Rapid-Prototyping of Rapid-Prototyping Machines

An Integrated Approach to Design, Manufacturing and Control

To accompany the presentations given at the FAB5 conference in Pune, India, August 20, 2009.

Abstract:

Fostering the spread of inexpensive, sustainable manufacturing processes and equipment empowers individuals and societies to create products which solve their particular challenges. We intend to advance that idea to its next logical step, offering not just the means of production, but also the mechanisms by which this empowerment can propagate and evolve. By creating rapid-prototyping machines which are designed with the idea of self-replication, we accomplish the dual purpose of enabling both the distribution and the customization of the technology.

A system capable of addressing these goals requires software and firmware mechanisms for design and control that are freely redistributable, easily modifiable, and allow for a simple workflow with rapid iterations. No combinations of conventional software for computer-aided design and manufacturing have been found which adequately address each of these requirements. To meet these needs, we have designed a new software architecture for computer-aided creation. This document describes the design principles and our implementation of each component of the software suite.

Introduction

In “Democratizing Innovation”, Eric Von Hippel describes the phenomenon of product innovation being driven largely by users, rather than manufacturers, and quantifies this effect with several case studies. Indeed, when the comparison is made, it seems self-evident that users – who have unique needs, and are the direct beneficiaries of customization and improvement – should be more strongly motivated and better equipped to innovate than manufacturers. When manufacturing processes are not available to users, innovation is stifled, but as Von Hippel concludes, “When the cost of high-quality resources for design and prototyping becomes very low ... The net result is and will be to democratize the opportunity to create.” [Von Hippel, p. 123]

This document describes a modest collection of software, with the rather more immodest goal of enabling machines that help achieve Von Hippel’s projected change in the ownership of the “opportunity to create.”

The initial target for the software described here is specifically to enable 3-axis milling and/or deposition machines that can be used to create other similar machines for use in the worldwide network of Fab Labs. However, with the appropriate level of flexibility designed into the system, the hope is that the software will prove useful both for a wider class of machines and for other deployment possibilities.

The ideas in this document coalesced through the contributions of many people, primarily in the Spring 2009 instance of the “MAS 961: How To Make Something That Makes (almost) Anything” class at MIT [http://fab.cba.mit.edu/classes/MIT/961.09/], led by Prof. Neil Gershenfeld.
The concepts for self-replicating machines described here were heavily inspired by the RepRap [http://www.reprap.org] and Fab@Home [http://fabathome.org] projects, represented in the HTMSTMaA class by guest lecturers Adrian Bowyer for the former, Hod Lipson and Evan Malone for the latter.

The goal of the software is to provide an integrated system for:

1. Designing 2D and 3D geometry
2. Controlling a flexible set of both additive and subtractive manufacturing machines with an arbitrary number of axes and an arbitrary scale.

The design considerations also include the following features as requirements:

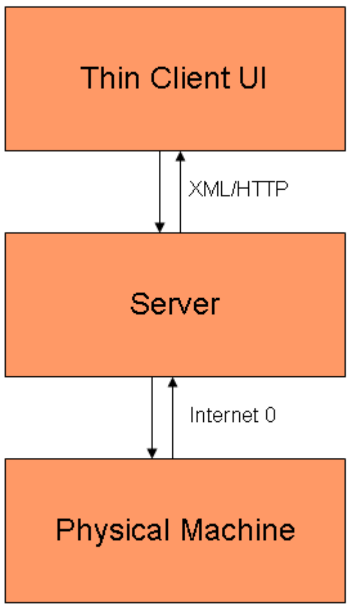
1. Store geometry in a scale-independent format.
2. Allow for rapid iteration and modification by encoding “design intent” in logical groupings and parameterized relationships between geometry components.
3. Use data formats that can easily be shared and used by others to create the physical objects they describe, even using other machines.
4. Provide a mechanism for simulating the machine activity prior to running the physical machine.
5. Consist of easily modifiable software components. Where practical, the implementation is done in a very high level language (VHLL) like Python.
6. Consist exclusively of freely modifiable and redistributable software. This is a strict requirement, in order to facilitate the growth and evolution of the system.

Software Architecture

Architecture Overview

At the highest level of abstraction, the system consists of a thin client user interface, connected to server software which is responsible for the core functionality, and which communicates directly with the rapid-prototyping device (referred to as the “physical machine” throughout this document).

No relative location is implied by the terms “client” and “server” – they may run on the same computer, or on different computers.

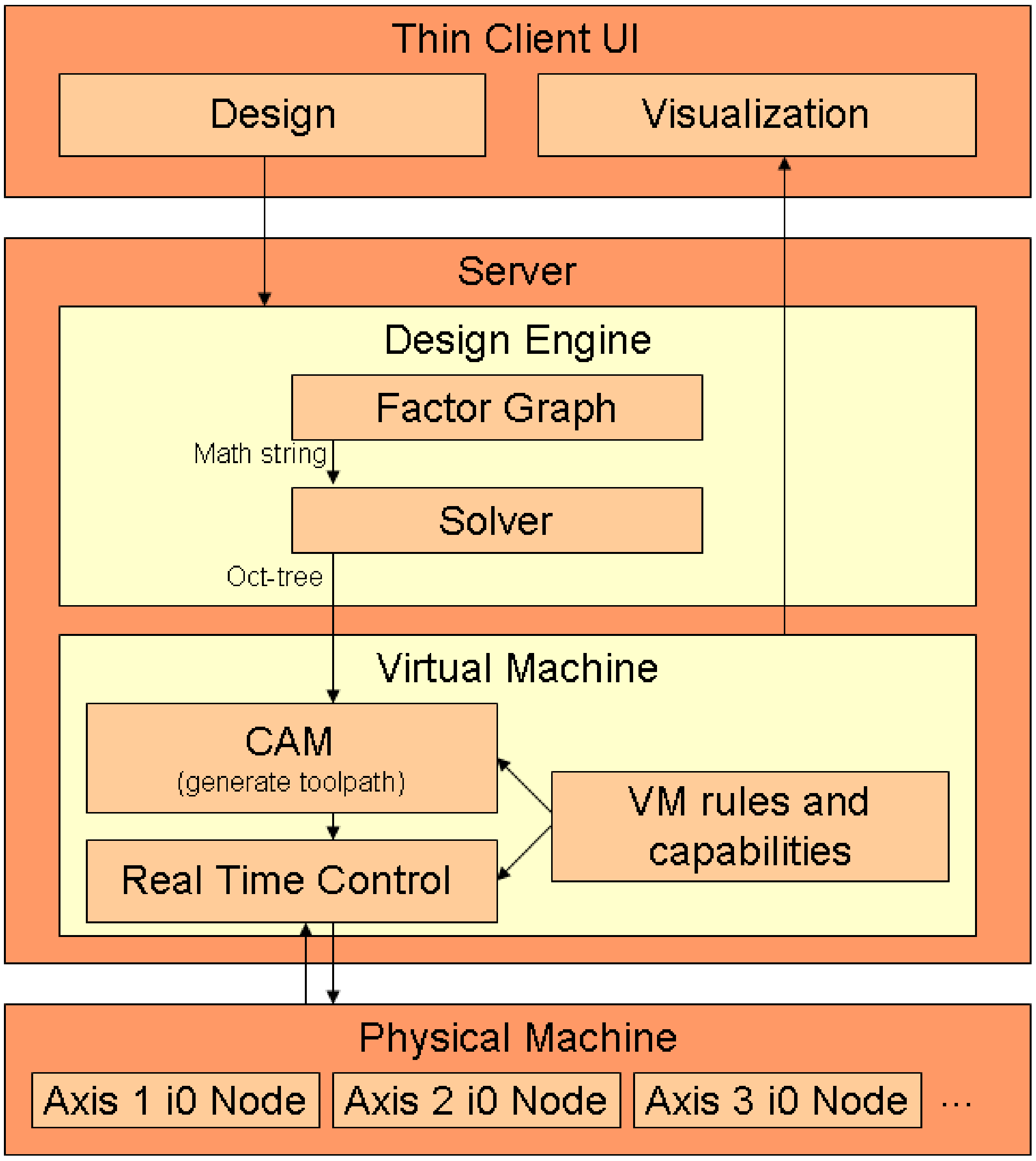


The client communicates with the server by sending XML formatted messages over HTTP, and thus may easily be configured for remote connectivity.

The server software sends machine control instructions to each axis of control in the physical machine using Internet 0, and thus the server and physical machine may in principle be in different locations. However, the assumption for the most common configuration is that the two will be physically proximate and that the i0 traffic will go over a local private network.
A complete description of Internet 0, is beyond the scope of this document. For the purposes of this architectural overview it is sufficient to assume that the motor controller for each axis is a node on a network, capable of receiving motor control instructions in the form of stateless packets. For additional details on Internet 0, see [ref].

Architecture Details

The following diagram provides a more complete view of each component and its relationship to the rest of the system:



These details will be elaborated upon in the discussion for each component.

Client User Interface

The client interface allows the user to see five types of information. These are:

1. A visual factor graph representation of the geometry
2. A python representation of the factor graph
3. A math string that represents satisfiability constraints to describe the geometry
4. [To be added in future revisions] A 3D visual rendering of the geometry
5. [To be added in future revisions] Visual feedback showing the state of the virtual and/or physical machine being used to construct the geometry

User Input

Users may provide input in a variety of ways, all supported by the primary user interface. Possibilities include, but are not limited to:

- Directly constructing/editing the factor graph (either using a mouse to manipulate factor graph elements or modifying the python representation)
- Directly writing the math string
- Importing images
- Importing other CAD formats

Factor Graph

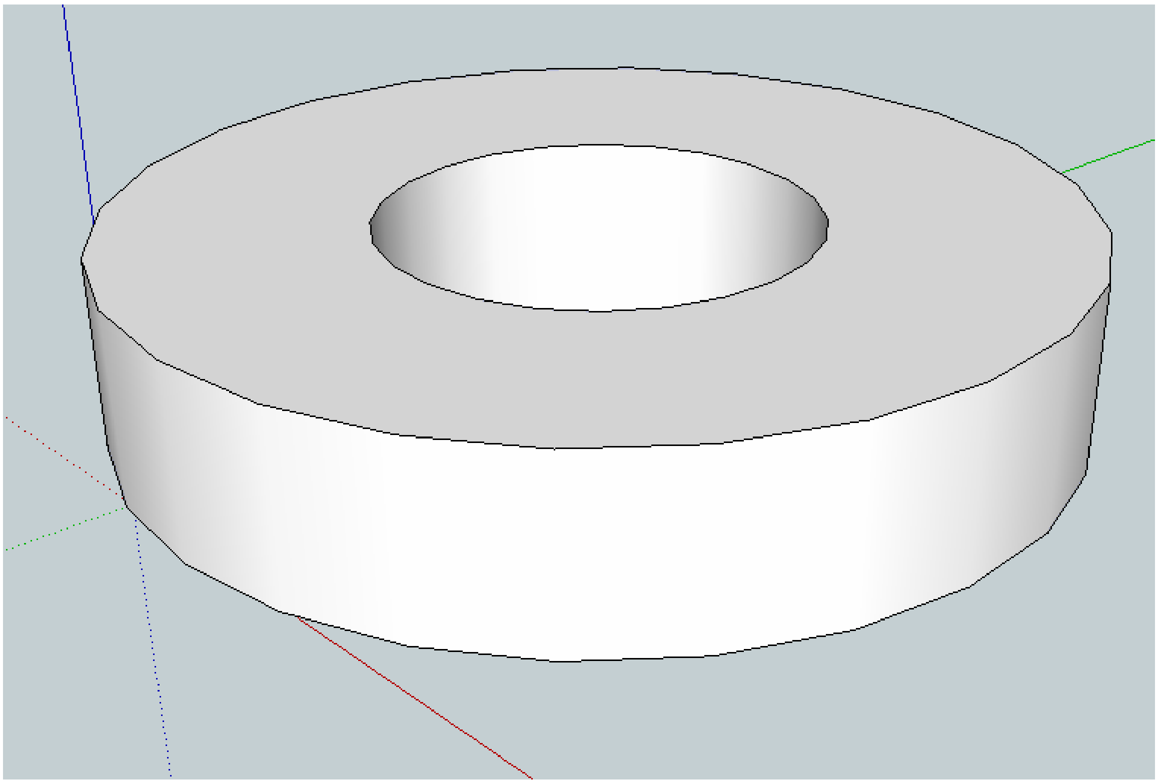
The factor graph describes physical and conceptual components and their relationships. For example, nodes in a factor graph might be “sphere”, “radius” and “cylinder”. The “sphere” and “cylinder” might have properties defining their location. The “radius” node might feed into both the “sphere” and the “cylinder” nodes, in order to constrain both of them to have the same radius.

The factor graph may be modified either through its visual representation of nodes and edges, or through the python code that supplies the concise form of the graph.

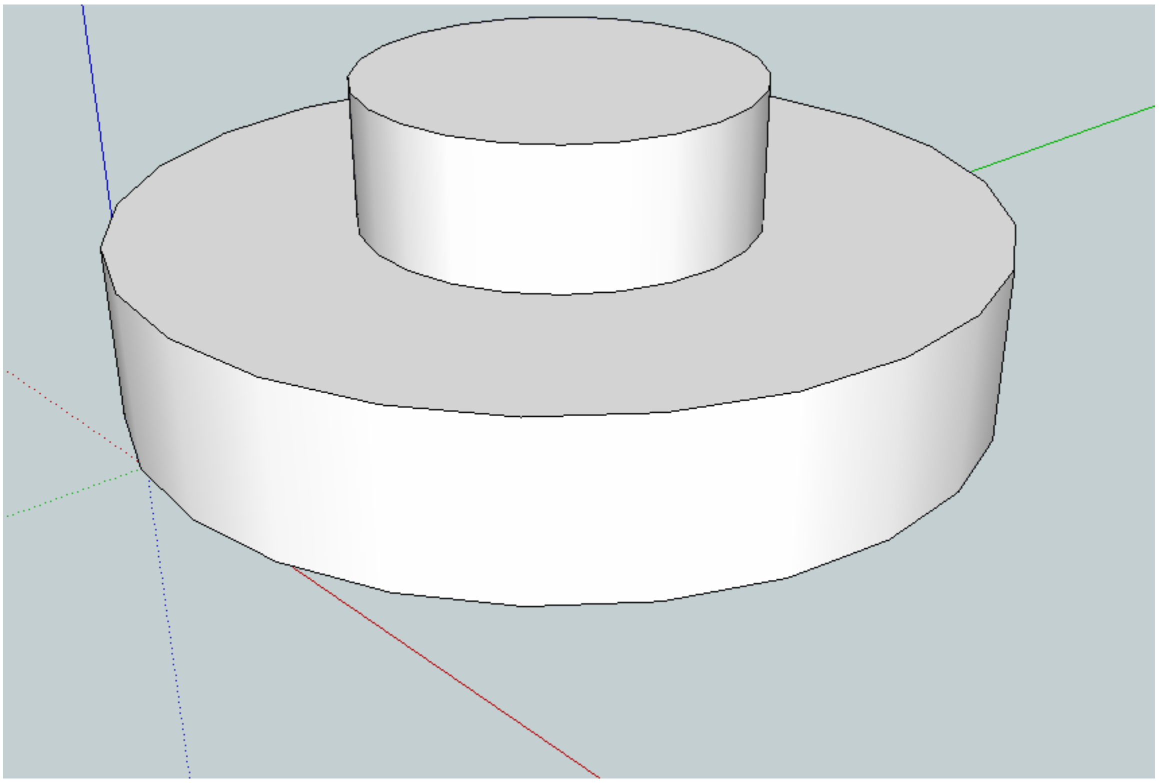
The python code is structured such that each node in the graph is a python object with a member field containing the math string that describes the node, and member fields with connectivity information (containing pointers to the objects representing connected nodes). When a connection between nodes causes a parameter to be overridden, this manifests itself as a string substitution in the math string for that node. Unlike other visual graph languages (Simulink, LabView, Iris Explorer), the connections to a given node are not restricted to a finite number of “ports” controlling a finite number of parameters. Instead, a node may make arbitrary string substitutions in the math string for a connected node.

In the visual form of the graph, nodes occupy a location in 3D space. By default, their location is used as meaningful input for parameters. For example, a sphere may be defined by a center point and a radius. The location of the sphere node in the factor graph will determine the default values for the X, Y and Z parameters describing the center point of the sphere. These can, however, be overridden by connecting other nodes.

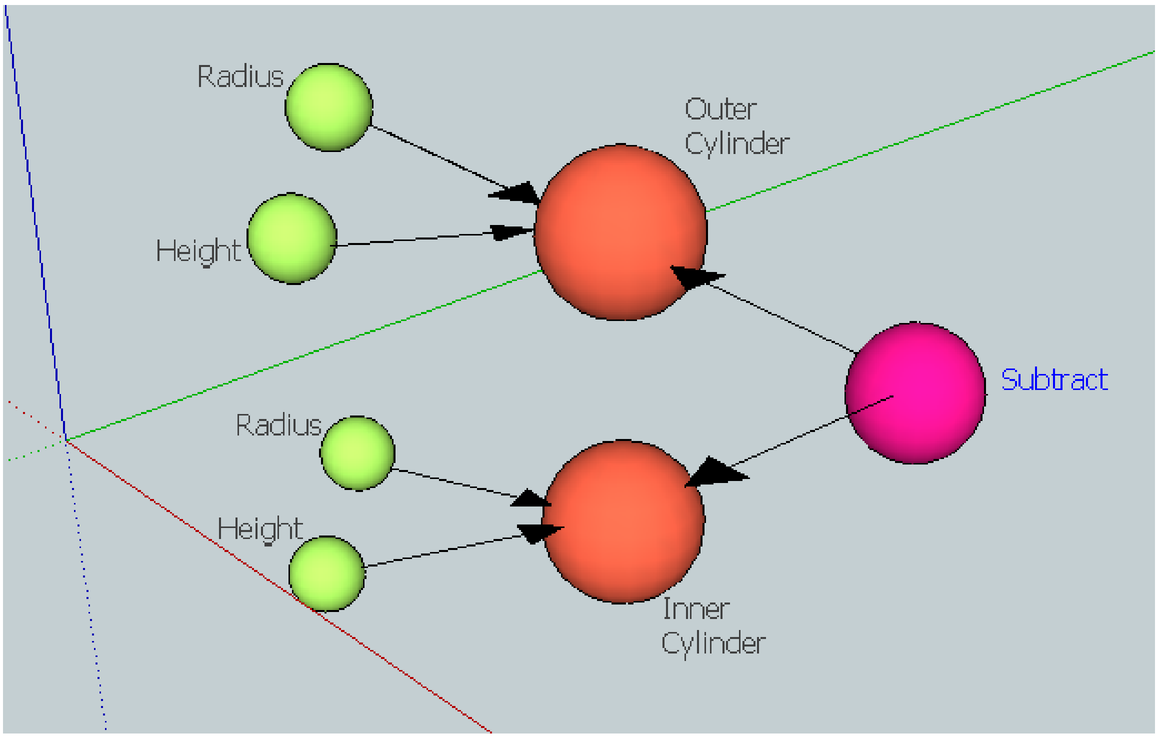
As an example, consider the geometry shown below. Note that the images of cylindrical object and factor graph are not actual screen shots of the software, however, as the factor graph UI reaches completion, it will be accessible via a public URL.



This was created by subtracting the inner cylinder from the outer cylinder in the following arrangement:



This set of operations can be represented via the factor graph shown here:



Math String

The math string will be familiar to users of Gershenfeld's cad.py software [http://fab.cba.mit.edu/about/fab/dist/cad.py], from which this concept is adopted. The math string consists of arithmetic and logical operators, with variables for orthogonal axes forming a basis for 2 (X,Y) or 3 (X,Y,Z) dimensional space. Coordinates for any point in space can be substituted into the math string, and its evaluation will result in a boolean value reflecting the presence or absence of material in the desired part at the specified location.

Consider, for example, the following math string:

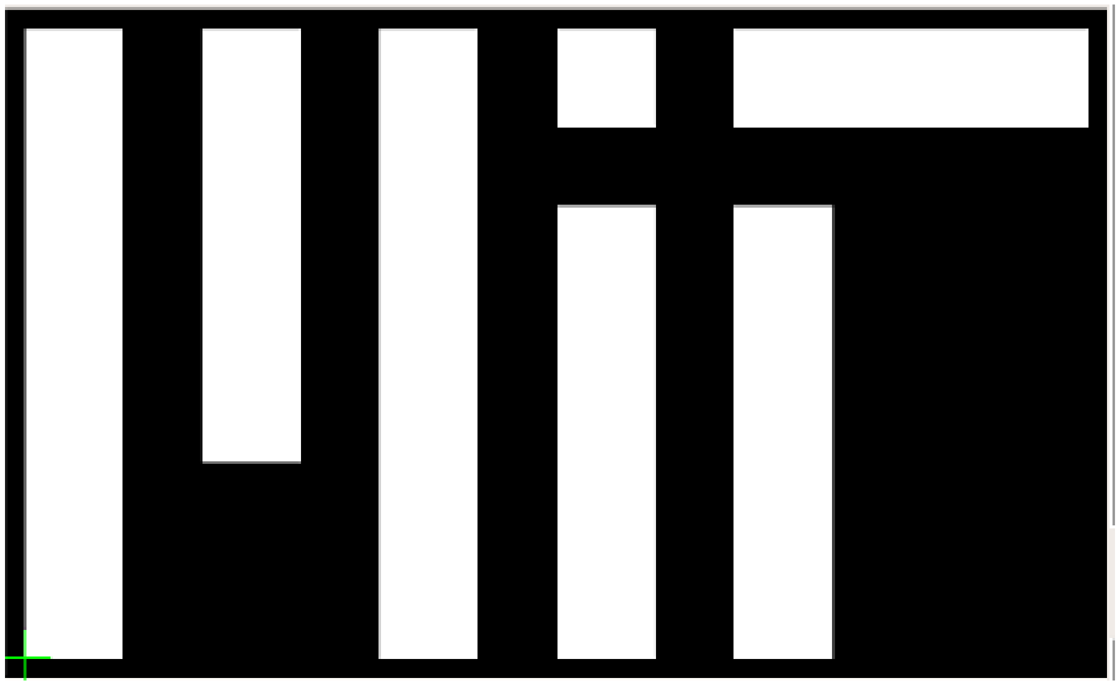
```
((X >= 0.5) & (X <= 1.5) & (Y >= 0.5) & (Y <= 1.5))
```

If we substitute values for X and Y coordinates that lie within the square whose lower left corner is at (0.5, 0.5) and whose upper right corner is at (1.5, 1.5), the expression will evaluate to True. If we substitute values for coordinates outside that square, the expression will evaluate to False. Therefore, this example math string represents a solid 2D square with the aforementioned size and position. Notice that Z is not represented in the string. This means that for any value of Z, we get the same result. Thus, if we use this math string in a 3D problem domain, it represents a rectangular solid, going to +/- infinity in the Z direction. For the purposes of constructing a physical object, we are free to choose a sensible action for an object whose extents go to infinity. We would typically take it to be fixed to some maximum height that may be supplied by the user as part of the configuration parameters for a given usage of a machine.

Notice that the math string provides a faithful representation of the geometry at all scales – coordinate values may be used with arbitrary levels of precision.

By using logical operators to combine mathematical expressions for primitive shapes, it is possible to build arbitrarily complex geometry. Transformations such as shearing, stretching and rotating can be achieved by adding arithmetic operations to the math string.

As a slightly more complex example, consider the MIT logo:



Which may be represented by the following math string:

```
((X >= 0) & (X <= 5) & (Y >= 0) & (Y <= 32)) | ((X >= 9) & (X <= 14) &
(Y >= 10) & (Y <= 32)) | ((X >= 18) & (X <= 23) & (Y >= 0) & (Y <= 32))
| ((X >= 27) & (X <= 32) & (Y >= 0) & (Y <= 23)) | ((X >= 27) & (X <=
32) & (Y >= 27) & (Y <= 32)) | ((X >= 36) & (X <= 41) & (Y >= 0) & (Y
<= 23)) | ((X >= 36) & (X <= 54) & (Y >= 27) & (Y <= 32))
```

Rendering

The term “rendering” could be used to refer to any of several different operations in different contexts in the system. In this document, we reserve the word “render” to describe the process of converting from the math string to representations suitable for output. This is the process analogous to converting from a scale-independent vector representation to a discretized raster representation when Postscript or PDF documents are printed or displayed.

The primary task of the rendering component is to convert the math string to a representation described by interval arithmetic on an octree, which is explained in the next section. However, the rendering system is also responsible for generating triangle facets for visualization, and will be extended to produce triangles for output to other geometry formats outside the Fab toolchain (e.g. output to STL).

The octree data structures have been implemented in C++, and are only accessible through the methods provided as the public API to the rendering component.

Octree and Interval Arithmetic

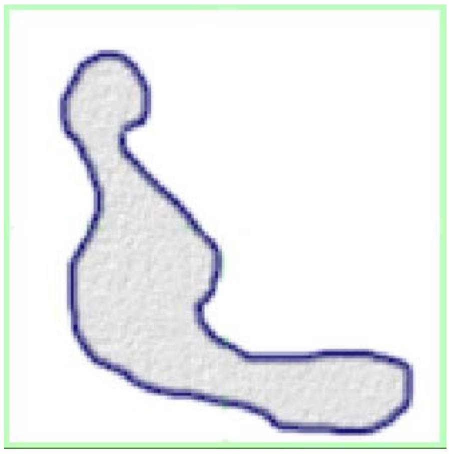
To explain the octree representation, we begin with a description of a discretization on a regular space-filling grid. The entire space (bounded at either user-specified limits or the extents of the geometry) is conceptually divided into a cubic lattice. The coordinates for a point in each lattice cell is substituted into the full math string, and the expression is evaluated to return a boolean value reflecting the presence or absence of material in each lattice cell.

The problem with such an approach is that the time required to do the computation limits the scope of its utility. For many 2D applications, such as milling a circuit board, the computation is slow but bearable, taking anywhere from several seconds to small numbers of minutes on a modern desktop. But it quickly becomes prohibitive when applied to 3D problems which consist of large numbers of 2D slices, and without any acceleration techniques, it would be a few orders of magnitude too slow.

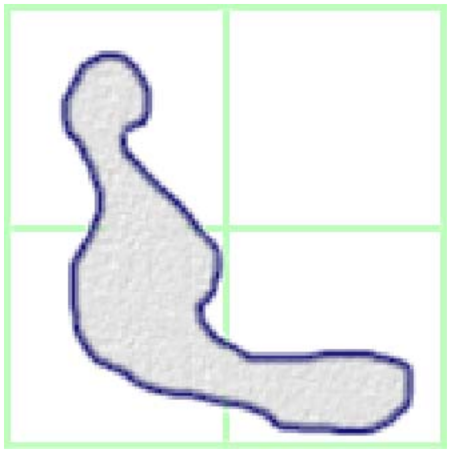
It should be immediately clear that there are optimizations that can be applied to reduce both the overall data size and the number of math string evaluations that need to be performed. It should be sufficient for us to compute and store information about the solid objects only near their surfaces, and to mark all the space between the surfaces as either filled or unfilled.

The spirit of this optimization can be achieved by using an octree where only the areas of interest are refined to a specified level of precision. “Octree” is the term for a data structure that is obtained by recursively slicing 3D space into eight parts by bisecting it with three orthogonal planes. A branch of the tree need not be further refined if the space represented by the leaf node is either all true (filled with solid material) or all false (empty).

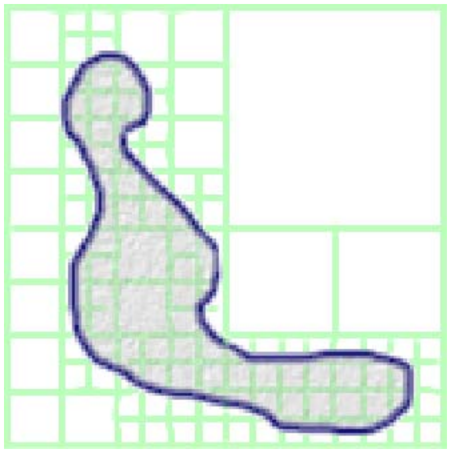
The concept can be explained further by giving an example with the 2D analogue of an octree – the quadtree. Consider the following object:



If there are any boundaries between solid and empty space in the entire region, then the region should be bisected in each dimension, resulting in four equal regions:



Each new sub-region is then considered, and if there are any transitions between solid and empty space in that subregion, then it too is divided. This algorithm proceeds recursively for as many subdivisions as desired, resulting in something like the following:



The complete set of subdivided regions is stored as a tree structure, with each region represented by a node in the tree. Each node will have either zero or exactly four (or in the case of a 3D octree, exactly 8) children. Those nodes with zero children are the leaf nodes of the tree.

In order to make use of an octree in the manner described, it is necessary to know whether a boundary exists within a given region of space. However, the actual use of the octree varies slightly from the preceding introductory description in that we do not provide the system with actual guarantees that a boundary exists in a given region as the octree is generated. Instead we consider a region to be of interest if it *might* include a boundary.

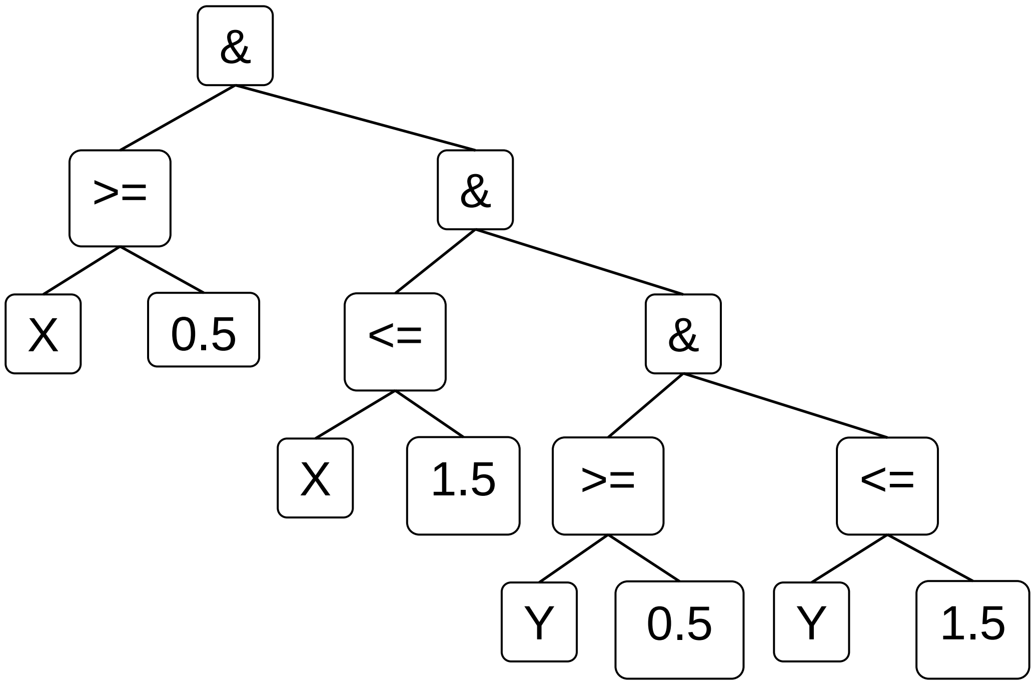
To explain, let us return to the simple example of a 2D unit square, defined by the following math string:

```
((X >= 0.5) & (X <= 1.5) & (Y >= 0.5) & (Y <= 1.5))
```

Without any optimizations, if we take geometry represented by a math string and wish to create a discrete rendering with a specified resolution, we must evaluate the full math string at every coordinate on a regularly spaced grid. There are two clear opportunities for optimization:

1. Don’t perform the evaluation everywhere
2. Don’t evaluate the entire expression

If we examine the math string for the square, and consider it in the manner a typical language parser would approach it, we see that it is an expression composed of several subexpressions. It can be viewed as a tree of operators and operands:



The topmost operator in the tree is a logical AND, which tells us that both the left and right branch must evaluate to True in order for there to be solid material at a given coordinate. Examining the left branch (X >= 0.5), it is clear that this will never be true for X<0.5, and therefore we immediately know that in the entire region X<0, there is no need to ever evaluate the expression at all. We have also learned something about the entire space where X>=0.5. In that portion of the plane, the (X>=0.5) branch will always evaluate to true. This doesn’t mean that the whole math string evaluates to true for all X>=0.5, but it does mean that there’s no need to evaluate the (X>=0.5) branch of the math string, because we already know the answer.

This information provides us the ability to make the following three conclusions about the resulting quadtree:

1. Subregions that only contain space in the X>=0.5 portion of the plane do not need to be resolved any further.
2. Subregions that span the X=0.5 line may (but are not guaranteed to) contain boundaries between solid material and empty space, so they need to be resolved further.
3. Subregions that only contain space in the X>0.5 portion of the plane may contain boundaries and need to be resolved further, but for these areas, the math string can be pruned to a shorter expression, eliminating the first condition.

If we continue to process the entire math expression tree and build the quadtree, we will end up with a quadtree that contains fine resolution for any areas that span the lines X=0.5, X=1.5, Y=0.5, Y=1.5. This includes finely resolved areas along these lines outside of the intended square geometry that will not turn out to contain any boundaries, but we have ruled out large swaths of area in which we know that nothing interesting is happening. In all of the areas that are finely resolved by the quadtree, we have also pruned the math string so that it is never necessary to evaluate the entire expression when it comes time to perform a discretization and identify where the actual boundaries are, to some specified level of precision.

This works extremely well for reducing the cost of time cost of rendering our square example. However, there is an additional trick required in order to accommodate more complex geometric descriptions. Imagine that, instead of a square, the left edge of the object is defined by a sine wave. For example:

```
( ( X >= sin(Y/(2*pi)) ) & (X <= 1.5) & (Y >= 0.5) & (Y <= 1.5))
```

In this case, it is no longer immediately clear how to build the quadtree, because it becomes difficult to determine whether a boundary might occur in any given quadtree region. The solution is to determine the minimum and maximum possible values for the `sin(Y/(2*pi))` portion of the expression. We happen to have special knowledge about the range of possible return values for the `sin()` function, and we need to take advantage of this information. For the purposes of refining our quadtree, the subexpression `( X >= sin(Y/(2*pi)) )` becomes `( X >= some value in the range [-1,1] )`. We can then apply that knowledge to reach conclusions similar the ones above, but with a somewhat wider band of possible locations where a boundary might occur, and a correspondingly wider region in which the quadtree needs to be refined.

When the math string includes even more complicated arithmetic expressions, we simply need to perform interval arithmetic by operating on combinations of the minimum and maximum values in the range operands.

It is possible for the approach to break down to the worst case in which the range information runs to infinity. It is also possible that the math expression may contain an operation that is not valid for interval arithmetic (many nonlinear operations and calls to python functions that perform something other than basic arithmetic operations would disqualify this approach). In those situations the technique reduces to the unoptimized case given in the introductory explanation. However, in practice, these problems are rare, since users can create any physically realizable object (and many unphysical objects) using a library of functions that is restricted to operators for which there are valid interval versions.

CAM

The CAM component is responsible for taking the octree representation as input, and providing machine instructions as output. In order to do this, it must have all the information describing the capabilities of the physical machine, and generate appropriate instructions for toolpaths, travel speeds, feed rates and/or bit-changing instructions, and any other relevant data.

It begins with the construction of a scale-specific discretized identification of the location of the surface of the object(s). Once the boundary has been identified, this can be used as the basis for filling the solid region (for additive processes) or the empty region (for subtractive machining) with contours at a user-defined spacing, typically corresponding to the tool width for a given machine.

The algorithm for finding the boundaries of the object is a modified grassfire approach. In the explanation of the octree above, we began with a description of a uniform lattice of 1-bit values, and then introduced the octree as an optimization. Similarly, to describe the edge-finding algorithm it helps to consider the task as applied to a regular grid of 1-bit values, and then examine the modifications required to handle data that begins as an octree representation.

If we are presented with a regular grid of 1-bit values and given the task of finding the edges and then filling the area adjacent to the edges with a specified number of contours, there are several possible approaches. For finding the edges, we could use the approach of choosing a starting pixel, and then expanding out from the initial pixel until we encounter boundaries. This is referred to as a grassfire algorithm – the name coming from the nature of the expanding front. The same basic technique can be used for growing the contour lines from the boundaries. If, however, we use this approach for finding the initial boundaries, it is not sufficient for us to stop expanding the front at the boundaries, because holes (nested boundaries) would never be found, and more work would have to be done to locate them. In practice, if we want to reliably locate all nested boundaries in our regular grid, it is necessary to visit every pixel in the space. Therefore, the initial edge-finding can just as easily be done with a simpler approach, like applying a stencil operator to the entire data set. We can then sensibly use either approach for growing the boundaries to generate contours (for this second task there are some tricks that can make the grassfire approach superior to the stencil approach, such as retaining a reference to each unique boundary, and only visiting the neighborhood immediately adjacent to the boundaries for all subsequent steps).

When the data is represented as an octree, our task is made more complex by the fact that there is not a constant number of neighbors bordering each cell, and the method of traversal is not as simple as accessing neighboring elements in an array. However, we have the advantage that the data has already been pruned so as to not contain an explicit representation of each unit of area in large contiguous regions with no boundaries.

In order to find the boundaries in the octree, we have chosen an approach wherein we first apply the logical equivalent of a stencil operator, visiting each explicitly represented region (each leaf node of the octree), and comparing its value with the immediate neighbors. Note that not all cells have the same number of neighbors, and that finding the neighbors requires traversal of the tree structure. In order to traverse the tree to find spatial neighbors, it is necessary to ascend and descend a portion of the tree. The worst case is for nodes representing regions adjacent to the centerline of the entire space. For these, in order to find the neighbor on the other side of the center line, it is necessary to ascend all the way to the root node of the tree and back down. Notice, however, that the octree is of a fixed depth which can be independent of the complexity of the geometry, and the overall cost is small.

Once we have found the boundaries, we go ahead and build the fully refined representation of the boundary (as though we returned to the regular lattice, but only for the areas on boundaries), and we are then free to ignore the octree structure and grow the contours from those boundaries. Even if we use a full regular lattice while growing these boundaries, we have saved ourselves from ever having to visit the interior of large contiguous regions.

While growing the contours, it is necessary to evaluate the math string subset in each cell of interest, in order to determine whether it is solid or empty space.

The time savings obtained from techniques used here depend on the ratio of the surface area of the object(s) to the total volume in the workspace. In practice, we find that this is noticeably helpful when used for a 2D operation or a single Z-slice (e.g. when milling a circuit board), and for 3D problems, it allows us to run successfully with problems and resolutions that would previously have been completely impractical.

STL

The ability to read and write STL files is a lower priority feature, since it is not part of the recommended workflow. However, there have been situations in which it has proven useful for interfacing with other tools.

The rendering component is responsible for converting the octree representation into triangles for STL output. This is dependent upon user-supplied parameters for identifying the minimum feature size in order to determine an appropriate level of resolution.

Visualization

Visualization is to be used both for displaying 3D views of the object during the design process, and for viewing the current state of the machine as it operates. Visualization is accomplished by converting the geometry to triangles in the rendering component, and shipping these back to the client.

A current working assumption is that JOGL [https://jogl.dev.java.net/] will be used for display purposes. However, this has not yet been implemented.

Virtual Machine

The primary task of the virtual machine is to receive the instructions from the CAM component, to simulate the machine movements necessary to carry out those instructions, and to forward the lower-level machine movement instructions to the physical machine.

Notice that the Virtual Machine is not an optional part of the workflow. It is used for driving the physical machine, in contrast to other possible system designs where the VM takes the same instructions as the physical machine for input, and plays the role of a simulator, outside the main workflow.

The rules and capabilities of the machine consist of both data and logic. For a 3-axis extrusion machine, we might have rules like the following:

- 5 primary controls (“dimensions”?): x, y, z, extrusion temperature, extrusion speed
- Constraints on minimum and maximum speed in the xi+yj direction
- Minimum and maximum values for each dimension
- Maximum path length
- Quantity of extrudable material to be ejected before pausing
- Step size in the z direction

The language for sending data to the virtual machine is described in the “Data Formats” section later in this document.

Physical Machine

The physical machine executes instructions supplied by the virtual machine, and where possible, supplies feedback to the virtual machine.

The control for the physical machine is accomplished via a network of motor controllers. Each controls an individual axis, where “axis” is used with a loose definition to include not just X, Y, Z positioning, but also material feed rates for deposition machines, bit rotation speed for milling machines, and any other controllable aspect of the system. The instructions are broadcast from the software implementation of the virtual machine, and each packet uses the destination port to indicate the axis for which the message is intended. The motor controllers for all axes are able to see all network traffic, but only act on instructions that are intended for their consumption.

The virtual machine is responsible for generating the i0 traffic that drives the physical system.

Data Formats

A key innovation in the geometry data format is to use a device-independent description that contains program logic to create the object, rather than a simple static representation. This is similar to the design used by John Warnock for the Adobe Postscript language for documents, later also used in PDF [ref]. It is a convenient mechanism for retaining a scale independent representation through as much of the tool chain as possible, which simplifies the task of producing the final output on an arbitrarily wide range of machines. In the case of the Adobe document formats, the language is a scale-independent programmatic description of vectors, which gets converted to a raster format only at the time that it gets printed or displayed.
In the case of the Fab toolchain, the description passes through three scale-independent programmatic representations and one discretized raster format. It may be a reasonable goal to eliminate the need to ever stop over in the discretized format as the system evolves. The set of representations are as follows:

1. The programmatic description of the factor graph.
2. A math string which can be evaluated at any point in space and at an arbitrary level of precision to provide a boolean indication of the presence or absence of material at that location.
3. A discretized representation of space. For simplicity, this can be thought of as a space-filling regular grid of a finite specified resolution, with a boolean value for each grid cell. In practice, to avoid the need to evaluate and store information on all points in space, this is represented with an octree.

4. A set of machine control instructions for creating the part. The machine control instructions nominally include specific scale information and are targeted to a particular output device, but are parameterized for some scaling flexibility.

The toolpath generated by the CAM component is encoded as python instructions that can be understood by the virtual machine. Another way of saying this is that python is the scripting language for the virtual machine, and that it provides a pre-specified API for machine control.

The most commonly used instruction is a call to the move() function. The CAM component attempts to retain as much parameterization as possible in the final instructions, but much is necessarily converted to absolute coordinates at this stage.

The following toolpath code was generated from the math string representing the MIT logo that was shown in an earlier example (the coordinates differ from those visible in the math string only because a scaling factor has been applied):

```
traverse_speed = 8
cutting_speed = 4.0
plunge_speed = 4.0
z_down = 0.0
z_up = 0.1
move(z=z_up, rate=plunge_speed)
move(0.0, 0.0, z_up, traverse_speed)
move(z=z_down, rate=plunge_speed)
move(0.0462962962963, 0.0, z_down, cutting_speed)
move(0.0462962962963, 0.296296296296, z_down, cutting_speed)
move(0.0, 0.296296296296, z_down, cutting_speed)
move(0.0, 0.0, z_down, cutting_speed)
move(z=z_up, rate=plunge_speed)
move(0.0833333333333, 0.0925925925926, z_up, traverse_speed)
move(z=z_down, rate=plunge_speed)
move(0.12962962963, 0.0925925925926, z_down, cutting_speed)
move(0.12962962963, 0.296296296296, z_down, cutting_speed)
move(0.0833333333333, 0.296296296296, z_down, cutting_speed)
move(0.0833333333333, 0.0925925925926, z_down, cutting_speed)
move(z=z_up, rate=plunge_speed)
move(0.166666666667, 0.0, z_up, traverse_speed)
move(z=z_down, rate=plunge_speed)
move(0.212962962963, 0.0, z_down, cutting_speed)
move(0.212962962963, 0.296296296296, z_down, cutting_speed)
move(0.166666666667, 0.296296296296, z_down, cutting_speed)
move(0.166666666667, 0.0, z_down, cutting_speed)
move(z=z_up, rate=plunge_speed)
move(0.25, 0.0, z_up, traverse_speed)
move(z=z_down, rate=plunge_speed)
move(0.296296296296, 0.0, z_down, cutting_speed)
move(0.296296296296, 0.212962962963, z_down, cutting_speed)
move(0.25, 0.212962962963, z_down, cutting_speed)
move(0.25, 0.0, z_down, cutting_speed)
move(z=z_up, rate=plunge_speed)
move(0.25, 0.25, z_up, traverse_speed)
move(z=z_down, rate=plunge_speed)
move(0.296296296296, 0.25, z_down, cutting_speed)
move(0.296296296296, 0.296296296296, z_down, cutting_speed)
move(0.25, 0.296296296296, z_down, cutting_speed)
move(0.25, 0.25, z_down, cutting_speed)
move(z=z_up, rate=plunge_speed)
move(0.333333333333, 0.0, z_up, traverse_speed)
move(z=z_down, rate=plunge_speed)
move(0.37962962963, 0.0, z_down, cutting_speed)
move(0.37962962963, 0.212962962963, z_down, cutting_speed)
move(0.333333333333, 0.212962962963, z_down, cutting_speed)
move(0.333333333333, 0.0, z_down, cutting_speed)
move(z=z_up, rate=plunge_speed)
move(0.333333333333, 0.25, z_up, traverse_speed)
move(z=z_down, rate=plunge_speed)
move(0.5, 0.25, z_down, cutting_speed)
move(0.5, 0.296296296296, z_down, cutting_speed)
move(0.333333333333, 0.296296296296, z_down, cutting_speed)
move(0.333333333333, 0.25, z_down, cutting_speed)
```

Future Directions

It should be apparent from the descriptions in this document that each component still has several features which remain to be implemented or improved. However, a large part of the intent behind this project is to take advantage of user-generated innovation, as discussed in the introduction. In that spirit, the system is being made publicly available, even at a relatively early development stage.

Next steps include:

- Completion of the factor graph UI
- Visualization of geometry during the design process
- Visualization of the state of the virtual machine
- Improved mesh quality in the STL output
- Active feedback to the virtual machine, for those physical machines which support it
- Performance enhancements
- Usability enhancement, based on user feedback and contributions

By deploying the entire system in the global network of Fab Labs, it is expected that iteration will indeed be rapid.

References:

Von Hippel, Eric, Democratizing Innovation, The MIT Press, Cambridge, MA, 2005.
http://web.mit.edu/evhippel/www/books.htm

The Internet of Things; October 2004; Scientific American Magazine; by Neil Gershenfeld, Raffi Krikorian and Danny Cohen;
http://www.sciamdigital.com/index.cfm?fa=Products.ViewIssue&ISSUEID_CHAR=E9FFCBE6-2B35-221B-661ED2EFA6A79697

John Warnock; The Computer Bulletin; Volume 46, Number 4; pp. 16-17, 2004.
http://itnow.oxfordjournals.org/cgi/reprint/46/4/16.pdf

http://www.reprap.org

http://fabathome.org