Proceedings of IDETC/CIE 2006
ASME 2006 International Design Engineering Technical Conferences &
Computers and Information in Engineering Conference
September 10-13, 2006, Philadelphia, USA

DETC2006-99666

FAST LAYERED MANUFACTURING SUPPORT VOLUME COMPUTATION ON GPUS

Rahul Khardekar
Department of Mechanical Engineering
University of California
Berkeley, CA 94720
Email: rahul@me.berkeley.edu

Sara McMains *
Mechanical Engineering
University of California, Berkeley
Berkeley, California, 94720
mcmains@me.berkeley.edu

ABSTRACT

We present a GPU-accelerated algorithm for computing a fast approximation of the volume of supports required for layered manufacturing in a given build direction, one criterion often used to choose a direction that requires less time and material. In a sequence of rendering passes that project the part in the given build direction, we use depth peeling to identify faces bounding supports. We exploit programmable graphics hardware to compute the total height of all supports at each projected pixel location, scale the values by pixel area, and finally sum over all pixels to find the total volume of supports. For sample parts tested, our algorithm achieves over 99% accuracy and running times ranging from .2 seconds, for a part with 1,252 facets and depth complexity 2, to 1.86 seconds, for a part with 419,798 facets and depth complexity 9.

INTRODUCTION

Prototypes are often built during product design to demonstrate a concept, to design tools or packaging, to analyze manufacturability, or for testing stress or vibration response of the product [1]. Layered manufacturing processes are a class of rapid prototyping processes that build 3-dimensional shapes by depositing layers of raw material as illustrated in Figure 1. These processes can quickly build a small number of prototypes without any part specific tooling. Fused deposition modeling (FDM), stereolithography and selective laser sintering are some examples of layered manufacturing processes [2], [3].

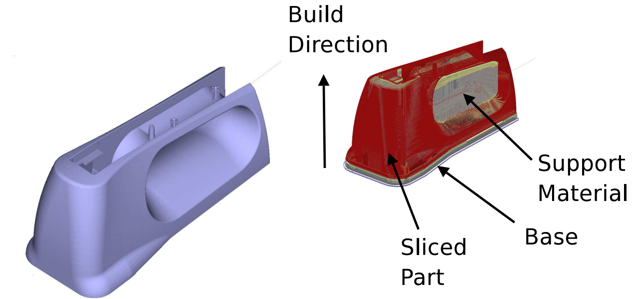

Figure 1. (a) A sample part (b) Sliced part along with the support material

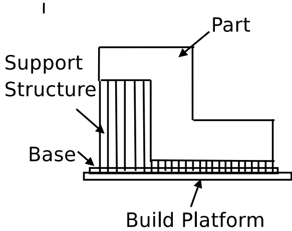

Figure 2. 2D schematic of part showing the support material holding up overhanging features and attaching the part to the build platform

As a pre-processing step for layered manufacturing, a build direction is chosen and the virtual model of the product is sliced by planes normal to that direction. The resulting slice contours are then sent to a layered manufacturing machine where raw ma-

*Address all correspondence to this author.

Copyright © 2006 by ASME

terial is deposited layer by layer to build the entire shape. The part is build on a build platform on which a thin base layer is first deposited as shown in Figure 2. For many processes, additional material is also needed to support overhanging geometry as illustrated in Figure 2. Adding this support material not only increases the manufacturing time, it also has to be removed upon completion, a process that can damage the part surface. Along with the amount of support material, factors like the accuracy of important features, desired anisotropic strength properties of a part, the desired surface finish of the part, and the manufacturing time also affect the choice of the build direction. With respect to support material alone, an orientation minimizing the volume of support is optimal. But in most cases, the final build direction is chosen by considering trade-offs between all the factors described above. If the metrics based on these factors could be computed interactively while comparing build directions, the manufacturer could make an informed orientation choice. The fast computation of the volume metric is difficult because of the high complexities of geometric algorithms involved in identifying faces needing supports; current commercial systems compute the volume of support after the slicing is completed by computing Boolean differences between adjacent layers. We propose to use the computational power of graphics processing units for computing this volume.

Graphics Processing Units (GPUs) have recently evolved into programmable processors capable of performing general-purpose computational tasks. Figure 3 shows a schematic layout of a modern GPU. Two programmable units, the Vertex Processing Unit (VPU) and the Fragment Processing Unit (FPU), can execute a user-defined set of instructions in place of a fixed sequence of geometric transformations, lighting (per vertex operations) and texturing operations (per-pixel operations). Because multiple vertices and pixels are processed in parallel, GPUs can achieve much higher computational speeds than conventional CPUs. We show that GPUs can be very useful for the fast computation of geometric metrics for manufacturing such as the support volume.

