Interactive Rendering of MCNP Models

Victor Roetman
Washington Statue University
vroetman@tricity.wsu.edu

Dr. Robert Lewis
Washington Statue University
School of Electrical Engineering and Computer Science
bobl@tricity.wsu.edu

ABSTRACT

In this paper, we describe our approach to interactively render the CSG models used in the MCNP nuclear transport code using OpenGL and standard graphics hardware. The approach uses the algorithm presented by T. F. Wiegand in "Interactive Rendering of CSG Models."

1. INTRODUCTION

It would be useful to have a tool that would allow interactive three-dimensional rendering of MCNP (Monte Carlo N-Particle) [1] models. It is helpful for the designer and reviewers of the calculations to view the input files graphically in three dimensions. This adds further verification that the model is built as intended, and it is also useful for presentations and demonstrations. Interactive viewing, as opposed to static viewing, is good to be able to view the model from different angles, and to see the objects moving and rotating. It gives the viewer a feel of it being a real object, and helps the viewer envision how it looks in 3-space. Furthermore, it's fun to be able to spin models around in real-time, and MCNP developers would want a tool like this even if they didn't need it.

MCNP is a nuclear physics code from Los Alamos National Laboratory that simulates the interaction of neutrons, photons (gamma radiation), and electrons (beta radiation) with various materials in a three-dimensional model. MCNP uses a Constructive Solid Geometry (CSG) format to describe its models.

Visual display of MCNP input files is generally done by looking at two-dimensional cross-sections of the geometry. These images can usually be generated quickly, but they are only two-dimensional. There is also a ray-tracing package called SABRINA [5] that renders low quality three-dimensional images MCNP geometries, but this is not interactive.

We used an implementation of Wiegand's algorithm [6] in this program. This algorithm was designed to render CSG images interactively using standard graphics hardware. We implemented it in OpenGL, and modified it to render MCNP's specific geometrical format.

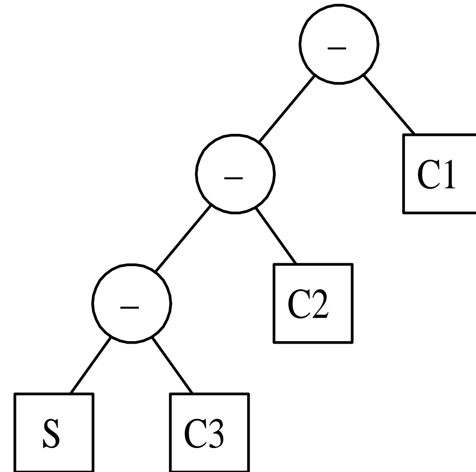

Figure 1: Bowling Ball Tree

2. CONSTRUCTIVE SOLID GEOMETRY

Constructive Solid Geometry (CSG) is a common method in computer graphics for describing three-dimensional models. It consists of applying Boolean operators (union, intersection, and subtraction) on the point sets of geometrical primitives (the space bounded by a sphere, cylinder, half-space, cone, etc). Each primitive is given a description (size, orientation, etc) and a location in space, and is then unioned with, subtracted from, or intersected with other primitives. With a union $A \cup B$, the point sets of objects A and B are combined to form a new point set. A subtraction $A - B$ removes the point set of B from the point set of A. An intersection $A \cap B$ is the point set that is in common between both A and B.

A bowling ball, for example, could be modeled by subtracting three cylinders from a sphere. Generally the CSG model is built into a tree structure made up of nodes. Each node is either a primitive or an operator with children. Each operator is binary and operates on two nodes, or sub-trees. Each sub-tree is itself a CSG tree.

The bowling ball example could be constructed by subtract-

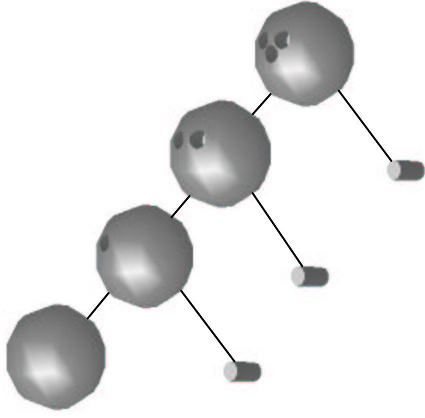

Figure 2: Bowling Ball Figure

ing three cylinders from a sphere. We could write this as $(((S - C3) - C2) - C1)$. If we assume left associativity, we could write this as $S - C3 - C2 - C1$. The CSG tree would look like Figure 2 , and the corresponding objects at each node would look like Figure 2.

3. RAY TRACING

Ray tracing is a method of building images from models by casting a ray (drawing a line) from a viewpoint (the viewer's eye) through a location on the screen (a pixel) and then into the model. What the ray "hits" is what the eye will "see" when it looks at that pixel. This point in the model is found by calculating the intersection of the ray with the object. Once this point is found, the lighting, color, and reflections properties are calculated, by calculating the angle that the ray intersects the surface of the object, and by casting other rays to and from light sources, and casting rays to other objects. A good description of ray tracing can be found in Foley, et al. [2].

3.1 Ray Tracing and CSG

Ray tracing works well with CSG models. The intersections of the ray with the primitives are calculated by solving the ray equation against the primitive equations. This will give multiple values, where the ray enters the object and where it exits the object. (It may also intersect tangentially at one point, or it may not intersect at all.) These values describe ranges of points that are the intersection of the ray with the point set of the object. Then the different point sets from the objects in the tree are compared to determine which one is in "front" by traversing the CSG tree and choosing which points are IN and which are OUT.

Although CSG is a good description model for ray tracing, the intersection calculations of each ray and all the objects in the model must be solved for each pixel in the image. Although there are some optimizations to this process, it is still very time consuming, and is much too slow for interactive graphics.

4. INTERACTIVE GRAPHICS

Most interactive graphics consists of drawing polygons (usually triangles) with lighting and color calculations performed at the vertices of the triangle. The triangle is drawn onto the screen and the triangle is "filled" with colors. When an object is drawn to the screen, the depth value (how far "into" the screen the image is) for each pixel is compared against the depth buffer. If the pixel value is behind the depth buffer value, the pixel is not drawn. If the pixel is in front of the depth buffer value, the pixel color is drawn into the color buffer (drawn onto the screen) and the depth value is entered into the depth buffer.

