Hierarchical Volumetric Object Representations for Digital Fabrication Workflows

by

Matthew Keeter

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author ....
Program in Media Arts and Sciences
May 10, 2013

Certified by..
Prof. Neil Gershenfeld
Director, MIT Center for Bits and Atoms
Thesis Supervisor

Accepted by ....
Prof. Patricia Maes
Associate Academic Head
Program in Media Arts and Sciences

Hierarchical Volumetric Object Representations for Digital Fabrication Workflows

by

Matthew Keeter

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
on May 10, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

Modern systems for computer-aided design and manufacturing (CAD/CAM) have a history dating back to drafting boards, early computers, and machine shops with specialized technicians for each stage in a manufacturing workflow. In recent years, personal-scale digital fabrication has challenged many of these workflows' build-in assumptions. A single individual may control the entire workflow, from design to manufacture; they will be using computers that are exponentially more powerful than those in the 1970s; and they may be using a wide variety of tools, machines, and processes.

The variety of tools and machines leads to a combinatorial explosion of possible workflows. In addition, tools are based on boundary representations, which are fragile and can easily describe nonsensical objects. This thesis addresses these issues with a set of tools for end-to-end digital fabrication based on volumetric solid models. Workflows are modular, making it easy to add new machines, and a shared core of path-planning operations reduces system complexity. Replacing boundary representations with volumetric representations guarantees that models represent reasonable real-world solids.

Adaptively sampled distance fields are used as a generic interchange format. Functional representations are used as a design representation, and we examine scaling behavior and efficient rendering. We present interactive design tools that use these representations as their geometry engine. Data from CT scans is also used to populate these distance fields, showing significant benefits in file size and resolution compared to meshes. Finally, these representations are used as inputs to a modular multi-machine CAM workflow. Toolpath generation is implemented, characterized, and tested on a complex solid model. We conclude with a summary of results and recommendations for future research directions.

Thesis Supervisor: Prof. Neil Gershenfeld
Title: Director, MIT Center for Bits and Atoms

Hierarchical Volumetric Object Representations

for Digital Fabrication Workflows

by

Matthew Keeter

The following people served as readers for this thesis:

Thesis Reader . . . .

Neri Oxman
Assistant Professor of Media Arts & Sciences
MIT Program in Media Arts and Sciences

Thesis Reader . . . .

Erik Demaine
Professor of Computer Science
MIT CSAIL

Acknowledgments

First, I would like to thank my advisor, Neil, for his support and feedback through this entire process. I'd also like to thank my other readers, Erik and Neri, for their insights.

Many thanks to all of those who alpha and beta-tested my design tools: the students in How to Make (Almost) Anything, Fab Academy, Sam Calisch (who made far cooler designs than I), and Christian Reed (expert ShopBot wrangler).

I'd like to thank all the people that keep things running smoothly: Joe and Theresa for their efforts in keeping CBA under control; John and Tom for their dedication to the CBA shop and the students that use it; Linda and Bill for their work in keeping us all organized. I would also like to thank DARPA and CBA's sponsors for funding my time here at MIT.

Finally, I owe a great deal to my family and friends who supported me through this process, both inside and outside the lab. It's been a lot of fun – thank you all.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The evolution of computer-aided design and computer-aided manufacturing (CAD/-CAM) tools have left users with a combinatorial explosion of different tools and workflows, illustrated in Fig. 1-1. A single user can scan a physical object with a depth-field camera, touch up the mesh in Rhino, perform path planning in Partworks, and manufacture the object on a Shopbot – or a myriad of other possible combinations.

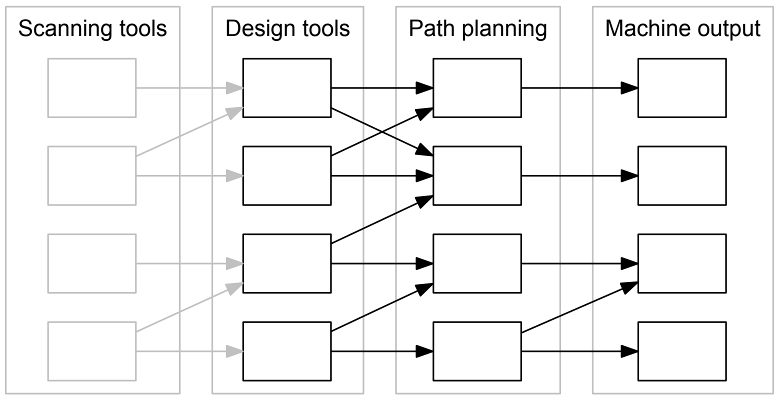


Figure 1-1: Combinatorial explosion

In the past, each of these processes was handled by a specialist. Complexity was managed through segmentation of knowledge. However, with the increased popularity of fab labs [33], hackerspaces, etc., there has been a focused interest in personal-scale fabrication. In these environments, the same individual is in control of the entire workflow, from design to manufacture; furthermore, this individual may be using a wide variety of machines with different parameters and interfaces.

The combinatorial explosion of machines and tools has several major disadvantages. A single user may find it challenging to master multiple dissimilar workflows. In addition, the chain of importing and exporting files from one tool to the next can lead to errors: files that look good in design tools can break in CAM tools, leading to nonsensical toolpaths or failed 3D prints. This problem is exacerbated by the fact that these tools usually use boundary representations (e.g. triangulated meshes), which can easily represent nonsensical solid models.

This thesis presents a complete CAD/CAM workflow based on hierarchical, volumetric object representations; Figure 1-2 illustrates the workflow. We begin with an implementation of adaptively sampled distance fields as a generic volumetric object format. Next, we discuss strategies for populating these data structures, including custom CAD software software and volumetric scanning. Finally, we conclude with a discussion of path planning and manufacturing, then showcase a complex model fabricated with this workflow.

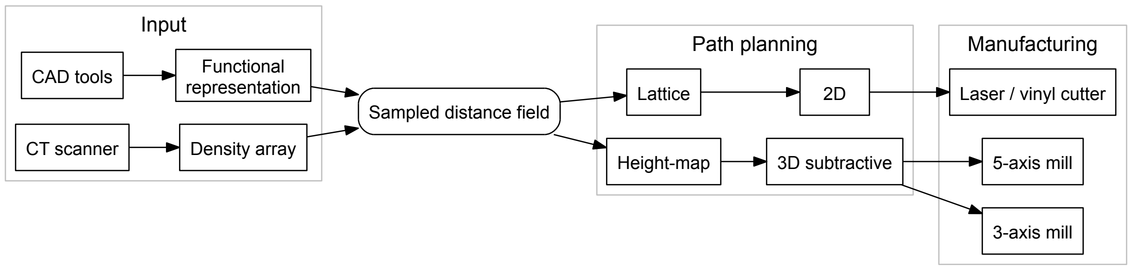


Figure 1-2: End-to-end CAD/CAM workflow

Chapter 2

Volumetric Object Representations

2.1 Historical considerations

Before the advent of computer-aided design, engineers designed on drafting boards. These systems were often mechanically assisted to allow engineers to draw precise lines and curves. Early CAD tools showed this legacy, focusing on accurace placement of points, lines and curves.

Operators such as extrusion, lofting, and turning turn these flat sketches into solid objects. The edges of a two-dimensional drawing are transformed into solid faces in a three-dimensional representation, formalized by the boundary-representation (b-rep) paradigm [7].

Thanks to this gradual evolution, nearly all modern CAD/CAM systems use b-reps and surface modeling. B-rep popularity was also influenced by early computing hardware: b-reps and triangulated meshes are simple to render, even with limited processing power. However, there are significant downsides to the widespread use of boundary representations.

To be valid, a boundary representation must be “water-tight”: it must define a complete, closed surface without cracks, holes, or co-incident faces. This is a challenging criterion to meet, as evidenced by tutorials and software tools specifically to help users clean broken meshes [27, 32]. The chain of exports and imports described in the previous chapter includes plenty of opportunities for meshes to become invalid or nonsensical.

A related downside is complexity. Developing a robust b-rep geometry kernel is a challenging task. There is a high barrier to entry for individuals who want to create or modify their own design tools. Rapid prototyping of CAD software, while somewhat uncommon, is hindered by the b-rep paradigm.

2.2 Volumetric representations

This chapter presents a general-purpose volumetric representation for solid objects, with details on implementation and performance. A volumetric data structure must represent a closed, solid volume in an unambiguous manner. It should be possible to query an (x, y, z) coordinate and learn whether this point is inside of the solid; such a query is challenging (or even nonsensical) on a boundary-representation, where the boundary could be open or discontinuous.

Voxel[1] data is a trivial volumetric structure. A solid can be represented by a 3D array of Boolean values, where each Boolean represents either empty or filled. Data acquired from a CT scanner is often in this format (although the values are density samples, rather than Booleans). However, this representation scales poorly.

Introducing hierarchy allows models to take advantage of spatial locality: filled regions are typically clustered near each other, not randomly spread out in space. One basic hierarchical volumetric structure is the octree [30]. In an octree, a three-dimensional spatial region is repeatedly split into eight subregions; cells are labelled as solid, empty, or branching. This has the disadvantage that model surfaces must be represented by many cells at the smallest level of recursion.

There are various extensions to the octree structure. Extended octrees [8] add a set of special nodes to represent planar faces, edges, and vertices. Dyllong [16] goes even farther, storing fragments of CSG descriptions in octree cells for use in later model reconstruction. Both of these octree variations are too specific for our purposes. The first is optimized for a triangle-based workflow, and the second for a

[1]A “voxel” is the three-dimensional equivalent to a pixel

purely CSG workflow.

Instead, we choose to base our volumetric representation on adaptively sampled distance fields (ASDFs), as described in [17, 13]. In the past, ASDFs have been used as the underlying representation for both sculpting-style design tools [43] and machining simulations [46], among other applications.

2.3 Data structure

In the ASDF data structure, the Euclidean distance to a solid's surface is sampled and stored in a hierarchical structure; positive distances are outside and negative distances are inside the solid. These distance samples reduce the number of cells required and allow for calculation of surface normals.

The ASDF described in this chapter is implemented as an extension to a traditional octree [30]. A discrete spatial lattice is subdivided with up to eight-fold splits at each level, as shown in Fig. 2-1. The region is discrete, with a minimum possible voxel size and larger cells that contain multiple smaller voxels.

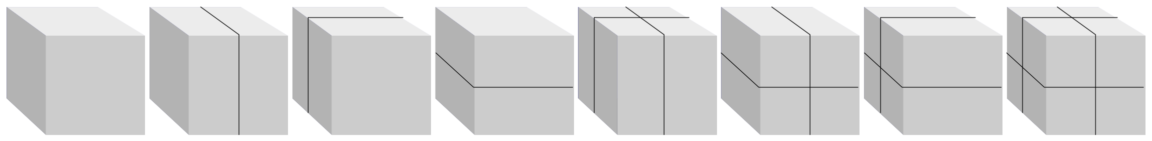


Figure 2-1: Octree cell splits

Each cell in the octree is labelled as full, empty, branching, or leaf. The first three states are common to traditional octrees; the last is unique to the ASDF. This implementation can be more precisely described as a superficial or compositive ASDF [5, 46], in which only cells containing an object boundary ("leaf" cells) store accurate distance samples.

The implemented data structure is shown in Table 2.1. The `Interval` data type used for bounds contains two floating-point values (`upper` and `lower`). The `data` member variable is used for operation-dependent storage (e.g. triangulation uses it to store vertex indices).

Name	Type	Description
`state`	`enum`	`FILLED`, `EMPTY`, `BRANCH`, or `LEAF`
`X, Y, Z`	`Interval`	Floating-point bounds
`branches`	`ASDF*`	Pointers to children
`d`	`float[8]`	Distance samples
`data`	`void*`	Miscellaneous data pointer

Table 2.1: Member variables of the `ASDF` structure

Some data duplication occurs within a full ASDF tree. Neighboring cells often share distance samples, and the cell bounds could be reconstructed from the bounds and lattice resolution of the top-level region. When implementing this data structure, RAM was deemed to be cheaper than both CPU and programmer time, so the duplication was allowed to persist; strategies to mitigate this are described in [5, 9].

2.4 Cell reconstruction and combination

Within an ASDF leaf cell, interpolation is used to determine the object's boundary. Given a position triplet (x, y, z), cell bounds, and distance values at cell corners, trilinear interpolation produces a value $d_{x,y,z}$. If this value is less than zero, then the point (x, y, z) is within the object; otherwise, it is outside.

In ordinary octrees, cells can be collapsed if they are all full or all empty. The ASDF structure adds another option: leaf cells can be combined if the resulting interpolation error is sufficiently small. In other words, as long as interpolation on the parent cell accurately reproduces its children, the data in the children is superfluous. This is an advantage compared to the octree, which must always recurse down to minimum voxel size along object edges.

The algorithm for octree simplification is given in Alg. 2.1. The algorithm examines each axis in turn, checking to see whether each pair of branches can be merged along this axis. This implementation does not require octrees to be "complete" (i.e. divided eight-fold at every stage) – requiring complete octrees would force volume sizes to be powers of two, which was deemed too strict a limit for a general-purpose volumetric format.

Algorithm 2.1 ASDF simplification (non-recursive)

```
function SIMPLIFY(asdf)
    for each axis $x$, $y$, and $z$ do
        merge ← true
        for each pair of branches that could be merged along this axis do
            $A$ ← first branch
            $B$ ← second branch
            if $A$ or $B$ is of type BRANCH then
                if $A$ splits on this axis or $B$ splits on this axis or
                        $A$ and $B$ split differently then
                    merge ← false
                end if
                if not ($A$ and $B$ are of type BRANCH) then
                    merge ← false
                end if
            else if $A$ or $B$ is of type LEAF then
                if interior samples can't be reconstructed accurately then
                    merge ← false
                end if
            else if $A$'s type is different from $B$'s type then
                merge ← false
            end if
        end for
        if merge is true then
            for each pair of branches that could be merged along this axis do
                $A$ ← first branch
                $B$ ← second branch
                Collapse $A$ and $B$ into $A$
                if $A$ or $B$ is of type LEAF then
                    Set $A$'s state to LEAF
                else if $A$ and $B$ are branches then
                    Move $B$'s branches to $A$.
                end if
                Remove $B$ from the tree
            end for
        end if
    end for
end function
```

Figure 2-2 shows an ASDF being simplified along the Y axis. The two leaf cells on the left are merged into a single leaf cell which reconstructs its interior values with interpolation. The two branch cells on the right are consolidated into a single branch, which is only possible because neither of them splits on the Y axis; this is tested in line 8 in the pseudo-code.

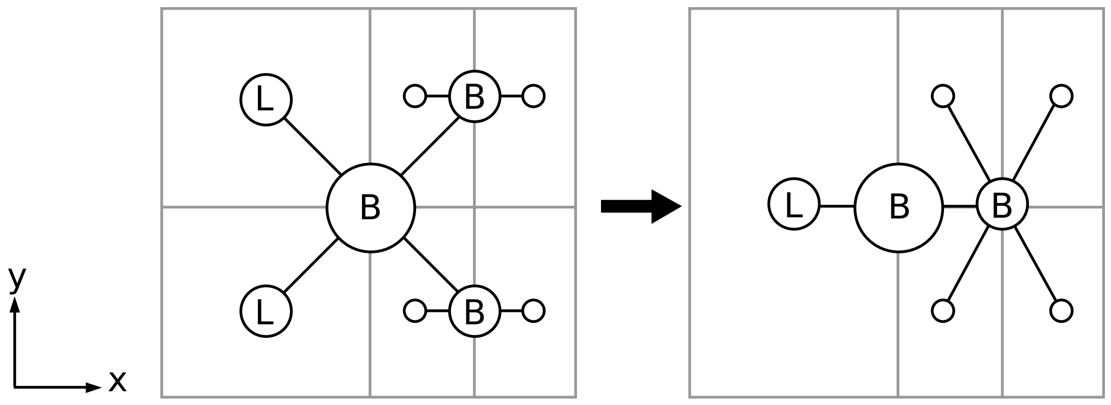


Figure 2-2: ASDF simplification on y axis

Note that this algorithm is non-recursive; it assumes that branches of the provided ASDF are already simplified as much as possible. This is consistent with "bottom-up" ASDF construction [13].

Figure 2-3 compares an octree and an ASDF representation of a planar model. Red cells are filled; blue are empty; green are leaf cells. Note that in the ASDF, long flat edges are stored in a single green leaf cell. The curving edges of the shape are still stored in small voxels; the reconstruction error threshold can be adjusted to allow for more or less cell combination.

Cell counts between ASDFs and pure octrees are compared in Fig. 2-4. The wheel model shown in Fig. 2-3 is rendered at a variety of resolutions. As expected, the ASDF consistently requires fewer cells than the octree. Furthermore, ASDF cell count stabilizes at a particular value, dependent on the acceptable error level used when merging leaf cells.

Figure 2-5 examines leaf cell combination in more detail. It was generated by representing a circle as ASDF, with an increasing voxel resolution and varying levels

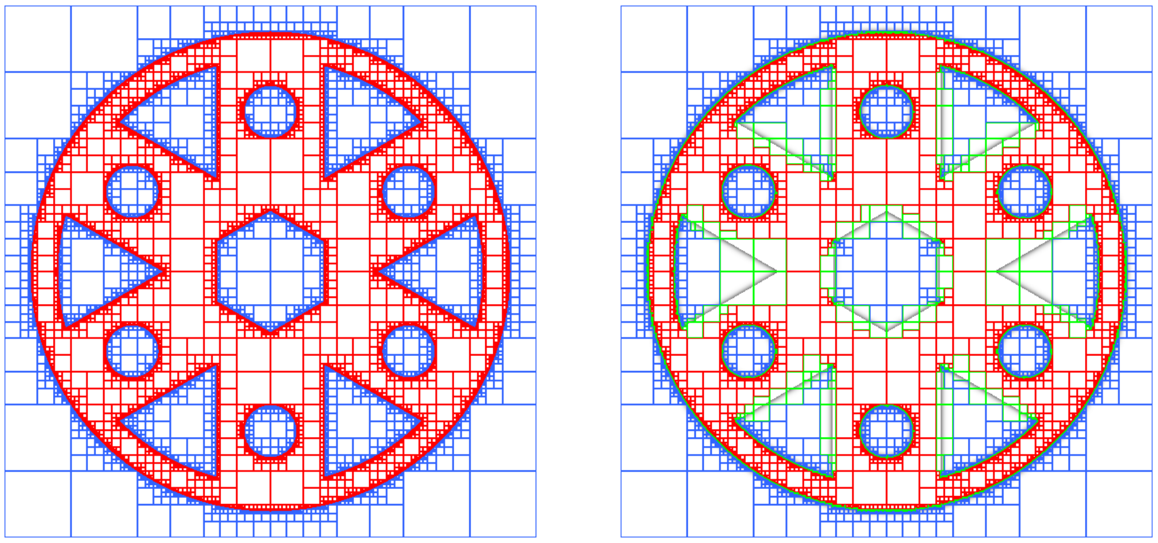


Figure 2-3: Cells in octree and ASDF

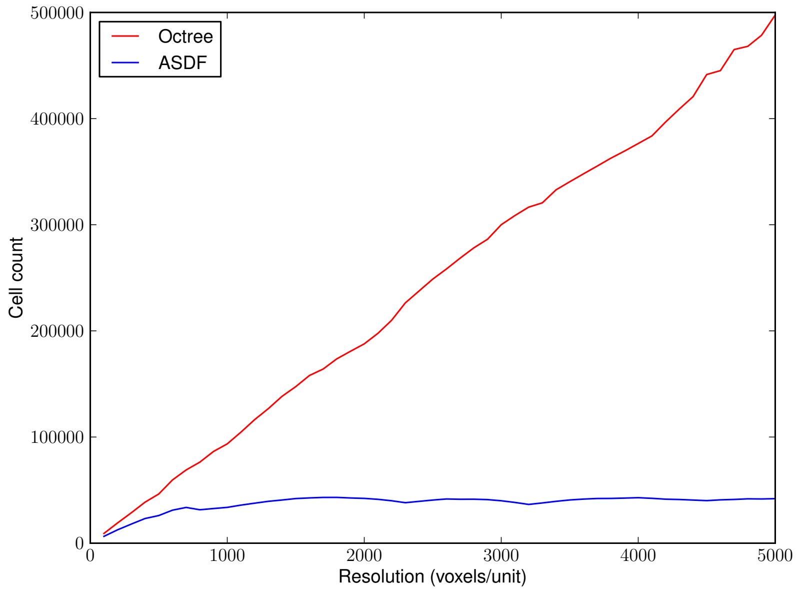


Figure 2-4: Cell count in octree and ASDF

of interpolation error. As expected, the number of voxels stabilizes at a level that increases as the maximum allowed interpolation error decreases. These results are comparable to results from [17] examining sample counts as a function of maximum error level.

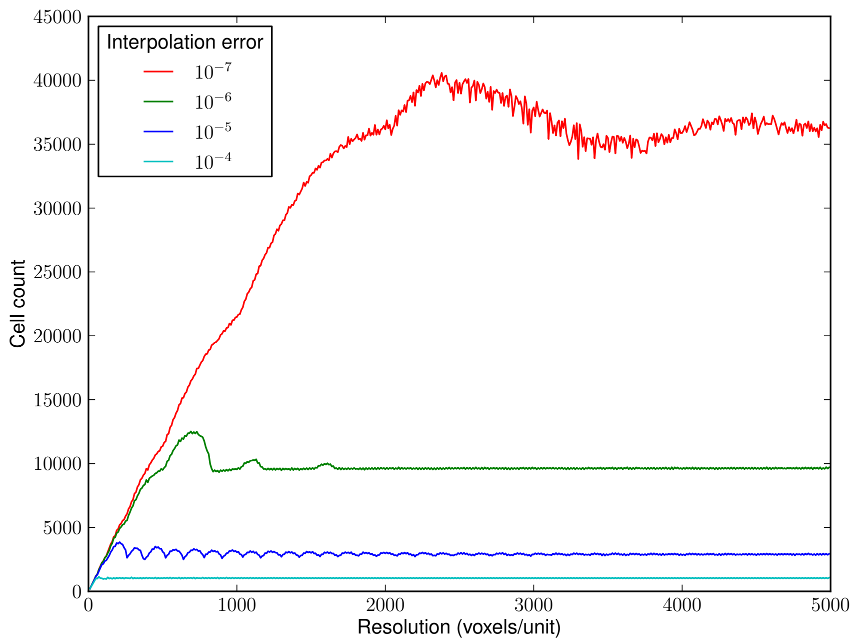


Figure 2-5: Number of cells defining circle

2.5 Rotation performance

Because octrees are non-isotropic data structures, they often show performance variation with rotation. Figure 2-6 examines ASDF cell count as a function of rotation about two axes. Each ASDF was generated from an cube of side length 1, rendered in a $2 \times 2 \times 2$ region with 128 voxels/unit resolution.

