Real-Time Visualization Of Bead Based Additive Manufacturing Toolpaths using Implicit Boundary Representations

Shane Peelar[1] , R. Jill Urbanic[2] , Robert W. Hedrick[3] Luis Rueda[4]

[1]University of Windsor, peelar@uwindsor.ca
[2]University of Windsor, jurbanic@uwindsor.ca
[3]CAMufacturing Solutions Inc., bob.hedrick@camufacturing.com
[4]University of Windsor, lrueda@uwindsor.ca

Corresponding author: R. Jill Urbanic, jurbanic@uwindsor.ca

ABSTRACT

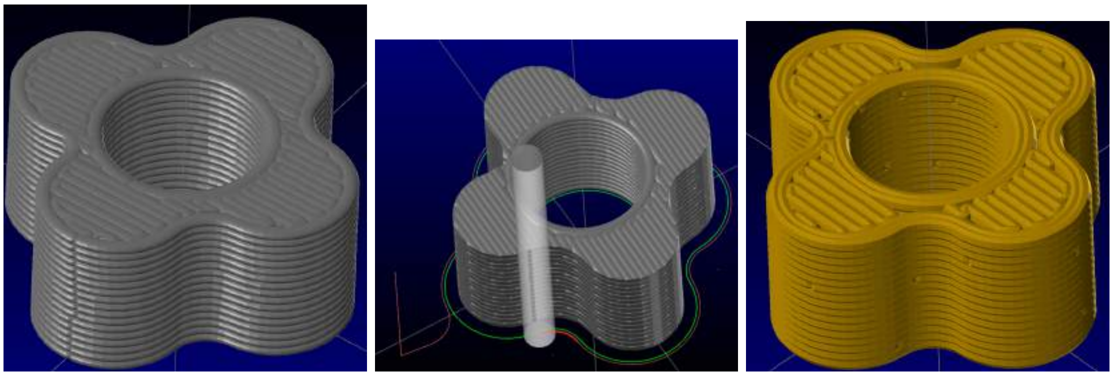
Constructing a boundary representation for an AM part is challenging due to the large number of CSG operations that need to be performed. To tackle the problem, we begin with a review of different numeric representations and their suitability for solving geometric problems. We then review the state of the art in explicit boundary representations, exploring why they are unsuitable for generating AM part models. Finally, we describe a hybrid implicit-explicit boundary representation which addresses the scalability and precision needs of AM part model generation. This merits of this approach are illustrated using several case studies.

Keywords: visualization, additive manufacturing, part model generation, simulation, constructive solid geometry, boundary representations, implicit surfaces
DOI: https://doi.org/10.14733/cadaps.2019.904-922

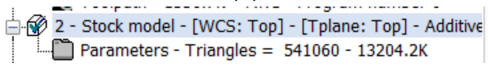
1 INTRODUCTION

Additive Manufacturing (AM) poses unique challenges in part model generation and visualization compared to subtractive methods. Building a part using AM involves first "slicing" it into layers, and then rasterizing each layer using a fill technique. Here, the toolpath is the path that is used in depositing material to form the part. The part model essentially serves as a boundary representation for the part being manufactured using the toolpath, process parameters and bead geometry. It is important to generate an accurate part model for visualization purposes, void detection, and further processing.

As an example, consider a simple tool path for a clover leaf (63.5 mm x 63.5 mm x 25.4 mm) shown in Fig. 1, where a 5.8 mm nozzle is employed for simulating a fused filament deposition process, with an allowable maximum 50% overlap for this operation. The slice height is 1.5 mm. There is a raster scan fill with a 90° rotation between layers. The stop and start points for the machining stock model are aligned. When using one boundary layer, there are some voids between the layers, typical of fused deposition filament processes, and some voids at the boundary-raster fill transition regions. Using this model as machining stock model, the interior voids are exposed after a machining



(a) (b) (c)



2 - Stock model - [WCS: Top] - [Tplane: Top] - Additive
Parameters - Triangles = 541060 - 13204.2K

(d)

Figure 1: Clover leaf pattern *.STL (63.5 mm x 63.5 mm x 25.4 mm): (a) stock model with 50% overlap, one boundary contour, (b) contour machining operation exposing the internal voids associated with a fused filament deposition process, (c) a two boundary curve model, with increased internal voids near the inner boundar, (d) stock model characteristics.

operation (Fig. 1(b)). Changing the number of boundary curves will reduce the exposed interior voids, but this will introduce additional no-fill conditions within a layer (Fig. 1(d)).

Boundary representations can be categorized into implicit and explicit approaches. Explicit boundary representations are commonly used to represent part models [24]. However, due to exactness requirements, these representations have difficulty scaling with AM. Each toolpath necessarily has one or more explicit boundaries that in turn represent the bead geometry along it. The explicit boundary representation of the entire part model must be computed from the union of all toolpath boundaries. As a result, the number of CSG unions can get into the hundreds of thousands quickly, taking unreasonable amounts of time to compute even on high end workstations. Furthermore, the number of facets of the final representation can grow unreasonably large, resulting in part models that consume gigabytes of disk space (Tab. 1), even for a relatively small component [25] (Fig. 3). Additional complexity is introduced when extracting void data, and the bead overlap regions. Both influence the functional characteristics of the component - the void regions are crack propagation sites, and the component's hardness, residual stresses, and distortion are influenced by the bead overlap conditions. Another interesting issue to consider is that there is material mixing occurring at the boundaries (side to side and with the layer stacking), so the boundaries have transition regions (Fig. 2)

Therefore, alternative methods are required represent the bead geometry. The objective of this research is to develop a scalable solution that can model the geometry of a large AM built component constructed from many small beads quickly at various levels of resolution, and provide a basis for simulating physical performance characteristics related to the void and transition areas.

An alternative to explicit boundary representations are implicit boundary representations. Whereas an explicit boundary representation uses a parametric, polygonal, or spatial partitioning scheme to define the boundaries of an object, an implicit boundary representation uses a scalar field. The boundary of the object in the field is the set of points that correspond to a specific isovalue. This is commonly called an implicit surface or *isosurface*. To address the shortcomings of explicit boundary representation approaches, we relax the requirement of having an exact result and propose a hybrid explict-implicit boundary representation approach as a scalable alternative for generating AM part models. In our approach, the individual toolpath boundaries are represented explicitly, however the resulting part-model boundary is computed implicitly.

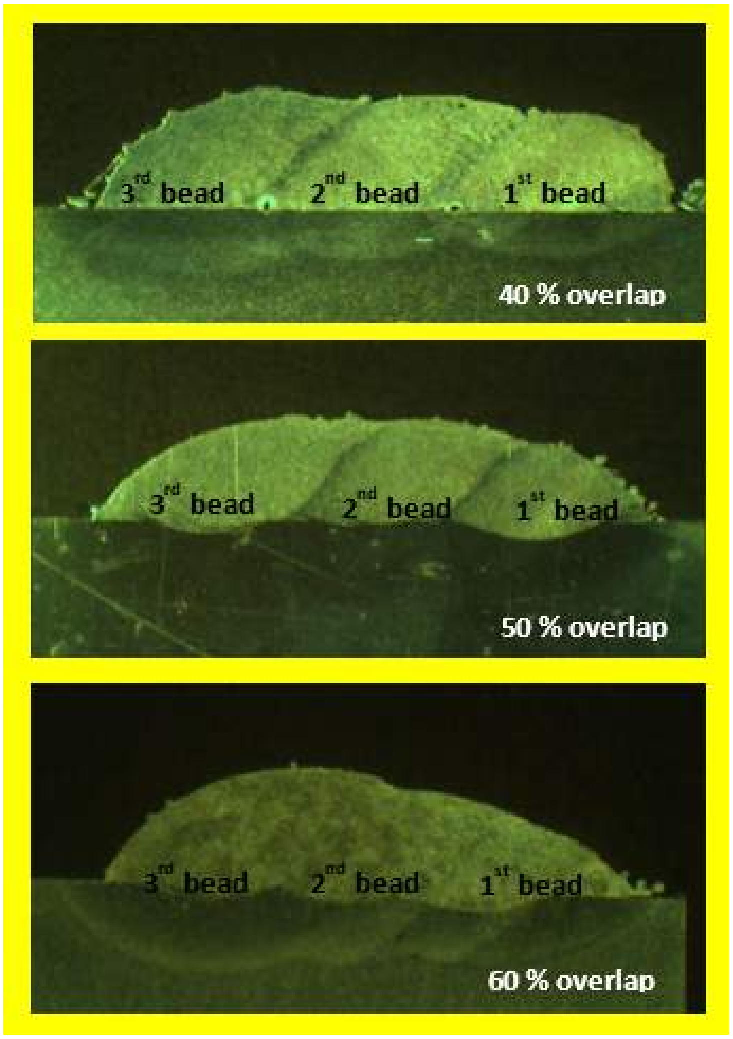


Figure 2: Bead overlap conditions illustrating the resulting geometry. [19]

