OpenFab: A Programmable Pipeline for Multi-Material Fabrication

Kiril Vidimče Szu-Po Wang Jonathan Ragan-Kelley Wojciech Matusik

Massachusetts Institute of Technology

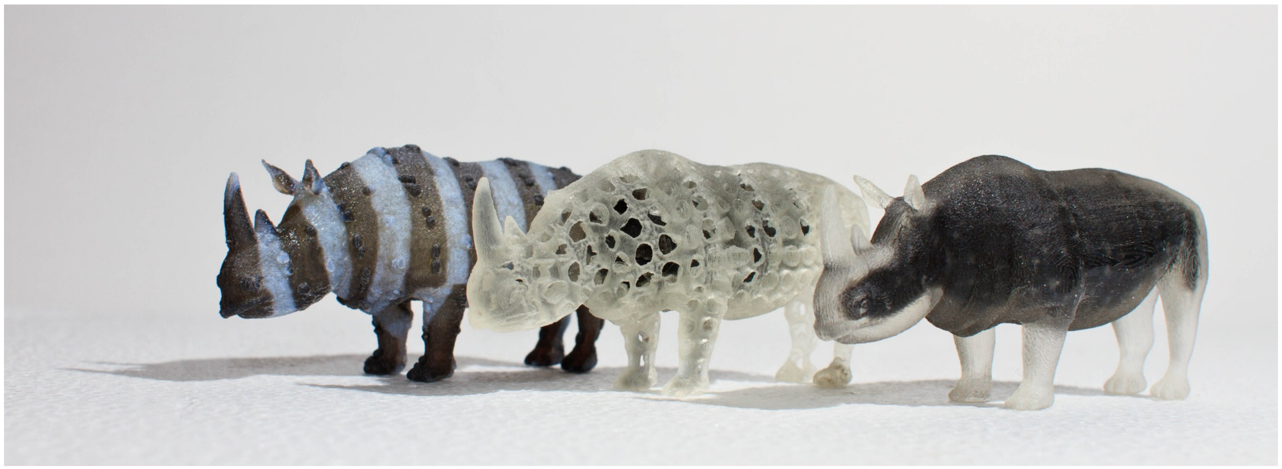

Figure 1: *Three rhinos, defined and printed using OpenFab. For each print, the same geometry was paired with a different* fablet*—a shader-like program which procedurally defines surface detail and material composition throughout the object volume. This produces three unique prints by using displacements, texture mapping, and continuous volumetric material variation as a function of distance from the surface.*

Abstract

3D printing hardware is rapidly scaling up to output continuous mixtures of multiple materials at increasing resolution over ever larger print volumes. This poses an enormous computational challenge: large high-resolution prints comprise trillions of voxels and petabytes of data and simply modeling and describing the input with spatially varying material mixtures at this scale is challenging. Existing 3D printing software is insufficient; in particular, most software is designed to support only a few million primitives, with discrete material choices per object. We present OpenFab, a programmable pipeline for synthesis of multi-material 3D printed objects that is inspired by RenderMan and modern GPU pipelines. The pipeline supports procedural evaluation of geometric detail and material composition, using shader-like *fablets*, allowing models to be specified easily and efficiently. We describe a streaming architecture for OpenFab; only a small fraction of the final volume is stored in memory and output is fed to the printer with little startup delay. We demonstrate it on a variety of multi-material objects.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representation I.3.8 [Computer Graphics]: Applications

Keywords: fabrication, 3D printing, API, materials

Links: DL PDF WEB

1 Introduction

State-of-the-art 3D printing hardware is capable of mixing many materials at up to 600 DPI resolution, using, for example, photopolymer phase-change inkjet technology. Each layer of the model is ultimately fed to the printer as a full-resolution bitmap where each "pixel" specifies a single material and all layers together define on the order of 10^8 voxels per cubic inch. This poses an enormous computational challenge as the resulting data is far too large to directly precompute and store; a single cubic foot at this resolution requires at least 10^{11} voxels, and terabytes of storage. Even for small objects, the computation, memory, and storage demands are large.

At the same time, it is challenging for users to specify continuous multi-material mixtures at high resolution. Current printer software is designed to process polygon meshes with a single material per object. This makes it impossible to provide a continuous gradation between multiple materials, an important capability of the underlying printer hardware that is essential to many advanced multi-material applications (e.g., [Wang et al. 2011]). Similarly, there is no support for decoupling material from geometry definition, and thus no ability to specify material templates that can be reused (e.g., repeating a pattern that defines a composite material, or defining a procedural gradation for functionally graded materials).

We think the right way to drive multi-material 3D printers is a programmable synthesis pipeline, akin to the rendering pipeline. Instead of a static mesh per piece of material, OpenFab describes a procedural method to synthesize the final voxels of material at full printer resolution, on demand. This provides efficient storage and communication, as well as resolution independence for different hardware and output contexts. It also decouples material definition from geometry. A domain-specific language and pipeline features specific to 3D printing make it much easier for users to specify many types of procedurally printed output than they could by writing standalone programs for every different material or fabrication application.

The OpenFab pipeline offers an expressive programming model for procedurally specifying the geometry and material of printable objects. A scene graph describes geometry and attributes, while *fablets* procedurally modify the geometry and define material composition much like shaders in the rendering pipeline. Fablets are written in a domain-specific language (OpenFL) and provide a flexible toolset that supports many common material specification tasks.

We also propose a scalable architecture for implementing the OpenFab pipeline. Since the total computational cost is large and it is impossible to fit the entire output volume into memory, the pipeline is designed to progressively stream output to the printer with minimal up-front precomputation and with only a small slab of the volume kept in memory at any one time. An OpenFL compiler analyzes and transforms the procedural computation described by the fablets as needed for efficient implementation in the fabrication pipeline.

We evaluate the system on a variety of multi-material 3D objects that have been specified and computed using our pipeline. We discuss how our system can be used to easily describe meta-materials, graded-materials, and objects that contain materials with varied appearance and deformation properties. We print a number of results using a commercial multi-material 3D printer and evaluate the performance of our prototype implementation.

2 Related Work

While the majority of current 3D printers use only a single material at a time, the emerging class of multi-material 3D printers (e.g., Objet Connex series [Objet ; Reisin 2009]) is capable of producing objects with almost arbitrary shape, deformation properties, and appearance by combining different materials at high resolution within a single object. Overall, there is substantial progress in the area of 3D printing hardware and use of multiple materials due to the efforts from industry, academia, and hobbyists.

Graphics and Printing APIs/File Formats: Traditionally, 3D printing has synthesized uniform material objects defined by unstructured surface meshes [3DSystems 1988]. Multiple materials are supported by statically assigning a single material to each mesh. Various companies have created proprietary formats to support their specific equipment. Nevertheless, with current printing software, it is unclear how the geometric data is translated to machine instructions, making the printing process difficult to control from outside. Open-source efforts (e.g., RepRap, Fab@Home) largely target fused deposition modeling (FDM) printing processes, which are motion vector-based, low-resolution, and low-throughput architectures with limited support for multiple materials (multiple materials are handled as separate STL files). The recent Additive Manufacturing File Format (AMF) standard [ASTMStandard 2011] allows description of object geometry, its composition and color. Colors and materials can be specified with limited proceduralism, using simple expressions from voxel coordinates to material choices. However, its per-voxel expressions have limited power, and no architecture has been proposed to efficiently implement it.