This method of drawing has a major advantage over ray tracing - it is much simpler to perform these calculations than to solve the intersection equations in ray tracing, and many computers have graphics hardware that do these calculations quickly. This combination can render very complicated models interactively.

There are graphics libraries that do most of these calculations for the programmer, and also take advantage of the hardware. OpenGL [7] is one such library that is commonly used, and is the library we chose to use in this project.

Graphics hardware and libraries have other features that are useful. Of particular use to this project is the stencil buffer. The stencil buffer is a set of integer values for each pixel, often thought of as multiple single bit planes. It is commonly used to mask (stencil) parts of the screen while drawing objects. OpenGL provides functionality to test and set the stencil buffer while drawing.

One problem with CSG models is that the CSG format does not lend itself well to rendering with OpenGL or a similar library. Each primitive can be converted into a set of polygons (tessellated) that approximates the original geometry (similar to approximating a circle with an octagon). However, performing the Boolean operations on the tessellated objects is very difficult, and determining the exact intersections between two arbitrary objects (such as CSG subtrees) is also very difficult. The conversion of the CSG model to a polygonalized model was something we considered. However, we found an easier method.

5. INTERACTIVE RENDERING OF CSG

Wiegand [6] describes a different way to accomplish interactive rendering of CSG models. It is based upon a method [3] to render CSG in a "Pixel Planes" system. Wiegand modified their approach to work with a conventional graphics workstation and its libraries (e.g., an SGI and OpenGL).

The basic approach is to "normalize" the CSG tree into groups that are joined by union. The objects of each group are joined by intersection and subtraction. Drawing each group consists of two operations—a "classification" operation, and a "render" operation. The classification and rendering portion of the algorithm handles the intersections and subtractions within each group, and the depth buffer handles the union of the separate groups.

The classification operation consists of drawing an object into the depth buffer (Z buffer), and then "trimming" away all the other members in the group. The result is a bit in the stencil buffer that is set only in the places where the object is visible in the group.

Table 1: Tree Normalization Rules

$X - (Y \cup Z)$	$\rightarrow$	$(X - Y) - Z$
$X \cap (Y \cup Z)$	$\rightarrow$	$(X \cap Y) \cup (X \cap Z)$
$X - (Y \cap Z)$	$\rightarrow$	$(X - Y) \cup (X - Z)$
$X \cap (Y \cap Z)$	$\rightarrow$	$(X \cap Y) \cap Z$
$X - (Y - Z)$	$\rightarrow$	$(X - Y) \cup (Z \cap Z)$
$X \cap (Y - Z)$	$\rightarrow$	$(X \cap Y) - Z$
$(X - Y) \cap Z$	$\rightarrow$	$(X \cap Z) - Y$
$(X \cup Y) - Z$	$\rightarrow$	$(X - Z) \cup (Y - Z)$
$(X \cup Y) \cap Z$	$\rightarrow$	$(X \cap Z) \cup (Y \cap Z)$

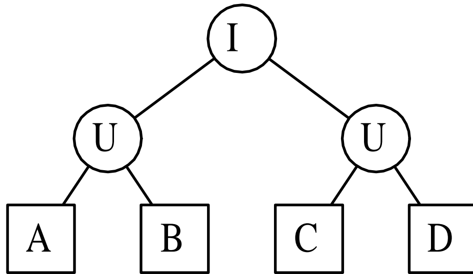

Figure 3: Sample CSG Tree

This is done for each object in the group, with a different bit in the stencil buffer being set for each object. Then the stencil buffer contains a mask for each object in the group, set only where that object is visible.

The render operation then draws each object into the color buffer and depth buffer only where the stencil bit is set. When each object is drawn this way, the final result is the group rendered properly.

This is then repeated for each group, and the depth buffer will ensure the unions are properly rendered.

Graphics workstations only have a limited number of stencil bits (usually eight). If the number of objects in a model exceeds the number of available stencil bits, the process will need to be repeated for more objects. Because the depth buffer is used both in the classification process and in the rendering process, the state of the depth buffer after rendering must be saved before classifying more objects, and restored before rendering them.

5.1 Tree Normalization

The CSG tree is normalized by recursively applying the rules shown in Table 1 until no further rules apply.

This set of rules moves all the unions to the top of the tree, and moves all the intersections and subtractions to the bottom. The final result can be thought of as a set of groups joined by union, where the groups are objects joined by intersection and subtraction.

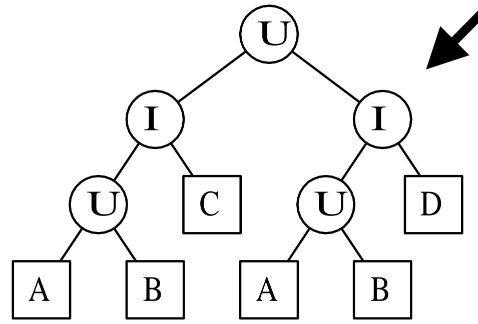

Figure 4: Applied Rule 2

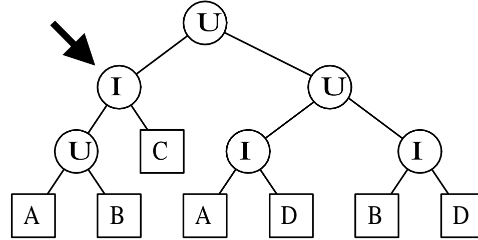

Figure 5: Applied Rule 9

For example, the tree for $((A \cup B) \cap (C \cup D))$ is represented by Figure 3.

After Rule 2 is applied at the top node of the tree, the transformed tree is shown in Figure 4.

Rule 9 is then applied to the right sub-tree (shown with an arrow on Figure 4), and the results are shown in Figure 5.

Rule 9 is applied again to left sub-tree (shown with an arrow on Figure 5), which results in the normalized tree shown in Figure 6.

The normalized tree can increase in size greatly. In the above example, the tree doubled in size after normalizing from four leaf nodes (primitives) to eight leaf nodes. A tree such as $(((A \cup B) \cap (C \cup D)) \cap (E \cup F)) \cap (G \cup H)$ will increase from eight leaf nodes to sixty-four leaf nodes after normalization. Many of the subsequent groups may be pruned if their objects do not intersect. This pruning may take place by comparing the bounding boxes of the objects and subtrees, and nodes with a null bounding box may be deleted.

