EPJ Web of Conferences **302**, 11006 (2024) https://doi.org/10.1051/epjconf/202430211006
*SNA + MC 2024*

# Add 3D Plotting to Your Monte Carlo Code In A Few Hours

*Gavin* Ridley[1,2,*], *Patrick* Shriwise[3], and *Benoit* Forget[1]

[1]Massachusetts Institute of Technology
[2]Aalo Atomics
[3]Argonne National Laboratory

**Abstract.** Any Monte Carlo transport code can produce 2D slice plots of geometric inputs using sampling of cell or material identifiers at each pixels' point in space; ray-traced 3D plots of simulation geometries are generally less available or require paid or closed-source software. We demonstrate the ease of implementing minimalistic ray-traced visualization algorithms using only the geometric querying operations found in any Monte Carlo transport code.

## 1 Introduction

Monte Carlo (MC) models are pivotal for criticality safety, research reactor core analysis, radiation shielding. The applications of MC methods vary as much as their geometric specifications thereof. Constructive solid geometry in the universe-based [1–3] or scene graph formalism [4, 5] is the standard for geometric representation. Within a scene graph or universe hierarchy, regions of space typically representing a single material are delineated by analytically defined surfaces or meshes. Each code may support its own menagerie of mesh or surface types. In contrast to mesh-based partial differential equation solvers with their variety of standard geometry formats (e.g. VTK) and standard visualization tools (e.g. Paraview [6]), no single visualization tool works for every MC code system, and geometric formats are rarely portable between MC codes. Consequently, the most effective visualization solution thus far is to let each code implement its own. Because 3D visualizations are less trivial compared to conventional slice plots, most codes have not natively adopted this technology. To remediate this, we present the simplest possible consolidation of computer graphics techniques to integrate 3D ray-traced plots into the reader's own MC code with a few hours of focused work.

Integrated 3D visualization can be useful to engineers, not only for analysis but training, communication, and advertising. The report [7] lays out yet more reasons why an integrated 3D visualization tool (which never materialized) would be beneficial for MCNP. Spencer, Kulesza, and Sood point out that engineers often inherit complex MCNP models which are difficult to grasp by reading line after line of surface definitions and poring over slice plots. 3D visualization can bring an analyst up to speed more effectively than conventional 2D slice plots. There are external tools, however, for visualizing MCNP models.

The VisEd tool for MCNP [8] has attracted many users. VisEd can approximately triangulate MCNP geometries for interactive model visualization via OpenGL. In addition, it

*e-mail: gavin@aalo.com

implements ray-traced plots to see geometries "as MCNP sees them." In other words, the actual particle tracking routines through CSG geometry are likely called to determine the color of each pixel. Similarly, the paid software Cyclone [9] presents a meticulously fully featured graphical user interface (GUI) employing ray tracing on MCNP models.

Other codes are not without 3D visualization capabilities. MONK also can produce ray-traced 3D visualizations as mentioned in [10]. Due to TRIPOLI's [11] use of the ROOT framework [5] for geometry visualization, it inherits the robust visualization tools that come along with it. The FLAIR [12] GUI for FLUKA [13] also presents 3D visualization capability, as does the T4G [14, 15] GUI for TRIPOLI-4. SuperMC [16] naturally incorporates 3D visualization by being fundamentally tied to CAD-based geometry; CAD tools practically always include 3D visualization. Lastly, to our knowledge, only one other MC code can produce 3D plots natively: SCALE [17]. Through the Fulcrum [18] user interface, 3D ray-traced plots can be produced to detail both geometry and simulation results.

In OpenMC, voxelization of arbitrary geometries for visualization in any standard mesh-based tool like Paraview [19] has been supported for years. Voxel plots are extremely straightforward to generate, as a simple loop over Cartesian coordinates using the native cell finding routines.

