



Visualizing Monte Carlo computed radiation energy values by using a density emitter
by Brett A Donahue

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in
Computer Science
Montana State University
© Copyright by Brett A Donahue (1994)

Abstract:

The goal of this thesis was to produce a program that will allow the visualization of the data produced by a program that uses the Monte Carlo method to simulate a neutron beam colliding with an environment. The Monte Carlo program referenced in this thesis approximates the points where neutrons lose energy as they pass through matter contained within a cube. It sums the energy lost in cubical regions (referred to as voxels) by millions of neutrons. Finally, these energy values are output as a three dimensional array. The problem is to produce a program allowing visualization of this enormous amount of three dimensional energy data. The viewers need to be able to see where the dense areas are in respect to the cube and in respect to other three dimensional information. The viewers need to be able to see through the data, so that they can see information deep within the cube.

For this project, the cube is treated as a "density emitter" which treats each voxel as if it were filled with tiny light emitting particles termed light emitters. A line integral is performed along the line from the viewer's eye, through the front side of the cube, to the point where this line exits the cube on the far side. The integral sums the light emitted along this path. The equations employ a Poisson distribution to take into account the probability of light from a particular point not arriving at the viewer's eye due to it having been obstructed by other particles.

The images output look much like a cloud. The cloud is dense where the energy values were high. The regions deep within the cube are still visible, but are hazy due to the cloud-like information in front of them. By moving the viewer's eye around the cube in a circular orbit, the program produces several images which are combined to produce an animation. The viewer can see the cube spinning which brings alive the depth of the image and the data.

VISUALIZING MONTE CARLO COMPUTED RADIATION ENERGY
VALUES BY USING A DENSITY EMITTER

by

Brett A. Donahue

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

June 1994

N378
D717

ii

APPROVAL

of a thesis submitted by

Brett A. Donahue

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

7/1/94
Date

J. Dumbig Stanley
Chairperson, Graduate Committee

Approved for the Major Department

7/1/94
Date

J. Dumbig Stanley
Head, Major Department

Approved for the College of Graduate Studies

7/1/94
Date

R. L. Brown
Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purpose, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature

Brett A. Donahue

Date

7-1-94

TABLE OF CONTENTS

	Page
CHAPTER 1	
INTRODUCTION	1
CHAPTER 2	
THE VARYING DENSITY EMITTER	3
CHAPTER 3	
GDER PROGRAM DESIGN	5
Program Overview	6
Class Instantiations	7
The MATRIX, POINT, and VECTOR Classes	8
The DE Class	10
The BNCT Class	13
The IMAGE Class	15
The YBUFFER Class	16
The VOXEL Class	20
The VIEWER Class	25
Main Function	27
CHAPTER 4	
RESULTS	29
GDER Usage	29
Future Work	30
Summary	31
REFERENCES	33
REFERENCES CITED	34
APPENDICES	35
APPENDIX A	
PARAMETER FILE FORMAT AND CONTENTS	36
APPENDIX B	
VOXEL FILE FORMAT	38

LIST OF TABLES

Table	Page
1. The MATRIX class.	9
2. The POINT class.	9
3. The VECTOR class.	10
4. The DE class.	11
5. The BNCT class.	14
6. The IMAGE class.	15
7. The YBUFFER class.	17
8. The VOXEL class.	21
9. The VIEWER class.	26

LIST OF FIGURES

Figure	Page
1. Class instantiations.	7
2. A Sliver projected onto a viewing plane pixel.	18
3. Scan lines across the projected edges.	19
4. The voxel cube.	22
5. Voxel cube traversal.	24
6. Output Example 1.	32
7. Output Example 2.	32
8. Example parameter file.	37
9. Example voxel data file.	38

ABSTRACT

The goal of this thesis was to produce a program that will allow the visualization of the data produced by a program that uses the Monte Carlo method to simulate a neutron beam colliding with an environment. The Monte Carlo program referenced in this thesis approximates the points where neutrons lose energy as they pass through matter contained within a cube. It sums the energy lost in cubical regions (referred to as voxels) by millions of neutrons. Finally, these energy values are output as a three dimensional array. The problem is to produce a program allowing visualization of this enormous amount of three dimensional energy data. The viewers need to be able to see where the dense areas are in respect to the cube and in respect to other three dimensional information. The viewers need to be able to see through the data, so that they can see information deep within the cube.

For this project, the cube is treated as a "density emitter" which treats each voxel as if it were filled with tiny light emitting particles termed light emitters. A line integral is performed along the line from the viewer's eye, through the front side of the cube, to the point where this line exits the cube on the far side. The integral sums the light emitted along this path. The equations employ a Poisson distribution to take into account the probability of light from a particular point not arriving at the viewer's eye due to it having been obstructed by other particles.

The images output look much like a cloud. The cloud is dense where the energy values were high. The regions deep within the cube are still visible, but are hazy due to the cloud-like information in front of them. By moving the viewer's eye around the cube in a circular orbit, the program produces several images which are combined to produce an animation. The viewer can see the cube spinning which brings alive the depth of the image and the data.

GLOSSARY OF TERMS AND ACRONYMS

BNCT	Boron Neutron Capture Therapy, the program sponsored by the Department of Energy through INEL for which GDER was written
BNCT	the class in GDER that handles the BNCT B-spline surfaces
DE	the class in GDER that performs the density emitter functionality [SABE88]
GDER	General Density Emitter Renderer, the name of the program discussed in this thesis
INEL	Idaho National Engineering Laboratories
MRI	magnetic resonance images
PRP	projection reference point [FOLE94]
sliver	the entire section of a voxel that is projected onto one pixel of the image. (See Figure 2, Page 18.)
VIEWER	the class in GDER that does viewing functions
voxel	volume element, similar to a pixel, but in three space
VOXEL	the class in GDER that handles voxels
VRC	viewing-reference coordinates [FOLE94]
VUP	the vector that defines the up direction [FOLE94]
ybuffer	my algorithm to project 3D voxels onto the viewing plane
YBUFFER	the class in GDER that implements the ybuffer algorithm

CHAPTER 1

INTRODUCTION

This thesis represents the development of a computer graphics program titled General Density Emitter Renderer (GDER) that will aid in the visualization of the data produced by programs used in the Boron Neutron Capture Therapy (BNCT) radiation treatment planning at Montana State University (MSU) and the Idaho National Engineering Laboratory (INEL) in Idaho Falls, Idaho. The BNCT radiation treatment planning program [WESS92] takes digitized medical images as input and produces three dimensional (3D) B-Spline surfaces defining various anatomical regions such as the skull, brain, tumors, eyes, and sinus cavity. These region definitions are then used as input to the Monte Carlo program at INEL [WHEE94]. Monte Carlo uses stochastic methods based upon physical properties to simulate hundreds of millions of neutrons interacting with the matter in the treatment environment. The cubical volume where these neutrons are traced is segmented into equal sized, small cubical regions or voxels (volume elements). The energies lost by the neutrons within each of these voxels are totaled and output to a file.