5.2 Wiegand's Drawing Algorithm

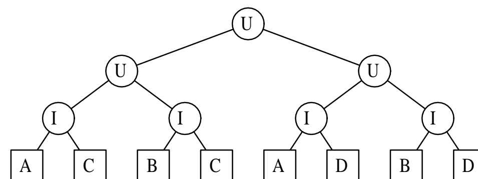

Figure 6: Normalized Tree

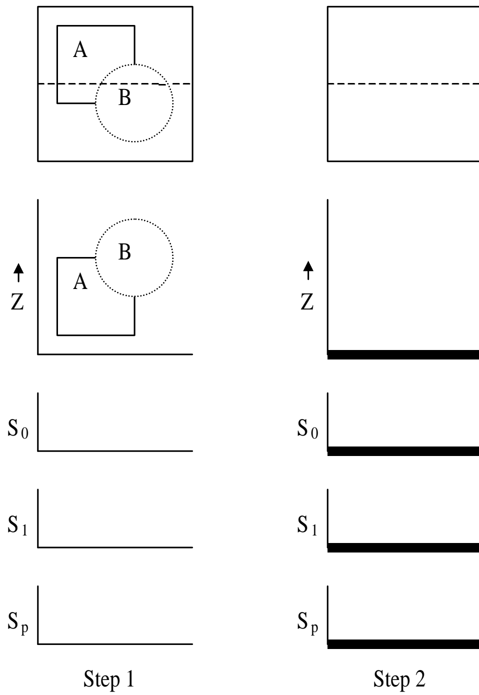

Figure 7: Rendering of A - B

Once the tree has been normalized, and we have a set of groups, we can apply the drawing portion of Wiegand's algorithm: Repeat for each Group: (save depth buffer if this is not the first time through) Classify: set n to zero For each object in the group: Draw object into depth buffer (if object is subtracted, draw inside face, or back face, of object). For all other objects in the group: Trim: Clear stencil plane S_p to 0 Draw object (front and back face) and toggle (S_p) where the pixel passes the depth test. (*This sets S_p where the object intersects with the object in the depth buffer*) If object is subtracted, clear depth buffer where S_p is 1, else clear depth buffer where S_p is 0 Set a stencil bit Sn where depth buffer is set. (*Sn is now set where object n is visible*) clear depth buffer increment n (restore depth buffer if this is not the first time through) set n to zero Render: For each object in group: Draw object into depth and color buffer where Sn is set (if object is negated, draw back face of object) increment n clear stencil buffer

This must be done for each new view of the model, so if the viewpoint is changed (or if the object is rotated or animated), the process must be repeated for the new viewpoint.

5.3 Algorithm Example

This algorithm will be demonstrated for a simple example, showing the state of all the buffers. Steps 1 through 17 are shown in Figure 7 through 7.

Step 1: Suppose we have two objects, a cube (A), and a sphere (B), and we want to render $(A - B)$. The box at the top of the figure shows objects A and B as they would be viewed from the screen. Below the box is a cross-section of the depth buffer (Z buffer) taken at the dashed line on box. The arrow points toward the viewer, so objects at the

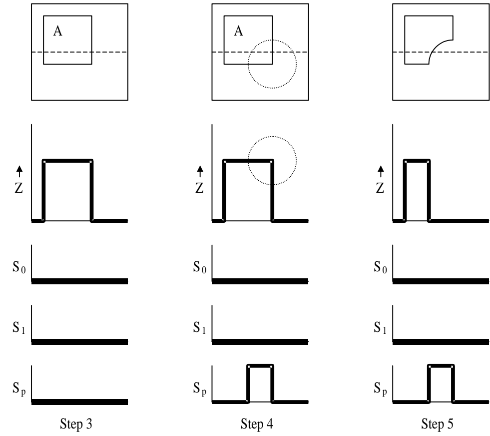

Figure 8: Rendering of A - B (cont.)

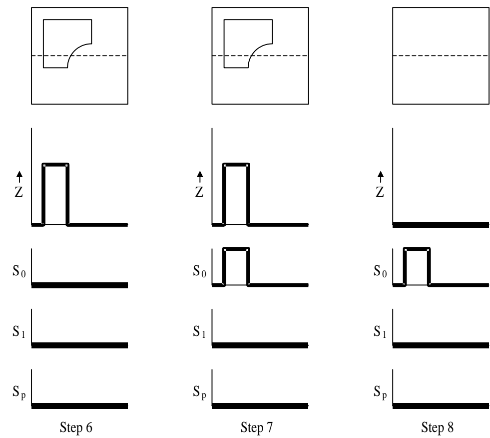

Figure 9: Rendering of A - B (cont.)

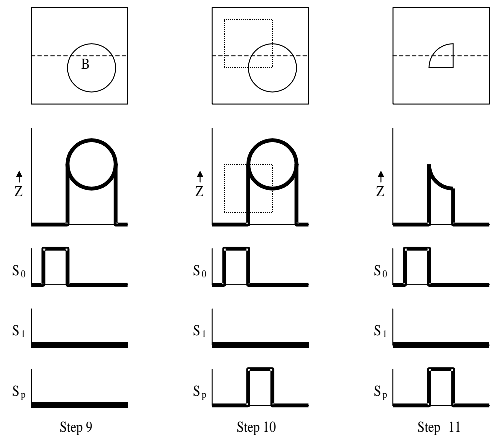

Figure 10: Rendering of A - B (cont.)

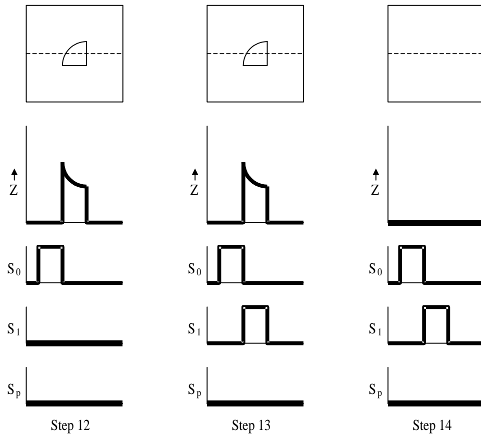

Figure 11: Rendering of A - B (cont.)

