Device Modelling of Field Emission Displays

by

Pei-Ning Wang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Electrical Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

© Pei-Ning Wang, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 28, 1996

Certified by ....
Tayo Akinwande
Associate Professor
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Eng.

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 11 1996

Device Modelling of Field Emission Displays

by

Pei-Ning Wang

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Electrical Science and Engineering

Abstract

In recent times, field emission displays (FEDs) have generated a lot of interest because they feature the advantages of both CRT and the prevalent flat panel display technologies. The fast pace of current developments in field emission arrays (FEAs) demonstrate a need for efficient device simulation software. The simulation should be capable of calculating electron trajectories from the emitter to the screen and current density distributions on the phosphor screen. It should also have the ability to simulate the distribution of electrons for a lone emitter, as well as for an entire field emitter array (FEA). In addition, the software should be robust, so that various cathode geometries may be implemented; and modular, so that many cathode array elements may be easily simulated together. This thesis describes such a tool for calculating the electron trajectories of field emission cathodes of various geometries.

Thesis Supervisor: Tayo Akinwande
Title: Associate Professor

Acknowledgments

I would like to thank everyone who have given me so much of their time and support during these last few months. Largely, I would like to thank my parents for their encouragement and generosity which has made my life at MIT so much easier and carefree.

Next, I would like to thank Professor Akinwande for his guidance and sensitivity. He is a marvelous advisor, both at a academic level and a personal level. I hope that many other students will take advantage of the opportunity to benefit from him, as I have.

And also I must thank Dr. John Gilbert for his insight and expertise. John has met with Professor Akinwande and myself at practically every meeting. Thanks for your time and advice! Your suggestions have directed my thesis, and put me on the right track toward graduating.

In addition, I would like to thank Professor Senturia and his research group for sharing a few of the machines in their laboratory. Thank you, I could not run my simulations without your help.

And a special thanks goes to Julian Verdejo, whose assurances has kept me smiling through these last days at MIT.

Contents

1 Introduction **7**

1.1 Problem Statement 8

1.2 Research Objectives 9

1.3 Overview 10

2 Background **13**

2.1 Prevalent Flat-Panel Technologies 14

2.2 Emissive Technologies 16

2.2.1 Field Emission Displays 16

2.2.2 Description of FEAs 17

2.2.3 Problems with FEAs 18

2.3 Existing Device Analysis Programs 19

3 Overview of Virtual-FED: the FED Device Simulator **23**

3.1 General Description 25

3.2 Virtual-FED Implementation Details 28

3.2.1 Special Constructs 28

3.2.2 Octree Field Estimation 34

3.2.3 Trajectory Convergence Implementation 34

3.2.4 Monte Carlo Implentation 35

3.3 Limitations and Requirements 36

3.4 Virtual-FED Usage 36

4 Virtual-FED Simulation Results **41**

4.1 Sample Trajectory Evaluations 42

4.1.1 Electron Self-Focusing 45

4.1.2 The Effects of Gate Voltages 48

4.1.3 Distribution Effects 51

4.1.4 The Effects of Initial Electron Energy 57

4.2 Comparison of Virtual-FED Results with Analytical Expressions 61

5 Conclusions and Recommendations **63**

5.1 Recommendations for Future Work 65

5.1.1 Integration with MEMCAD 65

5.1.2 Trajectory Test Programs 65

5.2 Conclusions 67

A trajectory.h **71**

B trajectory.h **77**

C octree.h **103**

D octree.m **107**

E initial.m **131**

F sample main.c **133**

List of Figures

2-1 Color LCD Cross-Section 15
2-2 A single emitter in a field emission display 17
2-3 Major Types of Field Emitter Arrays (FEAs) 18

3-1 Virtual-FED Processes: Flow Chart 26
3-2 Top `Octree Box` 29
3-3 Box Data Structure 30
3-4 Support Structures for `Box` 31
3-5 `Octree` Data-Structures 32
3-6 Diagram of Simplified `Octree Structure` 32
3-7 `Trajectory` Data-Structures 33
3-8 Diagram of `Trajectory Structure` 34
3-9 Semi-Circular Cone Emitter 35
3-10 Virtual-FED: Structures User Must Initialize 38
3-11 Virtual-FED Sample Code 40

4-1 Range of theta: $V_A = 2000V$, $V_G = 100V$, $E = 0.1eV$ 43
4-2 Self Focusing: Distribution of Electrons Positions 46
4-3 Self Focusing: Energy Distribution 47
4-4 Gate and Anode Voltages: Distribution of Electron 49
4-5 Gate and Anode Voltages: Energy Distribution 50
4-6 Uniform Probability Density Function $f_\theta(\theta)$ 52
4-7 Gaussian Probability Density Function $f_\theta(\theta), \sigma_\theta = 0.88$ 52
4-8 Gaussian Probability Density Function $f_\theta(\theta), \sigma_\theta = 0.2$ 53
4-9 Distribution Effects: Distribution of Electron I 54

4-10 Distribution Effects: Distribution of Electron II 55
4-11 Distribution Effects: Energy Distribution 56
4-12 Effects of Initial Electron Energy: Distribution of Electrons 59
4-13 Effects of Initial Electron Energy: Energy Distribution 60

5-1 IFE-FEA Structure with Focus Electrodes at Gate Level 66
5-2 IFE-FEA Structure with Focus Electrodes above Gate Level 67

List of Tables

2.1 Comparison of Flat-Panel Display Technologies 14
2.2 AMLCD Luminous Efficiency 15

4.1 Parameter Values for Virtual-FED Test Cases 45
4.2 Test Parameters: Self-Focusing Examples 45
4.3 Test Parameters: Effect of Gate Voltage 48
4.4 Test Parameters: Effect of Initial Velocity Distributions 53
4.5 Test Parameters: Effect of Initial Electron Energy 57

Chapter 1

Introduction

In recent years, flat-panel displays have found broad acceptance and application in the areas of portable information systems, such as lap-top computers used for in-field data-acquisitioning and record keeping. In addition, military and avionic groups have expressed interest in flat-panel technologies, with emphasis on developing interfaces for intelligent human-machines [1]. The commercial applications of flat-panels are endless, ranging from products such as personal notebooks to large-screen, HDTV systems. This growing demand for flat-panel technologies has demonstrated a need for developing a flat display technology with high resolution and high brightness, while still maintaining low power consumption and cost.

Although conventional CRTs have exceptional brightness and resolution characteristics, they are unsuited to the needs of portable systems due to bulkiness and weight constraints. Prevalent flat-panel technologies are lightweight and thin solutions, with tradeoffs in picture quality and power consumption. However, present day advances in integrated circuit fabrication technology have allowed Field Emission Displays (FEDs) to emerge as an alternative flat-panel display technology. This latest technology functions essentially as a flat panel CRT, and hence, brings the best qualities of the CRT together with the portability of flat-panel displays.

These desired qualities motivate FED researchers to advance at lighting speed. In order for the science of FEDs to continue at its current rapid pace, designers need efficient simulation models for FED environments. Thus far, FED development has been restrained by the lack of computer simulation tools. CAD tools are necessary to keep development

time and costs at a minimum.

1.1 Problem Statement

Despite the fast pace of development in FED technology, FED developers are still unsure about future directions in the field. There are a number of questions which need to be answered before the technology can reach its full potential. FED researchers would like to know the effects of specific design choices on FED performance *before* an FED is physically implemented. Their questions include:

- What are optimal gate, screen, and focus electrode (if present) operating voltages?
- Given device dimensions and voltage parameters, what is the current density distribution (spot size) on the phosphor screen?
- What parameters are necessary to produce a reliable and reproduceable device?
- How does emitter geometry affect current uniformity?
- How closely should field emitters be spaced in an FED?
- To what extent do focusing electrodes help narrow the electron beam?
- How far vertically from the emitter should the focusing electrode be spaced?
- How far can the distance between the emitter cathode and the screen be extended with the usage of various focusing electrodes?

If a suitable electron trajectory simulation program were available, it could have the capability of predicting accurate solutions to these questions for FED developers. To date, there are no available CAD tools which adequately model field emitter array (FEA) device performance. And prior to the start of this work, there appeared to be no leads for future designs.

The benefit of having such a CAD tool is that it would bring FED development time and costs to a minimum. It would be able to propel FED technology forward by allowing the simulation of novel devices, without the costs and hassle of physical implementation. In addition, a FED device simulator could allow FED designers to find the parameters to optimize an implementation. In order for a FED design to be an optimal design, it must demonstrate a high proficiency within four key attributes [2]:

- Feasibility of large size cathode with uniform electron emission and high manufacturing yield
- Feasiblility of full color device with brightness, resolution and high luminous efficiency at moderate anode voltage.
- Acceptable driving voltages.
- Long life.

A CAD tool, which calculates electron trajectories to predict spot size upon the screen, could effectively determine the first three characteristics for an optimal FED device. The key feature of a CAD utility is to estimate the current density distribution of a single emitter on screen of the display, taking into account the variations of emission current. After the calculation of the current distribution from a single emitter, the solution may be replicated at various positions on the FEA and superimposed to give an overall current density distribution for the screen. With these qualities, the CAD software would be a useful analytical and predictive tool for estimating the relationship between device spot size and the physical design parameters, such as the optimal spacing between the cathode and screen and the cone to cone spacings in FEAs.

Currently, the FED industry needs computer simulation tools equivalent to the design tools which have revolutionized other engineering fields, such as IC design and fabrication. VLSI CAD utilites including $CADENCE^{TM}$ and $MentorGraphics^{TM}$ have produced radical changes in the IC hardware industry by allowing product development times to decrease dramatically; thereby, lowering costs and increasing innovation. FED technology could benefit immensely with a complementary device simulation program, parallel to those of the VLSI industry.

1.2 Research Objectives

The research objective of this paper is to design Virtual-FED, a CAD utility which is suitable for modelling FEAs. The intended purpose of the program is to predict the current density distribution, along the monitor screen, as a function of gate and anode potentials and arbitrary emitter spacings. Using Virtual-FED software, FED designers will be able to predict the brightness, spot size, and their uniformity across the screen.

The Virtual-FED utility is similar to existing FEA simulating software. However, in addition to the features described by other methods, the discussed simulation predicts

a current density distribution on the screen and introduces a monte carlo scheme which models the effects from variations of emission current and estimate actual current density/uniformity at the screen. The following is a listing of the essential elements of the Virtual-FED software.

- Calculate electric field (boundary element method)
- Compute current density on emitter tip using the Fowler-Nordheim Theory
- Simulate the electron trajectory
- Calculate the current density distribution on the screen
- Use monte carlo technique to account for non-normal electron emission
- Correlate the calculated current density distribution with actual data

The unique contribution of this CAD design is that it calculates current density distributions, and uses a monte carlo technique for determining electron emissions which are non-normal to the surface of the emitter tip. The monte carlo scheme allows for predictions of actual experimental distributions; this capability can be extremely useful. For example, such a tool could be used to predict the minimum tip radius required for a particular brightness or uniformity. It could also be used to estimate the minimum spacings between emitters for a given spot size.

Other advantages of this proposed implementation are that it enables the design of novel FEA structures and displays, and the utility will be integrated with MEMCAD, which provides a fast and efficient means for calculating electric fields.

1.3 Overview

The next chapter, Chapter 2, presents a brief background of current flat-panel technologies. First, a description of the various display types is given in order to form a comparative benchmark with FED displays. Then, the advantages and disadvantages of emissive displays are discussed; and the physical nature of FED environments is introduced. Finally, previous work involving computer analysis to model current distributions on the FED monitor screen is presented.

A complete description of the Virtual-FED simulation program is given in Chapter 3. An overview of the simulation procedure for calculating electron trajectories and electron

energy distributions at the screen is given in the first section. The next section focuses on implementation details, such as the special constructs. Then, the limitations and requirements for using Virtual-FED are outlined. Lastly, the procedure for using the program is presented.

Several test cases, generated with the Virtual-FED utility, are presented in Chapter 4. The results of these examples form general conclusions about issues, such as (*i*) the anode operating voltage, or self-focusing, (*ii*) gate operating voltage, (*iii*) current distribution, and (*iv*) the initial energy of the electron. After each of these points are discussed, concerns about the simulator accuracy and practicality are addressed.

The final chapter contains a discussion of the performance of Virtual-FED and suggestions for future work. Possible future extensions of this thesis work are also introduced.

Chapter 2

Background

Recent popularity of portable information systems has led to a need for developing low cost and low power technologies for flat panel displays. Current flat-panel display technology offers a wide variety of implementations: passive and active-matrix liquid-crystal displays (LCDs), electroluminescent displays (ELDs), plasma display panels (PDPs), and vacuum fluorescent displays (VFDs), and so on. Each of these implementations has its own advantages and disadvantages (refer to [3] for a detailed overview). However, for a *typical* lap-top computer, the display itself embodies $40 - 50\%$ the cost of the system, and consumes approximately 50% of the power in the entire system. In fact, none of the current flat-panel technologies satisfy the requirement for portable information systems; which demand low power, high luminous efficiency, high contrast, high information content and full color. They must at the same time have small volume and be lightweigth. Table 2.1 summarizes the governing attributes of popular flat-panel technologies.

The most dominant display technology is the cathode ray tube (CRT). It is efficient in power however bulky and unportable. An ideal case would feature (1) the high efficiency of CRTs, (2) the portability of flat panel displays, and (3) low cost in manufacturing. FED technology has the potential for providing all of these features, since it is essentially a flat CRT.

To date, the LCD display is the most advanced of the plat-panel displays under development. In recent years, improvements in manufacturing technologies have allowed LCDs to gain wide acceptance. This flat-panel architecture has dominated the market for portable computers, lap-tops, and other portable computing systems. In addition, LCDs are becom-

Table 2.1: Comparison of Flat-Panel Display Technologies

	Shadow Mask Cathode Ray Tube	Liquid Crystal Display	Electro-luminescent Display	Vacuum Fluorescent Display	Plasma Display
Power Consumption	$200W$	$100W$	$40-50W$	$60W$	$60-80W$
Contrast Ratio	Excellent (with filter)	Medium	Good (with filter)	Good (with filter)	Good (with filter)
Viewing Angle	Excellent	Poor	Good	Excellent	Excellent
Luminance	$350cd/m^2$ (with filter)	$350cd/m^2$ (with filter)	$100cd/m^2$	$70cd/m^2$	$70cd/m^2$
Color	Best	Full	Green or yellow	Full	Full
Resolution	Excellent	Excellent	Excellent	Satisfactory	Satisfactory
High Ambient Light Readability	Good (with filter)	Excellent	Satisfactory (with filter)	Poor	Satisfactory (with filter)
Frame Rate	$60Hz$	$60Hz$	$60Hz$	$60Hz$	$60Hz$
Pixel Matrix	2048 x 2048	1024 x 1024	864 x 1024	400 x 640	2048 x 2048
Temperature Resistance	Excellent	Poor	Satisfactory	Satisfactory	Satisfactory
Humidity Resistance	Satisfactory	Poor	Satisfactory	Satisfactory	Satisfactory
Shock and Vibration Resistance	Satisfactory	Excellent	Excellent	Satisfactory	Excellent
Price	Very low	High	Moderate	Low	Moderate

ing incorporated into products, such as flat screen and projection entertainment systems.

2.1 Prevalent Flat-Panel Technologies

Liquid-crystal display (LCD), the prevalent flat panel technology, works essentially as an addressable light valve, composed of a polarizable liquid-crystal material sandwiched between two sheets of glass (See Figure 2-1). Along the surface of the glass sheets are layers of transparent conducting material. An electric field can be induced in the liquid-crystal by applying a voltage across the conductors; this field rotates the orientation of molecules in the liquid-crystal. Light generated by the backlight/diffuser is sent through a pre-polarizer and then this filter. The intensity of light at the screen is determined by the rotation angle of the liquid-crystal molecules. When the backlight passes through the filters, it loses intensity. Overall, in a full color LCD, only about 5% of light from the backlight actually

Table 2.2: AMLCD Luminous Efficiency

	Theoretical	Actual
Backlight		$50 lumen/watt$
Diffuser		30%
First Polarizer	50%	43 – 45%
Aperture Ratio of TFT/LC		50 – 50%
Color Filters	33%	25%
Small Losses (Absorption/Reflections)		90%
Exit Polarizer		87 – 89%
Luminous Efficiency (excluding Diffuser)		$\approx 5\%$
Luminous Efficiency (including Diffuser)		$< 2.5\%$
Luminous Efficiency of AMLCD		$\approx 1 lumen/watt$

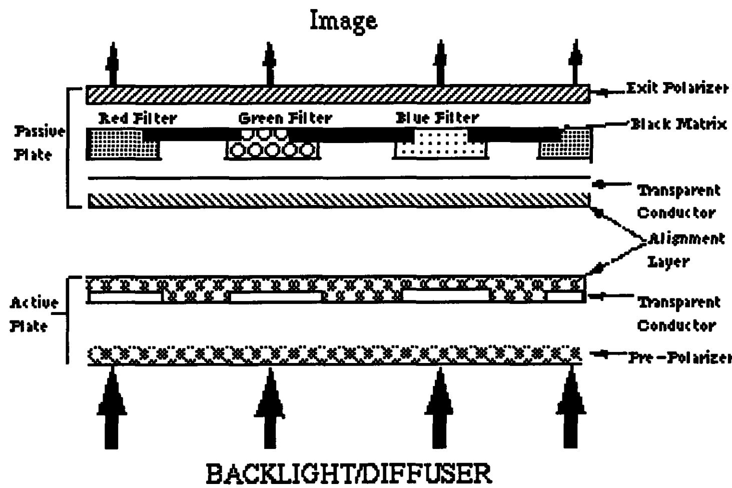


Figure 2-1: Color LCD Cross-Section

reaches the screen. This is summarized in Table 2.2.

Although LCDs are easily addressable and inexpensive to produce, they do have many shortcomings: (1) the requirement of a uniform and controlled light source, (2) restrictions in the dynamic range of color for each pixel, (3) the loss of at least one-half of the total light through the basic polarization process, (4) slow refresh rates due to the slow response time of the liquid-crystal, (5) a strong variation of light intensity at different viewing angles, (7) sensitivity to temperature and pressure, and (8) filters must be matched with the color spectrum of the light source [3].

2.2 Emissive Technologies

LCDs have other disadvantages which are non-existent with emissive display implementations such as CRTs and FEDs. CRTs produce light extremely efficiently through cathodolumminescence – the process by which light is produced from an electron striking a phosphor. The advantage of cathodolumminescence is that the quality and intensity of light may be controlled and also the light is unpolarized and may be viewed over a wide angle without variation in intensity, color purity, or contrast. Several advantages of cathodoluminescence are listed from Henry Gray in his article 'The field-emitter display' [3].

- High Iuminous efficiency
- High brightness
- High dynamic range in brightness
- Full color
- Outstanding color purity
- Wide viewing angle
- High spatial resolution
- Designable persistence (to prevent flickering, yet allow a fast refresh)
- "Simple" manufacturing processes

2.2.1 Field Emission Displays

The primary problem with the CRT display is that it is requires much volume and weight. These qualities are mainly attributed to the bulkiness of the cathode ray tube. The proposed

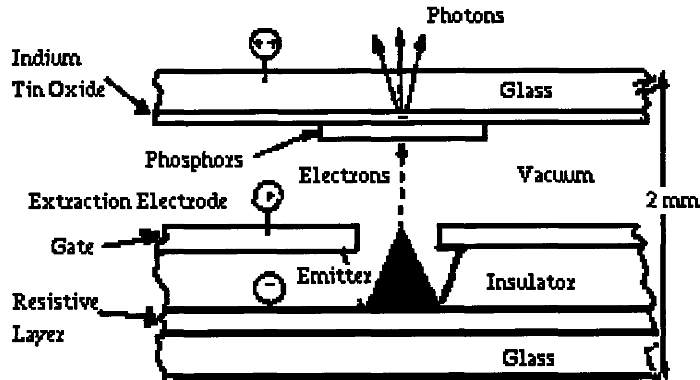


Figure 2-2: A single emitter in a field emission display

solution is to create FEDs which combine the advantages of cathodoluminescence with the desirable volume and weight of current flat panel displays.

The FEDs offers a more elegant implementation of the CRT, by replacing the single cathode ray tube with a two dimensional array of minute cathode emitters, known as field emitter arrays (FEAs). FEAs have the following properties [3],

- Individual pixel can be targeted on or off
- Has built in sub-pixel redundancy - if one elementary cells fail, others still work
- Can be batch fabricated
- Has high spatial resolution
- Capable of high current density
- Is temperature insensitive
- Can be made extremely thin or flat

2.2.2 Description of FEAs

A single emitter in a field emitter array (FEA) acts as a miniature electron gun (See Figure 2-2). When a sufficiently large voltage is applied between the emitter and the gate, by means of the extraction electrodes, electrons tunnel out of the emitter. The electrons accelerate through the gate aperture and onto the phosphor screen.

Electrons tunnel from the emitter when a large electric field on the order of $10^9 V/m$ is induced around the emitter structure. This process is described by quantum-physics.

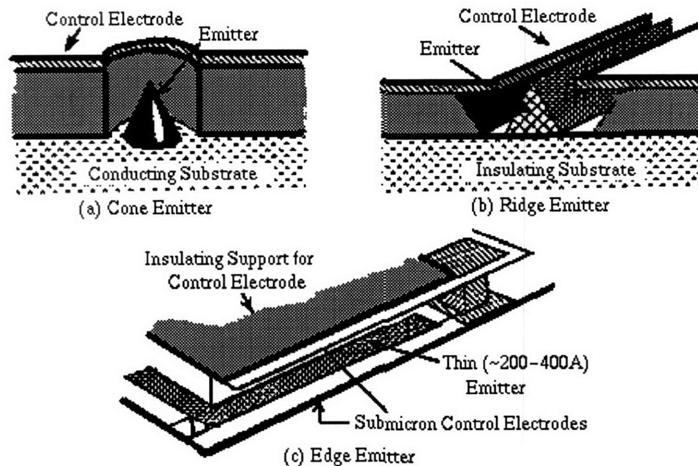


Figure 2-3: Major Types of Field Emitter Arrays (FEAs)

The rate of change of electrons across the area of the emitter may be described by the Fowler-Nordheim equation for current density [4],

$$J = \frac{AE^2}{\phi t^2(y)} exp[-B\phi^{1.5\nu(y)}/E] \tag{2.1}$$

where $t^2(y) = 1.1$, $v(y) = 0.95 - y^2$, $A = 1.54 \times 10^{-6}$, $B = 6.87 \times 10^7$, and $y = 3.79 \times 10^{-4}E^{1/2}/\phi$. Increasing the total current density from an emitter increases the brightness of the corresponding pixel [5]. This fact enables easy adjustment over the resolution and contrast of the image.

Since the electric field is naturally strongest around the tip of a sharp object, shaping a cathode emitter with the geometry of a sharp object, such as a cone, greatly reduces the potential necessary to make electrons arc from the emitter. With a pointed design, operational voltage ranges from 100V-300V, rather than 1000V-30000V [6, 1976]. The major designs of FEAs are shown in Figure 2-3.

2.2.3 Problems with FEAs

The main challenges confronting FED development are (*i*) development of spacers between the screen and gate substrate, and (*ii*) issues of uniformity, reliability, and stability. Spacers must be strong enough to sustain an evacuated cavity and thin enough so that they do not interfere with the current density distribution on the screen. In addition, current FEDs use low voltage phosphors which have poor efficiency, about $5lumen/Watt$, as compared to CRTs which use high voltage phosphors with an efficiency of $25lumen/Watt$. Current

FED technology does not allow for the usage of higher voltage phosphors because of spacer breakdown at shorter distances. Also, if higher voltages were used, a trade-off occurs between resolution and efficiency, since increasing the voltage level requires an increase in vacuum gap [7]. And thus, an increase in spot size follows. One solution is to add a focusing electrode aperture within the vacuum region itself. Designing such a structure would require CAD tools, which have yet to be developed. Discussed issues with FEAs are listed in a nutshell:

- Necessity for strong, long-lived spacers
- Current FEDs, using low voltage phosphors, are inefficient (5 Lumen/Watt)
- Trade-off between decreasing spot-size, i.e. increasing resolution, and decreasing luminous efficiency. This can be avoided by the use of higher screen voltages with the appropriate spacers and electron focusing optics.
- Need for focusing electrodes
- No appropriate CAD tools exist for modelling FEDs and FEAs.

2.3 Existing Device Analysis Programs

As mentioned in Chapter 1, to date, there are no available CAD tools which accurately model FEA performance.

In their paper *Modelling of a Field Emision Display Using the Adaptive Scheme Method*, Kyung C. Choi and Nagyoung Chang describe an adaptive scheme method which calculate the trajectories of electrons when a low voltage is applied to the anode of a cone-shaped emitter [8] (See Figure 2-2). Characteristics of the electron contact position with the phosphor screen were discussed, as well as the effects concerning varying the emitter tip radius, etc. However, the paper does not mention the capability to calculate current density distributions on the phosphor screen. Without this feature, their utility is not a practical predictor of spot size or resolution.

W. Dawson Kesling and Charles Hunt describe a technique for simulating the performance of field emitters in FEDs, combining both finite element and finite difference analysis [9]. Their program is similar to this work, using emission currents, along the cathode surface, which are calculated from the Fowler-Nordheim equation, and trajectories are calculated using a fourth-order finite difference scheme. However, it has several limitations. At far

distances from the cathode, trajectories are extrapolated using the exact solution for electrons in a uniform field. This shows that the difference scheme is not self adjusting, i.e. it does not self compensate for the changing strengths of the fields. In addition, they have assumed a cylindrically symmetric field about the emitter environment, so their program cannot explore outcomes based upon asymmetrical irregularities on the cone's surface.

In their paper *Electron-beam induced deposition for fabrication of vacuum field emitter devices*, the team of Weber, Rudolph, Kretz, and Koops relate a Monte-Carlo simulation program. The simulation is used to model the scattering of the primary electrons in an emitter tip, in order to calculate the spatial distribution of leftover energy at the tip [10]. Although, the program is capable of calculating beam energy, the current, and the effects of material properties at the tip. It does not go on to calculate actual electron trajectories and the current density distribution along the surface of the screen.

Fedirko, Belova, and Makhov describe a physical and mathematical simulation model for calculating the effectiveness of beam control optimization [11]. Their work takes advantage of the cylindrical symmetry for a cone-shaped FED device; however, this restricts the program to only symmetrically shaped devices. This confinement makes their program unsuitable for the calculation, of spot size and resolution, for all FED device geometries

Munro, Zhu, Rouse, Liu present a 3D finite difference program, that calculates electron trajectories using the fourth-order Runge-Kutta algorithm in their paper entitled *Computer Simulation of Vacuum Microlectronic Components* [12]. The program uses special equations at the interfaces between the electrodes and the free space regions. The potential values, calculated from the equations, are stored in a three-dimensional grid point system. This program is extremely similar to the proposed Virtual-FED implementation. Their paper does not mention whether the finite difference method, that they are using, is self adjusting to compensate for highly varying fields. In addition, they do not describe any special methods, involve in the implementation of their three-dimensional grid lookup table. Hence, one cannot conclude from their paper, if they have a fast and efficient simulation program for modelling FED devices.

In summary, existing FED device simulation programs have many design flaws. For example, most of the programs use finite differece methods; however none of them mention the capability to self adapt in step sizes. Without the ability to self adjust, they shall need to scale geometries much bigger or change anode sizing in order to provide computationally

reasonable results. In addition, most recent FED device simulators describe their programs to utilize cylindrical symmetry the cone-shaped emitters. These implementations will not be able to support any FED devices which are asymmetrical about the z-axis.

It is necessary to design a program that can start from a geomentry of fine granuarity, that self adjusts into coarser environment. In addition, it should be able to compute entire FEAs.

Chapter 3

Overview of Virtual-FED: the FED Device Simulator

The FED device simulator, Virtual-FED, models field emission devices, particularly the field emitter display. The simulation package is intended to be used to explore field emission design as well as to help develop and analyze experiments to illuminate the underlying factors that describe field emission from manufactuable material surfaces.

