12-2011

Automated Digital Machining for Parallel Processors

Joshua Tarbutton
Clemson University, tarbutton@gmail.com

Automated Digital Machining for Parallel Processors

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mechanical Engineering

---

by
Joshua Aaron Tarbutton
August 2011

Accepted by:
Dr. Thomas R. Kurfess, Committee Chair
Dr. Tommy Tucker
Dr. John Ziegert
Dr. Yong Huang

ABSTRACT

When a process engineer creates a tool path, a number of fixed decisions are made that inevitably produce sub-optimal results. This is because it is impossible to process all of the available information and tradeoffs a priori. The research presents a methodology to support a process engineer's attempt to create optimal tool paths by performing automated digital machining and tool path analysis. This methodology automatically generates and evaluates tool paths based on parallel processing of volumetric digital part models. Generalized cutting geometry is used to generate the tool paths permitting on an infinite range of cutting possibilities. Digital part models are created by voxelizing STL files. The digital part surfaces are obtained based on the intersection of the model and rays cast from an arbitrary plane.

Tool path points are based on a general path template and updated based on the generalized tool geometry and part surface information. The distance traveled and the material removed by the generalized cutter as it follows the path is used to obtain 3D path metrics. The paths are evaluated based on the path metrics of material removal rate, machining time, and scallop height. This methodology is a parallel processing accelerated framework suitable for generating tool paths in parallel enabling the process engineer to rank and select the best tool path for the job.

The parallel processing framework developed in this research was implemented on a single GPU and resulted in a 500X increase in speed compared to CPU benchmarks. Tool paths generated from this framework were post-processed into G-code and representative parts were machined.

DEDICATION

This dissertation is dedicated to God who has given us a world of limitless exploration, to my wife Antonina whose love and support made this work possible, to my daughters Alette and Violet, and my sons Noah and Elliot for their patience through this process and for all the giggles, hugs, and smiles along the way.

ACKNOWLEDGMENTS

I would like to thank Dr. Kurfess for nearly eight years of investment into my life through teaching, advising, and putting me in situations that caused me to grow. I would also like to thank Dr. Tucker for providing me the base code for this dissertation and advice on how to step through this process of research code development. I would also like to thank Dima Konobrytskyi for his help through some difficult software bumps and bugs along the way. I would also like to thank the staff at CGEC who supported my work as well as all of my friends and family who didn’t hear from me as often as they should have.

TABLE OF CONTENTS

Table of Contents (Continued)

Table of Contents (Continued)

Page

LIST OF TABLES

LIST OF FIGURES

List of Figures (Continued)

List of Figures (Continued)

Figure Page

List of Figures (Continued)

Figure Page

List of Figures (Continued)

List of Figures (Continued)

List of Figures (Continued)

Figure Page

CHAPTER ONE

INTRODUCTION

The last forty years of research have transformed manufacturing considerably. Perhaps the most significant change was the introduction of numerically controlled machine tools that increased productivity and quality exponentially. Machine operators received specific training to be able to program these machines. However, after years of experience and training, machine operators still often used trial and error methods to produce the desired part. Significant time investments went into making these parts.

With the introduction of Computer Aided Design (CAD) packages, increasingly complicated parts were introduced further burdening Computer Numerically Controlled (CNC) programmers. Computer Aided Manufacturing (CAM) packages were introduced to provide process planners and machine tool operators the necessary support to generate machine tool code from based on part surfaces. CAM packages were used as early as 1971 by Bezier at Renault in the production car bodies and tools. However, CAM technology has taken years to mature and still requires significant experience to be used properly. The CAM packages to date are quite impressive; a manufacturing engineer can take nearly any part and produce a gouge-free and collision-free tool path for a machine tool with the desired surface accuracy.

However, instead of automating the process, CAM packages have essentially taken the experienced operator from the shop floor to behind a computer. Often, a second operator is needed to run the machine. The process engineer is now expected to produce more complex parts by using these tools. However, a major limitation is that process engineer has to rely on years of manufacturing experience to be successful. Often, due to manufacturing deadlines, the CAM user only has one chance to produce the tool path.

The Need for Automated Path Planning

The greatest problem with CAM systems is that they are linear. A user provides the input parameters (cutter size, blank size, path width, depth of cut, and etc.) and an output is generated. Therefore, the quality of the output is directly related to the user input when it should be tied instead to the features of the part. The traditional workflow from "Art to Part" is shown below.

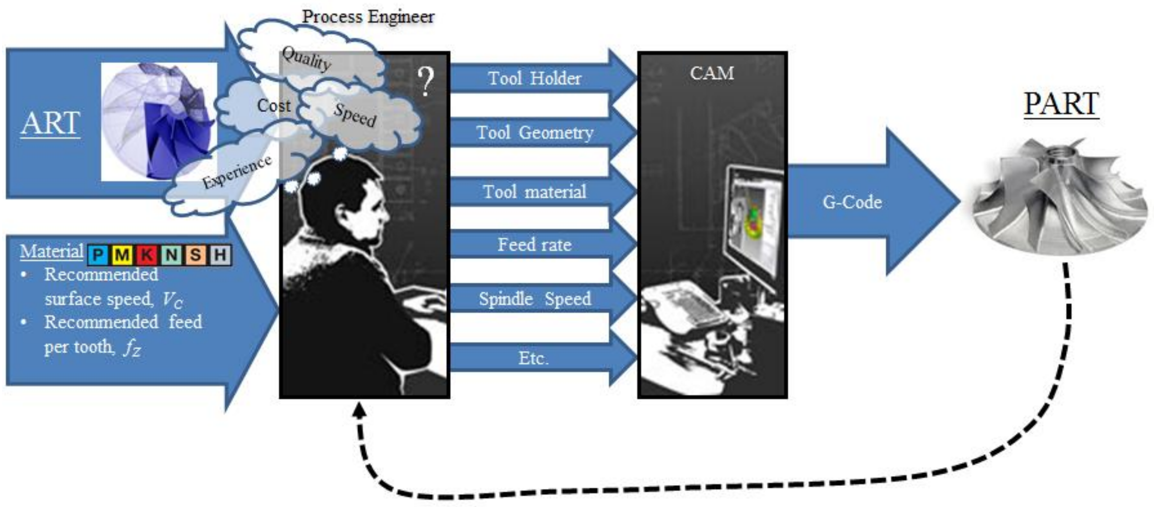


Figure 1-1: Traditional machining workflow

In addition, there are many options available to the process planner that are not fully considered due to a lack of time and analytical capability. Furthermore, new database and information sets (e.g., online tooling, coolant, and machine tool specification catalogues) are available that provide even more options for planners.

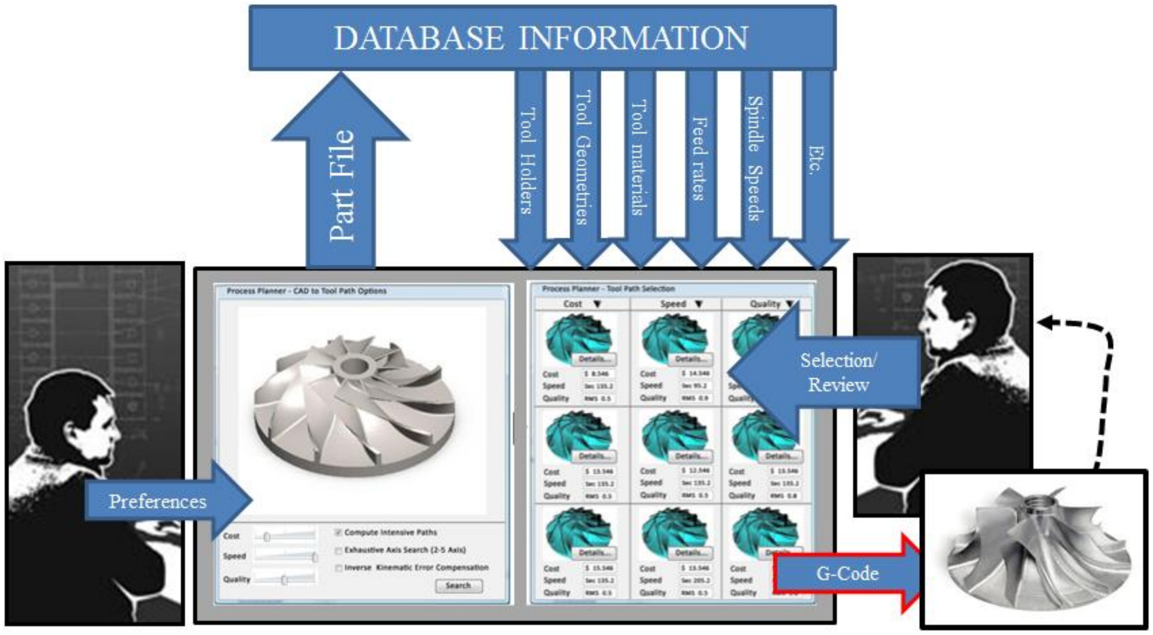


Figure 1-2: Automatic path planning

The increase of such information is overwhelming and is typically not used to its fullest extent. Even if a process engineer was capable of memorizing every available option for machining, it would be impossible and impractical to evaluate the tradeoffs between approaches due to a fundamental clash of optimality. Optimal tool paths ideally meet the conflicting objectives of cost, speed, and quality. Automation of this process is necessary to propel machining into the next era of manufacturing.

Figure 1-2 shows a how such a tool might be used by the process engineer. He or she would input a part file and preferences into the software support tool in terms of cost, speed and quality. The part file would be used to run hundreds of thousands of different tool paths on the part based tooling catalogues, material databases, and ranges of acceptable feeds and speeds. The software tool would then rank them based on the desired preferences.

This dissertation presents the necessary methodology and parallel processing framework to support automatic path planning for multi-axis machine tools to enable rapid evaluation and selection of tool paths based on preference. This methodology utilizes sophisticated algorithms that are designed to have exceptional performance when run using parallel code to calculate tool path trajectories automatically from a part model.

The input to this methodology is a part model in the standard STL file format. The output is a simulation of the cutting process, path metrics to evaluate the path performance, and G-Code that can be run on a production milling machine. This framework gives process engineers the ability to load an arbitrary part file with complex freeform shapes and generate G-code without ever using a CAM (computer aided manufacturing) system. In addition, this framework allows multiple tool paths to be generated based on a generalized cutter allowing for optimization of machining time, material removal rate, and scallop.

Chapter 2 provides background information on machining and the fundamental role of the tool path. The traditional path planning approaches and limitations are presented and discussed. The methodology described in this dissertation utilizes parallel processing to accelerate the algorithms. The GPU is used in this research as the parallel co-processor and the hardware, motivation, and parallel processing performance of the

GPU are presented. Chapter 3 presents the following core building blocks of this research:

- A voxel representation of a part and blank model and the voxelization process.
- A surface detection algorithm based on ray-casting and optimal cutting plane selection for 2+3 machining.
- A generalized cutter representation.
- A path creation algorithm.
- A path adjustment algorithm based on the generalized cutter.
- A blank model update algorithm based on the voxel distance to the cutter.

Chapter 4 discusses the hardware and performance of the core functions introduced in Chapter 3. The performance of the developed algorithms discussed in terms of time and memory. Chapter 5 presents the traditional path planning approach and path metrics and demonstrates how the parallel framework introduced in Chapter 3 can be used to obtain path metrics. Chapter 5 also discusses the potential to generate hundreds to thousands of tool paths using this framework to support process planning based on cost, time, and quality preferences. Chapter 6 presents experimental results of machining for both 3-axis and 2+3-axis parts. Chapter 7 concludes this dissertation with a summary of the major contributions of this work which are:

- A parallel accelerated processing framework to:
    - Create and utilize digital voxel models to represent part and blank models.
    - Utilize a ray-casting approach for accurate surface detection.
    - Generate a tool path based on a digital part and generalized cutter.
    - Simulate the removal of material.
    - Analyze the path metrics.

CHAPTER TWO

BACKGROUND

In this chapter the relevant background information for tool path planning and GPU hardware is presented. Classical tool path planning methods are first introduced followed by the current research in the area. The history of GPU hardware development is presented with specific examples of its uses in manufacturing and path planning.

Traditional tool path planning

Tool path planning is the art of transforming a surface representation from a part model into a series of commands that can be given to an arbitrary machine tool to produce a real part. Tool path planning plays a central role in the manufacturing sector and affects nearly every part found in the automobile, aviation, and shipping industries. Figure 2-1 show typical milling components highlighting the tool path.

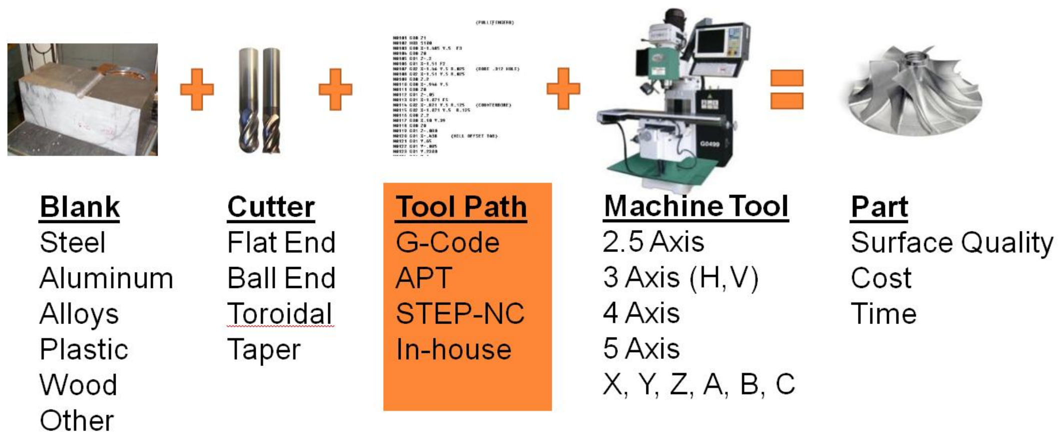


Figure 2-1: Typical milling operations

CAD packages have continually been increasing core functionality to represent and join multiple complex surfaces into single part models. CAD designers are taking advantage of this to design parts with higher surface complexity than ever before. CAM packages attempt to formulate methods to automatically generate tool paths based on these part models.

The goal of all tool path planning approaches is to generate accurate Cutter Location data (CL) in Euclidian space based on accurate surface Cutter Contact (CC) points so that the tool path produces the desired part surface without gouge or collision. The CL is the center point of the geometry used to describe the tool. For a flat end cutter shown in Figure 2-2 the CL is located at the center of the bottom of the tool. For a ball end cutter the CL is located at the center of the ball. The CC is the point on the surface of the tool that is in contact with the surface of the part.

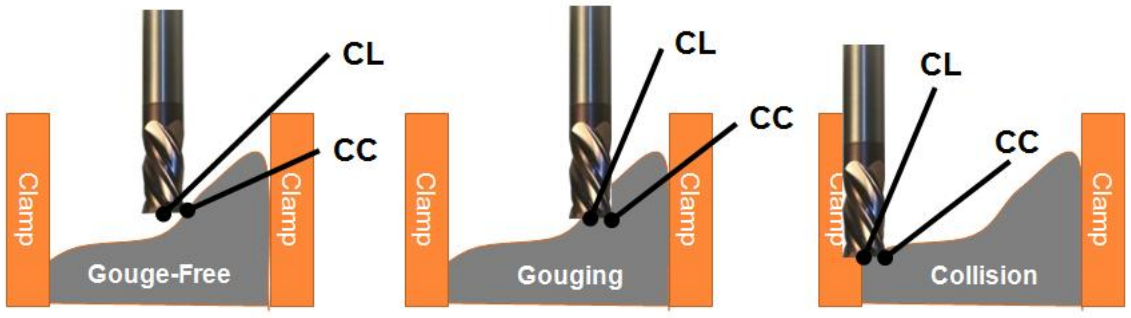


Figure 2-2: Tool path CC and CL points

The majority of tool path planning algorithms are based on these well-established methods: iso-parametric [1-3], iso-planar [4-6], iso-scallop [7, 8], and polyhedral. Iso-parametric and Iso-planar are the most widely used methods in CAM systems [9] but some of the most recent commercial CAM systems also use polyhedral machining [10].

Iso-parametric Path Planning

Iso-parametric path planning methods are the mapping of a parametric surface, s(u,v), onto Euclidean space. One of the parameters u, or v, of the surface is held constant while the other is incremented resulting in a path mapped out in Cartesian space. This mapped parametric path yields the cutter contact point, or CC. Then, the other iso-parameter is incremented and the process is repeated until the entire surface has been traversed. Once the CC points are obtained, an offset surface is created from the CC points to yield cutter contact, or CC, data [6]. A main drawback of this method is that the mapping of surfaces with large surface gradients results in large gaps between successive paths in Euclidian space as shown in Figure 2-3. Adaptive methods have been introduced to correct this phenomenon but at the cost of increased algorithm complexity [2].

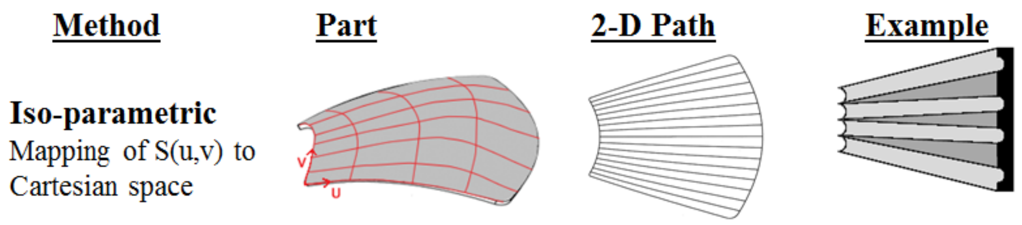


Figure 2-3: Iso-parametric path planning

Iso-parametric methods are useful when there is only one surface but multiple surfaces often result in gouging [7] and compound surfaces are commonly converted to triangular meshes [11]. When the underlying free-form surfaces are extend and combined, as is normally the case in complex surface modeling, the tool paths generated by the iso-parametric tool paths often are no longer boundary confined [11] and surface repair is necessary to ensure error-free tool paths [9].

Iso-planar Path Planning

Iso-planar methods (also referred to as drive surfaces) generate parametric curves by intersecting the part surface by infinite planes as shown in Figure 2-4. If the plane is vertical, the method is referred to as the Cartesian or constant z-level method. If the plane is horizontal, it is referred to as the contour parallel offset or CPO method, direction-parallel [12], one-way tool path, or zigzag tool path [13]. The plane-surface intersections are used to create parametric curves which are used to generate CC points and CL points.

According to Kim [13], determining the plane-surface intersections is very computationally demanding but, because the algorithm is very reliable when used for complex surfaces, it is still widely used in die and mold manufacturing. A disadvantage of this approach is that parts with large surface gradients will produce uneven paths if the intersecting planes are spaced evenly and adaptive methods must be used to account for these regions [14].

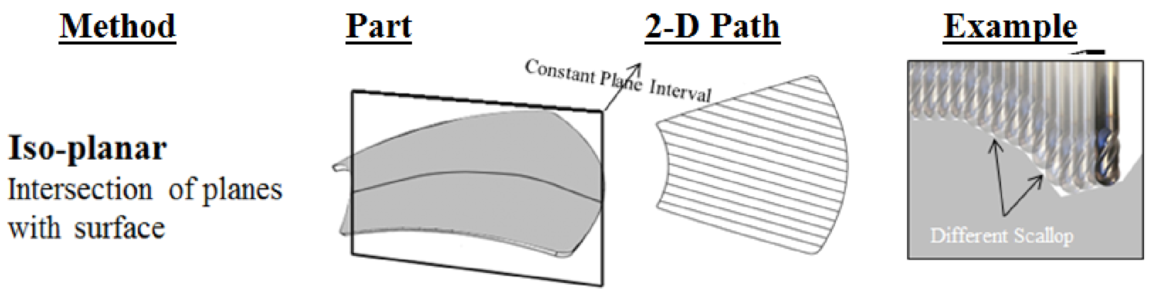


Figure 2-4: Iso-planar path planning

Iso-scallop Path Planning

Iso-scallop methods are extensions of the above two approaches and shown in Figure 2-5. The distance between the iso-plane is adjusted so that accounts for steep surfaces and results in uniform scallop height when the iso-scallop method is based on the iso-plane method. The parametric surface is manipulated so that the resulting tool paths are mapped uniformly to Euclidian space when the iso-scallop method is applied to the iso-parametric method.

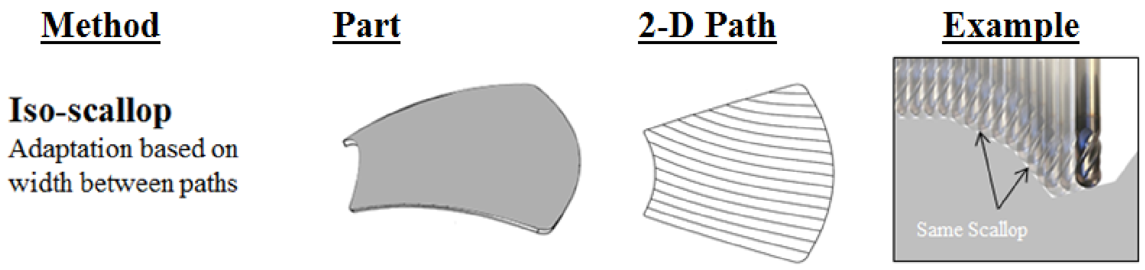


Figure 2-5: Iso-scallop path planning

Polyhedral Path Planning

Polyhedral machining was introduced by Duncan in 1983 [15] and is superior to the computationally intensive surface checking algorithms found in the above methods [10]. However, polyhedral machining was not widely used simply because of hardware cost and memory limitations; which are much less relevant with today's hardware. Polyhedral machining creates a tool path based on surfaces represented by polyhedrals. Polyhedral surface models are generated by tessellating a part's surfaces.

Nearly all CAD packages have tessellation algorithms that produce tessellated surfaces with the desired surface accuracy. Although the tessellated surface is an approximation of the true surface, the surface accuracy can be constrained to whatever tolerance is acceptable at the cost of more memory. The tessellation algorithms are extremely robust and capable of combining any number of surfaces into a single triangular mesh [11].

Tessellated surfaces have been widely used for rapid prototype systems where the surface accuracy of the final part is much less important than the creation of an initial representative part. In addition, the de-facto standard file format for rapid prototyping systems is the STL (Stereolithography) file format; which is little more than a header with a list of triangle vertices and their normal. Because hardware limitations are no longer an issue, polyhedral machining has made its way into commercial CAM packages [10].

Determining the CC and CL from an STL model has been accomplished by two main approaches (shown in Figure 2-6) which are: the point/curve-based approach [6, 10, 15-17], the inverse tool offset surface approach [18-20]. The point/curve based approach finds CL points based “dropping” the tool along the z-axis until it intersects the part surface. The inverse offset method places the tool upside down (inverse) with it’s center on the surface of the part and uses the tool’s tip (offset) to determine the CL data. These methods suffer from gouge when the surface geometry is concave. To correct for this concave regions must be detected and eliminated which proves to be computationally demanding and difficult [21].

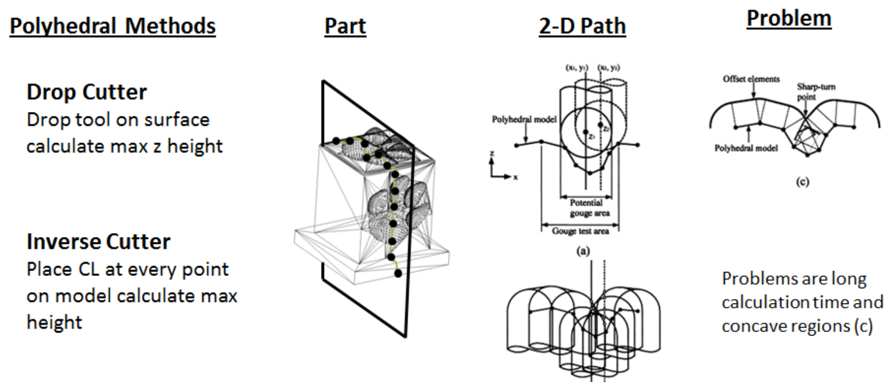


Figure 2-6: Polyhedral machining

Introduction to the GPU

Until recently, the advances in the GPU were strictly limited to games and animated movies. However, graphics card manufacturers realized the demand to use the greater computing capabilities afforded by the GPU. In 2005, graphics card manufacturers opened up the graphics card “pipeline” allowing programmers to use portions of the graphics as general purpose processors.

The programmer was granted the access and ability to manipulate raw vertex and pixel information stored in graphics memory. With this new capability, research exploded in what is commonly known as “General Purpose Graphics Processing Unit” computing or GPGPU computing. A tremendous amount of research in non-graphics fields such as physics, economics, medicine, and engineering soon followed [22]. Simulations that took days or that were simply unrealistic became tasks that could be completed in a fraction of the time due to the extremely low cost per floating point operation of the readily available and easily integrated graphics cards.

Subsequent to the availability of programmable GPUs, applications beyond graphics, were developed for the GPU. Naga K Govindaraju et. al. [23] implemented LU decomposition and Jin Hyuk Jung [24] implemented Cholesky decomposition on the GPU for use in linear algebra applications. A GPU finite element method to solve linear heat equations was demonstrated by Rumpf and Strzodka [25]. Their approach utilizes the hardware based pixel manipulation functions and graphics memory environment functions to not only perform algebraic operations but also optimize operations by reducing the number of rendering passes. They achieved more than 300 million operations per second (MOPS) in a single iteration of their Jacobi solver.

Later generation graphics cards continue to extend this performance. For example, the nVIDIA Tesla can perform more than 135 GFLOPS (135×10^9 Floating Point Operations Per Second), yielding capabilities significantly more advanced than highest end commercial CPUs (e.g. 70 GFLOPS Intel Quad Core). At this point, GPUs can run physics-based simulations as required for a computer game at several orders of magnitude faster than a CPU or even multiple CPUs. GPU advances are on a pace widely referred as “Moore’s Law Cubed” referring to the common perception that computing power doubles every eighteen months [26]. This trend is seen in Figure 2-8 where the data memory bandwidth (GB/s) and processing speed (FLOPS) are presented over time.

The CPU executes sequential code on general hardware suitable for a variety of computing tasks to support a wide range of software whereas the GPU executes parallel code on hardware dedicated to operating on millions of floating point numbers to support code specifically written to exploit the GPU. The CPU is designed so that each processing thread can deal with very large amounts of data. The latest CPUs allow for up to 8 simultaneous threads. The GPU is designed to process thousands of threads and each thread is significantly limited in the memory it can use.

The GPU supports parallel operations on multiple Streaming Multiprocessor Cores. These are similar to CPU processors except in their density. Current high-end CPUs have 2 – 4 cores (e.g. Intel Duo, Intel Quad) running at approximately 3GHz whereas current GPUs have 64 – 512 processor cores operating at speeds up to 1.5GHz. There is a tremendous difference in the computational horsepower of the CPU vs. GPU. As demonstrated in the next section, the nominal acceleration in computing applications afforded by the GPU is 10-100x’s [27] the speed up of the CPU which can result in significant implications for manufacturing.

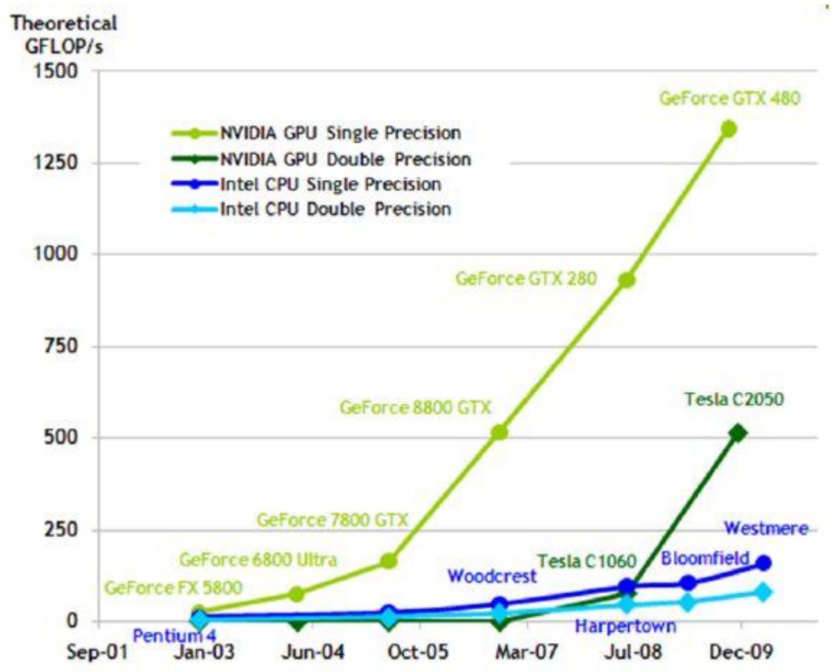


Figure 2-7: GPU peak floating point operations per second over time

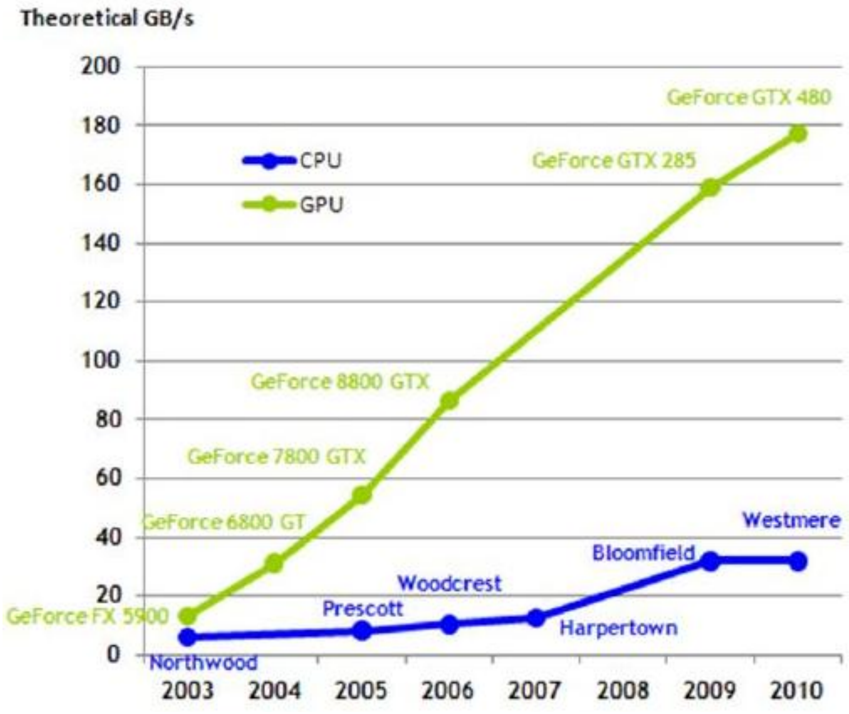


Figure 2-8: GPU memory bandwidth over time

GPU Accelerated Machining Research

Most CAD packages render models on the screen via a triangle mesh approximation of the actual model. Calculating the physics of light that interact with these triangles to obtain the color values for the millions of pixels on the screen is computationally expensive. GPUs are specially developed hardware used to render such images at the rate of 30-60 Hz required to satisfy the human eye. Prior to this decade, the GPU had a fixed set of operations or “pipeline” through which it processed multiple vertices and pixels in parallel using standard lighting physics.

A depth buffer, also called the z-buffer, is an integral part of the GPU’s ability to accelerate 3D rendering in graphics. The depth buffer is like an image that stores a number describing the depth of the object at each pixel rather than a color. It stores the distance from the eye to each visible point in the image and is primarily used for hidden surface removal also called “z culling.” Objects in a 3D scene are each given an opportunity to set the color value at a given pixel. When the rendering calculations determine an object may cover a particular pixel, the object is drawn at this pixel only if its depth value is less than the depth value currently set for the pixel. Thus, a depth buffer plays an instrumental role in avoiding draw calls for objects occluded by other objects having nearer depth values.

Techniques utilizing the depth buffer have been developed to determine the castability of geometric parts and assist with part redesign by McMains, Khardekar and Burton [28]. These algorithms are capable of identifying and displaying undercuts, and minimum and insufficient draft angles. In this approach, they utilize the occlusion query functionality (depth buffer) of the graphics card to determine whether or not a part is castable in a given direction when the object is viewed from the mold removal direction. This involves rendering the part geometry and storing the distance to the visible part facets also called “up-facets” from the eye-point in the depth buffer. The geometry is then re-rendered with the depth test enabled. In this pass, only those portions of the up-facets that were invisible earlier are rendered. Thus, if any pixels are rendered in the second pass, the object is not castable in that direction. The implementation of this algorithm on a NVIDIA Quadro FX 3000 series GPU was shown to determine the castability of parts with over 20,000 faces in less than a millisecond per direction. The algorithm is further extended to highlight the non-castable features of the part to allow the designer to make the necessary changes. The depth buffer obtained from the first rendering pass is transferred to a depth texture, and the orthogonal viewing matrix for the current camera position is saved. This procedure is repeated from the opposite casting direction, permitting the designer to rotate the object and examine the undercuts in real time.

A graphics hardware–assisted approach to 5-axis surface machining was developed by Gray et. al. [29] to build upon a tool positioning strategy called the Rolling Ball Method. In this method, a ball of varying radius is rolled along the tool path such that a small portion of the surface of the ball is used to approximate a small portion of the surface being machined by the cutting tool. The radius of the rolling ball is computed by checking a grid of points in the shadow of the cutting tool on the work piece surface. The hidden surface removal property of the depth buffer is used to expedite this checking process by clipping the part of work piece that is not in the shadow of the tool. This approach was shown to eliminate the need for parameterization of the surface to be machined, and enable the machining of multiple patch-triangulated surfaces. This method was extended to the Arc Intersection Method [30] by the same authors to ensure that the tool contacts the surface at two points in the cutting process.

