AN ADVANCED MACHINING PROCESS SIMULATOR FOR INDUSTRIAL APPLICATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

EMRE YEĞİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MECHANICAL ENGINEERING

DECEMBER 2010

Approval of the thesis:

AN ADVANCED MACHINING PROCESS SIMULATOR FOR INDUSTRIAL APPLICATION

submitted by **EMRE YEĞİN** in partial fulfillment of the requirements for the degree of **Master of Science in Mechanical Engineering, Middle East Technical University** by,

Prof. Dr. Canan Özgen _______________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Suha Oral _______________
Head of Department, **Mechanical Engineering Dept., METU**

Asst. Prof. Dr. Melik Dölen _______________
Supervisor, **Mechanical Engineering Dept., METU**

Asst. Prof. Dr. A. Buğra Koku _______________
Co-Supervisor, **Mechanical Engineering Dept., METU.**

Examining Committee Members:

Prof. Dr. Engin Kılıç _______________
Mechanical Engineering Dept., METU

Asst. Prof. Dr. Melik Dölen _______________
Mechanical Engineering Dept., METU

Asst. Prof. Dr. A. Buğra Koku _______________
Mechanical Engineering Dept., METU

Asst. Prof. Dr. Yiğit Yazıcıoğlu _______________
Mechanical Engineering Dept., METU

Dr. Atilla Özgit _______________
Computer Engineering Dept., METU

Date: 14 / 12 / 2010

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name : Emre Yeğin

Signature :

ABSTRACT

AN ADVANCED MACHINING PROCESS SIMULATOR FOR INDUSTRIAL APPLICATIONS

Yeğin, Emre
M.S., Department of Mechanical Engineering
Supervisor: Asst. Prof. Dr. Melik Dölen
Co- Supervisor: Asst Prof. Dr. A. Buğra Koku

December 2010, 107 pages

Turning and milling are the main manufacturing techniques in industry. A great deal of time and money is spent for machining operations. Although most of the time, the tool path of a CNC machine tool is generated by a Computer Aided Manufacturing (CAM) software package, to be sure that the result of the machining operation will be as required, it is necessary to use a simulation software. There are various machining simulation software packages available in the market. However, they are not only expensive, but also specialized for only one of the before mentioned machining techniques. Most of the companies in the industry are small or medium scale ones and, it is not so easy for them to afford a specialized simulation software for that purpose.

In this thesis, it is aimed to develop a software package, which will be used to simulate advanced industrial machining processes, including turning and milling. Dexel modeling, which is generated by ray casting, and sweep plane algorithm with polygon clipping technique are used for visualization. For polygon clipping

technique, outer surfaces of the resultant workpiece are generated from planar contours. The software is developed in C# programming language and DirectX libraries are utilized for visualization purposes. With the aid of this software, it is also aimed to visually confirm the validity of both mill and lathe NC-code, by representing highly accurate 3D displayed results of these simulations.

Keywords: Machining simulation, ray casting, dexel modeling, polygon clipping, surface generation

ÖZ

ENDÜSTRİYEL UYGULAMALAR İÇİN İLERİ TALAŞLI İMALAT SÜREÇ SİMÜLATÖRÜ

Yeğin, Emre
Yüksek Lisans, Makina Mühendisliği Bölümü
Tez Yöneticisi: Yard. Doç. Dr. Melik Dölen
Ortak Tez Yöneticisi: Yard. Doç. Dr. A. Buğra Koku

Aralık 2010, 107 sayfa

Tornalama ve frezeleme, endüstriyel uygulamalarda kullanılan başlıca talaşlı imalat teknikleridir. Talaşlı imalat uygulamaları için çok yüklü miktarda zaman ve para harcanmaktadır. Çoğu zaman sayısal kontrol verisi bir bilgisayar programı tarafından yaratılıyor olsa da, işleme operasyonu sonucunun istenilen neticeyi vereceğinden emin olmak için bir simülasyon programı kullanmak gerekir. Piyasada birçok ticari talaşlı imalat simülasyonu yapan yazılım mevcuttur. Ancak, çok pahalı olmakla birlikte, yukarıda bahsedilen işleme tekniklerinden birine yönelik olarak hazırlanmış olmaktadırlar. Endüstriyel uygulamaları olan şirketlerin çoğu, küçük ve orta ölçekli şirketlerdir ve bu firmalar için belirli bir işleme tekniğine yönelik hazırlanmış bir simülasyon programı için bütçe ayırmak kolay olmamaktadır.

Bu tez kapsamında, aralarında tornalama ve frezelemenin de bulunduğu, ileri endüstriyel talaşlı imalat süreçlerinin benzetiminde kullanılacak bir yazılım paketinin geliştirilmesi amaçlanmıştır. İş parçasının ve kesici takımın etkileşimini görselleştirmek için ışın göndeme metodu kullanarak oluşturulan dexel modeli ve yüzey süpürme algoritması ile polygon kırpma teknikleri kullanılacaktır. Bu yazılım paketinin geliştirilmesi için C# proglamlama dili, görselleştirme amacıyla ise

DirectX kütüphanelerinden faydalanılacaktır. Bu yazılım sayesinde, yapılacak benzetimlerin, yüksek hassasiyette 3 boyutlu görsel sonuçlarının sunulması ile, freze ve torna NC kodlarının doğruluğu teyit edilebilecektir.

Anahtar Kelimeler: İşleme simülasyonu, ışın gönderme, dexel modelleme, polygon kırpma, yüzey oluşturma

To my wife

ACKNOWLEDGMENTS

I would like to express my deepest gratitude and appreciation to my supervisor Asst. Prof. Dr. Melik Dölen and my co-supervisor Asst. Prof. Dr. A.Buğra Koku for giving me this great opportunity to get into the exciting world of numerical controlled machining simulation modeling. I deeply appreciate your priceless guidance, support, and motivation to share every bit of your knowledge and ideas until the very end.

In addition, my utmost and sincere thanks go to my wife, my mother-in-law, my sister-in-law, and my parents for their extreme contribution in making me an independent and confident person by their invaluable advices and guidance, and all their unconditional love and support.

I would like to express my appreciation to my friend Hasan Ali Hatipoğlu for his precious support, guidance and encouragement throughout my study

I would like to thank to my friends Alper Uyanık and Ali Serkan İnkaya for their excitement about my study and for their friendship and all the good times we had.

Finally, I would like to thank to Ercan Kiraz for being so generous by sharing his work, which was very helpful for me to improve my study.

TABLE OF CONTENTS

ABSTRACT iv
ÖZ vi
ACKNOWLEDGMENTS ix
TABLE OF CONTENTS x
LIST OF TABLES xiii
LIST OF FIGURES xiv
LIST OF ABBREVIATIONS xvi
CHAPTERS
1.INTRODUCTION 1
1.1 Motivation 4
1.2 Aim and Scope of This Study 5
1.3 Outline of the Study 6
2.LITERATURE SURVEY 7
2.1 Solid Modeling 7
2.2 Discrete Modeling 8
2.2.1 Z-buffer Algorithms 9
2.2.2 Surface Normal Algorithms 12
2.2.3 Voxel-based Algorithms 14
2.3 Polygon Clipping 15
3.NC CODE PARSING 16
3.1 Motion Types 17
3.1.1 Linear Motion 17
3.1.2 Rapid Motion 19
3.1.3 Circular Motion 19
4.TURNING SIMULATION 23
4.1 Introduction 23
4.2 Workpiece and Cutting Tool Definition 24
4.2.1 Pixel based technique 24
4.2.1.1 Workpiece Data Structure in pixel based technique 24

4.3.1.2 Cutting Tool Data Structure in pixel based technique 26
4.3.2 Polygon Clipping technique 27
4.3.2.1 Workpiece Data Structure in polygon clipping technique 27
4.3.2.2 Cutting Tool Data Structure polygon clipping technique 29
4.4 Cutting Methodology 31
4.4.1 Cutting methodology in pixel-based technique 31
4.4.2 Polygon clipping technique and its application in the simulation 33
4.4.2.1 Introduction 33
4.4.2.2 Polygon Clipping 34
4.4.2.2.1 Basics 34
4.4.2.2.2 The algorithm 36
4.4.2.2.3 Selecting the result edges 42
4.4.2.2.4 Connecting the result edges to form the solution 46
4.4.2.2.5 Performance analysis 47
4.4.2.2.6 Special cases 51
4.4.3 Polygon sweeping methodology 52
4.4.3.1 Introduction 53
4.4.3.2 Basics of Polygon Sweep 53
4.4.3.2 Polygon sweeping algorithm 54
4.4.4 Algorithm of Turning simulation with Polygon operations 57
4.5 Visualization Methodology 59
4.5.1 Visualization in the cross-section viewport 59
4.5.1.1 Visualization in the Cross-Section Viewport in pixel-based simulation 60
4.5.1.2 Visualization in the Cross-Section Viewport in polygon clipping based simulation 60
4.5.2 Visualization in the 3D Model Viewport 61
4.5.2.1 Boundary vertices data generation in pixel-based simulation 62
4.5.2.2 Basics of 3D objects in DirectX 62
4.5.2.3 Mesh data generation 64
4.6 Features of the Program 66
4.6.1 System requirements 66
4.6.2 User interface 66
4.6.2.1 User interface in pixel-based simulation 66
4.6.2.2 User interface in polygon clipping based simulation 68

4.7 Closure 70
5.MILLING SIMULATION 73
5.1 Introduction 73
5.2 Workpiece and Cutting Tool Definition 73
5.2.1 Dexel Data Structure 73
5.2.2 Ray Casting 74
5.3 Cutting Methodology 77
5.4 Visualization Methodology 79
5.5 User Interface 80
5.6 Closure 82
6.CASE STUDIES 83
6.1 Introduction 83
6.2. Turning simulation case study with pixel based technique 83
6.3. Turning simulation case studies with polygon clipping technique 86
6.3.1 Case study 1 86
6.3.1 Case study 2 88
6.3 Milling Simulation - Sample Application 90
7.CONCLUSIONS & FUTURE WORK 93
REFERENCES 100
APPENDIX A 104
NC PROGRAM LISTINGS 104
A.1 NC program used in Case Study 1 104
A.2 NC Program used in Case Study 2 105
A.3 NC Program used in Milling Simulation-Sample Application 106

LIST OF TABLES

TABLES

Table 3.1 Circular motion representations. 20

Table 4. 1 Workpiece data structure 28
Table 4. 2 Format of cutting tool vertex data file. 29
Table 4. 3 Cutting tool data structure. 30
Table 4. 4 Definition of workpiece and cutter matrices and modification of workpiece matrix with the application of Boolean operation. 32
Table 4. 5 Algorithm for Boolean operations. 39
Table 4. 6 *SweepEvent* structure. 44
Table 4. 7 *PolygonType*, *EdgeType*, and *BooleanOpType* enums. 44
Table 4. 8 Routine to set *inside* and *inOut* flags of edges. 45
Table 4. 9 Sorting procedure to obtain connected vertices Polygon vertices. 47
Table 4. 10 Execution times of Boolean operations on the polygons shown in Figures 4.11-4.14. Polygon 1 is composed of 94 vertices, polygon 2 is composed of 54 vertices. 48
Table 4. 11 Polygon Sweeping function. 56
Table 4. 12 Pseudo code of Turning Simulation algorithm. 59
Table 4. 13 Code snippet of scale factor. 61
Table 4. 14 Mesh generation algorithm. 65

Table 5. 1 Definition of Intersection function. 75
Table 5. 2 Cutting procedure. 78

Table 6. 1 Execution times and memory requirements in VirtualMachining for the first example NC program. 86
Table 6. 2 Execution times and memory requirements in CNCVerifier for the second example NC program. 89

LIST OF FIGURES

FIGURES

Figure 1. 1 CSG tree to construct a complex object with simple geometric............. 2
Figure 1. 2 (a) Z-buffer technique [33], (b) Surface normal technique [34]............. 3
Figure 1. 3 A torus shaped object, composed of voxels [34]................................... 4

Figure 3.1 Linear Motion [38]. 17
Figure 3.2 Circular Motion [38]... 20

Figure 4. 1 Taking a cross-section of a cylindrical workpiece. 24
Figure 4. 2 Discretization of the workpiece cross-section...................................... 25
Figure 4. 3 Converting workpiece into pixel matrix, then converting this pixel data, into a Boolean matrix.. 26
Figure 4. 4 Workpiece geometry.. 28
Figure 4. 5 Tool editor of the CNCSimulator 30
Figure 4. 6 Axes in turning operation. 33
Figure 4. 7 Boolean operations on polygons [21] 35
Figure 4. 8 Subdivision of edges of polygons at their intersection points [21] 36
Figure 4. 9 Types of intersection [21] 42
Figure 4. 10 Sweep-line technique [21] 42
Figure 4. 11 User interface of simple polygon clipping program for test purposes. In the figure, the screen view of the program, with two arbitrary polygons, is shown. The black one is the subject polygon, with 94 vertices. The red polygon is the clipping polygon, with 54 vertices. 49
Figure 4. 12 Result of Boolean subtraction.. 49
Figure 4. 13 Result of Boolean Intersection... 50
Figure 4. 14 Result of Boolean Union 50

Figure 4. 15 Inclusion of overlapping edges in result of Boolean operations [21] .. 52
Figure 4. 16 Result of sweeping a line segment 54
Figure 4. 17 When swept polygon resides in the starting polygon 56
Figure 4. 18 Sweeping operations as applied on different tool polygons. 57
Figure 4. 19 Vertex definition of a polygon and its front face [18]. 63
Figure 4. 20 Turning simulation user interface for pixel based simulation. 67
Figure 4.21 Turning simulation user interface for polygon clipping based simulation. 70

Figure 5. 1 The dexel data structure developed by Van Hook [18]. 74
Figure 5. 2 Cut procedure cases, Van Hook [18]. 78
Figure 5. 3 User interface of Milling Simulation. 81

Figure 6. 1 The workpiece is rotated to visualize it in 3D space. 84
Figure 6. 2 The results of the simulation as shown in solid mode. 85
Figure 6. 3 The result of the simulation in wireframe mode. 85
Figure 6. 4 (a) Result of simulation in 2D cross sectional view. (b) 3D object model representation of the workpiece in CNCVerifier. 87
Figure 6. 5 Screen view output of the CNCSimulator. (a) 2D cross section of the workpiece after simulation. (b) Isometric view of the workpiece. 88
Figure 6. 6 (a) Result of simulation in 2D cross sectional view. (b) 3D object model representation of the workpiece in CNCVerifier. 89
Figure 6. 7 Screen view output of the CNCSimulator. (a) 2D cross section of the workpiece after simulation. (b) Isometric view of the workpiece. 90
Figure 6. 8 The user interface for milling simulation before the simulation. 91
Figure 6. 9 Screen views after the completion of the milling simulation carried out in (a) CNCVerifier (b) CNCSimulator 92

LIST OF ABBREVIATIONS

ABBREVIATIONS

BMP	: Bitmap
CAD	: Computer Aided Design
CAM	: Computer Aided Manufacturing
CNC	: Computer Numerical Control
CPU	: Central Processing Unit
CSG	: Constructive Solid Geometry
CT	: Cutting Tool
CTP	: Cutting Tool Polygon
Dexel	: Depth element
DOF	: Degree of Freedom
GDI+	: Graphic Design Interface
GPU	: Graphics Processing Unit
GUI	: Graphical User Interface
IGES	: Initial Graphics Exchange Specification
ISO	: International Organization for Standardization
METU	: Middle East Technical University
NC	: Numerical Control
OOP	: Object Oriented Programming
PC	: Polygon Clipping
Pixel	: Picture Element
RAM	: Random Access Memory
STEP	: Standardized Exchange of Product
STL	: Stereo-lithograph
WP	: Workpiece
WPP	: Workpiece Polygon
Voxel	: Volume Element

CHAPTER 1

INTRODUCTION

In today's competitive industry, it is extremely important to use resources such as time, machine, and labor, etc., in the most effective way. In a wide variety of applications, the biggest proportion in industrial applications, are manufacturing processes. There are several types of manufacturing processes, but without any doubt, mechanical machining is the most widely applied one. Machining is a kind of application in which

- A vast amount of capital is spent in equipment,
- Highly skilled man power is required
- In most of the cases, considerably raw material is processed.

Thus, to be considered as competitive and efficient, cost effective and accurate applications are unavoidable. This demand becomes more and more important, which increases opportunities to develop new techniques to increase the quality and quantity of production for cheaper solutions. This trend could not be ignored in education, especially in engineering studies. Simulation takes place not only in both real industrial application but also in mechanical engineering studies. Moreover, machining simulation is not an exception. Mechanical machining simulation is widely studied and applied.

In the timeline of machining simulation's development, there is a one to one correspondence with the discoveries of the techniques used in computer graphics techniques. Several techniques are developed since 80's, and some of them

successfully adapted in generation of sculptured surface simulation and modeling. Below is the list of some techniques in the area:

- Solid modeling techniques
    - Wire frame modeling
    - Constructive Solid Geometry (CSG)
    - Boundary representations
- Discrete modeling techniques
    - Z-buffer algorithms
    - Surface normal algorithms
    - Voxel-based algorithms

In CSG, and geometric shape could be constructed as a composition of simpler entities like sphere, cylinder, cube, etc. For the buildup of the object Boolean operations, which are union, subtraction, and intersection, are used. In cases, where CSG techniques used for machining simulation, subtraction is the Boolean operation, applied to simulate material removal. In Figure 1.1 a simple example of CSG methodology is depicted.

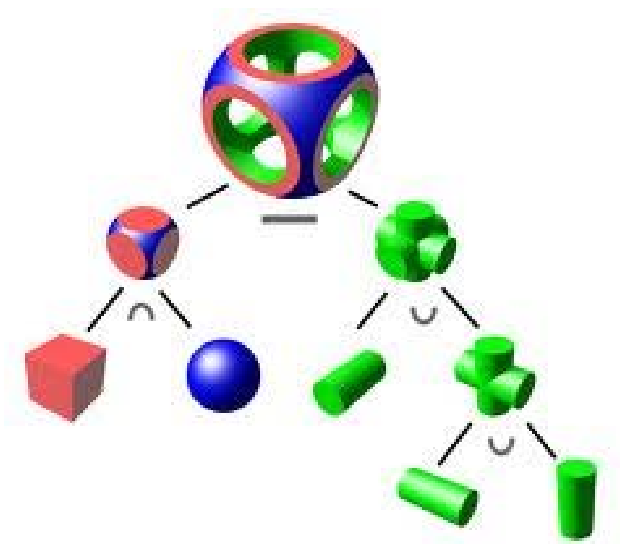


Figure 1. 1 CSG tree to construct a complex object with simple geometric entities [32].

Z-buffer techniques have been used in machining simulations early on. In these techniques, object in Z direction is considered and modeled as a one dimensional height elements and the cutting tool geometry is modeled, in a way to, crop height elements of workpiece object. Surface normal techniques are also very similar to Z-buffer techniques. The difference in surface normal technique is that, the workpiece model contains normal vectors on the outer surface and cutting tool once again crops these normal vectors. Surface normal technique lets user to see the amount of gouing and overcut, if there will be any, as a result of machining operation. Figures 1.2 illustrates these two techniques.

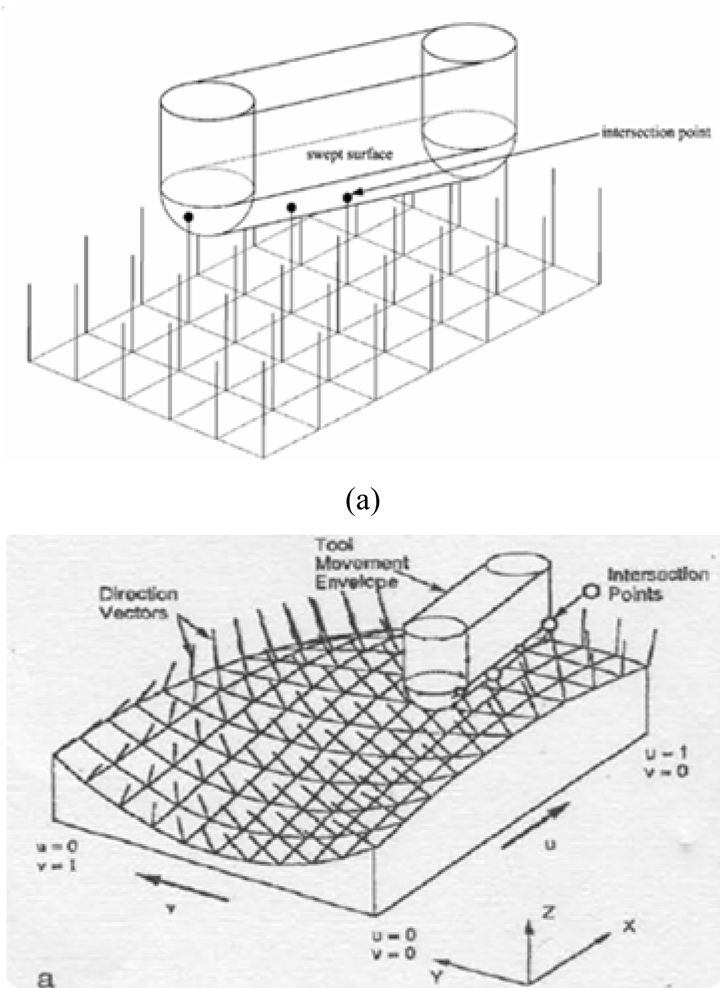


Figure 1. 2 (a) Z-buffer technique [33], (b) Surface normal technique [34].

With the increase in the computational capacity of computers, discretization in simulations increased as well. In voxel based technique, 3D space is divided into cubical elements, called voxel, and both workpiece and cutting tool are modeled with voxels. To generate solid model of the object with shaded surfaces, marching cubes algorithm, and its variances have been used. Later applications utilized the benefit of optimization algorithms like octree, etc. to decrease the amount of data in calculations, therefore to increase the simulation speeds. A voxelized object is shown in Figure 1.3 to give some idea.

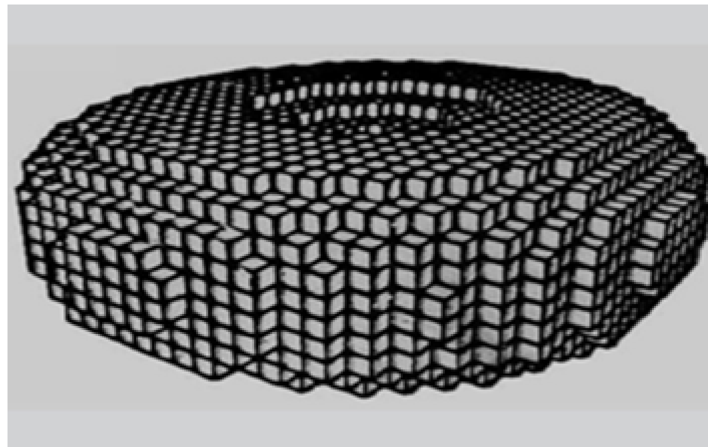


Figure 1. 3 A torus shaped object, composed of voxels [34].

1.1 Motivation

In the industry, mechanical machining is one of the mostly used techniques among all existing ones. In the timeline of developments in machining, since 1940s and '50s, numerically controlled (NC) machines made a leap in the rates of production. With the decrease in the cost of electronic components and other hardware, it became affordable even for medium or small-scale companies to have NC machines.

In early 1970s, with the development of high performance computers, automotive and aerospace companies started to use Computer Aided Manufacturing (CAM) software for the generation of tool path for NC machining. Early on, it was common

to apply a trial machining with a dummy raw, which is both soft and easy to machine. Yet it does not always assure the effectiveness of a NC code. Moreover, it is slow and costly, due to decrease in the tool life, waste of machining time and material.

Due to these difficulties and limitations, use of virtual machining software becomes advantageous. Especially automotive and aerospace industries, manufacturing of complex parts and dies require trustable NC code.

Many and different sources of errors may be contained in an NC program. It could be in some cases a simple syntax error, or in some others an unwanted interaction of cutting tool or spindle with work-holding fixtures and other parts of the machine. To get rid of these unwanted cases, there is a need for debugging NC programs and utilizing a computer program for that purpose is a wise approach.

Today, NC machining simulation and verification is carried out by the use software packages. The main drawback of these packages is their high costs. Obviously, it deserves to have a virtual machining software to avoid worse expenses due to an erroneous NC program.

As a result, it is believed that, development of a virtual machining software is useful for the related studies. Especially, to get benefit in educational studies, such a software is necessary. It is thought that it will decrease the dependency on other packages.

1.2 Aim and Scope of This Study

As a readily available NC simulation software CNCSimulator® [31] is used in course ME440 - Numerically Controlled Machine Tools, at Mechanical Engineering Department, METU. Due to reason that, CNCSimulator is not freeware and there is no way to get the source code of the software for further development, in this thesis study development of a similar and in some ways a more advanced virtual machining software is aimed.

In this study, it is aimed to develop a machining simulator to discover the errors of an NC program, before it is run to manufacture a part in a milling and turning machine. It is believed that, by this way, it will give the programmer, the confidence of running an error-free NC program on a valuable NC machine.