Virtual-FED is intended for generalized device analysis and design. It is currently built to be included as a module of an existing software system, MEMCAD. MEMCAD is designed to model arbitrary three-dimensional device structures, which helps build structures from mask and process information. It already has existing modules for solving 3D electrostatics, structural mechanics, and coupled-electromechanics. In future implementations, the Virtual-FED will use the MEMCAD system to construct models of arbitrary field emitter geometry and calculate electric fields throughout the device. However, in this thesis an analytical field solution is used. Electron emission current density at the field emitter tip are calculated by assuming the electrons are emitted normal to the surface. This is refined to take into account non-zero tangential electron energy. The Monte-Carlo and other statistical techniques are used to account for non-normal electron emission from the surface. Trajectory calculation is used to predict electron current density and its spatial variation on the phosphor screen. Eventually, the Virtual-FED will calculate the luminance of the phosphor screen and its uniformity using the electron density and known phosphor characteristics. Virtual-FED will also allow the design of FEAs that use focusing electrodes

to reduce the spot size. The simulator will model the characteristics of a FED pixel and it can be linked to other display simulation programs which model the full display system. In addition, it will also be used to understand complex experimental results from field emitter.

In this thesis, the Virtual-FED simulation tool includes the following elements:

- Electron trajectory calculations
- `Octree` data structure
- Electron distribution functions to account for non-normal electron emission

This version of Virtual-FED implements the routines and functions that will be able to predict electron trajectories from the emitter to the phosphor screen, electron distribution at the phosphor screen and with the addition/or implementation of the Fowler-Nordheim equation, the current density and its uniformity at the phosphor screen.

Virtual-FED functions essentially as a library in the C programming language to a very powerful environment for implementing electron trajectory paths. It determines field emitter characteristics based upon input parameters. All the functions for modelling the cone-shaped FED are provided: linear and fourth order Runge-Kutta difference methods, a linear convergence algorithm, and a fast lookup-table for storing precalculated fields. In addition, Virtual-FED uses spatial decomposition data structures (the `Octree`) as well as Monte-Carlo techniques for following charged particle trajectories. The bulk of the simulation functions may be divided into two main categories. The first group calculates and/or prestores the fields within the FED environment; the second determines the trajectory paths of electrons and generates a specified distribution of the current density upon the screen.

Initially, the electric field parameters are calculated and stored in a structure designed for field storage, or an `Octree`. The `Octree` is refined where the field is highly varying, and coarse where the field is unchanging. Following field storage, the Virtual-FED uses a monte carlo scenario, generating either a uniform or gaussian distribution of the initial electron direction from the tip of the emitter, with respect to angle off the axis of the emitter. Electron trajectories are repeatedly called through the monte-carlo scheme. The trajectory of an individual electron is calculated using a fourth-order Runge-Kutta adaptive difference method. When the electron is within areas of highly changing fields, the incremental step sizes vary accordingly. Next, a convergence algorithm checks whether the trajectory path

is reliable. If the trajectory is found inaccurate, then the trajectory is recalculated using a finer initial step size.

The following sections describe the program in greater detail. First, a general description of the Virtual-FED simulation procedure is given. The next section focuses on the implementation details, such as the special constructs. Then, the requirements and limitations of the program are outlined. And lastly in Section 3.4, a tutorial for using the program is presented.

3.1 General Description

Procedures for calculating current density distributions along a given area of the screen may be condensed onto a simple flow chart Figure 3-1. First, during the initialization stage, electric field values are calculated and loaded into the `Octree` storage structure. Then, the electron trajectory paths can be calculated, using the electric field values in the `Octree` lookup table. This simulation process is summarized under the following items:

1. Calculate or load fields of a FED environment
2. Refine fields and store fields into the `Octree`
3. Use monte carlo to generate an initial electron velocity, under a specific distribution
4. Calculate trajectory path through the difference method, using the `Octree` as a lookup table.
5. Test for trajectory path accuracy, if inaccurate go back to 4.
6. Store trajectory data into `Octree`
7. Calculate another trajectory path by returning to 3.

Each step performs an important role for determining trajectory paths in a fast and efficient manner.

The first step refers to initializing or scanning in the electric fields. Currently, this is accomplished by analytically determining the field at each desired point within an FED structure. Eventually, this utility will be integrated into MEMCAD, a 3D solid modelling program, which provides a fast and efficient algorithm for calculating fields. MEMCAD will make it easier for the user to define more complex and computationally intensive fields.

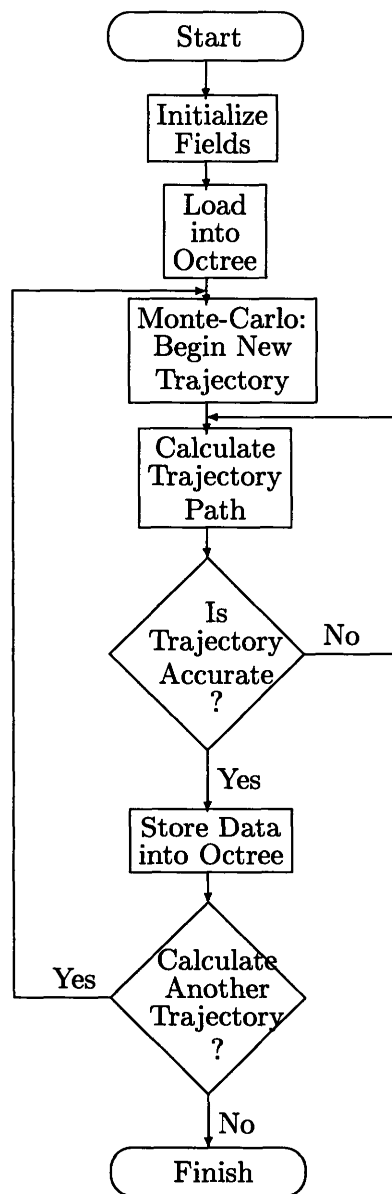


Figure 3-1: Virtual-FED Processes: Flow Chart

The ability to quickly incorporate any combination of field values into the system allows for a multitude of different configurations of FED device structures. Thus, the program is versatile enough in nature to embody the fields for any distinctively shaped emitter.

The second item on the list refers to refining and storing the fields into the `Octree`. This procedure may occur concurrently with the calculation of fields. The `Octree` is a data-structure, similar to a binary search tree, except it contains eight branches leading from a parent node rather than two branches. And it functions as an efficient table for finding fields, given a position within the environment. In this case, the average computation time for such a search would be $O(\log_8 n)$ as compared to $O(n)$ for the fully degenerate case, where n is the number of nodes within the `Octree` [13].

Step three begins the trajectory computation stage. After the fields are initialized into the `Octree`, the monte carlo scheme is used to randomly generate an initial electron velocity vector, under a specified distribution along the emitter tip. Presently, the user may choose to establish a gaussian or uniform distribution for electron inertial vectors normal to the emitter surface at an angle, *theta*, away from the z-axis.

For the next procedural item, the electron trajectory is calculated through an adaptive difference method, based upon changes in time. That is, the length of the time increment is a function of the refinement for electric fields at a particular position in the FED environment, the velocity of the electron at the previous calculation position, and an initial incremental guess in time. With a determined incremental time, changes in velocity and the next position may be estimated. By default a fourth-order Runge-Kutta method is used, but a linear method may be specified.

The fifth step includes testing for convergence in trajectory path accuracy. Unless the first guess for the incremental time is appropriate, the trajectory path may be inaccurate. For example, if the time increment is too large, the electron may pass through an area of highly varying electric field without being influenced. To correct this, a second trajectory path is calculated with the same input parameters except a finer first guess for incremental time. The two trajectories are then compared, and if the path is found inaccurate to a specified error percentage, a new trajectory path is calculated with an even finer initial guess. Essentially, the Virtual-FED simulation returns back to the previous procedure item, whenever the path is found to be inaccurate.

Next, the trajectory data path is mapped into the `Octree`. While trajectories are being

computed, the `Octree` functions as a field lookup table; after calculations, the `Octree` operates as a tabulation, mapping trajectory paths and total electron energies. In the seventh and final item, the decision of calculating a newly specified electron trajectory will return the the procedural stage to the third processing detail: using the Monte-Carlo methods to generate a new initial position and velocity vector.

3.2 Virtual-FED Implementation Details

This section presents an overview of the specific details for implementating the Virtual-FED simulation. The simulation uses several special data constructs and various calculation methods to perform its functions efficiently. The program was implemented under several design considerations. Data structures were devised with both high functionality and performance in mind. The data structure for storing the field values operates as a fast and efficient lookup table. Due to the varying granularity of the fields defined by the FED device settings, the data structures are dynamically allocated when necessary for the efficient usage of memory. Likewise, the data structure for calculating trajectories is also dynamically allocated, so that a desired accuracy of the electron trajectory path may be computed effectively. The following subsections deliver a summary of these special constructs and some of the simulation methods used. First, a complete description of each special data construct and its function is given. Finally, the implemented computational methods are explored in detail.

3.2.1 Special Constructs

As suggested previously, the simulator defines two major classes of data structures in order to perform rapidly and efficiently. One set of structures, the `Octree`, is used solely as a table for the storage of field values and trajectory path data. The other data structure, the `Trajectory`, is used to determine the actual trajectory path of electrons. Special variables have been defined, not very elegantly, in order to keep these two data structures abstracted from one another. For all pratical purposes, these special variables do not effect the program's functionality; however, they do place a few restrictions upon the simulation at its current stage. This is further discussed in the following Section 3.3: Limitations and Restrictions.

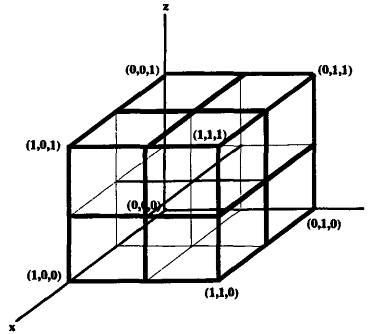


Figure 3-2: Top `Octree` `Box`

The `Octree` Data Structure

The `Octree` data structure stores field values and calculated trajectory path data within the FED device surroundings. The geometry of the environment is assumed to be a cube or rectangular prism. This volume is then mapped into an `Octree` structure. The entire capacity of the box like structure, may be represented as the top most box in an `Octree` structure. This top box may be divided into eight equally sized boxes (See figure 3-2). Each of these eight subboxes may further be divided eight times, for a total of 64 boxes defining the FED device.

Essentially, the '`Box`' element in an `Octree` is the basic building structure supporting fields and additional data. Other structures are defined inside the `Box` structure to help organize the data. The C programming language declaration for a `Box` is shown in Figure 3-3. Also the C declarations for the supporting data structures are listed in Figure 3-4. Below is an inventory of data components for the `Box` and a description of their functions:

- `up` — the pointer to the box on the upper level. This field, or component of the `Box`, in conjunction with the subboxes field define the basis for a double-linked tree structure with eight components, or an `Octree`.

- `subboxes` — an array of eight pointers to boxes at a deeper level. A particular subbox is reference in bit-ordered system, where subboxes$[x_1, x_2, x_3]$ is as defined in Figure 3-2. For example, the subbox positioned at $(1, 1, 0)$ within the top box, can be referenced by subboxes$[6]$.

- `fd` — an array of eight pointers to the structure that defines an analytical field. Each of these eight fields represent the exact analytical field at one of the eight

```
struct box{
  struct box * up;           /* pointers to upper level box */
  struct box * subboxes[8];  /* pointers to subboxes */
  struct analytical_field *fd[8];  /* field at box */
  struct data * dat; /* electron velocities through octree */
  int depth;         /* depth of current box */
  int num_terminated;/* number of terminated subboxes */
  Vector coordinate; /* coordindate of lower left corner of box */
  int isboundary;    /* boolean, denotes if boundary is inside octree */
  int tag;           /* integer tag which denotes type of boundary */
typedef struct box Box;
typedef struct box * BoxPtr;
```

Figure 3-3: Box Data Structure

corners of the Box. It is referenced in the same manner as the subboxes.

- dat — pointer to the structure, storing information of the electron trajectory path and total electron energy through the Box.

- depth — the depth of the current box. The depth of the top most box in an Octree is one. This depth increments with each level of subboxes.

- num_terminated — the number terminated, or NULL, subboxes from this Box.

- coordinate — the lower left corner position of the Box with respect to the unit-normalized Octree. That is, the top most Box is assumed to have dimensions of unity, and a bottom left hand corner located at the origin.

- isboundary — a boolean that denotes whether a boundary is inside the Box.

- tag — a label, designating a specific type of boundary. The tag may represent a specific surface, such as the emitter or gate surfaces in a FED device environment.

Using boxes as building blocks, Octree is an all inclusive structure, caching fields and trajectory path data. Its primary purpose is to provide an effective field lookup interface to the trajectory calculation functions, mapping positions and scaling the boxes into actual locations within the FED environment. The C programming language declaration for an

```
struct vector{
  double x;                   /* x component */
  double y;                   /* y component */
  double z;                   /* z component */
};
typedef struct vector Vector;

struct analytical_field{
  struct vector e;            /* Electric field */
  struct vector b;            /* Magnetic field */
};
typedef struct analytical_field Afield;

struct data{
  int num;          /* number of electrons passing through box */
  double energy;    /* total energy of passing electron */
};
typedef struct data Data;
typedef struct data * DataPtr;
```

Figure 3-4: Support Structures for `Box`

`Octree` and one supporting data structure is shown in Figure 3-5. The `Octree` fields are summarized in the following:

- `top` — the pointer to the top most box. This box has unit dimensions, a `Box` coordinate of the origin.
- `count` — total number of boxes in the `Octree`.
- `depth` — largest depth in the `Octree`.
- `axis` — pointer to the structure with values for mapping positions within the `Octree` into actual positions.
- `isboundary` — a boolean that denotes whether a boundary is inside the `Octree`.

While analyzing a particular FED device, the boxes of an `Octree` representing areas of highly varying fields may be refined to deeper levels. Whereas, boxes representing less varying fields do not need to be refined. In this manner, the rapidly changing electric fields in the FED device environment are not underepresented, and coarser electric field areas are

```
struct octree{
  struct box *top; /* ptr to top (largest) box */
  int count;       /* total number of boxes, including all layers */
  int depth;       /* largest depth */
  struct map *axis;/* scale factor for x,y,z axis */
  int isboundary;  /* boolean, denotes whether boundary is inside octree */
};
typedef struct octree Octree;
typedef struct octree * OctreePtr;

struct map{
  Vector * scale;  /* scale factor for x,y,z axis */
  Vector * offset; /* lower left hand position */
};
typedef struct map Map;
typedef struct map * MapPtr;
```

Figure 3-5: `Octree` Data-Structures

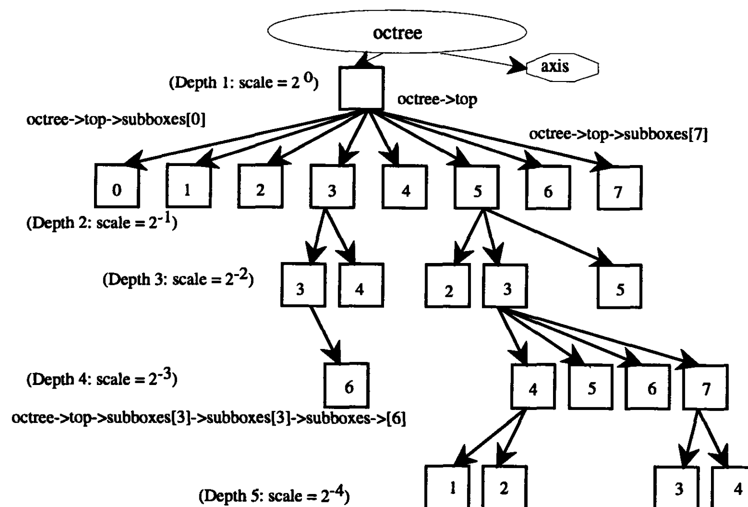


Figure 3-6: Diagram of Simplified `Octree Structure`

```
struct traj_elem{
  struct traj_elem * next;    /* pointer to next traj_elem */
  struct traj_elem * prev;    /* pointer to prev traj_elem */
  struct vector pos;          /* postion vector of particle */
  struct vector vel;          /* velocity vector of particle */
  double t;                   /* total time elapsed */
  double dt;                  /* increment in time */
};
typedef struct traj_elem Listelem;
typedef struct traj_elem * ListelemPtr;
typedef struct traj_elem * List;

struct trajectory{
  int num;                    /* total number of traj_elem's */
  struct traj_elem * front;   /* pointer to first elem of list*/
  struct traj_elem * rear;    /* pointer to last elem of list*/
  struct traj_elem * arry;    /* arry of trajectory elements */
};
typedef struct trajectory Trajectory;
typedef struct trajectory * TrajectoryPtr;
```

Figure 3-7: `Trajectory` Data-Structures

not needlessly over-represented, such that the octree provides a fast and efficient lookup table for fields. (For example, see Figure 3-6)

`Trajectory` Data-Structures

The second main data structure, or the `Trajectory` structure, calculates trajectory paths using field values stored within the `Octree`. The basic building block for the `Trajectory` is a `traj_elem`, or `Listelem`. It operates as a double-linked list of `Listelems`, basically functioning as a queue. The C programming language declaration for a `Trajectory` and its supporting structure is shown in Figure 3-7. Below is a list of data components of the `Listelem` and a brief description of functionality. A simple conceptual diagram is given in Figure 3-8.

- `next` and `prev` — pointers to other `Listelems`. These fields are used to form a double-linked list of `Listelems`, creating a queue.

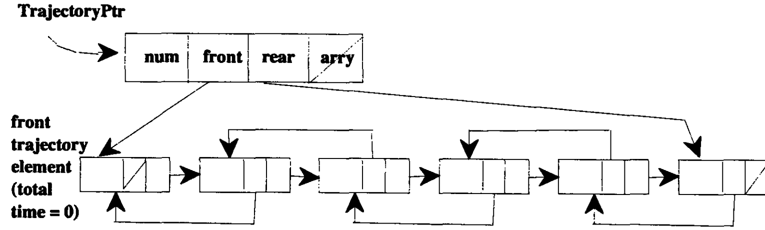


Figure 3-8: Diagram of `Trajectory Structure`

- `pos` — electron vector position with respect to the origin inside the FED environment.
- `vel` — electron vector velocity at given instance of time.
- `t` — total time elapsed, upon *entering* the current vector position.
- `dt` — increment in time, used to calculate the next trajectory item. Hence the total time for the next trajectory element, is $t + dt$.

3.2.2 Octree Field Estimation

For reasons of precision, the fields stored inside an `Octree` box must be representative of the field for the entire box. The `Box` structure within an `Octree` contains field values for each of its eight verticies. When field lookup is specified for a point within the volume of the box, a three-dimensional, linear interpolation of the fields is returned.

3.2.3 Trajectory Convergence Implementation

The first calculation of the trajectory path may be inaccurate, unless the original initial guess in incremental time is appropriate. In order to test for accuracy, a second trajectory path is calculated with the same input parameters with the exception of a finer initial guess for incremental time. The two trajectories are then compared by sampling both trajectories at the same instances in time, and identifying the maximum length of vector difference between two positional points. This maximum length represents the largest interval of error between the two trajectories. This is given by Equation 3.1. Next, the maximum length is normalized against the dimensions of the FED structure in order to ensure that all three axes have equal weight, and weighed with the maximum allowable percentage of

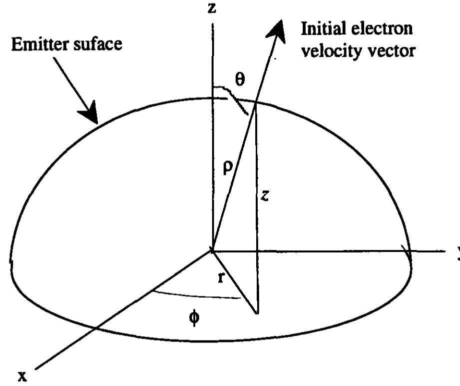


Figure 3-9: Semi-Circular Cone Emitter

error specified by the user.

$$\epsilon = max\{\mid \overline{a_n} - \overline{b_n} \mid\} = max\{\mid \sqrt{\sum_{i=1}^{3}(a_{n_i} - b_{n_i})^2} \mid\} \tag{3.1}$$

where n represent the sample number.

3.2.4 Monte Carlo Implentation

The Monte-Carlo method is used to initialize the velocity and position vectors for an electron trajectory, and it generates either a uniform or normal gaussian distribution of initial velocity at emitter tip surface. The initial position for the electron trajectory is located within an angle, θ, away from the z-axis, along the surface of the emitter. The original electron velocity vector is normal to the suface of the initial position (See figure 3-9). For the purposes of this work, the distribution in ϕ was assumed to be uniform around the emitter. The implementation of the monte-carlo methodology for a uniform distribution is a trivial matter. However, for a gaussian distribution, it is more complicated. The normal gaussian probability density function defined by:

$$f_\theta(\theta) = (2\pi\sigma^2)^{-1/2} \exp -\frac{(\theta - m)^2}{2\sigma^2} \tag{3.2}$$

The desired gaussian density fuction is known. Hence, a form of Bayes's theorem may be

used to determine whether to make a trajectory call at a particular (randomly generated) θ [14]. Bayes's theorem states: given a known gaussian probability density function for θ and $f_\theta(\theta)$, then the probability of event A (in this case, the event that the trajectory with an initial velocity vector directed θ should be calculated), given that the random variable θ took on the value θ is:

$$Pr(A|\theta = \theta) = \frac{f_\theta(\theta|A)Pr(A)}{f_\theta(\theta)} \tag{3.3}$$

3.3 Limitations and Requirements

With infinite memory, there are no limitations on the number of Trajectory structures or Octree boxes that may be generated. However, the simulations are restricted based upon computer memory limitations. In particular, boxes within the `Octree` data structure, require much memory in order to map the FED environment. This is true, even with the feature of selective refining inside the `Octree`. In fact, one `Box` structure claims 504 bytes of memory, and for any simulation result described in Chapter 4, 86,665 `Octree` boxes are necessary in order to accurately map the FED device surroundings. Thus, over 43.6 MBytes are necessary to map the `Octree` structure alone for those graphs.

As discussed earlier, special variables have been defined to keep these the `Octree` and `Trajectory` data structures abstracted from one another. The restrictions to the program is that fields cannot be recalculated anaytically, while using the same `Octree` as a lookup table. Also at any one period in time, only one `Octree` structure may be in existence. It may pose a problem when FEAs are to be implemented, since more than one `Octree` structure may need to be referenced at any one time. Nonetheless, the current implementation can be easily altered if necessary.

3.4 Virtual-FED Usage

The simulation code is intended to serve as a library in the C programming language for calculating various FED charateristics, such as the spot size of on the screen. Most of the procedures for calculating uniform and gaussian distributions are written. See the Appendicies for the Virtual-FED program and documentation. The user needs only to follow a simple procedure, common to many programs:

1. Initialize variables — device geometry values, fields, and `Octree` refinement.
2. Perform desired calculations
3. Output results
4. Free variables

Parameter Initialization

Before any trajectory calculations can be made, the user must initialize several data structures and other variable parameters. The data structures for these parameters are listed in Figure 3-10. These structures are described briefly below:

- `Fields` — field structure which has two components. One which contains an analytical field expression, or the field value itself. The other is a pointer to the `Octree`. The data structure of the analytical field was given in Figure 3-4.
- `Initiale` — the initial electron data structure, contains information such as the electron charge, mass, initial position before trajectory calculation, and initial velocity before trajectory calculation.
- `Bounds` — the boundaries for simulated volume are stored here.

In addition, the user must refine areas of interest in the FED device. There are a number of functions, defined in the Virtual-FED utility, which specializes in refining the octree. These function prototypes are listed below as well as in 'Appendix C: octree.m.'

- `refine` — given a depth, or number of levels to refine, this function refines the entire FED device volume to the number specified by depth.
- `refine2perc` — refines the entire FED device volume to the level of boxes, having a common parent box, with a specified percentage of variation between their fields. Note, that this function may overlook highly varying field points or singularities in the volume.
- `refine2point` — given a depth, this function refines to a specified position in the volume.

```
struct field_struct{ struct analytical_field af; /* analytical fields
  */ struct octree * oct; /* fields stored on an octree */ }; typedef
  struct field_struct Fields;

struct initiale{
  double q;            /* charge of particle */
  double m;            /* mass of particle */
  struct vector pos;   /* intial postion vector of particle */
  struct vector vel;   /* intial velocity vector of particle */
};
typedef struct initiale Initiale;

struct bounds{
  double xmax;  /* maximum x value of cube */
  double xmin;  /* minimum x value of cube */
  double ymax;  /* maximum y value of cube */
  double ymin;  /* minimum y value of cube */
  double zmax;  /* maximum z value of cube */
  double zmin;  /* minimum z value of cube */
};
typedef struct bounds Bounds;
```

Figure 3-10: Virtual-FED: Structures User Must Initialize

- `refine2pointperc` — given a percentage for the variation of field values, this function refines the `Octree` at a specified position to a level where neighboring subboxes fall within the percentage of field variation.
- `refinebds` — refines the boundaries of a box at specified incremental values.
- `refinetopbds` — refines uniformly at the top of the `Octree`, where the phosphor screen is located.
- `refinesemi` — refines a semi-circle above the x-y plane. This semi-circle is centered about the origin.

Perform Calculations

There are a few higher-order functions which specialize in generating electron trajectories. Their prototypes are listed 'Appendix A: trajectory.h.' A brief description of each are given in bullet form below:

- `transit` — calculates a single electron trajectory.
- `uniform_dist` — calculates trajectories with a uniform distribution of initial velocities; the uniform distribution is defined with respect to the range of θ.
- `gaussian_dist` — calculates trajectories with gaussian distribution of initial velocities.

Output Results

There are two functions which print out the results, which are stored in the `Octree` structure. These functions are listed in 'Appendix C: octree.h.' and also described below:

- `print_box_data` — print the data from each and every Octree box.
- `print_top_dat2file` — print data from the screen into a file.

Freeing Variables

After trajectories have been calculated and the results written to file, it is necessary to free certain variables. In particular, the `Octree` and the `Trajectory` data structures need to be freed. The following two procedures will free the `Octree` and the `Trajectory` structures respectively.

- `destroy_octree` — destroys the `Octree` data structure.
- `destroy_trajectory` — destroys the `Trajectory` data structure.

Sample FED Trajectory Program

Figure 3-11 represents a Virtual-FED sample program, which is used to calculate the results for uniform distibution functions, described in the next chapter. For the user, the most tedious procedure can be initializing the input values into the program. In this case, an inputfile could be helpful.

```
/* Virtual-FED Sample Program */
#include "trajectory.m"

void main()
{
  TrajectoryPtr t;  /* trajectory pointer */
  Initiale ch;      /* initial electron information */
  Bounds bd;        /* boundary information */
  Fields f;         /* fields structure */
  Afield af;        /* analytical field structure */
  double maxerror;  /* maximum allowable error between two trajectories */
  double energy;    /* magnitude of inital trajectory energy */
  int num_traj;     /* the desired number of trajectory calls */
  char s[50];    /* filename, cannot be more than 49 characters long */

  /* scan in values for:
   * V_a = anode voltage (in volts)
   * V_g = gate voltage (in volts)
   * energy = initial electron energy (eV)
   * num_traj = number of desired trajectores
   * s = string, or filename to write data into
   */
  scanf(``%lf %lf %lf %d %s'', &V_A, &V_G, &energy, &num_traj, s);

  /* the init_all fuction will prompt the user for various parameters
   * such as the electron information, the boundaries of the environment
   * initial fields, maximum allowable error between similar trajectory
   * paths, a refinement levels within the Octree are also prompted as
   * a user input.
   */
  init_all(&t, &ch, &bd, &f, &af, &maxerror);

  /* this fuction writes into the octree and calculates a uniform
   * distribution of initial velocities normal to the surface of the
   * emitter, with the characteristics of previously discussed variables,
   * within the range of 0.0 < theta < 2.0
   */
  uniform_dist(ch, f, bd, maxerror, energy, num_traj, 2.0);

  /* prints only  data from the top of the screen to a datafile */
  print_top_data2file(oct->top, oct, bd, s);

  destroy_octree(oct); /* free memory */
}
```

Figure 3-11: Virtual-FED Sample Code

Chapter 4

Virtual-FED Simulation Results

Virtual-FED is intended to be a Technology-Computer Aided Design (T-CAD) tool for field emitter display (FED) design and optimization. Amongst the design objectives of the simulation tool is the ability to analyze FED designs and predict important device characteristics such as resolution, luminance, contrast ratio, and power. Virtual-FED, when fully implemented, will be able to answer the following questions:

- What are optimal gate, screen, and focus electrode (if present) operating voltages?
- Given device dimensions and voltage parameters, what is the current density distribution (spot size) on the phosphor screen?
- How closely should field emitters be spaced in an FED?
- To what extent do focusing electrodes help narrow the electron beam?
- How far vertically from the emitter should the focusing electrode be spaced?
- How far can the distance between the emitter cathode and the screen be extended with the usage of various focusing electrodes?