A program is needed to combine the data from the BNCT.

program and the Monte Carlo program into one comprehensive image. The program needs to produce an image that will allow viewers to visualize the enormous amount of energy data with respect to itself, with respect to the BNCT B-spline surfaces, and with respect to the entire cubical volume considered by Monte Carlo. Also, the program should be able to produce a fly around animation of the cube to enhance the viewer's perception.

The major problem to overcome is how to perform volume rendering upon the cube of energy values, so that the viewers are able to see the "hot spots" where large amounts of energy were deposited by the neutrons, no matter how deep within the cube they reside. To do the rendering, the cube will be treated as a "varying density emitter", and the algorithm presented in the paper "A Rendering Algorithm for Visualizing 3D Scalar Fields" by Paolo Sabella [SABE88] will be followed with some improvements that will be discussed in chapter 2.

CHAPTER 2

THE VARYING DENSITY EMITTER

A density emitter is a volume filled with tiny light emitting particles termed light emitters. The density of these particles is defined by a 3D scalar field. In this application, we have discrete 3D scalar data defined by the voxels representing the cubical regions of energy, so that the density of the light emitters within a voxel is equal to the value stored in the cubic array. The voxels are equal sided and there are an equal number of voxels in each direction, so the entire volume has the shape of a cube.

To render the density emitter cube, a line integral is performed along a straight line from the viewer's eye, through a pixel on the image, through the front surface of the density emitter cube, and to the back surface of the cube. This integral sums the tiny amounts of light emitted by the light emitters. A Poisson distribution in the integrated equation accounts for the probability of light not arriving at the viewer's eye due to it colliding with other light emitters along the path. A summation is performed rather than analytic integration because of the discrete scalar field.

The algorithm to perform the volume rendering by Paolo Sabella [SABE88] is followed closely with one major deviation. Because the light emitters must be considered from the closest to the furthest, Sabella used a ray tracing approach to perform the summation that traced the path of the line integral for each pixel. In GDER, a different approach is used. A front to back traversal of voxels is performed. The traversal order must ensure that for each voxel being processed, every voxel that lies in front of it has previously been processed. A specialized scan line algorithm with a y-buffer [FOLE94] projects light emitted from within one voxel onto the image. This method was written to handle the projection of a volume instead of a polygon, so the depth is also taken into account. With my algorithm, each voxel needs to be considered only once per image, whereas the ray tracing approach considers every voxel that affects a particular pixel for each pixel in the image.

CHAPTER 3

GDER PROGRAM DESIGN

Because GDER is written in C++, object oriented terminology will be used to describe the program design and implementation. The functionality is contained within nine classes:

- MATRIX,
- POINT,
- VECTOR,
- DE,
- BNCT,
- IMAGE,
- YBUFFER,
- VOXEL, and
- VIEWER.

Each of these classes is discussed in detail in the following sections. The most basic classes will be discussed first because instantiations of the basic classes will be members of the more intricate classes.

Program Overview

The steps that GDER takes to produce the images are:

A) initialization which includes:

- 1) input of the parameter file,
- 2) input of the voxel data,
- 3) input of the BNCT B-splines surface data,
- 4) initialize the DE class member, and
- 5) initialize the IMAGE class members;

B) rendering the scene which involves:

1) rendering the density emitter by

- a) traversing the voxels from the nearest to the furthest,
- b) using a specialized scan line algorithm and y-buffer to provide each voxel's contents to the density emitter summation equations, and
- c) copying the density emitter's stored intensity values into the image;

2) rendering the BNCT B-spline surfaces into the image;

C) saving the image to disk; and

D) changing the viewing angle and looping to step B.

Class Instantiations

Figure 1 shows which classes use instantiations of other classes. Showing how these classes use instantiations of each other should help show the program flow. The function MAIN is the routine in the program that uses only the VIEWER class which uses all of the other classes.

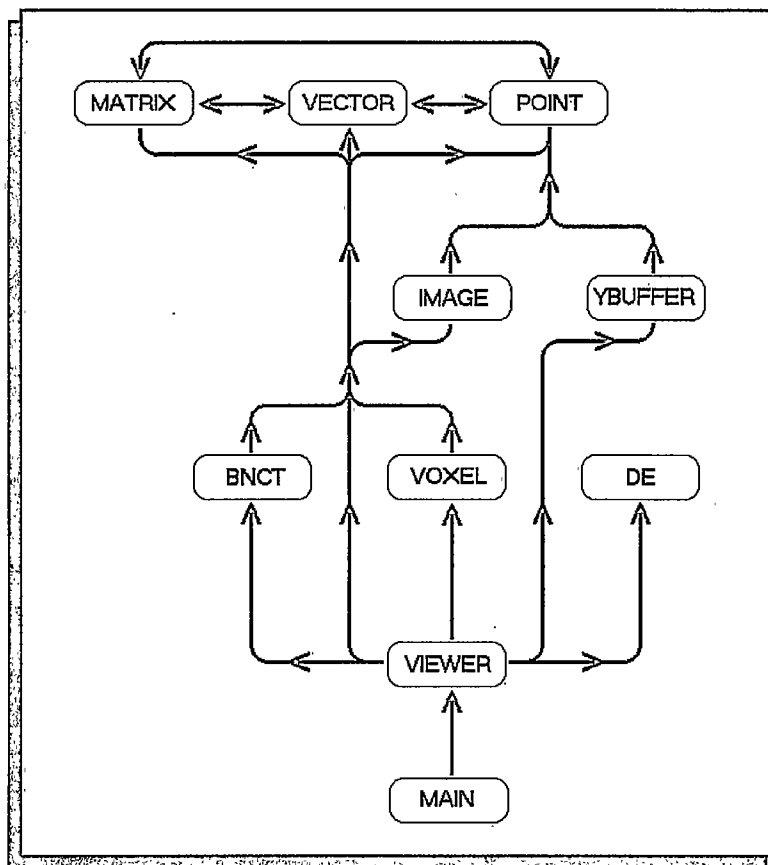


Figure 1. Class instantiations.

The MATRIX, POINT, and VECTOR Classes

The MATRIX, POINT, and VECTOR classes handle the storage and manipulation for matrices, points, and vectors. The methods perform the vector math associated with these types. All of the members of these classes are private, but some of the members are friends of the other classes. This allows vector math to be performed across classes. For example, a matrix multiplication may be performed upon a point, or a vector may be calculated by the location of two points.

The contents of the MATRIX, POINT, and VECTOR classes with brief definitions are in Tables 1, 2, and 3 respectfully.

Table 1. The MATRIX class.