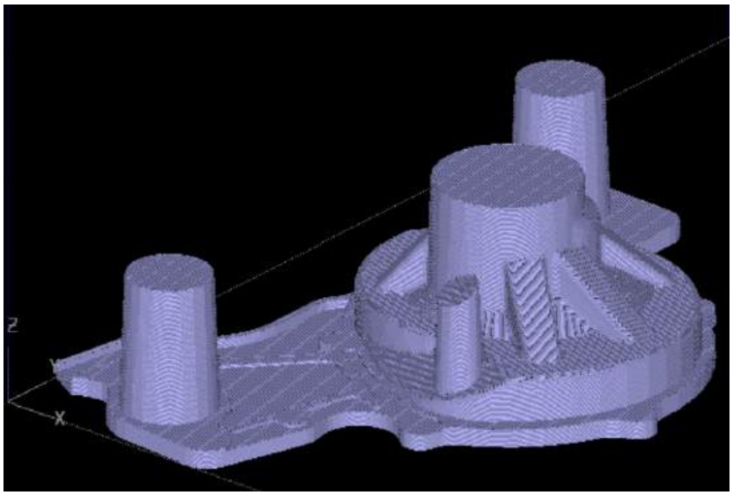


Figure 3: Water pump casting pattern *.STL (174 x 236 x 62 mm @ 0.25 mm slice height): 1,018,892 KB. [25]

IR Size	*Part model size (STL)*
2,509 KB	16,446 KB
6,547 KB	176,202 KB
8,256 KB	210,345 KB
93,387 KB	2,242,654 KB

Table 1: Part model sizes in relation to their toolpath Intermediate Representation (IR) size using explicit boundary representation.

2 NUMERIC REPRESENTATION

The choice of numeric representation used in geometric computations has a direct effect on performance characteristics and robustness of the result, regardless of which boundary representation is used. In this section, we compare and contrast common numeric representations and their associated arithmetic with respect to one another. Note that we distinguish between the representations themselves and *instances* of those representations.

The *width* of an instance is the number of bits it requires to represent it. If a representation mandates that all instances have a constant width, it said to have *fixed-width*. Otherwise, it is said to have *arbitrary-width*. For example, a IEEE-754 double precision floating point number always has a width of 64 bits, and the single precision variant always has a width of 32 bits [1]. These are both examples of fixed-width representations.

The *precision* of an instance is the distance between its value and the smallest next representable value in that instance's representation, or if that is not defined, the distance between its value and the largest previous representable value in that instance's representation, or if that is not defined, 0.

If all instances of a representation are guaranteed to have the same precision p, then if $p > 0$ it is said to have *fixed-precision* and if $p = 0$ it is said to have *arbitrary-precision*. Otherwise, it is said to have *bounded-precision*.

The *range* of a representation is the distance between the maximum value and the minimum value in that representation. In a fixed-width representation, precision and range are proportional. In an arbitrary-width representation, there are no restrictions on precision and range aside from available memory.

We can broadly describe existing categories of arithmetic and their associated numeric representations using these definitions.

2.1 Fixed-point Arithmetic

In *fixed-point arithmetic*, sometimes called *fixed-precision arithmetic*, the representation has fixed-precision, however it may be either fixed-width or arbitrary-width. These representations need only store an integer m, since the precision p is a known constant. The actual value represented by a fixed-point instance is mp. In other words, p acts as a scaling factor for the integer stored in the instance, and m can be thought of as a pre-scaled quantity.

An advantage of fixed-point arithmetic is that it requires no specialized hardware, and can generally be performed directly on the computer's standard integer Arithmetic and Logic Unit (ALU). If $p = 1$, then the instance is simply an integer.

Fixed-point numbers can be visualized on a number line where consecutive points are separated by a line segment of length p. Care needs to be taken if instances of different precisions are used together in the same arithmetic expression. In this case, one of the instances will need conversion to a new fixed point number with the precision of the other instance, which may overflow or underflow if it is computed at insufficient width. Visually, the number lines are "overlaid" on top of one another to form a new number line that contains all possible values of both previous instances. This is similar to adding two rational numbers where one of the denominators is a multiple of the other. Indeed, fixed-point numbers are a subset of the rationals.

Fixed-point numbers can be convenient for representing quantities such as currency, where the required precision is known in advance. Furthermore, comparing equality of fixed point numbers is well defined without requiring a distance threshold.

2.2 Floating-point Arithmetic

In *floating-point arithmetic*, the representation has a bounded precision and therefore fixed width. Although many floating point representations exist, the most common format used in CAD/CAM software is IEEE-754 floating point, which is commonly 16, 32, or 64 bits wide. Special instances include `NaN`, $+$`Inf`, and $-$`Inf`, which are used when the result of an arithmetic expression would be undefined or would overflow, for example when dividing by 0. In addition to this, an instance may be *subnormal* or *denormal* for extra precision near 0.

Floating point numbers can be visualized on a number line where the points are tightly clustered around the origin and grow exponentially distant as they move towards positive and negative infinity. Fundamentally, mantissa bits are traded for exponent bits which determine the precision and range of the instance. When floating point instances of different precisions are used together in an arithmetic expression, the result is subject to rounding.

Care must be taken to maintain the significant bits in the mantissa during certain operations, such as subtraction. A problem called *catastrophic cancellation* can occur if a subtraction between two *nearly equal* floating point numbers is performed, where the number of significant bits in the result can be heavily reduced. Algorithms must be designed such that they resist these effects. IEEE-754 floating point arithmetic is also non-associative, which makes it difficult to use to solve geometric problems reliably [5, 13, 21]. The non-associativity of IEEE-754 floating point can also make it difficult to compare instances against each other, particularly for equality tests. Base-2 floating point formats, such as IEEE-754, are also unable to accurately represent many common decimal numbers, such as 0.1. Base-10 floating point formats do exist that address this need, however they have not gained mainstream adoption. On the other hand, a fixed-point representation with the correct precision would have no trouble representing this number.

Floating-point representations attempt to be useful in a variety of contexts where the precision and range requirements may not be known ahead of time. If a floating point representation is found to not have enough precision or range at a later time, it can sometimes be substituted for a wider representation to achieve an acceptable result. Floating Point Units (FPUs) are very common in consumer CPUs and are becoming increasingly common in the embedded device space. In systems with FPUs, performance is comparable with fixed-point arithmetic. This, coupled with the fact that floating point numbers behave well enough under many circumstances has led to widespread adoption, even in domains where they may not be appropriate.