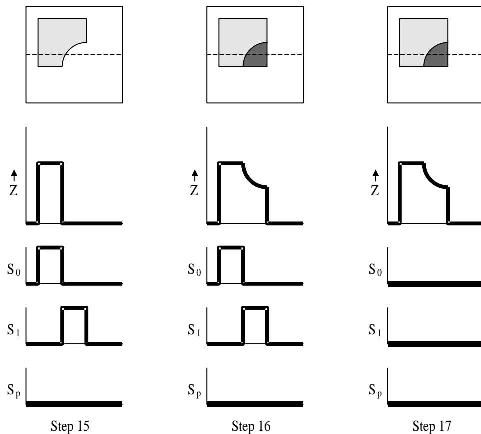

Figure 12: Rendering of A - B (cont.)

bottom are farther away than objects at the top. Sphere B is partially above and beside cube A. S_0, S_1, and S_p are bits in the stencil buffer.

Step 2: This is initial state of the frame buffers at the start of the algorithm. The depth buffer is cleared (set to FAR) and the stencil bits are cleared (set to 0).

Step 3: The classification process begins with cube A drawn into the depth buffer.

Step 4: Sphere B is trimmed against the depth buffer. As the sphere is rendered, S_p, initially 0, is toggled as the object is drawn. Where the front side and back side of the sphere are both drawn, S_p is toggled to 1 and back to 0, but where the back of the sphere is behind the depth buffer (where the shell of the sphere intersects with the cube), it will be drawn only once, setting S_p to 1. Thus S_p is set everywhere that sphere B intersects with cube A.

Step 5: Because sphere B is subtracted, the depth buffer is cleared (set to FAR) everywhere S_p is set.

Step 6: S_p is cleared. Steps 4 through 6 would be repeated if there were more objects to trim.

Step 7: After all the objects have been trimmed (there is only one object to trim in this example), the stencil bit S_0 is set (to 1) everywhere the depth buffer is set (not equal to FAR). Step 8: The depth buffer is cleared (set to FAR). Now S_0 is set everywhere that the cube is visible to the viewer.

Step 9: The classification process is repeated for sphere B. Sphere B is rendered into the depth buffer, but because B is subtracted, we render the inside face of B.

Step 10: Cube A is trimmed against the depth buffer. As before, S_p is toggled as cube A is rendered. Where cube A is behind the object in the depth buffer, S_p will not be toggled. The result is that S_p is set everywhere A intersects B.

Step 11: Because cube A is not subtracted, we clear the depth buffer everywhere S_p is 0.

Step 12: S_p is cleared, and Steps 10 through 12 are repeated if there are more objects to be trimmed.

Step 13: S_1 is set everywhere the depth buffer is set (not equal to FAR).

Step 14: The depth buffer is cleared, and the classification process is repeated for more objects, if there are more to render, up to the number of stencil bits available.

After all the objects have been "classified", each object has a one bit mask in the stencil buffer everywhere that object is visible. We will then render each object where the corresponding stencil bit is set.

Step 15: The render process begins by drawing cube A into the depth buffer and the color buffer (the screen) where S_0 is set.

Step 16: We draw the back face of sphere B (because it is subtracted) into the depth buffer and the color buffer where S_1 is set.

Step 17: Finally, we clear the stencil buffer, and we are finished. If there were more objects to render and there were not enough stencil bits, we would save the contents of the depth buffer at this point, and repeat the algorithm. If this was the case, and the depth buffer had previous information, it would be restored immediately before step 15.

5.4 Viewing Volume Issue

Each primitive must be a "closed" object (all parts must have a front face and a back face) so that as the object is drawn, the parity stencil bit S_p is toggled twice if the object does not intersect anything, leaving the parity bit in the same state as before. However, if the object intersects with the front or back clipping plane, this can cause problems because the object will not be drawn outside the viewing volume and the object is no longer closed. The back clipping plane can be set very far out without much loss of resolution to the depth buffer, solving that problem. But the front clipping plane is still of concern. It is more likely that objects will intersect with the front clipping plane.

Wiegand's solution to this is to toggle the parity bit while drawing the trimming object with no depth testing. This will set the parity bit everywhere the object intersects with the front clipping plane. Then the object is trimmed as normal. The stencil buffer is already set where the object is outside the viewing volume, and the algorithm works as intended. The drawback to this is that each trimming object must be drawn twice, thus slowing down the algorithm.

5.5 Half Spaces

In this algorithm, it is possible to render a single half space as a sufficiently large cube, encompassing the entire model, where one face of the cube is co-planar with the bounding half-space. But Wiegand offers a more efficient (and more elegant) method using clipping planes to render sets of intersecting half spaces.

Convex polyhedra may be represented as intersections of half spaces. Convex polyhedra can also be rendered as a single object in Wiegand's algorithm, provided they are "closed" as discussed earlier. We may then render a set of half spaces as a single convex polyhedron, and "cap" the far end of the potentially infinite object.

The solution uses clipping planes to aid in the drawing of the polyhedron. The number of half spaces that can be rendered as one polyhedron is limited to the number of auxiliary clipping planes in the library. OpenGL systems have at least six auxiliary clipping planes.

The clipping plane, when enabled, will keep any objects that lie on the trimming side of the clipping plane from being rendered. If we set the clipping planes to the same equations as the half-spaces, then we can draw all the planes as very large planes (larger than the viewing volume), and they will only be rendered where they are a face of the polyhedron.

The far end of the polyhedron is drawn by a back plane, filling the screen, and close to the FAR end of the viewing volume. Because of the clipping planes, the back plane is only drawn where it is "in" the polyhedron. The near end of the polyhedron is handled as if it were in front of the front clipping plane, as described earlier.

6. MCNP

The MCNP (Monte Carlo N-Particle) [1] nuclear physics code is written in FORTRAN IV in the 1960's. It is a Monte-Carlo code (using pseudo-random numbers) that simulates neutron, photon, or beta particle interaction in a three-dimensional environment. It is used to calculate radiation dose rate and shielding calculations, as well as nuclear criticality calculations. This is done using an extensive cross-section library of particle interactions with different materials, and the probabilities of such interactions.

A common calculation, for example, could contain a radioactive source in a container made of concrete and lined with lead. The code would calculate the probability of a nuclear decay in the source, calculate the probability of various energy levels and directions the photon particle may take, and then "roll the dice" to see which of these possibilities the particle will take, and assign it a direction and position. Then the code looks up the probability of interaction with the given material, "rolls the dice", and then if there is an interaction, calculates the probabilities for energy deposited and change of direction, "rolls the dice" and then assigns a new position and direction. If the particle leaves one material (such as lead) and enters another (such as concrete), then the code uses the new material properties to determine the interaction. Eventually the particle will be absorbed, will exit the model, or will be terminated for statistical reasons.