The main focus is to model and simulate machining operation especially on an NC lathe, and some simple cases in NC milling machine. To model the workpiece and cutting tool, dexel modeling and sweep plane algorithms are used. Polygon clipping technique, and afterwards generation of surfaces from contour data will be applied to obtain satisfactory visual results. For the turning simulation a cut-plane of the workpiece and cutting tool will be taken into consideration, and the 3D shaded view of the workpiece will be generated. This capability does not exist in CNCSimulator. For milling both dexel modeling and ray casting techniques are applied.

CNCVerifier is the name of the program, developed in this study. The program is written in C# language and DirectX 9.0c libraries are used for visualization of 3D objects. The program can depict the material removal by the interaction of cutting tool with the workpiece.

1.3 Outline of the Study

The whole study can be divided into six chapters. The first one is introduction chapter to give preliminary understanding about the study. The next one is literature survey. In the literature survey chapter, past studies, especially solid modeling, and special algorithms for virtual machining are discussed. In Chapter 4, turning operation will be emphasized. Developed algorithm and mathematical explanation of the algorithm and 3D modeling will be presented. The following chapter will be dedicated for the milling simulation. For the milling simulation, first ray casting and dexel modeling will be discussed. Explanation of the algorithms with the methodology are presented in the chapters. Finally, conclusion and future work chapter will conclude this thesis.

CHAPTER 2

LITERATURE SURVEY

The main goal of NC code verification is straightforward. Given an NC program, defining tool motion, and initial workpiece geometry, its purpose is to determine whether the program can form the workpiece as expected or not.

To accomplish this goal, it is necessary to represent the cutting tool and workpiece geometry before they place in the cutting operation. First, the part description has to be made. The swept volume of the cutting tool must be calculated in the same way of the workpiece and cutting tool so that, swept volume could be subtracted from the workpiece geometry.

Given this brief information, a method for removing the swept volume of each cutting tool motion from the workpiece model, must be provided. The tool movements must be processed one by one, if it is required to do the simulation of the workpiece at any time. Solid modeling and discrete modeling are the two major modeling techniques used in simulating machining. In the next section, Solid modeling technique will mentioned

2.1 Solid Modeling

Solid modeling is representation of complex surfaces by building up simple objects as Boolean combinations of them. A very simple explanation of machining operation would be subtraction of the tool movement envelopes from the

workpiece. With the aid of machining simulation, an accurate model of the current state of the workpiece geometry could be obtained, and there is also opportunity to determine the material removed by each tool motion.

Solid modeling systems especially for NC simulation and verification were first investigated by Voelcker and Hunt in [1] and [2], Frishdale in [3]. On the other hand, using solid modeling to do simulation and verification has a down side. Simulation requires performing intersections between surfaces which leads to very computationally intensive operation [35]. In addition, in the case where the surfaces are too complicated, no known algorithm for performing the intersection may be utilized. Later work, in which discrete modeling was applied, addressed that problem by trying to spatially subdivide the problem to reduce the amount of intersection operation necessary. On the verification side, the comparison between the finished workpiece and the desired part was nontrivial. Even if they are exactly the same, floating point errors would most probably prevent that from being realized. Thus, a method of measuring the proximity of the two surfaces was needed, also requiring potentially expensive surface-based operations.

2.2 Discrete Modeling

Although solid modeling has the advantage of maintaining an exact representation of the machining process, unfortunately it is computationally intensive. By trading total accuracy for easier computation, one may have opportunity to obtain an alternate approach. The general concept of discrete modeling could be defined as: representing an object by an approximation constructed of points, triangle or any other simple objects that could be dealt with easily. The part surface and/or the workpiece are most of the time represented by a set of points, each having an associated vector with the point. The advantage of this approach is that intersections and other computations, involving points and vectors, becomes easier and fast.

Several researches, some of them are [4]-[16], proposed methods of NC simulation and verification using discrete approximations. These proposals could be divided

into three groups, Z-buffer algorithms, surface normal algorithms and voxel-based algorithms.

2.2.1 Z-buffer Algorithms

An early approach to simulation was designed by Anderson [4] to detect whether the tool holder hits the workpiece in three-axis machining. He set up a rectangular array (each grid cell containing a height value) to represent the workpiece. For each tool movement, the grid squares that fall under the tool's path are checked against the height of the tool and reduced in the case the tool cuts the remaining stock in the corresponding square. This is closely related to the Z-buffer algorithm used for graphics displays [36].

The amount of material removed could be approximated with this method, as can the final shape of the workpiece. Which means that gouge and excess material could be found. In this approach, a regular grid greatly simplifies the calculations, but means that grid size is closely related to the accuracy of the simulation. Another disadvantage is that material can only be removed from the top. This means that it is not possible to simulate five-axis cutting and undercutting with the tool.

Wang [5], Van Hook[6], and Atherton [7] each updated the Z-buffer approach to perform simulation and verification. Their methods allow a view direction to be chosen. It is the direction in which the workpiece will store height values. Wang [5] stored multiple Z values at each pixel, which allowed him to simulate five-axis machining. Scan line operations allow quick determination of the cutting depths of the tool movement at every single pixel. The resultant swept envelope Z-buffer values are used to update the workpiece representation. Since the surface values were originally used to create starting entries in the workpiece Z-buffer, comparing the finished workpiece and the desired part is relatively easy to verify whether desired workpiece geometry is obtained.

Van Hook's [6] approach is quite similar, except that a pixel image of the cutting tool is precomputed. Likewise, it could be subtracted from the workpiece Z-buffer, which finally gives the finished surface. Since the tool envelope image is precomputed, the tool axis is kept fixed during machining, and thus the method is limited to three-axis cutting operations. Atherton [37] extended this approach to allow five-axis cutting.

All these approaches have common advantages. Due to the regular nature of the Z-buffer grids, intersections with the tool envelopes are relatively simple and fast. The values of the surface and the workpiece are obtained at each pixel, allowing easy comparison of the final product to the desired part geometry. In addition, gouges and excess material left behind could be determined easily. Finally, the amount of material removed is easily computed.

However, they suffer from some drawbacks as well. The simulations are view-point dependent so that they must be redone if an error does not show up in a given viewing direction. The magnitude of a given error at a given pixel is variable which depends on the orientation of the surface at that point. If the surface is nearly parallel to the Z direction at the pixel position, the error value will not correctly reflect the size of the gouge or bump for that location. In addition, the accuracy of the simulation is also dependent on the size of the grid. To space samples on the part surface evenly, pixels must be closer together when the surface normal is nearly perpendicular to Z direction. Lastly, preparing the starting Z-buffer requires determination of the intersection of lines in the Z direction at each pixel with part surface. This is also referred to as the inverse point problem.

In the case of the parametric curved surfaces which are often used in part descriptions, solving the inverse point problem often requires iterative methods to get a valid answer; hence taking a fair bit of computational effort. For more complex surfaces, it is likely to get errors, which may be introduced at this step.

Accuracy is a major problem for methods using a regular grid. Grid sizes have to be based on the tightest spacing necessary all over the surface. This results in many

unnecessary samples in areas where they are not actually needed. Drysdale and Jerard [8] propose a method very similar to those above, but on the contrary with variable spacing. Their approach includes choosing points on the part surface combined with vectors at each point to represent the workpiece. In addition, they showed how to determine the space allowable between points on the surface which depends on the desired accuracy of the simulation, the size and shape of the cutting tool, and the amount of the curvature of the local surface. Moreover, they generated the sample points parametrically. This allowed them to avoid the inverse point problem. When the axis of every tool movement is exactly the same, highly efficient calculations could be performed by just picking the Z direction to coincide with the tool axis. In other words, surface vectors only have to intersect the tool bottom [9].

Although technically not exactly a Z-buffer, the point set was used in a similar manner. As in Z-buffer approaches, each successive tool movement is intersected with the surface vectors, to reduce the height when an intersection occurs. Though a regular grid is no longer used, efficient computation was still achieved by placing points into a grid of buckets (essentially a 2D hash table). Tool movements find the buckets they intersect and perform intersection calculations just limited with the vectors in those corresponding buckets.

This method sidesteps the inverse point problem by generating accurately placed sample points on the surface, instead of by projecting points from a fixed grid. The variable spacing also ensures that the number of points used to represent the surface and workpiece is much enough and not excessive. On the downside, the variable spacing makes material removal calculation more complicated than in the case of regular spacing. Moreover, since Z aligned vectors are being used, error estimates are affected by the angle between the surface normal and Z direction.

2.2.2 Surface Normal Algorithms

Chappel [10] used the point-vector approach to surface representation just described. However, in Chapel's [10] work each vector uses the surface normal just for orientation. Points were chosen on the part surface. At each surface point, the surface normals are extended in both directions. The first surface normal is directed into the part until another part or face is reached. And the second surface normal is directed out of the part until another part, face or the end of the stock is reached. The tool movement was simulated as a series of fixed tool positions, where cutting is done by finding the intersection of the vectors and cylinder positions. Consequently, interference and gouging were reported. It is possible to simulate 3-, 4-, and 5- axis machining with this method. However, Chapel netiher discussed how surface points are selected, nor how tool positions were chosen.

Oliver and Goodman [11] took this approach, and brought it further. They also considered the surface normal at each point on the surface as the vector to be cut. The point selection is done visually by selecting an orientation for the surface. The pixels are then projected onto the part surface, whereupon the normal is calculated and used as a normal vector. Simulation proceeds as previously described. By using surface normals, it becomes easier to simulate five-axis machining. Moreover, the error at each point is now more accurate. On the other hand, once again by picking their points using a view-based method, they experienced problems due to the grid accuracy, the view direction selected, and the inverse point problem.

Jerard and Drysdale [12] extended their point set approach to use normal vectors in order to simulate five-axis machining. Points are still on the surface, but they are spaced evenly to guarantee the accuracy of the simulation, thereby avoiding the inverse point problem. At the same time this approach ensures that there are just enough points needed to simulate at the desired accuracy and no more.

Their contribution also addressed the issue of localization for five-axis simulations. The problem here is that in this case normals pointing in all directions over the

surface instead of in a single direction. Hence, two-dimensional bucketing does not work correctly. A point may reside in one -bucket, yet its normal cuts through several. [12] suggests two methods for dealing with this issue.

The first method (called the short normal method) is to use a relatively short vector at each surface point. Instead of having a long vector that extends till the ends of the stock material, the vector this time is limited to a small distance above and below the surface. In this case, once again it becomes possible the points to be placed into buckets. When a tool envelope is compared to the bucket boundaries, the short normal length is used to expand the envelope to guarantee getting all points that might intersect with the tool trajectory.

The second approach (average normal method) uses a set of preselected directions to be utilized as normals. Each direction is associated with a bucket set. Each point uses the normal whose direction is closest among available ones to its own normal and is put into the bucket set for that direction. Cutting is then performed for each direction. This approach takes advantage of working with parallel vectors while reducing the errors result since the real normals are close to the chosen direction. Actually the direction vector is an average of the surface normals of the points they represent. Also, even though the cutting process is repeated over and over, the time required does not increase since each point is only placed into a single bucket set. In other words, the number of intersections performed does not increase.

One final discrete modeling method was proposed by Ozair [13, 14]. He applied the common technique of approximating surfaces by triangles to NC verification. By doing this, fewer primitives could be used in very flat regions. While surface points must be close to each other to prevent the tool from sticking down into the surface without detection, only three are enough to define a triangle to represent the same region. Triangle representation is very useful when the subjected region is a flat one, yet this representation suffers when the surface is highly curved in various places, since the cost of performing intersections is much higher. One other deficiency of the method is that the excess material could no longer be effectively

tracked. Also a gouge is associated with a triangle, so that if the corner of a large triangle is nicked, the whole region is flagged as gouge.

2.2.3 Voxel-based Algorithms

Jang et al [15] proposed a voxel-based model. In their study they aimed to develop a voxel-based simulator for multi-axis computer numerical control (CNC) machining. The developed simulator displays the machining process in which the initial workpiece is incrementally converted into finished part. The voxel representation was used to model the changing state of the in-process workpiece. The voxelized tool swept volumes are subtracted from the workpiece. They consider that, voxel operation simplified the Boolean operation and material removal volumes. It was quite simple to determine the material removal rate, because the material removal was the number voxels removed. They use that simplified case to adjust the feed rate adaptively.

Yau et al [16] used an adaptive voxel representation for multi-axis solid machining. They use adaptive octree to develop a multi-axis simulation procedure. Their objective to develop an algorithm in which during the machining process takes place at any time, it could be possible to visualize the state of the workpiece and to verify the correctness of the cutting route. It was possible to adjust the resolution of the voxel size, which affects the level of approximation on the model. They use some implicit functions to represent various cutter geometries. They also intended (by the use of these implicit functions) to determine the contact points of the cutter with speed and accuracy. They aimed to allow the user to conduct error analysis and compare between the generated cutting model and the original computer aided design (CAD) data.

2.3 Polygon Clipping

Polygon overlay, which is also known as polygon clipping (PC), Boolean operations determine the spatial coincidence of two polygons. Polygon overlay techniques are generally used in by field scientists to extract relationships between spatial attributes, stored as layers in a geophysical data model. A Boolean operation on the polygons could also be used to visualize changes in parties of a machining simulation. Especially, due to simplicity of turning simulation, it is possible to assume both workpiece and cutting tool as polygons in a 2D plane.

Many efficient algorithms exist for polygon clipping. Most of them are limited to certain types of polygons. For example, Andereev [24] and Sutherland and Hodgman [25] algorithms require convex clip polygons while the algorithm by Liang and Barsky [26] requires a rectangular clip polygon.

For more complicated polygons (i.e. concave polygons with holes and self-intersections) less solutions exist. Moreover, some of the solutions need complex and specific data structures. Study of Weiler [27] suffer from complexity. Geiner and Hormann [28] propose a new algorithm for polygon clipping. Their algorithm is easy to implement, and treats degeneracy, which occurs when a vertex of a polygon lies on an edge of the other. Liu et al. [29] proposed some optimization for Greiner and Hormann's algorithm. Unfortunately, their solution also suffers for degeneracy. Vatti [30] has also a clipping algorithm. Which is based on plane sweep paradigm. Another algorithm developed by Martinez et al [21] is faster and more efficient even if it is also based on plane sweep. In this study, for the polygon clipping technique, the algorithm developed by Martinez et al [21] is base.

CHAPTER 3

NC CODE PARSING

NC program is the code that designates the whole sequence of a specific machining operation to be performed on a CNC machine tool [39]. NC-codes have been initially designed for CNC machine tools although they are also applicable for the robot manipulators. In fact, the NC code can be changed to describe the end-effector trajectory to carry out the task with the given tolerance.

The NC code developed in this study includes information about the end-effector orientation, motion type, manipulator's operation modes and orbit (a.k.a. the "pace"), as well as the coordinates. This NC code consists of line (i.e. blocks) each representing information about a segment of the motion. Sub-program/subroutines can be used for trajectories with repetitive features. G-codes defines the motion type and the operation mode. In this study, the G codes that are used are as follows:

- G0 - rapid linear motion;
- G1 - Rectilinear motion;
- G2/G3 - circular motion,
- G17, G18, G19 - selection of the working plane in the circular motion;
- G90 / G91 - absolute and incremental coordinate mode;
- G100 - specification of the local coordinate frame.

In the NC code, tool coordinates X, Y, Z [mm], and various functions M and feedrate F [mm/min] can be defined in addition to the G-codes. `M98`, `M99` M function references a subprogram and subroutine respectively while M30 calls the

end of the subprogram. Furthermore, the N word is used to mark the start of a subroutine and the name of the subroutine to be called is defined by the P-word, which is used in cooperation with `M98` function. In addition, D (sec) defines the dwell function. For the task which needs to maintain its pose during the specific time interval, for example the spot-welding operation, it is important to use the D-word. Modal coding has been made use of the efficiency of the code and the easiness to follow it. In this scheme, any change is observed in the modes fixed by the G codes or the coordinates being specified until new mode or coordinate is entered.

3.1 Motion Types

Manipulators are able to fulfill various motion types like rectilinear, circular, helical and parabolic motions. However, in this study, rapid, linear and circular motion are just performed for the simplicity.

3.1.1 Linear Motion

In linear motion, the user sets the constant feed rate and the tool moves to the destination at that feed rate. In this mode, all axes work in harmonization and tool moves the same amount in each axis as shown in Figure 3.1. The linear motion is designated as `G1 Xxf Yyf Zzf Ff` where xf, yf, zf are the coordinates of the end-point (in absolute or incremental mode) and f is the feedrate (mm/min) set by the user.

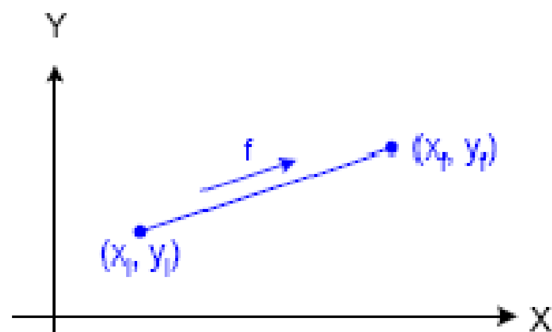


Figure 3.1 Linear Motion [38].

Note that at the start of each control cycle a linear interpolation is required to produce position commands to the controller. Firstly, the travel (Euclidian) distance should be calculated as

$$d = \sqrt{(x_f - x_i)^2 + (y_f - y_i)^2 + (z_f - z_i)^2} \quad (3.1)$$

Similarly, the time required to reach destination becomes

$$t = \frac{60d}{f} \quad (3.2)$$

Therefore, the number of commands to be generated along this linear trajectory can be calculated as

$$N = floor\left(\frac{t}{T}\right) \quad (3.3)$$

where T is the sampling rate of the control unit. The increments at each axis becomes

$$\Delta x = \frac{x_f - x_i}{N} \quad (3.4a)$$

$$\Delta y = \frac{y_f - y_i}{N} \quad (3.4b)$$

$$\Delta z = \frac{z_f - z_i}{N} \quad (3.4b)$$

Similarly, the coordinates of the tool at a particular time (kT) can be expressed as

$$x(k) = x_i + k\Delta x \quad (3.5a)$$

$$y(k) = y_i + k\Delta y \quad (3.5b)$$

$$z(k) = z_i + k\Delta z \quad (3.5c)$$

3.1.2 Rapid Motion

The rapid motion is utilized to move the tool from one point to another at the maximum speed. The difference between the linear motion is that feedrate is not required in rapid motion. Rapid motion is designated as `G0 Xxf Yyf Zzf`. Unlike point-to-point motion in formal `G0` (of RS-274D); here, all axes work in harmonization and tool moves the same amount in each axis as illustrated in Figure 3.1. The difference is that feedrate is selected automatically as the maximum feedrate (fmax) which can be achieved by the mechanism.

3.1.3 Circular Motion

The last interpolator motion, circular motion, differs by means of algorithm and definition. User has to define the working plane since circle should lie on a plane and also it is important to define the direction of rotation in order to draw the correct section of the circle. The block of the NC code consists of the information of the working plane, direction of rotation, the coordinates of the destination, the radius or the coordinates of the circle and the feedrate. There are two alternative definitions for circular motion. The representation options are presented in Table 3.1. In this table, X, Y and Z are the coordinates of the destination point, R is the radius of the circle and I, J, K are the distance from the center to the starting point. I, J, K values are used as a set of two keywords. I and J keywords are used for defining the center of the arc in XY Plane. Similarly, J, K is used for arcs in YZ plane and I, K are used for arcs in ZX plane. The sign of radius should be given different if the arc is designated with R. Radius should be designated as –R for the arc angles larger than 180°, as shown in Figure 3.2 with motion from A to D. Radius should be defined as R for smaller arc angles like motion from A to B. For the other cases, the incremental distance from center to starting point should be given as shown in Figure 3.2. In this case, the incremental distance on X-axis is identified as i, the incremental distance on Y axis is identified as j and the distance on Z axis is defined as k.

Table 3.1 Circular motion representations.

XY Plane	$G17\ G\begin{Bmatrix}2\\3\end{Bmatrix}\ Xx\ Yy\ \begin{Bmatrix}Rr\\Ii\ Jj\end{Bmatrix}\ Ff$
ZX Plane	$G18\ G\begin{Bmatrix}2\\3\end{Bmatrix}\ Xx\ Zz\ \begin{Bmatrix}Rr\\Ii\ Kk\end{Bmatrix}\ Ff$
YZ Plane	$G19\ G\begin{Bmatrix}2\\3\end{Bmatrix}\ Yy\ Zz\ \begin{Bmatrix}Rr\\Jj\ Kk\end{Bmatrix}\ Ff$
Complete Circle	$G\begin{Bmatrix}2\\3\end{Bmatrix}\ \begin{Bmatrix}G17\ Ii\ Jj\\G18\ Ii\ Kk\\G19\ Jj\ Kk\end{Bmatrix}\ Ff$

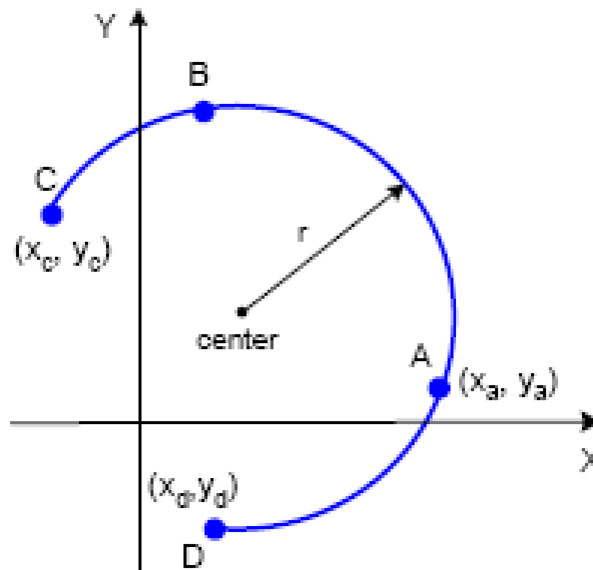


Figure 3.2 Circular Motion [38]

The last important definition is the working plane. `G17`, `G18` and `G19` are used for determining the working plane and they are used for motion on XY plane, ZX plane and YZ plane respectively.

The circulars motion calculations differ from the linear one by definition. If the parameter incremental distance to the center is known, radius of the circle can be calculated by Eqn. (3.5) and center coordinates are computed by Eqn. (3.6). If only the parameter radius is known while the center coordinates are not given, the center

position is achieved by geometric operations. If the center coordinates and radius of the circle is given, the angle of the starting point of the arc, θs, and the angle of the final point of the arc angle θf, is calculated by Eqn. (3.7a) respectively for XY, ZX and YZ planes.

$$r = \sqrt{i^2 + j^2 + k^2} \tag{3.5}$$

$$x_c = x_s + i \quad y_c = y_s + j \quad z_c = z_s + k \tag{3.6}$$

$$\theta_s = \operatorname{atan}\left(\frac{x_s - x_c}{y_s - y_c}\right), \qquad \theta_f = \operatorname{atan}\left(\frac{x_f - x_c}{y_f - y_c}\right) \tag{3.7a}$$

$$\theta_s = \operatorname{atan}\left(\frac{x_s - x_c}{z_s - z_c}\right), \qquad \theta_f = \operatorname{atan}\left(\frac{x_f - x_c}{z_f - z_c}\right) \tag{3.7b}$$

$$\theta_s = \operatorname{atan}\left(\frac{y_s - y_c}{z_s - z_c}\right), \qquad \theta_f = \operatorname{atan}\left(\frac{y_f - y_c}{z_f - z_c}\right) \tag{3.7c}$$

Position commands can be generated after all of the unknowns are computed. Firstly, the travel distance should be calculated to carry out this computation. The travel distance is the length of the arc. To compute the arc length, the sweep angle of the arc should be computed by Eqn. (3.8). Then the arc length, l, is found by Eqn. (3.9).

$$\theta_t = \theta_f - \theta_s \tag{3.8}$$

$$l = \theta_t \cdot r \tag{3.9}$$

Similarly, time required to complete the arc becomes

$$t = \frac{60 \cdot l}{f} \tag{3.10}$$

Therefore, the number of commands to be generated along this circular trajectory can be calculated as

$$N = floor\left(\frac{t}{T}\right) \tag{3.11}$$

where T is the sampling rate of the control unit. Since the trajectory is circular, increments should be projected into angles and the angular increments can be found by

$$\Delta\theta = \frac{\theta_t}{N} \tag{3.12}$$

Similarly, the angle values at a particular time (kT) can be expressed as

$$\theta(k) = \theta_s + \Delta\theta \cdot k \tag{3.13}$$

Finally, the coordinates of the tool at a particular time (kT) can be expressed as in Eqn. (3.14) which shows the procedures for circular motions in XY, ZX and YZ planes respectively.

$$x(k) = x(0) + r * [\cos(\theta(k))] \quad y(k) = y(0) + r * [\sin(\theta(k))] \tag{3.14a}$$

$$x(k) = x(0) + r * [\cos(\theta(k))] \quad z(k) = z(0) + r * [\sin(\theta(k))] \tag{3.14b}$$

$$y(k) = y(0) + r * [\cos(\theta(k))] \quad z(k) = z(0) + r * [\sin(\theta(k))] \tag{3.14c}$$

CHAPTER 4

TURNING SIMULATION

4.1 Introduction

Turning simulation feature of the program aims to simulate a turning operation with the given parameters. For that purpose, a specially designed graphical user interface (GUI) is used to input parameter and data input and to observe the simulation output.