In contrast to the model-oriented descriptions supported by current 3D printing software, standard APIs and formats in 3D rendering and 2D printing describe how an output device should synthesize an image [Segal and Akeley 2012; Blythe 2006; Pixar 2005; Adobe Systems 1985; Hewlett-Packard 1984]. Our programmable pipeline model takes a similar approach. Our scene description parallels standard scene graph representations [Bell et al. 1995], with extensions specific to fabrication, and without many complexities necessary for animation and interactivity.

Goal-Based Material Design Recent work has pursued goal-based fabrication (e.g. [Bickel et al. 2010; Hašan et al. 2010; Weyrich et al. 2009; Bermano et al. 2012]). These methods allow specification of a certain goal, such as a desired deformation under a given force, and then automatically solve for the shape and material composition of the object. Chen et al. describe a generalized framework that helps with implementing new goal-based methods [Chen et al. 2013]. The framework consists of an API and novel data structures used for parameterizing the space of material assignments and for describing and controlling the optimization process. In contrast, OpenFab allows the user to directly specify and precisely manipulate the geometric and material properties of the printed output.

Scalable Graphics Architectures: Our design is inspired in part by RenderMan's Reyes architecture [Cook et al. 1987]. Reyes was designed to render models with extreme geometric detail and programmable shading. All geometric primitives are discretized into micropolygons which provide a uniform representation through the rest of the pipeline. Reyes manages complexity by processing a scene in image-space tiles with limited memory footprint. Tile-based deferred rendering pipelines [Molnar et al. 1994] make a similar design choice.

Programmable Rendering Pipelines: Rendering pipelines like RenderMan, OpenGL, and Direct3D [Cook et al. 1987; Segal and Akeley 2012; Blythe 2006] provide flexibility, simplicity, and performance by combining a fixed pipeline with user-programmable stages. Programmable shaders decouple geometry from material description. Our fabrication pipeline is inspired by the success of programmable rendering pipelines, and uses shader-like *fablets* to describe microgeometry and material composition. However, the motivation is different: rendering focuses on simulating images of 3D scenes with lighting and surface reflectance, while our pipeline uses the programmable stages to procedurally synthesize material and geometric samples for each layer of printing based on coarse model description. The visibility problems are also different in both domains.

Procedural Modeling: The procedural geometric and material modeling aspects of our pipeline are conceptually similar to previous work on procedural solid modeling by Cutler et al. [2002]. In their work the authors describe a scripting language that allows volume decomposition of solids into layers and procedural assignment of materials within each layer using embedded code written in C. Similarly to OpenFab, they provide signed distance function queries that can be used when evaluating the procedural material function. Unlike OpenFab, their system is strictly designed as a modeling and animation tool; we focus on fabrication and describe a scalable and streaming architecture that can evaluate the object specification on demand while the object is being printed.

Procedural Shading: Many languages have been defined for programmable rendering pipelines [Cook 1984; Hanrahan and Lawson 1990; Mark et al. 2003; Blythe 2006]. A fablet is similar to a programmable shader used in rendering. A shader procedurally defines the appearance of an object to be rendered in computer graphics; similarly, a fablet procedurally defines the material content of an object to be fabricated using an additive manufacturing process. Many optimizations and analyses developed for shader compilers are also important for fablets. Interval analysis for conservative bounding of computed values, which has been used for sampling and culling in traditional renderers [Heidrich et al. 1998; Hasselgren and Akenine-Möller 2007; Hasselgren et al. 2009; Clarberg et al. 2010], is useful for bounding surface displacements and adaptively sampling the material volume in fabrication.

Functionally Graded Materials: In material science and mechanical engineering, *functionally graded materials* (FGM) are heterogeneous materials whose material composition varies over the volume of a given object. Prior work describes the difficulty of modeling FGM objects, and proposes a variety of volumetric representations based on tetrahedra and voxels [Jackson 2000]. MIT's *three-dimensional printing* group describe a system that uses a signed distance field to represent geometry while the material composition is defined by a *composition function*. They also define 2D and 3D dithering methods that consider anisotropic properties of fluids when 3D printing with an inkjet printhead [Liu et al. 2004; Zhou et al. 2004].

3 Design Philosophy

Mixing many materials with different optical and mechanical characteristics at inkjet printer resolution allows extremely complex objects with countless unique and spatially varying properties to be synthesized directly from a digital description. At the same time, print volumes and speed are growing, while cost is falling, putting additive multi-material manufacturing within reach of more and more applications. These trends led us to several major principles which guided our design:

- **Continuous material definition.** To unlock the full capabilities of printer hardware, our system should allow continuous material definition at full printer resolution.
- **Streaming architecture.** In order to achieve scalability necessary for printing large build volumes at native resolution, the OpenFab pipeline should only use local storage and computation wherever possible, streaming over the output volume in the order required by the printer. It should also require as little up-front precomputation as possible, to minimize printer startup delay.
- **Procedural synthesis.** Expressive tools, especially a shader-like language and programming model, provide a more natural way to describe complex optical and mechanical material logic than is currently possible with static meshes per material. Procedural synthesis also supports scalability, trading memory for computation: the material composition and geometric detail does not have to be stored explicitly, but can be computed procedurally, as required by the printer.
- **Decoupling material from geometry.** Complex material logic should be defined independently of the mesh geometry, and be reusable across models.
- **Automatic adaptation to hardware.** Procedural synthesis of surface and volume detail provides resolution independence for different output sizes and resolution. Automatically normalizing and dithering multi-material mixtures, accounting for physical constraints like different materials expanding or contracting when cured, dramatically simplifies the development of device-independent procedural materials.

4 Programming Model

To meet these design goals, we propose a programmable pipeline abstraction for 3D printing. The role of the pipeline is to process a combination of geometric input, image textures, and *fablets* to synthesize device-specific fabrication output. The user controls the process by defining geometry and textures, setting pipeline attributes and options, and defining fablets. User-programmable fablets procedurally transform and compute attributes at each vertex of the object mesh, and compute the material mixture output at each point within the mesh volume.

Our pipeline has a number of logical stages, shown in Fig. 2. Similar to rendering pipelines, some of the stages are fixed and others are programmable by the user. The input to the pipeline is a *fab world*, a scene graph-like description that consists of object boundary representations and associated attributes such as transforms, image texture inputs and fablets.

In the first stage of the pipeline, the surface of the input objects is discretized via *tessellation*. Tessellation generates micropolygon primitives that constitute the common surface representation throughout the pipeline. Next, the *surface phase* of the fablet is evaluated for all micropolygons. This stage is programmable, has access to surface user attributes, and can reference external image textures. It can optionally displace the surface geometry.

The next stage discretizes the volume enclosed by objects via *voxelization*. The *volume phase* of the fablet is then evaluated over each voxel. This stage is also programmable and allows the user to access externally defined resources such as image textures and material definitions. Its output is a continuously defined mixture of material quantities. Final volumetric quantization and discretization of material quantities is performed in the *dither* stage. Finally, the device-specific output is produced via different back-ends.

We now describe our input, pipeline stages, and output in more detail, highlighting the key elements of our programming model.