CLASS MEMBER	PURPOSE
M	a 4x4 array representing a matrix
CLASS METHOD	PURPOSE
MATRIX	initializes the matrix by calling ZeroMatrix
ZeroMatrix	zeros this matrix
Mult	multiplies this by the given MATRIX
MatMultPoint	multiplies and stores the given MATRIX by the given POINT
GetTransMatrix	creates and stores a translate matrix from the 3 floats given
GetRotateMatrix	creates and stores a rotate matrix from the 3 floats supplied
GetScaleMatrix	creates and stores a scale matrix from the 3 floats supplied

Table 2. The POINT class.

CLASS MEMBER	PURPOSE
x, y, z	the values representing a point
CLASS METHOD	PURPOSE
POINT	initializes this POINT to a given value
Set	assigns a value to this point
X	returns the value stored as the x member
Y	returns the value stored as the y member
Z	returns the value stored as the z member
MatMultPoint	multiplies the given POINT and the given MATRIX and stores the value
CalcDistance	returns the distance between two POINTS

Table 3. The VECTOR class.

CLASS MEMBER	PURPOSE
x, y, z	the values representing a vector
CLASS METHOD	PURPOSE
VECTOR	initializes this VECTOR to a given value
Set	assigns a value to this vector
X	returns the value stored as the x member
Y	returns the value stored as the y member
Z	returns the value stored as the z member
Normalize	divides this vector by the magnitude to make it unit length
CrossProd	calculates the cross product of this vector and one other provided VECTOR
DotProduct	calculates the dot product of this vector and one other provided VECTOR
Magnitude	returns the magnitude of this vector
Vectorize	calculates and stores the vector defined by the two POINTS provided

The DE Class

The DE (density emitter) class is responsible for performing all of the calculations related to the density emitter. The methods and members are listed in Table 4.

Table 4. The DE class.

CLASS MEMBER	PURPOSE
OverIntensity	the number of pixels that have an intensity more than the maximum value of 1.0
struct DEDataType	the structure that contains values stored for each pixel, which are: TransM - the pixel's transmittance Intensity - the pixel's intensity MaxDensity - the maximum density ever applied to this pixel DepthOfMaxDen - the depth at which MaxDensity occurred
DEData	the array of DEDataType that stores the values for each pixel
Tau	part of the density emitter equation, logarithmically adjusts the density values [SABE88]
DensityTran	part of the density emitter equation, determines how opaque the voxels are [SABE88]
Color	a boolean indicating of pseudo color is calculated
MaxMaxDensity	the highest density recorded for all pixels
MinDepth, MaxDepth	the max and min depths, used for calculated the saturation (HSV) [SABE88]
CLASS METHOD	PURPOSE
DE	initializes all members by calling clear
Clear	initializes all members, must be called prior to producing each image
Apply	applies a density of light emitters to a pixel
GetRGB	calculates and return the color of a pixel
SetDepthRange	stores the max and min depth of the voxel cube
OverInt	returns the value stored in OverIntensity

Initially, the transmittance values are set to 1.0 indicating that all of the light from the particle emitter reaches the viewing plane. The intensity is set to 0.0, so it contains a blank image.

The method `Apply` performs the summation equation that calculates the intensity of each pixel. This method has five parameters:

- `i, j`--the viewing plane coordinates of the pixel,
- `length`--the length of the voxel "sliver" (see Figure 2, page 18) that is projecting upon this pixel,
- `density`--the density of the particle emitters in the "sliver", and
- `depth`--the depth of the "sliver" in world coordinates.

The intensity of light reaching the viewing plane is calculated, summed, and stored within members of this class. The method `Apply` is invoked once for each pixel projected upon by each voxel. The successive calls of `Apply` for one pixel must have increasing depth values, or the equations will not work properly. The methods `VIEWER::Render`, `VIEWER::ViewVoxel`, and `VOXEL::GetVoxelTraversalData` perform this front to back traversal of voxels. The class `YBUFFER` handles the projection of the voxels onto the appropriate pixels of the viewing plane.

The BNCT Class

The BNCT class stores and manipulates the BNCT data. It inputs the data from a file; translates, rotates, and scales the data; and renders the data. The methods and members are listed in Table 5.

The BNCT data and the voxel data each have their own coordinate system. The rotate and scale are constant values, but the shift values are dependent upon the data set. When the voxel data is read from the file, its location in world coordinates is determined and the shift values are sent to the BNCT class via the method SetShift. When the method TRS is called, the necessary translates, rotates, and scales are performed on the BNCT data, so that it has the same coordinate system as the voxel data.

The method Render renders the BNCT data and stores the results in an instantiation of the IMAGE class. Rendering of the BNCT data consists of drawing a line from each point to each adjacent point; thus, drawing the meshes defining the BNCT bodies. The color of the lines for each body are read in as part of the BNCT data. The lines are depth cued, but that is performed within the method IMAGE::DrawLine of the IMAGE class. By replacing the Render method, the user may employ

other rendering algorithms. Included within the source is another, unused Render method which produces a file of "raw" triangle data. The program RAWTOPOV can convert the file into an input file for The Persistence Of Vision Ray Tracer (POV-Ray), so the BNCT data can be ray traced. Both RAWTOPOV and POV-Ray can be acquired via ftp at wuarchive.wustl.edu on internet.

Table 5. The BNCT class.

CLASS MEMBER	PURPOSE
shift_x, shift_y, shift_z	the values that translate the BNCT data to the voxel data
BNCTData	the "root" of linked list that contains the BNCT data
CLASS METHOD	PURPOSE
GetEmptyBNCTBody	returns an initialized linked list node
BNCT	initializes the class members
BNCTRead	reads in the BNCT data from a file and builds the data structure
SetShift	sets the three translate values
TRS	TRS (translates, rotates, and scales) the BNCT data
Free	frees the allocated memory in the BNCT data structure (not implemented)
Render	renders the BNCT image and stores the pixel values in the image class provided

The IMAGE Class

The IMAGE class provides for storage and manipulation of an image. Member functions plot points and draw depth cued lines into the image. The IMAGE class members and methods are tabulated in Table 6.

Table 6. The IMAGE class.

CLASS MEMBER	PURPOSE
ImageDataType	a structure defining a pixel in the image
image	a 2D array for storing the image
MaxDepth, MinDepth	the minimum and maximum depths which are used in depth cuing
CLASS METHOD	PURPOSE
IMAGE	initializes and clears the image
Set	sets one pixel's data (color)
Get	returns one pixel's data (color)
SaveTGA	saves the image as a Targa file
Clear	clears the image for future processing
SetDepthRange	sets the maximum and minimum depths which are used in depth cuing
DrawPixel	draws one pixel into the image
DrawLine	draws one depth cued line into the image

The image must be cleared by calling the method `Clear` prior to any processing. Then, the method `SetDepthRange` must be invoked to set the maximum and minimum possible depth values, so that depth cuing of lines may be performed. The Basic Incremental Algorithm [FOLE94] is used for line drawing. Because very little time is spent drawing lines compared to computing the density emitter equations, the time was not taken to implement a faster, more efficient algorithm.