Voxel plots suffer from a few shortcomings, however. Gigabytes of voxels must be generated to resolve detailed geometric features. Consequently, perusing a list of publications employing OpenMC has revealed to us that almost none of them presented their geometry of interest using voxel plots. One aspect of voxel plots' lack of adoption could be the unfamiliarity of users with Paraview; another may be that in routine engineering work, generating a voxel file in OpenMC (which may itself be time-consuming for useful levels of detail) and subsequently opening the file and applying material filters takes more time than is worthwhile to get a glance at the geometry. If it were simple to create an image then iterate on model parameters, it may be more useful. On top of that, for particularly large models, a voxelization may be impossible in a practical amount of time. In problems that mix CSG and meshes available in OpenMC since 2022 and Shift since 2023 [20], ray-traced plots present a natural means to visualize the integration of unstructured mesh with CSG, a capability pointed out as essential by Spencer et al. [7].

We recognize that the methods presented here are pedestrian to any computer graphics expert. To our knowledge, no prior work has concisely presented these algorithms in pseudocode to facilitate adoption into others' work. The objective of the paper is to straightforwardly present visualization algorithms unknown to the typical reactor physicist in terms of the basic geometric operation of an MC code, democratizing high-quality 3D visualization in MC codes.

## 2 Methods

### 2.1 Shading of points on a surface

We have implemented two main 3D visualization types in OpenMC. The first we have named a `ProjectionPlot` in OpenMC. This plot type allows volume rendering of materials and wireframing of material or cell outlines, a familiar view in almost any commercially available CAD software. The second we have deemed a `PhongPlot`, in which a more physically realistic view of a model is rendered. The latter plot type's name comes from the fact that ray tracing is used to approximately shade surfaces using Phong's approximation [21]. Due to constraints on document length, we only present our implementation of the `PhongPlot`, and direct readers to the OpenMC source code for information on the `ProjectionPlot`.

The Phong illumination model approximates the solution to the Boltzmann transport equation for visible light, a glimpse at the connection between reactor physics and computer graphics. The model specifically posits that relative luminous intensity can be approximated as:

$$I_p = k_a i_a + \sum_{m \in \text{lights}} k_d i_m \left(\hat{L}_{p,m} \cdot \hat{N}_p\right) \mathbb{1}(\text{p in line-of-sight of m}) \tag{1}$$

Where $I_p$ is the relative illumination of the point $p$ on an arbitrary surface; $k_a$ is specific to each surface, and represents the amount that the surface reflects ambient light; $i_a$ is the intensity of ambient light. In real life, ambient light originates from diffuse reflections from elsewhere or diffusely scattered light through clouds–any source which is not in line-of-sight to the light source. The summation index $m$ represents the index of a few isotropic light sources which illuminate the scene, and $k_d$ describes the extent that a material reflects singly-diffusely-scattered light; $\hat{L}_{p,m}$ is a normalized vector leading from the point $p$ to the light source $m$, and $\mathbb{1}$ is the indicator function; $\hat{N}_p$ is the normal vector directed off of the surface at point $p$. Some implementations of the Phong illumination model may include a specular reflection approximation term that can emulate metallic or otherwise shiny surfaces, which our work did not implement.

The values in the above equation can be calculated using ray tracing, a standard operation in any MC code. The first term does not depend on any geometric information. The terms in the sum over light sources are calculated using ray tracing. Once a ray has arrived at the point $p$ on a surface, $\hat{L}_{p,m}$ is a simple vector operation, no more than the vector difference between the point the light sits at and $p$ normalized. $\hat{N}_p$ can be calculated using the surface normal subroutines commonly found in MC codes; these are required for implementing white or reflecting boundary conditions. Lastly, the indicator function term must be found by tracing a ray from the point $p$ to the light $m$. If a surface which has been designated as opaque is encountered by the ray, the term is zero. Otherwise, the term is one.

In Serpent and OpenMC, RGB-format colors are typically associated with each cell or material. If the value $I_p$ can be made to be normalized between zero and one, a simple multiplication of each red, green, and blue component describes a realistic color for the point $p$ associated with a pixel. The only additional input over a standard 2D slice plot is a set of material or cell indices which are opaque. Moreover, if tally values are interpolated at the point $p$, this can similarly be made to display results from the calculation. The question, then, is how to find which points $p$ correspond to each pixel in a plot.