In the turning simulation, it is appropriate to consider workpiece (WP) as a cylindrical part. Length and diameter of the workpiece are given to define workpiece geometry in the cutting operation. For the turning simulation, two separate techniques are applied. The first one depends on an approach which will be based on pixel operations of the workpiece and cutter is considered as matrices with Boolean values which will be generated as a result of investigation of pixel values of their geometries on the screen. Subtraction operation is then conducted.

Due to simplicity of geometric features in turning operation, both workpiece and cutting tool could be considered as polygons. The second approach is derived from this idea. Polygon clipping technique is applied to obtain desired geometry throughout the simulation. The result of that interaction will dynamically affect the workpiece geometry and will be used for the generation of 3D model of the workpiece.

4.2 Workpiece and Cutting Tool Definition

In this part, cutting and workpiece tool data structure is mentioned in detail. Since both pixel-based and polygon based data are used, definitions are made separately.

4.2.1 Pixel based technique

In pixel-based technique, workpiece and cutting tool (CT) are modeled as a pixilated 2D region. In the following section workpiece data structure is discussed in detail. Later cutting tool data structure is explained which is similar to workpiece's structure.

4.2.1.1 Workpiece Data Structure in pixel based technique

A geometric model is discretized by dividing it into finite number of elements. Thus, a fully continuous geometry is represented by a piecewise continuous geometry by a finite number of nodal quantities. This approach can be effectively used for turning simulation, and provides considerably fast simulation speeds.

First, a cylindrical workpiece is symmetric in the axial direction. If a cut of such a cylinder from its axis is taken, the resultant geometry will look like a rectangle. Figure 4.1 illustrates the idea to obtain that rectangle.

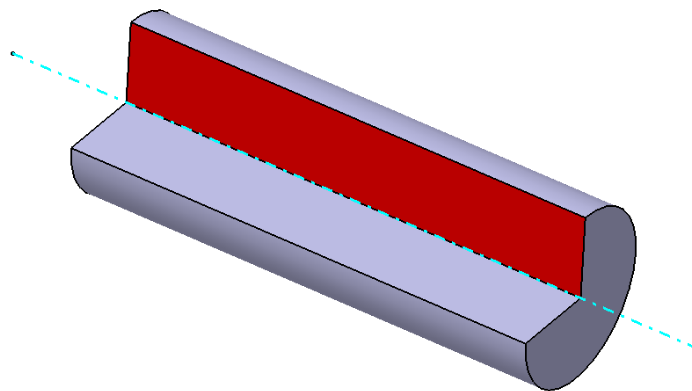


Figure 4. 1 Taking a cross-section of a cylindrical workpiece.

Because the part is axi-symmetric, by utilizing this feature, the part is cut across its axis, and only the section, shown in red, is used for calculations. If our resolution is limited by the screen resolution, it is sufficient to discretize the workpiece section according to that resolution. That is, the ultimate resolution in the screen, is the resolution it generates. This is the resolution of the screen device. If a shape is discretized to the screen resolution, the size of a single pixel is going to give us the minimum meaningful change in the process. Therefore, the simulator generates cutter movements according to that principle. As can be seen in Figure 4.2, the rectangular cross section is filled with some equally spaced black dots, which represents the dimensions of a pixel on the screen.

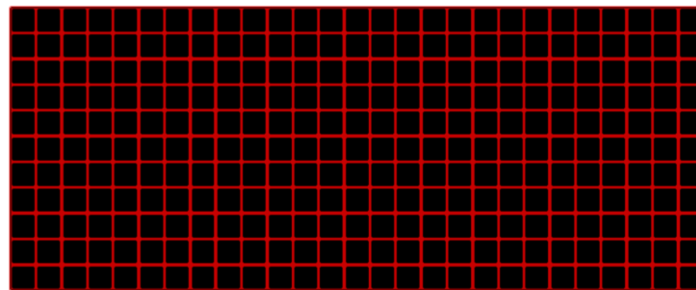


Figure 4. 2 Discretization of the workpiece cross-section.

The pixelized cross-section of the workpiece is somewhat similar to a Boolean matrix, in which Boolean value indicate pixels with black dots, and false values denote empty pixels. The data structure of the workpiece can be considered as a two dimensional matrix with Boolean values. The construction of workpiece Boolean matrix is shown in Figure 4.3. Since the data structure of the workpiece is a Boolean matrix, the cutting tool data must be in the same format to apply appropriate Boolean operations for the purpose of subtracting the cutting tool from the workpiece.

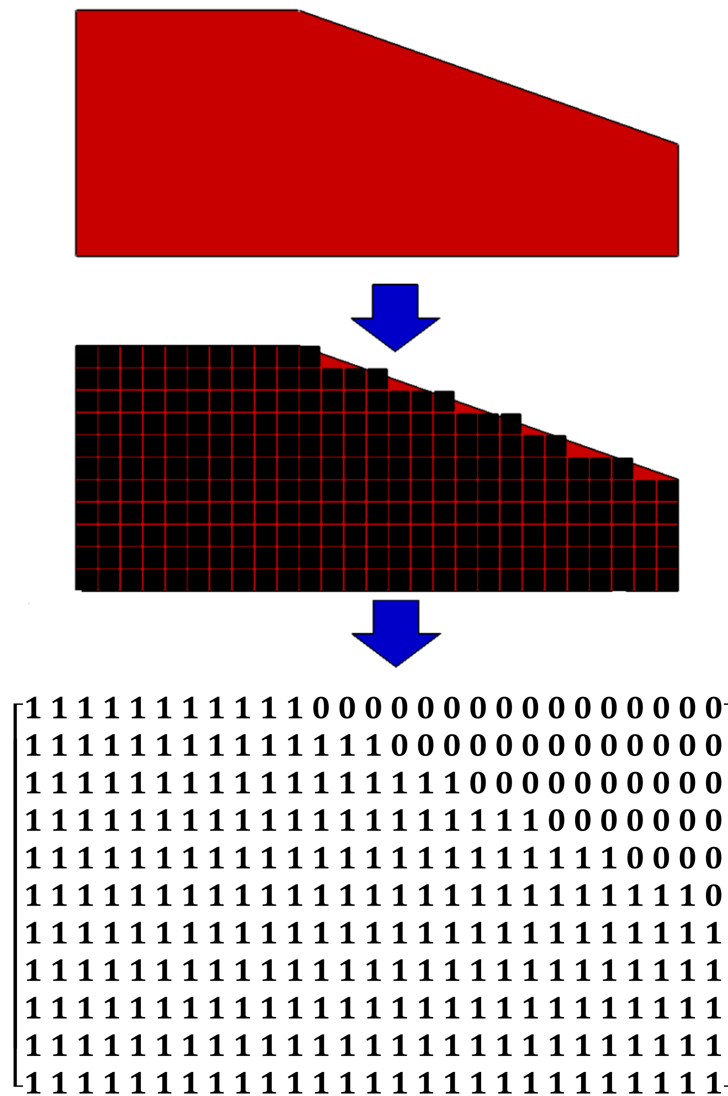


Figure 4. 3 Converting workpiece into pixel matrix, then converting this pixel data, into a Boolean matrix.

4.3.1.2 Cutting Tool Data Structure in pixel based technique

For the definition of cutting tool, a novel approach is proposed in this thesis. To make the input conversion as simple as possible, the cutting tool is also represented as a Boolean matrix. In fact, the cutting tool input file is a bitmap (bmp) file. As mentioned before, bmp file is a file, which directly holds the coordinates and color values of pixels in a picture. If a cutting tool is drawn as a bmp file, it can directly

be taken as an input to the program and can be converted into a matrix of Boolean type. For this purpose, a Bitmap (bmp) object is created in the program and pixel values are read. A matrix having the dimensions of the bmp width and height values, is produced. Then black colored pixels are considered as true values in the matrix while white colored pixels are regarded as false. As in the workpiece case, any line in the matrix is converted into `BitArray` data types to keep things simpler. After that, both of the inputs of the program are in the format of `BitArray` Vectors. It is easy to update the status of the workpiece during the cutting operation, as explained in the next section.

4.3.2 Polygon Clipping technique

To apply polygon clipping technique in machining simulation first it is necessary to define workpiece and cutting tool as polygons. In the following sections this procedure is explained in detail.

4.3.2.1 Workpiece Data Structure in polygon clipping technique

In this case, both the workpiece and cutting tool are represented as polygons. These polygons characterize their boundaries. A polygon is defined as an ordered list of vertices in our case. The number of vertices determines the number of edges on the polygon.

For the workpiece, the geometry starts out as a simple rectangle with four vertices. In the developed software, the workpiece dimensions are taken as vertex coordinates. The length input is used to define the width of the rectangle, where the diameter input defines height of the rectangle. Thus, a rectangular polygon is created with the vertex parameters as shown in Figure 4.4.

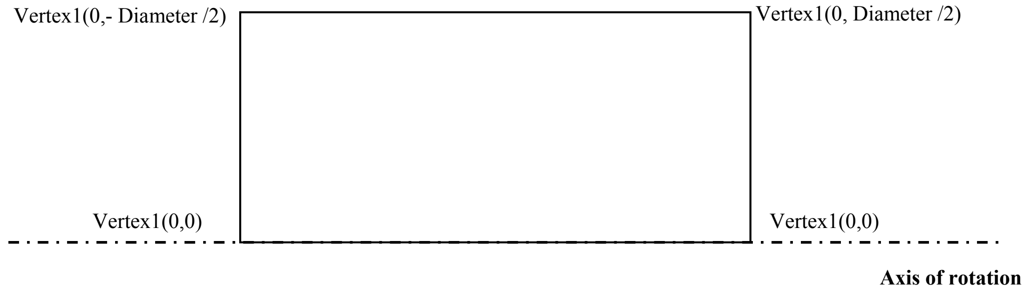


Figure 4. 4 Workpiece geometry

As shown in the Figure 4.4, vertices are defined with half of the diameter There is a symmetry about the axis of rotation. That feature, allows to make the calculations for one the half. Note that since the turning operation is applied mostly for cylindrical parts, the resultant vertex values could be rotated around the axis to get the final object. This will be explained later in the section, where 3D object generation is discussed. In the developed software, a special data structure is generated for the workpiece which keeps some specific geometric features of the workpiece. Table 4.1 shows that corresponding data structure.

Table 4. 1 Workpiece data structure

```
struct Workpiece
{
    double Length;
    double Diameter;
    PointD[] PointDArray;
    PointF[] PointFArray;
    PointF[] ClampedZoneF;
    PointD[] ClampedZoneD;
}
```

In the data structure of the workpiece, length and diameter values are in double precision. Due to the nature of the polygon clipping technique, the geometric calculations (like intersection of two line segments) are calculated frequently. In C# and DirectX, default Point, and Vector data types are composed of single precision coordinate values. Because of this lack of precision; specific Point and Vector data types are generated in the program. `PointD`, `Vector2D`, and `Vector3D` value types are developed for this purpose to coordinate values, that are in double precision.

`PointDArray` variable holds vertex data of the workpiece. `PointFArray` variable is used to hold single point precision equivalents, which has to be generated for the visualization. As mentioned before, DirectX and C# graphics design interface (GDI+) work with single precision floating point values. That is why, the `PointDArray` items are converted into `PointF` values and they are stored in `PointFArray` of the workpiece structure.

`ClampedZoneD`, and `ClampedZoneF` members of the workpiece data structure define the portion of the workpiece which has to be away from the machined part. In the simulation, if there is a cutter interaction with clamped zone, a warning is issued to the user. Since it implies a crash of the cutting tool with the chuck in the real cutting operation.

4.3.2.2 Cutting Tool Data Structure polygon clipping technique

Cutting tool is also modeled as a polygon. The cutting tool geometry is defined by the data retrieved from a library of common cutting tool geometries used in turning operations. The interpreted NC code includes the tool number and the related cutting tool geometry is called by the software from the library.

Actually cutting tool geometry is defined by a text file. This text file has an extension, which is the tool number. This notation is recognized by the software. In the text file the vertices of the cutting tool are given as Z and X coordinate pairs. A sample cutting tool data is shown in Table 4.2.

Table 4. 2 Format of cutting tool vertex data file.

$Vertex_{1,Z}$ $Vertex_{1,X}$ $Vertex_{2,Z}$ $Vertex_{2,X}$ . . . . $Vertex_{n,Z}$ $Vertex_{n,X}$

As can be seen in Table 4.2, the coordinate pairs define vertices of the polygon, which determines the boundary. As shown in Figure 4.5, in the CNCSimulator there is a tool editor, in which the tool geometries can be edited.

The cutting tool data structure is shown in Table 4.3. It is very similar to workpiece data structure. The only difference is that instead of length and diameter parameters, the width and height of the tool's bounding box is kept as double precision members. These values are used in the calculations for the determination of screen output in the 2D viewport during the simulation. As will be mentioned later, even the workpiece and cutting parameters and dimensions change, the 2D viewport adjusts the scaling dynamically to provide best visibility.

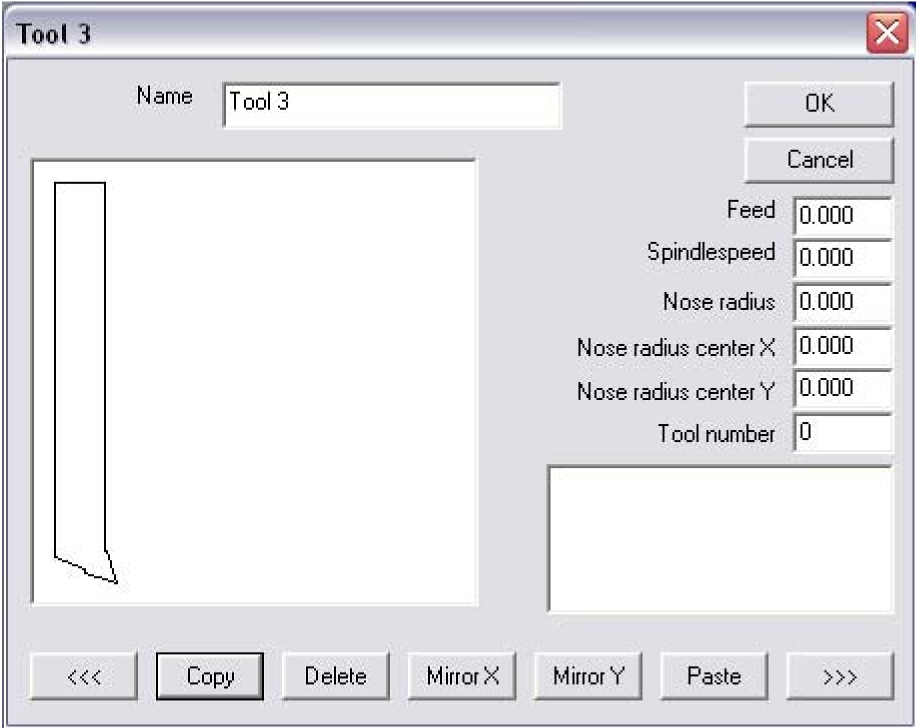


Figure 4. 5 Tool editor of the CNCSimulator

Table 4. 3 Cutting tool data structure.

```
struct CuttingTool
{
   List<PointD> PointDList;
   PointF[] PointFArray;
```

Table 4.3 (continued)

```
  double BoundingBoxWidth;
   double BoundingBoxHeight;
}
```

4.4 Cutting Methodology

Cutting methodology for pixel-based technique and polygon-based technique are completely different. In the next section pixel-based technique is explained. In the following section polygon-based technique will be discussed in detail.

4.4.1 Cutting methodology in pixel-based technique

As we have both workpiece and cutting tool have Boolean matrices format, one can apply logical operations to generate updated workpiece data. An example of this simple logic is shown in Table 4.4.

Note that in C# there is a data type, which is called BitArray. With the usage of BitArray data type, it becomes much more simpler and faster to make Boolean operations in a program. A BitArray manages a compact array of bit values which are represented as Booleans where true indicates that the bit is on (1) and false indicates the bit is off (0) [17].

To be able to use BitArrays as data storage type in the program, it is necessary to convert Boolean matrices into a 1-dimensional vector of BitArrays. Then one can apply special C# BitArray functions, logical “AND” and logical “NOT” on the `BitArrays` to make Boolean operations easier.

Table 4. 4 Definition of workpiece and cutter matrices and modification of workpiece matrix with the application of Boolean operation.

$$Wp = \begin{bmatrix} 1 & 1 & \cdots 1 & 1 & 1 \\ 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & \cdots & 1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \; Tool = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$Wp = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & \cdots & 1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \; (XOR) \;\; Tool\;(in\,Wp\;space) = \begin{bmatrix} 0 & 0 & \cdots & 1 & 1 \\ 0 & 0 & \cdots & 1 & 1 \\ 0 & 0 & \cdots & 1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

$$Wp = \begin{bmatrix} 1 & 1 & \cdots & 0 & 0 \\ 1 & 1 & \cdots & 0 & 0 \\ 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

Another requirement is that the workpiece position through the cutting operation is intact but it will only be affected by the cutting tool movements. On the other hand, the cutting insert will keep its by geometry all the time, provided that no tool wear is observed. Hence, only the position of the tool will change in time. Thus in the program, the tool matrix is displaced according to the position registered.

Another important feature of the developed program is that positional increments are as small as the pixel size on the screen. Thus, any cutter movement extracted from the cutter location (CL) data file is divided into portions. The size of these portions has to be same as the size of a pixel. To do so, if say a movement is composed of X and Z axis components, in the cutting space coordinates shown in Figure 4.6, in the case where the amount of movement component on one of the axis is more than the other component, the bigger amount is considered to find the amount of increment. In other words, if in one step X component of the move vector is 5 mm and Z component is 30 mm, then the both X and Z components are divided into 30 to get at most 1 mm increments. Of course, this amount is correct, if we have the pixel size as much as 1 mm. The program recursively applies the subtraction of cutter from the workpiece to simulate the turning operation.

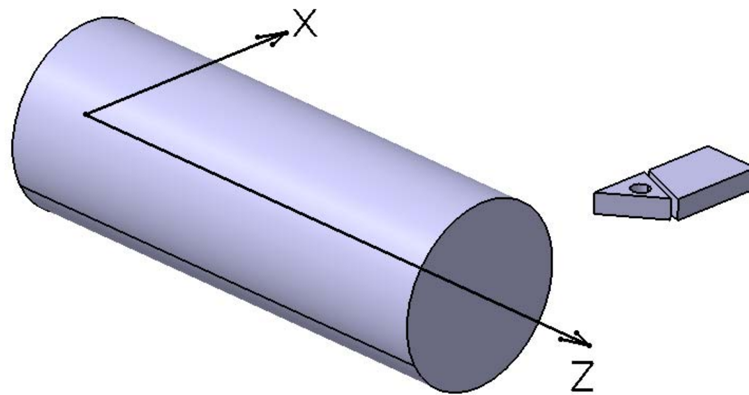


Figure 4. 6 Axes in turning operation.

4.4.2 Polygon clipping technique and its application in the simulation

Due to nature of turning operation, workpiece and CTPs as generated before are candidates to be applied some Boolean operations on them to simulate the machining operation. In the following section, Polygon Clipping operations and their application in the case of cutting simulation will be discussed in detail.

4.4.2.1 Introduction

First of all, it would be appropriate to explain the main idea behind performing the simulation with only polygons. At the beginning there exist two polygons, which are defined in terms of vertex values. The polygon, which represents cross-section of a rotating workpiece, is considered as stationary object in the simulation. Because there is no benefit of that rotation for neither simulation quality nor the calculation complexity. The only change occurring in the vertex data during the simulation is due to the clipping at workpiece polygon (WPP) with cutting tool polygon (CTP).

On the other hand, the CTP is not a stationary polygon. It replicates the interpreted movements defined in the NC code. Thus, the coordinate values of the cutting tool

is transformed continuously during the simulation as a whole. However, unlike WPP, the relative positions of the coordinate values of the CTP kept fixed. They are only subjected to the same transformation.

The simulation is conducted first in 2D with the above mentioned polygons and then the obtained data is transformed into 3D. The polygon clipping lies at the heart of the technique. The CTP is swept in ZX plane and hence the resultant swept polygon is used to clip WPP.

In this section, first the polygon clipping technique will be explained. How the CTP is swept, will be discussed. At the end, application of this operations into the turning simulation will be elaborated.

4.4.2.2 Polygon Clipping

Before discussing cutting methodology of the developed software, it would be better to give some insights about polygon clipping technique. As mentioned in the literature survey section, polygon clipping techniques have been utilized in computer graphics applications since the emergence of the computer graphics technology. The method considered in this thesis is developed by Martinez, et. al. [21].

4.4.2.2.1 Basics

A common way to represent a polygon is to list its vertices in counter-clockwise order: v_0; v_1; v_2; . . . ; v_n. The ordered list of edges v_0v_1; v_1v_2; . . . ; v_nv_0 defines the boundary of the polygon.

Depending on the type of operation result of a Boolean operation on two polygons is composed of those portions of the boundary for each polygon that lie (or do not lie) inside the other polygon. For example, Fig. 4.7 shows the results of different

Boolean operations on two polygons called P and Q. Therefore, the computation of a Boolean operation could be considered as finding these intersecting portions. Once found, they must be connected to form the resultant polygon.

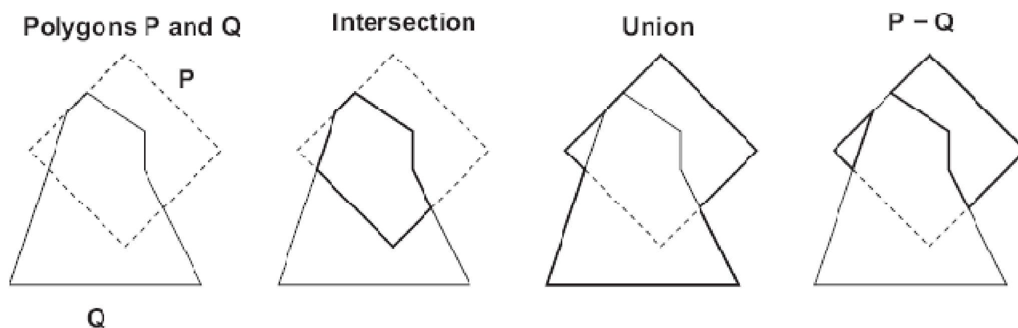


Figure 4. 7 Boolean operations on polygons [21]

Suppose that the edges of two polygons are divided into line portions at their intersection points as shown in Fig. 4.8. In that case, the boundaries of the polygons intersect at endpoints of those newly generated line portions. The problem of computing the resultant boundary of a Boolean operation on the polygons finding those related edges of each polygon that lie (or do not lie, which depends on the executed Boolean operation) inside the other polygon. At the end, once these edges are found, they have to be connected to generate the resultant polygon. The approach for computing Boolean operations on polygons can be described as follows:

(1) First, subdivide the edges of the polygons at the points where they intersect each other.

(2) Select those subdivided line portions that lie inside the other polygon (or that do not lie, which is determined according to the type of the Boolean operation)

(3) Join the edges, selected in Step 2 and sort them in an appropriate order to form the resultant polygon.

To apply the prescribed algorithm above, the plane sweep technique is used to efficiently to implement this approach.

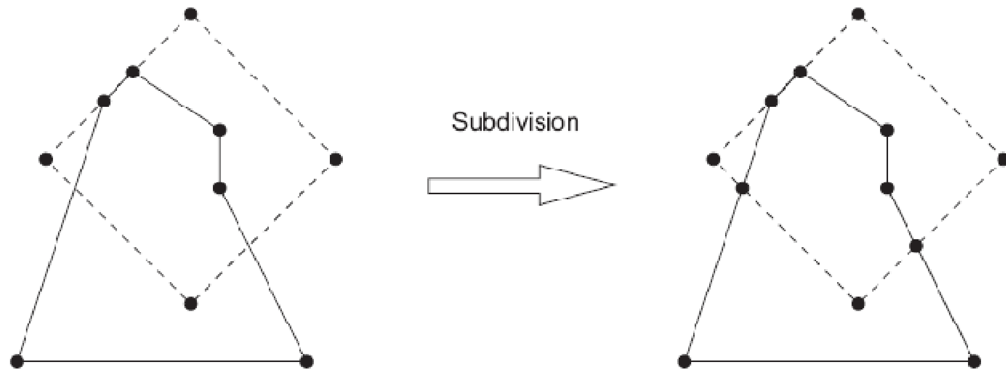


Figure 4. 8 Subdivision of edges of polygons at their intersection points [21]

4.4.2.2.2 The algorithm

In this section, the algorithm for computing Boolean operations on polygons, is described. To be able to subdivide the edges of the polygons, it is necessary to find their intersection points. This task can be efficiently carried out by using the following principle: consider that the plane is swept with an infinite length vertical line. At every moment, the edges that intersect with the vertical sweep-line are ordered from bottom to top and are stored in a data structure, which is called *S*. Then, it can be shown that:

(1) The status of S does not change, unless it reaches to an endpoint or an intersection point of the edges.

