Computer-Aided Design and Applications

http://www.cadanda.com

3-Axis CNC Path Planning Using Depth Buffer and Fragment Shader

Jeremy A. Carter[1], Thomas M. Tucker[2] and Thomas R. Kurfess[3]

[1]Clemson University, carter9@cs.clemson.edu
[2]Tucker Innovations, Incorporated, tommy@tuckerinnovations.com
[3]Clemson University, kurfess@clemson.edu

ABSTRACT

A CNC tool path planning technique is proposed which will take advantage of advances in the programmability of high-performance, low cost graphics acceleration hardware. Though similar to another recently proposed technique using the depth buffer, this method benefits from simpler geometric demands and solves the gouging problem through the use of fragment shader programs. This simple technique accelerates the path planning process and provides a foundation for future work exploiting this technology. A sample of the algorithm using this technology is presented, with promising results.

Keywords: path planning, GPU, tool nose radius compensation, fragment shader.
DOI: 10.3722/cadaps.2008.612-621

1. INTRODUCTION

The automation of tool path planning for computer-numerically controlled (CNC) machining is a long-standing open problem. Though many proposals have been made, advances toward a solution have been hindered by heavy processing demands due to the increasing complexity of desired geometry and the increasing capabilities of milling machines from 3- to 4- and 5-axis machines. This research proposes to leverage the programmability, parallel architecture, and vector operation optimizations of commodity graphics hardware to close that gap. Graphical processing units (GPUs) have advanced greatly over the last few years. Though the development of this technology is driven by the computer gaming and film special effects industries, the advent of programmable shaders as well as non-graphics-based interfaces have opened possibilities for their use in general purpose processing. This paper proposes an algorithm using this technology that is both efficient and easy to implement. Similar to previous work by Inui alone [8] and later Inui and Ohta [9], it uses the natural vector optimizations of the hardware through the graphics pipeline interface to automatically generate a reasonable, if not optimal, cutting path for both roughing and finishing 3-axis milling. Unlike their work, the proposed technique does not use an inverted cutter location surface constructed by geometric representations of the tool. Instead, the modeled part surface provides the basis working surface, and the gouging problem is solved through the novel use of the new programmability available on graphics hardware in the form of a fragment shader. The goal of this research is to provide an automated, easy-to-implement planning solution which gains orders of magnitude of speed over existing solutions due to the processing power of the GPU. The scope of the research is restricted to the generation of a tool trajectory and subsequent correction of that trajectory with relation to tool nose geometry. Tool velocity and material interaction are reserved as future research.

The remainder of the paper is organized as follows. First, previous work in machine tool path planning is discussed, as well as relevant research supporting the widespread applications of programmable GPU technology in CAD and other research. A review of the graphics pipeline and graphics hardware capabilities follows for those readers unfamiliar with basic graphics principles. The algorithm is presented in its basic form, and then the added functionality for tool

nose radius compensation using a fragment shader. Finally, some comparative results to CPU solutions are presented, along with a brief discussion of open problems and future research opportunities using this technology.

2. PREVIOUS WORK

Many attempts have been made at solving the tool path planning problem, though most of these are based upon parametric surface models of the target geometry. As parametric surfaces combined via boundary representation models can define most any shape in an exact and continuous fashion, these are the preferred modeling methods in most CAD programs. Also, parametric surface models benefit from concise representations, so data storage and transmission benefit from small overhead. Cho *et al.* explored path planning options given parametric surface patches [2]. In 2001, Lartigue *et al.* performed research on the use of B-splines as opposed to linear and circular paths [14]. However, with the increased use of graphics acceleration hardware, which uses polygonal primitives, non-parametric specifications of geometry have become more commonplace. Any attempt to discretize geometric specifications invariably introduces questions of tolerance. Austin *et al.* describe a comparison between several common discretization algorithms for both the model surface and the tool surface [1]. A patent by Jepson for Ford Motor Company describes a method using of building an image-based *z*-buffer analysis of a CAD model and the relation between pixel scale and product tolerance [10].

The automation of path planning algorithms is a difficult problem, largely due to the need for the machine to perform analysis of the part, both for capability and optimization. Fussell *et al.* developed techniques to dynamically set feedrates based on force constraints [4]. Jerard *et al.* investigated a dynamic approach to evaluating machine capability [11]. Khardekar *et al.* used programmable graphics hardware-accelerated algorithms for determining the 2-moldability of geometric parts [12].

Beginning in 2003, the first graphics cards featuring programmable hardware and associated high-level shading languages became readily available. This led to a boom in research to exploit this budding technology in both the graphical and physical modeling realms. Marinov *et al.* achieved a GPU-based solution to multiresolution deformation and reconstruction of CAD models [16]. Geist *et al.* employed the GPU to solve lattice-Boltzmann flows of light through clouds [5]. For foundations in GPU applications outside of graphics research, the reader is directed to [15],[17].