### 2.2 Camera Ray Generation

With a method to color each pixel of a plot in place, the question of generating rays which correspond to each pixel in a resulting image remains. To generate such an image, a few inputs are requisite. As in conventional slice plots, a number of pixels in both the horizontal and vertical directions must be specified. Secondly, the field-of-view of the camera must be specified: a number to emulate the human eye is 70°. The vertical field of view can then be calculated as the ratio of vertical to horizontal pixels, thus avoiding distortion of the image. The direction of the view must also be specified. The direction the camera points must also be specified, and in our implementation the direction is specified by listing a point in the model at which the camera should be oriented. This leaves one final degree of freedom: the rotation of the camera about the vector pointing from it to the point of interest. Table 1 names these variables with symbols.

A coordinate system which originates at the camera having basis vectors respectively along the direction of the camera to the point of interest, horizontal pixels, and the vertical

Table 1: Variables to specify a perspective ray-traced plot

| | |
|---|---|
| $n_h$ | number of horizontal pixels |
| $n_v$ | number of vertical pixels |
| $f_h$ | horizontal field of view (radians) |
| $f_v$ | vertical field of view (radians) |
| $\vec{c}$ | location of the camera |
| $\vec{l}$ | point in space to point camera at |
| $\vec{u}$ | direction which is up for the camera |

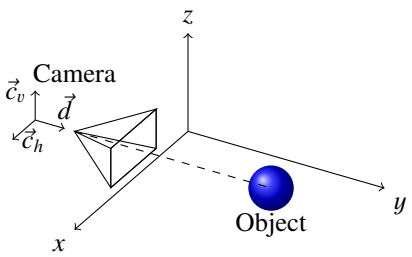



Figure 1: Depiction of camera coordinate system basis vectors of equation 3.

pixels can be defined as:

$$\boldsymbol{M} = \begin{pmatrix} | & | & | \\ \vec{d} & \vec{c}_v & \vec{c}_h \\ | & | & | \end{pmatrix} \tag{2}$$

where

$$\begin{aligned} \vec{d} &= \frac{\vec{l} - \vec{c}}{||\vec{l} - \vec{c}||} \\ \vec{c}_v &= \frac{\vec{l} \times \vec{u}}{||\vec{l} \times \vec{u}||} \\ \vec{c}_h &= \frac{\vec{c}_v \times \vec{d}}{||\vec{c}_v \times \vec{d}||} \end{aligned} \tag{3}$$

Where we have assumed that the horizontal pixel direction and vertical pixel direction in the image are orthogonal. These basis vectors and their relation to the camera location are depicted by figure 1. The matrix $\boldsymbol{M}$ maps directions in camera space to directions in physical space. It is used to convert a pixel coordinate in the image to a direction a ray should traverse the model. A so-called perspective projection image can be obtained by looping over pixels in the camera plane and running rays through the model; this procedure is made explicit by the algorithm in our presentation, or can be seen in the OpenMC source code.

Inside the outer loop over pixels, rays are created corresponding to the view of each individual pixel. The ray carries out a loop over surface intersections throughout the model. Notably, most MC codes will throw an error on cell finding carried out from outside the bounds of the model. Therefore, a specialized function is required to advance rays from

**Algorithm 1** The outer image creation loop that maps pixels to rays.

```
function createImage
    Allocate pixel-wise result array of size n_h × n_v
    focal_dist ← 10                                  ▷ An arbitrary number works here
    dx ← 2(focal_dist) tan(0.5 f_h)                  ▷ Camera-space pixel spacing
    dy ← n_v/n_h dx
    for i_v ← 0;  i_v < n_v;  i_v+ = 1 do            ▷ Loop over vertical pixel coordinate
        for i_h ← 0;  i_h < n_h;  i_h+ = 1 do        ▷ Loop over horizontal pixel coordinate
            local_vec ← (focal_dist, −dx/2 + i_h/n_h, −dy/2 + i_v/n_v)
            local_vec ← local_vec /||local_vec||
            initialize ray with direction matmul(M, local_vec) , position at camera
            if camera not in model then
                Advance ray to model boundary with surface-to-surface tracking
            end if
            while ray intersects next surface do
                OnIntersection                       ▷ Defined in alg. 2
            end while
            color pixel by ray result
        end for
    end for
end function
```