Dokken et. al. [31] utilize the GPU as an accelerator for CAD-type intersection algorithms. They propose the use of the fragment processor for subdividing surfaces to create a tight hull containing the surface, and a tight hull containing the normal field of the surface. The depth buffer is used as a tool for testing for overlap of the extent of hulls containing the surfaces. For two surfaces to intersect in a closed loop, their normal fields must overlap. The approach employs the depth buffer to determine the intersection of hulls containing the normal fields of various surfaces. A set of hulls that do not intersect, eliminates the possibility of closed intersection curves.

Roth et. al.[32, 33] propose a mechanistic model, based on an adaptive and local depth buffer, to compute milling forces when machining a part on a multi-axis milling machine. Their paper presents a novel method of calculating chip geometry and volume of material removed during machining in order to determine the cutting forces. The terms “adaptive” and “local” depth buffers are used as the depth buffer is changed to be constantly aligned with the tool axis and sized to the tool instead of the work piece respectively. Previous tool positions are rendered to the scene and the depth buffer is stored. The current tool position is then rendered and the depth buffer is stored again to obtain the in-process chip geometry, which is the difference between the two states of the depth buffer. This model is limited to only flat end mills and it is inefficient as most of the depth buffer holds previous tool positions as it is sized to cover the tool. To address these shortcomings the depth buffer is sized to the cutter teeth, thereby improving the memory requirements resulting in efficient usage. The updated implementation permits application to more complex tool shapes.

Inui [20] implemented the inverse offset method mentioned in the beginning of this chapter by using the depth buffer of the graphics cards to generate tool paths for mold and die manufacturing at Mazda [34]. The inverse offset is generated by using the depth information of the graphics card and is computed based on the cutter geometry. Inui also used the GPU to determine the optimal angle in deep mold milling. Carter [35] used GPU fragment shader programs to determine CL data from a surface generated by the depth buffer. In this approach, the part surface is determined by using depth buffer information and then the CL data is generated by super-sampling the surrounding area and calculated CL points based on the tool nose radius.

This research introduces an alternative tool path planning approach based on ray-casting and voxel models, both of which are concepts found ubiquitously in computer graphics. Voxel models have been used extensively in the area of “Volume Rendering” which is concerned with rendering, in a very realistic way, a scene with volumetric elements. Applications of volume rendering are predominantly concerned with the rendering of objects with various densities such as clouds and smoke for the movie and gaming industries or for rendering the bone and organ data obtained from an MRI [36]. Ray-casting is a cornerstone of graphics research [37, 38] and is used in CAD systems to render the modeled object to the GUI or “scene” [39]. Ray-casting is used in photo realistic rendering as a means to render a virtual scene as close to reality as possible by physically modeling rays of light [40]. These concepts are used to create the state of the art games and movies enjoyed by many.

The concept of ray casting has previously been used in tool path planning to find the highest patch surface in a work area [41] or specific intersection with a surface [42]. Voxel models have been used in virtual machining simulation [43] but have not been used to generate tool paths. Moreover, the method of using a voxel model and casting rays to detect surfaces has never been done. In this research, a STL file is used as the basis for voxelization and rays are cast at the iso-surface of the voxel model for fast surface detection. Tool paths are generated from these intersections by either traditional milling or climb milling. Experiments are carried out on a production machine tool demonstrating the results of this approach.

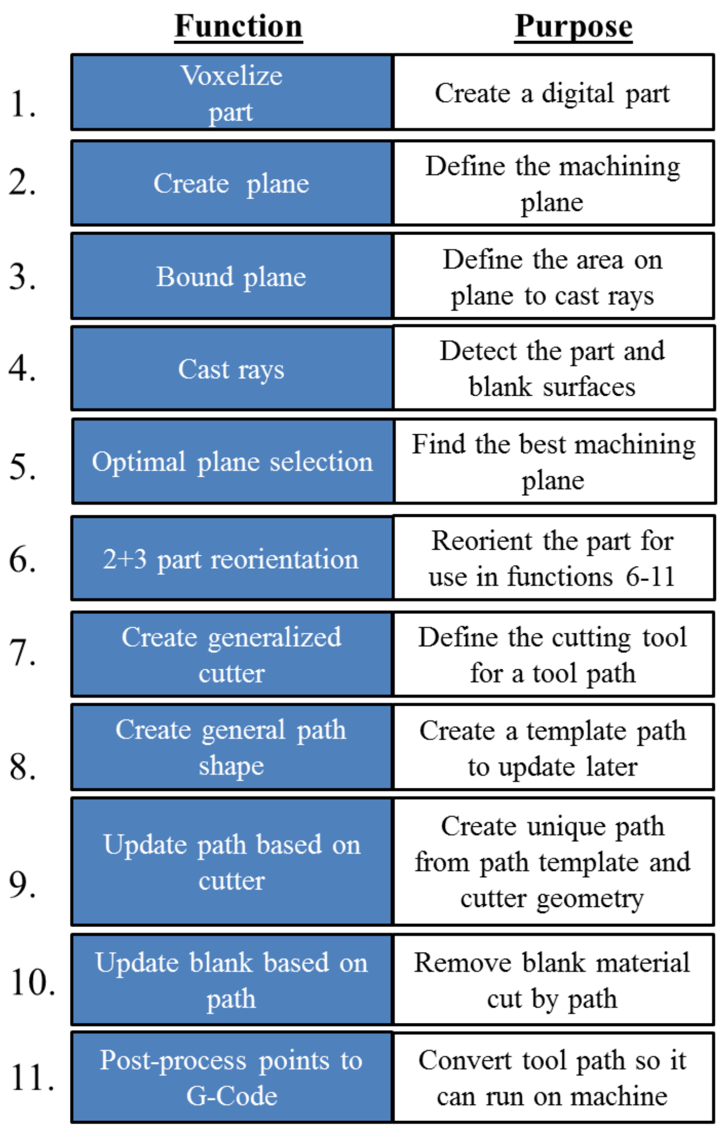
CHAPTER THREE

AUTOMATED PATH PLANNING

This chapter presents a methodology of path planning that automatically generates G-code capable of machining a part based on an input STL file. The major components of this methodology are converting the part file from an STL file into a digital discrete volume model, detecting the surface of the model by ray casting, planning a path based on a generalized cutter and template path shape, removing the volume of material swept by the cutter, and generating G-code. This methodology is implemented by the 11 core functions shown in Figure 3-1.

The first core function shown in Figure 3-1 converts the input STL file into a voxel (volume element) model. The voxel model is like a stack of digital photographs taken along planar slices of a part model. The second function creates an arbitrary plane in space that represents the machining plane. The third deals with placing, bounding, and selecting the best machining plane. The fourth function fires rays from the bounded machining plane into the model to detect its surfaces. The fifth function looks for the best machining plane by reorienting the plane and firing rays from each orientation to determine the best orientation from which to machine.

The seventh and eighth functions deal with creating and updating a tool path based on generalized cutter geometry and template path shape. The generalized cutter shape is the union of a torus and two cones and used to generate the shape of almost any cutter. The ninth function updates the template tool path based on the part and blank surfaces and the cutter geometry. The tenth function updates the voxelized part model based on the volume swept by the tool as is follows the path. The eleventh function post-processes the tool points to convert them into G-code. The generated G-code is used to machine the part as demonstrated in Chapter 6.



	Function	**Purpose**
1.	Voxelize part	Create a digital part
2.	Create plane	Define the machining plane
3.	Bound plane	Define the area on plane to cast rays
4.	Cast rays	Detect the part and blank surfaces
5.	Optimal plane selection	Find the best machining plane
6.	2+3 part reorientation	Reorient the part for use in functions 6-11
7.	Create generalized cutter	Define the cutting tool for a tool path
8.	Create general path shape	Create a template path to update later
9.	Update path based on cutter	Create unique path from path template and cutter geometry
10.	Update blank based on path	Remove blank material cut by path
11.	Post-process points to G-Code	Convert tool path so it can run on machine

Figure 3-1: Chapter overview by function and function purpose

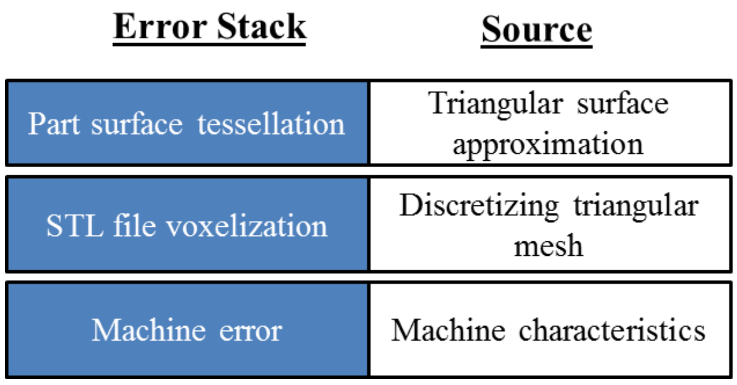
Part model representation

This research introduces a new tool path planning approach based on ray-casting into voxel models. A function converts the part models into voxel models from STL files. The STL file is a format used by nearly every cad package and in this research CAD models are imported as STL [44] files. STL files are generated by tessellation of a parts surface into triangle representations of the surface. The STL file contains triangle vertex and normal information. Although STL files are an approximation of parametric surface models, they can still represent virtually any part with acceptable surface accuracy and tolerance. The STL file is an excellent match to the GPU's capabilities.

Voxelization is the process of transforming an object into a volume element. A voxel is a 3D volume element much like a pixel is a 2D picture element. A voxel can be thought of as stacked photographs or layers of pixels in space. The medical industry utilizes voxel models to store the data from CT and MRI scans of bones and internal organs in medical imaging. Graphic artists use voxels in games to simulate a variety of effects including semi-transparent media such as smoke, fog, and clouds.

In this research, a voxel model is used to represent the part. The part is voxelized directly from the STL model and the resulting voxel is a digital discrete volume. The voxelization is carried out by exploiting native GPU functionality (such as depth testing) and native memory architecture (such as depth buffers and texture memory). The part is voxelized for the purposes of fast intersection detection with the rays, 3-D calculations on the voxel volume, and inherent simulation capability.

Analytically solving for plane-surface or point-surface intersections takes significant computation time when using the iso-parametric, iso-planar, iso-scallop, or polyhedral methods described in Chapter 2. Surface intersection is drastically simplified by using a voxel model. However, this speed comes at the expense of large memory consumption and the introduction of another layer of surface approximation error. In practice, the maximum overall error is expressed as a function of the tessellation, voxelization, and machine errors as shown in Figure 3-2. The total error can be expressed by Equation (3.1) .



Error Stack	Source
Part surface tessellation	Triangular surface approximation
STL file voxelization	Discretizing triangular mesh
Machine error	Machine characteristics

Figure 3-2: Methodology error stack

$$E_{Surface} = f\left(t,v,m\right) = \left|e\left(t\right)\right| + \left|e\left(v\right)\right| + \left|e\left(m\right)\right| \tag{3.1}$$

Where e(t) is the tessellation error, v(t) is the voxelization error, and e(m) is the machine error. The tessellation and voxel errors are both introduced by surface approximation and the user has control over tessellation resolution and voxel size. In this research, it is assumed that the STL file has been imported with acceptable accuracy and that the user determines the maximum acceptable voxel size.

The voxelization algorithm makes use of the native graphics rendering functionality and the fact that the STL file is a tessellated model comprised of polygons (triangles). When images are rendered to a computer screen, they pass through the GPU's rendering pipeline. The rendering pipeline takes the polygons and converts them to pixels on the screen. The GPU is set up to render to the screen using either an orthographic projection or a perspective projection. The voxelization process utilizes orthographic projection as shown in Figure 3-3. Figure 3-3 also shows the how the part surfaces are projected onto the viewing plane (screen). The projection of the part onto the viewing plane is performed by looking at the triangles on the tessellated surface and the order in which their vertexes were defined. If the vertexes were defined in a clockwise fashion, the surfaces are understood by the GPU to be front facing – meaning they face the viewing plane. If the vertexes were created in a counterclockwise fashion they are understood by the GPU to be rear-facing meaning they face away from the viewing plane. Whether the polygons seen by the viewing plane are rear-facing or front-facing allows the voxelization algorithm to determine if the viewing plane is inside a part or outside the part.

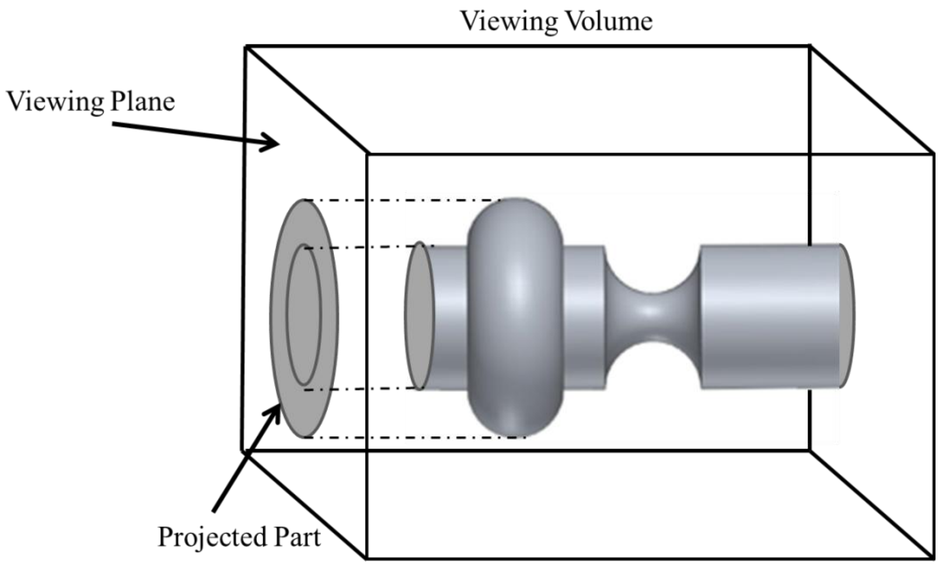


Figure 3-3: Part model rendered to viewing plane

The voxelization process is similar to the one described in [45] where a clipping plane is placed so that is slices the part as shown in Figure 3-4. The voxel model is created by moving the clip plane along the model and using the information from the rear-facing and front-facing polygons to render only the inside of the part onto texture memory as show in Figure 3-4. This process essential slices the model and then takes a picture of the inside of the model only.

The texture resolution is selected to be eight times greater than the desired voxel resolution. The factor of eight is chosen to balance voxel resolution and memory usage. Therefore, if the desired voxel resolution is 100x100x100 voxels, then 2D textures are created at a resolution of 800x800 and sampled in depth at 8Xs the desired voxel resolution. The texture memory is assigned as a char data type which means that each voxel has a value between 0-255. Voxels that are inside the part are assigned a value of

255 and voxels that are outside the part are assigned a value of 0. Voxels that are at the surface of the voxel model are defined as 128. Voxels that are near the surface are affected by their distance from the surface according to a ramp in values. The ramp in voxels goes down for those near the surface and outside the model and up for those near the surface and inside the voxel model. This effectively distributes the surface of the voxel model across more voxels which softens the surfaces when rendering. The final voxel value is updated based on the average of the high resolution voxel model.

Figure 3-4 shows the voxelization process with a resolution that is only four times the desired resolution for illustration. The voxelization algorithm performs the following steps to discretize the model:

1. Place a clip plane at the desired slice location along the part.
2. Allocate texture memory on the GPU at 8 times the desired resolution of the voxel model.
3. Draw all the background pixels as zero in the high resolution GPU texture memory.
4. Draw all back faces of the STL model as 255 in the GPU texture.
5. Draw all front faces as 0 in the GPU texture.
6. Test the depth of front facing polygons and rear facing polygons.
7. Render the front faces that are in front of the back faces.
8. Make 8 textures in depth and average each 8x8 grid to determine the voxel value for the final voxel volume and repeat for the next slice.

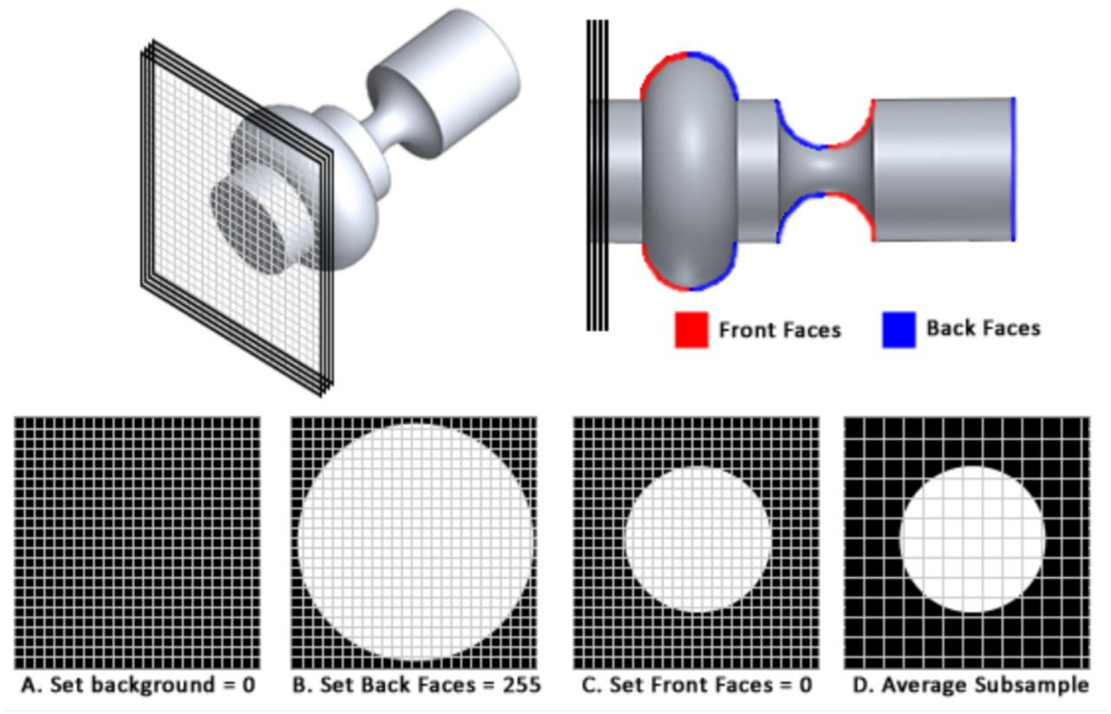


Figure 3-4: Rendering slices into memory

This algorithm was implemented in C++ and DirectX. Screen shots of the voxelization process are shown in Figure 3-5. The part to be tessellated is shown on the left and consists of a freeform surface intersected with three other surfaces from a cylinder, sphere, and plane. The tessellated part is shown in the top row on the left and the clipping plane is shown at the right side of that part. What is saved to GPU memory is shown in the box below the part. The clipping plane starts at the right side of the part and performs the operations described above before moving to the next plane. The right image shows the output of the algorithm where the tessellated part is now a volumetric part, or voxel model. Once the part has been voxelized, it represents a digital discrete volume.

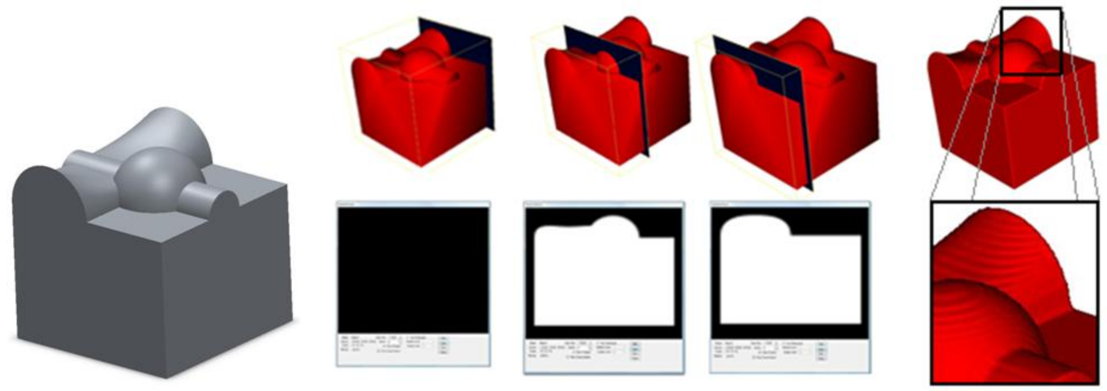


Figure 3-5: Voxelization process

The tool path planning algorithm described below relies on the definition of a blank as well as a part. The blank is also represented by a voxel model. The voxel model of the blank is created based on dimensions of a part volume. The blank is created by copying the voxel volume and assigning all the voxels to their “on” value then taking the part volume values and assigning those “off” in the blank volume. The blank and part volumes are show in Figure 3-6. The blank material is removed by successive tool paths based on the depth of cut until only the part remains. Defining a voxel blank also allows simulation of the material removed per pass.

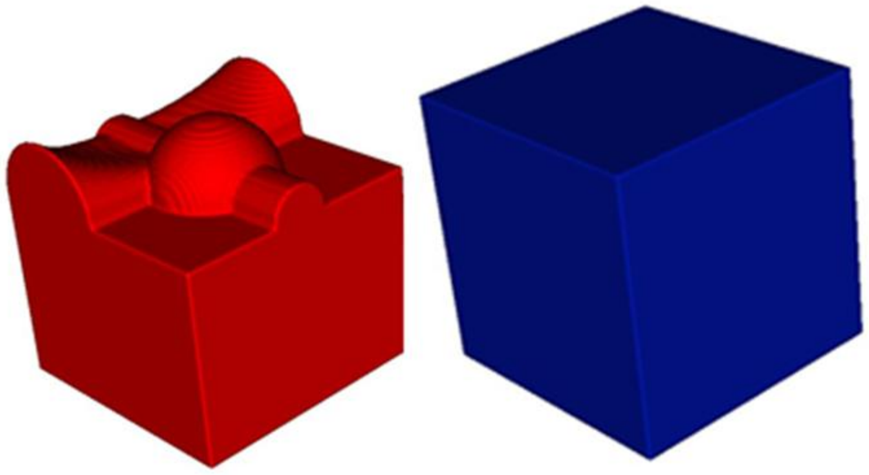


Figure 3-6: Voxel models of blank and part

Plane projection algorithm

The tool paths generated in this research are created based on the portion of the model seen from a machining plane. The machine tool physically cuts the part along the machining plane. The location machining plane can be anywhere around the part. The machining plane is infinite and to determine the area used in actual machining, it must be bounded to a smaller area based on the part visibility. The part visibility on the plane is determined by projecting the points of a bounding box that surrounds the part. A parametric planar square is created on the plane that represents the projected machining area.

The machining plane is described by a point $\mathbf{P_0}$ and a normal $\mathbf{n}$. The plane is constructed by selecting a point above the origin of the part and creating the plane normal vector from the cross product of the **X** and **Y** axes unit normal directions. These direction vectors are maintained within a plane class and used as the parametric directions of the plane. Each point of the bounding box is projected onto the plane by Equation (3.2) and shown in Figure 3-7.

$$\mathbf{q} = \mathbf{p} - \mathrm{proj}_{\mathbf{n}}\left(\mathbf{p} - \mathbf{p}_0\right) \tag{3.2}$$

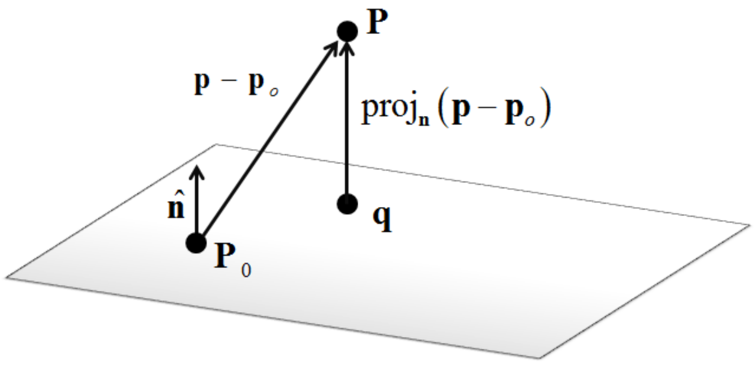


Figure 3-7: Point projected onto plane

Once the points have been projected onto the plane, the parametric distances *s* and *t* along the direction vectors used to define the plane are calculated. Based on a given plane origin point, **P₀**, the projected point **P,** and the unit direction vectors used in the definition of the plane **u**, and **v**, the parametric distance ***s***, and ***t*** can be determined. The projected point on the plane in terms of parametric distances can be represented by Equation (3.3).

$$\mathbf{p} = \mathbf{p}_0 + s \cdot \hat{\mathbf{u}} + t \cdot \hat{\mathbf{v}} = \begin{bmatrix} P_{0x} \\ P_{0y} \\ P_{0z} \end{bmatrix} + s \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} + t \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{3.3}$$

This results in 3 equations and two unknowns, ***s***, and ***t***. Solving for parametric distances ***s*** and ***t*** yields Equation (3.4) and (3.5).

$$s = \frac{p_x - p_{0x} - tv_x}{u_x} \tag{3.4}$$

$$t = \left[p_z - p_0 - \left(\frac{p_y - p_{0y}}{u_y} \right) u_z \right] \Bigg/ \left(v_z - \frac{v_y u_z}{u_y} \right) \tag{3.5}$$

The plane origin is moved so it is coincident to the projected point with the minimum parametric distances. This allows the maximum parametric distances, uMax and vMax, along the plane direction vectors to the same distance of the part bounds on the plane as shown in Figure 3-8.

Figure 3-8 shows the bounding box surrounding the part and the bounding box points when projected onto a plane located above the part. Figure 3-9 shows the output of the plane projection algorithm for planes located at different locations. The figures show that the parametric distances are updated based on the projected box for arbitrary locations in Cartesian space. The parametric distances are necessary for the ray-casing algorithm discussed in the next section.

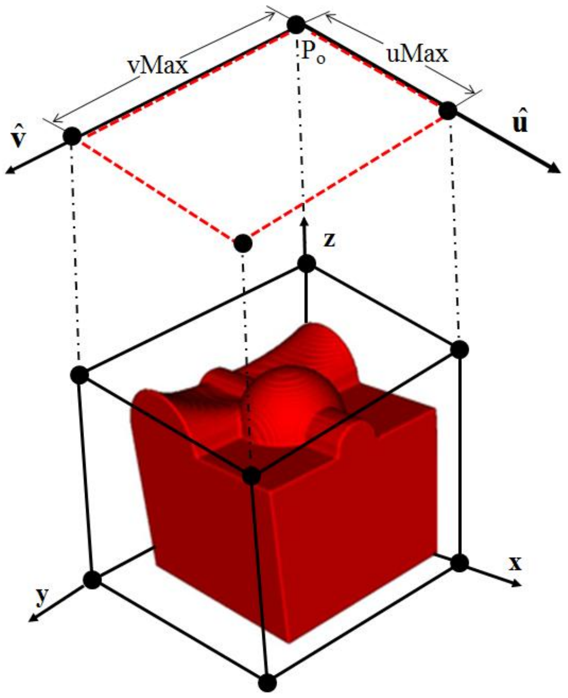


Figure 3-8: Parametric plane (red) with projected bounding box

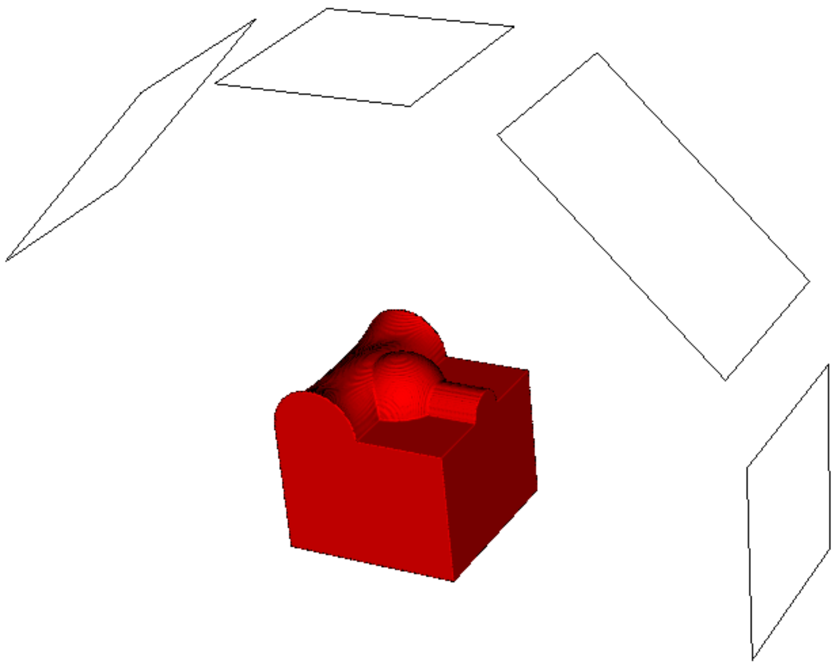


Figure 3-9: Output of plane projection algorithm

Ray casting algorithm

The part surfaces in this research are determined by the intersection of a ray with the voxel model. Ray casting is a technique used in computer graphics to model how light behaves in a virtual scene. The term ray comes from "light ray" and is typically used to model how light refracts off different objects in a virtual scene. In this research, the concept of ray casting is used to determine the surface of the part and blank model for later use in calculating the tool path. In the previous section, the creation of a bounded machining plane based the part dimensions was described. In this section, a grid is generated on that plane based on the parametric size (uMax and vMax) of the projected box. The grid and a single ray are shown in Figure 3-10.

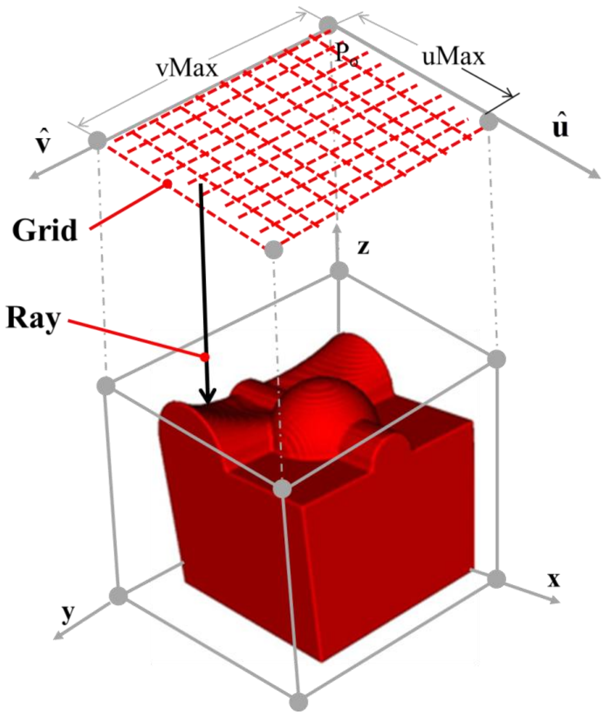


Figure 3-10: Single ray cast from grid

A ray is defined by a point and a direction vector. Any point on the ray is determined by the direction vector multiplied by the distance from the ray origin,

$$r = p_o + \mathbf{n} \cdot d \tag{3.6}$$

where, r is the ray location, $\boldsymbol{p}_o$ is the origin of the ray, $\mathbf{n}$ is the direction of the ray, and d is the distance along the ray. Rays are cast from the machining plane by increasing d until the voxel surface is "hit" as shown in Figure 3-11. The surface is hit when the ray passes the iso-surface of the part or more specifically when the value in the volume is equal to 128.

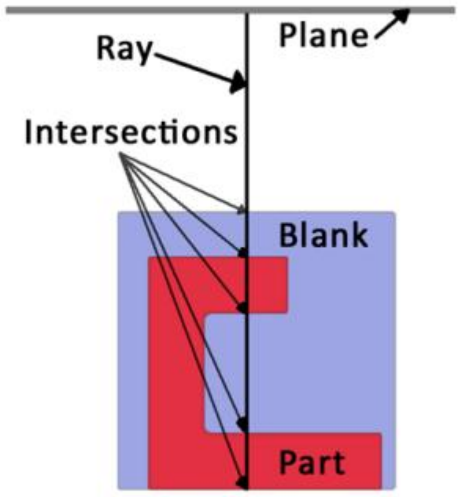


Figure 3-11: Ray casting into part and blank voxel volumes

The value of the voxel volume along the ray is determined by tri-linear interpolation of the voxel volume at that point along the ray. The tri-linear interpolation is performed based on the known voxel width and the location of the vertexes surrounding the voxel as shown in Figure 3-12.