2.2.1 Posit arithmetic

A relatively new floating point format, called the Posit [9], offers some interesting tradeoffs with respect to IEEE-754 floating point. One benefit of the Posit with respect to IEEE-754 is that it is associative, and may therefore be suitable

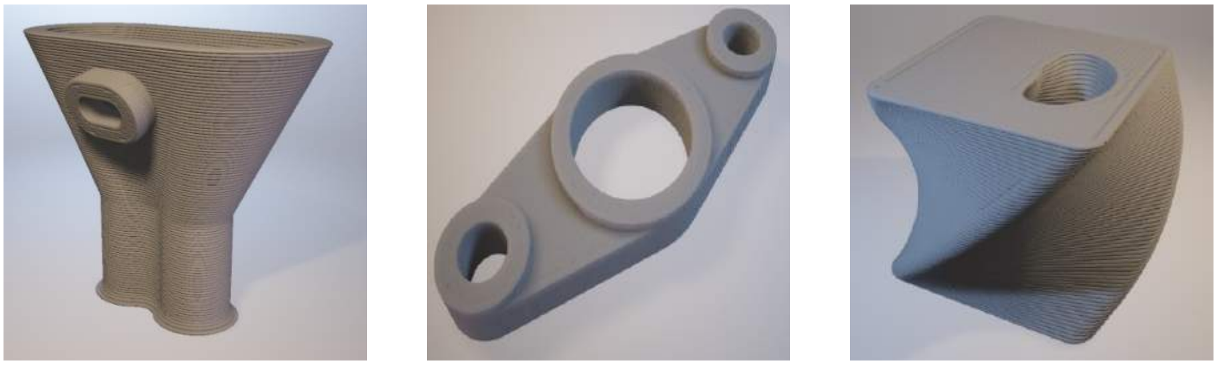


Figure 4: Explicit boundary representations for three part models. Left to right: multiplane.stl, widget.stl, and IRAP.stl.

for solving certain geometric problems. Only a reference software implementation exists currently, however it is possible to implement in a Field Programmable Gate Array (FPGA) for hardware acceleration.

2.3 Arbitrary-precision Arithmetic

Arbitrary-precision arithmetic is an umbrella term for arithmetic using arbitrary-width representations, which may have either fixed or arbitrary precision. Assuming a computer has infinite memory, an arbitrary-width arbitrary-precision representation can exactly represent all rational numbers and any calculations involving them. Indeed, these are commonly referred to as *exact-precision* representations. There are a number of libraries available that provide these representations, such as GNU Multiple Precision Arithmetic (GMP) [17] and Boost.Multiprecision [15].

Arbitrary-precision representations have a critical advantage over fixed or bounded precision approaches in that results are able to be calculated without rounding. In addition to this, these approaches are associative. This property is highly desirable for geometric calculations, with some libraries such as CGAL exclusively using arbitrary-precision arithmetic for solving geometric problems [5].

A significant drawback of arbitrary precision arithmetic is that often times the values stored will not fit into the native registers on the computer they are running on. The library itself must simulate an arbitrary-precision arithmetic and logic unit (ALU) that is in turn built upon the fixed-precision ALU available on the computer. Furthermore, many memory allocations must be performed since the width is allowed to grow as much as necessary, forcing heap allocation rather than stack allocation. Many operations on arbitrary-precision representations often require normalization before they can be performed as well, which also can affect performance.

2.4 Suitability for Solving Geometric Problems

In general, representations that are associative will be more robust than those that are not for solving geometric problems. The correct representation to use depends on robustness, performance, and accuracy requirements. Unfortunately, no representation exists that can simultaneously address all factors. IEEE-754 floating point, while widely available, can produce surprising results for even well formed input. Fixed-point arithmetic may not yield accurate enough results for intersection operations, but will behave predictably. Arbitrary precision may address both needs, however could increase run-time by several orders of magnitude. Posits have yet to be empirically tested, with only a reference implementation available, however may improve on the state of the art in bounded-precision representations for certain classes of geometric problems.
Our approach for part-model generation relaxes the requirement of computing exact unions of the explicit boundaries of AM toolpaths. This allows us to avoid using arbitrary-precision arithmetic in favour of the existing floating point facilities on the host system.

3 EXPLICIT BOUNDARY REPRESENTATIONS

Explicit boundary representations attempt to explicitly model the boundaries of geometry using parametric equations (such as in the case of B-Rep [18]), polygonal meshes, or a spatial partitioning scheme. Commonly, the object is defined by a tree of CSG operations starting on an initial primitive. The primary benefit of an explicit boundary representation (Fig. 4) is that the result can be computed exactly without rounding using arbitrary-precision arithmetic, improving robustness. Indeed, most state of the art research in this area uses arbitrary-precision arithmetic for this reason [5, 6]. While it is possible to use floating-point arithmetic with these approaches, considerable care must be taken in order to prevent degenerate output, even with well-formed input. Fixed-point arithmetic is an alternative that can work when the precision and range requirements are known in advance, and exact solutions are not required. Arbitrary-precision arithmetic does not suffer these drawbacks, however it carries a performance penalty, and can prevent the boundary representation from scaling to objects that are defined by large numbers of CSG operations, as is the case in AM part model generation (Tab. 2).

Number of Meshes	*Processing time (approx.)*
10	20 seconds
100	4 minutes
1,000	40 minutes
10,000	6 hours

Table 2: Processing time required for the CSG union of explicit triangle-mesh boundaries using arbitrary-precision arithmetic.

3.1 Parametric Approaches

Parametric approaches, commonly called *B-Reps*, attempt to model geometry by defining it through a tree of parametric equations [18]. The surfaces of primitives are treated as manifold surfaces, and CSG operations are defined in terms of operations on these surfaces. For example, if a boolean difference between a cube and a cylinder were performed, a circular hole may be “cut” from the facets of the cube affected by the operation. One advantage of these approaches is that geometry need not be converted to a piecewise-planar representation (such as a triangle mesh) in order to be operated upon, making these approaches very flexible. One problem with parametric approaches is that they have difficulty scaling up to hundreds of thousands of CSG operations, particularly because they attempt to compute an exact parametric description of the boundary, even if floating-point arithmetic is used.

3.2 Spatial Partitioning Approaches

A spatial partitioning approach hierarchically partitions space into convex regions that are “inside” and “outside” the object using hyperplanes. They may only represent the boundaries of objects whose facets are piecewise-planar. Of these, the most general approach is the Binary Space Partition (BSP) [22], which recursively partitions space into convex *half-spaces* using a binary tree and a hyperplane stored at each node of the tree. BSPs have been widely used to accelerate real-time rendering and retrieval operations on geometry. There could be many possible valid BSP-trees that represent the boundary of a given object. They perform best when their underlying binary tree is balanced. Unfortunately, generating optimal BSPs is an intractable problem [16]. Instead, heuristics are used during the construction of the tree to generate a result that is a compromise between optimality and generation speed.

CSG operations can be performed on two objects by constructing a BSP for each of them independently and then merging these trees, recursively pruning and merging branches as necessary throughout the operation. Once the final BSP is computed, the boundary of the object can be extracted into a triangle mesh since only the clipped polygons will remain.

One drawback of this approach is that it may result in more triangles than necessary to represent the boundary of the object. For example, consider Fig. 5, where the partitioning hierarchy splits a triangle across multiple hyperplane

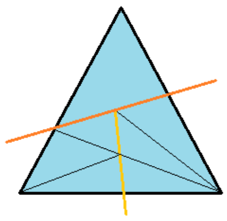


Figure 5: A triangle that has been cut by two hyperplanes during the BSP generation of a piecewise-planar object. The orange line represents the parent hyperplane, and the light orange line represents one of its child hyperplanes. The thin black lines delineate the new triangles that were formed during BSP generation.

boundaries. The hyperplanes are chosen randomly from other polygons in the object, which are not depicted. If all sub-triangles are kept, the resulting triangulation is clearly sub-optimal. This can have a drastic effect on representation size and performance. To make matters worse, re-merging the co-planar neighbouring sub-triangles is an intractable problem, and may even be impossible if floating-point arithmetic is used, due to the inherent difficulties in comparing equality of those representations.

Commonly, floating-point arithmetic is used in BSP generation and representation. This could result in degenerate outputs even for non-degenerate inputs, particularly during the triangle splitting process. Extracted boundaries may be non-manifold due to self-intersecting geometry, for example. Robustness can be improved using fixed-precision arithmetic if the precision and range requirements are known, or if those are not known, arbitrary-precision arithmetic can be used instead at a performance cost.

3.3 Mesh-oriented Approaches

Mesh-oriented boundary representations do not attempt to partition space into "inside" and "outside" regions directly. Instead, they depend on a metric which measures the degree of "insideness" a point has relative to an object. A number of metrics exist, such as Signed Distance Functions (SDFs) [7], and more recently the Generalized Winding Number (GWN) [10]. These approaches are restricted to objects that are piecewise-planar, and the boundary representation itself is commonly a set of vertices with an associated set of triangles.