4.1 Input specification

The fab world input is akin to an input specification to a renderer and can be specified via either a C++ API or an accompanying file format. We give a brief description of each and highlight the important features specific to 3D printing.

Our API currently supports the definition of geometry in the form of closed triangle and quadrilateral-based shapes. The shapes provide a boundary representation of the volume of the object being printed. Fablets are written in OpenFL and provide surface and material definition. Both the shape representation and the fablets can be reused across different printable objects. Each printable object couples a geometric shape with a fablet and accompanying data bindings. Complex objects such as a mechanical assembly may contain numerous instances of the same geometric shape (e.g., a bolt or a gear) and thus, we provide an ability to uniquely identify shapes and instance and transform them.

Given the ability to use the surface phase of the fablet to fine tune the geometric details at the surface level, the interface between two objects that are in contact can be very hard to define from a strictly geometric point of view (e.g., a procedurally displaced object embedded inside another object). Thus, we allow for the specification of object *priorities* defined as an integer value. If two or more objects end up populating the same voxel either by design or as a result of a displacement, OpenFab gives priority to the object with a higher priority value. This effectively allows constructive solid geometry (CSG) operations such as union and difference, but not intersection and is similar to the *precedence* operator described by Cutler et al. [2002].

4.2 Tessellation

The tessellation stage reduces the geometry input to micropolygons, a common internal surface representation throughout the pipeline. The tessellator uses the desired output resolution to produce micropolygons that match the target printer resolution. The tessellator also interpolates user-defined attributes and makes them available to the later stages in the pipeline.

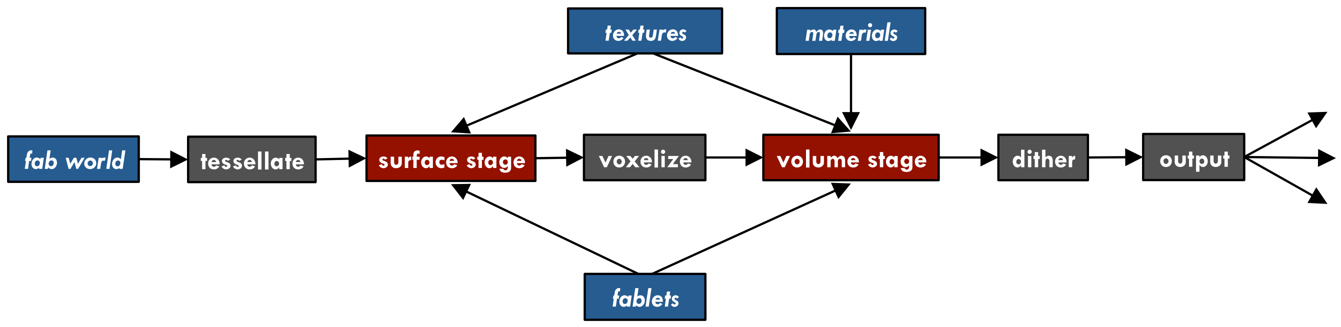

Figure 2: *The OpenFab pipeline defines a programming model for synthesizing continuous volumetric material mixtures for 3D printing. As an input (blue) it takes a scene graph describing a set of object boundary representations, textures, printer materials, and user-programmable* fablets*—similar to shaders. From this input, it generates a discrete volumetric material definition that is device specific. Some stages are fixed-function (grey), controlled by high-level parameters and printer characteristics, while* fablet *stages (red) are programmable by the user.*

4.3 Surface phase

The surface phase of the fablet is evaluated over the surface of the printable object. Conceptually, the surface fablet phase is evaluated point-wise, similarly to vertex shaders [Mark et al. 2003]. The fablet is given the vertex location and the normal as an input. The output consists of a list of user-defined attributes and the the displacement of the vertex. The additional surface user attributes will later be consumed by the volume phase of the fablet. The procedural displacement allows for increased geometric detail and can be an especially powerful mechanism for describing surface microgeometry that would be unfeasible to explicitly specify in the input. The fablet also has access to image textures which allow texture-driven procedural effects. Image textures are explicitly defined as an input to a particular fablet binding. This allows us to precisely track data dependencies and perform certain optimizations such as automatic creation of min-max textures for interval analysis.

4.4 Voxelize

The voxelization stage discretizes the volume enclosed by the tessellated and optionally displaced geometry. In order to voxelize objects in a consistent fashion, one has to define rules for determining whether a given voxel is inside or outside at the object's boundaries. Consider a multi-part assembly where parts are printed separately. To ensure the assembly fits together, one must follow consistent rules for defining the part boundaries. We follow the rules of 26-separating voxelization [Cohen-Or and Kaufman 1995]. Alternative rules can be used as long as they are applied consistently. Other discretization approaches that result in a different internal volumetric representations can also be used; examples include tetrahedral meshes or adaptively sampled distance fields [Frisken et al. 2000].

4.5 Volume phase

The volume phase of the fablet is evaluated over the volume of its corresponding printable object. The goal is to assign material mixtures to all voxels inside the object. This is the part of the pipeline that allows for a procedural material definition and makes it feasible to construct heterogeneous materials at the resolution of the printer. Each available material is given a globally unique id as part of the input specification. Similarly to image textures, all materials that the fablet will reference are explicitly defined as part of the object-fablet binding. The input to the volume phase consists of the voxel center and size. The output is a list of material-quantity pairs and is normalized to completely fill the voxel volume. If the output is empty, the voxel is marked as void.

When defining materials volumetrically, it is often useful to be able to determine the relative position of a given voxel with respect to the object boundary [Liu et al. 2004]. Consider the scenario where we would like to print a textured object. Unlike rendering, we cannot assign colors simply to the outer layer of the surface. In order to achieve a particular color, reflectance, and scattering behavior the printer needs to deposit a certain amount of layered material to achieve the desired appearance properties. Thus, one of the key features provided to the volume phase of the fablet is the ability to query the distance to the nearest point on the surface. Similarly, the user can query any user-defined surface attribute or any values generated by the surface phase at the same point.

4.6 Dither

The output of the prior stage consists of a mixture of materials for each volume element. However, 3D printers typically are only capable of depositing a single type of material at a given point. Therefore, we have to transform this description of the material mixture such that each voxel receives a single material assignment. This is similar to two dimensional dithering performed for color 2D printing; the key difference is that the number of materials is potentially much higher and the dithering should ideally be performed in 3D.

4.7 Output

The final output of our pipeline is device-dependent and targets a specific 3D printer. Different back-ends can be implemented. We currently implement a streaming raster slice format that's appropriate for a drop-on-demand 3D printer. We can also generate per-material geometric meshes that can be used with existing commercial software. Due to the size of the output, we are limited to printing small build volumes on commercial printers; this can be remedied by being able to directly provide the raster slices to the printer.

4.8 Discussion

The OpenFab pipeline bears strong resemblance to a modern programmable rendering pipeline. This is not surprising since both process similarly defined 3D input datasets. We note here the key differences between them and what makes our pipeline distinct.

Volumetric pipeline 3D printing is a process that produces physical 3D objects rather than images. Thus, our pipeline's nature is fundamentally *volumetric* and has to be able to generate and process orders of magnitude more data.