As mentioned above, the use of programmable graphics hardware is not new to manufacturing research; other noteworthy GPU-CAD applications follow. Roth *et al.* used an adaptive depth buffer to aid in modeling milling processes [19]. Hjelmervik and Hagen examined the tessellation of parametric surface for aid in CAD surface validation [7]. Vona and Rus implemented a toolpath planning solution for mechanical etch on the GPU [20]. Greß *et al.* explored GPU-based collision detection algorithms for deformable surfaces [6]. Last year, Ren *et al.* designed a haptic interface system to aid in CAD management of volumetric modeling on the GPU [18]. Quite recently, Kurfess *et al.* explored applications for several common CAD problems on the GPU, including iterative closest point mapping and registration processes [13].

3. GRAPHICS HARDWARE REVIEW

The majority of performance computers on the market contain some sort of graphics acceleration hardware. These cards contain dedicated, high-speed, cached memory and specialized graphical processing units (GPUs) that are optimized for vector and matrix operations. The processing units on these graphics cards are essentially multi-pipeline co-processors specialized to perform floating-point calculations on geometric data. Advancing performance demands from the computer gaming and film special effects industries have pushed the development of this branch of hardware to the point where most GPUs can out-perform even multiple CPUs by several orders of magnitude. Further, this technology is available through retail channels for a fraction of the cost of multiple CPU systems, with capable cards falling in the $100-$500 cost range in the current market. While graphics card capabilities vary, it is assumed that the minimum hardware capabilities and software proficiency needed for the approach discussed herein are available to the reader.

To extract this level of performance from the GPU requires a certain minimum amount of knowledge of the architecture, as well as some proficiency in one of several specialized graphics APIs. The most prominent of these are the open source OpenGL and Microsoft's Direct3D. (OpenGL is used throughout this paper, though equivalent functionality exists in both libraries.) To use these APIs, one must begin with an understanding of the specialized

architecture native to graphics acceleration cards. As previously stated, these processors are optimized as pipelines for operating upon floating-point data. The typical operations of this pipeline convert three-dimensional vertex data and scene specifications such as lighting into a two-dimensional rendered image. Figure 1 shows a simplified version of the pipelining operations that occur. Geometry is specified by the user in the form of a mesh of vertex data. (Though most modern APIs support parametric specifications of geometry, such requests are evaluated to a vertex mesh approximation by the hardware upon entering the pipeline.) Per-vertex operations, such as viewing transformations, lighting, texture coordinate generation and calculation of normals occur here. Next, polygons are decomposed into triangle and/or quadrilateral primitives. Rasterization converts the primitives into fragments, i.e. portions of data which represent potential pixels. It is at this stage that lit vertex colors and texture coordinates are interpolated between vertices. Per-fragment operations such as texture application and blending occur, and then the fragments pass through a series of tests to determine which will be displayed for each pixel. These include stencil testing for masking operations and depth testing for hidden surface removal, among others. The output is sent to an area of on-card memory known as the framebuffer. Though the framebuffer's primary function is to hold the color information that comprises the final image, it can be enabled to contain various other types of information in associated sub-buffers such as the depth buffer for occlusion testing, the accumulation buffer for additive effects such as second order illumination, and the stencil buffer for masking effects.

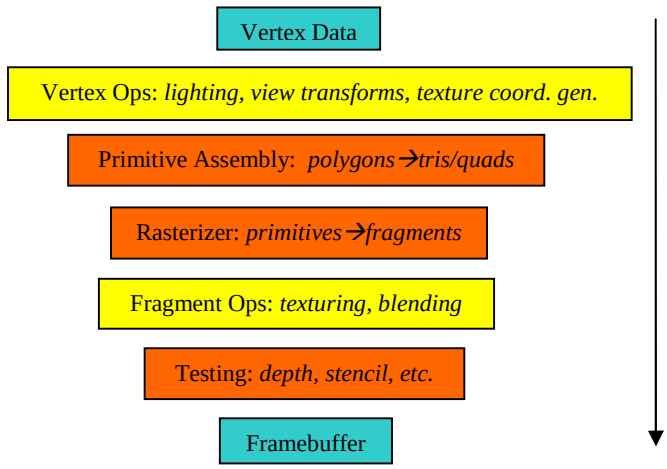

Fig. 1: Graphics Pipeline. Data enters as vertices and leaves as a framebuffer image. Vertex and fragment operations are programmable; others are fixed for now.

Of particular interest in this process are the vertex and fragment operation stages. It is in these stages that specialized programs called shaders may be inserted into the pipeline to replace the fixed functionality. The programs may be considered computational kernels that are executed on a per-vertex or per-fragment basis respectively. Both vertex and fragment shaders have many applications in computer graphics, but fragment shaders are more common in general-purpose GPU (GPGPU) applications. Currently, there are three shader languages in widespread use. The OpenGL Shading Language is used in combination with OpenGL, and the High-Level Shading Language (HLSL) corresponds to Direct3D. NVIDIA has also released Cg, which works with either API, but is restricted to use on NVIDIA hardware.

The common algorithm of GPGPU solutions is to use fragment shaders as asynchronous parallel processing kernels which operate on data stored in onboard memory units called textures. In common graphics usage, textures are blocks of image data that map to scene geometry, often with the intent of creating the illusion of complexity. For instance, rather than specifying a wall of individual bricks within the geometry, one might specify a single quadrilateral with a brick image texture pasted over it. Standard GPGPU approaches treat the texture memory spaces as large, 2-dimensional arrays, which, when mapped to a screen-aligned quadrilateral, can be acted upon by fragment programs for each pixel location in the render target. With the ability to render to textures, the number of fragment program instantiations increases further. Further, it is possible to set up a feedback loop and "ping-pong" between an input and output texture for synchronous update problems which map nicely to this common usage.