The focus of this chapter is to demonstate that the Virtual-FED utility has the capability of predicting accurate solutions for FED developers. With Virtual-FED, it is possible to model a device quantitatively before it is physically built. Virtual-FED has the capacity of predicting the relationship between spot size and other design parameters, such as the cone to cone spacing for a FEA and the spacing between the emitter and the screen.

The following sections describe the results for several test cases, which have been generated using Virtual-FED. Specific simulations were chosen to portray certain problematic

issues, such as (*i*) the anode operating voltage, and its effect on self-focusing, (*ii*) gate operating voltage, (*iii*) current distribution, and (*iv*) the initial energy of the electron. Based upon the results, general conclusions can be made for each issue.

4.1 Sample Trajectory Evaluations

Virtual-FED has capability of predicting the relationships between spot sizes and emitter geometry. However, for the purposes of this thesis, test cases were chosen specifically in order to demonstrate the effect of between the anode and gate voltages and the energy and distribution on the spot size. Thus, all generated simulations have common FED device parameters. The distance between the base plate, or the emitter, and the face plate, or the phosphor screen, is fixed at $d = 1mm$. An emitter tip radius (r_o) of $0.05\mu m$ and a gate aperature radius (R) of $0.5\mu m$ from the center of the emitter tip, is assumed for all simulations.

In addition, the simulations have a specific distribution of θ, or the angle from the emitter axis. The values for θ has been fixed between the range of $0° \leq \theta \leq 2°$. This narrow range of values was chosen because the majority of electrons which hit the screen were found to have an initial θ of less than one degree. Figure 4-1 shows how the range of theta may affect the results. In Figure 4-1 (*a*), (*c*), and (*e*), θ ranges uniformly from $0° \leq \theta \leq 2°$; in graphs (*b*), (*d*), and (*f*), θ ranges uniformly from $0° \leq \theta \leq 10°$. All plots were generated with a $2000V$ anode voltage (V_A), a $100V$ gate voltage (V_G), and an initial electron energy (E) of $0.1eV$ with uniform distribution about the emitter axis (U). Note for these graphs, that the gaussian distribution is denoted with a (G) and its variance. These parameter values are labelled above each plot. In addition, all plots have the same number of calculated trajectories; however, the number of electron 'hits', or collisions against the screen, within the $20\mu m \times 20\mu m$ dimensions of the screen differ dramatically. The number of these electron 'hits' are labelled individually below each plot.

Figure 4-1 (*a*) and (*b*) depict the distribution of electrons on the phosphor screen. Each point represents a box position in the Octree, along the surface of the screen, which contains ***at least*** one electron hit. Hence, the graphs represent the spatial distribution of electrons, with a viewing perspective of looking directly at the screen. The axes for both of these plots are the actual dimensions along the screen. The origin of the x-y plane represents the

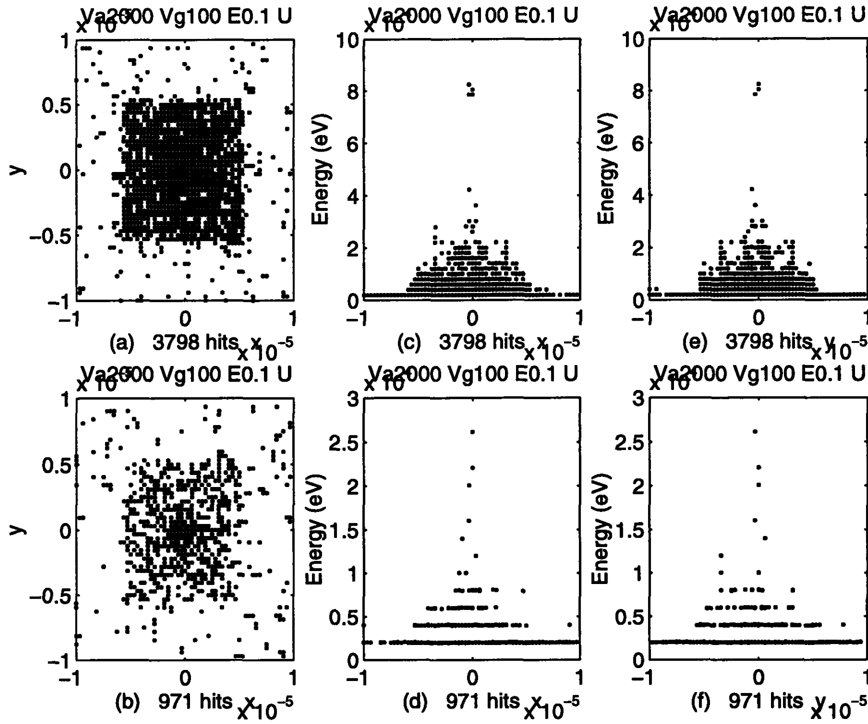


Figure 4-1: Range of theta: $V_A = 2000V$, $V_G = 100V$, $E = 0.1eV$

center location of the emitter tip. The x-axis ranges from $-10\mu m \leq x \leq 10\mu m$, which is depicted on the plots as $(-1 \leq x \leq 1) \times 10^{-5} meters$. The y-axis is simularly pictured in this fashion, ranging from $-10\mu m \leq x \leq 10\mu m$ as well.

Figures 4-1 (c) through (f) represent the summation of electron energies at the screen. Figures 4-1 (c) and (d) denote the energy distribution across the range of x. For these two plots, the x-axis describes horizontal locations on the surface of the screen for the ranges $-10\mu m \leq x \leq 10\mu m$, which is listed on the graph as $(-1 \leq x \leq 1) \times 10^{-5} meters$. The y-axis represents the magnitude of electron energies in eV. The multiple data points at each x location portray the energy values across a sweep of y, while holding x constant. Simliarly, Figures 4-1 (e) and (f) denote the energy distribution across the range of y, which is the vertical location component on the phosphor screen. As expected, the graphs of Figure 4-1 (e) and (f) are similar to (c) and (d) respectively, since these simulations are assumed to be symetrical about the emitter axis.

Together, Figures 4-1 (a), (c), and (e) paint a three dimensional picture of the energy density against the plane of the screen for a uniform distribution of $0° \leq \theta \leq 2°$. Likewise, Figures 4-1 (b), (d), and (f) describe a separate three dimensional graph for the distribution

about $0° \leq \theta \leq 10°$. From Figures 4-1 (a) and (b), one can see that the spatial distribution of electrons is gathered around the origin for both plots. However, in plot (a) the electrons are more densely concentrated at the center, than in plot (b). Figures 4-1 (c) and (d) both show the number of electrons which hit the screen is centered about $x = 0$. While Figures 4-1 (e) and (f) show that the number of electrons, hitting the screen, is also centered about $y = 0$. Altogether, these graphs conclude that the magnitude of electrons, which strike the screen, tends to peak within a small area about origin. By noticing the difference in the scaling of Figures 4-1 (c) and (e) (which is 10×10^4) vs. (d) and (f) (which is 3×10^4), one can also see that the number of electrons which hit the screen is much greater in the case where the distribution is $0° \leq \theta \leq 2°$. In fact, the uniform distribution of $0° \leq \theta \leq 2°$ produces 3798 electron hits out of 5000, or a 75.96% hit ratio. For the distribution of $0° \leq \theta \leq 10°$, only 971 electrons out of 5000 hit the screen, giving a 19.42% hit ratio.

One last inference that can be made with these plots, based upon the anode voltage (V_A). Since $V_A = 2000V$ is used for the set of simulations in Figure 4-1, the energy for a single electron when it hits the screen is approximately $2000eV$. Knowing this, one can deduce from Figure 4-1 (c) and (d) that at $x = 0$ and $y = 0$, the number of electrons is approximately $(8.2 \times 10^4 eV)/2000V$ or 41 electrons hit the screen at the origin. Correspondingly, the number of electrons at the origin for the wider range of θ is $(3 \times 10^4)/2000eV$ or 15 electrons (See Figures 4-1 (d) and (f)).

The purpose of this example is to demonstate how to interpret the data plots as well as exhibiting the influences of using various ranges of θ. Because the distribution of θ from $0° \leq \theta \leq 2°$ gives more information, all the following test cases take on that range. In addition, all screen dimensions are fixed at $-10\mu m \leq x \leq 10\mu m$. Again, this is listed on the graphs as $(-1 \leq x \leq 1) \times 10^{-5} meters$.

A summary of the input parameters to Virtual-FED simulations is listed in Table 4.1. A 'Yes' in the column labelled 'Fixed' denotes that those values are common in all simuations. Throughout the next four sections, the results of these test simulations have been arranged in order to answer questions regarding the anode and gate voltage, and how they influence the spot size and the energy distribution upon the screen. First, issues concerning the anode operating voltage and its effect on self-focusing is explored. Second, the results from varying gate voltages is used to determine its role in influencing the spot size. Next, gaussian and uniform current distributions are compared. Finally, the effects of initial energy is analyzed.

Table 4.1: Parameter Values for Virtual-FED Test Cases

Parameter	Value(s)	Fixed
V_A	$20V$, $100V$, $200V$, $1000V$, $2000V$	No No
V_G	$50V$ or $100V$	No
E initial energy	$0.1eV$ or $1eV$	No
Distribution	uniform or gaussian	No
d emitter/screen dist.	$1mm$	Yes
r_o emitter radius	$0.05\mu m$	Yes
R gate radius	$0.5\mu m$	Yes
d emitter/screen dist.	$1mm$	Yes
θ	$0° \leq \theta \leq 2.0°$	Yes
Number of Calculated trajectories	5000	Yes
Screen dimensions	$-10\mu m \leq x \leq 10\mu m$ $-10\mu m \leq y \leq 10\mu m$	Yes Yes

Table 4.2: Test Parameters: Self-Focusing Examples

	(a)	(b)	(c)	(d)
V_A	$20V$	$200V$	$1000V$	$2000V$
V_G	$100V$	$100V$	$100V$	$100V$
E	$0.1eV$	$0.1eV$	$0.1eV$	$0.1eV$
Distribution	Uniform	Uniform	Uniform	Uniform

4.1.1 Electron Self-Focusing

Four examples are used to show the effect of the anode voltage upon spot size and the energy density distribution on the screen, as a demonstration of the electrons self-focusing ability in higher anode or screen electric fields. The gate voltage, the initial electron, energy, and the type of distribution are all fixed at specifed values. In this case, only the anode voltages are varied. Table 4.2 provides a summary of the parameter values.

Figure 4-2 depicts the spatial electron distribution, or the spot size, for the uniformly distributed simulations. Similar to previous examples, the top of every graph is labelled with its anode and gate voltages, the initial energy, and its distribution. In Figure 4-2 (a), $V_A = 20V$, (b), $V_A = 200V$, (c), $V_A = 1000V$, and (d), $V_A = 2000V$; so that V_A is growing with each successive example. It is apparent that the electrons are self focusing with higher

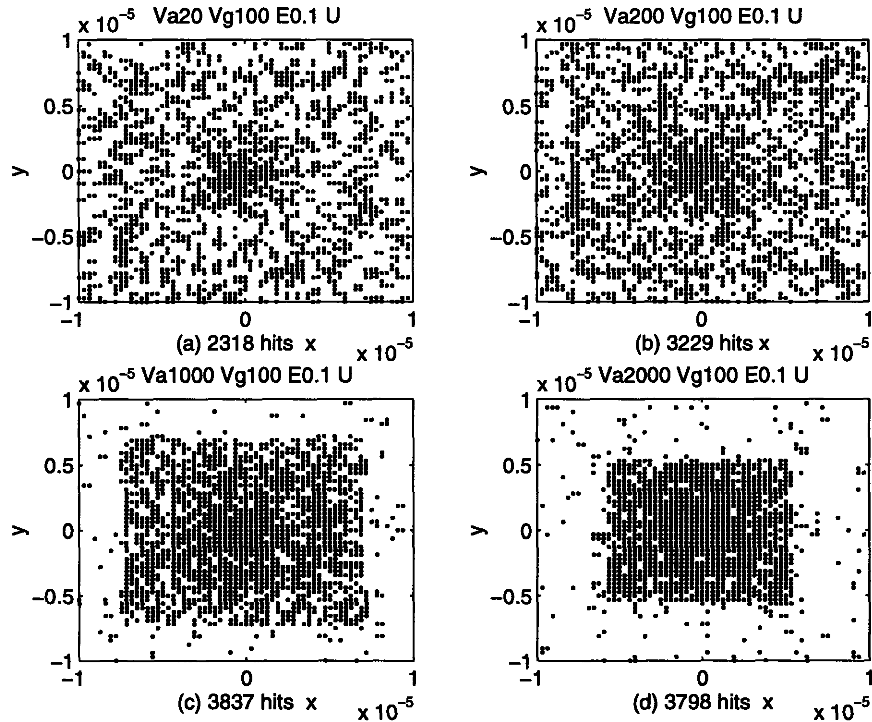


Figure 4-2: Self Focusing: Distribution of Electrons Positions

anode voltage. In all four plots, the spatial distribution of electrons striking the screen is most densely concentrated about the origin. However, this cluster of electrons becomes more narrow and compact at higher anode voltages. (See Figure 4-2 (c) and (d))

Figure 4-3 gives energy distribution, or the summation of electron energies which strike the screen, which is complement to Figure 4-2. Note that only the energy distribution in the x direction is shown here, but the results are similar in the y direction. Thus, one can assume that the energy is symmetrical in both x and y directions, and that Figures 4-3 and 4-2 provide the necessary information to describe three-dimensional images of the electron energy values (the z-parameter) plotted against the spot size image (on the x-y plane). The magnitude of energies increases with increasing V_A. In Figure 4-3 (a), $V_A = 20V$ and the energies range from 0 to approximately $180eV$. This suggests that about $180eV/20V$ or 19 electrons land at the center of the screen, which is also lined at the center of the emitter tip. In plot (b) $V_A = 200V$, electron energies reach up to about $4200eV$; hence 21 electrons strike the x-y origin. Likewise in plot (c) $V_A = 1000V$, electron energies increase up to about $37000eV$; and an estimated 37 electrons strike the origin. For plot (d) $V_A = 2000V$, electron energies range to about $82000eV$ for an estimated 41 electrons. With the rise in

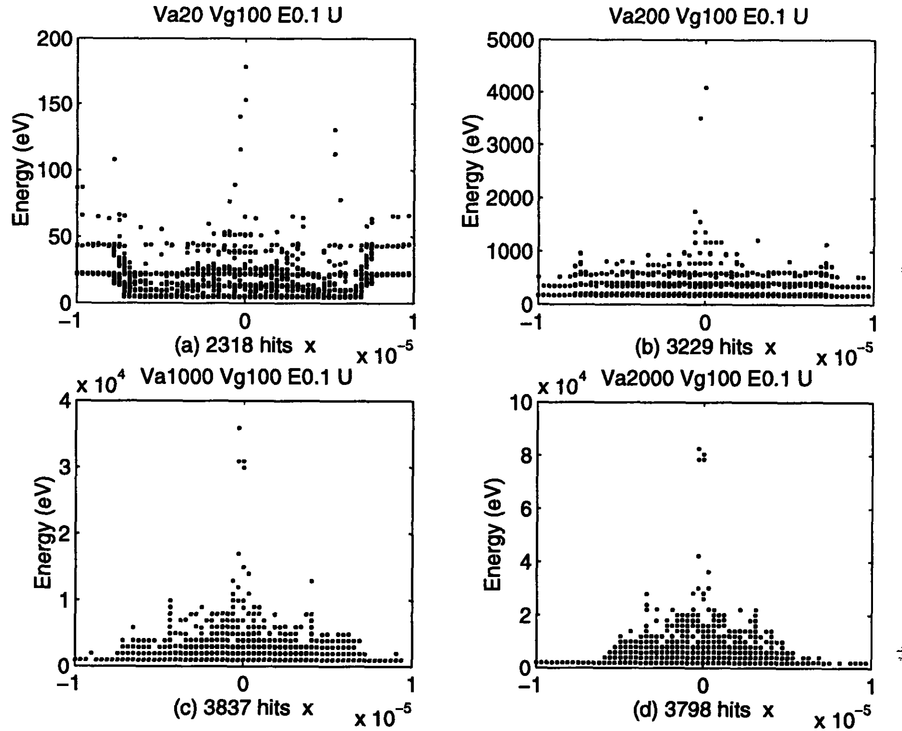


Figure 4-3: Self Focusing: Energy Distribution

V_A, the increase of electrons at the origin is consistent with self-focusing effects.

In addition, the maximum lateral extentent of energy concentration with respect from the cathode to the screen, becomes tighter at higher anode voltages. From Figures 4-3 (a) and (b), where the anode voltages are $20V$ and $200V$ respectively, the region of the phosphor screen with the larger concentration of electrons is greater than the width of the screen area of interest. Figures 4-3 (c) and (d), where the anode voltages are $1000V$ and $2000V$, the maximum lateral distance of electron concentration decreases to approximately $-7\mu m \leq x \leq 7\mu m$ and $-5.5\mu m \leq x \leq 5.5\mu m$ respectively. The number of electron 'hits' also tends to increase with a smaller maximum lateral extent of energy concentration although this is not always the case. The smaller spot size and increased counts of electron hits demonstrate higher screen resolution when the anode voltage is increased.

These results are consistent with the expected since the lateral distance, traversed by the electrons from the cathode to the screen, should decrease as V_A increases. In other words, as V_A increases, the electric field in the region between the emitter and screen increases. Hence, the electrons tend to become redirected onto the screen by means of the electric field.

Table 4.3: Test Parameters: Effect of Gate Voltage

	(a)	(b)	(c)	(d)
V_A	$2000V$	$400V$	$2000V$	$400V$
V_G	$50V$	$50V$	$100V$	$100V$
E	$0.1eV$	$0.1eV$	$0.1eV$	$0.1eV$
Distribution	Uniform	Uniform	Uniform	Uniform

The analyical espression for the estimated lateral movement of an electron, from the cathode to the screen, is given by:

$$\delta = 2 \times \frac{(V_G)^{1/2}}{(V_S - V_G)} \times \sin\theta[(V_S - V_G \sin^2\theta)^{1/2} - (V_G)^{1/2} \times \cos\theta]d \tag{4.1}$$

where d is the maximum distance from the gate aperature to the phosphor screen [15]. With $\theta = 2°$ this equation gives:

- (a) For $V_A = 20V$, $\delta = 47.30\mu m$
- (b) For $V_A = 200V$, $\delta = 28.92\mu m$
- (c) For $V_A = 1000V$, $\delta = 16.772\mu m$
- (d) For $V_A = 2000V$, $\delta = 12.758\mu m$

These lateral distances, calculated from the analytical formula, are found to be shorter than those in Figure 4-3. However, the trend, of decreasing lateral distances with increasing anode voltages, is consistent with the progression of the plots. Hence, a general conclusion can be made through the inspection of the plots in Figure 4-2 and Figure 4-3 that the anode voltage should be increased for further self-focusing, or the tightening of spot size and the increase in resolution.

4.1.2 The Effects of Gate Voltages

Four examples are used to show the effect of the gate voltage upon spot size and the energy density distribution on the screen. In this example, the initial electron energy and distribution are held fixed, while the anode and gate voltages are varied. Table 4.3 provides a summary of the parameters. The plots in Figure 4-4 represent the spot sizes produced by

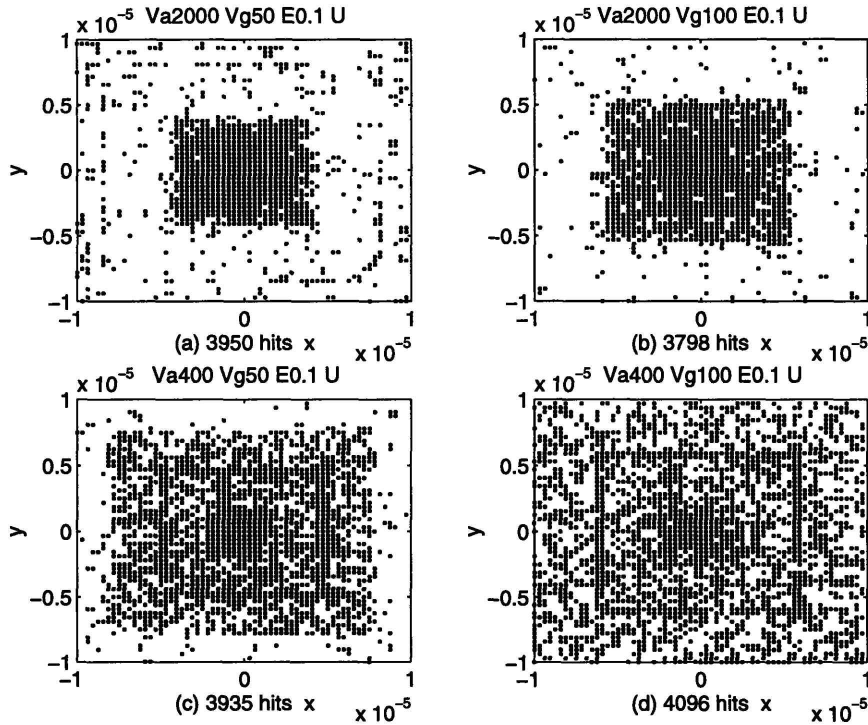


Figure 4-4: Gate and Anode Voltages: Distribution of Electron

varying gate and anode voltages. From Figure 4-4 (a) and (c), one can see an increase in self-focusing effects with the higher anode voltage V_A and a constant gate voltage V_G. This is consistent with the results discussed in the previous section.

Figures 4-4 (a) and (b) both have the same anode voltage $V_A = 2000V$ but varying gate voltages of $V_G = 50V$ and $V_G = 100V$ respectively. In this case, the spot size actually increases with increasing V_G. Similarly, Figures 4-4 (c) and (d) represent a pair of simulations which have a lower anode voltage at $V_A = 400V$, but varying gate voltages of $V_G = 50V$ and $100V$. Again, the spot size becomes less focused when V_G is increased from $50V$ to $100V$.

Figure 4-5 portrays the energy distribution, or the summation of electron energies at the screen. This graph is the complementary of Figure 4-4. As in the previous section regarding self-focusing, only the energy distribution in the x direction is shown, but one can assume that the energy is symmetrical in both x and y directions. The magnitude of energies decreases with increasing V_G. Figure 4-5 (a), which has $V_A = 4000V$ and $V_G = 50V$, shows a maximum energy at about $11600eV$ and approximately $116000eV/2000V$ or 58 electrons at the center of the screen. Plot (b), which has $V_A = 4000V$ and $V_G = 100V$

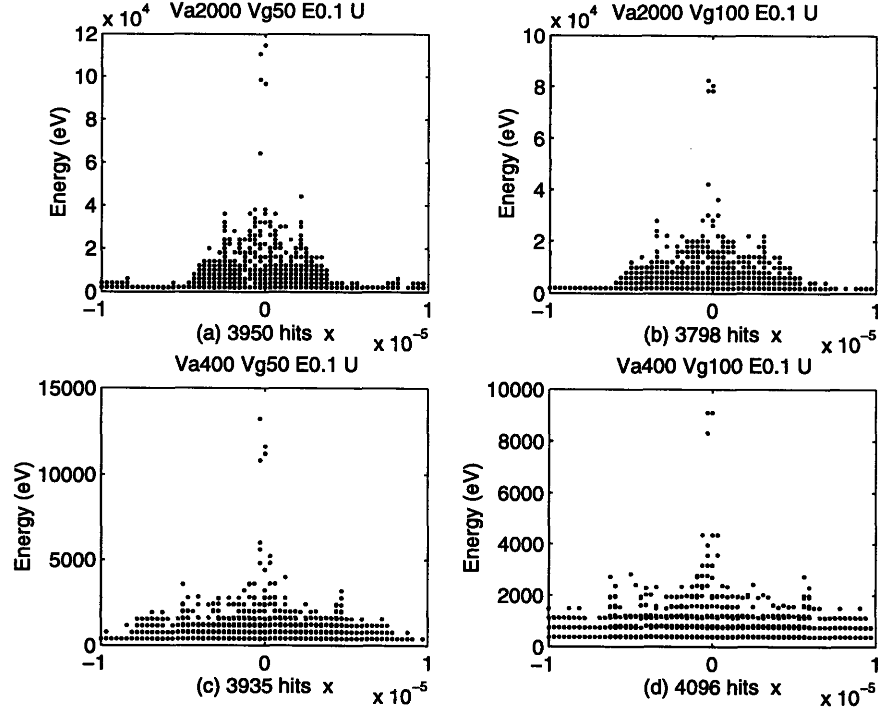


Figure 4-5: Gate and Anode Voltages: Energy Distribution

depicts approximately 41 electrons at the origin. These results are comparable with the observations from Figure 4-4, showing an tightening in spot size with a decrease in gate voltage. Likewise, in plot (c) $V_A = 400V$ and $V_G = 50V$, about $13600eV/400V$ or 34 electrons are focused at the center. Plot (d), where $V_A = 400V$ and $V_G = 50V$, holds approximately $9200eV/400V$ or 23 electrons at the origin. Thus, these last two plots are consistent with the trend shown in plots (a) and (b).

Also, one can determine visually that the distribution of energy deposited on the phosphor screen is narrower in Figure 4-5 (a) than in plot (b). This too is apparent between the plots (c) and (d). At larger gate voltages, the increased width of the lateral distances traversed by electrons reinforces the conclusions gained through Figure 4-4 and the discussion of energy concentration at the origin of the screen.

In summary, spot sizes were found to increase with increasing V_G, as demonstrated in Figures 4-4 and 4-5. This is consistent with the fact that the gate voltage causes electrons which are emitted at an angle to the normal to have a lateral trajectory.

By again using Equation 4.1 and $\theta = 2°$, the analytically calculated lateral distances are:

- (*a*) For $V_A = 2000V$ and $V_G = 50V$, $\delta = 7.019\mu m$
- (*b*) For $V_A = 400V$ and $V_G = 50V$, $\delta = 18.235\mu m$
- (*c*) For $V_A = 2000V$ and $V_G = 100V$, $\delta = 9.530\mu m$
- (*d*) For $V_A = 400V$ and $V_G = 100V$, $\delta = 37.996\mu m$

These lateral distances, calculated from the analytical formula, are slightly larger than those in Figure 4-5. For example, in Figure 4-5 (*a*) the simulated lateral distance is roughly $4\mu m$; however, the analytical lateral distance is $7.019\mu m$. One may still observe that the trend of increasing lateral distances in conjunction with increasing gate voltages, is consistent with the plots. The lateral distances, observed in the figures, are in proper proportion with the analytical distances.

From the past four figures (Figures 4-2 through Figure 4-5), one may infer that FEDs need to have low V_G and high V_A in order to have the smallest spot size and highest resolution. Determining FED geometries and parameters which increase resolution and decrease spot size is a subject for future work in FED development. In addition, FEA environments may be simulated by superpositioning the results of one field emitter.

4.1.3 Distribution Effects

In this thesis, the distribution of electron initial direction is studied. The motivation for implementing this ability to account for various distributions is derived from the real life FED scenario, where the initial velocities are highly peaked around the center of the emitter. In addition, the electric field varies with angle about the emitter axis also. Two distributions are used: (*a*) the uniform distribution and (*b*) the normal gaussian distribution. The study of the distribution of the initial electron direction examines two physical phenomena:

- the non-normal (or tangential) component of initial electron energy (or direction) with respect to the emitter surface.
- the observed decrease of surface electric field away from the tip center in cone-emitter simulations.

In both cases, the physical phenomena can be represnted with a distribution of the initial electron energy. Two types of distributions are assumed – the uniform and the normal gaussian.