Wide target range Rendering creates images that target a wide range of displays of disparate sizes, ranging from personal mobile devices, to large format TVs, to projection screens. However, when taking resolution into account, rendering across all of these devices have remarkably similar requirements. In contrast, the range of sizes and resolutions of objects that one can 3D print is much wider. Recent work on nanoscale 3D printing has demonstrated the ability to print 3D objects at resolutions as high as 100 nanometers/voxel [Cicha et al. 2011] whereas very large format 3D printers exist whose build volumes are measured in hundreds of cubic feet. For instance, the build volume of the VoxelJet VX4000 occupies on the order of 100 trillion voxels [VoxelJet 2013].

Physical constraints The pipeline has to handle additional constraints imposed by the mechanics of the underlying printing process. First, most 3D printers print 2D layers sequentially along one of the world axes. This constrains the order in which the input specification needs to be interpreted, and the order in which the output needs to be written. Drop-on-demand 3D printers and traditional stereolitography both require support material to be placed underneath parts of the printed object that do not lie directly on top of previous physical layers. Our pipeline not only needs to be able to calculate the form of these support structures but it also needs to be able to instruct the printer to place them at the very beginning of the print process regardless of the eventual position of the part that relies on them.

Visibility Unlike rendering, our pipeline cannot take advantage of traditional visibility culling, except when calculating placement of support material. Object priority does impose ordering similarly to how depth imposes front-to-back ordering in rendering. However, since the priority is specified per object, we can pre-determine visibility even before fablet execution has occurred. Unlike 3D rendering, we perform no clipping and no projection. Finally, the pipeline has no concept of a viewpoint and thus cannot take advantage of any view-dependent techniques or representations.

Importance of dithering Recall that the output of the volume fablet phase is a fractional mixture of materials. Most 3D printers can only deposit one material at any given position of the volume. Thus, our pipeline needs to be able to transform the abstract representation of the output into something that the printer will be able to directly consume.

5 Fablets and OpenFL

Fablets are written in OpenFL, a C-like programming language. OpenFL is similar in most respects to shader languages like Cg and HLSL [Mark et al. 2003; Blythe 2006]. Unlike most shader languages, OpenFL describes both surface and volume functionality together, as methods on a single fablet object. Uniform parameters, including texture and material IDs, are also declared in the object. OpenFL includes a standard library with common math, texturing, and other routines. Unique to our domain, the standard library also includes functions to query the distance to the nearest point on the surface, as well as any interpolated mesh attributes at that point.

OpenFL is compiled by our LLVM-based fablet compiler [Lattner and Adve 2004]. Compilation is staged, much like HLSL in Direct3D: the first phase statically compiles fablet source into an intermediate representation which is saved to disk; at run-time, this intermediate representation is loaded and JIT compiled for its use in the pipeline, potentially with concrete parameters bound.

Using a domain-specific language provides opportunities to both analyze and transform the computation defining fablets. For example, our compiler generates interval versions of each fablet to facilitate automatic inference of displacement bounds or other runtime optimization. It is also designed to allow fast data parallel code generation, as used for real-time shaders, though parallel code generation remains future work.

5.1 Example

To understand how fablets can be used to define procedural surface detail and continuous volumetric material variation, consider the example in Fig. 7. One side is flat and texture-mapped with the foreground image, while the other side is displaced according to the desired brightness of the shadow background image. This object is defined by the following fablet:

```
fablet MagicPostcard {
  @uniform {
    float2 border;
    float textureDepth, maxThickness;
    ImageTexture2D fg, bg;
    Material white, black;
  }

  const int CARD_FRONT = 0, CARD_BACK = 1;

  @Surface(@varying {
             SurfaceAttributes attr,
             float2 uv, int face,
             out float2 uvOut, out int faceOut
           })
  {
    // pass through attributes
    uvOut = uv;
    faceOut = face;
    if (face == CARD_BACK) { // back face
      float L = bg.Sample1(uv[0], uv[1], 0);
      float thickness;
      if (uv[0] < border[0] || uv[0] > 1 - border[0] ||
          uv[1] < border[1] || uv[1] > 1 - border[1]) {
        thickness = maxThickness;
      } else {
        // material approximation: transmission
        // has quadratic falloff with thickness
        thickness = sqrt(1 - L) * maxThickness;
      }
      return attr.n * thickness;
    } else {
      // no displacement on the front and sides
      return 0;
    }
  }

  @Volume(@varying {
             VolumeAttributes attr,
             @nearest float2 uv,
             @nearest int face
           })
  {
    MaterialComposition mc;
    if (face == CARD_FRONT && // front face
        abs(distance(attr.voxelCenter)) <= textureDepth) {
      // surface texture
      float L = fg.Sample1(uv[0], uv[1], 0);
      mc.Set(white, L);
      mc.Set(black, 1 - L);
    } else {
      // background/border
      mc.Set(white, 1);
    }
    return mc;
  }
}
```

Material and texture handles are declared as attributes of the fablet, along with parameters for the dimensions of the rectangular border, maximum thickness as well as the depth into the volume to which the texture should be deposited on the front face.

The surface phase takes as arguments the position, normal, and texture coordinates defined over the mesh, as well as a per-vertex enum-like flag indicating the face of the cube (front, back, or side). If the currently processed vertex is on the back face, the fablet computes a material thickness based on the luminance of the background image and displaces the mesh accordingly. It creates a fixed-depth border in a narrow band around the edges defined by the `border` parameter. Outside the back face, it performs no displacement and simply returns the original vertex position.

The volume phase takes as its argument the 3D position of the center of the currently-processed voxel. It then uses the `face` flag from the *nearest* surface point to determine if it near the front face. If it is and the `distance` to the surface is within `textureDepth`, it samples the foreground image texture based on the nearest surface texture coordinates, and mixes black and white materials based on the brightness at that point. Note that the texture cannot simply be deposited in an infinitesimal layer on the surface. To show up clearly in real materials, it is usually necessary to deposit colors down from the surface to some depth inside the interior volume. Elsewhere in the object, it outputs plain white material.

6 Architecture

The OpenFab pipeline bears resemblance to Reyes and modern real time rendering APIs such as OpenGL [Segal and Akeley 2012] and Direct3D [Blythe 2006] (see Figure 3). The pipeline is designed to facilitate efficient implementation. Specifically, it is designed to allow a streaming implementation, starting to produce output quickly after startup, and driving the printer on-demand within a fixed and controllable memory footprint. Additionally, the fablet programming model is designed to admit massively data parallel computation, in the same style as shaders in rendering.

Our reference implementation was built to stream output with a fixed memory budget and low startup time. It is a scalable foundation for a high performance implementation, but many individual stages are internally unoptimized. Nonetheless, it is more than fast enough to keep up with currently-available printers.

The architecture of our implementation is shown in Fig. 3, and proceeds as follows:

```
Precompute acceleration structures
for each slab, in printer order:
    for each shape overlapping slab:
        Compute surface microgeometry and attributes
        Compute voxels and material composition
    Normalize and dither materials to device capability
    Output slab to printer
```

The individual stages are described in order in more detail below.

6.1 Pipeline Stages

Bounds: We begin by calculating bounds for each shape in the scene. Users provide maximum displacement bounds but we additionally use interval arithmetic to automatically infer those bounds as well [Clarberg et al. 2010]. To infer the maximum displacement, we execute an interval variant of the surface phase of the fablet bound to each shape. We pick the minimum of the user-provided and inferred bound.