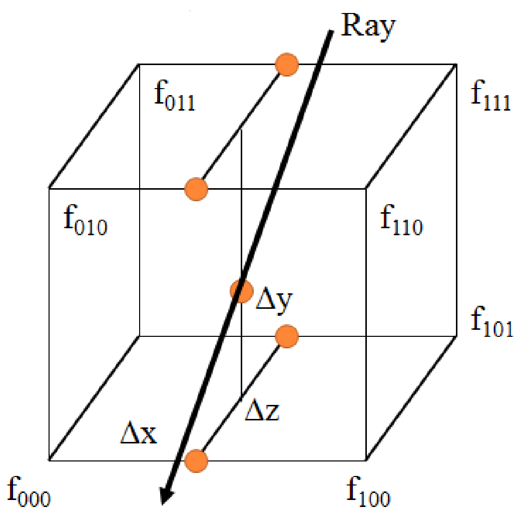


Figure 3-12: Interpolation of the voxel value along the ray

The value of the voxel model is obtained by using Equation (3.7).

$$\begin{aligned} val = &(1-\Delta x)(1-\Delta y)(1-\Delta z) f_{000} + (\Delta x)(1-\Delta x)(1-\Delta x) f_{100} \\ &(1-\Delta x)(\Delta y)(1-\Delta x) f_{010} + (1-\Delta x)(1-\Delta y)(\Delta z) f_{001} \\ &(\Delta x)(\Delta y)(1-\Delta z) f_{110} + (\Delta x)(1-\Delta y)(\Delta z) f_{101} \\ &(1-\Delta x)(\Delta y)(\Delta z) f_{011} + (\Delta x)(\Delta y)(\Delta z) f_{111} \end{aligned} \tag{3.7}$$

The surface intersection occurs when the interpolated value of the voxel volume is equal to 128. As presented in the voxelization section, voxel can take a value of 0-255 and 128 is chosen as the surface of the model. The location where the ray intersects the voxel surface is accurately calculated using a bisection algorithm. The bisection algorithm is as follows: the ray is cast by increasing the distance along the ray in increments it breaks the surface of the part (interpolated value > 128). Then the ray is moved back to the last increment along the ray and the voxel volume is re-sampled. This gives a “top” value and a “bottom” value. A middle value between the top and bottoms values is obtained by incrementing half the nominal increment along the ray from the “top” location. Each of the interpolated values is added to -128. The top and bottom values are multiplied by the middle value. The segment along the ray where the product is negative indicates that the surface lies along that segment. Bisection is repeated with the smaller segment until the value is within 128+/-tolerance. The steps along the ray and the tolerance of the bisection algorithm are chosen to result in fast traversal of the ray and accurate surface detection. Once the top surface of the part or blank is found in this way, then the distance along the ray and knowledge of the plane origin is used to determine the x,y,z location of the part surface. The bisection routine can be repeated to find each surface transition in the part.These surfaces intersections are stored and used to determine the cutter contact points and the maximum depth of cut at that location as well as undercut locations.

Surfaces that cannot be reached without additional axes or part re-positioning can automatically detected using the same ray casting approach. These surfaces are detected by allowing a ray cast from the tool cutting plane to pass through the part volume and continue to detect surface intersections. From these additional intersections, any un-machined surfaces that need to be removed are detected and stored. Figure 3-13 demonstrates the case where the machining plane is rotated by 45º.

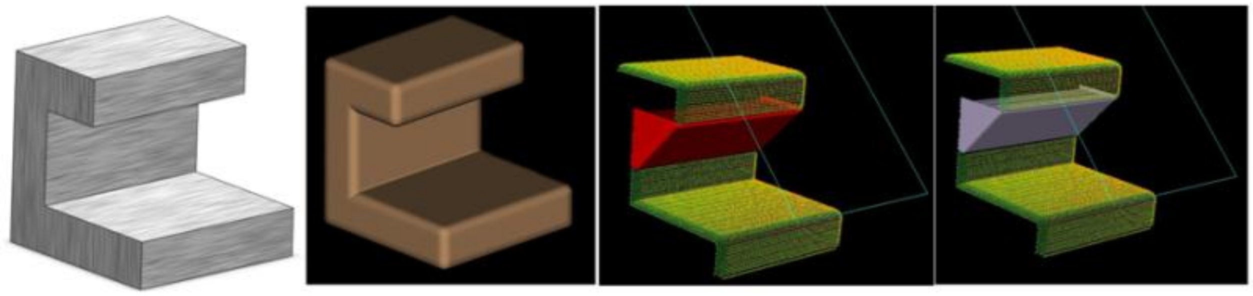


Figure 3-13: Detection of unreachable regions

Optimal cutting plane selection

Five axis machining can result in faster machining times and higher quality parts. To maintain the stiffest possible machining conditions, 5-axis machining is often performed by using the two rotary axes as fixed indexing axis to orient the part while the other three are used to machine the part; this is called 2+3 machining. In addition to stiffer machining characteristics, 2+3 machining is often employed because it is easier to manually create tool paths using CAM software. This research is 2+3 machining because all machining is performed from the machining plane in an arbitrary orientation.

The machining plane that results in the most material removed can be found by testing a series of planes around the part. The plane projection method described in the plane projection algorithm section is used to create bounding boxes on planes placed around the part. Rays are cast from these planes to detect the part and blank surface. An exhaustive search to determine the best plane is too time consuming because there are an infinite number of possible plane orientations. An initial search area was selected in this research based on constructing a plane at every edge, vertex, and face of the bounding box with each plane's normal pointing at the center of the part volume. This results in the creation of 26 planes. The geometric identity of this shape is the gyro-elongated square bi-cupola. The initial search plane orientations are shown in Figure 3-14. The rendered test planes based on this shape are shown in Figure 3-15. For each search plane, rays are cast to determine the part and blank surfaces. Snapshots of the height values of the intersection rays with the part surface are show in Figure 3-16.

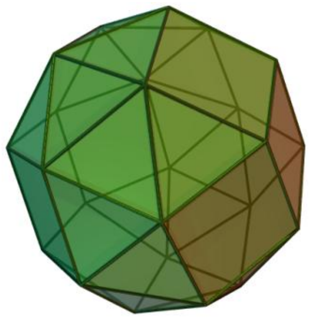


Figure 3-14: Test shape for optimal plane.

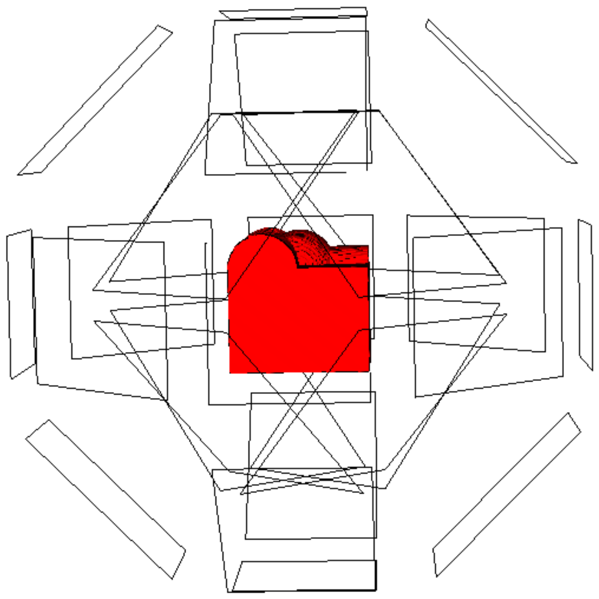


Figure 3-15: All 26 test planes.

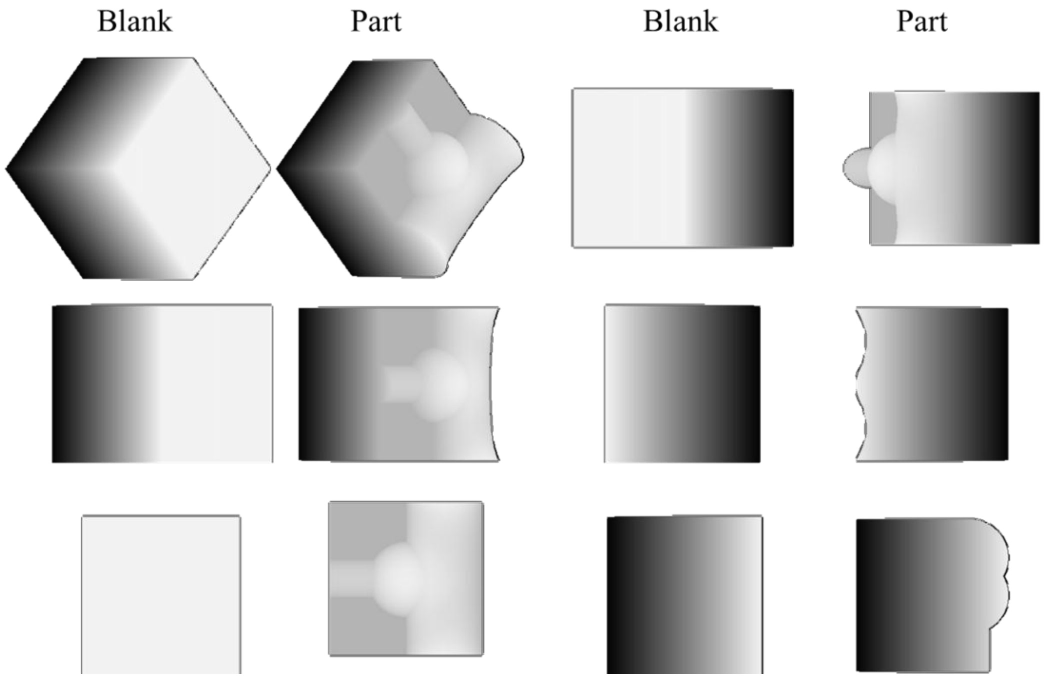


Figure 3-16: Blank and part images from ray casting from different planes.

The point of intersection for each ray with the part and blank models are stored in a grid. For each point in the resulting grid, the difference between an interior point in the part volume and the surface points of the blank and part is calculated as shown in Figure 3-17. These distances are summed to get an estimation of the total length of material removed from a given orientation. For the interior point, $\mathbf{P_0}$, blank surface point, $\mathbf{P}_{blank}$, and part surface point, $\mathbf{P_0}$, the estimated length of material removed is given by Equation (3.8).

$$MR = \sum_{i=0}^{N} \left(\left\| \mathbf{p}_0 - \mathbf{p}_{iBlank} \right\| - \left\| \mathbf{p}_0 - \mathbf{p}_{iPart} \right\| \right) \tag{3.8}$$

Although this yields the plane with the most material removed from a given orientation, it is not necessarily the ideal plane to machine from. Other criteria can be introduced using this framework that could account for other metrics to define the optimal plane as well as extend this search to local plane selection.

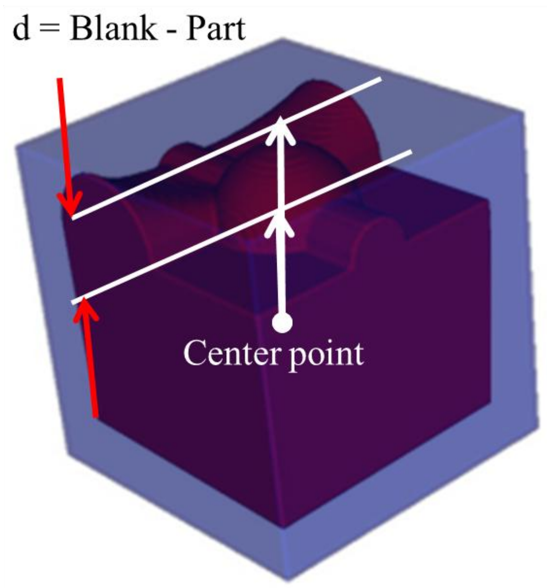


Figure 3-17: Distance between blank and part

Generalized cutter representation

The tool paths generated in this research are directly related to the cutter used for a given tool pass. A generalized cutter description is used to accurately capture cutters such as ball end mill, flat end mill, toroidal end mill, and etc. [46]. The generalized cutter is the combination of two cones and a torus. The generalized cutter is shown in Figure 3-18 and any point, $\boldsymbol{\Psi}$, on the cutter can be described by the following equations:

$$\Psi^{general}\left(\theta,\phi,a_1,a_2\right)=\begin{pmatrix}\left(a_1R_1+R_2\left[\sin\phi-\sin\beta_1\right]+a_2L\tan\beta_2\right)\cos\theta\\ \left(a_1R_1+R_2\left[\sin\phi-\sin\beta_1\right]+a_2L\tan\beta_2\right)\sin\theta\\ \left(a_1R_1\tan\beta_1+R_2\left(\cos\beta_1-\cos\phi\right)+a_2L\right)\end{pmatrix} \tag{3.9}$$

Where,

$$a_1\in[0,1],\ \ a_2\in[0,1],\ \ \phi\in\left[\beta_1,\frac{\pi}{2}-\beta_2\right],\ \ \theta\in[-\pi,\pi] \tag{3.10}$$

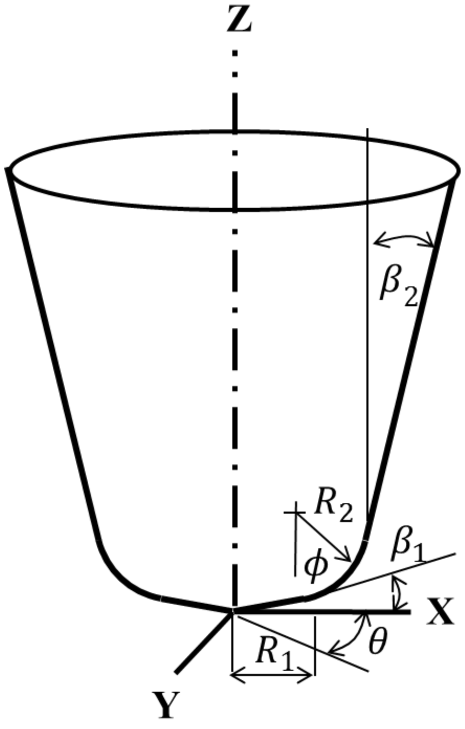


Figure 3-18: Generalized cutter

The generalized cutter is an input to the path creation algorithm described in the next section which uses the generalized cutter outer radius to determine the path interval. The generalized cutter object is also created for simulation purposes by creating a mesh object and rendering it to the screen. Examples of the generalized cutter are shown in Figure 3-19. The generalized cutter is vital to the performance evaluations of different tool paths as discussed in chapter 5.

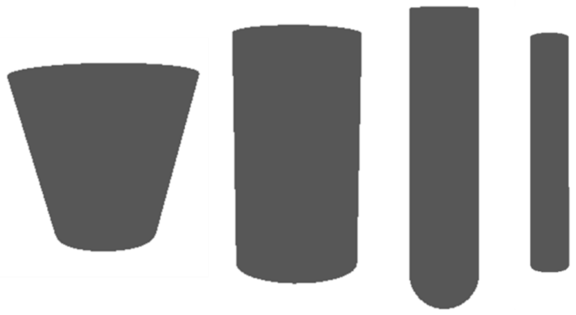


Figure 3-19: Generalized cutter model and representation of commonly used end mills

Path template algorithm

The final tool path is created by using the generalized cutter, the parametric distance uMax and vMax from the plane projection algorithm, the surface results from the ray casting algorithm, and an initial path template. The path template is discussed in this section and creates a planar path based on the generalized cutter radius and desired path overlap. The path interval is the given by Equation (3.11) where I_P is the interval of the path, R_C is the cutter radius and O is the desired overlap (Figure 3-20).

$$\mathrm{I}_P = R_C - O \tag{3.11}$$

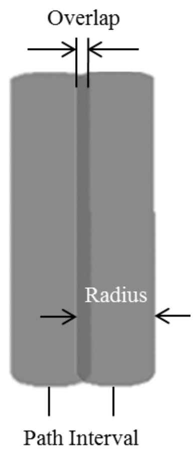


Figure 3-20: Path interval based on radius and overlap

The path template used in this research is a simple path for constant up (conventional) or down (climb) milling. The path template is created by starting the tool completely off the work piece and performing the following steps.

1. Calculate the distance X the tool has to move to reach the opposite side of the blank minus the interval width that the tool has already traversed.
2. Move that distance while adding the tool points to the path at the desired point interval.
3. Calculate the distance Y the tool has to move to reach the top of the surface minus the interval the tool has already been.
4. Move that distance while adding the tool points to the path at the desired point interval.
5. Repeat until the distance is equal to zero.

The above steps map out a simple but effective path to perform either up or down milling. The output path from this algorithm is shown in Figure 3-21. This template path is converted into a tool path by taking the part and blank surface and cutter geometry into account in the next section.

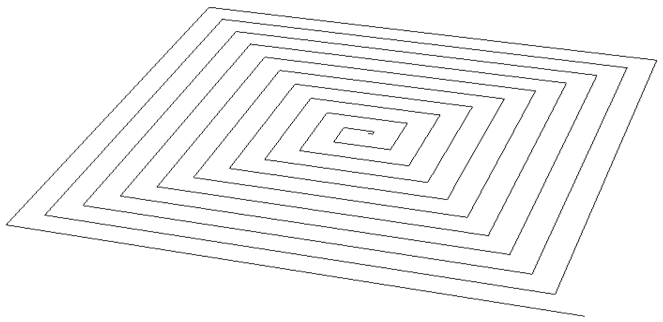


Figure 3-21: Template path

Path based on cutter geometry

The last section described how a template path was created based on the cutter radius and desired radial overlap. This section describes how the template path is converted into a final tool path as shown in Figure 3-23. To update the template path, the center of the generalized cutter is placed at each point in the path. Every path point is initially placed at the desired depth of cut or the top surface of the part; whichever is less distance from the blank surface. Then, the area underneath the cutter is checked for gouging. If there is any gouge, the tool center is updated (lifted) to a location above the tool point where there is no gouge.

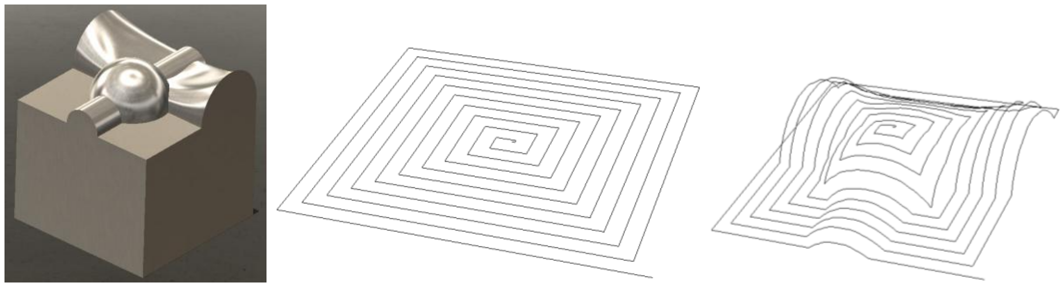


Figure 3-22: Original and updated path based on generalized cutter

The algorithm for checking the points underneath the tool accounts for the generalized cutting geometry. The checking algorithm is a two-step process. The first step is to get the vertical distance of each point on the part surface to the tool center point as shown in Figure 3-23. The distance for three axis calculation is simply the z-height difference. If the tool is machining from a plane that is not parallel to the x-y plane then the distance is found by the dot product.

$$d = \mathbf{A} \bullet \hat{\mathbf{B}} \tag{3.12}$$

Where, *d* is the distance to the plane, **A** is the vector from the *(x,y)* point under the cutter to the cutter center, and $\hat{\mathbf{B}}$ is the plane unit normal.

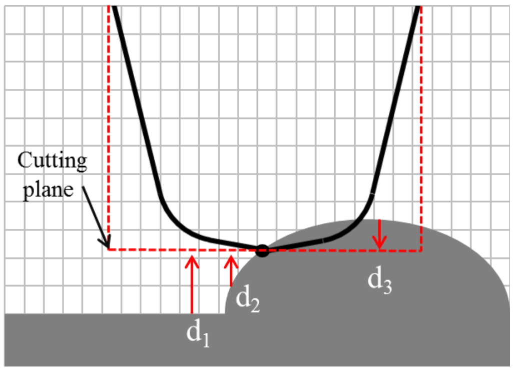


Figure 3-23: Step one of updating the path – get distance to tool center

The second step is to add the distance from the intersecting *(x,y)* point on the cutting plane to the cutter surface to the calculated distance *d* as shown in Figure 3-24. The x,y position in the plane is used to get the distance to the center of the cutter as shown in Figure 3-25. The height of the cutter at that point is calculated based on this location. The solution is analytical and is based on the fact that employs cutter symmetry. This calculation assumes that the surface swept by the cutter teeth is the intersecting surface.

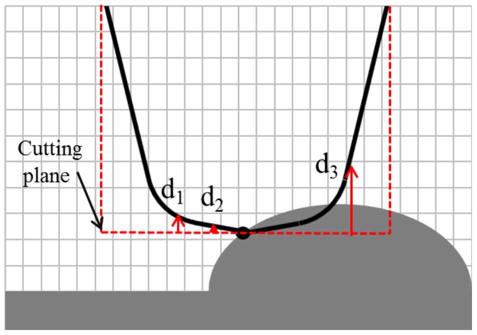


Figure 3-24: Distance from cutting plane to cutter surface

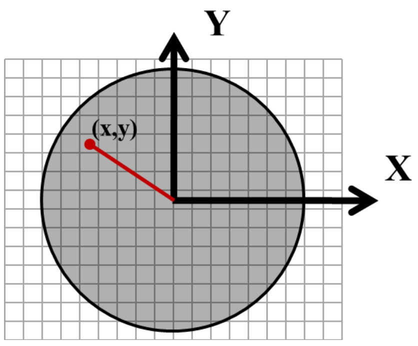


Figure 3-25: Example position under cutter used to solve for the surface point

The updated path is shown in Figure 3-22. The algorithm to calculate the distance from the cutting plane to the cutter surface is as follows:

```
1. For each cutter contact point
        2. Take each voxel x,y location in the cutting plane beneath the cutter
3. Calculate the z-clearance with a device function based on the x,y location of the voxel as follows
    if $\left(\sqrt{x^2+y^2} > a_1R_1\right)$
        $a_1 = 1;$
        if $\left(\sqrt{x^2+y^2} > R_1 + R_2\left[\sin\phi - \sin\beta_1\right]\right)$
            $\varphi = \frac{\pi}{2} - \beta_2$
            if $\left(\sqrt{x^2+y^2} > R_1 + R_2\left[\sin\left(\frac{\pi}{2}-\beta_2\right) - \sin\beta_1\right] + L\tan\beta_2\right)$
                then you are clear of the cutter
            else, solve for $a_2$  $z = R_1\tan\beta_1 + R_2\left(\cos\beta_1 - \cos\left(\frac{\pi}{2}-\beta_2\right)\right) + a_2L$
        else, solve for $\varphi$,  $z = R_1\tan\beta_1 + R_2\left(\cos\beta_1 - \cos(\varphi)\right)$
    else, solve for $a_1$,  $z = a_1R_1\tan\beta_1$
4. Get the blank and part depth
5. Add the calculated z-clearance
6. Calculate the acceptable depth as,   depth = blankTop - partTop + zClearance
```

7. Keep the minimum depth for all points under the cutter
8. Update the tool point with the minimum depth under the cutter

Removal of blank material from path

This section describes how the updated tool path from the last section is used to update the blank voxel model as shown in Figure 3-26. The blank model is updated by updating the values of the voxels affected by the cutter as it follows the tool path. Once the blank model is updated, it can be used as an input model for this methodology and the entire process repeated until there is no remaining blank material. In addition, the updated blank model is used to simulate the actual cutting process (Figure 3-26).

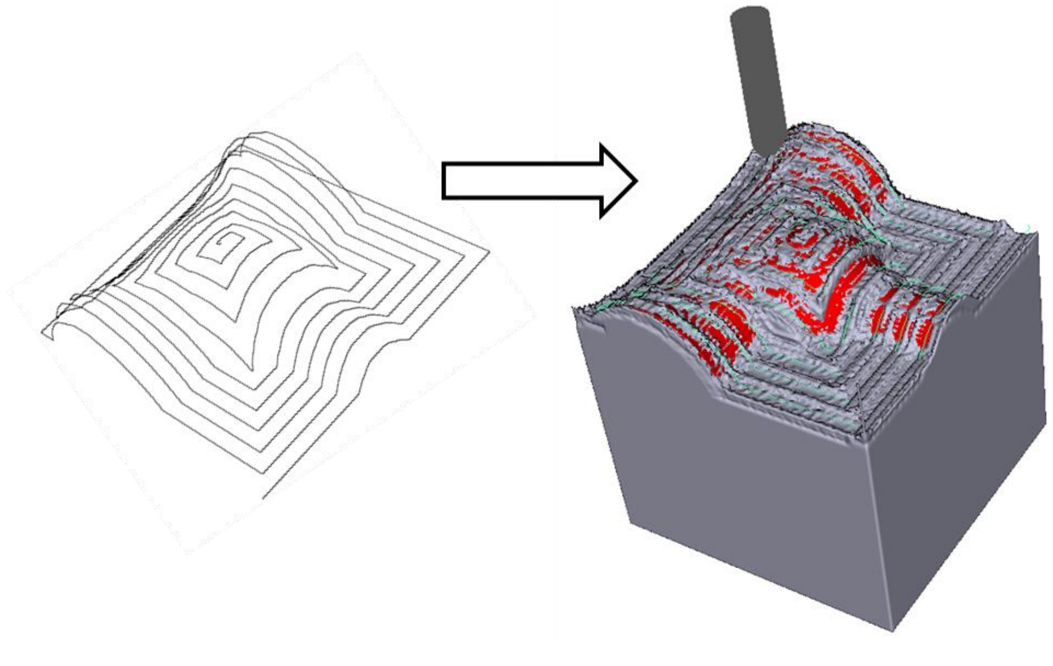


Figure 3-26: Automated path based on voxel model, ray casting, and generalized cutter

As the tool moves along the tool path it removes material from the blank. Each voxel value represents the amount of material in the given voxel using a range between 0-

255. The isosurface of the blank and part is defined as the location where the voxel values are 128. The blank model must be updated so the spatial location that the cutter surface sweeps during motion corresponds to the location in the blank model where the voxel value is 128. The voxels are updated according to the location of the cutter. For each tool point in the tool path, a voxel box similar to what is shown in Figure 3-27 is created around the cutter. Then, each voxel inside the voxel box surrounding the cutter is updated according to the minimum distance to the cutter surface. The distance is used to assign new values to the voxel model based on the effect of the cutter surface.

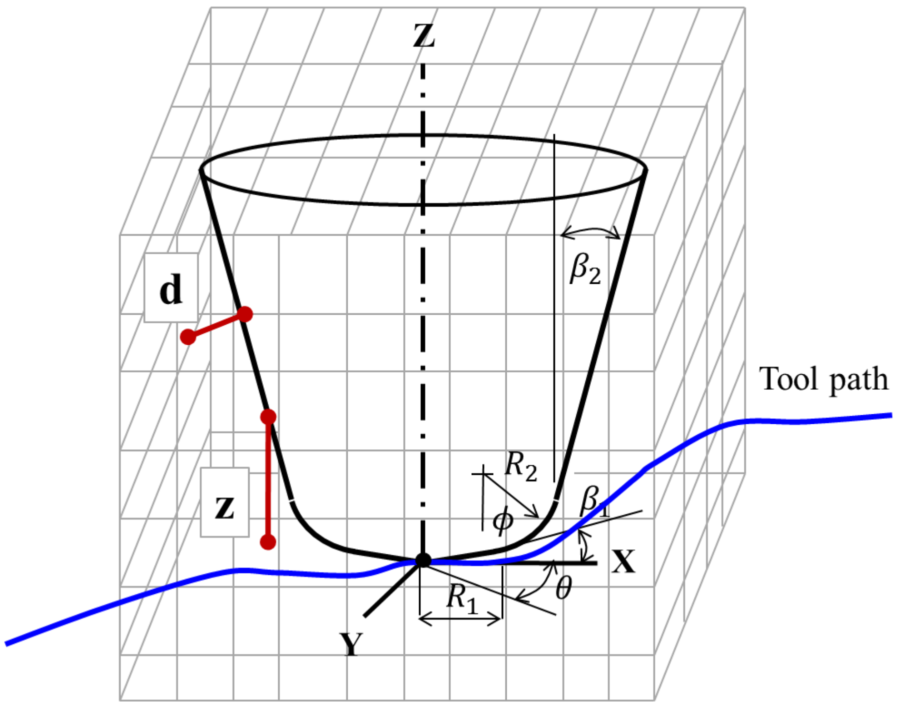


Figure 3-27: Voxel box

The problem of finding the closest distance from a given voxel and the cutter surface is simplified by mapping the 3D problem to 2D. The distance from a given voxel within the box is calculated based on the nearest distance from that point in the voxel volume and the 2D cutter profile intersecting the plane that goes through that point and the cutter contact point. The definition of the cutter which is two cones and a torus in 3D is reduced to two lines and a circle in 2D. The same point in 3D and on the 2D intersecting plane is show in Figure 3-28.

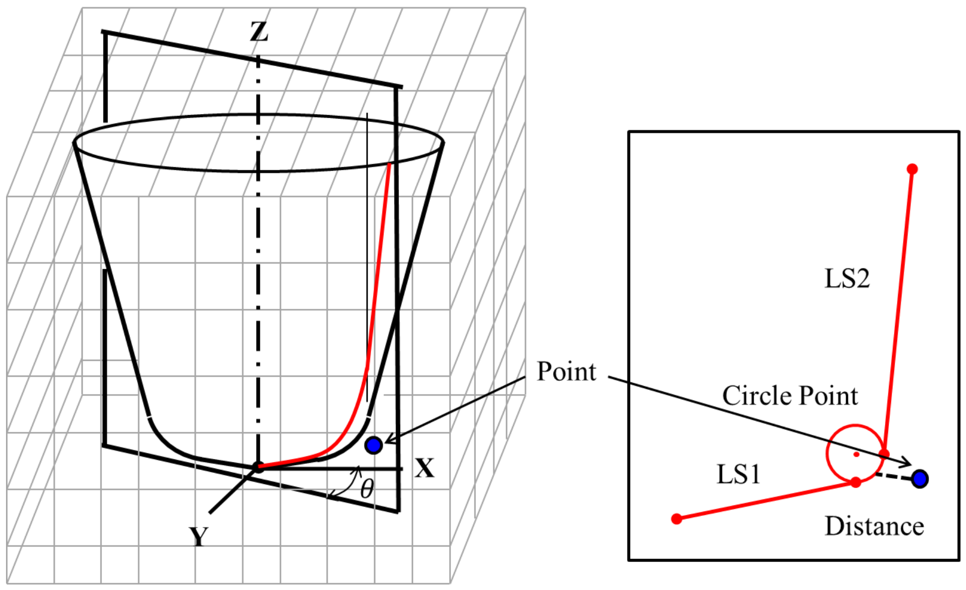


Figure 3-28: 3D profile and 2D profile from intersecting plane

The minimum distance to the 2D surface can be determined once the end points of the line segments and the center of the circle are calculated. The location of the line segment ends and center circle can be determined from the generalized cutter description and knowledge of the angle θ the plane intersects the cutter surface. The angle θ is determined by stripping the z-coordinate of the voxel box element, getting the vector from cutter center to the stripped point, and using the definition of the dot product to get the angle between this point and the tools x-axis. According to Equation (3.9), the line segment 1 (Figure 3-28 and Figure 3-18) is defined by the two points between the cutter center and the location where the bottom section joins the cutter torus section:

$$\mathrm{LS}_1 = \begin{cases} P_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ P_2 = \begin{pmatrix} R_1 \cos(\theta) \\ R_1 \sin(\theta) \\ R_1 \tan(\beta_1) \end{pmatrix} \end{cases} \tag{3.13}$$

Line segment 2 is the line segment from the torus section to the top of the cutter:

$$\mathrm{LS}_2 = \begin{cases} P_3 = \begin{pmatrix} \left(R_1 + R_2\left[\sin\left(\frac{\pi}{2} - \beta_2\right) - \sin(\beta_1)\right]\right)\cos(\theta) \\ \left(R_1 + R_2\left[\sin\left(\frac{\pi}{2} - \beta_2\right) - \sin(\beta_1)\right]\right)\sin(\theta) \\ R_1 \tan(\beta_1) + R_2\left[\cos(\beta_1) - \cos\left(\frac{\pi}{2} - \beta_2\right)\right] \end{pmatrix} \\ P_4 = \begin{pmatrix} \left(R_1 + R_2\left[\sin\left(\frac{\pi}{2} - \beta_2\right) - \sin(\beta_1)\right] + L\tan(\beta_1)\right)\cos(\theta) \\ \left(R_1 + R_2\left[\sin\left(\frac{\pi}{2} - \beta_2\right) - \sin(\beta_1)\right] + L\tan(\beta_1)\right)\sin(\theta) \\ R_1 \tan(\beta_1) + R_2\left[\cos(\beta_1) - \cos\left(\frac{\pi}{2} - \beta_2\right)\right] + L \end{pmatrix} \end{cases} \tag{3.14}$$

The location of the center of the circle is given by:

$$C = \begin{pmatrix} \left(R_1 - R_2\sin\left(\beta_1\right)\right)\cos\left(\theta\right) \\ \left(R_1 - R_2\sin\left(\beta_1\right)\right)\sin\left(\theta\right) \\ R_1\tan\left(\beta_1\right) + R_2\cos\left(\beta_1\right) \end{pmatrix} \tag{3.15}$$

Once the location of each of the points is found, it is mapped to the location in the voxel volume by multiplying it by the following translation matrix.

$$P_{\text{new}} = \begin{bmatrix} 1 & 0 & 0 & ccp_x \\ 0 & 1 & 0 & ccp_y \\ 0 & 0 & 1 & ccp_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_{\text{old-X}} \\ P_{\text{old-Y}} \\ P_{\text{old-Z}} \\ 1 \end{bmatrix} \tag{3.16}$$

Where *ccp* is the cutter contact point in Cartesian space and *Pold* is the local point and *Pnew* is the translated point in the space local to the cutter contact point. The distance between these points and the nearest point on the surface can be determined by getting the minimum of the distances between the point in the voxel box and the line segments defined above.