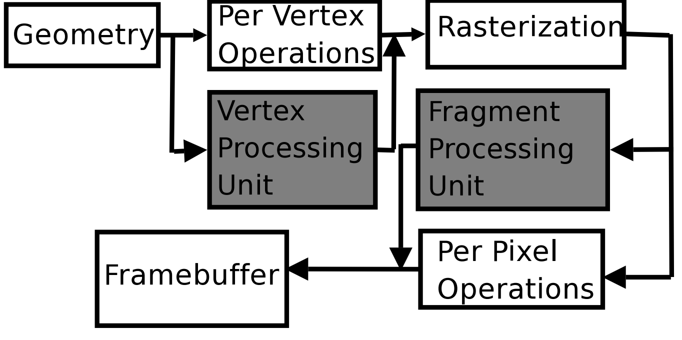

Figure 3. Modern GPU architecture.

In this paper, we propose a GPU-based algorithm for computing the volume of support material needed for layered manufacturing in a given build direction. Our algorithm can compute the volume of support material as the user changes the geometry as well as the build direction. We review the existing literature in this area in the next section before describing the algorithm and our results.

Previous work

Early work that looked at reducing support structure requirements for layered manufacturing focused on finding an orientation that would entirely eliminate the need for a support structure. Thompson and Crawford propose a heuristic search method for finding a build direction requiring no support [4]. Asberg et al. present an $O(n)$ algorithm for determining if there is any orientation in which a polygon or a polyhedron with n vertices can be built without supports [5].

Later work looked at minimizing the volume of support structures. Allen and Dutta present an algorithm for "generating optimal supports" for any polyhedron [6]. But Majhi et al. show that this algorithm, which essentially considers only directions parallel and orthogonal to edges of the convex hull, does not always find an optimal solution, and furthermore that the approximation error cannot be bounded [7]. In 2D, Majhi et al. present algorithms to find an orientation for a polygon that minimizes the contact length of the support/part interface and the support-structure area [7]. Their 3D algorithm for finding an orientation that minimizes the analogous contact area and support structure volume for a polyhedron are limited to *convex* polyhedra, however [8]. Agarwal and Desikan describe a more efficient randomized approximation algorithm for solving the same problem for convex polyhedra [9]. For the general case of a non-convex polyhedron, they show that the set of build directions that minimize the total area of faces that need support can have as many as $\omega(n^4)$ connected components, suggesting that solving the minimization problem for general polyhedra will be too slow for interactive feedback for all but the simplest input.

Researchers have also looked at the problem of finding optimal build direction based on multiple criteria including the volume of support material. Cheng et al. formulate a multi-criteria objective function that is evaluated over a set of candidate directions to find an optimum [10]. This function considers factors like the number of surfaces perpendicular to the build direction, the number of up-facing horizontal surfaces, the area of the base surface and the volume of support material. Lan et al. [11] propose a similar approach where candidate directions are tested for optimality with respect to either surface quality, build time, or the number of sample points needing supports. Frank and Fadel [12] propose a rule based system for choosing the optimal build orientation with respect to the volume of support material, surface finish and tolerances.

Copyright © 2006 by ASME

Related work on GPUs consists of various applications of graphics hardware to manufacturing and other general purpose problems. Previously, the z-buffer of graphics cards has been used to speed the solution of manufacturing problems such as tool path planning [13], [14], and inspection [15]. The advent of programmable GPUs has increased the use of GPUs for general purpose computing tasks such as computational geometry, fluid simulations, database operations, and CAD [16]. Khardekar et al. used programmable GPUs to interactively detect and display undercuts (for providing feedback to designers of parts to be molded or cast) [17]. The undercuts detected by this algorithm are a subset of features that need supports in layered manufacturing. These optimization algorithms could search over a larger search space in the same amount of time by using our new volume computation algorithm.

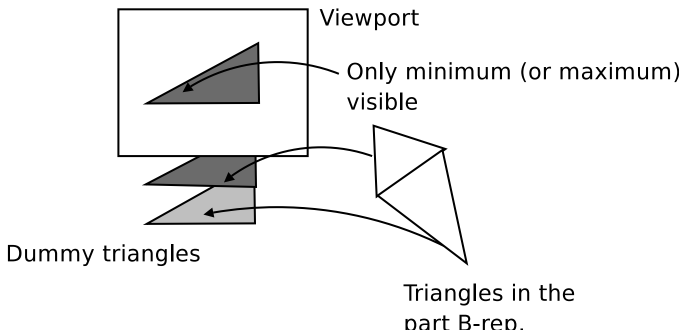

Figure 4. Dummy triangles are rendered instead of the primitives to find the minimum or the maximum value of scalar functions defined over all primitives.

Definitions and terminology

We will use the following standard graphics-oriented terms. As discussed earlier, modern GPUs contain two programmable units, a VPU and a FPU. The VPU executes a user-defined program called a "vertex program" for each vertex. A vertex program can access vertex attributes such as the position, the normal, the color, lighting parameters, and texture coordinates. It can also access a fixed system-dependent number of constant attributes and textures. The vertex program outputs the color, the texture coordinates and the transformed position for each vertex. After this stage, vertices are assembled to form triangles that are rasterized. During rasterization, triangles are subdivided into potential pixels called fragments; per-vertex attributes such as the color and the texture coordinates are linearly interpolated over each triangle. The FPU then executes a user-defined "fragment program" for each fragment. A fragment program can access the interpolated vertex attributes and textures stored in the GPU memory to determine the color and the depth value of each fragment. In modern GPUs, fragment programs can access full floating point textures (32 bits per color channel) as well as depth textures that store depth values instead of color values. Finally, fragments that pass a sequence of tests, including the z-test and the stencil test, become pixels in the frame-buffer. In the stencil test, a user-defined reference value is compared with the value stored at the fragment's location in a stencil buffer; the fragment is discarded if the test fails. In the z-test, the GPU compares the depth value of each fragment with the value stored in the corresponding location in a (usually) 24-bit depth buffer, discarding fragments that fail the user-chosen comparison. An "occlusion query" returns the number of fragments rendered in the frame-buffer for a given rendering pass.

