MIT Libraries | DSpace@MIT

MIT Open Access Articles

Foundry: Hierarchical Material Design for Multi-Material Fabrication

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Vidimce, Kiril, Alexandre Kaspar, Ye Wang, and Wojciech Matusik. “Foundry.” Proceedings of the 29th Annual Symposium on User Interface Software and Technology - UIST '16 (2016).

As Published: http://dx.doi.org/10.1145/2984511.2984516

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/112663

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

Foundry: Hierarchical Material Design for Multi-Material Fabrication

Kiril Vidimče Alexandre Kaspar Ye Wang Wojciech Matusik
Massachusetts Institute of Technology

Figure 1. Objects designed using Foundry. Left: a paddle with a perforated blade and a 'sweet' spot on its smooth side. Center: a simulation-driven ski design with retro-reflective surfaces. Right: a trycycle 'tweel' with lattice spokes for lateral strength and foam for improved suspension.

ABSTRACT
We demonstrate a new approach for designing functional material definitions for multi-material fabrication using our system called Foundry. Foundry provides an interactive and visual process for hierarchically designing spatially-varying material properties (e.g., appearance, mechanical, optical). The resulting meta-materials exhibit structure at the micro and macro level and can surpass the qualities of traditional composites. The material definitions are created by composing a set of *operators* into an *operator graph*. Each operator performs a volume decomposition operation, remaps space, or constructs and assigns a material composition. The operators are implemented using a domain-specific language for multi-material fabrication; users can easily extend the library by writing their own operators. Foundry can be used to build operator graphs that describe complex, parameterized, resolution-independent, and reusable material definitions. We also describe how to stage the evaluation of the final material definition which in conjunction with progressive refinement, allows for interactive material evaluation even for complex designs. We show sophisticated and functional parts designed with our system.

ACM Classification Keywords
I.3.5. Computer Graphics. Computational Geometry and Object Modeling. Curve, surface, solid, object representation. I.3.8. Computer Graphics. Applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
UIST 2016, October 16 - 19, 2016, Tokyo, Japan.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4189-9/16/10$15.00
DOI: http://dx.doi.org/10.1145/2984511.2984516

Author Keywords
fabrication; 3D printing; materials

INTRODUCTION
Multi-material 3D printing is an emerging technology for versatile prototyping and end-use part manufacturing. It enables fine material composition control at high resolution with feature sizes as low as 15 microns. Current multi-material 3D printers print with up to six distinct materials (e.g., Connex J750) out of a library of over twenty materials with diverse appearance, mechanical, optical and even bio-safe properties. The ability to vary the material composition at such precision supports numerous applications such as fabricating optical fibers [3, 28], lenticular prints [38], composite materials that maximize stiffness, strength and energy dissipation [42] or heat and electricity [39], biomimetic composite materials [24], actuated and deformable characters [34], colored and deformable characters for stop-motion animation [16], self-assembled objects [37] and functionally graded materials [26].

As compared to single material 3D printing where boundary representations are sufficient to describe the geometry and volume of the fabricated part, multi-material fabrication requires the user to specify the material composition over the entire volume at resolutions of up to 2,000 DPI. In its simplest form, a multi-material part could consist of a limited and discrete number of sub-parts each of which is assigned a single material. This is the approach taken by software that is bundled with existing commercial multi-material 3D printers.

By its very nature, multi-material printing of functional parts requires volumetric design of the part's underlying material composition. Furthermore, in order to push the boundaries of fabricated materials, we need to be able to specify structures at different spatial scales taking inspiration from materials that are found in nature. For example, an extremely tough and light

material of a human bone (Figure 2) exhibits different structures at different spatial scales that occupy the same volume. We believe that creating and reproducing these multi-scale structures will require hierarchical design tools.

Current modeling and material definition approaches are mostly driven by boundary representations used by traditional CAD and modeling packages. Tools such as Netfabb [25] or Materialise 3-Matic [21] can generate low-level geometry such as lattices or fine textures, but they cannot deal with hierarchical material composition or complex material assignments like gradients. On the other side, there are many volumetric modeling approaches described in the computer graphics community, but they are largely motivated by appearance-driven design and make appearance-driven decisions.

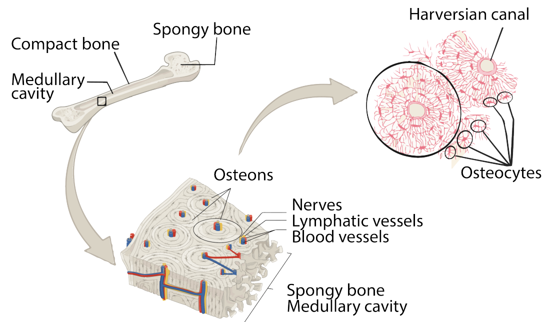


Figure 2. Biological materials such as the one made by the bone tissue can often be described in terms of a multi-scale hierarchy.

Building on the ideas described by OpenFab [41], we have developed Foundry – an interactive system for hierarchical authoring of spatially-varying, multi-material objects. Unlike OpenFab which is only suitable for experts, Foundry targets non-technical users by introducing the notion of hierarchical and modular material composition using a graph of operator nodes. This is a familiar representation reminiscent of shade trees [5] and image compositing tools [31]. In our authoring workflow, users start with an initial volume of the object defined by its boundary representation. They then use a visual interface to specify the material composition within the volume by building the operator graph (see Figure 3).

Foundry pushes the state-of-the-art along three axes simultaneously: (1) it enables non-technical users to create complex multi-material compositions, (2) it enables substantial increase in the complexity of multi-material designs as witnessed by our results and (3) it increases efficiencies in the design process by allowing users to build complex multi-material designs in minutes where the equivalent design would have taken many hours or even days with alternative tools. For instance, the initial design of the tweel shown in Figure 1 was created in less than an hour in Foundry; the same design would have taken a skilled programmer multiple days by using OpenFab.

We have used the system extensively to build a number of 3D printed objects and in conjunction created a core library of operators that we describe both in the paper and more exhaustively in the appendix. The operators can be hierarchically composed and perform one of three functions: decompose a volume into subvolumes, remap subvolumes to simplify the domain over which other (composed) operators will operate on, or construct and assign material compositions to subvolumes. Expert users can augment the library with new operators using a domain-specific language. The underlying graph-based representation is efficient (measured in KB) and similarly to OpenFab, printer and resolution-independent.

We achieve high performance and maintain interactive rates in Foundry by staging the execution of its OpenFab-like fabrication pipeline. Additionally, we use hierarchical (cache-evict) acceleration data structures to speed up global queries, and integrate progressive refinement and user-driven working-volume reduction techniques.

We evaluate Foundry by designing and fabricating several functional objects with hierarchical material structures and by conducting an informal case study.

In summary, Foundry is the first system designed specifically for interactive hierarchical multi-material design of functional parts at printer resolution. Our three key contributions are:

- A compact **operator graph** representation to visually, intuitively, and efficiently specify modular material compositions in a expressive and hierarchical manner.
- A description of our **operator library and classification** which enable a substantial increase in the complexity of achievable designs.
- An **interactive preview** system at native resolution of a high-resolution multi-material printer (15 microns) enabling complex material design and exploration.