The method `SaveTGA` saves the image as a Targa file without RLE compression. Within this method, there is unused source code that saves a PPM (Portable Pix Map) which may be more compatible with UNIX image utilities.

The YBUFFER Class

The algorithms used in the YBUFFER class were derived from the scan line algorithm for drawing filled polygons that uses a y-buffer [FOLE94]. Many modifications were necessary, so that depth is taken into account when voxels are projected onto the viewing plane. The YBUFFER class contents with a description is in Table 7.

Table 7. The YBUFFER class.

CLASS MEMBER	PURPOSE
const EMPTY	indicate that a y-buffer index is empty
struct { x[5] z[5] } YBuff[]	stores all y-buffer info. x array stores the x locations of the lines z array stores the depths of the lines
yIndex	stores the next index in YBuff to be drawn
yBuffSorted	indicates that the x arrays are sorted
CLASS MEMBER	PURPOSE
SetLine	puts a line into the y-buffer
GetNextRange	returns an x range and depths to work on and a boolean indicating another range existed
SortYBuff	sorts each of the x arrays in the YBuff
YBUFFER	constructor, initializes all of the data
Set	receives 8 points in usual order (A,B,...,G) and enters the cube into the y-buffer
GetNextPoint	returns the next pixel data to be drawn returns a boolean to indicate a range was available

The YBUFFER class takes a single voxel and breaks it into separate "slivers" where each sliver projects onto one pixel of the viewing plane. (See Figure 2.) Also, the member functions will output the distance that the sliver is from the viewing plane (the depth) and the total length of the sliver.

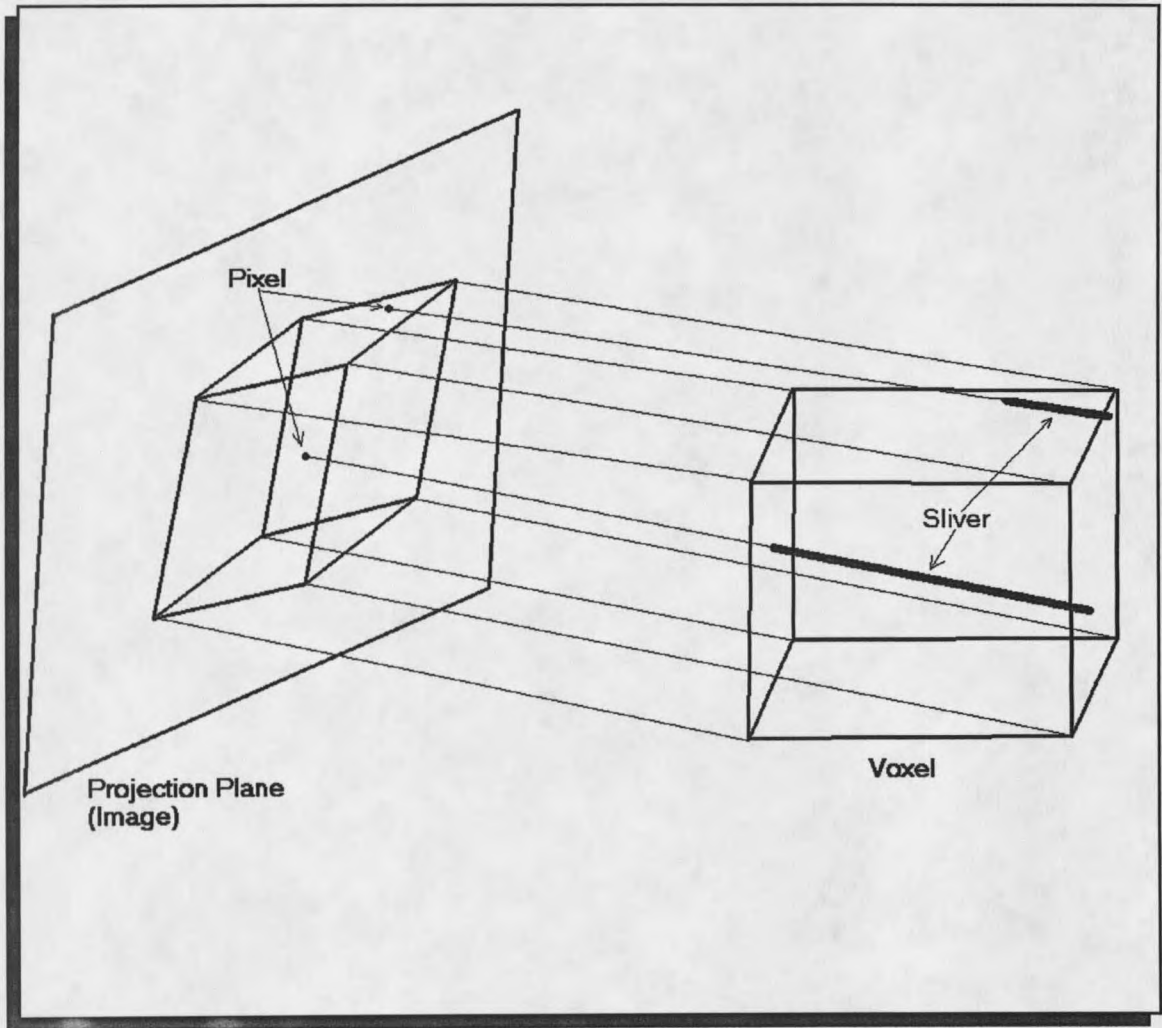


Figure 2. A Sliver projected onto a viewing plane pixel.

CCCCCCC

The user of this class must first call the method Set which takes 8 points defining a voxel and loads the 12 edges of the voxel into the y-buffer by calling the method SetLine 12 times. SetLine stores the x values and the depths of the