(2) Intersection between two edges can occur, if and only if these edges are adjacent along S. The classical algorithm for computing the intersection points between a set of segments is based on this principle given by Preparata and Shamos [22].

The algorithm used in the developed software also employs this approach efficiently to find the intersection points between the edges of the polygons. Furthermore, the information available during the plane sweep is facilitated to subdivide the edges and decide on the edges to be included in the result of the Boolean operation.

To elaborate this algorithm; a vertical line is used to sweep the plane from left to right. The sweep-line status, *S*, is composed of the ordered sequence for the edges of both polygons intersecting the vertical sweep-line. As mentioned before, *S* will only change at the end points of the edges:

- When the left end point of an edge is reached to the edge must be added to *S,* and it must take its proper location on the list.
- When the right end point is reached the edge must be removed from *S*, and S must be reordered to reflect necessary change in it.

In this thought process, every edge of polygons is composed of endpoints, which are called events. They are called events due to the fact that they cause a change in the *S*. To store those events, the event-point set is formed by the endpoints of the edges on the polygons. This set changes dynamically as when an edge is subdivided, two new endpoints appear. That is why, in the developed software both for *S* and for event-point set (called *Q*) are List variables. Unlike arrays, number of elements of a list can be dynamically changed during runtime. The elements of a list can be removed, or a new element can be inserted at any moment. Also, it is possible to copy the elements of a list into an array, if necessary. The algorithm implements the event-point set using a priority queue which stores the endpoints sorted from left to right.

The algorithm to compute Boolean operations on two polygons is shown in Table 4.5. To facilitate the object oriented programming (OOP) features of C# language, a specific class is created to define a polygon clipping object. It has several functions and data members. Some data types like structures and some enumerations are defined for the sake of simplicity. This class is called as `ClippingProcessorClass`. Once an instance of that class is created its functions are used to operate on it. To apply a Boolean operation on two polygons, the class instance object awaits to be called by its `fncMain` function. This function takes three parameters, two lists of vertices and Boolean operation type. Then clipping function is called internally.

Before going into details of the algorithm on clipping function, an important data structure, which is an event is necessary. In `ClippingProcessorClass`, any edge and endpoint of a polygon is defined by a structure called `SweepEvent`. `SweepEvent` structure and its members are shown in Table 4.6. Three important enumerations are `PolygonType`, `EdgeType`, and `BooleanOpType`. Their members are shown in Table 4.7.

The Boolean operation takes place in that clipping function. First of all, the priority queue is filled with the endpoints of the edges. After that these endpoints, which will be called as events there after that point, are sorted with respect to their X coordinate values. Thus, the events are processed (from left to right) as described. When an event, which is the left endpoint of an edge, is found, it is inserted into the sweep line status *(S)*. It is further processed if the associated edge resides in the other polygon. Details are explained in section 4.4.2.2.3. Consequently, the event is removed from the priority queue *Q*. Whether any intersection occurs with its neighbors along *S* must also be processed. For that purpose, `fncFindIntersectionOf2LineSegments` function is run. If any intersection is found, the function returns the number of intersections. If no intersections are found, function returns null. Note that it is also necessary to check whether the adjacent edges are overlapping. Overlapping edges are special cases and they must be dealt with. Details on how overlapping edges are dealt will be given in section 4.4.2.2.6.2. When a right endpoint is found, both the event itself is removed from the *Q*, and its associated edge is removed from *S*. Now, its two neighbors along *S* become adjacent and they need to be tested for possible intersection. Finally, the removed edge is considered for inclusion in the result of the Boolean operation.

As mentioned before, the procedure `fncFindIntersectionOf2LineSegments` is used to detect a possible intersection between two edges. In the case where the edges belong to the same polygon or they only intersect at only one of their endpoints no extra processing is required. Otherwise, if the edges belong to different polygons and they intersect at least at one point then they have to be

subdivided into segments. When an edge is subdivided, the data structures *Q* and *S* are need to be updated to reflect the new status. This is done with the procedure called `fncIntersectionProcessor`. Fig. 4.9 shows the types of intersections that may occur between two line segments. An intersection (depending on the kind) leads to a subdivision of the edge. In the procedure `fncFindIntersectionOf2LineSegments`, the intersection routine described in Schneider and Eberly [23] for detecting a possible intersection between two edges is used.

At this point, it would be beneficial to give an example on, how an edge is subdivided, and how it is reflected in *S*. As shown in Fig. 4.10, when the vertical sweep-line reaches to the point p_1, *S* is composed of edges (q_2q_3, q_1q_2). Then, the left endpoint of p_1p_2 is processed, and p_1p_2 is located in *S* $(S = \{q_2q_3, q_1q_2, p_1p_2\})$. p_1p_2 intersects with its neighbor q_1q_2 at point *i*, so that p_1p_2 and q_1q_2 must be subdivided into edges p_1i, ip_2, q_1i and iq_2. Similarly *Q* has to be updated to include the endpoints of these newly created edges. *S* will also change to $S = \{q_2q_3, iq_2, p_1i\}$. After that, it is time to process the left endpoint of p_0p_1, and p_0p_1 is inserted into *S* where $(S = \{q_2q_3, iq_2, p_1i, p_0p_1\})$. In the next section, the methodology on how an edge is selected to be included in the result polygon will be discussed.

Table 4. 5 Algorithm for Boolean operations.

```
object[] fncMain(PointD[] subjectPolygon, PointD[] clipPolygon,
        BooleanOpType operationType)
//Input:    A point array for subject polygon, a point array for
//          clipping polygon, type of boolean operation
//Return:   An object array, includes a list of points and a list
//          of point arrays.
{
      Set subject polygon and clipping polygon with sweepevent
      data types;
      //Call fncClippingFucntion to obtain returning values
      return fncClippingFunction(operationType);
}
//Input:    Boolean operation type.
//Return:   An object Array, includes a list of points and a list
//          of point arrays.
object[] fncClippingFunction(BooleanOpType operationType)
{
```

Table 4.5 (continued)

```
//Insert the endpoints of the edges of polygons into priority
//queue Q
Q.Events.Add (SubjectPoly);
Q.Events.Add (ClippingPoly);
Q.Sort();
//Loop as long as priority queue is not empty
while (Q.Count > 0)
{
      //Get the first event in the queue
      event = Q.Events[0];
      if(event.left_endpoint)
      {
            //Add the event to the Sweep Line Status
            S.Add(event);
            S.Sort();
            pos = S.IndexOf(event);
            fncSetInsideOtherPolygonFlag(event, pos);
            If(event intersects with previous or next event
            in S)
            {
                  Subdivide event and previous event;
            }
            else if (event and previous or next event in S
            overlaps)
            {
                  Change edge type parameters of events;
            }
      }
      else
      {
            pos = S.IndexOf(event.other);
            Q.Remove(event);
            event = S.Events(pos);
            switch(operationType)
            {
                  case Intersection:
                        if(event.Type == EventType.NORMAL &&
                           event.InsideOtherPoly == true)) ||
                           event.Type == Type.SAME_TRANSITION)
                              result.Add(event);
                  break;
                  case Union:
                        if((event.Type == EventType.NORMAL &&
                           event.InsideOtherPoly == false)) ||
                           event.Type == Type.SAME_TRANSITION)
                              result.Add(event);
                  break;
                  case Subtraction:
                        if(event.Type == Type.DIFF_TRANSITION)
```

Table 4.5 (continued)

```
                                    result.Add(event);
                              else if(event.Type ==
                                       EventType.NORMAL &&
                                      ((event.PolygonType ==
                                        PolygonType.SUBJECT &&
                                        event.InsideOtherPoly ==
                                        false) ||
                                       (event.PolygonType ==
                                        PolygonType.CLIPPING &&
                                        event.InsideOtherPoly ==
                                        true)))
                                    result.Add(event);
                  break;
            }
            S.Remove(pos);
            next = S.next(pos);
            prev = S.prev(pos);
            If(prev and next event intersects)
            {
                  Subdivide prev and next event;
            }
            else if (prev and next overlaps)
            {
                  Change edge type parameters of events;
            }
      }
   }
   Sort result data;
   return result;
}
```

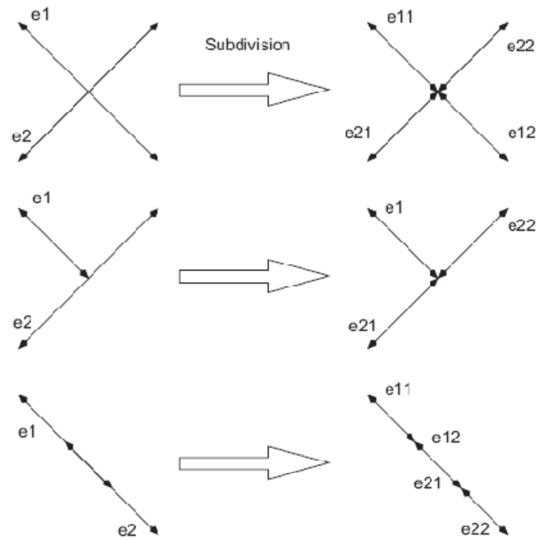


Figure 4. 9 Types of intersection [21]

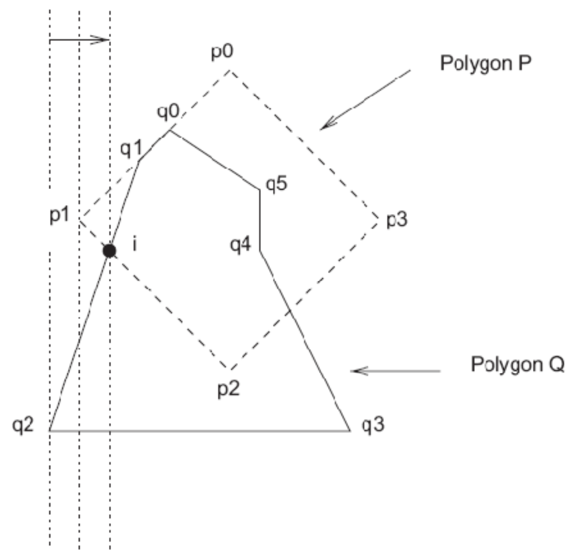


Figure 4. 10 Sweep-line technique [21]

4.4.2.2.3 Selecting the result edges

When the sweep-line reaches to the right endpoint of an edge (let's call it *e*) the algorithm is requested to decide whether *e* belongs to the result of the Boolean operation. The crucial thing at that point is to determine whether *e* lies inside the other polygon or not. In the `fncClippingFunction`, one computes if *e* lies inside

the other polygon; when *e* is first inserted into *S*. This computation is done by reading the three flags of information from the edge that precedes *e* in *S*. In Table 4.8 the algorithm of this procedure `fncSetInsideFlag` procedure is presented. Those three flags from the preceding edge that has to be read are as follows;

- `pl`: indicates whether the edge belongs to the subject or clipping polygon.
- `inOut`: indicates the inside/outside behavior of the polygon that the event belongs. For that purpose, some if statements are checked as:
    - If the previous event in *S* belongs to the same polygon, the `inOut` flag of the current event is set negation of the `inOut` flag of the previous event.
    - If the previous event is *S* belongs to the other polygon, the `inOut` flag of the current event is set equal to the inside flag of the previous event.
- `inside`: indicates if the edge is inside the other polygon. To determine this flag, once again some if conditions are checked as follows:
    - If the previous event in S belongs to the same polygon, the inside flag is set equal to the inside flag of the previous ones.
    - If the previous event in S, belongs to the other polygon, the inside flag of the current event is set negation of the `inOut` flag of the previous event.

However, in the case of both overlapping and vertical edges, there is a special case, which does not let the above procedure to yield correct result. Owing to the fact that a vertical edge does not imply any `inOut` relation for the polygon it belongs. Also in the case of an overlapping edge, whose `EdgeType` parameter must be else than `NORMAL`, there is possibility to set its index randomly because there is no way to put two edges in an order in neither *Q* nor in *S*. This ambiguity requires to be dealt with a more tedious, but achievable way. In that case, all the events, which are non-vertical, in the *S* up to current event, are counted to determine the number of events belonging to the subject and to the clipping polygons. If the number of events with the same polygon type is odd, the `inOut` flag is set to false. If the number of events with the different polygon type is odd, the inside flag is set to true.

Table 4.7 shows the algorithm of the routine, `fncSetInsideFlag`, that computes the `inOut` and `inside` flags of a left endpoint event call *le*. As mentioned before, `le` is compared with its immediate predecessor call `ple` in the *S*. If `ple` is null then *le* is simply the first event in S and it means that all the flags can be trivially set to false.

In the priority queue *Q*, events that share the same X coordinate, have to be processed from bottom to top to apply correctly the above mentioned routine. If two endpoints share the same point on the sweep plane, the right endpoints have to be processed before the left ones. If two left endpoints share the same point, It is necessary to check the position of the edges in the S, which has to be sorted in the ascending order.

Table 4. 6 *SweepEvent* structure.

```
public struct SweepEvent
{
      PointD p; //Coordinate of the point
      bool left; //Flag, whether the endpoint is the left one of
                 //the edge
      PointD other; //Coordinate of the other endpoint of the edge
      PolygonType Pl; //Polygon type of the edge
      bool inOut; //inOut flag of the edge
      bool inside; //inside flag of the edge
      EdgeType Type; //EdgeType of the edge. Possible values are
                     //NORMAL, SAME_TRANSITION,
                     //DIFFERENT_TRANSITION, and NON_CONTRIBUTING
      int ID; //a specific integer id number, which is used to find
              //the edge in the queues
      double M; //Slope of the edge (y=Mx+N)
      double N; //N in line equation (y=Mx+N)
      double Y; //Y coordinate value of the edge, in the curren x
                //coordinate of the vertical sweep line
      bool Vertical; //Flag, whether the edge is a vertical one.
}
```

Table 4. 7 *PolygonType*, *EdgeType*, and *BooleanOpType* enums.

```
enum PolygonType
{
      CLIPPING, //Clipping Polygon
      SUBJECT //Subject Polygon
```

Table 4.7 (Continued)

```
}

enum EdgeType
{
      NORMAL, //The edge type is normal
      NON_CONTRIBUTING, //The edge will not be considered in the
                        //result
      SAME_TRANSITION, //The edge has the same inOut flag with the
                      //edge with which it overlaps
      DIFFERENT_TRANSITION //The edge has opposite inOut flag of
                           //the edge with which it overlaps
}

enum BooleanOpType
{
      INTERSECTION, //Intersection type of boolean operation
      SUBTRACTION, //Subtraction type of boolean operation
      UNION //Union type of boolean operation
}
```

Table 4. 8 Routine to set *inside* and *inOut* flags of edges.

```
//Input     : A SweepEvent and the index number of event in S
void fncSetInsideFlag(SweepEvent event, int index)
{
      if (index == 0)
      {
            event.inside = false;
            event.inOut = false;
      }
      else
      {
            if (Previous event in S is vertical or EdgeType of
                previous event is not Normal)
            {
                  for each event in S up to index do
                  {
                        Count events with the same polygon type;
                  }
                  Set inside and inOut flags;
            }
            else
            {
                  if (event and previous event in S belong to the
                  same polygon)
                  {
                        event.inside = S.Events[index - 1].inside;
```

Table 4.8 (continued)

```
                        event.inOut = !S.Events[index - 1].inOut;
                  }
                  else
                  {
                        event.inside =!S.Events[index - 1].inOut;
                        event.inOut = S.Events[index - 1].inside;
                  }
            }
      }
}
```

4.4.2.2.4 Connecting the result edges to form the solution

The result of a Boolean operation on two polygons results in a set of polygons. There is possibility to have an empty set for the result in some cases. In the previous sections, it was described how to find the edges of the subject and clipping polygons. Now, it will be shown, how these edges can be sorted and connected to form the resultant polygons. For that purpose, a specific sorting algorithm is generated. `fncResultPolygonSort` procedure is utilized for that purpose. The algorithm is shown in Table 4.9. The difficulty of the problem comes from the nature of the sweep plane algorithm. To apply this algorithm, it is necessary to set the endpoint with smaller X coordinate value as left endpoint and the one with the higher X coordinate value as right endpoint. The natural result of this approach is that every edge is defined as if it is a vector directing to the right hand side at the plane. That situation does not lead to a closed polygon. For that reason, in the procedure, endpoints of the edges are checked for coincidence. Once a coincidence is observed, the endpoints of that edge are swapped. This process is repeated until the starting point of the polygon is reached.

There are some cases, result of clipping results more than one closed polygon. In such cases, procedure can suffer an infinite loop while trying to find a line segment to connect to the last one. This is considered as an indication of multiple cycles by

the procedure so that looping infinitely is prevented. A new cycle starts and the previous process is repeated, until all the edges are put into a polygon.

Table 4. 9 Sorting procedure to obtain connected vertices Polygon vertices.

```
//Input     : A reference SweepEvent list
//Return    : List of SweepEvent lists
List<List<SweepEvent>> fncResultPolySort(ref List<SweepEvent>
                                          incomingList)
        {
            //
            while (incomingList is not complete)
            {
                  If(check for cycles)
                        Set parameters for cycles;
                  for each event in the incomingList do
                  {
                        if (successive events are in order)
                              Add event to result;
                        else if(successive events having one
                        endpoint mating)
                              Swap endpoints of event;
                              Add event to result;

                  }
                  If(A chain is closed)
                        break while loop and start another chain;
            }

            Add each cycle to the result;
            return result;
        }
```

4.4.2.2.5 Performance analysis

In this section the performance of the presented algorithm is analyzed as described in [21]. The following notation will be adapted: let n be the total number of edges of the polygons involved in the Boolean operation and k be the number of intersections that take place in between the polygon edges. The algorithm starts out with inserting all the endpoints of the edges on priority queue Q, which takes $O(n\log(n))$. Then the

plane sweep starts and all the events are processed in the cycle from left to right. Let us analyze the cycle body:

- The lines, where *S* is manipulated with additions, removals, and where an event is searched in *S* are operations on *S*. *S* holds at most *n* edges and can be considered as a dictionary, so these lines take each $O(log(n))$.
- If the event is not a vertical one (or if it not an overlapping one) `fncSetInsideFlag` routine runs in time $O(log(n))$ since this is the time needed to determine the immediate predecessor of the event in *S*—the routine used to set the *inside* and `inOut` flags runs in time $O(1)$.
- The function `fncFindIntersectionOf2LineSegments` takes $O(log(n + k))$ time, because after an intersection test, which uses constant time, four insertions on *Q* can be done, and *Q* has an $O(n + k)$ size.
- Line, where an event is retrieved from *Q,* runs in constant time, and removing an event from *Q* takes $O(log(n + k))$.
- Finally, the inclusion of an edge in the result polygons takes $O(log(n))$.

Before implementing developed algorithm (i.e. application of Boolean operation on polygons) a test program is first developed. Algorithm and its performance is first tested in that small program and then the method is placed inside simulation program. As an example in Figures 4.11 - 4.14 show a sample case, where 2 considerably complex polygons with 94 and 54 vertices are examined with all three Boolean operations, (subtract, intersect, and union). Execution times of these three operations are given in Table 4.10.

Table 4. 10 Execution times of Boolean operations on the polygons shown in Figures 4.11-4.14. Polygon 1 is composed of 94 vertices, polygon 2 is composed of 54 vertices.

Boolean operation	Execution time (ms)
Subtraction	296.8
Intersection	281.2
Union	312.5

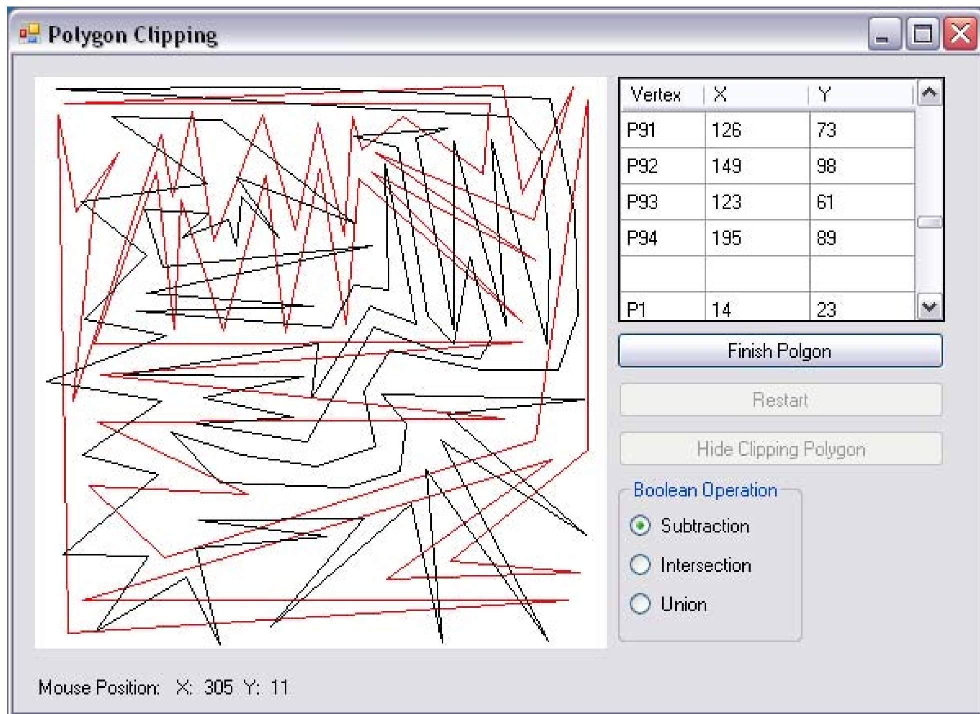


Figure 4. 11 User interface of simple polygon clipping program for test purposes. In the figure, the screen view of the program, with two arbitrary polygons, is shown. The black one is the subject polygon, with 94 vertices. The red polygon is the clipping polygon, with 54 vertices.

Figure 4. 12 Result of Boolean subtraction

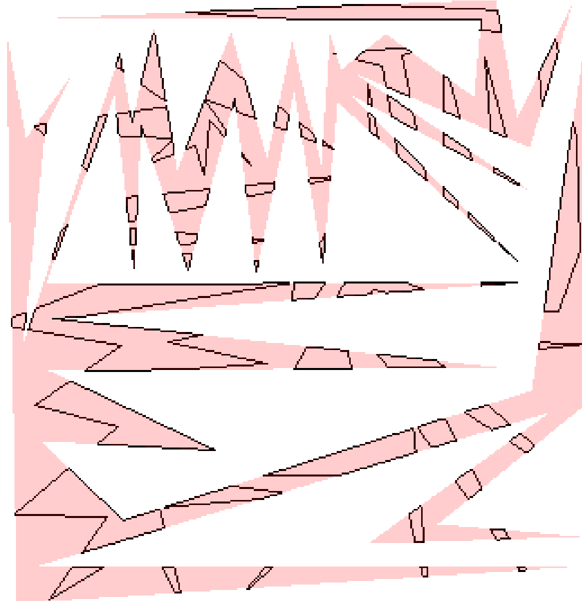


Figure 4. 13 Result of Boolean Intersection

Figure 4. 14 Result of Boolean Union

4.4.2.2.6 Special cases

In this section, special cases (and how they are treated) will be discussed. As will be shown they are treated in a simple, effective way.

4.4.2.2.6.1 Vertical edges

Vertical edges are special since their two endpoints are placed at the same X coordinate. This causes an ambiguity in the sorting algorithms. However, it is possible to process them in the algorithm as "normal edges" as long as the following rules applied:

(1) The lower endpoint of a vertical edge, has to be considered as its left endpoint and the upper endpoint as its right endpoint.
(2) To order the sweep-line status *(S)*, the lower endpoint of the edge must be considered as the one that intersects the sweep line. S is ordered by the Y coordinate at which edges intersect the sweep-line. Thus, a vertical edge takes its place in *S* according to its lower endpoint. If a non-vertical edge shares the same Y coordinate with a vertical edge, the non-vertical one is placed before the vertical one in *S*.

4.4.2.2.6.2 Overlapping edges

When two edges overlap, they are subdivided so that their overlapping portions become an edge of each polygon. This situation is illustrated in Figure 4.9 as the last type of intersection. It should be possible for the algorithm, to select at most one of these two "equal edges" while constructing the result of the Boolean operation. Unfortunately, the methods explained in section 4.4.2.2.3 to select the result edges does not work for overlapping edges. That is why, those two "equal edges" representing an overlapping portion, need a special processing, next.

When overlapping between two edges is detected, one of the edges representing the overlapping fragment is labeled as NON_CONTRIBUTING. This is done by changing the edge type parameter of the event. The meaning of this change is that the edge will not be considered for inclusion in the result of the Boolean operation. Hence other edge is labeled as SAME_TRANSITION or DIFFERENT_TRANSITION depending on the overlapping edges having the same inOut flags. Depending on the Boolean operation, the edge will be included in the result if its label is the one checked in the statement. Those edges that are labeled as SAME_TRANSITION are only included in the result of union and intersection operations. On the other hand, the edges labeled as DIFFERENT_TRANSITION are only included in the result of set theoretic subtraction operation. Fig. 4.15 shows, how overlapping fragments are included in the Boolean operations, depending on the inOut flags of the overlapping edges and on the type of Boolean operation.