We also use the following FDM-related terms in our algorithms. On an FDM machine, a platform is first built on which the actual part rests as shown in Figure 5. This platform is called the "base." The projected area of the base is slightly larger than the projected area of the part when projected on the base plane. We call the amount of offset applied to the boundary of the projection of the part to find the boundary of the base the "base offset." The overhanging surfaces that require supports and are not supported by the base are called "supported features." The sum of the volume of supports and the base gives us the total volume of support material required.

We will now discuss of our algorithm for the volume computation in detail.

Volume computation

We assume that the input for our algorithm consists of the build direction and a triangulated boundary representation of the part. Before starting our algorithm, we carry out the following initialization.

OpenGL Initialization

Before starting the main algorithm, we set the following OpenGL rendering parameters. We first set the look-at point (the location at which the camera points) at the center of the axis-aligned bounding box of the part (computed beforehand in the original world-space coordinate system by recording the minimum and the maximum value in each dimension while reading in the part file). We place the eye-point (the location of the camera) on the build direction outside the sphere bounding the axis-aligned bounding box such that the camera is facing towards the bottom surface of the part that is deposited first by the layered manufacturing machine, thus aligning the viewing direction (the direction from the eye-point to the look-at point) with the build direction. Until we compute the exact clipping planes bounding the part, we set them such that they enclose the sphere circumscribing the bounding box. For setting the near and far clipping planes for the given viewing direction, we find the minimum and the maximum distance between the part and the plane normal to

Copyright © 2006 by ASME

the viewing direction containing the eye-point by using the following method. We assign a dummy triangle to each triangle of the part and set the coordinates of the vertices of the dummy triangle such that all dummy triangles have the same shape and size and are rendered at the exact same location when projected in the viewport, completely overlapping each other. Figure 4 shows this process for two triangles. In actuality, these dummy triangles occupy a very small number of pixels in the viewport. For all three vertices of each dummy triangle, we compute the minimum (respectively maximum) distance from the eye-point along the viewing direction in a vertex program and set the z-coordinate value equal to the computed minimum (respectively maximum). We also set the color value of each vertex equal to the new z-value, because the color attributes can have full 32-bit floating point precision instead of the 24-bit precision of the depth attribute. We set the z-test such that the triangle with minimum (respectively maximum) z-value is visible. We read back the color value for one pixel where the dummy triangles are rendered to find the minimum (respectively maximum) distance from the eye-point.

For finding the left and the right clipping plane, we temporarily set our eye-point on the x-axis of the view-coordinate system defined by the viewing direction and the up-vector (a vector defined during defining the camera that defines the up direction in the eye coordinate system). We refer the reader to [18] for the details of obtaining the x-axis in world coordinates. We then find the near and the far clipping plane from this new eye-point and set them as the right and the left clipping plane respectively. Similarly, we find the top and the bottom clipping planes by setting the eye-point on the y-axis of the view-coordinate system.

We will now discuss our support volume computation algorithm.

a

0.1 The volume of support material

For computing the volume of support material for a given orientation, we render the first layer of front-facing facets (with normals making an obtuse angle with the viewing direction) and store their z-values in a color buffer. This corresponds to the height of any supports attached to the base. We then render consecutive layers of front-facing facets (relative to that orientation) and back-facing facets by using a depth-peeling technique that peels off unwanted layers of facets using the two-sided depth test that we will discuss in detail below [19]. We observe that supports are enclosed between the $(n+1)^{st}$ layer of front-facing facets and that portion of the nth layer of back-facing facets whose projection overlaps the $(n+1)^{st}$ layer of front-facing facets. We use the stencil buffer to render only those portions of the back-facing facets that overlap the corresponding layer of front-facing facets (to form supports). Figure 5 illustrates a two-dimensional example in which supports are enclosed between the second layer of front-facing facets and the portion of the first layer of back-facing facets where their projections overlap. For every such pair of layers, we identify the supports calculating a per-pixel cumulative sum of the distance between the layers, which is the height of the supports at every pixel. In the end, we add values stored at all pixel locations and multiply the sum by the area of a pixel in world coordinates to obtain the volume of the supports. The process can be described by the following equation:

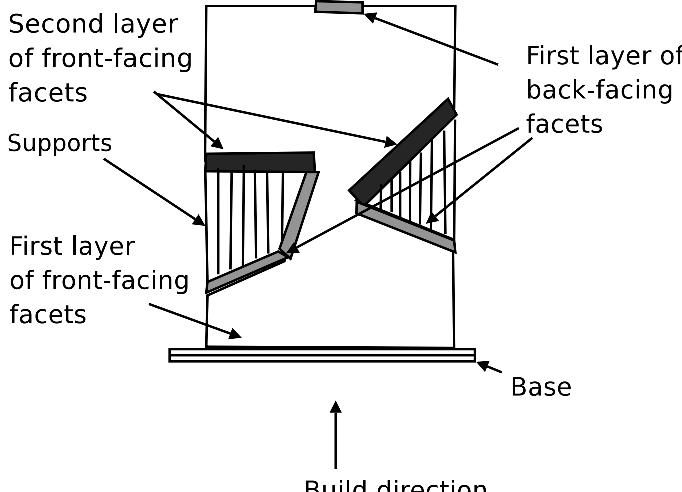

Figure 5. Example of a two dimensional part requiring supports

$$V = A_p \sum_{N_p} \left(h_{BASE} + h_1^F + \sum_{i=0}^{D_c-1} h_{i+1}^F - h_i^B \right) \quad (1)$$

In the above equation, V is the total volume of supports, A_p is the area of a pixel, N_p is the total number of pixels rendered, h_{BASE} is the height of the base layer, h_i^F is the height of the ith layer of front-facing facets, D_c is the depth complexity, and h_i^B is the height of the ith layer of the back-facing facets.

The complete algorithm is summarized in pseudo-code as follows:

```
1. Calculate tight clipping planes.
2. Set up projection,camera parameters.
3. Render the 1st layer of front-facing facets
   setting color=depth and store the z-buffer
   in a depth texture.
4. Grow the rendered region by an amount
   equal to the base offset and store the
   color buffer in a floating point
   texture HeightSum
5. n = 1;
```

 Copyright © 2006 by ASME

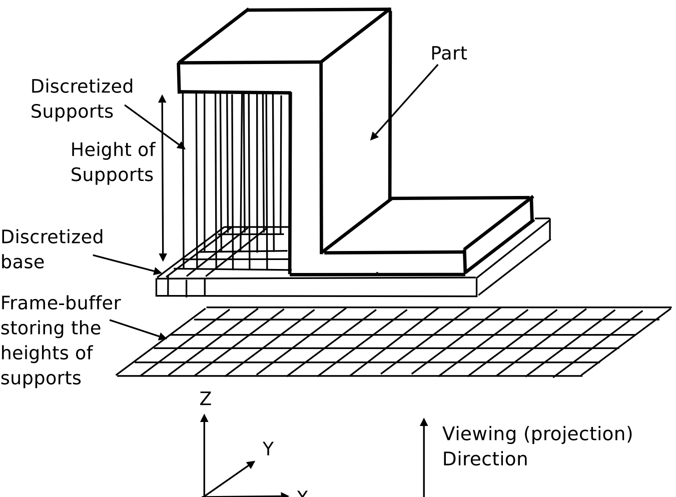

Figure 6. System set up during algorithm. The framebuffer contains the discretized sum of the heights of the base and supports above each pixel.

```
while(true)
{
   6. Render the (n+1)st layer of front-
      facing facets with an occlusion query
      and set stencil value for rendered
      pixels equal to 1. Set color = depth
      value. Copy depth buffer to the depth
      texture.
   7. Break if occlusion query returns zero.
   8. Store color values in a floating point
      texture FrontFaceHeights.
   9. Render nth layer of back-facing facets
      only for pixels rendered in 6 using
      stencil test. Set color=depth.
  10. Subtract the color value of every
      fragment from the corresponding value
      stored in FrontFaceHeights and add the
      result to corresponding value in
      HeightSum.
  11. Render a viewport sized square with
      stencil test such that fragments not
      rendered in 9 are rendered. Set the
      color value to the corresponding
      value in HeightSum.
  12. Store the result in HeightSum.
  13  n++;
}
14 Sum all values in HeightSum
```

We find the clipping planes as discussed in the previous section. We offset the near clipping plane by an amount equal to the thickness of the base below the bottom-most layer so that the distance from the near clip plane corresponds to the thickness of the base. We then render the first layer of front-facing facets (via standard z-buffer hidden surface removal). We use a vertex program to replace the color of each vertex with its post-view-transformation z-value so that we can use the full 32-bit floating point precision of color buffers instead of the 24-bit precision of the z-buffer. Figure 7(a) shows a sample 2D part and 7(b) shows the first layer of front-facing facets. We copy the z-buffer to a depth texture to be used later in depth peeling and copy the color buffer to a floating point texture, call it *HeightSum*.