Though the work in this paper is focused upon exploiting the natural functionality of the GPU through a graphics API, other methods of access exist in the form of general-purpose languages written specifically for GPUs. These include BrookGPU, Sh, and Shallows. More recently, NVIDIA has released a set of extensions to the C language called CUDA (Compute Unified Device Architecture) that allows direct access to the GPU, though it is only available for cards using NVIDIA's G80 architecture.

4. ALGORITHM

The algorithm proposed here takes advantage of the automatic depth testing available through graphics APIs. Information is gathered by rendering the part, and then a simple path is planned based upon that information. The goal of this work is first and foremost to reduce the time (chronological and person-hours) necessary to plan a cutting path. The former is accomplished by leveraging the computational power of the GPU. For the latter, a minimum of input is required of the user in order to plan a path for a given orientation; this is primarily concerned with the level of detail required and the difficulty of cutting the material. Human inspection is still necessary to determine part orientation. Another goal is for this to be a solution that is easy to implement. The programming techniques employed are current fare for undergraduate graphics courses. Finally, the target applications for this technique are small-scale manufacturing runs, such as prototyping and die and mold manufacture. For these applications, manufacturing efficiency is less imperative than planning efficiency.

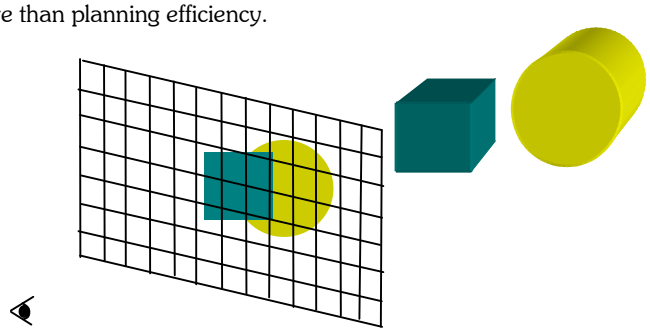

Fig. 2: Depth buffer. Since the cube is between the observer and the cylinder, it wins the depth test, and its color appears in the image projection where the shapes overlap.

4.1 Depth Buffer Path Planning

The algorithm takes advantage of the aforementioned depth buffer, also known as the z-buffer. When enabled, this memory space contains the distance along a ray from the viewing plane to the geometry displayed at each pixel. More specifically, as each polygon is rendered, it is divided into fragments which represent its potential contributions to some number of pixels. Each portion of fragment data is communicated to its pixel location asynchronously, communicating at least the fragment's color and distance from the viewing plane. Then, each fragment's depth is compared against a specified criterion stored in the depth buffer to see which should be kept and promoted to the framebuffer (the default selection being the nearest). An example of this occlusion testing is shown in Figure 2; the cube is nearer to the observer, so for pixels which align with both shapes, the cube's color is chosen. The cost of this functionality is dependent upon the number of polygons being rendered, the resolution of the framebuffer, and the percentage of filled screen space, though it may generally be said to be proportional to the polygon count.

The foundation of the proposal is the generation of a z-map of the part surface. The part is rendered from the perspective of the tool such that the centerline axis of the tool (z-axis) projects into the image plane (defined as x,y). Using an orthogonal projection, this ensures that the depth buffer contains the straight-line distance parallel to the z-axis at each pixel from the tool reference point to the top surface of the part at that discrete location. This depth information may be captured by the OpenGL command glReadPixels, which transfers the depth buffer information from card memory to main memory. From this array of depth values, the contributing surface points may be reconstructed by using the current rendering projection to reverse the viewing transformations performed on the geometric data, yielding a sheet of surface points specified in model-space coordinates. The OpenGL utility gluUnProject conveniently packages these operations. This algorithm only chooses the topmost surface points thanks to the automatic occlusion testing performed via the depth buffer; confusion with undercut areas is not a concern.

From this collection of points, it is possible to construct a naïve, but effective, tool path by performing an alternating raster scan across the points as shown in Figure 3(a).