RELATED WORK

Procedural Volume Modeling: Volumetric models can be efficiently created with procedural modeling. The material density can be computed using a function [11] or by combining simpler operators [29]. Cutler et al. [7] specifically address authoring volumetric multi-material models. However, their system does not provide an interactive design workflow and it focusses on creating virtual, non-fabricable models. Takayama et al. [36] propose an interactive system for authoring solid objects. Their representation is based on diffusion surfaces that require relatively sparse user input but adding detail is more difficult. Recently, Ijiri et al. [17] propose a semi-automated solid modeling system that fits geometric primitives to CT volume data. Multiscale vector volumes can describe volumes with complex internal structures using a binary tree of signed distance functions [43]. Foundry in comparison uses a more general volume decomposition approach. A recent system, Symvol by Uformia [40], allows for procedural modeling of microstructures in the context of single material 3D printing. An alternative approach is to use multi-phase implicit functions to model non-intersecting subvolumes as shown by Yuan et al [46]. Finally, Monolith [22] is a volumetric modeler that uses a stack of geometry and material assignment layers similar to image layers in imaging software. Monolith uses existing geometric models as constraints but generally assumes that the object is modeled via volumetric operations. In contrast, Foundry expects one or more boundary representations as an initial description of the object but then provides a powerful set of operators to further refine the geometric and material

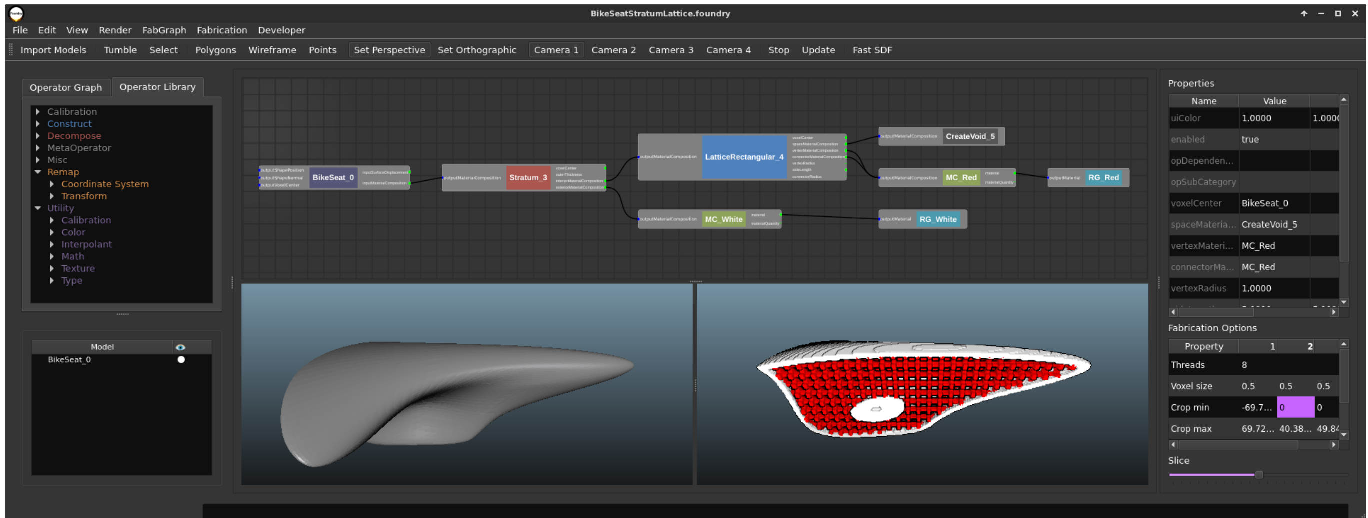


Figure 3. The Foundry UI: an operator browser (left), an operator property editor (right), a graph widget for manipulating the operator graph (top), and traditional 3D views and slice/volume preview panels (bottom).

representation. Unlike Monolith, our focus is entirely on multi-material design and thus the underlying representation, the workflow and our library of operators have been optimized for interactive multi-material design at printer resolution.

Data-driven Solid Texture Synthesis: Data-driven methods are an alternative to procedural volume modeling. 3D volumes can be synthesized from 2D or 3D texture exemplars. Ghazanfarpour and Dischler [12, 13] explore parametric approaches for solid texture synthesis. Jagnow et al. [18] analyze a 2D texture exemplar to synthesize a solid texture with a similar particle distribution. Wei [45] adapts a 2D neighbor-matching method for solid synthesis and Kopf et al. [19] improve it with a global optimization approach and an additional histogram-matching step. In lapped solid textures [35], a solid texture exemplar is applied to a tetrahedralization of an object. In addition, an authoring tool is developed that allows spatial variations and anisotropy. Dong et al. [9] extend this approach to support lazy evaluation, making the system more efficient. Another way to reduce storage requirements is by using vector solid textures [44]. Recent work tries to generate micro-structures: Dumas et al. [10] use exemplar-based synthesis to generate structural patterns. Both Panetta et al. [27] and Schumacher et al. [33] optimize the material micro-structure and its tiling to achieve elastic properties. Our system supports tiled solid textures in addition to procedural solid textures.

Material Specification for 3D Printing: Currently, multi-material objects are specified via separate boundary representation (an STL file) for each material. However, this specification method is very inefficient and limiting. The Additive Manufacturing File (AMF) Format [1] is a new standard that allows some proceduralism for direct specification of multi-material objects. The recent OpenFab [41] programming model and architecture provides an efficient and scalable way of directly specifying multi-material objects. Our system and design workflow builds upon this work. Finally, multi-material objects can also be defined by their desired functional characteristics rather than material composition. Spec2Fab [4] proposes the first attempt to build a functional specification pipeline. Spec2Fab's reducer tree exhibits a similar spatial decomposition approach to the one employed by Foundry.

Procedural Shading: Our research draws from both academic and commercial work on programmable rendering pipelines. Our underlying representation is inspired by Cook's Shade Trees [5]. A similar approach has been employed for 2D illustrations [20]. The system architecture and programming model have similarities to RenderMan [6] and shading languages [15]. Our authoring tool and workflow is partly inspired by the current commercial tools for shading design [2, 30]. There are, however, some key differences due to the fact that our system is geared for the design of volumes rather than surfaces. First, our representation makes distance query operators first class citizens. This also requires different acceleration data structures and staging of computation in order to make the system interactive. Finally, our design workflow requires tools for volume inspection and volume rendering.

FOUNDRY

We now present the key concepts and modules behind Foundry. We begin with an overview of the underlying framework, followed by the guiding principles employed when designing our system. We describe the typical user workflow for building a hierarchical operator graph that describes complex material definitions. We describe our library of operators, their classification and relation with real world materials. We discuss the importance of a rich set of distance queries that we used for our operators. We describe our execution model and how it can be staged to enable interactive design. Finally, in the results section we evaluate our system with multiple examples.