Nearest query acceleration: Nearest surface point queries are expensive to compute on demand. We create acceleration structures to speed up the queries performed in the volume phase of the fablet. We build a bounding volume hierarchy (BVH) that spatially partitions the base primitives of the input mesh. We conservatively account for possible displacement using the displacement bounds calculated in the prior stage. We refine the BVH until each subvolume contains no more than a given target number of candidate primitives. This up-front process is fast, since it is performed on the untessellated base primitives of the input.

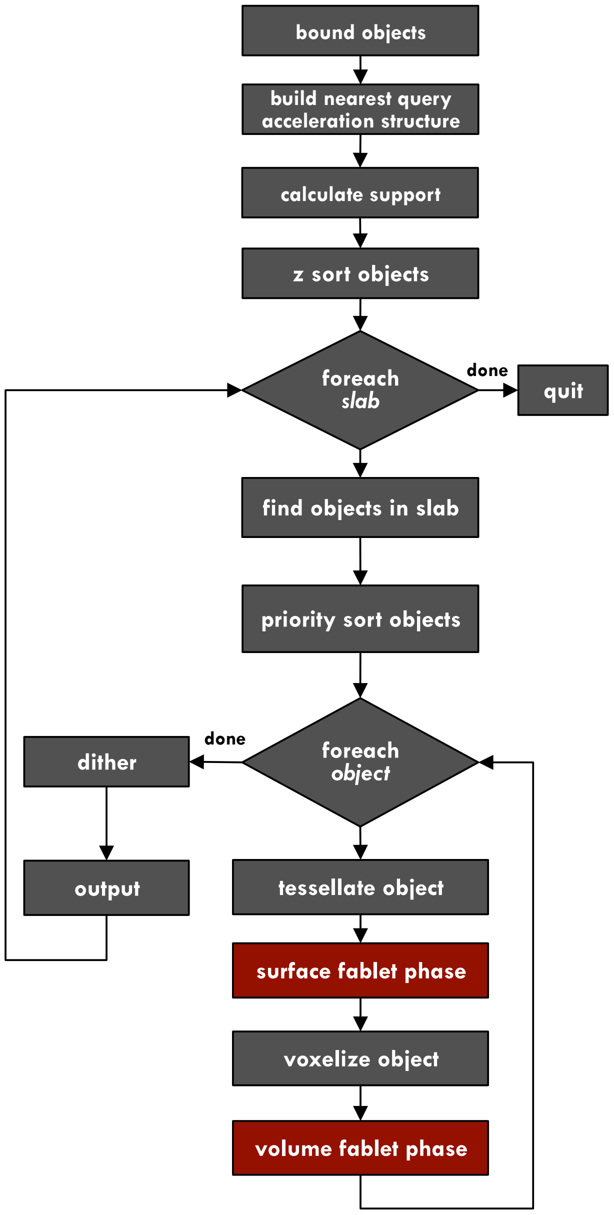

Figure 3: *The architecture of our OpenFab implementation is designed to stream over large, high-resolution print volumes with a fixed memory budget. The printing volume is divided into* slabs *along the primary printer axis, sized to bound memory usage. The pipeline processes one slab at a time and streams the output to the printer. Minimizing the amount of precomputation before streaming begins keeps startup time to a minimum, letting the printer start working almost immediately after OpenFab begins processing. Intermediate results like tessellated geometry that span slab boundaries are cached for reuse, and the caches are also set to a fixed maximum size.*

Calculate support: If the target printer requires support structures, we pre-calculate the places where such support is needed. We use a fast, high-resolution, fixed-point rasterizer to perform an orthographic render along the print platform movement axis (typically, the z axis). We dilate each primitive to account for any pos-

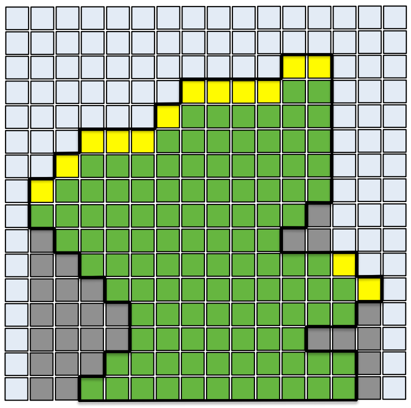

Figure 4: *A 2D representation of our support generation approach. Voxels in green and yellow are part of the object being printed. Voxels in grey are support voxels. Voxels in yellow are part of the depth map that is generated with a high-resolution, fixed-point rasterizer. Support voxels are generated for empty voxels iff there is a voxel in the depth map above them.*

sible displacement using the bounds calculated in the first stage. The resulting depth-map contains the highest point along the z axis at which material is present for each voxel column represented by that given depth sample. During the output phase, if a given voxel is void we output support if and only if the height of that voxel is lower than the highest populated voxel for that particular voxel column as recorded in the depth map (see Fig. 4). When printing with soft materials, one has to additionally create support structures on the sides; this remains future work.

Z-Sort shapes: To progressively fabricate each shape along the print (z) axis, we initially sort the candidate shapes into a priority queue. We use the minimum z value of their bounding boxes as sort keys. Each shape is then retrieved from this queue when the slab we process begins to intersect the bounding box of the shape.

Slab processing *(outer loop)*: We subdivide the print volume into slabs. The size of the slab is dynamically calculated based on target memory usage, and is a function of the resolution of the print and the total build volume. As we process each slab we maintain a working set of shapes whose bounding volume intersects the current slab. As we begin the processing of each slab we update the working set by removing shapes that are now beyond the current slab and adding ones that are now under the slab's domain.

Priority sort: Recall that each object has a user-provided priority that determines which object occupies a given voxel in case of overlap. This is similar to Z-buffer visibility in traditional rendering. However, given that the priority is assigned on a per-object basis, we can forgo traditional per-visibility sample comparisons and presort our shapes up front, akin to performing the reverse painter's algorithm at the object level. Thus, during this phase we sort all objects in the working set based on their priority value. When voxelizing and populating the voxel buffer, if a given voxel is already occupied, the newly arriving voxel can be immediately discarded, giving opportunities for early culling. Culling voxels due to object overlap makes fablet evaluation efficient: only one fablet (the one assigned to the highest priority object) gets evaluated per voxel.

Shape processing *(inner loop)*: We iterate over each shape in our working set in order of priority and perform the next five stages of the pipeline.

Tessellate shape: The first stage of the loop performs partial tessellation. Similarly to Reyes, regardless of the shape type, we always tessellate into micropolygons, our common 2D primitive for the remainder of the pipeline. Tessellated primitives are cached and reused if the primitive straddles multiple slabs. Primitives can also be tessellated on demand in order to respond to a distance function or nearest user attribute query; such tessellations are also cached and reused. The cache has a set size and entries are evicted using a simple LRU scheme.

Surface fablet phase: We evaluate the surface phase of the fablet on the resulting tessellated mesh. We evaluate a quad at a time in order to compute derivatives and thus calculate the filter width needed for filtered sampling of textures. We use the OpenImageIO library as our texture engine [Gritz 2012].