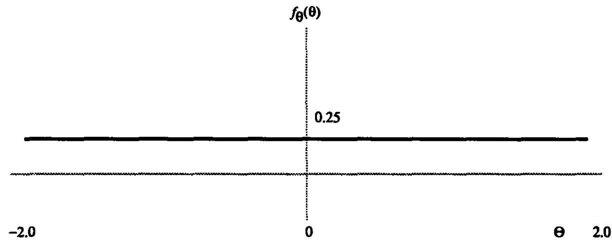


Figure 4-6: Uniform Probability Density Function $f_\theta(\theta)$

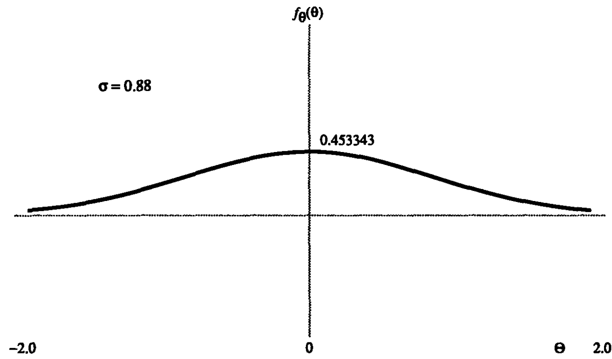


Figure 4-7: Gaussian Probability Density Function $f_\theta(\theta), \sigma_\theta = 0.88$

A total of twelve examples are generated to show the effects of distribution upon screen spot size. For each set of voltage and device parameters, three simulations are compared: one with a uniform distribution, another with a normal gaussian distribution with a variance (σ_θ) of 0.88, and the last with a normal gaussian distribution with a variance of 0.2. The probabililty density functions for the uniform distribution, the gaussian distribution with a σ_θ of 0.88, and the gaussian with a σ_θ of 0.2 are illustrated in Figures 4-6, 4-7, and 4-8 respectively. Table 4.4 provides a summary of voltage and distribution parameters.

Figures 4-9 and 4-10 depict the spot size along the screen for four sets of simulations; where each set has three differently distributed initial velocities. The layout of these plots are consistent with the graphs from previous examples. Each graph is labelled with the anode and gate voltages, the initial energy, and the distribution the top. The first set of test cases, in Figure 4-9 (a), (b), and (c) show the spot sizes for the uniform, gaussian with $\sigma_\theta = 0.88$, and gaussian with $\sigma_\theta = 0.2$ respectively. All three plots have the fixed parameters of $V_A = 200V$, $V_G = 100V$, and $E = 0.1eV$.

One can see that the spot size for Figure 4-9 (a), or the uniform distibution case, is

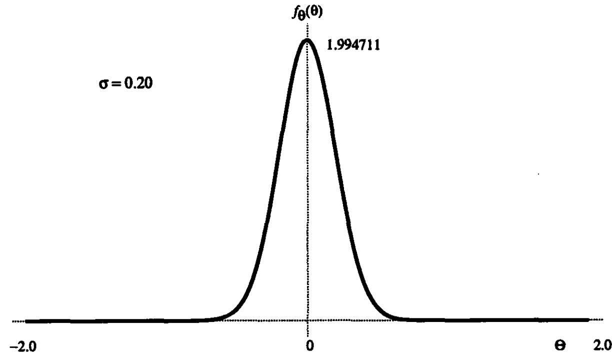


Figure 4-8: Gaussian Probability Density Function $f_\theta(\theta), \sigma_\theta = 0.2$

Table 4.4: Test Parameters: Effect of Initial Velocity Distributions

	(a), (b), (c)	(d), (e), (f)	(g), (h), (i)	(j), (k), (l)
V_A	$200V$	$20V$	$200V$	$1000V$
V_G	$100V$	$100V$	$100V$	$50V$
E	$0.1eV$	$0.1eV$	$1eV$	$0.1eV$
Distribution	Uniform Gauss $\sigma_\theta = 0.88$ Gauss $\sigma_\theta = 0.2$	Uniform Gauss $\sigma_\theta = 0.88$ Gauss $\sigma_\theta = 0.2$	Uniform Gauss $\sigma_\theta = 0.88$ Gauss $\sigma_\theta = 0.2$	Uniform Gauss $\sigma_\theta = 0.88$ Gauss $\sigma_\theta = 0.2$

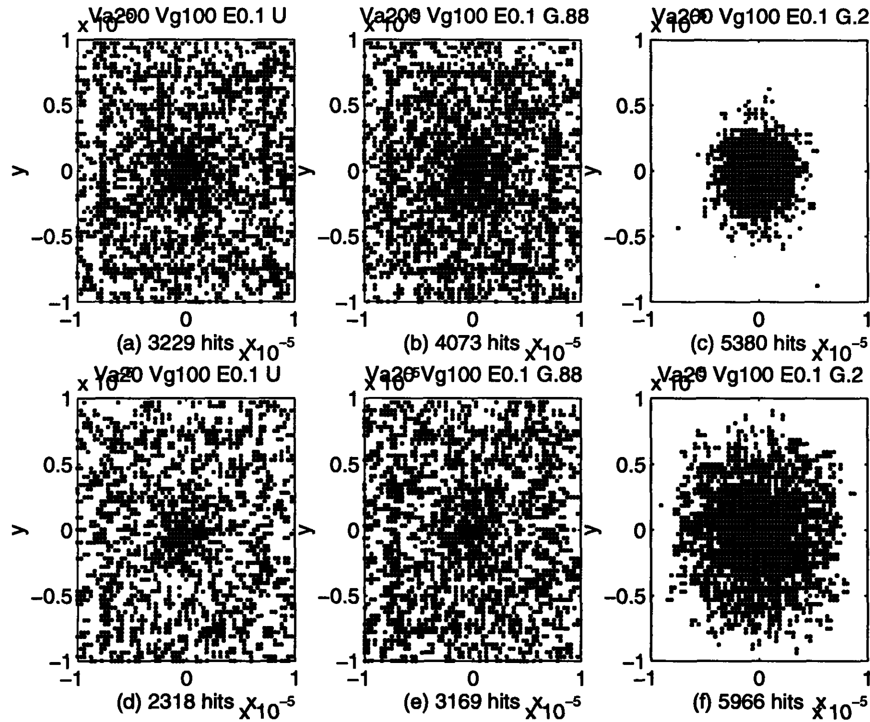


Figure 4-9: Distribution Effects: Distribution of Electron I

slightly less concentrated about the screen's origin as the spot size in plot (b), the gaussian with $\sigma = 0.88$, and much less concentrated than the spot size for plot (c), the gaussian with $\sigma = 0.2$. In addition, the number of hits increases after each case, establishing a higher resolution for each consecutive plot. The decrease in spot size and increase in resolution are expected results for the gaussian distribution of the initial electron velocities, with low variance values. This trend is also apparent in the other sets of examples.

Figure 4-9 (d), (e), and (f) are generated with a lower anode voltage of $V_A = 20V$ than that of plots (a), (b), and (c), where $V_A = 200V$. All other parameters remain the same. Comparision of these two sets of results verify that a higher V_A generates a tighter spot size. The focusing of spot size is most noticeable in Figure 4-9 (c) and (f), where the lateral distances increase from a radius of approximately $3.1\mu m$ for $V_A = 200V$ to $7.0\mu m$ for $V_A = 20V$.

The graphs (g), (h), and (i) in Figure 4-10 are parallel simulations to plots (a), (b) and (c) in Figure 4-9, except the initial electron energy is $E = 1V$ instead of $E = 0.1V$. All additional parameters are the same. These two data sets are similar. Even the number of hits are almost the same. These results are as expected; and the reasoning is to be given in

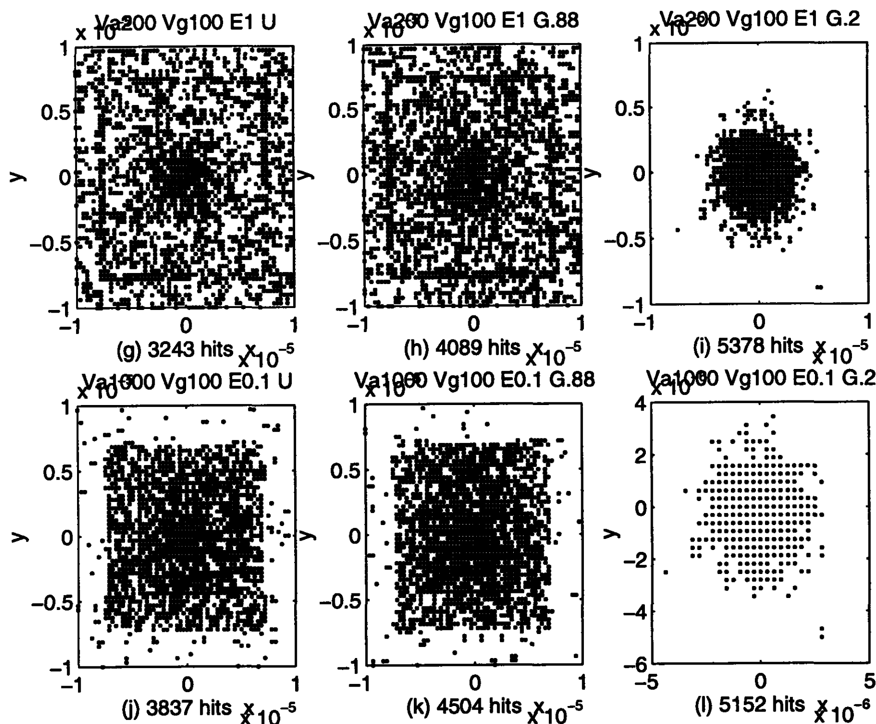


Figure 4-10: Distribution Effects: Distribution of Electron II

the next subsection, entitled 'The Effects of Initial Electron Energy.'

Figure 4-10 (j), (k), and (l) represent the higher anode voltage ($V_A = 1000V$) counterpart to plots (a), (b), and (c) of Figure 4-9 (where $V_A = 200V$). The other parameter values stay fixed. Again, the comparision of these data sets verify that a higher V_A generates a tighter spot size. The variation of spot sizes is most noticeable between plot (c) of Figure 4-9 and plot (l) of Figure 4-10. The lateral distance, traversed by an electron from the emitter to screen, decreases from approximately $3.1\mu m$ for $V_A = 200V$ to $2.8\mu m$ for $V_A = 1000V$. Note the change of scale for the plot (l), where graphing points range from $-5\mu m \leq x \leq 5\mu m$ and $-6\mu m \leq y \leq 4\mu m$, unlike all previous plots of graphing range $-10\mu m \leq x \leq 10\mu m$ and $-10\mu m \leq y \leq 10\mu m$. The data points in plot (l) have an illusion of spaciousness, since they represent the position of Octree boxes at close range.

Figure 4-11 depicts energy distribution, or the summation of electron energies which strike the screen, which is complementary to Figure 4-10. As in previous examples, only the energy distribution in the x direction is shown here; however, one can assume that the energy is symmetrical in both x and y directions. Figures 4-10 and 4-11 provide all the information required to describe three-dimensional images with the spot size image on the

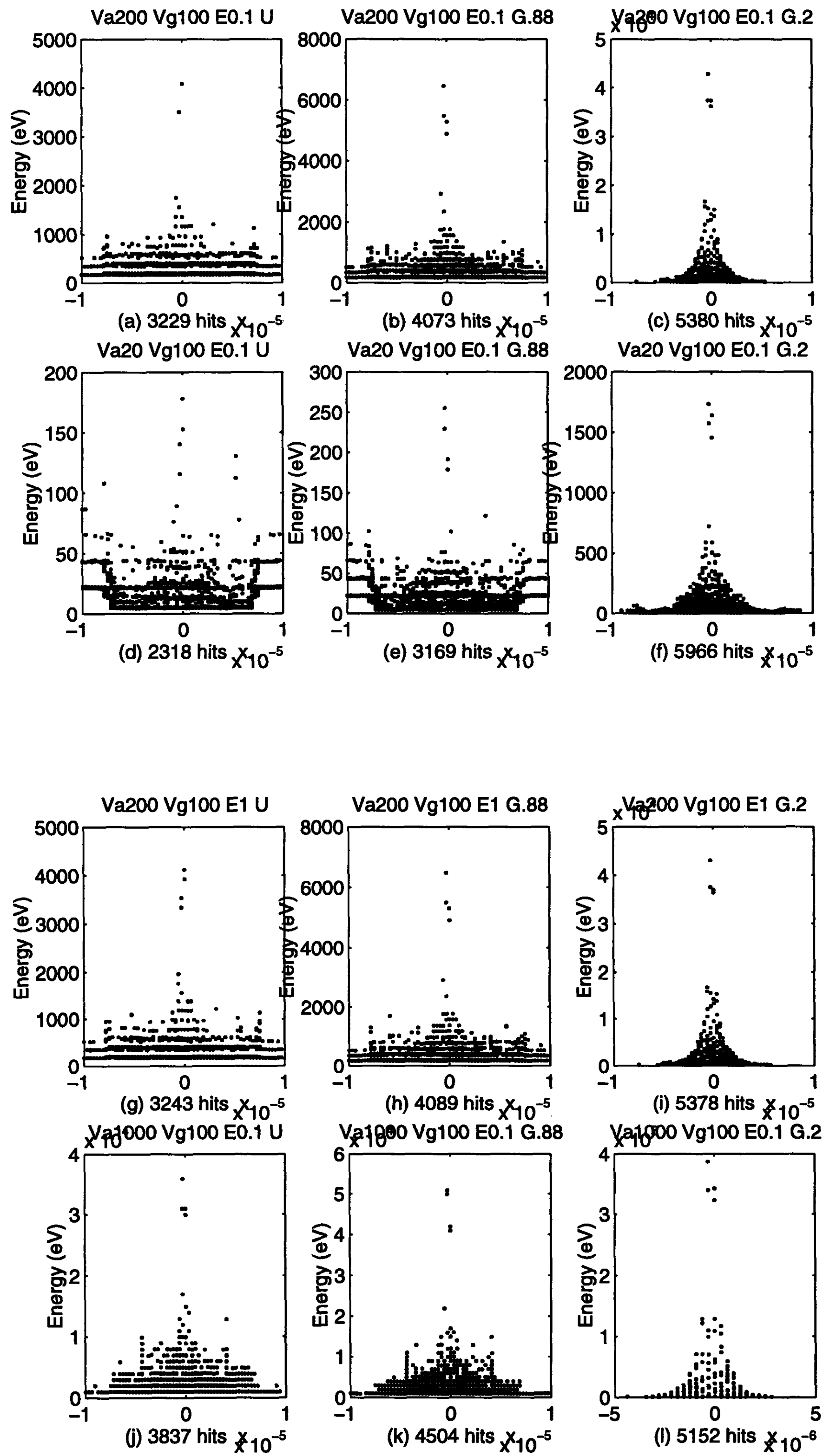


Figure 4-11: Distribution Effects: Energy Distribution

Table 4.5: Test Parameters: Effect of Initial Electron Energy

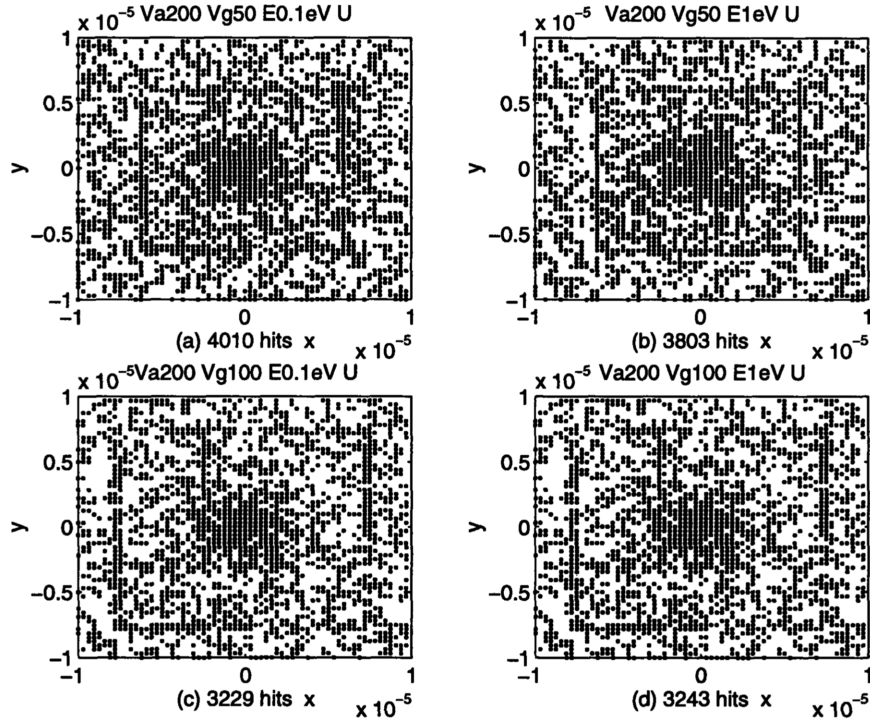
	(a), (b)	(c), (d)	(e), (f)	(g), (h)
V_A	$200V$	$200V$	$400V$	$400V$
V_G	$50V$	$100V$	$50V$	$100V$
E	$0.1eV$ $1eV$	$0.1eV$ $1eV$	$0.11eV$ $1eV$	$0.1eV$ $1eV$
Distribution	Uniform	Uniform	Uniform	Uniform

x-y plane, and the magnitude electron energies as values in z.

The plots in Figure 4-11 essentially reinforce the results obtained in Figure 4-10. The magnitude of energies about the emitter origin increases with the use of more focused initial velocity distributions. In Figure 4-11 (c), electron energies at the screen, for the gaussian distribution with a $\sigma_\theta = 0.2$, range to approximately $50000eV$. The plot suggests that almost $43000eV/200V$ or about 215 electrons land at the origin of the screen. In plot (a) the energy scale ranges to $5000eV$, indicating that approximately $4100eV/200eV$ or 21 electrons strike the screen's center. In addition, the number of electron hits increase consistently with a more focused distribution. The tendency for the cases, with a gaussian distribution and $\sigma_\theta = 0.2$, to have higher electron energies is evident in all sets of examples. From these simulations, one may infer that the resolution increases with distribution of smaller variance, and the spot size decreases in conjunction.

The purpose of this subsection is to demonstate Virtual-FED's capability of generating plots with different distributions of initial electron velocities. This characteristic of Virtual-FED allows for an accurate portrayal of real life FED devices. As previously mentioned, actual initial velocities are highly peaked about the tip of the emitter, and true electric fields are have a distributed about the center of the emitter tip, as well. The variation of results amongst the same set, i.e. those simulations with all the same initial parameters except for the distribution of initial velocities, demonstrate the importance of designating the type of distribution. A simulation with a uniform distribution gives completely different spot size and resolution results, compared with a simulation for a gaussian distribution of low variance.

4.1.4 The Effects of Initial Electron Energy



Eight examples are used to show the effects of electron energy on the screen spot size. For each set of voltage and device parameters, two simulations are compared; one with an initial energy of $0.1eV$, and the other with an initial energy of $1.0eV$. Table 4.5 provides a summary of voltage and initial energy values.

The results in Figures 4.1.4 represent the spatial distribution of electrons, or the spot size along the screen for four sets of simulations. Each set of data has two initial energy magnitudes. Similar to previous examples, all plots are labelled from above with the anode and gate voltages, the initial energy, and the distribution. The number of 'hits,' or electrons which strike the screen within the area of concern, is located in the subtitle below each plot. The first set of test cases, in Figure 4.1.4 (a) and (b) have common parameter values of $V_A = 200V$, $V_G = 50V$, and uniform distributions; however, the initial energies are $E = 0.1eV$ and $E = 1.0eV$ respectively. Comparing the two plots (a) and (b), it is apparent that there is very little difference between these two graphs. Even the hit ratios are practically the same. Plot (a), with an initial energy of $E = 0.1eV$, has 4010 hits out of 5000 electrons, or a 80.2% hit ratio. Starting with an initial energy of $E = 1eV$, plot (b) contains 3803 hits, giving a 76% hit ratio.

All other data sets introduced identical results as well. The hit ratio for Figure 4.1.4

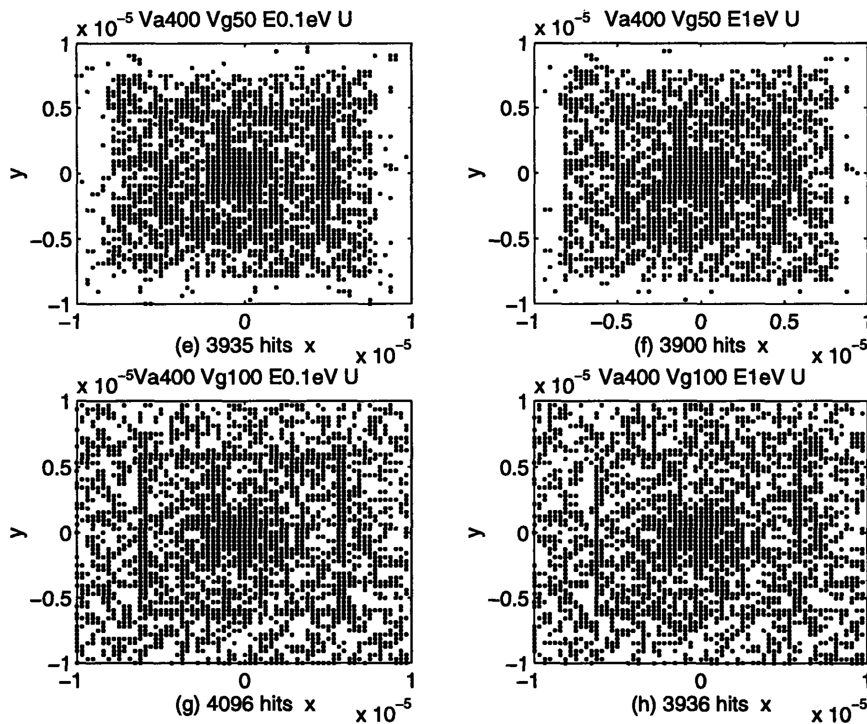


Figure 4-12: Effects of Initial Electron Energy: Distribution of Electrons

(c) is 3229/5000 or 64.58%; while for its counterpart plot (d) the ratio is 64.86%. Graphs (e) and (f) have hit ratios of 3935/5000 or 78.7%, and 3900/5000 or 78.0%. Similarly, the hit ratios for plots (g) and (h) are 4096/5000 or 81.9% and 3936/5000 or 78.7% in that order. The distribution of electrons in each set of plots are hardly distinguishable from one another.

These charateristics are also evident in Figure 4.1.4, which represents energy distribution counterpart of Figure 4.1.4, As in the previous test cases, only the energy distribution in the x direction is shown here; however, due to the symmetrical nature of the energy distribution, the plots from both Figures 4.1.4 and 4.1.4 adequately paint an image of a three-dimensional plot of the magnitude of electron energies at the screen vs. the spot size on the x-y plane. The similarities in electron energies show that the small change in initial energies, from $0.1eV$ to $1.0eV$, scarcely effect the resolution or the lateral distances travelled by electron en route from the emitter to the screen.

In conclusion, the small change in initial electron energies do not effect the energy distributions at the screen. This observation is as expected, since the anode voltages, at $200V$ and $400V$, are much higher than the initial energies. The energy of the electron at the

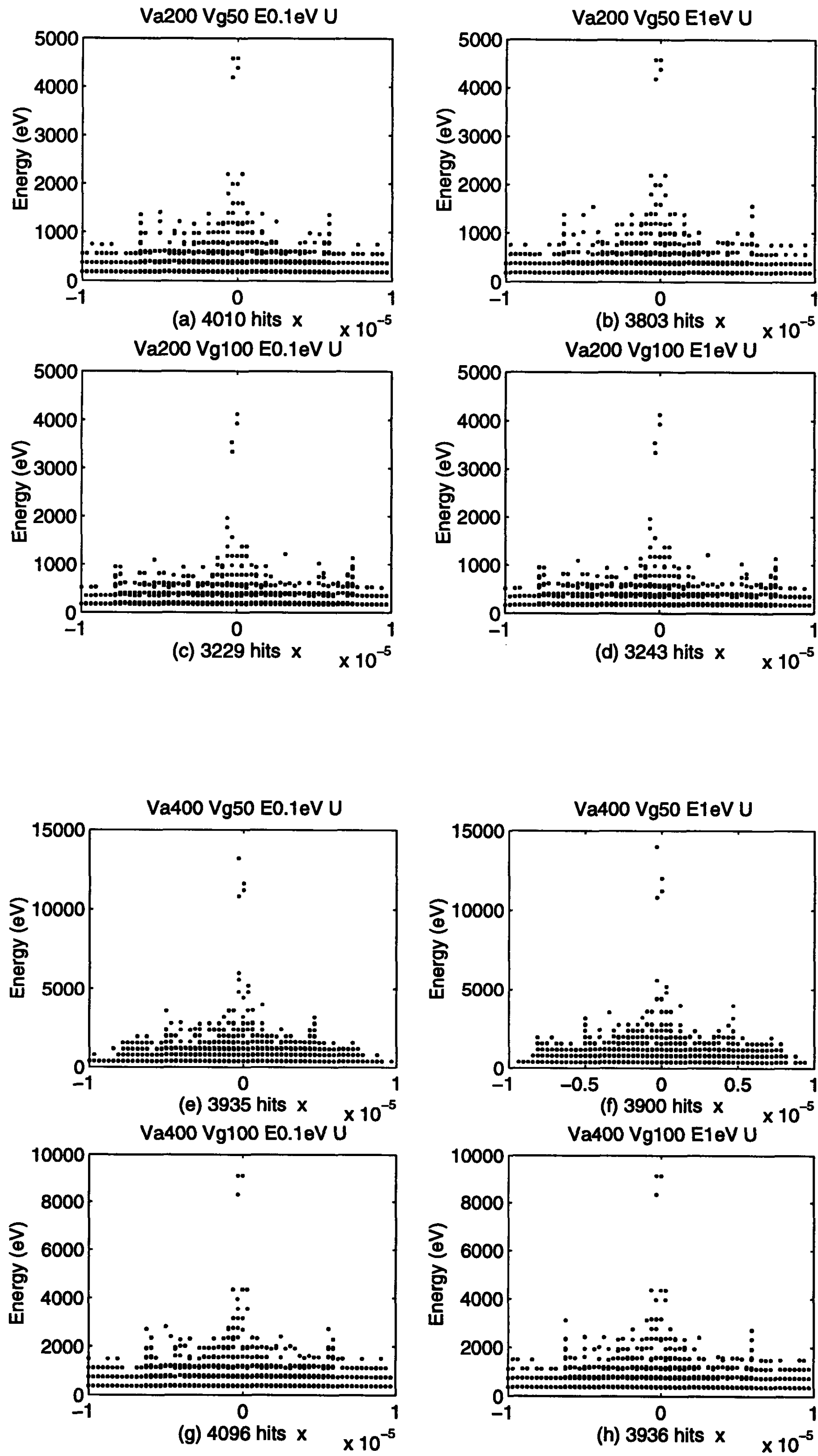


Figure 4-13: Effects of Initial Electron Energy: Energy Distribution

time it hits the screen is estimated to be the initial energy in addition to the anode voltage, scaled to the units of electron volts. So, assuming $V_A = 200V$, the estimated energy at the screen for an electron starting with $0.1eV$ is $200.1eV$. Likewise, the estimated energy at the screen for an electron starting with $1.0eV$ is $201eV$. This minuscule change in energy barely effects the spot size and resolution of the plots.

4.2 Comparison of Virtual-FED Results with Analytical Expressions

So far, simulations under Virtual-FED have produced results, very close to expected values. In subsections 4.1.1 'Electron Self-Focusing' and 4.1.2 'The Effects of Gate Voltages,' the observed electron lateral displacements, generated by Virtual-FED, from the tip of the emitter to the screen are consistent with the analytical expression of Equation 4.1. In those examples, the lateral distances, calculated by the analytical formula, are found to be larger than those depicted in the plots. The discrepancies however, may not be completely inaccurate. The analytical formula represents the lateral distance for one electron with an initial θ of $2°$. The examples generated in those sections have a range of θ uniformly distributed from $0° \leq \theta \leq 2°$. Hence, the bulk of the electrons are expected to fall inside of the lateral distances given the analytical expression. This is consistent with the results given by Virtual-FED.