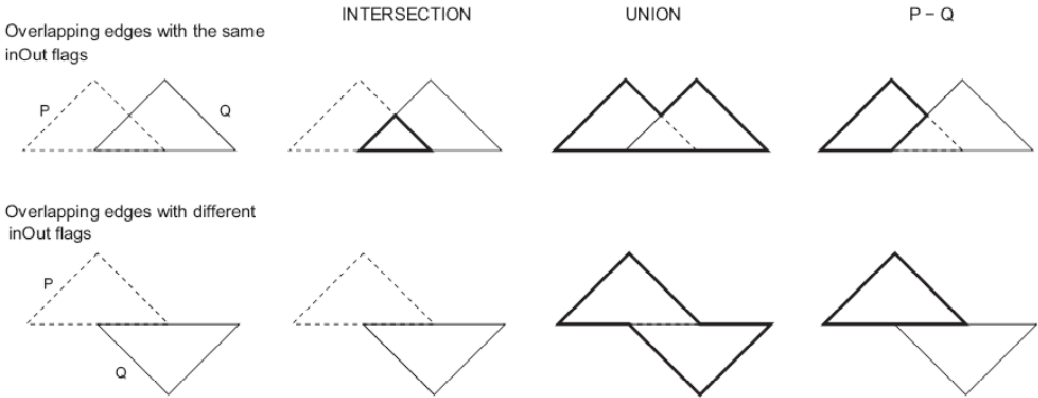


Figure 4. 15 Inclusion of overlapping edges in result of Boolean operations [21]

4.4.3 Polygon sweeping methodology

A new method which is derived from polygon union is developed to sweep a polygon on the plane it lies. In this section this new method is discussed in detail.

4.4.3.1 Introduction

In the previous sections, Boolean operations and their application on polygons are discussed. To make good use of polygon clipping, it is necessary to have a polygon to be clipped. The turning operation, as discussed earlier, is simplified by subtracting CTP from WPP throughout the operation. However, to be able to do this efficiently, the CTP must clip the WPP recursively. The number of steps could be a matter of precision. Better and more elegant way would be to apply this clipping by first generating swept polygon, and then subtracting the resulting polygon from the WPP.

In the developed software, the NC code is interpreted and cutter location data is obtained. This data consists of the tool location in space at specific times. The main parameter while generation this data is the sampling time, which is a property of the machine's controller (i.e. CNC unit).

Once the NC code is interpreted and the cutting locations are obtained accordingly, it becomes possible to use that data to sweep cutting tool with the vectors obtained by subtracting two successive position data from each other. This approach along with the algorithm applied to sweep a polygon on the plane where the polygon lies, are completely new ones.

4.4.3.2 Basics of Polygon Sweep

Polygon sweeping technique in this study, is actually an extension of Boolean union operation on the same polygon itself. The methodology could be summarized as follows:

- A polygon could be considered as a collection of line segments.
- A line segment, when swept on a plane it lies, generates a quadrilateral, unless the sweeping direction is parallel to the segment's direction. This is shown in Figure 4.16.

- If one applies the sweeping, to all the line segments of the polygon, and performs the Boolean union operation, the result will be the swept of polygon.
- The special case occurring only when the swept distance is small such that the final position of the polygon intersects with the first position of the polygon. How this special case is treated, will be discussed in the next sections.

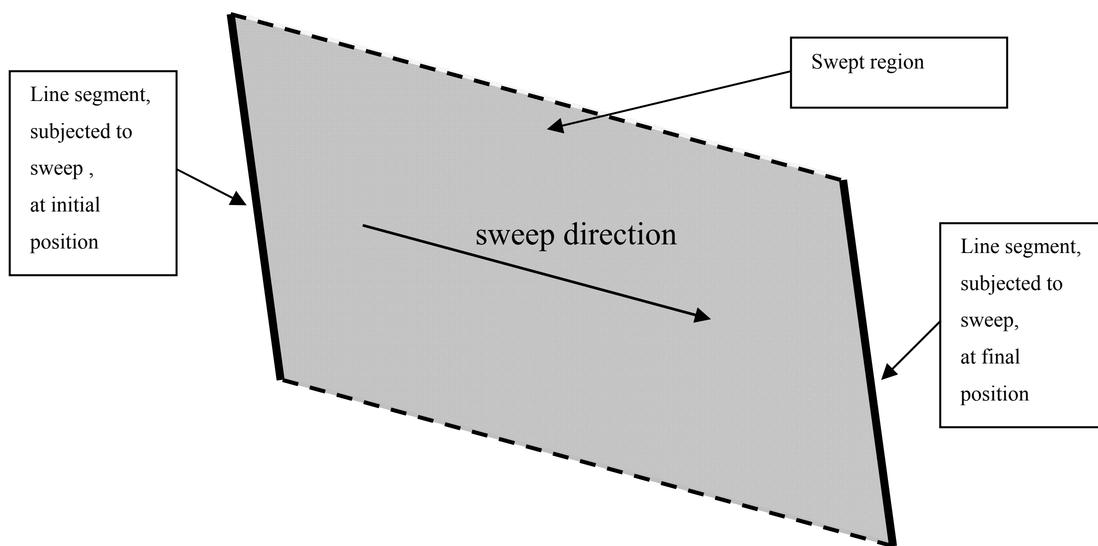


Figure 4. 16 Result of sweeping a line segment

The swept quadrilaterals are then added to each other to obtain the final polygon. The algorithm elaborated in next section.

4.4.3.2 Polygon sweeping algorithm

As in the case of Polygon Clipping case, the object oriented programming is facilitated in polygon sweeping. A specific class is generated to operate on polygon sweeping objects.

Polygon sweeping class has a `Main` function in which the operation takes place. In the Table 4.11, the pseudo code of that function is presented. In the `fncMain` procedure, first two vertex arrays are defined. One of them is to hold the polygon vertex data when polygon is at its initial position. The second one is used to store the vertex data of the polygon when it is in the final position. To start polygon sweep, one of the edges of the polygon is selected. The two endpoints of the line segment is transformed by the sweep vector. Thus, a quadrilateral is defined with the four vertices at hand. At that point, a special case occurs when the slope of the line segment is exactly the same as the sweep vector. When a line is intended to be swept with a vector parallel to it, this sweeping operation will not generate any quadrilateral at the end. Thus, before running the code for a line to be swept, it is necessary to check whether its slope is equal to the slope of the sweep vector. In that case, there is no need to sweep that segment it would not contribute to the result.

At this point, it is compulsory to check whether the sweep vector brings the polygon out of its bounds. In the case where the sweep polygon does not move the initial polygon completely out of its bounds the second special case, which is illustrated in Figure 4.17, occurs. When a polygon, (even after sweeping is applied) still stays within its bounds, only sweeping its boundary line segments would yield a wrong result. To overcome this, the program checks whether this case occurs. If such a situation is discovered, the quadrilateral, which was obtained by sweeping the first line segment is subjected to the union operation with the polygon when it is in its final position. Actually, this situation usually takes place in machining simulation because the circular interpolation and small sampling time causes very small sweeping vectors.

Table 4. 11 Polygon Sweeping function.

```
//Input      : Two double values dx, and dy, list of points
//             incomingPoly
//Return     : Point array
PointD[] fncMain(double dx, double dy, List<PointD> incomingPoly)
{
      Set a point array at initial posisition;
      Set a point array at final position;

      double slope = dy / dx;

      Sweep a segment of input polygon;

      if (dx and dy values are smaller than width and height of
      input polygon's bounding box)
      {
            Add swept line segment to the polygon at final
            position;
      }

      for each remaining line segments in the input polygon do
            Sweep line segment;
            Add swept segment to previous one;
      return result;
}
```

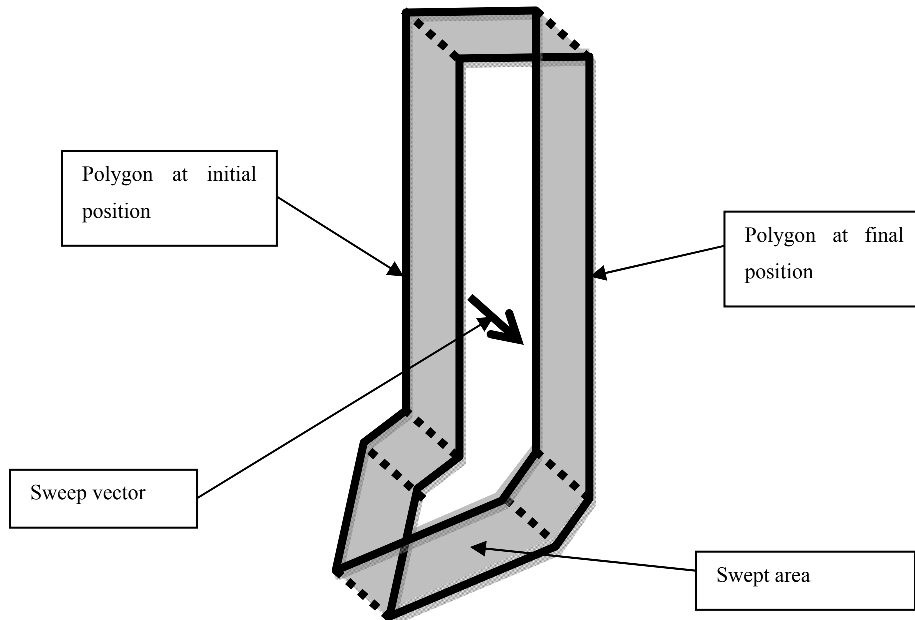


Figure 4. 17 When swept polygon resides in the starting polygon

Before implementing polygon sweeping algorithm into the simulation software, a small program is developed. In that program, which is composed of only a plain windows form, swept region by a polygon is shown. The data is represented as a polygon, which is extracted from tool geometry available in standard tool library. Several trials which includes special cases, are conducted. The algorithm gave the expected result. In Figure 4.18, some sample screen outputs for arbitrary sweeping operations, applied on different cutting tool boundary polygons are shown.

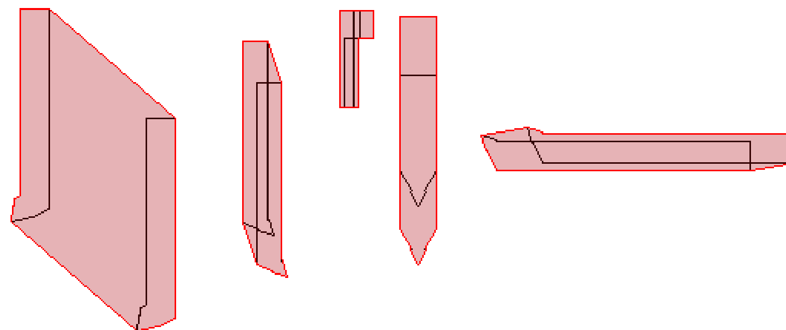


Figure 4. 18 Sweeping operations as applied on different tool polygons.

4.4.4 Algorithm of Turning simulation with Polygon operations

By applying Boolean operations on polygon, modifications on polygons can be achieved. For instance, by applying polygon sweep algorithm, it is possible to simulate cutting tools motion in 2D. At that point, whole algorithm will be emphasized to give a clear idea.

Turning simulation starts with first processing on NC code. Two possible ways to give this data to the program is either by manual input by writing code on the editor at the right hand side or by loading NC code data by browsing for an appropriate file on the hard drives. The first method requires user to write the code line by line. The format of NC code is standard (RS274B) ISO format. Four parameters that user can adjust are:

- Sampling time
- Maximum speed

- Diametral or radial coordinates
- Tool home position

These parameters naturally affect the result of NC code interpretation. Another important parameter (as set by the user), is the dimensions of the workpiece. After setting these parameters, the user inputs the NC code either by typing or by loading file. Then the algorithm starts to interpret the NC code. When finished, the generation of simulation data commences. The interpreted NC code is read line by line and the algorithm checks for the tool changes and tool motions. When a tool change is encountered, the related tool number in the library is called and the new tool geometry data is loaded. After that point the new data is processed, until a new cutting tool is asked by the code to be called.

When the processed code line is associated with a tool motion then the sweep vector for the current vertices of cutting tool is sent to polygon sweeping algorithm. This data is processed and another vertices data is returned by the polygon sweeping algorithm. At that point, enough data is available to apply Boolean subtraction operations on the WPP by subtracting swept CTP data. Therefore, these polygon data are sent to clipping processor object. At the end clipped WPP is obtained. In this stage, some specific algorithms are executed to generate 3D mesh data. Generation of the mesh data will be the subject of the next section.

After the production of all the data, the program is ready to generate visual results. With the instruction of the user, the program starts visualizing the simulation. In Table 4.12 pseudo code of the turning simulation algorithm is presented.

Table 4. 12 Pseudo code of Turning Simulation algorithm.

```
//Input     : Two double values as workpiece length and diameter,
//            A point to set cutting tool home position, Two double
//            values, one for sampling time and other for maximum
//            speed, and an array of NC code strings
void Turning(double workpieceLength, double workpieceDiameter,
             PointD cutterHome, double samplingTime,
             double maxSpeed, string[] ncCode)
{
      Set workpiece;
      for each NC code in ncCode[] do
      {
            Read ncCode;
            Set dx, dz;
            Set cutting tool and sweep it by dx, and dz;
            Clip workpiece by swept cutting tool;
            Rotate workpiece array about z axis and generate mesh
            data;
      }
}
```

4.5 Visualization Methodology

For the visualization of the turning, two viewports in the user interface, are used. The upper one is to show cross-sectional view of the operation. The below one is expected to demonstrate the 3D model of the workpiece.

4.5.1 Visualization in the cross-section viewport

Since clipping algorithms are completely different than each other, pixel-based and polygon clipping techniques have their own visualization procedures for 2D output. On the contrary, for 3D they share the same algorithm. In the next section, these algorithms will be discussed.

4.5.1.1 Visualization in the Cross-Section Viewport in pixel-based simulation

In DirectX, every pixel on the screen is defined as a vertex. Vertices could be draw independently or they could be grouped in a buffer and then they can be fed into graphics hardware (GPU) at once.

In the developed program, the cross-sectional view is generated by directly reading workpiece and cutting tool matrices. If a node in the matrix is true, the related program procedure adds a vertex into the vertex buffer. After completion of vertex buffer creation, the buffer is send to Direct3D device. To generate a view in DirectX, the program firstly needs to have necessary reference namespaces loaded in the project. At the beginning of the project, DirectX, DirectX.Direct3D and `DirectX.DirectInput` namespaces are added to the project to be able to utilize DirectX facilities. Then a DirectX device object has to be defined. After defining this object, it is possible to use it for any rendering operation on it.

In the user interface, the existing viewports for visualization, are controlled by the created device. The viewports are refreshed continuously to reflect changes in the current state of the view. The function called Render is used for this purpose. For the upper viewport, a vertex buffer type of stream source is defined. This definition is directly taken by the device and the data is sent to the frame buffer of the graphics hardware. In this way, DirectX is used as a communication interface between the program and graphics hardware.

4.5.1.2 Visualization in the Cross-Section Viewport in polygon clipping based simulation

Visualization in the cross section viewport in the second technique involves the display at boundary polygons for both the workpiece and CTPs during the simulation. This is easily achieved by storing polygon vertices for in an array and when it is requested by the user, the polygons are drawn on the viewport surface by the GDI+ functions.

To be able to get the most of the viewport a scaling factor is applied to the polygon drawn. This factor is calculated by

- Maximum and minimum X coordinate values, traveled by the cutter as defined by the NC code.
- Maximum and minimum Z coordinate values, traveled by the cutter in the NC code.
- Dimensions of the cutting tool.
- Dimensions of the workpiece.
- Dimensions of the viewport, in which 2D view is generated.

In Table 4.13, the formula of the scale factor is presented.

Table 4. 13 Code snippet of scale factor.

```
double widthParam = maxXCoord + cuttingTool.BoundingBoxWidth;
If (maxYCoord > (workpiece.Diameter / 2)) then
      double heigthParam = maxYCoord + workpiece.Diameter / 2
else if
      double heigthParam = workpiece.Diameter;
end if;

if (heigthParam / windowSize.Height) < (widthParam /
    windowSize.Width) then
      double scale = windowSize.Width / widthParam;
else if
      double scale = windowSize.Height / heigthParam;
end if;
```

4.5.2 Visualization in the 3D Model Viewport

For the creation of the 3D model, surfaces must be created. For this purpose, a primitive (in DirectX) called triangle is employed. Any 3D surface can be defined in terms of triangles. By combining triangles, even very complex surfaces could be generated.

In the next section, generation of 3D mesh data for pixel based simulation will be discussed. Generation of the 3D mesh data depends on the existence of boundary vertices on the workpiece. In polygon clipping technique, this data is readily available. However in pixel based technique, the exiting data must be processed to obtain boundary vertices. This will be discussed in the next section.

4.5.2.1 Boundary vertices data generation in pixel-based simulation

For the creation of 3D model of the workpiece, it is first necessary to extract the boundary of the cross-section for the workpiece in the above viewport. For this purpose, a simple search algorithm, which visits the workpiece data matrix, is developed.

When a boundary node is obtained, it is added to a new data list. Then, this new data list is analyzed whether successive nodes have similar coordinates as if they are on a line. If so, they are skipped, until a node whose coordinates are out of the trend. By doing so, boundary nodes are converted into vertices of boundary line segments.

4.5.2.2 Basics of 3D objects in DirectX

As mentioned before, to generate a 3D object, one of several method in DirectX can be used to generate a mesh. Meshes are collection of triangles .To create a mesh, one needs to define vertices of the triangles and indices of those triangle. That is, the DirectX processes three vertices data along with the given indices to create a triangle.

To form a triangle, just having three vertices is not enough. In addition, vertex normals of the vertices must be defined to apply. In DirectX, a 3D view can be obtained with the application of light. For the calculation of amount of shading in a surface, surface normal is a crucial parameter. To correctly define the normal of a

polygon, three vertices must be given in clockwise order. The normal also defines the front face of a polygon. An illustrative example of a DirectX polygon, which also show vertices and normal of a surface, is shown in Figure 4.19.

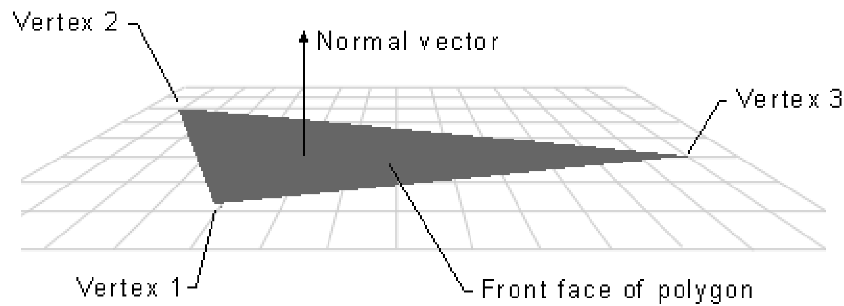


Figure 4. 19 Vertex definition of a polygon and its front face [18].

Another important data, that has to be provided to generate triangular polygons, is their indices. Index data is used to process vertices data in the correct order. As mentioned before, if vertices are not provided in the correct order, some visual artifacts may appear.

To avoid such problems and to utilize the capability of back face culling of DirectX; all the vertices must be given in the correct order. That is done by the help of an index array which keeps indices of the vertices in the order, in which vertices are grouped to form triangles.

In the developed, program meshes are used to generate 3D model of the object. The boundary data is converted to vertex data and then vertices are rotated about workpiece axis, Z axis, With the indices data, they are put into a mesh form to construct 3D view of the workpiece. In the next section, details about generation of mesh vertices and indices will be discussed.

4.5.2.3 Mesh data generation

In the turning simulation, 3D view of the workpiece is simply a cylindrical object. The boundary data with the vertices, one obtains by the application of the algorithms make it possible to easily generate 3D view of the workpiece. This is done by running the algorithm given in the Table 4.14.

To generate mesh data, it is first necessary to have vertex data. To obtain vertices of the workpiece, the following procedure is employed:

1. The boundary data of the workpiece is divided into segments first.
2. Every segment is composed of two vertex. These two vertices are rotated around Z axis to obtain corners of the triangles, which will build the mesh data (before rotating). Two vertices could be processed as follows:
    a. If one of the vertices lies on the Z axis, this vertex is kept fixed (during the rotation process). The other one is rotated.
    b. If both of the vertices lie on the Z axis, then there is no need to process this segment, as it will not generate any surfaces.
3. A special ordering algorithm is run to generate indices at the data.

Segments are called as stacks in the program's notation. Three different stack types are defined, *Empty*, *Disc*, and *Normal*. As explained before, a stack a vertex on Z axis is a disc; no vertex on Z axis is normal and if both vertices are on z axis, it is called empty stack. The algorithm considers a stack as a disc, whether Z coordinates of both vertices are the same. Thus, a conical surface and an exact disc is treated just like the same. In the algorithm, starting and ending vertices of a normal stack could have same Z coordinates (but neither have x coordinate equal to zero). Even in this case, the algorithm considers the stack as normal, and produces correct results.

Table 4. 14 Mesh generation algorithm.

```
// Input    : An array of points for boundary vertices, maximum
//            dimension of the visualization viewport as integer
// Output   : Vertex buffer as PositionNormal vertex array
// Return   : Index buffer array as unsigned short array
List<ushort[]> fncMeshDataGenerator (PointF[] boundaryVertices,
                                    int maxDim,
                                    out List<CustomVertex.
                                    PositionNormal[]>
                                    returningBuffer)
{
      Generate stack list from boundaryVertices array;

      for each stack in stack list do
            if(stack type is Disc)
                  fncDiscMesh(stack);//Run disc stack function
            else if(stack type is Normal)
                  fncNormalStack(stack);//Run normal stack function
      return result;
}

// Input    : A stack
// Output   : A vertex buffer and an index buffer
fncDiscMesh (StackFloat tempStack, ref List<CustomVertex.
             PositionNormal> tempVertices, ref List<ushort>
             tempIndices)
{
      Set the the endpoint lies on the z axis as center point;
      Rotate other point about z axis;
      Save them in vertex buffer;
      Set index buffer array;
}

// Input    : A stack
// Output   : A vertex buffer and an index buffer
fncRegularSectionMesh (StackFloat tempStack,
                       ref List<CustomVertex.PositionNormal>
                       tempVertices, ref List<ushort>
                       tempIndices)
{
      Rotate both endpoints about z axis;
      Save them in vertex buffer;
      Set index buffer array;
}
```

4.6 Features of the Program

In this part as the name implies features of the developed software is presented. In the first section development environment is mentioned briefly. Later user interfaces which are specific to two separate methods are discussed in detail.

4.6.1 System requirements

The simulation program is written in C# language. .Net framework 3.5, and DirectX 9.0c are necessary to run the program. Operating system is Microsoft XP with minimum Random Access Memory (RAM) of 2048 MBs.

4.6.2 User interface

User interface (UI) is where all user interaction takes place. Thus, it is aimed to make UI as functional and easier as possible. In the first part UI of pixel-based program and in the second part UI of polygon-based program are explained.

4.6.2.1 User interface in pixel-based simulation

In Turning simulation user interface window, there are two different viewports as shown in Figure 4.20. In the upper viewport, cross-sectional view of both the workpiece and insert (tool) are illustrated. In the bottom viewport, current state of the part in 3D workspace is shown. It is possible to rotate, zoom and pan the view.

On the right hand side of viewports, workpiece dimensions can be defined by using two numerical up/down controls. One for length of the workpiece, the other is for radius of the workpiece. Where all dimensions are in mm. and it is possible to define the dimensions with one decimal place.

On the right hand side of the window, there are two buttons: The upper one is titled "Load Files". For the pixel based simulation, it is used to load parameter files including cutter geometry. Cutter has to be in the format of a monochrome Bitmap (bmp) file, in which only black and white pixels exist. The second file is to define the cutter trajectory. When the files are correctly selected and loaded, the below button, named "Start Process", becomes active. Before the loading of the files, it is inactive and the program does not run, before successfully retrieving necessary data. The program immediately is executed when the user presses the "Start Process" button. The data generation and 3D visualization occurs simultaneously in the program.

