MASTERS OF ENGINEERING PROJECT

FabStudio Final Report

Software for the Fab@Home

Karl Gluck, ECE '10, M. Eng CS '10

May 2010

CORNELL COMPUTATIONAL SYNTHESIS LAB, ITHACA, NY 14850

TABLE OF CONTENTS

ABSTRACT

This semester, I led the development of FabStudio, a cross-platform application that enables printing using the Fab@Home Model 2. The project's primary objective was to implement a program that generates a valid .FAB-format file from user-selected 3D object models and materials. As the lead software developer, my semester was defined by two categories of work: organizational tasks and development tasks. It was my responsibility to coordinate and guide the programming team, most of whom were new to the project, and encourage contributions while familiarizing them with the Fab@Home, team programming and our development cycle. In the category of development tasks, I wrote and debugged the majority of the 23,000 lines of code that comprise FabStudio.

We have made significant progress on the project objective. At the writing of this report, FabStudio is running and has been demonstrated to create functional .FAB files. It has been successfully used by both technical and non-technical users and their feedback recorded or integrated. Additionally, the shared code-base is mature enough to begin developing other 3D printing applications with it. Finally, my team and I have produced documentation and video tutorials that detail how to use FabStudio. FabStudio is now Alpha-quality software, in that it is stable enough to be distributed to users, provided those users can tolerate occasional unexpected behavior and usability issues as well as the need to download periodic updates from developers.

OBJECTIVES

The technical development of FabStudio had three main objectives. First, minimize the amount of knowledge necessary to effectively utilize the program's basic functionality. The purpose of this goal is to bring 3D printing with the Fab@Home to a wider audience by reducing the technical barrier to utilizing the technology. Second, enable advanced, technical users to control detailed aspects of the printing process. The Fab@Home was originally developed as a research project and has been adopted and modified by academics and technical users the world over. The software needs to be powerful enough to be useful for these users. Third, create a C++/Qt code-base that is reusable, modular and professional so that it can be the basis for other 3D printing applications. While FabStudio is intended to be a general-purpose printing utility, it is foreseeable that developers may want to incorporate the ability to print on a 3D printer as a feature into other applications. This new suite of code should be created with such use as a design goal, so that it can be done with only a minimal number of changes to the core software.

I had two goals for the Fab@Home programming team. First, to recruit a new group of students and get them interested in working on the Fab@Home. The continued development of the Fab@Home at Cornell depends on passing knowledge of the software and an interest in the project to junior classes. To this end, I provided the team with a balanced workload that would allow this team to make meaningful contributions to FabStudio even though most initially lacked experience in C++ and none had used Qt. My second goal was to familiarize them with 3D printing technology and the software

development process so that those motivated to do so can lead the maintenance and development of the Fab@Home software suite.

Background

Fab@Home

The software that drives the Fab@Home into two main areas: the program that creates a FAB file containing printer instructions from files describing objects, and the program that takes that FAB file and issues hardware instructions to the printer. FabStudio represents the first program, and FabInterpreter the second.

Basic Technical Knowledge Assumptions

Most of this report is written for a general technical audience with programming experience. Familiarity with the following topics is assumed:

General programming and related concepts

Threads, mutexes, inheritance, file formats, object oriented design

3D and Vector Math

Operations with 3- and 4-component vectors, matrix transforms, quaternions

Software Engineering

Development process, encapsulation/abstraction

Definitions

The following terms appear throughout this report:

Mesh

A mesh is a representation of an object. It is a set of triangles that collectively the outer surface of some solid. A mesh can have one outer surface and zero or more inner surfaces.

STL File

STL stands for "Stereolithography", and this file format is the standard way of encoding meshes for 3D printing. It is also an unfortunate standard in that it can contain no information about an object other than its triangulated structure, and even that is problematic due to coincident points being stored in multiple locations. Use of this format, while commonplace, should be considered deprecated; the AMF format should be preferred.

AMF File

An "Additive Material Format" file, which is a superior alternative to STL files for representing printable meshes, developed in the CCSL by Jon Hiller.

TOOL-SCRIPT

Analogous to a driver for a hardware device; it provides the settings and instructions necessary to translate a set of meshes into a FAB file that can be used to drive the print job. ToolScript is a custom-developed XML-based file format.

SLICING

Taking a mesh and cutting it horizontally by planes perpendicular to the Z axis to find the inner and outer loops that define the edges of the object on that plane. This step is performed by a program component called a slicer, and is the first step in translating a mesh for printing.

PATHING

The algorithm that defines how a tool should deposit material given a slice of an input mesh is performing a pathing algorithm. In FabStudio, the 'simple crosshatch pather' creates a path that fills in the object entirely.

FAB FILE

An XML-format file that encodes tool settings and deposition paths that the FabInterpreter can use to drive the Fab@Home Model 2 printer.

FABINTERPRETER

A program that reads a FAB file and translates it into hardware instructions that operate the physical Fab@Home printer.

OPENGL

A library provided with Qt that enables the development of applications that render 3D scenes

SUPPLEMENTARY ADVANCED TECHNICAL KNOWLEDGE

Those aiming to use this report in order to become familiar enough with the project to continue its development should have knowledge of the following:

OPENGL

OpenGL is a graphics library used for rendering 3D scenes. Relevant topics include the rendering pipeline, shading models, display lists. For tutorials, try looking at game programming websites for introductions to creating OpenGL applications.

ADVANCED C++

Template classes, virtual functions, memory management

Qt

Qt is a free (LGPL) C++ library developed by Nokia that enables cross-platform GUI application development. One should be familiar with signals/slots, forms, Qt extensions to C++ and Qt classes. There are many guides to programming with Qt available online.

Organizational Approach

The semester began with recruiting new team-members to assist with the development of FabStudio. It was my responsibility to evaluate candidates and organize the team so it would be productive. This involved determining how to incorporate their contributions as to the rapidly-changing central program, as well as outlining their work and the flow of tasks through the semester. In our early meetings, I evaluated their ability to handle some basic programming tasks in a familiar, forgiving environment (MATLAB) that led directly into more relevant and complex challenges. We followed an iterative development process, each week adding on to or debugging what they had accomplished the following week. After the team became comfortable with 3d programming and the tasks I had assigned, I moved them from developing in MATLAB to working with C++ and, finally, importing their code into the now-stabilized FabStudio framework. The semester concluded with the team performing two weeks of technical and user testing to improve the usability of FabStudio.

Technical Approach

I began FabStudio's development by outlining the design goals for this semester after talking with Jeff Lipton. The primary essential feature of FabStudio, and first goal, was to create a functioning pipeline from end-to-end, where users could generate FAB files with minimal difficulty from source STL files. Feature-goals included object manipulation, support for additional file formats, automatic support material generator, 3D path previews, adding multiple objects to a scene and utilizing/extending the ToolScript concept from last semester.

The technological requirement for FabStudio was that development was to occur in Qt. Qt fulfills the requirement of developing a portable (cross-platform) application, it makes creating UIs very easy, and it works with C++. C++ is important for our development for legacy reasons, but also because it is low-level enough to support direct hardware interaction, most professional and academic developers have experience with the language.

I should note that, with development time for this project limited to exactly one semester, it was also important that I maintain a strict schedule and postpone any nonessential features for later addition or revision once the core product was working. This proved to be a major win for FabStudio, as being able to stay focused dramatically improved the quality of the end result and has allowed me to devote adequate time to the often-overlooked task of preparing documentation for my successors.

The coding process for FabStudio was thus planned to progress as follows. First, implement the minimum functionality: being able to create a FAB file from an STL. To determine the order of

programming tasks, I developed a feature dependency tree to achieve this functionality. Having programmed this pipeline before for the AppRunner made this task easier since I knew what to expect; for example, the object-loading code needs to be implemented before objects are rendered, matrix math precedes rendering, rendering precedes object manipulation, manipulation fits before printing, and materials also need to be assigned before printing. With this rough guide, the goal was to be finished with the minimum functionality of FabStudio approximately halfway through the semester. During this initial stage of development, parts of the program would be rewritten and changing frequently as its structure matured. It would be difficult to incorporate the team's code during this time, so that left approximately one-quarter of the remaining time in the semester for adding postponed features and team contributions, then the final quarter for testing and reporting. We nearly met this timeline, but ended up taking a bit longer to add final features and spent less time on testing than I would have liked.

Implementation

Team Progress

In February, Jeff presented me with a list of programmers that had applied to be part of the Fab@Home team. I evaluated each one's credentials and most seemed to have enough technical knowledge to be able to make a meaningful contribution to the project, so I ended up selecting a team of 5 new members (plus Nathan, who remained on the team from last semester). Unfortunately, though Sabina Sobhani was originally selected, she decided she would not be able to contribute and thus does not appear in this report. Jeremy Cohen joined a short while later but I will include him in this selection list for consistency:

- Chris Hogan - CS '13
- Karina Sobhani - CS '13
- Jeremy Cohen - CS '13
- Nathan Lloyd - CS '10
- Jimmy Liu - CS '13
- Jason Zhao - CS '13

I held weekly meetings on Monday from 5:00 PM - 8:00 PM in Upson 217. After an update on the state of FabStudio and a progress report from each member, I would talk about what we would be working on that day and our tasks for the week ahead. During meetings, I helped groups complete their tasks or explained enough that they could finish on their own the following week. To help keep things moving forward each week, I requested weekly updates by email every Friday from each group member.

By the beginning of March, our team began working in MATLAB as most were comfortable in the environment. Assignments were:

- Karina & Jimmy - Load and plot in 3D an ASCII-format STL file; then FAB file; then binary STL file
- Chris - Finding best print rotation by determining the center of gravity of an arbitrary triangle mesh (used loader developed by Karina & Jimmy)

- Nathan & Jason - Turning a black-and-white image into an STL
- Jeremy - Optimize build tray layout in standalone C++ program

In the middle of March, we began the transition to Qt/C++. I introduced the group to using SVN and had them do an introduction to programming with Qt using online tutorials for the program. After having them convert their respective MATLAB projects to C++ code, I created "stub" functions in the main FabStudio project and showed them where they would integrate their work. This went smoothly, and by the middle of April their code was integrated and ready for technical testing. The whole team performed technical testing, trying everything they could think of to break the program or create usability issues in on meeting as I recorded the problems for repair. Those issues, revealed in a later section, were fixed within the week. The team then performed user testing with 'real' users and brought back responses from a good cross-section of their friends from majors both within and outside of the engineering school. Finally, the team cleaned up (formatted/commented) and documented the C++ code they had written and created material for our group presentation to wrap up the semester.

My Development

I began development of FabStudio this January with an empty project in Qt Creator. Last semester, we had prototyped a few shared components that could be reused in this project (such as the slicing and pathing classes, toolscript loader, and graphics framework) but the majority of the code for this project was rewritten from scratch. I began by loading the GUI that Jeff and his team developed last semester in a human-computer interaction class and learned to program some of the novel stylistic elements. Once I was comfortable with implementing the toolbar, docking windows, menus and tabs, I emulated this project's layout in the new version of FabStudio. I decided to do this instead of just converting the existing program, because while its interface was great, the code was not set up properly to have back-end software plugged into it easily. I then imported code from last semester's work on the AppRunner to handle OpenGL rendering and camera controls.

The next task was to implement a dock-window that allowed users to load objects. To make this as user-friendly as possible, I wanted to have most of the interaction done visually and with the mouse rather than through dialogs. To this end, I created a panel that lists objects on the user's computer and would allow them to drag-and-drop the objects to the main window. There were several immediate issues with this.

First, drag-and-drop from a list in Qt is complicated; there's no way I could find to directly access which of the list entries was being dragged from the structure provided in the drop-event callback. To resolve this, I had to query the drop list to see if it was the source of the event (that event information is available) and, if it matches, just return the selected object. The assumption here is that the object would have to have been selected in order to be dragged.

Next, drop events are sometimes passed to the MainWindow class and sometimes to the OpenGLWidget class, depending on what the user has clicked on last to create focus. To solve this issue, I gave OpenGLWidget a reference to the MainWindow, and just had it feed its drag-and-drop callbacks

through to MainWindow whenever it received them. This established a pattern of doing most of the "thinking" in MainWindow through function calls by other objects. Indeed, all panels, the toolbar, etc. make callbacks into MainWindow in order to interact with the objects scene.

Finally, scanning all objects on the user's computer could take a significant and unpredictable amount of time. To prevent the interface locking up while scanning is progress, I moved the scan procedure to a different thread and had it add objects to the list through a mutex-protected LoadObjectsPanel callback. The pattern of scanning in a separate thread and sending a callback when information is discovered was also used later when looking for tool-scripts. A word of caution is in order here: there are subtle situations in which this pattern created deadlocks, so be careful when modifying the scanning code.

Once object loading had been implemented, the next step was to be able to load and render the objects that had been dropped. By implementing the PrintableObject class, I was able to abstract away the loading of files by routing the call through classes developed last semester for the purpose (STLFile, AMFFile) and just store an AMFMesh of triangles internally. Rendering was accomplished just by pushing each triangle in sequence through to OpenGL on each call to an object's glRender.

Now that loading and rendering had been implemented, I merged in all of the math, toolscript, mesh processing and other shared algorithms from last semester. This led directly into adding the object manipulation controls. With the math library in place, it was easy to compute object transforms such as scaling and rotation. I had to write a Matrix4x4 and Quaternion class, however, since once I started combining multiple transformations the math became much easier to do in a single step.

With objects on screen, there were only two steps left before the basic pathway was complete: adding materials, and integrating the tool-script engine.

To add materials, I scanned parameters from local tool-script files and inserted them into the MaterialsPanel as noted earlier. I introduced the idea of a "Material" vs. "Tool" in a tool-script in order to reduce the complexity of this screen. Basic users should only need to know that they are printing out of some kind of Silicone, for example, before hitting the "print" button, while advanced users could tune the exact script being used to print from this material. I also reformatted the script section of the tool-script to improve consistency and give the script finer control over the direction of the print and reporting its progress.

The PrintPanel, implemented next, began as just a button that hooked directly to a hard-coded pather/slicer. Once I had debugged the basic operation and was able to preview the results in 3D, I went ahead with integrating "real" tool-scripts that drive the print. At first, the tool-script just ran and locked up the UI until it was finished. This was unacceptable and gave the impression of the program freezing permanently for longer builds, so I added progress bars and put the printing in a second thread. This has been a constant source of bugs ever since, but I don't see another way around it at this point. The down-side to threading this is twofold: first, inter-thread communication via queued blocking slots is necessary for everything from reporting print progress to the UI to saving the results to the main window. This is a down-side because only objects with basic or Qt types (int/bool/pointer/QList) can be

passed as parameters in this structure without going through some complicated meta-registration process with Qt. This means that we're essentially restricted to using pointers to pass any kind of interesting data, which implies that some sort of policy about memory management is needed. However, since multiple slots can be connected to the same signal (and multiple signals to the same slot) things get a bit fuzzy about who actually "owns" an object's memory. This policy for FabStudio is that the callee is responsible for deallocating objects, and any signal that emits a signal with an allocated object may only be connected to a single slot. The second downside to threading the print job is that there is no way to cleanly abort the thread and deallocate its memory. The best way I have found to do this is to simply erase all external references to the thread, and have the thread delete itself on termination. This does mean, however, that a thread continues to process even after the user has terminated processing (by clicking Start Over, for example). It consumes extra CPU cycles, but I don't see a way around the problem without deadlocking FabStudio's execution.

Once the FAB-file output had been tested for objects using various tool-scripts, the core features were complete. By the beginning of April, I launched into feature and debugging mode. On April 4, I had stubbed out all of the contribution points for the team and integrated these stubs (for arranging the build tray, orienting objects, etc) into the UI and other program points. Jeff suggested that the UI was too complex for basic users, and I agreed, so I added a basic/advanced mode toggle in the "view" menu to hide most of the controls that are not necessary for basic printing. Other fixes included adding check-marks to the toolbar and disabling access to various panels to help guide users through a print job.

After preliminary technical testing with the group on April 12, which identified a variety of bugs and usability issues that I fixed later that week, We tested FabStudio by printing an object using the plastic extruder on the Model 2. Printing a face with the edge-only pather yielded great results, but to match the quality of something produced with the Rep-Rap, we wanted to print a larger version using a double-edged algorithm. I designed and implemented this class (DoubleEdgePather) that afternoon, and luckily discovered a bug with the LoopInXYPlane class's "expand" method. Briefly, I found two incorrect statements that were causing it to execute hundreds of times more operations than should have been necessary. By fixing the issues, the processing time for slicing and pathing was cut dramatically. Additionally, I moved away from using high-tolerance equality in calculations based on a template parameter in favor of double-precision tolerance and non-templated classes that derive types from static global typedefs. This move actually improved the quality of the crosshatch pathing algorithm, has reduced compile time and increased debuggability (Qt's debugger is, unfortunately, very bad at analyzing templated class types at runtime).

Most of my work after this successful print test has been dedicated to managing the team's user-testing and preparing this and other documentation.

TESTING

TECHNICAL TESTING

During the week of April 12-April 19, FabStudio was tested by all team members. At our group meeting, they attempted to break the software in every way they could think of, and were able to cause it to crash or behave unexpectedly in a number of situations. During the following few days, I was able to resolve nearly all of the issues raised (listed below for reference) and I re-submitted the program to them for further testing.

List of issues resolved after technical testing:

- Delete scanned object while scan in progress causes load objects panel to crash
- Click object to get properties panel
- Implement object transformation numerical values
- Implement quaternions to make rotations compose properly
- Rename "position objects panel" to "edit objects panel"
- Put all panels in view menu
- Implement selected object &
- Position objects panel too big
- Send to printer panel is locked big
- Light source in open GL to make object look like... an object
- Auto rescale super small/big objects on load
- Position object after loading
- Gray out load panel when loading an object
- After rescaling an object, snap it to the build tray
- Can drag objects off tray (re-purposed, see next)
- Can't delete object (drag off screen -> poof)
- New Controls:
    - LMB is for object actions
    - RMB + drag - up/down zoom, left/right rotate
    - RMB click - recenter view
- Can flip view upside down with zoom in, limit zoom out
- Print with no object = crash
- Too many left clicks -> mainwindow.cpp:381 crash error
- Controls not always on
- Controls for adjusting object are sticking around too long
- Implement a first-time load screen to display important info for new users:
- Control instructions
- Splash image
- Save basic/advanced, first time load, etc. between sessions
- "zoom to corner" bug
- Remove material from object / select material that is not compatible without having to delete objects

- Remove slicer direction from options in tool settings
- Put first layer at z = slice_height

On April 18, FabStudio was used to print two face models on the Fab@Home Model 2 using the new plastic extrusion tool. The first used the edge-only pather; the second, about used the double-edge pather. Both printed successfully without requiring modification to the .FAB file.

User Testing

User testing with non-team members occurred April 19-April 26. On April 19, the team created three tutorial videos:

- Basic Tutorial - Walks the user through the very fewest number of steps required to create a print job
- Intermediate Tutorial - Introduces more advanced features of FabStudio, including
- Advanced Tutorial - Describes FabStudio's complete functionality, touching on technical topics such as examining the tool-script contents, adjusting printer & pather settings, and saving results to an intermediate FAB file for viewing.

The following week, each team member was responsible for finding two people to use the software after watching one or more of the videos. These were given a task according to the video they watched:

- Those watching the Basic Tutorial were asked to print an object from ABS Plastic.
- A user who saw the Intermediate Tutorial was asked to print more than one object from different materials
- Advanced tutorial watchers were given the task of printing an object using the double-edge pather

As each tester watched the video and attempted the task, the team member would record their actions (especially mistakes or unusual actions), whether they had to give the user help or answer questions and any feedback from the user.

The responses gathered were largely positive, and the program performed well from a technical perspective. Most issues were with inconsistencies in, or the unintuitive behavior of, the user-interface.

Test Admin: Karina

Tutorial Video: Basic, Intermediate

- Didn't like moving with green/red arrows; would rather click on object and drag it
- Attempted to use scroll-wheel several times for zooming, but it was nonfunctional
- Got stuck in the tool-script editing panel, since on a MacBook there is no close button

Test Admin: Jason

Tutorial Video: Intermediate, Advanced

- Wanted to be able to adjust camera elevation (not just pan/zoom) but this feature is not available
- Would rather click on object to drag it rather than using handles
- Was bothered by the inconsistency of the "Edit Objects" panel automatically displaying itself
- Can't resize the startup splash screen, and the instructions are small / hard to read
- User was confused by objects not loading; attempted several times to load an invalid object file since there is no feedback about the operation failing

TEST ADMIN: JEREMY

Tutorial Video: Intermediate

- Engineer completed task successfully
- Suggested that instead of dragging material to object, double-clicking a material could add it to the currently selected object
- Was confused by the fact that material colors are not consistent; if several materials are assigned, then removed, then assigned again in a different order, they don't return with the same color as before

TEST ADMIN: CHRIS

Tutorial Video: Basic, Intermediate

- Non-enginer was confused at how to right-click on a MacBook. If possible, minimize the use of right-click to help alleviate this problem.
- Engineer user with 3d modeling experience said he really liked the layout
- Both were able to complete tasks successfully

TEST ADMIN: JIMMY

Tutorial Video: Basic

- Got lost because the very first model they loaded was bigger than the screen, which made the handles impossible to grasp for resizing
- Default window size when starting FabStudio is too small to do anything useful; suggests full-screen by default.
- A better indicator of the loading time for an object would be useful; the user throught the program was freezing on a complicated object
- When lost, was looking for some kind of 'reset' button that would clear everything and start over
- Suggested automatically rescaling objects for a good size for printing when loaded
- Make objects always 'stick' to tray, even in advanced mode

- Create indicators for why someone can't access "print scene" or "assign materials" instead of just disabling them; the user was trying to find an explanation for why they were unable to print by clicking on the disabled button

Test Admin: Nathan

Tutorial Video: Advanced

- User was confused by the fact that Z=0 does not put the object on the tray
- The tool-settings dialog where one selects the pather does not have a close button; breaks the standard of the "X" in a window not saving changes in a dialog (it always saves changes).
- Auto-scaling is broken for small objects; it supersizes them instead of scaling them up to a reasonable size
- User accidentally backspaced and deleted their object, then went looking for an "undo" button that didn't exist.

Results

The FabStudio project at revision 386 (April 26) is 10,000 lines of code. The shared library contributes another 13,000 for a grand total of 23,000 lines. The code-to-comments ratio is approximately 8:1.

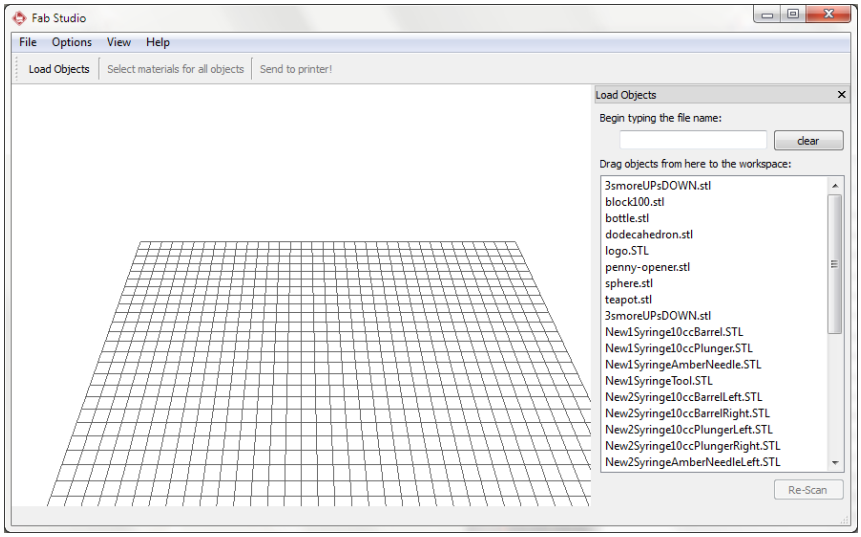
Tested Features

Complete printing process

As demonstrated below, the full basic printing process has been implemented in FabStudio.

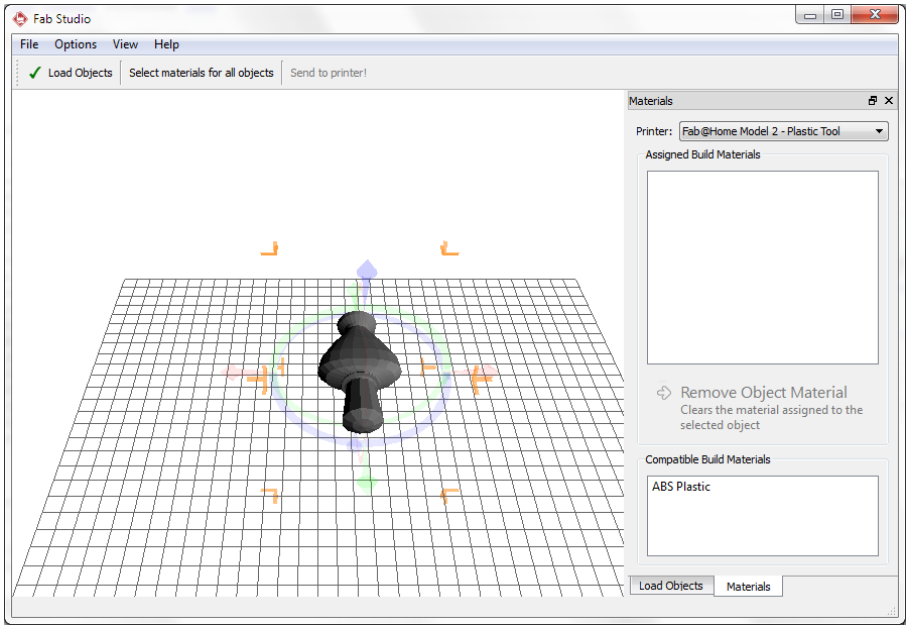
Launch FabStudio

When FabStudio starts up, the user notices three main interaction regions: the docking panels on the right-hand side, the 3D preview window, and the toolbar/menubar in the top-left. From this point, all actions that are not yet usable will be grayed out (such as selecting material or sending to printer) to help guide the user through the printing process.



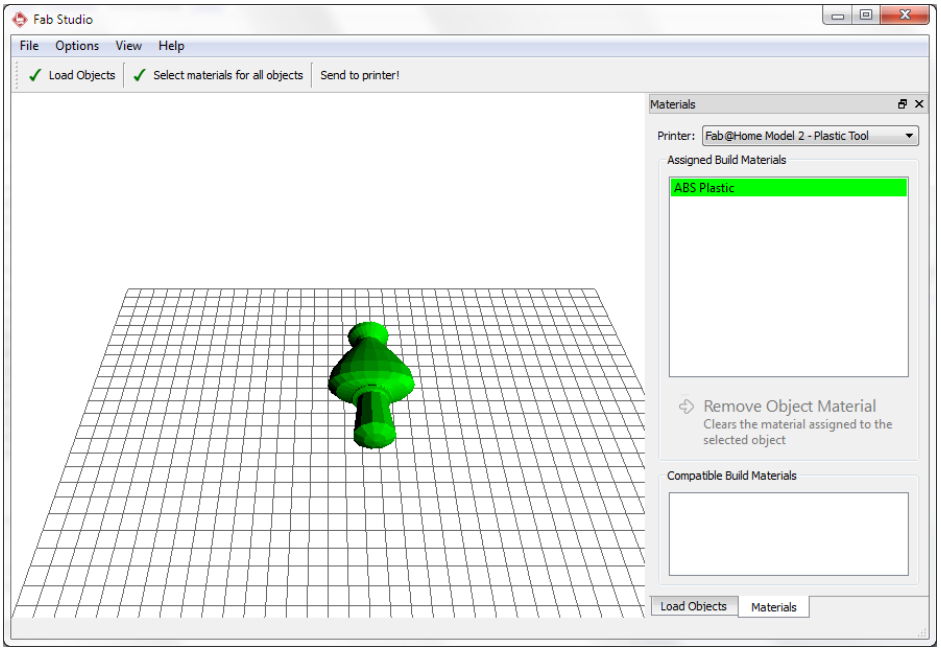
Drag an object into the scene

To begin printing, the user drags a model from the Load Objects panel on to the build tray. This automatically causes the Materials panel to be displayed. Also, the Load Objects toolbar item will get a check-mark next to it to indicate that the given stage has been completed. When the object is loaded, it is automatically rotated to its optimal printing orientation.



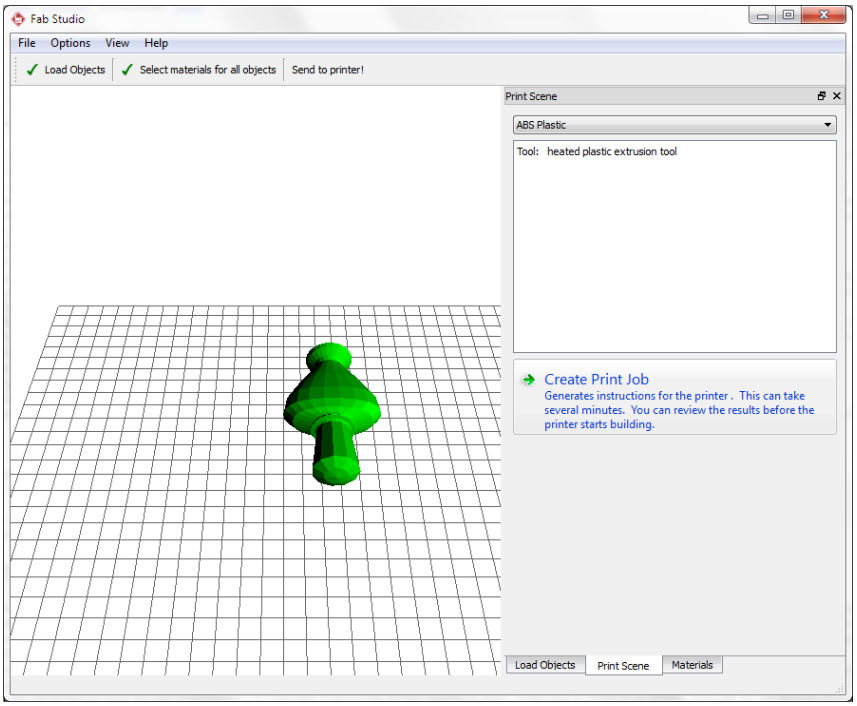
Assign an Object Material

Similarly to how an object is loaded, each object is assigned a material by clicking and dragging the name of a material from the right-hand panel to the object itself. In this case, we have chosen to print our bottle out of ABS plastic.



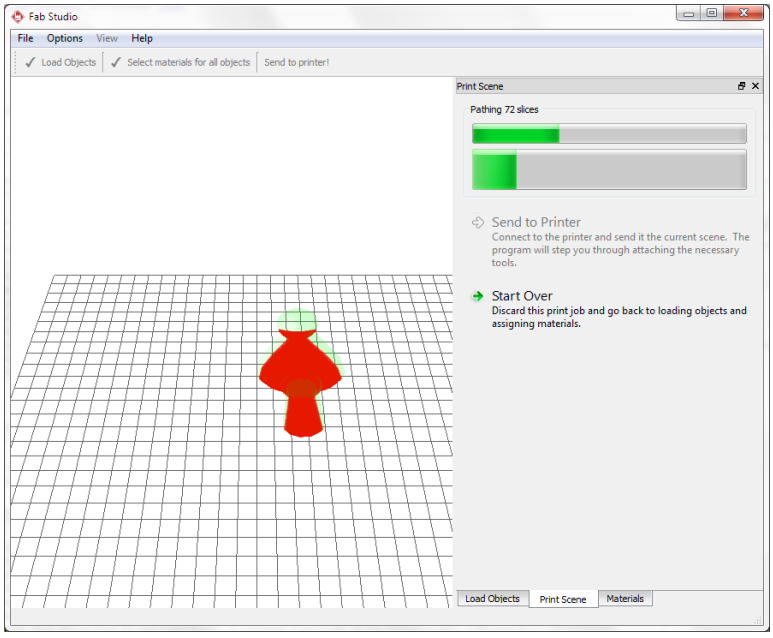
Click the "Send to printer!" button

Once the materials have been assigned to all objects, the select material toolbar item gets a green check-mark and the button to print the scene is enabled. The user need only click "Create Print Job" to begin the process.



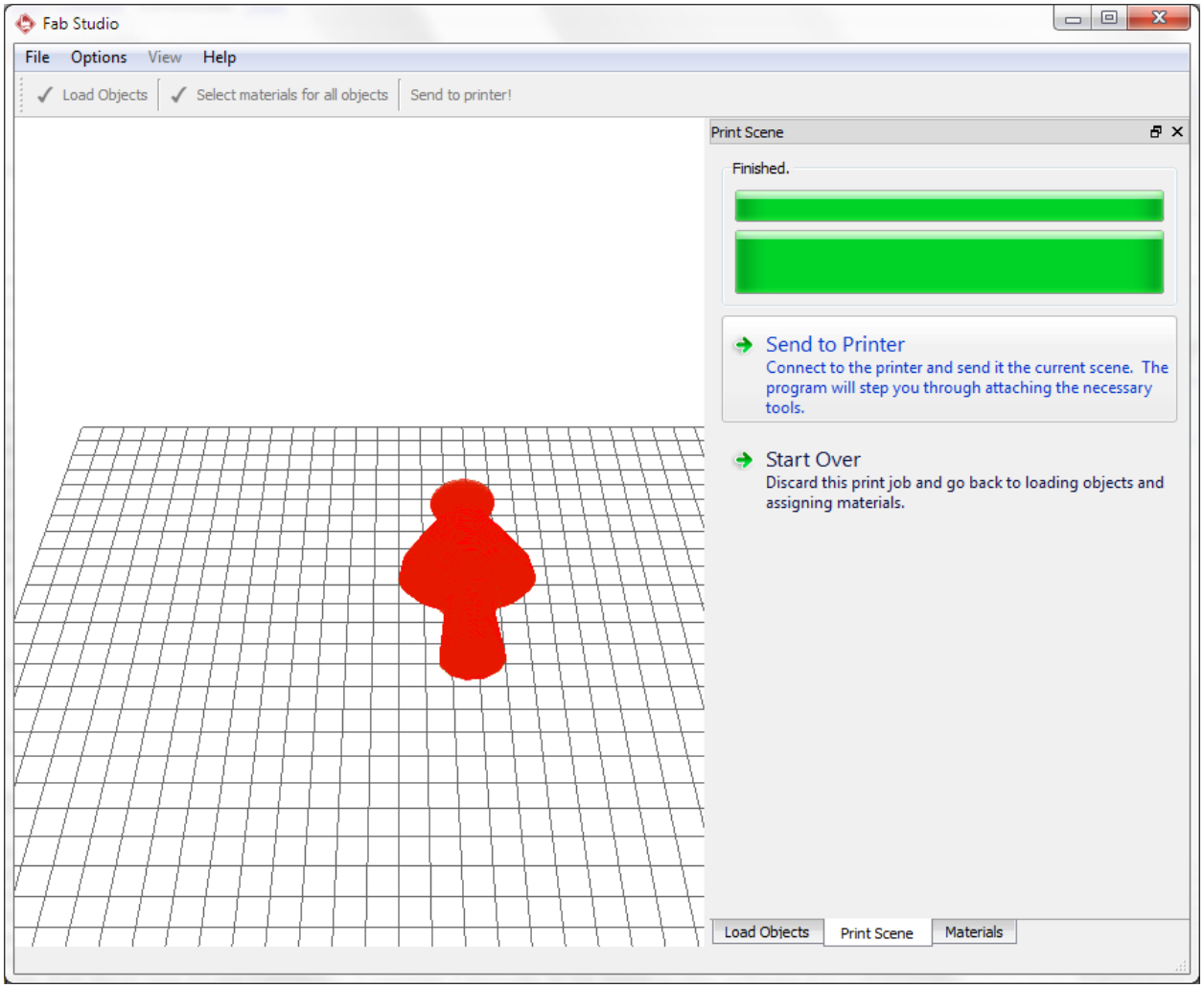
FabStudio runs the Tool Script

This step requires no user interaction. FabStudio will process the mesh automatically, and display the results as they are computed.



THE MODEL IS READY TO BE PRINTED

By clicking "Send to Printer", FabStudio will launch FabInterpreter with the given model file and printer settings pre-loaded.

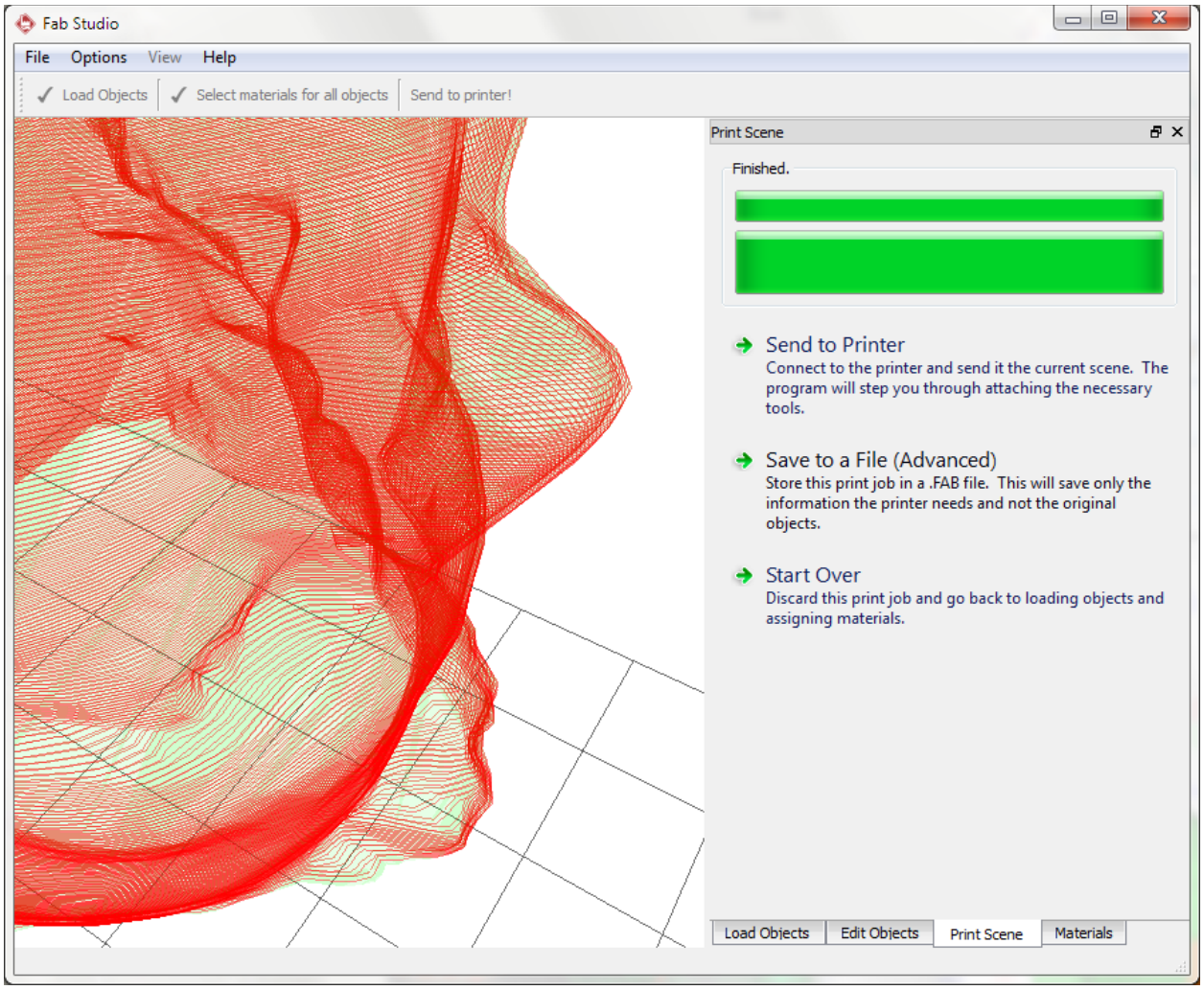


OBJECT FILE FORMATS

FabStudio currently supports ASCII and Binary STL files, as well as Additive Material Format (AMF) files. Creating meshes directly from object files, Nathan and Jason's work, is implemented as a separate program not part of the main FabStudio application.

Pather Implementations

Pathers define how a mesh's boundaries are filled. FabStudio supports three current pathers: the Crosshatch Pather, Double-Edge Pather and the Edge-Only pather. The Crosshatch pather traces boundaries of each layer, then raster-fills the inner space. The Double-Edge pather traces a doubled loop on the boundary of the object to make thicker walls, whereas the Edge-Only pather simply traces a single time along a layer's edge.



Automatic Support Material Generation

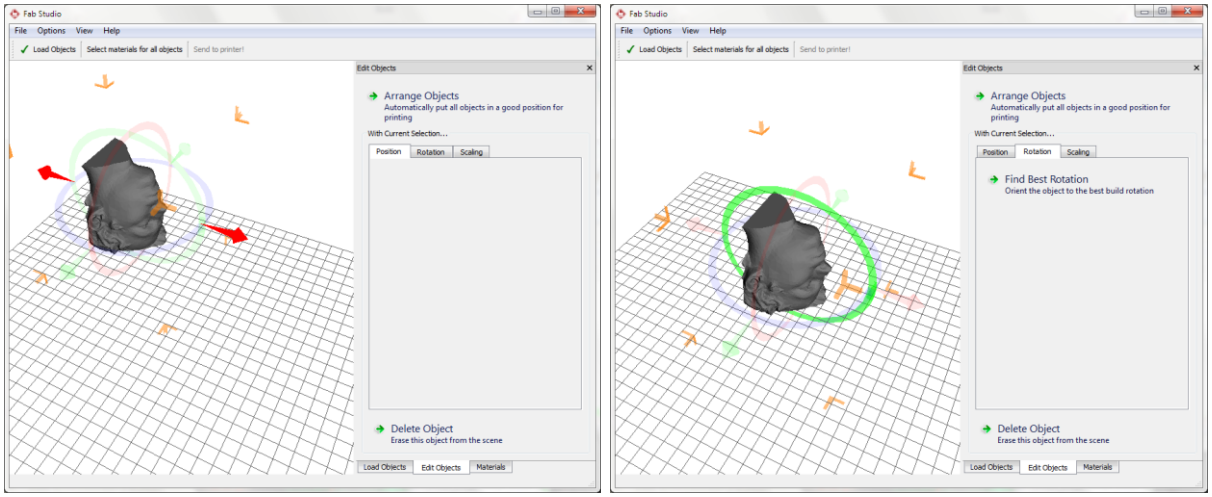
For multimaterial tool-scripts in which one material is designated as being support material, FabStudio will automatically create a surrounding mesh from the support material.

Build-Tray Organizer

For prints with many objects, it can be a hassle to arrange all of the footprints such that they do not interfere with one another. This operation can be done automatically with a single button-press in FabStudio.

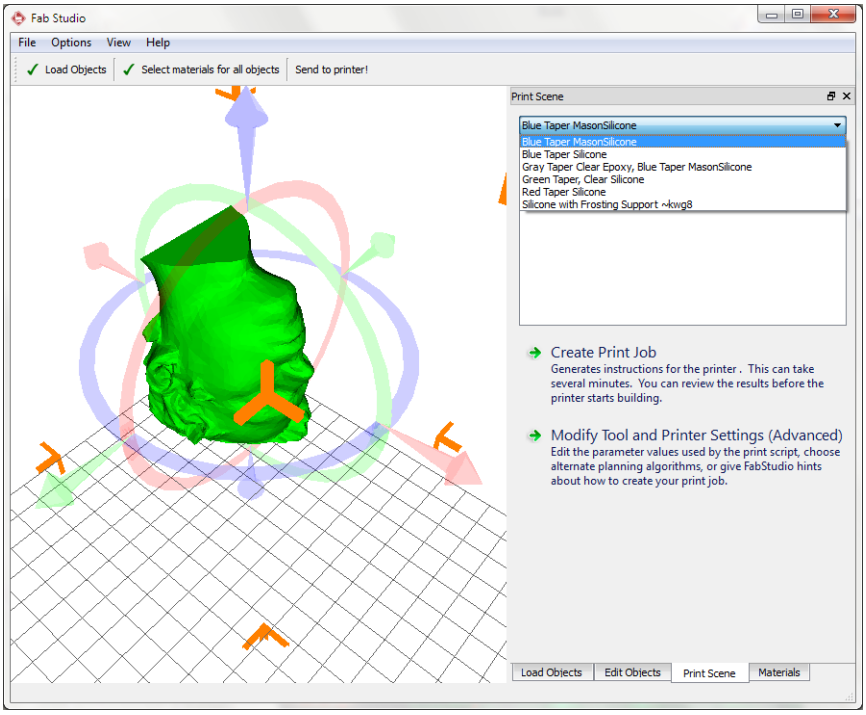
Visual Object Manipulation Controls

Rather than requiring users to change object locations, rotations and scaling values by entering numbers into a dialog, FabStudio provides visual controls that allow the user to grab and move, turn or resize objects dynamically.



Material-Based (rather than tool-based) Design

FabStudio decouples the ideas of material and tool. Rather than having to pick a specific tool-material combination with which to print a mesh, FabStudio allows the user to simply assign a generic material name first, then pick the exact tool that will be used during the print-job preparation as a second step. This gives the precise control needed for advanced users, while allowing basic users to ignore this added complexity.



Conclusions

FabStudio, having been tested for functionality, stability and usability, is now alpha-quality software. This semester's development has been successful in creating both a prototype FabStudio that meets the Fab@Home project's needs both for a new environment in which models can be processed for printing and for a platform on which novel 3D-printing enabled applications can be developed. The documentation and tutorials generated are sufficient to allow those continuing work on this project to train themselves in its design and development. Finally, the newly-recruited Fab@Home programmers have been introduced to the process of team software engineering and possess the fundamental skills necessary to perform maintainance on FabStudio. Those that are motivated should be capable of assuming a leadership role in the project within one year.

Future Work

To allow for this project to be wrapped up cleanly and for this report to be written, FabStudio underwent a progressive pausing of development during the last two weeks in April. Feature lock was imposed April 19, so only features implemented before that date were completed and tested. Those that were not are listed below. Next, code-freeze occurred on April 25 after initial technical testing. The second round of user-testing results was only gathered after that date so they have not been incorporated.

Below is a list of issues identified through user testing and our technical presentations that need to be addressed by future developers. Someone without experience in this code-base but possessing the supplementary advanced technical knowledge outlined in the background section should have no problem working through the issues in roughly the order presented here--usability, then technical, then feature requests. This is the general order of increasing difficulty and requisite familiarity with the software. Usability issues are the least technically challenging, but working on them will let the developer learn to navigate FabStudio's project files and code. Resolving technical issues usually involves writing more code and a deeper understanding of the software than fixing usability issues. After a developer is very familiar with the program's structure, they can begin attempting to add new features. However, keep in mind that this is only a general progression; some very simple technical issues have been left unfixed intentionally to ease the transition for future developers, and some usability issues will be rather difficult to remedy.

I have rated each issue on a 1-to-10 scale of relative difficulty, where 1 is the easiest and 10 is the most challenging among those tasks listed here. Each is also accompanied by some information that will be useful in resolving the problem.

Unresolved Usability Issues

1 - Ability to have newlines within description of tool

This should already be solved, the only thing needed here is to confirm that it can, in fact, be done. To add a newline in a tool's description, use the HTML «br/> tag; the Qt 'auto' text box in the toolscript

settings dialog should recognize this as an instruction to insert a newline. If it does not, simply replace all strings in the toolscript's description matching the regex (already excaped) "\\< *br *\\/\\>" with the end-of-line character '\n' when the script is loaded from disk.

1 - Add "drag support material" label to instruct user, similar to "drag object" label

Open the assign materials panel's ".ui" (User Interface) file, click on the group box that surrounds the materials panels, and change the text to include some instruction to drag the labels. Alternatively, add a text label as the top element within each group box and set that text label's displayed string to the instruction text.

1 - Does the camera rotate the wrong way when dragged? (add as an option in the future)

To add this as a checkable menu item under the "options" menu, one might want to add a new menu item to the main menu. To do this, open the mainwindow.ui file, click on the 'options' menu item, and type a new menu entry for "Reverse Camera Rotation". Then, assign the 'checkable' attribute in the panel near the bottom. Finally, right-click on the action associated with this menu item, and click "Go to slot...". Then, in the function that is created, add code to call a function on the MainWindow class's OpenGLWidget member (opengl_widget_) to toggle the value of an internal boolean variable in that class. You will have to write this function. Finally, query the boolean when handling the camera rotation in OpenGLWidget::mouseMoveEvent and multiply the value of the mouse's x-delta by +1 or -1 depending on whether the flag is set or not.

2 - Keyboard controls aren't subject to constraints on camera zoom or location, so the user can get lost in space

Keyboard controls pass their values through to the OpenGLWidget from MainWindow by calling functions. Install limiters on these functions to bounds-check the values of the various parameters. If a value goes beyond the bounds, just assign it to the nearest in-bounds value.

2 - Fake a mouse move when the user's object loads so that the controls show up if the mouse is already over the object when it loads

In the MainWindow's method which is called when a LoadObjectsFileThread completes the loading process (named addObject/loadedObject or something similar) install a call to `opengl_widget_->mouseMoveEvent`. You will need to provide a parameter to this class, which can be generated by polling properties of the global QCursor object for things like the cursor's state and current location.

2 - No exit button on printer & toolscript settings dialog

Open up the settings dialog's ".ui" file. Drag a button on to the bottom of the main dialog. Assign its object name and text. Right-click the object, "Go to slot..." and pick clicked(). In this function, enter "this->close()" or just "close()" to close the dialog window.

To create a pair of buttons that allow the user to either close the dialog or save the changes, add a ButtonBox and select the appropriate buttons. Then, each button will need to both have the close() call

within it, but preceded by a statement that sets an internal flag of whether to save the settings or not. Then, back in the PrintPanel's slot for pressing the edit printer/toolscript settings button (in which this dialog is activated) read out that flag from the object and check it before assigning the values from the dialog back into the local printer/toolscript settings.

3 - Before printing, warn if geometry is below the Z=0 plane or above the maximum build height

When the user presses the "Create Print Job" button, first call a new method on MainWindow with signature "bool checkIfObjectsAreBelowTray()". In this method, iterate over all PrintableObject classes in the printable_objects_ member. For each object, compute the bounding box (this will probably be a new function in PrintableObject). If the bottom of the bounding box is below z=0, return 'true' immediately; otherwise, at the end of the function, return false. I suggest computing the bounding box instead of just checking to see if any Z coordinate is below zero because that is a superset of the functionality needed by this function, and is probably used elsewhere in the program (so writing this in once place may let you eliminate code elsewhere).

3 - Can exit out of everything including toolbar, which makes it difficult to get any panels back

There don't appear to be settings in Qt for designating that a widget can't be closed. The only way to prevent might then to be to connect a slot to the signal that is emitted by the toolbar when it is closed that simply re-shows the toolbar; that way, anyone trying to close it will be unable to do so.

4 - Unless Z of an object is explicitly modified, always snap back to tray

This is an improvement that will resolve a number of usability issues regarding the Z location of an object. In PrintableObject, add a boolean flag and get/set methods that designates whether the object should be snapped to the Z=0 plane (something like "bool auto_snap_to_tray_;"). Add a new method "void checkAutoSnapToTray();". In this method, return if auto_snap_to_tray_ is false; otherwise, invoke the snapToXYPlane method. Next, in each method that adjusts the object's rotation or scaling values, call checkAutoSnapToTray(). In the constructor, set auto_snap_to_tray_ to be 'true'. In any method that explicitly modifies the object's Z coordinate (probably just the Z-axis handle manipulation routine) set auto_snap_to_tray_ to be 'false'. This should be all that is needed.

It is more complicated to allow the user to modify this value explicitly, and could be left as a "todo" with little consequence. However, to implement this functionality, add a checkbox to the Edit Objects panel's Position tab. See other tabs' code for examples of how to hook the checkbox's value to that of the PrintableObject.

5 - Sometimes can't see x/y rotation axes because they are infinitely thin

Two changes will need to be made: adjusting the handle's rendering code, and changing the handle-intersection code. First, change the handle rendering code to, instead of rendering a flat circle, render a loop whose cross-section is an equilateral triangle. This gives the handle size when viewed from any angle.

Next, in the handle-intersection routine of the PrintableObject, each incoming ray will need to be tested against this new torus rather than against a circle on a plane. Look up the algorithm for testing ray intersection with a torus and replace the current code with this new algorithm. If you're feeling particularly industrious, you might even want to create a new class in the /shared/math/ library called "Torus" with the appropriate attributes, and implement this algorithm there so others can benefit from your work.

5 - Remove "swing" delay on camera rotation/zoom; keep on translation

There is an exponential approach algorithm implemented in the OpenGLWidget class to make the camera smoothly translate to its new coordinates when moving. This approximation also was implemented for zooming and rotating, but it makes the controls feel more sluggish and imprecise than they should. Remove the code for the exponential approach on the camera angle and zoom, but leave it intact for the translation. Though conceptually simple, this will be a challenge as they are very interlocked at the moment and may require completely rewriting this algorithm.

6 - Group objects in load objects panel in tree, by last two directories in path

This will allow the user to more quickly identify the objects they want to use by their location, rather than having to rely on hovering the mouse and reading off different path strings. This is challenging because it requires replacing the list widget with a tree widget, and developing a way of filtering the list to respond to the text the user enters in the typeahead field.

Unlike previous issues, if I were implementing this functionality I would not be sure exactly what to do--mostly, I would implement it by experimenting and figuring out what worked. Generally, though, I would suggest you begin by deleting the list widget, creating a tree widget with the same name, and recompiling just to see what happens. I believe that some of the add*() methods are different between lists and trees and take different parameters. For guidance, see how the tree view control is managed in the printer/toolscript settings panel.

Unresolved Technical Issues

1 - Program halts after launching FabInterpreter until FabInterpreter is closed; this causes Windows to think it has crashed (it is marked as "not responding")

Simply take out the call to .waitForClosed in the PrintPanel's send to printer method.

2 - If FabInterpreter is not found and has to be selected by the user, it is not immediately launched after being selected; the user has to click "Send to Printer" a second time

This should be easy to resolve; one of three things could be happening: the string for executing the program is not being updated properly, the do...while loop is terminating prematurely, or the function is returning somewhere it should not be.

3 - The dimensions of the printer are scattered in several places (at least: the delete object when dragging routine, the grid rendering, when calling

"ARRANGE OBJECTS"). THESE VALUES SHOULD BE CENTRALIZED AND LOADED FROM A PRINTER CONFIGURATION FILE SO THE ACTUAL BUILD-TRAY SIZE IS USED.

Everywhere these parameters are used, instead query MainWindow for the parameter values. This will centralize the printer's dimensions, so they can later be read from a configuration file.

4 - SUPPORT FOR GLDRAWLISTS IS INCONSISTENT AND MAY CAUSE CRASHING BUGS; IT MIGHT BE GOOD TO HAVE A WAY TO DISABLE THE DRAW-LIST ACCELERATION WITHOUT RECOMPILING THE PROGRAM

A GLDrawList "memorizes" many OpenGL function calls and allows them to be executed much faster from memory as a single batch. It is a coarse optimization that FabStudio uses to speed up rendering of PrintableObject and the paths during printing. However, there seems to be some sort of limit on the amount, type or configuration of data one can store in them on different machines. For example, on a MacBook, per-vertex colors cannot be put into a draw list. On my ThinkPad laptop, I can store 3-dimensional colors (glColor3f) but not 4-dimensional ones with a non-1 alpha parameter (glColor4f). Since these are just an optimization, and on some computers causes FabStudio to crash entirely, it would be ideal to be able to turn off this functionality. This could be done by adding a new entry for enabling/disabling GL draw lists to the QSettings for the application. If the app ever crashes (monitor this with another QSetting that only gets reset if the app closes normally), disable draw lists on the next startup. Each place draw-lists are used, read the flag and only compile or execute them if the optimization flag is set.

4 - Y/Z RESCALING SPIN BOXES ON EDIT OBJECTS PANEL'S SCALING TAB DON'T FUNCTION

I believe this is a problem related to the "uniform scaling" button not re-enabling the Y/Z spin boxes.

5 - CLOSING ALL PANELS DOESN'T REDRAW THE SCREEN

It is going to be difficult to resolve this issue; there are supposed to be signals emitted when the panels are closed, but they do not seem to occur at the correct time to allow the drawable area to be resized. This has been a problem for some time, and I haven't found a way to fix it.

6 - "HEART.STL" DOESN'T PATH CORRECTLY

This particular STL file appears to be broken. While some broken STLs path correctly, this one does not. The solution is to implement a "welding" procedure in the STLFile::convertToAMF method that merges vertices from the original STL mesh to form a 'solid' AMF instead of keeping them as separate triangles.

7 - FREEZES WHILE LOADING LARGE BINARY STLS

The changes to fix this problem will be entirely in the STLFile::readBinary method. The performance issues here are due to several factors. First, reads are performed sequentially but in very short amounts, directly from disk. Instead, read the entire file into memory from disk first, then parse it into an STL structure from the in-memory object. An efficient way to do this would be to find the file's size, allocate a new buffer, read into the buffer, close the file, read from the buffer, then deallocate the buffer. The second source of performance issues is likely to be adding triangles to the list one-by-one,

since this causes lots of allocation. If the STL triangles were stored in a list instead of a vector, this would likely be less of an issue.

8 - Deleting an object while scanning for objects is in progress causes the load objects panel to malfunction

The QFileWatcher class monitors the top K folders with the greatest number of objects found by the scan for object files thread, where K is some constant defined in that thread implementation file. When the folder changes, it notifies the load object panel that it has changed so that the panel can update the folder's contents to match what is on disk. Unfortunately, if this occurs while the scanning thread is still in progress, the procedure that updates the conents of the list starts conflicting with the one that is scanning for folders. This is likely a synchronization issue. It has been temporarily resolved by just ignoring folder changes while scanning is still in progress.

Unimplemented Features

1 / ??? - Allow multiple tool-scripts to be used in the same scene

A scaling issue with the way tool-scripts are currently implemented was identified during FabStudio's final presentation. Since a script has to be defined per tool combination, per material, there will be an incredibly large number of scripts required once Fab@Home has more than a few materials available for use.

There is an immediate work-around for this problem: create a single tool-script that contains a <tool> entry for every tool usable on the Fab@Home. FabStudio will then discover that it can use those tools in any combination. During the slicing/pathing stages below, one would then need to slice/path meshes from every material (materials with no meshes would be ignored), and identify which of the tools had actually been used and only add their configuration information to the output FAB file. This can be done by only adding configuration information if the "materialName.meshes.length > 0" variable for each material "materialName".

A true solution to this problem that allows multiple tool-scripts to be used on a single scene would be extremely difficult to implement; additionally, I'm not sure that it even makes sense to do so. The point of a tool-script is to completely encapsulate the translation process. Mixing two translation processes, even if they were compatible, would be hard to automate. Doing so would require code analysis, verification, then rewriting before it could be executed. I believe the best solution is the work-around.

3 - Save main window state between executions

When the program ends, but before the main window is actually destroyed, read its window state and size parameters. Save these values into keys stored in QSettings. When the application starts, read these keys (or default values) and set the state of the main window.

4 - Save folder locations where models were found between executions to increase loading speed

Again, use QSettings. Every folder where a model is found, add it to a list in QSettings during the scan thread's process. Then, when the scan thread starts, prime the worklist by adding all of these folders to

the list. The trickiest part will be making sure these folders are not later re-scanned. To do this, add each folder to a set, and before scanning a folder, check to make sure it is not in the set.

4 - To improve loading speed, one could re-use files that are loaded into memory by PrintableObject instead of loading a file multiple times.

Save a PrintableObject's source file name when the file is loaded. Before loading any file, check all of the existing PrintableObject instances to see if they are that same file. If they are, invoke a "clone" method that returns a copy of the object instead, then position and scale it appropriately. The trick here is that both clones should use the exact same source AMF object (don't copy the AMF, if possible) to conserve memory and significantly improve speed. This is difficult because you will need to use either a QSharedPointer or careful reference monitoring in order to determine when the object is no longer needed and its memory can be freed.

8 - Enable editing of tool-script and parameter values from printer/toolscript editing dialog

Implementing this functionality requires several changes to the structure of FabStudio. First, the ToolScript and ToolScriptTool classes will need functions that allow their member values to be written as well as read. Second, the labels of the tree-widget in the dialog need to be made editable. When edits are made, they need to be saved to a list of things to update that is passed back to the PrintPanel so it the PrintPanel can apply them. Note that the changes cannot be made immediately to the ToolScript/ToolScriptTool object because the program needs to be able to undo changes that the user makes. Additionally, it would be useful if the script were validated somehow before it was saved; otherwise, even an advanced user might make a typo in the script and have a tough time trying to figure out what they are doing wrong. This will require creating a new method in the ToolScript that is essentially a stripped-down "execute" method that doesn't perform any actual work.

10 - The support material algorithm Professor Lipson requested

Professor Lipson would like a support material generator that does not waste as much extra material as this one. Specifically, he would like it to fit tighter with the object itself and not add material where there is no part of the original object to support. His request reads:

Please change the algorithm so that instead of using a fixed bounding box, you use the bounding box of the individual layer geometry AND the bounding box of the previous layer. I think that's a relatively simple modification that would be much more efficient in terms of material use.

If you have time, replace the bounding box with a convex hull and it will be even more efficient.

Since slices are only generated after support material is created, one can't actually implement this functionality as stated; however, his desire to reduce the amount of support material should be met by the following algorithm. It will need to be generalized to multiple objects in a complete implementation:

- Invoke the Slicer on an object for some reasonable slice height, say 5 mm
- For each slice, top to bottom:
- While the current slice's expanded outer boundary does not contain all outer boundaries of those slices above it, increase the boundary's expansion size
- Build a triangle mesh (AMFMesh) with inward-facing normals from the outer boundaries
- Duplicate the original mesh and reverse its normals
- Combine the two meshes into the output support-material mesh

The most difficult part of this algorithm will be implementing the mesh generation from the outer boundaries, since which vertices are used to form triangles will be especially important. One downside might be that protruding features with a finer resolution than the chosen slice-height may not be caught inside the support material algorithm. However, one could always increase the chosen resolution at the expense of creating more triangles (and hence increasing slicing/pathing time during print job creation).

Miscellaneous

1 - Rename all "FAHFloat, FAHVector3" etc to just "Math::Float" and "Math::Vector3" then remove #include "shared/fabathome-constants.h" and replace it with the #include for exactly which header file is required

This should be as simple as a find->replace all for each of the following entries:

FAHFloat -> Math::Float

FAHVector3 -> Math::Vector3

FAHLoopInXYPlane -> Math::LoopInXYPlane

FAHSphere -> Math::Sphere

FAHLine -> Math::Line

FAHTriangle -> Math::Triangle

FAHQuaternion -> Math::Quaternion

3 - Create a deployable version of FabStudio for Mac OSX

FabStudio must be rebuilt on a Mac, and the developer should follow these instructions: http://doc.trolltech.com/4.3/deployment-mac.html. This is necessary so that end-users don't have to have Qt installed in order to run FabStudio.

todo()

There is a "todo" function that creates a report in log.html when the program closes. This function is called wherever there is something left undone in the program that is not critical to resolve immediately, but could improve the performance, consistency or stability of the program. As of the last execution, this report contained the following entries:

```
..\..\..\shared\math\loopinxyplane.cpp(506):   "correctly implement the algorithm for picking valid loops" - kwg8 (hit 1586 times - 8ms)

..\..\..\shared\slicer\slicer.cpp(182):   "make this intersection test not check every single triangle in the source mesh" - kwg8 (hit 617 times - 873ms)

..\..\..\shared\pather\simplecrosshatchpather.cpp(296):  "Sanitize the paths to remove paths that double-back on themselves" - kwg8 (hit 427 times - 6ms)

mainwindow.cpp(708):  "Find items in the View menu less hackishly" - kwg8 (hit 16 times - 0ms)

materialspanel.cpp(126):  "optimize updateAssignedMaterials by not calling updateUI if the set of materials is the same as the current one" - kwg8 (hit 13 times - 16ms)

..\..\..\shared\toolscript\toolscript.cpp(107):    "check these names for validity; else, toolscript is not valid" - kwg8 (hit 11 times - 0ms)

mainwindow.cpp(110):  "add the ability to detect when a dock widget is closed and update the screen!" - kwg8 (hit 4 times - 0ms)

modifytoolandprintersettingsdialog.cpp(122):  "set toolscript's preferred pather" - kwg8 (hit 3 times - 0ms)

modifytoolandprintersettingsdialog.cpp(121):  "set toolscript's preferred slicer" - kwg8 (hit 3 times - 0ms)

..\..\..\shared\amf\amfmesh.cpp(230):  "clear out destination mesh" - kwg8 (hit 2 times - 7ms)

loadobjectfilethread.cpp(48):   "check the object's size based on attributes of printer, not random constants" - kwg8 (hit 2 times - 1731ms)

printableobject.cpp(99):  "instead of basing this purely on extension, we could attempt some sort of file analysis" - kwg8 (hit 2 times - 287ms)

generatesupportmaterial.cpp(40):  "implement an algorithm in the requiresSupportMaterial method" - jlz27 (hit 1 time - 0ms)

..\..\..\shared\stl\stlfile.cpp(586):  "This method doesn't actually weld vertices to create a solid mesh" - kwg8 (hit 1 time - 0ms)

loadobjectspanel.cpp(69):  "enable the 'browse' button in the LoadObjectsPanel" - kwg8 (hit 1 time - 0ms)
```

Appendix & Supplemental Material

The following files are included as part of this report package.

The complete Qt source code from FabStudio, revision 386

The complete Visual Studio source code from a compatible version of FabInterpreter, revision 22

Tool-Scripts for various materials, including the plastic extrusion tool

Basic/Intermediate/Advanced tutorial videos

These tutorials were recorded at a group meeting and employed during user-testing. They have also been uploaded to YouTube.

- /Documentation/FabStudio - Introductory Tutorial - Fab@Home.avi
- /Documentation/FabStudio - Intermediate Tutorial - Fab@Home.avi

DEMONSTRATION VIDEOS

Three videos showing three increasingly complex sets of Fab@Home features are provided. These videos were displayed during the group meeting presentation. They are also available on YouTube.

- /Documentation/Video 1 - Basic Printing Features.avi
- /Documentation/Video 2 - Multiple Objects, Multiple Materials.avi
- /Documentation/Video 3 - Changing the Pather, Multiple Toolscripts, Support Material.avi

Doxygen-generated software documentation

See the file "/Documentation/html/index.html" to browse the contents of this documentation.

TOOLSCRIPT DOCUMENTATION

A Tool-Script is an XML document that describes to FabStudio the sequence of operations and parameter values that transform a generic 3D scene descrpition into instructions that can be used to build that scene. These instructions are stored in a FAB file, which is in turn used by FabInterpreter to drive the print job on the Fab@Home.

Tool-Scripts are designed for flexibility. Each document allow a designer to fully specify any number of tools and material settings, and specify a sequence of processing instructions in a 3D-printing-aware programming language. The specification of a Tool-Script is intended to be universal enough to allow them to be used to generate print jobs for other printing platforms besides the Fab@Home.

Under the root «toolscript> node, there are two major sections of content: settings and instructions.

Each setting is defined in the following way:

<settingName text="Displayed Name" units="value units">value</settingName>

- settingName is a string with no spaces or non-alphanumeric characters that defines the setting's variable name
- The text attribute defines what to label this setting when it is displayed in the printer settings dialog
- The units attribute specifies w hat to put after the value of the setting; e.g. "mm" or "seconds"
- value declares the default value of the setting.

Settings are either per-tool or global. Global settings are constant for the entire build process, and can be used to define things that are properties of the whole printer. These settings appear as variables in the script with a name equal to the name of the tag under the <settings> ...</settings> pair.

There can be any number of <tool>...</tool> tags that define tool-specific settings. Each tool declares one material/print-head combination that can be used in this script. The tool tag has the following inline attributes:

- `name="Tool Name"` - How to refer to the tool in FabStudio
- `material="Material"` - What kind of material this tool prints out of; it should be generic, so the user can pick "Silicone" during the material assignment stage, for example, then select a specific tool-script to get a certain calibration of silicone
- `scriptVariable="variable"` - Defines how FabStudio should name the variable that refers to this tool when it is executing the script. The value must be a valid identifier in QtScript (best practice: just use an alphabetic name).

Settings are also defined beneath a tool tag in a `<settings>…</settings>` block. There can be any number of settings in this area, defined exactly as they were for the global settings block. The main difference, however, is in how these settings appear in the script. Each tool has a scriptVariable property that defines an object in the script. Settings for that tool are accessed through the object as properties of that object. For example, a tool whose scriptVariable is "plastic" with a setting tag "sliceHeight" would be exposed to the script as "plastic.sliceHeight".

The script commands are contained in CDATA section between the `<script>…</script>` tags. This is interpreted by FabStudio as a set of QtScript commands. Complete documentation for this language's syntax can be found online, but it suffices to know that:

- The syntax is similar to C++ and Java
- Each line terminates with ;
- Blocks are defined by braces { ... }
- Variables and functions are referenced like in Java and C++ objects using the "." operator
- There is one variable-type, "var" that can hold any kind of value; e.g. "`var x = 5;`"
- To declare structures, one simply creates a 'var' with properties as follows:

```
var thisIsAStruct = {
  nameOfAttribute : "value";
  anotherAttribute : 10;
};
```

There are several objects made available to the tool-script which it must use in order to print a scene.

PROGRESS

This object is used to control the main progress bar on the print panel. It has three functions:

- `setSteps(`*`integer`*`)`
  Use this method at the very start of the program to determine how many divisions to cut the progress bar into. Then, during the program, call `step()` that number of times.

- `step()`
  Invoke this method whenever progress has been made; for example, after slicing or pathing a mesh.
- `log(`*`string`*`)`
  Prints a message to the output log. Useful for debugging.
- `finish()`
  Call this at the end of printing to be sure both progress bars are filled

SLICER

The slicer object is used to compute the slices from triangle meshes. This must be done before those meshes are processed by the pather. Before slicing, the slice-height needs to be set by invoking the `setSliceHeight(`*`number`*`)` method. Usually, this is called where the parameter is taken from some tool's setting value; e.g. `setSliceHeight(plasticTool.sliceHeight)`. Then, each mesh is sliced with the doSlicing method; `slicer.doSlicing(`*`mesh`*`)`.

PATHER

The pather works almost exactly the same way as the slicer. To initialize the pather, it needs its variables set. Because which variables are available in a pather changes, one must use just a generic `set(`*`variableString`*`, `*`value`*`)` method and pass the name of the variable to set as the first parameter. All pathers need a path width variable value, so tool-scripts should make a call like `pather.set("PathWidth", plastic.pathWidth)` before pathing. After `doSlicing` has been called on a mesh, invoke `doPathing(`*`mesh`*`)`.

TOOLS

Each tool will be made available to the script as a variable with some properties. This variable has a property for each of its settings, and a property named "meshes". This property is an array that has a reference to each of the meshes that were assigned by the user to that tool. As an array, in QtScript it has two operations: one can access the number of elements in the array the "meshes.length" (useful for iterating over the elements) and each mesh is accessible by "meshes[i]" where 'i' is the index of the mesh in [0, meshes.length).

The mesh object itself has no attributes or functions; the only thing one can do with it is pass the mesh to `doSlicing` or `doPathing` as the parameter.

GLOBAL VARIABLES

Global variables come from the `<settings>...</settings>` section that is in the root of the fab-file tag. They can be accessed and manipulated like any other variable.

FABFILE

Once the meshes have been processed, this object is used to print instructions to the .FAB file. It provides a method to access a writer that implements this behavior for standard structures, so the script

doesn't have to specify the XML markup explicitly (although this is possible, if desired). To write a FabInterpreter-compatible file, get the writer using "`fabFile.fabAtHomeModel2Writer()`", and store its value into a new variable called "`fabWriter`". The next step is then to add all of the meshes that were processed in the tool-script. This will automatically take the paths from those meshes and insert them into the fab file; however, the script needs a bit more information. Every tool must have calibration settings. To create these, generate a calibration structure. Then, just invoke "`fabWriter.addMeshes(toolName, calibrationStruct, tool.meshes);`" where toolName is a string name for the material. Finally, sort the paths from bottom to top using "`fabWriter.sortBottomUp()`", specify the print-acceleration from the global variable with "`fabWriter.setPrintAcceleration(value);`" and write out the results with "`fabWriter.print();`"

```
// let there be some tool called "plastic"

var plasticMaterialCalibration = { pathSpeed: plastic.pathSpeed, depositionRate: plastic.depositionRate, pathWidth: plastic.pathWidth, depositionRate: plastic.depositionRate, pushout: plastic.pushout, suckback: plastic.suckback, suckbackDelay: plastic.suckbackDelay, clearance: plastic.clearance, pausePaths: plastic.pausePaths, pitch: plastic.pitch };

var fabWriter = fabFile.fabAtHomeModel2Writer();

fabWriter.addMeshes("plastic", plasticMaterialCalibration, plastic.meshes);

fabWriter.sortBottomUp();

fabWriter.setPrintAcceleration(printAcceleration);

fabWriter.print();
```