An advantage of these approaches is that input triangle meshes can generally be directly used in performing CSG operations without converting them to new representations. First, the meshes are *resolved* against one another, a process that creates additional vertices and triangles on both meshes such that any triangle intersections are resolved into new sub-triangles that are non-intersecting. This is similar to the hyperplane clipping process used in BSP generation. Next, a metric is evaluated on each vertex to decide whether to keep it or prune it from the geometry, which completes the CSG operation. Example libraries that follow this approach include CGAL [6], Cork [3], and libigl [11, 27] which appears to be the state of the art in this space.

Mesh-oriented approaches can be sensitive to degenerate inputs, for example non-manifold triangle meshes. Floating-point arithmetic performs poorly when used during the mesh resolution phase of these approaches. This is because exact intersection points are needed between triangles, and small quantities may interact with large quantities when solving those equations. Specifically, a distance metric that should be zero for a particular problem may be evaluated in floating-point as a near-zero value instead, which could cause an intersection test to fail or succeed unintentionally. Fixed-point arithmetic is also problematic, since the precision requirements can vary greatly even in small datasets. Arbitrary-precision arithmetic is generally the only viable option for mesh resolution. Regardless of which numeric representation is used, the results can be converted back to another representation for metric evaluation.

3.3.1 The Generalized Winding Number

The Generalized Winding Number [10] is a generalization of the winding number [2] (Fig. 6) to three dimensions. Briefly, it provides a measure of "insideness" of a point with respect to an object. For a watertight mesh, the GWN of every point inside the mesh is 1, and the GWN of every point outside the mesh is 0. Where the GWN really shines is in handling degenerate meshes: meshes that contain self-intersecting edges, have facets that are not oriented correctly, or meshes that are not watertight. In these cases, the GWN provides a smooth transition from regions of the mesh that are "inside" to ones that are "outside", providing robustness where an SDF would have trouble. We use a special case of the GWN [27] which can be used to efficiently compute the GWN for triangle-based meshes.

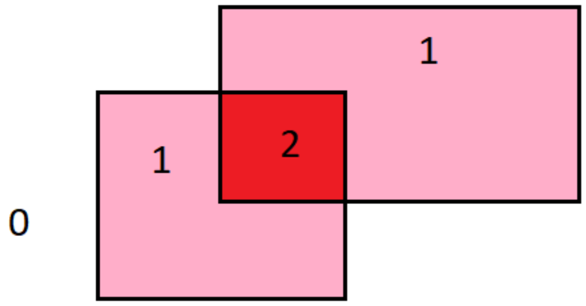


Figure 6: Winding numbers of points with respect to the two overlapping boxes. Notice that the overlapping region still has a positive winding number.

3.4 Suitability for AM Part Model Generation

A common theme with explict-boundary approaches is that they attempt to exactly model the boundary of an object that is defined through a tree of CSG operations, forcing them to use arbitrary-precision arithmetic to achieve generalizable results. Fortunately, exact results are not required for part model generation. An imprecise result within a confidence threshold is good enough for most visualization tasks, and even some hybrid manufacturing (machining operations performed on an additive manufactured model) tasks, as illustrated in Fig. 1(b). One might be tempted to use a floating point format in combination with an explicit boundary technique to achieve this; however, floating point arithmetic is non-associative and hence not suitable for performing geometric operations [13, 21]. Even non-degenerate input can result in degenerate output due to a sign flip or precision loss from rounding. Fixed-point arithmetic is also unsuitable since the range and precision requirements of AM part model generation can vary dramatically. Instead, we turn to an implicit boundary representation to address these problems.

4 IMPLICIT BOUNDARY REPRESENTATIONS

An *Implicit Boundary Representation* is any boundary representation that does views geometry as being embedded in a field $F : \mathbb{R}^3 \rightarrow \mathbb{R}$ where, without loss of generality, each point has a positive value if it is "inside" the object, and a non-positive value otherwise. Locations in the field that have a value of 0, called *isovertices*, therefore represent the boundary of the embedded geometry.

CSG operations are natural to define using an arithmetic operation *op* to combine points from the two input fields F_1 and F_2 into a new field F':

$$F'(x) = F_1(x) \textit{ op } F_2(x)$$

For example, a CSG union could be implemented using addition, a CSG difference could be implemented using subtraction, and a CSG intersection could be implemented using multiplication. The field can be represented using an analytic function or a voxel approximation. In the case of a voxel grid, interpolation is used to compute values of the field between samples. Note that these CSG operations do not require arithmetic associativity. Consider the possible ways to compute a 3-way CSG union, for example. Although the end results could vary due to non-associativity of the underlying arithmetic, the end results will be similar enough that the representations will not differ in a meaningful way. In particular, the signs will be preserved, and the magnitudes are likely to differ only in rounding. In an explicit boundary representation, this could easily result in degenerate output. Implicit boundary representations can therefore be soundly represented and operated on using IEEE-754 floating-point arithmetic.

4.1 Voxel Representation

The resolution of the geometry embedded in a field represented using voxels depends on the sampling resolution of the voxel grid. If the voxels are uniformly spaced apart in all dimensions then it is called a *dense* voxelization. Otherwise, it is called *sparse* voxelization.

Dense voxelizations sample the field uniformly in a grid-like fashion. This is more amenable to direct implementation, however can suffer from scalability problems with larger objects. It has the benefit of being readily computed on a Graphics Processing Unit (GPU), using the built-in trilinear interpolation hardware within it to interpolate within the already obtained samples for further processing and visualization. The dense model is represented as a multidimensional array of samples. It can be thought of as a "brute force" approach to sampling data – although one that can outperform a sparse approach in some cases with a powerful GPU.

Sparse voxelizations sample the field in dynamic intervals. An example of these is the Sparse Voxel Octree (SVO), which adaptively samples the implicit function. An SVO is essentially a hierarchical space-partitioning tree of samples with associated metadata. The original field can be approximated from the SVO using a form of interpolation. Sparse voxelizations are more amenable to being generated on a CPU, given the branching logic and sparse memory accesses required. They have the advantage of being more scalable than dense approaches due to being more compact. Commonly, Hermite data is recorded in the SVO to help preserve sharp features [20].

Both approaches have diminishing returns with the desired resolution chosen. For purely random data, a sparse approach will perform worse than a dense approach, since the SVO will become excessively large compared to an equivalent array in a dense approach. However, sparse approaches have the main benefit of performing fewer samples overall compared to dense approaches with non-pathological input. A dense approach with sufficient resolution can achieve acceptable results using linear interpolation even when the isosurface is non-linear. Sparse approaches must be more careful with the choice interpolation function to achieve reasonable results [8].

4.2 Isosurface Extraction

A need arises frequently with implicit boundary representations where the boundary, or *isosurface*, needs to be extracted into a polygonal mesh for various reasons, such as visualization and compatibility with existing software packages. The process of approximating this explicit boundary using a mesh is called *isosurface extraction*.

The field analogue of a normal vector is the gradient vector, which is a higher order difference of how the field changes along each component at a given point. In other words, it is the generalization of a derivative. At the boundaries of the geometry (where the field is 0), this models the normal vector in the explicit boundary sense, pointing "outward" from the object. Gradients play an important role in extracting high quality meshes that approximate isosurfaces. If the analytical gradient function is not known, it can be approximated using finite differences [8]. We summarize some existing isosurface extraction methods below. As with numeric representation, the right algorithm to choose depends on the balance of performance and accuracy required.

4.2.1 Marching Cubes

The Marching Cubes algorithm [14] extracts an isosurface from a field by sampling the field at fixed intervals in a grid-like fashion. The values at the corners of each "cube" in the grid are compared against a given isovalue threshold. Each of the 2^8 possible states of a cube have an associated polygon, which is then used to construct a shell that approximates the isosurface over the entire volume. Resolution can be increased by decreasing the cube size. The original Marching Cubes algorithm suffers from ambiguity in certain cases which have been addressed in subsequent work [23], and due to its use of linear interpolation for locating isovertices, it can produce visual artifacts when used with a non-linear implicit function. Marching Cubes may be used with both dense and sparse voxelizations.