System Architecture Overview

Foundry was built on top of a programmable pipeline for multi-material fabrication similar to OpenFab [41]. The system that implements this pipeline is called a *fabricator*. To fabricate an

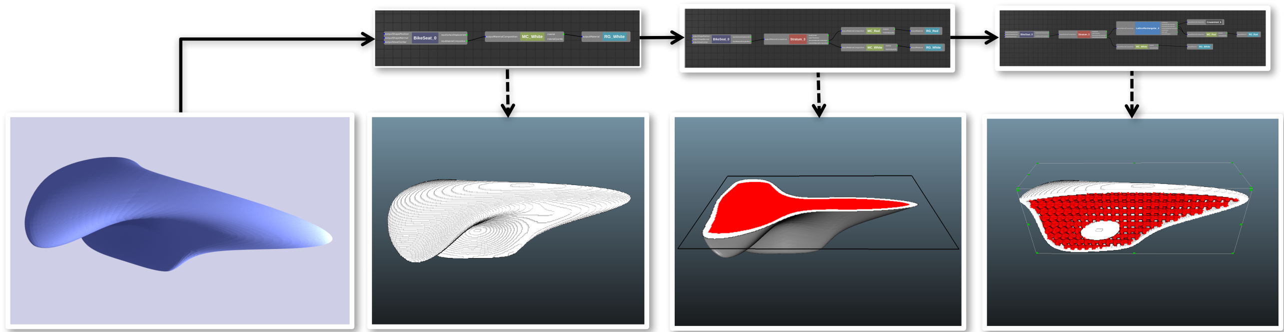


Figure 4. Sample session starts by importing a 3D model of a bike seat (left), then constructs and iterates on the operator graph (top row) to first produce a single material volume (first bottom). It uses a stratum operator to create two different colored subspaces (second column) inspected in slice preview (second bottom) and adds a lattice in the interior shell (third column). Final volume is inspected using the box widget (bottom right).

object, its initial volume is defined via a boundary representation. The object's surface is tessellated and then optionally displaced. The resulting volume is voxelized and for each voxel the material composition is determined. The surface displacement and volume material composition stages of the pipeline are fully programmable and exposed via *fablets* — programs written in OpenFL, a C-like, domain-specific language. OpenFL exposes a streaming, kernel-like programming model where conceptually, a single sample is evaluated at a time. Users implement the surface phase of the fablet which determines optional displacements. These displacements introduce minute, high-frequency surface textures that would be prohibitive to model via traditional modeling techniques. Additionally, the user implements the volume phase of the fablet which determines the material composition throughout the object's volume. To simplify defining complex, heterogeneous material compositions, the pipeline provides a key abstraction: users output arbitrary material compositions throughout the object's volume; the fabricator uses dithering to distribute the materials locally while achieving a single material per voxel and minimizing error. The final rasters are transmitted to a 3D printer via device-specific backends.

Design Principles

Our high-level goal was to build a system that enables quick exploration of the design space of multi-material fabrication. We achieve this with the following design principles:

Expressiveness: Hierarchical, heterogeneous material definitions can be very complex and designing them from scratch is time consuming. Our system introduces key operators that serve as building blocks and, when composed together, form complex material definitions. Some of these operators mimic materials that already exist while others are more general and can be used to hierarchically decompose the object's volume, optionally remap its coordinate system and then construct and assign material compositions for each subvolume. We describe these operator classifications in the next sections.

Extensibility: The operators in Foundry are implemented using OpenFL and thus, new operators can be easily developed, tested and refined. When developing new operators, the user can optionally implement a custom 3D widget via a plugin architecture exposed by the Foundry framework. Foundry will automatically detect, introspect and expose any new operators and associated 3D widgets to the user via its GUI.

Interactivity: Foundry was designed to provide an interactive experience. With each operator graph change, a meta-compiler converts the operator graph into an OpenFL-based description, thus forming a fablet. All operator property values are then bound and the fablet is JIT compiled to machine code. The fabricator can then evaluate the resulting fablet in a streaming fashion and provide a volumetric display of the material composition output. Initial results are provided quickly and then refined to the desired quality and printer resolution. We employ several strategies to optimize this process as described in the execution section.

Workflow

A modeling session starts by importing some existing geometry. The user then constructs the material composition by manipulating the operator graph (top section of Figure 3) while previewing and inspecting the resulting volume. As an example, Figure 4 follows a simple modeling session for a bike seat which starts with a single white material composition. We add a *stratum* operator to divide the structure into two shells. The external shell remains white and the internal one is assigned a new red material composition. We switch to the slice preview mode to visualize the internal volume. Finally, we add a *rectangular lattice* operator, set its filling composition to *void*, and use the red composition for its edges and vertices. The new operator is then connected to the internal shell, thus replacing the previous red composition, and we use the box widget to visualize the composition.

similar to Shade Trees [5]. Each node is a parameterized procedure with a set of input and output properties. Input properties have default values which the user can override via Foundry's interface. They can also be overridden by connecting them to output properties of other nodes.

Decompose, remap and assign: when building material definitions, a common pattern has emerged: the user typically uses a chain of decompose → remap → construct and assign material composition operators (see Figure 5). These *chains* of nodes are then nested and instanced in a hierarchical fashion.

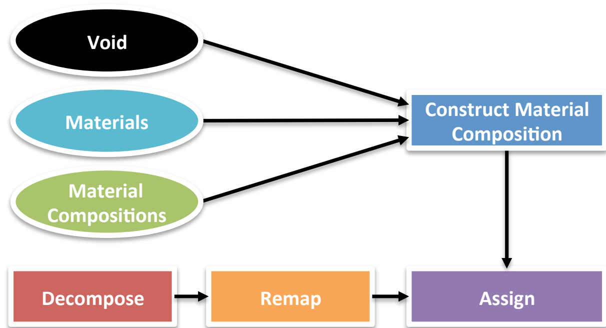
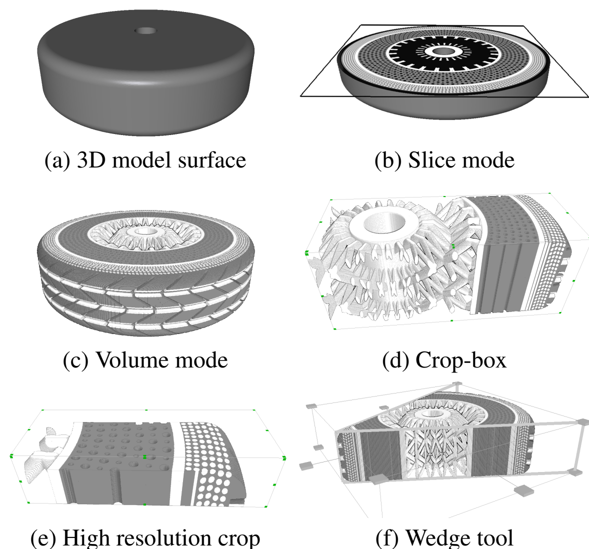


Figure 5. Typical operator work flow. The object's volume is decomposed into subvolumes. The space is optionally remapped. A material composition is constructed using void spaces, simple material-based compositions or complex compositions described using existing operators. The final material compositions are assigned to each subvolume.



(a) 3D model surface (b) Slice mode

(c) Volume mode (d) Crop-box

(e) High resolution crop (f) Wedge tool

Figure 6. Foundry can preview the material composition constructed from the initial geometric input shown in a). Slice and volume preview are shown in b) and c) respectively. The region of interest can be restricted as shown in d) and e). Once preview results are generated, they can be analyzed further, e.g., with the wedge tool shown in f).

Interactive exploration: the multi-scale nature of complex material compositions requires an equivalent multi-scale design exploration process. Users typically start with macro-structures and refine details iteratively, introducing new layers in the hierarchy. For volumetric exploration, we provide two preview modes: the user can either view individual *slices* and interactively browse through them with a slider or the whole voxelized volume can be explored using box and wedge manipulators that carve holes in the volume (therefore enabling the visualization of internal compositions) as illustrated in Figure 6.

The system ensures interactivity via progressive refinement or direct control of the preview resolution. The parameter space of operators can be sampled interactively with continuous sliders that automatically update the graph and produce direct visual feedback. Further implementations details are provided in the execution section.

OPERATORS AND MATERIALS

Based on our experience of designing multi-material functional parts, we propose and implement a set of over 100 different operators. The set provides a powerful basis for material design. We discuss their classification here. For a more detailed description, please see the supplementary material.