The MCNP program uses a CSG model for its model descriptions. This works well for MCNP because the particle calculations are essentially ray-tracing calculations. If a particle has a position and direction, then the distance from the particle to the next object can be calculated by casting a ray from that point in that direction, and solving the intersection equations with each primitive in the CSG tree, running through the Boolean operators, and determining the point of intersection with the next object, and thus the next material. Then the distance can be calculated, and the interaction probability can then be calculated.

6.1 MCNP Models

MCNP primitives are mathematical primitive objects (spheres, cylinders, half-spaces, cones, etc), and many of the MCNP primitives such as half-spaces and cylinders are infinite in extent. A finite cylinder is constructed by intersecting two half spaces with a cylinder. A cube is constructed by the intersection of six half-spaces.

The MCNP user usually thinks of these primitives as "surfaces" or shells that bound a volume, called a "cell". Each cell (a CSG tree) can consist of one material.

When each particle in an MCNP simulation leaves one cell, it must enter another adjacent cell. Adjacent cells will have at least one primitive in common. All space (to infinity) in a valid MCNP input file must be defined by cells, and no

Table 2: DeMorgan's Laws

$$\overline{A \cap B} = \overline{A} \cup \overline{B} \quad (1)$$
$$\overline{A \cup B} = \overline{A} \cap \overline{B} \quad (2)$$

two cells may intersect. This means that at least one cell will be infinite in extent, often surrounding the entire model. These infinite cells will be marked with zero "importance" meaning that all particles that enter the cell are killed, thus limiting the extent of the calculation.

The MCNP input file is built by describing a set of "surfaces" or primitives, and assigning to each one a numerical label. Each cell is described with these surfaces and Boolean operators. The surfaces are complemented with a "-" sign on the surface number. Intersections are implicit (a space), and unions are described with the ":" symbol. Intersections take precedence over unions, parenthesis may be used to group objects, and objects are left associated.

The intersection with a complemented object is generally used for simple subtraction in MCNP. However, MCNP does provide a subtraction operator, called the NOT operator (the "#" sign), which may be applied to geometry groupings and also to other cells. One cell may contain a description NOT another cell. MCNP simply replaces the NOT operator with the cell description of the complemented cell, and then applies DeMorgan's laws (shown in Table 2).

6.2 MCNP Input File

The MCNP input file, sometimes called the "input deck" is constructed as a set of "cards." This terminology comes from the old punch cards used in the original card reader input device. The input file is assembled in the following order:

- Title Card
- Cell Cards
- Blank Line
- Surface Cards
- Blank Line
- Data Cards
- Blank Line

Comment cards start with a "c" in the first five columns. A "$" character signifies a comment for the remainder of the line. Comments may be interspersed through the file.

The cell card must have the following components:

- Numerical label
- A material number (materials are defined elsewhere in the input file)
- A material density if the material is not zero. A negative number notes g/cc, and a positive number notes atoms/cc.
- A CSG cell description
- Optional data cards and comments.

For example, the following cell card

```
10  1  -1.0  6  7  -8  :  2  -3  4
```

is described as: cell number 10, material 1, density of 1.0 g/cc, and a geometrical description of: ((primitive 6 $\cap$ primitive 7) $\cap$ complemented primitive 8) $\cup$ ((primitive 2 $\cap$ complemented primitive 3) $\cap$ primitive 4)

The surface cards must have the following components:

- Numerical label
- Surface type
- Values
- Optional comments and data cards

The data cards contain information about the source, transformations, tallies (how results are computed), and material descriptions.

6.3 Sample Input File

Table 3 shows an example MCNP input file, with italicized line numbers added in front of each card. The description of the input file follows:

Line 1: Title card (required)

Line 2: Cell 1, material 1, 10 g/cc density, the geometry is -1 which means $-surface1$, or everything inside sphere one.

Line 3: Cell 2, material 231 at .00122 g/cc, $geometry - 2 \cap 1$ or everything inside sphere 2 and outside sphere 1

Line 4: Cell 3, material 288, 11.3 g/cc density, geometry $(-3) \cap 4 \cap (-5) \cap 6 \cap (-7) \cap 8 \cap (NOTcell2) \cap (NOTcell1)$ What this means is to take everything in the -z direction from plane 3 intersected with everything in the +z direction from plane 4 intersected with everything in the -y direction from plane 5 intersected with everything in the +y direction from plane 6 intersected with everything in the -z direction from plane 7 intersected with everything in the +z direction from plane 8 and subtract cell 2 and subtract cell 1. This describes a cube centered at 0,0,0 extending 80cm in each direction, but not excluding the two spherical cells inside it.

Line 5: This is a comment.

Line 6: Cell 4, no material (therefore no density is given) and it is $3 \cup -4 \cup 5 \cup -6 \cup 7 \cup -8$, or everything outside the cube described in cell 3.

Table 3: Sample MCNP Input File

```
1   Sample Input File
2       1     1  -10        -1 $ cell 1, 5% enriched fuel
3       2   231  -0.00122   -2 1
4       3   288  -11.3      -3 4 -5 6 -7 8 #2 #1
5   c cell 4 is the outside world
6       4     0  3 : -4 : 5 : -6 : 7 : -8 $ void
7
8       1        so         30
9       2        so         50
10      3        px         80
11      4        px         -80
12      5        py         80
13      6        py         -80
14      7        pz         80
15      8        pz         -80
16
17  mode  n
18  m1    92235.60c        -0.044 $Fuel, 5% enriched
19        92238.60c        -0.836 8016.60c          -0.12
20  m2    92235.60c       -0.0264 $Fuel, 3% enriched
21        92238.60c       -0.8536 8016.60c          -0.12
22  m231  7014.50c         -0.765 $Air
23        8016.50c         -0.235
24  m288  82000.50c            -1 $Lead
25  imp:n  1            2r 0         $ 1, 4
26  sdef
27
```

Line 7: Blank line delineates the end of the cell cards and starts the surface cards.

Line 8: Surface 1, a sphere centered at the origin with a radius of 30 cm.

Line 9: Surface 2, a sphere centered at the origin with a radius of 50 cm.