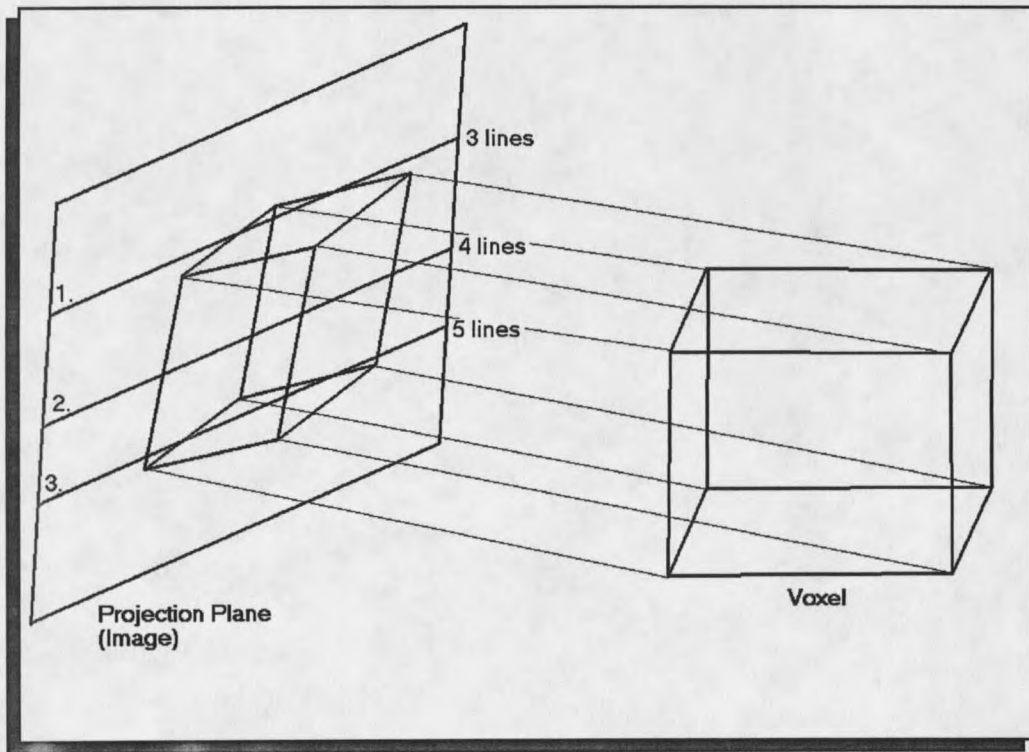


Figure 3. Scan lines across the projected edges.

line at the x values in the range of the y-buffer representing the y values of the line. As Figure 3 shows, at each of the scan lines, there will be either 0, 3, 4, or 5 edges of the voxel which is how many ranges must be processed. If there are 5 edges, then there are 4 ranges.

Once the method Set has finished, the user repeatedly calls the method `GetNextPoint` to return one sliver at a time from the current scan line range. If a range is empty (or one has not yet been retrieved), `GetNextPoint` calls the method `GetNextRange` which calculates and returns the definition of

the next range to be processed. GetNextRange linearly interpolates the depth at each end of one section of the scan line, marks that range as processed, and returns the values to the function GetNextPoint. If the entire voxel is processed and no ranges are available, GetNextRange returns a zero to GetNextPoint which returns a zero indicating that processing of this voxel has been completed.

The VOXEL Class

The VOXEL class provides all functionality and storage for the voxels and the voxel cube. The member and methods of the VOXEL class are listed in Table 8.

The voxel data are read and stored in a 3D array by the method ReadAndSet. For a description of the format of the voxel file, see Appendix B. The location of the voxels are calculated from the location of the voxel cube in world coordinates. Each of the eight corners of the cube is represented by an instantiation of the POINT class. The points are labeled A through H and are located in world coordinates as seen in Figure 4, page 22. The width, height, and length of the voxels are all one centimeter; the same distance as the voxel data values. The center of the cube is initially at the origin, but the method TRS translates,

rotates, and scales these eight points which moves the entire cube into the first quadrant in world coordinates. The viewing plane x and y coordinates are equal to the world x and y values, so when projections are performed, no calculations are necessary. (See Figure 4.)

Table 8. The VOXEL class.

CLASS MEMBER	PURPOSE
size	the max voxel array index
Max, Min	may need for adjusting values
A, B, C, D, E, F, G, H	define the cube in 3D space
AB, AD, AE	orthogonal, normalized vectors from A
VoxelLWH	voxel width, length, and depth in 3D
CLASS METHOD	PURPOSE
ReadAndSet	reads and stores voxel input file
Reset	restores cube to original location for the next TRS
GetVoxel	returns the value of the voxel
GetApproxVoxel	returns the linear approximate value of the voxels
SIZE	returns value in member size
VOXELLWH	return value in member VoxelLWH
TRS	translates, rotates, and scales the voxel cube
GetVoxelTraversalData	determines how to traverse the voxel cube from front to back
GetVoxelCoords	returns the 8 corner points of a voxel
GetCubeCoords	returns the 8 corner points of the voxel cube

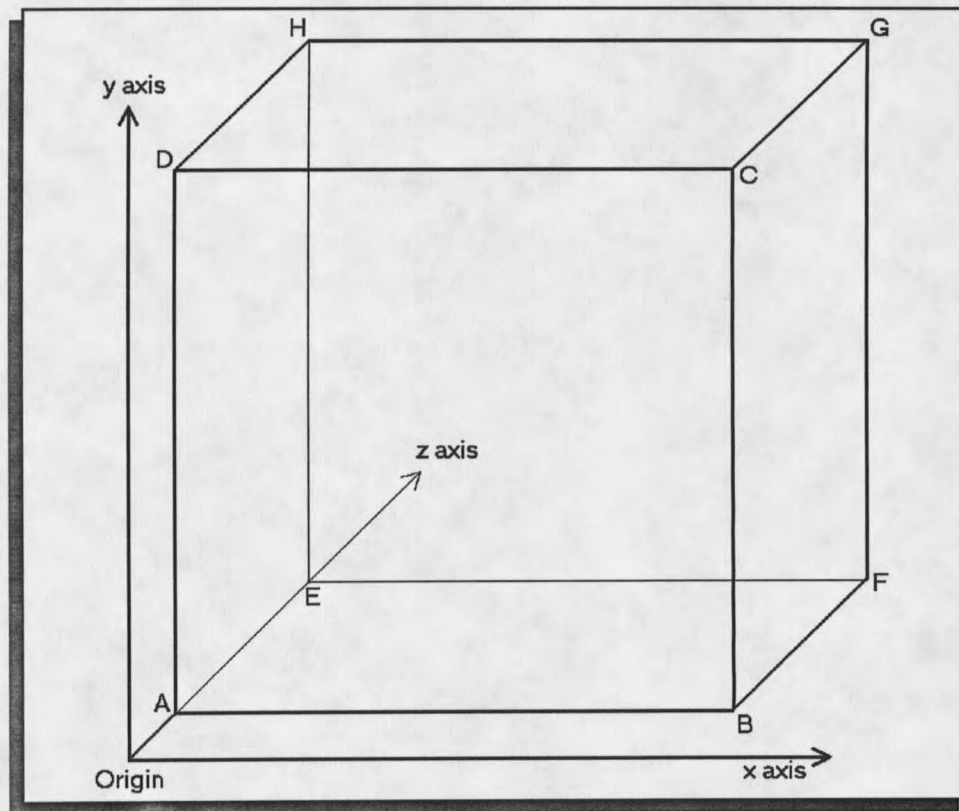


Figure 4. The voxel cube.

The method `GetVoxelTraversalData` determines in what order the voxels should be traversed, so they are processed from the closest to the furthest. By doing this, the density emitter equations in the method `DE::Apply` will process the slivers for each pixel from the closest to the furthest. The following argument explains how the order is determined, but the actual traversal of the voxels is performed in the method `VIEWER::RENDER`. First, the very closest corner of the cube is

found, then the second, and finally the third. The three voxels at these corners are numbered 1,2, and 3 respectively. (See Figure 5.) First, voxel 1 is processed, then each voxel on the path to voxel 2 is processed. This is line 'a' in Figure 5. Once that row has been processed, the adjacent parallel row in the direction of the third closest voxel is processed. Once that entire first layer of voxels is completed (lines 'a', 'b', 'c', and 'd'), the next layer of voxels is completed which starts with line 'e' in Figure 5. This process continues until the last layer is finished.