The only concerns with the simulator, thus far, have been the apparent blockiness in the results generated through the uniform distribution or a gaussian distribution of high variance. This problem is less noticeable in examples which have low-variance gaussian distribution in the initial velocities. This blockiness error may possibly be attributed to the Octree, which gives a linear approximate of field values. As an alternative explanation, the blockiness may be due to an insufficient gridding of Octree `Box` structures around the emitter tip, particularly in the $0° \leq \theta \leq 2°$ range.

Chapter 5

Conclusions and Recommendations

Virtual-FED was developed with the goals for exploring field emission device design, as well as for developing and analyzing experiments to illuminate the underlying factors that describe field emission from manufactuable material surfaces. The eventual focus of design objectives for this software tool includes the ability to analyze FED designs and predict important device characteristics such as resolution, luminance, contrast ratio, and power. As a primary design feature, the Virtual-FED utitlity should have the capability of estimating the current density distribution, along the monitor screen, as a function of gate potential and arbitrary emitter spacings.

The *overall* goal is for Virtual-FED to analyze and model general FED structures. This is a huge project; however, the implementation of Virtual-FED may be broken into smaller, incremental tasks. The evolution of the Virtual-FED utility should progress in a step-by-step fashion. These incremental targets are listed below:

1. Design a function for calculating an electron trajectory from the emitter to the screen.
2. Create a structure for the efficient lookup and storage of electric fields.
3. Use the Monte-Carlo and other statistical techniques to account for non-normal electron emission at the emitter surface.
4. Assuming the electrons are emitted normal to the emitter surface, calculate the electron emission current density at the field emitter tip. This may be refined to take into account non-zero tangential electron energy.

5. In order to calculate complex fields about the FED device quickly and efficiently, integrate the Virtual-FED with MEMCAD, an existing software utility for modelling three-dimensional device structures. MEMCAD would be able to generate the electric field values inside the emitter device environment, using the boundary element method implemented in FASTCAP.
6. Predict electron current density and its spatial variation on the phosphor screen, using the utilities implemented through the above goals.
7. Calculate the luminance of the phosphor screen and its uniformity using the electron density and known phosphor characteristics.
8. Design FEAs, by replicating and superpositioning the electron current density characteristics produced from a sole emitter.
9. Test the use of focusing electrodes to reduce the spot size.

The Virtual-FED simulator will be able to model the characteristics of a FED pixel. In addition, it will be capable of linking with other display simulation programs which model the full display system. In addition, it will also be used to understand complex experimental results from field emitter.

Due to time constraints, not all of the itemized goals were accomplished over the span of this thesis. However, most of the underlying groundwork for Virtual-FED has been established and implemented. For the purposes of this work, the following elements were created for the Virtual-FED simulation tool:

- Electron trajectory calculations
- The `Octree` data structure, which stores field values and functions as a lookup table for fields.
- Electron distribution functions to account for non-normal electron emission

These created elements satisfy the first three points from the previous list. So far, the Virtual-FED utility predicts electron trajectories from the emitter to the phosphor screen, with a specified distribution of electron velocities normal to the emitter tip. The routines for calculating electron distribution at the phosphor screen, based upon the Fowler-Nordheim equation, has been established although the results are not explored. These rountines allow for the analysis of current density and its uniformity at the phosphor screen; this calculation is the fourth item on the previous list.

5.1 Recommendations for Future Work

It is a hope that the implementation of the Virtual-FED utility shall be continued, and developed to its fullest extent. Some of the key elements for future design work, described earlier in this chapter, are discussed in detail in this section.

5.1.1 Integration with MEMCAD

Virtual-FED is currently built with the intention to be included as a module of MEMCAD. As previously mentioned, MEMCAD is a software utility for creating three-dimensional models of arbitrary three-dimensional MEMS device structures. In this case, it could be used to model the FED devices, since field emitters are essentially three-dimenional multi-material structures, constructed at similar dimensional scales with MEMS devices. MEMCAD can design structures from mask and process information, and currently has existing modules for solving three-dimensional electrostatics, structural mechanics, and coupled-electromechanics.

It is a future goal for Virtual-FED simulator to use the MEMCAD system for the construction of arbitrary-field emitter-geometry models. In addition, MEMCAD would be able to quickly and efficiently calculate the electric fields throughout the entire device. These precalculated field values could then be loaded into the Virtual-FED `Octree` structure for field lookup.

The advantages of integrating with MEMCAD, and introducing a fast and efficient system which for generating complex device fields, are enormous. The Virtual-FED utility shall then have access to the MEMCAD model construction methods, as well FASTCAP, its three-dimensional electrostatic solver.

5.1.2 Trajectory Test Programs

Various test examples can be used to test the capabilities of the Virtual-FED utility. Figure 5-1 depicts an Integrated Focus Electrode Field Emitter Array (IFE-FEA) structure. The IFE-FEA consists of a basic cone emitter, gate aperature, and a second gate (or focus electrode). The first gate which is biased at a positive voltage relative to the cathode, so that it extracts electrons from the emitter. The focus electrode is biased at a negative voltage, and is used to redirect electrons to the center of the screen; thus, reducing the electron

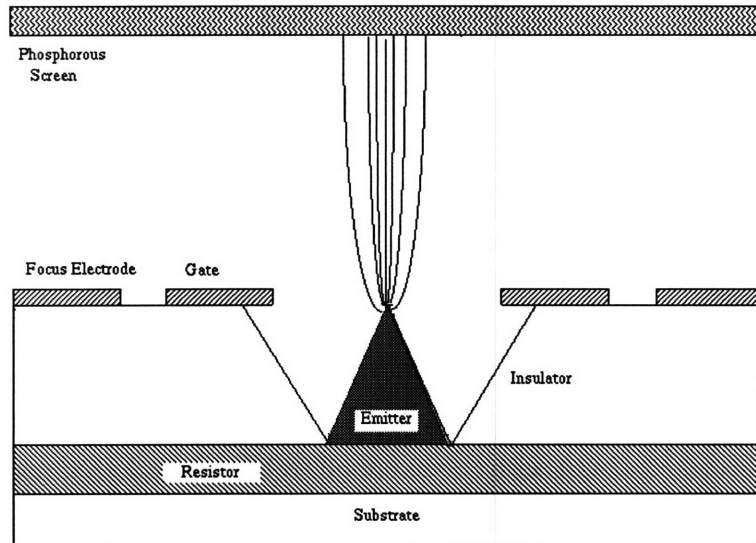


Figure 5-1: IFE-FEA Structure with Focus Electrodes at Gate Level

beam spread and spot size. By reducing the spread of the beam, the distance between the emitter and the phosphor screen may be increased. Designing a field emitter array with this device construction could solve many of the problems with the current implementation of FEDs. One of the biggest obstacles in FED design is finding a spacer material (*i*) thin and strong enough to keep the screen and the emitter plate from collapsing and (*ii*) has high enough dielectric breakdown to withstand high voltages required for high luminous efficiency phosphors. Increasing the distance from the emitter to the screen, while still maintaining spot size and resolution characteristics, allows the more common spacers, with lower dielectric breakdown voltage to be used.

Figure 5-1 gives a IFE-FED structure with the focus electrodes on the same plane as its gate. Future work may include the modelling of this device, and exploring the spot size and resolution characteristics at various emitter/screen distances. In Figure 5-2, a IFE-FED structure with the focus electrode above the emitter is shown. This device may be modelled in Virtual-FED and the results compared with those of the structure in Figure 5-1. These models will help determine the most efficient implementation of a FED device. More information about the actual implementation of these structures is given in the article *'Deflection Microwave Amplifier with Field-Emitter Arrays'*, written by Tang, Lau, and Swyden [16].

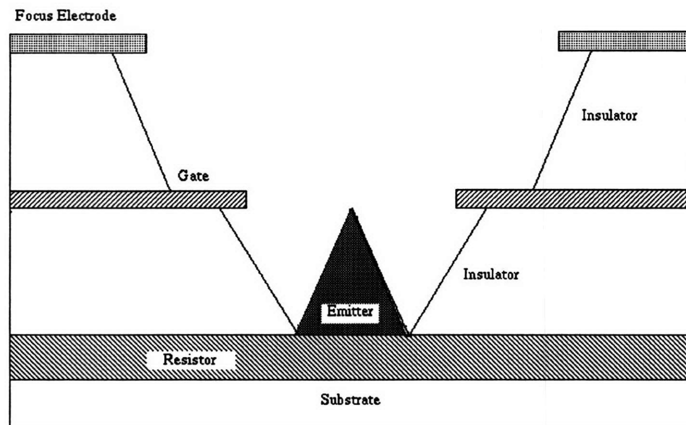


Figure 5-2: IFE-FEA Structure with Focus Electrodes above Gate Level

5.2 Conclusions

Overall, the Virtual-FED utility has been implemented and has been successful in predicting electron trajectories from the emitter to the phosphor screen, with a specified distribution of electron velocities normal to the emitter tip. Further optimizations and testing such as those suggested in the begining of this chapter will help Virtual-FED achieve its implementation goals and verify its performance. It will make Virtual-FED a powerful tool for the development of FED technologies.

Bibliography

[1] Honeywell: Sensor and System Development Center [10701 Lyndale Avenue South, Bloomington, Minnesota 55420.], *'Field Emission Display, Flat Panel Display Technology: A White Paper'*. August 1992.

[2] Meyer, R., *'Color field emission display: state of the art and prospects'*. Conference Proceedings: Euro-Display'93, 13th International Display Research Conference LETI, Grenoble, France, August 31-September 3, 1993, pages 189-192.

[3] Gray, Henry F. *'The field-emitter display'*. Information Display, Vol. 9, No. 3, March 1993, pages 9-14.

[4] Good, Jr., H. R. and Erwin W. Müller, *'Field Emission.'*. Handbouch der Physik, Vol. XXI, S. Ḟlugge Springer, 1956.

[5] Cathey, Jr., David A., *'Field Emission Display'*. Information Display, Vol. 11, No. 10, October 1995, pages 16-20.

[6] Spindt, C. A., I. Brodie, L.Humphrey, and E. R. Westerberg, *'Physical properties of thim-film field emission cathodes'*. Journal of Applied Physics, Vol. 47, No. 12, December 1976, pages 5248-5263.

[7] Kishino, Takao and Shigeo Itoh, *'Present Status of the FED Development and Problems to Solve'*. Research and Development Center, Futaba Corporation, 1080 Yabutsuka, Chosei, Chiba 299-43 Japan.

[8] Choi, Kyung Cheol and Nagyoung Chang, *'Modelling of a Field Emission Display Using the Adaptive Scheme Method'*. International Display Research Conference, Monterey, CA, October 10-13, 1994, pages 237-240.

[9] Kesling, W. Dawson and Charles E. Hunt, *'Beam Focusing for Field Emission Flat Panel Display'*. [Department of Electrical and Computer Engineering - University of California Davis, CA 95616], The 7th International Vacuum Microelectronics Conference (IVM), Grenoble, France, July 4-7, 1994, pages 135-138.

[10] Weber, M., M. Rudolph, J. Kretz, and H. W. P. Koops, *'Electron-beam induced deposition for fabrication of vacuum field emitter devices'*. [Institut fü Angewandte Physik, Technische Hochschule Darmstadt, Schlossgartenstr. 7, D-64289 Darmstadt.] The 7th International Vacuum Microelectronics Conference (IVM), Grenoble, France, July 4-7, 1994, pages 395-400.

[11] Fedirko, V.A., N. G. Belova, V. I. Makhov, *'Numerical Modelling of Microvacuum Cell with a Cylinrical Field Emitter'*. The 7th International Vacuum Microelectronics Conference (IVM), Grenoble, France, July 4-7, 1994, pages 155-157.

[12] Munro, E., X. Zhu, J. A. Rouse and H. Liu, *'Computer Simulation of Vacuum Microelectronic Components'*. [Munro's Electron Beam Software Ltd, 14 Cornwall Gardens, London SW7 4AN, England.] The 7th International Vacuum Microelectronics Conference (IVM), Grenoble, France, July 4-7, 1994, pages 143-146.

[13] Lerman, Steven R., *'Problem Solving and Computation for Scientist and Engineers'*. Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1993, pages 304-317.

[14] Helstrom, Carl W. *'Probablity and Stochastic Processes for Engineers'*. Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1991, pages 81-90.

[15] Spindt, Charles A., Christopher E. Holland, Ivor Brodie, John B. Mooney, and Eugene R. Westerberg, *'Field-Emitter Arrays Applied to Vacuum Fluorescent Display'*. IEEE Transactions on Electron Devices, Vol. 36, No. 1, January 1989, pages 225-189.

[16] Tang, Cha-Mei, Y. Y. Lau, and T. A. Swyden, *'Deflection Microwave Amplifier with Field-Emitter Arrays'*. The 7th International Vacuum Microelectronics Conference (IVM), Grenoble, France, July 4-7, 1994, pages 292-295.

Appendix A

trajectory.h

```
/*
 *  name of this file:   pixel/src/traj1/trajectory.h
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define  SAMPLING_NO    100
#define  Q              1.609e-19   /* charge of electron */
#define  M              9.11e-31    /* mass of electron */
#define  PI             3.14159265359
#define  TRUE   1
#define  FALSE  0

struct vector{
  double x;                   /* x component */
  double y;                   /* y component */
  double z;                   /* z component */
};
typedef struct vector Vector;

struct analytical_field{
  struct vector e;            /* Electric field */
  struct vector b;            /* Magnetic field */
};
typedef struct analytical_field Afield;

struct box{
  struct box * up;            /* pointers to upper level box */
  struct box * subboxes[8];  /* pointers to subboxes */
  struct analytical_field *fd[8];  /* field at box */
  struct data * dat; /* electron velocities through octree */
  int depth;         /* depth of current box */
  int num_terminated;/* number of terminated subboxes */
```

```
  Vector coordinate; /* coordindate of lower left corner of box */
  int isboundary;    /* boolean, denotes boundary is inside octree */
  int tag;           /* integer tag which denotes type of boundary */
};
typedef struct box Box;
typedef struct box * BoxPtr;

struct octree{
  struct box *top; /* ptr to top (largest) box */
  int count;       /* total number of boxes, including all layers */
  int depth;       /* largest depth */
  struct map *axis;/* scale factor for x,y,z axis */
  int isboundary;  /* boolean, denotes boundary is inside octree */
};
typedef struct octree Octree;
typedef struct octree * OctreePtr;

struct data{
  int num;      /* number of electrons passing through box */
  double energy;/*describes the total energy of passing electron*/
};
typedef struct data Data;
typedef struct data * DataPtr;

struct map{
  Vector * scale;  /* scale factor for x,y,z axis */
  Vector * offset; /* lower left hand position */
};
typedef struct map Map;
typedef struct map * MapPtr;

struct field_struct{
  struct analytical_field af;  /* analytical fields */
  struct octree * oct;         /* fields stored on an octree */
};
typedef struct field_struct Fields;

struct initiale{
  double q;           /* charge of particle */
  double m;           /* mass of particle */
  struct vector pos;  /* intial postion vector of particle */
  struct vector vel;  /* intial velocity vector of particle */
};
typedef struct initiale Initiale;

struct bounds{
  double xmax;  /* maximum x value of cube */
  double xmin;  /* minimum x value of cube */
  double ymax;  /* maximum y value of cube */
  double ymin;  /* minimum y value of cube */
  double zmax;  /* maximum z value of cube */
  double zmin;  /* minimum z value of cube */
};
typedef struct bounds Bounds;
```

```
/*************** trajectory structures **************/
struct traj_elem{
  struct traj_elem * next;   /* pointer to next traj_elem */
  struct traj_elem * prev;   /* pointer to prev traj_elem */
  struct vector pos;         /* postion vector of particle */
  struct vector vel;         /* velocity vector of particle */
  double t;                  /* total time elapsed */
  double dt;                 /* increment in time */
};
typedef struct traj_elem Listelem;
typedef struct traj_elem * ListelemPtr;
typedef struct traj_elem * List;

struct trajectory{
  int num;                   /* total number of traj_elem's */
  struct traj_elem * front;  /* pointer to first elem of list*/
  struct traj_elem * rear;   /* pointer to last elem of list*/
  struct traj_elem * arry;   /* arry of trajectory elements */
};
typedef struct trajectory Trajectory;
typedef struct trajectory * TrajectoryPtr;

/** function prototype **/
/** initialization functions **/
void init_particle(Initiale *ch);
void init_bounds(Bounds *b);
void init_fields(Fields *f);

void init_all(TrajectoryPtr *traj, Initiale *ch,
      Bounds *b, Fields *f, Afield *af, double *maxerror);

void init_trajectory(TrajectoryPtr *t, Initiale ch);

/* adding trajectory element functions */
TrajectoryPtr make_trajectory(void);
List make_list(void);
ListelemPtr make_listelem(void);
TrajectoryPtr add2front(TrajectoryPtr t, List elem);
TrajectoryPtr add2rear(TrajectoryPtr t, List elem);
TrajectoryPtr cp_trajectory(TrajectoryPtr traj);
ListelemPtr cp_listelem(ListelemPtr lptr);

/* remove trajectory element functions */
List rm_front(TrajectoryPtr t);
List rm_rear(TrajectoryPtr t);

/* freeing memory */
void destroy_trajectory(TrajectoryPtr t);
void destroy_list(List l);
void destroy_arry(struct traj_elem * a);

/* printing records */
void print2file(TrajectoryPtr t);
```

```
void print_transit(TrajectoryPtr t);

/* higher trajectory functions */
TrajectoryPtr transit(TrajectoryPtr t, Initiale ch,
      Fields f, Bounds b, double max_error_per);
void get_trajectory(TrajectoryPtr traj, Initiale ch,
    Fields f, Bounds b, double factor);
int next(TrajectoryPtr traj, Initiale ch, Fields f,
 Bounds b, double factor,
 List (*method)(TrajectoryPtr t, Initiale ch,
Fields f, double delt));

/* method functions */
List linear(TrajectoryPtr t, Initiale ch, Fields fs, double dt);
List runge_kutta(TrajectoryPtr traj, Initiale ch,
Fields fi, double dt);

/* lookup functions */
Vector position(List step);
double length(Vector k);
int get_traj_length(TrajectoryPtr t);
double min_vector(double x, double y, double z);
double min(double x, double y);

/* guessing dt functions */
double guess_dt1(List step,  Bounds b, double init_guess);
double guess_dt2(Initiale ch, Fields f);
double guess_dt3(Bounds b, int no);
Vector acceleration(Initiale ch, Fields f);

/* interpolation */
double difference(TrajectoryPtr t1, TrajectoryPtr t2,
  Bounds b, int sampling_no);
Vector iter(TrajectoryPtr t, double time,
    Vector (*method)(TrajectoryPtr t, double time));
Vector linear_interp(TrajectoryPtr t, double time);
double difference2(TrajectoryPtr t1, TrajectoryPtr t2, Bounds b);
Vector iter2(TrajectoryPtr t, double time, double dt,
    Vector (*method)(TrajectoryPtr t, double time));

/* other functions */
List *trajectory2arry(TrajectoryPtr t);
TrajectoryPtr coarse(TrajectoryPtr t1, TrajectoryPtr t2);

/* boundary checking */
int checkbounds(Vector p, Bounds b);
List bound_position(List in, List out, Bounds bd);

/* current velocity distribution */
void func(Initiale ch, Fields f, Bounds b, double max_error_per,
  double energy, int num_traj, double degree,
  int gauss, double variance);
void uniform_dist(Initiale ch, Fields f, Bounds b,
  double max_error_per, double energy,
```

```
  int num_traj, double degree);
void gaussian_dist(Initiale ch, Fields f, Bounds b,
  double max_error_per, double energy, int num_traj,
  double degree, int gauss, double variance);
Vector sphere2cart(double vel, double theta, double phi);
Vector uniform(double vel, double degree);
Vector gaussian(double v_t, double degree,
double mean, double variance);

/* current density distribution */
double fowler_nordheim(double elecfield, double phi);
```