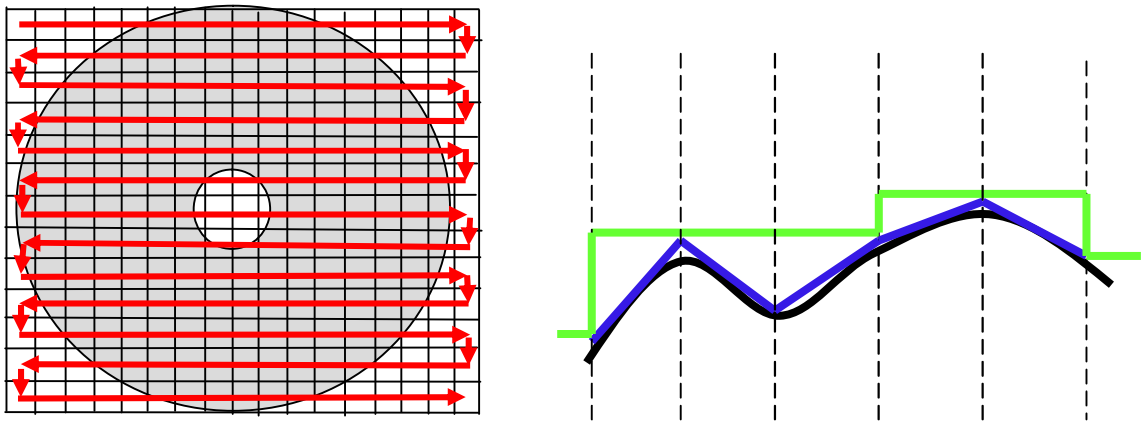

Fig. 3: a): Alternating raster scan, or "lawnmower path." b): Side detail of roughing trajectory (green) and finish trajectory (blue) over part surface (black).

The simplicity of the path design makes it appropriate for both roughing and finishing operations. Though this is a discrete approximation of the surface, machine tools also perform such approximations, as even parametric cutting instructions are decomposed into short linear moves. A simple scan directly along the unprojected points yields a finishing path, while roughing operations perform intermediate stair-step maneuver whenever adjacent pixel depths differ sufficiently. Figure 3(b) offers a detailed side view of both roughing and finishing operations. Figure 4(a) shows a sample roughing pass over an impeller model, and Figure 4(b) details the roughing maneuver in close-up.

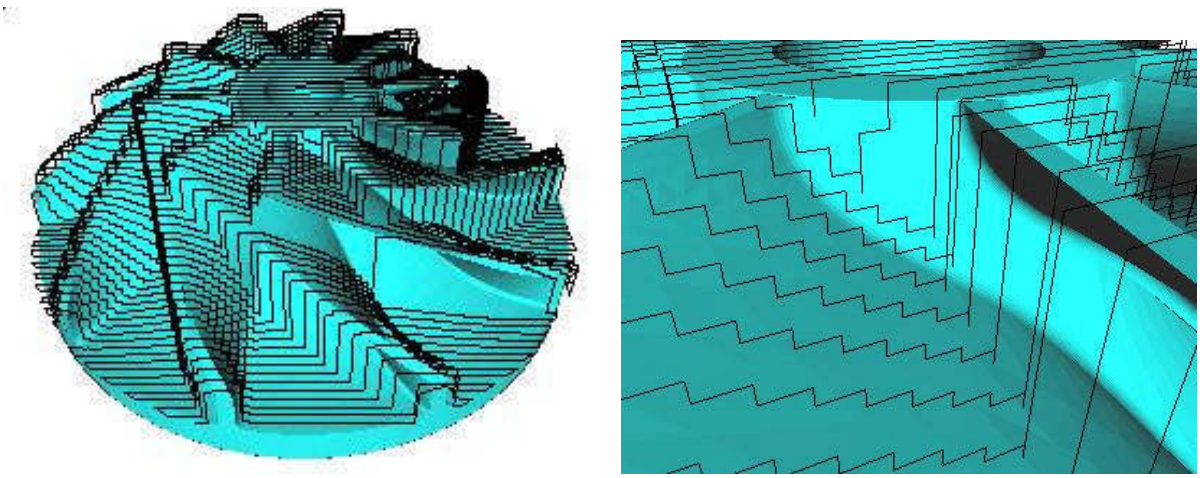

Fig. 4: a) Roughing pass over impeller model. b) Close-up of roughing stair step.

4.2 Tool Nose Radius Compensation

The above foundational algorithm operates on the assumption that the reference point of the tool and the resolution of the discrete sampling due to rendering will be sufficient information to prevent gouging of the part. Since the reference points of different tool types are not all located on the tip of the tool, this is an unfair assumption. Inui proposed a solution to this problem through the generation of a cutter location surface by rendering an inverse offset sweep of the surface [8]. That solution requires the generation of geometry to represent the tool nose at each vertex of the part model, as well as connective geometry representing the path between vertices. The generation of this quantity of

additional geometry is very computationally expensive in itself, and it also adds to the occlusion testing required by the graphics hardware. Though some optimizations may be made by preliminary occlusion testing, this additional computation remains unsatisfying.

