modeling structures such as bundles of fuel pins.

The repeated ellipse template allows for one or more ellipses located partially or wholly within the tile. The tile is finite in the *x*- and *y*-dimensions, but has infinite extent in the *z*-dimension. Any portion of an elliptical cylinder that extends beyond the tile boundary is truncated. An arbitrary number of ellipses can be defined in the interior of a tile. Transforming these 2D ellipses to the 3D space of the geometric cell is equivalent to extruding the ellipses infinitely along the *z*-axis. The ellipses may have arbitrary positions, rotations, and lengths along the semi-major and semi-minor axes. The bounds of any two ellipses cannot intersect within the tile, though they can be nested within one another. Figure 2 shows an example of an ellipse template.

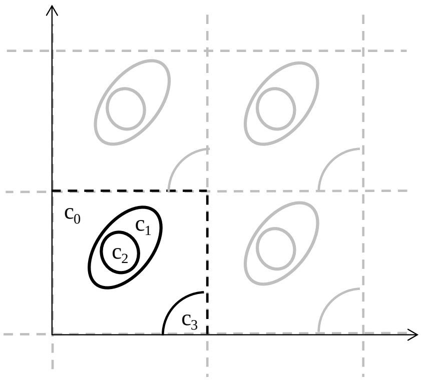

Figure 2: Example of a repeated ellipse template where c_0 corresponds to the background (i.e. grid cell) and c_1-c_3 correspond to ellipses

Each ellipse is stored internally with a matrix representation of a general conic polynomial to support `inside` and `distance` calculations in the template coordinate frame (without having to transform to individual ellipse coordinate frames).

The template stores the template cells in a cell tree data structure to accelerate tracking and point location. A *cell tree* is a limited form of the hierarchical CSG model that was described by Millman, et al.[6] Each cell *c* is represented by a node *n*. Each node has parent *b* and children k_1 through k_m, and stores a bounded region of space $R(n)$. Recall the following relevant properties:

1. $R(n) \subseteq R(b)$
2. $c = R(n) \setminus \bigcup_i R(k_i)$
3. $R(k_i) \cap R(k_j) = \varnothing$ for $i \neq j$

An example of a cell tree corresponding to the model illustrated in Figure 2 is given in Figure 3. In MC21, the cell tree is created during initialization by using a predicate that takes two ellipses, c_1 and c_2, and a unit cell *U* and returns true if the intersection of $R(c_1)$ and *U* is a subset of the intersection of $R(c_2)$ and *U*.

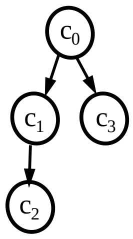

Figure 3: Example of a cell tree corresponding to the example illustrated in Figure 2

Recall that the `inside` routine takes in two arguments: a particle *p* and a template *T*. If *p* is contained in *T*, the routine returns the cell *c* that contains *p*. For the repeated ellipse template, the `inside` routine traverses the cell tree to find cell *c*, such that *p* is in *c* but *p* is not in any of the children of *c*.

The `distance` routine for the repeated ellipse template is broken down into several steps. Again, recall that this routine takes in two arguments: a particle *p* and the cell *c* of template *T* that the particle is inside. First, the distance is computed to the ellipse *c*, each child k_1 – k_m, and the boundary of the template. If the minimal distance is to ellipse *c*, the particle leaves *c* and enters the parent *b*. If the minimal distance is to a child k_i, the particle enters k_i. If the minimal distance is to the boundary of the template, then the particle moves to the next tile. Moving to the next tile is handled by applying a periodic boundary condition to the edges of the tile such that when a particle leaves the left boundary, it enters the right boundary, when it leaves the top boundary, it enters the bottom boundary, etc. Thus, the representation of the ellipses does not need to be transformed as the particle moves between lattice cells.

V. Box Template

The box template is a three-dimensional template used to model a collection of similarly-aligned boxes. The tile, on which the boxes are defined, is infinite in all directions.