4.2.2 Dual Contouring

Adaptive Dual Contouring [12] improves on Marching Cubes by using a Hermite-tagged Sparse Voxel Octree (SVO) to locate the boundaries of the isosurface. Notably, it requires the gradient of the field. The edges of each voxel are "tagged" with the gradient of the intersecting isosurface along the voxel edges, and an isovertex is added to each leaf node by minimizing a quadratic error function. The resulting SVO is then used to construct a smooth surface over the discovered boundaries. Resolution can be improved by increasing the depth of the SVO. The approach can be used with both dense and sparse voxelizations.

4.2.3 Manifold Dual Contouring

Manifold Dual Contouring [20] improves on Dual Contouring by ensuring the extracted mesh is manifold, a desirable property in many applications.

4.2.4 Accurate Isosurface Interpolation with Hermite Data

Accurate Isosurface Interpolation with Hermite Data [8] appears to be the state of the art in isosurface extraction. It improves on previous techniques by using arbitrary interpolation methods to improve isovertex placement along voxel edges, producing more accurate approximations of geometry embedded in the field.

4.3 Suitability for Representing an AM Part Model Explicit Boundary

These methods are all highly suitable for representing the explicit boundaries of AM part models. They can all be performed using non-associative arithmetic. If the implicit function is formed using the GWN, one can take advantage of the technique described in [8] and use harmonic interpolation for a more accurate reproduction of the explicit boundary.

5 HYBRID BOUNDARY REPRESENTATION FOR AM PART MODEL GENERATION

Our approach enables the bead geometry of AM toolpaths to be defined explicitly using a swept cross section, which is convenient for designers. The cross-section is defined as a piecewise-linear curve, however could be defined parametrically in future work. Each toolpath may have one or more explicit boundaries, called *contours*. In addition, a contour may have “end-caps”. These contours are used to generate *.STL files directly, which may involve writing each layer out to a separate file.

5.1 X3D Intermediate Representation

The X3D format [4] has been used previously [24] in part model generation in AM. The contours are mapped directly to `Extrusion` elements in the X3D specification. Each end of a contour could optionally have “end-caps”, which are also represented by `Extrusion` elements. A contour could therefore have either 1 or 3 extrusions associated with it. The primary reason for creating separate contours from a toolpath is to avoid sharp corners in the result, which would not accurately represent the deposition process. Since contours have end-caps, these moves are “rounded off” and retain a more realistic appearance.

An `Extrusion` element in the X3D specification is essentially a cross-section that is swept along a spine, with optional scaling and orientation transformations that can be performed at each spine point. The cross-section is defined as a piecewise-linear curve [4].

5.2 Improved Intermediate Representation

We improve upon the results of [24] by fusing the end-caps of a contour directly into the main `Extrusion` element. New spine points are created beyond the ends of the original element at fixed intervals, decreasing the scale parameter at each point to approximate the original cap shape in [24]. The resolution and the interval size between the spine points are inversely proportional and can be configured as desired. This reduces the total number of meshes significantly and retains a similar visual appearance.

5.3 Implicit Boundary Representation

The Boolean union of these extrusions are not directly computed as they would be in an explicit boundary approach in order to form the part model. Instead, we sample the GWN field induced by all explicit boundaries of all toolpaths using a dense voxelization approach where the resolution is chosen in advance.

The dense voxel approach allows us to use a GPU to accelerate all calculations. We have chosen to use CUDA in our approach, however it should be possible to use OpenCL to the same effect. We enabled full compiler optimizations in CUDA, including the "fast-math" option which uses fast approximations for many common operations, including the square root and division. Briefly, we use the large vector processing capabilities of the GPU to process one GWN sample per SIMD lane in blocks of 256 voxels (one sample per voxel) until the entire voxel space is exhausted. Once the voxelization is obtained, trilinear interpolation is used for all further GWN queries. This implicit boundary representation can then be directly visualized on a GPU using ray-casting, and in the future it may also be used to simulate additional hybrid machining operations.

Although we use the GWN in our approach, any function which measures "insideness" could be used as well, including an SDF [26, 7] (Fig. 7). We chose to use the GWN as it is faster to compute than SDFs for arbitrary geometry, and is well defined even when input is non-manifold. However, an SDF offers an interesting advantage where the boundaries of a part can be expanded or shrunk by modifying the isovalue during isosurface extraction. This is trickier to do with the GWN since a watertight mesh will have a piecewise-constant GWN field (it is not smooth as an SDF would be).

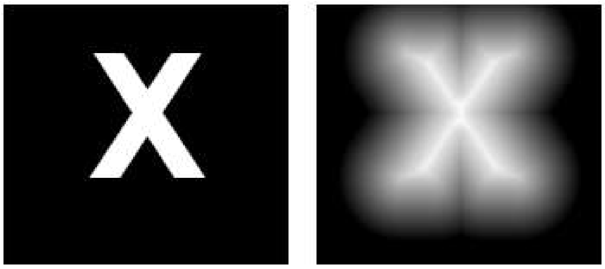


Figure 7: Boundary representations of the character 'X'. Left: explicit representation, Right: Implicit representation using a Euclidean SDF [26].

5.4 Visualization

The implicit boundary representation can be directly visualized on a GPU using well known volume rendering techniques without constructing an explicit boundary. In the images presented, we visualized the three part models shown in (Fig. 4) using ray-casting their voxelized implicit boundary representations. Each image (Fig. 8) (Fig. 9) was taken using a voxel grid at a resolution of 128^3 voxels, or 128 voxels per dimension.

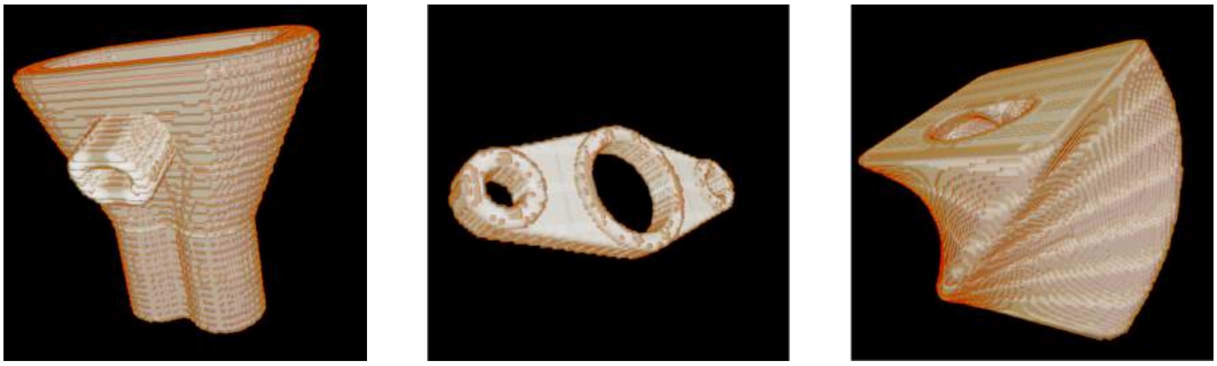


Figure 8: A solid visualization of the implicit boundary representations for three part models. Left to right: multi-plane.stl, widget.stl, and IRAP.stl.

In addition to these opaque models, it is trivial to make both inside and outside regions of the model transparent enabling easy viewing of part model boundaries (Fig. 9):

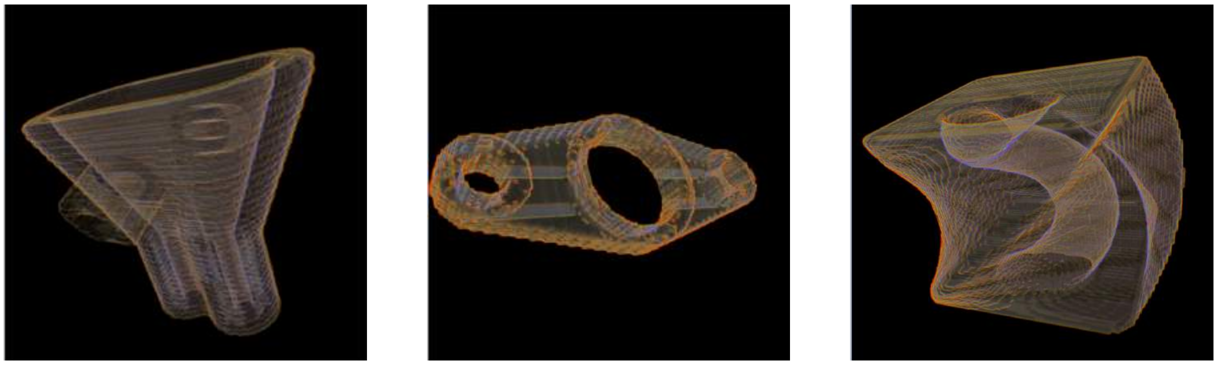


Figure 9: Boundary visualization of the implicit boundary representations for three part models. Left to right: multi-plane.stl, widget.stl, and IRAP.stl.