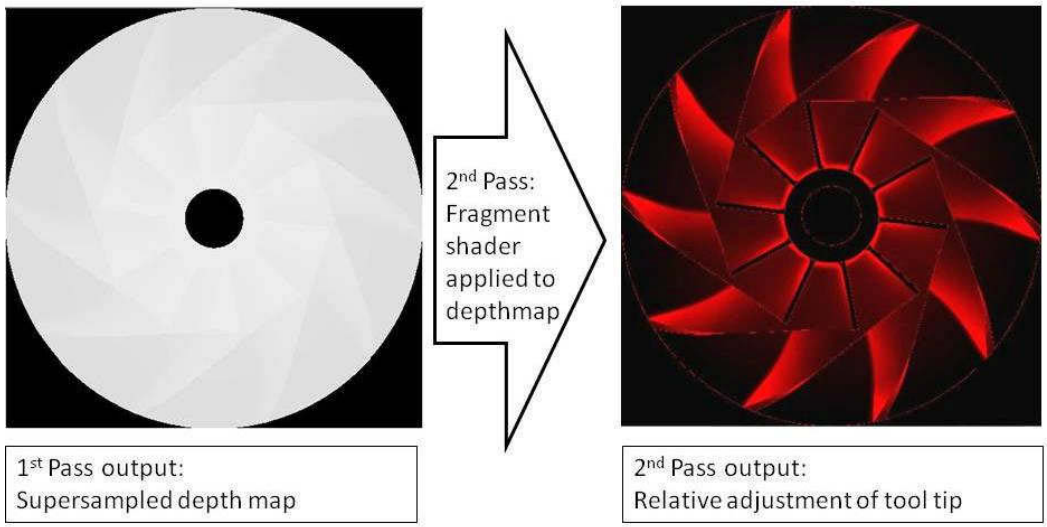

Fig. 5: Dataflow diagram for tool nose radius compensation.

This algorithm is a hybrid of traditional graphics and GPGPU techniques, performing the tool nose radius compensation as an intermediate refinement step. Taking a cue from Inui and Ohta's proposal of a high-resolution grid [9], and recognizing the greater resolution available by rendering to a texture instead of the framebuffer, the solution begins with a supersampled render of the part with a texture unit as the render target. The density of samples is determined by the precision of the tool and the tolerance requirements of the final product. Rendering to a texture instead of the common framebuffer allows for use of a larger buffer while still supporting depth testing. Next, a quadrilateral is rendered at the target final resolution. By mapping the supersampled depth texture to this quadrilateral, fragment shaders can perform random access on that data and downsample from any of the texture memory. During execution, fragment shader kernels are unable to sample values from their neighbor instances due to the asynchronous pipeline architecture, but they are able to perform random access on multiple textures simultaneously. By first rendering to a texture, each shader instance can employ a box filter technique on the values of its neighbors, similar to the convolution filters used in image processing for blurring and edge detection. In this way, the final depth value at each pixel is calculated in parallel by separate instances of the fragment shader program, each of which has access to the entirety of the supersampled depth data entering the pipeline. Figure 5 outlines the data flow through the algorithm. Here, the output of both stages of the algorithm can be seen: a depth map from the supersampling pass (scaled down for presentation), and a representation of the relative adjustment to the tool at each pixel location. For this representation, brighter red indicates a greater adjustment, i.e., a steeper slope at that point on the model, while flatter areas appear darker.

GPU Path Planning Algorithm with Tool Nose Radius Compensation

1) Render part from tool perspective at supersampled resolution to offscreen buffer
2) Render supersampled texture using TNRC shader to:
    a. Unproject supersampled depths to find part surface
    b. Calculate tool reference point and surface under filter
    c. Determine maximum gouge and correct tool reference point
    d. Reproject tool reference point to depth output
3) Unproject downsampled depth values
4) Run "lawnmower path" over constructed surface points to generate path

Listing 1: Pseudocode for tool nose radius compensation.

Each fragment program invocation can be made aware of its screen (pixel) coordinates. Given its location, it can choose the supersampled data appropriate to downsample to the final resolution. For the sample implementation, supersampling is performed in a 3x3 pattern to allow for a center point at each filter box (though a larger supersample is certainly feasible, based on the tool resolution needs). If the tool is assumed to be centered over this pattern, the shader can calculate first the reference point for the tool and then the tool nose location above each sample. Any overlap between the cutting surface and the part surface is resolved, and a refined depth value is then passed along to the depth buffer for final planning. Thus the tool compensation only affects the z-value of the cutter location, as described by Duncan [3]. The final path still passes through the center of each pixel (in the xy-image plane), and the remainder of the foundation algorithm proceeds as before. Surface depths are captured from the depth buffer and unprojected to recreate their model-space coordinates. Note that even though the screen-aligned quad is of a uniform depth in the scene, the default depth values can be overwritten by the fragment shader output. In this way, fragment shaders can output multiple values to the various buffers comprising the framebuffer, even ignoring the standard meanings of RGBA (red, green, blue, alpha) pixel color output. This ability to assign arbitrary semantic value to texture and framebuffer values is what makes the shader interface viable for GPGPU.

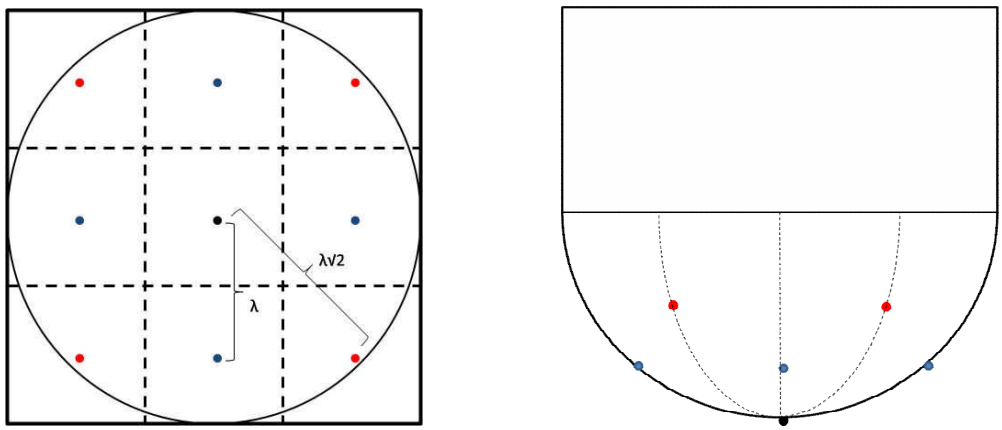

Fig. 6: a. Top view of downsample filter. b. Side view of cutter contact points above (or below) each part surface sample.