Operator classification

Users typically start by decomposing the volume into subspaces, then apply remappings to transform the coordinates and eventually assign a material composition to each of the regions of interest. Some of the operators in our library are specific to only one class of operations but often combine two or even all three functionalities into a single operator. For example, most of our decomposition operators also provide a way to directly assign a material composition to each subspace, and others remap and normalize each subspace to make further decomposition easier. Our library contains additional utility operators including mathematic operations, noise functions, texture manipulation and color conversions.

Decomposition Operators

Decomposition operators subdivide the initial volume into subvolumes (Figure 7, seats *a* to *d*). For instance, an operator may stratify the object by performing a signed distance function query from the object's surface. The volume can then be decomposed into a subvolume that is within a given distance of the surface and another that's further inside. This can be used to construct a coating of a given thickness that surrounds a core material. When implementing material definitions that construct several layers of materials (e.g., onion layers), these operators can be nested to decompose the volume into multiple stratifications. Another example decomposition operator is the tiling honeycomb operator which decomposes the space into individual truncated hexagonal prisms.

Distance Decomposition: We provide distance-based operators that can decompose the space based on a traditional distance function query from the surface. We also allow for full decoupling of the initial boundary representation used for the object and the distance queries; the distance queries can be made from associated (1) surfaces, (2) piecewise linear curves (e.g., skeletons), (3) splines or (4) points, the latter resulting in spherical decomposition.

Geometric Decomposition: Geometric operators decompose the space using simple geometric primitives to partition the volume into two subvolumes (inside vs. outside).

Lattice Decomposition: These operators subdivide the space into uniform grids and tilings. We can define these subdivisions both in R^2 (rectilinear, hexagonal, triangular) and in R^3 (hexahedral, tetrahedral). Different outputs are generated: (1) continuous subspaces for the cells with normalized coordinates and origin at the center of the cell, (2) discrete cell indices that identify each cell, and (3) regions of interest within the cells such as lattice edges, vertices or filling space.

Remapping Operators

Remap operators remap the underlying space into a different coordinate system. For instance, an operator that constructs a

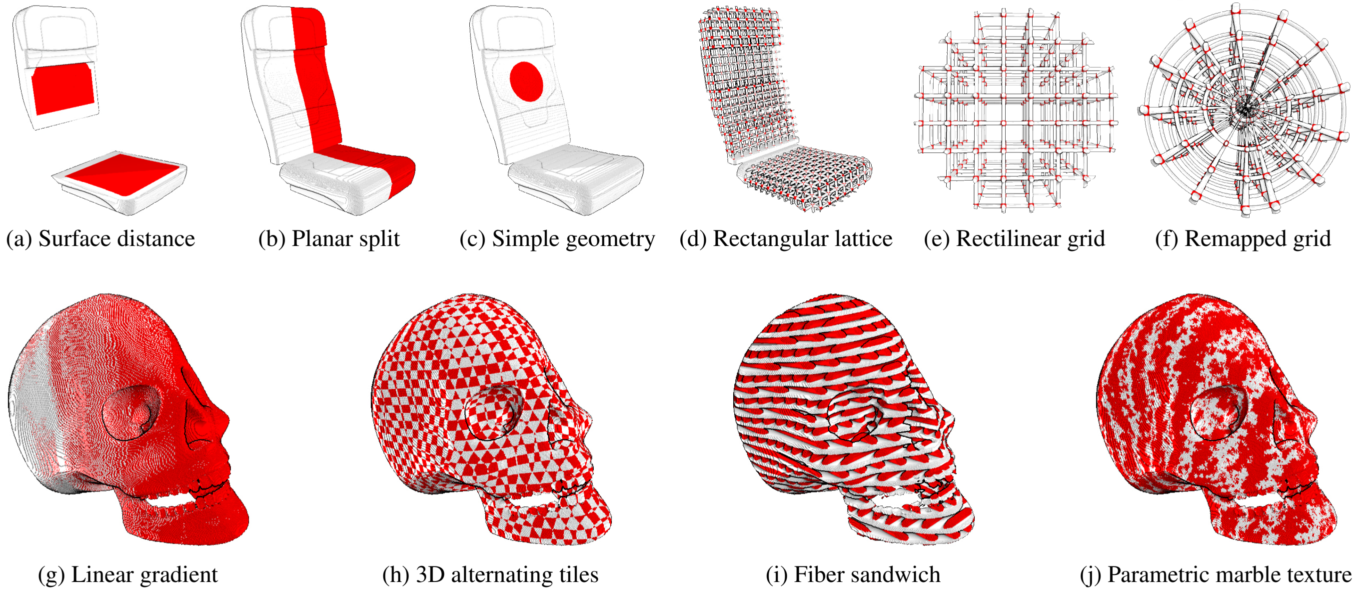


Figure 7. Illustration of various decomposition, remapping and construct/assign operators. The *decompositions* operators (a,b,c,d) use distinct color compositions to distinguish each subspace. The rectilinear grid of (e) is transformed into the cylindrical one of (f) using a single *remapping* operator. The *construct/assign* operators (g,h,i,j) display common material constructions from gradient (g) to composite (h,i), and even parametric textures (j).

classical hexagonal foam material may assume that it operates on a rectilinear grid. When using this operator on a cylindrical object such as a tire, the user may prefer to have the hexagonal pattern aligned with the tire curvature. By inserting a cylindrical to rectilinear remap operator into the graph, an existing foam operator can achieve this alignment. Similarly, when embedding fibers in the tire thread, in order to achieve cylindrical alignment of the fibers we can simply remap the space before connecting a traditional fiber composite operator. We have implemented operators that remap between Cartesian and non-Cartesian coordinates (spherical, cylindrical) or apply general linear transforms (see Figure 7 (e) and (f)).

Construct and Assign Operators

We provide operators that construct simple homogeneous material compositions from existing materials or null compositions. Other operators construct sophisticated hierarchical and heterogeneous material compositions such as patterns, traditional composites, cellular materials or various application-specific material compositions (Figure 7, skulls *g* to *j*). We leverage the abstraction exposed by the underlying system architecture where the material composition output from the operator can be an arbitrary and continuous linear combination of material and material quantity pairs. The back-end of the fabricator will distill the discrete material assignments for the fabrication device using a dithering-like scheme.

The final step is to assign the material composition to each decomposed volume. This is a conceptual step; in practice this is achieved by simply connecting the node that constructs the material compositions with the decompose or remap node.

Meta-Operators

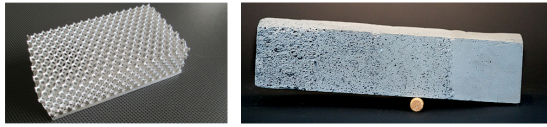
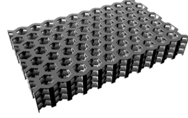
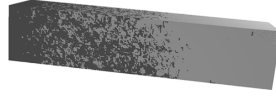
Non-technical users can create their own operators with no programming whatsoever. Once they construct an operator graph, a user can choose to convert any part of it into a meta-operator that is available for further instantiation and use. The user names the operator and selects a set of public parameters from the existing node inputs in the operator graph which are then exposed as part of the meta operator interface.

From Operators to Real Materials

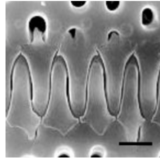
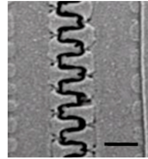
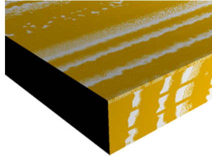
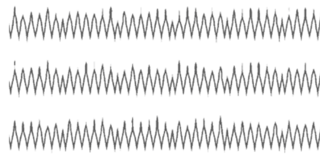
Given the extensive toolbox of operators, we provide examples and discuss how to use Foundry to build complex heterogeneous materials used in engineering and found in nature. We look at the common classes of materials and translate them to the operator representation.