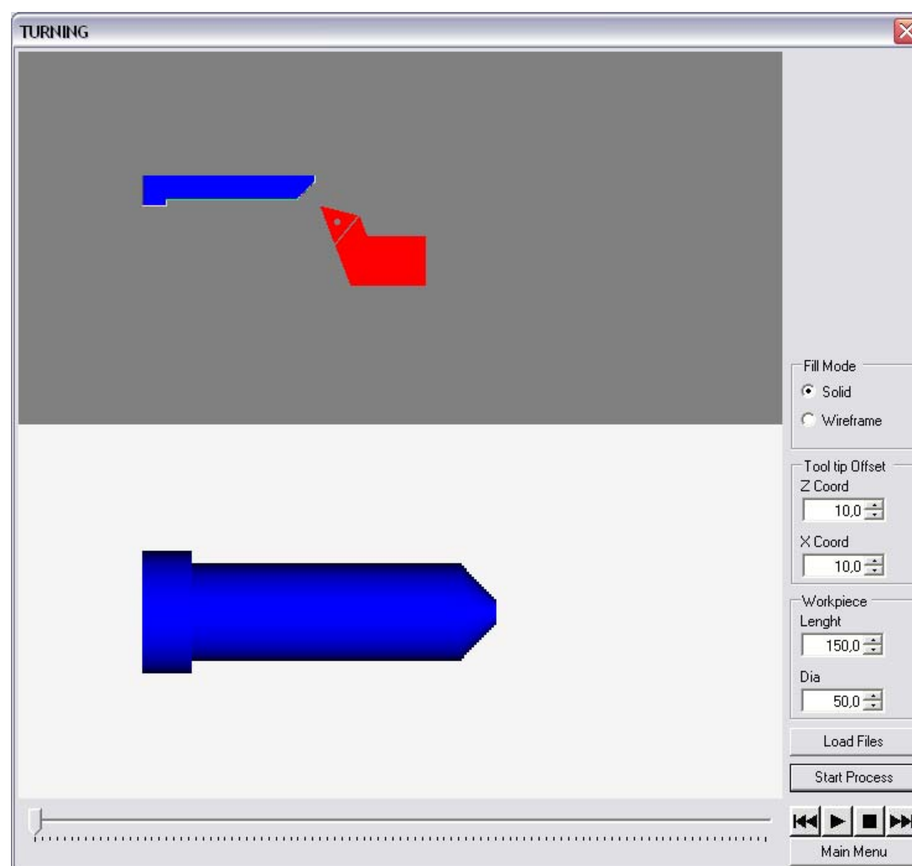


Figure 4. 20 Turning simulation user interface for pixel based simulation.

4.6.2.2 User interface in polygon clipping based simulation

A sample screen shot of the user interface is shown in Figure 4.21. In polygon clipping based simulation case, in the user interface, once again there are two viewports for the visualization. The upper one (#1) is used to perform the simulation on the cross section of the cut. The lower one (#2), visualizes the 3D model of the workpiece.

On the right hand side of the user interface, there is an NC code editor (#3). User can type/edit the NC code line by line or load it from a file. In the NC code editor, it is also possible to insert new line, delete a selected line, and sort the line numbers. To do this, there is a context menu, that appears when the user right clicks on the editor.

Machine settings are defined in the pane #4. There are two text boxes to set sampling time (in sec), and maximum speed (mm/min). In pane 5 tool home position is set. Once again two text boxes accept numerical values for initial X and Z coordinates of the tool tip. In pane 6, there are two radio buttons. They are used to switch between diametral and radial coordinates.

“Process Data in Editor” (#7) button is used to execute simulation with the current code in the editor. When user presses this button, the program starts to interpret NC code. As the name on “Clear editor” (8) button implies, this button clears the text in editor. The progress bar (#9) shows the level of completion during the runtime when the program works on the data for NC code interpretation and data generation for visualization.

There are two radio buttons in pane #10. They are used to switch between wireframe and solid views. Workpiece dimensions are set in two text boxes (#11) where length and diameter are the two parameters here.

“Load File” button (#12) is used to load an NC file from a folder. An open-file dialog appears and user can browse thru the available directories to select a file. After processing of the data in the file or in the editor, “Start Process” button (13) becomes activated. Then, the user can utilize this button to start the simulation. “Stop Process” button (#14) is used to stop simulation. Similarly “Reset view” button (#15) is used to reset camera view. Camera view can be adjusted with some keyboard input. Below is the list of keyboard keys, can be employed to adjust some viewing operations:

- ‘A’ or ‘a’ : Moves camera right
- ‘Q’ or ‘q’ : Moves camera left
- ‘S’ or ‘s’ : Moves camera down
- ‘W’ or ‘w’ : Moves camera up
- ‘D’ or ‘d’ : Zooms in
- ‘E’ or ‘e’ : Zooms out
- ‘Z’ or ‘z’ : Rotates camera about -x direction
- ‘X’ or ‘x’ : Rotates camera about +x direction
- ‘C’ or ‘c’ : Rotates camera about –y direction
- ‘V’ or ‘v’ : Rotates camera about +y direction
- ‘B’ or ‘b’ : Rotates camera about +z direction
- ‘N’ or ‘n’ : Rotates camera about –z direction

Horizontal scroll bar (#16) can be used to adjust simulation speed. Main Menu button (17) is used to return to the main menu. Finally, trackbar at the very bottom (18), is employed to navigate through simulation. User can place the indicator to navigate in the simulation.

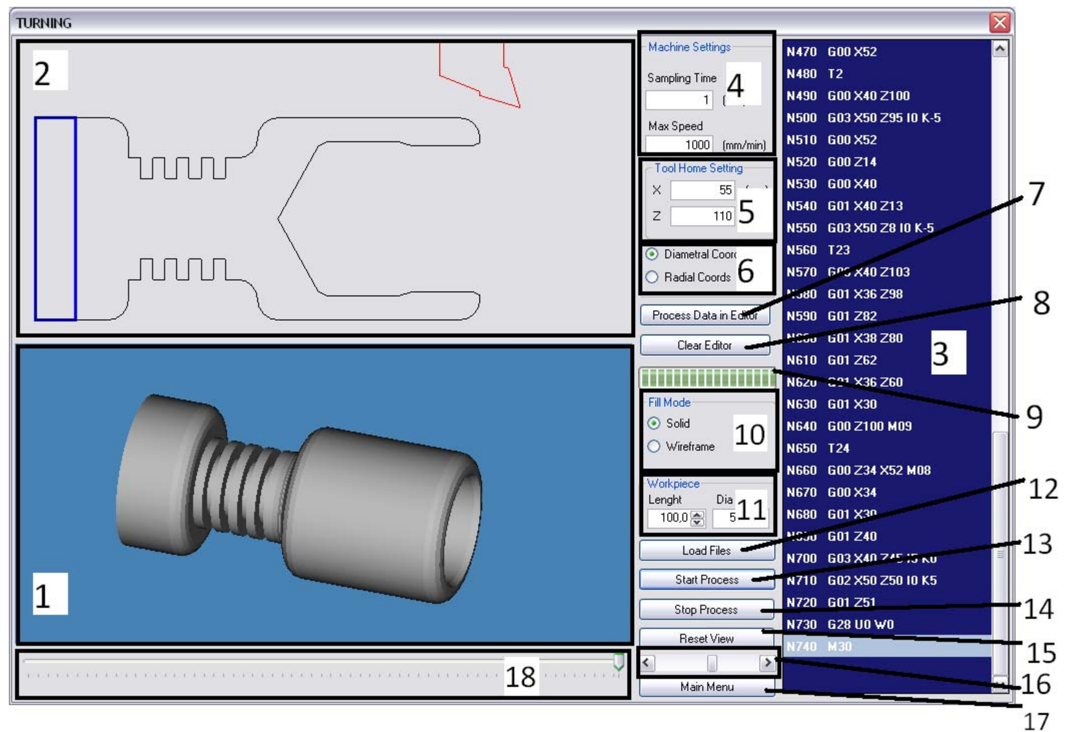


Figure 4.21 Turning simulation user interface for polygon clipping based simulation.

4.7 Closure

In this chapter, two different simulation methods are devised, to simulate turning operation in machining. The first technique was a pixel-based one. The workpiece, along with the cutting tool were modeled as matrices with Boolean values. Dimensions of the workpiece matrix is adjusted by the given length and diameter data. On the other hand, the cutting tool is geometry is defined by a bitmap image file. Color values of the pixels in the image file is checked to generate cutting tool matrix. After generation of the workpiece and cutting tool matrices, the cutting tool locations throughout the cutting operation, is extracted by the algorithm. The algorithm took that location information and applied the transformation to the cutting tool matrix. At every location of the cutting tool matrix, Logical XOR operation is applied to the workpiece- and cutting tool matrices. After this logical operation, the workpiece matrix is searched to get its boundary points. Once boundary points are obtained, they are processed to extract significant boundary

vertices. At the end, boundary vertices are rotated around the Z axis to generate mesh data.

In the second technique, Boolean operations on polygons are utilized. The workpiece- and cutting tool geometry are defined as polygons at the beginning. In this method, a polygon is constructed as an ordered list of vertices. In this second technique, cutting tool locations are retrieved by interpreting NC code. NC code is either loaded from a file or typed by the user using the editor of the user interface. The interpreted NC code is then used to transform vertices of the CTP in the space. To generate clipping polygon, the boundary vertices of the cutting tool is swept on ZX plane as defined in the NC code. For that purpose, a new sweeping algorithm is developed. The result of the sweep is used to clip WPP. Clipped WPP is once again rotated around Z axis and the 3D mesh data is formed.

There is a big difference in the quality of the results for two techniques. In the first one, the simulation resolution is limited by the size of the generated matrices. The size of matrices, which consist of the Boolean values, could be increased in that case, there is a physical limit, imposed by the memory of the computer. Also, simulation result is highly dependent on the success of search in the boundary vertices. In some cases, serration effect could be encountered. That is, the search algorithm finds it necessary not to skip some of intermediate points. This situation increases the amount of stacks in the result meshing, which leads to a more complex 3D object. A more complex 3D object may cause difficulty in rendering.

The second technique, is on the other hand, works directly with boundary vertices. Therefore, in that case no search algorithm is necessary. Smoothness in the circular interpolation is a matter of sampling time, which is an adjustable parameter. The developed program lets user to define NC code (so called G code) following RS 274D formalism. This lets user to develop an NC program in the runtime. The simulation time is highly dependent to the sampling time. The computation speed is an inverse proportional to the sampling time.

The algorithm developed using the polygon clipping technique is considered to be more suitable for the turning simulation. The main reason that is the vertices data

are processed throughout the calculations. This data simulates the real situation successively and is very easy to be used in the generation of 3D model for the workpiece object in the simulation. Due to those capabilities of the second technique, it is considered as more effective and advantageous over the first one for the turning simulation.

CHAPTER 5

MILLING SIMULATION

5.1 Introduction

Milling simulation feature of the program aims to simulate a turning operation with the given parameters. For that purpose, a specially designed GUI is used to make parameter and data input and to observe the simulation. In the milling simulation, the workpiece is considered as a rectangular block, and the cutter is defined as a flat end mill.

5.2 Workpiece and Cutting Tool Definition

In this section, use of ray casting to create dexel data for both cutting tool and workpiece is discussed in detail.

5.2.1 Dexel Data Structure

In this milling simulation process; a simple but fast technique, which depends on dexel modeling, is used. This modeling technique was first proposed by Van Hook [6]. His model is just adapted to C#, but the idea is somewhat similar.

To obtain real-time shaded NC milling display, an extended Z-buffer data structure is used. In this data structure, each pixel is regarded as a rectangular solid extending

along the Z axis, and consists of a near Z depth, a far Z depth a color and a pointer. Since in C# cannot be used in managed safe code, another data type, called List is used instead of pointers. In keeping with the convention of the names pixel and voxel, this rectangular solid is called a dexel, or depth element. The dexels are organized in an X by Y matrix called a dexel structure which corresponds to the raster display frame buffer. Multiple dexels in a cell are maintained in near to far Z order. The dexel structure developed by Van Hook [6] is shown in Figure 5.1. The dexel structure is created by scan conversion of a surface data where scan conversion is done by Ray Casting.

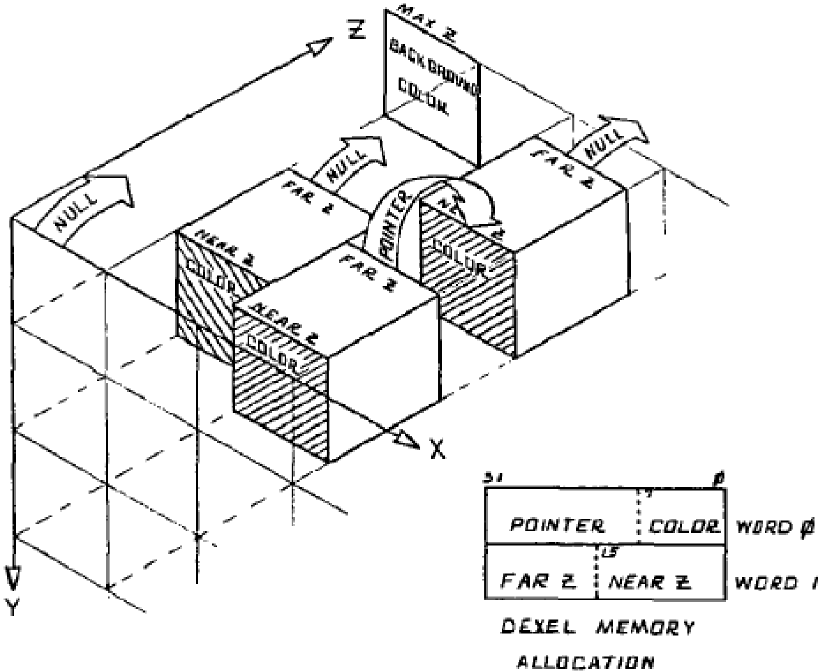


Figure 5. 1 The dexel data structure developed by Van Hook [18].

5.2.2 Ray Casting

Ray casting algorithm is first proposed by Roth [19]. In his definition, a ray is simply a straight line in 3D space eminating from the camera model. It is best defined in parameterized form as a point (X_0, Y_0, Z_0) and a direction vector (D_x, D_y, D_z). In this form, points on the line are ordered and accessed via a single parameter t. For every value of t, a corresponding point (X, Y, Z) on the line is defined:

$$
\begin{aligned}
X &= X_0 + t.D_x \\
Y &= Y_0 + t.D_y \\
Z &= Z_0 + t.D_z
\end{aligned}
\tag{5.1}
$$

A ray in a parallel view that passes through pixel (X, Y) in the screen is simply represented as (X, Y, 0)(0, 0, 1). At this point the algorithm developed by Roth [19] is left and a specific derivation is applied. For our case in 3D space, all the objects have outer surfaces in the form of triangles. In the developed program, both the workpiece and the cutter are formed by meshes. Meshes are as explained earlier in Turning simulation chapter, are composed of vertices and indices data. In CNCVerifier, the application of ray casting starts with extracting triangles vertex data. For this purpose, specific commands of DirectX are used. They are `Mesh.LockVertexBuffer`, and `Mesh.LockIndexBuffer`. Their definition are given below.

- `Mesh.LockVertexBuffer(Type typeVertex, LockFlags flags, params int[] ranks)`
- `Mesh.LockIndexBuffer(Type typeIndex, LockFlags flags, params int[] ranks)`

The first one is to extract vertex data from the meshes. To put vertices in correct order, index buffer data is read. For this purpose, at any pixel location in the display frame (in our case simulation viewport) a parallel ray is cast through the viewer to the object in the scene. The ray casting requires calculation of pixel data on the display frame. Since the object must be a combination of triangles, it is necessary to determine the intersections of rays with these triangles. The ray triangle intersection is computed by DirectX with readily available function, `Geometry.IntersectTri`. Its definition is given in Table 5.1.

Table 5. 1 Definition of Intersection function.

- `Geometry.IntersectTri(Vector3 zero, Vector3 one, Vector3 two, Vector3 rayPos, Vector3 rayDir, out IntersectInformation hitLocation)`

The output of this function is a Boolean value, which is true or false. If the function returns true value, it means that the ray intersects the triangle, and the `hitLocation` variable has a value. Otherwise, it is understood that the triangle does not have an intersection with the ray and `hitLocation` variable has a null value.

In fact, `hitValue` variable is `IntersectInformation` type. `IntersectInformation` type has barycentric coordinates.

Bartcentric Coordinates [20]: In geometry, the barycentric coordinate system is a coordinate system in which the location of a point is specified as the center of mass, or barycenter, of masses placed at the vertices of a simplex (a triangle, tetrahedron, etc). First, let us consider a triangle T defined by three vertices V_1, V_2, V_3. Any point P located on this triangle may then be written as a weighted sum of these three vertices, i.e.

$$P = \lambda_1 V_1 + \lambda_2 V_2 + \lambda_3 V_3 \quad (5.2)$$

Where λ_1, λ_2, and λ_3 are the area coordinates. These are subjected to the constraint

$$\lambda_1 + \lambda_2 + \lambda_3 = 1 \quad (5.3)$$

If the `Geometry.IntersectTri` returns true, then it turns out that it is easy to determine the point of intersection with the data included in `hitLocation`. To calculate intersection point, the *U* and *V* data of `hitLocation` can be utilized:

$$P = V_1 + U(V_2 - V_1) + V(V_3 - V_1) \quad (5.4)$$

Any point on the triangle can be represented by the barycentric coordinate *(U, V)*. The *U* parameter controls how much V_2 gets blended into the result, and the *V* parameter controls how much V_3gets weighted into the result. Lastly, *1-U-V* can be calculated to determine the weight of V_1 into the result.

Armed with the information given above, and embedded functions in DirectX libraries, it becomes easy to determine intersection calculations. Every single triangle is intersected with a bunch of rays surrounding a triangle. Both screen X, and Y coordinates, and Z coordinates (indicating the intersection point's depth along the ray) is determined from the intersection information.

All these intersection information is collected into a matrix having the dimensions, exactly the same as the simulation viewport. Since the number of intersections could vary, the data stored in that matrix is a list which gives the flexibility to add or remove points later in the simulation. The procedure is the same for both objects in the simulation, workpiece and cutter. When applied, the above methodology yields matrices which contains coordinate values of intersection points in Z axis with a color value combined.

5.3 Cutting Methodology

Cutting methodology in the developed program is unique for a system, which utilizes dexel data structure. This procedure was first explained in Van Hook's study [18]. The cutting procedure consists of three nested loops that step through each Y and X tool dexel, and subtract it from each block dexel in the list. In order to subtract an object with internal voids, a fourth loop can be added to step through the linked list of dexels in the tool. This capability would provide CSG subtraction operation. However it is unnecessary in CNC milling display where the tool is a convex object. The offsets in X and Y from the tool dexel to the block dexel along with the Z offset of the tool are three degrees of freedom (DOF) in the cutting procedure. For more than three degrees of freedom, the tool dexel structure is recreated from a transformed tool surface description. Since the tool is typically small in proportion if compared to the entire scene, and the recreation is equivalent to scan conversion, 5-axis milling can be effectively displayed. In the cutting function, the cases shown in Figure 5.2 are checked. The cutting procedure pseudo-code is given in Table 5.2.

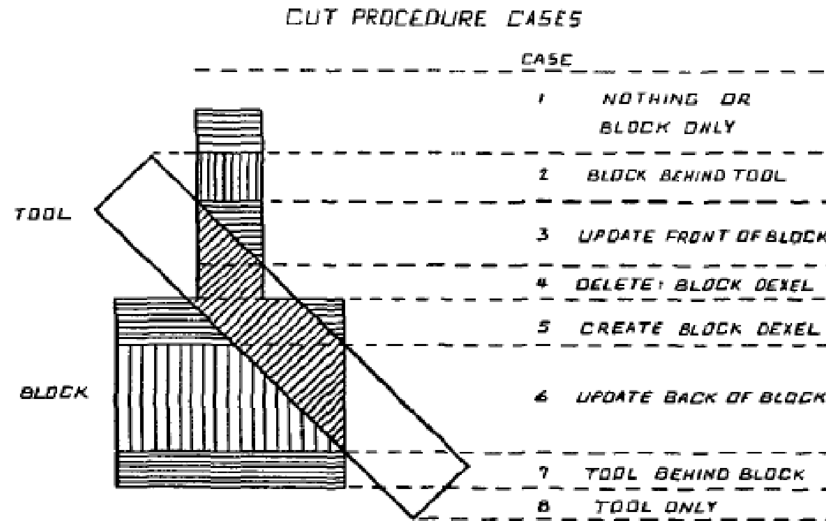


Figure 5. 2 Cut procedure cases, Van Hook [18].

Table 5. 2 Cutting procedure.

```
Procedure    cut(cutter_x_min,    cutter_y_min,    cutter_x_max,
cutter_y_max)
{
      Structure cutter_dexel{cutter_near_z, cutter_far_z,
                             cutter_color}
      structure workpiece_dexel{workpiece_near_z, workpiece_far_z,
                                workpiece_color}
      for cutter_y = cutter_y_min to cutter_y_max {
            for cutter_x = cutter_x_min to cutter_x_max {
      CASE 1:     if cutter_dexel =empty;
                        next cutter_x;
      CASE 8:     if workpiece_dexel =empty;
                        next cutter_x;
      CASE 2:     if cutter_far_z < workpiece_far_z {
                        If cutter_far_z < workpiece_near_z
                              next cutter_x;
      CASE 3:           else if cutter_near_z < workpiece_near_z
                              workpiece_near_z = cutter_far_z;
      CASE 5:           else
                              add workpiece_dexel;
                  }
```

Table 5.2 (continued)

```
                else  {
      CASE 7:         if cutter_near_z > workpiece_far_z
                            next workpiece_dexel;
      CASE 6:         else if cutter_near_z > workpiece_near_z
                            workpiece_far_z = cutter_near_z;
      CASE 4:         else  delete workpiece_dexel;
                      next workpiece_dexel;
                      }
                }
          }
      }
}
```

5.4 Visualization Methodology

For the visualization before the ray casting (as mentioned in Chapter 2) a Direct 3D device is created. Direct3D camera is also defined and light is applied in the scene. The 3D models of the cutter and workpiece is generated by constructing meshes with the user defined dimension data. At the beginning, as default values, dimensions of the workpiece are 200.0 mm width, 200.0 mm length, 100.0 mm height and the dimensions of the cutting tool is 10.0 mm radius and 100.0 mm length. As these values are changed, their generated views also change before ray casting.

After ray casting is done, the dimensions of the workpiece are kept as they were before the ray casting performed. The dimensions of the cutting tool can be adjusted even after the ray casting executed. As mentioned in Chapter 2, the view after ray casting is just generated by a vertex buffer. As every element in the scene is just an ordinary vertex with only a color value.

Only exception is cutting tool. Even after ray casting cutting tool is still in mesh form and still a 3D object. The one shown in the view is just a dummy cutting tool.

For the calculations, a ray casted cutting tool is used. However for visualization, a cutting tool mesh is still shown to simulate the operation better. This also gives the opportunity to hide cutting tool whenever required for some reason.

5.5 User Interface

The user interface of milling is shown in Fig. 5.3. In milling simulation user interface, the main viewport is where all the simulation and cutting operation takes place. In this viewport, before ray casting starts, user can make any viewing operations, like zooming, rotating or panning etc. It is important to make necessary adjustments to obtain a good viewing point before the machining starts. Because, in ray casting technique, after the viewing direction and point is chosen, all the calculations are done for this viewing direction.

On the right side of the simulation viewport, there is a number of group boxes and buttons for specific operations. The one at the top (#1) is “Move Cutter group box. It includes three text boxes, labeled as “dX”, “dY”, “dZ”, and a checkbox “Absolute”. The purpose of this group box is to make a single cutter movement and see the result. When the check box is checked, labels of text boxes appear as X, Y, and Z. Conversely, when unchecked they return to their default appear dX, dY, and dZ. Default state of the check box is unchecked in which the cutting tool makes an incremental movement set by values dX, dY, and dZ. However when the check box is checked, cutting tool moves to the coordinates set by the user in the X, Y, and Z boxes. Since the simulation is just for three-axis milling; thus inputting only translation of the cutter is sufficient.