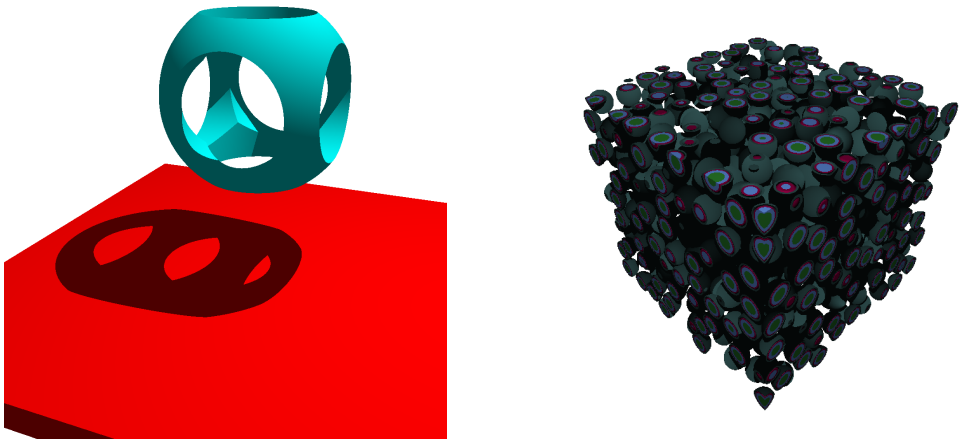
outside the model to the boundary. Because cells are not defined outside the model, surface-to-surface tracking has to be carried out until a valid cell is found in the case of universe-based geometry. A bounding box hierarchy could likely improve the performance of this step, but for a universe-based code, simply nesting the outer universe inside one with a small number of cells solves the problem. This also allows arbitrary clipping of the geometry; the cutting surfaces need only be placed in an outer level, and they can be any surface allowable in the MC code in this case.

## 3 Results

Any valid MC input for OpenMC can be visualized using these methods, so figure 3 shows how aesthetically appealing views of the typical computational solid geometry (CSG) model of a reactor and some TRISO fuel can be created. Similarly, figure 2b highlights how a model automatically converted from an MCNP file format to OpenMC can be visualized to clearly glean more geometric information than a slice plot. The amount of compute time to generate these plots is only marginally more than standard 2D slice plotting.

**Algorithm 2** Handle Intersection in Phong Ray

```
function onIntersection
    Determine hit cell or material ID
    if ray is reflected and beyond the camera then
        Stop the ray
        return
    end if
    if hit ID is not opaque then
        return
    end if
    if not reflected then
        Set ray in direction towards light
        normal ← Calculate the normal vector at the surface
        // Convert surface normal in inner universe to root universe
        for loop over universes from inner to outer do
            normal ← matmul(transpose of rotation matrix of this universe, normal)
        end for
        Calculate I_p from eq. 1
        Set pixel color to RGB value of ID times I_p
        Save the original hit ID of this surface
        Orient the ray towards the camera
        Recalculate distance to next boundary
    else
        Indicator function of eq. 1 is false, set I_p = k_a i_a
        Update pixel color with updated I_p value, using saved original hit ID
        Stop the ray
    end if
end function
```



(a) The typical CSG example object

(b) Cut into some TRISO particles

Figure 2: Any MC input geometry can be visualized with these methods.

