

2012

## Visualization And Collision Detection Of Direct Metal Deposition

Sufyan S.D. Fehe

*North Carolina Agricultural and Technical State University*

Follow this and additional works at: <https://digital.library.ncat.edu/theses>

---

### Recommended Citation

Fehe, Sufyan S.D., "Visualization And Collision Detection Of Direct Metal Deposition" (2012). *Theses*. 83.  
<https://digital.library.ncat.edu/theses/83>

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at Aggie Digital Collections and Scholarship. It has been accepted for inclusion in Theses by an authorized administrator of Aggie Digital Collections and Scholarship. For more information, please contact [iyanna@ncat.edu](mailto:iyanna@ncat.edu).

# VISUALIZATION AND COLLISION DETECTION OF DIRECT METAL DEPOSITION

by

Sufyan S.D. Fehe

A thesis submitted to the graduate faculty  
in partial fulfillment of the requirements for the degree of  
MASTER OF SCIENCE

Department: Mechanical Engineering  
Major: Mechanical Engineering  
Major Professor: Dr. Shih-Liang Wang

North Carolina A&T State University  
Greensboro, North Carolina  
2012

School of Graduate Studies  
North Carolina Agricultural and Technical State University

This is to certify that the Master's Thesis of

Sufyan S.D. Fehe

has met the dissertation requirements of  
North Carolina Agricultural and Technical State University

Greensboro, North Carolina  
2012

Approved by:

---

Dr. Shih-Liang Wang  
Major Professor

---

Dr. Samuel Owusu-Ofori  
Committee Member

---

Dr. Sun Yi  
Committee Member

---

Dr. Samuel Owusu-Ofori  
Department Chairperson

---

Dr. Sanjiv Sarin  
Associate Vice Chancellor for Research  
and Dean of Graduate Studies

## **DEDICATION**

I dedicate this Master's Thesis to my parents; my late father Salifu Moro and my mother Binta Daud for their support and comfort through my entire education. I dedicate this thesis to God, for the protection and favor he bestowed on me from birth to this time.

## **BIOGRAPHICAL SKETCH**

Sufyan S.D. Fehe was born June 22, 1976 in Accra, Ghana. He received the Bachelor of Science in Agricultural Engineering from Kwame Nkrumah University of Science and Technology in 2001. He is currently a candidate for the Master of Science degree in Mechanical Engineering.

## **ACKNOWLEDGMENT**

I am very grateful to my advisor, Dr. Shih-Liang Wang for the guidance and support during challenging times in the course of the research. I wish to express my appreciation to my thesis committee members, Dr. Samuel Owusu-Ofori and Dr. Sun Yi for their time and review of my thesis. I thank my colleagues for all discussions, it helped me a lot.

From Missouri S&T University I wish to express my appreciation to Dr. Frank Liou (Professor in Mechanical Engineering & Director of the Interdisciplinary Manufacturing Engineering Education Program), Dr. Jianzhong Ruan (Post Doctoral Fellow in Mechanical Engineering) and Todd Sparks (Ph.D. Student in Mechanical Engineering) for their support, discussions and answers to questions I ask them in a prompt manner. I thank them for giving me materials to use as a head start in the research. I also appreciate the reception and courtesy they gave me when I visited Missouri for more enlightenment on this research.

I also want to acknowledge Product Innovation and Engineering (PINE), LLC for the financial support in the course of this work. PINE's funding is through a NSF Small Business Technology Transfer Project SBIR/STTR Phase IIA supplement research project entitled "Process Visualization for Hybrid Manufacturing Systems." I also thank Amit Raj Thatipalli former Master Engineering student at Missouri S&T University for sharing his ideas with me pertaining to his research. Lastly, I wish to thank my family for all the prayers and support.

## TABLE OF CONTENTS

|   |     |
|---|-----|
| LIST OF FIGURES .....   | vii |
| ABBREVIATIONS .....   | ix  |
| ABSTRACT .....  | x   |
| CHAPTER 1. INTRODUCTION .....   | 1   |
| 1.1 Direct Metal Deposition .....   | 1   |
| 1.2 Problem Statement.....  | 2   |
| 1.3 Research Objectives .....   | 3   |
| 1.4 Thesis Layout .....   | 3   |
| CHAPTER 2. VISUALIZATION OF THE DEPOSITION PROCESS USING SWEEP<br>VOLUME .....                          | 6   |
| 2.1 Literature Review .....   | 6   |
| 2.2 Selection of the Computer Programming Language .....  | 7   |
| 2.3 Importance of G-codes in the Visualization Process.....   | 8   |
| 2.4 Extraction of the Coordinate Values from the G-codes .....  | 8   |
| 2.5 Swept Volume Procedure .....  | 10  |
| 2.6 Cross Section Overlap .....   | 12  |
| 2.7 Direct Metal deposition Visualization Results .....   | 14  |
| 2.8 Comparison with Previous Work .....   | 17  |
| 2.9 Observations and Conclusions.....   | 18  |
| CHAPTER 3. COLLISION DETECTION OF THE DEPOSITION PROCESS USING<br>OCTREE AND ORIENTED BOUNDING BOX..... | 19  |
| 3.1 Literature Review .....   | 19  |

|  |    |
|--|----|
| 3.2 Selection of the Computer Programming Language .....                                   | 22 |
| 3.3 Octree Data Structure .....  | 23 |
| 3.4 Creation and Split of the Octree Boxes .....   | 24 |
| 3.5 Oriented Bounding Box Collision Detection Test and Results .....                       | 28 |
| 3.6 Comparison with Previous Work .....  | 35 |
| 3.7 Observations and Conclusions.....  | 36 |
| CHAPTER 4. DISCUSSION AND RECOMMENDATION.....  | 38 |
| 4.1 Summary and Discussion .....   | 38 |
| 4.2. Recommendation and Future Work.....   | 39 |
| REFERENCES .....   | 40 |
| APPENDIX A. VYPYTHON, NUMPY AND PYOGENGL OVERVIEW .....                                    | 43 |
| APPENDIX B. INSTRUCTIONS TO INSTALL AND RUN VISUALIZATION<br>PROGRAM ON WINDOWS.....       | 44 |
| APPENDIX C. INSTRUCTIONS TO INSTALL AND RUN COLLISION DETECTION<br>PROGRAM ON WINDOWS..... | 49 |
| APPENDIX D. HOW TO INPUT OR CHANGE THE CAD MODLES AND THEIR<br>RELATIVE POSITIONS.....     | 53 |
| APPENDIX E. G-CODE TEXT FILE.....  | 56 |
| APPENDIX F. VISUALIZATION PROGRAM MODULES.....   | 57 |
| APPENDIX G. COLLISION DETECTION PROGRAM MODULES .....                                      | 65 |



## LIST OF FIGURES

| FIGURE  | PAGE |
|---|------|
| 1.1. Laser Metal Deposition Process .....   | 3    |
| 1.2. Hybrid manufacturing system in Missouri S&T LAMP Laboratory .....  | 4    |
| 1.3. Direct metal deposition process with part rotated through orientations (a), (b), (c) and (d) to complete the process ..... | 5    |
| 2.1. Algorithm for parsing G-codes.....   | 9    |
| 2.2. A cross-sectional shape swept along a straight and a curve path. ....  | 10   |
| 2.3. Angle measured between two line segments.....  | 11   |
| 2.4. Sweep algorithm and extrusion with and without spikes .....  | 12   |
| 2.5. (a) is 3D view of a cross section swept with 50% width overlap, (b) 2D top view showing the overlaps.....                  | 13   |
| 2.6. Flow chart for visualization of the deposition process.....  | 14   |
| 2.7. (a) 3D CAD model, (b) Actual deposited part from MST .....   | 15   |
| 2.8. (a) Inccurate toolpath, (b) Desired toolpath .....   | 16   |
| 2.9. (a) a 2.0 mm cross-sectional width, (b) a 0.8 mm cross-sectional width visualization results.....                          | 17   |
| 2.10. (a) Voxel placement result, (b) Swept volume result. ....   | 17   |
| 3.1. (a) 3D Structure of an octree and index codes of octants, (b) The tree structure of an octree .....                        | 23   |
| 3.2. A simplified 2D axis aligned bounding box.....   | 24   |
| 3.3. Simplified 2D axis aligned bounding box split into two axis aligned bounding boxes .....                                   | 25   |
| 3.4. A partitioned box showing it's occupancy values.....   | 26   |

|   |    |
|---|----|
| 3.5. Octree representation of a 3D CAD model showing (a) 9 depth partitioning, (b) 15 depth partitioning .....            | 27 |
| 3.6. Algorithm for creating Octree-OBB of the CAD models .....  | 28 |
| 3.7. Oriented bounding box (OBB) with local axis.....   | 29 |
| 3.8. The vector L forms a separating axis .....   | 29 |
| 3.9. Algorithm for separating axis collision detection .....  | 30 |
| 3.10. Two sets of OBBs in collision.....  | 30 |
| 3.11. Flow chart for OBB collision detection .....  | 33 |
| 3.12. Flow chart for Octree creation.....   | 34 |
| 3.13. (a) No collision between component I and nozzle, (b) Octree representation of the model.....                        | 35 |
| 3.14. (a) Nozzle collides with component I, (b) Octree representation of the model ..                                     | 35 |
| 3.15. (a) single axis aligned bounding box collision test, (b) Octree oriented bounding box collision detection test..... | 36 |

## **ABBREVIATIONS**

|        |   |
|--------|---|
| CAD    | Computer Aided Design                                     |
| AM     | Additive Manufacturing                                    |
| DMD    | Direct Metal Deposition                                   |
| STL    | Standard Tessellation Language                            |
| CNC    | Computer Numerical Control                                |
| AABB   | Axis Aligned Bounding Box                                 |
| CAM    | Computer Aided manufacturing                              |
| 3D     | Three Dimensional   |
| BSP    | Binary Spatial Partitioning                               |
| HBV    | Hierarchical Bounding Volume                              |
| CGAL   | Computational Geometry Algorithms Libraries               |
| OBB    | Oriented Bounding Box                                     |
| LAMP   | Laser Additive Manufacturing Process                      |
| GAMMA  | Geometric Algorithms for Modeling Motion and<br>Animation |
| K-DOPs | K planes Discrete Oriented Polytope                       |

## ABSTRACT

**Sufyan S.D. Fehe.** VISUALIZATION AND COLLISION DETECTION OF DIRECT METAL DEPOSITION. (Major Professor: **Dr. Shih-Liang Wang**), North Carolina Agricultural and Technical State University.

Direct metal deposition (DMD) is a manufacturing technique that manufactures solid metal parts from bottom to top using powdered metal and a focused laser. In this research, the swept volume technique was used as framework to develop a computer program to perform volumetric visualization of the deposition process as a pre-processor, before the actual metal deposition commences. The program extracts coordinate values from a G-code; these extracted values constitute a point. These points are then defined as a swept path using VPython extrusion object library. A cross section is then swept through these points to perform the volumetric visualization of the deposition process.

In a DMD system computer numerical control (CNC) machine components can collide during deposition, a computer program can be used to facilitate collision detection if components within the build perimeter collide. In this research, open source oriented bounding boxes (OBB) intersection and open source octree implementation were used to develop a computer program, to detect the collision between CAD models of two components within a graphic scene. The collision detection test is performed by holding one CAD model fixed while the other model is set into translation. The CAD models will collide, if the line distance from the center between their OBBs is equal to the sum of their projected radii onto the reference axis of approach.

# **CHAPTER 1**

## **INTRODUCTION**

### **1.1 Direct Metal Deposition**

Direct metal deposition (DMD) is a technique used to manufacture or build solid metal parts from bottom to top using powdered metal and a focused laser. There are many different DMD systems which are commercially available or are currently being developed. Each system may use different materials, different G-codes and different techniques for the build process. Below are the pre-processing stages for DMD.

- I. Create a 3D computer aided design (CAD) model
- II. Convert CAD model into Standard Tessellation language (STL) model
- III. Slice STL model into layers
- IV. Generate the tool path for each sliced layer

Volumetric visualization of the deposition process starts at stage IV before actual deposition commences. A lot of research has been done in this field and many techniques and algorithms have been proposed for DMD visualization. Visualization processes before actual execution of work finds its application in medical fields, oil and gas fields, NASA space programs and many more diverse engineering applications. POM Group is a company in Michigan that has done a lot of work in this field and has manufactured DMD machines for sale with visualization and collision detection capabilities. Missouri S & T University has done extensive work in this area too. Figure 1.1(Ch. Sweta Dhaveji, 2011) below is a DMD process at Missouri S&T University laboratory. Some of the

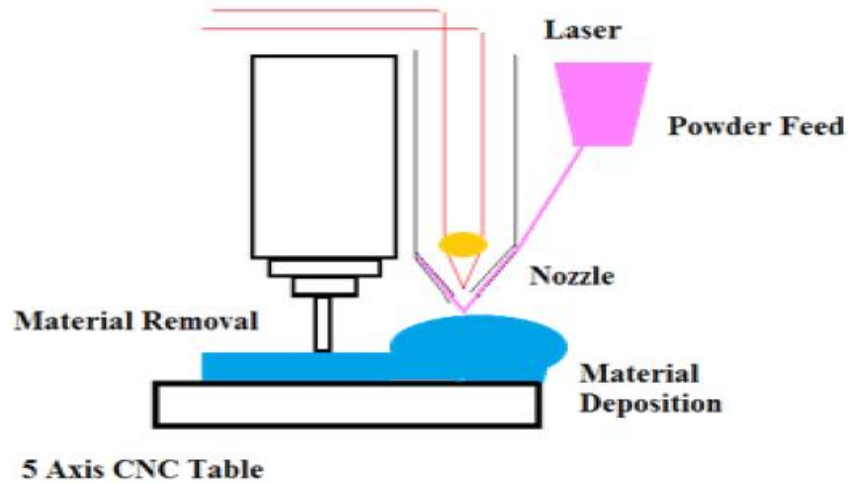
techniques used for volumetric visualization processes are voxel-based, swept volumes, octrees etc. The choice of technique to use for the visualization process is determined by the nature of the problem and the desired output.

Collision detection starts after stage IV when deposition has to commence. The deposition or build process poses possible collision between stationary or dynamic computer numerical control (CNC) machine components. A lot of research work has been done in this area and new techniques have been developed to solve and improve current algorithms. Collision detection is widely used and applied in the game industry, robotics and in CNC machining. There are many different techniques used in collision detection. Examples of bounding volumes techniques are; hierarchical bounding volumes (HBV), axis aligned bounding box (AABB), oriented bounding box (OBB), k-planes discrete oriented polytopes (K-DOPs), convex hulls, ray-triangle etc.

## **1.2 Problem Statement**

Below are problems during a deposition process that necessitated the research:

- I. Parts are either completely built, or sample deposition is performed for user to determine the accurateness of the part built from the generated G-code. This may lead to material wastage if it is realized the deposition will not build the part to its required specifications.
- II. In a DMD system, stationary or moving CNC machine components within the build perimeter can collide into each other during the deposition process. This can cause damage to the nozzle assembly, work piece etc.



**Figure 1.1. Laser Metal Deposition Process**

### **1.3 Research Objectives**

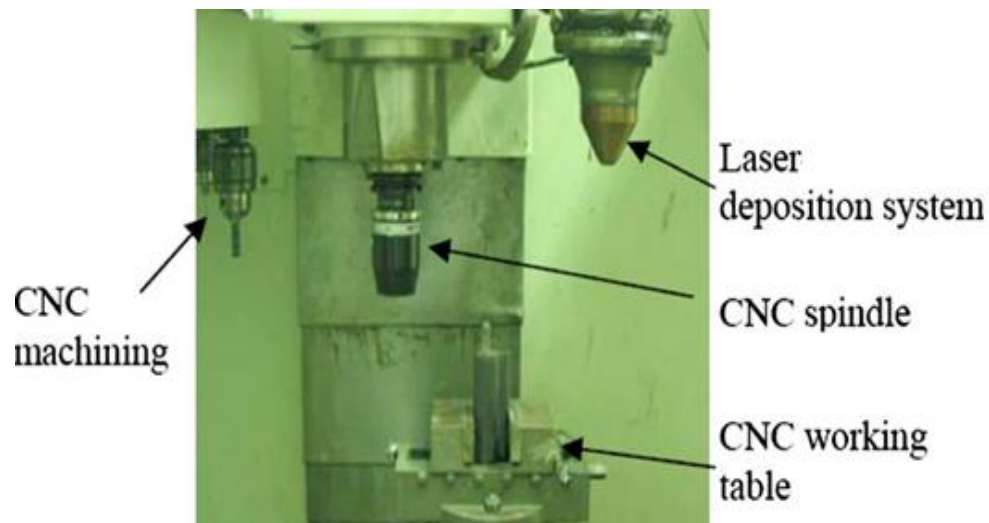
The main objectives of this research are outlined below:

- I. Use the swept volume technique to develop a computer program to be used as a pre-processor, for volumetric visualization of the deposition process before actual deposition commences.
- II. Use open source oriented bounding box (OBB) intersection technique (Gomez, 1999) and open source octree implementation (Kenwright, 2002) as framework to develop a computer program, to detect the collision between CAD models of two colliding components in a graphic scene.

### **1.4 Thesis Layout**

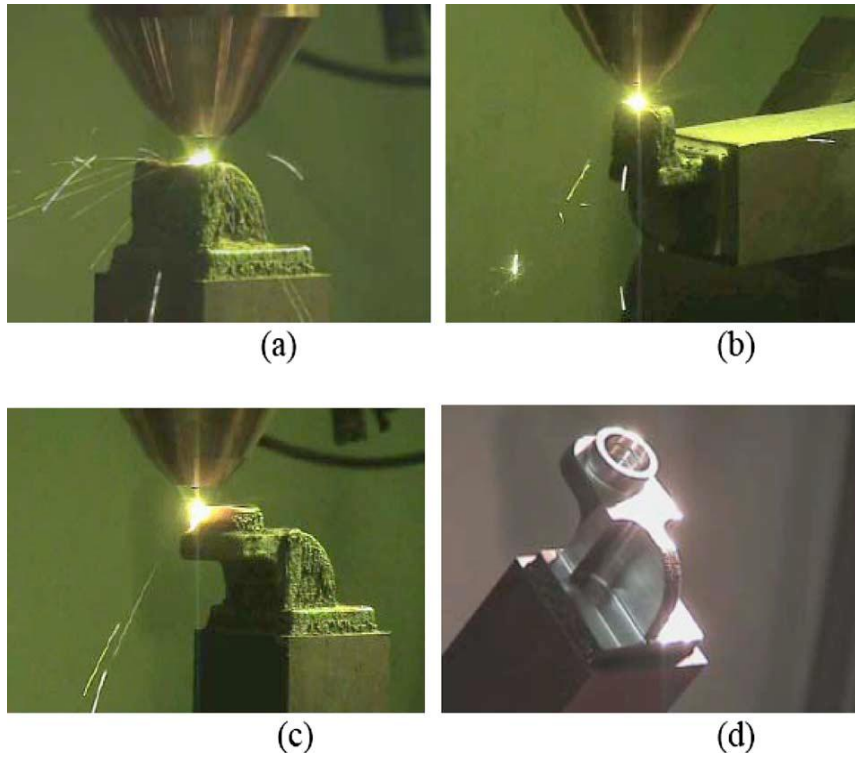
This thesis has been organized into four chapters. Chapter 1 is dedicated to introduction of the subject matter and the specific objective of the research work. Chapter

2 is solely dedicated to DMD visualization, literature review, methodology, algorithms, mathematical concepts, and the computer programming language used to develop the program. Chapter 3 focuses on collision detection during deposition, literature review, methodology, algorithms, mathematical concepts, and the computer programming language used to develop the program. Chapter 4 discusses the methods and algorithms used. Other ensuing difficulties encountered are discussed and recommendations are made for future work. Figure 1.2 modified (Ren *et al.*, 2010) below is a hybrid direct metal deposition (DMD) manufacturing system. This system builds solid parts from bottom to top and performs the surface finishing on the same system. Figure 1.3 (Ren, *et al.*, 2010) below is a DMD process.



**Figure 1.2. Hybrid manufacturing system in Missouri S&T LAMP Laboratory**





**Figure 1.3. Direct metal deposition process with part rotated through orientations (a), (b), (c) and (d) to complete the process**

## **CHAPTER 2**

### **VISUALIZATION OF THE DEPOSITION PROCESS USING SWEEPED VOLUME**

#### **2.1 Literature Review**

Thatipalli in his master's project work used the voxel-based technique to perform visualization of 5-axis direct metal deposition (DMD) process (Thatipalli, 2011). His visualization program sequentially places voxel size of 0.1 in x 0.1 in at the center of the 3D trajectory (X, Y, and Z) of the G-code to visualize the deposition process. Because voxels are sequentially placed by insertion, voxels placed on curved path lines will have to be rotated again to give a smooth rendered surface. This technique works well if the G-codes are discretely generated to fill every space. The technique is inefficient if the G-codes are made of line segments.

An octree approach that is used for simulation and visualization of multi-axis additive manufacturing (AM) system was presented (Dhoveji et al., 2011; Don, 1991). The algorithm uses octree technique to simulate and visualize the deposition of the part geometry and its progressive changes. The concept of this algorithm is in three folds; i.e. a 3D object is generated, an octree document is created and the AM process is then implemented. A general overview on voxel-based visualization techniques used in AM processes was catalogued (Chandru et al., 1995). In their paper, they presented various technical issues that borders on memory requirements and rendering complexities when using voxel algorithm for visualization of AM process. They also indicated in their paper

the development of software called “G-WoRP” that will in future solve many of the voxel-based techniques problems in AM processes.

3D printing is one out of the many techniques used in AM. The process planning is similar to that of DMD. The difference between the two manufacturing processes is DMD uses either an electron beam or laser beam as the source of energy to melt metal powder as deposits to build the part, while 3D printing traditionally uses an ultra violet (UV) light to cure extruded resin.

Direct placement primitive technique (DPP) (Jee *et al.*, 2000; Sachs *et al.*, 1990) was also researched. The algorithm presented in their paper is only restricted to the part geometry of the object being deposited using DPP. The size of a resulting DPP corresponds to the powder-binder agglomerate formed by a single droplet. The DPPs are successively deposited to form the virtual shape of the actual object. The resolution of the virtual object depends on the size of the DPP.

## **2.2 Selection of the Computer Programming Language**

Python programming language, its visual module Vpython and the extrusion object library was chosen as the platform to develop the computer program. Python was chosen over other programming languages like C++ and Computational Geometry Algorithms Libraries (CGAL) because, Python is a high level programming language that is far easy to learn and implement with very fast OpenGL rendering graphics in a short time. Python is also cross-platform for Windows, Mac and Linux and can be converted to C++ as well. Overview of VPython is provided in Appendix A.

### **2.3 Importance of G-codes in the Visualization Process**

The single most important data required for visualization of the deposition process is the G-code text file. Most G-codes are generated from computer aided manufacturing (CAM) software or other special stand-alone software. Geometric and volumetric information for the actual deposition process is extracted from the G-code. Similarly extracted data from G-codes are used for the volumetric visualization of DMD.

G-code instructs a DMD machine nozzle/tool what type of action to take at a given point in time. In DMD, G01 code means linear interpolation, it is the common workhorse for material deposition. M-codes are ignored because the program seeks to perform a real time volumetric visualization of DMD where M-codes are not required. A sample G-code text file with zigzag path used in this research to perform the visualization test is provided in Appendix E.