Since the reference point for flat-end mills is on the tool surface, a simple filter is used for these tools which merely selects the minimum depth value (corresponding to the highest part sample) under the tool. For 3-axis milling over an evenly-discretized grid, this is a trivial case. Calculation for ball-end mills is straightforward if not quite so trivial. This requires a reverse projection from each of the sampled values under the tool to convert the screen-space coordinates and captured depth values into model-space coordinates. The tool's reference point (at the center of the spherical end) is assumed to reside one radial length above the center sample. In a best-case scenario, this is the closest the tool could be to the surface without gouging. To check for gouging, the tool surface is calculated at each point around the center. The fragment shader is able to accept relevant input such as the tool radius and the radius/pixel ratio to assist in these trigonometric calculations. Taking advantage of symmetry, only three tool surface depths need to be calculated, the center, a side, and a corner of the box filter. These depths are given by:

$$d_{center} = R_z - r \tag{4.1}$$

$$d_{side} = R_z - \sqrt{r^2 - \lambda^2} \tag{4.2}$$

$$d_{corner} = R_z - \sqrt{r^2 - 2\lambda^2} \tag{4.3}$$

where R_z is the reference point depth, r is the tool radius, and λ is the spacing between samples. Should the tool height be less than the height of the material at the corresponding site, this overlap is noted. Figure 6 illustrates how the box filter examines the samples and shows the cutter surface points calculated by the shader. The maximum of all overlaps is selected and added to the reference point height, and this corrected reference height is projected back into the scene

and returned to the planning algorithm. This process is illustrated in Figure 7. Finally, the corrected downsampled cutter location values are assigned to the fragment depth and become the depth values acquired by glReadPixels in the original algorithm. Listing 1 outlines the algorithm.

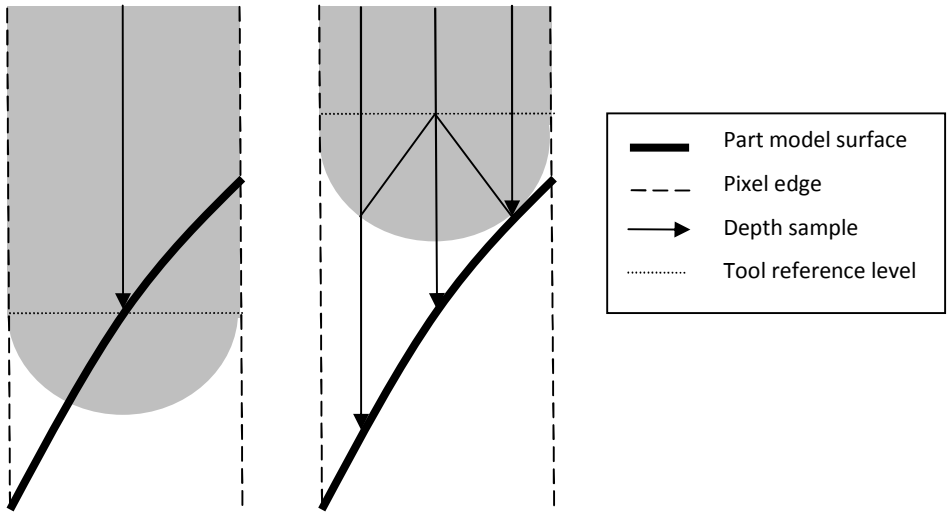

Fig. 7: Side view of tool nose radius compensation. The left side shows the output of the basic algorithm. The right shows the corrected results after compensation due to a 3x3 supersampling.

5. RESULTS

The presented planning algorithm shows great promise. Traditional approaches to this problem have been recorded to take as much as 3 to 4 hours to compute [9], while this method produces paths for non-trivial parts in a half-second. Figure 8 shows a sample close roughing path that was computed on non-trivial geometry in seconds. This model of human teeth was derived from a 3D scan of a patient and contains around 475,000 triangles.