This can be accomplished by simply modifying the transfer function in the volume renderer to treat regions with a negative GWN or a GWN of 1 or greater as not contributing to any final pixel values. In the above pictures, the part model is colourized by the GWNs that are sampled by each ray using the colours of the rainbow. The more red a pixel is, the lower the GWN is at the point shown. On the other hand, the more violet a pixel is, the higher the GWN is at that point. Regions that are white have a GWN of 1 or greater and are entirely contained inside the part model.

It is interesting to explore the differences in visual fidelity for each implicit boundary representation. In all cases, regardless of voxel grid resolution, rendering was able to be performed in real time at the monitor's refresh rate (60hz), with the dense voxel grid taking only up to 5% of available video memory.

Even at a low fidelity 32^3 voxels, the interior boundaries of the part are apparent. Furthermore, as shown in Section 6, the generation of a 32^3 dense voxelization for even the most pathological input cases has a reasonable run time, making this suitable for a rapid previewing technique. For each model, there is a clear point where increasing the number of samples of the voxelization does not improve visual quality.

6 EVALUATION

To evaluate the effectiveness of our solution in generating implicit boundary representations of part models, we test constructing an implicit boundary using our method on 4 part models in STL format and examine the amount of time taken to construct each implicit boundary at different resolutions.

6.1 Testing System

The test system has the following specifications:

- CPU: Core-i7 8700k (4.9ghz core clock on all cores, 4.8ghz cache clock)
- Memory: 32GB 3866mhz DDR4
- GPU: Nvidia GeForce GTX 1080Ti FE
- Storage: 3x250GB Samsung 840 Evo Solid State Drives

The tests were run with the GPU running in compute-only mode, with the display of the machine connected to the integrated graphics port. This isolates the running time of each test from other background activities on the system, such as desktop compositing. In each case, the time measured is from the time to upload the triangle data to the GPU to the time the results were completely streamed back to the host system.

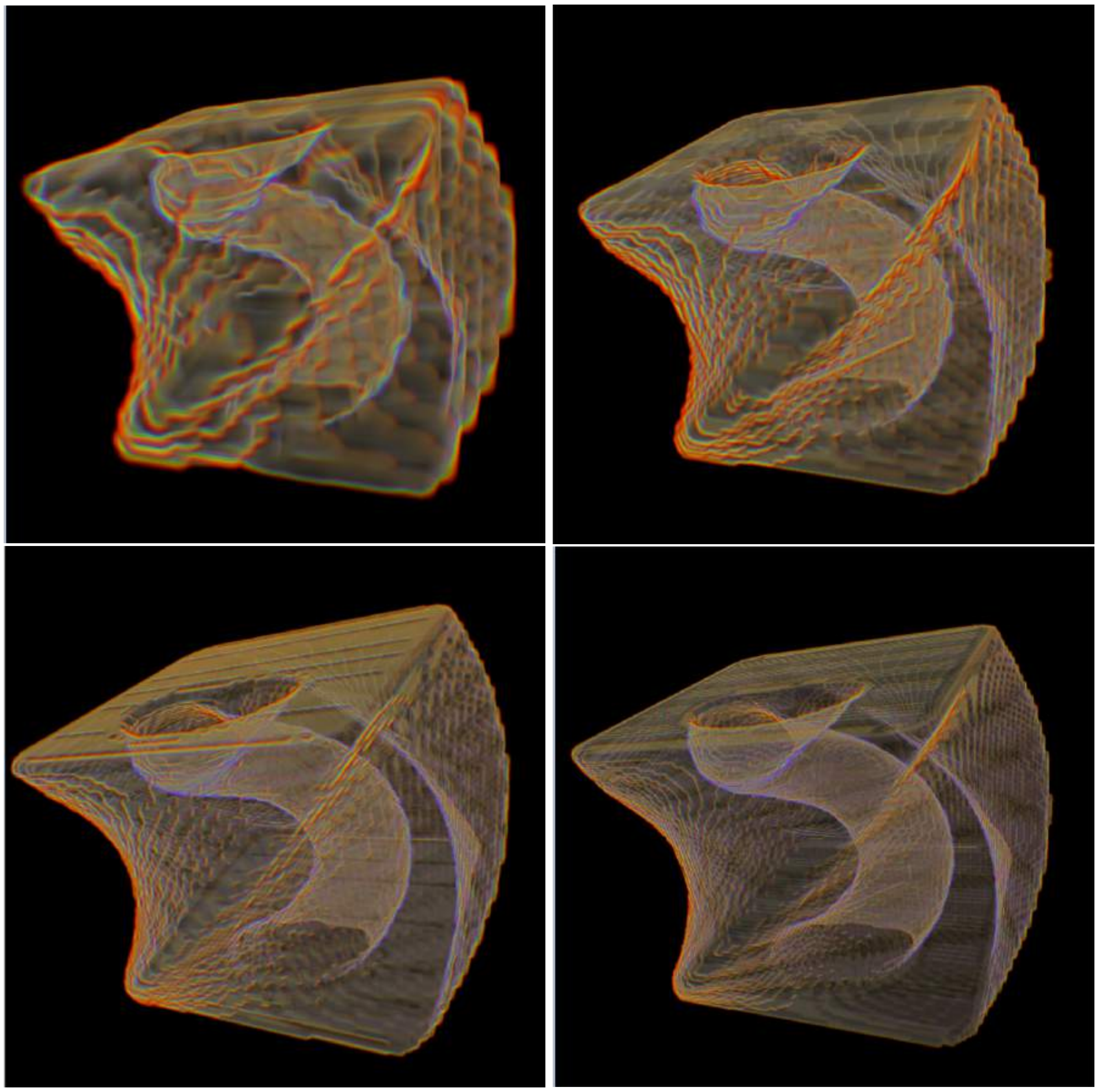


Figure 10: A comparison of the implicit boundary representations for IRAP.stl at different voxel resolutions. Top-left to lower-right: 32^3, 64^3, 96^3, 128^3 total voxels respectively.

6.2 Results

(Tab. 3) summarizes the time it took to construct each implicit boundary representation. (Tab. 4) summarizes how many voxels per second were processed for each dataset at different voxel grid resolutions.

Dataset	*STL Size*	32^3 voxels	64^3 voxels	96^3 voxels	128^3 voxels
multiplane.stl	45,358KB	0.57s	4.12s	13.81s	34.3929s
widget.stl	210,099 KB	2.61s	20.31s	71.63s	176.817s
IRAP.stl	176,202 KB	2.22s	17.45s	60.64s	153.047s
MANIFOLDTP.stl	2,242,654 KB	29.82s	235.26s	854.38s	1976.42s

Table 3: Processing time required for the CSG union of explicit triangle-mesh boundaries using a dense voxelization of the GWN as the implicit boundary representation.

Dataset	*STL Size*	32^3 voxels	64^3 voxels	96^3 voxels	128^3 voxels
multiplane.stl	45,358KB	57487.72	63627.18	64064.88	60016.54
widget.stl	210,099 KB	12554.79	12907.14	12351.47	11860.58
IRAP.stl	176,202 KB	14760.36	15023.01	14590.6	13702.67
MANIFOLDTP.stl	2,242,654 KB	1098.86	1114.3	1035.54	1061.1

Table 4: The number of voxels per second that were processed for each dataset at different voxel grid resolutions.