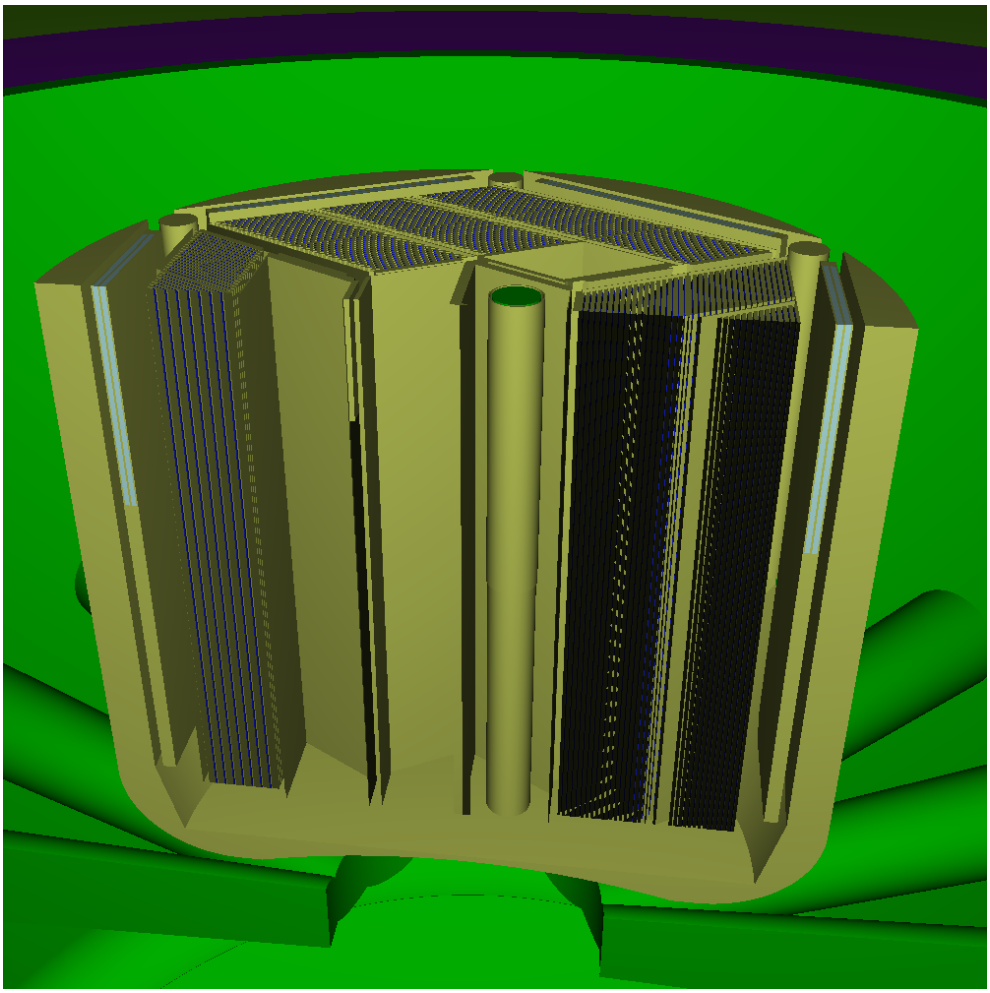


Figure 3: Sliced view of the MIT reactor core created from an MCNP-converted OpenMC model.

## 4 Conclusion

In conclusion, we have presented in compact form the core operations required to create 3D visualizations of MC input geometries. Implementing this took about one thousand lines of code in OpenMC, so we hope that other MC codes will be able to straightforwardly implement these plotting methods. The associated source code can be found in the OpenMC repository in `plot.cpp`, with in-depth discussion in OpenMC pull request #2655, https://github.com/openmc-dev/openmc/pull/2655.

There are a few other directions to expand this work in the future. For one, the coloring methods for the surface are arbitrary, so information far beyond cell or material ID can be visualized. For example, tally values mapped into a color space could be used instead of RGB values for the cell or material. As an *in-situ* power iteration convergence monitor, source site densities tallied on a mesh could be displayed from generation-to-generation. Cell

temperature could also be plotted to monitor the convergence of long-running multiphysics calculations.

Moreover, we envision the potential integration of these visualization methods with virtual reality (VR) technologies, allowing engineers and students to explore complex geometries in an immersive 3D environment. In this case, rays would be traced from two starting locations to form a 3D representation of a Monte Carlo model.

Overall, our hope with this work is to increase the productivity of both users and developers of MC codes by enhancing visual understanding of models. None of the algorithms presented here are new; rather we have presented basic computer graphics techniques shoehorned into the context of MC transport codes. Considering the fairly sparse adoption of native 3D plotting in mainstream MC codes despite the hardest programming aspects already being in place (namely robust ray tracing), we hope others can adopt these methods based on our publicly available implementation in OpenMC.