### **2.4 Extraction of the Coordinate Values from the G-codes**

The program matches G-code lines starting with letters G, X, Y, and Z, characters. The corresponding coordinate values of X, Y and Z represents a 3D trajectory or a point. Coordinate values of X, Y, and Z constitute the path which a cross section will be swept through. Each parsed G-code line is given a line name and stored by the program in sequence from the first G-code line to the last G-code line. Instructions to install and run the program are found in Appendix B. The algorithm for parsing the G-code lines to extract the required coordinate values is provided in Figure 2.1.

**ALGORITHM FOR PARSING G-CODE:**

1. Import G-code as a text file
2. Match letters X, Y, Z and G-code in text file
3. Match floating points in text file
4. Match spaces in text file
5. Check G-code line for validity
6. Lines with X,Y,Z and characters G constitute a valid line: else line is invalid
7. Set line number= 0
8. Start parsing the G-code text file
9. Line number = line number +1 (naming the parsed lines)
10. Elements = X,Y,Z coordinate values in line (9)
11. For all non zero elements: store X,Y,Z values in a string format
12. Define Ret =[]
13. Append Ret (elements)
14. Return Ret (return all the coordinate values for every line parsed in sequence)

**Figure 2.1. Algorithm for parsing G-codes**

Below is an example of a three G-code line to be parsed:

M108 R3.146

G1 X28.63 Y28.64 Z0.15 F1080.0

G1 X28.63 Y-28.64 Z0.15 F1080.0

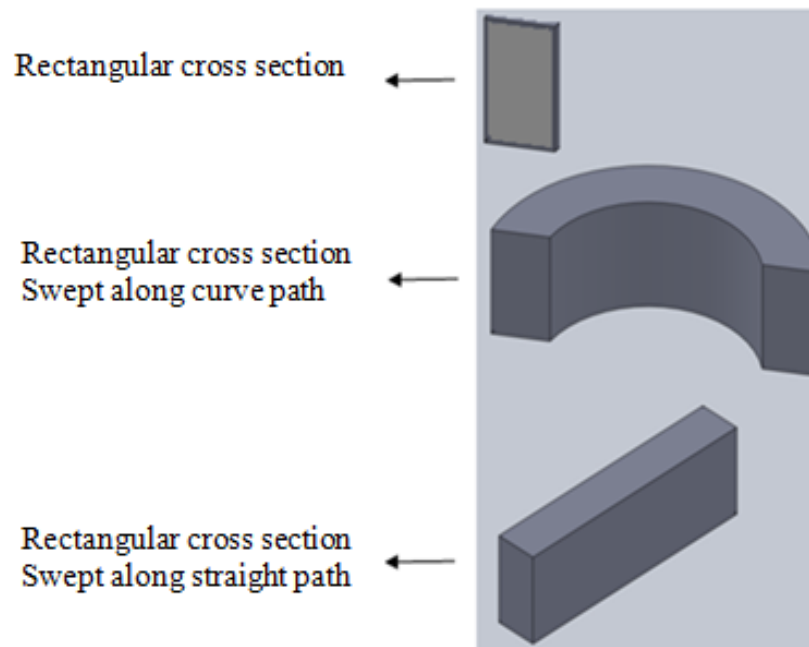
After parsing the G-code above, the information below is extracted for the DMD visualization process.

('G1', {'Y': 28.64, 'X': 28.63, 'Z': 0.15})),

('G1', {'Y': -28.64, 'X': 28.63, 'Z': 0.15})),

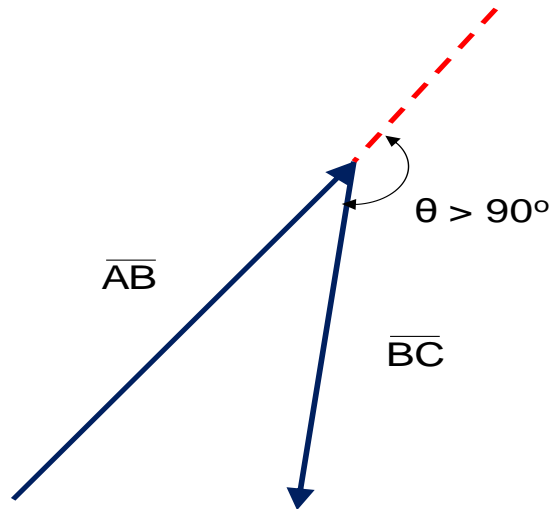
## 2.5 Swept Volume Procedure

The extrusion object library of Vpython is used to define all the 3D trajectories or points of the G-code as a swept path. Sweep is performed by sweeping a cross section through a defined swept path; the volume created through the sweep process is called the swept volume. The paths are defined by the 3D trajectories or points (X, Y, Z) extracted from the G-codes. The path starts from the first G-code point ( $X_1, Y_1, Z_1$ ) to the last G-code point ( $X_n, Y_n, Z_n$ ) in sequence. The process of sweeping the cross section in real time forms the basis for the volumetric visualization of the deposition process. The rendered model should then look the same as the original 3D CAD model but a little bigger because of overlap during the sweeping process. Figure 2.2 below shows examples of a cross section swept along different paths.

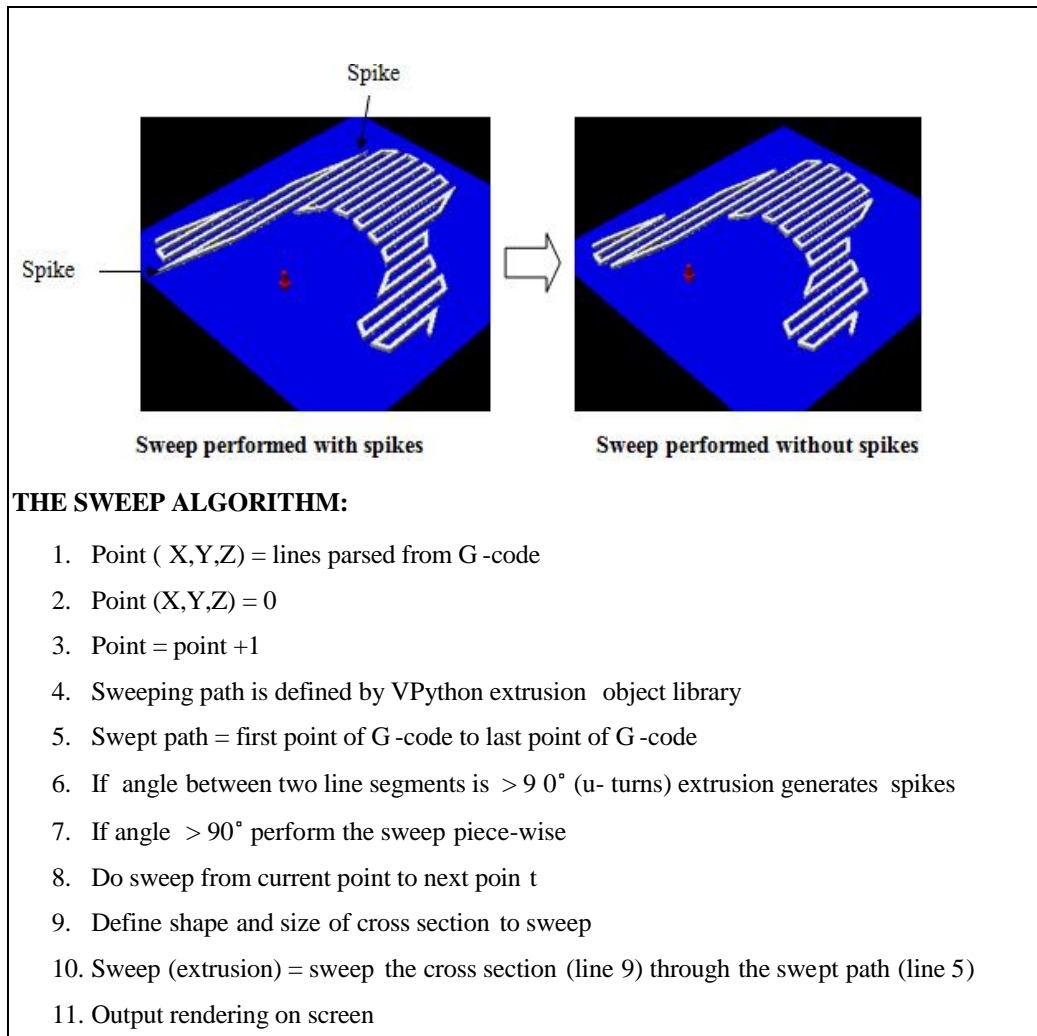


**Figure 2.2. A cross-sectional shape swept along a straight and a curve path.**

The extrusion object library is used to define the 3D trajectories or points for the sweep process. A line segment is defined by two successive points. If the angle between two line segments is greater than  $90^\circ$  as shown in Figure 2.3 below, the extrusion object library generates unwanted spikes when performing the sweep at the corner points. This problem is resolved by performing a piecewise extrusion when the angle is greater than  $90^\circ$ . In performing the piecewise extrusion, the extrusion object stops at the corner point where the angle is greater  $90^\circ$  and extrudes from that current point to the next point, hence the artificial spikes are not generated. This makes the sweep process and the rendered model look smoother. Figure 2.4 is the sweep implementation algorithm; it shows also extrusions with and without spikes. In the visualization program, “AnimatedPath.py” is the program compiler that defines all the points extracted from the G-code for the sweep process.



**Figure 2.3. Angle measured between two line segments**



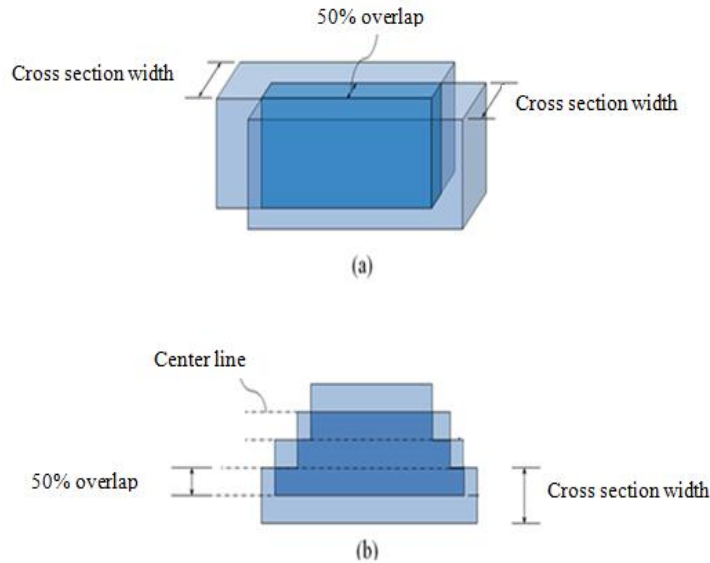
**Figure 2.4. Sweep algorithm and extrusion with and without spikes**

## 2.6 Cross Section Overlap

The track width used in the actual deposition process is taken into account when performing the sweep. The deposition track width becomes the width of the cross section for the sweep process. Solidness of the rendered model is a function of the cross section width and the extent to which the cross section will overlap. Experiments shows that the track width is close to the laser spot diameter, which is approximately  $2.54 \cdot 10^{-3}$  m

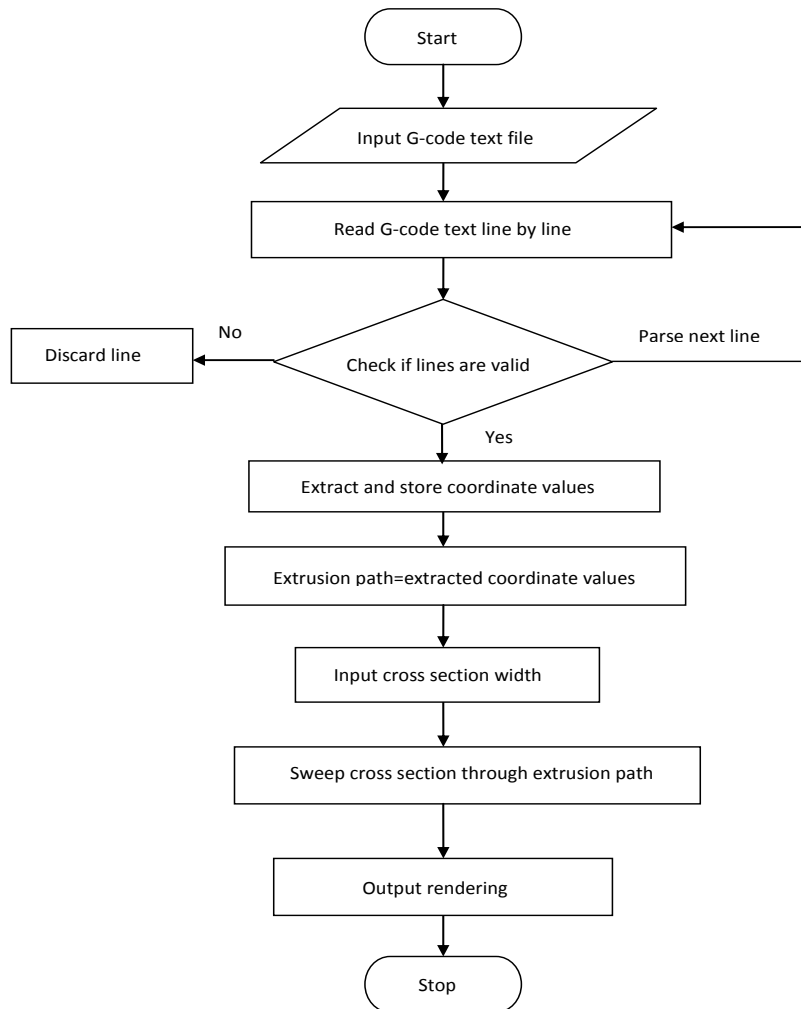


(0.1in/2.54 mm) with overlap of 50% at a nozzle standoff distance of  $1.27 \cdot 10^{-2}$  m (Lie *et al.*, 2009; Ren, *et al.*, 2010). Figure 2.5 below shows a 3D and 2D view of an overlapping cross section. The experimental parameters above are used for the visualization of the deposition process in this research unless otherwise stated. Figure 2.6 is the flow chart for visualization of the deposition process.



**Figure 2.5. (a) is 3D view of a cross section swept with 50% width overlap, (b) 2D top view showing the overlaps**

The cross section width and the percent overlap used in this research are based on actual deposition experiments performed at Missouri S&T University. The G-code used in this research to test the program was generated with a track width of 2.5 mm and center line spacing of 1.25 mm based on 50% track width overlap. The track width and the spacing between center lines in DMD are determined by the type of material (metal powder).



**Figure 2.6. Flow chart for visualization of the deposition process**

## 2.7 Direct Metal deposition Visualization Results

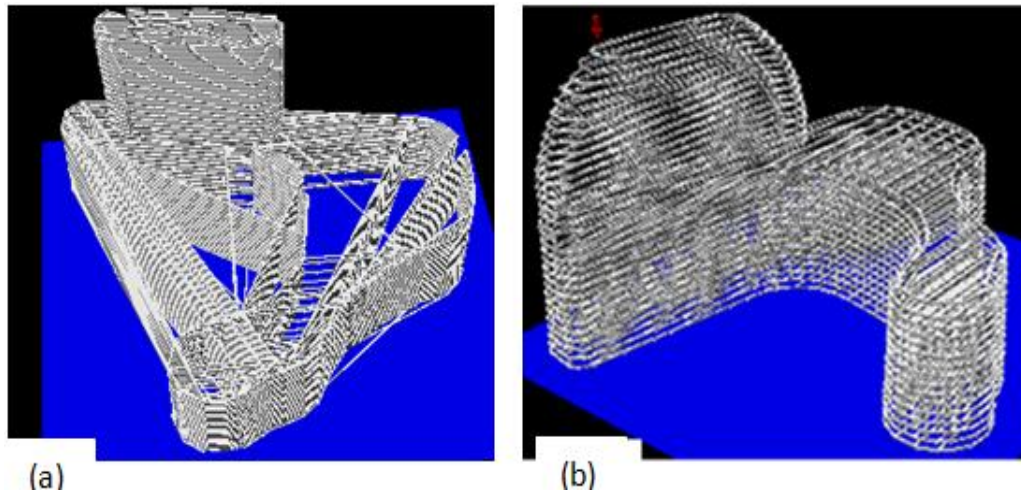
The program was tested using data from Missouri S &T University. The G-code text files can be found in the Appendix E. Data from Missouri S &T University includes a 3D CAD model, photo of the actual deposited part and the G-code used for the actual deposition. Figure 2.7 below shows the 3D CAD model and picture of the deposited part.

The G-code is generated for Fadal CNC machine (model VMC3016) as shown in section 1.4 Figure 1.2. Instructions to run the visualization program are in Appendix B.



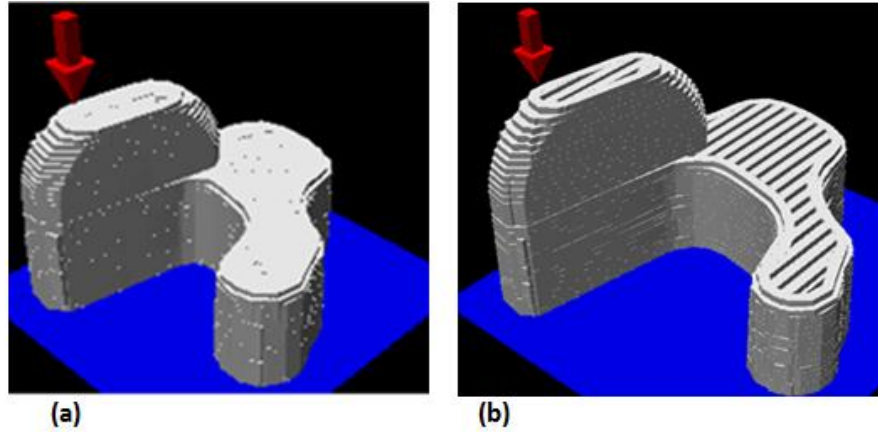
(a) (b)  
**Figure 2.7. (a) 3D CAD model, (b) Actual deposited part from MST**

The program was used to test and visualize the deposition process using the first G-code generated for the model. The visualization results as indicated in Figure 2.8 shows inaccurate G-code path that will not build the part to its actual geometry or specifications. This pre-process visualization result indicates to the user to check, correct or regenerate a new G-code for the deposition process because of inaccuracies with the first attempt. For every generated G-code, pre-process verification is performed until the correct G-code is generated to build a solid part to look like the actual model.



**Figure 2.8. (a) Inaccurate toolpath, (b) Desired toolpath**

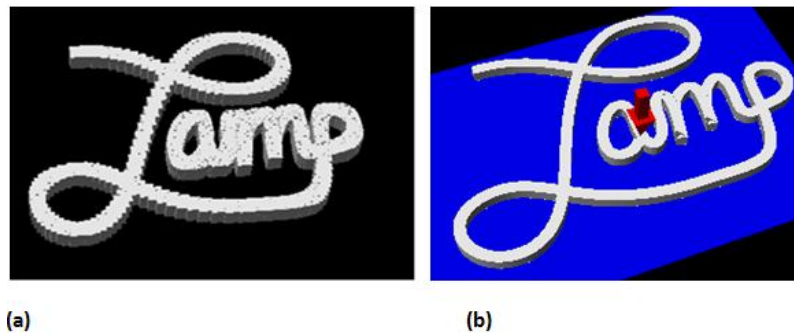
The rendered model should always retain the geometric features of the actual 3D CAD model and it must be solid along the build path without unwanted voids. The exactness of the rendered model geometry to that of the actual 3D CAD model depends on the width of the cross section, the extent of the cross section overlap within the center lines and the shape of the cross section. The cross section width used is normally the track width used in the actual deposition process. As indicated in section 2.6, the experimental parameters of 2.5 mm cross section width and 50% cross section overlap renders a part that is solid without voids. The rendered model will always come out solid when the cross section width overlaps within the center lines or when the cross section width is greater than 1.25 mm. Figure 2.9 (a) is a 2.0 mm cross-sectional width visualization result without voids, while Figure 2.9 (b) is a 0.8 mm cross-sectional width visualization result with voids because 0.8 mm is less than the 1.25 mm centerline spacing.



**Figure 2.9. (a) a 2.0 mm cross-sectional width, (b) a 0.8 mm cross-sectional width visualization results**

## 2.8 Comparison with Previous Work

The G-code generated for the LAMP logo of Missouri S&T University is tested using voxel placement (insertion) technique (Thatipalli, 2011) and swept volume technique used in this research. Whereas voxel placement needs further rotation of the placed voxel to get smooth surface the swept volume technique does not and it gives better surface smoothness. Figure 2.10 below shows the visualization results of the two techniques compared.



**Figure 2.10. (a) Voxel placement result, (b) Swept volume result.**

## **2.9 Observations and Conclusions**

- I. When models with changing geometry or slope sides are animated, the surface of the rendered model around those sides comes out uneven (stair case effect).
- II. The program sometimes runs slowly during real time animation of bigger models with massive G-codes.

Computer program for real time visualization of DMD process has been developed and implemented. The program is robust and capable of visualizing G-codes during the deposition process. The program is designed to output a one-time rendering or mimic the steady deposition process in real time. Users can pause the visualization process, zoom in or out for closer view. Users can also rotate and pan during the visualization process for different angle of views. The program is capable of visualizing any DMD process when the G-code format is the same as the one used in this research. The program can also be tweaked to parse other characters from the G-code other than X, Y, Z and G characters.

## CHAPTER 3

### COLLISION DETECTION OF THE DEPOSITION PROCESS USING OCTREE AND ORIENTED BOUNDING BOX

#### 3.1 Literature Review

Literature review on some of the techniques used for collision detection in 5-axis CNC manufacturing was presented (Tang *et al.*, 2007). Their collision detection algorithm was based on a sweep plane approach, where octree of bounding sphere algorithm is used. The colliding bounding spheres are further checked with the sweep plane algorithm to ensure false collision or interference is not reported. The algorithm presented in their paper is capable of performing collision detections between tool and work piece as well as between other parts of the CNC machine.

The method based on hierarchical orientated bounding box (OBB) and octree space partition for the global interference detection was also presented in 5-axis NC machining (Ding *et al.*, 2004). In this algorithm, the cutter and cutter holder are modeled by a hierarchical OBB structure, whereas the work piece surfaces are approximated by an octree. The interference detection is conducted between the tool OBBs and the gray octants of the surface octree with the separating axis theorem. They summarized and classified the various collision techniques into the following broad form i.e. Vector based methods, convex hull based methods, bounding volume methods, C-space based methods, Analytical methods, Swept volume methods and Space partition methods.

An algorithm for rapidly detecting, and correcting collision between a manually predefined tool and an arbitrary work piece was presented (Balasubramaniam *et al.*, 2002;

Balasubramaniam et al., 2003). The tool is modeled by using implicit equations and the work piece is modeled as a cloud of points. The algorithm is based on hierarchical bounding boxes, called k-DOPs, for both tool and work piece. The tool and work piece are preprocessed to form their respective hierarchical bounding boxes. The collision detection algorithm returns the points of the object that are in collision with the tool. However, the point-cloud representation for the work piece tends to lose efficiency as the number of sampled points is increased in order to obtain a good approximation for the machined part. Configuration space (C-space) approach to tool path generation that provides gouge-free and collision-free tool paths was also presented (Choi et al., 1997). However, the proposed approach is limited to three-axis machining. Jun et al proposed a searching method in the machining C-space to find the optimal tool orientation by considering the local gouging, rear gouging and global tool collision in five-axis machining (Jun *et al.*, 2003).

In conducting collision check and collision avoidance, tool path generation module, post processing and machine simulations are integrated into one system (Lauwers et al., 2003). The algorithm is use to detect collisions between the tool and work piece, the machine and part, the tool and machine or among moving machine components. However, this algorithm cannot be applied for a general form of the tool since a cylindrical approximation was assumed. Moreover, the change of the work piece geometry was not taken into account.