Voxelize shape: We perform solid voxelization by using the odd-even rule (Jordan curve theorem). We cast a ray along one of the principal axes and for each triangle hit we flip the inside/outside bit for all voxels behind the hit. For each hit within a given voxel, we only consider that voxel to be inside the mesh if the center of the voxel is in front of the hit, thus following rasterization rules similar to the ones in the OpenGL and DirectX pipeline. More efficient hierarchical edge-equation based voxelizers exist [Schwarz and Seidel 2010]. Applying them is future work.

Volume fablet phase: We evaluate the volume phase of the fablet for each voxel in our grid. The underlying voxel grid is optimized to store up to 16 materials out of a total of 64 materials that can be defined in the fab world. Careful consideration is given to keeping the memory footprint as small as possible.

Surface distance and attribute queries are evaluated on demand by searching the corresponding acceleration structure. To allow fast startup, the acceleration structure encodes base mesh primitives (expanded conservatively to account for displacement bounds). At search time, candidate base primitives are tessellated and displaced by the surface fablet, and their microgeometry recursively searched for the nearest point or attributes. The results of tessellation and fablet evaluation are cached in the post-tessellation surface cache, so that they are rarely recomputed, but the cache size limits potential memory overhead at the cost of redundant recomputation of surface geometry required in multiple places.

Dither: We apply Floyd-Steinberg dithering [Floyd and Steinberg 1976] for each slice when using multiple materials. We use a sliding window to satisfy our fixed memory requirements and reduce storage pressure for large slabs. We perform the dithering on a grid of the same resolution as the voxelized grid; any errors due to the difference in final effective resolution is simply distributed around the local neighborhood. Error diffusion achieves the right balance: if the fablet outputs one material, the dithered output matches the resolution of the printer. If the fablet outputs multiple materials, the dithered output gracefully reduces resolution in order to achieve the requested material ratios. Our current implementation dithers each slice in 2D. By using a 3D kernel we could diffuse error across slices instead and avoid streaks [Cho et al. 2003].

Output: We output a custom raster format. When targeting commercial printers that only take STL as input, we generate a set of boundary meshes for each material used, using a method similar to marching cubes [Lorensen and Cline 1987]. Given the presence of multiple coordinate systems and resolutions within a given 3D printer (e.g., from the motion system, linear encoders, arrays of

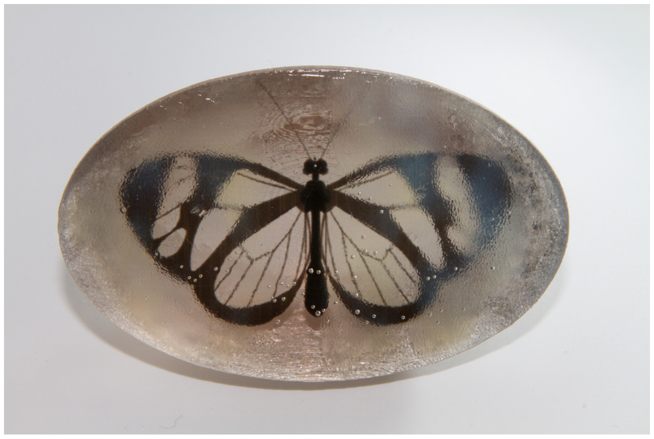

Figure 5: *Insect embedded in amber. Object priority is used to embed the procedurally displaced insect mesh inside the outer amber hemisphere. The amber region mixes small amounts of white material according to procedural noise to model cloudiness and variation in the amber.*

printhead nozzles, variably-sized droplets, different material properties), our native output is abstract enough that it allows a printer-specific backend to perform the necessary mapping to low-level commands that take these various sources of resolution into account.

7 Results

We have designed and fabricated a variety of different objects that highlight features of the OpenFab pipeline.

7.1 3D Prints

Our results were printed on an Objet Connex 500, a high-end multi-material 3D printer that uses photopolymer phase-change inkjet technology and is capable of simultaneously printing with two primary materials and one support material. It supports a variety of polymer-based materials that vary in color, elasticity and optical qualities. It takes per-material geometry meshes as an input. The build volume of the results is limited by the maximum number of primitives allowed by the Objet driver software—at most about 10 million.

Our first result, shown in the teaser (Fig. 1), highlights the ability to easily apply different fablets to the same base geometry. The appearance of the rhinos varies significantly, and each uses a variety of features in OpenFab. For instance, the left rhino uses displacement mapping in the surface phase of the fablet to create micro-spikes over the rhino's skin. The volume phase of the fablet samples from a zebra-like texture to apply a layer of textured material near the surface. It uses the ability to query the nearest point to both retrieve the texture coordinate necessary to sample the texture and to determine whether to apply the textured material. The center rhino has holes carved out throughout its body by returning void in the volume phase of the fablet. We use a distance function to separate the transparent outer shell of the rhino from the black inner core. The right rhino achieves its look in a similar fashion.

Our next result, the butterfly (Fig. 5), highlights the use of object priority to achieve a CSG difference-like operation. The butterfly is placed within a transparent casing to simulate an amber fossil (the butterfly geometry has higher priority than the casing). We procedurally define volumetric cloudiness and particles in order to increase the appearance realism of the amber.

The bunny and the teddy bear pair (Fig. 6) demonstrate the ability to reuse the same fablet across different models. The material used to print these objects is flexible but volume-preserving. The fablet introduces procedurally-defined and repeated void spaces in order to achieve a compressible, foam-like material. This demonstrates the ability to easily define and apply patterned materials. One could also make the 2D or 3D pattern be texture-driven. OpenFab allows one to build a library of such fablets similarly to how material and light libraries are built for image rendering.

The magic postcard (Fig. 7) demonstrates a creative use of texture-driven displacement mapping in its fablet (code in Sec. 5.1). The front face of the postcard (shown left) is textured using a foreground layer of image texture. The back of the postcard (shown right) displaces the surface to create a spatially varying transmission effect. The amplitude of the displacement at each point is driven by the luminance of the background image. When illuminated solely from the front, the background layer is not visible. When another illumination source is added from the back, the whole image becomes visible (shown center). Similarly to other textured objects, the postcard fablet uses nearest point query and distance from the surface to perform texture-driven material assignment.

The marble table in Fig. 8 (center) procedurally recreates the appearance of marble. It uses Perlin noise [Perlin 1985] to define a solid texture in the volume phase of the fablet. Note that the material distribution changes continuously to create a graded material.

The microlens in Fig. 8 (right) demonstrates a working, procedurally-defined microlens array. The surface phase of the fablet transforms a slab of material into an array of aspherical lenses by using displacement mapping. The volume phase of the fablet adds baffles in between the lenslets and assigns the two materials used (clear for lenses and black for the baffles). The baffles reduce the light leakage between neighboring lenses.

Finally, in Fig. 8 (left) we show two examples of objects made of procedurally-defined materials with anisotropic mechanical properties. The core of the material is made of transparent and elastic material. We procedurally insert helical (left) or straight (right) rods made of white and rigid material. These rods influence the mechanical behavior: the helical rods allow twisting motion of the object in clockwise direction and very little twist in the opposite direction; the straight rods transform downward side pressure into transverse motion that causes elongation.

7.2 Performance

We ran a number of simulations to test the scalability of our initial implementation and its ability to provide fabrication data in real time to the 3D printer. Despite the lack of optimizations, our OpenFab implementation meets our design goals and provides satisfactory performance; it can stream the data as fast or faster than a high-end, multi-material 3D printer can output material (in our case, an Objet Connex 500).