## References

[1] T. Goorley, M. James, T. Booth, F. Brown, J. Bull, L.J. Cox, J. Durkee, J. Elson, M. Fensin, R.A. Forster et al., Annals of Nuclear Energy **87**, 772 (2016)

[2] J. Leppänen, M. Pusa, T. Viitanen, V. Valtavirta, T. Kaltiaisenaho, Annals of Nuclear Energy **82**, 142 (2015)

[3] P.K. Romano, N.E. Horelik, B.R. Herman, A.G. Nelson, B. Forget, K. Smith, Annals of Nuclear Energy **82**, 90 (2015)

[4] T. Adams, S. Nolen, J. Sweezy, A. Zukaitis, J. Campbell, T. Goorley, S. Greene, R. Aulwes, Annals of Nuclear Energy **82**, 41 (2015)

[5] R. Brun, A. Gheata, M. Gheata, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **502**, 676 (2003)

[6] A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, J. Favre, *Remote Large Data Visualization in the ParaView Framework* (The Eurographics Association 2006), ISBN 978-3-905673-40-1, ISSN: 1727-348X, https://doi.org/10.2312/EGPGV/EGPGV06/163-170

[7] J. Spencer, J.A. Kulesza, A. Sood, Tech. Rep. LA-UR-17-24668, Los Alamos National Laboratory (LANL) (2017), https://sites.nationalacademies.org/cs/groups/dbassesite/documents/webpage/dbasse_179890.pdf

[8] *The Visual Editor for MCNPX*, http://www.mcnpvised.com/visualeditor/visualeditor.html

[9] P. Taylor, *Cyclone*, https://orthrussoftware.com/cyclone/

[10] B. Jones, J. Fildes, S. Richards, J. Sakurai-Hale, Annals of Nuclear Energy **192**, 109943 (2023)

[11] E. Brun, F. Damian, C.M. Diop, E. Dumonteil, F.X. Hugot, C. Jouanne, Y.K. Lee, F. Malvagi, A. Mazzolo, O. Petit et al., Annals of Nuclear Energy **82**, 151 (2015)

[12] V. Vlachoudis, *FLAIR: A POWERFUL BUT USER FRIENDLY GRAPHICAL INTERFACE FOR FLUKA*, in *International Conference on Mathematics, Computational Methods & Reactor Physics 2009* (2009)

[13] G. Battistoni, T. Boehlen, F. Cerutti, P.W. Chin, L.S. Esposito, A. Fassò, A. Ferrari, A. Lechner, A. Empl, A. Mairani et al., Annals of Nuclear Energy **82**, 10 (2015)

[14] F.X. Hugot, Y.K. Lee, Progress in Nuclear Science and Technology **2**, 851 (2011), publisher: Atomic Energy Society of Japan

[15] Y.K. Lee, F.X. Hugot, *TRIPOLI-4 Monte Carlo Code Verification and Validation using T4G Tool*, in *Proceedings of ICONE 31, 31st International Conference on Nuclear Engineering* (2024)

[16] Y. Wu, J. Song, H. Zheng, G. Sun, L. Hao, P. Long, L. Hu, Annals of Nuclear Energy **82**, 161 (2015)

[17] W. Wieselquist, R.A. Lefebvre, Tech. Rep. ORNL/TM-SCALE-6.3.1, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States) (2023), `https://www.osti.gov/biblio/1959594`

[18] R.A. Lefebvre, A.B. Thompson, B.R. Langley, B.T. Rearden, Tech. rep., Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States) (2017), `https://www.osti.gov/biblio/1400159`

[19] in *High Performance Visualization* (Chapman and Hall/CRC, 2012), ISBN 978-0-429-10535-7, num Pages: 18

[20] E. Biondo, G. Davidson, B. Ade, Annals of Nuclear Energy **181**, 109569 (2023)

[21] B.T. Phong, Communications of the ACM **18**, 311 (1975)