The quality parameter from the parameter file determines whether the method `GetVoxel` or `GetApproxVoxel` is used to retrieve the density of light emitters in a sliver. The method `GetVoxel` takes 3 integers as parameter and returns the value of the voxel. The method `GetApproxVoxel` takes 3 floats as parameters and returns a single value which is the linear interpolated density of the 8 voxels nearest that point. This produces a much better image because of the smooth transition from one voxel to the next, but the time requirements are very high. The images produced with the method `GetVoxel` might have more analytical value because the true data values are presented, but do not look nearly as good due to the coarseness of the data representing the voxels and the abrupt transitions of voxel values.

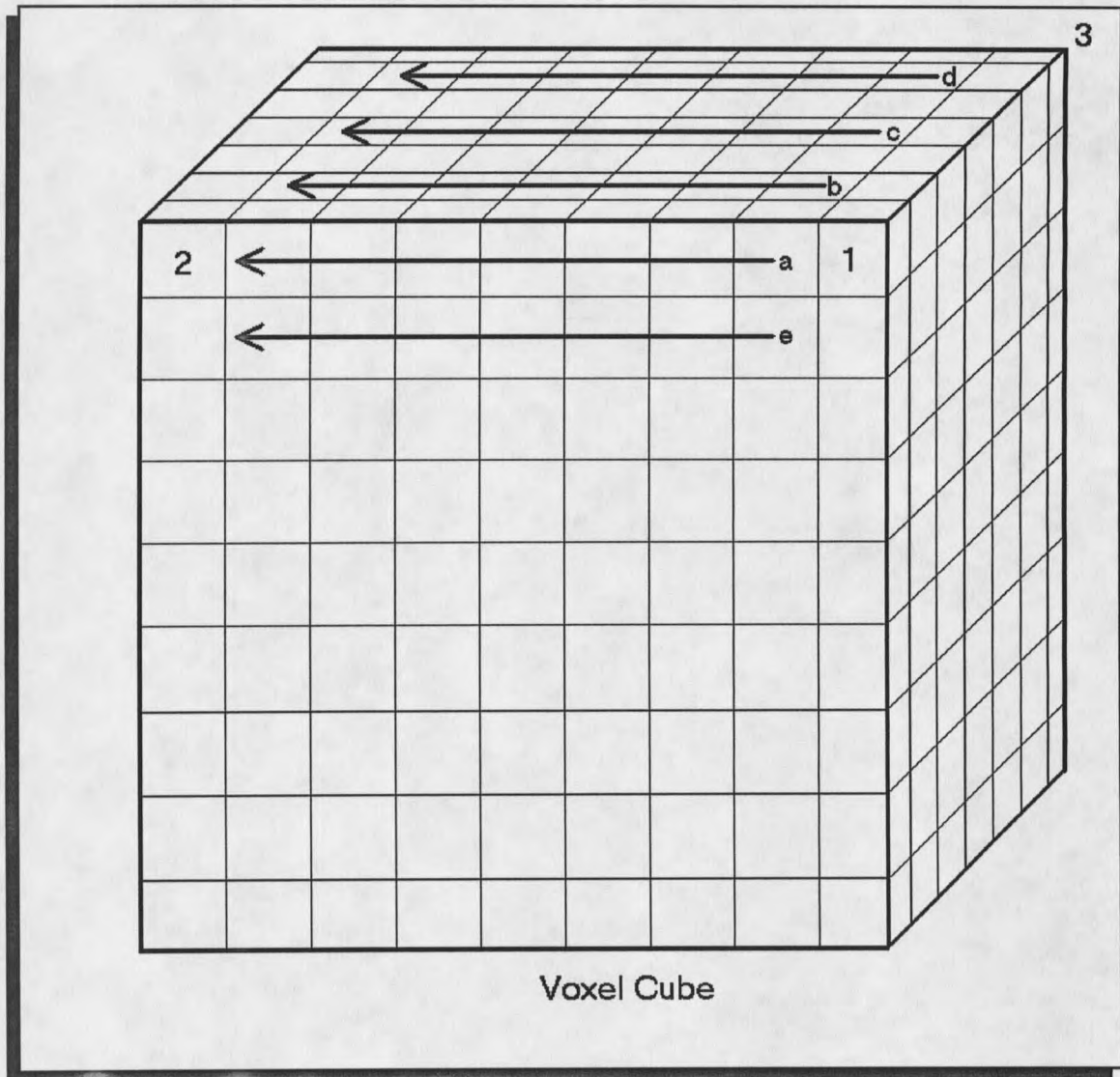


Figure 5. Voxel cube traversal.

The VIEWER Class

This class uses instantiations of the previously mentioned classes to provide the necessary functionality to produce the images. The user of the VIEWER class provides it with only a few parameters via the members `init` and `SetView`. The user calls the method `Render` to produce the image and the method `SaveImage` to save the image. All of the members and methods are listed in Table 9.

This class could produce images from any viewing point, but for this application the viewing point is always at forty five degrees up from the xz plane and at any angle in the xz plane. To change these defaults, modify the method `SetViewParams`.

Table 9. The VIEWER class.

CLASS MEMBER	PURPOSE
vox	VOXEL instantiation
umin, vmin, umax, vmax	viewing window size in VRC
angle	angle of the viewing plane
quality	quality level (0,1)
PRPworld	POINT where the eye is in world coordinates
PRP	POINT where the eye is in VRC
PRPdist	Z value of PRP in VRC
VRP	POINT where the view window is located, it is the origin of VRC
VUP	VECTOR defining the up direction
LookAtPoint	POINT where the camera is aimed
image	IMAGE instantiation for produced images
ybuffer	YBUFFER instantiation
de	DE instantiation
bnct	BNCT instantiation
CLASS METHOD	PURPOSE
ViewVoxel	displays one voxel in IMAGE
SetViewParams	sets all of the viewing parameters, view location, etc
WorldToViewer Coords	calculates and performs the translate, rotate, and scale (TRS)
init	initializes all data and reads data files
SetView	sets viewing parameters and calls method to perform the TRS
Render	performs all steps to render the image
SaveImage	calls the method IMAGE::SaveTGA

The method `Render` first calls the method `VOXEL::GetVoxelTraversalData` to determine how the voxels must be traversed, then it performs the traversal. For each voxel being processed, the method `ViewVoxel` is called which calls the method `YBUFFER::Set` to put the voxel into the y-buffer. Then, within `ViewVoxel`, a loop is started which:

- 1) calls `YBUFFER::GetNextPoint` to get the slivers from the y-buffer one at a time,
- 2) calls either `VOXEL::GetVoxel` or `VOXEL::GetApproxVoxel` depending on the quality parameter of the parameter file to retrieve the density of the density emitter,
- 3) calls `DE::Apply` to calculate the effect of that sliver on the image.