Classification of collision detection base on their systematic solving characteristics was presented (Jiménez *et al.*, 2001). The reason is that most collision



detection techniques are tailored to particular applications, others stem from theoretical concerns and their diverse origins and aims often hide their common ground which they all originate from. They broadly classified collision detection techniques as spatio-temporal intersection, swept volume interference, multiple interface detection and trajectory parameterization.

GAMMA research group of University of North Carolina (UNC) have compiled many open source collision detection libraries on their website. “V-COLLIDE” (Jonathan D. Cohen, 1998) is a collision detection library for large dynamic environments, and unite the N-body processing algorithm of I-COLLIDE with the pair processing algorithm of RAPID. Consequently, it is designed to operate on large numbers of static or moving polygonal objects, and the models may be unstructured.

“SOLID” (“SOLID 3.5,” 2007) is an open source library for interference detection of multiple 3D polygonal objects undergoing rigid motion. The shapes used by SOLID are polygon soups. The library exploits frame coherence by maintaining a set of pairs of proximate objects using incremental sweep and pruning on hierarchies of axis-aligned bounding boxes. Though slower for close proximity scenarios, its performance is comparable to that of V-COLLIDE in other cases.

“SWIFT++” (Ming Lin, 2001) from the GAMMA group is an open source library for collision detection approximation, exact distance computation, and contact determination between closed and bounded polyhedral models. It decomposes the boundary of each polyhedra into convex patches and pre-computes a hierarchy of convex

polytopes. It uses the SWIFT library to perform the underlying computations between the bounding volumes.

“PIVOT2D” (Kenneth E. Hoff III, 2001) from the GAMMA group is an open source software for collision detection. It computes generalized proximity information between arbitrary objects using graphics hardware. It uses multi-pass rendering techniques and accelerated distance computation, and provides an approximate solution for different proximity queries. These include collision detection, distance computation, local penetration depth, contact region and normals, etc. It involves no preprocessing and can handle deformable models.

“DeformCD” (Min Tang, 2007 ) from the GAMMA group is a fast collision detection library designed to accelerate calculation for deforming objects. Deforming objects, whose vertices are vibrating, an AABB refitting solution is used for collision detection. The efficiency of the AABB-refitting schema is compared with OBB-rebuild and AABB-rebuild schemas with timing. They achieved 5-10 times of speed up. Currently the library supports only windows platforms.

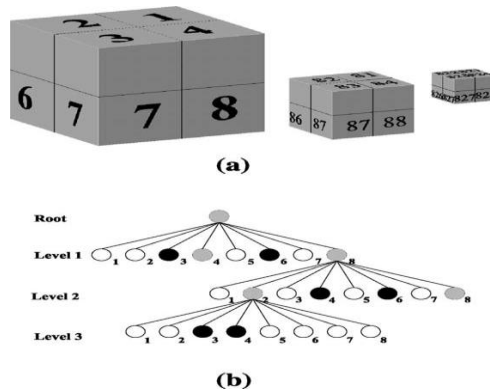
### **3.2 Selection of the Computer Programming Language**

C++ programming language was considered first because most of the open source collision detection libraries (SWIFT++ and SOLID) are implemented using C++. C++ was not chosen because more time is required for its mastery. PyOpenGL module which is built on Python was chosen so that the computer programs for visualization and collision detection will have the same platform. Open source Oriented Bounding Box

(OBB) intersection (Gomez, 1999) and open source Octree implementation (Kenwright, 2002) libraries are used as a framework to develop the collision detection program. Overview of PyOpenGL is in Appendix A.

### 3.3 Octree Data Structure

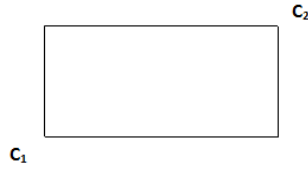
Octree is a hierarchical data structure that describes how the objects in a scene are distributed throughout the three dimensional space occupied by the scene (Trung Thanh *et al.*, 2007). Octrees as shown in figure 3.1 (Ding, *et al.*, 2004) below are created by recursive subdivision of a cube representing the parent node into eight smaller cubes called children; each child is then divided further into eight octants. Subdivision can proceed to any desired level of accuracy determined by the type of application and intended results. The level of subdivision is termed depth in this research. Blank octants are empty (0), grey octants (1) are partially full and black octants (2) are full. Octrees can therefore be used to represent a solid 3D part to its near-net shape.



**Figure 3.1. (a) 3D Structure of an octree and index codes of octants, (b) the tree structure of an octree**

### 3.4 Creation and Split of the Octree Boxes

The algorithm for creating the octree is based on (Kenwright, 2002) open source octree implementation algorithm. A bounding box is created to enclose the entire 3D CAD model; the bounding box is then split with three evenly divided planes to create eight child boxes within the top box. The procedure is repeated on each child until the desired depth is reached. The bounding boxes are represented in two ways in the program. When the box is axis-aligned then it is represented by two diagonal points, one at the corner of the box with the lowest coordinate values i.e.  $C_1$ , and the other one with the highest coordinate values i.e.  $C_2$  as shown in Figure 3.2.



**Figure 3.2. A simplified 2D axis aligned bounding box**

Splitting of the boxes is performed in axis-aligned boxes. This is because, first the tree is built and then they are rotated along with the mesh. The box in figure 3.2 above is defined by the coordinates  $C_1=(X_1, Y_1, Z_1)$  and  $C_2=(X_2, Y_2, Z_2)$ . The coordinates in  $C_1$  are lower than the ones in  $C_2$  i.e.  $X_1 < X_2$ ,  $Y_1 < Y_2$  and  $Z_1 < Z_2$ .

The center of the box is:

$$Ctr = \frac{(C_1 + C_2)}{2} \quad (3.1)$$

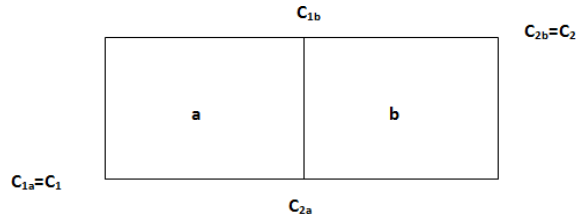
Dimension of the box is:

$$\mathbf{Size} = \mathbf{C}_2 - \mathbf{C}_1 \quad (3.2)$$

The distance between the two boxes from the center is:

$$dist = \sqrt{\Delta X^2 + \Delta Y^2 + \Delta Z^2} \quad (3.3)$$

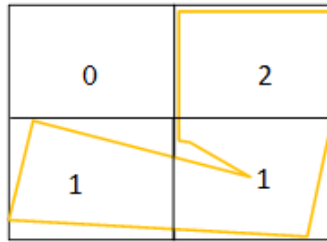
To split the box into two, there are three possibilities, left to right, front to back, or top to bottom. Suppose the x-axis is the one that goes horizontally, so if we are splitting the box defined by  $\mathbf{C}_1$  and  $\mathbf{C}_2$  into a right and left halves, the two new boxes (a and b) upper and lower coordinates will be given as:  $\mathbf{C}_{1a} = \mathbf{C}_1$ ,  $\mathbf{C}_{1b} = ((X_1+X_2)/2, Y_2, Z_2)$  and that of the second box will be  $\mathbf{C}_{2a} = ((X_1+X_2)/2, Y_1, Z_1)$  and  $\mathbf{C}_{2b} = \mathbf{C}_2$  as shown in Figure 3.3 below.



**Figure 3.3. Simplified 2D axis aligned bounding box split into two axis aligned bounding boxes**

The coordinate information of the octree boxes, equations 3.1, 3.2 and 3.3 above are used to develop and implement the collision detection program. In the collision detection program, “splitBox” is a function in “octree.py” compiler; this is the method that does the actual splitting of the box into two. In the program, “OctreeBoxIter” is a function in “octree.py” compiler; this is an iterator that gives the coordinates of the eight cells that would result from splitting a single box into eight sub-cells. That is, applying all three cuts: X, Y and Z at a time.

Each of those 8 cells is checked to see if it is fully occupied by the object, fully empty, or partially occupied. In the program, that value is stored in the occupancy list in “octree.py”, which stores an “int” for each of the eight cells. If the box content is 0, it means it is empty. When it is 1 means intersecting or partially full and when it is 2 means the box is fully enclosing its object. The idea is to leave one of the halves fully empty or fully contained if possible. For example, if it was 2D, then there would be two possible cuts: left-right, or top-bottom. Suppose a box as shown in Figure 3.4 below had the 4 sub-cells with occupancy values 0 2 and 1 1 .Here the program will choose to perform the splitting leaving one of the quarters fully empty (just the 0) and the other quarter fully occupied (2). The bottom quarters will be partially filled with occupancy values of 1.

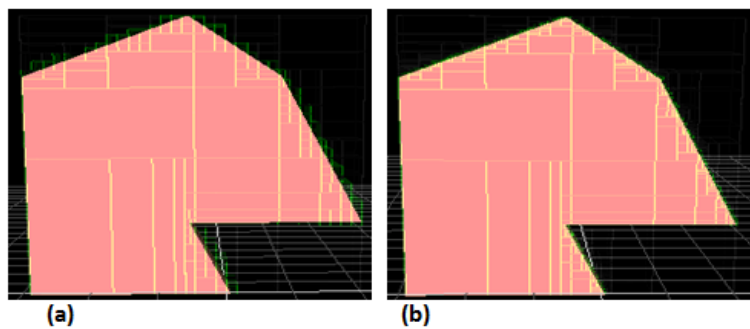


**Figure 3.4. A partitioned box showing it's occupancy values**

In “octree.py”, the occupancy list contains 8 values, one for each sub-cell. The list "test\_box\_indices" in octree.py tells the program which cells lie on the same half with respect to a given axis. For example: ((0, 2, 4, 6), 2) means, in axis 2 (z), cells 0, 2, 4 and 6 lie on the same side. In other words, if occupancy [0], occupancy [2], occupancy [4] and occupancy [6] are all 0 (empty) or all 2 (full), then the box should be cut slicing axis Z. Now, once a box is split into two, one half would be union of the 4 cells that were

tested that lied in that half, and the other half would be the other 4 cells. For example, if one half would be the union of cells (0,2,4,6), the other half would contain (1,3,5,7).

The number of partition required determines how close the octree structure will represent the 3D CAD model to its near- net shape. The algorithm used in this research sets the portioning depth to 9 so as to free up some computer graphic memory. Figure 3.5 below is an example of an octree representation of a 3D CAD model. It shows a 9 and 15 depth of partitioning of the 3D model and its octant occupancy. The 9 depth portioning as can be seen shows partially filled grids along the slant side of the model; hence the OBBs are not tightly fitting the model. The 15 depth of portioning as can be seen shows the OBBs are tightly fitting the model along its slant sides. This is so because, during the 15 depth partitioning all partially filled grids are further refined by splitting. This makes the OBBs of the model fit tightly along the slant sides having partially filled grids. Figure 3.6 is the algorithm for creating octree-OBBs of the CAD models. The Python compiler that creates the octrees is “octree.py” in Appendix C, it also references “cube.py” and “intersections.py” compilers.



**Figure 3.5. Octree representation of a 3D CAD model showing (a) 9 depth partitioning, (b) 15 depth partitioning**

#### **OCTREE CREATION ALGORITHM**

1. Input 3D CAD in STL ASCII format
2. Extract min and max X,Y,Z coordinates of the CAD model
3. Create the bounding box to contain the CAD model
4.  $C_1$  = min lower end of bounding box coordinates ( $X_1, Y_1, Z_1$ )
5.  $C_2$  = max upper end of bounding box coordinates ( $X_2, Y_2, Z_2$ )
6. Center of box =  $(C_1 + C_2)/2$
7. Size of box =  $C_2 - C_1$
8. Split the bounding box into eight octants using three planes (top-down, front-back, right-left)
9. Check split boxes occupancy: empty=0, partially full=1, full =2
10. If octants in (9) are partially full further split octants into eight children
11. For created octants in step 10, update the bounding box coordinates of the octants
12. Perform octant splitting nine times (depth =9)
13. Create OBB-Octrees of CAD model

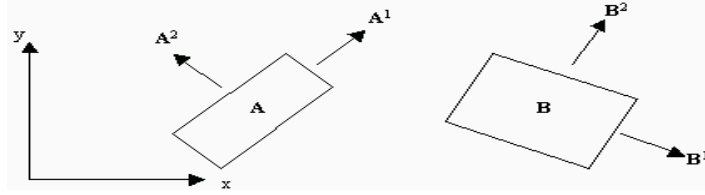
**Figure 3.6. Algorithm for creating Octree-OBB of the CAD models**

### **3.5 Oriented Bounding Box Collision Detection Test and Results**

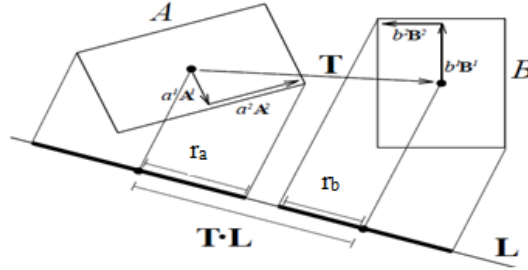
The separating axis theorem as illustrated in Figures 3.7 (Gomez, 1999) and Figure 3.8 (Gottschalk *et al.*, 1996) are used to implement the collision test between OBBs of the 3D CAD models. Splitting of the 3D CAD model using octrees as outlined in section 3.4 effectively represents the actual 3D model to a certain resolution. The octrees are rotated with their objects so are the bounding boxes too. Hence, the axis aligned boxes of the octants behave as oriented bounding boxes (OBB) rotating with their local coordinates. The collision detection is performed at the local point where the two CAD models make contact. The contacting OBBs at that point are checked for collision. Interference occurs when there is an overlap along the local maximum and minimum



coordinate axis  $x, y, z$  of the contacting OBBs. If the line distance between two OBBs is greater than the sum of the two radii then the OBBs do not intersect.  $\mathbf{L}$  is the reference axis of approach to project the radii of the two OBBs onto.  $\mathbf{T}$  is the vector from one OBB center to another.  $\mathbf{A}^1, \mathbf{A}^2, \mathbf{B}^1$  and  $\mathbf{B}^2$  are the local coordinate axis of boxes A and B. Figures 3.7 and Figure 3.8 below explain the principle where  $\mathbf{L}$  is a unit vector.  $a_1, a_2$  and  $a_3$  are half-widths (or radii) of box A.



**Figure 3.7. Oriented bounding box (OBB) with local axis**



**Figure 3.8. The vector  $\mathbf{L}$  forms a separating axis**

The radius of the projection of box A onto  $\mathbf{L}$  is

$$r_a = a_1 |\mathbf{A}^1 \cdot \mathbf{L}| + a_2 |\mathbf{A}^2 \cdot \mathbf{L}| + a_3 |\mathbf{A}^3 \cdot \mathbf{L}| \quad (3.5)$$

The same is true for B, and  $\mathbf{L}$  forms a separating axis if

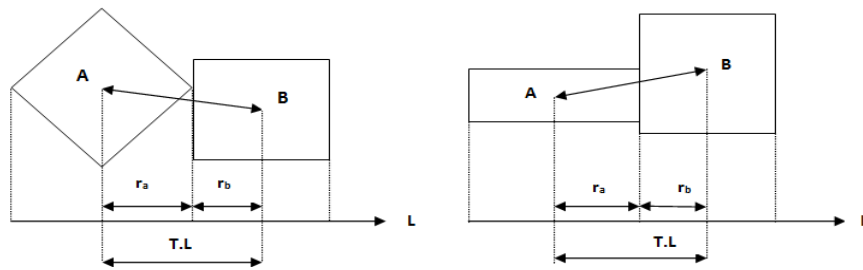
$$|\mathbf{T} \cdot \mathbf{L}| > r_a + r_b \quad (3.6)$$

Note that  $\mathbf{L}$  does not have to be a unit vector for this test to work. The boxes A and B are disjoint if none of the 6 principal local axes and their 9 cross products forms a separating axis (Gomez, 1999). Figure 3.9 below is the algorithm for collision detection base on the separating axis theorem, Python compiler “intsersections.py” in Appendix C performs the collision detection which also references “linear.py”. Figure 3.10 below is a case where two OBBs will collide when  $|\mathbf{T} \cdot \mathbf{L}| = r_a + r_b$ .

**COLLISION DETECTION ALGORITHM:**

1. From separating axis theorem
2. Check collision between any two OBBs of CAD models within proximity
3.  $r_a$  and  $r_b$  define the projected radiuses of two bounding boxes(OBB)
4.  $\mathbf{T}$  is the line distance between the two bounding boxes
5. If  $|\mathbf{T} \cdot \mathbf{L}| > r_a + r_b$  no collision between boxes
6. If  $|\mathbf{T} \cdot \mathbf{L}| < r_a + r_b$  will intersect
7. If  $|\mathbf{T} \cdot \mathbf{L}| = r_a + r_b$  the boxes will collide
8. If step (7) is satisfied
9. Report collision = true
10. Else report collision = false

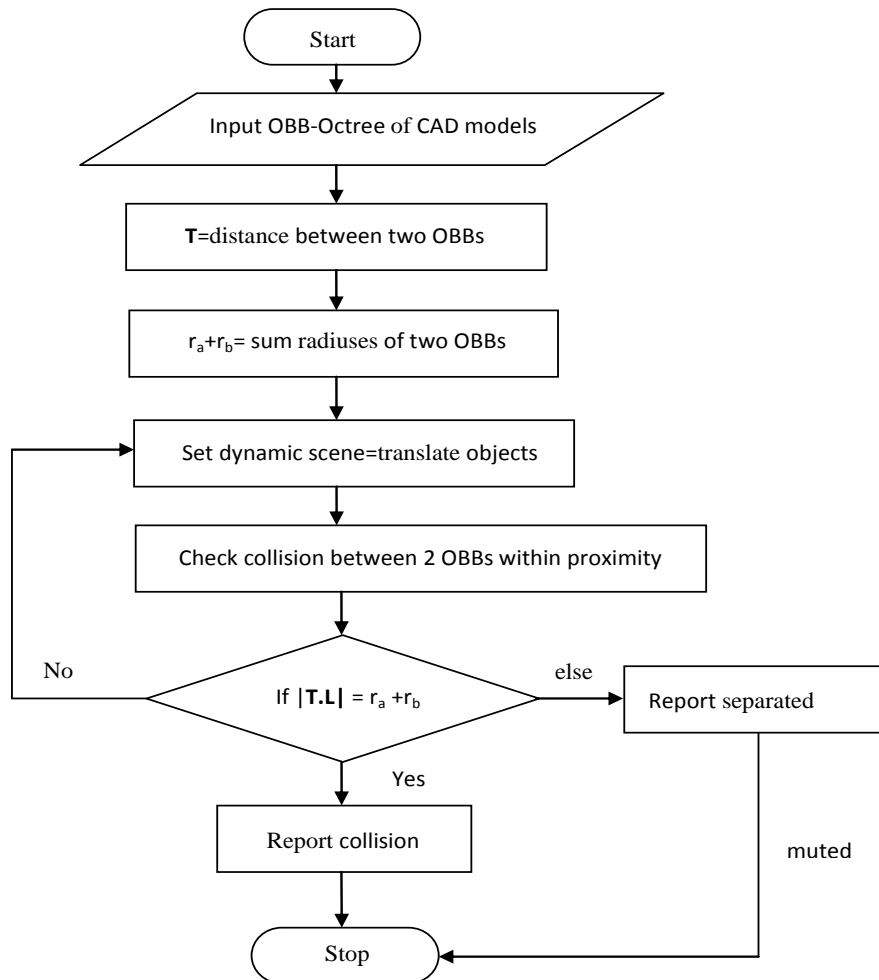
**Figure 3.9. Algorithm for separating axis collision detection**



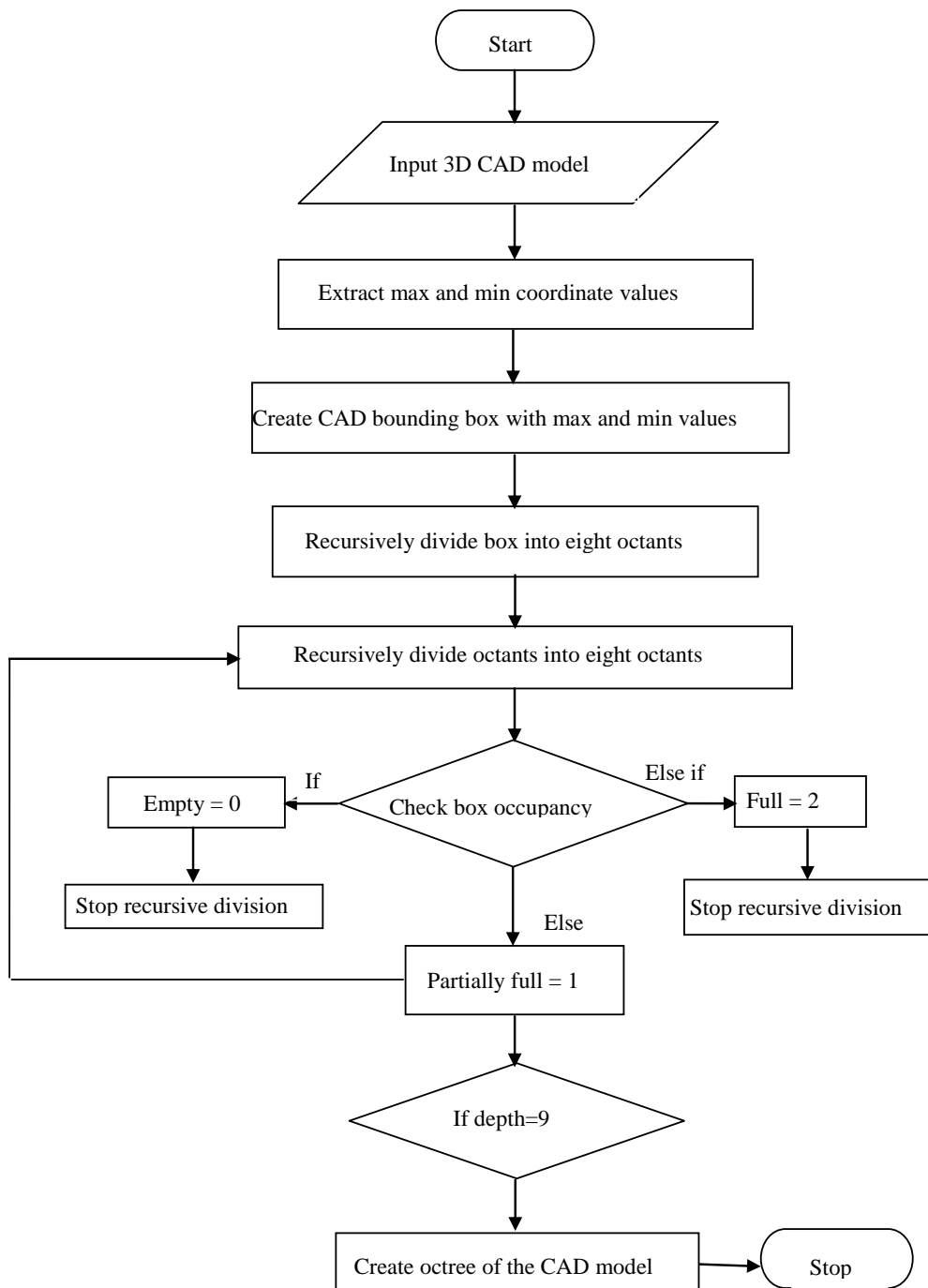
**Figure 3.10. Two sets of OBBs in collision**

In the program, “boxBoxOverlap” is a function in “intersections.py” compiler that performs the collision detection between any two contacting OBBs in the graphic scene. When the two OBBs collide then the line distance between the centers of the touching OBBs is equal to the sum of the projected radii of the two contacting OBBs. If we were interested in the intersection of the contacting OBBs, the sum of the two contacting OBBs radii will have to be greater than the line distance between the OBBs centers. It also means the OBBs are overlapping along all six coordinate axis of x, y and z. Figure 3.11 is the flow chart for the OBB collision detection based on the separating axis theorem, refer to “intersections.py” which also references “linear.py” in Appendix C. Figure 3.12 is the flow chart for the creation of octrees, “Octree.py” which also references “cube.py” and “intsersections.py” in Appendix C performs the octree creation of the CAD models. In Appendix D are instructions on how to input or change the CAD models. The collision detection test starts with an input of the 3D CAD models, the octree representation of the CAD model is then created with each octant or child placed in its OBB. Base on the separating axis theorem, the CAD models are translated within the graphic scene to test for collision. If the sum of the projected radii and the line distance between two contacting OBBs are equal then the OBBs are colliding which also means the two components are colliding at that point. At the point where the OBBs make contact, the program performs a routine operation to determine if other octants within proximity are also colliding. If any two OBBs of the CAD models at the point of contact satisfy the condition of the separating axis theorem, the program will detect collision or the translation of the movable component comes to a halt.