Cellular Materials: These materials include materials with regular lattice topology or more irregular foam structures [14]. Many lattice truss structures have been explored (e.g., honeycomb, pyramidal, tetrahedral, 3D-Kagome), each providing different large-scale properties. The foam structures are typically classified into open-cell-structured foams (where pores are connected to each other) and closed-cell-foams. One of the main advantages of cellular materials is that they are light, since most of the space is not filled with any material. The properties of cellular materials are governed by the topology of the structure, the fraction of the cell occupied by material, and the properties of the constituent material. Cellular materials have many applications, including aerospace, automotive, and naval. There are already many cellular materials implemented as individual operators in our library: a variety of parameterized latices and parameterized foams. We show a typical engineering cellular material and a similar structure implemented in Foundry (Figure 8).

Composite Materials: These materials are made from two or more constituent materials: at least one matrix and one reinforcement material [8]. The reinforcement (e.g., fiber)

Cellular and functionally graded materials

Composite and biomimetic materials

Figure 8. Pairs of real, physical materials (top) and corresponding design in Foundry (bottom).

adds rigidity and the matrix (e.g., a polymer) surrounds it. When these two material types are combined, they potentially produce a material with different properties, such as improved strength or lighter weight. Some classic examples of composites include: concrete (loose stones held with a matrix of cement) and plywood. Fiber-reinforced composites have become extremely popular in objects that need to be strong yet lightweight. Foundry can easily realize composite materials either by combining basic operators or existing parameterized composites. We show an example of a plywood material and a corresponding structure implemented in Foundry (Figure 8).

Functionally Graded Materials: A Functionally Graded Material (FGM) is a composite where the composition and structure of the constituent materials can gradually change over the volume [23]. An FGM replaces the sharp interface of a traditional composite with a smooth transition. For example, the material composition can linearly change from material A on one side of the object to material B on the other side. FGMs can be used for applications that have extreme operating conditions, e.g., heat shields, armor plates, and GRIN lenses. FGMs are found in nature as well, e.g., teeth. FGMs are trivially implemented in Foundry using a composition operator or a variety of interpolation operators. We compare real FGMs and materials produced in Foundry in Figure 8.

Biomimetic Materials: These materials are developed by drawing inspiration from nature. Typically, spatial arrangements of the constituent materials are copied, while the constituent materials can be both copied or replaced. Examples of biomimetic materials include honeycomb structures, wood fibers, and bone structures. Some interesting applications of biomimetic materials include adhesive surfaces (mussels and geckos), self-cleaning, water-repellent surfaces (lotus leaves), extremely strong materials (marine mollusc teeth), structurally iridescent materials (butterfly wings), and velcro (burrs). We have experimented with using Foundry to represent biomimetic materials. Biomimetic materials are implemented as material composition operators; e.g., we have implemented a suture structure that mimics tough materials in turtle shells. However, biomimetic materials can also be implemented using an operator graph. We show an example bone we generated in the results section.

```
@name Decompose Rectangular Lattice 2D
@category Volume Decomposition
@subcategory Lattice and Foam
@description Decompose 3D space into a 2D lattice
operator DecomposeRectangularLattice2D {
  double Mod(double m, double n) {
    if (m > 0) return fmod(m, n);
    else if (m == 0.0) return 0.0;
    else return fmod(m, n) + abs(n);
  }

  @Evaluate(@input double3 voxelCenter,
            @input double2 cellSize = ...)
            @output double3 localCoords) {
    for (int i = 0; i < 2; ++i) {
      localCoords[i] = Mod(voxelCenter[i], cellSize[i]);
      localCoords[i] /= cellSize[i];
    }
    localCoords[2] = voxelCenter[2];
  }
}

@name Line Fibers
@category Construct Material Composition
@subcategory Composites
@description Fibers driven by bound skeleton
operator LineFibers {
  @Evaluate(@input double3 voxelCenter,
            @input int lineSetIndex = ...,
            @input double thickness = ...,
            @input MaterialComposition fiberMC,
            @input MaterialComposition fillMC,
            @output MaterialComposition outputMC) {
    double dist = lineDistance(lineSetIndex, voxelCenter);
    if (dist <= thickness) outputMC = fiberMC;
    else outputMC = fillMC;
  }
}

@name Construct Cylindrical Foam
@category Construct Material Composition
@subcategory Cellular Materials
@description Construct a cylindrical foam
operator ConstructCylindricalFoam {
  @Evaluate(@input double3 voxelCenter,
            @input double2 cellSize = ...,
            @input double cylinderRadius = ...,
            @input MaterialComposition cylinderMC,
            @input MaterialComposition fillMC,
            @output MaterialComposition outputMC) {
    double3 localCoords;
    DecomposeRectangularLattice2D decomp2D;
    decomp2D.Evaluate(voxelCenter, cellSize, localCoords);
    double3 origin = double3(0, 0, 0);
    if (distance(localCoords, origin) <= cylinderRadius)
      outputMC = cylinderMC;
    else
      outputMC = fillMC;
  }
}
```

Figure 9. Top: volume decomposition operator. Middle: construct material composition operator. Bottom: combined decompose/material composition operator. The code annotations are used to document the operator functionality and provide automatic operator search.

Example

We describe several example operator implementations. First, in Figure 9 (top) we show an example volume decomposition operator. This operator decomposes the given volume into a rectangular grid of 2D cells. The cell size is provided as an input. The coordinates in each cell are normalized to be unit sized. The third dimension (Z) is passed through, so, the resulting 2D cells are extruded along the Z axis.

Figure 9 (middle) shows the implementation of a simple operator that constructs a material composition by filling in fibers

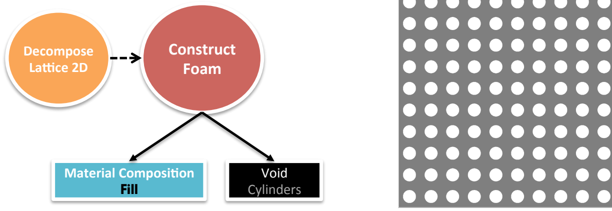


Figure 10. Example operator diagram and generated slice.

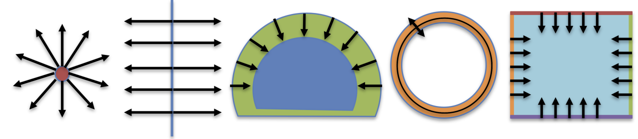


Figure 11. Distance function operators from left to right: a point distance (spherical decomposition), a line distance (skeletons), traditional signed distance function, a distance function from a coupled mesh (e.g., medial-axis), and a distance from discrete object surfaces.

with one material composition and the rest (the matrix or the fill) with another. The fibers in this case are defined as cylinders with thickness (or diameter) specified via an input property and their position is driven via a set of piecewise-linear curves. The curve set is defined as a separate geometric shape in the scenegraph and is bound and indexed to the object. Each object can be coupled with any number of shapes that can then be referenced when performing a distance query; hence the *lineSetIndex* parameter to the operator and the use of the *lineDistance* function call.