The data indicates some degree of isotropy: cell count varies by a factor of three across the range of rotation angles. Dark lines at appear 90° intervals, corresponding to rotated axial alignment.

2.6 Height-map rendering

In Chapter 6, we discuss path planning on discrete lattices and distance-transformed images. As such, a rendering pipeline from ASDFs to bitmaps and height-maps is necessary.

Others have used sphere tracing [19] to render ASDFs; however, this strategy requires either a complete ASDF or a superficial ASDF with various guarantees that

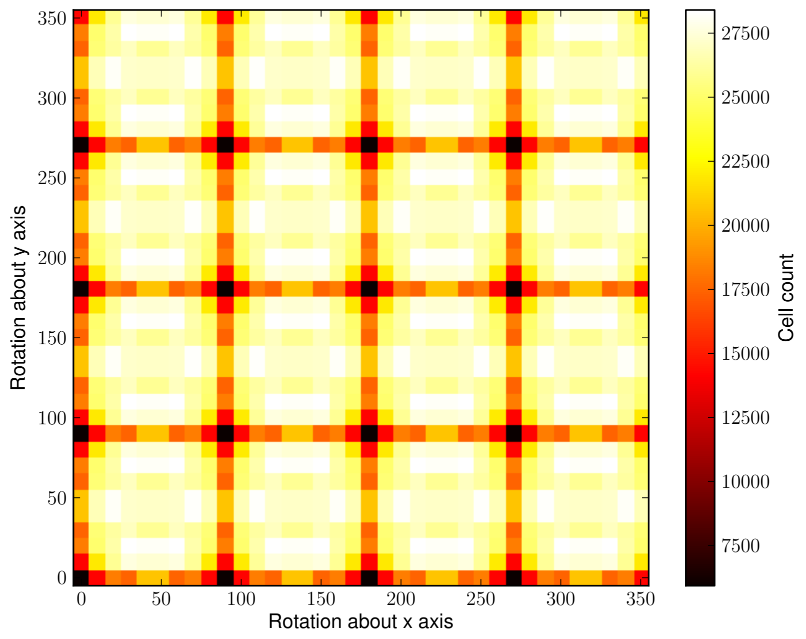


Figure 2-6: Rotation performance of ASDF

cannot be made in our models [5]. It is possible to render ASDFs by treating them as implicit surfaces, then applying spatial subdivision and interval arithmetic as described in [15] and in Alg. 3.2. This strategy queries into the ASDF data structure to check whether a region is filled or empty.

In practice, we have found that recursion into the ASDF tree is significantly more efficient than recursion on the spatial region. Algorithm 2.2 describes the general process of rendering an ASDF data structure. It allows for coordinate transforms between the view frame (which defines the render region) and the ASDFs cells (which define frames for interpolation). We implemented rotation about the x and z axes parameterized by α and β variables, but this could be extended to arbitrary coordinate mappings (e.g. for rendering with perspective).

Figure 2-7 shows an ASDF rendered to a height-map; the source file is a sphere with internal microstructure (inspired by [42]).

The running time of ASDF to height-map rendering depends on both ASDF resolution and output image resolution. The relationship is examined in Fig. 2-8. Surprisingly, lower-resolution ASDFs are not necessarily faster to render using this strategy. For low-resolution ASDFs, the loop over voxels in leaf cells begins to domi-

Algorithm 2.2 ASDF rendering algorithm

```
function ASDFtoHeightMap(asdf, region)
    if region is entirely outside the bounds of asdf then
        return
    end if
    Shrink region based on bounds of asdf
         (taking coordinate transforms into account)
    if asdf is a LEAF or FILLED cell then
        for each voxel in region do
            Transform voxel into asdf's coordinate frame
            if the voxel is outside of asdf's bounds then
                continue with next iteration of loop
            end if
            if asdf is a FILLED cell or interpolated value at voxel is $< 0$ then
                Fill in pixel based on voxel's height
            end if
        end for
    else
        ASDFtoHeightMap(branch, region) for each branch of asdf
    end if
end function
```

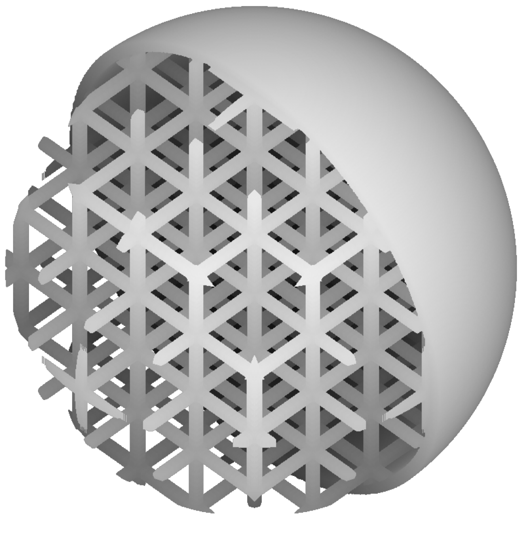


Figure 2-7: Height-map rendering of ASDF

nate the running time, as each leaf cell spans a larger number of voxels. Switching to a recursive strategy for leaf cell rendering would eliminate this discrepancy, but was not implemented.

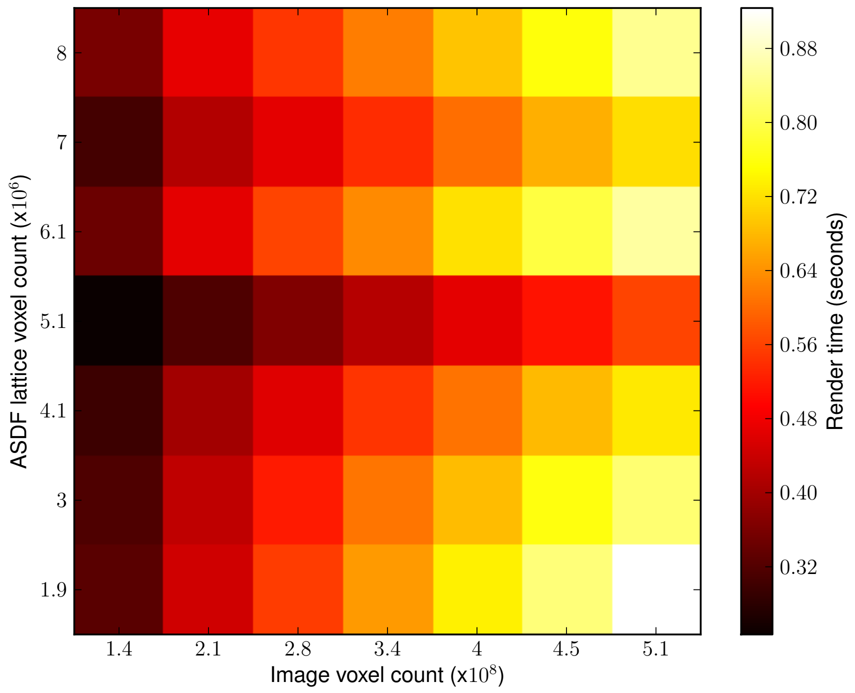


Figure 2-8: Height-map rendering of ASDF

2.7 Shaded rendering

Height-maps are useful for three-axis machining, but for full five-axis machining, surface normals are also often necessary. Normals can be estimated by taking the local gradient of the distance field within leaf cells, though this leads to C^1 discontinuities [5]. Algorithm 2.2 is used with slight modifications: instead of checking single pixels, all corners of a projected pixel-cubes are checked. This prevents errors due to sampling frequency.

The same microstructure file is rendered with normals and shading in Fig. 2-9. In the left image, x, y, and z normals are interpreted as R, G, and B color values; the second image is shaded based on the normal angle.

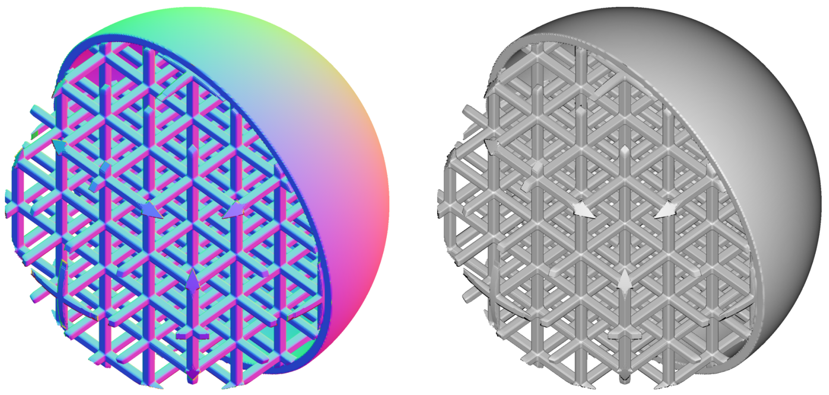


Figure 2-9: Normal and shaded rendering of ASDF

Running time for shaded rendering is shown in Fig. 2-10. The effect discussed above – in which lower-resolution ASDFs take longer to render – is more severe, since the work done to render a single pixel is more significant. Still, optimizing both height-map and shaded rendering speed was a low priority, as the time cost is incurred only once in the CAD/CAM workflow.

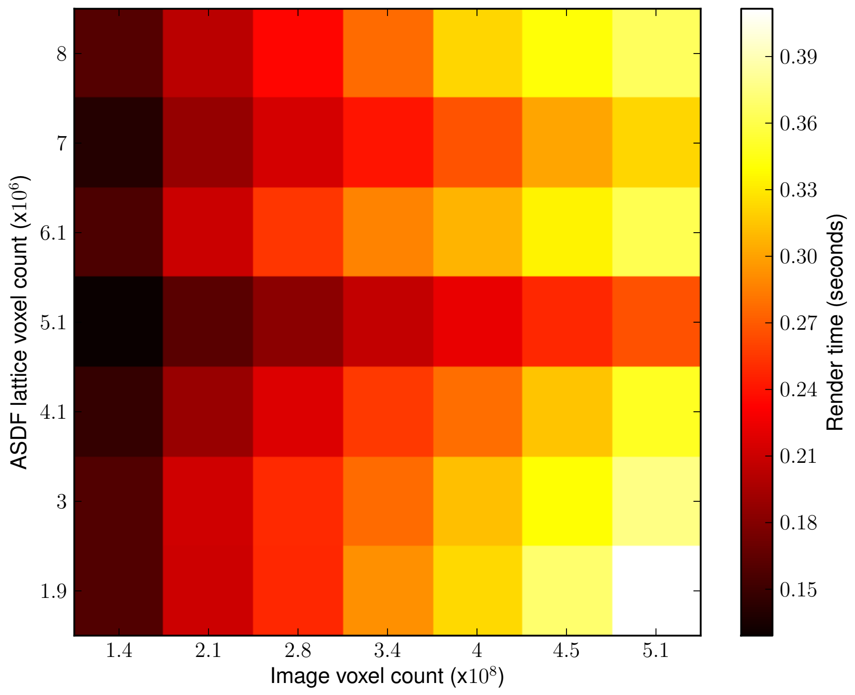


Figure 2-10: Shaded rendering of ASDF

2.8 Triangulation

Despite the thesis-wide emphasis on volumetric representations, triangulated meshes have their place. Modern graphics hardware can easily render thousands or millions of shaded triangular faces. As such, ASDF triangulation routines were developed to visualize the volumetric data files.

Marching cubes [29, 20] is a fundamental algorithm in computer graphics. Basic implementations can produced flawed meshes due to ambiguities in neighbouring cells; there are a variety of strategies to resolve these flaws [37, 28]. The chosen meshing algorithm, marching tetrahedrons, is an alternative to marching cubes that – while still simple to implement – does not require special cases to resolve inter-cell ambiguities [36].

The implementation relies on the fact that the solid's isosurface will only pass through leaf cells. Filled and empty cells can be ignored. The algorithm traverses the tree recursively, keeping track of both the current cell and neighboring cells with which it shares faces. When a leaf cell is found, it is subdivided into six tetrahedrons (shown in Fig. 2-11), each of which may contain up to two triangle faces.

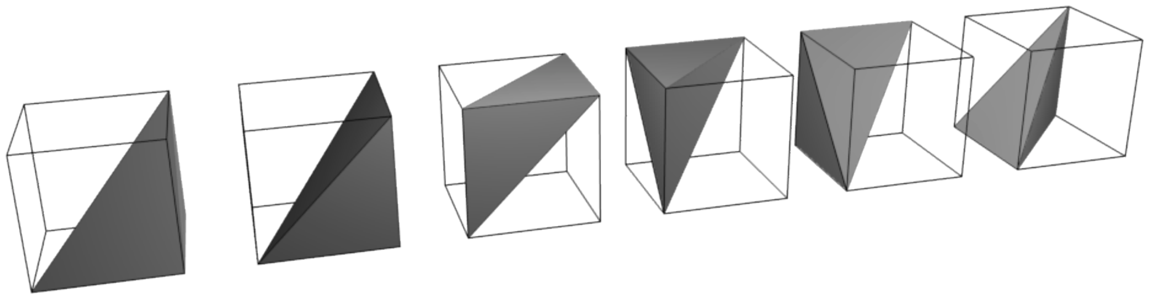


Figure 2-11: Tetrahedral decomposition of cube

Interpolation is used along tetrahedron edges to find the isosurface's point of intersection. The vertex normal is found by taking the gradient of the leaf cell's distance field at the chosen position; vertices shared between cells are given an averaged normal.

To reduce output geometry size, the algorithm produces indexed geometry: vertices are stored only once and triangles are defined as a set of three indices into the vertex buffer. The algorithm keeps track of neighbouring cells to merge shared vertices. Before creating a new vertex in a given leaf cell edge, we check that the vertex has not yet been created by any of the neighbours which may share it. Examples of shared edges are shown in Fig. 2-12.

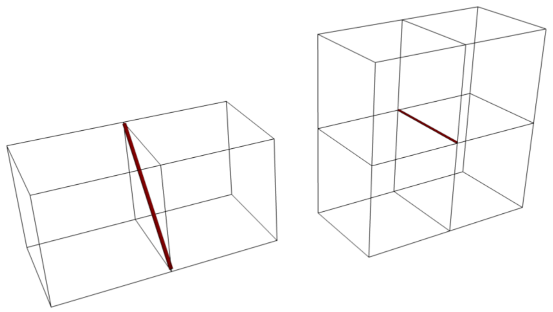


Figure 2-12: Edges shared by two and four neighbors

Because the ASDF contains leaf cells of various sizes, the resulting mesh is effectively decimated: large leaf cells become large triangles, and small leaf cells become small triangles. Figure 2-13 shows a triangulated model. No additional decimation was performed on this mesh; the large triangles are simply the triangulation of large ASDF leaf cells.

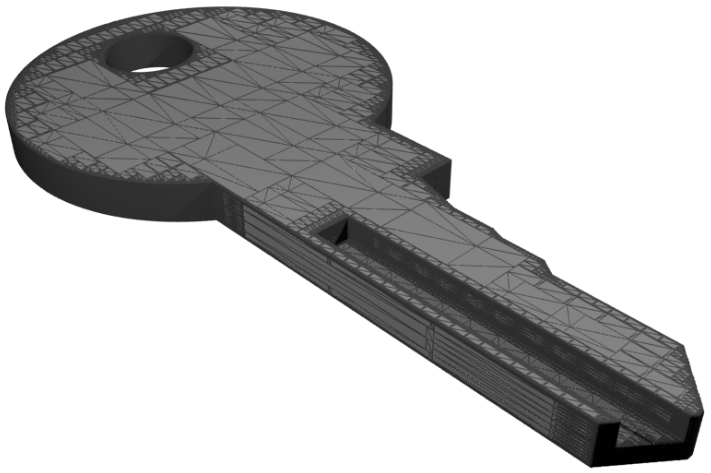


Figure 2-13: Triangulated mesh, showing decimation

Note that meshes generated with this strategy may not be water-tight. In particular, there will be zero-area cracks in between leaf cells at differing depths, visually explained in Fig. 2-14. As the mesh is only being used for visualization purposes, these cracks are not a concern. If watertight meshing is required, there are many relevant algorithms in the literature (e.g. [23]).

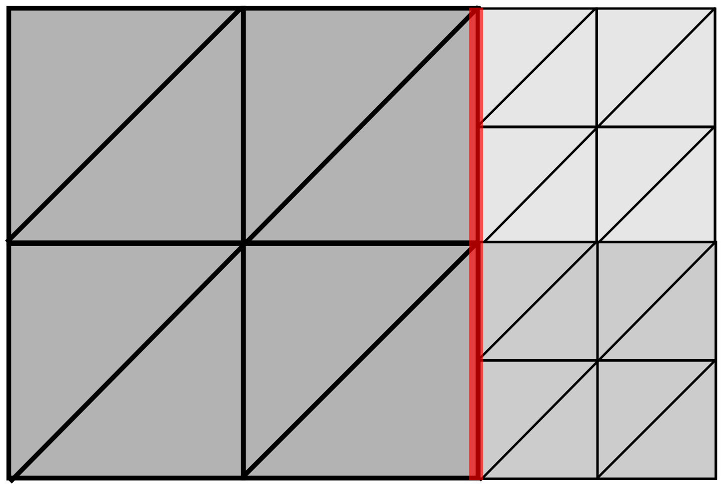


Figure 2-14: Zero-area crack on multi-scale intersection

Marching tetrahedrons is also simple to parallelize, as each cell is evaluated independently. Our parallel implementation uses eight threads to evaluate the eight branches of the top-level octree. The resulting vertex and index buffers are concatenated (and indices are modified to account for the offset of their vertices within the larger buffer). Note that indexed geometry is not shared between independent threads; vertices on the edges between octants may be duplicated in the resulting vertex buffer.

Figure 2-15 shows triangulation time on the key model from Fig. 2-13. The time taken appears to be sublinear when plotted against voxel count, and multithreading gives a 2.5× improvement.

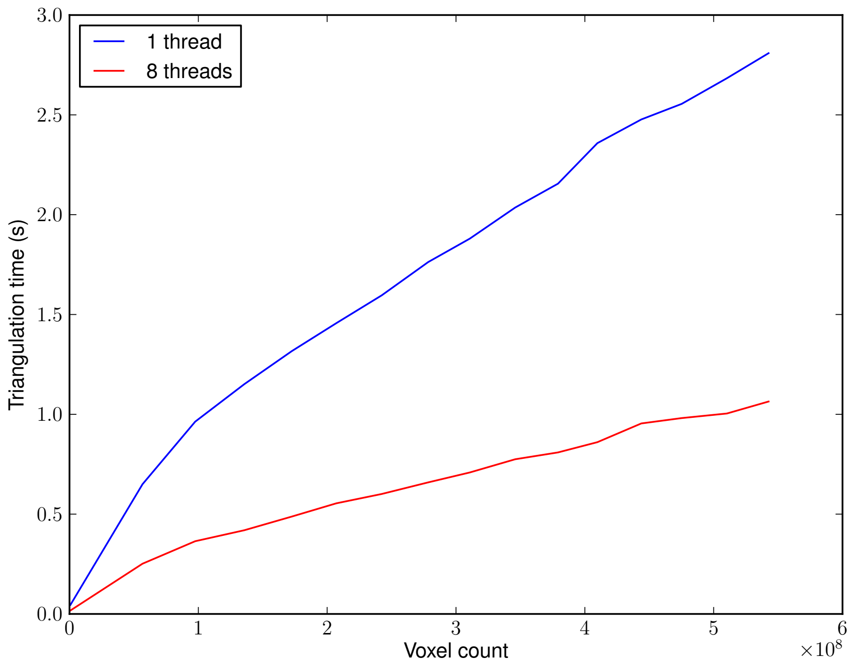


Figure 2-15: Triangulation time

Chapter 3

Functional Representations

3.1 Motivation

Though ASDFs are a suitable representation for solid objects, they do not lend themselves to direct design. A higher level of abstraction is desirable; ideally, a higher-level representation that easily populates ASDF cell corners.

This chapter introduces the use of functional representations (or F-reps) as a backend for CAD tools. A functional representation defines a solid model as an implicit surface that takes in coordinates and returns a value, i.e. $f(x, y, z) \to \mathbb{R}$. Values greater than zero are outside of the object; equal to zero are on the surface of the object; and less than zero are inside.

Functional representations have several advantages over boundary representations. When working with boundary representations, rendering is easy and computational solid geometry is algorithmically challenging. F-reps have the opposite properties: rendering is computationally intensive, but CSG is trivial. Given the trends in computing power, computational difficulty is preferable to algorithmic difficulty. F-reps also have arbitrarily high resolution, as the expression can be evaluated at as many points as desired. Finally, there exists an efficient rendering strategy based on spatial subdivision which dovetails with ASDF creation and population.

3.2 Prior work

The use of f-reps for design dates back to the 1970s. The Hyperfun project is one of the earliest such projects [14]; it defines a standard language for describing f-rep based shapes and provides plugins for rendering f-reps in Pov-ray (an open-source raytracing package).

In recent years, individuals from the Hyperfun team have created a commercial plug-in for Rhino named Symvol™[47], which uses f-reps as a flexible backend for computational solid geometry. Finally, the Shapeshop project allows users to interactively sculpt three-dimensional shapes by drawing lines and curves; again, f-reps are used as a backend for CGS [45].

These projects are similar in that they are purely design tools; they do not integrate into a complete CAD/CAM workflow without first exporting to boundary representations. Furthermore, they each impose different limitations on workflows in order to create a particular user experience. The f-rep engine described in this chapter is both very general and designed for use in a complete CAD/CAM workflow.

3.3 Operations

There is a wealth of literature on modeling with implicit surfaces, dating back to the 1970s and continuing up into the present day [44, 41, 6, 18]; this section gives an overview of common operations that can be performed on distance-metric implicit surfaces.

Computational solid geometry can be performed using the `min` and `max` operations, as shown in Table 3.1.

CSG Operation	Implementation
$A \cup B$	`min(A,B)`
$A \cap B$	`max(A,B)`
$\bar{A}$	`-A`

Table 3.1: Basic CSG operators

Coordinate transforms allow for more sophisticated deformations, including scaling, tapering, shearing, etc [4]. A few basic coordinate transforms are listed in Table 3.2. More sophisticated transforms including shears and tapers were also implemented in the library of standard functions.

Operation	Implementation
Translation	$(x, y, z) \to (x - x_0, y - y_0, z - z_0)$
Scaling about origin	$(x, y, z) \to (x/s_x, y/s_y, z/s_z)$
Rotation about z axis	$(x, y, z) \to (\cos\alpha x + \sin\alpha y, -\sin\alpha x + \cos\alpha y, z)$

Table 3.2: Sample coordinate transforms

All of the operations listed above ignore distance metric values; they would work equally well on a Boolean function $f(x, y, z) \to [\texttt{True}, \texttt{False}]$. Using the distance metric opens up new possibilities for object combination and blending [39, 25]. A few examples are shown in Table 3.3, and the morph operator is demonstrated in Fig. 3-1.

Operation	Implementation
Blend	$\texttt{min}(\texttt{min}(\texttt{A}, \texttt{B}), \sqrt{\lvert\texttt{A}\rvert} + \sqrt{\lvert\texttt{B}\rvert} - r)$
Morph	$d \times \texttt{A} + (1 - d) \times \texttt{B}$
Shell	$\texttt{max}(\texttt{A} - t/2, -t/2 - \texttt{A})$

Table 3.3: Multi-object combination

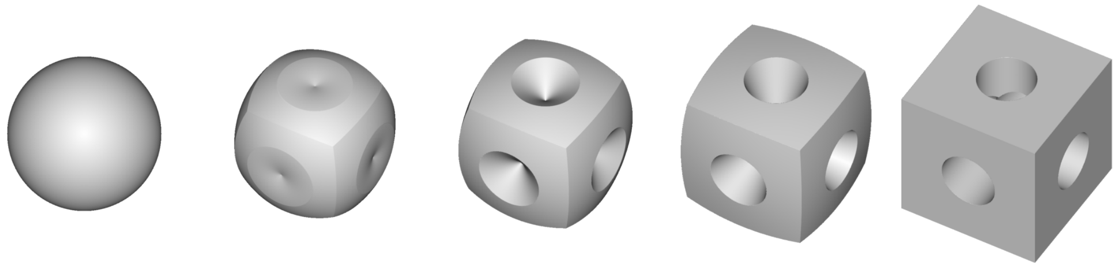
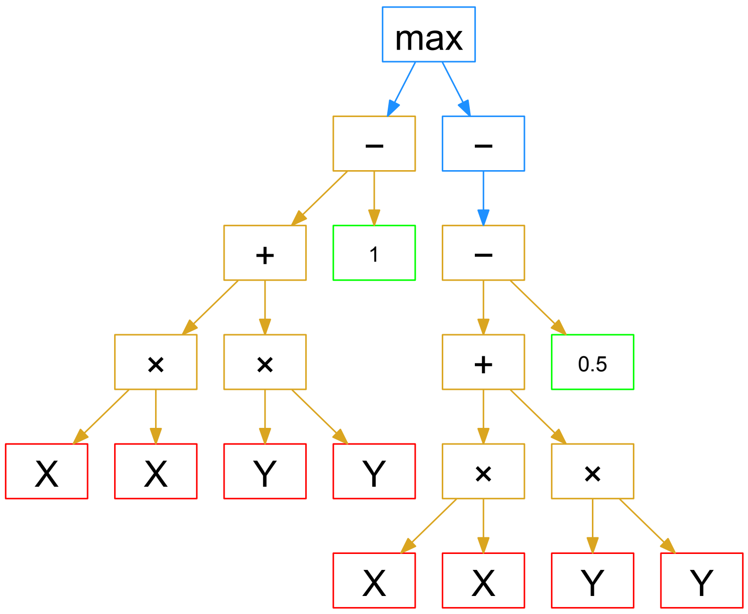


Figure 3-1: Smooth morphing between two shapes

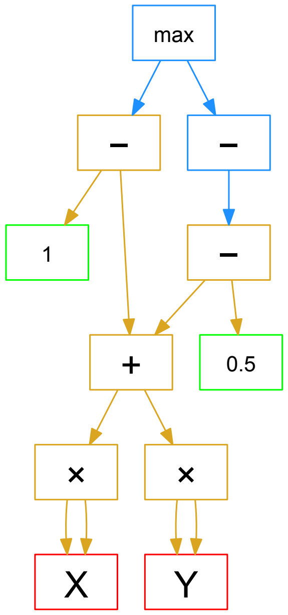
3.4 Graph representation

An f-rep expression can be viewed as a directed acyclic graph of operators. Each operator has a fixed number of arguments which point to either other operators, numerical constants, or the x, y, and z position variables.

The evaluation time for such a graph is proportional to the number of nodes. To make evaluation more efficient, duplicate nodes should be merged. Consider the expression `max(X*X+Y*Y-1, -(X*X+Y*Y-0.5))`, representing a annulus. Without deduplication, it is described by the graph shown in Fig. 3-2a; with deduplication, it is described by the sparser graph in Fig. 3-2b.



(a) Original tree



(b) Deduplicated tree

Figure 3-2: Effects of deduplication

3.5 Language design & parser

For ease of use, the f-rep workflow accepts strings and parses them into expression graphs. This enables simple design tools which perform basic string manipulation then pass expressions to the geometry engine. The chosen syntax is a sparse, prefix-notation language (with details described in Appendix A). The use of prefix notation allows the parser to be implemented with basic recursion, as shown in Alg. 3.1.

The parser performs deduplication on incoming clauses. Graph nodes are cached in arrays sorted by operation (e.g. `min`, `max`) and tree depth (defined as maximum distance to a constant, x, y, or z node). The DEDUPLICATE procedure checks for

Algorithm 3.1 Parser pseudo-code

```
function GetToken(x, y, z)
    Read next token t from input stream
    if t is an atom then
        Construct node n from t
    else if t is a map operation then
        x'←GetToken(x, y, z)
        y'←GetToken(x, y, z)
        z'←GetToken(x, y, z)
        n←GetToken(x', y', z')
    else
        Call GetToken to get appropriate number of arguments
        Construct node n from t and arguments
    end if
    n ← Deduplicate(n)
    return n
end function
```

a node with matching opcode, rank, and arguments. If none is found, then the argument is saved in the cache and returned; otherwise, the argument is freed and the cached node is returned instead.

As defined above, the map operator applies a coordinate transform $m(x, y, z) = (x', y', z')$ to another expression. Naïvely, map operations can be implemented using find-and-replace: to apply the coordinate transform $x' = y, y' = x$, simply replace all instances of `X` in the expression with `Y` and vice versa. However, creating a specific operator has a major advantages. The naïve implementation increases the size of a string by $O(n_xL_x + n_yL_y + n_zL_z)$, where $n_{x,y,z}$ is the number of occurrences of `X`,`Y` or `Z` and $L_{x,y,z}$ is the length of each symbol's replacement. This leads to a geometric increase in string size. Using the dedicated map operator increases the string length by only $O(L_x + L_y + L_z)$, changing the growth rate from geometric to arithmetic and reducing parser load.

3.6 Packed data structure

When looking at expression trees, one's first instinct is to evaluate them recursively. However, simple recursive evaluation invites a host of problems related to caching.

The evaluator should ensure that each node is only evaluated once, so a cached result must be stored. When nodes are activated and deactivated (as discussed in Section 3.8), cache invalidation becomes challenging.

Instead, our implementation evaluates trees from the bottom up. Each node is assigned a rank, where constants are rank -1; x, y, and z are rank 0; and all other nodes are one greater than their largest child's rank. Nodes are then "packed" into rows (implemented as arrays of pointers) by rank order. A visualization of a graph "packed" in this manner can be seen in Fig. 3-3.

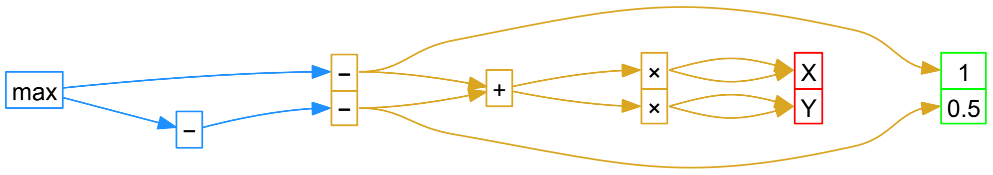


Figure 3-3: Packed tree

Evaluation proceeds up the rows in increasing rank order, with each node storing the result of its operation. This ordering can be interpreted as a topological sort on the graph nodes [12, Chapter 22]. Evaluation on the graph above would first solve `X` and `Y` (storing results locally), then `X*X` and `Y*Y` (looking up the stored results from `X` and `Y`), then `X*X+Y*Y`, and so on (the constants are not evaluated, as their values are unchanging).

3.7 Render strategy

Interval arithmetic [35] is used to efficiently render a math expression into an ASDF or height-map image, as described in [15]. Pseudo-code for the evaluation is shown in Alg. 3.2.

Evaluating the tree on an interval produces upper and lower bounds for the result. If the upper bound is below zero, then this region is unambiguously filled; similarly, if the lower bound is above zero, then the region is unambiguously empty. In ambiguous cases, the region is divided and Render is called recursively.

Recall that the region is a discrete lattice, rather than a continuous spatial region.

As such, single-voxel regions are not subdivided. Finally, the function OUTPUT is a placeholder that depends on the output format (i.e. height-map, ASDF).

Algorithm 3.2 Evaluation pseudo-code

```
function RENDER(tree, region)
    if region is of volume 1 then
        OUTPUT(region, LEAF)
    else
        $d \leftarrow$ EVALUATEINTERVAL(tree, region)
        if $d$.upper $< 0$ then
            OUTPUT(region, FILLED)
        else if $d$.lower $>= 0$ then
            OUTPUT(region, EMPTY)
        else
            OUTPUT(region, BRANCH)
            PRUNETREE(tree)
            Subdivide region into subregions
            Call RENDER(tree, subregion) on each subregion
            UNPRUNETREE(tree)
        end if
    end if
end function
```

3.8 Tree pruning

Evaluating the entire math expression at every point is inefficient, as there are often sections of the tree that can be skipped.

Consider evaluating two expressions `A` and `B` on a region `R`, using interval arithmetic as discussed above. Assume the results are `A(R) = (5,10)` and `B(R) = (-3, 3)`. Because `B`'s value is strictly below that of `A` in this region, `min(A,B) = B` for any region contained within `R`. Therefore, when recursing on subregions within `R`, `A` can be ignored and should not be evaluated.

Tree pruning is implemented as a single pass from the root to the leafs of the packed tree, with pseudocode given in Alg. 3.3. Each node has a Boolean mark variable. If this mark is true when the node is reached, then the node is disabled.

Nodes are disabled by swapping them to the back of their row and decrementing the count of active nodes in that row. The total number of nodes disabled for each row and recursion level is stored. This allows the system to re-activate them when popping out to a shallower recursion level.

Algorithm 3.3 Tree pruning

```
function PruneTree(tree)
    Mark every node in the tree
    Unmark the root node of the tree
    for each row in the tree, from root to leafs do
        Initialize disabled node count to 0 for this row and recursion level
        for each node N in this row do
            A ← N's first argument
            B ← N's second argument
            if N is marked then
                Swap N to the back of this row's array
                Decrement active node count of this row
                Increment disabled node count
            else if N is a max operator then
                if A.result $\leq$ B.result then
                    Unmark B
                else if A.result $\geq$ B.result then
                    Unmark A
                else
                    Unmark A and B
                end if
            else if N is a min operator then
                if A.result $\geq$ B.result then
                    Unmark B
                else if A.result $\leq$ B.result then
                    Unmark A
                else
                    Unmark A and B
                end if
            else
                Unmark A and B
            end if
        end for
    end for
end function
```

In certain rendering modes, only the occupancy value of the result matters. When rendering a height-map lattice, only occupancy matters; in contrast, when constructing a distance field, the actual sample value is needed. When only occupancy matters, a second pruning pass can deactivate a larger set of nodes. To explain this second pass, we will define the term "Boolean" in the context of an numerical interval to be true if zero is not contained within this interval, i.e. an interval $A \mid 0 \notin A$. A Boolean interval can be interpreted as an interval without ambiguity in regards to occupancy.

The Boolean pruning pass checks two conditions to determine if a node can be disabled. First, the intermediate result stored in that node must be a Boolean interval. Second, that node must be connected to the root of the tree only by nodes where tightening of the intermediate interval will not change the top-level result from non-Boolean to Boolean, i.e. nodes with operators in the set

$$\{f \mid \forall a \in \{\text{set of Boolean intervals}\} \nexists a' \subseteq a \text{ such that } 0 \in f(a) \text{ and } 0 \notin f(a')\}$$

This ensures that nodes aren't disabled when ambiguity can be resolved by tightening the interval.

To illustrate the second condition, constrast `max(A, B)` with `A + B`. If `A` results in a Boolean interval, then tightening that interval won't change the result of `max(A, B)` from non-Boolean to Boolean – at this point, that is determined by `B`. In contrast, tightening the interval of `A` *could* change `A + B` from non-Boolean to Boolean:

`A`	`B`	`max(A, B)`	`A+B`
$[-4, -1]$	$[-2, 2]$	$[-2, 2]$	$[-6, 1]$ (not Boolean)
$[-4, -3]$	$[-2, 2]$	$[-2, 2]$	$[-4, -1]$ (Boolean)

Therefore, `A` could be disabled if it appears in `max(A, B)`, but not if it appears in `A + B`.

Pseudo-code for this pruning pass is shown in Alg. 3.4. This pseudo-code assumes that the disabled node count has already been initialized (i.e. by Alg. 3.3). The pseudo-code re-uses the same mark variable; in this case, it refers to whether or not a node meets the second condition for pruning discussed above.

The disabled node count is used for unpruning, which is shown in Alg. 3.5. In a single pass through the rows, the active node count is increased by the number of nodes

Algorithm 3.4 Boolean tree pruning

```
function PruneTreeB(tree)
    Mark every node in the tree
    for each row in the tree, from root to leafs do
        for each node N in this row do
            if N is marked and N's result interval is Boolean then
                Swap N to the back of this row's array
                Decrement active node count of this row
                Increment disabled node count
            else if N is not marked or N is an operator other than
                    max, min, neg, mult then
                Unmark N's arguments.
            end if
        end for
    end for
end function
```

disabled in the previous PruneTree (and possibly PruneTreeB) operation(s).

Algorithm 3.5 Tree unpruning

```
function UnpruneTree(tree)
    for each row in the tree do
        $c \leftarrow$ disabled node count for this row and recursion level
        Increase active node count for this row by $c$
    end for
end function
```

Figure 3-4 shows the effects of this tree pruning, tested on the f-rep font showed in Fig. 3-5. The x axis is render recursion level: the full region is level 0, the first subdivision of the region is level 1, the next subdivision is level 2, etc. The y axis shows the average number of nodes active at that level.

The results are dramatic: at the lowest level of recursion, there are on average 200$\times$ fewer active nodes. As expression evaluation time is directly proportional to the number of active nodes in the tree, this provides a significant improvement in running time; this gain is compounded by the fact that there are geometrically more regions to be evaluated as recursion level increases.

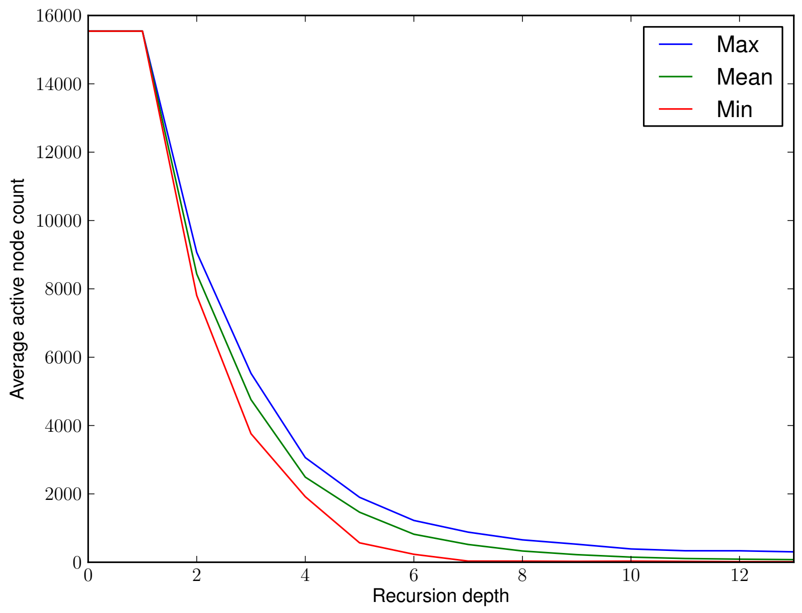
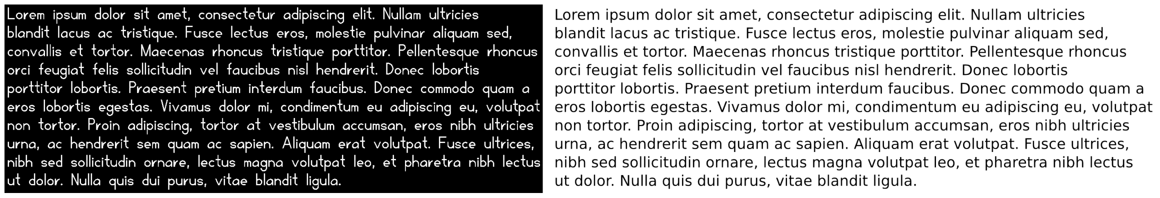


Figure 3-4: F-rep pruning in action

3.9 Scaling behavior

To examine scaling behavior, we compared the performance of an f-rep based font to a similar vector font. The chosen text was ten lines of "lorem ipsum", a standard placeholder text string commonly used in design templates, and the two fonts can be seen in Fig. 3-5.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam ultricies
blandit lacus ac tristique. Fusce lectus eros, molestie pulvinar aliquam sed,
convallis et tortor. Maecenas rhoncus tristique porttitor. Pellentesque rhoncus
orci feugiat felis sollicitudin vel faucibus nisl hendrerit. Donec lobortis
porttitor lobortis. Praesent pretium interdum faucibus. Donec commodo quam a
eros lobortis egestas. Vivamus dolor mi, condimentum eu adipiscing eu, volutpat
non tortor. Proin adipiscing, tortor at vestibulum accumsan, eros nibh ultricies
urna, ac hendrerit sem quam ac sapien. Aliquam erat volutpat. Fusce ultrices,
nibh sed sollicitudin ornare, lectus magna volutpat leo, et pharetra nibh lectus
ut dolor. Nulla quis dui purus, vitae blandit ligula.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam ultricies
blandit lacus ac tristique. Fusce lectus eros, molestie pulvinar aliquam sed,
convallis et tortor. Maecenas rhoncus tristique porttitor. Pellentesque rhoncus
orci feugiat felis sollicitudin vel faucibus nisl hendrerit. Donec lobortis
porttitor lobortis. Praesent pretium interdum faucibus. Donec commodo quam a
eros lobortis egestas. Vivamus dolor mi, condimentum eu adipiscing eu, volutpat
non tortor. Proin adipiscing, tortor at vestibulum accumsan, eros nibh ultricies
urna, ac hendrerit sem quam ac sapien. Aliquam erat volutpat. Fusce ultrices,
nibh sed sollicitudin ornare, lectus magna volutpat leo, et pharetra nibh lectus
ut dolor. Nulla quis dui purus, vitae blandit ligula.

Figure 3-5: F-rep (left) and vector (right) depictions of lorem ipsum text

As designs become more complex, the representation should grow at a manageable rate. Figure 3-7 compares the scaling of f-rep and vector font objects as more lines are added to the lorem ipsum string. Grown rate is examined through two comparisons. The first comparison examines the number of vector nodes (shown in Fig. 3-6) and math-tree clauses; the second compares the length of the math string representation and the size of the `.svg` file.

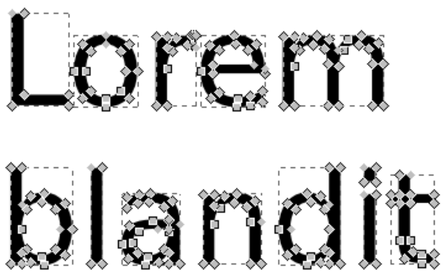


Figure 3-6: Close-up of vector font, showing nodes

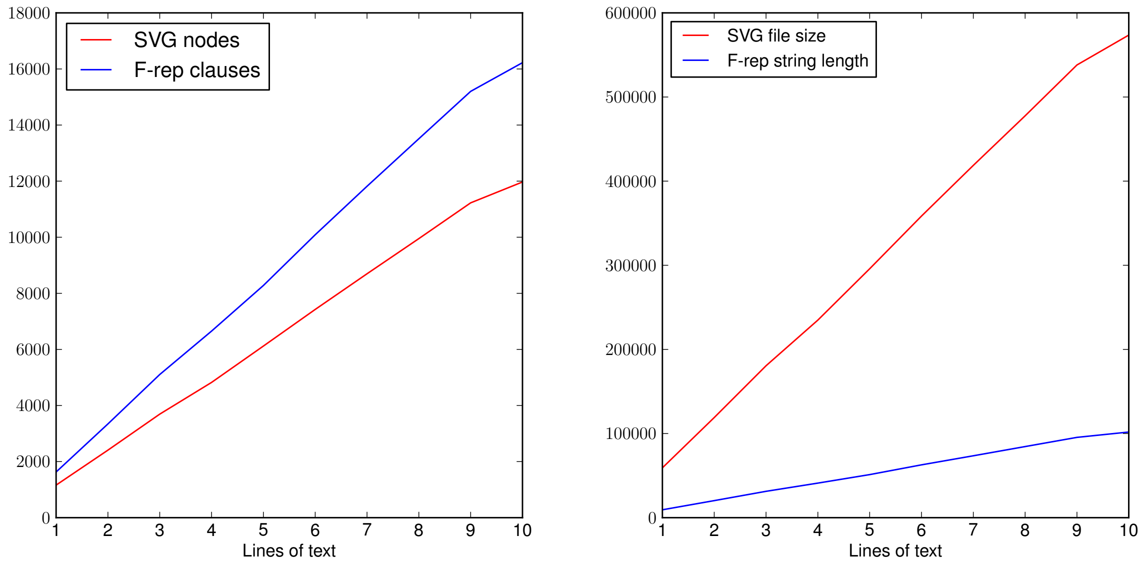


Figure 3-7: Scaling behavior on lorem ipsum strings

Fair comparisons of absolute values are not meaningful; the point of interest is ratios. For both comparisons, the ratio is nearly constant: the number of math clauses is $\approx 1.4\times$ the number of vector nodes, and the `.svg` files are $\approx 5.7\times$ the size of the equivalent math string. In this example, the data indicates that f-reps grow at an equivalent rate to more traditional representations.

3.10 Bitmap Rendering

In certain workflows, f-rep expressions are rendered directly to bitmap images (rather than to volumetric representations) for display on the screen and toolpath generation. This section examines f-rep to height-map performance, rendering models at varying resolutions and with varying numbers of threads. Multithreading is implemented by dividing the spatial region and assigning each thread to its own region.

First, we examine the two-dimensional lorem ipsum example discussed throughout this chapter. A graph showing render time as a function of voxel count appears in Fig. 3-8. Render time appears roughly linear with voxel count. Adding threads decreases render time at what appears to be a logarithmic rate.

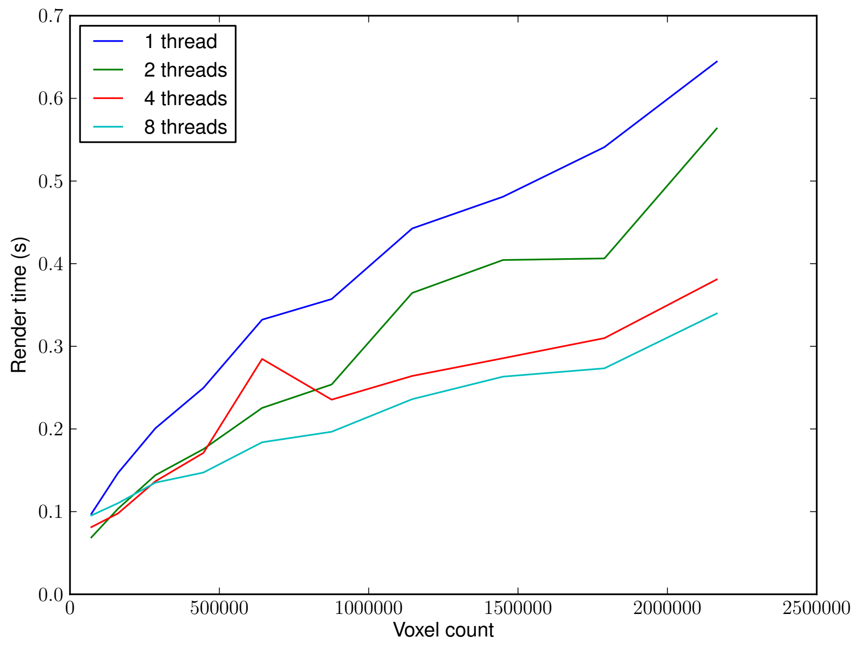


Figure 3-8: Speed performance of lorem ipsum text

Next, we consider a three-dimensional mold design, shown in Fig. 3-9. This model is rendered as a height-map, taking advantage of depth-based culling as well as tree pruning strategies discussed above.

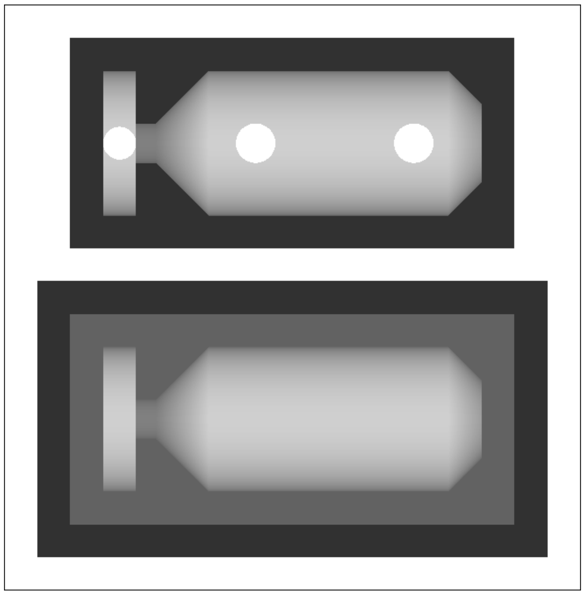


Figure 3-9: Mold height-map

Rendering time is shown in Fig. 3-10. As with the text, render speed is roughly linear with voxel count and decreases logarithmically with more threads.

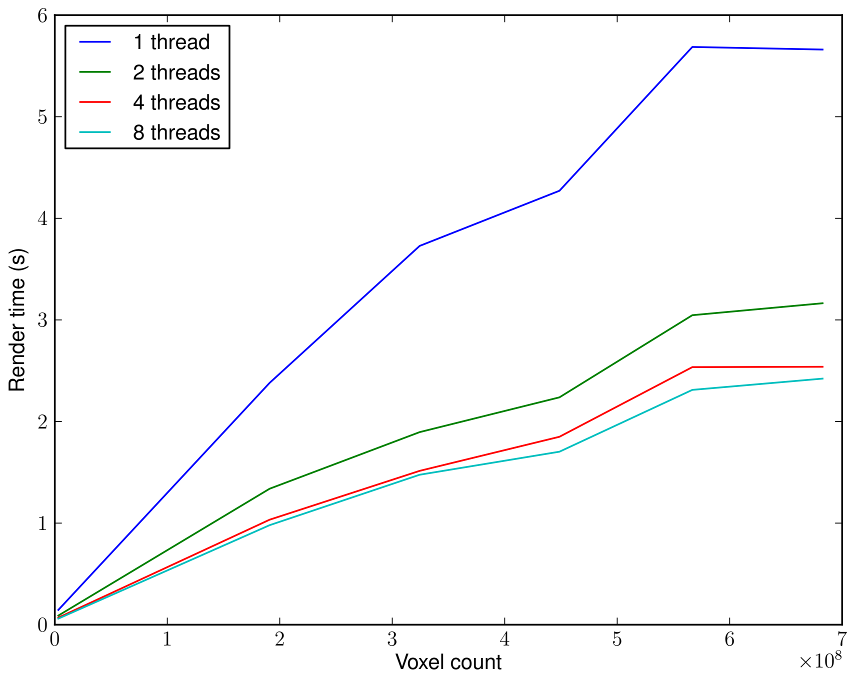


Figure 3-10: Speed performance of mold

Chapter 4

Design Tools

4.1 Motivation

The use of f-reps as a geometry backend requires an appropriate user interface, as users expect a higher level of abstraction than raw math expressions in their design tools. This chapter presents a pair of design tools that use f-reps as a backend; this illustrates the argument that a simple, clean geometry engine allows for rapid prototyping and iteration of design tools themselves.

The first tool, `fabserver`, is a web-based tool that represents solids as graphs of information flow; the real-space positions of nodes modify model parameters. The second, `kokopelli`, uses Python as a hardware description language, allowing for metaprogramming and encoded design intent.

4.2 `fabserver`

`fabserver` is a design tool written by Neil Gershenfeld as part of the fab modules [34]. It runs in modern web browsers using JavaScript and SVG for rendering. Computational geometry is offloaded to a separate server. Users design models as a graph of information flow, where the real-space coordinates of nodes become parameters in the model.

4.2.1 Client-server model

`fabserver` is implemented with a client-server model, illustrated in Figure 4-1.

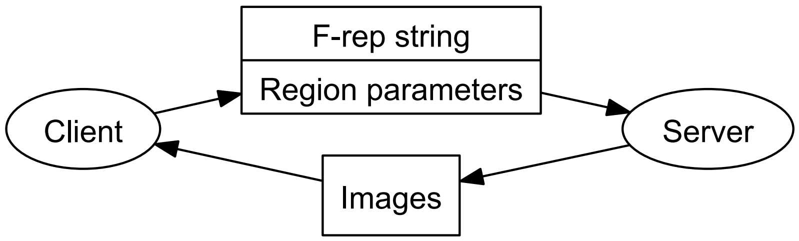


Figure 4-1: `fabserver` client-server model

The client passes f-rep strings and descriptions of the render bounds to a server, which renders the expression and returns bitmap images to the client. The server can be run locally, but this model also allows for decentralization and "cloud CAD", where dedicated servers provide the computing horsepower for model rendering and the client's operations are limited to string manipulation and bitmap display.

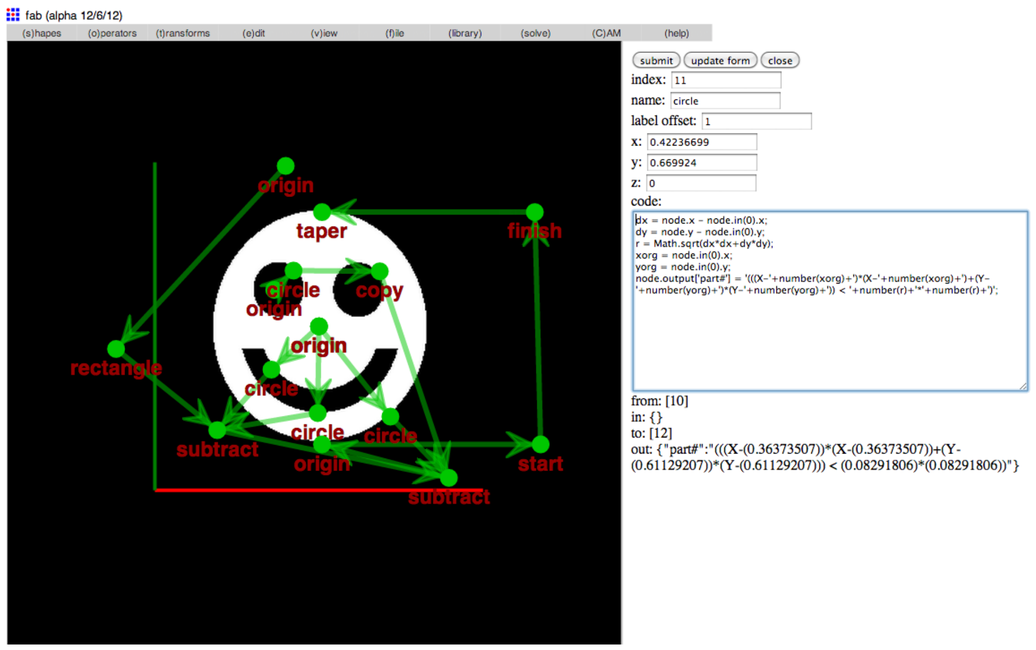
4.2.2 Design philosophy

Figure 4-2 shows `fabserver` running in a browser window. Each green node propagates some amount of information to its children; the information may include position and math expressions among other parameters. In this example, the circle node takes in the position of the previous node in the graph, calculates a radius, then outputs a math string representing a circle.

Nodes are fragments of JavaScript code, so the tool is easily extensible. Users can modify the code in provided nodes to change their behavior, or write their own custom nodes to add new functionality.

4.3 `kokopelli`

`kokopelli` was written as both a platform for experimentation and as a practical design tool. It uses Python as a hardware description language for solid models, defining standard libraries of shapes and transform that in turn manipulate f-rep



index: 11
name: circle
label offset: 1
x: 0.42236699
y: 0.669924
z: 0
code:

```
dx = node.x - node.in(0).x;
dy = node.y - node.in(0).y;
r = Math.sqrt(dx*dx+dy*dy);
xorg = node.in(0).x;
yorg = node.in(0).y;
node.output['part#'] = '(((X-'+number(xorg)+')*(X-'+number(xorg)+')+(Y-
'+number(yorg)+')*(Y-'+number(yorg)+')) < '+number(r)+'*'+number(r)+')';
```

from: [10]
in: {}
to: [12]
out: {"part#":"(((X-(0.36373507))*(X-(0.36373507))+(Y-(0.61129207))*(Y-(0.61129207))) < (0.08291806)*(0.08291806))"}

Figure 4-2: `fabserver` window with simple model

strings. Users interact with this higher level of abstraction, rather than directly manipulating f-rep strings; the full power of Python and its extension libraries can be used in designing solid models.

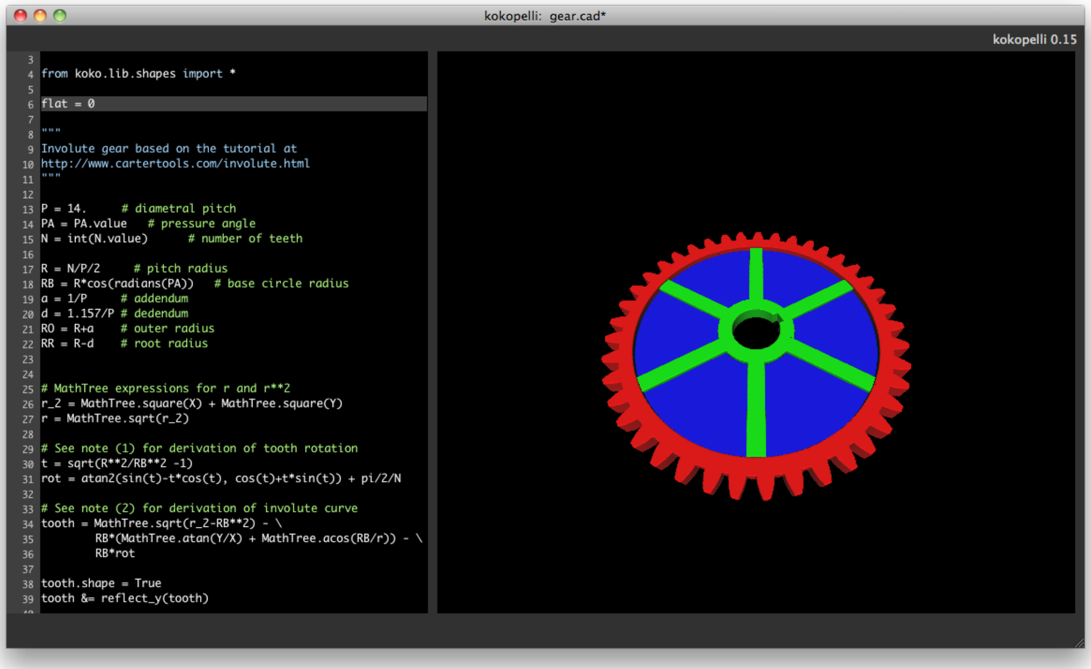
4.3.1 Similar work

The notion of an f-rep language dates back to the Hyperfun project [14, 40], which defines a language for f-rep based models. Other examples of scripting-based CAD include OpenSCAD [26], DesignScript [3], and ImplicitCAD [38].

The first two design environments use boundary-representations (with the Shape-Manager and CGAL kernels respectively); the latter uses implicit surfaces but is quite slow at present. In addition, all but ImplicitCAD use custom languages (without the widespread usage and extension libraries of Python); ImplicitCAD uses Haskell, which is less widespread as Python.

4.3.2 User Interface

The user interface in `kokopelli` is inspired by Bret Victor's talk titled "Inventing on Principle" [48]. It includes two main panels, shown in Fig. 4-3. One panel shows the object's source code; the other shows a rendering of the object (either as height-map or shaded solid). A set of callbacks re-renders the visualization when the code changes.



```

from koko.lib.shapes import *

flat = 0

"""
Involute gear based on the tutorial at
http://www.cartertools.com/involute.html
"""

P = 14.      # diametral pitch
PA = PA.value   # pressure angle
N = int(N.value)      # number of teeth

R = N/P/2     # pitch radius
RB = R*cos(radians(PA))   # base circle radius
a = 1/P      # addendum
d = 1.157/P # dedendum
RO = R+a     # outer radius
RR = R-d     # root radius


# MathTree expressions for r and r**2
r_2 = MathTree.square(X) + MathTree.square(Y)
r = MathTree.sqrt(r_2)

# See note (1) for derivation of tooth rotation
t = sqrt(R**2/RB**2 -1)
rot = atan2(sin(t)-t*cos(t), cos(t)+t*sin(t)) + pi/2/N

# See note (2) for derivation of involute curve
tooth = MathTree.sqrt(r_2-RB**2) - \
        RB*(MathTree.atan(Y/X) + MathTree.acos(RB/r)) - \
        RB*rot

tooth.shape = True
tooth &= reflect_y(tooth)
```

Figure 4-3: Gear model in `kokopelli`

4.3.3 Model refinement

`kokopelli` includes two rendering modes, each of which supports dynamic re-rendering based on the region of interest. In the first mode, objects are rendered as height-map images. When the user zooms and pans around the image, the f-rep design is re-evaluated based on the window bounds and zoom level so that is always pixel-for-pixel accurate to the viewport. This render operation takes place in a background thread and does not disrupt the UI.

The second rendering mode uses OpenGL to display triangulated meshes. F-reps are first converted to ASDFs, then triangulated using the strategies discussed in Section 2.8. In this render mode, re-rendering is more challenging: it is difficult to tell which parts of a model are closest to the camera.

Model refinement is decided experimentally. Each mesh includes a link back to its source data (which may be an f-rep expression or an ASDF file). In a hidden rendering pass, meshes are drawn with unique flat colors. These colors are then counted, pixel by pixel. If any mesh occupies a large fraction of the viewport, it is subdivided and refined. If a subdivided region occupies a small fraction of the viewport, it is collapsed. Figure 4-4 shows a cube being subdivided as we zoom in; each submesh is drawn with a unique flat color.

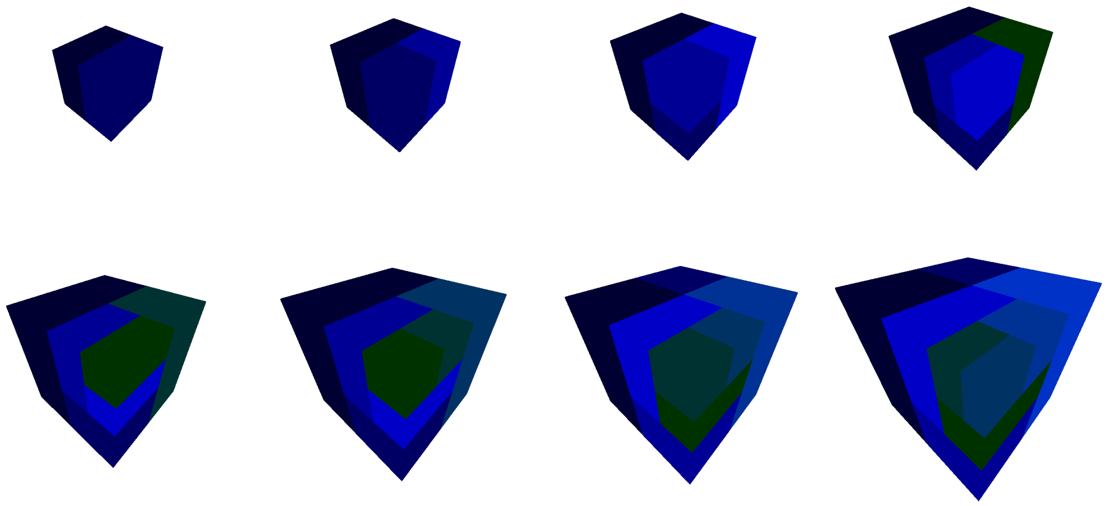


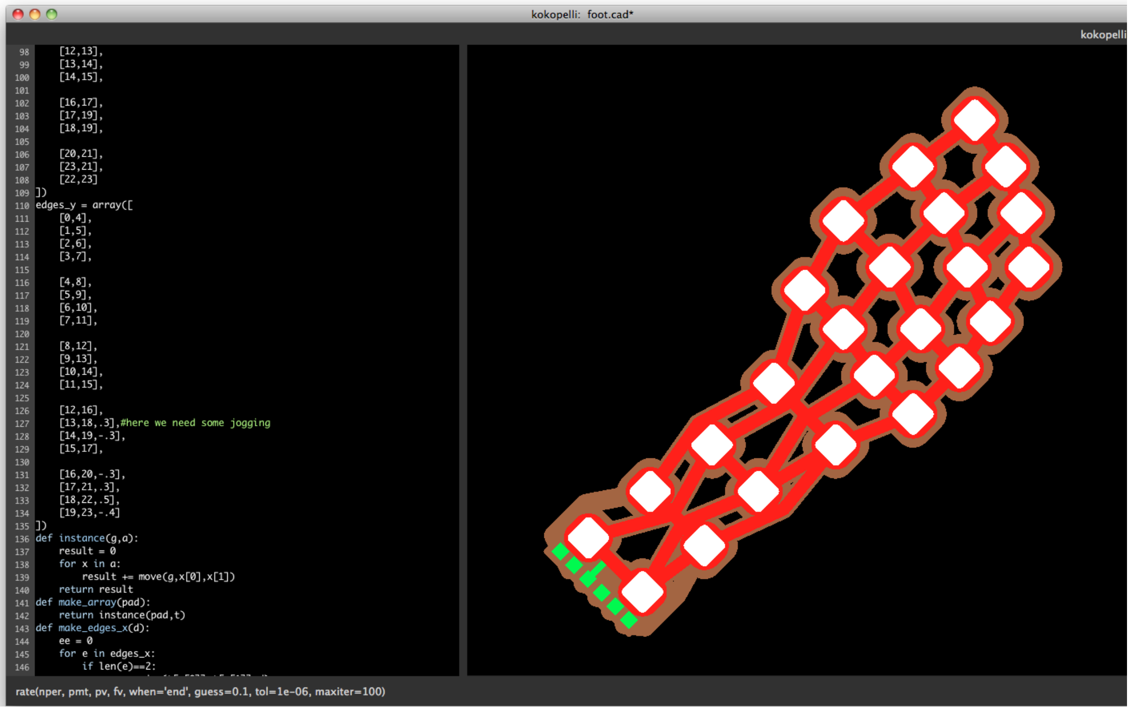
Figure 4-4: Subdivision meshes

This implementation of level-of-detail rendering is extremely flexible, as the refine operation depends on the mesh's source. For an f-rep, the refine operation renders subregions at a higher resolution; for a multi-scale ASDF file (as discussed in Section 5.4), it simply loads the next level-of-detail model from a file.

4.3.4 Extensibility

The use of Python allows designs to be hierarchical, modular, and functional. A standard library provides a set of basic shapes, but new shapes can be defined as simple functions that take arbitrary arguments. In the involute gear model from Fig. 4-3, the involute curve was defined as a custom function then used to generate the array of teeth.

In addition, there is a wealth of third-party libraries that extend Python in various ways. Figure 4-5 shows a design by Sam Calisch for a multi-point pressure sensor intended to go into a running shoe [10].



```
    [12,13],
    [13,14],
    [14,15],

    [16,17],
    [17,19],
    [18,19],

    [20,21],
    [23,21],
    [22,23]
])
edges_y = array([
    [0,4],
    [1,5],
    [2,6],
    [3,7],

    [4,8],
    [5,9],
    [6,10],
    [7,11],

    [8,12],
    [9,13],
    [10,14],
    [11,15],

    [12,16],
    [13,18,.3],#here we need some jogging
    [14,19,-.3],
    [15,17],

    [16,20,-.3],
    [17,21,.3],
    [18,22,.5],
    [19,23,-.4]
])
def instance(g,a):
    result = 0
    for x in a:
        result += move(g,x[0],x[1])
    return result
def make_array(pad):
    return instance(pad,t)
def make_edges_x(d):
    ee = 0
    for e in edges_x:
        if len(e)==2:
```

```
rate(nper, pmt, pv, fv, when='end', guess=0.1, tol=1e-06, maxiter=100)
```

Figure 4-5: Pressure sensor array in `kokopelli`

This design is notable because it uses NumPy [1] for efficient operations on arrays of points. The positions of sensor pads are defined as an array of measurements; pads and links are automatically generated with array operations.

4.3.5 Sample design

As a non-trivial proof-of-concept, Jonathan Ward's MTM A-Z PCB mill [49] was re-implemented as a parametric `kokopelli` file, shown in Fig. 4-6

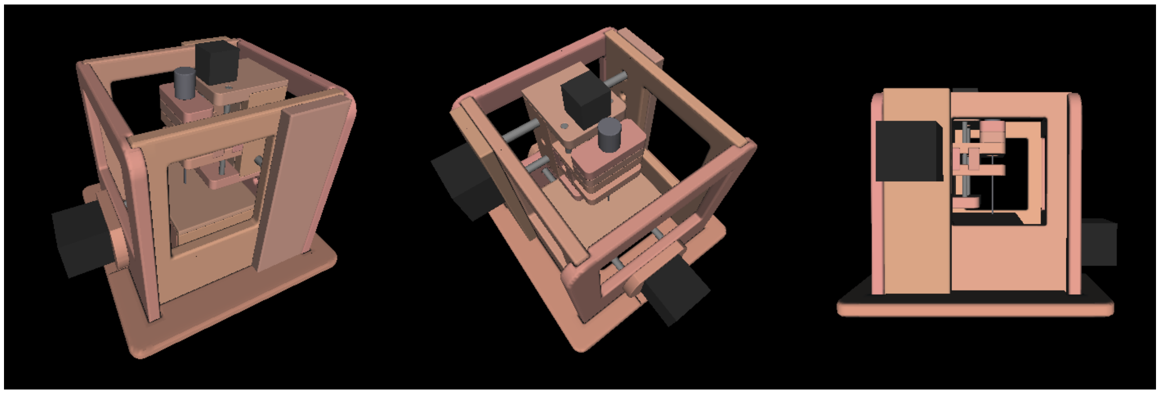


Figure 4-6: MTM A-Z PCB mill

The script-based design system allows extensive encoding of design intent. This model was parameterized by a set of variables, most notably sheet thickness and milling area. Modifying these variables creates plausible modified designs, as shown in Fig. 4-7.

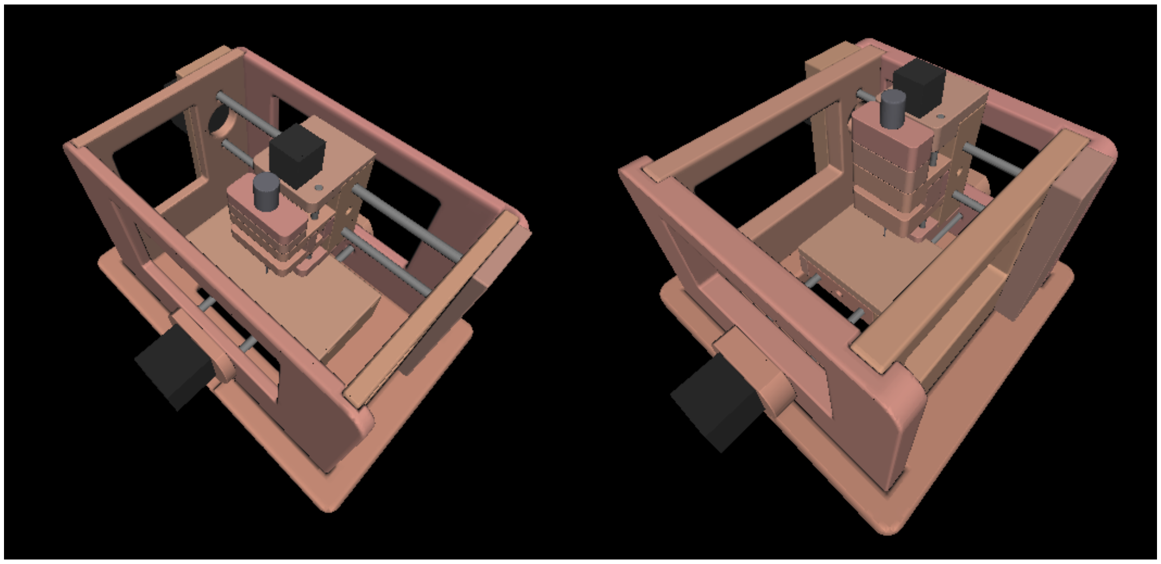


Figure 4-7: Modified MTM A-Z, with larger area (left) and thicker stock (right)

This model features thirty-five distinct parts: twenty-six model parts, three motor placeholders, and six rods. The model is regenerated in 0.55 seconds upon parameter modifications; the refinement process described above is critical in keeping initial render times low.

4.3.6 Design scaling

To examine scaling to many parts, the MTM A-Z model above was arrayed, as illustrated in Fig. 4-8. Each MTM machine instance has complexity described in Table 4.1.

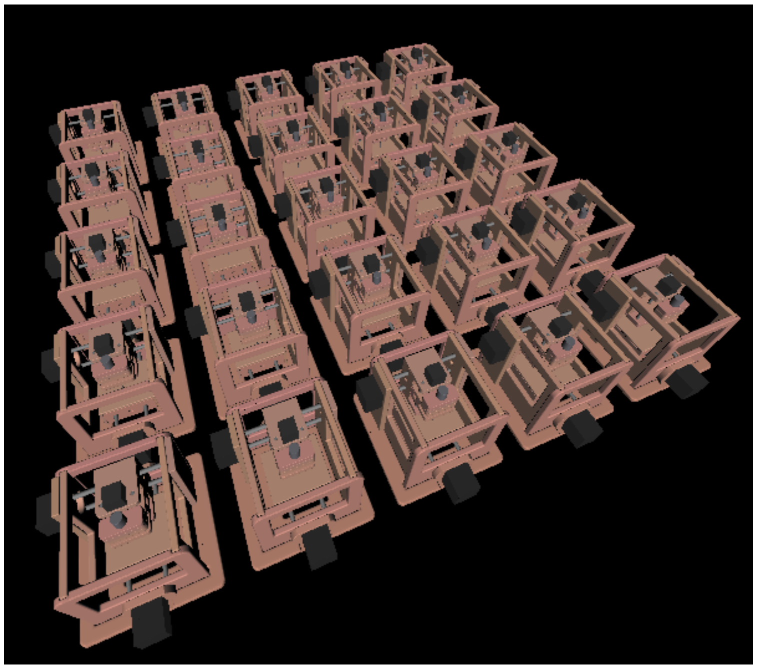


Figure 4-8: MTM machine array

Metric	Count
Parts	35
Math string length (bytes)	7401
Math tree nodes (deduplicated)	2167

Table 4.1: MTM model complexity metrics

Render time and memory usage are showed in Fig. 4-9. The system did not take advantage of instancing, so both render time and memory usage are linear with number of machines. In this demonstration, over one hundred separate parts are rendered in around two seconds. This is not unreasonably slow, but approaches the limits of user-perceived responsiveness. Further work in caching and instancing could be valuable for designing assemblies with hundreds of distinct parts.

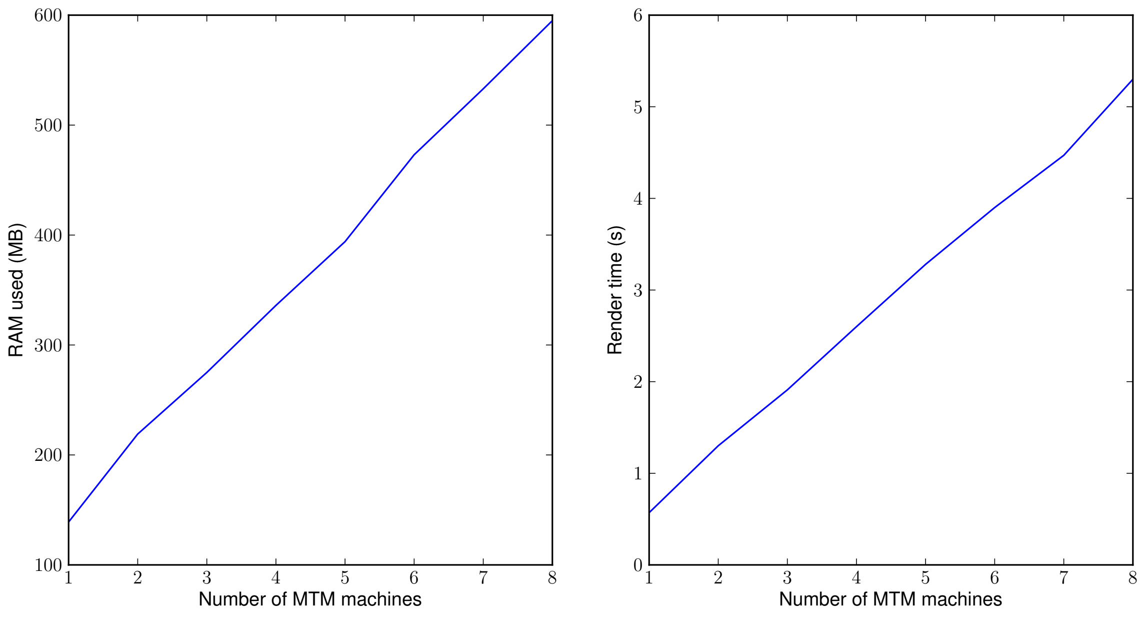


Figure 4-9: MTM array scaling

Chapter 5

Volumetric Scanning

5.1 Motivation

When digitizing physical objects, there are two primary strategies. Surface scanning uses structured light, lasers, etc. to produce a point cloud describing an object's surface. These points can then be stitched together into a mesh, often with color information, to produce a triangulated object model. In contrast, volumetric scanning (e.g. CT scanning) produces a volumetric data set with millions of object density samples stored on a three-dimensional lattice.

This chapter describes a strategy for importing volumetric scan data into our CAD/CAM workflow. The chosen import strategy uses density samples and a user-defined threshold to produces an "isosurface" ASDF.

This work is inspired by a collaboration with Jeff Koons, whose work often involves digitizing physical artifacts then recreating them out of unusual materials and at much larger scales. His team's existing workflow involves converting CT data into triangulated meshes, then laboriously post-processing the meshes to ensure that they are suitable for machining. This process takes hundreds of person-hours to complete for complex meshes. As such, a workflow that skips the triangulation step entirely is very appealing.

5.2 Import strategy

It has been observed that density samples are somewhat smoothed between neighboring voxels in CT scanning [24]. As such, when constructing the isosurface ASDF, density samples on minimum-sized voxels are interpreted as distance values from the user-defined threshold density. This allows the system to extract smoothly curving surfaces, rather than millions of voxel cubes.

The general algorithm to import density samples is shown in Alg. 5.1. It uses a "bottom-up" strategy where minimum-sized voxels are constructed then merged using SIMPLIFY (Alg. 2.1).

Algorithm 5.1 CT data import algorithm

```
function IMPORTRECURSIVE(data, region, threshold)
    if all values in region are below threshold then
        return an empty ASDF cell
    else if all values in region are above threshold then
        return a filled ASDF cell
    else if region is a single voxel then
        Construct a leaf-type ASDF named leaf
        Set leaf's d values to density samples (offset by threshold)
        return leaf
    else
        Divide region into subregions
        IMPORTRECURSIVE(data, subregion, threshold) for each subregion
        Construct an ASDF asdf that contains subtrees as branches
        SIMPLIFY(asdf)
        return asdf
    end if
end function
```

5.3 Performance

This form of isosurface construction produces efficient, sparse representations. It was tested on a piece of lace from a ceramic figurine, shown in Fig. 5-1. The original CT data is $255 \times 255 \times 511$ samples, with a voxel side of 0.1 mm.

Table 5.1 compares file sizes across a variety of representations. Compressed file

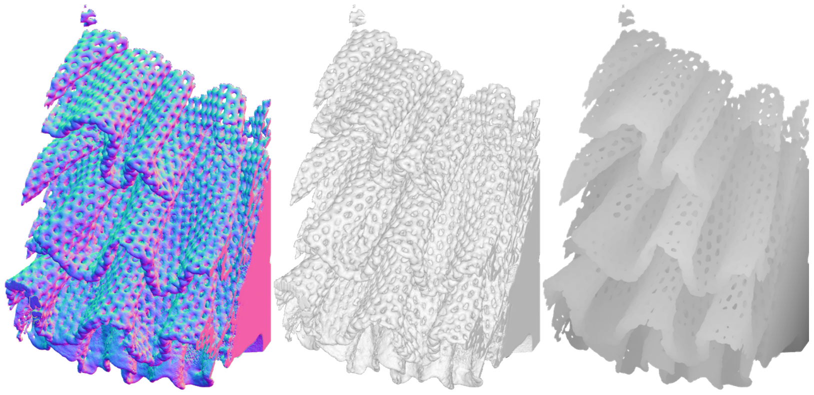


Figure 5-1: CT scanned lace (normals, shaded, and height-map)

sizes are also shown (to reduce dependance on the particular structure of each file format).

Description	File type	File size (MB)	Compressed size (MB)
Raw CT data	`.vol` float array	127	113
Naïve marching cubes	Binary `.stl`	194	25
Smoothed mesh	Binary `.stl`	30	27
ASDF isosurface	`.asdf` file	14	11

Table 5.1: File sizes in CT workflow

From these results, we conclude that that the CT to ASDF workflow is effective at reducing file size while preserving isosurface detail.

5.4 Large files and multi-resolution importing

Importing a 3.6 GB voxel data set (992x992x992) into an ASDF requires a significant amount of RAM. It was possible on a high-end workstation, but could be difficult for commodity hardware. By sampling a subset of density values, we can create a set of smaller ASDFs that exchange coverage region for sample count to maintain consistent file sizes.

The large file was converted into a three levels of ASDF files. The top-level file contained the entire region and used 1/64 of the density samples; files on the middle level contained 1/8 of the bounding region and used 1/8 of the samples; files on the lowest level contained 1/64 of the bounding region and used every sample.

Even though the multi-resolution ASDF includes redundant data between levels, we find that it reduces the file size by more than an order of magnitude, as shown in Table 5.2. Meshes are absent from this table, as they were prohibitively large.

Description	File type	File size (MB)	Compressed size (MB)
Raw CT data	`.vol` float array	3724	2688
ASDF isosurface	Single `.asdf` file	212	159
ASDF isosurface	Multiple `.asdf` files	260	203

Table 5.2: File sizes in multi-resolution CT workflow

Figure 5-2 shows a multi-resolution ASDF triangulated at two different detail levels. The left model is visually low-resolution but suitable for high zoom levels. The model on the right is higher-resolution and can be dynamically loaded when appropriate. In practice, multi-resolution ASDFs are build into the system described in Subsection 4.3.3, so they are dynamically refined in the same manner as f-rep expressions.

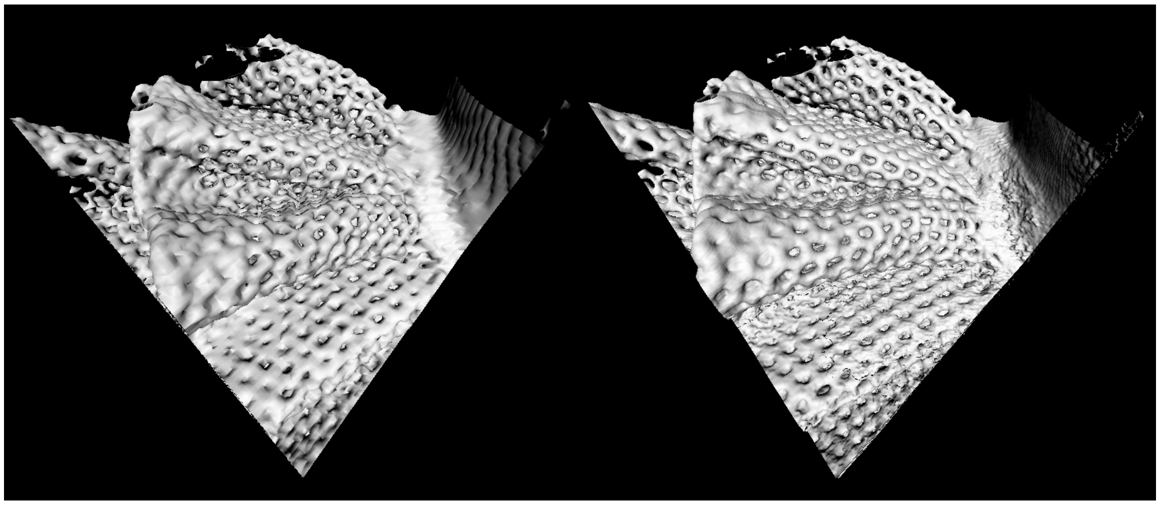


Figure 5-2: Multi-resolution ASDF models

Chapter 6

Manufacturing

Previous chapters have discussed ways to create solid models; this chapter covers transforming these models into physical artifacts. We begin by discussing a strategy for generic CAM workflow construction. Next, we discuss path planning on discrete lattices, including a brief detour to describe a space-efficient contouring algorithm for ASDFs. Finally, we show a challenging worked example fabricated on a five-axis mill.

This chapter has its genesis in the fab modules project [34], a set of tools for CAM on many machines and with a variety of input formats. Major improvements include better workflow generation, simple extensibility (to add new machines), the use of ASDFs rather than greyscale images, and clean integration with the CAD user interface.

6.1 Generic CAM workflow construction

Figure 6-1 shows the set of workflows for a wide variety of inputs and machine outputs. Though the graph is complex, end-users are not exposed to the full graph of possibilities. The system computes the needed intermediate stages based on the input and output types.

Nodes in the graph are implemented in `kokopelli` as UI panels with defined inputs and outputs. The panels for a given workflow are stacked into a complete CAM UI. Figure 6-2 shows a example workflow from f-rep to five-axis shopbot. Each

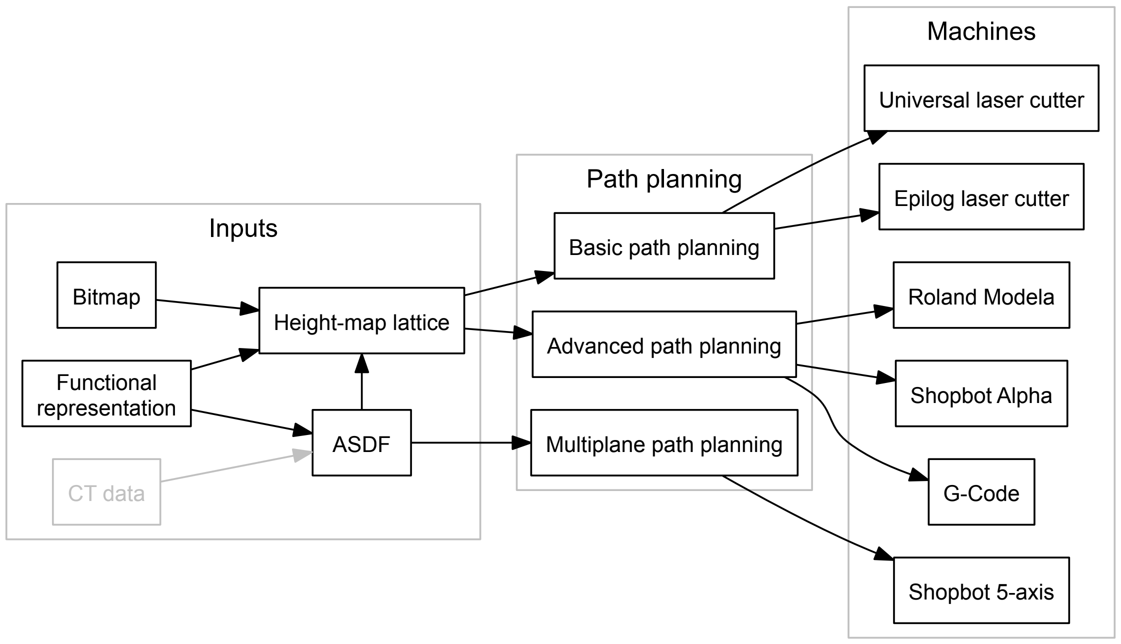


Figure 6-1: CAM workflow graph

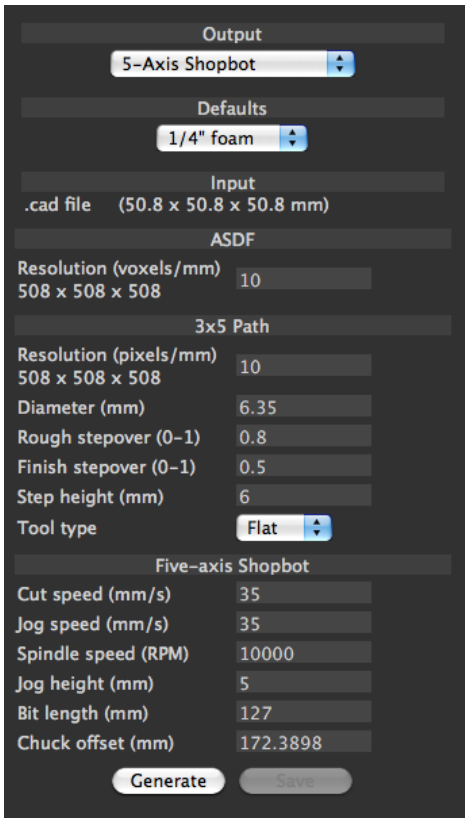
of the CAM graph nodes has a UI panel (other than CT data import, which uses a separate command-line utility).

6.2 ASDFs versus bitmaps

Most numerically controlled machines are fundamentally discrete: somewhere in the machine, a stepper motor is turning in discrete ticks to move an axis. As such, there is a robust body of work relating to populating lattice occupancy grids from continuous surfaces (one overview is [2]). If the lattice voxels are smaller than the machine's discrete steps, then the output will be optimally smooth.

Previous work in the fab modules [34] uses greyscale bitmap images as general purpose height-map lattices. Such a height-map is sufficient for two and three-axis machining, e.g. laser cutting or milling. However, starting the workflow from an ASDF offers advantages over simple bitmap images.

Because ASDFs are reconstructed with interpolation, they can be stored at a lower lattice resolution than a bitmap of equivalent error rate; a higher-resolution lattice


Output

5-Axis Shopbot

Defaults

1/4" foam

Input

.cad file (50.8 x 50.8 x 50.8 mm)

ASDF

Resolution (voxels/mm) 508 x 508 x 508	10

3x5 Path

Resolution (pixels/mm) 508 x 508 x 508	10
Diameter (mm)	6.35
Rough stepover (0-1)	0.8
Finish stepover (0-1)	0.5
Step height (mm)	6
Tool type	Flat

Five-axis Shopbot

Cut speed (mm/s)	35
Jog speed (mm/s)	35
Spindle speed (RPM)	10000
Jog height (mm)	5
Bit length (mm)	127
Chuck offset (mm)	172.3898

Figure 6-2: CAM workflow panels

can then be produced by upsampling the ASDF. A simple example is shown in Fig. 6-3. Both the bitmap and the ASDF were rendered on a 22x22 lattice. However, when reconstructed at 10x the original resolution, the bitmap is pixelated while the ASDF remains clean.

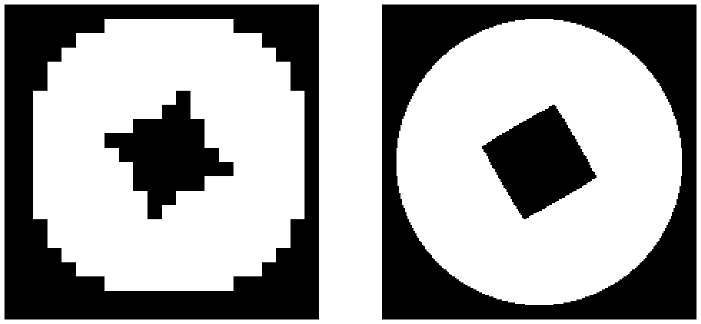


Figure 6-3: Upsampled bitmap and ASDF

Figure 6-4 examines this property in more detail. A circle of radius 1 was created and exported at a variety of lattice resolutions. Ideally, tracing a contour around this circle should give a set of points at radius 1. The graph shows the mean squared error of contour points (found using a strategy such as the one described in Section 6.4). Across the range of resolutions, we find that doubling the upsampling level decreases the mean squared error by a factor of four.

These results have implications for model file size. For a given error rate, we expect an ASDF to more efficiently encode a model. Figure 6-5 examines the relationship between mean squared error and file size, with compression used to account for discrepancies in file formats. The same circle was used as a test file, and error rates are based on 8x upsampling on the ASDF. We find that the compressed ASDF file more efficiently represents the model at all error levels.

6.3 Image offsetting

Subtractive fabrication operations typically use a cutting head – whether an endmill, waterjet, or laser beam – to remove material from an original piece of stock. Offsetting is a fundamental operation: toolpaths should be generated such that the edge (rather

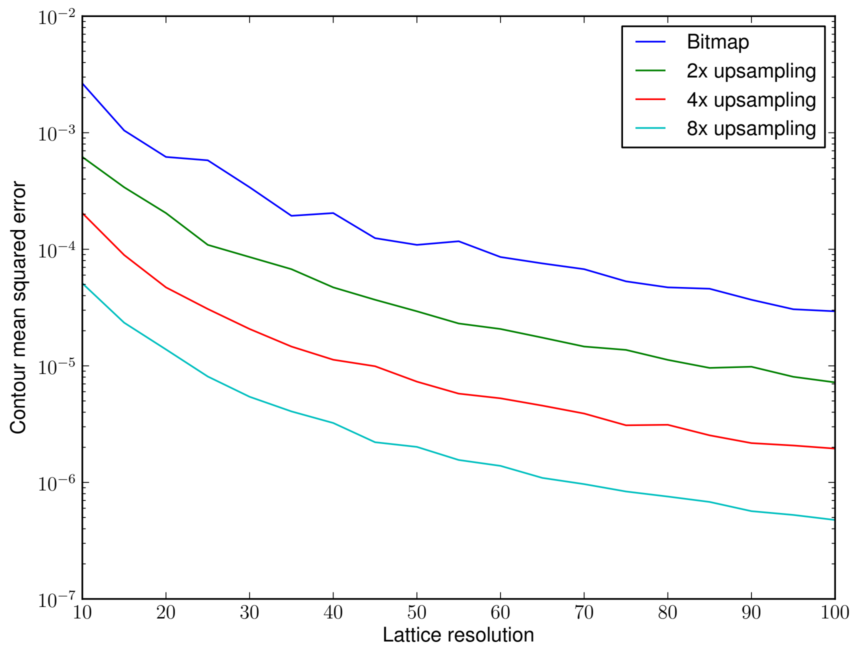


Figure 6-4: Mean squared errors in circle contours

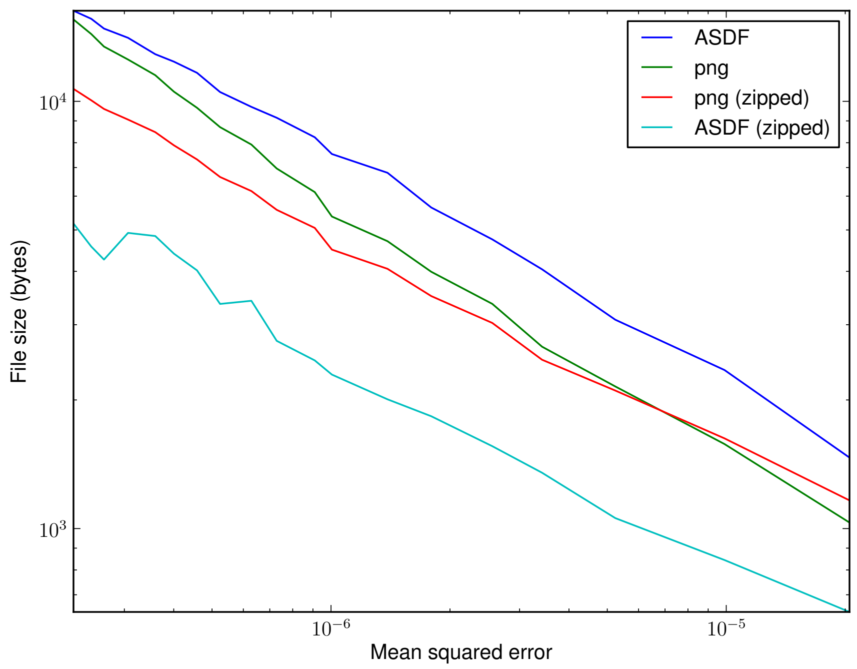


Figure 6-5: File sizes as a function of mean squared error

than the center) of the cutting head traces the contour of the desired shape. Much of the literature focuses on generating Voronoi diagrams from continuous shapes (e.g. [21, 22]); in our work we use a simple distance transform on a discrete lattice.

A Euclidean distance transform was implemented effectively verbatim from Meijster's 2000 paper [31]. This transform takes a binary bitmap image and returns a lattice containing Euclidean distances to the solid. The result of the transform on a printed circuit board design is shown in Fig. 6-6.

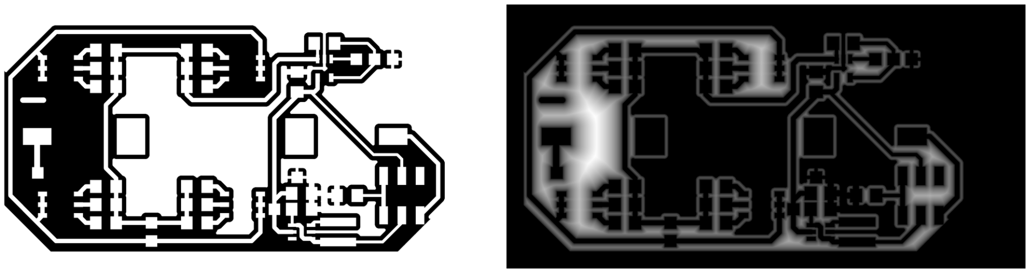


Figure 6-6: Original model and distance transform

The Meijster distance transform is trivial to parallelize, as rows and columns are evaluated independently. Testing on the circuit board from Fig. 6-6 found a local optimum at two, with more threads leading to degraded performance. These results are shown in Fig. 6-7.

6.4 Lattice contouring

The marching squares algorithm transforms a distance field to a set of contours. The algorithm is an adaptation of marching cubes run in two dimensions rather than three. For a given pixel, there are sixteen possible corner occupancy states; the look-up table in Fig. 6-8 defines between zero and two edges for each corner occupancy state. The distance field is scalar, rather than binary, so interpolation between corner values is used to improve the quality of the resulting contour.

Contouring is a two-stage process. In the first pass, lattice locations are checked against the look-up table and edges are constructed. In the second pass, edges are traced to construct full contours. The second pass takes advantage of edge direction-

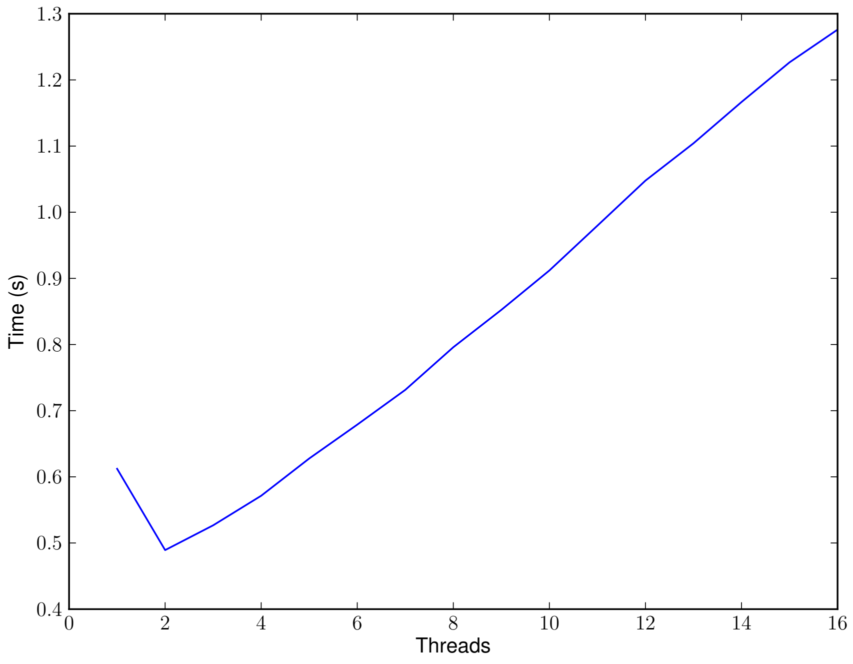


Figure 6-7: Time taken for distance transform

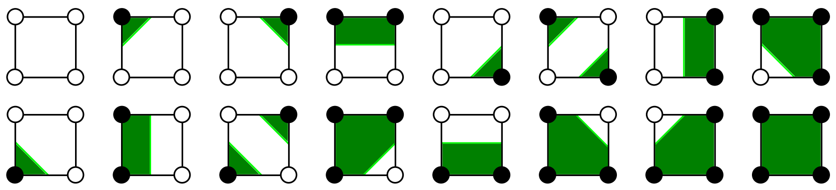


Figure 6-8: Marching squares lookup table

ality: edges are always placed so that they travel counter-clockwise around the solid, which makes tracing contours simple.

We expect contouring to run in $O(n)$ time, where n is the number of lattice points in the distance field. This was tested on a complex model by Sam Calisch, showed in Fig. 6-9.

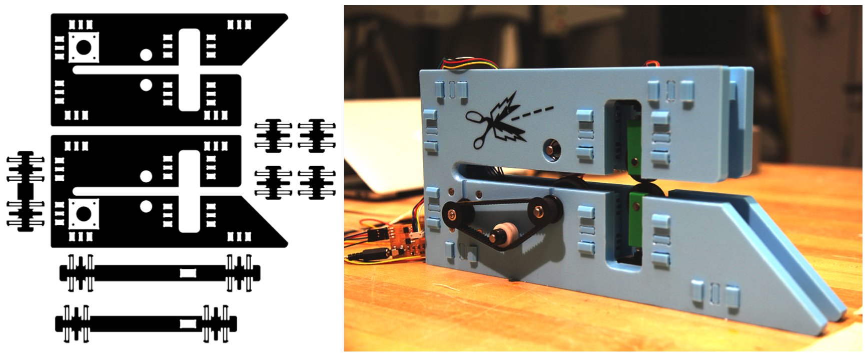


Figure 6-9: CNC fabric cutter (credit: Sam Calisch)

Figure 6-10 shows contour time as a function of image resolution (with a 1/16" offset). As expected, time taken in the contouring pass is roughly linear with pixel count (equivalently, $O(n^2)$ with DPI).

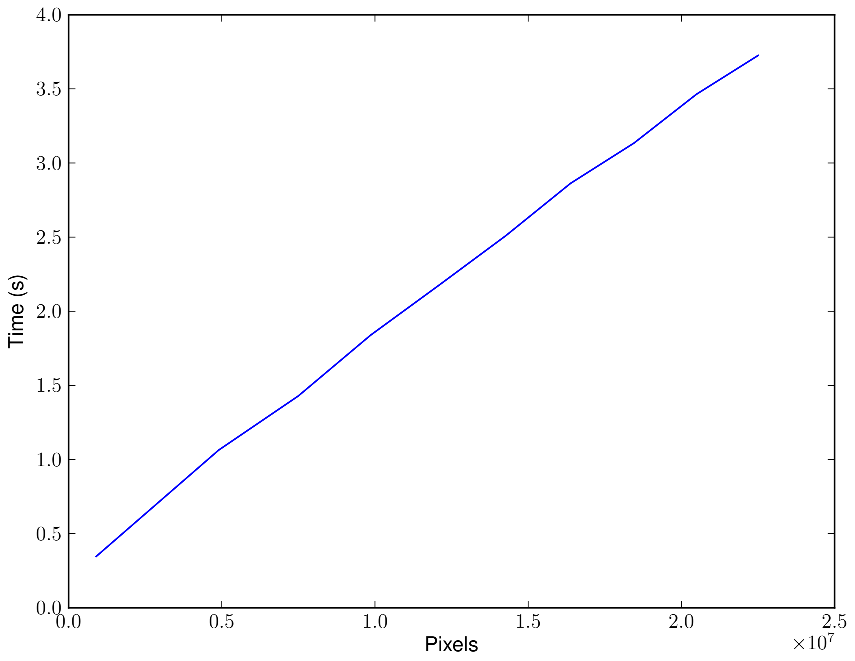


Figure 6-10: Contour time

6.5 ASDF offsetting

In the course of this research, we also developed a strategy for direct offsetting of ASDF structures without the intermediate step of a lattice. This strategy is hierarchical in space but not in time; it evaluates the same number of points as a lattice distance transform but efficiently collects them into an ASDF tree. This strategy is also applicable for direct offsetting from f-rep expressions.

Our strategy is based on the Meijster distance transform. This distance transform has two stages. In the first stage, columns are processed one by one to generate the G lattice (which records minimum vertical distance to the solid). In the second stage, rows are processed one by one to generate the D lattice (which records minimum distance to the solid with the chosen metric).

Our modified transform operates in a similar manner, but constructs G and D ASDFs instead of lattices. This preserves the hierarchical nature of the input data structure. To generate the G ASDF, we generate g columns as in the Meijster distance transform, then collect them into ASDF cells. Two g columns are active at a time, representing data on the left and right edges of ASDF cells. We repeat this process for all of the columns in the image, simplifying ASDF cells when all of their children have been populated.

Figure 6-11 shows the first two columns being collected into an ASDF. Green circles represent cells with all children populated; when a node changes from white to green, it should be simplified to reduce tree size.

The same process is repeated for the D ASDF: rows are generated as in the Meijster distance transform, then collected into an ASDF with simplification when cells finish populating all of their children.

In a slight deviation from the Meijster transform, our implementation calculates both positive and negative distances. This makes leaf cell combination more likely by giving cells better distance values as their basis for interpolation.

To reduce the size of the resulting trees, our implementation only generates a single offset, defined by a distance values v. When generating the G ASDF, values

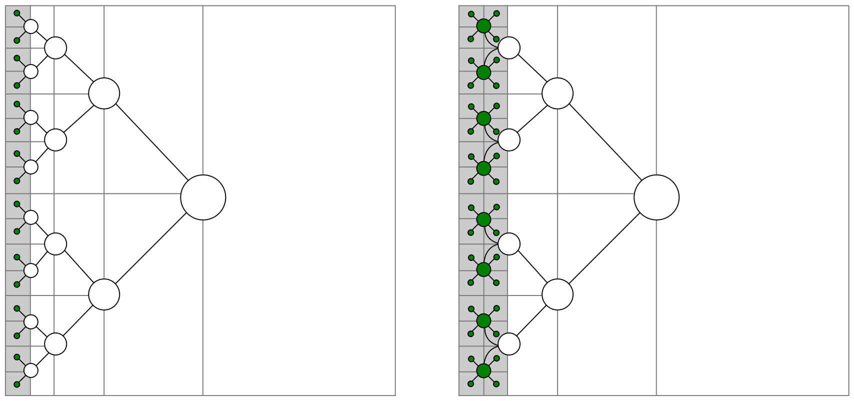


Figure 6-11: Two columns of ASDF distance transform

greater than $\sqrt{2}v$ are classified as empty and values less than zero are classified as filled. When generating the D ASDF, an isocontour at distance v is saved in leaf cells; cells above or below this isocontour are marked as empty or filled. Figure 6-12 shows multiple offsets generated using this strategy.

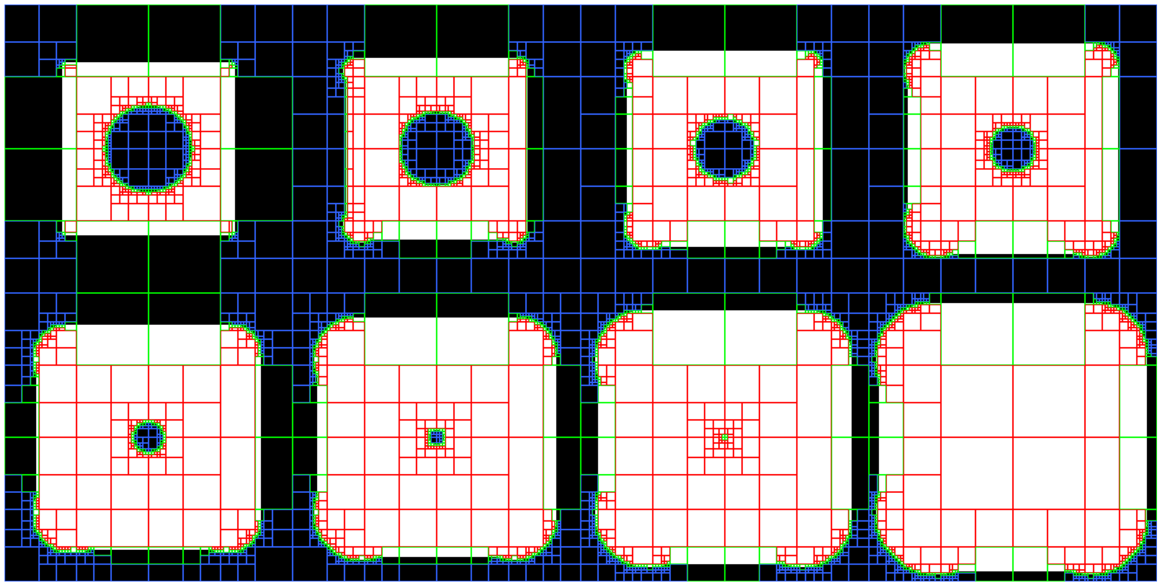


Figure 6-12: Multiple ASDF offsets

In practice, this algorithm runs slightly slower than a plain lattice distance transform, as extra time is required for tree manipulations. The algorithm is interesting from a theoretical perspective, but proved not to be of practical use: files that are too large for the original algorithm for reasons of size are infeasible for the modified algorithm for reasons of running time.

6.6 Rough cuts

Rough cuts are intended to efficiently clear material, generating a set of flat contours that can be further refined. They are calculated on the XY plane at a variety of Z heights. The input heightmap or ASDF is thresholded at each Z value to produce a binary image. A distance transform produces a continuous distance field from this binary image, and a set of contours are extracted.

Generating rough cuts in this manner is relevant to both two-dimensional and three-dimensional machining. Figure 6-13 shows offset contours for machining of PCB traces, while Fig. 6-14 shows rough cuts on multiple Z levels.

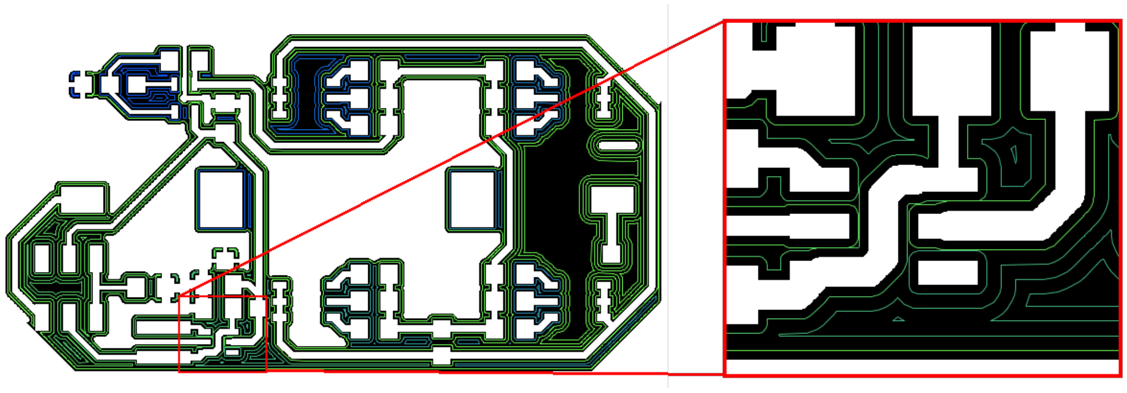


Figure 6-13: Contour cuts on a PCB bitmap

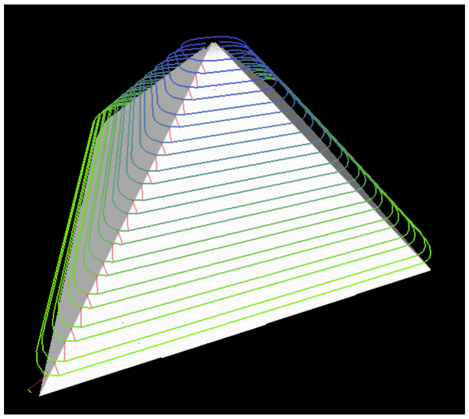


Figure 6-14: Rough cuts on a solid model

6.7 Toolpath sorting

Each layer of a rough cut is described by a list of paths (which may be either closed or open). Before sending them to the machine, it is useful to sort them to minimize jog distance.

This sorting must obey the rule that if path A is completely enclosed by path B, then A should be cut before B. If B was cut before A, then the piece containing A would be free to move as A was being cut out; obviously, this is not desirable.

We check for enclosure by finding the XY bounding box of each path, then declaring that path A is enclosed by path B if the bounding box of A is entirely within the bounding box of B. This rule will occasionally yield a false positive (i.e. saying that A is within B when this is not the case), but will never yield a false negative (which would produce an invalid sorting). Figure 6-15 shows this rule applied in four cases. Case 1 is a correct positive, cases 2 and 3 are correct negatives, and case 4 is a false positive.

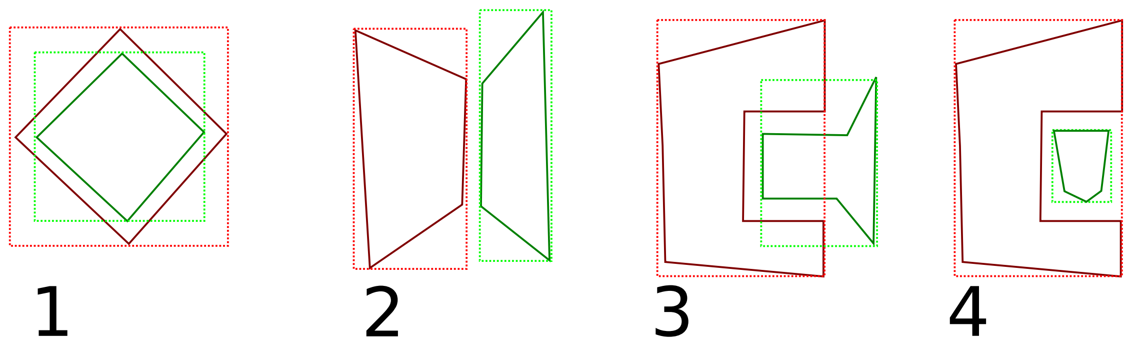


Figure 6-15: Paths A (green) and B (red) with their bounding boxes (dotted)

A greedy algorithm is used for path sorting. At each stage, we choose the path whose start point is nearest to the previous path's end point and whose enclosed paths have all already been selected. This is an $O(n^2)$ strategy (where n is the number of paths to sort), as we generate pairwise distances from path start and endpoints. Figure 6-16 shows a set of sorted toolpaths. Color shades from blue (first toolpath) to green (last toolpath), and red lines show traverses.

This strategy is non-optimal. For closed toolpaths, the start and end points could

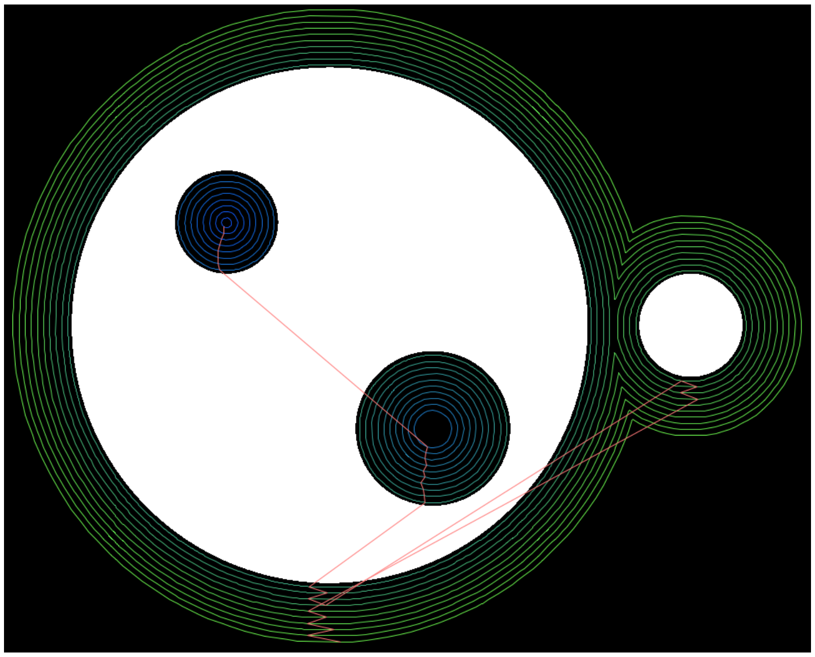


Figure 6-16: Sorted toolpaths

be anywhere along the loop, rather than at specific locations. In addition, the greedy algorithm does not necessarily give optimal orderings. As toolpath sorting is equivalent to the traveling salesman problem (with precedence constraints), developing an polynomial-time algorithm to find the optimal solution is beyond the scope of this thesis [11].

6.8 Finish cuts

A finish cut travels along the XZ plane at many Y values or the YZ plane at many X values, tracing the contours of the shape's surface. Performing a finish cut following rough cuts improves the quality of the final model.

On a height-map lattice, finish cuts are implemented using pseudo-convolution. A finish cut is parameterized by endmill radius and shape (e.g. flat or ball). Given these parameters, we generate two lattices. The first is a heightmap representing the endmill (at the same pixel resolution as the original lattice); the second is a binary mask recording occupancy. Examples of endmill height-maps are shown in Fig. 6-17

We then "convolve" the endmill with the model's height-map lattice. In normal

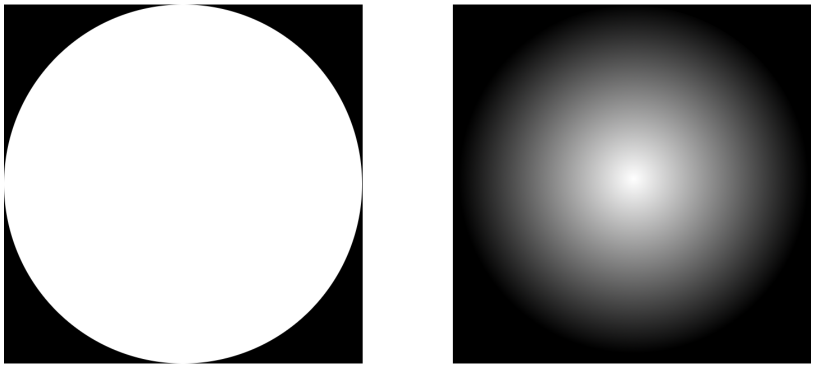


Figure 6-17: Flat and ball endmill heightmaps

convolution, we calculate the sum of many multiplication operations; this pseudo-convolution replaces multiplication with subtraction and summing with `max`, while only checking points where the endmill mask is true. This operation returns a z height for the endmill such that it barely touches the surface at some point on its area. Figure 6-18 shows finish cuts on a simple cube with flat and ball endmills. As expected, the ball endmill traces a curving path around the cube's edges.

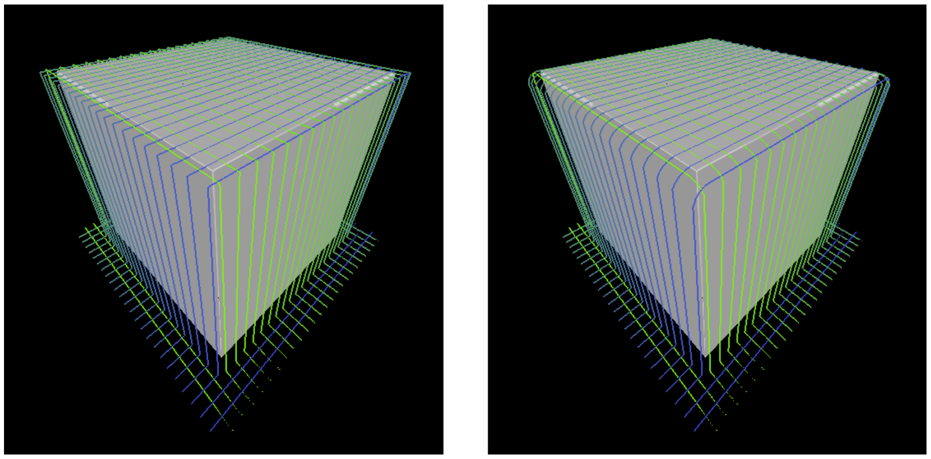


Figure 6-18: Finish cuts with flat and ball endmills

6.9 Multi-plane machining

For complex models, we developed a workflow to perform three-axis rough and/or finish cuts on multiple distinct cut planes. This workflow was targeted at a Shopbot 5-Axis machine, shown in Fig. 6-19, but the toolpath generation stage is generic and applicable to other five-axis machines.

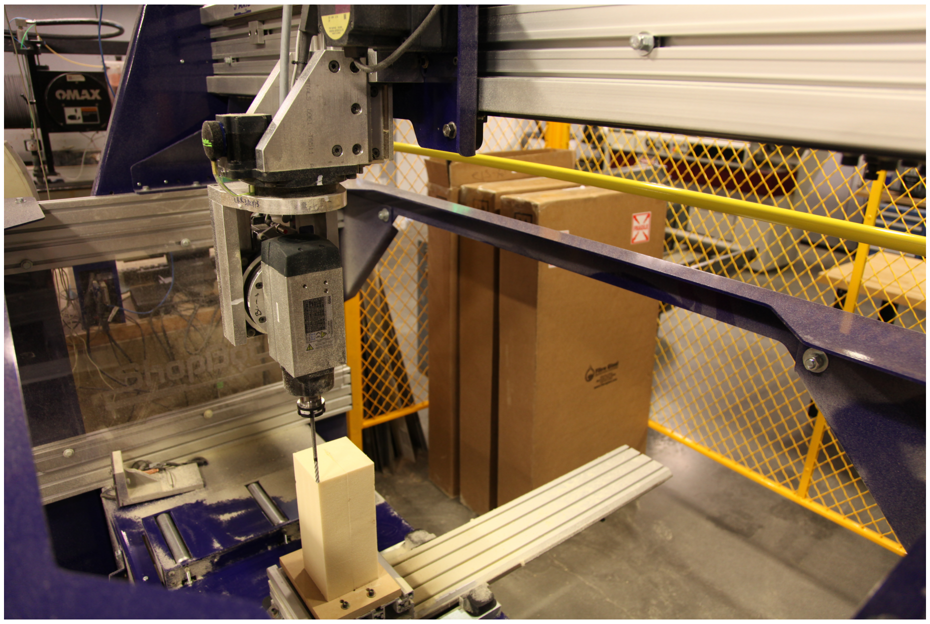


Figure 6-19: Shopbot 5-axis

This workflow takes place in two stages. In the first stage, a target ASDF is rendered to heightmaps with various rotations, then contoured with rough and finish cuts as described above. The cut positions are transformed from the rotated coordinate system to the global coordinate system and the endmill direction is recorded as a unit vector. The result of the first stage is a set of paths, where each path contains some number of six-element points (representing position on the solid's surface and endmill direction vector). These paths are grouped by cut plane. This stage is generalizable to any five-axis machine.

In the second stage, these planes are combined into a single path file based on the Shopbot's mechanics. Because the Shopbot uses a rotating head (rather than a trunion table), the surface position must be offset by the bit length plus the gauge length in the direction opposite the endmill pointing vector.

Jogs between cuts on a plane must take place along a safe vector. Our strategy for these jogs is very conservative. Given a point on a safe plane s, an endmill direction vector v (with $|v| = 1$), and a cutting point p, the appropriate safe position is found at $p' = p - v\,((p - s) \cdot v)$ (shown in Fig. 6-20). All jogs are made along the safe plane.

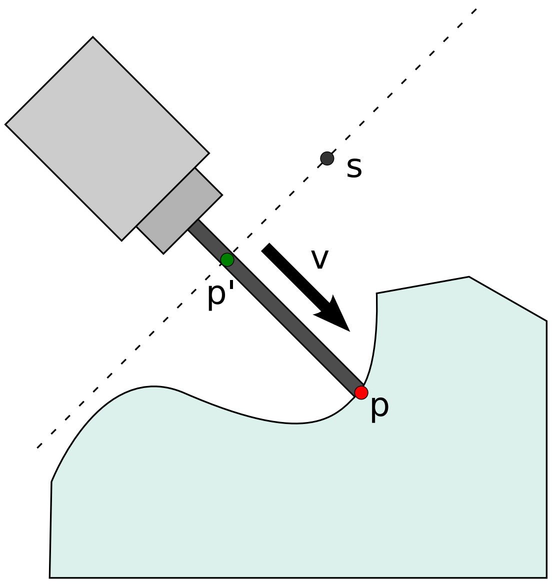


Figure 6-20: Safe plane for angled cuts

Using similar math, we check the endmill depth based on the bit length provided by the user. If the endmill depth exceeds the bit length, it is backed out to an appropriate distance. This prevents the chuck from colliding with the body of the model. Again, this is a conservative strategy. Depending on model and chuck geometry, it is often possible for the chuck to enter the model's bounding box without collision, but this is more complex to check.

Finally, special care must be taken when transitioning between cut planes. The strategy is described in Algorithm 6.1. Following this sequence ensures that the head will not cut through an existing part of the model, with the limitation that the cutting planes cannot be overhanging (i.e. the z coordinate of the endmill direction vector must be ≤ 0).

Algorithm 6.1 Cut plane transfer

```
  Retreat to safe plane
  Move up on the $z$ axis to above the top of model
  Rotate so that the endmill is pointing down ($A = B = 0$)
  Travel to the $xy$ coordinates of the next plane's first point
  Rotate head based on this plane's pointing vector
  Travel to the $z$ coordinate of the next plane's first point
```

The implemented workflow includes three different plane selection strategies. The first uses the five visible planes (top, front, back, left, and right). The second allows a user-defined plane, which can also be taken from the viewport rotation. The third option generates evenly spaces planes based on user-provided step sizes (this is a generalization of the first strategy).

Machining separate planes works best when the planes are independent. In practice, some amount of air cutting occurs when a path attempts to clear material that was already removed on a previous plane.

6.10 Worked example

As a challenging example, we chose to machine the lace discussed in the previous chapter. The model was machined on the five-axis ShopBot on all five visible faces, with rough and finish cuts on each face. The original model represented a 1" $\times$ 1" $\times$ 2" volume; it was scaled up by a factor of four and machined as a 4" $\times$ 4" $\times$ 8" model.

Figure 6-21 depicts the results. Though the chosen end-mill (1/4" ball) was not small enough to machine individual holes, the output is a reasonable replication of the CT scan data.

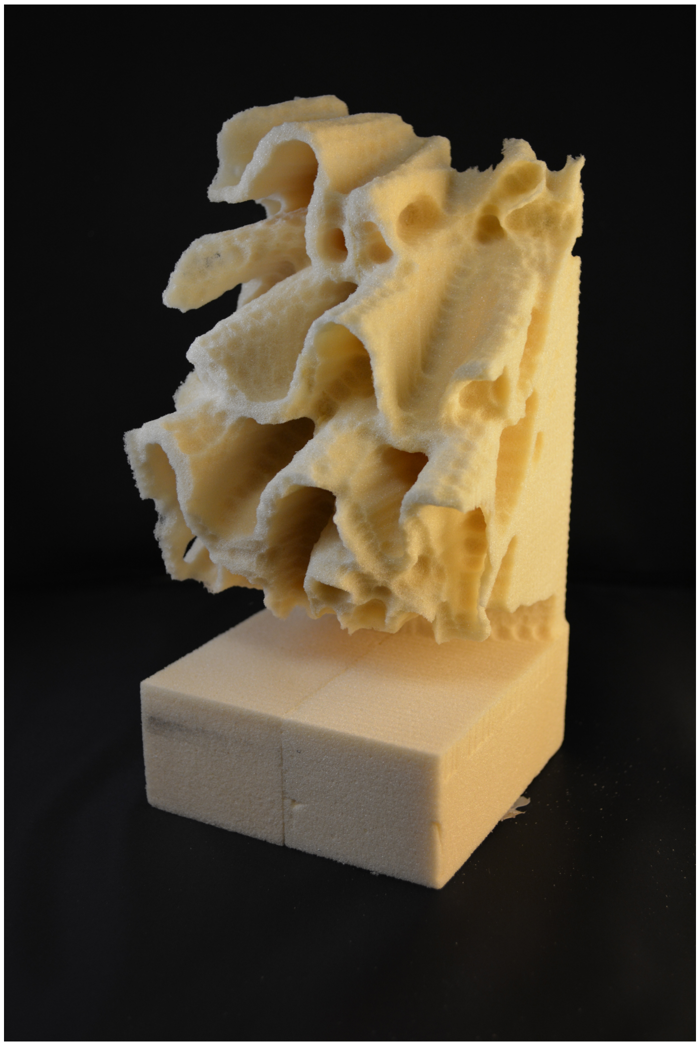


Figure 6-21: Machined lace

Chapter 7

Conclusions

This document has described a range of research into novel CAD/CAM workflows, from design with functional representations to reproduction with CT scanners to lattice-based path planning. A complete CAD workflow was implemented by a single researcher over the course of one year, demonstrating rapid prototyping of design tools for rapid prototyping. Though this workflow has fewer features than high-end commercial CAD packages, its quick development time is equally due to its lack of historical baggage.

This chapter summarizes results, discusses workflow limitations, and synthesizes recommendations for future work in the area.

7.1 ASDF performance and behavior

We found that using adaptively sample distance fields on an octree is an efficient strategy for representing spatially complex objects. The octree takes advantage of spatial locality and distance samples allow for accurate surface and normal reconstruction.

A trade-off can be made between model accuracy and cell count by tuning the minimum interpolation level: experimentally, we found that a $10\times$ increase in allowed interpolation error results in a $3\times$ decrease in the number of cells needed to represent a model at peak accuracy.

These models show a degree of non-isotropy. In a simple test, worst-case rotations required 5× as many cells as an axially-aligned model. However, both of these cases require far fewer cells than a full octree, as the ASDF does not always recurse to minimum voxel size along surfaces.

Render time depends on both ASDF and image resolution. Our fairly naïve implementation performs poorly when rendering a low-resolution ASDF to a high-resolution image, but generally renders thousands-of-pixel images in tens of seconds using multiple threads. Render time appears linear with voxel count. Finally, ASDF triangulation runs in slightly sub-linear time relative to the number of voxels being evaluated.

7.2 Functional representations

Distance metric functional representations proved to be an excellent way to populate ASDFs. Even though the distance field from an f-rep may not be completely Euclidean, they are often linear enough to allow for ASDF leaf cell combination (especially if they are locally Lipschitz continuous).

F-rep representations appear to be comparable to vector-based representations for two-dimensional text; both scale at linear rates with text length. In practice, vector representations have slightly fewer nodes than f-reps have clauses, but f-rep strings are smaller than equivalent `.svg` files.

We showed very significant f-rep tree reduction was often possible: in one experiment, an average of 99.5% of nodes could be ignored when rendering small chunks of the spatial region. Speed improvements over a naïve, pixel-by-pixel approach were primarily due to this tree pruning, enabled by spatial subdivision and interval arithmetic. In general, we found render time was approximately linear with voxel count and decreased logarithmically with thread count.

7.3 CT data import

Importing CT data into an ASDF (rather than a mesh) appears to offer major advantages. This representation drastically reduces file size while preserving intricate details. In practice, ASDF "isosurfaces" were between two and three times smaller than meshes generated from the same model; in addition, the ASDF preserves surface curvature more accurately than flat triangles.

Though ASDFs have small file sizes, they are often large when loaded into RAM; as such, commodity laptops may not be able to load an entire CT scan into a single ASDF tree. Instead, the scan can be downsampled and used to create a series of ASDFs at various resolutions. In this test, the full set of "multi-resolution ASDFs" was 15$\times$ smaller than the raw scan data.

7.4 Toolpath generation

We found that ASDFs can represent shapes more efficiently than plain lattices, thanks to both hierarchy and interpolation. Upsampling an ASDF (that is, rendering it above its native lattice resolution) produces smooth images and decreases contouring error.

The Meijster distance transform was used on greyscale lattices and modified for use directly on ASDFs. The modified transform was not practical for large images, as it had a similar running time; the efficiency improvement was only in required storage space. Generating offset contours was found to be linear with image pixel count, as expected.

7.5 Workflow limitations

Size remains a major workflow limitation. The Shopbot has a bed size of 2.4×1.2 meters at about 10 pixels per mm, equivalent to a 24K by 12K pixel image. This lattice size (equivalent to a 1.2 GB image at 4 bytes per pixel) is too large to process on commodity hardware.

Our design tools can create ASDFs at that size without trouble, assuming the model is not uniformly high-resolution and intricate. However, the workflow for path planning does not scale well. If using greyscale lattices, the required RAM becomes excessive: the lattice described above requires 3 GB for storage of intermediary and final distance values. If using our ASDF offsetting algorithm, storage sizes remain reasonable but running time is too long: a simple model on a ShopBot-sized lattice took six minutes to generate a single offset.

A second workflow limitation is aspect ratio. Parts that are exceedingly thin require high ASDF cell counts to avoid being overlooked by aliasing. Designs that use sheet-metal or fabric are not suitable for our CAD system.

7.6 Future work

Extending our workflow to include physical modeling and simulation will enable closed-loop design, in which parameters extracted from simulation can feed back into the design description. Future work will likely involve a multiphysics modeling workflow that operates on ASDFs, extracting physical, mechanical, and electro-magnetic properties from ASDF models.

Declarative design is a catch-all phrase for design tools that translate very high-level descriptions (often specified in terms of constraint satisfaction and maximization) into design files. Such design is tightly integrated with techniques from the optimization research community. Combining our design tools with physical modeling will create a strong platform for future declarative design research.

Feature extraction (either algebraically from an f-rep or numerically from an ASDF) will enable more advanced CAD operations. Though feature-based CAD introduces topological dependancies not present in f-reps, it allows for operations like fillets (which are defined on a specific edge or face). A combination of f-rep and feature-based operations could enable CAD tools competitive with commercial offerings at a fraction of their complexity.

More generally, it would be valuable to develop editing tools that modify ASDFs directly (rather than treating them as read-only after their initial creation). ASDFs have already been used at the basis for a sculpting-style design tool [43]; fusing this style of interactive creation with the rest of our powerful workflow would open up interesting possibilities for more creative design.

There are also a set of interesting questions relating to efficient CAM operations on these distance fields. Our lattice-based strategy is effective, but discards all of the hierarchy in the structure and scales poorly to very high-resolution models. A fully hierarchical CAM workflow would solve the size limitations discussed above and enable very high-resolution design and fabrication. Finally, extending the workflow to full five-axis machining leads to another set of research challenges.

Appendix A

Math String Syntax

The f-rep solver includes a parser for math strings. This parser accepts strings written in a sparse prefix-notation syntax, with functions listed in Tables A.1 and A.2. Note that all trigonometric functions operate on radians.

Function	Code
Sine	`s`
Cosine	`c`
Tangent	`t`
Arcsine	`S`
Arccosine	`C`
Arctangent	`T`
Absolute value	`b`
Square	`q`
Square root	`r`
Negation	`n`

Table A.1: Unary F-rep functions

Function	Code
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Minimum	`i`
Maximum	`a`
Power	`p`

Table A.2: Binary F-rep functions

The f-rep syntax accepts four distinct types of atoms. `X`, `Y`, and `Z` are replaced by position in the world's coordinate system at any given evaluation point. Floating-point constants are preceded by `f`, followed by the value (e.g. `f3.14159` or `f6.023e23`). For example, a circle ($x^2 + y^2 - 1$) is written as `-+qXqYf1`.

Finally, the map operation allows for coordinate transforms to be represented in a space-efficient manner. The map operator is written as `m` followed by four arguments. The first three arguments represent transformed expressions x', y' and z' respectively (in terms of x, y, and z); if a coordinate is not modified then a space appears instead. The last argument is the expression to evaluate with the transformed coordinates. The example `m+XYX X` applies the coordinate transform `X'=X+Y`, `Y'=X`, `Z'=Z` to the expression `X`, producing `X+Y` (note that the z coordinate is not modified, because the third argument following `m` is a space).

Appendix B

Sample Machine Description

Machine descriptions are kept in the folder `koko/cam/machines/`. Each machine is defined by a single Python file ending in `.py`. This module must define the following global (module-level) variables:

Name	Type	Description
`NAME`	String	Machine name
`INPUT`	`FabPanel`	Preceeding panel
`PANEL`	`OutputPanel`	Machine panel
`DEFAULTS`	List of tuples	Set of defaults

This appendix contains a heavily annotated machine description, intended for use as a template for new machines. Note that new machines also need to be added to the `MACHINES` global variable in `koko/cam/machines/__init__.py`. This will cause them to appear in the drop-down menu.

```
## koko/cam/machines/epilog.py

## NAME is the machine's name.  It will appear in the drop-down menu
## of machines in the CAM panel.
NAME = 'Epilog'

import  tempfile
```

```python
import  koko
from    koko.cam.panel import OutputPanel

## This section defines a UI panel for this machine.
## It's a subclass of OutputPanel, which is defined in
##      koko/cam/panels.py

class EpilogOutput(OutputPanel):

    ## Every machine needs to define an extension for saved files
    extension = '.epi'

    ## Every machine has to have a constructor
    def __init__(self, parent):
        OutputPanel.__init__(self, parent)

        ## The construct function takes in a panel name and
        ## a list of tuples.  Each tuple contains
        ##      Parameter label
        ##      Parameter name
        ##      Parameter type
        ##      Checking function (optional)

        self.construct('Epilog laser cutter', [
            ('2D power (%)', 'power', int, lambda f: 0 <= f <= 100),
            ('Speed (%)', 'speed', int, lambda f: 0 <= f <= 100),
            ('Rate','rate', int, lambda f: f > 0),
            ('xmin (mm)', 'xmin', float, lambda f: f >= 0),
            ('ymin (mm)', 'ymin', float, lambda f: f >= 0),
            ('autofocus', 'autofocus', bool)
        ])


    ## A machine needs a run function.  The argument to this
    ## function is whatever is returned from the machine's
    ## INPUT panel (in this case, it's a ContourPanel,
    ## which returns one variable named 'paths').  The run
    ## function should convert the paths to a machine-specific
    ## file.
    def run(self, paths):

        ## self.get_values returns a dictionary mapping parameter
        ## names (as defined above in self.construct) to values,
        ## or False if any of the parameters is invalid
        values = self.get_values()
        if not values:  return False

        ## It's often helpful to change the status bar to
        ## give the user feedback on what's going on.
        koko.FRAME.status = 'Converting to .epi file'

        ## A machine needs to define self.file as a
        ## NamedTemporaryFile.  This file should contain the
        ## actual machine output instructions
        self.file = tempfile.NamedTemporaryFile(suffix=self.
            extension)

        ## Lines below here describe how to generate the laser
        ## output file. This will vary from machine to machine.

        job_name = koko.APP.filename if koko.APP.filename else '
            untitled'
        self.file.write("%%-12345X@PJL JOB NAME=%s\r\n E@PJL ENTER
            LANGUAGE=PCL\r\n &y%iA &l0U &l0Z &u600D *p0X *p0Y *t600R
           *r0F &y50P &z50S *r6600T *r5100S *r1A *rC %%1BIN;XR%d;YP%
           d;ZS%d;\n" %
            (job_name, 1 if values['autofocus'] else 0,
             values['rate'], values['power'], values['speed']))

        scale = 600/25.4 # The laser's tick is 600 DPI
        xoffset = values['xmin']*scale
```

```
        yoffset = values['ymin']*scale
        xy = lambda x,y: (xoffset + scale*x, yoffset + scale*y)

        for path in paths:
            self.file.write("PU%d,%d;" % xy(*path.points[0][0:2]))
            for pt in path.points[1:]:
                self.file.write("PD%d,%d;" % xy(*pt[0:2]))
            self.file.write("\n")

        self.file.write(" %%0B %%1BPUtE %%-12345X@PJL EOJ \r\n")

        ## It's important to flush the file when you're done
        ## writing to it, since it may be used after this point
        self.file.flush()

        koko.FRAME.status = ''

        return True


################################################

## INPUT is whatever panel should preceed this machine in a
## CAM workflow.  Typically, it will be some form of path panel.
from koko.cam.path_panels   import ContourPanel
INPUT = ContourPanel

## PANEL is the name of this machine's output panel (in
## this case, it's the EpilogOutput defined above)
PANEL = EpilogOutput

################################################

from koko.cam.inputs.cad import CadImgPanel

## Defaults is a list of tuples.
## Each tuple contains a default name and dictionary.
## The dictionary maps FabPanel names to tuples of parameter names
## and desired values.
## For example,
##      ('CardBoard', <=== Default name
##          {CadImgPanel: <=== FabPanel to which the
##                             following settings apply:
##               [('res',    <=== Parameter name
##                  5        <=== Parameter value
##                 )]})

## When a set of defaults is applied to a workflow, each panel
## is looked up in the defaults dictionary.  If found, each
## parameter in the list of tuples is set to the desired value.

DEFAULTS = [
    ('<None>', {}),

    ('Cardboard',
        {CadImgPanel:  [('res',5)],
         ContourPanel: [('diameter', 0.25)],
         EpilogOutput: [('power', 25), ('speed', 75),
                        ('rate', 500), ('xmin', 0), ('ymin', 0)]
        }
    )
]
```

Appendix C

Core classes

This appendix describes the core classes used in this CAD/CAM workflow. These classes are used in `kokopelli`, but they are not tightly coupled to the rest of the system; they could be used in other workflows. Note that this documentation only includes a select subset of methods and parameters. The source is heavily annotated with Doxygen docstrings, which can be processed into a complete manual.

C.1 `koko.fab.asdf.ASDF`

The `ASDF` class wraps a pointer to a C data structure storing the actual ASDF. The pointer is stored in the `ptr` data attribute. When the Python destructor is called, this structure is freed if the pointer is to the top of an ASDF tree. This is determined by the `free` data attribute.

C.1.1 Instance methods

`contour(self, interrupt=None)`

`interrupt`	A `threading.Event` object used to halt run

Finds an isocontour from the ASDF, returning a list of `Paths`. `render(self, region=None, threads=8, alpha=0, beta=0, resolution=10)`

`region`	Render region (if None, bounding box is used)
`threads`	Threads to use in rendering
`alpha`	Rotation about Z axis
`beta`	Rotation about X axis
`resolution`	Resolution in voxels/mm

Renders the ASDF as a height-map, returning an `Image`.

`rescale(self, mult)`

`mult`	Scale factor

Rescales the ASDF in-place by the given scale factor.

`save(self, filename)`

`filename`	Target filename

Saves the ASDF to a file.

`slice(self, z)`

`z`	Slice height

Slices the ASDF at the given Z value, returning a 2D ASDF.

`triangulate(self, threads, interrupt)`

`threads`	Boolean; if `True`, multiple threads are used
`interrupt`	A `threading.Event` object used to interrupt render

Triangulates the ASDF, returning a `Mesh`.

C.1.2 Class methods

`from_vol(cls, ni, nj, nk, offset, mm_per_voxel, merge_leafs=True)`

`ni`	X sample count
`nj`	Y sample count
`nk`	Z sample count
`offset`	Isosurface density
`mm_per_voxel`	Scaling factor (mm/voxel)
`merge_leafs`	Boolean determining whether leaf cells are merged

Loads an ASDF from a `.vol` file, returning it.

`load(cls, filename)`

`filename`	ASDF filename

Loads an ASDF from an `.asdf` file, returning it.

C.2 `koko.fab.image.Image`

The `Image` class wraps a NumPy array, using it as an array of pixels. Each instance has `depth` and `channels` data attributes that keep track of image parameters. Valid depth values are 8, 16, 32, or 'f' (representing integers of various sizes or floating-point values). Valid channel counts are 1 or 3 (representing greyscale or RGB images).

C.2.1 Instance methods

`contour(self, bit_diameter, count=1, overlap=0.5)`

`bit_diameter`	Endmill (or beam) diameter (in mm)
`count`	Number of offsets
`overlap`	Overlap between offsets (0-1)

Finds a set of contours on a one-channel distance field image (with a depth of `'f'`). Returns a list of `Path` objects.

`copy(self, channels=None, depth=None)`

`channels`	New channel count
`depth`	New image depth

Duplicates the image, optionally modifying depth and channel count.

`distance(self, threads=2)`

`threads`	Number of threads to use

Applies the Meijster distance transform, returning a one-channel floating-point image.

`save(self, filename)`

`filename`	Image filename (must end in `.png`)

Saves the image as a bitmap. Three-channel images are saved without metadata; one-channel images are saved with bounds metadata. X and Y dimensions are encoded in the image's scale; Z bounds are specified in `tEXt` chunks with labels `zmin` and `zmax` (in mm).

`threshold(self, z)`

`z`	Threshold height (in mm)

Returns a thresholded copy of the original image.

C.2.2 Class methods

`load(cls, filename)`

`filename`	Name of `.png` file to load

Loads and returns a single `.png` file. The file is converted into a 1-channel 16-bit heightmap.

`merge(cls, images)`

`images`	List of `Image` objects to merge

Merges a set of images into a single 3-channel 8-bit image, which is returned. Input images must have same z bounds and scale. The `color` data attribute is used to colorize images as they are merged.

C.3 `koko.fab.path.Path`

A `Path` contains a set of points for a toolpath or vector contour. Each instance constains a data attribute named `points`, which stores toolpath points in the rows of a NumPy array. The data attribute `closed` defines whether the path is a loop.

C.3.1 Static methods

`sort(paths)`

`paths`	List of `Path` objects to sort

Sorts the paths as described in Section 6.7, returning a sorted list of paths.

C.4 `koko.fab.tree.MathTree`

The `MathTree` class stores a distance metric expression. This expression is stored in the `math` data attribute as a string in the sparse prefix notation described in Appendix A. When necessary, this expression is parsed into a C data structure. A pointer to this structure is stored in the `_ptr` data attribute (which is `None` if the string has not yet been parsed).

C.4.1 Instance methods

All common arithmetic and logic operators are overloaded ($+, \times, /, -, \&, |$). The `shape` data attribute modifies their behavior: if true, then addition and subtraction are interpreted as logical operators rather than arithmetic operators.

`asdf(self, region=None, resolution=None, mm_per_unit=None, merge_leafs=True, interrupt=None)`

`region`	Evaluation region (if None, taken from expression bounds)
`resolution`	Render resolution in voxels/unit
`mm_per_unit`	Real-world scale
`merge_leafs`	Boolean determining if ASDF leafs are merge
`interrupt`	`threading.Event` to halt rendering

Renders and returns an `ASDF`.

`clone(self)`

Returns a clone of the target `MathTree`.

`map(self, X=None, Y=None, Z=None)`

`X`	New X expression or None
`Y`	New Y expression or None
`Z`	New Z expression or None

Applies a coordinate transform, returning a new `MathTree`.

`render(self, region=None, resolution=None,`

`mm_per_unit=None, threads=8, interrupt=None)`

`region`	Evaluation region (if None, taken from expression bounds)
`resolution`	Render resolution in voxels/unit
`mm_per_unit`	Real-world scale
`threads`	Number of threads to use
`interrupt`	`threading.Event` to halt rendering

Renders and returns an `Image`.

`save_dot(self, filename, arrays=False)`

`filename`	Target filename
`arrays`	Boolean determining whether nodes are packed in dot graph

Converts the math expression into a `.dot` graph description.

C.4.2 Static methods

`wrap(self, value)`

`value`	Target to convert to `MathTree`

Converts a value into a `MathTree`. Strings are assumed to be valid sparse prefix expressions; numbers (floats or ints) are converted; other input types raise an error.

Appendix D

Notes on tools

This thesis describes a complete workflow, but the design tools emerged in iterative stages. This appendix includes notes on this evolution, emphasizing tools that were found to be useful.

Early development used C++ as the implementation language for the solver. In this implementation, Boost.Thread is used to implement multithreading; the STL is also used for miscellaneous data structures. This implementation resulted in a modular design, with small programs that do one operation (e.g. `math_png`, `math_stl`).

The newer implementation presented in this document uses C for core algorithms and the Python ctypes library to wrap these algorithms. This proved to be a significant improvement. The Boost dependancy is eliminated, as threading is handled by Python's threading module. Python threading is often criticized, as only one thread can execute Python code at a time (due to the global interpreter lock or GIL). However, the GIL is released when C code is called, eliminating this concern.

The Python/ctypes implementation made rapid iteration much simpler. Instead of implementing simple routines in C/C++, they could be written in Python, which led to faster development. C was only used for speed-critical sections of the code. By saving pointers to C data structures in Python objects and defining the `__del__` operator (called by the Python garbage collector), we re-used Python's garbage collector to deallocate C data structures.

NumPy, a Python library for efficient array operations, was also instrumental in developing these tools. Pointers to rows of NumPy arrays can be passed directly into C code, allowing render functions to populate images while keeping memory allocation and managment in the hands of Python.

The `kokopelli` UI is written in wxPython. This UI toolkit includes a text editor (in `wx.py.editwindow.EditWindow`) which was adapted into the editor seen in `kokopelli`. The OpenGL canvas is created using the `wx.glcanvas.GLCanvas` object, which is very poorly documented. PyOpenGL is also used for various OpenGL operations. OpenGL 2.1 and GLSL 1.20 were targeted, using a fairly minimal set of features. This OpenGL version was released in 2006, so we expect it to be supported on most modern hardware. Finally, Doxygen is used to automatically generate well-formatted documentation for the project.

This thesis was written in LaTeX, and graphs were generated using matplotlib.

Bibliography

[1] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant, *Numerical Python*, ucrl-ma-128569 ed., Lawrence Livermore National Laboratory, Livermore, CA, 1999.

[2] S. P. Austin, R. B. Jerard, and R. L. Drysdale, "Comparison of discretization algorithms for nurbs surfaces with application to numerically controlled machining," *Computer-Aided Design*, vol. 29, no. 1, pp. 71 – 83, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010448596000516

[3] Autodesk Labs. Designscript. [Online]. Available: http://labs.autodesk.com/utilities/designscript

[4] A. H. Barr, "Global and local deformations of solid primitives," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 21–30, Jan. 1984. [Online]. Available: http://doi.acm.org/10.1145/964965.808573

[5] T. Bastos and W. Celes, "GPU-accelerated adaptively sampled distance fields," in *Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on*, 2008, pp. 171–178.

[6] J. Bloomenthal and B. Wyvill, "Interactive techniques for implicit modeling," *SIGGRAPH Comput. Graph.*, vol. 24, no. 2, pp. 109–116, Feb. 1990. [Online]. Available: http://doi.acm.org/10.1145/91394.91427

[7] I. Braid, "Designing with volumes," Ph.D. dissertation, University of Cambridge, 1974.

[8] P. Brunet and I. Navazo, "Solid representation and operation using extended octrees," *ACM Trans. Graph.*, vol. 9, no. 2, pp. 170–197, Apr. 1990. [Online]. Available: http://doi.acm.org/10.1145/78956.78959

[9] J. A. Brentzen and K. Lyngby, "Manipulation of volumetric solids with applications to sculpting," Tech. Rep., 2002.

[10] S. Calisch. (2013, December) Shoe insert pressure sensor array. [Online]. Available: http://fab.cba.mit.edu/classes/MAS.863/people/calisch/14/foot.html

[11] Clay Mathematics Institute. (2013) P vs NP problem. [Online]. Available: http://www.claymath.org/millennium/P_vs_NP/

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[13] L. de Figueiredo, L. Velho, and J. de Oliveira, "Revisiting adaptively sampled distance fields," in *Computer Graphics and Image Processing, 2001 Proceedings of XIV Brazilian Symposium on*, 2001, pp. 377–.

[14] Digital Materialization Group. Hyperfun. [Online]. Available: http://hyperfun.org/

[15] T. Duff, "Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry," in *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '92. New York, NY, USA: ACM, 1992, pp. 131–138. [Online]. Available: http://doi.acm.org/10.1145/133994.134027

[16] E. Dyllong and C. Grimm, "A reliable extended octree representation of csg objects with an adaptive subdivision depth," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Springer Berlin Heidelberg, 2008, vol. 4967, pp. 1341–1350. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68111-3_142

[17] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones, "Adaptively sampled distance fields: a general representation of shape for computer graphics," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 249–254. [Online]. Available: http://dx.doi.org/10.1145/344779.344899

[18] A. J. P. Gomez, J. J. Voiculescu, B. Wyvill, and C. Galbraith, *Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms.* Springer, 2009.

[19] J. C. Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12, pp. 527–545, 1994.

[20] S. N. Harvey E. Cline and B. L. N. William E. Lorensen, "System and method for the display of surface structures contained within the interior region of a solid body," Patent, 12 1987, uS 4710876. [Online]. Available: http://www.patentlens.net/patentlens/patent/US_4710876/en/

[21] M. Held, "Voronoi diagrams and offset curves of curvilinear polygons," *Computer-Aided Design*, vol. 30, no. 4, pp. 287 – 300, 1998, ¡ce:title¿Computational Geometry and Computer-Aided Design and Manufacturing¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010448597000717

[22] ——, "Vroni: An engineering approach to the reliable and efficient computation of voronoi diagrams of points and line segments," *Computational Geometry*, vol. 18, no. 2, pp. 95 – 123, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925772101000037

[23] C. Ho, F. Wu, B. Chen, and M. Ouhyoung, "Cubical marching squares: Adaptive feature preserving surface extraction from volume data," *Computer Graphics Forum*, vol. 24, p. 2005, 2005.

[24] K. H. Höhne, M. Bomans, A. Pommert, M. Riemer, C. Schiers, U. Tiede, and G. Wiebecke, "3d visualization of tomographic volume data using the generalized voxel model," *Vis. Comput.*, vol. 6, no. 1, pp. 28–36, Feb. 1990. [Online]. Available: http://dx.doi.org/10.1007/BF01902627

[25] P.-C. Hsu and C. Lee, "The scale method for blending operations in functionally-based constructive geometry," *Computer Graphics Forum*, vol. 22, no. 2, pp. 143–158, 2003. [Online]. Available: http://dx.doi.org/10.1111/1467-8659.00656

[26] M. Kintel and C. Wolf. OpenSCAD. [Online]. Available: http://www.openscad.org/

[27] J. LaMarche. Prepping blender files for 3d printing. [Online]. Available: http://www.shapeways.com/tutorials/prepping_blender_files_for_3d_printing

[28] T. Lewiner, H. Lopes, A. W. Vieira, and G. Tavares, "Efficient implementation of marching cubes' cases with topological guarantees," *Journal of Graphics Tools*, vol. 8, p. 2003, 2003.

[29] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, Aug. 1987. [Online]. Available: http://doi.acm.org/10.1145/37402.37422

[30] D. Meagher, "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, vol. 19, no. 2, pp. 129–147, June 1982.

[31] A. Meijster, J. B. T. M. Roerdink, and W. H. Hesselink, "A general algorithm for computing distance transforms in linear time," 2000.

[32] Meshlab Stuff. On the subtle art of mesh cleaning. [Online]. Available: http://meshlabstuff.blogspot.com/2009/03/on-subtle-art-of-mesh-cleaning.html

[33] MIT Center for Bits and Atoms. Fab central. [Online]. Available: http://fab.cba.mit.edu/

[34] ——. Fab modules. [Online]. Available: http://kokompe.cba.mit.edu/

[35] R. Moore and C. Yang, "Interval analysis I," Lockheed General Research Program, Tech. Rep., September 1959.

[36] H. Müller and M. Wehle, "Visualization of implicit surfaces using adaptive tetrahedrizations," *Scientific Visualization Conference*, vol. 0, p. 243, 1997.

[37] G. M. Nielson and B. Hamann, "The asymptotic decider: resolving the ambiguity in marching cubes," in *Proceedings of the 2nd conference on Visualization '91*, ser. VIS '91. Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, pp. 83–91. [Online]. Available: http://dl.acm.org/citation.cfm?id=949607.949621

[38] C. Olah. ImplicitCAD. [Online]. Available: http://www.implicitcad.org/

[39] A. Opalach and S. Maddock, "Implicit surfaces: Appearance, blending and consistency," 1993.

[40] A. Pasko and V. Adzhiev, "Function-based shape modeling: mathematical framework and specialized language," in *Automated Deduction in Geometry, Lecture Notes in Artificial Intelligence 2930*, 2004, pp. 132–160.

[41] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko, "Function representation in geometric modeling: Concepts, implementation and applications," 1995.

[42] A. Pasko, O. Fryazinov, T. Vilbrandt, P.-A. Fayolle, and V. Adzhiev, "Procedural function-based modelling of volumetric microstructures," *Graphical Models*, vol. 73, no. 5, pp. 165 – 181, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1524070311000087

[43] R. N. Perry and S. F. Frisken, "Kizamu: a system for sculpting digital characters," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 47–56. [Online]. Available: http://doi.acm.org/10.1145/383259.383264

[44] A. Ricci, "A constructive geometry for computer graphics," *The Computer Journal*, vol. 16, no. 2, pp. 157–160, 1973. [Online]. Available: http://comjnl.oxfordjournals.org/content/16/2/157.abstract

[45] R. Schmidt, "Interactive modeling with implicit surfaces," Master's thesis, University of Calgary, August 2006.

[46] A. Sullivan, H. Erdim, R. N. Perry, and S. F. Frisken, "High accuracy nc milling simulation using composite adaptively sampled distance fields," *Comput. Aided Des.*, vol. 44, no. 6, pp. 522–536, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.cad.2012.02.002

[47] Uformia. Symvol. [Online]. Available: http://uformia.com/index.php/symvol-maker

[48] B. Victor, "Inventing on principle." Presented at CUSEC 2012, 2012. [Online]. Available: http://vimeo.com/36579366

[49] J. Ward. Mtm a-z pcb mill. [Online]. Available: http://mtm.cba.mit.edu/machines/mtm_az/