Thus, *HeightSum* contains the projection of the whole bottom surface of the part. The color value stored in each texel location is equal to the sum of the height of the base layer and the heights of supports that support the bottom surface. We then offset the rendered region for a number of pixels in the x and y direction respectively that is obtained by dividing the base offset distance by the length and the height of the pixel in world coordinates. (The x and y values may be different if the aspect ratio of part projection is not $1:1$ and if the GPU is limited to using square depth/color textures.) In Figure 7, since the part is two-dimensional and the frame-buffer is one dimensional, the offset is only done in the x direction. For two dimensional frame-buffers, we achieve this offset in multiple passes where for every pass, we grow the region by a pixel in directions in a fragment program. Suppose the region needs to be grown by d_x pixels in the x direction and d_y pixels in the y direction with $d_x < d_y$. For every fragment, we look up the color texture value at that fragment's location as well as the four fully adjacent pixel locations. If the color value in *HeightSum* at the current fragment location is zero and any one of the four adjacent texels is non-zero, we set the color value of the current fragment equal to the thickness of the base. This in effect offsets the region by one pixel on all sides. We continue this for d_x number of passes. After that, we grow the region only in the y direction for $d_y - d_x$ passes. Thus the color buffer contains the sum of the pixel-wise height of the base and the bottom-most portion of supports (those connected to the base) as shown for the sample part in in Figure 7(c). We store the resulting color buffer in a floating point texture *HeightSum*.

We then initiate a loop that executes the following for every n starting from $n = 1$:

Note that at the starting point of the loop, we have a depth texture created during the execution of the loop for the previous value of n. (For n=1, the depth texture is created while rendering the first layer of front-facing facets as discussed previously.) We also have a floating point color texture *HeightSum* that stores the cumulative sum of the heights of supports rendered at each pixel up to this time. We render the $(n+1)^{st}$ layer of front-facing facets by using Everitt's two sided depth test [19] in a fragment program as follows . We discard a fragment if its depth value is less than or equal to the corresponding depth value stored in the

Copyright © 2006 by ASME

depth texture. Along with the usual z-test, this second depth test peels off layers of facets that are in front of the nth layer of front-facing facets, thus rendering the $(n+1)^{st}$ layer of front-facing facets.

The color buffer now contains the projection of the $(n+1)^{st}$ layer of front-facing facets as shown in Figure 7(d) for $n = 1$. If there were no front-facing facets in this layer, the occlusion query returns that zero pixels were rendered and we terminate the loop. In practice, we use a vertex program to change the colors of vertices to store their depth value. We also set the stencil bits of fragments rendered in this pass to one. We copy the rendered color buffer into a floating point texture, *FrontFaceHeights*, and copy the z-buffer into the depth texture to be used for depth peeling for larger values of n.

We then render the nth layer of back-facing facets, rendering only pixels with the stencil bit set to one. Thus in the example in Figure 7, the portion of the first back-facing layer that does not bound supports, in this case, namely that portion of the first back-facing layer that is on the top face of the part is not rendered. Figure 7(e) shows the facets rendered in this pass for $n = 1$. Similar to the previous pass, we use a vertex program to set the color value to store the depth value. Still restricted by the stencil buffer, our fragment program subtracts the color value at every fragment that holds the first layer of back back-face depths from the value at that fragment's location in *FrontFaceHeights* (rendered in (c)), and add the result to the value stored in *HeightSum* (rendered in (b)). This gives us the cumulative sum of the heights of supports being rendered at each pixel as shown Figure 7(e).

Note that because of the stencil test, this fragment program only updates fragments that have the corresponding stencil bits set to one. Other fragments remain untouched even if *HeightSum* has non-zero values at their location. Due to the stencil test, only the fragments occupied by the shaded rectangle in Figure 7(d) are rendered in this pass. For the remaining fragments, values from *HeightSum* (values obtained at Figure 7(b)) need to be copied to the current color buffer. We render a viewport sized rectangle and render fragments for which stencil bits are set to zero. We use a fragment program to transfer values stored in *HeightSum* to the color-buffer, giving us a buffer with a pixel-wise cumulative sum of the height of all supports up to the $(n+1)^{st}$ layer of front-facing facets as shown in Figure 7(f). Before incrementing the value of n and going back to the start of the loop, we copy the buffer to *HeightSum*. The process continues until there are no additional front-facing facet layers.

Finally, we find the sum of values stored in *HeightSum* by using Kruger and Westerman's multi-pass reduction technique summarized below [20] .

For summing the values in an NxN texture, this algorithm requires $\log N$ passes. For the ith pass, we render a square of size $N_i x N_i$ where N_{i-1} is the size of the square rendered in the $(i-1)^{th}$ pass, $N_0 = N$, $N_i = N_{i-1}/2$. At every pixel location (i, j), we