An arbitrary number of boxes can be defined within the tile. The bounds of any two boxes cannot intersect, though boxes can be nested within one another and they can share boundaries. Each box can be assigned a non-uniform Cartesian grid consisting of lines parallel to the *x*-, *y*-, and *z*-axes. This grid provides additional geometric detail within the box, without extending past its bounds. An example of this concept can be seen in Figure 4.

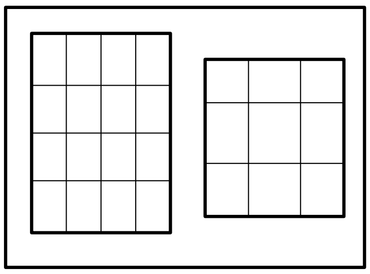

Figure 4: Example of a box template consisting of an outer box containing two disjoint boxes, which each have a different internal grid

Allowing each box to be individually gridded prevents geometric objects from being over-defined (i.e., represented by many more cells than are necessary). For example, if the model in Figure 4 was represented using only grid lines (which extend throughout the component), the equivalent representation would be greatly over-defined as shown in Figure 5. This representation has 100 cells instead of only 26, as shown in Figure 4. This can drastically increase memory usage, as well as lower tracking performance since particles must track to additional grid planes and score tallies in the extra grid cells.

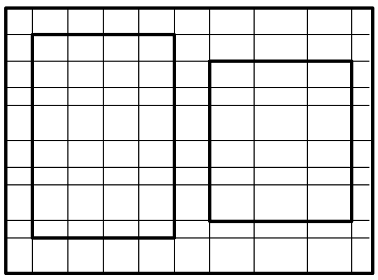

Figure 5: Illustration of what the problem shown in Figure 4 would look like if it were instead defined as a component with a grid.

Similar to the repeated ellipse template, the box template uses a cell tree data structure to represent the arrangement of the boxes.

The *inside* routine for the box template follows the same procedure as the *inside* routine for the repeated ellipse template, since both templates use the cell tree data structure.

The *distance* routine computes the distance to the box *c*, each child $k_1 - k_m$, and the internal grid of *c*. If the minimal distance is to box *c*, then the particle leaves the box. Since boxes can share boundaries, the code then walks up the cell tree until it finds the first ancestor that contains it. Then it walks down the cell tree starting from that ancestor until it finds the lowest node that contains it. If the minimal distance is to child k_i, then the code walks down the tree until it reaches the lowest node that contains it. If the minimal distance is to the internal grid of *c*, then the particle simply moves to the next internal grid cell.

VI. Testing and Results

Test problems were developed to emphasize the benefits of the repeated ellipse template and box template. To specifically focus on the tracking efficiency, all cases were run as a fixed source calculation in vacuum with 100,000 total particles. Particle starting locations were sampled uniformly throughout the model and their starting directions were sampled isotropically. All tests were run on a single Intel Xeon 2.6 GHz processor that had 48 GB of memory.

1. Test Problem - Repeated Ellipse Template

The test problem for the repeated ellipse template consisted of a single square component with an edge length of 10 cm. This component was equally subdivided into $N \times N$ subcells, each with edge length $\delta = 10/N$ cm. Each subcell contained two nested cylinders, representing the cladding and fuel. The diameter of the outer cylinder was $0.8 \cdot \delta$, and the diameter of the inner cylinder was $0.66\overline{6} \cdot \delta$. The number of pins was scaled to show how the tracking speed and memory usage was varied for different representations. Figure 6 shows an illustration of the model for a 20×20 lattice.