This loop is repeated until `YBUFFER::GetNextPoint` returns a zero indicating that this voxel has been completed.

Main Function

When the program is executed, the main function requires that the first and only command line parameter is the name of a parameter file which contains information that determines how `GDER` will function. The format of the file and the contents are listed in Appendix A.

The main function has one instantiation of the viewer class. Once the parameter file is read, the VIEWER method init is called which reads the data files and prepares the VIEWER instantiation and all of its instantiated classes to produce images. Appendix B contains a description of the voxel data file, and Appendix C describes the BNCT file.

Next, a loop is started from the beginning angle to the final angle. During this loop, the method VIEWER::SetView is called to set the viewing parameters for this angle. Then, VIEWER::Render is called, and finally, VIEWER::SaveImage is called.

Last, the main function outputs various files that may be used to batch build a FLC animation file from all of the independent Targa image files.

CHAPTER 4

RESULTS

GDER Usage

The two parameters, tau and density transmittance, change the appearance of the images output. A discussion with many examples of the effect of these parameters is in [SABE88]. It is best to experiment with the values to find the desired output, but 1.0 and 1.0 are generally good starting values. By increasing tau, the difference between voxel values is increased, so hot spots become more apparent. By increasing density transmittance, the effect of a voxel value upon the final image is decreased, so the entire image will be darker.

GDER can produce a 500 by 500 pixel image with the quality parameter set to high in about 2 minutes on a DEC 5100. So, a 36 frame animation takes approximately 72 minutes to produce the images plus the time to create the animation file from the individual images. Also, GDER may use up to 12 megabytes of memory and 50 megabytes of disk space.

The density emitter can produce an image from any scalar data, however, the data must be within the range 0.0 to 1.0. If the data is not within this range, it can be scaled by the same method as in the method `VOXEL::ReadAndSet`. This is discussed in Appendix B.

To eliminate the parts of GDER that are specific to this application, the BNCT class needs to be removed. The method `VOXEL::ReadAndSet` could be rewritten to read voxel data that is stored in other formats.

Future Work

There are a number of modifications that could be made to decrease the time requirements of GDER.

Several code sections incrementally interpolate points from strait lines. This code could be replaced by a Bresenham type incremental algorithm [BRES77].

The processing of each voxel could be distributed across processors, or by using the ray tracing approach [SABE88] every pixel could be traced in parallel.

It may be possible to derive an equation that would consider the orientation of a voxel in world coordinates and project it onto the viewing plane with the depth values. Thus, the procedures in the YBUFFER class could be avoided.

The fact that each voxel has the same exact orientation with a different location could be taken advantage of. The calculations to break one voxel into "slivers" could be performed only once.

Summary

GDER, the Generic Density Emitter Renderer, is a generic tool that could be used for the visualization of 3D scalar fields in many applications. The program can produce images from any discrete scalar field such as densities of earth, densities of particles in any medium, quantities of ground water, intensities of light, or, like in this case, energy fields. For example, the Monte Carlo program outputs eight separate scalar values for each voxel, and GDER has been very successful in visualizing each of them.

The images output from GDER do provide excellent visualization of the Monte Carlo energy density values combined with the BNCT B-spline surfaces. (See Figures 6 and 7.) GDER produces many images from one set of data by moving the viewing plane in an orbit around the data. These images can be combined into a fly around animation which makes the depth of the data much more apparent, thus providing excellent visualization.

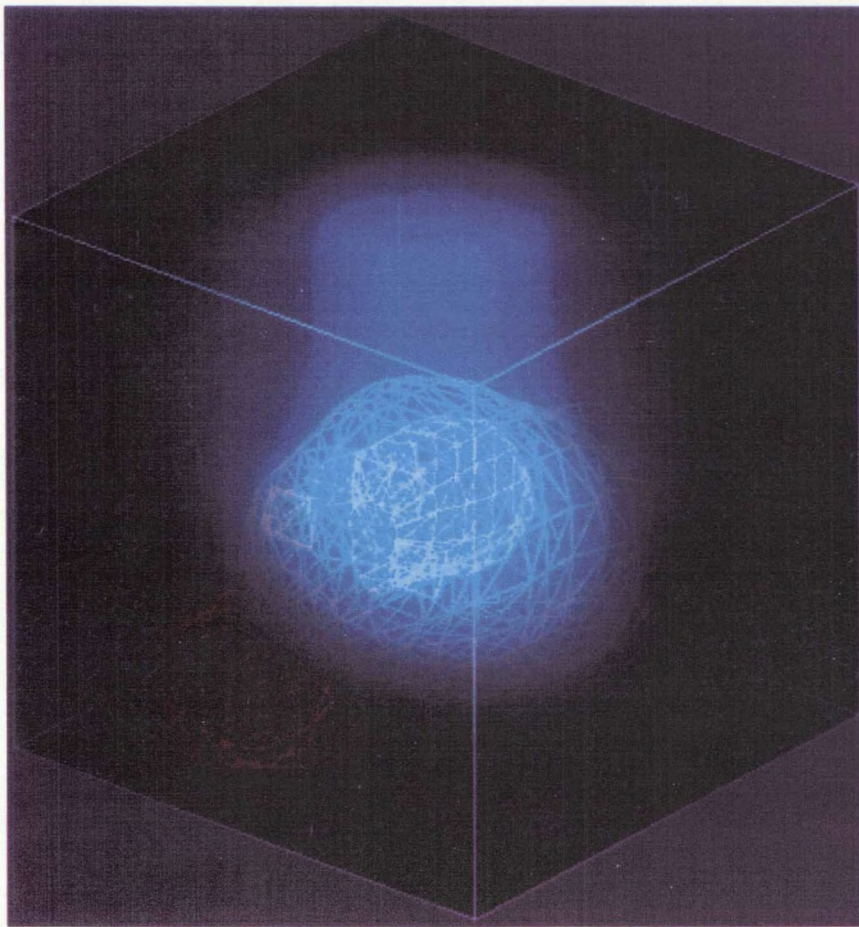


Figure 6. Output Example1.