Line 10: Surface 3, a plane at $x = 80$ cm.

Line 11: Surface 4, a plane at $x = -80$ cm.

Line 12: Surface 5, a plane at $y = 80$ cm.

Line 13: Surface 6, a plane at $y = -80$ cm.

Line 14: Surface 7, a plane at $z = 80$ cm.

Line 15: Surface 8, a plane at $z = -80$ cm.

Line 16: A blank line delineates the end of the surface cards, and the beginning of the data cards. The data cards are not used presently by our program for rendering MCNP models, but the data cards can contain geometry information such as matrix transformations applied to the geometries.

Line 17: Run this calculation in neutron mode.

Lines 18, 19: Material 1 consists of Uranium 235, Uranium 238, and Oxygen (Uranium Oxide)

Lines 20, 21: Material 2 is the same but at a higher 235 enrichment.

Lines 22, 23: Material 231 is air (oxygen and nitrogen)

Line 24: Material 288 is lead.

Line 25: Cell importances (used to improve calculation statistics and results). What this means is an importance of 1, then repeated twice (2r) and then an importance of 0, so 1 will be applied to cell 1, cell 2, cell 3, and then cell 4 will have an importance of 0, which will kill the particles that reach cell 4 (otherwise they would continue for infinity, and the program will either crash when it cannot determine the intersection of a ray with infinity, or will run indefinitely)

Line 26: Source description card. This one uses the defaults (14 MeV neutrons at 0,0,0).

Line 27: Blank line signifies the end of the input file. Everything after this line will be ignored.

Although this particular input file is valid and will run, computationally it will not halt, nor will it give results as no "tallies" or information is requested in the input file. However, for viewing, only a model description is required, and the data file need not be complete.

Table 4 lists the surface types supported in this project.

7. INTERACTIVE RENDERING OF MCNP

Table 4: Supported Surface Types

INPUT LINE	OBJECT	EQUATION
p A B C D	General plane	$Ax + By + Cz + D = 0$
px D	Plane normal to x axis	$x - D = 0$
py D	Plane normal to y axis	$y - D = 0$
pz D	Plane normal to z axis	$z - D = 0$
so R	Sphere at origin	$x^2 + y^2 + z^2 - R^2 = 0$
s X Y Z R	General sphere radius R	$(x - X)^2 + (y - Y)^2 + (z - Z)^2 - R^2 = 0$
sx X R	Sphere on x axis radius R	$(x - X)^2 + y^2 + z^2 - R^2 = 0$
sy Y R	Sphere on y axis radius R	$x^2 + (y - Y)^2 + z^2 - R^2 = 0$
sz Z R	Sphere on z axis radius R	$x^2 + y^2 + (z - Z)^2 - R^2 = 0$
c/x Y Z R	Cylinder parallel to x axis radius R	$(y - Y)^2 + (z - Z)^2 - R^2 = 0$
c/y X Z R	Cylinder parallel to y axis radius R	$(x - X)^2 + (z - Z)^2 - R^2 = 0$
c/z X Y R	Cylinder parallel to z axis radius R	$(x - X)^2 + (y - Y)^2 - R^2 = 0$
cx R	Cylinder on x axis radius R	$y^2 + z^2 - R^2 = 0$
cy R	Cylinder on y axis radius R	$x^2 + z^2 - R^2 = 0$
cz R	Cylinder on z axis radius R	$x^2 + y^2 - R^2 = 0$

The first task was to read in the MCNP input file, and convert the model description into a CSG tree. This was done using YACC and LEX.

The parsed input file is stored as two linked lists: a list of cells and a list of surfaces. Each cell description contains information such as material, density, cell number, whether the cell contains a NOT, and a pointer to the geometry tree.

The geometry tree can contain nodes that are of type INTERSECTION, UNION, GEOMETRY NOT, CELL NOT, and SURFACE. If the node is type INTERSECTION or UNION, it contains left and right pointers that point to the left and right subtrees. If the node is type GEOMETRY NOT, it contains a pointer to the subtree that should be subtracted. If the node is type CELL NOT, it contains the cell number of the cell to be subtracted. And if the node is type SURFACE, it has both the surface id (signed integer), and a pointer to the surface description. It also contains a pointer used in a linked list of objects to be rendered, and a color value to color the object.

The surfaces are stored as a linked list, and contain the surface number, the surface type (plane, sphere, etc), the parameters that describe the surface (radius, position, etc.), a display type (an OpenGL display list is used to draw this object), a 4x4 transformation matrix (to transform the display list to the coordinates described by the surface card), and a special integer value initially set to 0 that is used to check for redundant objects.

The intersection of a complemented object may be rendered exactly like subtraction in Wiegand's algorithm, so the algorithm may be modified to account for the sign of the intersected object.

Because MCNP does not use a "-" operator, since subtractions are handled as the intersection of complemented objects, NOTS (#) are therefore resolved by applying DeMorgan's laws to the subtracted subtree. This is done in two steps:

First, we resolve all the GEOMETRY NOTS by applying DeMorgan's laws to the subtree.

Table 5: Tree Normalization Rules

$$X \cap (Y \cup Z) \rightarrow (X \cap Y) \cup (X \cap Z)$$
$$X \cap (Y \cap Z) \rightarrow (X \cap Y) \cap Z$$
$$(X \cup Y) \cap Z \rightarrow (X \cap Z) \cup (Y \cap Z)$$

Second we resolve all the CELL NOTS using the following algorithm:

Mark each cell that contains a NOT. While some cells still contain a NOT For each cell that contains a NOT: For each NOT in the cell: If the negated cell is resolved (does not contain a NOT) Copy the cell into the tree in place of the NOT Apply DeMorgan's laws If all NOTS are resolved, mark the cell as resolved If no cells were resolved on this pass, then set error condition and exit.

If the algorithm exits with an error after a pass where nothing is resolved, then there is a circular definition (or an illegal definition).

The next step is to normalize the tree. Because MCNP doesn't use a "-" operator, and all the NOTs were eliminated by applying DeMorgan's laws, only three of the rules in Table 1 are needed to normalize a tree. These are shown in Table 5. After applying these rules, some of the normalized groups may contain redundant objects (objects that are in the group more than once: $A \cap A = A$), or may be NULL (objects that are in the group more than once with opposite signs: $A \cap -A = NULL$ and $A \cap NULL = NULL$). The redundant objects and NULL groups are identified and deleted. To search for such groups and nodes, we used the following algorithm:

set special value for each surface to 0 set i to 0 for each group: increment i for each object in the group: if object is negated, ov = -i else ov = i sv = special value for the

surface to which the object points if sv = ov then this node is redundant delete node if sv = -ov then this group is NULL delete group else set sv = ov

This can be accomplished in a single pass through the tree.