Appendix B

trajectory.h

```
/*
 *  name of this file: pixel/src/traj1/trajectory.m
 */

#include "octree.m"

double GUESS = 5e-3;

/********* trajectory initialization functions ***********/
/*
 * function:     init_all
 * returns:      void
 * arguments:    TrajectoryPtr *t, uninitialized output trajectory
 *               Initiale *ch,     initial charge characteristics
 *               Bounds *b,        boundary
 *               Fields *f,        electric and magnetic fields
 *               double *maxerror, maximum tolerable error between simuls
 * description:  This function initializes everything from the arguments
 *               to the Octree structure.
 */
void init_all(TrajectoryPtr *traj, Initiale *ch, Bounds *b,
      Fields *f, Afield *af, double *maxerror)
{
  Vector pt;
  int min_depth = 0;/* define minimum depth here */
  int ref = 1;      /* number of Octree refinement layers is at least 1*/

  /* initialize parameters for the initial electron values*/
  init_particle(ch);  /* allocate charge paritcle */
  /* initial charge and mass */
  printf("charge:%e mass:%e\n", ch->q, ch->m);
  /* input initial electron velocity */
  printf("Initial velocities %f %f %f\n",
 ch->vel.x, ch->vel.y, ch->vel.z);
  /* input initial electron position */
```

```
  printf("Input initial position %f %f %f\n",
 ch->pos.x, ch->pos.y, ch->pos.z);

  /* initialize parameters for the initial electron values*/
  init_bounds(b);  /* allocate memory for boundary structure */
  printf("\nupper-right-corner of boundary %f %f %f",
 b->xmax, b->ymax, b->zmax);  /* input upper right hand corner */
  printf("\nInput lower-left-corner of boundary %f %f %f\n",
 b->xmin, b->ymin, b->zmin);  /* input lower left hand corner */

  /* initialize parameters for the initial electron values*/
  init_fields(f);  /* allocate memory for field structure */
  printf("\nFields E_x, E_y, E_z, B_x, B_y, B_z:\n%f %f %f %f %f %f",
f->af.e.x,f->af.e.y,f->af.e.z,f->af.b.x,f->af.b.y,f->af.b.z);
  *af = (*f).af;

  /* allocate memory for field structure */
  init_trajectory(traj, *ch);

  printf("\nEnter the maximum percent error:  ");
  scanf("%lf", maxerror);
  printf("%f", *maxerror);

  /* Octree initializations occur here */
  printf("\nOctree refinement: "); /* refinement percentage */
  scanf("%d", &ref);
  printf("%d", ref);

  oct = make_octree(af,*b);  /* allocate for Octree */
  printf("\nMinimum depth ");/* define minimum depth */
  scanf("%d", &min_depth);
  printf("%d", min_depth);

  /* define singularity point to refine */
  printf(" point (to refine)");
  scanf("%lf %lf %lf", &pt.x, &pt.y, &pt.z);

  refine(oct->top, oct, af, min_depth, *b); /* refine min depth */
  /* refine down to a certain percentage around singular point */
  refine2pointperc(oct->top, oct, af, pt, ref, *b);
  /* refine down to level level (ref) within a semi-cicular volume
     centered about the origin */
  refinesemi(oct->top, oct, af, ref, 30, 0.5e-6, 0.05e-6, *b);
 /* refine the top boundary */
  refinetopbds(oct->top, oct, af, 6, *b);
#ifdef DEBUG
  printf("octree box count %d\n", count);
#endif
  (*f).oct = oct;
}

/*
 * function:     init_trajectory
 * returns:      void
```

```
 * arguments:    TrajectoryPtr *traj,uninitialized output trajectory
 *               Initiale ch,  initial charge characteristics
 * description:  This function initializes the trajectory with
 *               one list element (traj_struct) with the same
 *               characteristics as the initial charge.
 */
void init_trajectory(TrajectoryPtr *t, Initiale ch)
{
  TrajectoryPtr tmp;/* temporary trajectory pointer */
  List l;          /* list element - the first and last in
       the queue since it is the only element */

  tmp = make_trajectory();/* allocate trajectory */
  l= make_list();         /* allocate list strucuture*/
  l->pos = ch.pos;        /* write initial position into listelem */
  l->vel = ch.vel;        /* write initial velocity into listelem */
  l->t = 0.0;             /* totaltime is zero before trajectory begins*/
  l->dt = 0.0;            /* change in time assigned zero also */
  l->prev = NULL;         /* only one element in list so no previous */
  l->next = NULL;         /* or next elements */

  tmp->front = l;         /* assign listelem to front of queue */
  tmp->rear = l;          /* assign listelem to back of queue also*/
  tmp->num = 1;           /* one listelem in queue */
  tmp->arry= NULL;

  *t=tmp;
}

/********* adding to trajectory functions ***********/

/*
 * function:     make_trajectory
 * returns:      TrajectoryPtr
 * arguments:    none
 * description:  This function allocates space for a trajectory.
 */
TrajectoryPtr make_trajectory(void)
{
  TrajectoryPtr tmp;  /* temporary trajectory pointer */

  /* dynamically allocating trajectory */
  if ((tmp = malloc(sizeof(Trajectory))) == NULL)  {
    printf("Memory allocation error for additional trajectory.");
    printf("  Exiting program.\n");
    exit(-1);
  }

  tmp->num = 0;   /* assign number of list elements to be zero */
  tmp->front = NULL;
  tmp->rear = NULL;
  tmp->arry = NULL;
  return tmp;
}
```

```
/*
 * function:    make_list
 * returns:     List
 * arguments:   none
 * description: This function allocates space for an empty list.
 */
List make_list(void)
{
  List l;
  /* dynamically allocating list */
  if ((l = malloc(sizeof(struct traj_elem))) == NULL)  {
    printf("Memory allocation error for trajectory list.");
    printf("  Exiting program.\n");
    exit(-1);
  }
  l->next = NULL; /* initiate next and prev */
  l->prev = NULL;
  return l;
}


/*
 * function:     make_listelem
 * returns:      ListelemPtr
 * arguments:    none
 * description:  This function allocates space for an empty listelem.
 */
ListelemPtr make_listelem(void)
{
  ListelemPtr l;

  /* dynamically allocating listelem */
  if ((l = malloc(sizeof(Listelem))) == NULL)  {
    printf("Memory allocation error for trajectory list element.");
    printf("  Exiting program.\n");
    exit(-1);
  }
  l->next = NULL; /* initiate next and prev */
  l->prev = NULL;
  return l;
}

/*
 * function:     add2rear
 * returns:      Updated TrajectoryPtr
 * arguments:    TrajectoryPtr traj, trajectory
 *               List elem,          list element
 * description:  this function adds the list element to the rear
 *               of the trajectory of list structure.
 */
TrajectoryPtr add2rear(TrajectoryPtr traj, List elem)
{
  elem->prev = traj->rear;
```

```
  traj->rear->next = elem;
  traj->rear = elem; /* element is now at rear of trajectory list*/
  (traj->num)++;     /* increment number of traj. elements */
  return traj;
}

/*
 * function:     add2front
 * returns:      Updated TrajectoryPtr
 * arguments:    TrajectoryPtr traj, trajectory
 *               List elem,          list element
 * description:  this function adds the list element to the front
 *               of the trajectory of list structure.
 */
TrajectoryPtr add2front(TrajectoryPtr traj, List elem)
{
  traj->front->prev = elem;
  elem->next = traj->front;
  traj->front = elem;/* element is at front of trajectory list*/
  (traj->num)++;     /* increment number of traj. elements */
  return traj;
}

/*
 * function:     cp_trajectory
 * returns:      TrajectoryPtr
 * arguments:    TrajectoryPtr traj, trajectory
 * description:  Returns copy of trajectory
 */
TrajectoryPtr cp_trajectory(TrajectoryPtr traj)
{
  TrajectoryPtr tmp;
  List l;
  ListelemPtr elem;

  tmp = make_trajectory(); /* allocate for new trajectory */
  tmp->num = traj->num;    /* the both have same # of listelements */
  tmp->arry = traj->arry;

  l = traj->front;
  if(l == NULL){          /* if trajectory list empty */
    tmp->front = NULL;    /* return empty trajectory list */
    tmp->rear = NULL;
    return tmp;
  }

  elem = cp_listelem(l);  /* copy the first element of */
  tmp->front = elem;      /* trajectory list into element */
  tmp->rear = elem;
  l=l->next;

  while(l != NULL){         /* while other elements exist in */
    elem = cp_listelem(l);  /* the trajectory list, copy a new */
    elem->prev = tmp->rear; /* element */
```

```
    tmp->rear->next = elem;
    tmp->rear = elem;
    l=l->next;
  }
  return tmp;
}

/*
 * function:     cp_listelem
 * returns:      ListelemPtr
 * arguments:    ListelemPtr lptr, point to list
 * description:  Returns copy of list element
 */
ListelemPtr cp_listelem(ListelemPtr lptr)
{
  ListelemPtr elem;

  elem = make_listelem(); /* allocate new listelem */
  elem->pos = lptr->pos;  /* make a copy of all values */
  elem->vel = lptr->vel;
  elem->t = lptr->t;
  elem->dt = lptr->dt;
  elem->next = NULL;
  elem->prev = NULL;
  return elem;
  }

/********* remove trajectory functions ***********/
/*
 * function:     rm_front
 * returns:      List item element being removed
 * arguments:    Trajectory traj,    trajectory
 * description:  Removes the front item of trajectory list.
 */
List rm_front(TrajectoryPtr traj)
{
  List tmp = traj->front;
  traj->front = traj->front->next;
  traj->front->prev=NULL;
  (traj->num)--; /* decrement # of trajectory elements */
  return tmp;    /* return listelem so it can be freed */
}

/*
 * function:     rm_rear
 * returns:      List item element being removed
 * arguments:    TrajectoryPtr traj, trajectory
 * description:  Removes the rear item of trajectory list.
 */
List rm_rear(TrajectoryPtr traj)
{
  List tmp = traj->rear;
  traj->rear=traj->rear->prev;
  traj->rear->next=NULL;
```

```
  (traj->num)--; /* decrement # of trajectory elements */
  return tmp;    /* return listelem so it can be freed */
}

/********* freeing memory ***********/
/*
 * function:     destroy_trajectory
 * returns:      void
 * arguments:    TrajectoryPtr traj, trajectory
 * description:  Frees trajectory.
 */
void destroy_trajectory(TrajectoryPtr traj)
{
  destroy_list(traj->rear);
  free(traj->arry);
  free(traj);
}

/*
 * function:     destroy_list
 * returns:      void
 * arguments:    List list (the rear element of a trajectory list)
 * description:  Frees list.
 */
void destroy_list(List l)
{
  while(l != NULL){
    free(l->next);
    l = l->prev;
  }
}

/*
 * function:     destroy_arry
 * returns:      void
 * arguments:    struct traj_elem * a
 * description:  Frees arry.
 */
void destroy_arry(struct traj_elem * a)
{
  free(a);
}

/********* printing functions ***********/
/*
 * function:     print2file
 * returns:      void
 * arguments:    TrajectoryPtr traj, non-empty trajectory
 * description:  Prints trajectory information to specified file.
 */
void print2file(TrajectoryPtr traj)
{
  int i;
  char s[50];    /* filenames cannot be more than 49 chars long */
```

```
  FILE *fptr;    /* pointer to file */
  List l = traj->front;/*assign reference pointer to trajectory list*/

  printf("\nEnter filename to be written:");
  scanf("%s", s);

  /* test for problems opening file */
  if((fptr = fopen(s, "w")) == NULL ){
    printf("Error opening file %s. Exiting program\n", s);
    exit(1);
  }

  /* print trajectory information to file */
  fprintf(fptr, "Number of elements in trajectory: %d\n",traj->num);
  for(i=0; i<traj->num ; i++, l=l->next)
    /* print information from each listelem into file */
    fprintf(fptr, "%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\n",
    l->pos.x, l->pos.y, l->pos.z,
    l->vel.x, l->vel.y, l->vel.z, l->t, l->dt);
  fclose(fptr);
}

/*
 * function:     print_transit
 * returns:      void
 * arguments:    TrajectoryPtr traj,  non-empty trajectory
 * description:  Prints contents of trajectory to screen.
 */
void print_transit(TrajectoryPtr traj)
{
  int i;
  List l = traj->front;

  /* print trajectory information to screen */
  printf("Number of elements in trajectory: %d\n", traj->num);
  for(i=0; i<traj->num ; i++, l=l->next)
    printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\n",
   l->pos.x, l->pos.y, l->pos.z,
   l->vel.x, l->vel.y, l->vel.z, l->t, l->dt);
}

/*
 * function:     print_list
 * returns:      void
 * arguments:    List l,
 * description:  Prints contents of list to screen.
 */
void print_list(List l)
{
  List tmp = l;
  /* print list information to screen */
  for(; tmp != NULL ; tmp=tmp->next)
    printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\n",
   tmp->pos.x, tmp->pos.y, tmp->pos.z,
```

```
   tmp->vel.x, tmp->vel.y, tmp->vel.z, tmp->t, tmp->dt);
}


/********* higher trajectory functions ***********/
/*
 * function:     transit
 * returns:      void
 * arguments:    TrajectoryPtr traj,    trajectory to be modified
 *               Initiale ch, Fields f, initial charge and fields
 *               Bounds b,              boundary
 *               double max_error_per   maximum tolerable error
 *                                      between simulations
 * description:  This function calculates trajectory which first
 *               falls within the maximum tolerable error percentage,
 *               given initial charge and fields boundary parameters.
 */
TrajectoryPtr transit(TrajectoryPtr traj, Initiale ch,
      Fields f, Bounds b, double max_error_per)
{
  TrajectoryPtr t1, t2, tmp;
  double max_error = max_error_per/100.0;
  double factor = 1.0;  /* factor for guess initial change in time */

  /* copy initial trajectory to get its initial
     eletron values, in the first listelem */
  t1=cp_trajectory(traj);
  t2=cp_trajectory(traj);

  /* get values for t1 with the initial change in time value (delt) */
  get_trajectory(t1,ch,f,b,factor);
  /* get values for t2 with the half the value of initial change in time */
  factor /= 2.0;
  get_trajectory(t2,ch,f,b,factor);

  /* While the difference between the (linear) interpolation in time is
   * greater than the maximum tolerable error, recalculate with delta /= 2
   */
  for(factor/=2.0;
      fabs(difference(t1,t2,b,SAMPLING_NO))>max_error;
      factor/=2.0){
    destroy_trajectory(t1);          /* destroy t1 */
    t1 = cp_trajectory(t2);          /* assign t1 = t2 */
    destroy_trajectory(t2);          /* destroy t1 */
    t2 = cp_trajectory(traj);

    /* recalculate t2 with new (smaller) factor */
    get_trajectory(t2,ch,f,b,factor);
  }

  /* when the trajectory differences are satisfied, throw away t1,
   * and assign the output trajectory = t2
   */
  destroy_trajectory(t1);
```

```
  /* write data */
  tmp = t2;
  while(tmp->front->next != NULL){
    if(different_box(oct, tmp->front->pos, tmp->front->next->pos))
      write_data(oct, tmp->front->pos, tmp->front->vel, ch);
    tmp->front = tmp->front->next;
  }
  /* write last data */
/*  if(different_box(oct, tmp->front->prev->pos, tmp->front->pos))*/
    write_data(oct, tmp->front->pos, tmp->front->vel, ch);

  return t2;
}

/*
 * function:     get_trajectory
 * returns:      void
 * arguments:    TrajectoryPtr traj,     trajectory to be modified
 *               Initiale ch, Fields f,  initial charge and fields
 *               Bounds b,               boundary
 * description:  This function calls for the calculation of individual
 *               trajectory elements.
 */
void get_trajectory(TrajectoryPtr traj, Initiale ch, Fields f,
    Bounds b, double factor)
{
  /* while next() returns TRUE, i.e. the trajectory is still within
   * the boundary, calculate the next trajectory list element.
   * Conclude when trajectory element is out of bounds.
   */
  while(next(traj, ch, f, b, factor, runge_kutta));
}

/*
 * function:     next
 * returns:      TRUE or FALSE
 * arguments:    TrajectoryPtr traj,   trajectory to be modified
 *               Initiale ch, Fields f,initial charge and fields
 *               Bounds b,             boundary
 *               double factor,        change in time times FACTOR
 *          List (*method)    function which returns a list element
 * description:  This function calculates the next trajectory list element
 *               by means of the method function specified by user.  If
 *               the next trajectory element is inside the boundary, it
 *               is appended to the end of the trajectory list.  This
 *               function returns true if the next list element is inside
 *               boundary, false otherwise.
 */
int next(TrajectoryPtr traj, Initiale ch,
 Fields f, Bounds b, double factor,
 List (*method)(TrajectoryPtr t, Initiale ch,
Fields f, double delt))
{
```

```
  List step;
  double delt = factor*guess_dt1(traj->rear, b, GUESS);
  traj->rear->dt = delt;

  /* calculate next element using method */
  step = method(traj, ch, f, delt);
  /* check if element (step) is instide boundary */

#ifdef DEBUG2
  printf("check %d count %d", checkbounds(step->pos, b), count);
  printf("p: %e %e %e\n",step->pos.x, step->pos.y, step->pos.z);
#endif

  if(checkbounds(step->pos, b)){     /* if step inside boundary */
    traj = add2rear(traj, step);     /* add to rear of list */
    return TRUE;                     /* return true */
  }
  else {
    traj = add2rear(traj, bound_position(traj->rear, step, b));
    /* add interpolated boundary position to list,return false */
    return FALSE;
  }
}

/********* method functions ***********/
/*
 * function:     linear
 * returns:      List
 * arguments:    TrajectoryPtr traj,     trajectory
 *               Initiale ch, Fields f,  initial charge and fields
 *               double delt,            change in time
 * description:  This function calculates the next trajectory list
 *               element by means of the linear method.  Returns a list
 *               TrajectoryPtr element with the calculated values.
 */
List linear(TrajectoryPtr traj, Initiale ch, Fields fi, double delt)
{
  List tmp;
  Vector dv, v = traj->rear->vel; /* use velocity from last element*/
  Afield *anf = fieldop_oct(oct, traj->rear->pos); /* lookup fields */
  double k = ch.q/ch.m;

  tmp = make_list();

  /* incremental change in velocity */
  dv.x = k * (anf->e.x + v.y*anf->b.z - v.z*anf->b.y) * delt;
  dv.y = k * (anf->e.y + v.z*anf->b.x - v.x*anf->b.z) * delt;
  dv.z = k * (anf->e.z + v.x*anf->b.y - v.y*anf->b.x) * delt;

  /* assign those new values into tmp */
  tmp->vel.x = v.x+dv.x;
  tmp->vel.y = v.y+dv.y;
  tmp->vel.z = v.z+dv.z;
  tmp->pos.x = traj->rear->pos.x + 0.5 * (tmp->vel.x+v.x) * delt;
```

```
  tmp->pos.y = traj->rear->pos.y + 0.5 * (tmp->vel.y+v.y) * delt;
  tmp->pos.z = traj->rear->pos.z + 0.5 * (tmp->vel.z+v.z) * delt;

#ifdef DEBUG2
 printf("anf->e %f %f %f pos %f %f %f vel %f %f %f dv %f %f %f\n",
       anf->e.x, anf->e.y, anf->e.z, tmp->pos.x, tmp->pos.y, tmp->pos.z,
       tmp->vel.x, tmp->vel.y, tmp->vel.z, dv.x, dv.y, dv.z);
#endif

  tmp->dt = delt;   /* change in time = delt */
  tmp->t = traj->rear->t + delt;/*total time = prev_time + delt*/
  tmp->prev=NULL;
  tmp->next=NULL;
  free(anf);        /* free the field */
  return tmp;
}

/*
 * function:     runge_kutta
 * returns:      List
 * arguments:    TrajectoryPtr traj,     trajectory
 *               Initiale ch, Fields f,  initial charge and fields
 *               double delt,            change in time
 * description:  This function calculates the next trajectory list
 *               element by means of the runga-kutta method.  Returns
 *               a list trajectory element with the calculated values.
 */
List runge_kutta(TrajectoryPtr traj, Initiale ch, Fields fi, double delt)
{
  List tmp;
  Vector dv0, dv1, dv2, dv3, v = traj->rear->vel;
  Afield *anf = fieldop_oct(oct, traj->rear->pos); /* lookup fields */
  double k = ch.q/ch.m;
  tmp = make_list();

  /* incremental change in velocity */
  dv0.x = k * (anf->e.x + v.y*anf->b.z - v.z*anf->b.y) * delt;
  dv0.y = k * (anf->e.y + v.z*anf->b.x - v.x*anf->b.z) * delt;
  dv0.z = k * (anf->e.z + v.x*anf->b.y - v.y*anf->b.x) * delt;

  dv1.x = k * (anf->e.x + (v.y+dv0.x/2)*anf->b.z
       - (v.z+dv0.z/2)*anf->b.y) * delt;
  dv1.y = k * (anf->e.y + (v.z+dv0.z/2)*anf->b.x
       - (v.x+dv0.x/2)*anf->b.z) * delt;
  dv1.z = k * (anf->e.z + (v.x+dv0.x/2)*anf->b.y
       - (v.y+dv0.y/2)*anf->b.x) * delt;

  dv2.x = k * (anf->e.x + (v.y+dv1.x/2)*anf->b.z
       - (v.z+dv1.z/2)*anf->b.y) * delt;
  dv2.y = k * (anf->e.y + (v.z+dv1.z/2)*anf->b.x
       - (v.x+dv1.x/2)*anf->b.z) * delt;
  dv2.z = k * (anf->e.z + (v.x+dv1.x/2)*anf->b.y
       - (v.y+dv1.y/2)*anf->b.x) * delt;
```

```
  dv3.x = k * (anf->e.x + (v.y+dv2.y)*anf->b.z
       - (v.z+dv2.z)*anf->b.y) * delt;
  dv3.y = k * (anf->e.y + (v.z+dv2.z)*anf->b.x
       - (v.x+dv2.x)*anf->b.z) * delt;
  dv3.z = k * (anf->e.z + (v.x+dv2.x)*anf->b.y
       - (v.y+dv2.y)*anf->b.x) * delt;

  /* assign calculated values into tmp */
  tmp->vel.x = v.x + (dv0.x+2*dv1.x+2*dv2.x+dv3.x)/6.0;
  tmp->vel.y = v.y + (dv0.y+2*dv1.y+2*dv2.y+dv3.y)/6.0;
  tmp->vel.z = v.z + (dv0.z+2*dv1.z+2*dv2.z+dv3.z)/6.0;
  tmp->pos.x = traj->rear->pos.x + tmp->vel.x * delt;
  tmp->pos.y = traj->rear->pos.y + tmp->vel.y * delt;
  tmp->pos.z = traj->rear->pos.z + tmp->vel.z * delt;

  tmp->dt = delt;   /* change in time = delt */
  tmp->t = traj->rear->t + delt;/*total time = prev_time + delt*/
  tmp->prev=NULL;
  tmp->next=NULL;
  free(anf);        /* free the field */
  return tmp;
}

/********* lookup functions ***********/
/*
 * function:     postition
 * returns:      Vector
 * arguments:    List step
 * description:  This function returns the position vector of step.
 */
Vector position(List step)
{
  return step->pos;
}

/*
 * function:     length
 * returns:      double
 * arguments:    Vector
 * description:  Returns the amplitude of vector k
 */
double length(Vector k)
{
  return sqrt((pow(k.x,2)+pow(k.y,2)+pow(k.z,2)));
}

/*
 * function:     get_traj_length
 * returns:      int
 * arguments:    TrajectoryPtr traj
 * description:  Returns length of trajectory
 */
int get_traj_length(TrajectoryPtr traj)
{
```

```
  return traj->num;
}

/*
 * function:     min_vector
 * returns:      double
 * arguments:    doubles... individual vector components (x,y,z)
 * description:  Returns the mininum of x,y,z
 */
double min_vector(double x, double y, double z)
{
  if(fabs(x) < fabs(y))
    if(fabs(x) < fabs(z))
      return x;
  else if(fabs(y) < fabs(z))
    return y;
  return z;
}

/*
 * function:     min
 * returns:      double
 * arguments:    double x, double y
 * description:  Returns the mininum of x,y
 */
double min(double x, double y)
{
  if(x<y)
    return x;
  return y;
}

/********* guessing dt functions ***********/

/*
 * function:     guess_dt1
 * returns:      double
 * arguments:    double x, double y
 * description:  Returns a guess for the incremental change in time,
 *               based upon an adaptive scheme, of the octree box
 *               size and the current velocity.
 */
double guess_dt1(List step,  Bounds b, double init_guess)
{
  return
    init_guess*pow(2,-search(oct->top,
     axis2oct(step->pos, oct))->depth)/
       vec2scalar(step->vel);
}


/*
 * function:     guess_dt2
 * returns:      double
```

```
 * arguments:    Initiale ch, Fields f
 * description:  Returns a guess for the incremental change in time,
 *               based upon the accelaration of particle.
 *               (doesn't work very well)
 */
double guess_dt2(Initiale ch, Fields f)
{
  Vector a, v;
  v = ch.vel;
  a = acceleration(ch, f);

  return length(v)/length(a);
}


/*
 * function:     guess_dt
 * returns:      double
 * arguments:    double x, double y
 * description:  Returns a guess for the incremental change in time,
 *               based upon the accelaration of particle.
 *               (doesn't work very well)
 */
double guess_dt3(Bounds b, int no)
{
  /* incremental change in time based upon minimum boundary */
  return 1e-6*fabs(min_vector(b.xmax-b.xmin,
      b.ymax-b.ymin,
      b.zmax-b.zmin)/(no-1));
}


/*
 * function:     acceleration
 * returns:      acceleration vector
 * arguments:    Initiale ch, Fields f
 * description:  calculates acceleration vector based
 *               upon fields and electron paramenters.
 */
Vector acceleration(Initiale ch, Fields f)
{
  Vector a;
  double k = ch.q/ch.m;

  a.x = k * (f.af.e.x + ch.vel.y*f.af.b.z - ch.vel.z*f.af.b.y);
  a.y = k * (f.af.e.y + ch.vel.z*f.af.b.x - ch.vel.x*f.af.b.z);
  a.z = k * (f.af.e.z + ch.vel.x*f.af.b.y - ch.vel.y*f.af.b.x);

  return a;
}

/********* interpolation functions  ***********/
/*
 * function:     difference
```

```
 * returns:      double
 * arguments:    TrajectoryPtr t1, TrajectoryPtr t2, int sampling_no
 * description:  Returns the maximum difference between the (linear) time
 *               interpolations of trajectories t1 and t2.  For constant
 *               increment time, which is determined by the total time
 *               divided by sampling number.
 */
double difference(TrajectoryPtr traj1, TrajectoryPtr traj2,
  Bounds bd, int sampling_no)
{
  double diff, max_diff = 0.0;
  double xlength, ylength, zlength;
  double time, delt;
  Vector pos1, pos2, pos3;
  TrajectoryPtr t1, t2;

  xlength = bd.xmax - bd.xmin; /* length of x boundary */
  ylength = bd.ymax - bd.ymin; /* length of y boundary */
  zlength = bd.zmax - bd.zmin; /* length of z boundary */

  t1=cp_trajectory(traj1);
  t2=cp_trajectory(traj2);
  /* take the minimum of the two trajectory times */
  time = min(t1->rear->t, t2->rear->t);
  delt = time/(sampling_no-1); /* sample at these time increments*/

  /* as long as the end of either the trajectory is not reached
     continue to compare differences */
  while(t1->front != NULL && t2->front !=NULL){
    /* linearly interpolate both trajectory lists
       to get the exact position at exact sampled time */
    pos1 = iter(t1, time, linear_interp);
    pos2 = iter(t2, time, linear_interp);
    /* move on to next list elements, which are later in time */
    t1->front=t1->front->next;
    t2->front=t2->front->next;
    /* take the vector difference of the two positions which were
       found linearly interpolated in time */
    pos3 = scalar2vec((pos1.x-pos2.x)/xlength,
      (pos1.y-pos2.y)/ylength,
      (pos1.z-pos2.z)/zlength);
    if((diff=length(pos3)) > max_diff)
      max_diff=diff;
  }
  destroy_trajectory(t1);
  destroy_trajectory(t2);

#ifdef DEBUG2
  printf("\nmax_diff is %20.8e delta %20.8e\n",
 max_diff, traj2->rear->dt);
#endif
  return max_diff/sqrt(3.0); /* return normalized max difference */
}
```

```
/*
 * function:     iter
 * returns:      Vector (position)
 * arguments:    TrajectoryPtr traj, double time, Vector (*method)
 * description:  Returns the interpolated position at time, using
 *               the method specified by user.
 */
Vector iter(TrajectoryPtr traj, double time,
    Vector (*method)(TrajectoryPtr traj, double time))
{
  TrajectoryPtr tmp;

  if(traj->front->next == NULL) /* return position if at end of list */
    return traj->front->pos;

  /* find trajectory element with correct time span
   * while time in tmp is less than the target time
   * next element of tmp is the new front of list */
  tmp = traj;
  while(tmp->front->next->t < time)
    tmp->front = tmp->front->next;

                            /* if the trajectory time is equal to target */
  if(tmp->front->t == time) /* time then interpolation not necessary, */
    return tmp->front->pos; /* return position */

  /* if trajectory time does not exactly match target time,
   *  then execute interpolation. */
  return method(tmp, time);
}

/*
 * function:     linear_interp
 * returns:      Vector (position)
 * arguments:    TrajectoryPtr traj, double time
 * description:  Returns the linearly interpolated position at time.
 */
Vector linear_interp(TrajectoryPtr traj, double time)
{
  Vector tpos, slope;
  Vector pos1 = traj->front->pos;
  Vector pos2 = traj->front->next->pos;
  double delt = traj->front->dt;
  double incr = fmod(time,delt);

  /* the slope is the average of the two positions (or the first
   * position which corresponds to before the time of sampling,
   * and the second, which corresponds to after the sampling time),
   * divided by incremental time */
  slope.x = (pos2.x-pos1.x)/delt;
  slope.y = (pos2.y-pos1.y)/delt;
  slope.z = (pos2.z-pos1.z)/delt;

  /* the interpolated position is the previous position
```

```
   * plus the difference in time (incr) times the slope
   * of the line drawn between the two positions */
  tpos.x = pos1.x + slope.x*incr;
  tpos.y = pos1.y + slope.y*incr;
  tpos.z = pos1.z + slope.z*incr;

  return tpos;
}

/*
 * function:     difference2
 * returns:      double
 * arguments:    TrajectoryPtr t1, TrajectoryPtr t2
 * description:  Returns the maximum difference between the (linear)
 *               time interpolations of trajectories t1 and t2.
 *               For variable incremental time.  t1 is the coarser
 *               of the two trajectories.  Sample at each point
 *               of the coarser trajectory.
 */
double difference2(TrajectoryPtr traj1, TrajectoryPtr traj2, Bounds bd)
{
  double diff, max_diff = 0;
  double xlength, ylength, zlength;
  double time = 0.0;
  Vector pos1, pos2, pos3;
  TrajectoryPtr t1, t2;

  xlength = bd.xmax - bd.xmin; /* length of x boundary */
  ylength = bd.ymax - bd.ymin; /* length of y boundary */
  zlength = bd.zmax - bd.zmin; /* length of z boundary */

  t1=cp_trajectory(traj1);
  t2=cp_trajectory(traj2);

  while(t1->front != NULL && t2->front !=NULL
&& time<=t1->rear->t && time<=t2->rear->t){
    pos1 = t1->front->pos;
    /* sample at the rate of the coarser trajectory */
    pos2 = iter2(t2, t1->front->t, t1->front->dt, linear_interp);

    while(t2->front->t < t1->front->t)
      t2->front=t2->front->next;
    t1->front=t1->front->next;
    /* take the vector difference of the two positions which were
       found linearly interpolated in time */
    pos3 = scalar2vec((pos1.x-pos2.x)/xlength,
      (pos1.y-pos2.y)/ylength,
      (pos1.z-pos2.z)/zlength);
    if((diff=length(pos3)) > max_diff)
      max_diff=diff;
  }
  destroy_trajectory(t1);
  destroy_trajectory(t2);
#ifdef DEBUG2
```

```
  printf("\nmax_diff is %20.8e delta %20.8e",
 max_diff, traj2->rear->dt);
#endif
  return max_diff/sqrt(3.0);
}

/*
 * function:     iter2
 * returns:      Vector (position)
 * arguments:    TrajectoryPtr traj, double time, Vector (*method)
 * description:  Returns the interpolated position at time, using
 *               the method specified by user.
 */
Vector iter2(TrajectoryPtr traj, double time, double dt,
    Vector (*method)(TrajectoryPtr traj, double time))
{
  TrajectoryPtr tmp;

  if(traj->front->next == NULL)  /* return position if at end of list */
    return traj->front->pos;

  tmp = traj;
  /* find trajectory element with correct time span
   * while time in tmp is less than the target time
   * next element of tmp is the new front of list */
  while(tmp->front->next->t < time)
    tmp->front = tmp->front->next;

                              /* if the trajectory time is equal to target */
  if(tmp->front->t == time)   /* time then interpolation not necessary, */
    return tmp->front->pos;   /* return position */

  /* if trajectory time does not exactly match target time, then
     execute interpolation... */
  return method(tmp, time);
}


/********* other functions ***********/

/*
 * function:     trajectory2array
 * returns:      An array of listelems
 * arguments:    TrajectoryPtr t,
 * description:  Given a trajectory, creates an array of listelems
 *               based upon the list of that trajectory.
 */
List *trajectory2arry(TrajectoryPtr t)
{
  List *a;
  int i = get_traj_length(t);

  /* dynamic memory allocation */
  if((a = malloc(i*sizeof(ListelemPtr))) == NULL){
```

```
    printf("Error using malloc.  Exiting program\n");
    exit(0);
  }

  for(i=0; i<get_traj_length(t); i++)
    if((a[i] = malloc(sizeof(Listelem))) == NULL){
      printf("Error using malloc.  Exiting program\n");
      exit(0);
    }

  for(i=0; i<get_traj_length(t); i++){
    a[i] = t->front;
    t->front=t->front->next;
  }
  return a;
}

/*
 * function:     coarse
 * returns:      double
 * arguments:    TrajectoryPtr t1, TrajectoryPtr t2
 * description:  Returns the coarser trajectory of t1 and t2
 */
TrajectoryPtr coarse(TrajectoryPtr t1, TrajectoryPtr t2)
{
  if(t1->front->dt > t2->front->dt)
    return t1;
  return t2;
}

/****************** boundary checking ********************/
/*
 * function:     checkbounds
 * returns:      TRUE or FALSE
 * arguments:    Vector p, Bounds b ... position p and bounds b
 * description:  This function returns true if the position of
 *               step is inside bounds and false otherwise.
 */
int checkbounds(Vector p, Bounds b)
{
  if(p.x <= b.xmax && p.x >= b.xmin
     && p.y <= b.ymax && p.y >= b.ymin
     && p.z <= b.zmax && p.z >= b.zmin)
    return TRUE;
  return FALSE;
}


/*
 * function:     bound_position
 * returns:      Updated List
 * arguments:    List in, trajectory elementinside boundary
 *               List out, trajectory element outside boundary
 *               Bounds bd, boundary of envirnment
```

```
 * description:  gives the linearly interpolated trajectory
 *               position directly at the boundary
 */
List bound_position(List in, List out, Bounds bd)
{
  double xslope, yslope, zslope;
  double dx_vel, dy_vel, dz_vel;

  dx_vel = (out->vel.x-in->vel.x)/out->dt;
  dy_vel = (out->vel.y-in->vel.y)/out->dt;
  dz_vel = (out->vel.z-in->vel.z)/out->dt;

  /* linearly interpolate position if the
   * electron hits the maximum x boundary */
  if(out->pos.x > bd.xmax){
    xslope = (out->pos.x - in->pos.x)/out->dt;
    yslope = (out->pos.y - in->pos.y)/out->dt;
    zslope = (out->pos.z - in->pos.z)/out->dt;
    out->dt = out->dt*fabs((bd.xmax-in->pos.x)/(out->pos.x-in->pos.x));
    out->pos.x = bd.xmax;
    out->pos.y = in->pos.y + yslope*out->dt;
    out->pos.z = in->pos.z + zslope*out->dt;
  }
  /* linearly interpolate position if the
   * electron hits the minimum x boundary */
  else if(out->pos.x < bd.xmin){
    xslope = (out->pos.x - in->pos.x)/out->dt;
    yslope = (out->pos.y - in->pos.y)/out->dt;
    zslope = (out->pos.z - in->pos.z)/out->dt;
    out->dt = out->dt*fabs((bd.xmin-in->pos.x)/(out->pos.x-in->pos.x));
    out->pos.x = bd.xmin;
    out->pos.y = in->pos.y + yslope*out->dt;
    out->pos.z = in->pos.z + zslope*out->dt;
  }
  /* linearly interpolate position if the
   * electron hits the maximum y boundary */
  if(out->pos.y > bd.ymax){
    xslope = (out->pos.x - in->pos.x)/out->dt;
    yslope = (out->pos.y - in->pos.y)/out->dt;
    zslope = (out->pos.z - in->pos.z)/out->dt;
    out->dt = out->dt*fabs((bd.ymax-in->pos.y)/(out->pos.y-in->pos.y));
    out->pos.x = in->pos.x + xslope*out->dt;
    out->pos.y = bd.ymax;
    out->pos.z = in->pos.z + zslope*out->dt;
  }
  /* linearly interpolate position if the
   * electron hits the minimum y boundary */
  else if(out->pos.x < bd.xmin){
    xslope = (out->pos.x - in->pos.x)/out->dt;
    yslope = (out->pos.y - in->pos.y)/out->dt;
    zslope = (out->pos.z - in->pos.z)/out->dt;
    out->dt = out->dt*fabs((bd.ymin-in->pos.y)/(out->pos.y-in->pos.y));
    out->pos.x = in->pos.x + xslope*out->dt;
    out->pos.y = bd.ymin;
```

```
    out->pos.z = in->pos.z + zslope*out->dt;
  }
  /* linearly interpolate position if the
   * electron hits the maximum z boundary */
  if(out->pos.z > bd.zmax){
    xslope = (out->pos.x - in->pos.x)/out->dt;
    yslope = (out->pos.y - in->pos.y)/out->dt;
    zslope = (out->pos.z - in->pos.z)/out->dt;
    out->dt = out->dt*fabs((bd.zmax-in->pos.z)/(out->pos.z-in->pos.z));
    out->pos.x = in->pos.x + xslope*out->dt;
    out->pos.y = in->pos.y + yslope*out->dt;
    out->pos.z = bd.zmax;
  }
  /* linearly interpolate position if the
   * electron hits the minimum z boundary */
  else if(out->pos.z < bd.zmin){
    xslope = (out->pos.x - in->pos.x)/out->dt;
    yslope = (out->pos.y - in->pos.y)/out->dt;
    zslope = (out->pos.z - in->pos.z)/out->dt;
    out->dt = out->dt*fabs((bd.zmin-in->pos.z)/(out->pos.z-in->pos.z));
    out->pos.x = in->pos.x + xslope*out->dt;
    out->pos.y = in->pos.y + yslope*out->dt;
    out->pos.z = bd.zmin;
  }
  out->vel.x = in->vel.x + dx_vel*out->dt;
  out->vel.y = in->vel.y + dy_vel*out->dt;
  out->vel.z = in->vel.z + dz_vel*out->dt;

  out->t = in->t + out->dt;
  return out;
}

/**************** current distribution ***************/

/*
 * function:     func
 * returns:      void
 * arguments:    Initiale ch, Fields f, Bounds b, double max_error_per...
 * description:  calculates trajectory through a range of initial
 *               thetas, phis
 */
void func(Initiale ch, Fields f, Bounds b, double max_error_per,
  double energy, int num_traj, double degree,
  int gauss, double variance)
{

  TrajectoryPtr traj;
  /*  double theta, phi;*/
  double v_t = energy2vel(energy, ch);
  int num =0 ;
#ifdef DEBUG
  time_t before, after;
#endif
```

```
  for(num = 0; num < num_traj; num++){
    if(gauss == TRUE)
      /* perform gaussian distribution if that is specified */
      ch.vel = gaussian(v_t, degree, 0.0, variance);
    /* else perform uniform distribution*/
    else  ch.vel = uniform(v_t, degree);
    init_trajectory(&traj, ch);

#ifdef DEBUG
    time(&before);
#endif
    traj = transit(traj, ch, f, b, max_error_per);
#ifdef DEBUG
    time(&after);
#endif

#ifdef DEBUG2
    printf("trajectory #%d count %d time %f secs.\n",
   num, get_traj_length(traj),difftime(after, before));
#endif
    destroy_trajectory(traj);
  }
}

/*
 * function:     uniform_dist
 * returns:      void
 * arguments:    Initiale ch, Fields f, Bounds b, double max_error_per...
 * description:  calculates trajectory through a uniformly distributed
 *               range of thetas (from zero to degree) and phis (from
 *               zero to 360 degrees).
 */
void uniform_dist(Initiale ch, Fields f, Bounds b,
  double max_error_per, double energy, int num_traj,
  double degree, int gauss, double variance)
{

  TrajectoryPtr traj;
  double v_t = energy2vel(energy, ch);
  int num =0 ;

  for(num = 0; num < num_traj; num++){
    ch.vel = uniform(v_t, degree);
    init_trajectory(&traj, ch);

    traj = transit(traj, ch, f, b, max_error_per);
#ifdef DEBUG2
    printf("trajectory #%d count %d time %f secs.\n",
   num, get_traj_length(traj),difftime(after, before));
#endif
    destroy_trajectory(traj);
  }
}
```

```
/*
 * function:     gaussian_dist
 * returns:      void
 * arguments:    Initiale ch, Fields f, Bounds b, double max_error_per...
 * description:  calculates trajectories with a gaussian distributed
 *               range of thetas (from zero to degree) and a uniform phi
 *               (from zero to 360 degrees).
 */
void gaussian_dist(Initiale ch, Fields f, Bounds b,
  double max_error_per, double energy, int num_traj,
  double degree, int gauss, double variance)
{

  TrajectoryPtr traj;
  double v_t = energy2vel(energy, ch);
  int num =0 ;

  for(num = 0; num < num_traj; num++){
    ch.vel = gaussian(v_t, degree, 0.0, variance);
    init_trajectory(&traj, ch);

    traj = transit(traj, ch, f, b, max_error_per);
#ifdef DEBUG2
    printf("trajectory #%d count %d time %f secs.\n",
   num, get_traj_length(traj),difftime(after, before));
#endif
    destroy_trajectory(traj);
  }
}

/*
 * function:     sphere2cart
 * returns:      velocity vector
 * arguments:    double vel, double theta, double phi
 *               (in spherical coordinates)
 * description:  Returns velocity vector in xyz cooridinates
 */
Vector sphere2cart(double vel, double theta, double phi)
{
  Vector v;
  v.x = vel*sin(theta*PI/180.0)*cos(phi*PI/180.0);
  v.y = vel*sin(theta*PI/180.0)*sin(phi*PI/180.0);
  v.z = vel*cos(theta*PI/180.0);
  return v;
}

/*
 * function:     uniform
 * returns:      random uniformly distributed velocity vector
 * arguments:    double initial velocity
 * description:  returns random velocity in semi-sphere distribution
 */
Vector uniform(double vel, double degree)
{
```

```
  double theta, phi;

  /* phi ranges from 0 to 360 */
  phi=(double)rand()/RAND_MAX*360;
  /* theta ranges from 0 to degree */
  theta=(double)rand()/RAND_MAX*degree;
  return sphere2cart(vel, theta, phi);
}

/*
 * function:     gaussian
 * returns:      Gaussian velocity vector
 * arguments:    Vector vel, double mean, double variance
 * description:
 */
Vector gaussian(double vel, double degree, double mean, double variance)
{
  int test = 0;
  double theta, phi;

  while(test == 0){
    /* assign value for theta between 0 and degree */
    theta = (double) rand()/RAND_MAX*degree;
    /* if the probability D(theta) is less than a randomly generated
       number (uniformly distributed from 0 to 1), use the value for
       theta.  Else pick another theta */
    if((exp(-0.5*pow((theta-mean)/variance,2)/(variance*sqrt(2*PI))))
       > (double)rand()/RAND_MAX)
      test = 1;
  }
 /* value for phi between 0 and 360*/
  phi = (double) rand()/RAND_MAX*360;
 /* convert to cartesian coordinates */
  return sphere2cart(vel, theta, phi);
}

/* current density distribution */

/*
 * function:    fowler_nordeim
 * returns:     double Fowler-Nordheim current density
 * arguments:   double elecfield, double phi -- elecfield=electric field
 * description: returns fowler-Nordheim equation for calculating current
 *              density.
 */
double fowler_nordheim(double elecfield, double phi)
{
  double A = 1.54e-6;
  double B = 6.87e7;
  double t_sq = 1.1;
  double y = (3.79e-4)*sqrt(elecfield)/phi;
  double v = 0.95 - pow(y,2);

  return A*pow(elecfield,2)*exp(-B*pow(phi,1.5*v)/elecfield)/(phi*t_sq);
```

```
}
```