Distance to the cutter surface

For every point affected by the cutter volume, the nearest distance to the cutter surface is determined by first finding the angle between that point and a specified direction (such as the x-axis). This angle specifies the plane where the two line segments and circle center are located and allows the nearest distance between the points affected by the cutter and the cutter surface to be determined. The following is adapted from [47]. A given line segment between **x1 =** (x1, y1, z1) and **x2 =** (x2, y2, z2) can be represented by the following parametric equation.

$$\mathbf{v} = \begin{bmatrix} x_1 + (x_2 - x_1)t \\ y_1 + (y_2 - y_1)t \\ z_1 + (z_2 - z_1)t \end{bmatrix} \tag{3.17}$$

For a point (x0, y0, z0) that is not on the line the square distance between that point and some point on the line is:

$$d^2 = \left[(x_1 - x_0) + (x_2 - x_1)t\right]^2 + \left[(y_1 - y_0) + (y_2 - y_1)t\right]^2 + \left[(z_1 - z_0) + (z_2 - z_1)t\right]^2 \tag{3.18}$$

The minimum distance is when the derivative of the above equation is zero. By taking the derivative of the above equation and setting it to zero the parameter t is:

$$t = \frac{(\mathbf{x}_1 - \mathbf{x}_0) \bullet (\mathbf{x}_2 - \mathbf{x}_1)}{|\mathbf{x}_2 - \mathbf{x}_1|^2} \tag{3.19}$$

Inserting the parameter t that minimizes the distance back into the distance equation yields Equation (3.20):

$$d = \frac{\left|(\mathbf{x}_2 - \mathbf{x}_1) \times (\mathbf{x}_1 - \mathbf{x}_0)\right|}{\left|\mathbf{x}_2 - \mathbf{x}_1\right|} \tag{3.20}$$

This distance from the cutter surface is used to update the value of the voxel model. The points affected by the cutter are those inside the cutter and those surrounding the cutter specified by a voxel ramp.

The voxel ramp is used to update the voxels adjacent to the voxel containing the calculated surface of the voxel model. When the voxel box is created it the dimensions of the box are padded to allow for calculation of the ramp. The voxel ramp is the integer number of voxels that the voxel value is updated over. A voxel ramp of 2 is shown in Figure 3-29 where, red is the blank material, grey is the part material, and white is the cutter. Figure 3-29 shows how the values are updated so there is an inside ramp to zero and an outside ramp to 255.

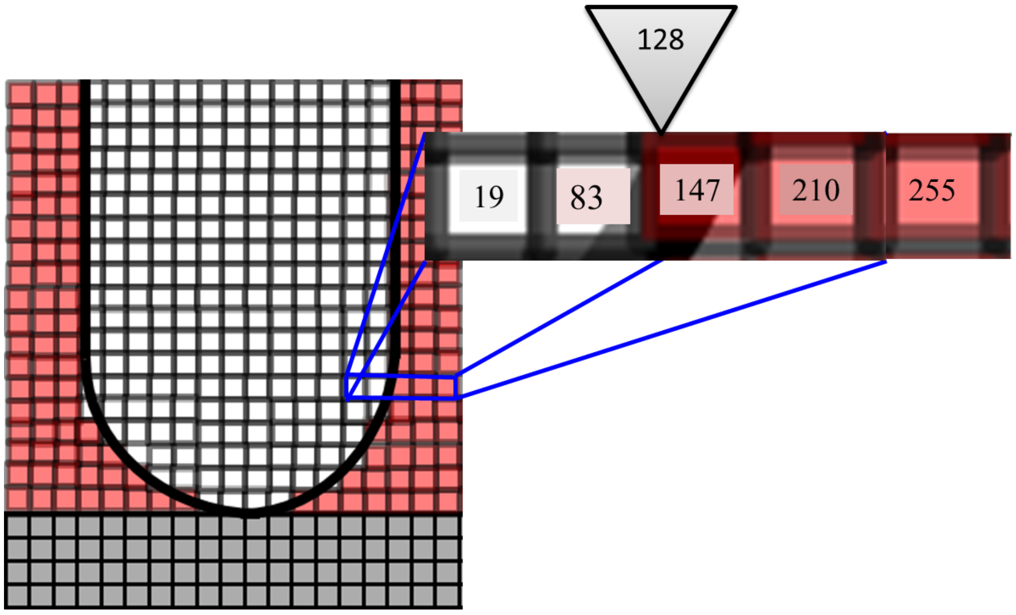


Figure 3-29: Voxel ramp

To determine if a voxel is inside the cutter, the z-height of the current voxel being sampled is taken. The z-height is then used to solve for any point on the surface of the cutter with that height as shown in Figure 3-30. For any z-height, there is a circle of possible points on the cutter surface as shown in red in the figure. The point on the surface is determined by testing which piecewise interval of the generalized cutter the point lies within and solving for the unknown angles and/or distances. The intervals are broken up into three sections I, II, and III for the cone in contact with the surface, for the torus, and for the cone along the shaft of the cutter respectively.

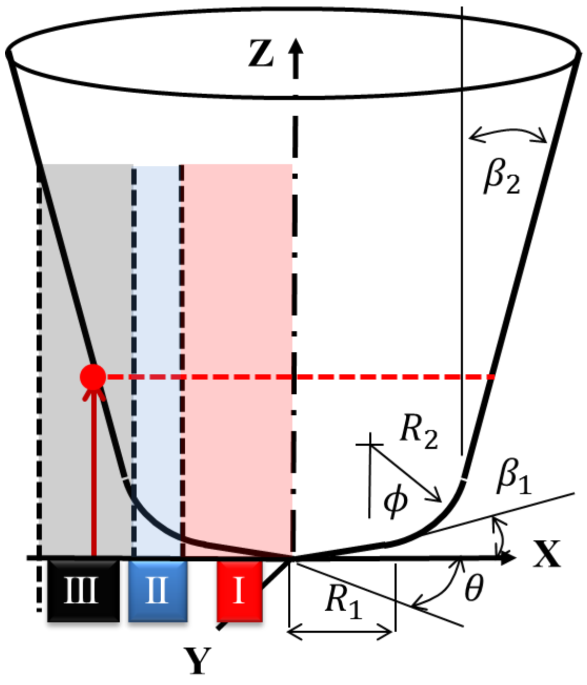


Figure 3-30: Surface point determined by z-height

The algorithm to get a surface point along the cutter at the z height of the sampled voxel is as follows:

1. Get the z height of the voxel relative to the cutter contact point. $z = P_{Vox\_z} - P_{Cutter\_z}$

2. Based on the z-expression for the generalized cutter, determine which section of the cutter the point is in and solve for a surface point at that height.

```
if $\left(z > R_1 \tan(\phi)\right)$
    if $\left(z > R_1 \tan(\phi) + R_2\left[\cos(\beta_1) - \cos\left(\frac{\pi}{2} - \beta_2\right)\right]\right)$
  //point is in section III solve for $a_2$
    $a_2 = \frac{z - R_1 \tan(\beta_1) - R_2\left[\cos(\beta_1) - \cos\left(\frac{\pi}{2} - \beta_2\right)\right]}{L}$
  else
    //point is in section II; solve for $\phi$
    $\phi = \cos^{-1}\left(\frac{R_1 \tan(\beta_1) + R_2 \cos(\beta_1) - z}{R_2}\right)$
else
   //point is in section I; solve for $a_1$
   $a_1 = z / R_1 \tan(\beta_1)$
```

3. Get the point on the surface at that height using a_1, ϕ, and a_2

4. Calculate the distance to the center of the cutter d_{surf} from that point $\boldsymbol{P_{surf}}$ using the cross product of the point and the normalized tool vector, $\boldsymbol{T_{axis}}$:

$$d_{surf} = \left\| \hat{T}_{axis} \times \left(\mathbf{P}_{surf} - \mathbf{P}_{CCP}\right) \right\|$$

5. Calculate the distance d_{vox}, of the point in the voxel box to the tool center axis, $\boldsymbol{T_{axis}}$,

$$d_{vox} = \left\| \hat{T}_{axis} \times \left(\mathbf{P}_{vox} - \mathbf{P}_{CCP} \right) \right\|$$

6. Determine if the point in the voxel box is inside the cutter.

```
if $\left(d_{vox} < d_{surf}\right)$
    inside = true;
else
    outside = true;
```

7. Update the value of the current voxel $Val_{current}$ based on the ramp r, and the voxel size v *as follows*:

```
if (inside)
    $Val_{current} = \begin{cases} 128\left(1 - \frac{d_{vox}}{r \cdot v}\right); & \frac{d_{vox}}{r \cdot v} > 1 \\ 0; & \text{otherwise} \end{cases}$
else
    $Val_{vox} = 128\left(1 + \frac{d_{vox}}{r \cdot v}\right)$
    if $\left(Val_{vox} < Val_{current}\right)$
        $Val_{current} = Val_{vox}$
    else
    //$Val_{current}$ is unchanged
```

A picture of the generalized cutter, tool path, part and updated blank is shown in Figure 3-26.

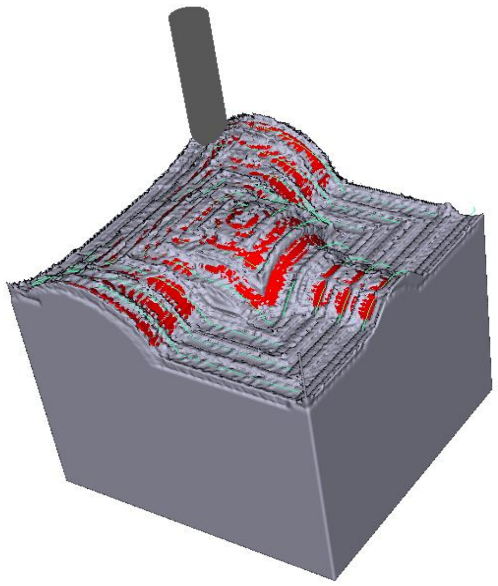


Figure 3-31: Output of methodology

Summary

This section introduced the path planning approach based on voxel models of the blank and part. A plane was created based on the bounding box of the models and rays were cast from the plane using tri-linear interpolation and bi-section. A generalized cutter was created based on two cones and a torus. A path was generated based on the generalized cutter radius. The path was updated based on the blank and part heights obtained from ray-casting. The blank material was removed by updating the blank volume based on the volume swept by the generalized cutter as it followed the tool path. All of these algorithms were implemented both on the CPU and the GPU. The performance of these algorithms is discussed in Chapter 4. The metrics of the path are discussed in Chapter 5. Experimental results of this method are presented in Chapter 6.

CHAPTER FOUR

HARDWARE AND PERFORMANCE

This chapter compares the performance of the CPU and GPU. The performance is measured by implementing the core functions introduced in Chapter 3 on both the CPU and GPU and measuring the time to perform each function. The time overhead when copying memory (voxel models) to the GPU and when copying back the GPU results back to the CPU is also included as well as the overall time for all the functions.

The two major components affecting the computing times for the CPU are the voxel size and the path resolution. The major influence on computing times for the GPU is voxel resolution as seen in the memory copy times and the voxel model update times. Overall, the computing time on the GPU was 550X's faster than the CPU for the smallest voxel models and 740X's faster for the larger voxel model. The timing reported for each of the functions is a subset of the overall time to perform the path planning and a summary plot of the overall time is presented at the end of this chapter.

It is very important to note that although the GPU based approach is over 500X's faster than the CPU approach, the performance on the GPU can still be dramatically increased far beyond what is reported in this section. The GPU performance can be improved by use of multiple GPUs and the latest GPU technologies such as OpenCL and the CUDA toolkit 4.0.

In addition, cloud GPU computing is now available. For example, Amazon.com offers a GPU computing node consisting of two nVIDIA Teslsa GPUs [48]. Each Tesla GPU has 448 Cuda processor cores and 3GB of memory giving twice the memory capacity of the GPU used for this research.

A single GPU is used throughout this research and both the part and blank voxel models are copied to the GPU for the duration of the algorithms presented in Chapter 3. In addition, the results of the ray casting algorithm for both the blank and part are stored on the GPU further reducing the available memory. The use of another GPU would allow for much smaller voxel sizes **and** result in **increased** performance.

Hardware

The code in this research was developed first on a Quadro FX 2800M nVIDIA GPU with and Intel i7-Q820 CPU and later on an nVIDA GeFORCE GTX 580 with an Intel i7-2600K CPU. The Intel i7-Q820 is a quad core processor with a nominal clock speed of 1.73 GHz and the i7-2600K is a quad core processor with a clock speed of 3.4 GHz. Both processors are capable of 8 simultaneous threads. The specifications of both GPUs are show in Table 4.1.

The graphics card executes parallel code by running parallel functions called kernels. Any function that can be ran in parallel can therefore by run by a number of threads processing the same kernel in parallel. Kernels are launched in parallel by using the special CUDA C API extensions.

The GPU allocates significantly more ALUs (Arithmetic Logic Units) to processing data than the CPU as seen conceptually in Figure 4-1[49]. This is because the CPU is used to run software programs with large memory demands whereas the GPU is designed to process large amounts of data quickly with kernels. However, the GPU architecture comes at the cost of limited local memory available for use by the kernels. Therefore good GPU code is designed to have limited memory demands and high computational demands.

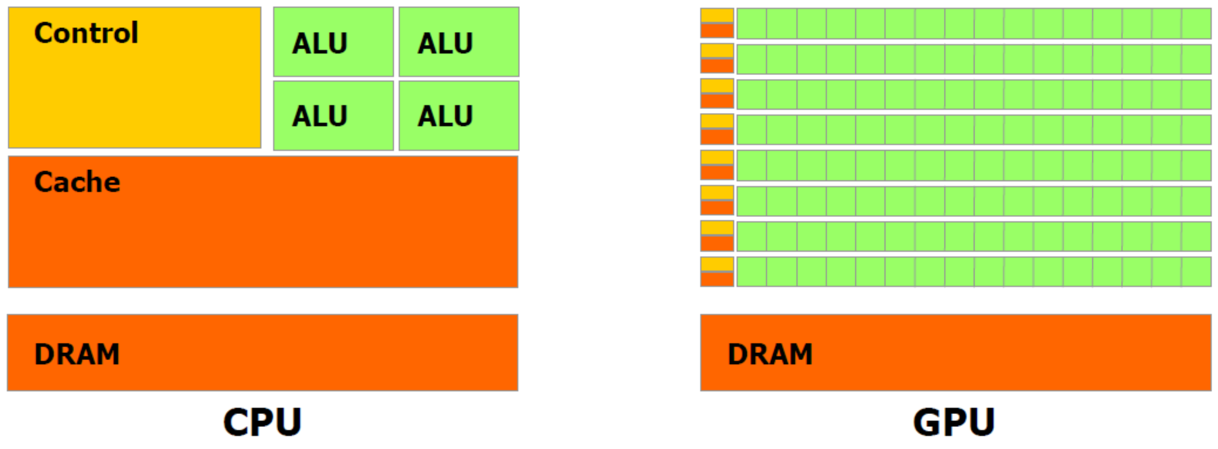


Figure 4-1: Architecture of CPU vs GPU

The differences among CUDA enabled GPUs are found in the allocation and size of local, constant and shared memory, compute capability, clock speed, and amount of global memory. The compute capability is related to the GPU architecture and the latest nVIDIA Fermi architecture has a compute capability of 2.0 which means it has four times the amount of CUDA processor cores than those introduced only a few years ago.

The theoretical maximum number of threads that can be launched at any given time by a kernel launch of GPU function call is 65,535 for all CUDA enabled GPUs. These threads are then scheduled by the GPUs multi-processors. At any given time, each multiprocessor can run a number of threads. As seen in Table 4.1, the Quadro FX 2800M card has 12 multi-processors. Each multi-processor can run multiple blocks of threads but is limited to 8 thread blocks. The maximum number of threads for all blocks running on a multi-processor at any time is 768. Each multi-processor processes 32 threads at a time (called a warp). The amount of shared memory and registers per multi-processor shown in Table 4.1 is what is available for the entire launch of threads for a single GPU.

Table 4.1: Graphics card specifications

Property	**GeForce GTX 580**	**Quadro FX 2800M**
Major compute capability:	2.0	1.1
Clock rate:	1594 MHz	1500 MHz
Total global memory:	1.5 GB	1.0 GB
Total constant memory:	64 KB	64 KB
No. of multiprocessors:	16	12
Cuda cores per mp:	32	8
Shared memory per mp:	48 KB	16 KB
32-bit registers per mp:	32 K	8 K
Threads in warp:	32	32
Max threads per block:	1024	512

The amount of shared memory, constant memory, register usage, and local memory allocated for a kernel is managed by the CUDA runtime. If the kernel has too much memory requirements then a launch error will occur and the size of the thread block must be reduced. There is a tradeoff between the amount of memory each kernel is permitted to consume and the number of kernels that can be launched. nVIDIA has provided an occupancy calculator to help determine the number of threads to launch in a block based on the memory demands of a given kernel.

Because the kernel launches are limited to the available memory, and different GPU architectures have different memory allocations, there can be significant time differences from GPU to GPU. As an example, the overall path planning method discussed in Chapter 3 was run on both the Quadro FX 2800M and the GeForce 580 GTX with a voxel size of 0.5mm and a path size of 3,164 points. The same GPU code resulted in computation times of 2.9ms on the Quadro and 0.89ms on the GeForce making the GeForce in excess of 3x's faster.

Performance

Each of the core algorithms described in Chapter 3 was implemented both on the CPU and the GPU to evaluate the overall performance of the algorithms in terms of time and the amount of performance gained by use of the GPU. Evaluation is based on varying the voxel sizes of the part and blank models and varying the path. There is no way to compare some of the functions with the GPU because the CPU has no such hardware implementations. For example, steps of the voxelization algorithm utilize the front and rear face culling hardware functions of the GPU and these do not exist on CPU hardware.

Each core function of this work is discussed individually and areas where parallel processing offers significant savings are highlighted. Then the results from the timing tests for that core function are presented demonstrating how the GPU performance compares to the CPU performance. First the algorithm to create the grid to cast the rays from is introduced and compared to a similar implementation on the CPU. Then this is repeated for the ray-casting algorithm, the path update algorithm, and the voxel removal algorithm.

The timing performance is measured for a maximum voxel resolution of 1mm and a minimum voxel resolution of 0.2mm. Figure 4-2 shows the voxel resolution of 1mm and Figure 4-3 shows the voxel resolution of 0.2mm. The underlying mesh from the input STL file can be seen in Figure 4-3 indicating that the voxel model is close to the resolution of the input file.

The timing analysis was carried out for a single roughing pass using a flat end cutter with a radius of 6.35mm. The overlap between paths is 65% of the cutter diameter. The results of the roughing pass for large and small voxel resolutions are shown in Figure 4-2 and Figure 4-3.

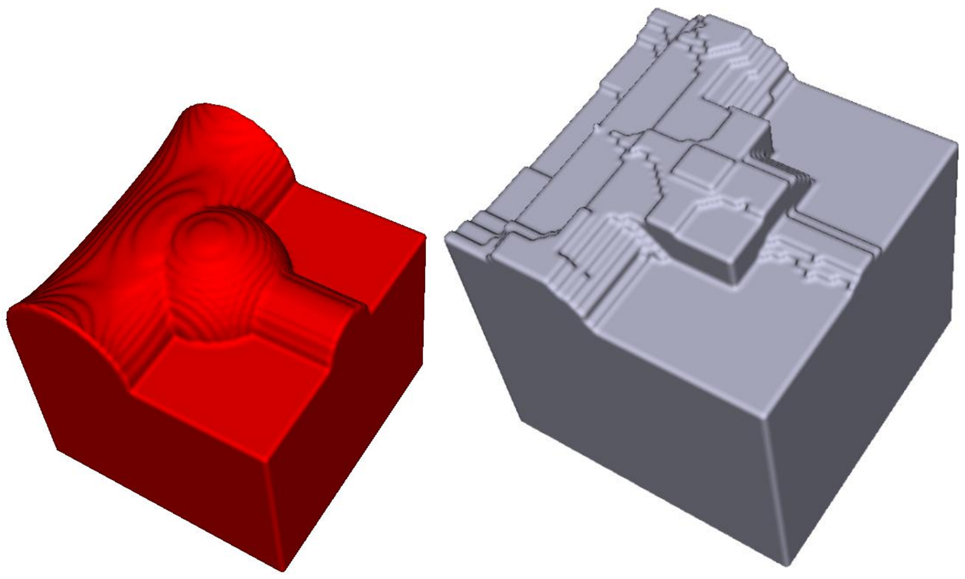


Figure 4-2: Voxel model with a resolution of 1mm and roughing pass results

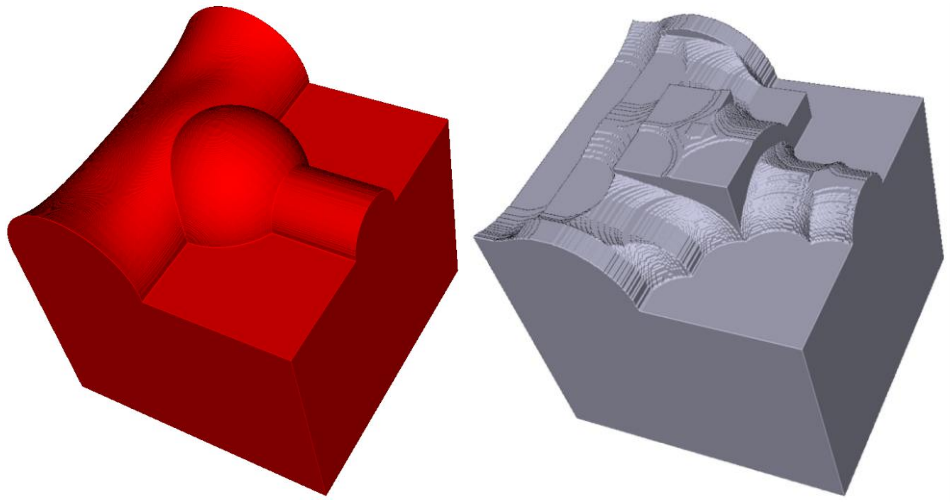


Figure 4-3: Voxel model with resolution of 0.2 mm and rouging pass results

Each voxel model used to benchmark the timing is of a 50.8x50.8x50.8mm part. Therefore, the dimensions of the voxel model were varied from 56x56x56 to 256x256x256. Each voxel model is comprised of voxels in the range of 0-255 is represented by an array of characters where each character is a byte. Therefore, the smaller model (larger voxels) has a memory footprint of 8x56x56x56 = 1.4 MB and the larger model (smaller voxels) has a memory footprint of 8x256x256x256 = 134 MB. The voxel model of the part and blank are the same size so the total memory that the voxel models of the blank and part occupy on the GPU is 2.8MB – 264MB. The memory limit of the GPU is 1.5 GB which means voxel models up to 578x578x578 can theoretically be used if both the blank and the part are to be contained on the GPU at the same time as in this research.

Because of the memory demands of the ray casting results, global variables, and code footprint, the voxel size for a part with square dimensions is limited to 0.2mm.

Smaller voxels caused GPU kernel launch errors. This is only a limitation of the current hardware configuration. Server based GPUs such as the nVIDIA Tesla with 3 GB global memory or the use of multiple GPUs would allow for significantly smaller voxels.

Grid Creation and Ray Casting

Chapter 3 presented the grid creation and ray casting algorithm where a grid is created on a plane and the rays are cast into the part from the grid. The grid is created based on the dimensions of projected bounding box that contains the part. The dimensions are the same as the number of voxels seen from the grid. Rays are cast from each point on the grid at the part and blank models. The intersections of the rays with the respective model are stored in an array.

The rays are incremented towards the voxel models from the grid in intervals of 3.0 voxels. At each point along the ray, the voxel volume is queried using tri-linear interpolation. Once the interpolated value of the part or blank volume along ray is greater than 128, then the bisection algorithm is used to determine the distance along the ray where the value is 128+/-0.001. The grid creation and ray casting algorithm is implemented the same way on both the GPU and the CPU with very small implementation differences.

The major time difference between the GPU and CPU for the ray casting algorithm is attributed to the linear nature of the CPU. On the CPU, each ray must be cast serially meaning every ray must be incremented individually with tri-linear interpolation performed at each step. In the GPU implementation, a kernel is launched allowing a single thread to calculate the intersection of each ray. This difference is highlighted in Figure 4-4.

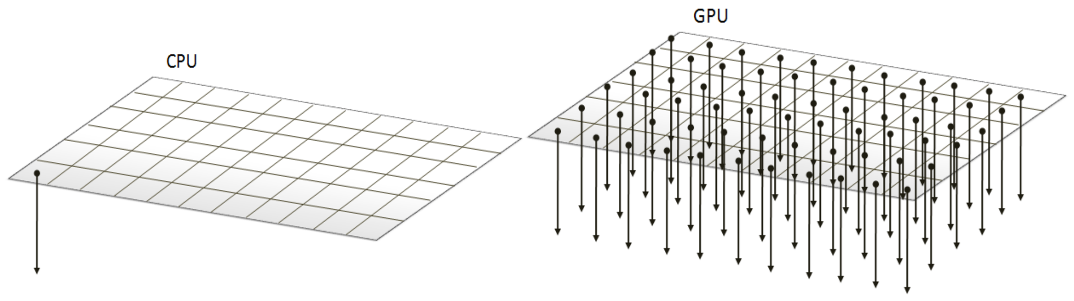


Figure 4-4: Ray casting from CPU vs. GPU

The time to create the grid and cast the rays for both the CPU and GPU vs. voxel size is shown in Figure 4-5. The implementation varies only with voxel size because the grid created on the plane only depends on the voxel size only as it is not affected by the size of the path. The GPU outperforms the CPU by 1,000X's for the larger voxels size of 1mm (smaller grid) and by nearly 14,000X's for the smaller voxel size of 0.2mm. The GPU timing remains nearly constant because the GPU kernels are all able to launch in parallel for the range of voxel sizes tested.

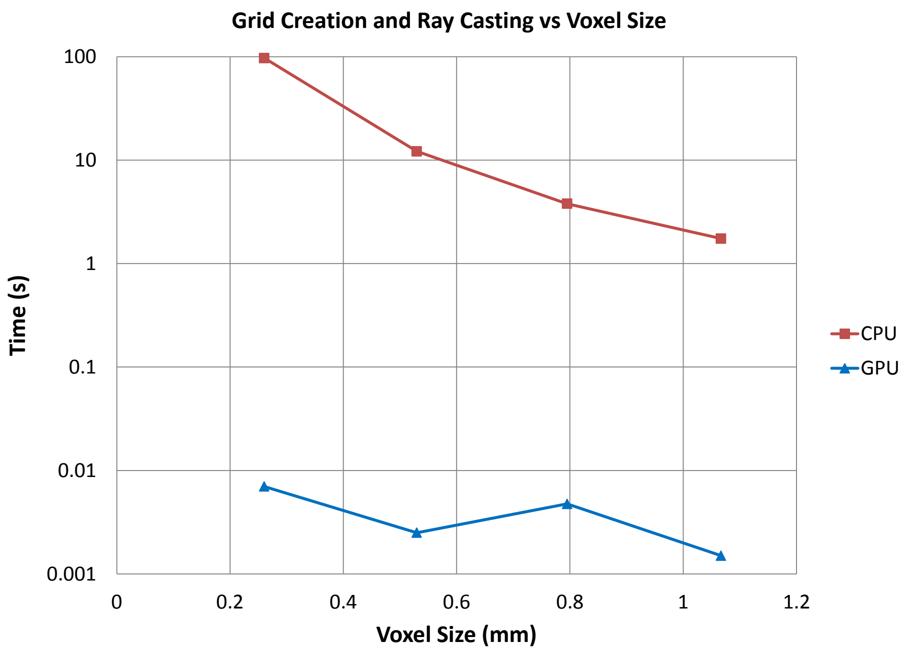


Figure 4-5: Time for grid creation and ray casting into voxel models

Optimal Plane Selection

The algorithm presented in Chapter 3 to determine the optimal plane uses the grid creation and ray casting algorithm for each orientation. A search plane is placed at each face of the gyro-elongated bi-cupola shown in Figure 4-6.

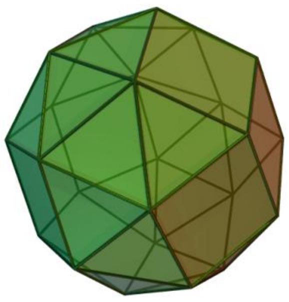


Figure 4-6: Optimal plane search planes

For this shape, there are 26 planes. According to Figure 4-5, utilizing the GPU to determine the optimal plane instead of the CPU results in a speed up of nearly 360,000X's.

Path Update Based On Cutter Surfaces

The tool path is generically created based on the dimensions of the blank, the size of the cutter, the overlap between cutter passes, and the path resolution. This generic path provides the locations that are used in the path update algorithm. Although the number of tool points is based on the cutter size and path overlap it is affected the most by the path resolution. The path resolution is selected as a fraction of the voxel size to ensure that every point in the tool path is updated at a higher resolution than the path grid. This is accomplished by bilinear interpolation of the results from the ray casting approach. For each point in the tool path, the location of the cutter contact point is updated so that it is either at the top of the part surface or at the depth of cut without gouging the surface.

The path update algorithm utilizes the plane of voxels underneath the cutter and centered at the cutter contact point. For each voxel in the plane beneath the cutter, the distance to the cutter surface and the distance to the part surface are used to update the center tool point. The maximum distance is selected that does not cause gouge or go beyond the desired depth of cut. Therefore, the time taken by the update path function is related to the number of inquiries – which is a 1:1 relationship with the number of path points and the number of voxels in the update plane.

The number of voxels in the update plane is related to the cutter radius (and ramp) and the voxel size. For example, a diameter of 6.35mm, a voxel size of 0.5mm, and a ramp of 3 voxels results in a plane under the cutter that is (6.35/0.5 + 6)^2 = 350 voxels. The results of the path update times for the GPU and CPU are shown in Figure 4-7 and Figure 4-8. For the case of the larger voxels, the time it takes to update the path on the GPU is approximately 100X's faster. For the smaller voxels, the GPU is nearly 375X's faster. The graphs also show that voxels at half the size at with similar number of tool points take roughly 10X's longer for the CPU to process whereas there is very little change in the time taken by the GPU due to the parallel processing of the path. The figures include the time it takes to convert the tool points from the CPU data type to the CPU data type before processing and copying them back after processing.

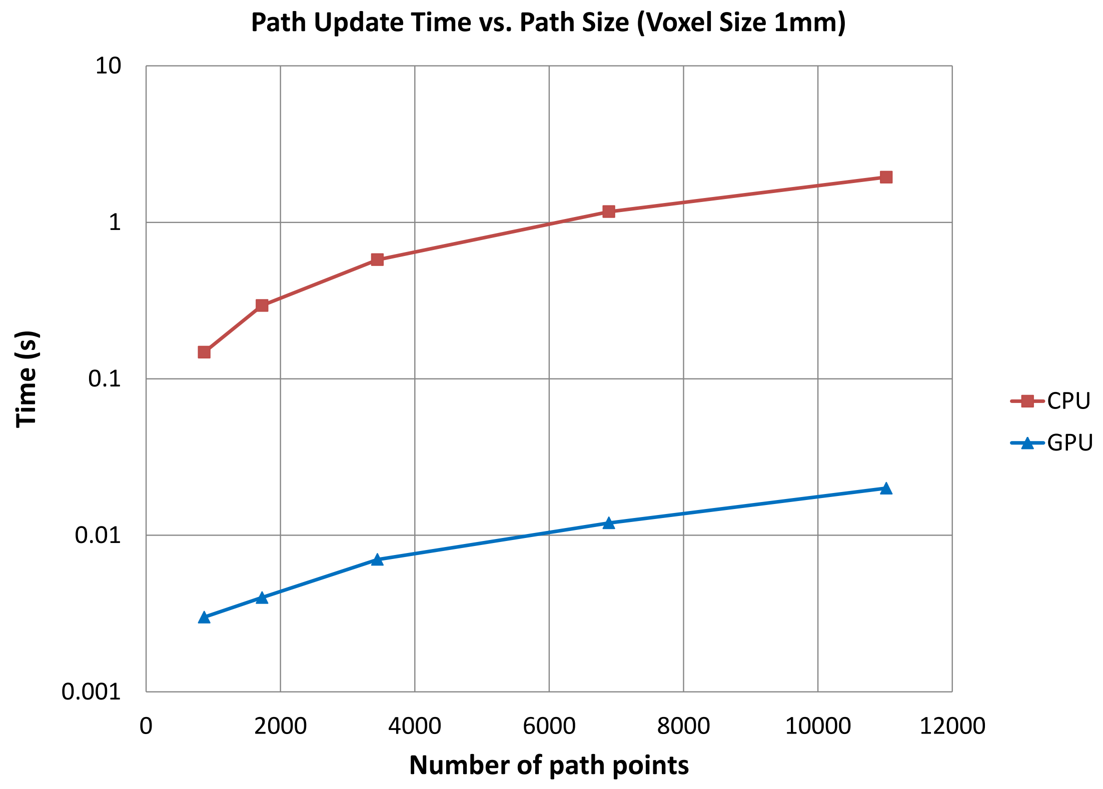


Figure 4-7: Path update time vs. number of points for 1mm voxels

Figure 4-9 shows the results when the GPU is used for high resolution paths that result in significantly more tool points. Even for 47,000 tool points the path update takes a fraction of a second.