Since operators are implemented as self-contained objects in OpenFL, new operators can refer to existing operators directly in the code, making it even easier to reuse code and implement meta operators directly using OpenFL. Figure 9 (bottom) shows a sample meta operator — a cylindrical foam operator that constructs a more sophisticated material composition. The cylindrical foam operator tiles the space into cells using the existing decomposition operator. In each cell it then constructs a cylinder. The cylinder space is then filled with the cylinder material composition; the rest with the fill material composition. Figure 10 shows the simple operator graph and resulting material composition. To achieve an open-cell cylindrical foam, the user can instantiate a null material composition and assign it to the cylinder material composition input.

Distance Queries

Many of our operators rely on making distance queries. Unlike prior work [7, 46, 41] that uses distance function queries to stratify the object or to decompose it into arbitrary internal subvolumes, we identify other uses for distance functions, especially when used with ancillary geometry that is coupled with the object being printed (see Figure 11).

For instance, our fabric operators create minute patterns (woven loops, knits, grids) along the surface of a given object and are driven by a surface parameterization provided by the user. However, thin surfaces are problematic in this case; when querying the parameterization via a surface distance query, the resulting parameterization may come from either side of the thin surface. Creating a consistent parameterization that would be ideally symmetrical is infeasible. Instead, we calculate or embed a medial-axis surface within the fabric volume and drive all distance queries from it. Creating a parameterization for this surface is straightforward and is consistent across the entire volume of the fabric.

Another usage of surface distance functions is when making distance queries from a discrete set of surfaces associated with the fabricated object. Consider a ski; its surfaces consist of the top sheet, the bottom sheet and the side (edge) surface. If we were to implement an operator that creates a spatially-varying material composition near the top surface, using a generalized signed distance function would not be able to distinguish between being close to the top sheet vs. the bottom sheet. When performing surface distance queries against specified (indexed) object surfaces, this is possible.

Finally, consider building skeletons such as bones in a character. Instead of modeling each bone, one can construct a skeleton and then build a bone operator that constructs a complex spatially varying material composition by using piecewise-linear curve distance queries. The bone operator would use the distance value to determine the ratio of collagen-like flexible material and hard mineral-like rigid material. Similarly, linear distances can be used to strategically place fibers whose material compositions would be determined programatically.

EXECUTION ARCHITECTURE

Our execution engine design was inspired by the OpenFab software architecture. However, unlike OpenFab, Foundry's fabricator focuses on providing interactivity and thus requires (1) an API for communicating partial updates and (2) further staging of the execution pipeline in order to attain partial evaluation. The fabricator can operate in two different modes: slice preview and volume preview. Different evaluation strategies are employed in each mode. All fabrication is multi-threaded and performed in one or more background threads. The UI is thus kept "alive" at all times; fabrication can always be stopped or restarted with new user changes.

The Meta-Compiler

The first step during execution is to translate the operator graph into a corresponding fablet. Our meta-compiler performs depth-first traversal of the operator graph and follows all input-output connections. The necessary glue code is generated so that top-level inputs flow where necessary; we generate the necessary variables to carry output values from invoked operators that return values. The meta-compiler collects all operator parameters and builds a *bind map* that's used by the fabricator to bind operator inputs to the Foundry scenegraph.

Partial Fabrication

Similar to OpenFab, our fabricator has a streaming, fixed-memory pipeline. It splits the build volume into manageable chunks (slabs) and processes the volume one layer at a time. During the initial fabrication, the fabricator will *prefabricate* — prepare initial intermediate data. Subsequently, the fabricator is invoked in *refabricate* mode where the actual output is computed. Depending on which preview mode the fabricator operates in, it performs different work during the prefabricate and refabricate stages.

In **slice preview mode**, we assume that the entire slice can be stored in memory and prefabrication consists of object preparation work, tessellation, displacement evaluation of the surfaces that intersect the requested slice, and voxelization of the thin volume that occupies the given slice. The resulting grid of voxels and their associated fablets is cached; all other intermediate data is discarded. When refabricating, we evaluate the volume phase of fablets associated with the voxels in the slice and dither the final output. In order to allow extremely high resolution preview of large objects, the slice is progressively refined and the output streamed to the Foundry's viewer widget. Dithering typically is uninformative during these intermediate stages, so, we simply accumulate a linear combination of the material colors and corresponding quantity. Once the full resolution evaluation is completed, we perform a final dithering stage to accurately portray the resulting slice.

In **volume preview mode**, caching intermediate volume data would be storage-prohibitive; thus, refabrication is largely the same as offline fabrication. Large volumes can be fabricated, though; the output is streamed and carefully stored in an efficient manner (a few bits per voxel). Voxels occupied by the same material are grouped to allow for quick toggle of each material's visualization. All intermediate data is discarded after each slab computation. When a reduced working volume is specified, the slabs are computed against the crop box.

Common to both preview modes, we cache per-object data that gets generated during prefabrication. This includes bounds and coarse acceleration structures such as kd-trees for accelerating distance queries that get built with the input geometry. In order to implement a robust system, it's crucial to consistently define the user operations that lead to discarding of caches and re-execution of the prefabricate stage. Some of the dependencies that Foundry keeps track of are: operator connections and input values, texture file names, material assignment, visualization color of materials, etc.

Model	Paddle	Tweel	Bone
Triangle count	1,312	83,916	3,784
Voxel count (B)	3.9	3.0	15.8
Memory usage (GB)	2.3	1.9	8.5
Initial slice preview (s)	0.16	0.34	0.46
Final slice preview (s)	23.3	7.5	13.9
Naive fabrication (s)	49.4	14.39	14.5

Table 1. Memory usage and performance for previewing and final fabrication. For each model we show triangle count, voxel count in billions, memory usage (in GB) in interactive preview mode, time to first preview results in slice preview mode, time to final high-resolution results in slice preview mode and finally, the time it takes to fabricate a single slice of the model without staging the fabrication pipeline.

System Performance

Foundry was designed to operate under a constrained memory budget. The coarse kd-tree only uses single digit MB. A second-level, finer kd-tree is refined and cached in an LRU fashion while evaluating the volume and is constrained to no more than a few tens of MB. When previewing the material composition, we analyze the operators and determine the maximum number of materials (N) output at any voxel. We allocate N bits per voxel for each slab. The slab is dynamically sized based on N, the preview grid resolution and the target memory usage. The volume is sliced in multiple slabs and we evaluate the operator graph one slab at a time in a typical streaming fashion (voxels are grouped for efficient data-parallel evaluation). The final output is stored directly in video memory; assuming support for no more than 16 materials, we can represent each voxel with only 4 bits. The preview is rendered on the GPU using a group of vertex, geometry and pixels shaders that dynamically generate voxels with the correct color. Due to progressive refinement, Foundry is very responsive and initial results are always displayed in subseconds; the final, high-resolution preview can take multiple seconds but is entirely computed in a background thread while the UI is kept alive. The system memory usage and execution performance is shown in Table 1.

RESULT EXAMPLES

We used Foundry to design and fabricate several functional examples that demonstrate the versatility of the system and the ability to quickly build objects with complex material designs. We verify the feasibility of complex material design using our system with a user case study (see supplementary materials).

Examples

The examples were built by the authors and a novice user of Foundry. The results were then 3D printed using a Stratasys Connex 500 [32], a high-end multi-material 3D printer that uses a photopolymer phase-change inkjet printing process and can simultaneously print with two primary materials and one support material. We used several different materials: Vero Clear (a transparent, rigid material), VeroWhite+ (a white, rigid material) and TangoBlack+ (a black, rubber-like material). For each example we provide some background information and design goals. We then briefly explain how the example's material definition was constructed using Foundry.