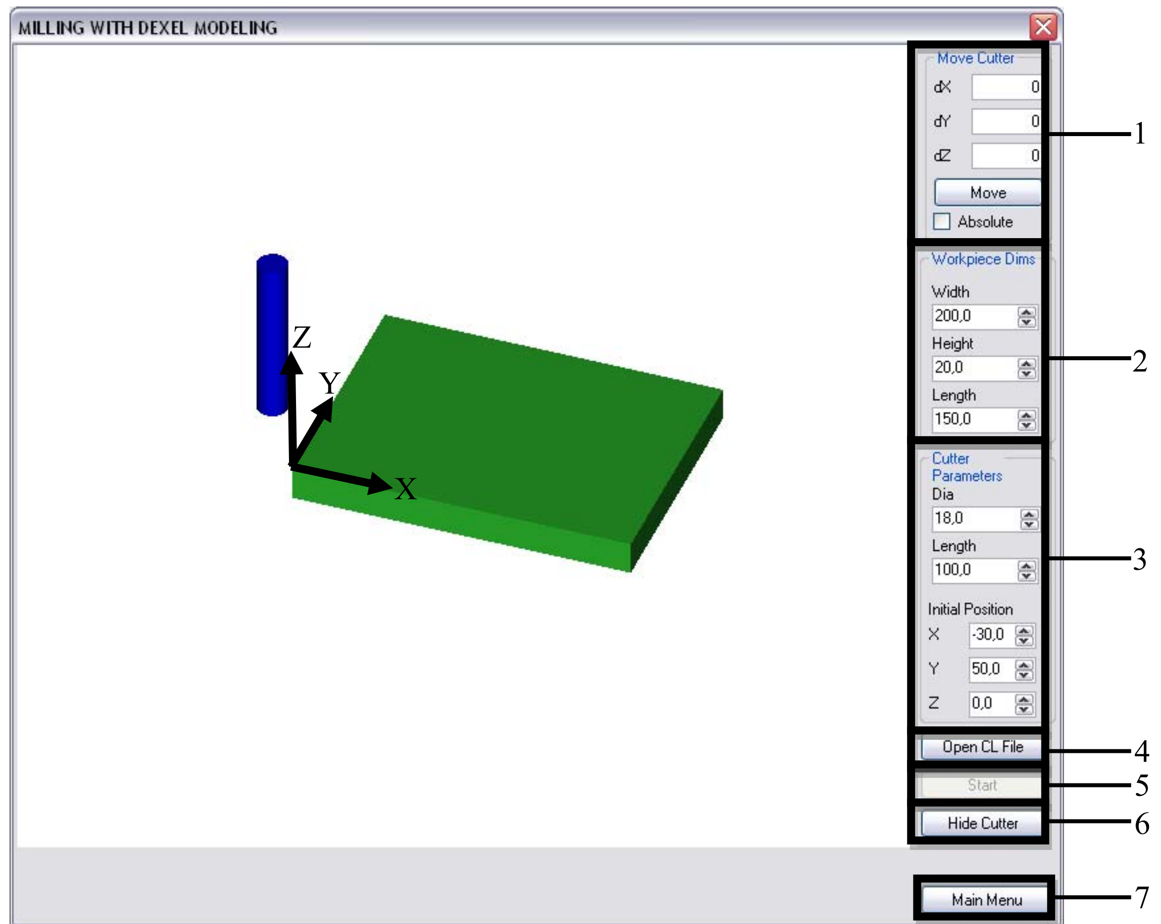


Figure 5. 3 User interface of Milling Simulation.

Below the "Move Cutter" group box, "Workpiece Dims" group box (#2) is located. As the name implies, in this group box, dimensions of the workpiece is defined. The width of the workpiece refers to the length in the X direction is shown in Figure 5.3. The height is the length in Z direction, and the length is the depth into the viewing direction which is Y direction. All the dimensions are in mm. Just under the "Workpiece Dims" group box, "Cutter Dims" group box (#3) is located. In this one, the diameter and the length of the cutter is defined. In addition, there are three to set the initial location of the cutting tool in the workspace. When their value are changed, cutting tool moves to the new coordinate. The values set in these three boxes are absolute coordinates.

When the "Open CL File" button (#4) is clicked, an open file dialog box appears. The format of CL file should be in "csv" (comma separated value) file format. If a CL file is loaded successfully, the "Start" button becomes active. The button is used

to run the loaded CL file. The cutter performs movements according to CL file data. Under “Start” button, “Hide Cutter” button (#6) is located, which gives user the option to hide the cutter, maybe for some more visibility. The last button is labeled as “Main Menu” (#7). When pressed program returns to main menu to switch between turning and milling user interfaces.

5.6 Closure

It is worth mentioning about the limitations of the milling simulation with dexel model. Developed program applies the technique derived by Van Hook [6] and Roth [19]. Their approach is adapted to utilize the capabilities of a 3D graphics hardware managed by DirectX and ease of a managed programming language C#. Although the performance characteristics and programming simplicity of these two approaches developed by [6] and [19] are considered as convincing, they have still some limitations which cannot be avoided because of the nature of the methods.

Firstly, the view of the final part at the completion of the cutting operation cannot be redisplayed from another viewing angle. The machined part exists only as an image without any object-space surface description. Although the voxel-based view transforms can be applied, the quality of results are associated with voxel data is generally lower than that expected of solid model display. Shading for a new view by calculating new surface normals based on the dexel structure is much less accurate than the surface descriptions of the tool and its path. However, since the milling display runs in real-time, a new view can be generated by running the path from a different viewpoint (or running a path from multiple viewpoints in different image windows at the same time). Of course, this will bring extra load on the processor and will affect the running speed of the simulation. The view of the final part at the completion of the simulation is a image-resolution model that does not provide any information regarding tolerancing. This simulation will not entirely replace the object-space intersection calculations as done in general purpose machining verification software (or test runs on the actual milling machine). However, it will limit expensive operations to verify the paths visually and encourage interactive experiment and optimization for better use production resources.

CHAPTER 6

CASE STUDIES

6.1 Introduction

Sample applications are done for turning and milling cases. For both simulations, results are compared to the results of CNCSimulator [31] software. For the simulations a PC with given configuration below is used.

- CPU: Intel Core2 T7400 @2.16GHz
- RAM: 2 GB DDR2
- Graphics Card: ATI X1600 256 MB DD2 Ram

6.2. Turning simulation case study with pixel based technique

In this application, turning simulation of a part with the given dimension below is performed. One of the cutting tool available in the library is selected for the simulation.

- Workpiece Length = 150 mm
- Workpiece Diameter = 50 mm

In the user interface, the parameters are defined as described above. To feel the 3D view, the workpiece is rotated little bit and presented in Figure 6.1. Another feature of the program in turning mode is that the 3D view of the workpiece can be viewed

as wireframe-solid modes. The radio buttons on the right-hand side of the user interface, are used for that purpose.

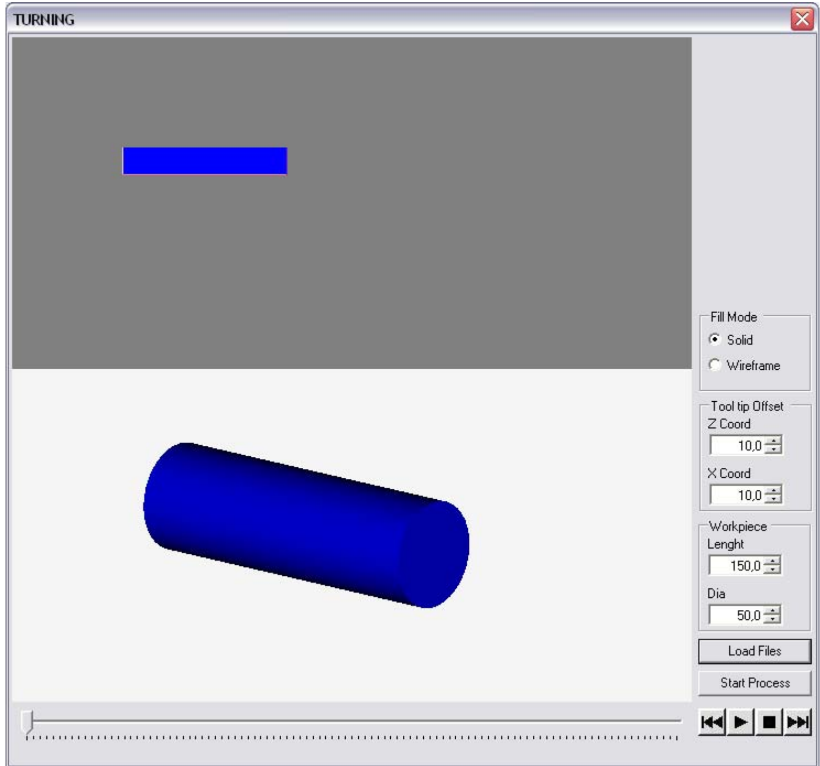


Figure 6. 1 The workpiece is rotated to visualize it in 3D space.

The next step is to press "Start" button. After that, the simulation starts immediately with maximum processing and visualizing speed. The machining operation is simultaneously shown in both viewports. The part program consists of around 50 lines of movement and the simulation takes about 2 seconds. The result of the loaded part program and its output as solid and wireframe models are shown in Figures 6.2 and 6.3.

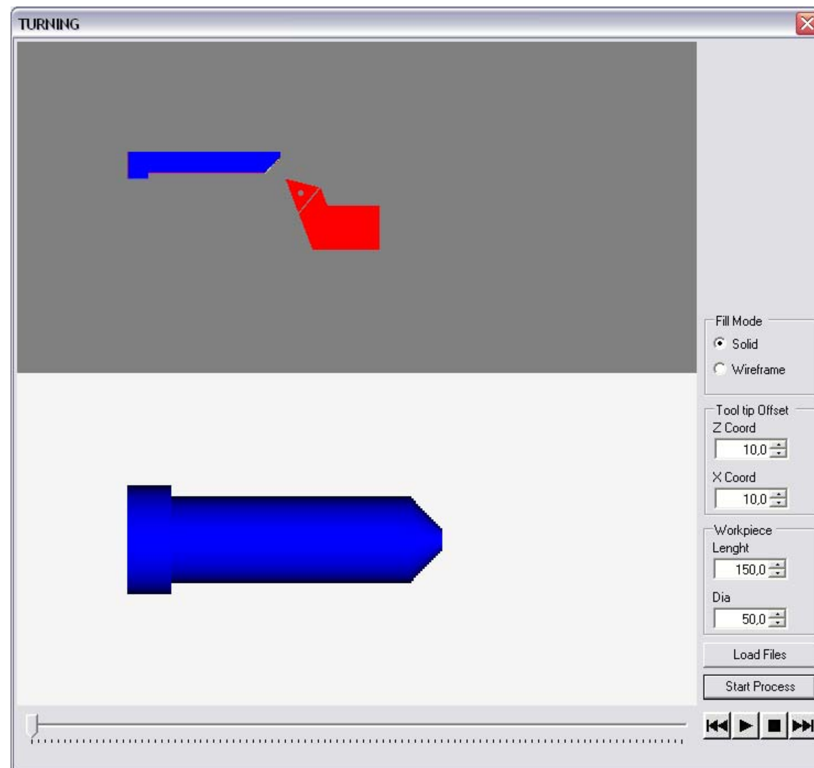


Figure 6. 2 The results of the simulation as shown in solid mode.

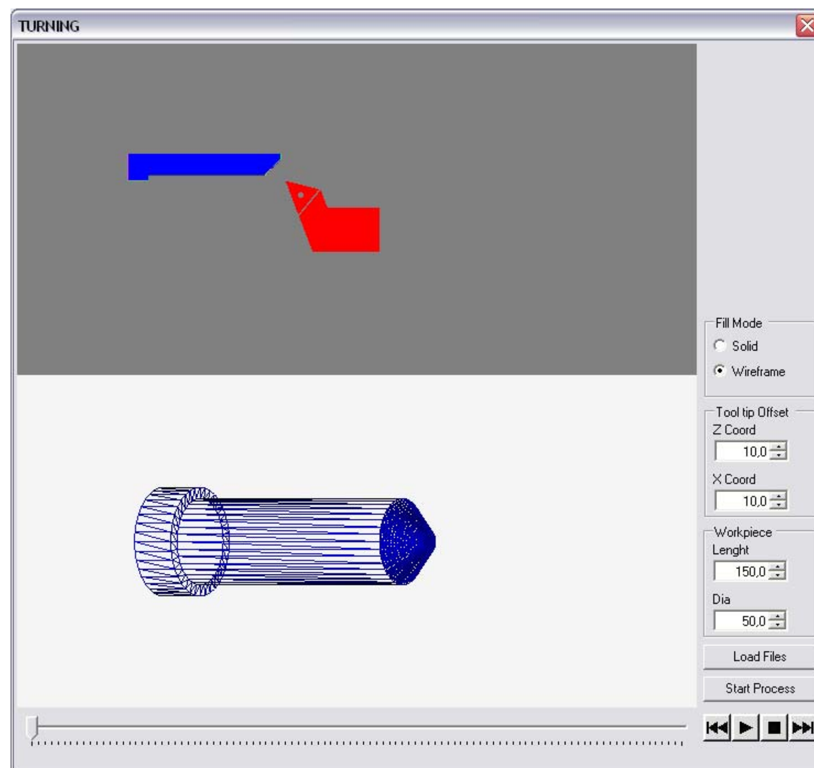


Figure 6. 3 The result of the simulation in wireframe mode.

6.3. Turning simulation case studies with polygon clipping technique

6.3.1 Case study 1

To test the capabilities of CNCVerifier, same NC code is run on both CncSimulator and developed software. Some machine and workpiece parameter settings, used in the simulation, are given as below:

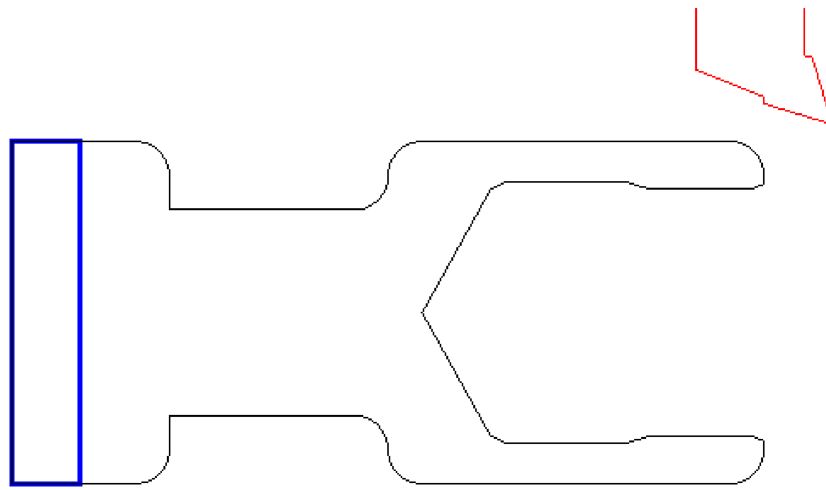
- Sampling time = 0.5 sec
- Maximum speed = 500 mm/min
- Workpiece length = 100 mm
- Workpiece diameter = 50 mm

Data for the first example, in the NC program file format, is presented in Appendix A1. Memory Requirements and time for the execution of the calculations in Virtual Machining with different sampling times for the first example NC program are given in Table 6.1

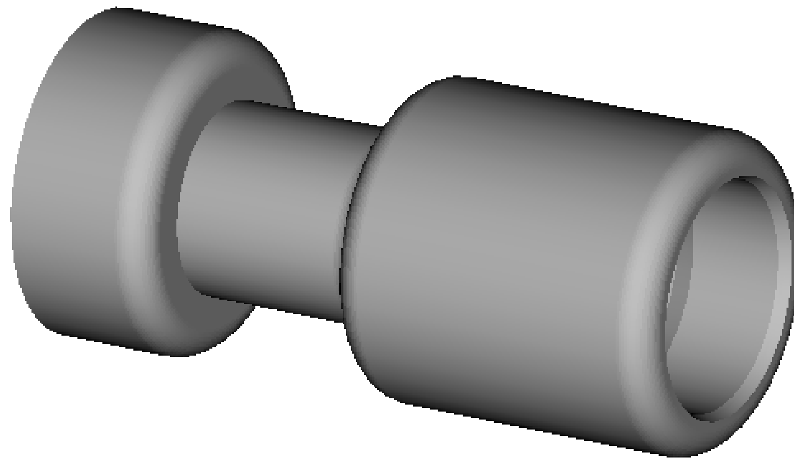
Table 6. 1 Execution times and memory requirements in VirtualMachining for the first example NC program

Sampling Time	Execution time	Memory requirement
1 sec	3 sec	40 MB
0.5 msec	8 sec	52 MB
0.25 msec	17 sec	90 MB

Simulation results are shown in Figure 6.4 and in Figure 6.5 for both programs respectively.



(a)



(b)

Figure 6. 4 (a) Result of simulation in 2D cross sectional view. (b) 3D object model representation of the workpiece in CNCVerifier.

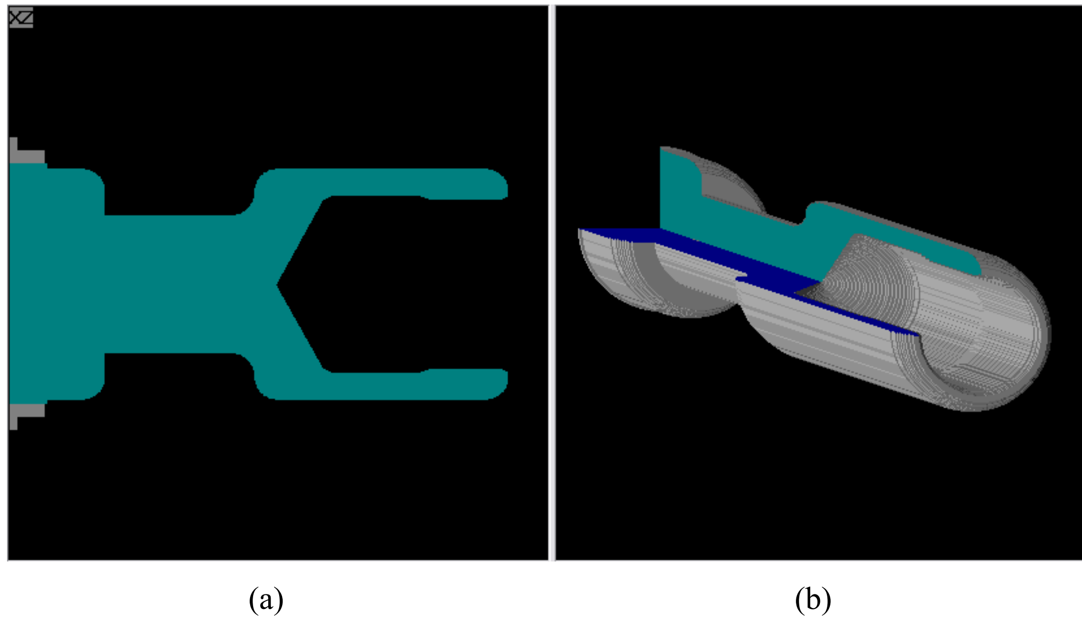


Figure 6. 5 Screen view output of the CNCSimulator. (a) 2D cross section of the workpiece after simulation. (b) Isometric view of the workpiece.

Results show that CNCVerifier gives perfectly shaded 3D model of the workpiece at the end of the simulation. CNCSimulator does not produce a realistic 3D model. The view in the isometric viewport of CNCSimulator could not be zoomed or rotated. On the contrary, the object in 3D viewport of the CNCVerifier could be zoomed, panned, and rotated freely. Furthermore, it is possible to view that model in wireframe mode.

Simulation times are almost the same for both of the software packages. In both programs, it is possible to adjust simulation speed and both programs highlight current NC code during the simulation. Visual quality makes CNCVerifier more advantageous to get more out of the simulation results.

6.3.1 Case study 2

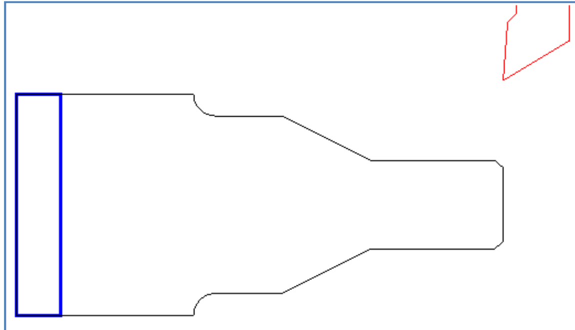
In the second application another NC program is employed which is loaded and run on both programs. Machine and workpiece settings are the same as the first

example. In Appendix A2, content of the NC program is presented. Once again sampling time is set to three different values to see the performance of CNCVerifier. The execution time of the NC program interpreting and all other calculations, with the memory requirements are given in Table 6.2.

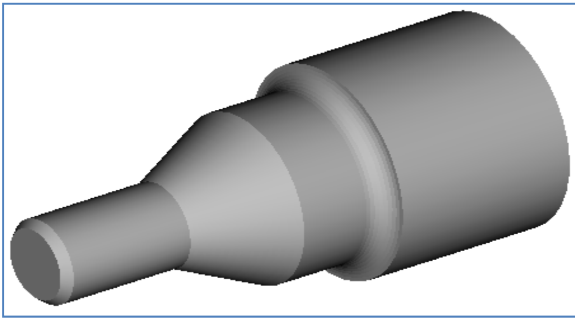
Table 6. 2 Execution times and memory requirements in CNCVerifier for the second example NC program.

Sampling Time	Execution time	Memory requirement
1 sec	3 sec	37 MB
0.5 msec	5 sec	40 MB
0.25 msec	10 sec	57 MB

Simulation results are shown in Figure 6.6 and in Figure 6.7 for both programs respectively. As shown from the figures, both programs yields the same result.

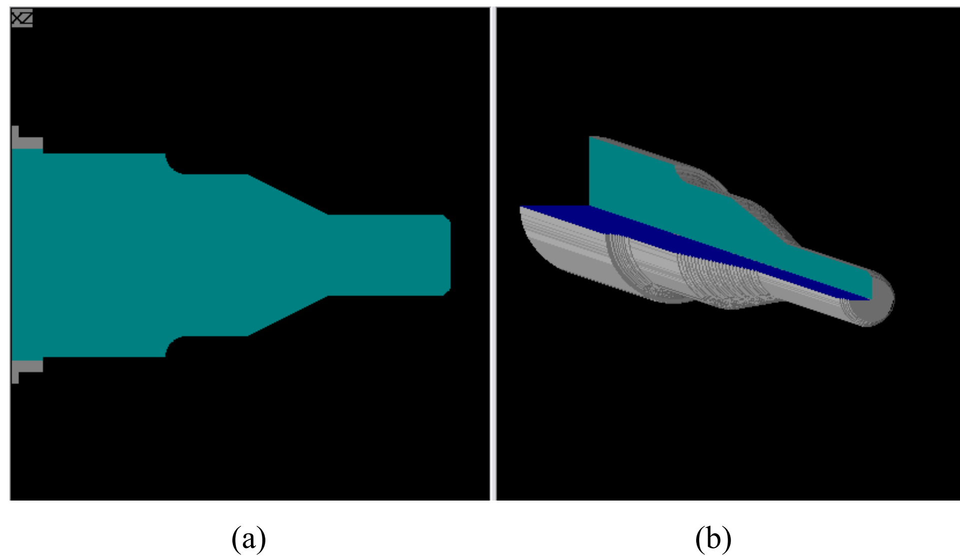


(a)



(b)

Figure 6. 6 (a) Result of simulation in 2D cross sectional view. (b) 3D object model representation of the workpiece in CNCVerifier.



(a) (b)

Figure 6. 7 Screen view output of the CNCSimulator. (a) 2D cross section of the workpiece after simulation. (b) Isometric view of the workpiece.

6.3 Milling Simulation - Sample Application

For milling simulation a block of dimensions listed below is machined with a flat end mill with 18 mm. in diameter. The NC code for this study is given in Appendix A3.

- Workpiece Width = 100 mm
- Workpiece Length = 100 mm
- Workpiece Height = 20 mm

Figure 6.8 shows the GUI before the simulation has started. As mentioned before, since there is no way to change the viewing direction after simulation starts, it is crucial to arrange the viewing direction in the scene to have a good understanding about the simulated CNC. The viewing direction is set as shown in Figure 6.8 to get a similar viewing angle that CNCSimulator provides. The capability of setting viewing angle in CNCVerifier does not exist in CNCSimulator. Also in CNCSimulator neither dimensions in Z direction are represented in scale nor the shading is natural. In other words, surfaces emit same amount of light and have same color gradient even if they have different normals with respect to the looking

and lighting direction. Simulation result of CNCVerifier is shown in Figure 6.9 (a) and simulation result of CNCSimulator is shown in Figure 6.9 (b).