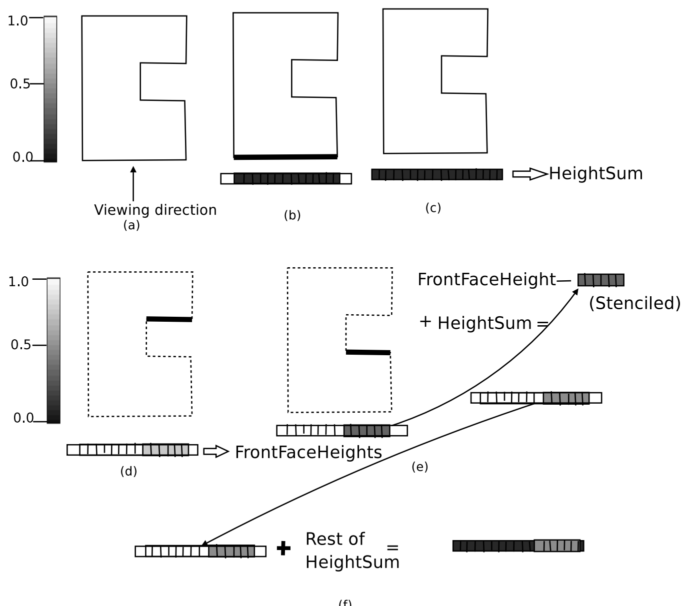

Figure 7. Steps in algorithm (a) Part (b) First layer of front-facing facets (c) Framebuffer after growing the region in b. (d) Second layer of front-facing facets forming supports (e) First layer of back-facing facets is subtracted from *FrontFaceHeights* and added to *HeightSum* (f) Final color buffer after copying the rest of *HeightSum*

use a fragment program that adds values in the texture stored at locations (i, j), $(i+N_n, j)$, $(i+N_n, j+N_n)$ and $(i, j+N_n)$. After $\log N$ passes, the single pixel rendered contains the sum. Figure 8 shows a two dimensional texture and the pixels that are added in a single pass. We found that on our graphics card, using an offset equal to half the size of texture is faster than using an offset of 1. In future, the offset should be selected by testing the relative speed of two approaches on the available platform.

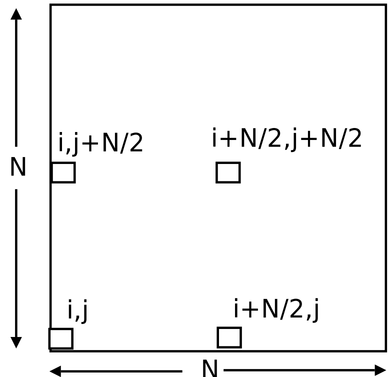

Figure 8. Four pixels are added at a time to reduce the size of a texture to one-fourth of the original size

Copyright © 2006 by ASME

The number of passes required to find the clipping plane is constant and does not depend on the input. The number of passes required for growing the base is equal to d, where d is equal to the maximum of d_x and d_y. The number of passes required for the volume computation is equal to the depth complexity of the part for a given viewing direction. We also require $\log N$ number of passes for adding the values in the frame-buffer where N is the width of frame-buffer in pixels. The total number of passes required for our algorithm is $O(d + nk + \log N)$ where d is the amount of the maximum base offset in pixels, n is the number of facets, and k is the depth complexity. The time required for a pass depends on the number of facets as well as the resolution of framebuffer. Figure 9 shows the effect of resolution on the timings for 4 different parts. For very low resolutions, the algorithm misses a few depth layers due to coarse discretization reducing the total time.

Results

We first tested our algorithm on several parts for which we were able to compute the amount of support material needed analytically because the volume of supports was known. For the sake of comparison, we did not include the volume of the base in these test parts. Table 1 provides the timings and accuracy data for these test examples. All timings were measured on an NVIDIA 6800 GO GPU and a 1.6 GHz Pentium M CPU with 512 MB RAM and 256 MB video RAM on a Mandrake Linux operating system. We used a buffer size of $1024x1024$ for all examples. There are three main sources of errors in our algorithm as follows:

1. Errors due to triangulation: Input to our algorithm is a triangulated boundary representation of the part. If the part consists of curved surfaces, this approximation by planar triangulation introduces errors in volume computation. We postulate that if view-dependent triangulation as discussed in [21] are used to limit the approximation error to less than a pixel, the error due to triangulation will be minimized. The computation of a view-dependent triangulation will increase the running time. Thus, this approach may not be efficient when the user is changing the build direction constantly.
2. Errors due to discretization: Triangles on the boundary representation are discretized into fragments by the GPU. This discretization introduces errors in the computation as the area of the pixels rendered will not be exactly equal to the actual area of the triangles.
3. Floating point error: GPU's use a non-standard single precision floating point format. The volume computation is also affected by the use of single precision instead of double precision.

We further tested our algorithm on a number of parts to determine the speed of the algorithm for complex industrial parts.

Example Parts			
Actual volume	12,000	10,079	83,553,885
GPU volume	11,906	10,020	82,799,800
Error	.8 %	.5 %	.9 %

Table 1. TESTING ACCURACY OF OUT ALGORITHM ON A FEW TEST PARTS

The graph in Figure 10 documents the time required for a number of example parts shown in Figure 12. We also measured the time required for the support material volume computation in Insight, the commercial software for the FDM machine. Note that Insight computes the volume of supports based on the tool paths generated after the part is sliced. Even though this is not a fair comparison as the software first slices the part and computes the tool-paths, this is the only commercially available way a designer can find an estimation of the volume of support material required. Thus, the timings shown in Figure 10 include the time required for slicing the part and computing the tool path. Figure 11 shows the effect of change in the number of triangles on the timings for two different parts. Part 2 had a higher depth complexity than part 1.

Conclusion

In this paper, we presented a GPU-based algorithm for computing the volume of support material needed for manufacturing a part in a given build direction. We have implemented our algorithm and tested it on a number of test parts. Our algorithm is 99% accurate in all the examples that we have tested. The running time varies with the number of facets and the depth complexity, but is less than a second for parts of up to 50,000 facets and depth complexity up to 19. This is one to two orders of magnitude faster that the commercial software, which must slice the part first in order to estimate support volume.