Ping-Pong Paddle

We reproduced a paddle design with a smooth surface on one side and a rough surface on the other (Figure 1). The paddle was designed entirely with existing operators, from geometric primitives to perform volume decomposition to pattern operators to carve out the rough surface. Geometric operators carve out holes in the handle and perform further volume decomposition. The marble operator generates a marble-like solid texture on the handle and a wood operator emulates wood rings on the blade. We reduced the paddle weight with an internal lattice structure for the handle and the blade. Finally, interpolant operators dynamically adjust the ratio of flexible and rigid material to modify the sweet spot stiffness.

Ski

Modern skis have complex internal structures and are built from a wooden core with composites providing additional strength. Geometrically, the 3D model matches modern ski design (Figure 1) - it's parabolic and contains side cuts. The core is flexible, made of a rubber-like material and its diameter is determined in a data-driven fashion. We ran a stress-strain simulation and identified the stress on the skis when bearing weight from a 3-year old. We stored the output data in a texture and used it in a custom operator to convert the simulated

Figure 12. A 3D print of the bone designed in Foundry. Left: intricate hierarchical details of the osteons, bone marrow and spongy bone visible in a cross-cut of the print. Right: the full 3D printed bone with the coin showed for scale.

Figure 13. The tweel mounted on a toddler tricycle.

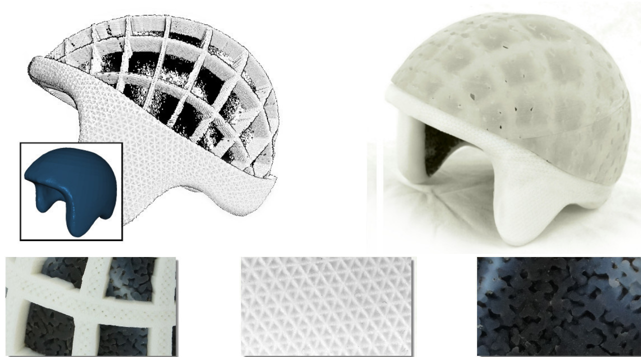


Figure 14. Top left inset: a boundary representation of the helmet. Top left: a voxel rendering of the helmet, excluding the transparent outer shell. Top Right: full printed model. Bottom: zoom on the lattice structure under the shell, the retroreflective pattern, and the brain-like layer.

stress-strain into the desired core diameter. To minimize ski torsion and provide rigidity, we used the *short fibers* operator to reinforce the rubber core with beams vertically and across the ski width, thus constructing a traditional composite.

The ski surface received additional treatment. We used the *honeycomb sandwich* operator to tile the ski tip with truncated prisms filled with transparent material. The top sheet of the ski is made retroreflective (to minimize glare) by carving tiny retroreflectors using the *corner retroreflector* operator. The edge of the ski is a combination of rigid clear material and rubber-like material to simulate edge bumpers. The bottom of the ski has special grooves that induce anisotropic friction – the ski cannot slide backwards.

Tweel

The tweel eschews the traditional pneumatic design and combines the tire, suspension and the wheel into a unified unit. We used Foundry to design a tweel from a rudimentary 3D model (shown in Figure 6) that models the basic torus-like structure of the tweel. The geometric details and the material composition were fully defined within Foundry.

Foundry was used to decompose the tweel into its logical components (spoke hub, spokes, foam, tire carcass, and tire thread). All but one of the operators used in the tweel's operator graph were already part of the Foundry library. A *radial stratum* operator was extensively used to decompose the volume of the tweel into its radially emanating subvolumes. The *micro-truss* operator was used to create the lattice for the spokes. The *cylinder foam* operator was used to create the sponge-like layer that provides suspension. One custom operator was written to model the sipes and the ribs in a parameterized way. The user can choose the number of each and the operator will automatically carve out the negative spaces and continuously lay the rigid material that lines up the ribs. The final 3D printed tweel was mounted on a toddler tricycle to test its ability to bear weight (see Figure 13).

Bone

Bones have a highly hierarchical structure. We built ours using a human right humerus from the BodyParts3D database. We reproduce key aspects of the bone structure: the outer shell, the compact bone layer containing the osteons, the spongy bone layer and the medullary cavity.

The operator graph uses a medial axis distance to drive the decomposition of the bone into its distinct layers ($d = 0$ on the medial axis, and $d = 1$ at the surface of the bone). The outer shell is modeled as a mix of dark gray and white materials. The compact bone is decomposed using a hexagonal grid to tightly pack cylindrical osteons that are further subdivided into multiple cylindrical layers up to their individual Harvesian canal. We used 3D Perlin noise at multiple occasions: (1) thresholded to generate the spongy bone, then (2) to warp the space and produce the non-uniform cylindrical layer separations of the bone and the osteons, and (3) to perturbate the surface of the bone ensuring that it mimics a real bone. Figure 12 shows the full print as well as a close-up of a slice.

Helmet

Helmet design must usually consider multiple constraints such as weight, aerodynamicity, aesthetic concern, and the protection they provide. Our design (shown in Figure 14) uses a cardboard inside the helmet shell to provide a more lightweight structure without compromising safety.

The composition uses two external surfaces: a planar surface separating the bottom part of the design from the upper part, and the outer surface of the volume used for the distance queries and texture coordinates. The outer shell is made of a transparent material displaying the internal design. The base has a retro-reflective surface. The core of the helmet is separated into the inner section using Perlin noise to mimick the appearance of the brain, and a lattice whose edges are further decomposed into sandwiches with a rectangular lattice filling similar to cardboard. Randomized holes on the outer shell provide ventilation.

CONCLUSION

We have developed Foundry – an interactive authoring system for designing materials and objects for multi-material 3D printing. It uses an extensible library of composable operators to create complex hierarchical material structures. While we have not performed a formal user study yet (primarily because there are no competing systems for multi-material design to compare against), we believe the design workflow is intuitive and easy to use. The operator graph does represent a principled representation for hierarchical material design and one can imagine even more sophisticated user workflows that ultimately map to the same operations we describe in this paper. The underlying system architecture achieves the necessary interactive performance. We have demonstrated system use by creating and fabricating a number of functional objects. We think that Foundry will be used for a wide range of applications as multi-material printing becomes a commodity.

For future work, we believe that coupling Foundry with a multi-physics simulator could be very beneficial. The simulation could inform users whether their design goals are met. However, simulating the behavior at full resolution will be very challenging requiring multi-scale modeling such as numerical coarsening. Foundry is currently a stand-alone tool but we envision integrating it into an existing modeling package. This will make Foundry better integrate into existing workflows, allow for an iterative design over both the geometry and the material definition and would let us leverage existing geometric tools in the target modeling package.

Acknowledgements

We would like to thank Desai Chen, Szu-Po Wang, and Alex Goins for contributing to the Foundry implementation. Caroline Morganti, Junda Huang, and Erica Lin helped build the operator library. Jacqueline Hung, Emma Steinhardt, Skyler Adams, Lars Johnson and Javier Ramos helped produce the results. Pitchaya Sitthi-Amorn, Shinjiro Sueda and Yahan Zhou helped print the results. We also thank David I.W. Levin and Frédo Durand for feedback on the manuscript. Tom Buehler helped edit the Foundry videos.

The 3D models and images of physical materials were obtained from TurboSquid, GrabCAD, and Wikimedia.

This work was funded in part by NSF grants 1138967, 1409310, and 1547088. Kiril Vidimče was supported by an NSF Graduate Research Fellowship and several contributing undergraduates were funded by the MIT Undergraduate Research Opportunities Program.