Since the half-spaces and cylinders (and other objects) are infinite in MCNP, and the objects must be closed within the rear clipping plane for this algorithm to work, it was determined that each object would be drawn within a bounding sphere. In many cases, a bounding sphere could be calculated for a given group, but for some groups the calculation was difficult, therefore a default bounding sphere is used, and can be changed as an input parameter.

The infinite object problem is also an issue for lighting when drawing cylinders and planes. We could easily draw a plane with four vertices at a far distance away, and it would be clipped to within the viewing volume by the graphics library. However, since the lighting calculations are computed at the vertices, and the vertices would be very far away in this case, all the close shading and specular components on the plane would be lost. Instead, we can draw our planes with several vertices within the bounding sphere so that lighting calculations will take place within the area of interest. This is the same with the cylinders, which are drawn with several (ten) segments inside the bounding sphere, and then the ends are capped with a disc to close the cylinder.

Currently, the program renders the complete input file, and cycles through colors and assigns them to each cell. The infinite cells must be removed as well as air space cells and any other cell that may block the view of the objects of interest. This program will render incomplete and invalid input files, as well as intersecting cells, but every surface referenced in the cell section must be defined.

8. CONCLUSIONS

With current workstation graphics hardware, the algorithm will render relatively simple models in real time. An example that could be rendered in real time on an SGI O2 with an R5000 processor and CRM graphics is a model with about ten cells, and about fifty objects in the tree. More complex models take longer, as do larger window sizes. Relatively large and complex MCNP models take on the order of a minute to render each frame on this machine.

There are three major areas that affect this performance:

First, with some larger and more complex models, the exponential growth of the CSG tree during normalization was a major problem, often causing the program to halt due to memory limits. In order to render the model, such problem cells were eliminated, but some of the cells in the test cases still became very large.

Second, the time complexity of the algorithm is $O(n^2)$ for the number of objects in each group. If the groups are large, the time complexity will severely affect performance.

Third, the time spent saving and restoring the depth buffer is significant. OpenGL does not do this efficiently.

Future work and optimization would be needed to make this a useful product. However, for some simpler input files, the program works quite well. Perhaps future graphics hardware enhancements will allow this algorithm to run in real-time for larger and more complex models.

9. FUTURE WORK

MCNP contains many features that could be implemented in this algorithm (repeated structures, lattices, transformations), and should be implemented in order to be able to render all legal input files. We have thought of approaches to handle most of these features, and how to incorporate them into the parse tree. MCNP also has several primitive types (hyperboloids, one and two sheet cones, torii) which were not implemented at this time. The algorithm is able to handle objects that are not convex (like a torus). However, these additional features were outside of the scope of this project, and therefore we are able to render only a subset of the legal MCNP input files.

The addition of a good user interface would significantly help this program. It would be better for the user to be able to choose cells and materials to render and not render, and to be able to choose color properties to assign to these cells and materials. The user should also be able to toggle whether or not to render with the front clipping plane fix. Quality and correctness may be lost, but speed will increase. Furthermore, the user should be able to decide whether to save the depth buffer (less memory) or save the stencil buffer (probably much better performance with OpenGL).

The program should also be able to discard cells that are too complicated or are giving problems, report an error, and continue. A tool of this sort would be useful in the creation and debugging of invalid geometries.

It is possible to perform bounding box calculations on some parts of the trees, storing "infinite" values for parts of the bounding box that cannot be determined. This may help to eliminate some of the exponential explosion of normalized cells, especially where cell NOTs are used.

MCNP geometries may also contain transformation matrices that may be applied to objects (transformation matrices are described in data cards). MCNP geometries also may contain repeated structures in the form of "universes" (defined objects referenced by a single value) and lattices (three-dimensional square arrays, and hexagonal arrays), but we chose to implement only a subset of MCNP features at this time to limit the scope of the project, so none of these features are supported at this time. Transformations, universes, and lattices should be able to work within the current program structure with some modifications.

10. SAMPLE MODELS AND IMAGES

Figures 10 and 10 are rendered from the following input file. This file has been modified from its original form by commenting out a number of cells, and adding in a "clipping plane" (surface 99) to reveal some detail.

```
cask criticality
c
```