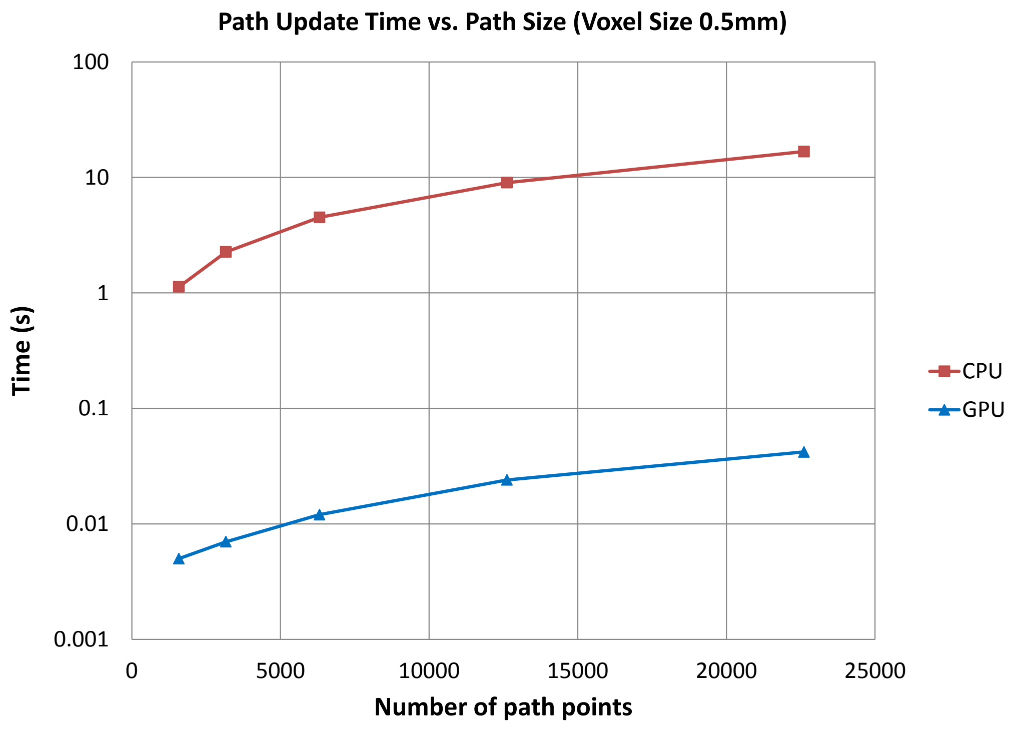


Figure 4-8: Path update time vs. number of points for 0.5mm voxels

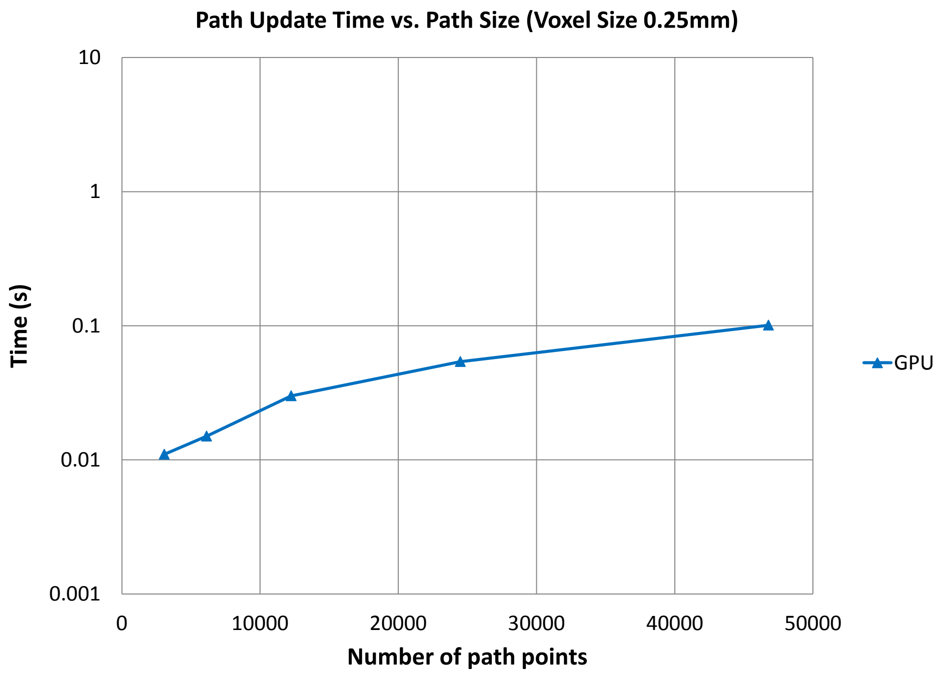


Figure 4-9: Path update time vs. number of points for 0.25mm voxels GPU only

Model Update Based on Path

Once grid has been created based on the ray-casting algorithm and the path has been updated, then the material is removed according to the path update algorithm discussed in Chapter 3. For each tool point, a voxel box is created around the cutter based on the dimensions of the cutter and the desired voxel ramp. This is the most time consuming portion of the code because each point in the voxel box must be visited and the blank must be updated according to the calculated distance from each voxel to the surface of the generalized cutter.

The amount of time it takes to update the voxel bank model for the CPU and GPU implementations is shown in Figure 4-10-Figure 4-12. Figure 4-10 shows that for the coarse voxel model that the CPU takes about 5 minutes and the GPU code takes about 500 milliseconds.

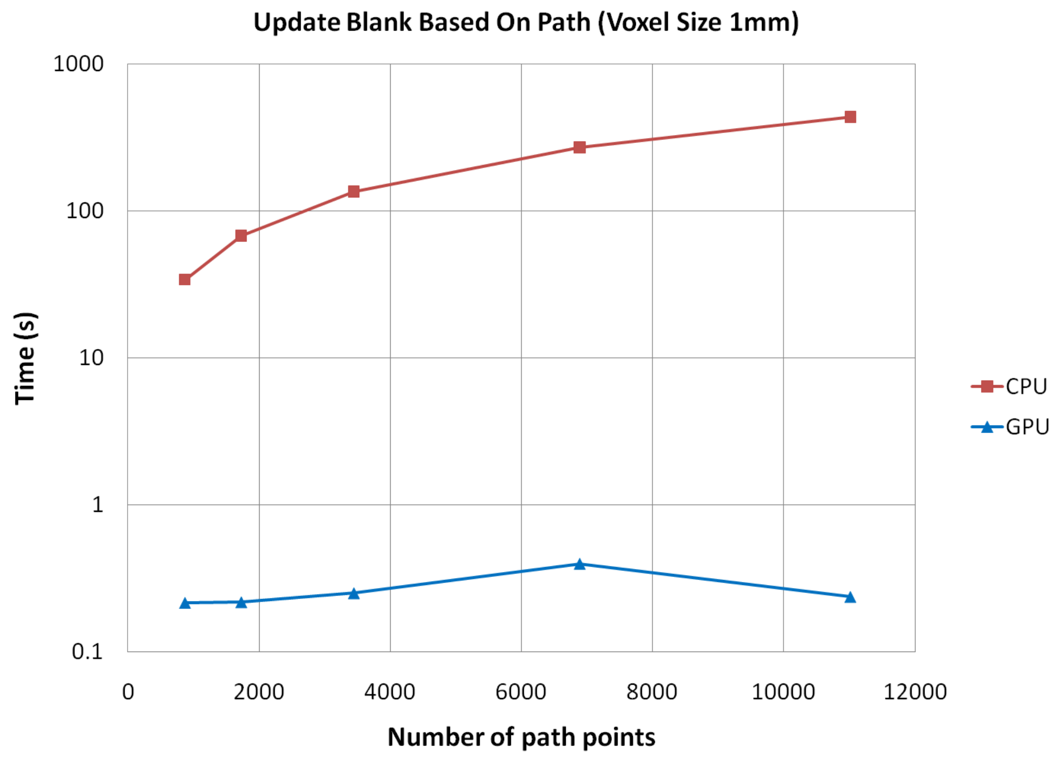


Figure 4-10: Update of blank timing vs. path size for 1mm voxels

The results for the 0.5mm voxels are shown in Figure 4-11. The CPU takes about 70 minutes whereas the GPU only takes about 7 seconds.

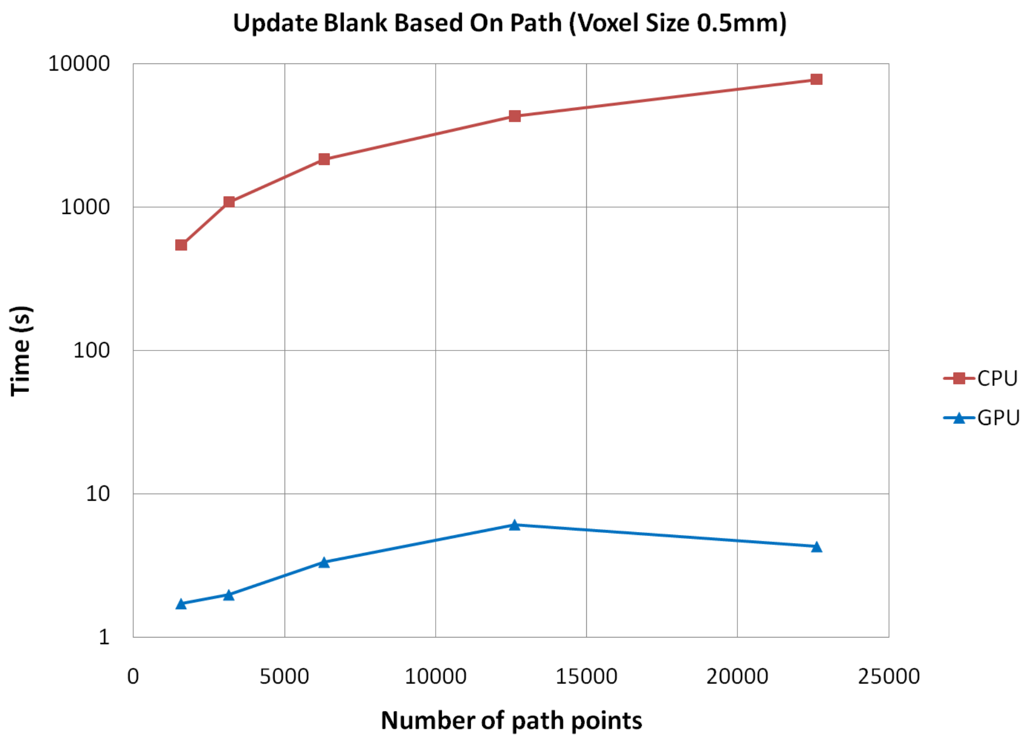


Figure 4-11: Update of blank timing vs. path size for 0.5mm voxels

Figure 4-12 shows the timing of the GPU for a higher resolution voxel model of 0.2mm when evaluating values beyond those tested on the CPU. Even for a high-resolution voxel model and nearly 25,000 path points the GPU only takes approximately 3 minutes. The GPU takes only 12 minutes when the number of path points is increased to 100,000.

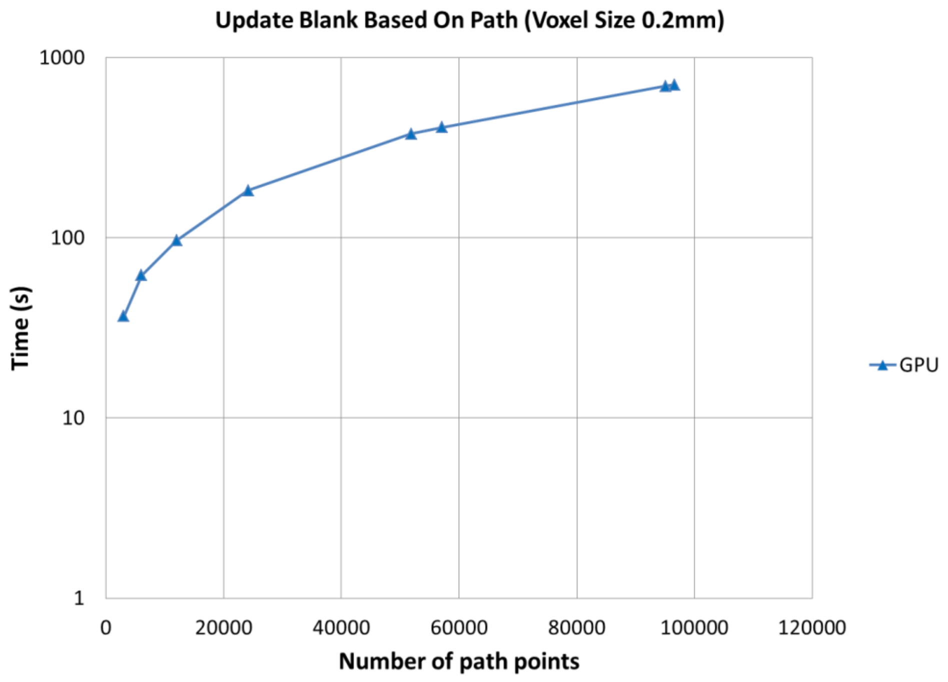


Figure 4-12: Update of blank GPU timing vs. path size for 0.2mm voxels

Performance Summary

The above charts did not address the time consumed by the GPU when performing the necessary voxel model memory copies from the CPU to the GPU for processing. However, the calculated results obtained from the GPU were included in the timing information for updating the blank. The additional overhead incurred when copying the voxel models is related to the size of the voxel and blank models that are copied to the GPU. The amount of overhead vs. the voxel resolution is shown in Figure 4-13.Voxel sizes higher than 1.1 mm resulted paths with insufficient resolution. Voxel sizes lower than 0.2 mm resulted in launch errors although the blank and voxel models were successfully copied to GPU memory.

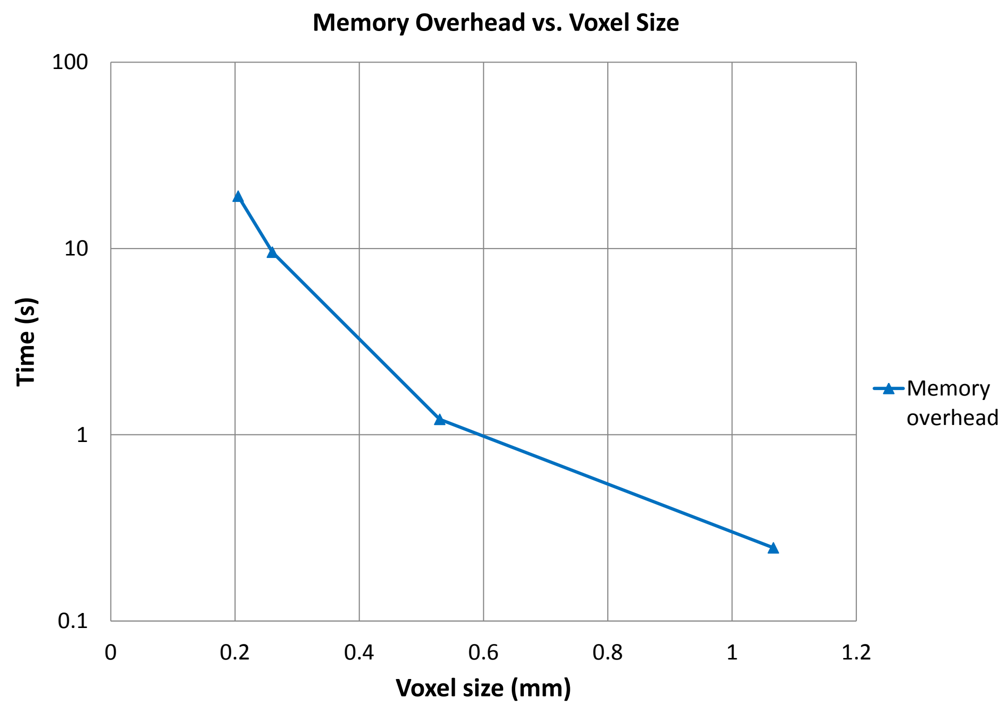


Figure 4-13: Memory copy overhead vs. voxel resolution

The overall timing of the algorithms benchmarked in this chapter for voxel sizes of 1mm and 0.5mm are shown in Figure 4-14 and Figure 4-15.

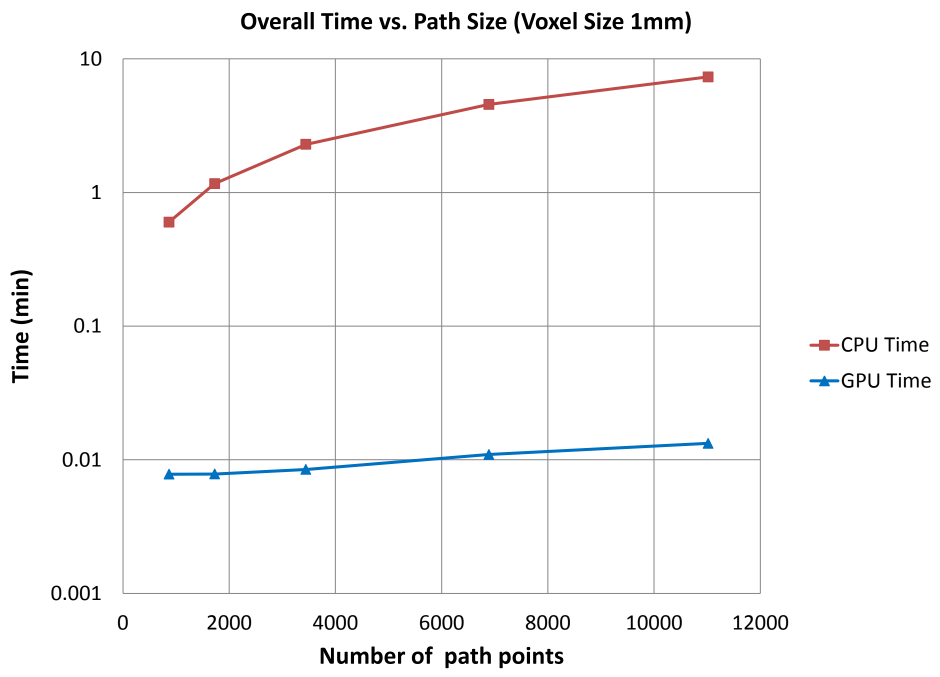


Figure 4-14: Overall time on GPU and CPU for 1mm voxels

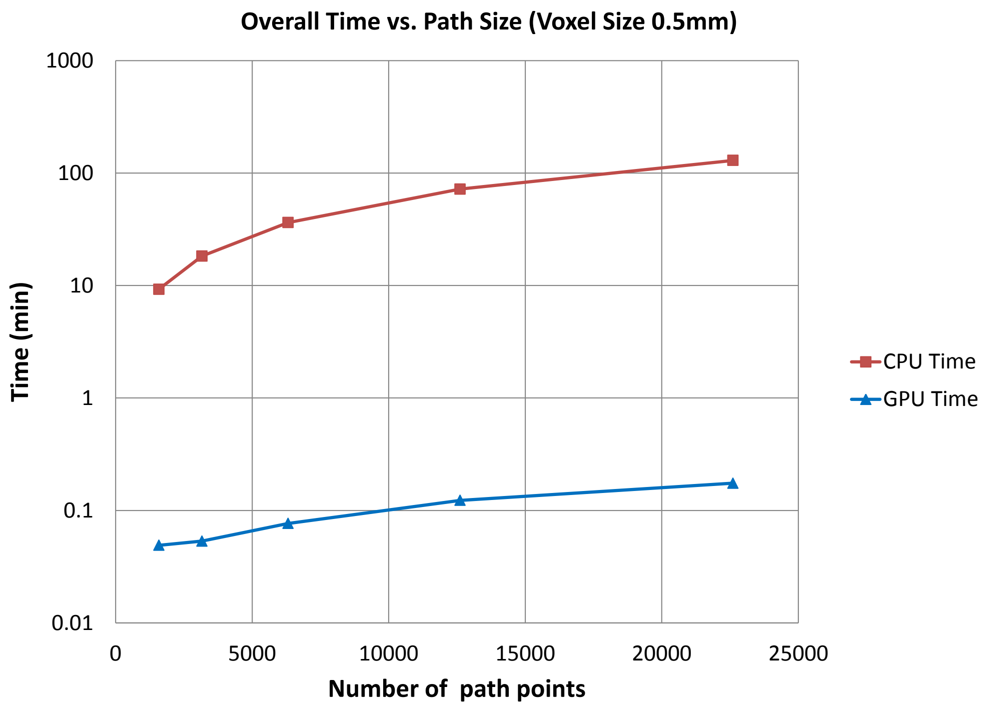


Figure 4-15: Overall time on GPU and CPU for 0.5mm voxels

The overall time for the algorithm to create cast rays and perform bisection, update the path, and remove the voxel material for 1mm voxels took 7.32min on the CPU and 0.013min on the GPU resulting in greater than 550Xs the CPU speed. The overall time for the smaller voxel size of 0.5mm took 130minutes on the CPU and 0.2 minutes on the GPU resulting in greater than 742Xs the CPU speed.

This dramatic increase in speed is by no means at the limit of what is possible by parallel processing. The code developed in this research can be extended to multiple GPUs either locally or in the cloud. The importance of the timing results of this chapter is that they demonstrate that this framework can be used to run thousands of paths in very close to real time allowing for optimization of the path planning process based on metrics of the resulting path.

Limitations and hardware options

Although the parallel approach benchmarked in this chapter results in over 500X's the performance, it is still limited by the available memory (global, local, constant, and register memory) and the number of parallel processor cores (and their clock speed) of the single GPU used. The global memory limitation restricts either the voxel model resolution or the size of the model that can be analyzed with this approach. For example, the timing measurements in this chapter were limited to a small example part of 50.8x50.8x50.8mm constrained to a minimum voxel size of 0.5mm. To either increase the accuracy of the part model or represent larger parts requires more voxels. However, any increase would exhaust the available memory of a single GPU.

The limitation of constant, shared, local, and register memory limits the number of threads than can be launched by a single multiprocessor. This is because each thread consumes a portion of each memory type as dictated by the CUDA runtime. The speed is limited not only by the amount of constant, shared, local, and register memory but also by the number of parallel processors. The clock speed for a parallel processing approach is less important than the amount of threads it can process at a single time. The evidence of this memory and processor limitation is seen between different GPU's (Table 4.1) where the GTX 580 results in 3X's the performance of the FX 2800M. This increase in performance is the result of more threads launched (due to increased constant, shared, local, and register memory) and more processor cores.

The limitations imposed by the available GPU memory and number of processor cores can be eliminated in one of two ways. For a single GPU, these limitations will be removed by future hardware that will have more global, constant, shared, and register memory and more parallel processors per GPU with higher clock speeds. Since 2001, the available global memory per GPU has increased from 64MB to 3GB as shown in Figure 4-16. Market demands will cause this trend to continue to increase.

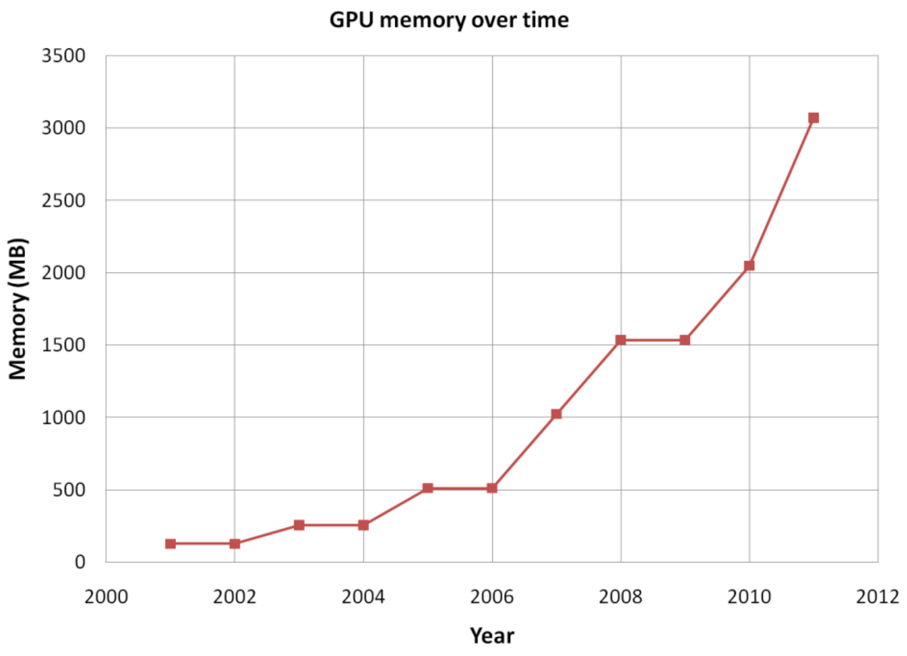


Figure 4-16: GPU memory increase over time

The other way to eliminate these performance limitations is by exploiting the parallel approach introduced in this research to utilize multiple GPUs and take advantage of each GPU's memory and processor cores.

The amount of memory necessary for a given model can be determined by the number of voxels as defined by Equation(1.1),

$$Voxels = width \cdot height \cdot length \cdot res^3 \qquad (1.1)$$

Where *width*, *height*, and *length* are the dimensions of the part in *units* and *res* is the resolutions of voxels in PPT (parts per thousand units). Each voxel is represented by a char data type and takes 8 bytes. The total memory consumed by a given part model can be expressed by Equation (1.2).

$$mem = voxels \cdot \frac{byte}{voxel} \tag{1.2}$$

To store both the part and blank models requires twice the *mem* value. The maximum memory limit of the current GPUs is 1.5GB for high end desktop cards and 3GB for server-based cards. Therefore, for a desktop GPU, when the number of voxels exceeds approximately 196K (for a single model) multiple GPUs are necessary. The amount of memory per normalized unit cube of 1x1x1 units^3 can be estimated from the voxel resolution in parts per thousand units as seen in Figure 4-17. Figure 4-18 shows the voxel resolution of a model in parts per thousand and the number of 1.5GB GPUs that would be necessary to store the model in global memory.

The memory limitation is based on part and blank models that are not stored optimally. Algorithms to “brickerize” voxel models can be used to create a single large voxel in the interior of a part or blank model file reducing the memory dramatically. Brickerizing is a method to compress the voxel model so that interior voxel do not consume as much memory and only the important surface voxels are represented.

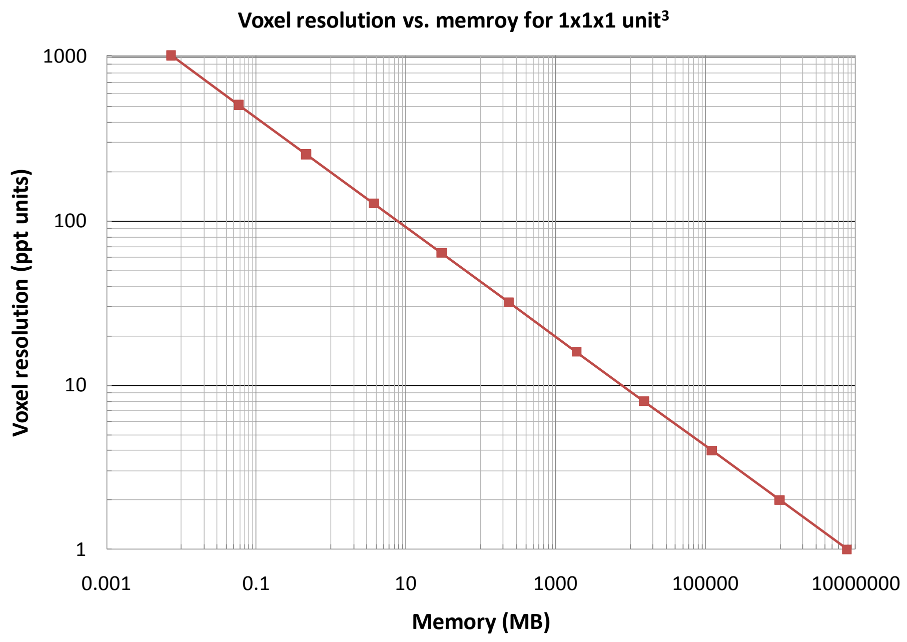


Figure 4-17: Voxel resolution in parts per thousand vs. memory consumption

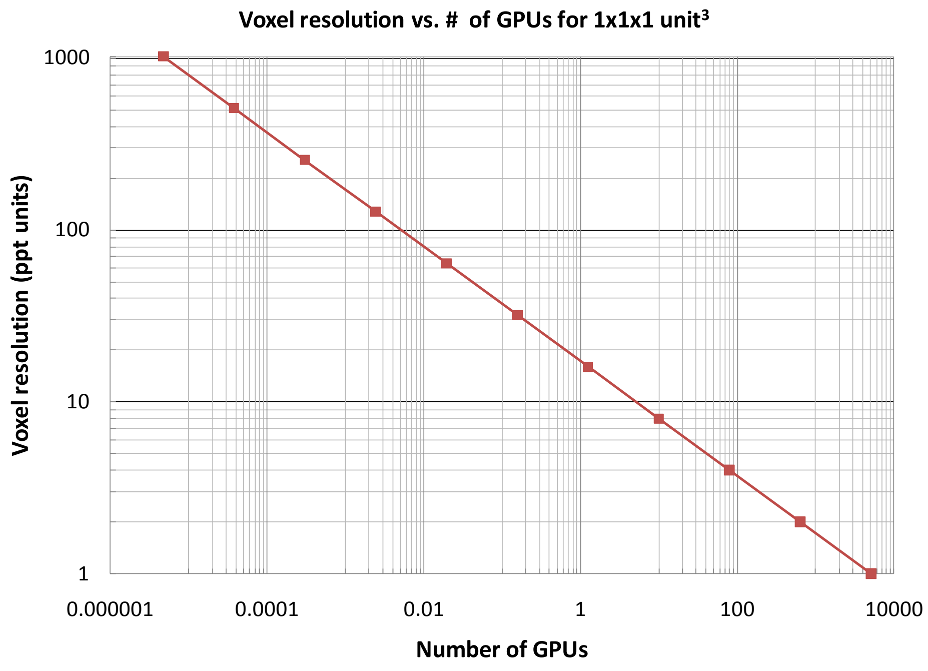


Figure 4-18: Voxel resolution in parts per thousand vs. number of GPUs

Figure 4-17and Figure 4-18 can be used to estimate the memory requirements for an arbitrary part size. For example, a 10x10x10 cm^3 part with a voxel resolution of 10 parts per thousand (100μm) would require approximately 1800MB or 1.8 GB and require 2 GPUs to store the part and the same amount of memory to store the blank. Therefore to store both part and blank models would require 4 desktop GPUs or 2 Tesla server GPUs.

Although these numbers may seem high, there are already turnkey solutions for GPU cluster processing. For example, the Dell PowerEdge C410x PCIe Expansion Chassis is designed to hold 16 GPUs. In addition, Amazon Cloud services gives access to any number of server GPUs [48]. MATLAB's latest release has a Parallel Computing Toolbox that can be used with or without Amazon's clouds services to utilize multiple GPUs. Therefore, the only limitation to this approach is the effort required to extend the code to utilize multiple GPUs. Increasing the number of GPUs not only has the advantage of more global memory to store the voxel models but also increases the local, constant, shared, and register memory which means that for each additional GPU, 2Xs the threads can be run simultaneously drastically reducing the computational time.

The computation time for a single GPU is comprised of the time to copy the memory to the GPU, the time to create the grid and cast the rays, and the time to update the voxel blank as the tool traverses the tool path. The test results at the beginning of this chapter can be used to get an estimate of these times in terms of the number of voxels, the dimensions of the plane, and the number of tool points. The time to copy the voxel to the GPU or the memory overhead is linearly related to the number of voxels by Equation (4.3).

$$Time_{Copy} = 0.000001 \cdot Vox_{Count} \tag{4.3}$$

The time to cast the rays is a function of the dimensions of the ray casting plane's *width* and *height* in voxels as shown in Equation (4.4).

$$Time_{Ray} = 2 \times 10^{-7} length_{Vox} \cdot width_{Vox} + 0.0012 \tag{4.4}$$

The time to update the path based on the path template and results of the ray casting algorithm is related to the number of path points, *TP*, as shown in (4.5)

$$Time_{Update} = 2 \times 10^{-6} TP + 0.0012 \tag{4.5}$$

The time it takes for the parallel algorithm to process all of the tool points and update the blank model is estimated by Equation (4.6).

$$Time_{Points} = 6 \times 10^{7} \cdot VS^{2.6} \cdot TP \tag{4.6}$$

Where $Time_{Points}$ is the time consumed to remove the blank voxels, *VS* is the voxel size, and *TP* is the number of tool points. The total time is calculated by summing the time for the memory overhead, ray casting, path update, and tool point processing as shown in Equation (4.7).

$$\begin{aligned} Time = {} & Time_{Copy}(\text{num vox}) + Time_{Ray}(\text{plane dims}) \\ & + Time_{Update}(\text{tool points}) + Time_{Points}(\text{tool points, vox size}) \end{aligned} \tag{4.7}$$

Because the methodology presented in this research is parallel, it can be run on any number of GPUs for significant time savings. There are a number of strategies that can be employed to utilize multiple GPUs. Each additional GPU will result in approximately 2X the speed up. Figure 4-19 shows the effect that increasing the number of GPUs has on the normalized time. The normalized time is the time calculated by dividing *Time* in Equation (4.7) by *Time* multiplied by the number of GPUs. Utilizing multiple GPUs will not result in exactly linear increases in performance due to memory overhead and the processing time associated with managing multiple GPUs. However, Figure 4-19 provides a good approximation.