REFERENCES

1. ASTM Standard. Standard specification for additive manufacturing file format (AMF) version 1.1. July 2011.
2. Autodesk, Inc. Autodesk Maya 2013, 2013.
3. Brockmeyer, E., Poupyrev, I., and Hudson, S. Papillon: Designing curved display surfaces with printed optics. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13 (2013), 457–462.
4. Chen, D., Matusik, W., Sitthi-Amorn, P., Didyk, P., and Levin, D. Spec2Fab: A reducer-tuner model for translating specifications to 3D prints. *ACM Trans. Graph. 32*, 4 (July 2013).
5. Cook, R. L. Shade trees. In *Proc. SIGGRAPH* (1984), 223–231.
6. Cook, R. L., Carpenter, L., and Catmull, E. The Reyes image rendering architecture. In *Proc. SIGGRAPH* (1987), 95–102.
7. Cutler, B., Dorsey, J., McMillan, L., Müller, M., and Jagnow, R. A procedural approach to authoring solid models. In *Proc. SIGGRAPH* (2002), 302–311.
8. Daniel, I., and Ishai, O. *Engineering Mechanics of Composite Materials*. 1994.
9. Dong, Y., Lefebvre, S., Tong, X., and Drettakis, G. Lazy solid texture synthesis. In *Computer Graphics Forum* (2008).
10. Dumas, J., Lu, A., Lefebvre, S., Wu, J., München, T. U., Dick, C., and München, T. U. By-example synthesis of structurally sound patterns. *ACM Trans. Graph. 34*, 4 (July 2015), 137:1–137:12.
11. Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. *Texturing and Modeling: A Procedural Approach*, 3rd ed. 2002.
12. Ghazanfarpour, D., and Dischler, J.-M. Spectral analysis for automatic 3-d texture generation. *Computers & Graphics 19*, 3 (1995), 413–422.
13. Ghazanfarpour, D., and Dischler, J.-M. Generation of 3d texture using multiple 2d models analysis. *Comput. Graph. Forum 15*, 3 (1996), 311–323.
14. Gibson, L., and Ashby, M. *Cellular Solids: Structure and Properties*. 1997.
15. Hanrahan, P., and Lawson, J. A language for shading and lighting calculations. In *Proc. SIGGRAPH* (1990), 289–298.
16. Hayns, G., and McLean, B. The seamless fusion of stop-motion and visual effects technologies in laika's feature films. ACM SIGGRAPH 2013 Course Notes, July 2013.
17. Ijiri, T., Yoshizawa, S., Yokota, H., and Igarashi, T. Flower Modeling via X-ray Computed Tomography. *ACM Trans. Graph. 33*, 4 (2014), 48:1–48:10.

18. Jagnow, R., Dorsey, J., and Rushmeier, H. Stereological techniques for solid textures. *ACM Transactions on Graphics 23*, 3 (2004), 329–335.

19. Kopf, J., Fu, C.-W., Cohen-Or, D., Deussen, O., Lischinski, D., and Wong, T.-T. Solid texture synthesis from 2d exemplars. *ACM Transactions on Graphics 26*, 3 (2007), 2:1–2:9.

20. Lopez-Moreno, J., Popov, S., Bousseau, A., Agrawala, M., and Drettakis, G. Depicting stylized materials with vector shade trees. *ACM Trans. Graph. 32*, 4 (July 2013), 118:1–118:10.

21. Materialise. Software for additive manufacturing, 2013.

22. Michalatos, P., and Payne, A. O. Voxel-based modeling engine for multimaterial 3d printing, 2015.

23. Miyamoto, Y. *Functionally Graded Materials: Design, Processing and Applications*. Materials Technology Series. 1999.

24. Nair, A. K., Gautieri, A., Chang, S.-W., and Buehler, M. J. Molecular mechanics of mineralized collagen fibrils in bone. *Nature Communications 4* (Apr 2013).

25. Netfabb. The software for 3d printing, 2012.

26. Oxman, N., Keating, S., and Tsai, E. Functionally graded rapid prototyping. In *Proceedings of VRAP: Advanced Research in Virtual and Rapid Prototyping*, P. Bartolo, Ed. (2011).

27. Panetta, J., Zhou, Q., Malomo, L., Pietroni, N., Cignoni, P., and Zorin, D. Elastic textures for additive fabrication. *ACM Trans. on Graphics - Siggraph 2015 34*, 4 (aug 2015), 12.

28. Pereira, T., Matusik, W., and Rusinkiewicz, S. Fibers for sensing and display. *ACM Trans. Graph. 33*, 4 (July 2014), to appear.

29. Perlin, K., and Hoffert, E. M. Hypertexture. In *Proc. SIGGRAPH* (1989), 253–262.

30. Pixar. Slim™- the shader tool, 2012.

31. Porter, T., and Duff, T. Compositing digital images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, ACM (New York, NY, USA, 1984), 253–259.

32. Reisin, Z. B. Expanding applications and opportunities with PolyJet™rapid prototyping technology. Tech. rep., Objet, 2009.

33. Schumacher, C., Bickel, B., Rys, J., Marschner, S., Daraio, C., and Gross, M. Microstructures to control elasticity in 3d printing. *ACM Trans. Graph. 34*, 4 (July 2015), 136:1–136:13.

34. Skouras, M., Thomaszewski, B., Coros, S., Bickel, B., and Gross, M. Computational design of actuated deformable characters. *ACM Trans. Graph. 32*, 4 (July 2013), 82:1–82:10.

35. Takayama, K., Okabe, M., Ijiri, T., and Igarashi, T. Lapped solid textures: Filling a model with anisotropic textures. *ACM Transactions on Graphics 27*, 3 (2008), 1–9.

36. Takayama, K., Sorkine, O., Nealen, A., and Igarashi, T. Volumetric modeling with diffusion surfaces. *ACM Transactions on Graphics 29*, 6 (2010), 180:1–180:8.

37. Tibbits, S. 4d printing: Multi-material shape change: Mit 2013, 2013.

38. Tompkin, J., Heinzle, S., Kautz, J., and Matusik, W. Content-adaptive lenticular prints. *ACM Trans. Graph. 32*, 4 (July 2013), 133:1–133:10.

39. Torquato, S., Hyun, S., and Donev, A. Multifunctional composites: Optimizing microstructures for simultaneous transport of heat and electricity. *Phys. Rev. Lett. 89* (Dec 2002), 266601.

40. Uformia. Symvol for Rhino, 2014.

41. Vidimče, K., Wang, S.-P., Ragan-Kelley, J., and Matusik, W. OpenFab: A programmable pipeline for multi-material fabrication. *ACM Trans. Graph. 32*, 4 (July 2013), 136:1–136:12.

42. Wang, L., Lau, J., Thomas, E. L., and Boyce, M. C. Co-continuous composite materials for stiffness, strength, and energy dissipation. *Advanced Materials 23*, 13 (2011), 1524–9.

43. Wang, L., Yu, Y., Zhou, K., and Guo, B. Multiscale vector volumes. *ACM Trans. Graph. 30*, 6 (Dec. 2011), 167:1–167:8.

44. Wang, L., Zhou, K., Yu, Y., and Guo, B. Vector solid textures. *ACM Transactions on Graphics 29*, 4 (2010), 86:1–86:8.

45. Wei, L.-Y. *Texture synthesis by fixed neighborhood searching*. PhD thesis, Stanford University, 2002.

46. Yuan, Z., Yu, Y., and Wang, W. Object-space multiphase implicit functions. *ACM Trans. Graph. 31*, 4 (July 2012), 114:1–114:10.