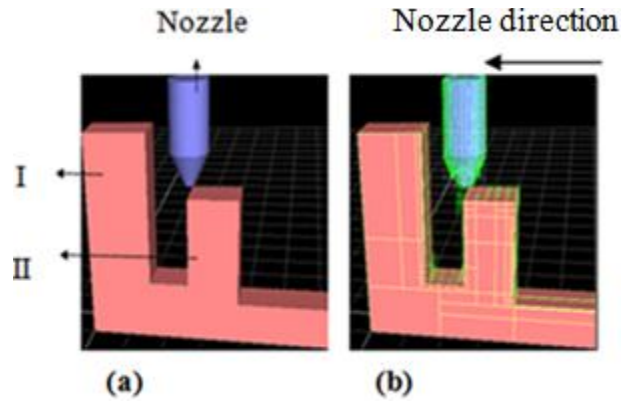
The collision detection program was used to perform rigid body collision detection test between a 3D CAD STL model of a workpiece and a 3D CAD STL model of a cutting tool in a graphic scene. The CAD model of the workpiece was design to have an already built component I. Figure 3.13 **(a)** and **(b)** shows a build process where component II is to be built. As can be seen, the nozzle does not collide with component I and will build component II without the nozzle colliding with component I. Figure 3.14 **(a)** and **(b)** also shows a built process where the nozzle collides with component I as it builds component II to its middle section. Instructions to run the collision detection program are in Appendix C.



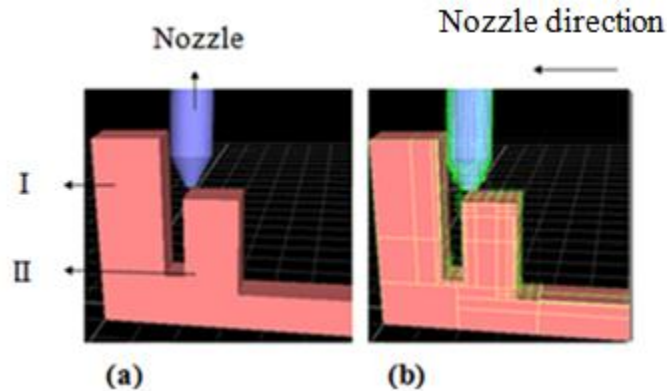
**Figure 3.11. Flow chart for OBB collision detection**



**Figure 3.12. Flow chart for Octree creation**



**Figure 3.13. (a) No collision between component I and nozzle, (b) Octree representation of the model**

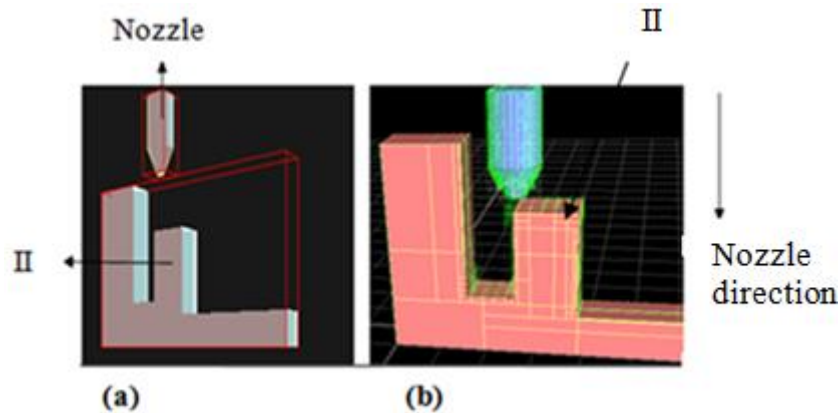


**Figure 3.14. (a) Nozzle collides with component I, (b) Octree representation of the model**

### 3.6 Comparison with Previous Work

The octree based collision detection program in this research was compared with AABB collision detection (Thatipalli, 2011) by testing. In Figure 3.15 the two techniques (a) and (b) are used to perform a collision detection test between a CAD model of a workpiece and a CAD model of a nozzle. The nozzle target point is to the top middle of component II. In Figure 3.15 (b), the octree-OBB collision detection technique represents

the CAD models better with the OBBs tightly fitting the CAD models; collision between the CAD models is more realistic since the models are closely touching. In Figure 3.15 (b), the single AABB technique and its representation of the CAD models are very conservative. Collision between CAD models bounded by a single AABB is not very realistic since the models are not closely touching. The AABB technique leaves a lot of space between the CAD model and its bounding box.



**Figure 3.15. (a) Single axis aligned bounding box collision test, (b) Octree oriented bounding box collision detection test**

### **3.7 Observations and Conclusions**

- I. For the octree to represent a CAD model with tightly fitting OBBs, the splitting process requires higher depth of partitioning this makes the program run slow. The program sometimes runs slow when the CAD models to be partitioned have quadratic surfaces.
- II. The program is ideal for rigid body collision detection where the CAD model is decomposed to plan the build sequence before deposition commences.



Computer program for collision detection has been developed and implemented using octree-OBBs. The program can detect the collision between CAD models of two components within a graphic scene; the program used octrees to represent the CAD models to their near-net shapes for enhanced collision detection tests using higher depth of partitioning. Users can pan and rotate the models within the graphic scene for different angle of views, perform zoom in and out operations. Users can also toggle between STL mesh surfaces and octree-OBB boxes, or have the STL mesh lines and octree-OBB boxes removed.

## **CHAPTER 4**

### **DISCUSSION AND RECOMMENDATION**

#### **4.1 Summary and Discussion**

Swept volume technique has been used to develop and implement computer program for visualization of the deposition process in this research. The program extracts coordinate information from G-codes and uses them as swept path. Users can pause the active visualization process, zoom in and out for closer view. Users can rotate and pan during the visualization process for different angle of views during the sweep process.

Octree-OBBs have been used to develop and implement computer program, to detect collision between CAD models of two components within a graphic scene. The CAD models of the components are well represented to their near-net shapes by the octrees for enhanced collision detection. Users can rotate the models in the graphic scene for different angle of views. Users can choose which model to translate at any time and flip between octrees boxes and STL meshes.

“SWIFT++” is open source collision detection library software developed by GAMMA group at University of North Carolina at Chapel Hill. It is a robust program that can perform collision detection tests such as exact distance computation, contact determination and tolerance. This program could not be used because there are bugs which needed to be fixed. “SOLID” is another open source collision detection software considered, the bugs were fairly easy to fix. The program is version 3.5.6 developed with a single axis aligned bounding box. SOLID reports objects penetration depth, it does not

use hierarchies of bounding boxes which makes it not very suitable for contact collision detection.

#### **4.2. Recommendation and Future Work**

The stair case effect that makes the surface of rendered models uneven due to change of object geometry or slope sides should be looked at and made finer. Optimization of the program to speed up the visualization process of models with massive G-code data should also be looked at.

Collision detection with octree should look at growth of the part being deposited. This requires updating the oriented bounding boxes (OBBs) as the part geometry changes. OBB-Rebuild can solve this problem or hierarchies of axis aligned bounding boxes (AABB). “DeformCD” is an open source collision detection library used for collision detection of deformable objects. It uses the OBB-Rebuild technique for collision detection test. This program shows great promise for future work of DMD collision detection because it accounts for part changes or deformity during the deposition process.

Future work should also be geared towards transitioning the collision detection program to follow the tool path during deposition. One way to do this is to develop a python script compiler that will extract and update the coordinate values from the G-code for the translation of the CAD models. This will effectively eliminate the manual keyboard manipulation used to translate the models to perform collision detection.

## REFERENCES

- Balasubramaniam, M., Ho, S., Sarma, S., & Adachi, Y. (2002). Generation of collision-free 5-axis tool paths using a haptic surface. *Computer-Aided Design*, 34(4), 267-279.
- Balasubramaniam, M., Sarma, S. E., & Marciniak, K. (2003). Collision-free finishing toolpaths from visibility data. *Computer-Aided Design*, 35(4), 359-374.
- Ch. Sweta Dhaveji, T. E. S., Jianzhong Ruan, Frank W. Liou. (2011). Generic Visual Simulation of Manufacturing Equipment.
- Chandru, V., Manohar, S., & Prakash, C. E. (1995). Voxel-based modeling for layered manufacturing. *Computer Graphics and Applications, IEEE*, 15(6), 42-47.
- Choi, B. K., Kim, D. H., & Jerard, R. B. (1997). C-space approach to tool-path generation for die and mould machining. *Computer-Aided Design*, 29(9), 657-669.
- Dhaveji, C. S., Sparks, T. E., Ruan, J., & Liou, F. W. (2011). *Octree Approach for Simulation of Additive Manufacturing Toolpath*. Paper presented at the International Solid Freeform Fabrication Symposium.
- Ding, S., Mannan, M. A., & Poo, A. N. (2004). Oriented bounding box and octree based global interference detection in 5-axis machining of free-form surfaces. *Computer-Aided Design*, 36(13), 1281-1294.
- Don, L. (1991). Modeling dynamic surfaces with octrees. *Computers & Graphics*, 15(3), 383-387.
- Gomez, M. (1999). Simple Intersection Tests For Games. Retrieved 03/14/2012, from [http://www.gamasutra.com/view/feature/131790/simple\\_intersection\\_tests\\_for\\_games.php?page=5](http://www.gamasutra.com/view/feature/131790/simple_intersection_tests_for_games.php?page=5)
- Gottschalk, S., Lin, M. C., & Manocha, D. (1996). OBBTree: {A} Hierarchical Structure for Rapid Interference Detection. *Computer Graphics*, 30(Annual Conference Series), 171-180.
- Jee, H. J., & Sachs, E. (2000). A visual simulation technique for 3D printing. *Adv. Eng. Softw.*, 31(2), 97-106.
- Jiménez, P., Thomas, F., & Torras, C. (2001). 3D collision detection: a survey. *Computers & Graphics*, 25(2), 269-285.

- Jonathan D. Cohen, S. G., Dinesh Manocha. (1998). V-COLLIDE. (03/05/2012). Retrieved from <http://gamma.cs.unc.edu/V-COLLIDE/>
- Jun, C.-S., Cha, K., & Lee, Y.-S. (2003). Optimizing tool orientations for 5-axis machining by configuration-space search method. *Computer-Aided Design*, 35(6), 549-566.
- Kenneth E. Hoff III, A. Z., Ming Lin, Dinesh Manocha. (2001). PIVOT. (03/14/2012). Retrieved from <http://gamma.cs.unc.edu/PIVOT>
- Kenwright, B. (2002). Octrees and 3D Worlds. Retrieved 03/14/2012, 2012, from <http://www.xbdev.net/mathsf3d/octree/tutorial/index.php>
- Lauwers, B., Dejonghe, P., & Kruth, J. P. (2003). Optimal and collision free tool posture in five-axis machining through the tight integration of tool path generation and machine simulation. *Computer-Aided Design*, 35(5), 421-432.
- Lie, T., Jianzhong, R., Sparks, T. E., Landers, R. G., & Liou, F. (2009, 10-12 June 2009). *Layer-to-layer height control of Laser Metal Deposition processes*. Paper presented at the American Control Conference, 2009. ACC '09.
- Min Tang, D. M. (2007 ). DeformCD 1.0. 2012(03/15/2012). Retrieved from <http://gamma.cs.unc.edu/DEFORMCD/index.html>
- Ming Lin, S. E. (2001). SWIFT++. Retrieved from <http://gamma.cs.unc.edu/SWIFT++/download.html>
- Ren, L., Sparks, T., Ruan, J., & Liou, F. (2010). Integrated Process Planning for a Multiaxis Hybrid Manufacturing System. *Journal of Manufacturing Science and Engineering*, 132(2).
- Sachs, E., Cima, M., & Cornie, J. (1990). Three-Dimensional Printing: Rapid Tooling and Prototypes Directly from a CAD Model. *CIRP Annals - Manufacturing Technology*, 39(1), 201-204.
- SOLID 3.5. (2007). 2012(18/01/2012). Retrieved from <http://www.dtectta.com/>
- Tang, T. D., Bohez, E. L. J., & Koomsap, P. (2007). The sweep plane algorithm for global collision detection with workpiece geometry update for five-axis NC machining. *Comput. Aided Des.*, 39(11), 1012-1024.
- Thatipalli, A. R. (2011). *Visualization Of Laser Deposition Process Of A 5-Axis CNC Machine Using A Voxel Based System*. Unpublished Masters Project Report, Missouri S&T University, Missouri.

Trung Thanh, P., Yong Hyun, K., & Sung Lim, K. (2007, 26-29 Aug. 2007). *Development of a Software for Effective Cutting Simulation using Advanced Octree Algorithm*. Paper presented at the Computational Science and its Applications, 2007. ICCSA 2007. International Conference on Computational Science and Its Applications (ICCSA).

## APPENDIX A

### VYPYTHON, NUMPY AND PYOPENGL OVERVIEW

#### ***VYPYTHON:***

VPython is the Python programming language plus a 3D graphics module called "Visual". VPython is a simple rendering tool for 3D objects and graphs. VPython allows users to create objects such as spheres and cones in 3D space and displays these objects in a window. Real-time, navigable 3D animations are generated as a side effect of computations. (<http://www.vpython.org/index.html>).

VPython is still being developed with a lot of contributions and modifications coming from independent contributors. The visualization program developed in this thesis made use of some of the latest additions to VPython library. The research program was developed using the new extrusion object library of VPython aided by the shape and path libraries. These libraries constituted the main frame of the research. The program was designed and implemented on windows 7 platform. How to install VPython on windows is in Appendix B.

#### ***PYOPENGL:***

PyOpenGL is the cross platform Python binding to OpenGL and related APIs. The binding is created using the standard Python 2.5 ctypes library, and is provided under an extremely liberal BSD-style Open-Source license. PyOpenGL includes support for OpenGL v1.1 through 3.2, GLU, GLUT v3.7 (and FreeGLUT), and GLE 3. It also includes support for hundreds of OpenGL extensions (<http://pyopengl.sourceforge.net/>).

PyOpenGL was chosen as the platform to develop the collision detection program in this research because of prior experience with OpenGL. NeHe Productions ([http://nehe.gamedev.net/tutorial/lessons\\_01\\_05/22004/](http://nehe.gamedev.net/tutorial/lessons_01_05/22004/)) has a lot of tutorials on various 3D topics in OpenGL. This made the use and learning of PyOpenGL fairly easy because of free available tutorials with source codes. PyOpenGL was mainly used to handle the graphic aspect of the collision detection program. The OpenGL graphic library does the rendering when STL CAD files are imported and displayed in the window. Most of PyOpenGL dependencies come with the installation of Python-2.7. How to install PyOpenGL is provided in Appendix C or from (<http://www.lfd.uci.edu/~gohlke/pythonlibs/>).

#### ***NUMPY:***

Numpy is an extension of the Python programming language. It adds support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays and matrices. The Numpy libraries are used in this research to perform the matrix and vector computations when translating or rotating the CAD models. How to install Numpy is provided in Appendix C or from (<http://www.lfd.uci.edu/~gohlke/pythonlibs/>)

## APPENDIX B

### INSTRUCTIONS TO INSTALL AND RUN VISUALIZATION PROGRAM ON WINDOWS

#### *INSTALL PYTHON AND ITS EXTENSIONS TO RUN THE PROGRAM:*

1. Install “**Python-2.7**” from this link ([http://www.vpython.org/contents/download\\_windows.html](http://www.vpython.org/contents/download_windows.html)).  
When the page is displayed scroll down the screen till you get to this portion of the screen shown below, click “**python-2.7**” in oval to install.

**Windows downloads (XP, Vista, Windows 7)**  
  
Several versions of Python are in common use. The versions of Python and VPython must match. If you are new to VPython, see [How to run VPython](#).  
  
**Python 2.7**  
  
First, download and install **Python-2.7.2** (Important: Let it install in C:\Python27)  
(There is not a VPython version for Windows 64-bit Python, but this 32-bit version of Python works fine on 64-bit Windows machines)  
Second, download and install **VPython-Win-Py2.7-5.73**

2. Follow this link to download and install “**VPython and python 2.7**” from:
  - ([VPython-Win-Py2.7-5.73](#)) this page is direct link to download VPython and Python
3. Or from ([http://www.vpython.org/contents/download\\_windows.html](http://www.vpython.org/contents/download_windows.html)). This will install **Python 2.7** and **VPython 5.7** simultaneously. When the page is displayed scroll down the screen till you get to this portion of the screen shown below, click “**VPython-Py-2.7-5.72**” in oval to install:

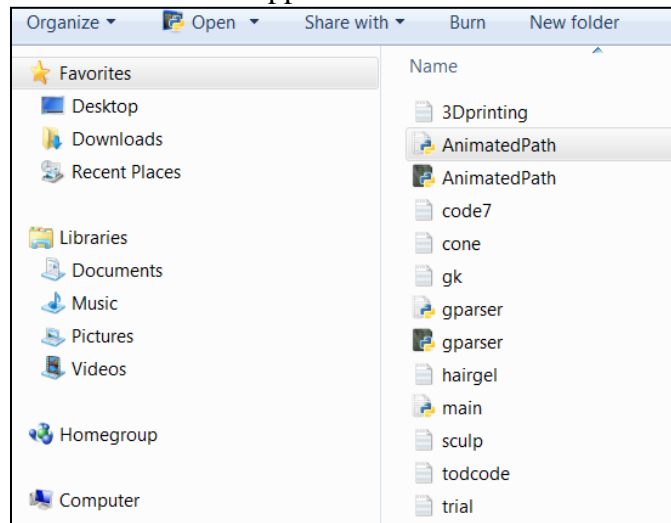
**Windows downloads (XP, Vista, Windows 7)**  
  
Several versions of Python are in common use. The versions of Python and VPython must match. If you are new to VPython, see [How to run VPython](#).  
  
**Python 2.7**  
  
First, download and install **Python-2.7.2** (Important: Let it install in C:\Python27)  
(There is not a VPython version for Windows 64-bit Python, but this 32-bit version of Python works fine on 64-bit Windows machines)  
Second, download and install **VPython-Win-Py2.7-5.73**

This last stage completes the installation to run the visualization program.

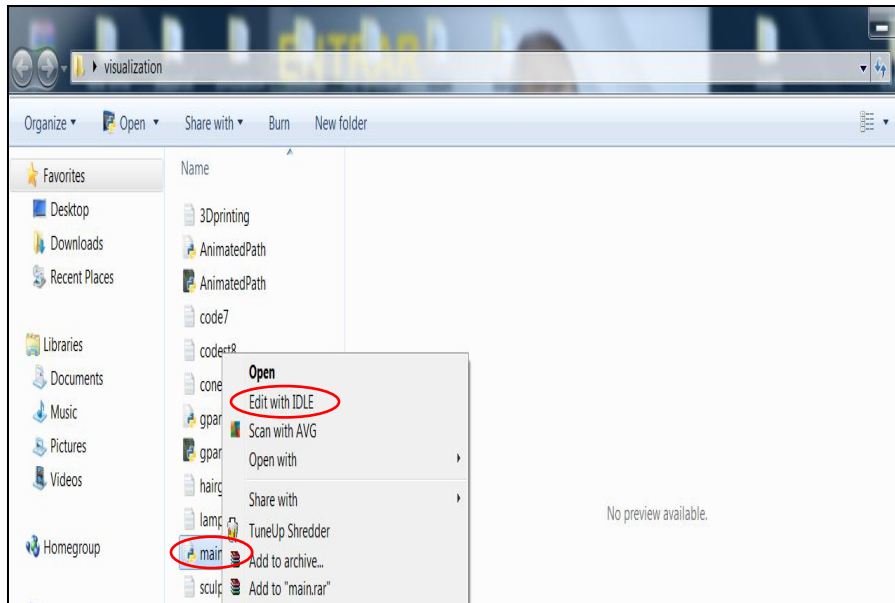


### **HOW TO RUN THE VISUALIZATION PROGRAM:**

1. Unzip folder “**visualization**” extract and save all the files into one folder. All G-codes are text files and have (.txt) extension. All Python files have (.py) extension. The files in the folder appear on screen like this below when extracted:



- File “**gparser.py**” is a python script compiler for extracting X,Y and Z coordinate values
  - File “**AnimatedPath.py**” is a python script compiler for defining the swept path
  - File “**main.py**” is the main python script used to run the visualization program, it references compilers “**gparser.py**” and “**AnimatedPath.py**”
  - Files “**todcode, lamplogo, gkn,trial**” are G-code text files from MST
  - Files “**sculp, hairgel, cone, 3Dprinting**” are G-code text files generated using open source 3D printing G-code generator called **ReplicatorG** to test the program.
2. To run the program execute: “**main.py**”. Right click “**main**” and then click (**Edit with Idle**) this opens the python script in the idle. See snap shot below



When this operation is performed, the window shown below will be displayed on the screen. This is the first seventeen command lines of the python script.

```

main.py - C:\Users\hamza\Desktop\visualization\main.py
File Edit Format Run Options Windows Help

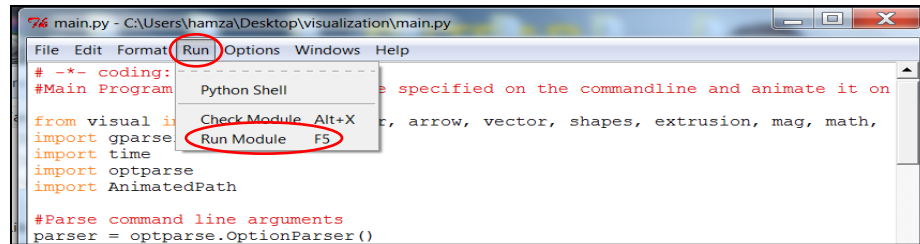
from visual import display, color, arrow, vector, shapes, extrusion, mag, math, box
import gparser
import time
import optparse
import AnimatedPath

#Parse command line arguments
parser = optparse.OptionParser()
parser.add_option("-t", "--time", default=750,
                  action="store", type="float", dest="animation_duration",
                  help="How many seconds the animation should last", metavar="SECONDS")
parser.add_option("-n", "--nsides", default=4,
                  action="store", type="int", dest="extrusion_ngon",
                  help="How many facets on the extrusion", metavar="NSIDES")
parser.add_option("-w", "--width", default=0.8,
                  action="store", type="int", dest="extrusion_width",
                  help="Width of the extrusion", metavar="WIDTH")
parser.add_option("-f", "--full", "--full-screen", default=False,
                  action="store_true", dest="fullscreen",
                  help="Full screen mode")
(options, args) = parser.parse_args()
print " KEYBOARD FUNCTIONS"
print "press space bar : to pause/play animation"
print "press + and - : to speed/slow animation"
print "press arrow keys: to play animation back and forth"
print "press 0 and 9 : move to begining and end of animation"
print "left and right click mouse and hold: drag to zoom in and out"

#User can specify a single file to open. If none is specified, we open a default file
if len(args) == 0:
    args=["todcode.txt"]
if len(args) != 1:
    parser.error("incorrect number of arguments, you should specify ONE G-code file")

```

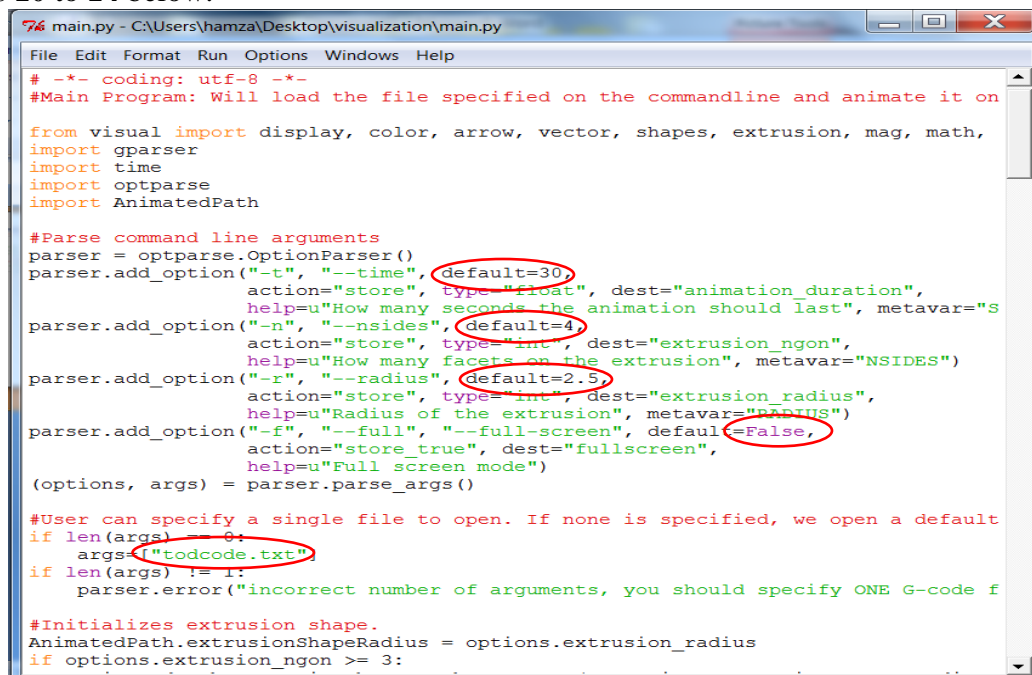
- When the window above appears press key “F5” on the keyboard to run the animation/simulation. Or from the task bar click ‘run’ click “Run Module with F5” see snap shot below.