Appendix C

octree.h

```
/*
 *  name of this file:    pixel/src/traj1/octree.h
 */

/** octree-related  -- function prototypes **/

/* initialization functions */
void init_scale(Vector *scale);
OctreePtr make_octree(Afield *f,
      Bounds bd);
BoxPtr make_topbox(Afield *f,
   OctreePtr oc,
   Vector coor,
   Bounds bd);
BoxPtr make_subbox(BoxPtr parent,
   OctreePtr oc,
   Afield *f,
   Vector coor,
   Bounds bd);
BoxPtr make_box(BoxPtr parent,
OctreePtr oc,
Afield *f,
Vector coor,
Bounds bd,
int depth);
void init_octree(OctreePtr *oc);
void init_box(BoxPtr *b);
void init_subboxes(BoxPtr b);
void init_boxfield(BoxPtr *b, OctreePtr oc, Afield *f,
   Vector oct_coor, int depth);
void init_data(DataPtr *dat);

/* freeing memory */
void destroy_octree(OctreePtr oc);
```

```
void destroy_box(BoxPtr box);

/* lookup functions */
BoxPtr whereami(OctreePtr oc, Vector coor);
BoxPtr search(BoxPtr box, Vector coor);
int whichoct(BoxPtr box, Vector coor);

/* boundary checking */
int checkoctbounds(OctreePtr oc, Vector coor, Bounds bd);

/* refine */
void refine(BoxPtr top, OctreePtr oc, Afield *f, int depth, Bounds bd);
void refine2perc(BoxPtr top, OctreePtr oc,
 Afield *f, double percentage, Bounds bd);
void refine2point(BoxPtr top, OctreePtr oc, Afield *f,
  Vector coor, int depth, Bounds bd);
void refine2pointperc(BoxPtr top, OctreePtr oc, Afield *f,
      Vector axis_coor, double percentage, Bounds bd);
void refinebds(BoxPtr top, OctreePtr oc, Afield *af, int depth,
       double incr, Bounds bd);
void refinetopbds(BoxPtr top, OctreePtr oc,
  Afield *af, int depth, Bounds bd);
void refinesemi(BoxPtr top, OctreePtr oc, Afield *af, int depth,
double incr, double R, double r_o, Bounds bd);
/* refine auxiliary functions */
void refpoint(BoxPtr top, OctreePtr oc, Afield *f,
      Vector target, int depth, Bounds bd);
void refpointperc(BoxPtr top, OctreePtr oc, Afield *f,
  Vector axis_coor, double percentage, Bounds bd);

/* print functions */
void print_box(BoxPtr top, OctreePtr oc);
void print_box_pos(BoxPtr top, OctreePtr oc);
void print_box_data(BoxPtr top, OctreePtr oc);
void print_top_data2file(BoxPtr top, OctreePtr oc,
 Bounds bd, char filename[50]);
void print_coor(Vector coor);

/* functions for assigning analytic fields */
Afield *field_lookup(OctreePtr oc, Vector axis_coor);
Afield *fieldop(Afield *af, Vector point);
Afield *fieldop_oct(OctreePtr oc, Vector point);
Afield *field_copy(Afield *af);
Afield *oct_fieldop(Fields f, Vector point);

/** functions for assigning analytic fields **/
Afield *constant_fieldop(Afield *af, Vector point);
Afield *linear_fieldop(Afield *f, Vector point);
Afield *quadratic_fieldop(Afield *f, Vector point);
Afield *cone_fieldop(Afield *f, Vector point,
     double r_o, double R,
     double V_a, double V_g,
     double d);
```

```
double linear_field(double x, double slope, double b);
double quadratic_field(double x, double a);

/** functions to write data into octree structure **/
void write_data(OctreePtr oc, Vector coor, Vector vel, Initiale c);
DataPtr add_data(DataPtr dp, Data *new);
int different_box(OctreePtr oc, Vector p1, Vector p2);

/********** Miscellaneous **********/
double vec2scalar(Vector vel);
Vector scalar2vec(double x, double y, double z);
Vector oct2axis(Vector coor, OctreePtr oc);
Vector axis2oct(Vector axis_coor, OctreePtr oc);
double vec2energy(Vector vel, Initiale c);
double energy2vel(double energy, Initiale c);
```