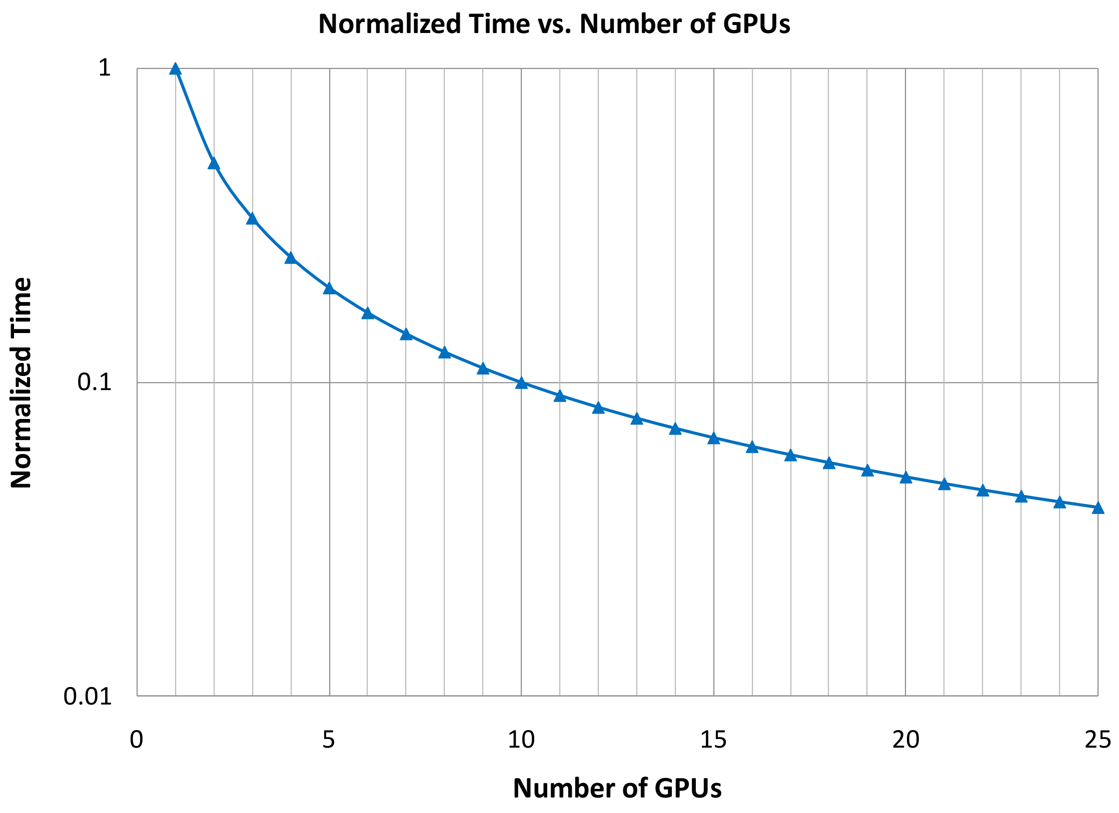


Figure 4-19: Example of speed increase due to multiple GPUs

The performance of the parallel approach will also be accelerated by the increase in GPU clock speed. Every year market demands have pushed the clock speed of the GPU to nearly 1.25Xs that of the previous year. Figure 4-20 shows the effect that increased processor speed has on the time spent processing on the GPU. The increase in speed is due to the increased density of transistors per integrated circuit according to Moore's Law [50].

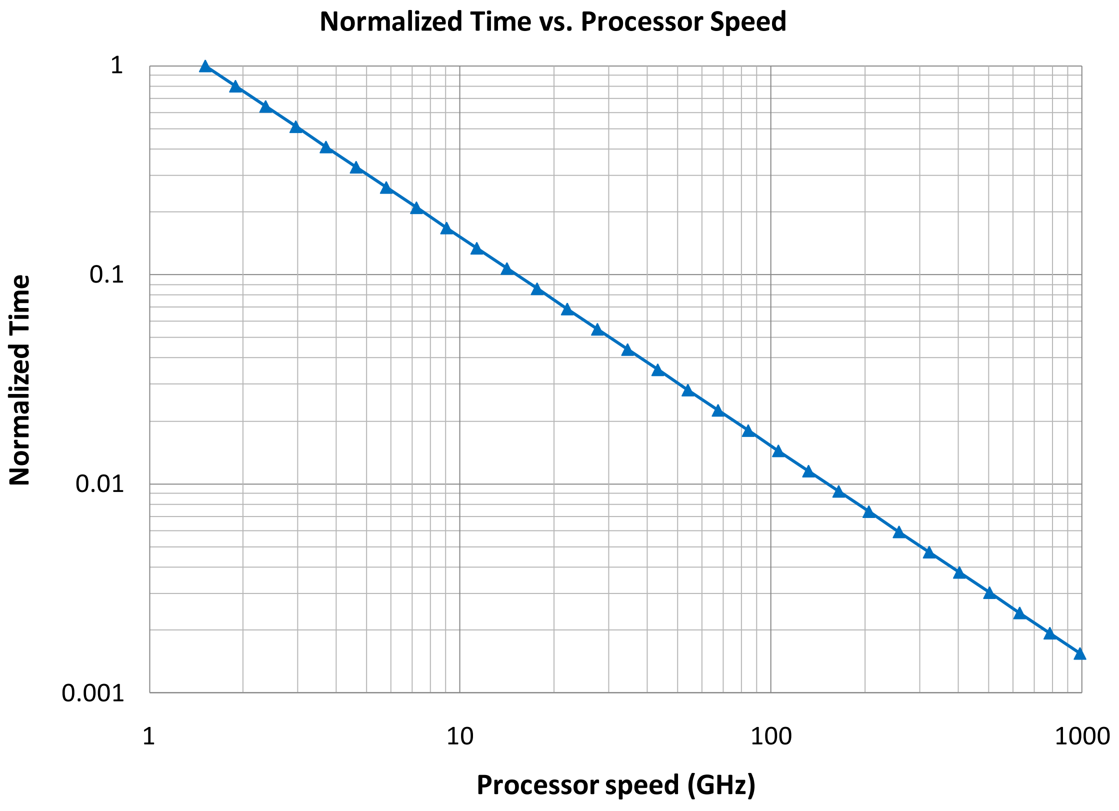


Figure 4-20: Effect of processor speed on GPU computing time

The actual computing speed increase, in terms of number of GFLOPS (giga-floating point operations per second), is about 1.5Xs every year. This is because the GPU is parallel and increased transistor density translates to more kernel memory and more processing speed per multi-processor per year. The normalized time vs. increased capabilities per year and additional GPUs is shown in Figure 4-21. Figure 4-21 shows that in approximately 3 years that 3 GPUs will reduce the computation time by 10X.

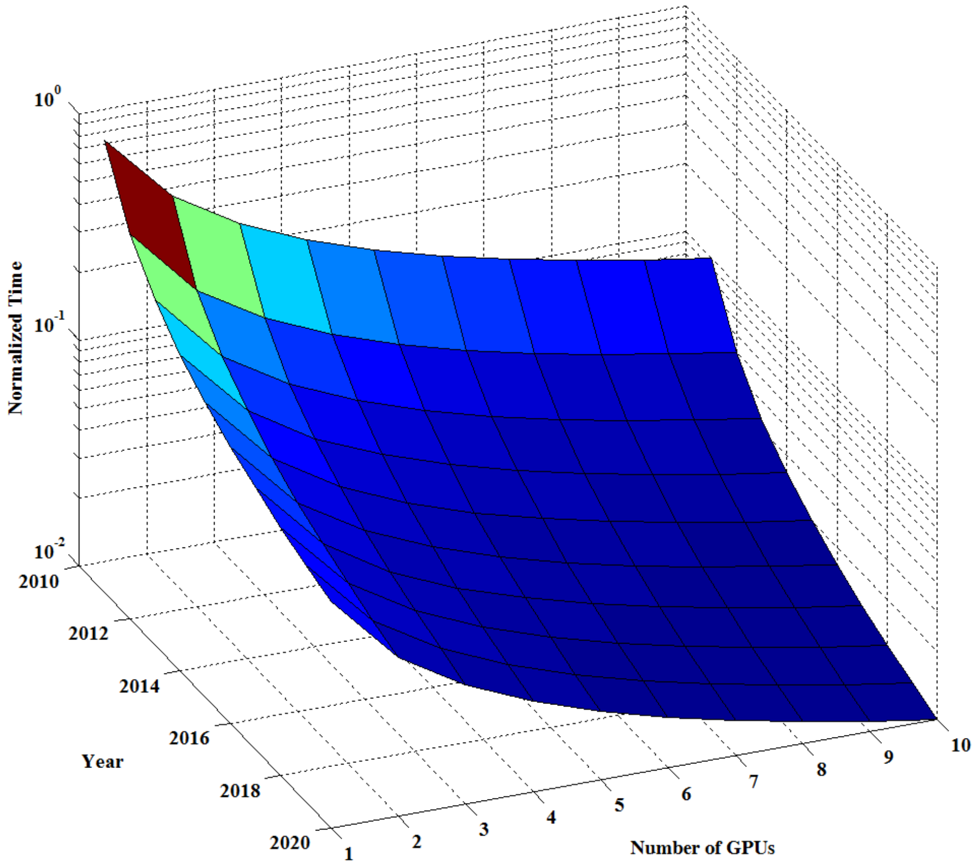


Figure 4-21: Normalized time as a function of year and number of GPUs

This chapter demonstrates that the methodology presented in this research can be used to generate any number of tool paths in close to real time by properly extending the code to multiple GPUs. The results of this chapter can also be used to understand what would be required with today's hardware for a specific part size and desired resolution.

CHAPTER FIVE

PATH METRICS

This chapter highlights the importance of the automated tool path planning approach presented in Chapter 3 and the significance of the timing results from Chapter 4. In Chapter 1, the need for an automated path planning tool was presented. This chapter presents how the GPU accelerated framework enables this need to be met by generating paths and then calculating the metrics of the path to evaluate its performance.

As mentioned in Chapter 1, due to the nature of manufacturing schedules and market competition, process engineers typically have a limited amount of time to generate a tool path. Therefore, they take the traditional design route, rules of the trade, and a reliable CAM package to create a safe and reliable tool path. Unfortunately, this tool path is by no means optimal because the amount of available information is not fully considered. In fact, the amount of information available to optimize a tool path is beyond human ability to process in any reasonable amount of time; if at all. Options such as tool holder geometry, cutter material choice and selection of inserts and insert material, coolant concentrations, alternative tool geometries, and etc. are much too difficult to optimize without support.

In this chapter, the typical design process used to select a tool and determine the path metrics is presented. This is an input-output process with a number of fixed decisions along the way. The method described in Chapter 3 is also an input-output process. However, the difference is that by utilizing parallel processing by GPU accelerated algorithms, the process engineer can now evaluate hundreds to thousands of tool paths in the same amount of time it would take to evaluate one tool path manually or by using a CAM software package. As mentioned in Chapter 4, the timing of the approach described in this research is scalable with the use of additional GPUs meaning that a very large number of tool paths could be evaluated in close to real-time with this framework.

Traditional approach to path planning

Traditional path planning begins with a material and desired cutting operation or sequence of operations such as rouging, semi-finishing, finishing, and etc. Tool manufacturers provide values for the recommended tool surface speed, v_c, and feed per tooth, f_z, or – in the case of a non-flat end cutter – the effective chip width, h_{ex}, for a given material. ISO standard ITS 4949:2003, classifies materials as steel (P), stainless steel (M), cast iron (K), aluminum (N), heat resistant super-alloys (S), and hardened steel (H).

Figure 5-1: ISO material classification

The material properties are typically a consideration of the design engineer not the process engineer. However, the material properties significantly impact the machining performance. Sandvik, a machine tool manufacturer, ranks materials according to the performance of their cutters in that particular material using their Coromat Material

Classification (CMC). For example, the ranking system for steel when cut by using a tool with round GC4220 inserts is shown in Table 5.1. Also shown is the recommended surface speed, v_C, and recommended effective chip width *hex*. The values in Table 5.1 are only for a single type of insert and for large engagement. Another similar table is used for cutters with small engagement.

Table 5.1: Steel cutting recommended surface speeds and effective chip widths

Material		Insert - GC4220			
Steel P	CMC Index	hex (mm) =	***0.1***	***0.2***	***0.3***
Unalloyed; C 0.10-0.25%	1.1	Vc (mm/min) =	570	560	520
Unalloyed; C 0.25-0.55%	1.2	Vc (mm/min) =	510	500	470
Unalloyed; C 0.55-0.85%	1.4	Vc (mm/min) =	425	415	390
Low alloyed; Hardened	2.2	Vc (mm/min) =	260	255	240
High alloyed; Tool Steel	3.22	Vc (mm/min) =	135	135	120

For the same material, with the same insert geometry, the selection of the cutting tool is not complete as there are a wide variety of cutting grades to choose from. Figure 5-2 shows the available insert grades from Sandvik for steel:

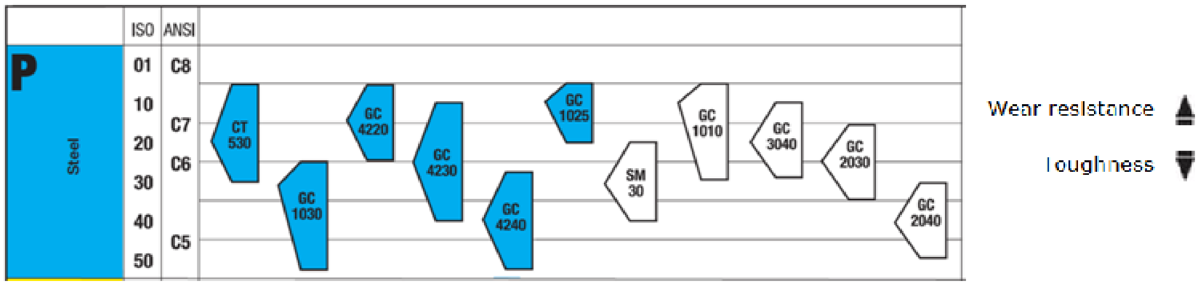


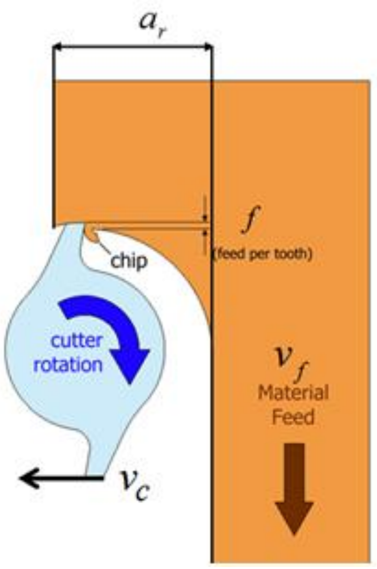
Figure 5-2: ISO steel grade and insert selection [51]

As seen in the above figure, there is an overlap between acceptable inserts that can be chosen. For example, low-alloy steel can be cut using 6 different inserts with speeds ranging from 60 m/min to 260 m/min.

It is clear from the Table 5.1 and Figure 5-2 that even if a specific tool and insert size are selected that there is a significant amount of information to consider simply based on the blank material. In addition to understanding the effect of the material, the process engineer must select the tool holder to support the desired operation, the cutter geometry (flat end, ball, end, toroidal, and etc.), radius, number of cutter teeth (2-N), and spacing between the teeth (coarse, fine). The tradeoffs between the various options are not well defined and it is likely that there is a more productive milling strategy in nearly every path planning case.

Traditional Path Parameters and Metrics

After the process engineer selects the tool geometry and obtains the recommended surface speed and the feed per tooth (or effective chip width) for the given material, then the cutting model [52] can be used to calculate the spindle speed and feed rate.



v_c: Cutting speed or tool surface speed $[m/\min]$

MRR : Material Removal Rate $\left[mm^3/\min\right]$

a_r : Radial depth of cut $[mm]$

a_p : Depth of cut $[mm]$

v_f : Feed rate or linear speed $[mm/\min]$

N : Spindle speed $[rev/\min]$

n : Number of teeth

f = Feed per tooth $[mm/(rev \cdot tooth)]$

t = Machining time $[\min]$

l = Cutting length $[mm]$

Figure 5-3: Cutting parameters for path metrics

The cutting speed, v_c, is the peripheral speed of the cutter and is related to the diameter, *D*, of the cutter and the spindle speed, *N*.

$$v_c = \frac{\pi DN}{1000} \tag{5.1}$$

For a flat end mill the cutting diameter is the same as the tool diameter. However, in non-flat end cutting, the cutting diameter is replaced by the actual cutting diameter at a specific depth of cut.

$$v_c = \frac{\pi D_{cap} N}{1000} \tag{5.2}$$

For ball and toroidal end cutters, the actual cutting diameter is based then the depth of cut, a_p, and radial depth of cut, a_r. The actual cutting diameter can be calculated using Table 5.2 [51]**.** The spindle RPM is calculated by the recommended surface speed and actual diameter by use of Equation (5.3) when the value of the diameter is given in mm.

$$N = \frac{1000 v_c}{\pi D_{cap}} \tag{5.3}$$

Table 5.2: Diameter at actual depth of cut

Cutter Type	**Actual Cutting Diameter at *a_p* (mm)**	**Feed per tooth (mm/tooth)**
Toroidal	$D_{capT} = D - \sqrt{iC^2 - \left(iC - 2a_p\right)^2}$	$f_z = \frac{h_{ex} \cdot iC \cdot D_{cap}}{4 \cdot \sqrt{a_p \cdot iC - a_p^{\,2}} \cdot \sqrt{D_{cap} \cdot a_r - a_r^{\,2}}}$
Ball End	$D_{capB} = \sqrt{D^2 - \left(D - 2a_p\right)^2}$	$f_z = \frac{D \cdot h_{ex}}{\sqrt{D_{cap}^{\,2} - \left(D_{cap} - 2a_r\right)^2}}$
$D =$ tool outer diameter; $f_z =$ feed per tooth; $iC =$ insert diameter; $a_p =$ depth of cut	$D_{cap} =$ actual diameter at cut; $h_{ex} =$ effective chip thickness; $a_r =$ radial depth of cut	

The spindle speed *N*, number of tool teeth *n*, and the feed per tooth f_z are used to calculate the table feed rate. The feed per tooth for a flat end mill is constant and recommended values are provided by tool manufacturers based on the material that is being cut. The feed per tooth for ball or toroidal end mills is calculated using Table 5.2 and is based on the effective chip thickness for a given cutter ,the selected depth of cut, and radial depth of cut. The feed rate is calculated using Eqation (5.4). The feed rate and spindle speed are used with the tool path to post process the path into G-code by the process engineer.

$$v_f = N \cdot n \cdot f_z \tag{5.4}$$

The metrics of the path are the machining time and MRR (Material Removal Rate). The machining time is the length of the path divided by the table feed rate:

$$t = \frac{l}{v_f} \tag{5.5}$$

The material removed by the cuter as it follows a path is related to the tool radial depth, of cut a_r, the tool axial depth of cut, a_p and the feed rate v_f . The volume or material removed per unit time, or MRR, is a measure of cutting efficiency. The MRR is related to the feed rate by the Equation (5.6).

$$MRR = a_r \cdot a_p \cdot v_f \tag{5.6}$$

GPU Based Path Metrics

The path metrics for the GPU based approach described in Chapter 3 are based on the tool path length, feed rate, and the amount of voxel material removed from the blank. The tool paths generated in Chapter 3 are 3D points in Cartesian space. Therefore, the tool path distance can be calculated by summing up the distances between adjacent points. The distance *d,* between points is the norm of the difference between them.

$$d = \sum_{\mathrm{P = P_0}}^{N} \left\| \mathrm{ToolPoint}_{P+1} - \mathrm{ToolPoint}_{P} \right\| \tag{5.7}$$

The material removal rate could be calculated based on the theoretical rate in equation (5.6) but this not as accurate as the actual material removal rate. The difference between the theoretical MRR and actual MRR can occur at the part borders or in regions where the cutter does not engage the full depth of cut, or full radial depth of cut, or in areas cut by a previous tool pass.

The actual material removal rate can be accurately estimated directly from the voxel model by summing the voxel values before and after a cutting pass. The voxel value represents the portion of the voxel that is “full” of material and ranges from 0-255. The amount of material in a voxel is given by the following Equation (5.8).

$$V_M = \frac{VoxelValue}{255} \cdot VoxelSize^3 \tag{5.8}$$

The material removed by a tool pass is the difference in the volume of material before the tool pass minus the volume of the material after the pass. The material removal rate is this amount of material divided by the time calculated by Equation (5.5) with *l* replaced by *d*.

$$MR = \sum V_{M_{Before}} - \sum V_{M_{After}} \tag{5.9}$$

$$MRR = \frac{MR}{d/v_f} \tag{5.10}$$

The material removed (MR) was calculated for an example case using the test part shown in Figure 5-4 and Figure 5-5. The blank is bar stock with the same outer dimensions of the part. The dimensions of the part are shown in Figure 5-6 in millimeters. A flat end cutter was selected for this test to show how the MR can be calculated for a roughing pass. The cutter diameter was 6.35 mm and the distance between path interval was 57.5% of the radius or 3.65mm.

The volume of material that should be removed from the bar stock based on the dimensions is 22.5 cm^3. To evaluate the accuracy of the MR calculation based Equation (5.9) the part was voxelized with a voxel size of 0.2mm. A blank model was created based on the dimensions of the part model and has a padding of 4 voxels between the blank and part for rendering purposes. The amount of material removed by evaluating the voxel model according to Equation (5.9) is 25.2cm^3. However, when the padding of blank material is accounted for by subtracting the padded volume, the MR is 22.0 cm^3 indicating that this approach can be used to accurately measure the material removed. The actual material removed is smaller because of the material left by the cutter that was selected as seen in Figure 5-5.

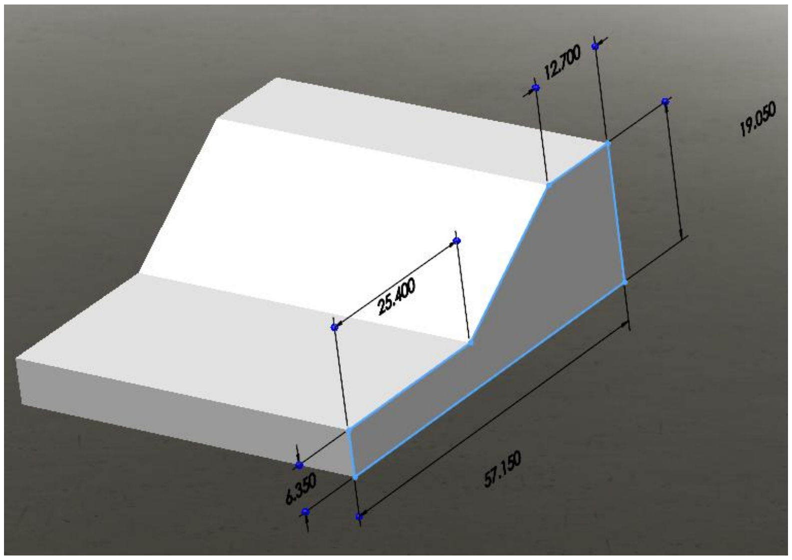


Figure 5-4: Simple test part for material removal (dimensions in mm)

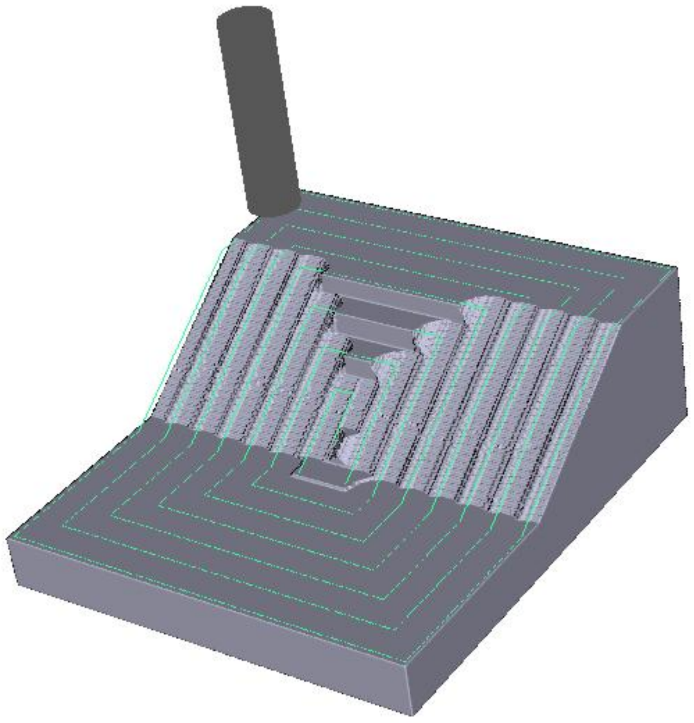


Figure 5-5: Flat end cutter and material removed

Surface Metrics

The material removal rate and machining time provide metrics of a tool path's cutting efficiency. However, they give no indication of the surface quality left by the path on the part or the amount of remaining material. When a non-flat end generalized cutter is used, and there is an arbitrary distance between paths, the cutter leaves a scallop. The scallop for the simplest case of a planar surface is shown in Figure 5-6.

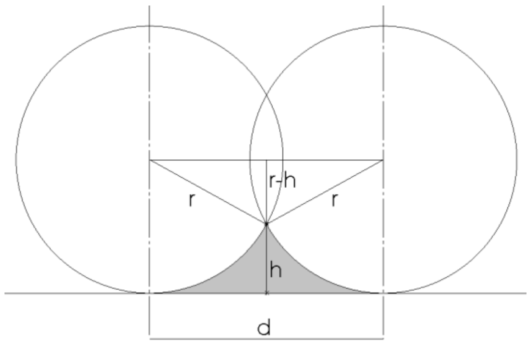


Figure 5-6: Scallop for planar surface

$$(r-h)^2 = d^2 - \left(\frac{d}{2}\right)^2 \tag{5.11}$$

Where r is the cutter radius, d is the step distance between successive passes in a path, and h is a scallop height. As a result, the scallop height can be calculates as:

$$h = r - \sqrt{r^2 - \left(\frac{d}{2}\right)^2} \tag{5.12}$$

This height will be smaller for convex regions of the surface and larger for concave regions of the surface.

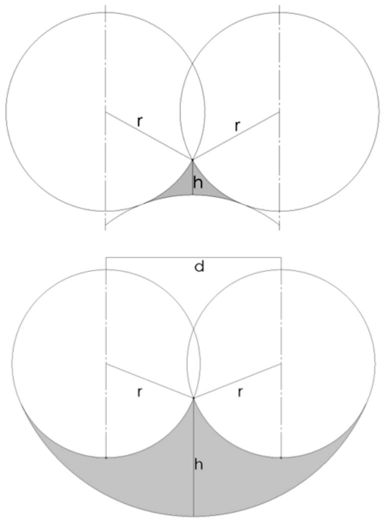


Figure 5-7 – Concave or Convex regions

For an inclined flat plane (Figure 5-8), a scallop height can be even larger than in a convex region and is calculated by:

$$h = CE \cdot \cos\left(\frac{\pi}{2} - \alpha\right) \tag{5.13}$$

$$CE = FH - FC + HE \tag{5.14}$$

$$FH = \frac{d}{\tan(\alpha)}; FC = \sqrt{r^2 - (d-r)^2}; HE = r \cdot \tan\left(\frac{\beta}{2}\right) \tag{5.15}$$

$$h = \left[\frac{d}{\tan(\alpha)} - \sqrt{r^2 - (d-r)^2} + r \cdot \tan\left(\frac{\beta}{2}\right)\right] \cdot \cos\left(\frac{\pi}{2} - \alpha\right) \tag{5.16}$$

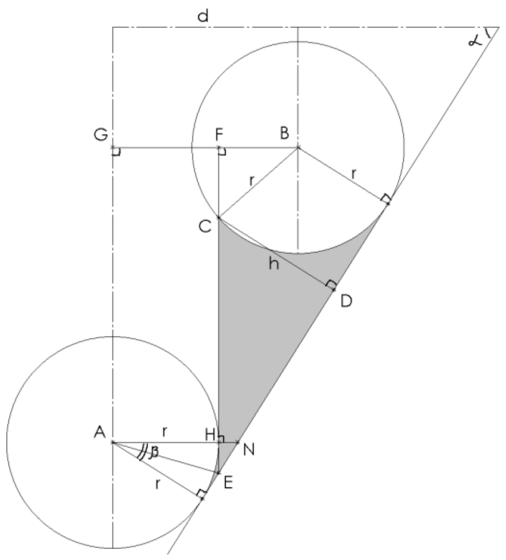


Figure 5-8 - Inclined surface machining

However, the scallop height for any non-planar surface will depend on the surface curvature and the height difference between points.

The scallop height is estimated using the ray-casting approach developed in Chapter 3. To estimate the scallop (or the maximum amount of remaining material) after a pass, rays are cast from the machining plane into the part and blank at the resolution of the voxel model. The bisection routine is used as the volume is queried along the ray to find the top surface of the part and blank. Then the results are stored in a grid and the maximum difference between the blank and part is calculated based on the difference between the two surface values.

This is not the exact value of the voxel but provides an excellent estimate. For example, if the cutter is a 3mm ball end mill, and the voxel size is 0.5mm, then there is a width and depth of 12 voxels underneath the cutter. If the overlap between passes is 25%

of the cutter radius then there are approximately 3 voxels under the cutter in the scallop area. Smaller voxels sizes yield better estimates for the scallop and reducing the voxel size from 0.5mm to 0.25mm results in a width of 6 voxels in the scallop area.

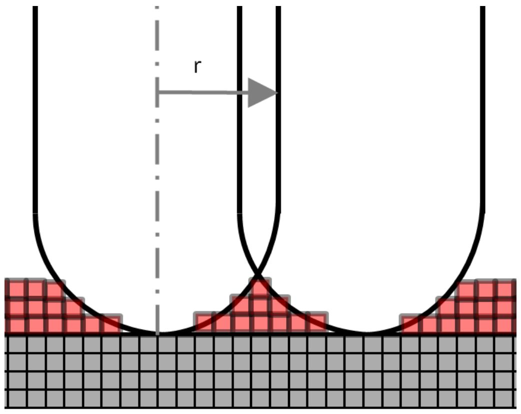


Figure 5-9: Scallop height approximated by voxels

The voxel value ranges from 0-255. Therefore each height value in a voxel can capture the height of a surface to a height resolution of (voxel resolution * ramp)/256. In the case of a voxel size of 0.5mm and a ramp of 3 voxels, the scallop height can be calculated with an approximately accuracy of 0.006mm.

The part shown in Figure 5-6 is used to illustrate the ability to calculate scallop with this framework. The tab in the figure is used to make sure there is ample blank height to generate scallop. The example part was machined using 6.35mm ball end mill with a 65% overlap between paths (4.13mm). The scallops can be clearly seen in Figure 5-10. The theoretical scallop based on Equation (5.12) is 0.762mm. For the path shown in Figure 5-10, the scallop will be between 0.76mm and 1.9mm because the distance between the path intervals is lager at the corners. The scallop was calculated as 1.1mm by taking maximum difference in surface location from the ray casting results. This indicates that the scallop can be accurately measured using this approach.

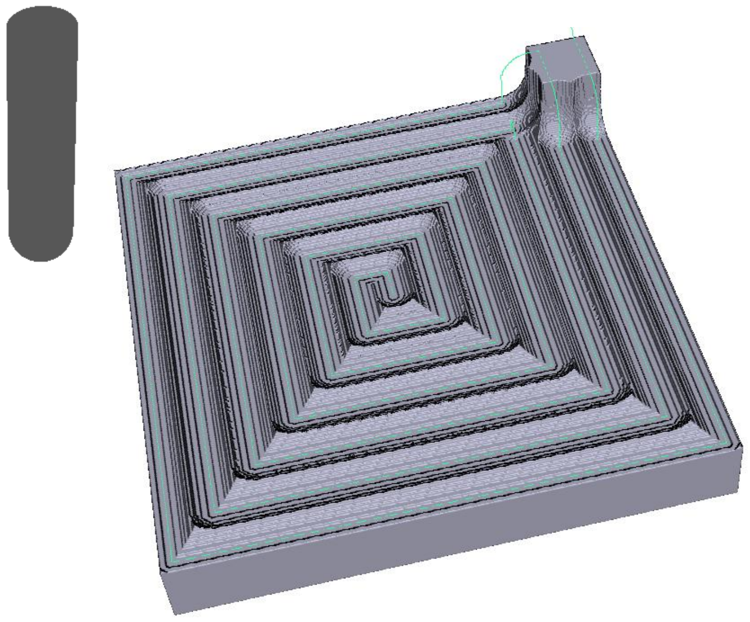


Figure 5-10: Scallop left by ball end mill

The material removal rate can be calculated by Equation (5.10) based on the path distance and feed rate. The path distance in Figure 5-9 is 693mm and was calculated by summing the point differences by using Equation (5.7). The feed rate is based on the feed per tooth, number of cutter teeth, and spindle speed. The feed per tooth depends on the material and the selected cutter. The spindle speed is based on the cutter diameter and recommended surface speed based on the cutter. Therefore given a material (recommended feeds and speed) and a generalized cutter geometry the framework above can automatically evaluate the quality of the path in terms of machining time, material removal rate, and surface quality (scallop).

Travelocity for Machining

The path metrics of machining time, material removal rate, and scallop are used to evaluate a specific path. Every manufacturer wants the best machining time, highest material removal rate, and best surface quality. But there is a tradeoff which makes the selection of machining parameters non-trivial. Figure 5-11 show what these tradeoffs may look like for a given process.