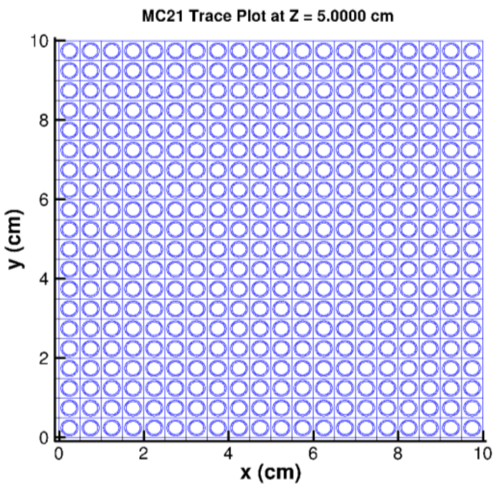

Figure 6: Test problem - 20×20 pin arrangement

Runtime performance was tested using different lattice sizes after constructing the model in three different ways. In the first way, each pin was represented by two components. This was considered the reference case, since it did not use templates at all. In the second way, a Cartesian grid was applied to the problem. Each grid cell was assigned an overlay consisting of two nested ellipses. The repeat capability of the template was not used in this case. In the third way, the box was not gridded. Instead, a single overlay was applied to the box and the template repeat capability was allowed to fill the entire space. Table 1 shows the results of the runtime comparisons for four different lattice sizes: 10×10, 20×20, 30×30, and 40×40. Memory usage was also tested using a very large pin arrangement of 1000×1000. The results of the memory usage test are shown in Table 2.

As can be seen from Table 1, the ellipse template significantly reduced the runtime when compared to a component-only model. This was especially true as the number of pins increased. There was also a noticeable decrease in runtime between the non-repeating template and the repeating template cases. The repeating template case tracked approximately 50% faster than the non-repeating case in these tests.

	Runtime (in seconds)		
Problem Size	Case 1 Component	Case 2 No Repeat	Case 3 Repeat
10×10	3.33	1.03	0.77
20×20	13.83	1.70	1.18
30×30	40.63	2.37	1.57
40×40	92.59	2.99	2.01

Table 1: Runtime for component case, non-repeating template case, and repeating ellipse template case

	Memory Usage (in MB)		
Problem Size	Case 1 Component	Case 2 No Repeat	Case 3 Repeat
1000×1000	3959	71	0.26

Table 2: Memory usage for component case, non-repeating template case, and repeating ellipse template

The runtime results in Table 1 were expected based on the tracking algorithm in each case. In the component-only case, every time a particle left a fuel pin and entered the background material, it had to check all other fuel pins to determine which it would intersect next. Thus, for a particle that intersected *k* pins in an $N \times N$ pin configuration, there were $O\left(k \cdot N^2\right)$ intersection tests performed. The repeating and non-repeating template versions of the problem used the underlying regularity of the problem to avoid extra intersection tests. Since every grid or lattice cell had only one pin to check for intersections (due to the nesting), there were only $O(k)$ intersection tests performed for the same configuration.

The differences in runtime were also expected between the repeating and non-repeating template cases. In the non-repeating case, each grid cell contained a separate overlay. When a particle left a pin and entered the background material, it still had to check distances to the grid cell's boundaries to determine which it would intersect next. This extra event processing did not occur in the repeating template case, which is why it had the shortest runtime. However, the only a single set of properties was assigned to all pins in the model. This means that if the model were to be depleted, then all of the pins would deplete together.

Use of templates also significantly reduced memory usage, as shown in Table 2. In the component-only case, every cylinder was stored as an individual component, which was quite expensive. In the template cases, the geometric structure was only stored once. The non-repeating template case was more expensive than the repeating template case because separate overlays were stored for every pin in the non-repeating case.

2. Test Problem - Box Template

The test problem for the box template consisted of an arrangement of boxes. The outer component contained *N*×*N* cubes, each of which had an edge length of 2.5 cm. Each cube was separated from the other cubes by 1 cm on each side. Cubes near the boundary of the outer component were a distance of 0.5 cm from the boundary. Each of the cubes also had an interior evenly spaced 5×5 grid, where each grid cell had an edge length of 0.1 cm. Figure 7 illustrates how a 4×4 arrangement of boxes would appear.