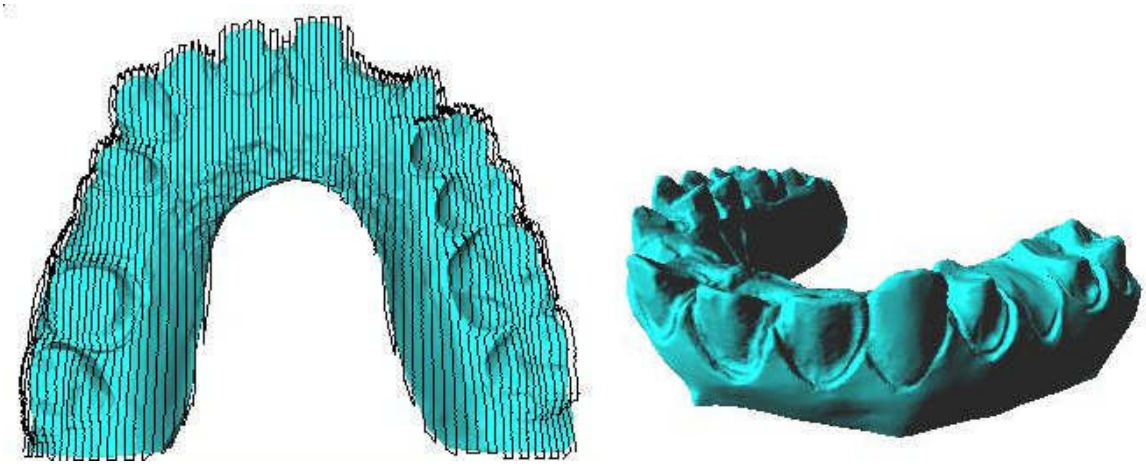

Fig. 8: Teeth model, ~475,000 triangles, shown with and without the cutting path overlay (perspective view).

Unfortunately, as this technique relies heavily on the graphics API, and as it is a relatively new idea, developing a reasonable benchmark is difficult. The results shown in Figure 7 are a comparison between algorithm as executed on the GPU, and the same algorithm executed using the Mesa 3D software rendering libraries. Use of these libraries

allows largely the same functionality without the benefit of hardware acceleration. For these trials, the reference machine was a 3.2 GHz Pentium 4 with 3.2 GB of RAM, running a Linux operating system. Hardware rendering acceleration was performed on an NVIDIA 6800 Ultra graphics card with 512 MB of onboard RAM. The time trials were executed on a progressive decomposition of the teeth model, from its original polygon count to a 95% simplification. Figure 9a illustrates the gains available with the basic, naïve algorithm. Figure 9b shows the increasing gap between the GPU and CPU's computational abilities when the problem is extended to include tool-nose radius compensation. Though the GPU solution does show some rise in cost as the polygon count climbs, it is a negligible rise in relation to the CPU's cost for the same model.

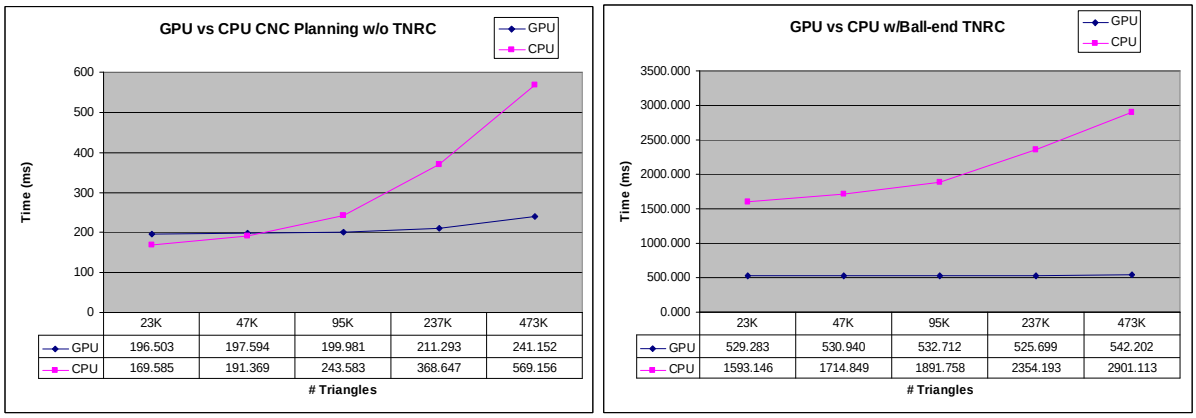

Fig. 9: Running times for CPU vs. GPU implementations. Comparison based on teeth model from Figure 6, shown at various polygon decompositions. a) Basic algorithm b) Tool nose radius compensation via 3x3 supersampling.

This technique is not without its limitations. Unlike mainstream CAM packages, this algorithm expects that model inputs are specified as polygon meshes. However, mesh export formats such as STL and OBJ are ubiquitous among CAD programs. Detractors of z-buffer path planning methods complain about the limited resolution available by such discrete methods. This need not be prohibitive to such methods, however, as simple fixes like the tiling method suggested in [9] can greatly increase the planning resolution at a mere linear (per tile) increase in computation time, assuring unbounded scalability. Such objections will become still less relevant as the size of GPU offscreen rendering buffers continues to increase. Current APIs support up to 4096 pixels per dimension, with hardware having as much as 4GB of onboard, texture-dedicated memory. Though tiling the planning space removes any bound on accuracy, the maximum accuracy of this technique when using only one buffer is currently bound by the aforementioned 4096 pixels per dimension. Therefore, assuming tolerance within 0.001 in. for a pixel width, a single tile implementation of the algorithm provides up to a 16 in^2 surface. Execution time scales linearly with the expansion into additional tiled areas. It is assumed that, in practice, this technique would use the tiling approach appropriate to the needed tolerance, thereby removing any limitations due to variations in hardware availability.