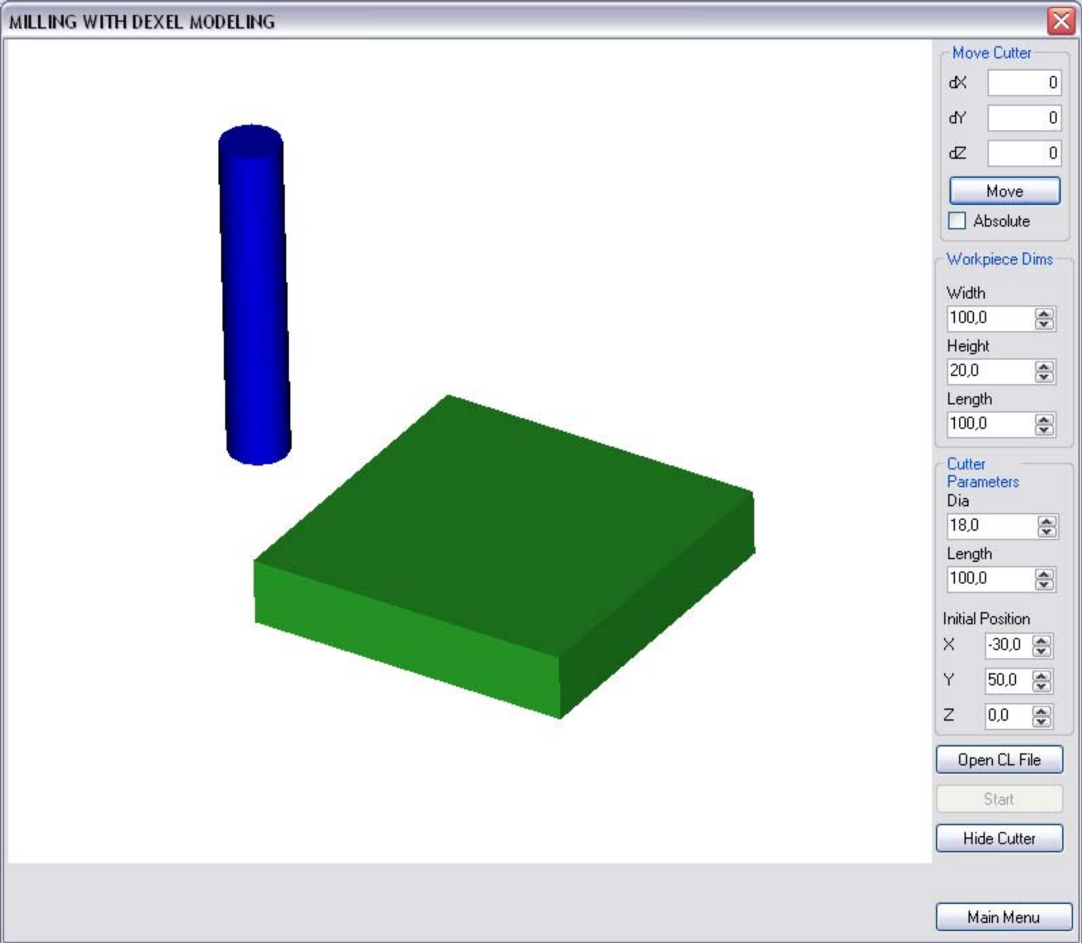
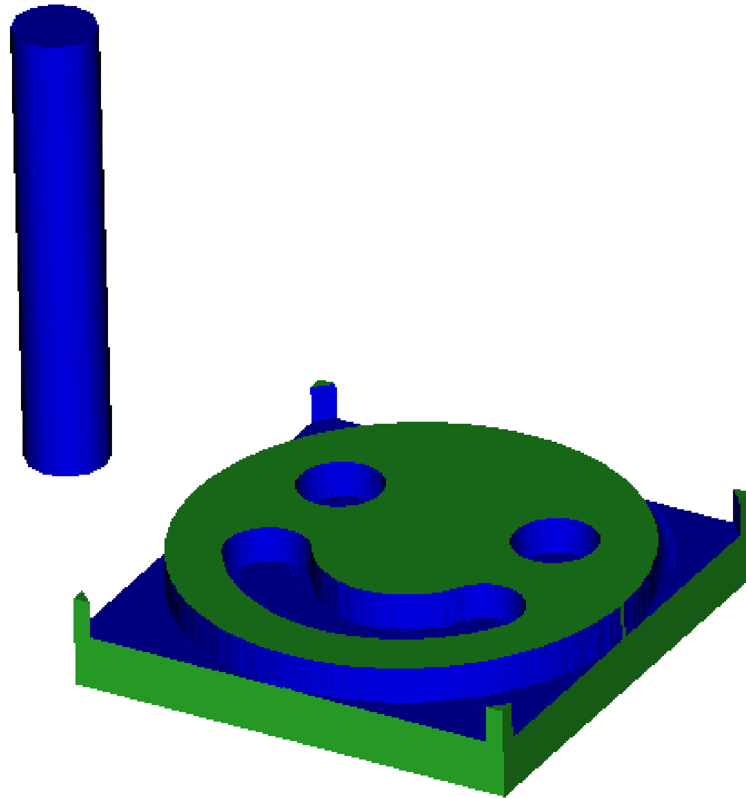
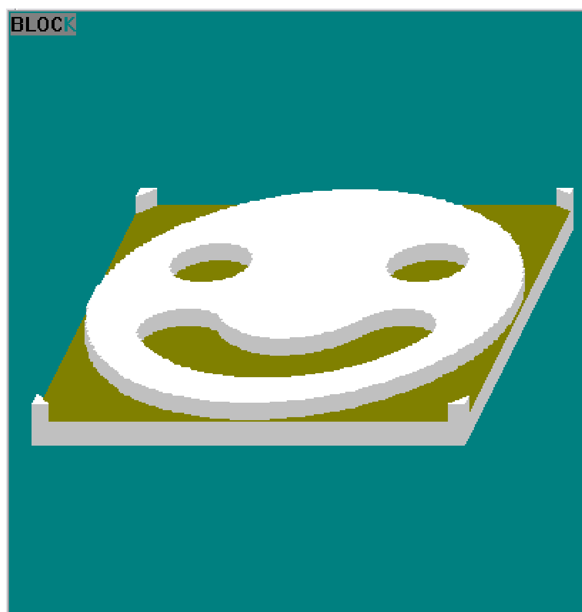


Figure 6. 8 The user interface for milling simulation before the simulation.



(a)



(b)

Figure 6. 9 Screen views after the completion of the milling simulation carried out in (a) CNCVerifier (b) CNCSimulator

CHAPTER 7

CONCLUSIONS & FUTURE WORK

The goal of this study was to develop a machining process simulator for milling and especially for turning operations. The lack of visual quality and inability of further development, existing programs limit the flexibility in the current studies. For instance, CNCSimulator is such a program. To overcome these limitations, CNCVerifier is developed.

For the simulation of turning operation, two separate techniques, pixel-based and polygon clipping techniques, are applied. In pixel-base technique, for workpiece and cutting tool, BitArray matrices were used. As a special data type in C#, BitArrays are very suitable to use in such Boolean applications. Their use made it easy to execute subtraction of cutting tool from the workpiece matrix by applying Logical "AND" operation. During the simulation, the cutting tool matrix is translated in space with respect to tool movements defined by the loaded part program.

Cutting tool is loaded to the program as a bmp picture file. It is converted into a Boolean matrix first and then another conversion took place to generate `BitArray` matrix. The workpiece is defined exactly the same way the cutting tool is done. The updated workpiece matrix is sent to a search algorithm to extract boundary points first, and then boundary points are inspected for their contribution to boundary line segments. If any is realized then boundary points are considered as endpoints of those boundary lines. This last operation is quite important, and the success of it

affects the performance of the simulation a lot. It is mainly because, the 3D generation of the workpiece is done as follow:

1. It is considered that there is a boundary line segment in between any two boundary points.
2. To obtain the 3D view of the workpiece, this boundary line is revolved about workpiece's Z-axis.
3. For the generation of this 3D model, mesh modeling is done using DirectX routines.
4. More the number of boundary segments, more the number of triangles in the mesh.
5. The rotation angle defines the number of triangles in the mesh. When the rotation angle gets smaller, the number of triangles in the mesh increases proportionally. The rotation angle is the slices in a full circle. The number of slices is defined as 36 at the beginning of the program as a default value, which corresponds a 10° rotation angle.
6. After making necessary calculations, mesh vertices and mesh indices are found and as a result, workpiece mesh is defined.
7. Finally, workpiece is shown in the 3D viewport of the user interface.

Because of the limitations in the boundary points search algorithm, for some cases it has been discovered that more than expected boundary segments are generated in the calculations which affects both the performance of the simulation and the quality of the final 3D view of the workpiece.

The current methodology applied for the definition of cutting tool and its geometry is a pictorial one. This affects the results of the simulation. The current format of the part program only allows utilizing a single cutting tool for the simulation.

All above mentioned limitations or drawbacks could be eliminated with the application of the below listed solutions which are thought to be studied as future work.

1. The search algorithm for boundary line segments could be improved. It is considered that a better search algorithm would improve the performance of the simulation which will yield decreased simulation times. In addition, a better search algorithm will eliminate unnecessary line segments, which will affect visual quality of the resultant workpiece in the better way.
2. A better cutting tool definition methodology could be developed. As explained in turning simulation chapter, polygon clipping methodology is considered to give better results. As a future work, this methodology is planned to be applied to obtain satisfactory results.
3. The part program could be in the format of standard ISO G code format. This would make the generated program to become a general purpose simulation media for the outputs of common CAM software.
4. Simulation resolution could be increased by increasing the matrix sizes. There is an upper bound, due to physical memory limitations, but any increase in matrix sizes would result in higher accuracy in the simulation. This will also increase the quality of the visual output, it will further decrease simulation speed.

The second technique applied for turning simulation was polygon clipping technique. In this case, workpiece and cutting tool are considered as polygons and they are modeled in that way. Polygon data is defined as an ordered list of vertices. For turning simulation with polygon clipping, an NC code interpreter is incorporated into the software. For that purpose, an NC code editor is added to the user interface and user was able to load NC files or type them in the editor. The code is processed by the algorithm and cutting tool location data is generated. Polygon clipping algorithm is applied in the steps listed below:

1. Workpiece and cutting tool boundary polygons are defined. For this purpose, workpiece dimensions are assigned as X and Z coordinate values of a rectangular polygon, and cutting tool data is retrieved from a tool library.
2. Before the application of clipping, the amount of cutting tool travel, in a single step, is applied as a transformation to the CTP by sweeping it. For the

sweeping, a new approach is developed. The result of that sweeping operation was also a polygon.

3. After that polygon clipping algorithm is executed and, WPP is clipped with CTP.
4. At the end the data available was a modified list of vertices and there were no need for any search algorithms. The data was sent to mesh data generation procedures.
5. Finally, not only 2D polygons of workpiece and cutting tool are drawn, but also 3D model of the workpiece is generated successfully.

Most of the difficulties and drawbacks of the pixel-based simulation technique are eliminated by polygon clipping technique. For turning simulation, the results obtained in CNCVerifier are much better than CNCSimulator. The program execution time, NC code definition and other user friendly features of CNCSimulator are replicated in CNCVerifier. The feature, which makes CNCVerifier more advanced than CNCSimulator, is its capability of generating 3D model of the workpiece. The model is shaded perfectly and it was possible to apply any viewing operations. It is also possible to render the workpiece in wireframe form.

For future work, the below listed subjects could be studied for the improvement of CNCVerifier.

1. Algorithms could be revised to increase simulation speeds.
2. In the current state, tools are just called from a library. An editor could be added for the generation of new tools.
3. Actually, mesh data generated in the program could be easily converted into stereolithography (STL) format. If any need arises in the future, it could be implemented to the algorithm to obtain a universally recognized copy of machined workpiece geometry.

For the milling simulation, dexel data structure is constructed. For the construction of dexel data structure, ray casting is applied on the 3D boundary surfaces of both

cutting tool and workpiece. Ray casting operation is done with the steps listed below.

1. All the screen pixel coordinates with respect to screen coordinate axes defined the X and Y coordinates of the rays. A suitable Z coordinate, away enough from surfaces of both workpiece and cutting tool objects, is determined to use as the starting point of the ray.
2. From these starting points, rays are casted parallel to the viewing direction targeting the objects.
3. The intersection points are collected in a matrix, in which lists of intersection points and other surface characteristics like color are hold.
4. The procedure is repeated for both workpiece and cutting tool.
5. After ray casting, the view on the screen is refreshed with the intersection points having the same visual characteristics of the objects. But from that point on, the viewing direction has to be kept.
6. The loaded part program is applied as in the turning simulation case.
7. The intersection points' Z coordinates which are stored in the dexel structure of the workpiece, are updated as a result of comparison of them with cutting tool's dexel data structure.
8. The procedure is repeated for any movement of the cutting tool.
9. The result is shown immediately on the screen inside the simulation viewport.

Dexel data structure is very easy to handle and the calculations are in the order of O(n). On the other hand, there are some drawbacks associated with this methodology:

1. The viewing direction is fixed. It has to be adjusted carefully before the simulation starts, to get the best visual output as a result of the simulation.
2. All the calculations have to be repeated to make the simulation from another viewing direction.

3. The resolution of the simulation is exactly the same of the screen resolution. There is no way to analyze the specific features of the machined object during or after the simulation.
4. There is no way to determine the volumetric material removal rate with this method. In future, it might be aimed to improve the capabilities of this program to make mechanistic analysis where current methodology won't let it be done.
5. In the current phase of the program, there is no way to make five-axis milling simulation.
6. The user interface is not adequate to provide functionality to the user.
7. Only the available option is to use flat end mill as cutting tool.
8. The part program format does not let the user to define NC data in the format of G codes.

To eliminate most of the deficiencies, in the below list, some alternative solutions are listed for future study on this subject.

1. With the application of a new data structure, all the drawbacks or limitations sourced by dexel modeling could be eliminated. For example, polygon clipping technique could be implemented in the milling simulation, too. For this purpose, the 3D models of the workpiece and cutting tool can be sliced by planar surfaces and the boundary generated by the slicing could be used as polygons. These polygons could be clipped and the resultant contours could be utilized to generate surface. If this approach could be realized, it is assumed that, the listed limitations below are tend to be resolved.

    a. Viewing direction limitation. Because the final view would be generated by the surface built on the polygons, it will be possible to view the final part from any direction.
    b. It will not be necessary to repeat the calculations in any change of the viewing direction.
    c. The resolution will not be affected by the screen resolution. Only the factor that will affect the accuracy will be the precision of the

calculations and the floating-point number representation errors due to machine epsilon.

d. The current polygon clipping algorithm used for turning simulation, gives the intersection of two polygons. This data is actually the material removed by the cutter in the sense of machining. Thus, by the application of polygon clipping, it is possible to determine the volumetric material remove rate. This will add mechanistic analysis to the program.

2. The user interface could be improved for more functionality. The arrangement of the buttons could be re-emphasized.
3. A cutting tool generation module could be integrated to the program. This will let the user to define new cutting tools, like ball end mill, slot mill, etc.

Although it is not within the scope of this study, some other features could be added the developed program in the future. These are listed as follows:

1. Other machining techniques could be integrated to the program.
2. The simulation results, if generated in surface format, could be saved in one of the universally recognized file formats like Initial Graphics Exchange Specification (IGES) format, STandardized Exchange of Product (STEP) format, STL format, etc. Thus, it would be reused for some specific purposes, like comparing it for verification.
3. Machine simulation could be done. For this purpose machine models should be prepared and loaded into the program. Also fixtures and clamping devices could be added to the model.
4. The simulation could be integrated with verification.
5. Tool path optimization could be done according to the result of verification routines.
6. Machining parameters input could be improved, if material removal rate could be calculated during the simulation.
7. Dynamic system model would be added as well.

REFERENCES

[1] H.B. Voelcker and W.A. Hunt., "The Role of Solid Modeling in Machining-Process Modeling and NC Verification", SAE technical Paper 810195, 1981.

[2] W.A. Hunt, and H.B. Voelcker, "An Exploratory Study of Automatic Verification of Programs for Numerically Controlled Machine Tools", Production Automation Project Tech Memo No.34, University of Rochester, Jan 1982.

[3] R. Frishdale, K.P. Cheng, D. Duncan, and W. Zucker, "Numerical Control Part Program Verification System", Proceedings of Conference on CAD/CAM Technology in Mechanical Engineering, MIT Press, Mar 1982. pp. 236-254.

[4] R.O. Anderson, "Detecting and Eliminating Collisions in NC Machining", Computer Aided Design, Vol. 10, No 4, 1978. pp. 231-237.

[5] W.P. Wang and K.K. Wang, "Geometric Modeling for Swept Volume of Moving Solids", IEEE Comuter Graphics & Applications, Vol 6, No 6, 1986. pp. 8-17.

[6] T.Van Hook, "Real-Time Shaded NC Milling Display", Computer Graphics (proc. SIGGRAPH), Vol 20, No 4, Aug 1986. pp. 15-20.

[7] P.R. Atherton, C. Earl, and C. Fred, "A Graphical Simulation System for Dynamic Five-Axis NC Verification", Proc. Autofact. SME, Dearborn, Mi., Nov 1987. pp. 2-1 – 2-12.

[8] Robert L. Drysdale, Robert B. Jerard, Barry Schaudt, and Ken Hauck, "Discrete Simulation of NC Machining", Algorithmica, No 4, 1989. pp. 33-60.

[9] Robert B. Jerard, Robert L. Drysdale, and Ken Hauck, "Geometric Simulation of Numerical Control Machining", ASME International Computers in Engineering Conference. San Fransisco, 1988.

[10] I.T. Chappel, "The Use of Vectors to Simulate Material Removal by Numerically Controlled Milling", Computer Aided Design, Vol 15, No 3, May 1983. pp 156-158.

[11] J.H. Oliver and E.D. Goodman, "Color Graphic Verification of NC Milling Programs for Sculptured Surface Parts". First Symposium on Integrated Intelligent Manufacturing. ASME Winter Annual Meeting, Anabein, Ca. 1986.

[12] Robert B. Jerard, S.Z. Hussaini, Robert L. Drysdale, and Barry Schaudt, "Approximate Methods for Simulation and Verification of Numerically Controlled Machining Programs", Visual Computer, No 5, 1989. pp 329-348.

[13] Robert L. Drysdale, Jerome L. Quinn, Kamran Ozair, and Robert B. Jerard, "Discrete Surface Representations for Simulation, Verification, and Generation of Numerical Conrol Programs", Proceedings of NSF Design and Manufacturing Systems Conference. 1991.

[14] Kamran Ozair, NC Machining Simulation and Verification Using Triangles Rather than Points. Honors Thesis, Dartmouth College, 1990.

[15] D. Jang, K. Kim, and J. Jung, "Voxel-Based Virtual Multi-Axis Machining", The International Journal of Advanced Manufacturing Tehnology, Volume 16, No 10, pp. 709-1713

[16] Hong T. Yau, Tsou S. Lee, and Yu C. Tong, "Adaptive NC Simulation for Multi-axis Solid Machining", Computer-Aided Design & Applications, Vol. 2, Nos. 1-4, 2005, pp. 95-104.

[17] Microsoft Developers Network, http://msdn.microsoft.com/en-us/library/system.collections.bitarray.aspx, last visited on August 2010.

[18] DirectX 9.0 for Managed Code, Microsoft 2005.

[19] Roth D. Scott, "Ray Casting for Modeling Solids", Journal, Computer Graphics and Image Processing, Vol. 18, No. 2, 1980, pp. 109—144.

[20] Barycentric Coordinate System, http://en.wikipedia.org/wiki/Barycentric_coordinate_system_%28mathematics%29, last visited on September 2010.

[21] Francisco Martinez, Antonio Jesus Rueda, Francisco Ramon Feito, "A new algorithm for computing Boolean operations on polygons", Computers & GeoSciences, Volume 35, issue 6, 2008, pp. 1177-1185.

[22] Preparata, F., Shamos, M., Computational Geometry an Introduction, Springer, New York, 1985, 398 pp.

[23] Schneider, P.J., Eberly, D.H., Geometric Tools for Computer Graphics, Elsevier Science, San Francisco, 2003, 1060 pp.

[24] Andereev, R.D., “Algorithm for clipping arbitrary polygons”, Computer Graphics Forum 8 (2), 1989, pp. 183–191.

[25] Sutherland, I.E., Hodgeman, G.W., “Reentrant polygon clipping”, Communications of the Association for Computing Machinery, 1974, 17 (1), pp. 32–42.

[26] Liang, Y.D., Barsky, B.A., “An analysis and algorithm for polygon clipping”, Communications of the Association for Computing Machinery, 1983, 26 (11), pp. 868–877.

[27] Weiler, K., “Polygon comparison using a graph representation”, Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), Seattle, Washington, USA, 1980, pp. 10–18.

[28] Greiner, G., Hormann, K., ”Efficient clipping of arbitrary polygons”, Association for Computing Machinery — TransactionsonGraphics 17 (2), 1998, pp. 71–83.

[29] Liu, Y.K., Wang,X.Q., Bao,S.Z., Gombôi, M.,Zâlik, B., “An algorithm for polygon clipping, and for determining polygon intersection sand unions”, Computers&Geosciences, 33, 2007, pp. 589–598.

[30] Vatti, B.R., “A generic solution to polygon clipping”, Communications of the Association for Computing Machinery, 35 (7), 1992, pp. 56–63.

[31] CNCSimulator, http://www.cncsimulator.com/, last visited: December 8, 2010.

[32] CSG Tree, http://en.wikipedia.org/wiki/Constructive_solid_geometry, last visited: December 8, 2010.

[33] Maeng, S.R., Baek, N., Shin, S.Y., Choi, B.K., “A Z-map update method for linearly moving tools”, Computer Aided Design, Volume 35, No.11, 2003, pp 995-1009.

[34] Karunakaran, K.P., Shringi, R., Singh, “Virtual Machining”, Modern Machine Tools, Vol 1 No. 3, 2004, pp. 62-68.

[35] Quinn, J.L., “Accurate Verification of Five-Axis Numerically Controlled Machining”, Ph.D. Thesis, Dartmouth College, United States, May 1997.

[36] Catmull, E., “A hidden-surface algorithm with anti-aliasing”, ACM SIGGRAPH Computer Graphics, Vol. 12 Issue 3, Aug. 1978, pp.6 6-11.

[37] Atherton, P.R. “A scan-line hidden surface removal procedure for constructive solid geometry”, Computer Graphics, Vol. 17 Number 3, 1983, pp. 73-82.

[38] Dölen, M., "Lecture Notes on ME440: Numerically Controlled Machine Tools", METU, Mechanical Engineering Department, 2010

[39] Chapra, Steven C. and Canale, Raymond P. "Numerical Methods for Engineers", Mc Graw Hill, 2002.

APPENDIX A

NC PROGRAM LISTINGS

A.1 NC program used in Case Study 1

```
N10 T26 M03
N20 G00 X54 Z100
N30 S2000 F300 M8
N40 G01 X-2
N50 G00 X0 Z102
N60 G00 X50
N70 G01 Z0.5
N80 G00 Z12 X52 M09
N90 T21
N100 G00 X0 Z102 M08
N110 G01 Z50 F100
N115 G00 Z102 M09
N120 T09
N130 G00 X52 Z40 M08
N140 G01 X32
N150 G00 X52
N160 G00 Z37
N170 G01 X32
N180 G00 X52
N190 G00 Z34
N200 G01 X32
N210 G00 X52
N220 G00 Z31
N230 G01 X30
N240 G00 X52
N250 G00 Z28
N260 G01 X30
N270 G00 X52
N280 G00 Z25
N290 G01 X30
N300 G00 X52
N310 G00 Z22
```

```
N320 G01 X30
N330 G00 X52
N340 G00 Z19
N350 G01 X30
N360 G00 X52
N370 G00 Z16
N380 G01 X30
N390 G00 X52
N400 G00 Z13
N410 G01 X30
N420 G00 X52
N430 T2
N435 G00 X40 Z100
N440 G03 X50 Z95 I0 K-5
N441 G00 X52
N442 G00 Z14
N443 G00 X40
N444 G01 X40 Z13
N445 G03 X50 Z8 I0 K-5
N450 T23
N455 G00 X40 Z103
N470 G01 X36 Z98
N480 G01 Z82
N481 G01 X38 Z80
N482 G01 Z62
N483 G01 X36 Z60
N484 G01 X30
N485 G00 Z100 M09
N490 T24
N500 G00 Z34 X52 M08
N510 G00 X34
N520 G01 X30
N530 G01 Z40
N540 G03 X40 Z45 I5 K0
N550 G02 X50 Z50 I0 K5
N555 G01 Z51
N600 M30
```

A.2 NC Program used in Case Study 2

```
N05 T2
N10 G00 X27 Z105
N20 G01 Z100 F200 S1500 M03 M08
N30 G01 X-2
N40 G00 X4 Z104 M09
N50 G00 X28 Z104
N55 T1
```

```
N60 G00 X26
N70 G01 Z31.732 M08
N80 G01 X27 Z32.732
N90 G00 Z106 M09
N100 G00 X24
N110 G01 Z32.172 M08
N120 G01 X25 Z33.172
N130 G00 Z106 M09
N140 G00 X22
N150 G01 Z50.472 M08
N160 G01 X23 Z51.472
N170 G00 Z106 M09
N180 G00 X20
N190 G01 Z54.472 M08
N200 G01 X21 Z55.472
N210 G00 Z106 M09
N220 G00 X18
N230 G01 Z58.472 M08
N240 G01 X19 Z59.472
N250 G00 Z106 M09
N260 G00 X16
N270 G01 Z62.472 M08
N280 G01 X17 Z63.472
N290 G00 Z106 M09
N300 G00 X14
N310 G01 Z66.472 M08
N320 G01 X15 Z67.472
N330 G00 Z106 M09
N340 G00 X12
N350 G01 Z98.828 M08
N360 G01 X13 Z99.828
N370 G00 X7.5 Z101 M08
N390 G01 X10 Z98
N400 G01 Z70
N410 G01 X20 Z50
N420 G01 Z35
N430 G02 X25 Z30 I5 K0
N440 G01 Z0
N460 G00 X28 Z6 M09
N470 G00 Z100
N520 M30
```

A.3 NC Program used in Milling Simulation-Sample Application

```
N01 G21 G90 G0 X-30 Y50 T15 S1200 M3
N02 G0 X-9.0 Z-8.0
N03 M8
```

```
N04 G17 G2 I59.0 J0 F400.0
N05 G0 Z5.0
N06 X25.0 Y65.0
N08 G1 Z-8.0
N09 Z5.0
N10 G0 X75.0
N11 G1 Z-8.0
N12 Z5.0
N13 G0 Y35.0
N14 G1 Z-8.0
N15 G2 X25.0 Y35.0 I-25.0 J15.0
N16 G1 Z5.0
N17 G0 X-30.0 Y50.0 M9
N18 Z0 M5
N19 M30
```