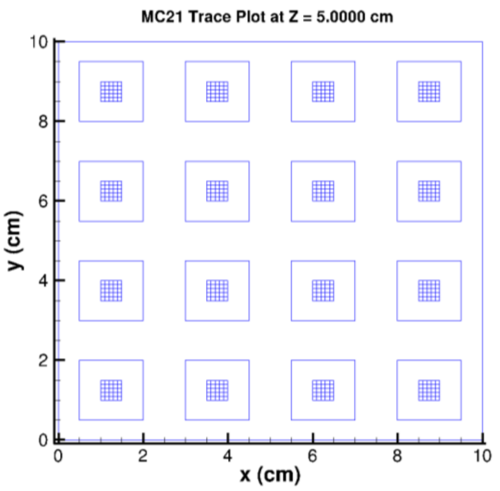

Figure 7: Test problem - 4×4 box arrangement

The number of boxes was scaled to show how the template becomes more efficient as the problem size increases. Three cases were examined for different arrangements of boxes. In the first case, each box was represented as two nested components with a Cartesian grid assigned to the inner component. In the second case, the boxes were not represented using components. Instead, a single grid was applied to the outer component. While this case had the benefit of not using the slower component tracking, it was considerably over-defined, which was expected to increase the memory usage. In the third case, the outer component was gridded such that each box was contained in its own grid cell. The boxes themselves were represented using the box template, and the inner box was gridded. The runtime results are shown in Table 3 and the memory usage results are shown in Table 4.

	Runtime (in seconds)		
Problem Size	Case 1 Component	Case 2 Grid	Case 3 Template
10×10	3.92	0.65	0.55
20×20	16.38	0.82	0.65
30×30	40.04	0.94	0.72
40×40	75.32	1.04	0.77

Table 3: Runtime for component case, grid case, and box template case

	Memory Usage (in MB)		
Problem Size	Case 1 Component	Case 2 Grid	Case 3 Template
10×10	1.11	0.61	0.41
20×20	4.13	2.00	1.01
30×30	9.54	4.23	2.01
40×40	17.72	7.42	3.41

Table 4: Memory usage for component case, grid case, and box template case

The runtime for the component case was similar to the runtime for the component case from the repeated ellipse test (Table 1). This was expected because of arrangement of boxes was similar to the arrangement of cylinders from that test. The memory usage for the component case was considerably more than both the grid and template cases, due to the expense of representing each component and its surfaces. The grid case had a smaller runtime and memory usage, though not to the extent of the template case. The grid case tracked slower than the template case because of the additional event processing that occurred as particles travelled through the extra grid cells. If this test had included other event processing, such as tallies, it is expected that the difference between the grid case and the template case would be even greater. The memory usage in the grid case was also much higher than in the template case because of the properties that were stored for the extra grid cells.

VI. Conclusion

A new geometry framework has been proposed for Monte Carlo codes that can simplify user input, improve tracking speed, and reduce memory usage. While most Monte Carlo codes have ways to address each of these aspects, none of those codes use a general framework based on templates. The templates framework allows for routines to be individually tailored to model simple, commonly repeated geometric shapes. This approach is highly amenable to object oriented programming since each template has its methods associated with it. In this paper, the MC21 implementation of the repeated ellipse template and box template were discussed. Results showed that for a problem with N components, the use of these templates could reduce the runtime from $O(N)$ to $O(1)$. However, the exact performance gains are dependent on how well a code developer can tailor each template to a given design application.

References

1) D.P. Griesheimer, et al., "MC21 v6.0 – A Continuous-Energy Monte Carlo Particle Transport Code with Integrated Reactor Feedback Capabilities," Proc. Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo (SNA+MC), 2013.
2) TRIPOLI-4 Project team, "TRIPOLI-4 Version 7 User Guide," CEA, serma/ltsd/rt/10-4941/a edition (2010).
3) X-5 Monte Carlo Team, "MCNP—A General Monte Carlo N-Particle Transport Code, Version 5," LA-UR-03-1987, Los Alamos National Laboratory (2003).
4) T. J. Donovan, L. J. Tyburski, "Geometric Representations in the Developmental Monte Carlo Transport Code MC21," *Proc. PHYSOR-2006*, Vancouver, BC, Canada, September 10–14, on CD-ROM (2006).
5) I. Lux and L. Koblinger, *Monte Carlo Particle Transport Methods: Neutron and Photon Calculations*, CRC Press 1991.
6) D.L. Millman, D.P. Griesheimer, B.R. Nease, and J. Snoeyink, "Robust Volume Calculations for Constructive Solid Geometry (CSG) Components in Monte Carlo Transport Calculations," Proc. PHYSOR: Advances in Reactor Physics, 2012.