6.3 Discussion

On even the most pathological input set, we achieved reasonable running times for low-fidelity approximations (32^3 voxels). However, it is clear that the dense voxel representation in our approach has trouble scaling beyond 128^3 voxels when accounting for pathological input. Indeed, the MANIFOLDTP dataset had poor performance at 128^3 voxels, taking approximately 65 minutes to complete at a resolution of 128^3 voxels.

In each case, however, the implicit boundary was able to be found very quickly in comparison to exact boundary approaches. The MANIFOLDTP.stl dataset contains over 45 million triangles and yet a low-fidelity result was able to be produced in just under 30 seconds. Computing the explicit boundary of this STL directly would likely not be possible in under 12 hours with an explicit boundary approach. Directly visualizing all triangles without computing the explicit boundaries induces a load time of 5 minutes or more on common software, and is non-interactive since the refresh rate of the render is approximately 1 frame per second.

One interesting aspect to look at is how many voxels per second were processed for each input mesh at each chosen resolution. Clearly, each mesh had more voxels per second processed at some resolutions than others, which could help guide a sparse voxel approach in the future.

7 FUTURE WORK AND APPLICATIONS

7.1 Sparse Representation

There are two key problems with the dense voxel representation: space, and processing time. A dense voxelization that contains M samples requires M samples for storage without compression. Consider that $M = N^3$ in the worst case, i.e., when the number of samples taken along each dimension is N, and this requires N^3 storage space. Increasing N clearly has diminishing returns as shown in Subsection 5.4, therefore it would be ideal to find a balance between quality and representation size.

Next, consider that if there are T triangles in the STL, and the dense voxel grid has a total of N^3 samples, then the worst-case time complexity of our approach is $O(TN^3)$ for the number of arithmetic operations performed, since the GWN needs to be evaluated at each point (an $O(N)$ process). If the number of triangles equals the number of samples, then this becomes $O(N^4)$. The reason the GPU works so well for this is that the problem is embarassingly parallel.

As an observation, note that many samples are taken in areas that aren't needed in the dense voxelization process, for example regions that have a mostly constant GWN. The regions of interest in visualization are the boundaries, and it would make more sense to design a voxelization scheme that attempts to place more samples at these regions than in empty areas. One way of doing this is using a Sparse Voxel Octree, as discussed in Subsection 4.1.

As we have shown, it is possible to easily compute groups of 32^3 voxels on the GPU for large datasets. One way of increasing fidelity is to create an octree where each node contains a group of 32^3 samples, and use the standard deviation of the samples within each node of the tree as the subdivision condition. If the standard deviation is small, then the node is kept as a leaf, otherwise it is recursively subdivided and the process continues. The end result is that SVO hierarchically hones down towards the boundaries of the part model. The depth of the SVO is directly related to how much information and detail is preserved. The parameters of the SVO are tunable and could be adjusted to find a reasonable compromise between fidelity and speed. Additionally, the SVO could be tagged with Hermite data at the edges to help preserve sharp features in the underlying GWN field [20].

This would simultaneously address the space and time problems with the dense approach, since the visual fidelity would directly correspond to the number of samples per leaf and the depth of the SVO. Using an SVO also invites the possibility of using non-linear interpolation, which could further improve visual quality during volume rendering [8] and isosurface extraction.

7.2 Void Detection

A voxel representation for part models could prove useful for void detection on parts manufactured using AM. Consider that a void in an AM part will have a GWN that is zero in that region. Machine learning could be applied directly to the voxel model to identify these problematic regions and provide a metric to users about the quality of the AM toolpath that has been chosen. The machine learning component would be used to differentiate these unintentional voids against intentional empty pockets within the part as the basis for a quality metric.

7.3 Thin-wall Detection

Consider the water pump STL that was shown in (Fig. 3). This part is difficult to generate a good toolpath for since it contains many thin-wall regions. It is critical to identify these regions as they need to be specially handled during the toolpath generation process. In (Fig. 11), we show the internal boundaries of the part using a dense voxelization with 256^3 samples, and we have colourized the interior of the part. Here, the thin-wall regions are immediately visually apparent. In the original STL, it would only be apparent if cross sections were examined throughout the part. This could be used as the basis for a thin-wall detection metric for guiding toolpath generation.

7.4 Heat-transfer Modelling

Voxelizations could also be useful for modelling heat propagation throughout a part during the AM process. Consider that a voxel sample could be generalized to contain material information and temperature. The process of bead deposition could be modelled as the print head moves across the field, updating sample values as it traverses the model. Voxels could share information with their neighbours to propagate properties across neighbouring regions.

7.5 Additional Machining Operations

Additional machining operations can be directly simulated on the implicit boundary representation of the part as CSG operations. A simulator could be designed that uses the implicit boundary representation and uses isosurface extraction at the final step for compatibility with major software packages, which expect explicit boundary representations.

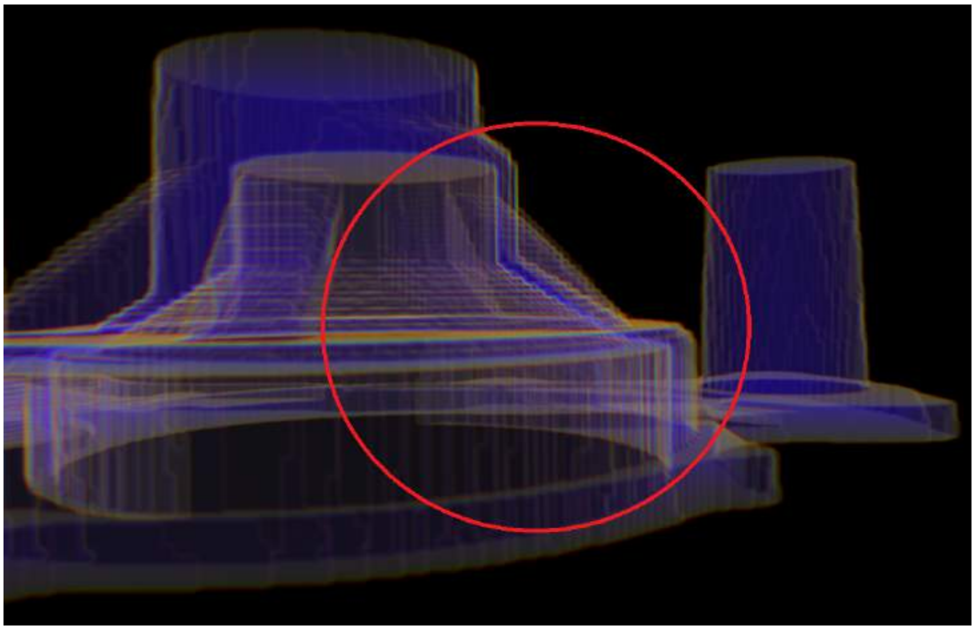


Figure 11: Volume rendering of the 128^3 sample dense voxelization of the implicit boundary representation of (Fig. 3). Indigo regions are interior to the part. Circled: a thin-wall region.

7.6 Parametrically-defined Explicit Toolpath Boundaries

Currently, explicit toolpath boundaries are defined in terms of triangle meshes. However, it may be desirable to parametrically define these boundaries instead in some cases. For example, the X3D intermediate format contains no configuration parameters for how geometry should behave around bends in the Extrusion element, which is limiting and can produce inaccurate results during visualization. To work around this, any sharp bend that is taken in a toolpath is separated into separate extrusions with end-caps added to make the geometry appear continuous. A parametrically defined toolpath boundary should be able to elegantly remove these limitations such that these workarounds are no longer necessary. Since the GWN depends only on being able to compute the solid angle of the surfaces that make up the geometry, it should be possible to directly use the method in Section 5 with a parametrically defined explicit boundary representation instead of the current triangle-mesh representation.