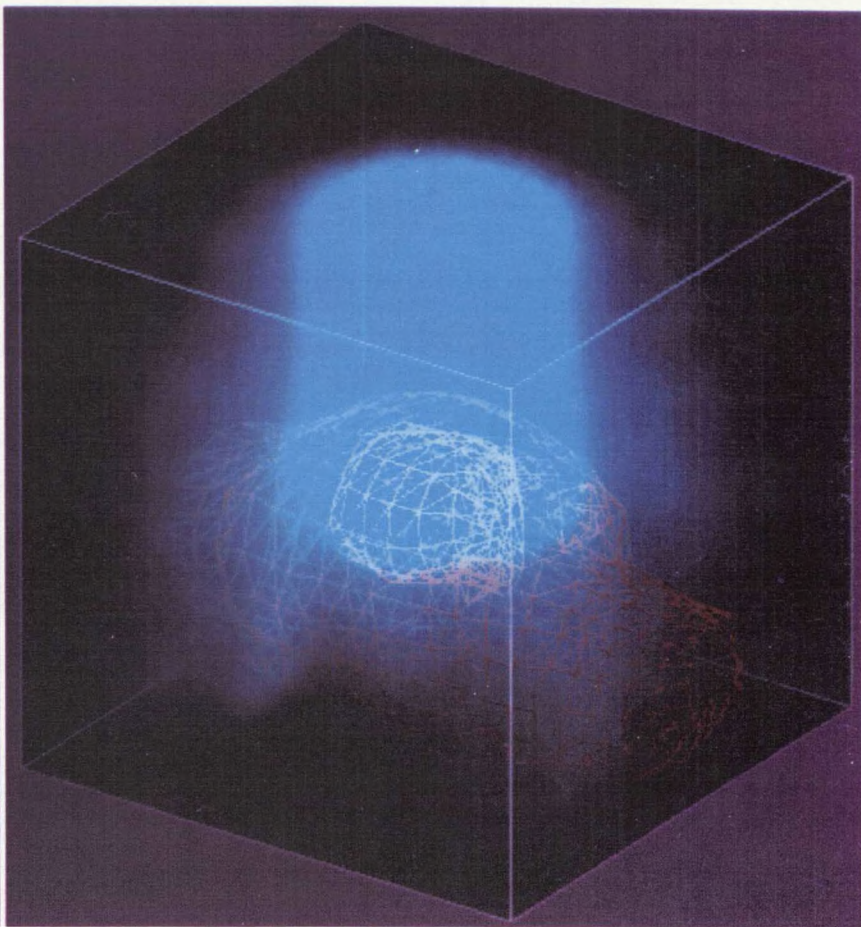


Figure 7. Output Example 2.

REFERENCES

REFERENCES CITED

- BRES77: Bresenham, J.E., "A Linear Algorithm for Incremental Digital Display of Circular Arcs," Communications of the ACM, 20(2), February 1977, 100-106
- FOLE94: Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., Computer Graphics: Principles and Practice, Addison-Wesley, Reading, MA, 1994.
- WHEE94: Wheeler, F.J., "Radiation Transport in Tissue by Monte Carlo--Version X02," EGG-BNCT-11178, Idaho National Engineering Laboratory, January 1994.
- SABE88: Sabella, P., "A Rendering Algorithm for Visualizing 3D Scalar Fields," SIGGRAPH 88, 51-58.
- WESS92: Wessol, D.E., Wheeler, F.J., Babcock, R.S., and Starkey, J.D., "Medical Image Reconstruction and Geometry Generation Software," Proceedings of the Fifth International Symposium on Neutron Capture Therapy for Cancer, Held in Columbus, Ohio, September 13-18, 1992.

APPENDICES

APPENDIX A
PARAMETER FILE FORMAT AND CONTENTS

The parameter file consists of a list of user definable settings which determine the flow of GDER. There are eleven separate parameters and each must be located on its own line starting in the first column. The parameters are:

- 1) the voxel data input file,
- 2) the BNCT data input file,
- 3) the name used to create the output filenames,
- 4) the initial viewing angle,
- 5) the final viewing angle,
- 6) the angle increment,
- 7) tau (a float value used to logarithmically adjust the density values) [SABE88],
- 8) the density transmittance (a float that determines how opaque the voxels are) [SABE88],
- 9) a boolean to indicate pseudo color:
 0--grey scale, or
 1--color,
- 10) a boolean to indicate the quality:
 0--low, fast, and analytical, or 1--high, and
- 11) a number indicating which of the columns of

values from voxel data to use as the density
data:

1--gamma dose,
2--fast dose,
3--total dose,
4--B-10,
5--g-prod,
6--N-14 dose,
7--phi 1,
8--phi 2, or
9--phi 3.

Figure 8 contains an example of the parameter file.

```
voxel.dat  
bnct.dat  
name  
45.0  
45.0  
1.0  
0.6  
1.4  
1  
1  
3
```

Figure 8. Example parameter
file.

APPENDIX B

VOXEL FILE FORMAT

The voxel file contains the energy data calculated by the program Monte Carlo at INEL. The first line of the file contains a single integer which represents how many data points are in any one dimension. The data must have an equal number of data points in each of the three dimensions. The data values start on the second line, and consist of the following fields: x, y, z coordinates, gamma dose, fast dose, total dose, B-10, g-prod, N-14 dose, phi 1, phi 2, and phi 3 in that order. All data values must be on one line with white space between each field. Below is an example of an input file. (See Figure 9.) To change the input file format for other applications, rewrite the method VOXEL::ReadAndSet.

```

2
-0.5 -0.5 -0.5 4.3E-3 3.7E-4 5.5E-3 8.7E-4 0 5.9E-5 1.6E+4 1.4E+5 8.5E+4
-0.5 -0.5 0.5 3.8E-3 7.6E-4 5.7E-3 1.0E-3 0 7.3E-5 1.4E+4 1.5E+5 1.0E+5
-0.5 0.5 -0.5 3.4E-3 4.1E-4 4.9E-3 1.0E-3 0 7.0E-5 9.3E+3 1.5E+5 1.0E+5
-0.5 0.5 0.5 4.7E-3 9.8E-5 5.9E-3 1.1E-3 0 7.5E-5 3.2E+3 1.6E+5 1.0E+5
0.5 -0.5 -0.5 5.2E-3 3.1E-4 6.7E-3 1.1E-3 0 7.8E-5 1.6E+4 1.9E+5 1.1E+5
0.5 -0.5 0.5 4.9E-3 4.7E-4 6.8E-3 1.4E-3 0 9.6E-5 1.0E+4 2.2E+5 1.3E+5
0.5 0.5 -0.5 5.5E-3 1.9E-4 7.5E-3 1.7E-3 0 1.1E-4 6.8E+3 2.4E+5 1.6E+5
0.5 0.5 0.5 5.7E-3 4.3E-4 8.1E-3 1.9E-3 0 1.3E-4 1.2E+4 2.8E+5 1.8E+5

```

Figure 9. Example voxel data file.

The values stored in the voxels should be between 0.0 and 1.0 inclusively. ReadAndSet verifies this, and if negative values do exist, the smallest negative value is subtracted from all of the numbers to eliminate the negative numbers. If there are any values larger than 1.0, all values will be divided by the largest data value to normalize the data between 0.0 and 1.0. These operations will affect the output, and should be prevented if possible.

MONTANA STATE UNIVERSITY LIBRARIES



3 1762 10223530 4