Appendix D

octree.m

```
/*
 *  name of this file: /pixel/src/traj1/octree.m
 */

#include "initial.m"
#include "octree.h"
#include <time.h>

#define MAXDEPTH 20
#define MINDEPTH 2

OctreePtr oct;
unsigned int count = 0;
double V_A;
double V_G;

/********** initialization functions **********/

/*
 * function:     init_axis
 * returns:      void
 * arguments:    MapPtr *axis, Bounds bd
 * description:  Initializes the component of octree which retains
 *               information for the scaling of the octree axis.
 */
void init_axis(MapPtr *axis, Bounds bd)
{
  if((*axis = malloc(sizeof(Map))) == NULL){
    printf("Error using malloc in init_axis.  Exiting program\n");
    exit(0);
  }
  if(((*axis)->scale = malloc(sizeof(Vector))) == NULL){
    printf("Error using malloc in init_axis.  Exiting program\n");
    exit(0);
```

```
  }
  if(((*axis)->offset = malloc(sizeof(Vector))) == NULL){
    printf("Error using malloc in init_axis.  Exiting program\n");
    exit(0);
  }

  (*axis)->scale->x = bd.xmax - bd.xmin;
  (*axis)->scale->y = bd.ymax - bd.ymin;
  (*axis)->scale->z = bd.zmax - bd.zmin;
  if((*axis)->scale->x <= 0.0
     || (*axis)->scale->y <= 0.0
     || (*axis)->scale->z <= 0.0){
    printf("Error: scale must be greater than zero.  Exiting program\n");
    exit(0);
  }
  (*axis)->offset->x = bd.xmin;
  (*axis)->offset->y = bd.ymin;
  (*axis)->offset->z = bd.zmin;
}

/*
 * function:     make_octree
 * returns:      OctreePtr
 * arguments:    Afield *f, Bounds bd
 * description:  Allocates and Initializes the Octree Structure
 */
OctreePtr make_octree(Afield *f,
      Bounds bd)
{
  OctreePtr new_oct;
  Vector coor;

  init_octree(&new_oct);
  init_axis(&new_oct->axis, bd);
  coor.x = coor.y = coor.z = 0.0;
  new_oct->top =  make_topbox(f, new_oct, coor, bd);
  return new_oct;
}


/*
 * function:     make_topbox
 * returns:      BoxPtr
 * arguments:    Afield *f,  OctreePtr oc,  Vector coor, Bounds bd
 * description:  Allocates and Initializes the top box in an Octree
 *               This is the parent box of all boxes
 */
BoxPtr make_topbox(Afield *f,
   OctreePtr oc,
   Vector coor,
   Bounds bd)
{
  BoxPtr topbox = NULL;
  return make_box(topbox, oc, f, coor, bd, 1);
```

```
}


/*
 * function:     make_subbox
 * returns:      BoxPtr
 * arguments:    BoxPtr parent, OctreePtr oc, Afield *f,
 *               Vector coor, Bounds bd
 * description:  Allocates and Initializes any box in an Octree,
 *               except for the top box, which does not have a
 *               parent box.
 */
BoxPtr make_subbox(BoxPtr parent,
   OctreePtr oc,
   Afield *f,
   Vector coor,
   Bounds bd)
{
  BoxPtr box;
  box = make_box(parent, oc, f, coor, bd, parent->depth+1);
  (box->up->num_terminated)--;
  return box;
}

/*
 * function:     make_box
 * returns:      BoxPtr
 * arguments:    BoxPtr parent, OctreePtr oc, Afield *f,
 *               Vector coor, Bounds bd, int depth
 * description:  This is an auxiliary function for allocation
 *               and initiating boxes.
 */
BoxPtr make_box(BoxPtr parent,
OctreePtr oc,
Afield *f,
Vector oct_coor,
Bounds bd,
int depth)
{
  BoxPtr new_box;
  init_box(&new_box);
  new_box->up = parent;
  init_subboxes(new_box);
  /* data contains information of electrons thru box */
  init_data(&(new_box->dat));
  new_box->depth = depth;     /* depth of current box */
  /* coordindate of bottom left corner of box */
  new_box->coordinate = oct_coor;
  /* fields at box's corners */
  init_boxfield(&new_box, oc, f, oct_coor, depth);

  if(checkoctbounds(oc, oct_coor, bd))
    new_box->isboundary = 0;
  else  new_box->isboundary = 1;
```

```
  if(new_box->isboundary)
    new_box->tag = 1;
  else new_box->tag = 0;
  return new_box;
}


/*
 * function:     init_octree
 * returns:      void
 * arguments:    OctreePtr *new_oct, pointer to uninitialized Octree
 * description:  Allocates memory for the Octree.
 */
void init_octree(OctreePtr *new_oct)
{
  /* exit if there are memory allocation problems */
  if ((*new_oct = malloc(sizeof(Octree))) == NULL)  {
    printf("Memory allocation error for new octree.  Exiting program.\n");
    exit(-1);
  }
  (*new_oct)->top = NULL;  /* ptr to top (largest) box */
  (*new_oct)->count = 0;   /* total number of boxes, including all layers */
  (*new_oct)->depth = 0;   /* largest depth */
  /* denotes whether boundary is inside octree */
  (*new_oct)->isboundary = FALSE;
  (*new_oct)->axis = NULL;
}

/*
 * function:     init_box
 * returns:      void
 * arguments:    BoxPtr *b, pointer to uninitialized box
 * description:  Allocates memory for the Box
 */
void init_box(BoxPtr *b)
{
  int i;
  Vector coor;
  if ((*b = malloc(sizeof(Box))) == NULL)  {
    printf("Memory allocation error for new box element.");
    printf("Exiting program.\n");
    printf("Number of boxes == %d\n", count);
    exit(-2);
  }
count++;
  init_subboxes(*b);

  for(i=0; i<8; i++)
    (*b)->fd[i] = NULL;     /* field at box's corners */
  (*b)->depth = 0;          /* depth of box */
  (*b)->num_terminated = 8; /* number of terminated subboxes */
  coor.x = coor.y = coor.z = 0.0;
  (*b)->coordinate = coor;  /* coordindate of bottom left corner of box */
  /* boolean, denotes whehter boundary is inside octree */
```

```
  (*b)->isboundary = FALSE;
}


/*
 * function:     init_data
 * returns:      void
 * arguments:    DataPtr *p, pointer to uninitialized Data
 * description:  Allocates memory for the Box
 */
void init_data(DataPtr *dat)
{
  if ((*dat = malloc(sizeof(Data))) == NULL) {
    printf("Memory allocation error for new data element.");
    printf("  Exiting program.\n");
    exit(-2);
  }
  (*dat)->num = 0;       /* number of data_elem in the link list */
  (*dat)->energy = 0.0;  /* velocity of trajectory element */
}

/*
 * function:     init_subboxes
 * returns:      void
 * arguments:    BoxPtr b
 */
void init_subboxes(BoxPtr b)
{
  int i;
  for(i=0; i<8; i++)
    b->subboxes[i] = NULL;
}

/*
 * function:     init_boxfield
 * returns:      BoxPtr *b, OctreePtr oc, Afield *f,
 *  Vector oct_coor, int depth
 * arguments:    Initializes all eight fields in the corner of the box
 */
void init_boxfield(BoxPtr *b, OctreePtr oc, Afield *f,
   Vector oct_coor, int depth)
{
  int i;
  Vector corner;
  for(i=0; i<8; i++){
    if (((*b)->fd[i] = malloc(sizeof(struct analytical_field)))==NULL){
      printf("Memory allocation error for new analytical field.");
      printf("Exiting program.\n");
      printf("Number of boxes == %d\n", count);
      exit(-2);
    }
    corner = scalar2vec(oct_coor.x+pow(2,-depth)*floor(i/4),
oct_coor.y+pow(2,-depth)*floor(fmod(i,4)/2),
oct_coor.z+pow(2,-depth)*fmod(i,2));
```

```
    (*b)->fd[i] = fieldop(f, oct2axis(corner, oc));
  }
}

/*************** delete functions **************/

/*
 * function:     destroy_octree
 * returns:      void
 * arguments:    OctreePtr oc
 * description:  frees the octree
 */
void destroy_octree(OctreePtr oc)
{
  destroy_box(oc->top);
  free(oc);
}

/*
 * function:     destroy_box
 * returns:      void
 * arguments:    BoxPtr box
 * description:  frees the box
 */
void destroy_box(BoxPtr box)
{
  int i;
  for(i=0; i<8; i++){
    if(box->subboxes[i] != NULL){
      destroy_box(box->subboxes[i]);
      if(box->up != NULL)
(box->up->num_terminated)++;
    }
    if(box->fd[i] != NULL)
      free(box->fd[i]);
  }
  free(box->fd);
  free(box->dat);
  free(box);
}

/*************** lookup functions **************/

/*
 * function:     whereami
 * returns:      BoxPtr
 * arguments:    OctreePtr oc, Vector axis_coor
 * description:  given a coordinate point in the FED device
 *               volume gives the boxptr to the corresponding
 *               Octree box
 */
BoxPtr whereami(OctreePtr oc, Vector axis_coor)
{
  return search(oc->top, axis2oct(axis_coor, oc));
```

```
}


/*
 * function:     search
 * returns:      BoxPtr
 * arguments:    BoxPtr box, Vector coor
 * description:  given a coordinate point in the Octree itself
 *               this function gives the boxptr to the corresponding
 *               Octree box
 */
BoxPtr search(BoxPtr box, Vector coor)
{
  if(box == NULL)
    return box;
  if(box->subboxes[whichoct(box,coor)] == NULL)
    return box;
  else
    return search(box->subboxes[whichoct(box,coor)], coor);
}


/*
 * function:     whichoct
 * returns:      int
 * arguments:    BoxPtr box, Vector coor
 * description:  gives integer number used reference the subbox
 *               of box, in which coor resides
 */
int whichoct(BoxPtr box, Vector coor)
{
  int val = 0;
  if(coor.x >= (box->coordinate.x+pow(2,-(box->depth))))
    val += 4;
  if(coor.y >= (box->coordinate.y+pow(2,-(box->depth))))
    val += 2;
  if(coor.z >= (box->coordinate.z+pow(2,-(box->depth))))
    val += 1;
  return val;
}


/*
 * function:     checkoctbounds
 * returns:      boolean,
 * arguments:    OctreePtr oc, Vector coor, Bounds bd
 * description:  returns true if coor, is inside the limits
 *               described by bounds, and false otherwise.
 */
int checkoctbounds(OctreePtr oc, Vector coor, Bounds bd)
{
  Vector p = oct2axis(coor, oc);

  if(p.x < bd.xmax && p.x > bd.xmin
```

```
     && p.y < bd.ymax && p.y > bd.ymin
     && p.z < bd.zmax && p.z > bd.zmin)
    return TRUE;
  else return FALSE;
}

/****************** refine *******************/

/*
 * function:    refine
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc, Afield *f, int depth, Bounds bd
 * description: uniformly refines the entire volume of the FED device,
 *              with a level of subboxes at (depth-1).
 */
void refine(BoxPtr top, OctreePtr oc, Afield *f, int depth, Bounds bd)
{
  int i;
  Vector coor;

  if(depth < 1 || top->depth >= MAXDEPTH || top == NULL)
    return;

  for(i=0; i<8; i++){
    coor = scalar2vec((top->coordinate.x+pow(2,-(top->depth))*floor(i/4)),
      (top->coordinate.y+
       pow(2,-(top->depth))*floor(fmod(i,4)/2)),
      (top->coordinate.z+pow(2,-(top->depth))*fmod(i,2)));
    if(top->subboxes[i] == NULL)
      top->subboxes[i] = make_subbox(top, oc, f, coor, bd);
    refine(top->subboxes[i], oc, f, depth-1, bd);
  }
}


/*
 * function:    refine2perc
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc, Afield *f,
 *              double percentage, Bounds bd
 * description: refines the entire volume of the FED device, to
 *              a level of subboxes, whose neighboring fields
 *              vary below the specified percentage.
 */
void refine2perc(BoxPtr top, OctreePtr oc,
 Afield *f, double percentage, Bounds bd)
{
  int i, j;
  Vector coor, neigh;
  Afield *af, *top_af;
  double per = percentage/100.0;

  if(top->depth >= MAXDEPTH || top == NULL)
    return;
```

```
  for(i=0; i<8; i++){
    coor = scalar2vec((top->coordinate.x+pow(2,-(top->depth))*floor(i/4)),
      (top->coordinate.y
       +pow(2,-(top->depth))*floor(fmod(i,4)/2)),
      (top->coordinate.z+pow(2,-(top->depth))*fmod(i,2)));
    if(coor.x < 1.0  && coor.y < 1.0 && coor.z < 1.0
       && coor.x >= 0.0 && coor.y >= 0.0 && coor.z >= 0.0){
      af = fieldop(f, oct2axis(coor, oc));
      top_af=field_lookup(oc,
  oct2axis(scalar2vec(top->coordinate.x+
      pow(2,-(top->depth)),
      top->coordinate.y+
      pow(2,-(top->depth)),
      top->coordinate.z+
      pow(2,-(top->depth))),
   oc));
      if(fabs(top_af->e.x-af->e.x) > per ||
 fabs(top_af->e.y-af->e.y) > per ||
 fabs(top_af->e.z-af->e.z) > per ||
 fabs(top_af->b.x-af->b.x) > per ||
 fabs(top_af->b.y-af->b.y) > per ||
 fabs(top_af->b.z-af->b.z) > per){
if(top->subboxes[i] == NULL)
  top->subboxes[i] = make_subbox(top, oc, f, coor, bd);
refine2perc(top->subboxes[i], oc, f, percentage, bd);
      }
      else
for(j=0; j<8; j++){
  neigh = scalar2vec((coor.x+pow(2,-(top->depth))*
      pow(-1,floor(j/4))),
     (coor.y+pow(2,-(top->depth))*
      pow(-1,floor(fmod(j,4)/2))),
     (coor.z+pow(2,-(top->depth))*
      pow(-1,fmod(j,2))));
  if(neigh.x < 1.0 || neigh.y < 1.0 || neigh.z < 1.0){
    free(af);
    af = fieldop(f, oct2axis(neigh, oc));
    if(fabs(top_af->e.x-af->e.x) > per ||
       fabs(top_af->e.y-af->e.y) > per ||
       fabs(top_af->e.z-af->e.z) > per ||
       fabs(top_af->b.x-af->b.x) > per ||
       fabs(top_af->b.y-af->b.y) > per ||
       fabs(top_af->b.z-af->b.z) > per){
      if(top->subboxes[i] == NULL)
top->subboxes[i] = make_subbox(top, oc, f, coor, bd);
      refine2perc(top->subboxes[i], oc, f, percentage, bd);
    }
  }
}
      free(af);
      free(top_af);
    }
  }
```

```
}


/*
 * function:    refine2point
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc, Afield *f,
 *              Vector axis_coor, int depth, Bounds bd
 * description: refines at a position, given by axis_coor within
 *              the FED device, to level of subboxes, designated by
 *              depth.
 */
void refine2point(BoxPtr top, OctreePtr oc, Afield *f,
  Vector axis_coor, int depth, Bounds bd)
{
  int i;
  Vector coor, target = axis2oct(axis_coor, oc);

  refpoint(top, oc, f, target, depth, bd);

  /* refine over eight neighboring coordinates
     where the original is the center of the cube */

  for(i=0; i<8; i++){
    coor = scalar2vec((target.x+pow(2,-depth)*pow(-1,floor(i/4))),
      (target.y+pow(2,-depth)*pow(-1,floor(fmod(i,4)/2))),
      (target.z+pow(2,-depth)*pow(-1,fmod(i,2))));
    if(coor.x < 1.0 && coor.y < 1.0 && coor.z < 1.0
       && coor.x >= 0.0 && coor.y >= 0.0 && coor.z >= 0.0)
      refpoint(top, oc, f, coor, depth, bd);
  }
}

/*
 * function:    refpoint
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc, Afield *f,
 *              Vector oct_coor, int depth, Bounds bd
 * description: this is an auxiliary function which refines to
 *              a position in octree coordinates, or oct_coor
 */
void refpoint(BoxPtr top, OctreePtr oc, Afield *f,
      Vector oct_coor, int depth, Bounds bd)
{
  int i;
  Vector coor;

  if(depth < 1 || top->depth >= MAXDEPTH || top == NULL)
    return;

  for(i=0; i<8; i++){
    coor = scalar2vec((top->coordinate.x+pow(2,-(top->depth))*floor(i/4)),
      (top->coordinate.y+
       pow(2,-(top->depth))*floor(fmod(i,4)/2)),
```

```
      (top->coordinate.z+pow(2,-(top->depth))*fmod(i,2)));
    if(coor.x < 1.0  && coor.y < 1.0 && coor.z < 1.0
       && coor.x >= 0.0 && coor.y >= 0.0 && coor.z >= 0.0)
      if(top->subboxes[i] == NULL)
top->subboxes[i] = make_subbox(top, oc, f, coor, bd);
  }
  refpoint(top->subboxes[whichoct(top,oct_coor)],
   oc, f, oct_coor, depth-1, bd);
}


/*
 * function:    refine2pointperc
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc, Afield *f,
 *              Vector oct_coor, double percentage, Bounds bd
 * description: refines the down to a single position with the FED
 *              device volume, axis_coor, with a level of subboxes,
 *              whose neighboring fields vary below the specified
 *              percentage.
 */
void refine2pointperc(BoxPtr top, OctreePtr oc, Afield *f,
      Vector axis_coor, double percentage, Bounds bd)
{
  BoxPtr tmp;
  Afield *af, *top_af;
  double per = percentage/100.0;

  if(top->depth >= MAXDEPTH || top == NULL)
    return;

  tmp = top;
  af = fieldop(f, axis_coor);
  top_af=field_lookup(oc,
      oct2axis(scalar2vec(top->coordinate.x+
  pow(2,-(top->depth)),
  top->coordinate.y+
  pow(2,-(top->depth)),
  top->coordinate.z+
  pow(2,-(top->depth))),
       oc));
  while((fabs(top_af->e.x-af->e.x) > per ||
 fabs(top_af->e.y-af->e.y) > per ||
 fabs(top_af->e.z-af->e.z) > per ||
 fabs(top_af->b.x-af->b.x) > per ||
 fabs(top_af->b.y-af->b.y) > per ||
 fabs(top_af->b.z-af->b.z) > per) &&
tmp->depth < MAXDEPTH){
    refpoint(tmp, oc, f, axis_coor, 1, bd);
    tmp = tmp->subboxes[whichoct(tmp, axis2oct(axis_coor, oc))];
  }
  free(af);
  free(top_af);
  refine2point(top, oc, f, axis_coor, tmp->depth-top->depth, bd);
```

```
}

/*
 * function:    refinebds
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc, Afield *f,  int depth,
 *         double incr, Bounds bd
 * description: refines the boundary walls of the FED volume,
 *              This is a box-like structre.
 */
void refinebds(BoxPtr top, OctreePtr oc, Afield *af, int depth,
       double incr, Bounds bd)
{
  double i, j, k;
  double xincr, yincr, zincr;
  xincr = (bd.xmax-bd.xmin)/(incr-1);
  yincr = (bd.ymax-bd.ymin)/(incr-1);
  zincr = (bd.zmax-bd.zmin)/(incr-1);

  /*top and bottom boundary planes*/
  for(i=bd.xmin; i<=bd.xmax; i+=xincr)
    for(j=bd.ymin; j<=bd.ymax; j+=yincr){

      k = bd.zmin;
      refine2point(top, oc, af, scalar2vec(i,j,k), depth,  bd);

      k = bd.zmax;
      refine2point(top, oc, af, scalar2vec(i,j,k), depth,  bd);
    }
  /*front and back boundary planes*/

  for(j=bd.ymin; j<=bd.ymax; j+=yincr)
    for(k=bd.zmin; k<=bd.zmax; k+=zincr){
      i = bd.xmin;
      refine2point(top, oc, af, scalar2vec(i,j,k), depth,  bd);
      i = bd.xmax;
      refine2point(top, oc, af, scalar2vec(i,j,k), depth,  bd);
    }

  /*side boundary planes*/

  for(i=bd.xmin; i<=bd.xmax; i+=xincr)
    for(k=bd.zmin; k<=bd.zmax; k+=zincr){
      j = bd.xmin;
      refine2point(top, oc, af, scalar2vec(i,j,k), depth,  bd);
      j = bd.xmax;
      refine2point(top, oc, af, scalar2vec(i,j,k), depth,  bd);
    }

}


/*
 * function:    refinetopbds
```

```
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc, Afield *f,  int depth,
 *         Bounds bd
 * description: Uniformly refines the top boundary, or the screen,
 *              to level specified by depth.
 */
void refinetopbds(BoxPtr top, OctreePtr oc,
  Afield *af, int depth, Bounds bd)
{
  int i;
  Vector coor;

  if(depth < 1 || top->depth >= MAXDEPTH || top == NULL)
    return;

  for(i=1; i<8; i+=2){
    coor = scalar2vec((top->coordinate.x+pow(2,-(top->depth))*floor(i/4)),
      (top->coordinate.y+
       pow(2,-(top->depth))*floor(fmod(i,4)/2)),
      (top->coordinate.z+pow(2,-(top->depth))*fmod(i,2)));
    if(top->subboxes[i] == NULL){
      top->subboxes[i] = make_subbox(top, oc, af, coor, bd);
    }
    refinebds2(top->subboxes[i], oc, af, depth-1, bd);
  }
}


/*
 * function:    refinesemi
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc, Afield *f,  int depth,
 *         double no_incr, double R, double r_o, Bounds bd)
 * description: Refines a semi-circle above the $x$-$y$ plane.
 *              This semi-circle is centered about the origin.
 */
void refinesemi(BoxPtr top, OctreePtr oc, Afield *af, int depth,
double no_incr, double R, double r_o, Bounds bd)
{
  double i, j, k;
  double rho;
  double incr = (2*R)/(no_incr-1.0);

  for(i=-R; i<=R; i+=incr)
    for(j=-R; j<=R; j+=incr)
      for(k=0.0; k<=R; k+=incr){
rho = sqrt(pow(i,2)+pow(j,2)+pow(k,2));
if(rho <= R && rho >= r_o)
  refine2point(top, oc, af, scalar2vec(i,j,k), depth,  bd);
  }
}

/****************** print *******************/
```

```
/*
 * function:    print_box
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc
 * description: prints all the information in an octree,
 *              Scaled in Octree coordinates.
 */
void print_box(BoxPtr top, OctreePtr oc)
{
  int i;
  Afield *top_af;

  if(top == NULL)
    return;

  top_af=field_lookup(oc,
      oct2axis(scalar2vec(top->coordinate.x+
  pow(2,-(top->depth)),
  top->coordinate.y+
  pow(2,-(top->depth)),
  top->coordinate.z+
  pow(2,-(top->depth))),
       oc));
  printf("\nE: %f\t%f\t%f\nB: %f\t%f\t%f\n",
 top_af->e.x, top_af->e.y, top_af->e.z,
 top_af->b.x, top_af->b.y, top_af->b.z);
  printf("depth:%d\t term:%d\t%f\t%f\t%f\t%d\n",
 top->depth,  top->num_terminated, top->coordinate.x,
 top->coordinate.y, top->coordinate.z, top->isboundary);
  printf("num_of_electronsns %d\tenergy %f\n",
 top->dat->num, top->dat->energy);
  for(i=0; i<8 ; i++)
    print_box(top->subboxes[i], oc);
}


/*
 * function:    print_box_pos
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc
 * description: prints all position of Octree boxes, scaled in
 *              actual FED coordinates
 */
void print_box_pos(BoxPtr top, OctreePtr oc)
{
  int i;
  if(top == NULL)
    return;

  printf("%f\t%f\t%f\n", top->coordinate.x*oc->axis->scale->x
 + oc->axis->offset->x, top->coordinate.y*oc->axis->scale->y
 + oc->axis->offset->y, top->coordinate.z*oc->axis->scale->z
 + oc->axis->offset->z);
```

```
  for(i=0; i<8 ; i++)
    print_box_pos(top->subboxes[i], oc);
}


/*
 * function:    print_box_data
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc
 * description: prints all data of all Octree boxes, scaled in
 *              actual FED coordinates
 */
void print_box_data(BoxPtr top, OctreePtr oc)
{
  int i;
  if(top == NULL)
    return;

  printf("%e\t%e\t%e\t%d\t%f\n", top->coordinate.x*oc->axis->scale->x
 + oc->axis->offset->x, top->coordinate.y*oc->axis->scale->y
 + oc->axis->offset->y, top->coordinate.z*oc->axis->scale->z
 + oc->axis->offset->z, top->dat->num, top->dat->energy);
  for(i=0; i<8 ; i++)
    print_box_data(top->subboxes[i], oc);
}

/*
 * function:    print_top_data2file
 * returns:     void
 * arguments:   BoxPtr top, OctreePtr oc, Bounds bd, char filename[50]
 * description: prints only data at the top Octree boxes, which represent
 *              the screen, to a file specifed by filename.  In addition,
 *              this only prints data, if an electron has passed through
 *              the box.
 */
void print_top_data2file(BoxPtr top, OctreePtr oc,
 Bounds bd, char filename[50])
{
  int i;
  FILE *fptr;

  if(top == NULL)
    return;

  if(top->subboxes[1] == NULL || top->subboxes[3] == NULL
     || top->subboxes[5] == NULL || top->subboxes[7] == NULL)
    if(top->dat->num > 0){
      if((fptr = fopen(filename, "a")) == NULL ){
        printf("Error opening file %s. Exiting program\n", filename);
        exit(1);
      }
      fprintf(fptr, "%e\t%e\t%e\t%d\t%f\n",
      top->coordinate.x*oc->axis->scale->x
      + oc->axis->offset->x, top->coordinate.y*oc->axis->scale->y
```

```
      + oc->axis->offset->y, top->coordinate.z*oc->axis->scale->z
      + oc->axis->offset->z, top->dat->num, top->dat->energy);
      fclose(fptr);
    }
  for(i=1; i<8 ; i+=2)
    print_top_data2file(top->subboxes[i], oc, bd, filename);
}


/*
 * function:    print_coor
 * returns:     void
 * arguments:   Vector coor
 * description: prints a vector
 */
void print_coor(Vector coor)
{
  printf("\n%f %f %f\n", coor.x, coor.y, coor.z);
}

/****** functions for assigning analytic fields ******/
/*
 * function:    field_lookup
 * returns:     coor
 * arguments:   OctreePtr oc, Vector axis_coor)
 * description: given an axis_location, i.e. actual coordinate of
 *              FED environment, this lookups up field values from
 *              the corresponding Octree box and performs a 3D
 *              linear interpolation to find the field at the
 *              exact position of axis_coor.
 */
Afield *field_lookup(OctreePtr oc, Vector axis_coor)
{
  Afield *f;
  double x1, x2, x3;
  BoxPtr box = whereami(oc, axis_coor);
  Vector box_coor = oct2axis(box->coordinate, oc);

  if((f = malloc(sizeof(Afield))) == NULL){
    printf("Error using malloc in field_lookup.  Exiting program\n");
    exit(0);
  }
  x1 = fabs(axis_coor.x - box_coor.x)/(pow(2,-box->depth)*oc->axis->scale->x);
  x2 = fabs(axis_coor.y - box_coor.y)/(pow(2,-box->depth)*oc->axis->scale->y);
  x3 = fabs(axis_coor.z - box_coor.z)/(pow(2,-box->depth)*oc->axis->scale->z);

  f->e.x = (1-x1)*(1-x2)*(1-x3)*box->fd[0]->e.x
    + (1-x1)*(1-x2)*x3*box->fd[1]->e.x + (1-x1)*x2*(1-x3)*box->fd[2]->e.x
    + (1-x1)*x2*x3*box->fd[3]->e.x + x1*(1-x2)*(1-x3)*box->fd[4]->e.x
    + x1*(1-x2)*x3*box->fd[5]->e.x + x1*x2*(1-x3)*box->fd[6]->e.x
    + x1*x2*x3*box->fd[7]->e.x;

  f->e.y = (1-x1)*(1-x2)*(1-x3)*box->fd[0]->e.y
    + (1-x1)*(1-x2)*x3*box->fd[1]->e.y + (1-x1)*x2*(1-x3)*box->fd[2]->e.y
```

```
    + (1-x1)*x2*x3*box->fd[3]->e.y + x1*(1-x2)*(1-x3)*box->fd[4]->e.y
    + x1*(1-x2)*x3*box->fd[5]->e.y + x1*x2*(1-x3)*box->fd[6]->e.y
    + x1*x2*x3*box->fd[7]->e.y;

  f->e.z = (1-x1)*(1-x2)*(1-x3)*box->fd[0]->e.z
    + (1-x1)*(1-x2)*x3*box->fd[1]->e.z + (1-x1)*x2*(1-x3)*box->fd[2]->e.z
    + (1-x1)*x2*x3*box->fd[3]->e.z + x1*(1-x2)*(1-x3)*box->fd[4]->e.z
    + x1*(1-x2)*x3*box->fd[5]->e.z + x1*x2*(1-x3)*box->fd[6]->e.z
    + x1*x2*x3*box->fd[7]->e.z;

  f->b.x = (1-x1)*(1-x2)*(1-x3)*box->fd[0]->b.x
    + (1-x1)*(1-x2)*x3*box->fd[1]->b.x + (1-x1)*x2*(1-x3)*box->fd[2]->b.x
    + (1-x1)*x2*x3*box->fd[3]->b.x + x1*(1-x2)*(1-x3)*box->fd[4]->b.x
    + x1*(1-x2)*x3*box->fd[5]->b.x + x1*x2*(1-x3)*box->fd[6]->b.x
    + x1*x2*x3*box->fd[7]->b.x;

  f->b.y = (1-x1)*(1-x2)*(1-x3)*box->fd[0]->b.y
    + (1-x1)*(1-x2)*x3*box->fd[1]->b.y + (1-x1)*x2*(1-x3)*box->fd[2]->b.y
    + (1-x1)*x2*x3*box->fd[3]->b.y + x1*(1-x2)*(1-x3)*box->fd[4]->b.y
    + x1*(1-x2)*x3*box->fd[5]->b.y + x1*x2*(1-x3)*box->fd[6]->b.y
    + x1*x2*x3*box->fd[7]->b.y;

  f->b.z = (1-x1)*(1-x2)*(1-x3)*box->fd[0]->b.z
    + (1-x1)*(1-x2)*x3*box->fd[1]->b.z + (1-x1)*x2*(1-x3)*box->fd[2]->b.z
    + (1-x1)*x2*x3*box->fd[3]->b.z + x1*(1-x2)*(1-x3)*box->fd[4]->b.z
    + x1*(1-x2)*x3*box->fd[5]->b.z + x1*x2*(1-x3)*box->fd[6]->b.z
    + x1*x2*x3*box->fd[7]->b.z;

  return f;
}

/*
 * function:    field_op
 * returns:     Afield *
 * arguments:   Afield *f, Vector axis_coor
 * description: This function is the field lookup interface between
 *              the Trajectory calculation program and the Octree
 *              data structure.  In this case, the fields are
 *              set up for a cone-shaped emitter, r_o = 0.05e-6,
 *              R =0.5e-6, and d = 1000e-6
 */
Afield *fieldop(Afield *f, Vector axis_coor)
{
/*  return linear_fieldop(f, point);*/
  double r_o = 0.05e-6,  R =0.5e-6, V_a = V_A, V_g = V_G, d = 1000e-6;
  return cone_fieldop(f, axis_coor,  r_o, R, V_a, V_g, d);
}


/*
 * function:    field_copy
 * returns:     Afield *
 * arguments:   Afield *f
 * description: Returns a point to the copy of the af
```

```
 */
Afield *field_copy(Afield *af)
{
  Afield *f;
  /* dynamic memory allocation */
  if((f = malloc(sizeof(Afield))) == NULL){
    printf("Error using malloc in field_copy.  Exiting program\n");
    exit(0);
  }

  f->e.x = af->e.x;
  f->e.y = af->e.y;
  f->e.z = af->e.z;
  f->b.x = af->b.x;
  f->b.y = af->b.y;
  f->b.z = af->b.z;
  return f;
}

/*
 * function:     fieldop_oct
 * returns:      Afield*
 * arguments:    Fields f, Vector point
 * description:  Looks up octree field at vector point....
 */
Afield *fieldop_oct(OctreePtr oc, Vector point)
{
  return field_lookup(oc, point);
}

/*************** analytical field operations ****************/
/*
 * function:     constant_fieldop
 * returns:      Afield *
 * arguments:    AFields af, Vector point
 * description:  Returns a constant analytical field at vector point....
 */
Afield *constant_fieldop(Afield *af, Vector point)
{
  return af;
}

/*
 * function:     linear_fieldop
 * returns:      Afield *
 * arguments:    AFields af, Vector point
 * description:  Returns a linear analytical electrical field
 *               in the x direction.
 */
Afield *linear_fieldop(Afield *f, Vector point)
{
  Afield *af;
  af = field_copy(f);
  af->e.x = linear_field(point.x, 1, f->e.x);
```

```
  return af;
}


/*
 * function:     quadratic_fieldop
 * returns:      Afield *
 * arguments:    AFields af, Vector point
 * description:  Returns a quadratic analytical electrical field
 *               in the x direction.
 */
Afield *quadratic_fieldop(Afield *f, Vector point)
{
  Afield *af;
  af = field_copy(f);
  af->e.x = quadratic_field(point.x, f->e.x);
  return af;
}


/*
 * function:     cone_fieldop
 * returns:      Afield *
 * arguments:    AFields af, Vector point, double r_o, double R,
 *               double V_a, double V_g, double d
 * description:  Returns a analytical electrical field for the
 *               region around a cone shaped emitter.
 */
Afield *cone_fieldop(Afield *f, Vector point,
     double r_o, double R,
     double V_a, double V_g,
     double d)
{
  Afield *af;
  double rho, theta, phi;
  double E;
  af = field_copy(f);

  rho = sqrt(pow(point.x,2)+pow(point.y,2)+pow(point.z,2));
  if(rho > R){
    af->e.x = 0;
    af->e.y = 0;
    af->e.z = (V_a - V_g)/d;
    af->b.x = 0;
    af->b.y = 0;
    af->b.z = 0;
    return af;
  }
  else if (rho <= r_o){
    af->e.x = 0;
    af->e.y = 0;
    af->e.z = 0;
    af->b.x = 0;
    af->b.y = 0;
```

```
    af->b.z = 0;
    return af;
  }
  else{
    theta = acos(point.z/rho);
    if(point.x == 0.0 && point.y == 0.0)
phi = PI/2.0;
    else if(point.y > 0.0)
phi = acos(point.x/sqrt(pow(point.x,2)+pow(point.y,2)));
    else
phi = -acos(point.x/sqrt(pow(point.x,2)+pow(point.y,2)));
#ifdef DEBUG2
    printf("z %e rho %e x %e r %e theta %f phi %f\n", point.z,
   rho, point.x, sqrt(pow(point.x,2)+pow(point.y,2)), theta, phi);
#endif
    E = V_g/(pow(rho,2)*(1/r_o - 1/R));
    af->e.x = E*sin(theta)*cos(phi);
    af->e.y = E*sin(theta)*sin(phi);
    af->e.z = E*cos(theta);

#ifdef DEBUG2
    printf("pos %e %e %e\n", point.x, point.y, point.z);
    printf("rho %f theta %f phi %f af %e %e %e\n",
   rho*180/PI, theta*180/PI, phi*180/PI, af->e.x, af->e.y,af->e.z);
#endif
    af->b.x = 0;
    af->b.y = 0;
    af->b.z = 0;
    return af;
  }
}


/*
 * function:     linear_field
 * returns:      double
 * arguments:    double x, double slope, double b
 * description:  Auxiliary function for calculating linear
 *               electric fields.  This just returns a
 *               linear equation
 */
double linear_field(double x, double slope, double b)
{
  return slope*x+b;
}

/*
 * function:     quadratic_field
 * returns:      double
 * arguments:    double x, double a
 * description:  Auxiliary function for calculating quadratic
 *               electric fields.  This just returns a
 *               quadratic equation
 */
```

```
double quadratic_field(double x, double a)
{
  return x*x+a;
}

/********** writing data into octree functions **********/

/*
 * function:     write_data
 * returns:      void
 * arguments:    OctreePtr oc, Vector axis_coor, Vector vel, Initiale c
 * description:  Writes data from the Trajectory structure, into
 *               the Octree, after the trajectory has been calculated
 */
void write_data(OctreePtr oc, Vector axis_coor, Vector vel, Initiale c)
{
  BoxPtr b;

  b = search(oc->top, axis2oct(axis_coor, oc));
  if(b){
    b->dat->energy += vec2energy(vel, c);
    (b->dat->num)++;
  }
}


/*
 * function:     different_box
 * returns:      boolean
 * arguments:    OctreePtr oc, Vector p1, Vector p2
 * description:  Compares to see if positions, p1 and p2,
 *               are represented by the same Octree box
 */
int different_box(OctreePtr oc, Vector p1, Vector p2)
{
  if(search(oc->top, axis2oct(p1, oc)) == search(oc->top, axis2oct(p2, oc)))
    return FALSE;
  return TRUE;
}

/********** Miscellaneous **********/


/*
 * function:     vec2scalar
 * returns:      double
 * arguments:    Vector vel
 * description:  Gives the scalar of a vector
 */
double vec2scalar(Vector vel)
{
  return sqrt(pow(vel.x,2)+pow(vel.y,2)+pow(vel.z,2));
}
```

```
/*
 * function:     scalar2vec
 * returns:      Vector
 * arguments:    double x, double y, double z
 * description:  Returns the vector of the three coordinate points
 */
Vector scalar2vec(double x, double y, double z)
{
  Vector vec;
  vec.x = x;
  vec.y = y;
  vec.z = z;
  return vec;
}


/*
 * function:     oct2axis
 * returns:      Vector
 * arguments:    Vector coor, OctreePtr oc
 * description:  Converts the octree coor into the actual FED coor
 */
Vector oct2axis(Vector coor, OctreePtr oc)
{
  Vector a;
  a.x = (coor.x * oc->axis->scale->x) + oc->axis->offset->x;
  a.y = (coor.y * oc->axis->scale->y) + oc->axis->offset->y;
  a.z = (coor.z * oc->axis->scale->z) + oc->axis->offset->z;
  return a;
}

/*
 * function:     axis2oct
 * returns:      Vector
 * arguments:    Vector axis_coor, OctreePtr oc
 * description:  Converts axis_coor a coordinate in the Octree
 *               frame of reference.
 */
Vector axis2oct(Vector axis_coor, OctreePtr oc)
{
  Vector c;
  c.x = (axis_coor.x - oc->axis->offset->x)/oc->axis->scale->x;
  c.y = (axis_coor.y - oc->axis->offset->y)/oc->axis->scale->y;
  c.z = (axis_coor.z - oc->axis->offset->z)/oc->axis->scale->z;
  return c;
}

/*
 * function:     vec2energy
 * returns:      double scalar energy value
 * arguments:    Vector velocity
 * description:  given velocity determines energy
 */
```

```
double vec2energy(Vector vel, Initiale c)
{
  return 0.5*c.m*pow(vec2scalar(vel),2.0)/1.60e-19;
}

double energy2vel(double energy, Initiale c)
{
  return sqrt(2.0*energy/c.m*1.60e-19);
}
```

Appendix E

initial.m

```
/*
 *  name of this file:   pixel/src/traj1/initial.m
 */
#include "trajectory.h"

/******* initialization functions **********/

/*
 * function:     init_particle
 * returns:      void
 * arguments:    Initiale *ch,  initial charge characteristics
 * description:  This function the asks user for initial
 *               charge characteristics.
 */
void init_particle(Initiale *ch)
{
  printf("Enter the charge and mass of particle:  ");
  scanf("%lf %lf", &ch->q, &ch->m);
  printf("\nInput initial velocities V.x, V.y, V.z:");
  scanf("%lf %lf %lf", &ch->vel.x, &ch->vel.y, &ch->vel.z);
  printf("\nInput initial position P.x, P.y, P.z:");
  scanf("%lf %lf %lf", &ch->pos.x, &ch->pos.y, &ch->pos.z);
}

/*
 * function:     init_bounds
 * returns:      void
 * arguments:    Bounds *b,        boundary values
 * description:  This function initializes boundary values.
 *               The boundary restricted to be in the shape of a cube.
 *               The user is asked to input the upper-right and the
 *               lower left-corner of this cube.
 */
void init_bounds(Bounds *b)
```

```
{
  printf("\nInput upper-right-corner of boundary (x,y,z): ");
  scanf("%lf %lf %lf", &b->xmax, &b->ymax, &b->zmax);
  printf("\nInput lower-left-corner of boundary (x,y,z): ");
  scanf("%lf %lf %lf", &b->xmin, &b->ymin, &b->zmin);

}

/*
 * function:     init_fields
 * returns:      void
 * arguments:    Fields *f,    electric and magnetic field values
 * description:  This function initializes electric and magnetic fields
 *               to constant values.
 */
void init_fields(Fields *f)
{
  printf("\nInput E_x, E_y, E_z, B_x, B_y, B_z:\n");
  scanf("%lf %lf %lf %lf %lf %lf",
&f->af.e.x,&f->af.e.y,&f->af.e.z,
&f->af.b.x,&f->af.b.y,&f->af.b.z);
}
```

Appendix F

sample main.c

```
/*
 *  name of this file:  pixel/src/traj1/main.c
 */

#include "trajectory.m"
#include <time.h>

void main()
{

  TrajectoryPtr t;/* trajectory pointer */
  Initiale ch;    /* initial electron information */
  Bounds bd;      /* boundary information */
  Fields f;       /* fields structure */
  Afield af;      /* analytical field structure */
  double maxerror;/* maximum allowable error between two trajectories */
  double energy;  /* magnitude of inital trajectory energy */
  int num_traj;   /* the desired number of trajectory calls */
  char s[50];    /* filename, cannot be more than 49 characters long */
  int gauss;   /* boolean denotes whether to perform gaussian dist */
  double variance = 0.2; /* variance to gaussian distribution */
  time_t before, after;

  /* scan in values for:
   * V_a = anode voltage (in volts)
   * V_g = gate voltage (in volts)
   * energy = initial electron energy (eV)
   * num_traj = number of desired trajectores
   * s = string, or filename to write data into
   * gaussian = boolean (True denotes use gaussian, false uniform)
   */
  scanf("%lf %lf %lf %d %s %d", &V_A, &V_G, &energy, &num_traj, s, &gauss);

  /* the init_all fuction will prompt the user for various parameters
```

```
   * such as the electron information, the boundaries of the environment
   * initial fields, maximum allowable error between similar trajectory
   * paths, a refinement levels within the Octree are also prompted as
   * a user input.
   */
  init_all(&t, &ch, &bd, &f, &af, &maxerror);


  /* this will set the initial guess multiply factor in the begining
   *  of the program, useful for customization */
  scanf("%lf", &GUESS);


  /* this fuction writes into the octree and calculates a uniform
   * distribution of initial velocities normal to the surface of the
   * emitter, with the characteristics of previously discussed variables,
   * within the range of 0.0 < theta < 2.0
   */
  time(&before);
  func(ch, f, bd, maxerror, energy, num_traj, 2.0, gauss, variance);
  time(&after);
  /* prints the total time for running simulations */
  printf("\nTime to run program %f secs.\n", difftime(after, before));

  print_top_data2file(oct->top, oct, bd, s);
  destroy_octree(oct);
}
```