7.7 Hierarchical Evaluation of the GWN

The approach described in Section 5 relied on the brute-force power of the GPU to naively evaluate many winding numbers concurrently, enabling the GWN field of STL files to be directly evaluated without preprocessing. However, hierarchical methods do exist for evaluating the GWN on large meshes [10] which feature better asymptotic time complexity. Currently, these rely on heavy conditional branching and divergent memory access patterns which make them more suitable for use on CPUs rather than GPUs. Furthermore, the approach described in [10] relies on arbitrary-precision arithmetic when constructing the hierarchy which could pose scalability problems for large datasets. A key feature of their method is that it exactly computes the GWN. If that constraint were relaxed, it may be possible to create an approximation which does not require arbitrary-precision arithmetic. Furthermore, it may be possible to develop a hierarchical evaluation scheme where memory access patterns are able to be coalesced for efficient GPU processing using a compact octree representation.

7.8 Explicit Boundary Extraction

An isosurface mesh of the result can be extracted at the implicit boundary where the GWN field is 0.5. The result could be made manifold if desired [20]. This represents the explicit boundary of the part model, which can be saved in the conventional STL format for compatibility with major software packages. The resolution of the final mesh will depend on the sampling resolution and isosurface generation method chosen. This gives flexibility to users, as they can choose their own balance between part model size and detail preservation.

8 CONCLUSIONS

In conclusion, we have developed a framework for scalable part-model generation that addresses the specific needs of additive manufacturing. First, we explored the effect numeric representation has on robustness. Next, we explored how explicit boundary representations have difficulty being simultaneously robust and scalable. We then presented implicit boundary representations as an inexact, however scalable and robust alternative. Our approach describes bead boundaries for toolpaths explictly, however the actual part model is computed implicitly, and the result can be directly visualized on the GPU. Interestingly, the implicit view of geometry could have other applications in AM as well for functional modelling, including void detection [25], and heat-transfer. In particular, machine learning could be useful to identify voids from the implicit boundary representation of the part model as part of a quality control metric. We explored some potential ways to improve scalability using sparse voxel approaches and hierarchical evaluation methods for the Generalized Winding Number. In summary, for Additive Manufacturing, it may be worth considering viewing geometry as an implicit, rather than an explicit feature of space for scalable visualization, process simulation, and feature detection.

ACKNOWLEDGEMENTS

An Ontario Centres of Excellence (OCE) TalentEdge Academic Internship grant with CAMufacturing Solutions Inc. funded this research.

ORCID

Shane Peelar, http://orcid.org/0000-0001-7391-0951
R. Jill Urbanic, http://orcid.org/0000-0002-2906-7618
Robert W. Hedrick, http://orcid.org/0000-0003-2708-6943
Luis Rueda, http://orcid.org/0000-0001-7988-2058

REFERENCES

[1] IEEE standard for floating-point arithmetic. http://doi.org/10.1109/ieeestd.2008.4610935.

[2] Alexander, J.W.: Topological invariants of knots and links. Transactions of the American Mathematical Society, 30(2), 275–306, 1928. http://doi.org/10.1090/S0002-9947-1928-1501429-1.

[3] Bernstein, G.; Fussell, D.: Fast, exact, linear booleans. Computer Graphics Forum, 28(5), 1269–1278, 2009. http://doi.org/10.1111/j.1467-8659.2009.01504.x.

[4] Daly, L.; Brutzman, D.: X3d: Extensible 3d graphics standard [standards in a nutshell]. IEEE Signal Processing Magazine, 24(99), 130–135, 2007. http://doi.org/10.1109/msp.2007.4317479.

[5] Fabri, A.; Pion, S.: The exact computation paradigm. https://www.cgal.org/exact.html. Accessed: 2018-04-05.

[6] Fabri, A.; Pion, S.: Cgal: The computational geometry algorithms library. In Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems, 538–539. ACM, 2009. http://doi.org/10.1145/1653771.1653865.

[7] Fedkiw, S.O.R.; Osher, S.: Signed distance functions, level set methods and dynamic implicit surfaces. Applied Mathematical Sciences, 153, 17–22, 2003. http://doi.org/10.1007/0-387-22746-6_2.

[8] Fuhrmann, S.; Kazhdan, M.; Goesele, M.: Accurate isosurface interpolation with hermite data. In 2015 International Conference on 3D Vision. IEEE, 2015. http://doi.org/10.1109/3dv.2015.36.

[9] Gustafson, J.L.; Yonemoto, I.T.: Beating floating point at its own game: Posit arithmetic. Supercomputing Frontiers and Innovations, 4(2), 71–86, 2017. http://doi.org/10.2165/00128413-199510020-00017.

[10] Jacobson, A.; Kavan, L.; Sorkine-Hornung, O.: Robust inside-outside segmentation using generalized winding numbers. ACM Transactions on Graphics (TOG), 32(4), 33, 2013. http://doi.org/10.1145/2461912.2461916.

[11] Jacobson, A.; Panozzo, D.: libigl. In SIGGRAPH Asia 2017 Courses on - SA '17. ACM Press, 2017. http://doi.org/10.1145/3134472.3134497.

[12] Ju, T.; Losasso, F.; Schaefer, S.; Warren, J.: Dual contouring of hermite data. ACM Transactions on Graphics, 21(3), 2002. http://doi.org/10.1145/566654.566586.

[13] Kettner, L.; Mehlhorn, K.; Pion, S.; Schirra, S.; Yap, C.: Classroom examples of robustness problems in geometric computations. Computational Geometry, 40(1), 61–78, 2008. http://doi.org/10.1016/j.comgeo.2007.06.003.

[14] Lorensen, W.E.; Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. ACM SIGGRAPH Computer Graphics, 21(4), 163–169, 1987. http://doi.org/10.1145/37402.37422.

[15] Maddock, J.; Kormanyos, C.: Boost multiprecision, 2016.

[16] Naylor, B.: Constructing good partitioning trees. In Graphics Interface, 181–181. Canadian Information Processing Society, 1993.

[17] Nikolaevskaya, E.A.; Khimich, A.N.; Chistyakova, T.V.: About GMP. In Programming with Multiple Precision, 13–30. Springer Berlin Heidelberg, 2012. http://doi.org/10.1007/978-3-642-25673-8_2.

[18] Requicha, A.G.; Voelcker, H.: Solid modeling: Current status and research directions. IEEE Computer Graphics and Applications, 3(7), 25–37, 1983. http://doi.org/10.1109/mcg.1983.263271.

[19] Saqib, S.M.: Experimental investigation of laser cladding bead morphology and process parameter relationship for additive manufacturing process characterization. Ph.D. thesis, University of Windsor (Canada), 2016.

[20] Schaefer, S.; Ju, T.; Warren, J.: Manifold dual contouring. IEEE Transactions on Visualization and Computer Graphics, 13(3), 610–619, 2007. http://doi.org/10.1109/tvcg.2007.1012.

[21] Schirra, S.: Robustness and precision issues in geometric computation. In Handbook of Computational Geometry, 597–632. Elsevier, 2000. http://doi.org/10.1016/b978-044482537-7/50015-2.

[22] Thibault, W.C.; Naylor, B.F.: Set operations on polyhedra using binary space partitioning trees. ACM SIGGRAPH Computer Graphics, 21(4), 153–162, 1987. http://doi.org/10.1145/37402.37421.

[23] Treece, G.; Prager, R.; Gee, A.: Regularised marching tetrahedra: improved iso-surface extraction. Computers & Graphics, 23(4), 583–598, 1999. http://doi.org/10.1016/s0097-8493(99)00076-x.

[24] Urbanic, R.; Hedrick, R.: Developing a virtual model for the fused deposition rapid prototyping process. In Proceedings of the Life Cycle Engineering Conference, 131–137, 2009.

[25] Urbanic, R.J.; Burford, C.G.; Hedrick, R.W.: Virtual quality assessment tools for material extrusion processes. Computer-Aided Design and Applications, 15(4), 520–531, 2018. http://doi.org/10.1080/16864360.2017.1419640.

[26] Yv, D.: Node.js package bitmap-sdf. https://www.npmjs.com/package/bitmap-sdf. Accessed: 2018-04-05.

[27] Zhou, Q.; Grinspun, E.; Zorin, D.; Jacobson, A.: Mesh arrangements for solid geometry. ACM Transactions on Graphics, 35(4), 1–15, 2016. http://doi.org/10.1145/2897824.2925901.