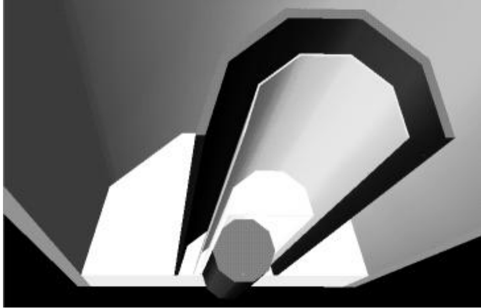

Figure 13: Cask Model

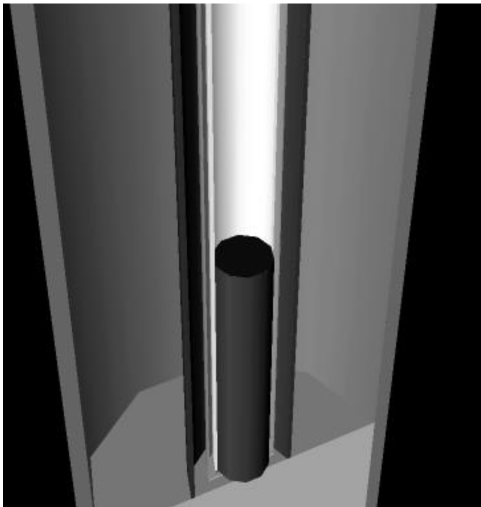

Figure 14: Cask Model

```
00000111111111122222222223333333333444444444455555
c
5678901234567890123456789012345678901234567890123455
c   #1 source region
    1     239 9.7651E-02  9 -17 -1        imp:n=1
c   #3 sfc top cap
    3     304 -8.03       15 -16 -2 99    imp:n=1
c   #4 sfc bottom cap
    4     304 -8.03       -9 10 -2 99     imp:n=1
c   #5 sfc wall
    5     304 -8.03       -2 1 -15 9 99   imp:n=1
c   #10 ident69 wall
   10     304 -8.03       3 -4 -19 11 99  imp:n=1
c   #11 ident69 bottom
   11     304 -8.03       -4 -11 12 99    imp:n=1
c   #12 water outside ident69
   14     304 -8.03       13 5 -6 -19 99  imp:n=1
c   #15 steel bottom of t3
   15     304 -8.03       -13 14 -7 99    imp:n=1
c   #16 outer steel t3 liner
   16     304 -8.03       7 -8 14 -19 99  imp:n=1

    1        cz      5.2240000
    2        cz      5.7150000
    3        cz      6.8800000
    4        cz      7.0650000
    5        cz      10.135000
    6        cz      10.955000
    7        cz      30.515000
    8        cz      33.580000
c   bottom of fissile material
    9        pz      0.0000000
   10        pz     -0.6020000
   11        pz     -1.6770000
   12        pz     -1.9520000
   13        pz     -5.0220000
   14        pz     -40.582000
   15        pz      96.586000
   16        pz      97.188000
c  top of fissile material
   17        pz      50.000000
   18        pz      110.00000
   19        pz      150.00000
   20        pz     -50.000000
   21        pz     -100.00000
   22        cz      45.000000
   23        cz      100.00000
   99        px      0
```

Figures 10 and 10 are rendered from the following input file.

```
Tie Fighter
c fighter  body
1 0 -1 2
c face
2 0 31 -32 33 -34 : -2 32 (-35 : -36 )
c pilot body
 3 0 -61 : -62 -63 64
c 3 0 -61
c 4 0 -62 -63 64
c left connector
11 0 -11 1 22 -10
```

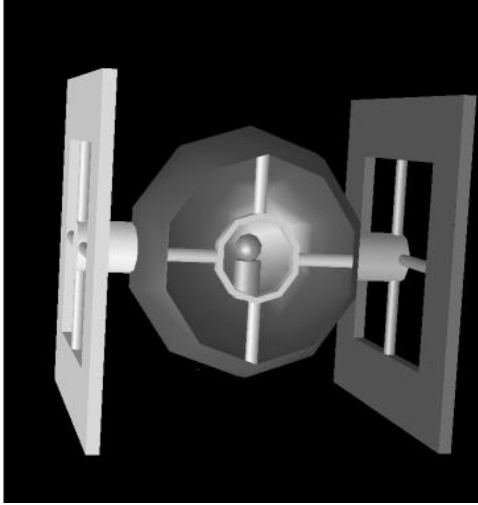

Figure 15: Tie Fighter Model

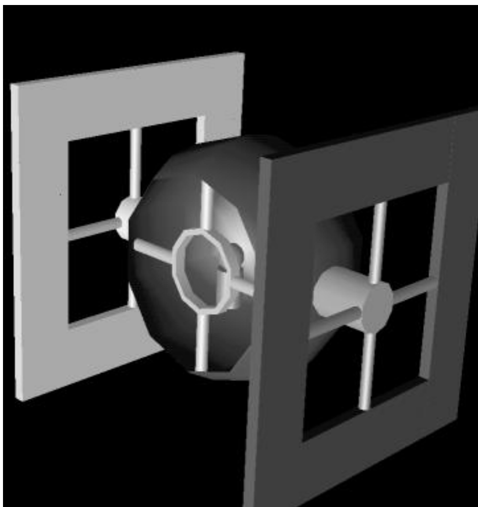

Figure 16: Tie Fighter Model

```
c right connector
12 0 -11 1 -24 10
c left wing
21 0 -21 22 -51 52 -53 54 #(-55 56 -57 58)
c right wing
22 0  23 -24 -51 52 -53 54 #(-55 56 -57 58)
c left wing supports
24 0 (-43 : -44) -55 56 -57 58
c right wing supports
24 0 (-41 : -42) -55 56 -57 58

1  so  10 $ body
2  sz   2 9 $ hollow in body
10 px   0 $ center plane
11 cx   2 $ connector
21 px -13 $ left wing inside
22 px -14 $ left wing outside
23 px  13 $ right wing inside
24 px  14 $ right wing outside31 cz   3 $ face ring
32 cz   3.5 $ face ring
33 pz   4 $ face ring
34 pz   5 $ face ring
35 c/x  0 4.5 .5 $ face bars
36 c/y  0 4.5 .5 $ face bars
41 c/y  13.5 0 .48
42 c/z  13.5 0 .48
43 c/y  -13.5 0 .48
44 c/z  -13.5 0 .48
51 pz  13 $ wing front
52 pz -13 $ wing back
53 py  13 $ wing top
54 py -13 $ wing bottom
55 pz  8 $ wing front hole
56 pz -8 $ wing back hole
57 py  8 $ wing top hole
58 py -8 $ wing bottom hole
61 s 0 .5 1.5  1.1 $ pilot head
62 c/y 0 1.5 1.1 $ pilot body
63 py  -.5  $ pilot body top
64 py  -3.5  $ pilot body bottom
```

11. REFERENCES

[1] Breismeister, J. F, Editor, 1993, MCNP–A General Monte Carlo N-Particle Transport Code, Version 4a, LA-12625, Los Alamos National Laboratory, Los Alamos, New Mexico, November 1993.

[2] J. Foley, A. van Dam, S. Feiner, J. Hughes, Computer Graphics: Principles and Practice, Second Edition in C, Addison-Wesley, Reading, MA, 1995

[3] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs, "Near real-time CSG rendering using tree normalization and geometric pruning," IEEE Computer Graphics and Applications, 9(3), pp. 20-28 (1989).

[4] Tom McReynolds, Organizer. SGI Programming with OpenGL: Advanced Rendering. Presentation notes from SIGGRAPH '97 Course, 1997.

[5] Van Riper, K. A., SABRINA User's Guide, LA-UR-93-3696, Los Alamos National Laboratory, Los Alamos, New Mexico, 1993.

[6] T. F. Wiegand. Interactive rendering of CSG models. In Computer Graphics Forum, volume 15, pages 249-261, 1996.

[7] M. Woo, J. Neider, T. Davis, OpenGL programming guide: the official guide to learning OpenGL, version 1.1, 2nd ed., Addison-Wesley Developers Press, Reading, MA, 1996.