We used three different models (center rhino shown in Fig. 1, butterfly shown in Fig. 5, and marble table shown in Fig. 8) and varied their build volume from as little as 3" to as high as 12" across their longest dimension. The simulation assumed 300 DPI printer resolution. The results were collected on an Intel Xeon E5-2650 processor running at only 2.0 GHz.

Performance is summarized in Table 1. We report startup time (time to first slice delivery), per-slice time, and overall run time performance. Note that startup time is always small relative to print time:

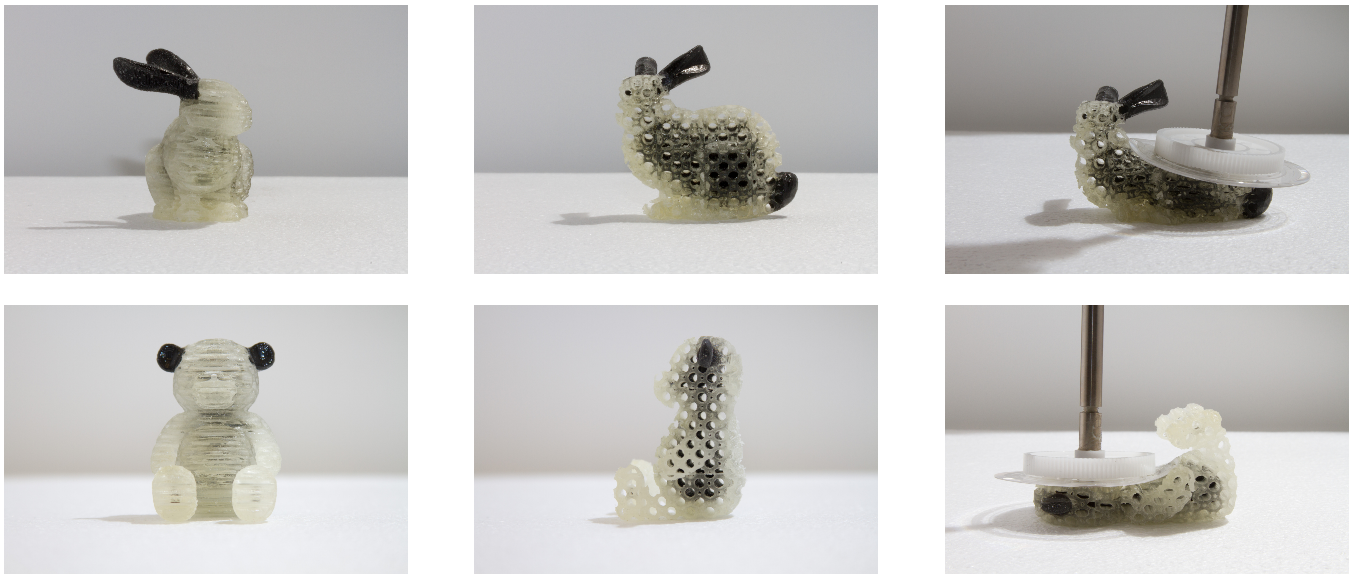

Figure 6: *A procedurally-defined foam material makes the bunny and bear squishy. Color and squishiness vary procedurally over the models.*

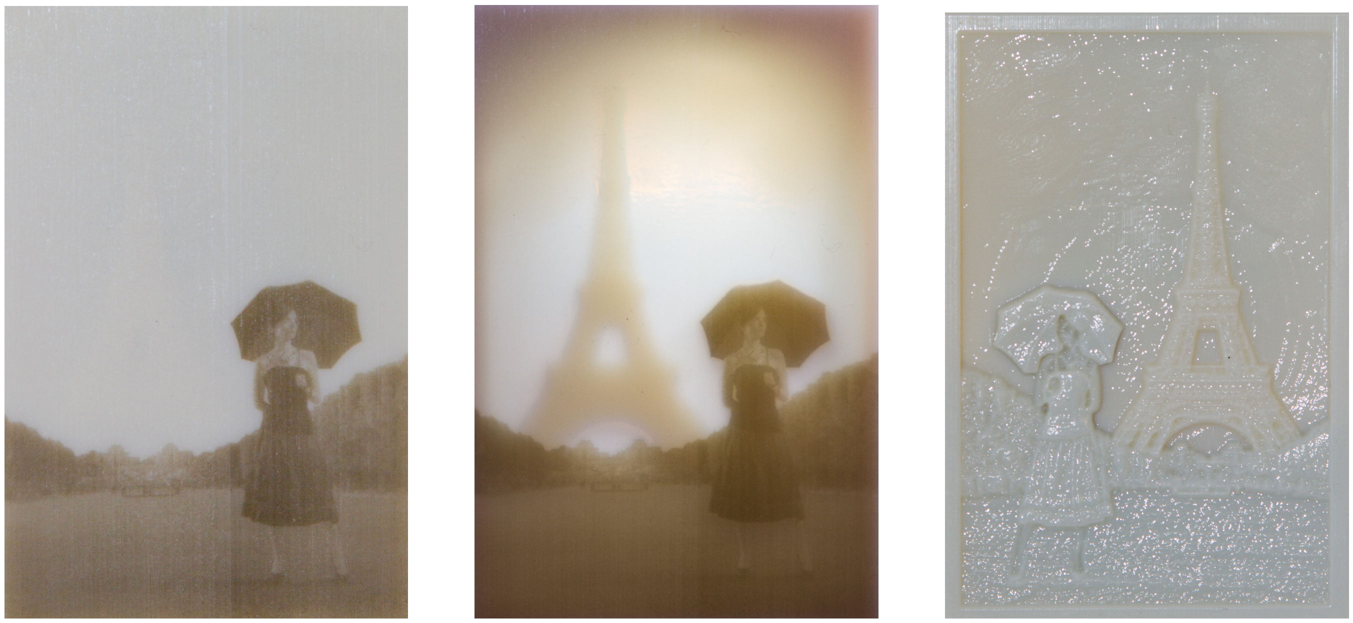

Figure 7: *The front face of the postcard (left) is texture mapped using a foreground image. The back of the postcard (right) displaces the surface to create a spatially varying transmission according to a combined foreground and background image. The result is a hidden background image which only appears when backlit (center).*

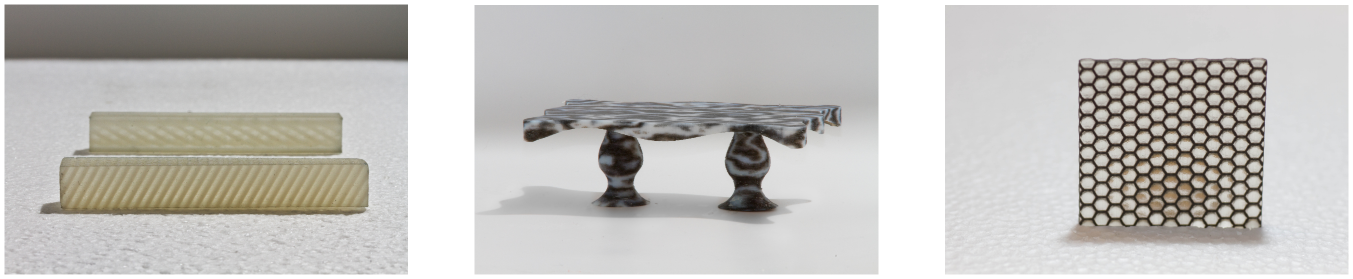

Figure 8: Left*: procedurally-defined materials with anisotropic mechanical properties.* Center*: Marble-like material generated using Perlin noise.* Right*: Procedurally-defined and fully parameterized aspherical microlens array with baffles.*