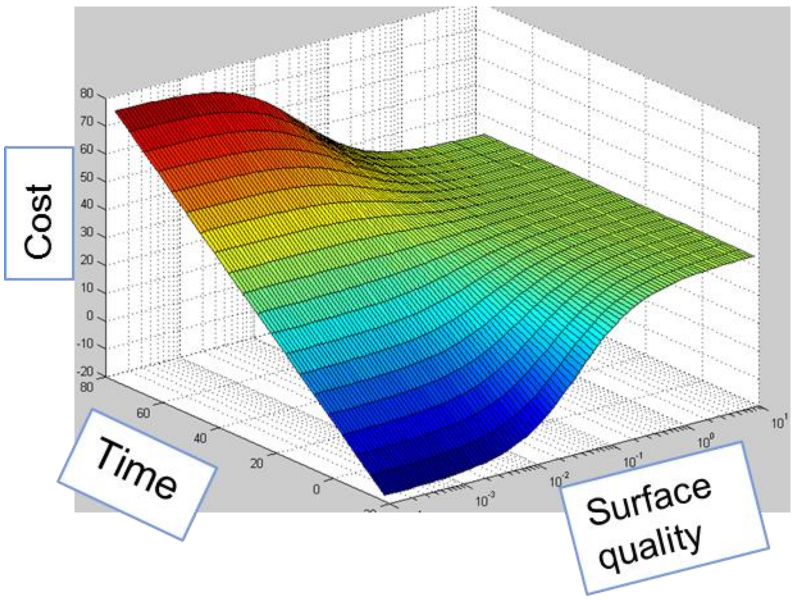


Figure 5-11: Machining tradeoff

Unfortunately, it is difficult to capture the tradeoffs before selecting parameters for machining. In practice, if time permits, tool paths are constantly improved by slowing changing the inputs to the path planning process. Figure 5-12 shows this process.

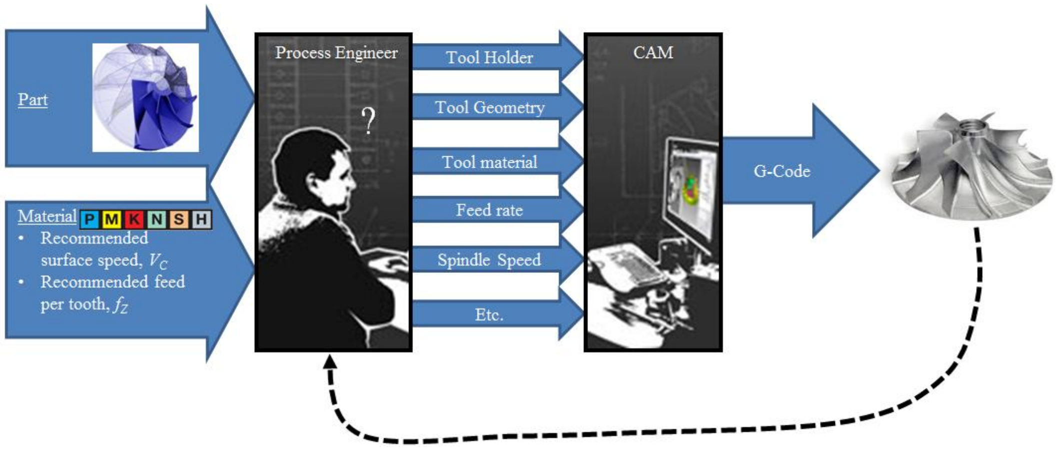


Figure 5-12: Traditional path planning process

However, the feedback from the resulting part does not always make it back into the tool path design process. This can be the result of limited resources and demanding manufacturing schedules but is more often the result of paths that are good enough but not necessarily optimal.

Choosing alternative tool geometry, slightly larger depth of cut, smaller radial immersion, a larger feed rate, and etc. can move a specific point on the surface shown in Figure 5-11 to other points on the surface. How these choices are made is determined by where on the surface the process engineer would like to operate. The process engineer may have more time and want to maximize the life of the tool and choose to operate based on cost alone. A 2D cross section of this preference might look like what is shown in Figure 5-13.

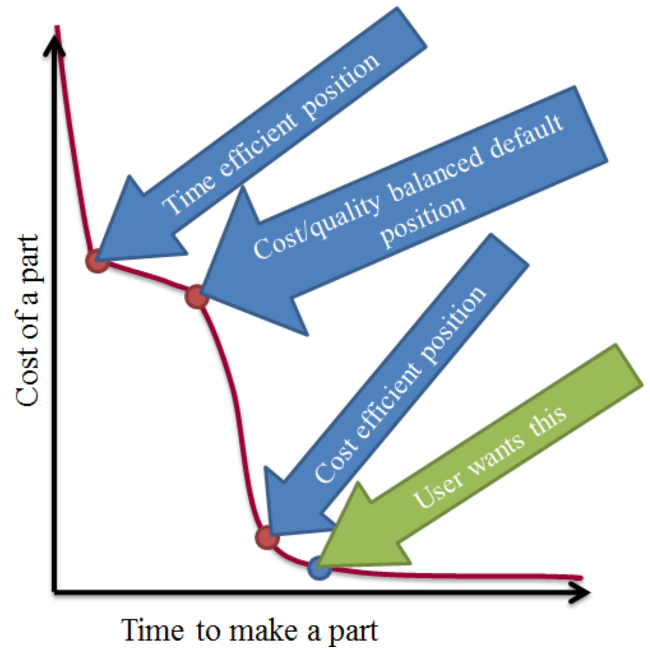
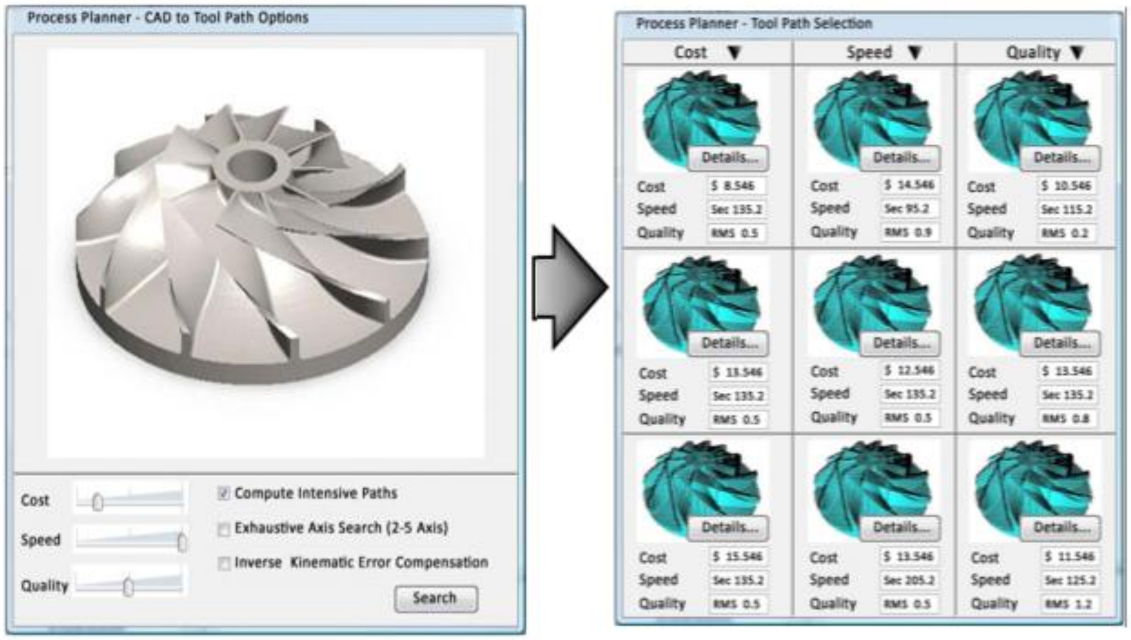


Figure 5-13: Time-cost tradeoff

Path planning is similar to planning a trip. What the process engineer needs is a tool similar to Travelocity for machining. Travelocity plans a flight (or path) based on user preference in terms of the travel time (number of stops, layovers), trip quality (business class, number of stops, layovers, preferred departure/arrival time), and ticket cost. A tool such as Travelocity for machining requires that tool paths are accurately generated and their performance analyzed in real time. This tool would be capable of generating thousands of tool paths complete with simulated results and metrics of cost, speed, and quality. The process engineer could then filter paths based on preference as shown in the mockup in Figure 5-14.


Process Planner - CAD to Tool Path Options
Search
Process Planner - Tool Path Selection
Details...

Cost
Speed
Quality

[✓] Compute Intensive Paths
[ ] Exhaustive Axis Search (2-5 Axis)
[ ] Inverse Kinematic Error Compensation

Cost ▼	Speed ▼	Quality ▼
Cost $ 8.546 Speed Sec 135.2 Quality RMS 0.5	Cost $ 14.546 Speed Sec 95.2 Quality RMS 0.9	Cost $ 10.546 Speed Sec 115.2 Quality RMS 0.2
Cost $ 13.546 Speed Sec 135.2 Quality RMS 0.5	Cost $ 12.546 Speed Sec 135.2 Quality RMS 0.5	Cost $ 13.546 Speed Sec 135.2 Quality RMS 0.8
Cost $ 15.546 Speed Sec 135.2 Quality RMS 0.5	Cost $ 13.546 Speed Sec 205.2 Quality RMS 0.5	Cost $ 11.546 Speed Sec 125.2 Quality RMS 1.2

Figure 5-14: Travelocity for machining

Although a complete tool such as this is beyond the scope of this research, the GPU accelerated approach presented in Chapter 3, the GPU timing results from Chapter 4, and the path metrics presented in this chapter provide a framework for this tool to be completed. This framework automatically generates tool paths based on an input part file, material, and cutting geometry and calculates the material removal rate (efficiency), remaining material (quality), and machining time.

In addition, the algorithms developed in this research are parallel algorithms created on the GPU. Chapter 4 showed that with this framework that a tool path can be generated for a given part with a voxel resolution of 0.5mm in less than 5 seconds. However, this is far from the limit of actual parallel computing ability. Moreover, the introduction of cloud computing changes the computing landscape for this research dramatically. It is possible to upload a single part file to any number of cloud GPUs with difference cutter parameters and immediately get the path metrics described in this chapter.

To demonstrate how the methodology introduced in the preceding chapters can be used in a Travelocity approach, a number of different tool passes were performed on the part shown in Figure 5-15. It is clear from the part geometry that the selection of the best cutter is not obvious and that the traditional path metrics would not be accurate due to the surface differences. The material selected for this example was polyurethane tooling board material commonly used to prototype parts. The recommended spindle speed is 2500 RPM for a radius of 12.7mm-25.4mm cutters with depths of cut up to 50.8mm for roughing. The cutters and their corresponding values are shown in Table 5.3.

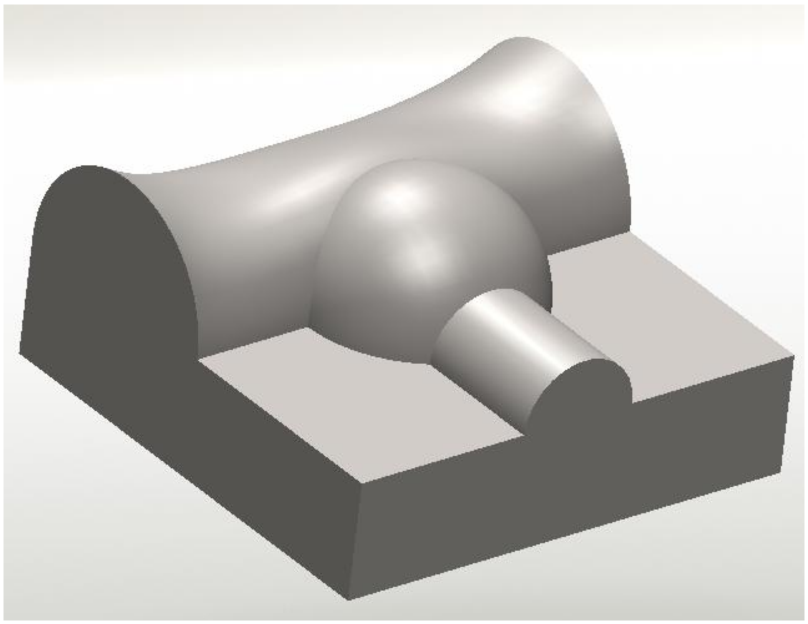


Figure 5-15: Travelocity example part

Table 5.3: Cutting performance

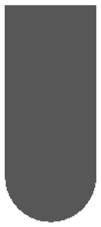
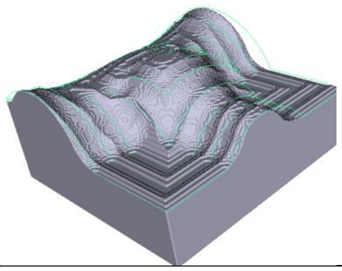
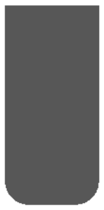
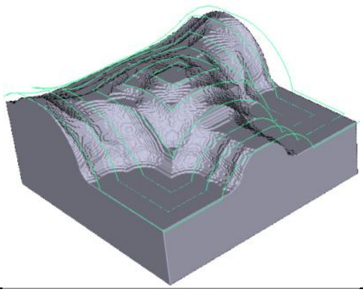
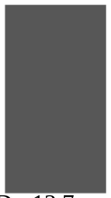
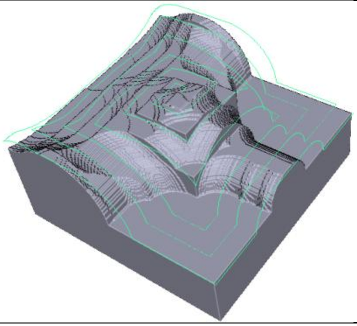
Cutter	Path Length (mm)	Machining Time (s)	Material Removal Rate MRR (cm^3/min)	Max Scallop (mm)	Output Path
 D = 12.7mm	80.26	9.48	137.02	5.98	
 D =12.7mm D_R =3.175mm	80.35	9.49	136.99	7.80	
 D =12.7mm	81.31	9.60	133.74	9.36	

Table 5.3 shows the results of the metrics when run using the framework presented in research. This example demonstrates how this tool can potentially enable process engineers to see tradeoffs in real time. Although an experienced process engineer would likely not need such a tool for the example part it would not be clear to someone less experienced which path is best. This highlights the fact that although tool paths can be created that perform as desired they are not always optimal. For example, if the flat end cutter in the example shown in Table 5.3 is selected based on the fact that a roughing

option is being performed there would be a loss in performance in time, material removed, and scallop. It is not hard to see that minor tweaks to the path such as changing the path interval, slightly changing the radius, and etc. will have small effects on the performance of the path. The luxury of working with a path until it is “perfect” is something that the process engineer is not afforded due to demanding schedules and limited resources. By taking advantage of this parallel framework the process engineer could work within preference instead of repeated sub-optimal calculations.

This chapter highlighted the tremendous amount of information available to the process engineer and how the decisions made upstream affect the material removal rate, machining time, and part surface quality. This chapter also presented how the traditional path metrics are calculated and how the same metrics can be automatically calculated based on the framework presented in Chapter 3. The fact that there is too much information for the process engineer to create optimal paths was highlighted and the need for a tool like Travelocity to support better productivity was established.

CHAPTER SIX

EXPERIMENTAL RESULTS

Chapter 3 introduced a GPU accelerated tool path planning methodology based on ray-casting into voxel models. This chapter presents experimental results of the methodology when used to machine the three part models shown in Figure 6-1. The parts machined are: a compound surface made of multiple parametric surface patches, a free-form surface, and a model file of a BMW 3 series. These parts are representative of a family of parts and highlight the methodology's performance and limitations.

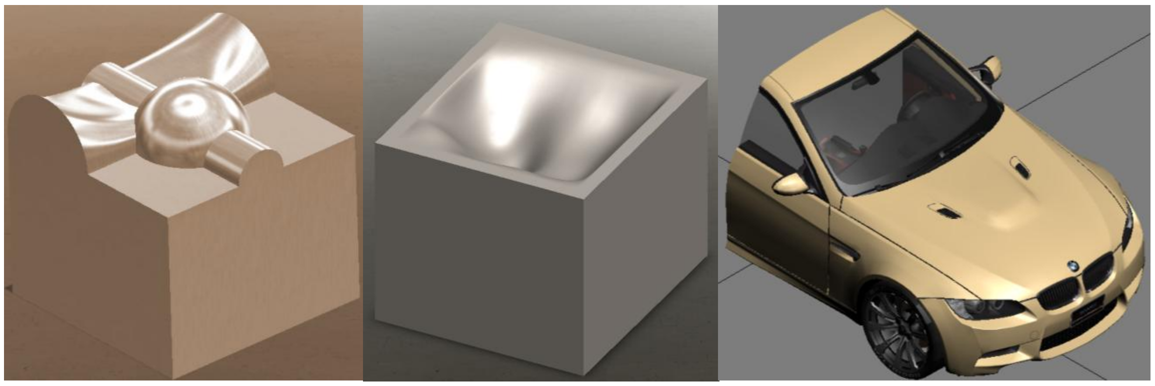


Figure 6-1: Test parts: compound surface, free-form surface, complex surfaces

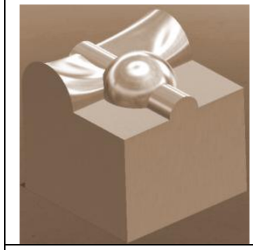
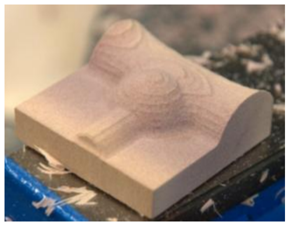
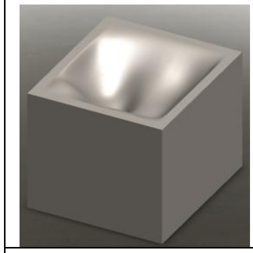
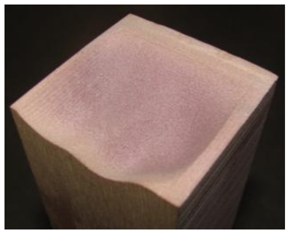
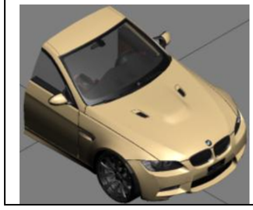
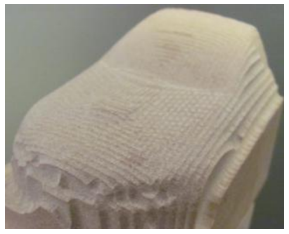
The tool paths generated by the methodology are automatically post-processed by translating the Euclidian path points into G-code that can be understood by machine tools.

Equipment

An Okuma MU500VA 5-axis vertical machining center was used to machine the parts shown in Figure 6-1Figure 6-1. Each of the parts were roughed with a larger cutter and finished with a smaller cutter from the G-code generated from the methodology presented in Chapter 3. The material used to machine the parts is made of dense polyurethane foam called tooling board. It takes very little force to cut and yet retains surface accuracy. It is commonly used as a prototype material. The cutter sizes and path information are shown in Table 6.1.

Figure 6-2: Okuma MU-500VA 5-axis machine tool

Table 6.1: Cutter sizes for experiments

Part	Roughing Cutter	Finishing Cutter	Voxel Size / Path Interval	Result
Compound Surface 	Ball End Mill: 6.35mm diameter	Ball End Mill: 3.175mm diameter	0.53mm / 0.254mm	
Freeform Surface 	Flat End Mill: 8mm diameter	Ball End Mill: 3.175mm diameter	0.26 mm / 0.0127mm	
Complex Surfaces 	Flat End Mill: 8mm diameter	Ball End Mill: 3.175mm diameter	0.26 mm / 0.0127mm	

Compound surface part

The first part shown in the first row of Table 6.1 represents a compound surface and is a more difficult part to generate tool path from using one of the traditional approaches discussed in Chapter 2. This is because it is comprised of multiple parametric surfaces that are joined together making it necessary for surface repair when offset surfaces are created. The voxelization of the part eliminates this difficulty by discretizing the part. However, small enough voxel sizes must be used to maintain surface accuracy. A larger voxel size of 0.53mm was used for this part the ridges that look like horizontal slices in the part surface indicate that the resolution was too small. However, the is no gouge in the part surface indicating that the path update algorithm correctly avoids gouging the surface with the cutter. Figure 6-3 shows an earlier version of the part without the path update algorithm on the left and with the algorithm on the right. The left image indicates that the cutter seriously gouges the part. To increase the surface finish of the part smaller voxel sizes would be necessary.

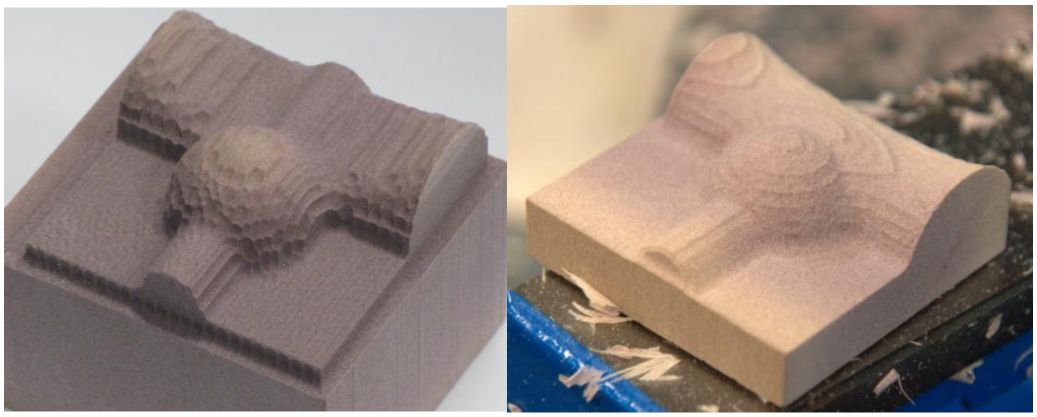


Figure 6-3: Compound surface part with and without gouge

Freeform surface part

The freeform surface part in the second row of Table 6.1 represents a part that might be used in the die and mold industry. It was created with a voxel size of 0.26mm. The result of the roughing pass is shown in A slice of the part model and machined part are shown in Figure 6-5. Other views of the part and a close up of the top view are shown in Figure 6-6 and Figure 6-7. The freeform surface part is free form gouge but has scallop height due to the path interval and steep side slope. There is no indication of voxelization in Figure 6-5 or Figure 6-6. However, some very small artifacts are shown Figure 6-7 that are similar to the planar slices shown in the compound surface part. These artifacts are likely due to the limit of the voxel accuracy. Smaller voxel sizes can be used by extending this work to take advantage of multiple GPUs or utilizing GPUs with larger amounts of RAM.

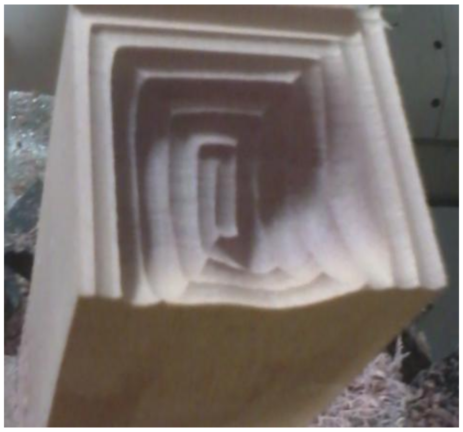


Figure 6-4: Freeform surface roughing pass

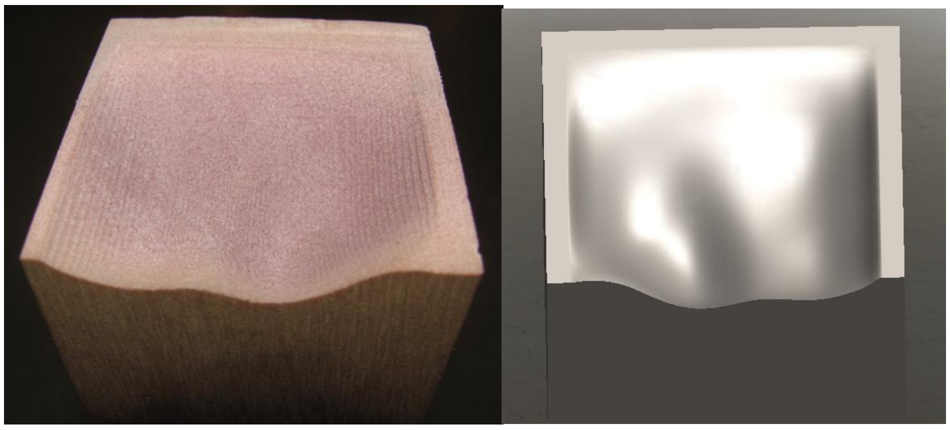


Figure 6-5: Freeform surface part

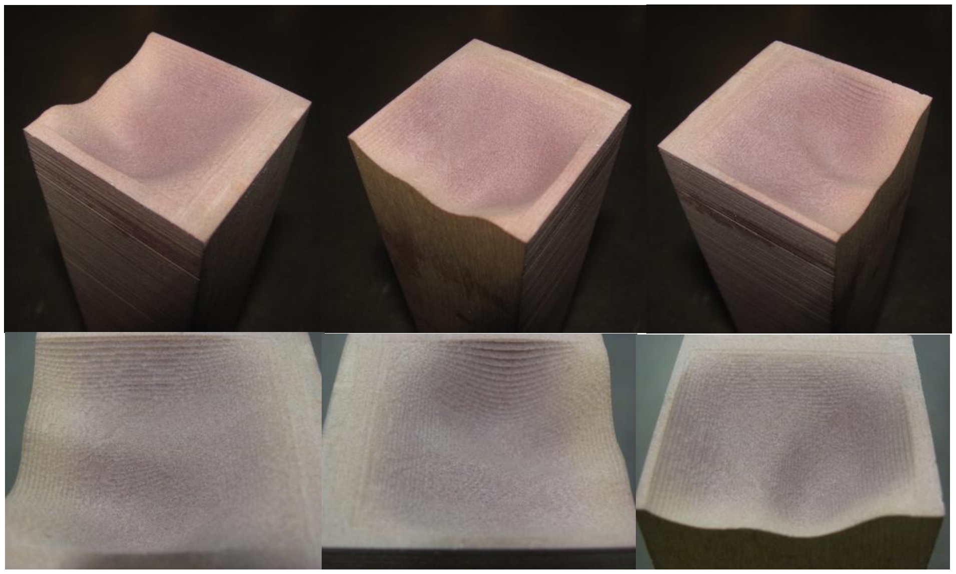


Figure 6-6: Freeform surface part different views

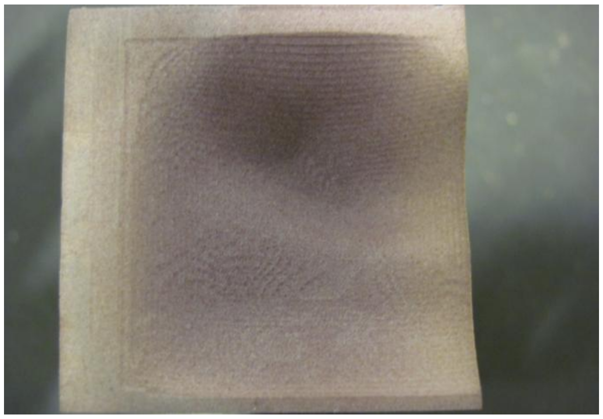


Figure 6-7: Freeform surface top view

Part with Complex Surfaces

The part shown in the third column is a BMW M3 model that is comprised of numerous surfaces. Such a model is typically used for design. In fact, the model was likely created by a game designer. Many of the model surfaces are not connected and the traditional tool path planning methods would likely fail to create accurate tool path due to requirements of “water tight” models. This model was converted to a STL file and voxelized. Because the voxelization algorithm presented in Chapter 3 utilizes the direction of the polygon faces it is able to successfully voxelize the part. The BMW part was machined both in 3-axis mode and in 2+3 mode with the A and C axis rotated. The three axis roughing and finishing operations are shown in Figure 6-8 and Figure 6-9.The results of the ray casting algorithm are shown in Figure 6-10 and show the part, two roughing passes, and the finishing pass. The part was machined by two roughing passes and three finishing passes.

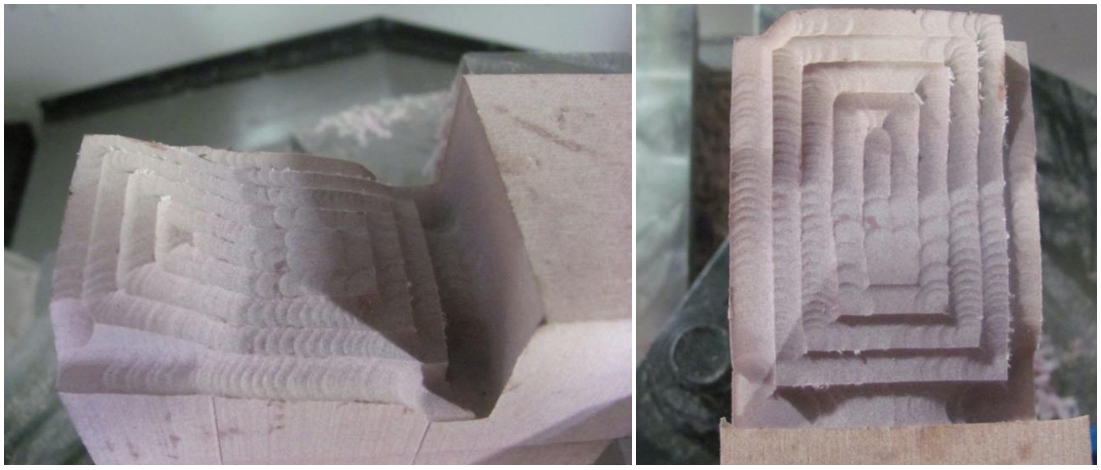


Figure 6-8: Complex surface model roughing

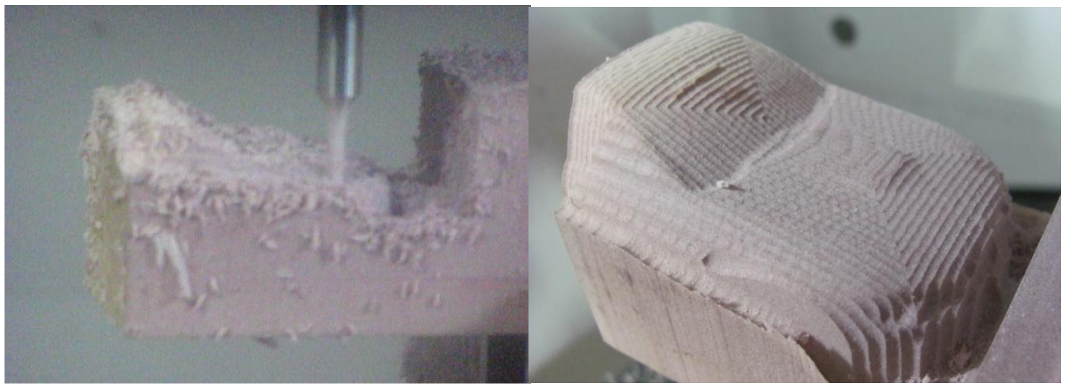


Figure 6-9: Complex surface finishing

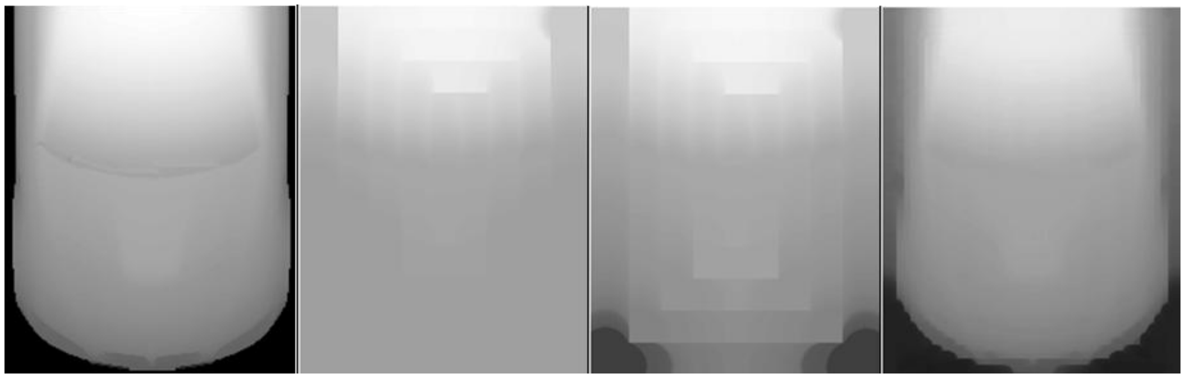


Figure 6-10: Results of ray-casting: part, first roughing pass, second roughing pass, finish pass

The BMW was rotated 45 degrees to demonstrate the 2+3 machining ability of this research. The part was reoriented in the scene as shown in Figure 6-11. The part as seen from the machining plane rotated by 45 degrees is shown

Figure 6-12. The rotated part in the actual machine tool is shown in Figure 6-13.