REFERENCES

[1] Otto, K., and Wood, K., 2001. *Product Design : Techniques in reverse engineering and new product development*. Prentice Hall.

[2] Jacobs, P. F., 1993. *Rapid Prototyping and Manufacturing: Fundamentals of stereolithography*. McGraw-Hill Inc, New York, NY.

[3] Beaman, J. J., et al., 1997. *Solid Freeform Fabrication : A*

Copyright © 2006 by ASME

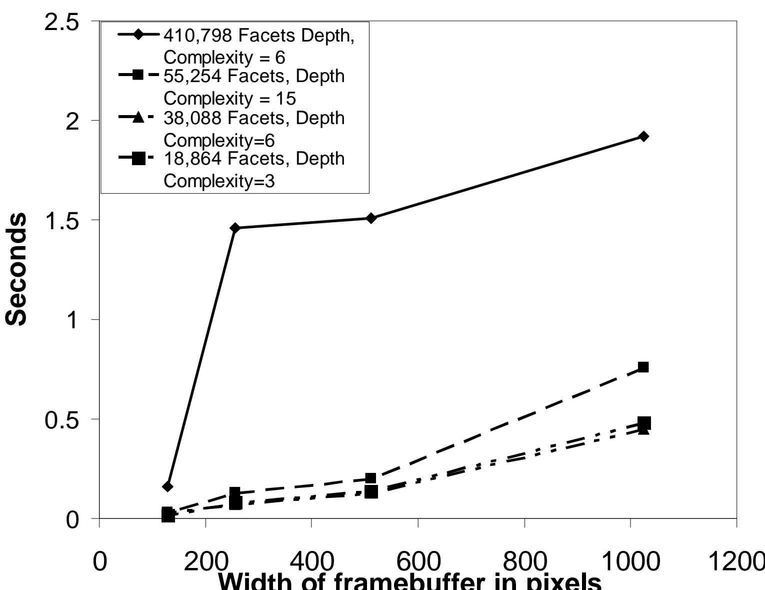

Figure 9. Effect of increase in resolution on the timings

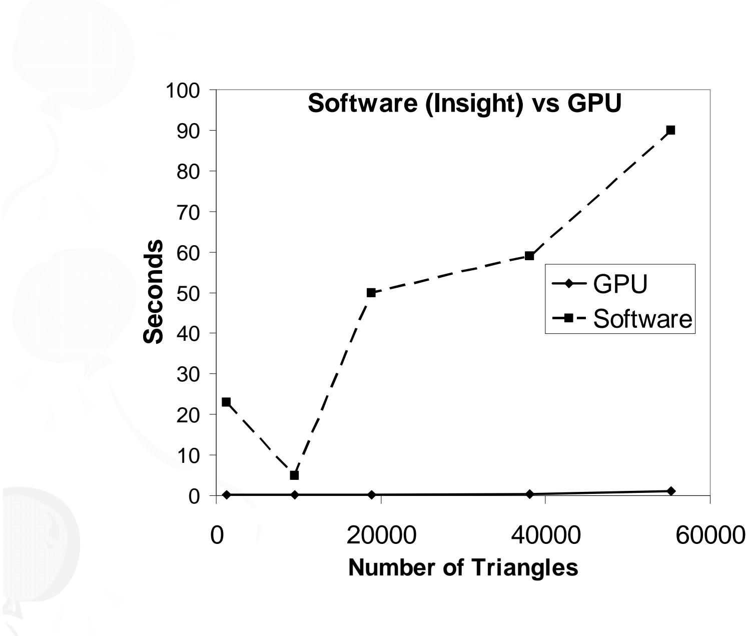

Figure 10. Comparison of our algorithm and Insight software

New Direction in Manufacturing. Kluwer Academic Publishers, Dordrecht.

[4] Thompson, D., and Crawford, R., 1997. "Computational quality measires for evaluation of part orientation in freeform fabrication". *Journal of Manufacturing Systems,* **16**(4), pp. 273–289.

[5] Asberg, B., Blanco, G., Bose, P., Garcia-Lopez, J., Overmars, M., Toussaint, G., Wilfong, G., and Zhu, B., 1997. "Feasibility of design in stereolithography". *Algorithmica,* **19**(1-2), pp. 61–83.

Copyright © 2006 by ASME

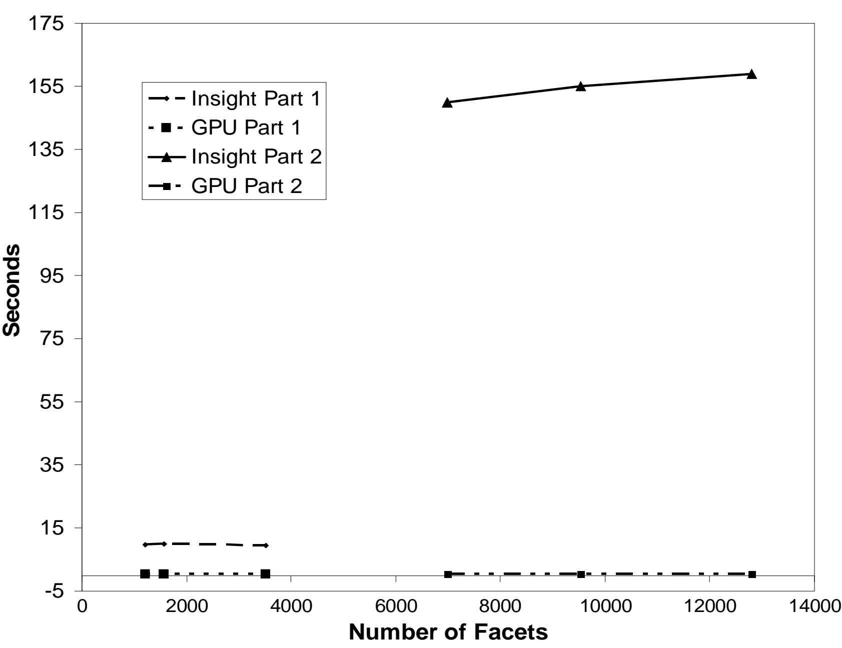

Figure 11. Comparison of our algorithm and Insight software for two parts with different resolution

Example Part	Number of Facets	Volume (mm^3) Orientation 1	Depth Complexity	Time in Hardware (sec)	Time in Software (sec)
	1,252	32,674	2	.21	23
	9,538	82,799,800	6	.16	5
	18,864	15,645,100	10	.19	50
	38,088	509,121	6	.42	59
	55,254	880,998	8	15	90

Figure 12. PARTS USED FOR OUR COMPARISON WITH INSIGHT

[6] Allen, S. W., and Dutta, D., 1995. "Determination and evluation of support structures in layered manufacturing". *Journal of Design and Manufacturing,* **5**, pp. 153–162.

[7] Majhi, J., Janardan, R., Schwerdt, J., and Smid, M., 1999. "Minimizing support structures and trapped area in two-dimensional layered manufacturing". *Computational geometry : Theory and Applications,* **12**(3-4), p. 241.

[8] Majhi, J., Janardan, R., Smid, M., and Schwerdt, J., 1998. "Multi-criteria geometric optimization problems in layered manufacturing". In Proceedings of the fourteenth annual

Copyright © 2006 by ASME

symposium on computational geometry, pp. 19–28.
[9] Agarwal, P., and Desikan, P., 2000. "Approximation algorithms for layered manufacturing". In Proceedings of the eleventh annual ACM-SIAM symposium on discreet algorithms, pp. 528–537.
[10] Cheng, W., Fuh, J. Y. H., Nee, A. Y. C., Wong, Y. S., and Loh, H. T., 1995. "Multi-objective optimization of part-building orientation in stereolithography". *Rapid Prototyping journal,* **1**(4), pp. 12–33.
[11] Lan, P., Chou, S., Chen, L., and Gemmill, D., 1997. "Determination of fabrication orientations for rapid prototyping with stereolithography apparatus". *Computer-Aided Design,* **29**(1), pp. 53–62.
[12] Frank, D., and Fadel, G., 1995. "Expert system-based selection of the preferred direction of build for rapid prototyping processes". *Journal of Intelligent Manufacturing,* **6**(5), pp. 339–345.
[13] Saito, T., and Takahashi, T., 1991. "NC machining with G-buffer method". *SIGGRAPH 91,* **25**(4), July, pp. 207–16.
[14] Balasubramaniam, M., Laxmiprasad, P., Sarma, S., and Shaikh, Z., 2000. "Generating 5-axis NC roughing paths directly from a tessellated representation". *Computer-Aided Design,* **32**(4), April, pp. 261–77.
[15] Spitz, S., Spyridi, A., and Requicha, A., 1999. "Accessibility analysis for planning of dimensional inspection with coordinate measuring machines". *IEEE Transactions on Robotics and Automation*, pp. 714–27.
[16] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A. E., and Purcell, T. J., 2005. "A survey of general-purpose computation on graphics hardware". In Eurographics 2005, State of the Art Reports, pp. 21–51.
[17] Khardekar, R., Burton, G., and McMains, S., 2005. "Finding feasible mold parting directions using graphics hardware". In Proceedings of the 2005 ACM Symposium on Solid and Physical Modeling, pp. 233–243.
[18] Foley, van Dam, Feiner, and Hughes, 1997. *Computer graphucs : Principles and practice*. Addison-Wesley, Reading, MA.
[19] Everitt, C., 2002. Interactive order-independent transparency. Tech. rep., NVIDIA corporation. See also URL http://developer.nvidia.com.
[20] Kruger, J., and Westermann, R., 2003. "Linear algebra operator for GPU implementation of numerical algorithms". *ACM Transactions on Graphics,* **22**(3), pp. 908–916.
[21] Khardekar, R., and Thompson, D., 2003. "Rendering higher order finite element surfaces in hardware". In Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia, pp. 211–219.

Copyright © 2006 by ASME