No consideration has yet been made for milling from different orientations (multi-axis), though this work is forthcoming. Also, human interaction is still needed for primary inspection and orientation of the part, as well as for post-cutting analysis. Again, these issues are already under review.

6. CONCLUSION

The development of programmable graphics hardware has opened many new possibilities for research. Once the sole realm of the gaming and film industries this resource is now available for use in manufacturing and CAD research, as well as a variety of other fields. Initial results of CNC tool path planners empowered by GPU technology show marked improvement over traditional techniques with regard to processing time. Particularly when more complex problems, such as tool-nose radius compensation are considered, the GPU performs orders of magnitude better than its sibling CPU processor. Further possibilities exist for this research, including the addition of collision detection for the work envelope in planning a cutter path. Perhaps the expansion into higher-axis machine planning would present the greatest challenge, as well as inroads into a difficult problem domain.

7. ACKNOWLEDGEMENTS

This work was partially funded by the National Science Foundation under Grant Number DMII-0600902. The government has certain rights in this material. Any opinions, findings and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] Austin, S.; Jerard, R. B.; Drysdale, S.: Comparison of Discretization Algorithms for NURBS Surfaces with Application to Numerically Controlled Machining, Computer Aided Design, 29(1), 1997, 71-83.

[2] Cho, J. H.; Kim, J. W.; Kim K.: CNC tool path planning for multi-patch sculptured surfaces, International Journal of Production Research, 38(7), 2000, 1677-1687.

[3] Duncan, J. P.; Mair, S. G.: Sculptured Surfaces in Engineering and Medicine, Cambridge University Press, Cambridge, 1983.

[4] Fussell, B. K.; Jerard, R. B.; and Hemmett, J. G.: Robust feedrate selection for 3-axis NC machining using discrete models, Journal of Manufacturing Science and Engineering, 123(2), 2001, 214-224.

[5] Geist, R.; Rasche, K.; Westall, J.; Schalkoff, R.: Lattice-Boltzmann lighting, Proceedings of Eurographics Symposium on Rendering, June, 2004, 355-362.

[6] Greß, A.; Guthe, M.; Klein, R.: GPU-based collision detection for deformable parameterized surfaces, Eurographics 2006, 25(3).

[7] Hjelmervik, J.; Hagen, T.: GPU-based screen space tessellation, Mathematical Methods for Curves and Surfaces, Tromsø, 2004. M. Dæhlen, K. Mørken, and L. L. Schumaker (eds.), Nashboro Press, 2005.

[8] Inui, M.: Fast inverse offset computation using polygon rendering hardware, Computer-Aided Design, 35(2), 2003, 191-201.

[9] Inui, M.; Ohta, A.: Using a GPU to accelerate die and mold fabrication, IEEE Computer Graphics and Applications, 27(1), 2007, 82-88.

[10] Jepson, C. T.: Absolute Gouge Avoidance for Computer Aided Control of Cutter Paths, U. S. Patent #5,043,906, issued August 1991, assigned to Ford Motor Company.

[11] Jerard, R. B.; Fussell, B. K.; Doherty, D.; Schulyer, C.; Ryou, O.: Dynamic evaluation of machine tool process capability, 2004 NSF Design, Service and Manufacturing Grantees and Research Conference, Dallas, TX, January 5-8, 2004.

[12] Khardekar, R.; Burton, G.; McMains, S.: Finding feasible mold parting directions using graphics hardware, Computer-Aided Design, 38(4), 2006, 327-341.

[13] Kurfess, T. R.; Tucker, T. M.; Aravalli, K.; Panyam, M.: GPU for CAD, Computer Aided Design and Applications, 4(6), 2007, 853-862.

[14] Lartigue, C.; Thiebaut, F.; Maekawa, T.: CNC tool path in terms of B-spline curves, Computer-Aided Design, 33(4), 2001, 307-319.

[15] Luebke, D.; Harris, M.; Krüger, J.; Purcell, T.; Govindaraju, N.; Buck, I.; Woolley, C.; Lefohn, A.: GPGPU: general-purpose computation on graphics hardware, course notes, ACM SIGGRAPH 2004.

[16] Marinov, M.; Botsch, M.; Kobbelt, L.: GPU-based multiresolution deformation using approximate normal field reconstruction, Journal of Graphics Tools, 12(1), 2007.

[17] Pharr, M. (ed.): GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, Upper Saddle River, NJ, 2005.

[18] Ren, Y.; Lai-Yuen, S.; Lee, Y.: Virtual prototyping and manufacturing planning by using tri-dexel models and haptic force feedback, Virtual and Physical Prototyping, 1(1), 2006, 3-18.

[19] Roth, D.; Ismail, F.; Bedi, S.: Mechanistic modeling of the milling process using an adaptive depth buffer, Computer-Aided Design, 35(14), 2003, 1287-1303.

[20] Vona, M.; Rus, D.: Voronoi toolpaths for PCB mechanical etch: simple and intuitive algorithms with the 3D GPU, Proceedings of the 2005 IEEE International Conference on Robotics and Automation. April 18-22, 2005, 2759-2766.