This will run the animation/simulation process with “**todcode**” as input G-code text file. While the animation/simulation is running the following operations can be performed i.e. lines 4 to 9

3. Pressing **space key** on the keyboard will toggle **play/pause**
4. Pressing “+” (plus sign) and “-” (minus sign) keys will make animation **faster or slower**
5. **Left/Right** arrow keys will play animation **forward/backwards**
6. **Home/End** keys seek **begin/end** of animation
7. Pressing “0” key moves to **begin**, pressing “9” key moves to the **end**
8. **Right click** mouse, hold and move will give different **angle of views**
9. **Simultaneously right and left** click mouse and **hold, drag** back and forth to **zoom in and zoom out**

To change the **cross section width** (track width), **G-code text file**, screen size and animation time close current visualization window and repeat **stage 2** above and follow lines 10 to 14 below.



10. Command line argument "-t 30" will set the speed to play the animation. The time is based on the feed rate. Current animation time is set to 750 seconds based on the feed rate of the input G-code.

- `parser.add_option("-t", "--time", default=750,`

11. Command line argument "args=["todcode.txt"]" inputs or changes G-code text file. Current G-code file name is **green** "todcode"

- `args=["todcode.txt"]`

12. Command line argument "-w width" sets the cross section width. Current cross section width is set to 2.5.

- `parser.add_option("-w", "--width", default=0.3,`

13. Command line argument "-n nsides" sets the shape of the cross section. Current cross section shape is 4 (square). 0 to 2 will sweep a circular shape.

- `parser.add_option("-n", "--nsides", default=4,`

14. Command line argument "-f" toggles full-screen. Current setting is "False" to get a full screen change to "True"

- `parser.add_option("-f", "--full", "--full-screen", default=False,`

## APPENDIX C

### INSTRUCTIONS TO INSTALL AND RUN COLLISION DETECTION PROGRAM ON WINDOWS

#### INSTALL THE EXTENSIONS: NUMPY 2.7 AND PYOPENGL

1. Install **Python 2.7** if not already installed. Follow **step 1 in Appendix B** to install Python 2.7.
2. Follow link (<http://www.lfd.uci.edu/~gohlke/pythonlibs/>) to download and install (**numpy-MKL-1.6.1.-win32-py2.7.exe**). When the page is displayed scroll down the screen till you get to this portion of the screen shown below, click name in oval to install numpy.

**NumPy** is a fundamental package needed for scientific computing with Python.  
*Note:* these builds are not completely compatible with the official SciPy binaries.  
*Note:* the MKL builds are linked to Intel's high performance [Math Kernel Library](#).  
*Note:* you may not redistribute the MKL builds unless you own an appropriate license from Intel.

- [numpy-MKL-1.6.1.win-amd64-py2.6.exe](#) [8.7 MB] [Python 2.6] [64 bit] [Mar 03, 2012]
- [numpy-MKL-1.6.1.win-amd64-py2.7.exe](#) [8.7 MB] [Python 2.7] [64 bit] [Mar 03, 2012]
- [numpy-MKL-1.6.1.win-amd64-py3.1.exe](#) [8.7 MB] [Python 3.1] [64 bit] [Mar 03, 2012]
- [numpy-MKL-1.6.1.win-amd64-py3.2.exe](#) [8.7 MB] [Python 3.2] [64 bit] [Mar 03, 2012]
- [numpy-MKL-1.6.1.win32-py2.6.exe](#) [7.0 MB] [Python 2.6] [32 bit] [Mar 03, 2012]
- [numpy-MKL-1.6.1.win32-py2.7.exe](#) [7.0 MB] [Python 2.7] [32 bit] [Mar 03, 2012]
- [numpy-MKL-1.6.1.win32-py3.1.exe](#) [7.0 MB] [Python 3.1] [32 bit] [Mar 03, 2012]
- [numpy-MKL-1.6.1.win32-py3.2.exe](#) [7.0 MB] [Python 3.2] [32 bit] [Mar 03, 2012]

3. Follow link (<http://www.lfd.uci.edu/~gohlke/pythonlibs/>) to download and install (**PyOpenGL-3.0.1.win32-py2.7.exe**). When the page is displayed scroll down the screen till you get to this portion of the screen shown below, click name in oval to PyOpenGL:

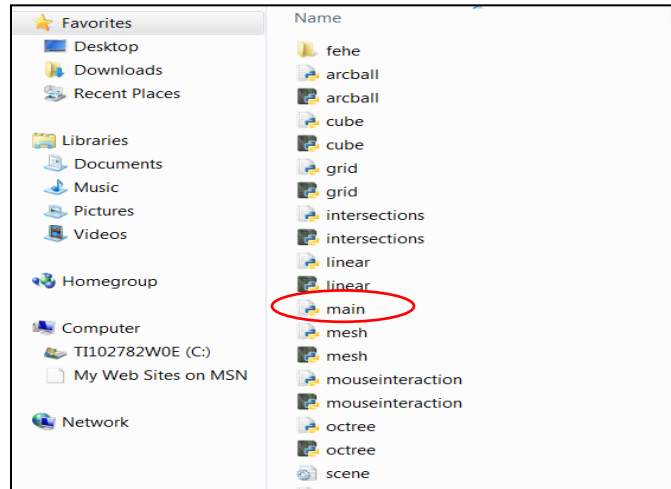
**PyOpenGL** provides bindings to OpenGL, GLUT, and GLE.

- [PyOpenGL-3.0.1.win-amd64-py2.5.exe](#) [1.4 MB] [Python 2.5] [64 bit] [Jan 19, 2011]
- [PyOpenGL-3.0.1.win-amd64-py2.6.exe](#) [1.4 MB] [Python 2.6] [64 bit] [Jan 19, 2011]
- [PyOpenGL-3.0.1.win-amd64-py2.7.exe](#) [1.4 MB] [Python 2.7] [64 bit] [Jan 19, 2011]
- [PyOpenGL-3.0.1.win32-py2.5.exe](#) [1.2 MB] [Python 2.5] [32 bit] [Jan 19, 2011]
- [PyOpenGL-3.0.1.win32-py2.6.exe](#) [1.3 MB] [Python 2.6] [32 bit] [Jan 19, 2011]
- [PyOpenGL-3.0.1.win32-py2.7.exe](#) [1.3 MB] [Python 2.7] [32 bit] [Jan 19, 2011]

This last stage completes the installation process to successfully run the collision detection program.

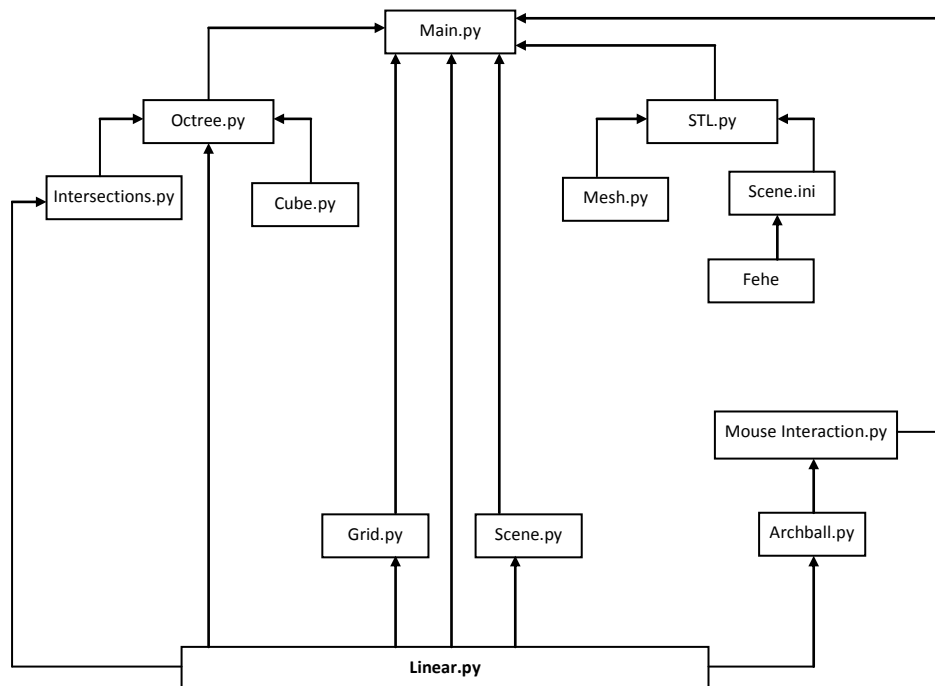
### **HOW TO RUN THE COLLISION DETECTION PROGRAM:**

1. Unzip folder “Octree-OBB” and save all the files into one folder. The files in the folder appear on screen like this below:

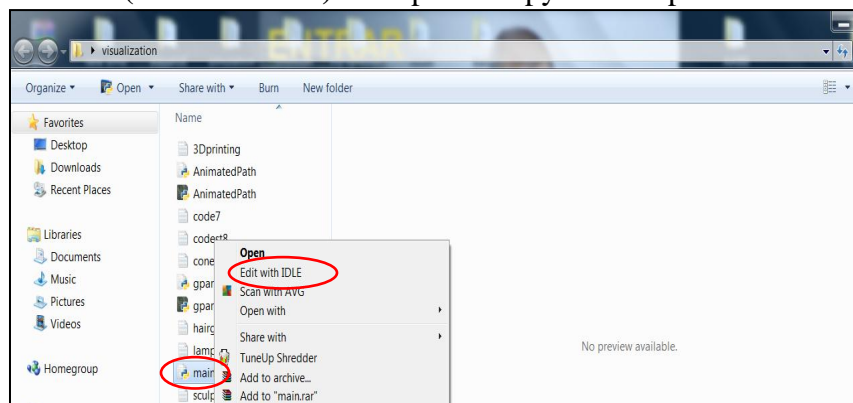


- Folder “fehe” contains list of ASCII CAD STL files
- File “linear.py” is a python script compiler for setting (computing) the translations and rotations of the CAD object in the scene.
- File “arcball.py” is a python script compiler that sets the scene dimension based on the size of the CAD and to allow mouse interactions and rotations. It can be downloaded from [http://nehe.gamedev.net/tutorial/arcball\\_rotation/19003/](http://nehe.gamedev.net/tutorial/arcball_rotation/19003/)
- File “grid.py” is a python script compiler that divides the scene into grids.
- File “Intersections.py” is a python script compiler that uses OBBs to perform the collision detection test. It can be downloaded from [http://www.gamasutra.com/view/feature/131790/simple\\_intersection\\_tests\\_for\\_games.php?page=5](http://www.gamasutra.com/view/feature/131790/simple_intersection_tests_for_games.php?page=5)
- File “Mouseinteractions.py” is a python script compiler used to perform zoom, pan and rotations operations.
- File “scene.py” is a python script compiler keep records of the models, and allows interactions with grids, octrees, translation and rotation of the CAD models.
- File “cube.py” is a python compiler that creates OBBs of the CAD models.
- File “mesh.py” is a python script compiler that represents the vertexes, points and faces of the CAD models.
- File “stl.py” is a python script that imports the 3D CAD file in stl formats.
- File “octree.py” is a python script that creates the octree of the CAD models. It can be downloaded from <http://www.xbdev.net/mathsof3d/octree/tutorial/index.php>

- File “**scene.ini**” is for changing the CAD models and their relative distances and depth of partitioning.
- File “**main**” is the main python script use to run the collision detection program.
- **Below is the Program Compilers Dependency**



- To run program execute “**main.py**”. Right click file “**main**” and then click (**Edit with Idle**) this opens the python script in the idle.



The first sixteen command lines of the python script in idle appears on the screen as shown below:

```

7% main.py - C:\Users\hamza\Desktop\OCTREE-OB\main.py
File Edit Format Run Options Windows Help
from OpenGL.GLUT import *
from OpenGL.GLU import *

import ConfigParser
from scene import Scene
from octree import Octree
from grid import Grid
from stl import loadSTL
from linear import *
from mouseinteraction import MouseInteraction

scene = None
action = None
click_pos = None

WINDOW_WIDTH = 640
WINDOW_HEIGHT = 480

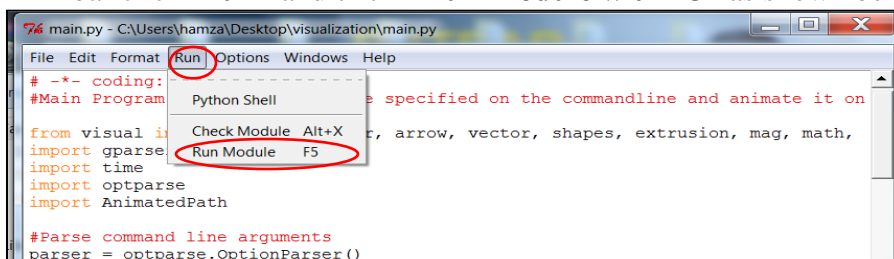
mouse_interaction = MouseInteraction(WINDOW_WIDTH, WINDOW_HEIGHT)

# this function gets called by GLUT when the scene needs to be redrawn, for exa
# - when the viewport is first created
# - after whe viewport changes (window resize)
# - when glutPostRedisplay is called

def DrawScene():
    # clear the viewport and the depth buffer (z-buffer)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    # set up the position of the camera

```

- Press key “F5” on the keyboard to run the simulation. Or from the task bar click “run” and click “Run Module with F5” as shown below.



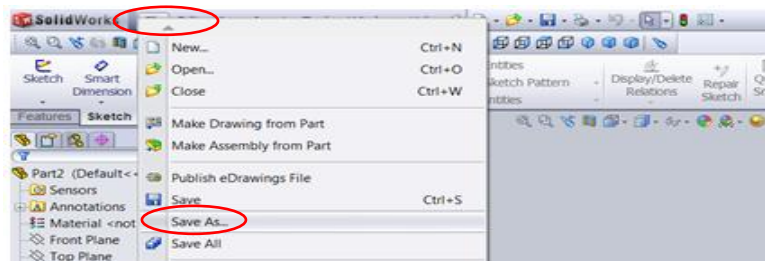
- While the animation/simulation is still active, the following operations can be performed i.e. lines 2 to 10
2. **F1** – will toggle **octree leaves**
  3. **F2** – will toggle **mesh lines**
  4. **F3** – will toggle **transparency**
  5. Left click the **mouse** and move to rotate the scene around (different angle of views).
  6. Press **CONTROL** and hold, left click and hold mouse, move around to pan the view.
  7. Press and hold down **SHIFT** key, drag mouse back and forth, to **zoom in** or **zoom out** of the scene.
  8. To **select** which **object to move**, press number **0** on keyboard to **translate** the first object while holding the second object **fixed**. Press number **1** on keyboard to reverse the process.
  9. To move the **object around**, use the **arrows**, **HOME** and **END** keys.
  10. To **rotate the object**, press **CONTROL** and use the **arrows**, **HOME** and **END** keys.



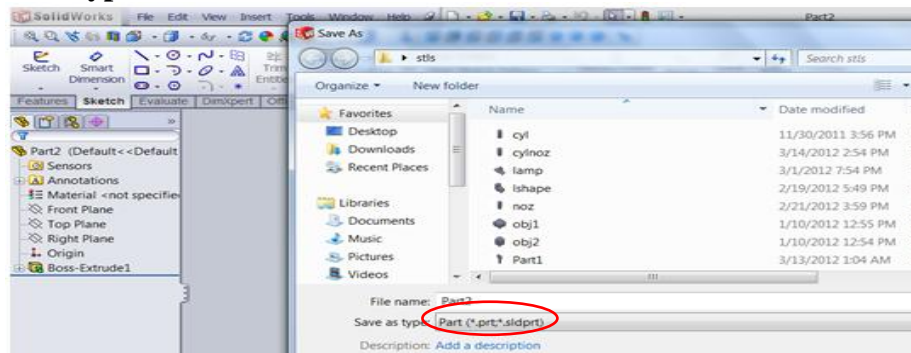
## APPENDIX D

### HOW TO INPUT OR CHANGE THE CAD MODELS AND THEIR RELATIVE POSITIONS.

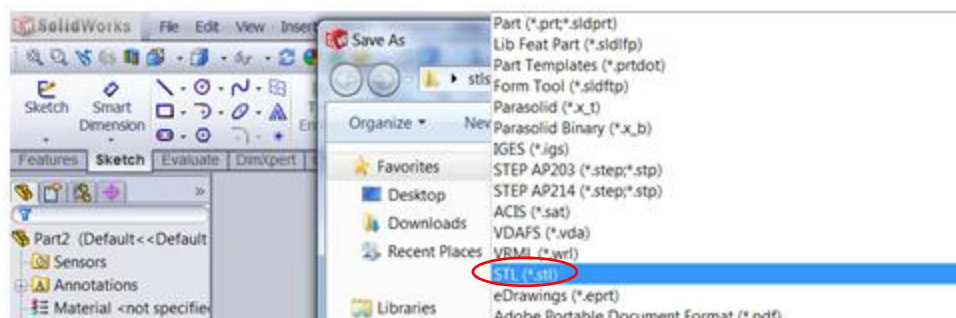
1. Use SolidWorks to Create CAD STL files. Create the CAD object, go to **file** from task bar and choose save as.



- Click on **Save As** a new window below will show up. Click on **save as type**

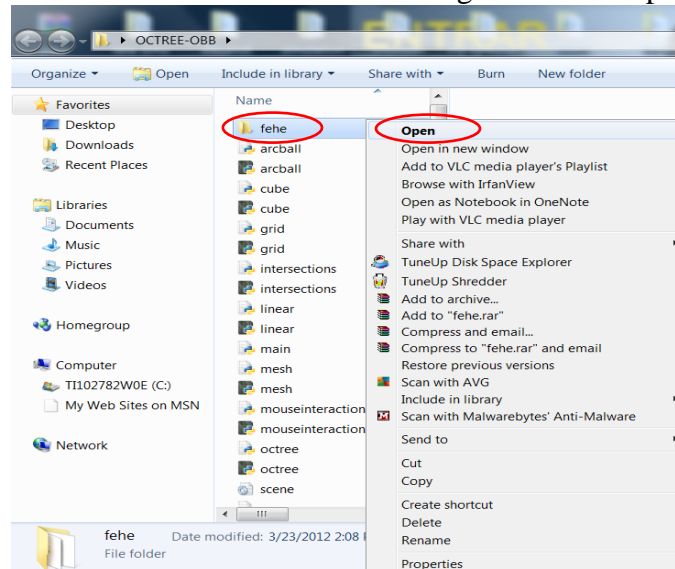


- When **save as type** is clicked, the window below will show up

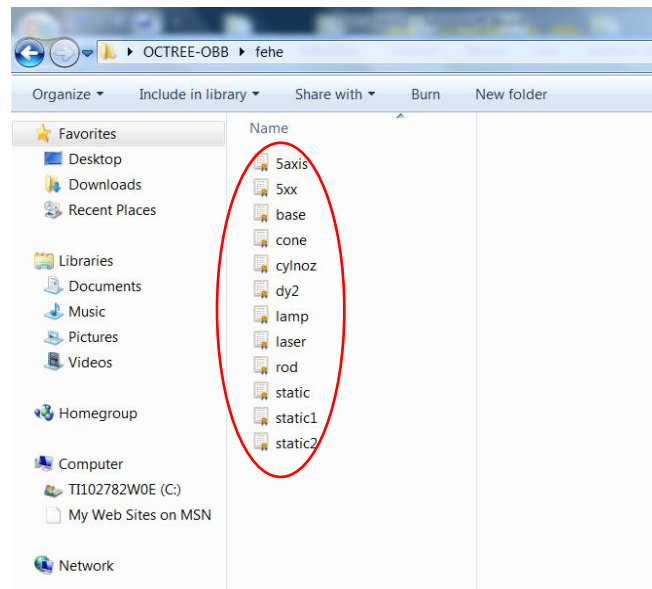


- Click on **STL (\*.stl)** to save the CAD object to a chosen folder or “fehe”.