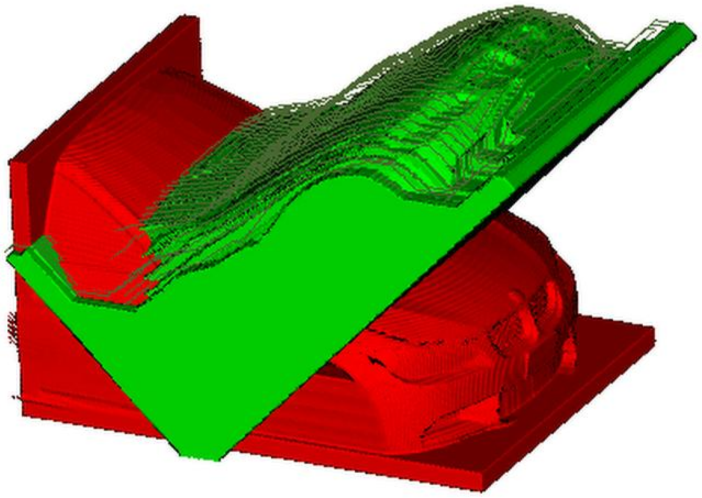


Figure 6-11: Part reorientation for 2+3 machining

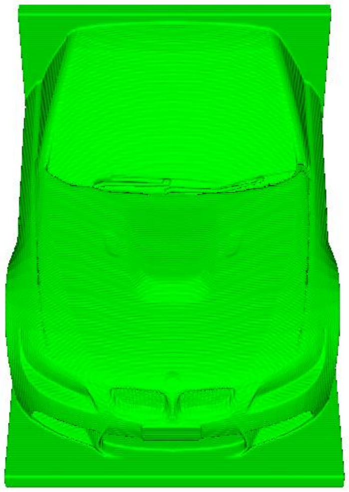


Figure 6-12: Part as seen from rotated machining plane

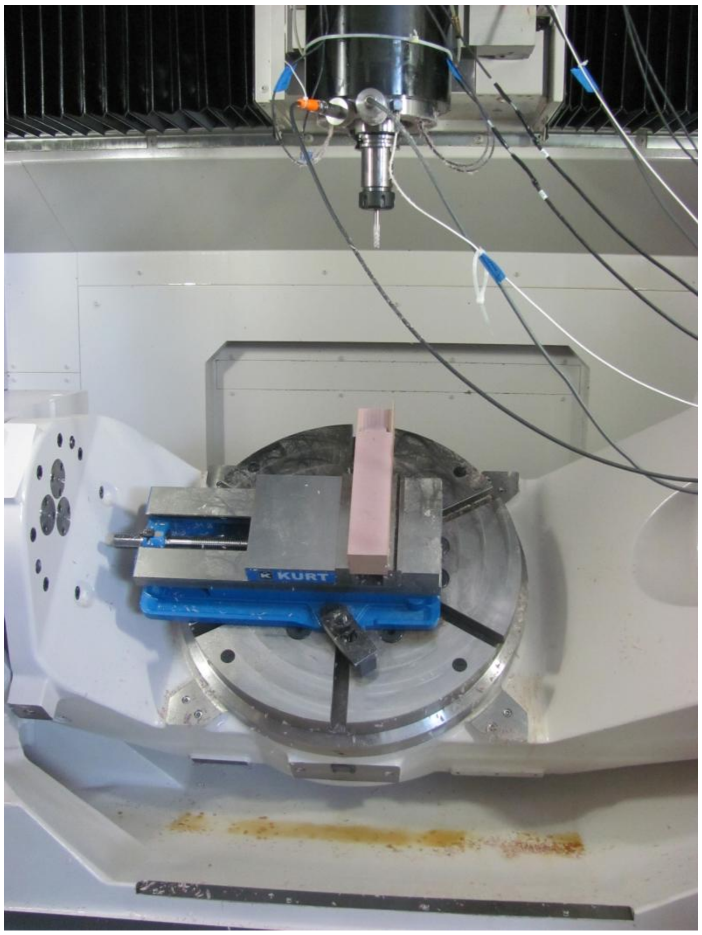


Figure 6-13: 5-axis machine with re-oriented part

Rays are cast at the re-oriented plane and roughing and finishing passes are generated based on the new orientation. The output of the ray casting algorithm is shown in Figure 6-14 for the initial part, after two roughing passes and after a finishing pass. The actual roughing and finishing passes are shown in Figure 6-15 and Figure 6-16. Figure 6-17 shows the final result.

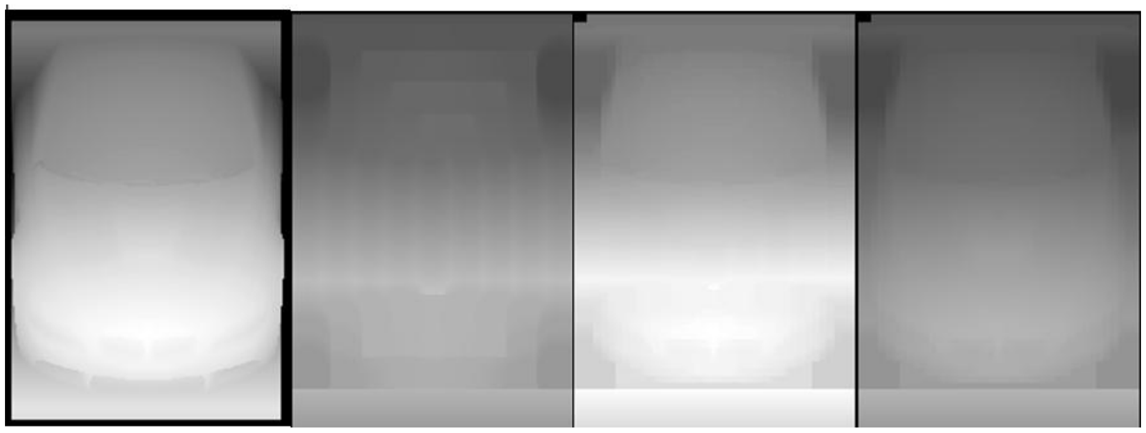


Figure 6-14: Results of ray-casting on rotated part

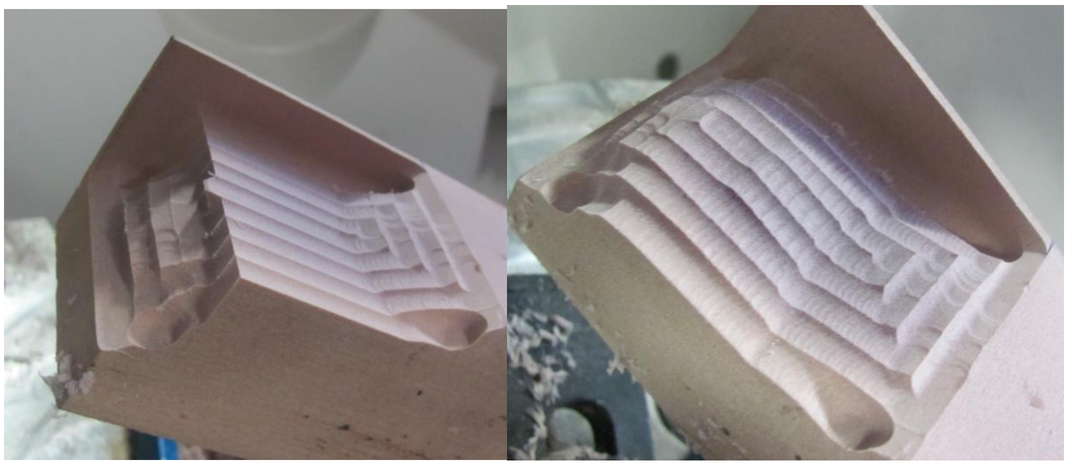


Figure 6-15: First and second 2+3 roughing passes

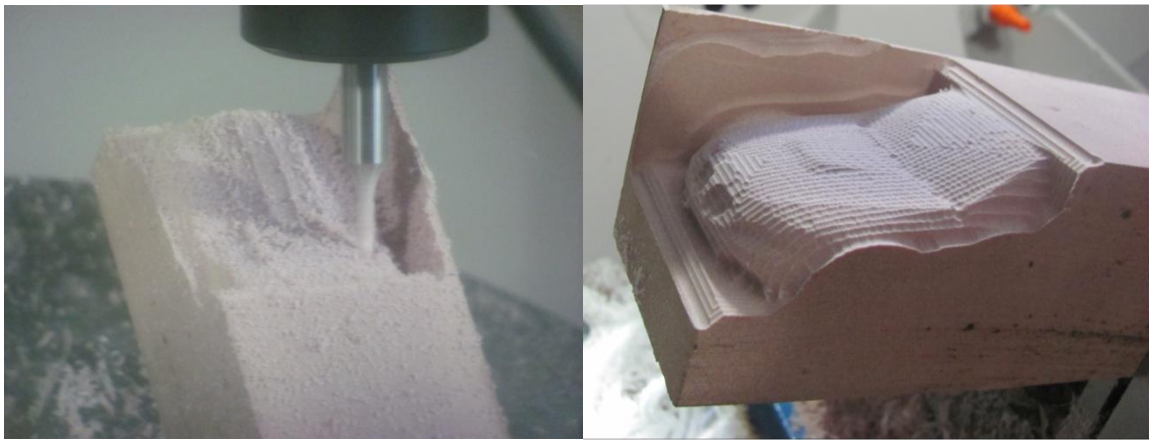


Figure 6-16: 2+3 Finishing pass

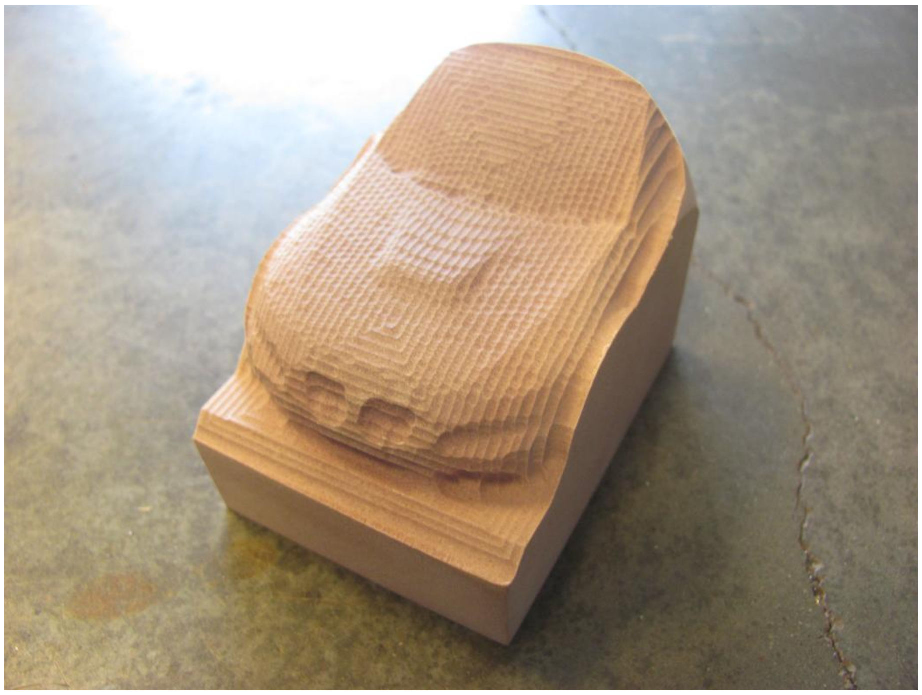


Figure 6-17: 2+3 Final part

The same cutters were used for both the 3-axis and 2+3-axis BMW models and both showed nearly identical surface quality. Both suffered from noticeable scallop due to the 65% step over distance which is the largest near the doors. The underlying surface is clearly seen through the scallop and by decreasing the step-over distance the scallop can be significantly reduced.

Summary

This chapter demonstrated the ability of this framework to automatically generate tool paths for compound surfaces, freeform surfaces, and multiple complex surfaces in both 3-axis and 2+3-axis configurations that can be used on a machine tool.

CHAPTER SEVEN

CONCLUSIONS AND RECOMMENDATIONS

Currently, process engineers create tool paths that produce sub-optimal results. Determining the tradeoffs of early decisions while creating the tool path is nearly impossible. This dissertation presents a methodology that performs automated digital machining and analysis on a part. This methodology contributes the following:

- A parallel framework to machine digital parts represented by voxel models.
- A digital surface detection approach by ray casting.
- Path planning based on generalized cutting geometry and digital voxel models.
- Gouge avoidance for digital parts and generalized cutting geometry.
- Digital model metrics of material removal rate, machining time, and scallop.

Representing the part models as digital voxels eliminates the complexity of offsetting and merging multiple surfaces encountered in traditional machining and enables native simulation and visualization capabilities. However, accurately capturing the surfaces is vital in generating tool paths that are precise and gouge free. Parallel ray casting is used to accurately capture the digital part surfaces. The speed and accuracy of the ray casting approach is enhanced by using a bisection routine for each ray. This research utilizes parallel processing so that all rays are cast simultaneously resulting in dramatic time savings compared to serial benchmarks from the CPU (Chapter 4).

In this research, rays are cast from an arbitrary plane in Cartesian space allowing the determination of the best machining plane by analyzing which plane results in the most material removed. Although this approach ignores valid reasons for machining from other planes, it demonstrates how this approach can be used and suggests ways to search for a better machining plane than a primary axis as in 3-axis machining.

The ray casting algorithm results in a 2D surface of height points. These points are used to determine the location of the blank and part voxel volumes. Once the surfaces are known, a template bath is generated based on cutter geometry. A generalized cutter geometry is utilized as part of this research and the template path is updated based on the generalized cutting geometry. The template path is updated by parallel algorithms that process the tool points in parallel on the GPU as opposed to serially on the CPU. Chapter 4 demonstrates that this parallel approach is 375Xs faster than the CPU.

Once the paths are generated and updated based on the generalized cutter, the material of the blank model is removed in parallel according to the updated tool path and cutting geometry. The updated tool path accounts for all the voxels affected by the cutter as it moves through the voxel blank volume. This approach results in an accurate tool path and simulation of a single tool pass. The process of casting rays and generating tool paths is repeated to create subsequent roughing and finishing passes. The update of the model is also processed in parallel, resulting in a speed up of 600Xs when compared to the CPU (Chapter 4).

The metrics of the generated paths are calculated based on the voxel volume and machining parameters (Chapter 5). The volume of material removed by the cutter as it traverses the path is calculated to provide an accurate measure of the material removed during that pass. In addition, the machining time is calculated based on the distance of the updated path and the feed rate. The material removal rate for a given pass is calculated based on the machining time and volume of material removed. The ray casting approach enables calculation of the distance between the blank and the part. This fact is exploited after a tool path is generated to estimate the scallop or remaining material left by a specific tool pass. This information can be used to determine the appropriate cutter for subsequent tool passes. This methodology can be used to evaluate multiple paths to support an optimization approach to machining by enabling process engineers to pick paths based on preferences (Chapter 5).

The overall performance of this parallel methodology showed in excess of 500Xs the increase in speed when implemented on a GPU vs a CPU (Chapter 4). The example parts in this research were small parts due to the GPU memory limitations. However, with today's hardware and software technology, multiple GPUs can be used with the parallel approach presented in this research. There are commercially available turn-key solutions and programming technologies that suggest that this framework can be run in real-time simply by extending the already parallel code to multiple GPUs.

The tool paths produced by this methodology were post processed into machine tool G-Code. Example parts were machined using both 3-axis and 5-axis machine tools. Because the ray-casting approach is planar, the capabilities of this approach are limited to

2+3 machining. Experimental results of different part geometries demonstrated the versatility of this methodology when applied to compound surfaces, free-form surfaces, and complex surfaces.

The following implications are a direct result of this research:

- Hundreds of tool paths can be generated, analyzed, and ranked by parallel processing based on generalized cutting geometry and template paths.

- Process engineers can use this framework to create parts based on preference of cost speed and quality and get immediate feedback from tool selection and path strategy.

- This framework can be used for multiple parallel processors and is not limited by current hardware.

This research is merely the beginning of what is possible with digital machining. The following fundamental research issues remain unsolved:

- Identify the best strategy to combine multiple local hardware devices and create code abstractions to manage memories and work load.

- Identify best strategies and input-output protocols to utilize cloud computing resources.

- Extend the path template framework to consider part surface information to generate non-uniform and iso-scallop paths.

- Identify part features and automatically create tool paths to machine them (e.g. flank milling sidewalls, pocketing, drilling, and chamfering).

- Organize tool libraries into a database and create parametric cutter abstractions that allow for rapid integration of a wide range of cutting tools.

- Design an interface to search material databases and create methods to relate material information to cutting parameters.

- Create search algorithms and optimization schemes that utilize this methodology to rapidly and automatically evaluate cutter performance and iterate towards a set of ideal combinations of cutting and machining parameters based on tool and material databases.

This research can be summarized as digital machining and its current state closely resembles the state of the first digital camera introduced in 1975 and shown in Figure 7-1. This camera had a resolution of 0.01 Megapixel (102x102 pixels) and took 23 seconds to capture a single picture [53]. This research is the 3D equivalent to the digital camera for a single GPU. As stated in Chapter 4, the maximum resolution voxel model if both the blank and the part are stored on the same desktop GPU is about 256x256x256 voxels or 1.5 GB.

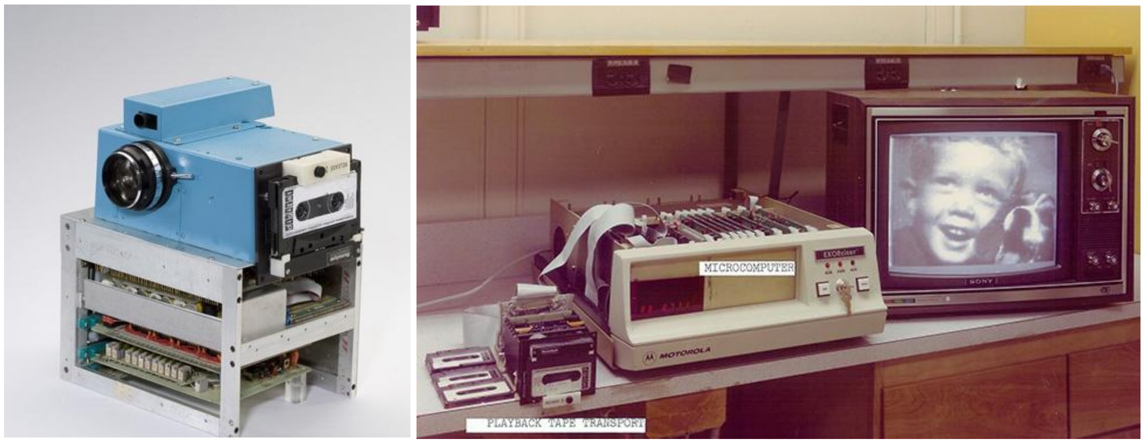


Figure 7-1: The first digital camera

Moore’s law [50], which predicts that the number of transistors that can be “crammed” onto a chip doubles every two years (later shortened to 18 mos.) predicted devices such as the digital camera a decade before it was created. Moore’s law is shown in Figure 7-2 and shows the number of transistors packed onto a single integrated circuit continues to follow this trend. Moore's law directly affects computing processing speed (shorter distances), memory (density), and the resolution of digital cameras (denser sensors).

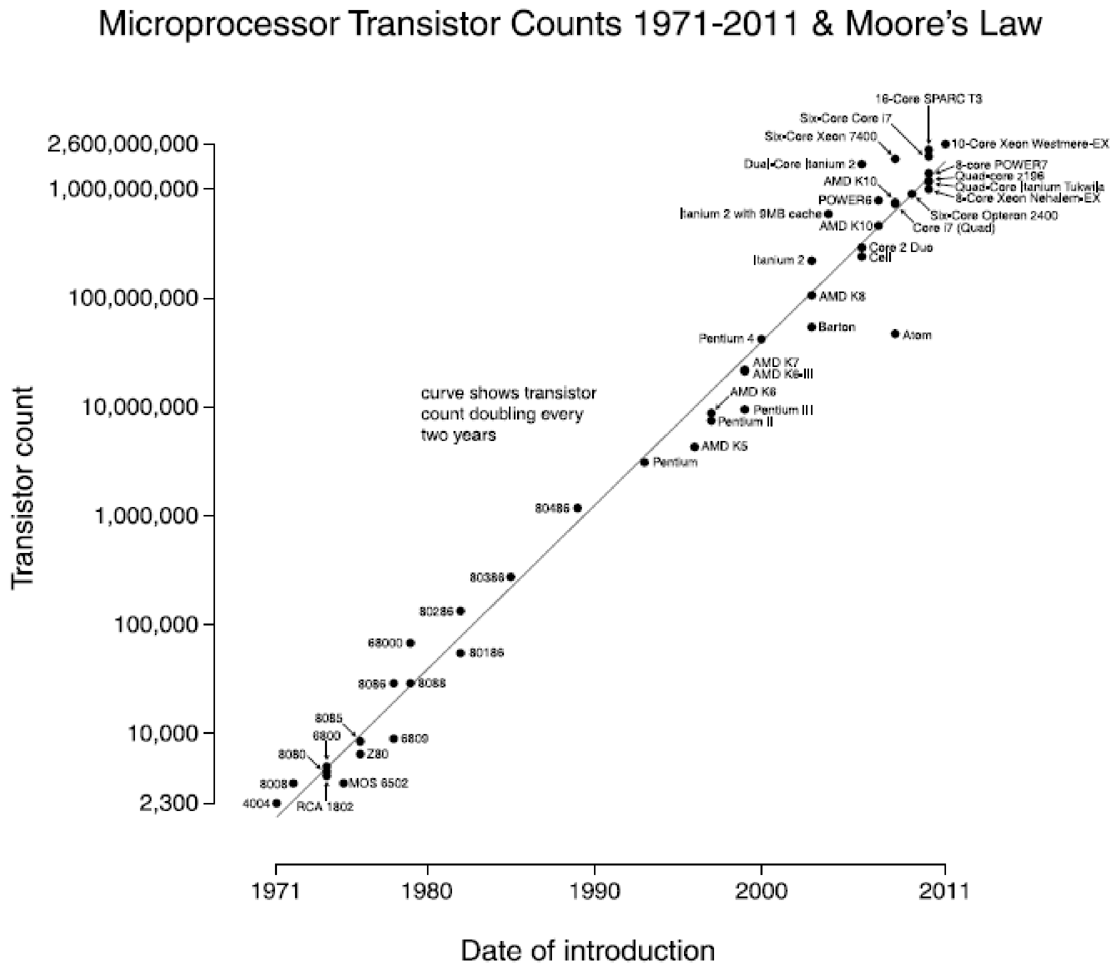


Figure 7-2: Moore's law [54]

The way that Moore's law has affected processor computing capabilities in terms of the number of GFLOPS (giga-floating point operations per second) for both CPUs and

GPUs is shown in Figure 7-3. The computing speed of the GPU increases much faster than that of the CPU because each of the GPUs parallel processors benefits from faster speeds. For the GPU, denser transistors means higher parallel processing power, with more local memory available for kernel launches, with higher precision.

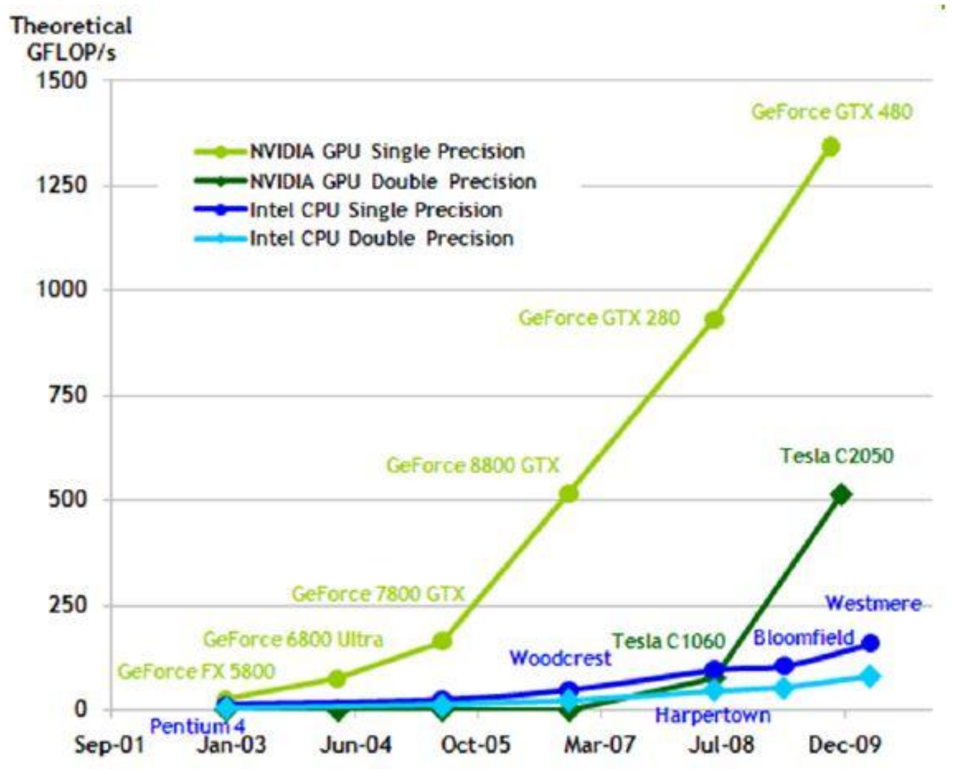


Figure 7-3: Computing speed vs. time for CPU and GPU

It took nearly 30 years, but the digital camera went from an oddity that consumers would not consider using to replace their 35mm analog cameras, to a device that consumers would expect and demand in nearly all cell phones with 500-1000x’s the resolution of the first camera. The capabilities for digital machining will follow a similar trend. However, due to the parallel nature of the methodology presented in this research, it will take much less time to realize high accuracy digital models for a single GPU. When considering multiple GPUs, this capability was available yesterday.

REFERENCES

1. Broomhead, P. and M. Edkins, *Generating NC data at the machine tool for the manufacture of free-form surfaces.* International Journal of Production Research, 1986. **24**(1): p. 1-14.
2. Elber, G. and E. Cohen, *Tool path generation for freeform surface models.* Proceedings on the second ACM symposium on Solid modeling and applications, 1993: p. 419-428.
3. Loney, G.C. and T.M. Ozsoy, *NC machining of free form surfaces.* Computer-Aided design, 1987. **19**(2): p. 85-90.
4. Bobrow, J.E., *NC machine tool path generation from CSG part representations.* Computer-aided design, 1985. **17**(2): p. 69-76.
5. Huang, Y. and J.H. Oliver, *Non-constant parameter NC tool path generation on sculptured surfaces.* The International Journal of Advanced Manufacturing Technology, 1994. **9**(5): p. 281-290.
6. Hwang, J.S., *Interference-free tool-path generation in the NC machining of parametric compound surfaces.* Computer-aided design, 1992. **24**(12): p. 667-676.
7. Kim, S.-J. and M.-Y. Yang, *A CL surface deformation approach for constant scallop height tool path generation from triangular mesh.* International Journal of Advanced Manufacturing Technology, 2006. **28**: p. 314-320.
8. Suresh, K. and D.C.H. Yang, *Constant scallop-height machining of free-form surfaces.* Journal of engineering for industry, 1994. **116**(2): p. 253-259.
9. Yang, D.C.H., et al., *Boundary-conformed toolpath generation for trimmed free-form surfaces via Coons reparametrization.* Journal of Materials Processing Tech., 2003. **138**(1-3): p. 138-144.
10. Jun, C.S., D.S. Kim, and S. Park, *A new curve-based approach to polyhedral machining.* Computer-Aided Design, 2002. **34**(5): p. 379-389.
11. Yuwen, S., et al., *Iso-parametric tool path generation from triangular meshes for free-form surface machining.* The International Journal of Advanced Manufacturing Technology, 2006. **28**(7): p. 721-726.
12. Park, S. and B.K. Choi, *Tool-path planning for directio-parallel area milling.* Computer-Aided Design, 2000. **32**(1): p. 17-25.
13. Kim, B.H. and B.K. Choi, *Guide surface based tool path generation in 3-axis milling: an extension of the guide plane method.* Computer-Aided design, 2000. **32**(3): p. 191-199.
14. Ding, S., et al., *Adaptive iso-planar tool path generation for machining of free-form surfaces.* Computer-Aided Design, 2003. **35**(2): p. 141-153.
15. Duncan, J.P. and S.G. Mair, *Sculptured Surfaces in Engineering and Medicine*1983: Cambridge University Press.
16. Choi, B.K. and C.S. Jun, *Ball-end cutter interference avoidance in NC machining of sculptured surfaces.* Computer-Aided Design, 1989. **21**(6): p. 371-378.

17. Choi, B.K., et al., *Compound surface modelling and machining.* Computer-Aided Design, 1988. **20**(3): p. 127-136.
18. Choi, B.K., D.H. Km, and R.B. Jerad, *C-Space approach to tool-path generation for die and mold machning* Computer-Aided Design, 1997. **29**(9): p. 657-669.
19. Takeuchi, Y., et al., *Development of a personal CAD/CAM system for mold manufacture based on solid modeling techniques.* Annals of the CIRP, 1989. **38**(1): p. 429-432.
20. Inui, M., *Fast Inverse offset computation using polygon rendering hardware.* Computer-Aided Design, 2003. **35**(2).
21. *www.wikipedia.com*.
22. nVIDIA. *GPU Acclerated Research*. 2009 9/25/2009]; Available from: http://www.nvidia.com/object/cuda_home.html#.
23. Galoppo, N., N.K. Govindaraju, and M.M. Henson, D. *LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware*. in *Proceedings of the ACM/IEEE Supercomputing Conf.* 2005.
24. Jung, J.H. and D.P. O'Leary, *Cholesky Decomposition and Linear Programming on GPU*, 2006, Scholarly Paper: Department of Computer Science, University of Maryland.
25. Rumpf, M. and R. Strzodka, *Level set segmentation in graphics hardware.* International Conference on Image Processing, 2001. **3**: p. 1103-1106.
26. Cohen, P. [cited 2003]; Available from: http://www.macworld.com/article/21485/2001/02/geforceannc.html.
27. Hwu, W.-M., et al., *Compute Unified Device Architecture Application Suitability*, in *Computing in Science & Engineering*2009. p. 16-26.
28. McMains, S., R. Kardekar, and G. Burton, *Finding Feasible Mold Parting Directions using Graphics Hardware.* Computer-Aided Design, 2006. **38**(4): p. 327-341.
29. Gray, P.J., F. Ismail, and S. Bedi, *Graphics-assisted Rolling Ball Method for 5-axis Surface Machining.* Computer-Aided Design, 2004. **36**(653-663).
30. Gray, P., F. Ismail, and S. Bedi, *Arc-intersect method for 5-axis tool positioning.* Computer-Aided Design, 2004. **37**: p. 663-674.
31. Dokken, T., T.R. Hagen, and J.M. Hjelmervik, *The GPU as a high performance computational resource.* Proceedings of the 21st spring conference on Computer graphics, 2005: p. 21-26.
32. Roth, D., F. Ismail, and S. Bedi, *Mechanistic modelling of the milling process using complex tool geometry.* The International Journal of Advanced Manufacturing Technology, 2005. **25**(1): p. 140-144.
33. Roth, D., F. Ismail, and S. Bedi, *Mechanistic Modeling of the Milling Process Using an Adaptive Depth Buffer.* Computer-Aided Design, 2003. **35**: p. 1287–1303.
34. Inui, M. and A. Ohta, *Using a GPU to Accelerate Die and Mold Fabrication.* IEEE Computer Graphics and Applications, 2007: p. 82-88.

35. Carter, J.A., T.M. Tucker, and T.R. Kurfess, *3-Axis CNC Path Planning Using Depth Buffer and Fragment Shader.* Computer Aided Design and Applications, 2008. **5**(5): p. 612-621.
36. Stone, S.S., et al., *Accelerating advanced MRI reconstructions on GPUs.* Journal of Parallel Distributed Computing, 2008. **68**(10).
37. Appel, A. *Some techniques for shading machine renderings of solids*. in *Proceedings of the AFIPS Joint Computer Conferences*. 1968. Atlantic City, New Jersey.
38. Cook, R.L., T. Porter, and L. Carpenter, *Distributed ray tracing.* ACM SIGGRAPH Computer Graphics, 1984. **18**(3).
39. Roth, S.D., *Ray casting for modeling solids.* Computer Graphics and Image Processing. , 1982. **18**: p. 109-144.
40. Glassner, *Principles of Digital Image Synthesis*1995, San Fransisco: Morgan Kaufman.
41. Griffiths, *Toolpath based on Hilbert's curve* 1994. **26**(11): p. 839-844.
42. Zhang, D. and A. Bowyer. *CSG set-theoretic solid modelling and NC machining of blend surfaces*. in *Proceedings of the second annual symposium on Computational geometry*. 1986. Yorktown Heights, New York.
43. Jang, D., K. Kim, and J. Jung, *Voxel-Based Virtual Multi-Axis Machining* International Journal of Advanced Manufacturing Technology, 2000. **16**(10): p. 709-713.
44. Jacobs, P.F. and D.T. Reid, *Rapid prototyping, and manufacturing: fundamentals of stereolithography.*1992, New York: McGraw-Hill.
45. Engle, K., et al., *Real-Time Volume Graphics*2006, Wellesley, MA: A K Peters, Ltd.
46. Chiou, C.J. and Y.S. Lee, *A shape-generating approach for multi-axis machining G-buffer models.* Computer-Aided Design, 1999. **31**(12): p. 761-776.
47. Weisstein, E.W. *Point-Line Distance--3-Dimensional*. From MathWorld--A Wolfram Web Resource 2011 [cited 2011 August]; Available from: http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html.
48. Amazon.com. *High Performance Computing Using Amazon EC2*. 2011; Available from: http://aws.amazon.com/ec2/hpc-applications/.
49. nVIDIA, *CUDA Toolkit 4.0.* 2011.
50. Moore, G.E., *Cramming more components onto integrated circuits.* Proceedings of the IEEE, 1998. **86**(1): p. 82-85.
51. Coromant, S., *Metalcutting Technical Guide.* Sandviken, Sweden, 2005.
52. Kalpakjian, S. and S.R. Schmid, *Manufacturing processes for engineering materials*. 4th ed2003, Upper Saddle River, N.J.: Prentice Hall. xvii, 954 p.
53. Grahame, J. *Kodak's First Digital Camera*. 2008 [cited 2011 August]; Available from: http://www.retrothing.com/2008/05/kodaks-first-di.html.
54. Wgsimon. *Transistor_Count_and_Moore's_Law_-_2008.svg*. 2011; Available from: http://en.wikipedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg.