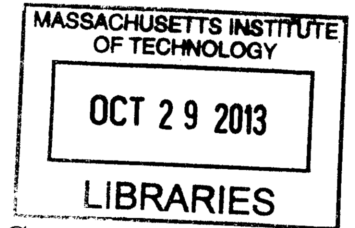
# Fab Trees for Designing Complex 3D Printable Materials


ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 29 2013
LIBRARIES

by

Ye Wang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

© Massachusetts Institute of Technology 2013. All rights reserved.

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
Aug 15, 2013

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Wojciech Matusik
Associate Professor
Thesis Supervisor

Accepted by . . . . . . . . . . . . .
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

# Fab Trees for Designing Complex 3D Printable Materials

by

Ye Wang

Submitted to the Department of Electrical Engineering and Computer Science
on Aug 15, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering

## Abstract

With more 3D printable materials being invented, 3D printers nowadays could replicate not only geometries, but also appearance and physical properties. On the software side, the tight coupling between geometry and material specification, and the lack of tools in specifying materials volumetrically, however, hinder the full usage of the multi-material capability of 3D printers. The heavy dependency on traditional modeling software also makes casual users, who are becoming one of the most important user groups, unwelcome in this rising area.

This thesis aims to solve the above problems by proposing fab trees for creating and combining procedural material specifications defined in OpenFL, a language for fabrication similar to the shading language for rendering. The fab tree representation allows users 1) to decouple material specification from geometry; hence, to be able to reuse the created materials on different models; 2) to easily create complicated materials systematically; 3) to have enough freedom to design materials procedurally, and fully utilize the functionality of today's multi-material 3D printers. In addition, I provide a fully functional user interface to explore desired visualization methods and user interactions for casual users in the 3D printing context.

Thesis Supervisor: Wojciech Matusik
Title: Associate Professor

# Acknowledgments

I would like to thank my supervisor, Prof. Matusik, for introducing me to 3D fabrication. He encouraged me to take ownership of my project. His devotion to his research is admirable and inspiring. There have been tough times while working on this thesis. I want to thank him for always trying to be supportive.

I would like to thank my mom for always telling me things will go fine. Her encouragement never fails to charge my positive energy.

I want to thank my teammates, especially Yam, Kiril, and Szu-po, for contributing to our discussions and providing me with help whenever needed.

Last but not the least, I want to thank my friends for their constant support.

# Contents

**1 Introduction** **17**

1.1 Approach . . . . . 18

1.2 Contributions . . . . . 19

1.3 Thesis Overview . . . . . 19

**2 Related Work** **21**

2.1 3D Printing . . . . . 21

2.2 Procedural Shading . . . . . 22

2.3 Procedural Solid Texture . . . . . 23

2.4 Functionally Graded Materials . . . . . 23

2.5 OpenFab . . . . . 24

**3 Fab Trees** **25**

3.1 Leaf Nodes . . . . . 25

3.1.1 Uniform Material . . . . . 25

3.1.2 Noise . . . . . 26

3.2 Internal Nodes . . . . . 28

3.2.1 Regular Structures . . . . . 28

3.2.2 Recursive Structures . . . . . 32

3.2.3 Interpolation in Object Space . . . . . 34

3.2.4 Interpolation in Parameter Space . . . . . 36

**4 User Interface and Workflow** **39**

4.1 Workflow . . . 39
4.2 Features . . . 41
4.2.1 Planar-cut View Mode . . . 41
4.2.2 Material Composition Visualization in Lab* Color Space . . . 42
4.2.3 FXAA Post Processing . . . 42

**5 Implementation** **45**
5.1 System . . . 45
5.1.1 External Dependency . . . 45
5.1.2 Library Import . . . 47
5.1.3 Model Geometry Rendering . . . 47
5.1.4 Model Appearance Rendering . . . 47
5.1.5 Planar-cut Mode Rendering . . . 48
5.2 Fab Tree Generation and Evaluation . . . 48
5.2.1 OpenFL Snippets . . . 49
5.2.2 Metafile Specification . . . 50
5.2.3 Fablet Concatenation . . . 52
5.2.4 Fablet Evaluation . . . 54
5.2.5 Texture Generation . . . 55

**6 Results** **57**
6.1 Performance . . . 57
6.1.1 Geometry Importing and Conversion . . . 57
6.1.2 Fab Tree Importing and Generation . . . 59
6.1.3 Texture Generation . . . 60
6.2 Rendered Results . . . 60

**7 Conclusions and Future Work** **63**
7.1 Conclusions . . . 63
7.2 Future Work . . . 64

# List of Figures

1-1 An example of a possible material specification. Blue and green indicate two different kinds of materials. . . . . . . . . . . . . . . . . . . . 18

3-1 Fab tree node `SingleMaterial`. For the three rendered results on the unit cube, `mateiral` = Material 0 in (b), Material 1 in (c), and Material 2 in (d). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

3-2 Fab tree node `ConstantComposition`. For the three rendered results on the unit cube, `materialA` = Material 0, `materialB` = Material 1, and `materialC` = Material 2. In (b), `percentA` = 0.25, `percentB` = 0.75. In (c), `percentA` = 0.50, `percentB` = 0.50. In (d), `percentA` = 0.75, `percentB` = 0.25. . . . . . . . . . . . . . . . . . . . . . . . . . 26

3-3 Fab tree node `PerlinNoise`. For the three rendered results on the right, `materialA` = Material 0, `materialB` = Material 1, `scale` = 4. `octave` is 1 in (b), 2 in ( c ), and 4 in (d). . . . . . . . . . . . . . . . 27

3-4 Fab tree node `Marble`. For the rendered result on the right, `materialA` = Material 2, `materialB` = Material 0, `scale` = 7, `octave` = 4. . . . 28

3-5 Fab tree node `Grid`. For the result on the left, `materialA` = Material 0, `materialB` = Material 1, `material_space` = Material 2, `spacing`= 0.01, `offset` = 0.05, `height` = 0.09, `widthA` = 0.08, `widthB` = 0.08. For the result on the right, `materialA` =`Marble` as defined in 3-4, `materialB` = Material 1, `spacing`= 0.01, `offset` = 0.05, `height` = 0.09, `widthA` = 0.2, `widthB` = 0.02. . . . . . . . . . . . . . . . . . . . 29

3-6 Fab tree node GridMap. The Map node takes Grid defined in Figure 3-5b as the material, and uses PolarMapping to map the position before passing into Grid. . . . . . . . . . . . . . . . . . . . . . . . . . 29

3-7 A 3D grid created using Grid. The Grid node is the same as the one in 3-5a. SimpleGrid is a simplified version of Grid with one material instead of two. materialA is set to Grid, material_space = Material 2, spacing= 0.01, offset = 0.05, height = 0.09 same as in the Grid node. The function node switches the z-plane and x-plane, and then shifts the x and y-coordinate by offset. . . . . . . . . . . . . . . . . 30

3-8 Fab tree node Foam. For the rendered result on a unit cube in (b), I have horizontal_gap = 0.05, vertical_gap = 0.05, horizontal_radius = 0.07, vertical_radius = 0.1, material_main is Material 0, and material_space is Material 2. For the result in (c), horizontal_gap = 0.02, vertical_gap = 0.02, horizontal_radius = 0.03, vertical_radius = 0.3, material_main is Material 1, and material_space is Material 2. 30

3-9 Fab tree node FoamMap. For (b) and (d), material points to the foam structure in 3-8b; for (c), material points to the foam structure in 3-8c. PerlinMapping uses Perlin noise to remap the position before passing it into the Foam node. To see the detailed mathematical definitions, refer to the context. . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3-10 Fab tree node Pattern. material points to the each FoamMap in Figure 3-9 respectively. unit defines the size of the material block. Here unit = 0.4. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

3-11 Fab tree node SpaceDivider. For the result on the left, materialA = Material 0, materialB = Material 1. gap_width = 0.03, and recursion_depth = 9. For the result on the right, materialA = Material 0, materialB = Marble as defined in 3-4. gap_width = 0.03, and recursion_depth = 6. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

3-12 Fab tree node Tree. branch_angle = $\frac{PI}{8}$, branch_width = 0.03, and recursion_depth = 8. material_tree is set to Material 0. material_other is set to Material 2. . . . . . . . . . . . . . . . . . . . . . . . . 33

3-13 Fab tree node LinearGradient. For the two rendered results on the right, materialA = Material 1, materialB = Material 0. direction = $(1, 0, 0)$ for (b) and $(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0)$ for (c). . . . . . . . . . . . . . . . . 34

3-14 Fab tree node RadialGradient. For the result on the left, materialA = Material 0, materialB = Material 1. For the result on the right, materialA = Material 0, materialB = Marble as defined in 3-4. . . . 35

3-15 Fab tree node Mask. The fab tree node Tree is the same as defined in Figure 3-12. LinearGradient is the same as defined in Figure 3-13a with the direction vector set as $(0, 1, 0)$. Foam is the same as defined in Figure 3-8c. For (b), mask_material_ID is set as Material 0. For (d), mask_material_ID is set as Material 2. . . . . . . . . . . . . . . . 35

3-16 Fab tree node Add. Add takes two materials, materialA and materialB, and adds them up together. Mask is defined in Figure 3-15a, and Mask1 is defined in Figure 3-15c. . . . . . . . . . . . . . . . . . . . . . . 36

3-17 Fab tree node ParameterInterpolation interpolating the two Grid in 3-5. mapping = LinearMapping in the $x$-direction.GridInterpolatable is a function node. materialB = Material 1, material_space = Material 2, spacing= 0.01, offset = 0.05, height = 0.09. GridInterpolation is a volume node. materialA_begin = Material 0, widthA_begin = 0.08, widthB_begin = 0.08; materialA_end = Material 1, widthA_end = 0.2, widthB_end = 0.02. . . . . . . . . . . . . . . . . . . . . . . . 37

3-18 Fab tree node ParameterInterpolation interpolating the two Foam fab nodes in Figure 3-8. . . . . . . . . . . . . . . . . . . . . . . . 37

4-1 Layout of the user interface. *Material list* is on the top of the page, demonstrating the base materials and colors they are rendered with. *Fab tree list* is on the left side of the page. It lists the fab trees loaded from the local directory. Users can pick the fab tree by clicking the icon. The selected fab tree is highlighted in light gray. The *rendered model* is at the center of the page. Users can rotate the model in real time. *Planar-cut slider* is on the right of the page. Users drag the slider to slice through the volume of the model. . . . . . . . . . . . . 40

4-2 Planar-cut view for the pig model. The first row has step 4, and the second row has step 10. From left to right, the material is `SingleMaterial` with base Material 0, `SingleMaterial` with base Material 1, and `Marble`. The artifact on the back of the pig is caused by non-consistent face normal directions in the input model. . . . . . . . . . . . . . . . 41

4-3 Demonstration of material composition visualization in Lab* color space. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

4-4 Before and after FXAA post processing on the rendering of the pig's tail. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

5-1 System architecture for Crumb. . . . . . . . . . . . . . . . . . . . . . . 46

5-2 Fab tree for LinearGradient with generated IDs. . . . . . . . . . . . . 54

5-3 Illustration for generating the texture for the surface of a cube model. The triangle being evaluated is highlighted in red both on the 3D cube and on the texture plane. The system checks all pixels in the bounding box of the triangle on the texture plane. and finds the corresponding 3D position in the 3D cube. The system queries the compiled fablet to get the material composition at the position, translates that composition to a color, and stores in the texture. On the right is a more detailed view of the generated texture with multiple triangles. Only the boundaries are marked with colors. . . . . . . . . . . . . . . . . . 55

6-1 The "pig" model with three different foam materials . . . . . . . . . . 61

6-2 The “dinosaur” model with three different foam materials. . . . . . . 61
6-3 Buildings with a marble material and a grid material. . . . . . . . . . 61

# List of Tables

6.1 Performance on importing model of different sizes. The number in the brackets behind the model names refers to the number of faces in the model. *b* stands backend, *f* stands for frontend, and *bf* stands for backend and frontend. . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.2 Performance on importing and generating fab trees. . . . . . . . . . . 59
6.3 Performance on texture generation for different fab trees on different models with different texture sizes. . . . . . . . . . . . . . . . . . . . . 60

# Chapter 1

# Introduction

3D printing technology has been called "a third industrial revolution" by The *Economist* magazine. In contrast to traditional subtractive manufacturing technologies, such as moulding, 3D printing builds models by laying out materials layer by layer. This gives fine control to every "voxel" of the printed model, and requires much less preparation time, space, or technical knowledge on the part of users.

Nowadays, 3D printers are no longer limited to printing prototypes in research labs or big design firms. As the technology emerges, 3D printers can print models with very high resolution and with various materials. They are used in printing medical devices, engineering parts, optical devices, and even human tissues. As the price drops, 3D printers become more and more available to casual users. People can order 3D printed jewelry and toys online and get them shipped to their homes. Designers design clothes, lamp shades, and shoes with 3D printers.

Despite the advances in material science and hardware, the workflow on the software side, however, has stayed underexplored. In a traditional workflow, users start by generating a 3D model with CAD software. If they want to print a model with multiple materials, users have to split the model into separate parts, and convert each part to an STL format. This conversion only preserves the geometric information of the model. The models are then imported in a printer-specific user interface. Users assign a material to each part manually.

In traditional modeling, only the appearance on the model surface is represented.

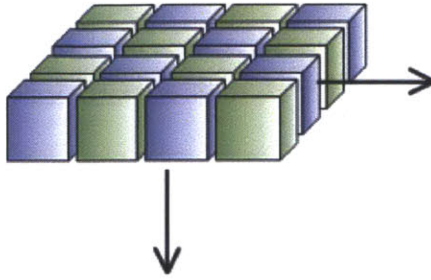


Figure 1-1: An example of a possible material specification. Blue and green indicate two different kinds of materials.

Directly readapting traditional modeling to doing material specification for 3D printing leaves materials inside model volumes unrepresented. This hinders the full usage of the capability of 3D printing. For example, users might want to specify some inner structures with a stiffer material to support the soft outer model. Alternatively, users might want to have a lighter material distributed across the model to modify its weight or center of mass.

The software workflow tightly couples material specification with modeling. Users have to remodel every time to apply the same material specification on a different model. For example, for a simple grid design as shown in Figure 1-1, the pipeline right now requires users to split a model into a huge number of cube meshes, and assign each cube one material. The operation is tedious. If users want to apply the above material specification on a different model, they would have to redo the same operations. This workflow is especially unwelcome to casual users with limited modeling experience.

## 1.1 Approach

In this thesis, I use a fab tree representation for designing materials. This representation decouples material specifications from geometries, thus allowing users to easily apply the designed materials on different models. The leaf nodes are volumetric material specifications defined procedurally using OpenFL, a language similar to the shading language in rendering. I provide a small collection of leaf nodes, such as

uniform composite materials and noise patterns. Leaf nodes are combined by internal nodes through structures, blending, and interpolation. For instance, I include linear gradient, add, and mask nodes. Each node has a set of parameters to which users could bind different values.

I provide a web-based interface that lets users apply the fab-tree-designed materials to models. The interface is designed specifically for casual users with limited modeling experience. Users can easily apply a library of predefined material specifications on a model they desire. The interface visualizes material specifications on model surfaces. I provide a planar-cut mode, where users can slice through a model with a plane to view the inner material specification.

## 1.2 Contributions

The fab tree representation has the following advantages:

1. Material specification is decoupled from geometry. Users can reuse the created materials on different models without remodeling.

2. Materials are designed procedurally. Any given position within a volume is defined with a material composition. This procedural material specification can fully utilize the functionality of today's 3D printers.

3. Creating materials is more systematic. Users can easily create complicated materials through combining simpler fab trees, instead of writing a new program from scratch.

In addition, I provide a fully functional user interface to explore desired visualization methods and user interactions for casual users in the 3D printing context.

## 1.3 Thesis Overview

In Chapter 2, I introduce 3D printing in more depth. I compare my work to previous work in procedural shading and solid texturing in computer graphics, and to func-

tional graded materials in material science and mechanical engineering. In addition, I include an explanation of the OpenFab project, in particular, OpenFL. Chapter 3 describes the fab tree nodes, and shows the corresponding materials rendered on a unit cube. Chapter 4 demonstrates the user interface and workflow. I also explain some features, such as color visualization for materials and planar-cut view mode. System and implementation details are described in Chapter 5. Chapter 6 evaluates the performance, and shows more rendered results. I sum up the work and suggest future directions in Chapter 7.

# Chapter 2

# Related Work

This thesis extends work in 3D printing and procedural shading. I give a more detailed introduction to today's multi-material 3D printers and software pipelines in 2.1. In 2.2, I introduce previous work in procedural shading, which serves as a main inspiration for my work. Some of the fab tree materials are inspired by previous work in procedural solid textures, which are introduced in 2.3. 2.4 explains similar attempts for designing 3D printable materials in mechanical engineering and material science. In the last subsection 2.5, I explain OpenFab, the 3D printing framework this thesis is based on.

## 2.1 3D Printing

There are several product lines that support multi-material or multi-color 3D printers. Objet manufactured the world's first multi-material 3D printer, the Objet260 Connex. The Objet260 Connex can handle up to 51 composite materials, simulating anything from rubber to transparency to rigid ABS-grade plastics. The ZPrinter 450 from 3DSystems can support 180,000 colors with two print heads. ZPrinter 650 can support 390,000 colors with five print heads. The MakerBot Replicator supports dual extruders that can print models with 2 colors.

These multi-material 3D printing technologies enable people to use them in ways that were not imagined before. In fashion, dedicated jewelry, dresses and footwear

can be 3D printed. In architecture and interior design, 3D printers can build from realistic prototypes to usable lamps and furniture. In medicine, 3D printing can be used to print lattices for growing tissues, dental plaster-models, and artificial limbs. In manufacturing, rapid prototyping with 3D printing can save a great amount of modeling and shipping time. The produced prototypes are more akin to their 3D designs. In film production, 3D printed figures can better match the appearance of real or animated characters.

3D printing has had a fast growth in popularity among casual users in recent years. With services such as Shapeways and Ponoko, people can order 3D printed models online, and get them shipped within weeks. Free online CAD libraries, such as GrabCAD, provides thousands of printable models for people to download. Online tools such as *nervous system* help users to design more customized models.

The current 3D printing workflow starts with users modeling in a traditional CAD/CAM software. CAD/CAM software have been widely used in the professional design and manufacturing context for over thirty years. Most well-known commercial software packages include Rhinoceros 3D, Solidworks, Autodesk Maya, and AutoCAD. There are also many free and open-source packages, such as Blender. Most CAD/CAM software provides a rich set of features; however, it is hard to use without professional training.

Objet recently released plug-ins for Solidworks and several other CAD/CAM software packages. The plug-in, while saving users from switching between applications, does not change the underlying workflow. Generating material specifications relies on geometry modeling. 3DSystems' ZPrinter series provide user interfaces which support models with colors, but color specifications are limited to surfaces.

## 2.2 Procedural Shading

The Shade Trees paper by Cook [3] introduced the concept of programmable shading to the realistic rendering community. Shade trees represent programmable shaders as nodes in a tree, and they can be combined together to generate more complicated

shadings. Shade trees are widely adapted in professional rendering pipeline, such as RenderMan. The shade tree serves as a strong inspiration to my work.

Grabli et al. [5] adapt programmable shaders to the stylization of strokes in line drawings. Their programmable shaders are used to control color, texture and thickness of strokes. Lopez-Moren et al. [10] adapt the shade tree approach for vector graphics to depict stylized materials, giving control to material attributes like shading and reflections.

## 2.3 Procedural Solid Texture

In computer graphics, solid textures are used to perform high-quality sub-surface scattering, and efficient internal surface rendering for fracturing objects. Different algorithms are used to synthesize solid textures. Pietroni et al. survey solid texture synthesis methods in [11]. My work is most related to procedural solid texturing, which synthesizes textures as a function of coordinates and a set of tuning parameters. Cook in his paper [3] define 3D noise functions for creating realistic solid patterns. Culter et al. [4] developed a scripting language to procedurally authoring layered, solid models using a tetrahedral representation. Previous procedural solid textures are used for rendering purposes. The output of the fab trees can be used to generate printable models directly. The fab tree approach is also a more systematical way for designing materials.

## 2.4 Functionally Graded Materials

To address varying material composition within a volume, in material science and mechanical engineering, there has been prior work on functionally graded materials (FGM). Jackson [7] has proposed a volumetric representation based on tetrahedra and voxels in 2000. MIT's three-dimensional printing group describes a system using signed distance filed for representing geometry, and based on that, represent material specifications as a composition function [9] [13]. The system decouples material speci-

fication from geometry, and is more flexible and systematic in generating complicated material specifications.

## 2.5 OpenFab

The OpenFab pipeline [12] offers a programming model for procedurally specifying material of printable objects, and synthesizing the final voxels of material at full printer resolution. This pipeline provides efficient storage and communication, as well as resolution independence for different hardware and output contexts. *Fablets* procedurally modify the geometry and define material composition. Fablets are defined in OpenFL, a domain-specific language, similar to the shading language in rendering. The pipeline is designed to progressively stream output to the printer with minimal pre-computation and with only a small slab of the volume kept in memory at any one time.

# Chapter 3

# Fab Trees

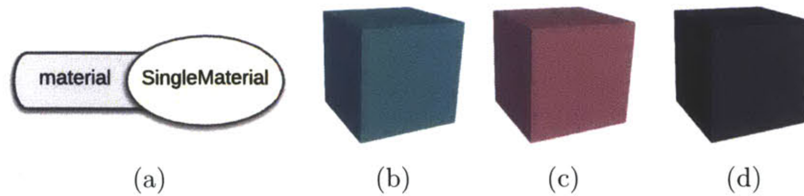
Materials are created by combining fab nodes into fab trees. The collection of fab nodes I choose include a small number of leaf fab nodes (3.1). These leaf nodes represent uniform materials and some natural noise patterns. The leaf nodes are combined by internal nodes (3.2) through designed structures, blending or parameter interpolation. In order to make materials fit geometry better, I also provide a collection of "mapping" nodes, marked in color red in the diagrams in this chapter.

## 3.1 Leaf Nodes

### 3.1.1 Uniform Material

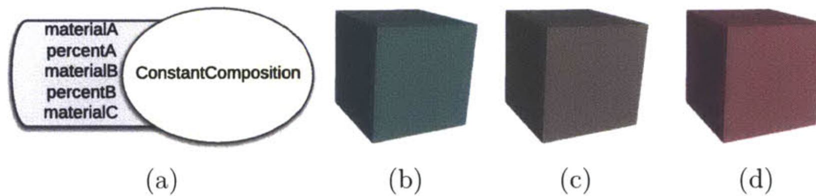
**Single Material** `SingleMaterial` represents a uniform material with one base material. The node takes the material ID of a base material as its variable input. Figure 3-1 shows the rendered results on a unit cube with three different base materials - Material 0, 1 and 2. The color scheme is explained further in 4.2.2.

**Constant Composition** `ConstantComposition` represents uniform composite materials, that are defined by specifying the percentage of each base material. In the example in Figure 3-2, `materialA`, `materialB` and `materialC` are base material IDs.




(a) (b) (c) (d)

Figure 3-1: Fab tree node `SingleMaterial`. For the three rendered results on the unit cube, `mateiral` = Material 0 in (b), Material 1 in (c), and Material 2 in (d).




(a) (b) (c) (d)

Figure 3-2: Fab tree node `ConstantComposition`. For the three rendered results on the unit cube, `materialA` = Material 0, `materialB` = Material 1, and `materialC` = Material 2. In (b), `percentA` = 0.25, `percentB` = 0.75. In (c), `percentA` = 0.50, `percentB` = 0.50. In (d), `percentA` = 0.75, `percentB` = 0.25.

`percentA` and `percentB` represent the percentages of `materialA` and `materialB`.

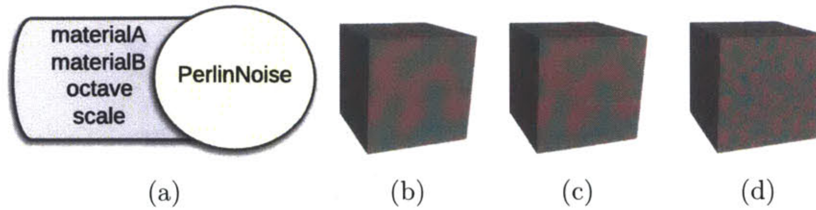
$$\texttt{percentA} + \texttt{percentB} \leq 1$$
$$\texttt{percentC} = 1 - \texttt{percentA} - \texttt{percentB}$$

For the three rendered results in Figure 3-2, the percentage for `materialC` is 0. The percentage for `materialA`, from left to right, is 0.25, 0.5, and 0.75; 0.75, 0.5, and 0.25 for `materialB`.

### 3.1.2 Noise

Noise is the texture primitive in computer graphics. When combined with math functions, noise can generate interesting and realistic-looking textures. I here use Perlin noise [3] as the primitive to generate speckle- and marble-looking materials.

**Speckle** `Speckle` takes two input material nodes, `materialA` and `materialB`. `octave` is an integer, and `scale` is a double.




(a) (b) (c) (d)

Figure 3-3: Fab tree node `PerlinNoise`. For the three rendered results on the right, `materialA` = Material 0, `materialB` = Material 1, `scale` = 4. `octave` is 1 in (b), 2 in ( c ), and 4 in (d).

The `Speckle` fab node first scales the geometry by `scale`,

$$\dot{P}_{new} = \texttt{scale} \cdot \dot{P}$$

Then it computes the turbulence by

$$\texttt{turbulence}(\dot{Q}) = \sum_{i=0}^{\texttt{octave-1}} \frac{perlinnoise(2^i \cdot \dot{Q})}{2^i}$$

The turbulence is adjusted to fit the 0 to 1 range,

$$\texttt{turbulence}'(\dot{Q}) = clamp(0.5 \cdot \texttt{turbulence}(\dot{Q}) + 0.5, 0, 1)$$

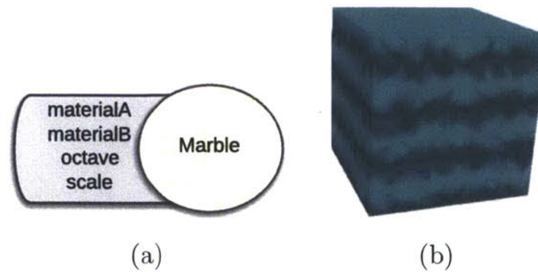
`materialA` is assigned with percentage $\texttt{turbulence}'(\dot{P}_{new})$, and `materialB` with percentage $1 - \texttt{turbulence}'(\dot{P}_{new})$.

Figure 3-3 shows the rendered results with `materialA` set to Material 0, `materialB` set to Material 1, `scale` set to 4, and `octave` set to 1, 2, 4 respectively from left to right.

**Marble** `Marble` takes the same set of input variables and nodes as `Speckle`. It uses the sin function to mimic the oscillation appearance in marbles.

$$\texttt{marble}(\dot{Q}) = 0.5 \cdot \sin(\dot{Q}_y + \texttt{turbulence}(\dot{Q})) + 0.5$$

`materialA` is assigned with percentage $\texttt{marble}(\dot{P}_{new})$ and `materialB` with $1-$




(a) (b)

Figure 3-4: Fab tree node `Marble`. For the rendered result on the right, `materialA` = Material 2, `materialB` = Material 0, `scale` = 7, `octave` = 4.

`marble`$(\dot{P}_{new})$.

Figure 3-4 shows the rendered result with `materialA` set to Material 0, `materialB` set to Material 2, `scale` set to 7, and `octave` set to 4.

## 3.2 Internal Nodes

### 3.2.1 Regular Structures

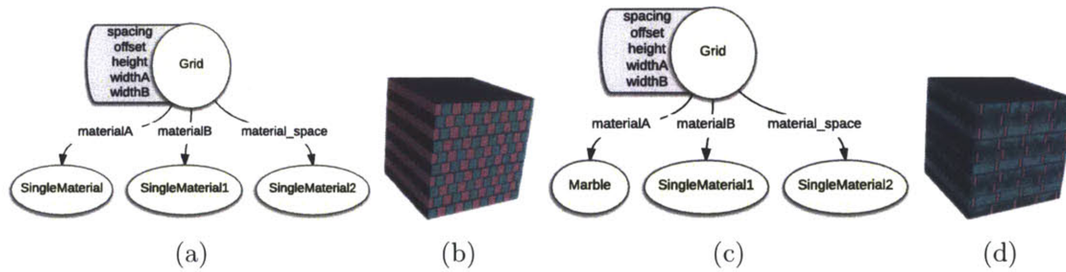
**Grid** `Grid` represents a brick wall pattern. The pattern consists of bricks with `materialA` and `materialB`, and gaps with `material_space`. `offset` is the offset in $x$-direction of the odd horizontal rows from the even ones. `widthA` is the width of bricks with `materialA`; respectively `widthB` for bricks with `materialB`. `height` is the height of each brick. `spacing` is the size of gaps.

Figure 3-5b shows the rendered result of a wall of equal-sized bricks. When changing `materialA` to `Marble`, and adjusts the widths of bricks, I get a wall in Figure 3-5d.

To pack the bricks circularly for a tube, I simply add a `PolarMapping` node on top of the Grid node. `PolarMapping` maps the original position in Cartesian coordinate system into Polar coordinate system.

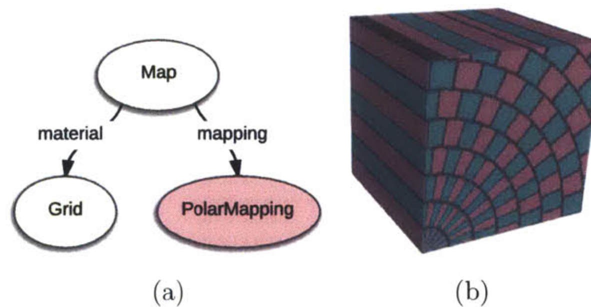
$$\texttt{PolarMapping}(\dot{P}) = (\frac{2 \cdot atan2(P_y, P_x)}{PI}, \sqrt{\frac{{P_x}^2 + {P_y}^2}{2}}, P_z)$$

See Figure 3-6 for an example.




(a) (b) (c) (d)

Figure 3-5: Fab tree node `Grid`. For the result on the left, `materialA` = Material 0, `materialB` = Material 1, `material_space` = Material 2, `spacing`= 0.01, `offset` = 0.05, `height` = 0.09, `widthA` = 0.08, `widthB` = 0.08. For the result on the right, `materialA` =`Marble` as defined in 3-4, `materialB` = Material 1, `spacing`= 0.01, `offset` = 0.05, `height` = 0.09, `widthA` = 0.2, `widthB` = 0.02.



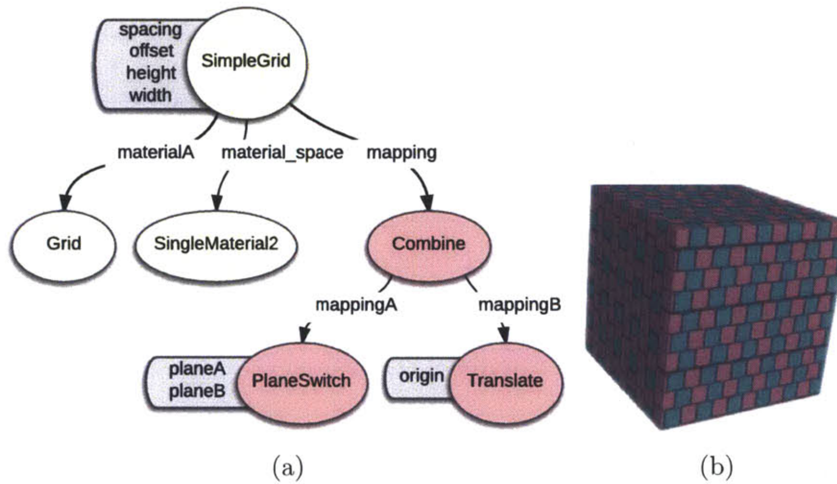

(a) (b)

Figure 3-6: Fab tree node `GridMap`. The `Map` node takes `Grid` defined in Figure 3-5b as the `material`, and uses `PolarMapping` to map the position before passing into `Grid`.

So far, the bricks are packed in the $xy$-plane. To generate a 3D grid, I add a `SimpleGrid` node to combine `Grid`. `SimpleGrid`, besides taking the width, height and materials, takes an additional mapping node, `mapping`. It switches the $x$ and $z$-plane, and offset the $x$ and $y$-coordinate for the odd rows. See Figure 3-7 for the rendered result.

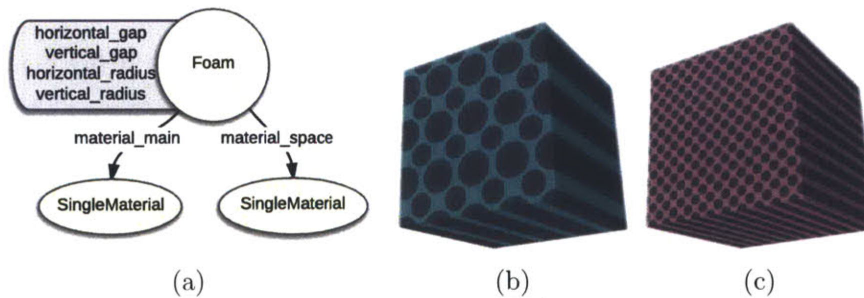
**Foam** `Foam` represents a material structure that contains regular holes of a different material from the main structure material. Figure 3-8 shows two examples of foams with different hole sizes and structure materials.

To generate more interesting foam structures, I add a `Map` node on top of `Foam`. See Figure 3-9a for the fab tree. The `Map` node takes a mapping node, which uses



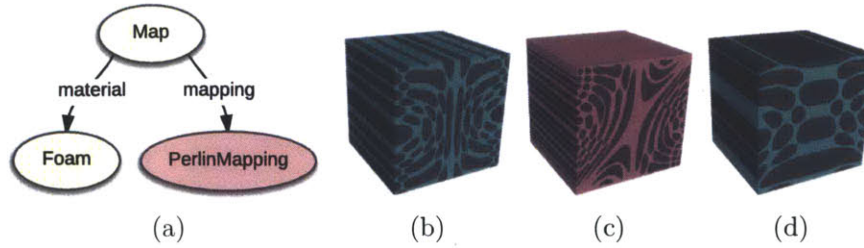

(a) (b)

Figure 3-7: A 3D grid created using `Grid`. The `Grid` node is the same as the one in 3-5a. `SimpleGrid` is a simplified version of `Grid` with one material instead of two. `materialA` is set to `Grid`, `material_space` = Material 2, `spacing`= 0.01, `offset` = 0.05, `height` = 0.09 same as in the `Grid` node. The function node switches the z-plane and x-plane, and then shifts the x and y-coordinate by `offset`.




(a) (b) (c)

Figure 3-8: Fab tree node `Foam`. For the rendered result on a unit cube in (b), I have `horizontal_gap` = 0.05, `vertical_gap` = 0.05, `horizontal_radius` = 0.07, `vertical_radius` = 0.1, `material_main` is Material 0, and `material_space` is Material 2. For the result in (c), `horizontal_gap` = 0.02, `vertical_gap` = 0.02, `horizontal_radius` = 0.03, `vertical_radius` = 0.3, `material_main` is Material 1, and `material_space` is Material 2.




(a) (b) (c) (d)

Figure 3-9: Fab tree node `FoamMap`. For (b) and (d), `material` points to the foam structure in 3-8b; for (c), `material` points to the foam structure in 3-8c. `PerlinMapping` uses Perlin noise to remap the position before passing it into the `Foam` node. To see the detailed mathematical definitions, refer to the context.

Perlin Noise to remap the position before passing it into `Foam`.

Let $pn$ stand for *perlinnoise*, for the example in Figure 3-9b, `mapping` is defined as

$$\texttt{mapping}_x(\dot{P}) = pn(P_x, P_y) + pn(1 - P_x, 1 - P_y)$$
$$\texttt{mapping}_y(\dot{P}) = pn(P_x, P_y) + pn(P_x, 1 - P_y)$$
$$\texttt{mapping}_z(\dot{P}) = P_z$$

In Figure 3-9c,

$$\texttt{mapping}_x(\dot{P}) = pn(P_x, P_y)(1 - P_x) + pn(1 - P_x, 1 - P_y)P_x$$
$$\texttt{mapping}_y(\dot{P}) = pn(P_x, P_y)(1 - P_y) + pn(1 - P_x, 1 - P_y)P_y$$
$$\texttt{mapping}_z(\dot{P}) = P_z$$

And in Figure 3-9d,

$$\texttt{mapping}_x(\dot{P}) = pn(P_x, P_y)(1 - P_x) + pn(1 - P_x, P_y)P_x$$
$$\texttt{mapping}_y(\dot{P}) = P_y$$
$$\texttt{mapping}_z(\dot{P}) = P_z$$

I can stack these Perlin foam structures together using the `Pattern` node. The results are shown in Figure 3-10.

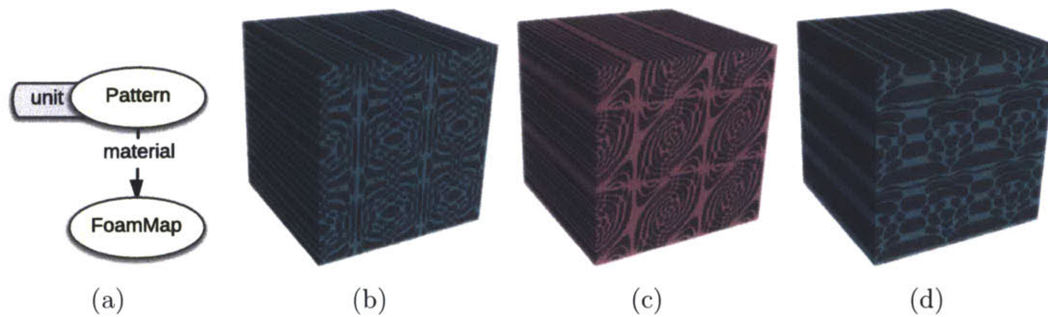



Figure 3-10: Fab tree node `Pattern`. `material` points to the each `FoamMap` in Figure 3-9 respectively. `unit` defines the size of the material block. Here `unit` = 0.4.

### 3.2.2 Recursive Structures

The following two nodes are defined recursively. They demonstrate the capability of fab trees to create fractal- and L-system-like structures.

**SpaceDivider** `SpaceDivider`, at each step, picks a plane and splits the volume into two using function `isInGap`. The node uses function `getNewPos` to transform positions. The divided volumes recursively call `SpaceDivider` to further divide.

In Figure 3-11a, the tree uses a straight line in the picked plane to divide. The recursion level is set to 9, and the width of the dividing lines is 0.03. See 3-11b for the rendered result.

The tree in Figure 3-11c uses *sin* curves to divide. The gap lines are curved like tree branches. For the rendered result in Figure 3-11d, the recursion level is set to 6, and the width of the dividing lines is 0.03.

**Tree** The `Tree` node represents a tree pattern. At each level, the node generates six branches, and simultaneously divides the volume into six sub-volumes centered by each branch. Each sub-volume then recursively calls `Tree` to keep generating tree-like structure.

`branch_angle` defines the angle between each adjacent branches, and `branch_width` defines the width of the initial branch. For the rendered result in Figure 3-12, `branch_angle` is set to $\frac{PI}{8}$ with 8 recursion levels.

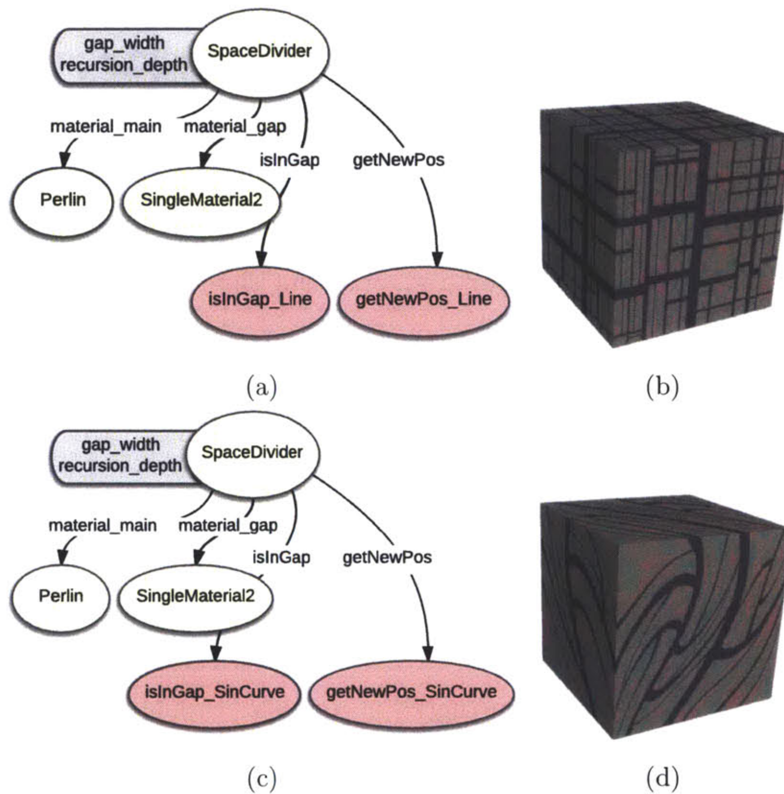



Figure 3-11: Fab tree node `SpaceDivider`. For the result on the left, `materialA` = Material 0, `materialB` = Material 1. `gap_width` = 0.03, and `recursion_depth` = 9. For the result on the right, `materialA` = Material 0, `materialB` = `Marble` as defined in 3-4. `gap_width` = 0.03, and `recursion_depth` = 6.

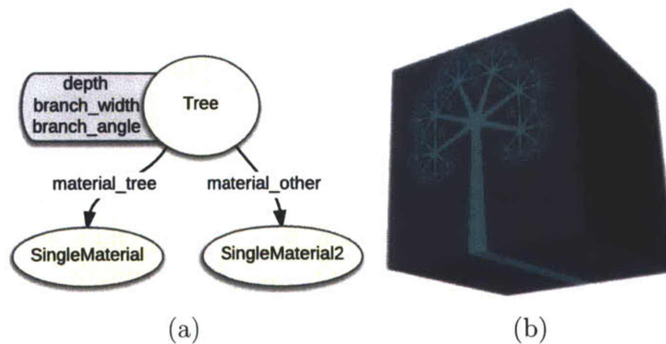
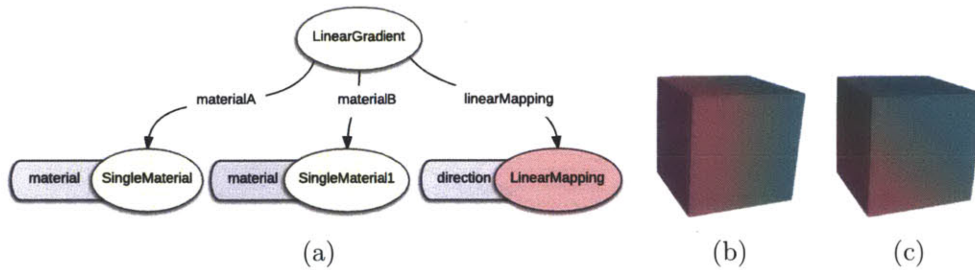



Figure 3-12: Fab tree node `Tree`. `branch_angle` = $\frac{PI}{8}$, `branch_width` = 0.03, and `recursion_depth` = 8. `material_tree` is set to Material 0. `material_other` is set to Material 2.




(a) (b) (c)

Figure 3-13: Fab tree node `LinearGradient`. For the two rendered results on the right, `materialA` = Material 1, `materialB` = Material 0. `direction` = $(1, 0, 0)$ for (b) and $(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0)$ for (c).

### 3.2.3 Interpolation in Object Space

**LinearGradient** `LinearGradient` represents blending of two materials, `materialA` and `materialB` along a vector, $\vec{d}$.

$$\texttt{linearMapping}(\dot{P}) = clamp(\vec{d} \cdot \dot{P}, 0, 1)$$

`materialA` has percentage $1 -$ `linearMapping`$(\dot{P})$, and `materialB` has percentage `linearMapping`$(\dot{P})$. See Figure 3-13 for examples of `LinearGradient` with two different direction vectors blending Material 0 and Material 1.

**RadialGradient** `RadialGradient` blends two materials, `materialA` and `materialB` in the radial direction using

$$\texttt{radialMapping}(\dot{P}) = clamp(|\dot{P}|, 0, 1)$$

`materialA` is set with percentage `radialMapping`$(\dot{P})$, and `materialB` with percentage $1 -$ `radialMapping`$(\dot{P})$.

Figure 3-14a shows the fab tree of `RadialGradient` blending from Material 1 to Material 0. Figure 3-14c shows the fab tree of `RadialGradient` blending from `Marble` to Material 0. The rendered results are shown in 3-14c and 3-14d.

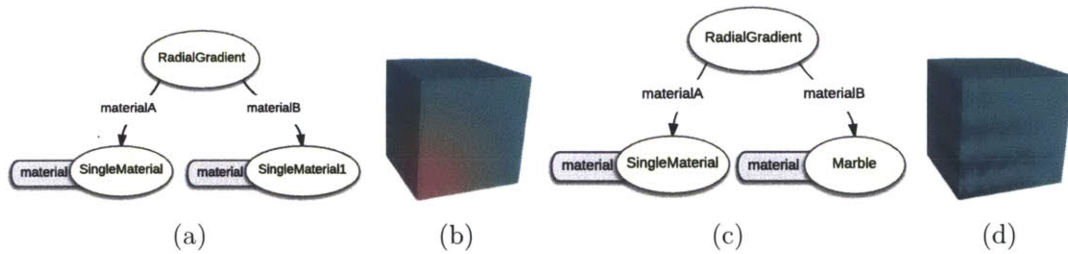



Figure 3-14: Fab tree node `RadialGradient`. For the result on the left, `materialA` = Material 0, `materialB` = Material 1. For the result on the right, `materialA` = Material 0, `materialB` = `Marble` as defined in 3-4.

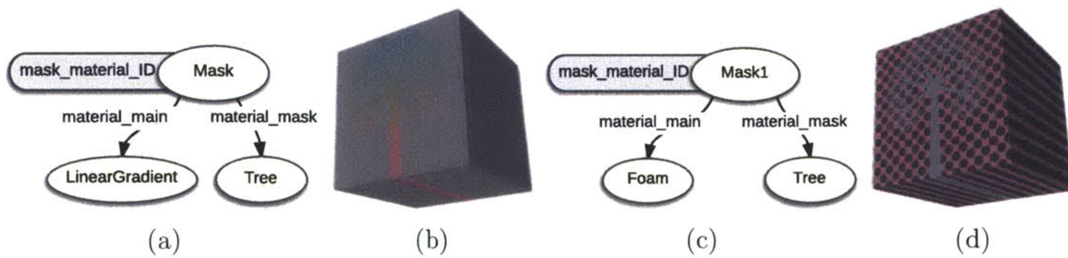



Figure 3-15: Fab tree node `Mask`. The fab tree node `Tree` is the same as defined in Figure 3-12. `LinearGradient` is the same as defined in Figure 3-13a with the direction vector set as $(0, 1, 0)$. `Foam` is the same as defined in Figure 3-8c. For (b), `mask_material_ID` is set as Material 0. For (d), `mask_material_ID` is set as Material 2.

**Mask** `Mask` uses one material as a mask to mask out the other material. The node takes two nodes `material_mask` and `material_main`, and a variable `mask_material_ID`. The percentage of base material `mask_material_ID` determines the percentage of `material_main` in the result material.

See Figure 3-15 for two examples using the `Tree` node as defined in Figure 3-12. The example on the left uses Material 0 as the mask to mask out a linear gradient material; and the example on the right uses Material 2 as the mask to mask out a foam material.

**Add** `Add` adds two materials together, and rescales the composition to make the percentages of each base material add up to 1. See Figure 3-16 for an example of adding `Mask` and `Mask1` defined in Figure 3-15.

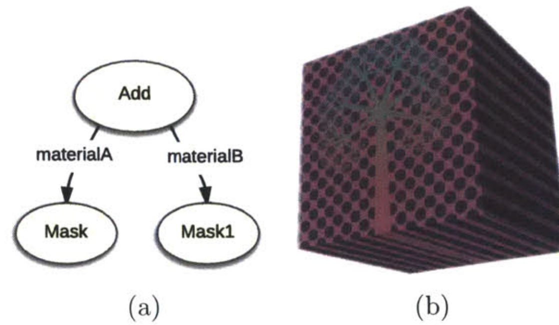



Figure 3-16: Fab tree node `Add`. `Add` takes two materials, `materialA` and `materialB`, and adds them up together. `Mask` is defined in Figure 3-15a, and `Mask1` is defined in Figure 3-15c.

### 3.2.4 Interpolation in Parameter Space

Besides blending, I provide parameter interpolation to give more node-specific controls to interpolation. The `ParameterInterpolation` node takes two nodes, a volume fab node `interpolation` and a mapping node `interpolatable`. Different from its normal counterpart fab node, `interpolation` fab node contains a set of variables and nodes that are interpolatable. `interpolation` takes a blending node, `mapping`, to blend the values of those interpolatable variables. It then passes in the interpolated variables to `interpolatable`, and evaluates the materials within a volume.

Figure 3-17 gives an example of parameter interpolation for two grid material specifications as defined in Figure 1-1. A normal `Grid` fab node contains variables `spacing`, `height`, `offset`, `widthA`, `widthB`, and three volume fab nodes `materialA`, `materialB` and `material_space`. The `GridInterpolatable` node has `spacing`, `height`, `offset`, `materialB` and `material_space` fixed. `GridInterpolation` uses `LinearMapping` in the $(1, 0, 0)$ direction to interpolate `widthA`, `widthB`, and `materialA`.

Figure 3-18 gives an example of interpolating two foams, defined in Figure 3-8, in a radial direction. All variables of the two foams are interpolated using `RadialMapping`.

These interpolation nodes in object space and parameter space allow users to easily create more complicated materials from simple nodes, and give users fine control on the final result. Users with little experience can quickly adapt a material on their model by updating the parameters.

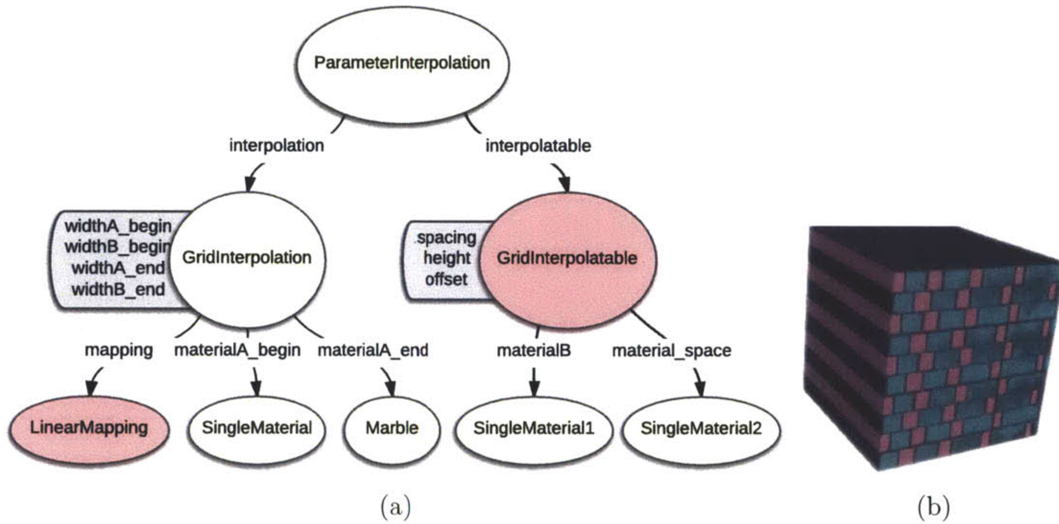



Figure 3-17: Fab tree node `ParameterInterpolation` interpolating the two `Grid` in 3-5. `mapping` = `LinearMapping` in the $x$-direction.`GridInterpolatable` is a function node. `materialB` = Material 1, `material_space` = Material 2, `spacing`= 0.01, `offset` = 0.05, `height` = 0.09. `GridInterpolation` is a volume node. `materialA_begin` = Material 0, `widthA_begin` = 0.08, `widthB_begin` = 0.08; `materialA_end` = Material 1, `widthA_end` = 0.2, `widthB_end` = 0.02.

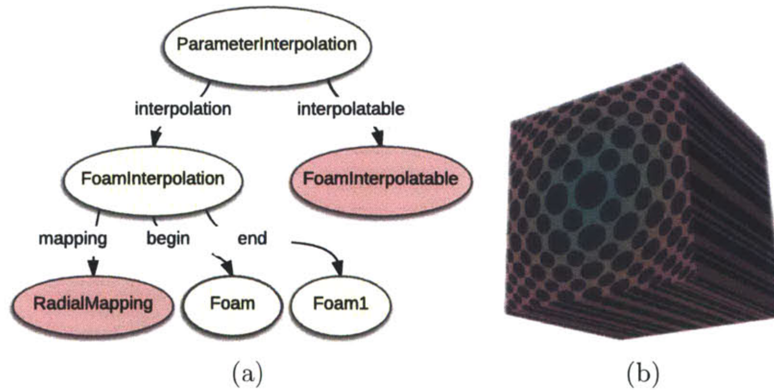



Figure 3-18: Fab tree node `ParameterInterpolation` interpolating the two `Foam` fab nodes in Figure 3-8.

# Chapter 4

# User Interface and Workflow

I implement the user interface for specifying and visualizing fab tree materials. The user interface is implemented as a browser plugin, and is interactive. After loading a model, users can select a fab tree material from an user-defined list. The model is automatically visualized with the material on the surface. To view the material inside of the volume, users can enter the planar-cut mode. In this chapter, I first explain the user interface workflow in 4.1. I explain the features, including planar-cut mode, material composition visualization scheme, and post processing for anti-aliasing, in 4.2.

## 4.1 Workflow

Users start by picking the *material list*, marked in red in Figure 4-1. In the screenshot, I use three materials from Stratasys Objet500 Connex for demonstration. The *fab tree list* is loaded on the left side of the page. The set of fab trees are read from the local directory, and could be easily modified by users. The selected fab tree is highlighted in light gray. The rendered model is at center of the page. Users can rotate the model to view from different angles. The model is first rendered with the default material, `SingleMaterial`. Users change materials by clicking on a fab tree listed on the left. The model is updated with the new material specification within seconds.

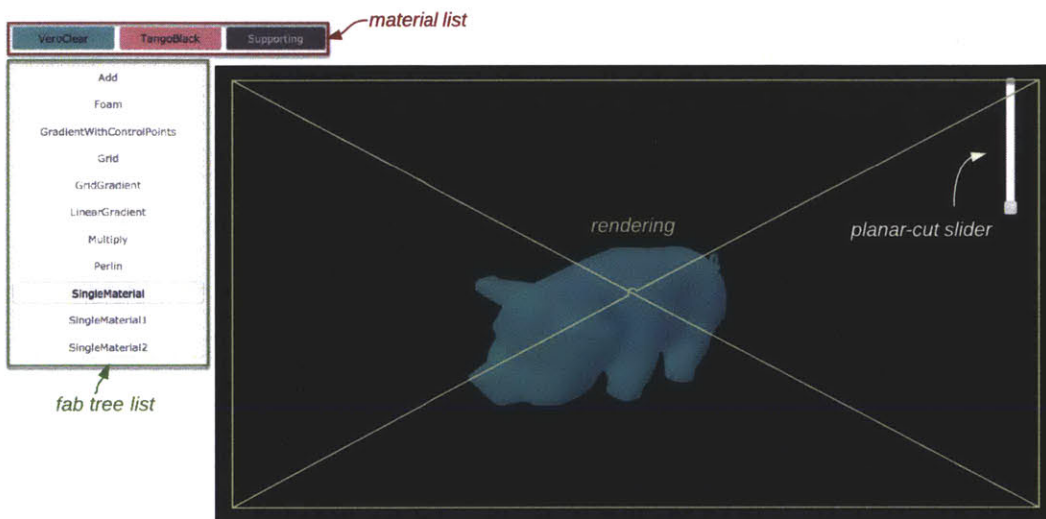



Figure 4-1: Layout of the user interface. *Material list* is on the top of the page, demonstrating the base materials and colors they are rendered with. *Fab tree list* is on the left side of the page. It lists the fab trees loaded from the local directory. Users can pick the fab tree by clicking the icon. The selected fab tree is highlighted in light gray. The *rendered model* is at the center of the page. Users can rotate the model in real time. *Planar-cut slider* is on the right of the page. Users drag the slider to slice through the volume of the model.

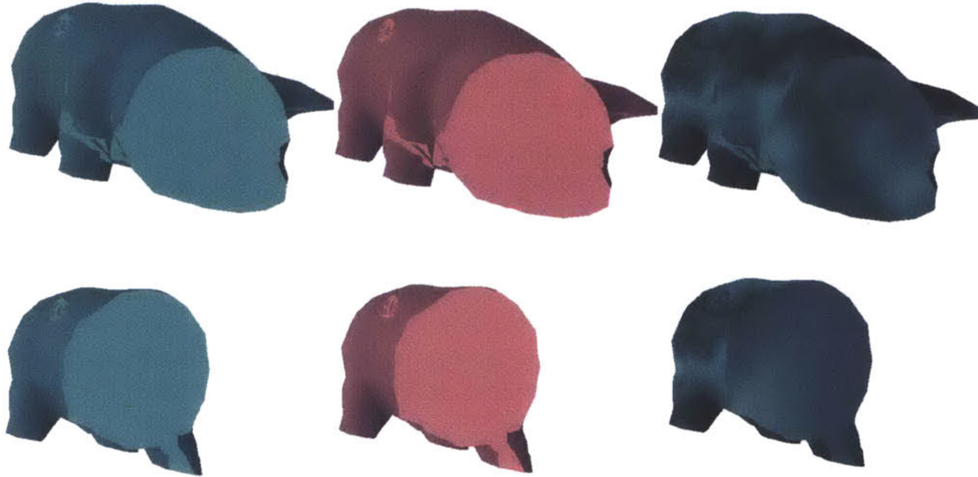


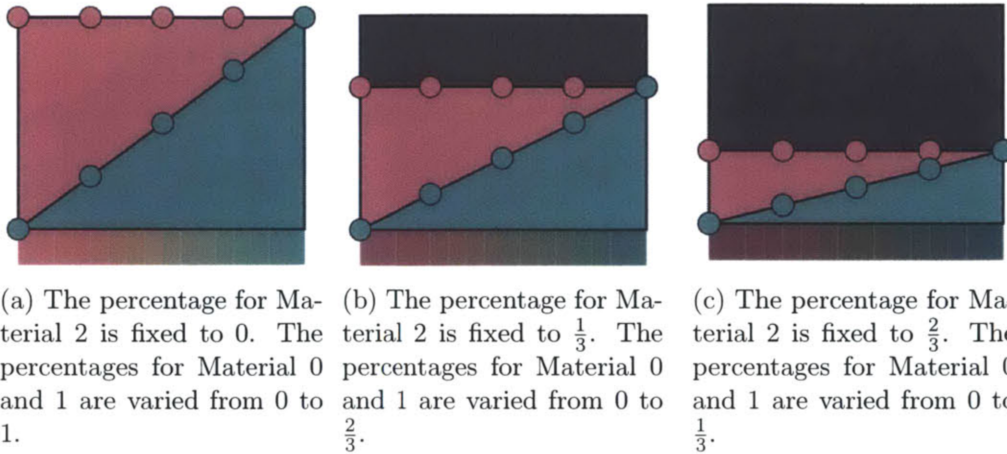
Figure 4-2: Planar-cut view for the pig model. The first row has step 4, and the second row has step 10. From left to right, the material is `SingleMaterial` with base Material 0, `SingleMaterial` with base Material 1, and `Marble`. The artifact on the back of the pig is caused by non-consistent face normal directions in the input model.

## 4.2 Features

To provide better user experience, I add three features. Planar-cut view mode enables users to visualize the materials inside of the model volume by slicing through the model with a plane. I develop a color scheme based on Lab* color space to visualize material compositions. This scheme can be more intuitively understood by users. Last, to remove jagged edges, I add a FXAA post processing step.

### 4.2.1 Planar-cut View Mode

Planar-cut view mode allows users to view materials inside of a model. The mode is controlled by the slider on the right of the page. Users can drag the slider to slice through the model as a clipping plane perpendicular to the viewing angle in real time. See Figure 4-2 for screenshots of the planar-cut mode.



(a) The percentage for Material 2 is fixed to 0. The percentages for Material 0 and 1 are varied from 0 to 1.

(b) The percentage for Material 2 is fixed to $\frac{1}{3}$. The percentages for Material 0 and 1 are varied from 0 to $\frac{2}{3}$.

(c) The percentage for Material 2 is fixed to $\frac{2}{3}$. The percentages for Material 0 and 1 are varied from 0 to $\frac{1}{3}$.

Figure 4-3: Demonstration of material composition visualization in Lab* color space.

### 4.2.2 Material Composition Visualization in Lab* Color Space

Since human's eyes are not good at decomposing colors into red, blue and green, I use Lab* color space [6] to provide more intuitive color visualization for material compositions. See Figure 4-3 for an illustration of different material compositions visualized in Lab* color space.

Material 2 is visualized as L*, the lightness of the color. Material 0 is visualized with large b* and small a* as green, and Material 1 is visualized with large a* and small b*, as pink red. The higher the percentage Material 2 in the composition the darker the color is. The higher the percentage Material 0 in the composition, the greener and colder the color is. The higher the percentage Material 1 in the composition, the redder and warmer the color is. If Material 0 and 1 have the same percentage, the composition is visualized in a yellow color.

### 4.2.3 FXAA Post Processing

The rendering with WebGL gives jagged edges. See Figure 4-4 for an example. I use Fast Approximate Anti-Aliasing (FXAA) [8] to smooth out the jagged edges in real time. FXAA is added as an additional rendering pass, and is fast to compute.

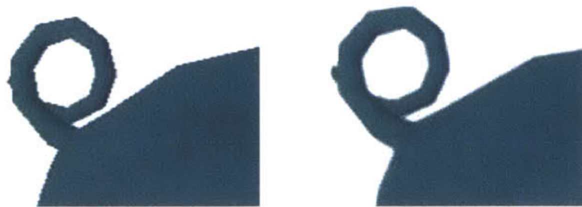


Figure 4-4: Before and after FXAA post processing on the rendering of the pig's tail.

# Chapter 5

# Implementation

This chapter explains implementation details of fab tree generation and evaluation, user interface, and the communication between the backend and frontend. I give an overview of the system architecture in 5.1. In that same subsection, I explain each modules from external dependency and library import in the backend to rendering in the frontend. In 5.2, I explain fab tree generation and evaluation in detail.

## 5.1 System

The frontend user interface is a webpage, and the fab tree backend is a browser plugin written with Netscape Plugin Application Programming Interface (NPAPI). The frontend and backend communicate through the `Plug-in API` layer. Figure 5-1 shows the system architecture.

### 5.1.1 External Dependency

External dependencies are marked in green in the diagram. I use THREE.js [2], a JavaScript library built on top of WebGL, for rendering. WebGL is a JavaScript API for rendering 2D and 3D interactive graphics within browsers. WebGL is developed based on OpenGL ES 2.0. JQuery UI is a common package for building user interface in a browser. It provides a rich set of widgets. Firebreath [1] is a library built on top

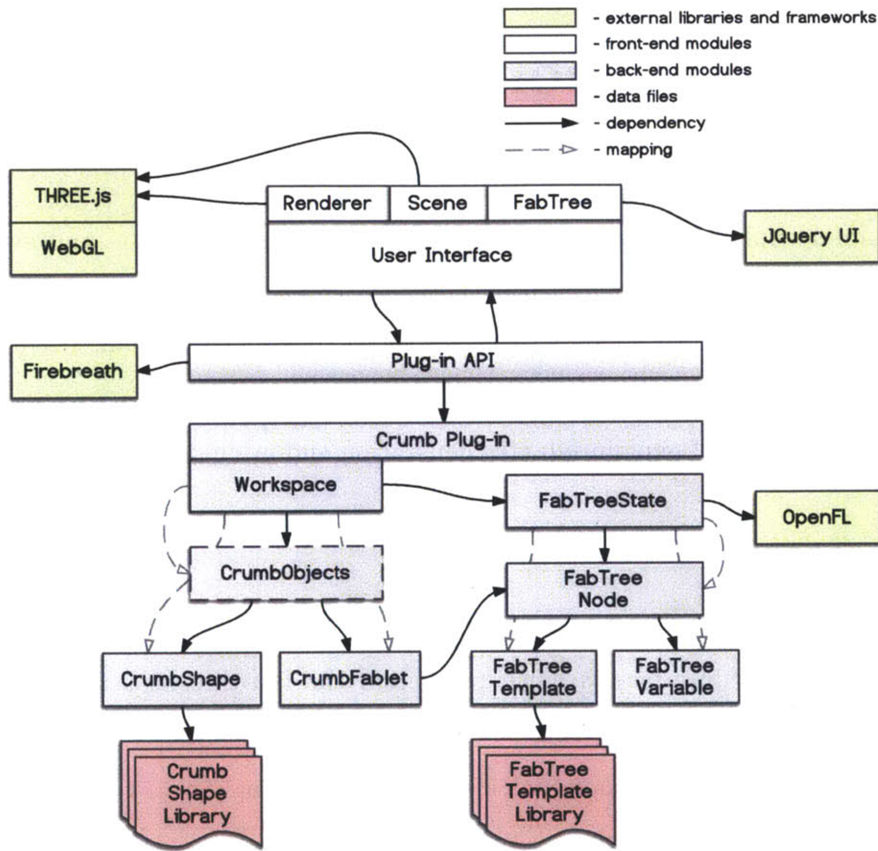



Figure 5-1: System architecture for Crumb.

of the NPAPI for building browser plugins.

### 5.1.2 Library Import

The frontend passes the location of model and fab tree libraries along with a callback function to the backend through `Plug-in API`. `Plug-in API` then starts a new thread for importing libraries. After finishing importing, the callback is used to notify the frontend. This asynchronous design allows the frontend and backend jobs to run in parallel.

The library files are written in JSON. The backend reads in the libraries as `CrumbShapeLibrary` and `FabTreeTemplateLibrary`, marked in red in the diagram. For each shape in the library, a `CrumbShape` object is created, and assigned a unique ID. `Workspace` uses the ID to hash the object. Similarly, for each fab tree node, a `FabTreeNode` object is created. Each `FabTreeNode` has a unique ID, which is later used for generating fablets. `FabTreeState` keeps a hash table of all `FabTreeNode` objects, and compiled fablets. Initially only the default fab tree is compiled.

### 5.1.3 Model Geometry Rendering

`CrumbShape` contains vertices of the mesh, indices of the vertices on each triangular face, and texture coordinates for vertices on each face. When a query of the geometry is made from the frontend, the `Plug-in API` starts a new thread converting the queried `CrumbShape` object to a Boost VariantMap, that is later converted to a JavaScript object by Firebreath. When the thread is finished, the system uses the passed-in callback to to pass back the JavaScript object. The frontend then creates a THREE.Geometry for rendering.

### 5.1.4 Model Appearance Rendering

Selecting a fab tree from the fab tree list on the user interface triggers a call to the API for evaluating the appearance on the model surface. `Workspace` first finds the `CrumbObject` for the current model, and reassigns the selected `FabTreeNode` to

`CrumbFablet`. The system evaluates the fablet on the surface of the model, and generates a default 600-by-600 texture. The texture is encoded as a Base64 image, and passed back to the frontend. The user interface uses the image as the texture for rendering the mesh.

### 5.1.5 Planar-cut Mode Rendering

When users move the planar-cut slider on the right of the user interface, the user interface enters the planar-cut mode. The slider has 20 steps with each step of size $\frac{1}{20}$. Step 0 refers to the plane closest to camera while touching the bounding box of the model, and Step 1 refers to the plane farthest away from the camera.

The user interface first uses the value of the slider to calculate the depth of the plane perpendicular to the view angle. The user interface then generates a quad as the intersection of the plane and the bounding box of the model. The quad is assigned with texture coordinates, and passed to the backend. The backend evaluates the material composition on the quad, and returns a texture.

The user interface sets the camera nearest depth to the depth of the plane. It first renders the scene, and stores the scene in a buffer. It then renders the quad with the returned texture and stores that in a buffer. In the next rendering pass, it uses stencil buffer to clip the quad, so that only the part inside of the model is showing. The stencil is used to mask out the rendered quad, and the opposite stencil is used to mask out the rendered scene.

## 5.2 Fab Tree Generation and Evaluation

Fab trees are used to generate fablet material specifications. Fab trees are formed by fab tree nodes. Each fab tree node is defined by an OpenFL code snippet and a metafile. To generate a valid fablet, the system traverses the tree to concatenate the code snippets together in a meaningful way. The generated fablet file is then compiled, and evaluated in real time to generate rendered results on models.

### 5.2.1 OpenFL Snippets

There are two types of fab nodes - volume and mapping. The code snippets for mapping nodes include a collection of functions without any restrictions; however, for volume nodes, the system requires a function which takes a 3D position as input, and outputs a material composition.

The following is an example of the code snippet `single_material.fab` for the volume `SingleMaterial` fab node:

```
MaterialComposition SingleMaterial(double3 pos) {
    MaterialComposition mc;
    initializeMaterialComposition(mc);
    setMaterialQuantity(mc, material, 1);
    return mc;
}
```

`MaterialComposition` is an array indexed by material IDs. It stores the percentage for each material. `initializeMaterialComposition` initializes the whole array to 0. `setMaterialQuantity` assigns a particular material with a percentage.

`SingleMaterial` defines a uniform material with one base material. The returned material composition is 1 for the specified base material, `material`, and 0 otherwhere. The value for `material` is assigned in the metafile `single_material.json`, which is further explained in 5.2.2.

The following is the code snippet for `LinearGradient`.

```
MaterialComposition LinearGradient(double3 pos) {
    MaterialComposition mc;
    double t = linearMapping(pos);
    mc = materialA(pos) * t + materialB(pos) * (1 - t);
    return mc;
}
```

The function `LinearGradient` uses `linearMapping` to project the input position onto an 1D variable `t`. `t` is used as the weight to linearly combine material compositions of `materialA` and `materialB` at position `pos`. `materialA` and `materialB` are volume nodes here that take a position and return a material composition. The returned material compositions are treated as vectors with $k$ dimensions, while $k$ is the number of base materials.

### 5.2.2 Metafile Specification

There are five fields in each metafile.

- `name` defines the name of the fab tree node. It is later displayed in the user interface.
- `type` describes the type of the node. There are two supported types - volume and mapping. Volume nodes describe material compositions within a volume. Mapping nodes contain a collection of helper functions.
- `variables` is an array of global variables used in OpenFL code snippets. It binds values to variables. The supported types of variables include material, float, double, int, boolean, float3, and double3.
- `functions` is an array that specifies the names and filenames of functions in the code snippets. The first one in the list is considered as the main function.
- `nodes` is an array of fab tree node references used in the code snippets.

See the following metafile `single_material.json` for `SingleMaterial`.

```
{
    "name" : "SingleMaterial",
    "type" : "volume",
    "variables" : [
        {
            "name" : "material",
```

```
            "type" : "material",
            "value" : 0
        }
    ],
    "functions" : [
        {
            "name" : "SingleMaterial",
            "filename": "single_material.fab"
        }
    ],
    "nodes" : [
    ]
}
```

`SingleMaterial` is the most basic leaf node. See Figure 3-1 for an illustration of the fab tree node `SingleMaterial`. The code snippet does not make references to any other node; hence the `nodes` field in the metafile is empty. There is one variable, `material`, which is assigned with Material 0. There is one function, `SingleMaterial`, defined in file `single_material.fab`.

`LinearGradient` is an internal node. See Figure 3-13. `LinearGradient` makes references to three nodes: two volume nodes, `materialA` and `materialB`, and one mapping node, `linearMapping`. `materialA` is assigned with `SingleMaterial`. `materialB` is assigned with `SingleMaterial1`. `linearMapping` is assigned with `MapTo1D_x` that takes a position and returns its x-coordinate. `linear_gradient.json` is as below.

```
{
    "name" : "LinearGradient",
    "type" : "volume",
    "nodes": [
        {
            "name": "materialA",
```

```
                "type": "volume",
                "value": "SingleMaterial"
            },
            {
                "name": "materialB",
                "type": "volume",
                "value": "SingleMaterial1"
            },
            {
                "name": "linearMapping",
                "type": "map",
                "value": "MapTo1D_x"
            }
        ],
        "variables": [
        ],
        "functions": [
            {
                "name": "LinearGradient",
                "filename": "linear_gradient.fab"
            }
        ]
}
```

### 5.2.3 Fablet Concatenation

To generate a valid fablet, the system traverses the fab tree using references in the metafiles, and concatenates the code snippets in `.fab` files. OpenFL does not support structs or scopes by the time this work is done; therefore, the system generates an ID for each fab node to identify them. All function and variable names in each fab node is concatenated with the ID. See the generated fablet for `LinearGradient` as follows.

```
fablet LinearGradient {
   @uniform Material material_I2;
   @uniform Material material_I1;

   double MapTo1D_x_I3(double3 pos) {
       return clamp(pos[0], 0, 1);
   }

   MaterialComposition SingleMaterial_I2(double3 pos) {
       MaterialComposition mc;
       initializeMaterialComposition(mc);
       setMaterialQuantity(mc, material_I2, 1);
       return mc;
   }

   MaterialComposition SingleMaterial_I1(double3 pos) {
       MaterialComposition mc;
       initializeMaterialComposition(mc);
       setMaterialQuantity(mc, material_I1, 1);
       return mc;
   }

   MaterialComposition LinearGradient_I0(double3 pos) {
       MaterialComposition mc;
       double t = MapTo1D_x_I3(pos);
       t = clamp(t, 0, 1);
       mc = SingleMaterial_I1(pos) * t + SingleMaterial_I2(pos) * (1-t);
       return mc;
   }
```

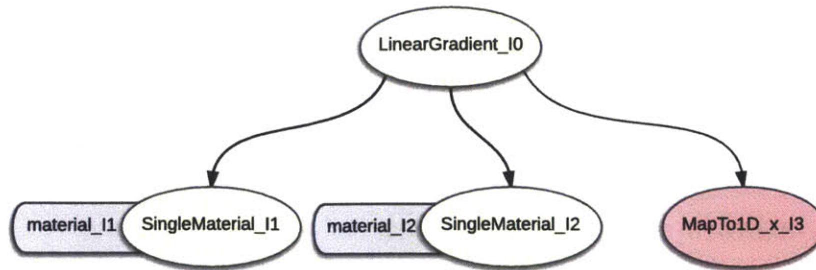



Figure 5-2: Fab tree for LinearGradient with generated IDs.

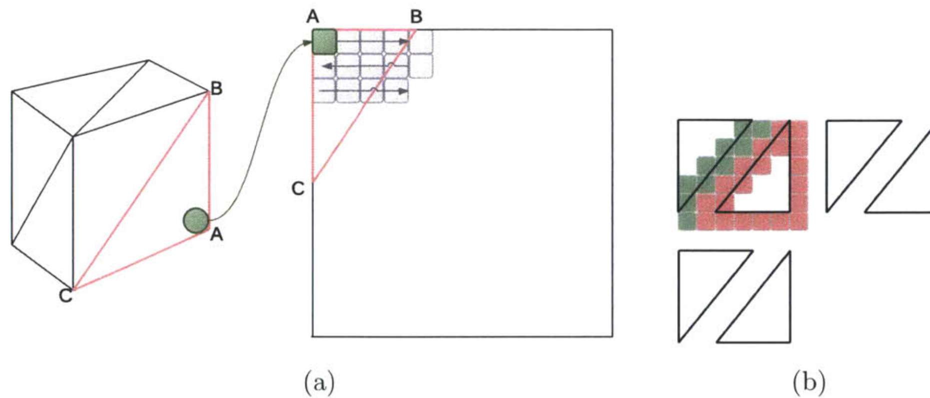
```
    @Surface(double3 pos, double3 normal) {
        return pos;
    }

    @Volume(double3 pos) {
        return LinearGradient_I0(pos);
    }
}
```

All functions and variables for `LinearGradient` are concatenated with ID `I0`. Its fab node references are substituted with the right function names; e.g. `linearGradient` is substituted with `MapTo1D_x_I3`, and `materialA` with `SingleMaterial_I1`.

See Figure 5-2 for an illustration of the `LinearGradient` fab tree with ID marked. The system first assigns `LinearGradient` with ID `I0`, `SingleMaterial` with `I1`, `SingleMaterial1` with `I2`, and `MapTo1D_x` with `I3`. Node reference `materialA` is replaced with `SingleMaterial_I1`, `materialB` with `SingleMaterial_I2`, and `linearMapping` with `MapTo1D_x_I3`.

### 5.2.4 Fablet Evaluation

The generated fablets are compiled with the OpenFL compiler to generate `.fabo` files. The system first binds values specified in metafiles to the uniform variables simultaneously. When a query at a given position is made, the system then binds




(a) (b)

Figure 5-3: Illustration for generating the texture for the surface of a cube model. The triangle being evaluated is highlighted in red both on the 3D cube and on the texture plane. The system checks all pixels in the bounding box of the triangle on the texture plane. and finds the corresponding 3D position in the 3D cube. The system queries the compiled fablet to get the material composition at the position, translates that composition to a color, and stores in the texture. On the right is a more detailed view of the generated texture with multiple triangles. Only the boundaries are marked with colors.

the position to the fablet's varying variable, and runs the compiled fablet to get the material composition at the queried position.

### 5.2.5 Texture Generation

A texture is generated for visualizing the material composition on the surface or for a cutting plane. For each triangular face on the surface of a mesh, the system samples in its texture coordinate space. See Figure 5-3a for an illustration. The system first finds the bounding box of the 2D triangle, and then samples 12 points to see if at least 2 of them fall in the triangle. If yes, the system uses the barycentric coordinates to find its corresponding 3D position, and queries the compiled fablet to get its material composition. The material composition then is translated to color using the color scheme explained in 4.2.2. The resulting texture is used for rendering.

# Chapter 6

# Results

In this chapter, I give an evaluation of the performance at each stage of the system. I also show some rendered results.

## 6.1 Performance

All experiments are run on MacBook Pro with 2.4 GHz Intel Core 2 Duo processor and 4 GB memory. The graphics card is NVIDIA GeForce 320M 256 MB. The system is Mac OS X 10.8.2, and the browser is Mozilla Firefox 22.0.

### 6.1.1 Geometry Importing and Conversion

I tested geometry importing and conversion performance on three models of different sizes. The *cube* model contains eight vertices and twelve faces. The *pig* model contains 606 vertices and 1208 faces. The *dinosaur* model contains 2309 vertices and 4587 faces.

I measure five operations in this stage. For these operations, I use three labels, *b*, *f*, and *bf*, where *b* stands for backend operations, *f* stands for frontend operations, and *bf* stands for operations both in the backend and frontend.

1. **Importing Model (b)** - reading the model from the local system in the backend;

| | Cube (12) | Pig (1208) | Dinosaur (4587) |
|---|---|---|---|
| Importing model (b) | 0.381 ms | 16.608 ms | 18.166 ms |
| Importing model (bf) | 5 ms | 7 ms | 24 ms |
| Conversion to VariantMap (b) | 0.979 ms | 72.448 ms | 276.889 ms |
| Conversion to JavaScript object (bf) | 102 ms | 3592 ms | 21726 ms |
| Creating THREE.js Geometry (f) | 3 ms | 12 ms | 80 ms |
| Total | 110 ms | 3612 ms | 21830 ms |

Table 6.1: Performance on importing model of different sizes. The number in the brackets behind the model names refers to the number of faces in the model. *b* stands backend, *f* stands for frontend, and *bf* stands for backend and frontend.

2. **Importing Model (bf)** - sending a request from the frontend to the backend to import a model, and sending back an acknowledgement from the backend to the frontend that the model has been read;

3. **Conversion to VariantMap (b)** - converting the imported model to Boost VariantMap (a format the Firebreath API requires for converting to an JavaScript object later);

4. **Conversion to JavaScript object (bf)** - sending a request from the frontend to the backend to get the geometry of the model, and sending back the geometry as a JavaScript object from the backend to the frontend;

5. **Creating THREE.js Geometry** - creating a THREE.js geometry object for rendering.

Operation 1 is part of Operation 2. Operation 3 is part of Operation 4. See Table 6.1 for the measured time of each operation on the three models.

The bottleneck in this stage is the conversion to a JavaScript object within Firebreath. This step constitutes 92.7% for small models, and more than 99% for large models. It takes 110 ms to import a cube model, and more than 21 seconds to import a dinosaur model with 4587 faces.

| | SingleM. | Marble | LinearG. | Foam | Tree |
|---|---|---|---|---|---|
| Importing | 0.309 ms | 0.264 ms | 0.213 ms | 0.333 ms | 0.326 ms |
| Generation | 0.425 ms | 0.416 ms | 0.287 ms | 0.302 ms | 0.314 ms |
| Compiling | 1879.99 ms | 2065.63 ms | 1773.2 ms | 1789.22 ms | 1870.93 ms |
| Evaluation | 3225.61 ms | 3829.29 ms | 3148.75 ms | 2998.45 ms | 3230.35 ms |
| Total | 5106.334 ms | 5895.6 ms | 4922.45 ms | 4788.305 ms | 5101.92 ms |

Table 6.2: Performance on importing and generating fab trees.

### 6.1.2 Fab Tree Importing and Generation

This stage is for reading in the fab tree specification and preparing it for generating the texture later. I test the performance on five different fab trees - `SingleMaterial`, `Marble`, `LinearGradient`, `Foam`, and `Tree`. These fab trees are explained in Figure 3-1, 3-4, 3-13, 3-8 and 3-12, respectively. `SingleMaterial` and `Marble` are trees of a single node. `LinearGradient`, `Foam` and `Tree` have two levels.

This stage breaks down to 4 operations:

1. **Importing** - importing a fab tree from JSON metafiles and fablet snippets in the local directory.

2. **Generation** - generating a compilable fablet for the fab tree by concatenating fablet snippets together.

3. **Compiling** - compiling the generated fablet into a .fabo format.

4. **Evaluation** - evaluating the .fabo format, and binding the initial uniform variables.

All these operations happen in the backend. See Table 6.2 for the measured time of each operation on the five selected fab trees.

Compiling and evaluating constitute more than 99% of the total operation time for all fab trees. Given that the complexity of fablets are similar, the operation time for compiling and evaluation between different models differs less than 0.2 ms. The total time for importing and generating fab trees is around 5 seconds.

| | SingleM. | Marble | LinearG. | Foam | Tree |
|---|---|---|---|---|---|
| Cube 200×200 | 411.768 ms | 486.119 ms | 441.451 ms | 403.872 ms | 517.182 ms |
| Cube 600×600 | 3527.6 ms | 4128.62 ms | 3785.18 ms | 3451.06 ms | 4407.26 ms |
| Pig 200×200 | 674.661 ms | 729.686 ms | 689.829 ms | 620.753 ms | 776.26 ms |
| Pig 600×600 | 3548.38 ms | 4132.28 ms | 3884.4 ms | 3476.04 ms | 4445.13 ms |
| Dino. 200×200 | 901.574 ms | 1043.42 ms | 975.041 ms | 891.488 ms | 1086.73 ms |
| Dino. 600×600 | 3642.34 ms | 4238.33 ms | 3980.31 ms | 3604.53 ms | 4531.56 ms |

Table 6.3: Performance on texture generation for different fab trees on different models with different texture sizes.

### 6.1.3 Texture Generation

For visualizing the material specification defined by a fab tree, I evaluate the fab tree on the surface of a model, and generate a texture image. I test the three models mentioned above with two different texture sizes, 200×200 and 600×600. *Cube* has 12 faces, *pig* 1208 faces and *dinosaur* 4587 faces. I use the same set of fab trees as above. The performance is shown in Table 6.3.

For the same fab tree on the same model, the time on generating textures is linear with respect to the texture size. The complexity of fab trees matters to the performance. For simple fab trees, such as `SingleMaterial` and `Foam`, it takes 3.6 seconds to generate a 600×600 texture for the dinosaur model, but around 4.5 seconds for more complicated fab tree such as `Tree`. The model sizes affect the performance within 0.25 seconds given the models I pick.

For the user interface, I use textures of size 600×600. The time spent on generating the texture is roughly 0.7 of that spent on importing and generating fab trees.

## 6.2 Rendered Results

I show a set of results generated with the system. Figure 6-1 shows a pig model with three different foam materials. The foam materials are defined as in Figure 3-8 and 3-18. The same foam materials are applied on a dinosaur model in Figure 6-2.

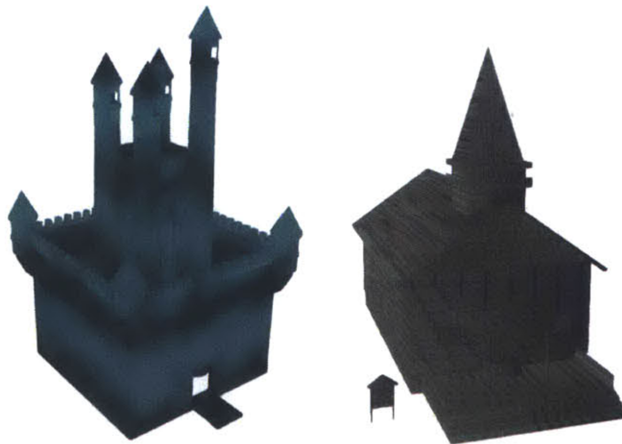
The parametrization method I provide can generate discontinuities along the edges of triangles on the surface for larger models. See Figure 6-2 for examples along the

(a) (b) (c)

Figure 6-1: The "pig" model with three different foam materials

(a) (b) (c)

Figure 6-2: The "dinosaur" model with three different foam materials.



(a) (b)

Figure 6-3: Buildings with a marble material and a grid material.

neck of the dinosaur. Since the sizes of triangles on the mesh does not correspond to the sizes on the texture plane, the resolution of the generated texture is limited by the largest triangle. This results as inconsistent resolution across the volume. In the future, the system should provide a better parametrization component for assigning texture coordinates.

Figure 6-3 shows two different building models with a marble texture, and a grid texture. The marble texture is defined as in Figure 3-4, and grid is defined as in Figure 1-1 but with `ConstantComposition` instead of `SingleMaterial`. The left building model has 2328 vertices and 4460 faces. The right building model has 246 vertices and 274 faces.

Aliasing artifact is present in the building example on the right. This is caused by only querying one 3D position for one pixel in the texture image. In the future, multiple sampling is needed for preserving high-frequency component in the texture.

# Chapter 7

# Conclusions and Future Work

In this thesis, I borrow inspiration from shade trees in rendering, and introduce fab trees to procedurally design materials for 3D printing purposes. I also provide an user interface for users to interactively specify and visualize materials on 3D models. In 7.1, I summarize the contributions. In 7.2, I discuss the limitations to my work, while outlining some of the possible future directions.

## 7.1 Conclusions

In this thesis, I provide a set of base nodes, including uniform materials and noise patterns, and internal nodes to combine the base nodes into a more complicated tree through designed structures, blending, and parameter interpolation. The volumetric material specification is represented in OpenFL, a language that procedurally defines material composition for any given 3D position in a volume.

In addition, I provide a web user interface to visualize created fab trees on any given 3D model in real time. When a fab tree is selected, the appearance on the surface of the model is evaluated in the backend and passed into the user interface as a texture. In order to view the material specification inside the volume, the user interface supports a planar-cut view mode. Users can drag a slider to slice through the model with the plane perpendicular to the view direction.

Compared to the current 3D printing software pipeline, the fab tree representation

gives users freedom to decouple material specification from geometry. They can reapply their designed materials on different models. Generating complicated materials is more systematic and approachable. The system allows users to fully utilize the capabilities of today's 3D printers.

## 7.2 Future Work

There are several limitations to the work presented in the thesis. First, the performance of the system is not optimal. For more complicated fab tree materials, the evaluation time for the generated fablets constitutes the performance. In the planar-cut mode, with the stencil buffer approach, the system evaluates many points that might not be inside the model. In the future, this could be improved by having the frontend generating the intersected planar geometries and passing them to the backend to get evaluated. To reduce evaluation delay, the system can cache 20 planes for the given view angle, thereby allowing users to slice through the model more seamlessly.

Second, I have only tried to build the system on Mac OS X as a Firefox plugin. Having more platform and browser supports can be future work.

Third, users can not share their designed materials easily. In the future, I can adapt the server-client architecture. Users can then build fab trees, and share them online with others. More domain- and usage-specific fab trees can be created.

Fourth, this work only supports designing materials in the volume space, and that are not geometry-aware. By the time this thesis is written, OpenFL also supports surface offsets and sign distance functions. In the future, I would like to include boundary-aware fab tree nodes and surface-texturing fab tree nodes. Surface offsets can be visualized with bump maps.

Last, the user interface does not support exporting the generated material specification to a print-ready format. In the future, the user interface can be connected to the Fabricator, part of the OpenFab project. So after users are content with their creation, they can print their models with their designed materials on a 3D printer

right away.

# Bibliography

[1] Richard Bateman. Firebreath. *URL: http://www.firebreath.org/*, 2013.

[2] Ricardo Cabello. Three.js. *URL: https://github. com/mrdoob/three.js*, 2013.

[3] Robert L. Cook. Shade trees. *SIGGRAPH Comput. Graph.*, 18(3):223–231, January 1984.

[4] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller, and Robert Jagnow. A procedural approach to authoring solid models. *ACM Trans. Graph.*, 21(3):302–311, July 2002.

[5] Stéphane Grabli, Emmanuel Turquin, Frédo Durand, and François X Sillion. Programmable rendering of line drawing from 3d scenes. *ACM Transactions on Graphics (TOG)*, 29(2):18, 2010.

[6] Richard S Hunter. Photoelectric color difference meter. *Josa*, 48(12):985–993, 1958.

[7] Todd Robert Jackson. *Analysis of functionally graded material object representation methods*. PhD thesis, Citeseer, 2000.

[8] Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, et al. Filtering approaches for real-time anti-aliasing. *ACM SIGGRAPH Courses*, 2(3):4, 2011.

[9] H Liu, T Maekawa, NM Patrikalakis, EM Sachs, and W Cho. Methods for feature-based design of heterogeneous solids. *Computer-Aided Design*, 36(12):1141–1159, 2004.

[10] Jorge Lopez-Moreno, Adrien Bousseau, Maneesh Agrawala, George Drettakis, et al. Depicting stylized materials with vector shade trees. *ACM Transactions on Graphics*, 32(4), 2013.

[11] Nico Pietroni, Paolo Cignoni, Miguel Otaduy, and Roberto Scopigno. Solid-texture synthesis: a survey. *Computer Graphics and Applications, IEEE*, 30(4):74–89, 2010.

[12] Kiril Vidimče, Szu-Po Wang, Jonathan Ragan-Kelley, and Wojciech Matusik. Openfab: A programmable pipeline for multi-material fabrication. *SIGGRAPH Comput. Graph.*, 32(4):223–231, January 2013.

[13] MY Zhou, JT Xi, and JQ Yan. Modeling and processing of functionally graded materials for rapid prototyping. *Journal of materials processing technology*, 146(3):396–402, 2004.