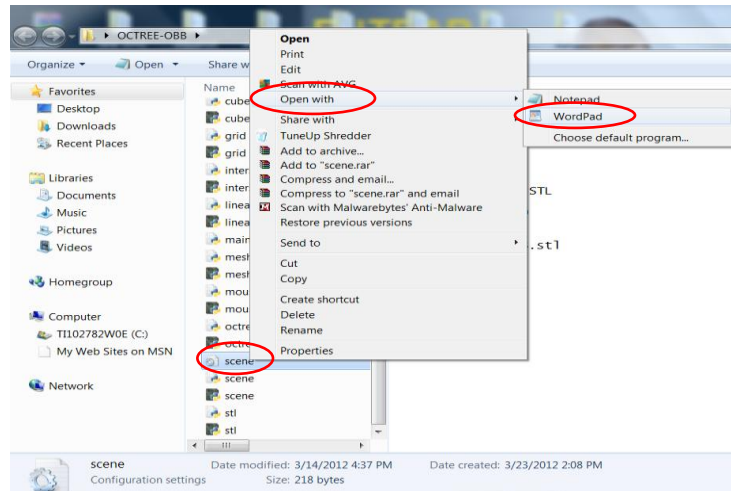
2. To **change** the **CAD** models, set the **depth** of partitioning and the **positions** of the CAD models in the scene follow lines **2** to **3** below.
3. The 3D STL CAD files are in folder “**fehe**” right click and open the folder



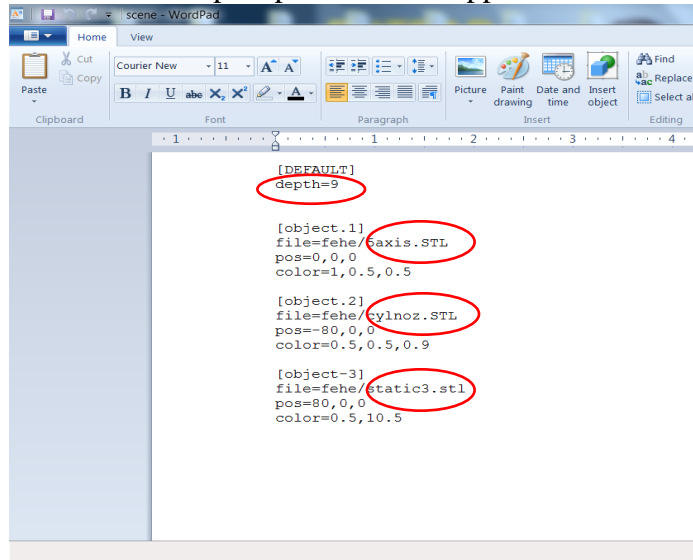
Below in the oval are the CAD files in the folder “**fehe**”



4. To **input or change** the 3D CAD models (STL ASCII format), depth of partitioning and positioning of the CAD models: open file “**scene.ini**” in WordPad and make the changes by typing the file name of the CAD model from folder “**fehe**”



- When “scene.ini” is open the WordPad window below will show up. In oval user can change CAD files by choosing from the folder “fehe”. “pos” is the relative position of the CAD models in the scene. “Color” will give the CAD model its color. “depth” will set the number of times to split a CAD model. Save the file after the changes are made. To run the animation repeat procedure in Appendix C.



## APPENDIX E

### G-CODE TEXT FILE

Below is a truncated sample G-code (todcode). “todcode” can be found in the zipped visualization program folder.

```
0      0      M64
0      12     G90
900    12     G4P20000
0      12     F750.0
900    12     G1X 8.4951 Y 0.135 Z 0.0
0      12     M65
900    12     G4P50
0      12     G1X 10.4263 Y 0.135 Z 0.0
900    12     G1X 12.6851 Y 1.4553 Z 0.0
0      12     G1X 6.8531 Y 1.385 Z 0.0
900    12     G1X 6.3717 Y 2.635 Z 0.0
0      12     G1X 15.5004 Y 2.635 Z 0.0
900    12     G1X 16.4491 Y 3.885 Z 0.0
0      12     G1X 6.4343 Y 3.885 Z 0.0
900    12     G1X 7.0835 Y 5.135 Z 0.0
0      12     G1X 16.9854 Y 5.135 Z 0.0
900    12     G1X 17.7818 Y 6.385 Z 0.0
0      12     G1X 11.098 Y 6.3696 Z 0.0
900    12     G1X 13.3519 Y 7.635 Z 0.0
0      12     G1X 18.6918 Y 7.635 Z 0.0
900    12     G1X 20.0562 Y 8.885 Z 0.0
0      12     G1X 14.5672 Y 8.885 Z 0.0
900    12     G1X 15.3469 Y 10.135 Z 0.0
0      12     G1X 23.5652 Y 10.135 Z 0.0
900    12     G1X 26.8813 Y 11.385 Z 0.0
0      12     G1X 15.8331 Y 11.385 Z 0.0
900    12     G1X 16.0977 Y 12.635 Z 0.0
0      12     G1X 27.7038 Y 12.635 Z 0.0
900    12     G1X 28.0823 Y 13.885 Z 0.0
0      12     G1X 16.3181 Y 13.885 Z 0.0
900    12     G1X 16.5385 Y 15.13
```

Where, F is the feed rate specifications in mm/min or in/min, G1 is linear motion mode, M64/M65 are on and off, respectively, of the optional M-code, G90 is absolute coordinate mode, G4 is a dwell cycle and X,Y,Z all refer to coordinate values.

## APPENDIX F

### VISUALIZATION PROGRAM MODULES

#### main.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#Main Program: Will load the file specified on the command line and animate it on the screen
from visual import display, color, arrow, vector, shapes, extrusion, mag, math, box
import gparser
import time
import optparse
import AnimatedPath

#Parse command line arguments
parser = optparse.OptionParser()
parser.add_option("-t", "--time", default=50,
                  action="store", type="float", dest="animation_duration",
                  help=u"How many seconds the animation should last", metavar="SECONDS")
parser.add_option("-n", "--nsides", default=4,
                  action="store", type="int", dest="extrusion_ngon",
                  help=u"How many facets on the extrusion", metavar="NSIDES")
parser.add_option("-w", "--width", default=2,
                  action="store", type="int", dest="extrusion_width",
                  help=u"Width of the extrusion", metavar="WIDTH")
parser.add_option("-f", "--full", "--full-screen", default=False,
                  action="store_true", dest="fullscreen",
                  help=u"Full screen mode")
(options, args) = parser.parse_args()
print " KEYBOARD FUNCTIONS"
print "--press space bar : to pause/play animation"
print "--press + and - : to speed/slow animation"
print "--press arrow keys: to play animation back and forth"
print "--press 0 and 9 : move to begining and end of animation"
print "--left and right click mouse and hold: drag to zoom in and out"
print "-- right click mouse and hold, move to get diffrent angle views"

#User can specify a single file to open. If none is specified, we open a default file
if len(args) == 0:
    args=["hairgel.txt"]
if len(args) != 1:
    parser.error("incorrect number of arguments, you should specify ONE G-code file")

#Initializes extrusion shape.
AnimatedPath.extrusionShapeWidth = options.extrusion_width
if options.extrusion_ngon >= 3:
    AnimatedPath.extrusionShape =
    shapes.rectangle(width=options.extrusion_width,height=options.extrusion_width,roundness=0)
else:
    AnimatedPath.extrusionShape = shapes.circle(radius=options.extrusion_width)

#Creates the scene object
scene=display(
    title="simulation",
```

```

width=650,height=400,
fullscreen=options.fullscreen,
autocenter=False,
autoscale=False,
background=color.black,
forward=vector(1,1,-1),
up=(0,0,1)
)

fileContents = gparser.parseFile(args[0]) #Parsed G-Code file
unrecognized_instrs = set() #Set if unrecognized intructions, to warn after the parsing is done
totalDistance = 0 #Total distance the printer will have to move. Used as time key during animation
on = True #Printer defaults to "On"
currentPos = vector(0,0,0) #Current position of the printer header
range_min = currentPos #Range of coordinates, used to center and zoom the scene properly
range_max = currentPos

animated_paths = [] #List of the AnimatedPaths printed
current_path_points = [] #List of tuples (time, point) of the current path being drawn
head_path = [ (0,currentPos) ] #List of tuples (time, point) where the head has passed

#"Executes" each instruction in the file
for instr in fileContents:
    instrType = instr[0] #Type of the instruction: G1, M64, etc
    instrArgs = instr[1] #Instruction parameters: X,Y,Z,etc

    #I'm not sure about instructions M101, M102, M103 and M108, but it seems to be working OK this way
    if False: #To make a regular indentation
        pass

    #Instructions to set the printing "ON"
    elif instrType=="M101" or instrType=="M108" or instrType=="M103" or instrType=="M65" :
        on=True
    #Instructions to set the printing "OFF"
    elif instrType=="M102" or instrType=="M64" :
        on=False
        #If there is pending set of points, add a new AnimatedPath for them
        if len(current_path_points):
            animated_paths.append( AnimatedPath.AnimatedPath(current_path_points) )
            current_path_points=[]

    #Instruction to move the printer header. The most import instruction!
    elif instrType=="G1":
        #new position is in the arguments
        pos=vector(instrArgs['X'],instrArgs['Y'],instrArgs['Z'])
        #Updates total distance travelled and scene ranges
        totalDistance += mag(pos-currentPos)
        range_min = vector(min(range_min[0], pos[0]), min(range_min[1], pos[1]), min(range_min[2], pos[2]))
        range_max = vector(max(range_max[0], pos[0]), max(range_max[1], pos[1]), max(range_max[2], pos[2]))

        #Add the new point to the header path animation
        head_path.append( (totalDistance, pos) )
        if on:
            #If on, add the new point to the current path being drawn
            current_path_points.append( (totalDistance, pos) )

```

```

    #Updates current position
    currentPos=pos

    #Instruction not recognised. Add to warning list
    else:
        unrecognized_instrs.add(instrType)

#If there is pending set of points after EOF, add a new AnimatedPath for them
if len(current_path_points):
    animated_paths.append( AnimatedPath.AnimatedPath(current_path_points) )
    current_path_points=[]

#Display "Finished" message, with some useful information
if len(unrecognized_instrs):
    print "Warn: G-Codes not recognized: "+ " ".join(unrecognized_instrs)
print "Parse completed. Segments:" + str(len(animated_paths)) + ", Length:" + str(totalDistance)

#sets the scene center/size
range_min = vector(
    range_min[0]-options.extrusion_width,
    range_min[1]-options.extrusion_width,
    range_min[2]-options.extrusion_width )
range_max = vector(
    range_max[0]+options.extrusion_width,
    range_max[1]+options.extrusion_width,
    range_max[2]+options.extrusion_width + 5*options.extrusion_width)
scene_radius = mag(range_max-range_min) + options.extrusion_width
scene.center = (range_min+range_max)/2
scene.range = scene_radius*vector(1,1,1)

#Printer header - A cone pointing to the current position of the printer header
printer_head_height=1*options.extrusion_width
printer_head = arrow(pos=(0,0,-options.extrusion_width), axis=(0,0,-printer_head_height),
shaftwidth=printer_head_height/3, color=color.red)

#deposition platform
box_padding=scene_radius/20
floor_center = vector( (range_min[0]+range_max[0])/2, (range_min[1]+range_max[1])/2, range_min[2])
floor_size = (range_max-range_min) + (2*box_padding, 2*box_padding, 0)
floor = box(pos=floor_center, length=floor_size[0], height=floor_size[1], width=options.extrusion_width/10, color=
color.blue )

#Draw a wired box around the scene

box_vertexes = [ ]
for z in ( range_min[2], range_max[2]+box_padding ):
    for y in ( range_min[1]-box_padding, range_max[1]+box_padding ):
        for x in ( range_min[0]-box_padding, range_max[0]+box_padding ):
            box_vertexes.append( (x,y,z) )

box_edges = [ #(0,1), (1,3), (3,2), (2,0),
#             (0,4), (1,5), (2,6), (3,7),
#             (4,5), (5,7), (7,6), (6,4)
#             ]
for x in box_edges:

```

```

    extrusion( pos=[ box_vertexes[x[0]], box_vertexes[x[1]] ], shape=shapes.ngon(np=4, radius=0.5), color=(.2,.5,1))#
actual deposition

#Runs the animation
lastNow = time.time()
animation_time = 0
speed = totalDistance / options.animation_duration
running=True
while True:
    now = time.time()
    delta_time = now -lastNow
    lastNow = now

    # Handle keyboard events
    while scene.kb.keys:
        s = scene.kb.getkey() # get keyboard info
        #print "Key typed: " + s + ""

        #Space: toggle running/not running
        if s == ' ':
            running = not running
            if running:
                start = time.time() - animation_time

        #Home/End/Numbers - Move to absolute position
        if s == 'home':
            animation_time = 0
        if s == 'end':
            animation_time = totalDistance
        if len(s)==1 and s >= "0" and s<="9":
            animation_time = int(s) / 9.0 * totalDistance

        #Right/Left - Rogle moving forward/backward
        if s == 'right':
            speed = abs(speed)
        if s == 'left':
            speed = -abs(speed)
        #+/- Faster/Slower
        if s == '+':
            speed = speed*1.3
        if s == '-':
            speed = speed/1.3

        #If running, move the animation one step
        if running:
            animation_time += delta_time * speed
        #animation_time is bound to the interval [0, totalDistance]
        animation_time = max(0, min(totalDistance, animation_time))

        #Updates printer header position and extruded paths accordingly to the current time
        printer_head.pos = AnimatedPath.interpolatePath(head_path, animation_time) - printer_head.axis +
(0,0,options.extrusion_width)
        for p in animated_paths:
            p.setTime(animation_time)

```



### AnimatedPath.py

```
# -*- coding: utf-8 -*-
from visual import extrusion, color, vector
import math

#Will be initialized on the main method
extrusionShape = None
extrusionShapeWidth = 0

#An animated path is a 3D object which will display incrementally based on the time, creating an animation
class AnimatedPath:

    #Creates the path from a list of tuples from parsed g-code lines
    def __init__(self, points, color=color.white):
        self.extrusion=extrusion(pos=[], shape=extrusionShape, color=color,visible=False)# actual deposition
        self.points = points
        self.firstTime = points[0][0]
        self.lastTime = points[len(points)-1][0]
        self.now = self.firstTime

    def setTime(self, time):
        #Don't do any modification if not needed
        if time == self.lastTime:
            return
        if time <= self.firstTime and self.now <= self.firstTime:
            return
        if time >= self.lastTime and self.now >= self.lastTime:
            return
        self.now = time

        newpos=[]
        prev=None
        for p in self.points:
            if p[0] < time:
                newpos.append(p[1])
            else: #The current segment is still being created, will interpolate this (last) point to create the animation
                if prev:
                    alpha = (time-prev[0]) / (p[0]-prev[0])
                    newpos.append(alpha*p[1] + (1-alpha)*prev[1] )
                break;
            prev = p

        i=1
        while i < len(newpos):
            prev = newpos[i-1]
            cur = newpos[i]

            if (cur-prev).mag < extrusionShapeWidth/10.0:
                newpos.pop(i)
                continue
            i=i+1

    #Post-processing step 2 - If we have U turnrs,i.e anngle >90 the extrusion library generates unpleasing glitches.
    #workaround.
```

```

i=1
while i < len(newpos)-1:
    #An angle is defined by 3 points: Current, previous and next
    p_prev = newpos[i-1]
    p_cur = newpos[i]
    p_next = newpos[i+1]

    #Gets length and direction of the segments prev<->current and current<->next
    dist1 = (p_cur - p_prev).mag
    dist2 = (p_next - p_cur).mag
    dir1 = (p_cur - p_prev).norm()
    dir2 = (p_next - p_cur).norm()

    #If the angle between the segments is too big (ang>90 == U-turn), we will break this corner in two
    ang = math.degrees( vector.diff_angle(dir1, dir2))
    if ang > 90:
        #Distance between the new endpoint to cur.
        #Should be approx extrusionShapeRadius to look good, and must be smaller than dist1 and dist2
        d = min(dist1/30, dist2/30, extrusionShapeWidth)

        #Uses linear interpolation to find the new endpoints
        alpha1 = d/dist1
        alpha2 = d/dist2
        p1 = alpha1*p_prev + (1-alpha1)*p_cur
        p2 = alpha2*p_next + (1-alpha2)*p_cur

        #Replaces the old endpoint with the new values
        newpos[i] = p1
        newpos.insert(i+1, p2)
        i=i+1
    i=i+1

if len(newpos) == 0:    # We can optimize a little if the list is empty
    self.extrusion.visible=False
else:                  # Updates the extrusion object with new values
    self.extrusion.pos=newpos
    self.extrusion.visible=True

def interpolatePath(points, time):
    prev=None
    for p in points:
        if p[0] > time:
            if prev:
                alpha = (time-prev[0]) / (p[0]-prev[0])
                return alpha*p[1] + (1-alpha)*prev[1]
            else:
                return p[1]
        prev = p
    return prev[1]

```

### gparse.py

```
# -*- coding: utf-8 -*-
import re

#This will parse a G-Codes file into a list of instructions.
#The first element in the G-Code identifies the instruction. This is kept as the "key" of each element
#The remaining elements are stored in a dictionary
#Invalid lines are ignored
#
# e.g.
# ;M113 S1.0
# M108 R3.146
# (<surroundingLoop>)
# (<loop> outer )
# G1 X28.63 Y28.64 Z0.15 F1080.0
# G1 X28.63 Y-28.64 Z0.15 F1080.0
# G1 X-28.41 Y-28.64 Z0.15 F1080.0
# M108 R35.0
# M102
# G1 X-28.63 Y-28.64 Z0.
#
# Becomes
# [
# ('M108', {'R': 3.146}),
# ('G1', {'Y': 28.64, 'X': 28.63, 'Z': 0.15, 'F': 1080.0}),
# ('G1', {'Y': -28.64, 'X': 28.63, 'Z': 0.15, 'F': 1080.0}),
# ('G1', {'Y': -28.64, 'X': -28.41, 'Z': 0.15, 'F': 1080.0}),
# ('M108', {'R': 35.0}),
# ('M102', {}),
# ('G1', {'Y': -28.64, 'X': -28.63, 'Z': 0.0})
# ]
def parse(infile):
    linenum=0

    letter_re="[A-Z]" #Regular expression to match the letter
    double_re="[-+]?[d*\.]?[d+]" #Regular expression to match floating point numbers
    space_re="\s*" #Regular expression to match spaces, if present

    #Regular expression to match each pair Letter+Nubmer in the G_codes.
    #May not be the best way, but seems to be working
    regexp_element=re.compile("(" + letter_re + ")" + space_re + "(" + double_re + ")")

    #Regular expression to check if line is valid.
    #It will be valid if it starts with a capital letter (GOOD) or a number (A problema with the TOD sample?)
    regexp_validLine=re.compile(space_re + "(" + letter_re + "|" + double_re + ")")
    ret=[]

    while True:
        linenum=linenum+1
        line = infile.readline()
        if not line:
            break
        if not regexp_validLine.match(line):
            #print "Ignoring " + line
            continue
```

```

elements = regexp_element.findall(line)
if len(elements) == 0:
    continue
firstElement=None
values={ }
for e in elements:
    if firstElement is None:
        firstElement = e[0]+e[1]
    else:
        values[e[0]] = float(e[1])
ret.append( (firstElement, values) )
#print str(firstElement) + " - " + str(values)
return ret

#Same as Parse, but receives a file path instead of a file object
def parseFile(path):
    infile = open(path, "r")
    ret = parse(infile)
    infile.close()
    return ret

if __name__ == "__main__":
    f = open("objects/cone.txt", "r")
    print parseFile("objects/cone.txt")

```

## APPENDIX G

### COLLISION DETECTION PROGRAM MODULES

#### main.py

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

import ConfigParser
from scene import Scene
from octree import Octree
from grid import Grid
from stl import loadSTL
from linear import *

from mouseinteraction import MouseInteraction

scene = None
action = None
click_pos = None

WINDOW_WIDTH = 1500
WINDOW_HEIGHT = 1000

mouse_interaction = MouseInteraction(WINDOW_WIDTH, WINDOW_HEIGHT)

# this function gets called by GLUT when the scene needs to be redrawn, for example:
# - when the viewport is first created
# - after the viewport changes (window resize)
# - when glutPostRedisplay is called
print "--KEYBOARD FUNCTIONS--"
print "--left click mouse and hold, move to get diff angle of views"
print "--Press 0 key to activate first object to translate, press 1 will activate second object"
print "--use arrows,HOME and END keys to translate the selected object around"
print "--Press CTRL key and hold : use arrows, HOME and END keys to rotate object"
print "--Press and hold SHIFT key: left click and hold mouse, drag mouse back and forth to Zoom in and out"
print "--press and hold CTRL key: left click and mouse, move mouse around to pan scene"
print "--Press F1 to show only octree boxes"
print "--Press F2 to show only STL mesh lines"
print "--press F3 will make the CAD models transparent"

def DrawScene():
    # clear the viewport and the depth buffer (z-buffer)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    # set up the position of the camera
    glLoadIdentity();
```

```

glTranslatef(0,0,-200.0)

mouse_interaction.applyTransform()
scene.draw()

# tell GLUT to swap the front and back buffers
try:
    # sometimes this fails (bug in pyopengl?)
    # if that happens (program seems to freeze), try resizing the viewport
    glutSwapBuffers();
except:
    pass

def InitOpenGL(Width, Height):
    glShadeModel(GL_FLAT)
    # color we want to use when clearing the viewport
    glClearColor(0.0, 0.0, 0.0, 0.0)

    # enable the z-buffer (so objects get occluded correctly)
    glEnable(GL_DEPTH_TEST)

    # enable lights in general, and in particular light #0
    glEnable (GL_LIGHT0)
    glEnable (GL_LIGHTING)

    # tell OpenGL it should use the color we assign to the object as the diffuse component
    # (makes specifying the color of the object easier)
    glEnable (GL_COLOR_MATERIAL)

    # make sure normals are always normalized, even if we change the scale of things
    glEnable (GL_NORMALIZE);

# this function gets called by GLUT when the viewport is first created and when
# the viewport is resized.
def onResize(w, h):
    if h <= 2: h = 2
    if w <= 2: w = 2

    glViewport(0, 0, w, h)

    # select the projection matrix (to set up the perspective projection)
    glMatrixMode(GL_PROJECTION)

    glLoadIdentity()
    # view frustum angle set to 45 degrees, viewport aspect ratio, near plane, far plane
    gluPerspective(45.0, float(w)/float(h), 1, 1000.0)

    # select the modelview matrix now
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();

    mouse_interaction.vpResize(w,h)
    return

# this function gets called by GLUT when a key is pressed

```

```

def onKeyPressed(key, x, y):
    global scene

    if key == chr(27): # 27 is the ascii value of the ESC key
        sys.exit(0)

    if key >= '0' and key <= '9':
        scene.selectObject(ord(key) - ord('0'))

def onSpecialKeyPressed(key, x, y):

    if key in
(GLUT_KEY_LEFT, GLUT_KEY_RIGHT, GLUT_KEY_UP, GLUT_KEY_DOWN, GLUT_KEY_END, GLUT_KEY_
HOME):

        scene.storeTransform()

        if glutGetModifiers() & GLUT_ACTIVE_CTRL:
            # pressing control + arrow makes the object rotate
            if key == GLUT_KEY_LEFT: scene.rotate(Vector3f(-1,0,0),10)
            elif key == GLUT_KEY_RIGHT: scene.rotate(Vector3f(1,0,0),10)
            elif key == GLUT_KEY_UP: scene.rotate(Vector3f(0,1,0),10)
            elif key == GLUT_KEY_DOWN: scene.rotate(Vector3f(0,-1,0),10)
            elif key == GLUT_KEY_END: scene.rotate(Vector3f(0,0,-1),10)
            elif key == GLUT_KEY_HOME: scene.rotate(Vector3f(0,0,1),10)
        else:
            # pressing control + arrow makes the object move
            if key == GLUT_KEY_LEFT: scene.translate(Vector3f(-1,0,0))
            elif key == GLUT_KEY_RIGHT: scene.translate(Vector3f(1,0,0))
            elif key == GLUT_KEY_UP: scene.translate(Vector3f(0,1,0))
            elif key == GLUT_KEY_DOWN: scene.translate(Vector3f(0,-1,0))
            elif key == GLUT_KEY_END: scene.translate(Vector3f(0,0,-1))
            elif key == GLUT_KEY_HOME: scene.translate(Vector3f(0,0,1))

        if scene.checkCollisions():
            scene.restoreTransform()

    elif key == GLUT_KEY_F1:
        scene.toggleHelperGroup("octree")
    elif key == GLUT_KEY_F2:
        scene.toggleFaceEdges()
    elif key == GLUT_KEY_F3:
        scene.toggleTranslucent()

    glutPostRedisplay()

def onClick(button, state, x, y):
    if state == GLUT_DOWN:
        if glutGetModifiers() & GLUT_ACTIVE_CTRL:
            action = mouse_interaction.PAN_ACTION
        elif glutGetModifiers() & GLUT_ACTIVE_SHIFT:
            action = mouse_interaction.ZOOM_ACTION
        else:
            action = mouse_interaction.ROTATE_ACTION
        mouse_interaction.click(x,y, action)
    elif state == GLUT_UP:

```

```

        mouse_interaction.release()
def onDrag(x, y):
    mouse_interaction.drag(x,y)
    glutPostRedisplay();

def loadScene():
    # split a vector represented as a string into float components
    def asVector(txt):
        return map(lambda a:float(a.strip()), txt.split(','))

    scene = Scene()

    config = ConfigParser.SafeConfigParser()

    config.read("scene.ini")

    for s_name in config.sections():
        print s_name
        if s_name.lower().startswith("object."):

            objname = s_name[7:]
            filename = config.get(s_name,'file')

            if filename is not None:
                print "Loading file %s"%filename

                if config.has_option(s_name,'pos'):
                    position = asVector(config.get(s_name,'pos'))
                else:
                    position = Vector3f()

                if config.has_option(s_name,'color'):
                    color = asVector(config.get(s_name, 'color'))
                else:
                    color = None

                mesh = loadSTL(filename)

                if config.has_option(s_name,'depth'):
                    depth = int(config.get(s_name,'depth'))
                else:
                    depth = 9

                mesh.setPosition(position)
                mesh.setName(objname or "object")
                mesh.setColor(color or (0.5,0.5,0.5))

                scene.addInteractive(mesh)

                oct = Octree(mesh, max_depth = depth)

                scene.addHelper(oct,"octree")

    scene.addHelper(Grid(200, 20),"grid")

    return scene

```



```

def main():
    global scene

    # boilerplate GLUT initialization code
    glutInit(sys.argv)

    # tell glut we need a viewport with:
    # - truecolor capabilities with alpha channel (transparency)
    # - double buffer (to avoid flicker when drawing)
    # - depth buffer (to have correct occlusion of objects)
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH)
    glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT)

    # open the window
    window = glutCreateWindow("")

    # register some callbacks that will call GLUT when something happens (key pressed, mouse move, etc)
    glutReshapeFunc(onResize)
    glutKeyboardFunc(onKeyPressed)
    glutSpecialFunc(onSpecialKeyPressed)

    glutDisplayFunc(DrawScene)

    # load everything
    scene = loadScene()

    #glutIdleFunc(Draw)
    glutMouseFunc (onClick)
    glutMotionFunc (onDrag)

    # initialize openGL
    InitOpenGL(WINDOW_WIDTH, WINDOW_HEIGHT)

    glutMainLoop()

if __name__=="__main__":
    main()

```

### **intersections.py**

```

import numpy as N
from linear import *

# returns (t,u,v) where
# t - distance along the ray where intersects the triangle
# u,v - barycentric coordinates on the triangle

def lineno():
    import inspect
    """Returns the current line number in our program."""
    return inspect.currentframe().f_back.f_lineno

def rayTriangleIntersection(ray_origin, ray_direction, triverts):

```

```

e1 = triverts[1]-triverts[0]
e2 = triverts[2]-triverts[0]

pvec = N.cross(ray_direction, e2)
det = N.dot(e1, pvec)

if det > -EPSILON and det < EPSILON:
    return None

inv_det = 1.0 / det

tvec = ray_origin - triverts[0]

u = N.dot(tvec,pvec) * inv_det
if u < 0.0 or u > 1.0:
    return None

qvec = N.cross(tvec, e1)

v = N.dot(ray_direction, qvec) * inv_det
if v < 0.0 or (u+v) > 1.0:
    return None

t = N.dot(e2, qvec) * inv_det

return t,u,v

# The following code is based on code by Tomas Akenine-Moller
# http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/

def planeBoxOverlap(normal, vert, maxbox):
    vmin = Vector3f()
    vmax = Vector3f()

    for coord in (0,1,2):
        v = vert[coord]
        if normal[coord] > 0.0:
            vmin[coord] = -maxbox[coord] - v
            vmax[coord] = maxbox[coord] - v
        else:
            vmin[coord] = maxbox[coord] - v
            vmax[coord] = -maxbox[coord] - v

    if N.dot(normal, vmin) > 0.0:
        return False

    if N.dot(normal, vmax) >= 0.0:
        return True

    return False

def triangleBoxOverlap(box_center, box_half_size, triverts):
    """
    Use separating axis theorem to test overlap between triangle and box
    need to test for overlap in these directions:
    1) the {x,y,z}-directions (actually, since we use the AABB of the triangle

```

```

    we do not even need to test these)
2) normal of the triangle
3) crossproduct(edge from tri, {x,y,z}-directin)
    this gives 3x3=9 more tests
"""

# test the triangle against a box centered at the origin
v0 = triverts[0] - box_center
v1 = triverts[1] - box_center
v2 = triverts[2] - box_center

#===== X-tests =====

def axisTestX01(a, b, fa, fb):
    p0 = a * v0[1] - b*v0[2]
    p2 = a * v2[1] - b*v2[2]

    if p0 < p2:
        min_val = p0
        max_val = p2
    else:
        min_val = p2
        max_val = p0

    rad = fa * box_half_size[1] + fb * box_half_size[2]

    return min_val > rad or max_val < -rad

def axisTestX2(a, b, fa, fb):
    p0 = a * v0[1] - b*v0[2]
    p1 = a * v1[1] - b*v1[2]

    if p0 < p1:
        min_val = p0
        max_val = p1
    else:
        min_val = p1
        max_val = p0

    rad = fa * box_half_size[1] + fb * box_half_size[2]

    return min_val > rad or max_val < -rad

#===== Y-tests =====

def axisTestY02(a, b, fa, fb):
    p0 = -a * v0[0] + b*v0[2]
    p2 = -a * v2[0] + b*v2[2]

    if p0 < p2:
        min_val = p0
        max_val = p2
    else:
        min_val = p2
        max_val = p0

    rad = fa * box_half_size[0] + fb * box_half_size[2]

```

```

        return min_val > rad or max_val < -rad

def axisTestY1(a, b, fa, fb):
    p0 = -a * v0[0] + b*v0[2]
    p1 = -a * v1[0] + b*v1[2]

    if p0 < p1:
        min_val = p0
        max_val = p1
    else:
        min_val = p1
        max_val = p0

    rad = fa * box_half_size[0] + fb * box_half_size[2]

    return min_val > rad or max_val < -rad

#===== Z-tests =====

def axisTestZ12(a, b, fa, fb):
    p1 = a * v1[0] - b*v1[1]
    p2 = a * v2[0] - b*v2[1]

    if p2 < p1:
        min_val = p2
        max_val = p1
    else:
        min_val = p1
        max_val = p
    rad = fa * box_half_size[0] + fb * box_half_size[1]

    return min_val > rad or max_val < -rad

def axisTestZ0(a, b, fa, fb):
    p0 = a * v0[0] - b*v0[1]
    p1 = a * v1[0] - b*v1[1]

    if p0 < p1:
        min_val = p0
        max_val = p1
    else:
        min_val = p1
        max_val = p0

    rad = fa * box_half_size[0] + fb * box_half_size[1]

    return min_val > rad or max_val < -rad

# edges

e0 = v1-v0
e1 = v2-v1
e2 = v0-v2

# TEST 1
fex = abs(e0[0])

```

```

fey = abs(e0[1])
fez = abs(e0[2])

if (axisTestX01(e0[2], e0[1], fez, fey) or
    axisTestY02(e0[2], e0[0], fez, fex) or
    axisTestZ12(e0[1], e0[0], fey, fex)):
    return False

fex = abs(e1[0])
fey = abs(e1[1])
fez = abs(e1[2])

if (axisTestX01(e1[2], e1[1], fez, fey) or
    axisTestY02(e1[2], e1[0], fez, fex) or
    axisTestZ0(e1[1], e1[0], fey, fex)):
    return False

fex = abs(e2[0])
fey = abs(e2[1])
fez = abs(e2[2])

if (axisTestX2(e2[2], e2[1], fez, fey) or
    axisTestY1(e2[2], e2[0], fez, fex) or
    axisTestZ12(e2[1], e2[0], fey, fex)):
    return False

# TEST 2
# first test overlap in the {x,y,z}-directions
# find min, max of the triangle each direction, and test for overlap in
# that direction -- this is equivalent to testing a minimal AABB around
# the triangle against the AABB

# test in X
if (min(v0[0], v1[0], v2[0]) > box_half_size[0] or
    max(v0[0], v1[0], v2[0]) < -box_half_size[0]):
    return False

# test in Y
if (min(v0[1], v1[1], v2[1]) > box_half_size[1] or
    max(v0[1], v1[1], v2[1]) < -box_half_size[1]):
    return False

# test in Z
if (min(v0[2], v1[2], v2[2]) > box_half_size[2] or
    max(v0[2], v1[2], v2[2]) < -box_half_size[2]):
    return False

# TEST 3
# test if the box intersects the plane of the triangle
# compute plane equation of triangle: normal*x+d=0

normal = N.cross(e0,e1)

if not planeBoxOverlap(normal,v0,box_half_size):
    return False

```

```

        return True # box and triangle overlap
# The following is code based on code by Miguel Gomez
# http://www.gamasutra.com/view/feature/131790/simple_intersection_tests_for_games.php?page=5

# box_frame_1, box_frame_2 must be of type numpy.matrix

def boxBoxOverlap(box_center_1, box_half_size_1, box_frame_1, box_center_2, box_half_size_2, box_frame_2,
R=None):
    if R is None:
        R = box_frame_1 * box_frame_2.T

    v = (box_center_2 - box_center_1)
    T = N.dot(box_frame_1, v.T) # T is column vector

    three = (0,1,2)

    a = box_half_size_1
    b = box_half_size_2

    Rabs = N.abs(R)

    for i in three:
        rb = N.dot(b, Rabs[i].T)
        t = abs(T[i])
        if t > a[i] + rb:
            return False

    for i in three:
        ra = N.dot(a, Rabs[:,i])
        t = abs(N.dot(R[:,i].T,T))
        if t > box_half_size_2[i] + ra:
            return False

    # L = A0 x B0

    ra = a[1]*abs(R[2,0]) + a[2]*abs(R[1,0])
    rb = b[1]*abs(R[0,2]) + b[2]*abs(R[0,1])

    t = abs(T[2]*R[1,0] - T[1]*R[2,0])
    if t > ra+rb:
        return False
    # L = A0 x B1

    ra = a[1]*abs(R[2,1]) + a[2]*abs(R[1,1])
    rb = b[0]*abs(R[0,2]) + b[2]*abs(R[0,0])

    t = abs(T[2]*R[1,1] - T[1]*R[2,1])
    if t > ra+rb:
        return False

    # L = A0 x B2

    ra = a[1]*abs(R[2,2]) + a[2]*abs(R[1,2])
    rb = b[0]*abs(R[0,1]) + b[1]*abs(R[0,0])

    t = abs(T[2]*R[1,2] - T[1]*R[2,2])

```

```

if t > ra+rb:
    return False

# L = A1 x B0

ra = a[0]*abs(R[2,0]) + a[2]*abs(R[0,0])
rb = b[1]*abs(R[1,2]) + b[2]*abs(R[1,1])

t = abs(T[0]*R[2,0] - T[2]*R[0,0])
if t > ra+rb:
    return False

# L = A1 x B1

ra = a[0]*abs(R[2,1]) + a[2]*abs(R[0,1])
rb = b[0]*abs(R[1,2]) + b[2]*abs(R[1,0])

t = abs(T[0]*R[2,1] - T[2]*R[0,1])
if t > ra+rb:
    return False

# L = A1 x B2

ra = a[0]*abs(R[2,2]) + a[2]*abs(R[0,2])
rb = b[0]*abs(R[1,1]) + b[1]*abs(R[1,0])

t = abs(T[0]*R[2,2] - T[2]*R[0,2])
if t > ra+rb:
    return False

# L = A2 x B0

ra = a[0]*abs(R[1,0]) + a[1]*abs(R[0,0])
rb = b[1]*abs(R[2,2]) + b[2]*abs(R[2,1])

t = abs(T[1]*R[0,0] - T[0]*R[1,0])
if t > ra+rb:
    return False

# L = A2 x B1

ra = a[0]*abs(R[1,1]) + a[1]*abs(R[0,1])
rb = b[0]*abs(R[2,2]) + b[2]*abs(R[2,0])

t = abs(T[1]*R[0,1] - T[0]*R[1,1])
if t > ra+rb:
    return False

# L = A2 x B2

ra = a[0]*abs(R[1,2]) + a[1]*abs(R[0,2])
rb = b[0]*abs(R[2,1]) + b[1]*abs(R[2,0])

t = abs(T[1]*R[0,2] - T[0]*R[1,2])
if t > ra+rb:
    return False

```

```
return True
```

### **octree.py**

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
```

```
from linear import *
from cube import drawCube
```

```
from intersections import boxBoxOverlap
" ideas from http://www.xbdev.net/math3d/octree/tutorial/index.php"
```

```
class Octree:
```

```
    class OctreeLeaf:
```

```
        def __init__(self, box):
            self.center = (box[0] + box[1]) * 0.5
            self.size_2 = (box[1] - box[0]) * 0.5
            # this is so it can
            self.is_leaf = True
            # this is so it can get inserted in the to_test_queue (in the Octree.intersects method)
            self.cells = [self]
```

```
    class OctreeNode:
```

```
        def largestAxis(self):
            size = self.bounds[1] - self.bounds[0]
```

```
            axis = 0
            d = size[0]
```

```
            if size[1] > d:
                axis = 1
                d = size[1]
```

```
            if size[2] > d:
                axis = 2
                d = size[2]
```

```
            return axis
```

```
        def KDboxIter(self):
            axis = self.largestAxis()
            return self.splitBox(axis)
```

```
        def intersects(self, node, c1, t1, c2, t2, R):
            return boxBoxOverlap(self.center*t1 + c1, self.size_2, t1, node.center*t2 + c2,
```

```
node.size_2, t2, R)
```

```
        # split the bounds of this OctreeNode into 8 boxes
```

```
        def octreeBoxIter(self):
            size_2 = self.size_2
            pos = Vector3f()
```



```

pos[0] = self.bounds[0][0]
for x in (0,1):
    pos[1] = self.bounds[0][1]
    for y in (0,1):
        # unroll the z loop
        pos[2] = self.bounds[0][2]
        yield (pos.copy(), pos+size_2)
        pos[2] += size_2[2]
        yield (pos.copy(), pos+size_2)
        pos[1] += size_2[1]
    pos[0] += size_2[0]

# list of boxes contained in a given half (with respect to a given axis).
# for example, for axis 0 (X), boxes 0,1,2 and 3 lie on the "left" half, while
# boxes 4,5,6 and 7 lie on the "right" half
test_box_indices=[
    ((0,2,4,6),2),
    ((1,3,5,7),2),
    ((0,1,4,5),1),
    ((2,3,6,7),1),
    ((0,1,2,3),0),
    ((4,5,6,7),0),
]

def splitBox(self, axis):
    size = self.bounds[1] - self.bounds[0]
    d2 = size[axis] * 0.5
    pos = self.bounds[0].copy()

    if axis == 0:
        corner = (d2, size[1], size[2])
        boxes = [(pos.copy(), pos + corner), None]
        pos[0] += d2
        boxes[1] = (pos.copy(), pos + corner)

    elif axis == 1:
        corner = (size[0], d2, size[2])
        boxes = [(pos.copy(), pos + corner), None]
        pos[1] += d2
        boxes[1] = (pos.copy(), pos + corner)

    elif axis == 2:
        corner = (size[0], size[1], d2)
        boxes = [(pos.copy(), pos + corner), None]
        pos[2] += d2
        boxes[1] = (pos.copy(), pos + corner)

    return boxes

def __init__(self, mesh, bounds, max_depth, max_error):
    # check if all the boxes listed in box_indices taken from the occupancy list
    # are either empty (code returned by mesh.intersectsAABB is 0) or
    # full (code == 2).
    def checkBoxesStatus(occupancy, box_indices):
        empty_test_passes = True
        full_test_passes = True

```

```

        for box_num in box_indices:
            if occupancy[box_num] != 0: empty_test_passes = False
            if occupancy[box_num] != 2: full_test_passes = False
        return empty_test_passes, full_test_passes

    self.is_leaf = False
    self.bounds = N.array(bounds,"f")

    size = bounds[1] - bounds[0]
    self.size_2 = size * 0.5
    self.center = (bounds[0] + bounds[1])*0.5

    occupancy = []
    boxes = []

    for box in self.octreeBoxIter():
        occupancy.append( mesh.intersectsAABB(box) )
        boxes.append(box)

    if max_depth <= 0 or max(size) < max_error:
        # if we're at the max depth, then just store true (if contains any portion of the
        mesh)

        # or false at the cells
        self.cells = [ oc != 0 for oc in occupancy ]
        self.boxes = boxes
    else:
        # otherwise, pick a good axis to split the cell in two. If possible, choose
        # an axis that leaves one half either completely full or completely empty
        # if such axis doesn't exist, then pick the largest one
        split_axis = None
        for test, axis in self.test_box_indices:
            test)

            empty_test_passes, full_test_passes = checkBoxesStatus(occupancy,

            if empty_test_passes or full_test_passes:
                split_axis = axis
                break

        if split_axis is None:
            # no half is either full empty or completely full
            split_axis = self.largestAxis()
            self.boxes = self.splitBox(split_axis)
            self.cells = [
                Octree.OctreeNode(mesh, self.boxes[0],
                Octree.OctreeNode(mesh, self.boxes[1],
                ]
        else:
            # one (or both) of the halves is completely empty or completely full

            # complementary test (the other boxes not in test)
            ctest = [x for x in range(8) if x not in test]

            passed_test_boxes = [box for box_num,box in enumerate(boxes) if
            box_num in test]

```

```

other_test_boxes = [box for box_num,box in enumerate(boxes) if
box_num in ctest]

# the min/max of the first/last box in "test" and "ctest" are actually
the corners of the
# combined boxes in each group.
self.boxes = [(passed_test_boxes[0][0],passed_test_boxes[-1][1] ),
(other_test_boxes[0][0],other_test_boxes[-1][1] )]

# full_test_passes and empty_test_passes refer to the contents of
self.boxes[0],
# that is to the boxes in passed_test_boxes
self.cells = [full_test_passes, None]

# now check the occupancy of boxes -not- in test (those in ctest)
empty_test_passes, full_test_passes = checkBoxesStatus(occupancy,
ctest)

if empty_test_passes or full_test_passes:
    self.cells[1] = full_test_passes
else:
    self.cells[1] = Octree.OctreeNode(mesh, self.boxes[1],
max_depth-1, max_error)

def draw(self, level=0):
    bounds = self.bounds
    size_2 = self.size_2
    cells = self.cells
    i = 0

    for cell, box in zip(self.cells, self.boxes):
        if cell is False:
            continue
        if cell is True:
            glColor4f(0.0,1.0,0.0,0.5)
            drawCube(box[0], box[1])
        else:
            cell.draw(level+1)

    glColor4f(1.0,1.0,1.0,0.1)
    drawCube(self.bounds[0], self.bounds[1])

def __init__(self, mesh, max_depth = 9, max_error = 0.1):
    print "Building octree (max depth=%s)"%max_depth

    self.mesh = mesh
    mesh.octree = self

    bounds = (mesh.bounds[0], mesh.bounds[1])

    self.root = Octree.OctreeNode(mesh, bounds, max_depth, max_error)

```

```

def intersects(self, octree):
    # translation part
    c1 = self.mesh.transform[3,0:3]
    c2 = octree.mesh.transform[3,0:3]

    # rotation part
    t1 = self.mesh.transform[0:3,0:3]
    t2 = octree.mesh.transform[0:3,0:3]

    # mapping from one orientation to the other
    R = t1 * t2.T

    if not boxBoxOverlap(self.root.center * t1 + c1, self.root.size_2, t1,
                        octree.root.center*t2 + c2, octree.root.size_2, t2,
R):

        return False

    #inter = self.root.intersects(octree.root, c1,t1,c2,t2,R)
    #if inter == False:
    #    return False

    to_test_queue = [ (self.root, octree.root) ]

    while len(to_test_queue)>0:
        g1,g2 = to_test_queue.pop(0)

        # we put this cycle here so we don't do the tests and instantiation of OctreeLeaf
        # many times inside the cycle of i,cell
        cell2_list = []
        for j,cell2 in enumerate(g2.cells):
            if cell2 == False: continue
            if cell2 == True:
                cell2 = self.OctreeLeaf(g2.bboxes[j])
                cell2_list.append(cell2)

        # check all the cells in the first group against all the cells in the second group
        for i,cell1 in enumerate(g1.cells):
            if cell1 == False: continue
            if cell1 == True:
                cell1 = self.OctreeLeaf(g1.bboxes[i])
                for cell2 in cell2_list:
                    if not boxBoxOverlap(cell1.center * t1 + c1, cell1.size_2, t1,
cell2.center*t2 + c2, cell2.size_2, t2, R):

                        continue

                    if not (cell1.is_leaf and cell2.is_leaf):
                        to_test_queue.append( (cell1,cell2) )
                    else:
                        return True

        return False

def draw(self):
    glPushMatrix()
    glMultMatrixf(self.mesh.transform)

```

```

glEnable(GL_BLEND)
glBlendFunc(GL_SRC_ALPHA, GL_ONE)
if self.mesh.getTranslucent():
    glDisable(GL_DEPTH_TEST)
self.root.draw()
if self.mesh.getTranslucent():
    glEnable(GL_DEPTH_TEST)
glDisable(GL_BLEND)
glPopMatrix()

```

### **linear.py**

```

import numpy as N

#inspired by code from NeHe

EPSILON = 0.00001

def Matrix4f ():
    return N.matrix(N.identity (4, 'f'))

def Matrix3f ():
    return N.matrix(N.identity (3, 'f'))

def Quat4f (x=None, y=None, z=None, w=None):
    if x is None:
        return N.zeros (4, 'f')
    else:
        return N.array((float(x),float(y),float(z),float(w)), 'f')

def Vector3f (x=None, y=None, z=None):
    if x is None:
        return N.zeros (3, 'f')
    else:
        return N.array((float(x),float(y),float(z)), 'f')

def Vector2f (x=None, y=None):
    if x is None:
        return N.zeros (2, 'f')
    else:
        return N.array((float(x),float(y)), 'f')

def Point3f (x=None, y=None, z=None):
    if x is None:
        return N.zeros (3, 'f')
    else:
        return N.array((float(x),float(y),float(z)), 'f')

def Point2f (x = 0.0, y = 0.0):
    pt = N.zeros (2, 'f')
    pt [0] = x
    pt [1] = y
    return pt

def VectorDot(u, v):
    return N.dot (u,v)

```

```

def VectorCross(u, v):
    return N.cross(u,v)

def VectorLength (u):
    return N.linalg.norm(u)

def Matrix3fSetIdentity ():
    return N.identity (3, 'f')

def Matrix3fMulMatrix3f (matrix_a, matrix_b):
    return N.dot( matrix_a, matrix_b )

def Matrix4fDet (matrix):
    X = 0
    Y = 1
    Z = 2
    s = sqrt (
        ( (matrix [X,X] * matrix [X,X]) + (matrix [X,Y] * matrix [X,Y]) + (matrix [X,Z] * matrix [X,Z]) +
          (matrix [Y,X] * matrix [Y,X]) + (matrix [Y,Y] * matrix [Y,Y]) + (matrix [Y,Z] * matrix [Y,Z]) +
          (matrix [Z,X] * matrix [Z,X]) + (matrix [Z,Y] * matrix [Z,Y]) + (matrix [Z,Z] * matrix [Z,Z]) ) /
    3.0 )
    return s

def Matrix4fSetRotationScaleFromMatrix3f(three_by_three_matrix):
    matrix = Matrix4f ()
    matrix [0:3,0:3] = three_by_three_matrix
    return matrix

def Matrix3fSetRotationFromQuat4f (q1):
    # Converts the quaternion q1 into a new equivalent 3x3 rotation matrix.
    X = 0
    Y = 1
    Z = 2
    W = 3

    matrix = Matrix3f ()
    n = N.dot(q1, q1)
    s = 0.0
    if (n > 0.0):
        s = 2.0 / n
    xs = q1 [X] * s; ys = q1 [Y] * s; zs = q1 [Z] * s
    wx = q1 [W] * xs; wy = q1 [W] * ys; wz = q1 [W] * zs
    xx = q1 [X] * xs; xy = q1 [X] * ys; xz = q1 [X] * zs
    yy = q1 [Y] * ys; yz = q1 [Y] * zs; zz = q1 [Z] * zs
    # This math all comes about by way of algebra, complex math, and trig identities.
    # See Lengyel pages 88-92

    matrix [X,X] = 1.0 - (yy + zz);          matrix [Y,X] = xy - wz;                      matrix [Z,X] = xz +
wy;
    matrix [X,Y] =    xy + wz;                matrix [Y,Y] = 1.0 - (xx + zz);          matrix [Z,Y] = yz - wx;
    matrix [X,Z] =    xz - wy;                matrix [Y,Z] = yz + wx;                      matrix [Z,Z] = 1.0 - (xx + yy)

```

```
return matrix
```

### **mesh.py**

```
from OpenGL.GL import *
from intersections import triangleBoxOverlap, rayTriangleIntersection, EPSILON
import numpy as N
import random
from linear import *
class Vertex:
    """ This class represents a single point in 3D along with its normal """

    def __init__(self, x, y, z, nx, ny, nz):
        self.pos = Point3f(x,y,z)
        self.normal = Vector3f(nx,ny,nz)
        self.octree = None

    def isSimilarTo(self, pt):
        v = self.pos - pt.pos
        d = N.dot(v,v) # get the squared magnitude of the vector v

        if d > EPSILON:
            return False

        v = self.normal - pt.normal
        d = N.dot(v,v)

        if d > EPSILON:
            return False

        return True

    def __str__(self):
        return "<%f,%f,%f> %s"%(self.x, self.y, self.z, self.pos)

# some properties to access the components by name (x,y,z,nx,ny,nz) rather than by index

    def set_x(self,val):
        self.pos[0] = val

    def set_y(self,val):
        self.pos[1] = val

    def set_z(self,val):
        self.pos[2] = val

    def set_nx(self,val):
        self.normal[0] = val

    def set_ny(self,val):
        self.normal[1] = val

    def set_nz(self,val):
        self.normal[2] = val
```

```

def get_x(self):
    return self.pos[0]
def get_y(self):
    return self.pos[1]

def get_z(self):
    return self.pos[2]

def get_nx(self):
    return self.normal[0]

def get_ny(self):
    return self.normal[1]

def get_nz(self):
    return self.normal[2]

x = property(get_x, set_x)
y = property(get_y, set_y)
z = property(get_z, set_z)

nx = property(get_nx, set_nx)
ny = property(get_ny, set_ny)
nz = property(get_nz, set_nz)

class Mesh:
    """ This class represents a mesh (points and faces) """

    DRAW_TRANSLUCENT = 1
    DRAW_EDGES = 2

    def __init__(self):
        self.name = ""
        self.orientation = Matrix3f()
        self.translation = Vector3f()
        self.color = (0.5,0.5,0.5)
        self.transform = Matrix4f()

        # list of all Point instances used in the mesh
        self.vertices= []

        # list of all Vector instances used in the mesh
        self.faces = []

        self._draw = None
        self.draw_mode = 0

    def getTranslucent(self):
        return bool(self.draw_mode & self.DRAW_TRANSLUCENT)

    def setTranslucent(self, enable):
        if enable:
            self.draw_mode |= self.DRAW_TRANSLUCENT
        else:

```



```

        self.draw_mode &= ~self.DRAW_TRANSLUCENT

def getDrawEdges(self):
    return bool(self.draw_mode & self.DRAW_EDGES)

def setEdges(self, enable):
    if enable:
        self.draw_mode |= self.DRAW_EDGES
    else:
        self.draw_mode &= ~self.DRAW_EDGES

def intersects(self, mesh):
    inter = self.octree.intersects(mesh.octree)
    print inter
    return inter

# see if we have another point already with the same (or almost same)
# position and normal. Otherwise, store the new point.
# return the index of the vertex in the list.
def _getVertexIndex(self, vtx):
    for i,v in enumerate(self.vertices):
        if vtx.isSimilarTo(v):
            return i

    # this is a new vertex. Add it to the list
    self.vertices.append(vtx)
    return len(self.vertices)-1

# vertices and normals are lists of Vector3f
def addPoly(self, vertices, normals=None):
    if normals == None:
        normal = VectorCross(vertices[1] - vertices[0], vertices[2] - vertices[0]);
        norm = VectorLength(normal)
        if norm > 0.0001:
            normal /= norm
        else:
            normal = Vector3f(0,0,0)

        normals = [normal] * len(vertices)

    v_indices = []
    for v,n in zip(vertices,normals):
        vtx = self._getVertexIndex( Vertex(v[0], v[1], v[2], n[0], n[1], n[2]) )
        v_indices.append( vtx )

    self.faces.append( tuple(v_indices) )

def setColor(self, rgb):
    self.color = rgb

def setName(self, name):
    self.name = name

```

```

@property
def bounds(self):
    if not self._draw:
        # make sure self.pos_data has been calculated
        self._setupOpenGLData()

    return N.min(self.pos_data, axis=0), N.max(self.pos_data, axis=0)

# vec must be a 3-element vector
def setPosition(self, vec):
    self.transform[3,0:3] = vec

# mtx must be a 3x3 matrix
def setOrientationMatrix(self, mtx):
    self.transform[0:3,0:3] = mtx

def _setupOpenGLData(self):
    """based on code at:
    http://bazaar.launchpad.net/~mcfletch/openglcontext/trunk/view/head:/OpenGLContext/drawcube.py """

    triangle_vertex_indices = []
    edge_vertex_indices = []

    # to keep track of edges already processed
    edges = { }

    tot_vertices = len(self.vertices)

    pos_data = self.pos_data = N.zeros((tot_vertices,3),"f")
    normal_data = N.zeros((tot_vertices,3),"f")

    i = 0
    for v in self.vertices:
        pos_data[i] = v.pos
        normal_data[i] = v.normal
        i+=1

    for f in self.faces:
        triangle_vertex_indices.extend(f)

        # store in vtx_pair the two indices that conform an edge of the facet, with lowest index
        # this is so we can test if we have already added that edge and skip it
        # otherwise, we add the indices of both end points to the edge_vertex_indices list

        prev_v_idx = f[-1] # last vertex
        for v_idx in f:

            if prev_v_idx > v_idx:
                vtx_pair = (v_idx, prev_v_idx)
            else:
                vtx_pair = (prev_v_idx, v_idx)

            if vtx_pair not in edges:
                edges[vtx_pair] = True
                edge_vertex_indices.extend( vtx_pair )

```

first

```

        prev_v_idx = v_idx

triangle_vertex_indices = N.array( triangle_vertex_indices, N.uint32)
edge_vertex_indices = N.array( edge_vertex_indices, N.uint32)

def draw(self):
    glPushMatrix()
    glMultMatrixf(self.transform)

    glPushClientAttrib(GL_CLIENT_ALL_ATTRIB_BITS)
    try:
        glEnable (GL_POLYGON_OFFSET_FILL);

        if self.draw_mode & self.DRAW_TRANSLUCENT:
            glEnable(GL_BLEND)
            glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)

        glPolygonOffset(1.0,1.0);
        glEnableClientState(GL_VERTEX_ARRAY)
        glEnableClientState(GL_NORMAL_ARRAY)
        glVertexPointerf(pos_data)
        glNormalPointerf(normal_data)

        glColor4f(self.color[0], self.color[1], self.color[2],0.3);

        glDrawElementsui( GL_TRIANGLES, triangle_vertex_indices)

        if self.draw_mode & self.DRAW_EDGES:
            glDisable(GL_LIGHTING)
            glDisableClientState(GL_NORMAL_ARRAY)
            glDrawElementsui( GL_LINES, edge_vertex_indices)
            glEnable(GL_LIGHTING)

        if self.draw_mode & self.DRAW_TRANSLUCENT:
            glDisable(GL_BLEND)

    finally:
        glDisable (GL_POLYGON_OFFSET_FILL);
        glPopClientAttrib()
        glPopMatrix()

self._draw = draw;

def draw(self):
    """ Send everything to OpenGL for rendering """
    if not self._draw:
        self._setupOpenGLData()

    self._draw(self)

# get a tuple with the vertices (as Vector3f) of the given face
def faceVertices(self, face):
    return tuple(self.vertices[i].pos for i in face)

def rayFaceIntersection(self, face, ray_origin, ray_dir):

```

```

        return rayTriangleIntersection(ray_origin, ray_dir, self.faceVertices(face))

# Tests whether the passed AABB ("the box") and the polyhedron ("the mesh") intersect, and returns:
# 0 - the box and the mesh don't intersect at all
# 1 - the box intersects the mesh, or the mesh is fully contained in the box
# 2 - the box is fully contained in the mesh
def intersectsAABB(self, box):
    c_min, c_max = box

    # make sure the bounds can be calculated
    if not self._draw:
        self._setupOpenGLData()

    # TEST 1
    # If all the vertices of the mesh are all to the same side of the box, then the box and the mesh don't
intersect.    # That is, we're checking if the bounding box of the mesh doesn't intersect the other box.

    mesh_min, mesh_max = self.bounds

    if ((mesh_min[0] > c_max[0]).all() or (mesh_max[0] < c_min[0]).all() or # test in X
        (mesh_min[1] > c_max[1]).all() or (mesh_max[1] < c_min[1]).all() or # test in Y
        (mesh_min[2] > c_max[2]).all() or (mesh_max[2] < c_min[2]).all()): # test in Z
        return 0

    # TEST 2
    # If all the vertices of the mesh are inside the box, then it is fully contained in the box
    if ((mesh_min[0] >= c_min[0]).all() and (mesh_max[0] <= c_max[0]).all() and
        (mesh_min[1] >= c_min[1]).all() and (mesh_max[1] <= c_max[1]).all() and
        (mesh_min[2] >= c_min[2]).all() and (mesh_max[2] <= c_max[2]).all()):
        return 1

    # TEST 3
    # Check if any face of the mesh intersects the box
    # NOTE: this assumes all faces are triangles
    box_center = (c_max + c_min)/2
    # we subtract epsilon so boxes that are touching the edges of the mesh don't get marked as
intersecting    # the mesh
    #
    box_half_size = (c_max - c_min)/2 - EPSILON
    triverts = N.zeros((3,3), "f")
    for f in self.faces:
        for v_num, v_idx in enumerate(f):
            triverts[v_num] = self.vertices[v_idx].pos

        if triangleBoxOverlap(box_center, box_half_size, triverts):
            return 1

    # TEST 4
    # At this point, we know the box is either fully contained in or fully outside the mesh.
    # Shoot a random ray from the center of the box. If it hits an even number
    # of mesh faces, then the center (and hence the box) is outside the polyhedron,
    # otherwise it's inside.
    # To avoid floating point imprecisions, if the ray hits a polygon (almost) on its edge,

```

```

# we discard the ray and try another random one

while True:
    # generate a random vector
    ray_dir = N.array(map(lambda a:random.uniform(-1,1), range(3)), "f")

    # make sure the vector is not too small... if it is, pick another one
    mag = N.dot(ray_dir, ray_dir)

    if mag < 0.01:
        continue

    # flip the parity whenever we hit a polygon (False means we're outside)
    parity = False

    ok = True
    for face in self.faces:
        inter = self.rayFaceIntersection(face, box_center, ray_dir)
        if inter is None:
            continue

        t,u,v = inter
        if t < EPSILON: continue

        parity = not parity

        # we hit an edge of a triangle, generate another ray
        if u < EPSILON or v < EPSILON or (1-u-v) < EPSILON:
            ok = False
            break

    if ok:
        break

    if parity:
        return 2

    return 0

```

### **archball.py**

```

from linear import *
from math import sqrt
EPSILON = 0.00001

# based on Arcball from the NeHe site (example 48)
# plus ideas from http://www.opengl.org/wiki/Trackball

class Arcball:
    def __init__(self, NewWidth, NewHeight):
        self.v1 = Vector3f()
        self.v2 = Vector3f()
        self.m_AdjustWidth = 1.0
        self.m_AdjustHeight = 1.0
        self.setBounds(NewWidth, NewHeight)

```

```

def setBounds (self, NewWidth, NewHeight):
    # //Set new bounds
    assert (NewWidth > 1.0 and NewHeight > 1.0), "Invalid width or height for bounds."
    # //Set adjustment factor for width/height
    self.m_AdjustWidth = 1.0 / ((NewWidth - 1.0) * 0.5)
    self.m_AdjustHeight = 1.0 / ((NewHeight - 1.0) * 0.5)

def _mapToSphere (self, screen_pt):
    # Given a new window coordinate, will modify NewVec in place
    X = 0
    Y = 1
    Z = 2

    # //Copy paramter into temp point
    pt = Vector3f(screen_pt[0], screen_pt[1], 0)
    # //Adjust point coords and scale down to range of [-1 ... 1]
    pt[X] = (screen_pt[X] * self.m_AdjustWidth) - 1.0
    pt[Y] = 1.0 - (screen_pt[Y] * self.m_AdjustHeight)
    pt[Z] = 0

    sphere_radius = 0.8
    r2 = sphere_radius*sphere_radius

    len_sq = N.dot(pt, pt)

    if len_sq < (r2*0.5):
        z = sqrt(r2 - len_sq)
    else:
        z = (r2*0.5) / sqrt(len_sq)

    pt[Z] = z

    len_sq = N.dot(pt,pt)

    return pt / sqrt(len_sq)

def click (self, NewPt):
    # //Mouse down (Point2f)
    self.v1 = self._mapToSphere (NewPt)
    return

def drag (self, NewPt):
    # //Mouse drag, calculate rotation (Point2f Quat4f)
    """ drag (Point2f mouse_coord) -> new_quaternion_rotation_vec
    """
    X = 0
    Y = 1
    Z = 2
    W = 3

    self.v2 = self._mapToSphere (NewPt)

    # //Compute the vector perpendicular to the begin and end vectors
    # Perp = Vector3f ()
    perp = N.cross(self.v1, self.v2);

```

```

    quat = Quat4f ()
    # //Compute the length of the perpendicular vector
    if (N.dot(perp,perp) > EPSILON):          # //if its non-zero
        # //We're ok, so return the perpendicular vector as the transform after all
        quat[0:3] = perp
        # //In the quaternion values, w is cosine (theta / 2), where theta is rotation angle
        quat[W] = N.dot(self.v1, self.v2);

    return quat

```

### stl.py

```

import numpy as N

from mesh import Mesh

def loadSTL(fname):

    # an iterator that will yield all the lines in the file,
    # skipping empty lines
    def getLines(fname):
        with open(fname) as f:
            for line in f.readlines():
                line = line.strip()
                if line == "": continue
                yield line

    mesh = Mesh()

    for line in getLines(fname):
        if line.startswith("solid"):
            mesh.name = line.split()[1:]
            continue

        if line.startswith("outer loop"):
            vertices = []
            continue

        if line.startswith("vertex"):
            pieces = line.split()
            vertices.append( N.array(map(float, pieces[1:4]),"f") )
            continue

        # we add a face to the mesh whenever we find a line that starts with "endloop"
        if line.startswith("endloop"):
            mesh.addPoly(vertices)

    return mesh

if __name__ == "__main__":
    mesh = loadSTL("fehe/static.STL")

    mesh._setupOpenGLData()

```

### scene.py

```
import math
from linear import *
import numpy as N

# this class keeps record of the elements to draw and allows interaction
# with them:
# - pieces
# - octrees
# - grid
class Scene:
    def __init__(self):
        self.inter = []
        self.helpers = {}
        self.disabled = set()
        self.current_object = 0

    def addInteractive(self, obj):
        self.inter.append(obj)

    def addHelper(self, helper, group=0):
        try:
            self.helpers[group].append(helper)
        except:
            self.helpers[group] = [helper]

    def disableHelperGroup(self, helper_group):
        self.disabled.add(helper_group)

    def enableHelperGroup(self, helper_group):
        self.disabled.remove(helper_group)

    def toggleHelperGroup(self, helper_group):
        if helper_group in self.disabled:
            self.enableHelperGroup(helper_group)
        else:
            self.disableHelperGroup(helper_group)

    def toggleTranslucent(self):
        t = not self.inter[0].getTranslucent()

        for d in self.inter:
            d.setTranslucent(t)

    def toggleFaceEdges(self):
        t = not self.inter[0].getDrawEdges()

        for d in self.inter:
            d.setEdges(t)

    def draw(self):
        for d in self.inter:
```



```

        d.draw()

    for group in self.helpers:
        if group not in self.disabled:
            for h in self.helpers[group]:
                h.draw()

    def setCameraPosition(self, cam_pos, cam_target, up = None):
        glLoadIdentity();
        if up_vector is None:
            up = (0,1,0)
        gluLookAt(cam_pos[0], cam_pos[1], cam_pos[2], cam_target[0], cam_target[1], cam_target[2],
up[0], up[1], up[2])

    def selectObject(self, num):
        self.current_object = max(min(len(self.inter)-1, num), 0)
        print "Object selected: ",self.current_object

    def translate(self, amount):
        t = self.inter[self.current_object].transform
        t[3,0] += amount[0]
        t[3,1] += amount[1]
        t[3,2] += amount[2]

    def rotate(self, axis, amount = 10):
        len = N.dot(axis,axis)
        matrix = Matrix3f()
        if len < EPSILON:
            return matrix
        axis /= len
        angle = amount * (math.pi/180.0)
        s = math.sin(angle*0.5)
        quat = Quat4f(axis[0]*s, axis[1]*s, axis[2]*s, math.cos(angle*0.5))
        matrix = Matrix3fSetRotationFromQuat4f(quat)
        t = self.inter[self.current_object].transform[0:3,0:3]
        self.inter[self.current_object].transform[0:3,0:3] = t*matrix

    def storeTransform(self):
        self.stored_transform = self.inter[self.current_object].transform.copy()

    def restoreTransform(self):
        self.inter[self.current_object].transform = self.stored_transform

    def checkCollisions(self):
        # check the current object against all others
        current = self.inter[self.current_object]

        for i,obj in enumerate(self.inter):
            if i == self.current_object:
                continue

            if obj.intersects(current):
                return True

        return False

```

### mouseinteractions.py

```
from OpenGL.GL import *
from arcball import Arcball
from linear import *
import math
class MouseInteraction:
    PAN_ACTION = 0
    ROTATE_ACTION = 1
    ZOOM_ACTION = 2

    def __init__(self, vp_width, vp_height):
        self.arcball = Arcball(vp_width, vp_height)
        self.vp_width = vp_width
        self.vp_height = vp_height

        self.transform_matrix = Matrix4f()
        self.manipulation_matrix = None

    def vpResize(self, vp_width, vp_height):
        self.arcball.setBounds(vp_width, vp_height)
    def applyTransform(self):

        if self.manipulation_matrix is not None:
            glMultMatrixf(self.manipulation_matrix)

            glMultMatrixf(self.transform_matrix)
    def click(self, x, y, action):
        cp = (x,y)
        self.click_pos = cp
        self.action = action
        if action == self.ROTATE_ACTION:
            self.arcball.click( cp )
        elif self.action == self.PAN_ACTION:
            self.manipulation_matrix = Matrix4f()
        elif self.action == self.ZOOM_ACTION:
            self.manipulation_matrix = Matrix4f()

    def release(self):
        if self.manipulation_matrix is not None:
            self.transform_matrix = self.transform_matrix * self.manipulation_matrix
            self.manipulation_matrix = None

    def drag(self, x, y):
        if self.action == self.ROTATE_ACTION:
            rot_quat = self.arcball.drag( (x,y) )
            self.manipulation_matrix =
Matrix4fSetRotationScaleFromMatrix3f(Matrix3fSetRotationFromQuat4f(rot_quat))
        elif self.action == self.PAN_ACTION:
            self.manipulation_matrix[3,0] = (x - self.click_pos[0])*0.2
            self.manipulation_matrix[3,1] = (self.click_pos[1] - y)*0.2
        elif self.action == self.ZOOM_ACTION:
            s = math.exp(float(x - self.click_pos[0]) / self.vp_width)
```

```

self.manipulation_matrix[0,0] = s
self.manipulation_matrix[1,1] = s
self.manipulation_matrix[2,2] = s

```

### *grid.py*

```

from OpenGL.GL import *
import numpy
from linear import *
class Grid:
    def __init__(self, scale, divisions):
        self.points = []

        points = []
        indices1 = []
        indices2 = []

        self.color2 = (0.8,0.8,0.8)
        self.color1 = (0.3,0.3,0.3)

        j = 0
        for i in range(-divisions, divisions+1):
            if i == 0: continue # skip the axes for now

            t = float(i) / float(divisions)

            points.append( Vector3f(t * scale, scale, 0) )
            points.append( Vector3f(t * scale, -scale, 0) )
            points.append( Vector3f(scale, t*scale, 0) )
            points.append( Vector3f(-scale, t*scale, 0) )

            indices1.extend( [j,j+1, j+2, j+3] )
            j += 4

        # positions of the axes

        points.append( Vector3f(0, scale, 0) )
        points.append( Vector3f(0, -scale, 0) )
        points.append( Vector3f(scale, 0, 0) )
        points.append( Vector3f(-scale, 0, 0) )

        indices2 = [j,j+1, j+2, j+3]

        self.points = numpy.array( points, "f" )
        self.indices1 = numpy.array( indices1, N.uint32)
        self.indices2 = numpy.array( indices2, N.uint32)

    def draw(self):
        glPushClientAttrib(GL_CLIENT_ALL_ATTRIB_BITS)
        try:
            glEnableClientState(GL_VERTEX_ARRAY)
            glVertexPointerf(self.points)

            # we don't want lighting to affect the grid
            glDisable(GL_LIGHTING)

```

```

        # draw the grid

        glColor3f(self.color1[0], self.color1[1], self.color1[2]);

        glDrawElementsui( GL_LINES, self.indices1)

        # draw the axes

        glColor3f(self.color2[0], self.color2[1], self.color2[2]);

        glDrawElementsui( GL_LINES, self.indices2)

        glEnable(GL_LIGHTING)

    finally:
        glPopClientAttrib()

```

### **cube.py**

```

from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

# c1 and c2 are opposite corners of the cube
def drawCube(c1, c2):
    glPushMatrix()
    center = (c1 + c2)/2;
    size = (c2 - c1)

    glTranslatef( center[0], center[1], center[2])
    glScalef( size[0], size[1], size[2])
    glutWireCube(1.0);

    glPopMatrix()

```