Models	3 inches	6 inches	12 inches
butterfly	3.18 secs / 0.48 secs / 5 mins *(0.22B voxels)*	4.44 secs / 1.75 secs / 33 mins *(1.8B voxels)*	9.32 secs / 6.63 secs / 249 mins *(14.2B voxels)*
rhino	1.26 secs / 0.46 secs / 7 mins *(0.12B voxels)*	2.70 secs / 1.89 secs / 56 mins *(0.94B voxels)*	7.80 secs / 6.99 secs / 417 mins *(7.5B voxels)*
marble table	2.44 secs / 0.70 secs / 4 mins *(0.15B voxels)*	2.99 secs / 1.25 secs / 25 mins *(1.2B voxels)*	20.52 secs / 12.79 secs / 187 mins *(9.6B voxels)*

Table 1: *OpenFab computational performance, as print volume increases (time to first slice delivery / computation time per-slice / total time). All model sizes are printed at 300 DPI (the highest supported by the Object Connex 500 for two materials). Total number of synthesized voxels is shown below, and ranges from 120M to 14.2B. Startup cost is always negligible relative to print time, and time per-slice is substantially faster than the printer speed. All results are synthesized with a fixed 1.5GB memory budget on a single processor, while computation time per voxel grows slightly* sub-linearly *with print volume, suggesting that our architecture is scalable to large, high resolution prints.*

we are able to start providing print data within at worst 20 seconds. Memory footprint is kept under 1.5 GB, of which 1 GB is reserved for slab data and the rest for ancillary caches and working set data structures. Across different sizes of slices, the system is able to keep up with the printer (e.g., the per-slice print time on the Connex 500 is about 24 seconds for a 12" slice). We observe that a significant amount of our runtime is spent in nearest distance and nearest point queries, which is unsurprising given their global nature. Between parallel code generation of fablets and optimization throughout the individual pipeline stages, we think there is an opportunity for at least an order of magnitude performance increase in the near future.

8 Discussion and Future Work

We have found the programmable pipeline abstraction a surprisingly powerful way to describe complex multi-material 3D prints with a wide range of mechanical and optical properties. We think the OpenFab pipeline provides a solid and scalable foundation on which to build many multi-material fabrication techniques.

The current programming model is powerful, but it is not the most natural way to describe all possible results. In the future, we think there is a great opportunity to spread proceduralism throughout the pipeline. Procedural geometry plugins could be more natural than the existing fablets for some types of geometry (e.g., synthesizing L-systems) and would be complementary to the existing stages. Programmable dithering could also increase the flexibility of the pipeline and the degree of user control over the exact printed output.

Designing a full ecosystem around this pipeline is a natural direction for follow-up work. This could include a procedural modeling tool, a visual fablet authoring tool, and print preview based on measured material properties. It is also desirable to extend the pipeline to integrate various mesh optimizations for automatic partitioning of large prints [Luo et al. 2012] and automatic detection and correction of structural stability [Stava et al. 2012].

Performance is another area of possible future work. Our current implementation is more than fast enough to keep up with current printers. But, as printers get faster, build volumes grow, and fablets become more complex, it will be important to improve performance. Fortunately, there is enormous room for optimization and parallelization in our implementation. Nearest surface queries from the volume fablet phase are a major component of our programming model, and the single most expensive operation in our implementation. There is an opportunity to make these queries more efficient. Further, it will be interesting to define more complex surface-volume attribute relationships, including alternative attribute interpolation methods.

Finally, native backends for many types of printer hardware will be important to realizing the full potential of the OpenFab pipeline. OpenFab was designed from the outset to drive continuous material output at full printer resolution. Current commercial printer software, however, is limited to STL format input and fails when given more than a few million polygons. This significantly limits the scale of spatially varying output we can feed to current commercially available printers. The printer backends, however, take raw full-resolution bitmaps of each slice. Interacting with printers at the raster level will allow streaming prints of continuous material variation at much larger scale.

Given the high-frequency details in dithered multi-material slices, implementing a back-end for vector path 3D printers (e.g., FDM) remains a challenge. Recent work on "multiplexer" extruders that combine multiple filaments is promising, though. We imagine targeting such printers by using dither masks that map local dither patterns to linearly-weighted combinations of the input filaments.

Acknowledgements

We would like to thank Jaakko Lehtinen and Mark Leone for providing seed code for our fixed-point rasterizer and OpenFL compiler (respectively), Mark Leone and Frédo Durand for extensive feedback on our writing, Pitchaya Sitthi-Amorn for providing feedback during design discussions, Ye Wang and Moira Forberg for helping produce our results.

Some of the models were obtained from TurboSquid. The Eiffel Tower photo was obtained through 123RF.

This work was funded in part by NSF grants CCF-1138967 and IIS-1116296, and DARPA grant #N66001-12-1-4242. K. Vidimče was supported by an NSF Graduate Research Fellowship and S. Wang by the MIT Undergraduate Research Opportunities Program.

References

3DSYSTEMS, 1988. StereoLithography interface specification.

ADOBE SYSTEMS, 1985. PostScript language reference.

ASTMSTANDARD. 2011. Standard specification for additive manufacturing file format (AMF) version 1.1. July.

BELL, G., PARISI, A., AND PESCE, M. 1995. The virtual reality modeling language version 1.0 specification. Tech. rep.

BERMANO, A., BARAN, I., ALEXA, M., AND MATUSIK, W. 2012. ShadowPix: Multiple images from self shadowing. *Computer Graphics Forum 31*, 2pt3 (May), 593–602.

BICKEL, B., BÄCHER, M., OTADUY, M. A., LEE, H. R., PFISTER, H., GROSS, M., AND MATUSIK, W. 2010. Design and fabrication of materials with desired deformation behavior. *ACM Trans. Graph. 29* (July), 63:1–63:10.

BLYTHE, D. 2006. The Direct3D 10 system. *ACM Trans. Graph. 25*, 3 (July), 724–734.

CHEN, D., MATUSIK, W., SITTHI-AMORN, P., DIDYK, P., AND LEVIN, D. 2013. Spec2Fab: A reducer-tuner model for translating specifications to 3D prints. *ACM Trans. Graph. 32*, 4 (July).

CHO, W., SACHS, E. M., PATRIKALAKIS, N. M., AND TROXEL, D. E. 2003. A dithering algorithm for local composition control with three-dimensional printing. *Computer-Aided Design 35*, 9, 851–867.

CICHA, K., LI, Z., STADLMANN, K., OVSIANIKOV, A., MARKUT-KOHL, R., LISKA, R., AND STAMPFL, J. 2011. Evaluation of 3D structures fabricated with two-photon-photopolymerization by using FTIR spectroscopy. *Journal of Applied Physics 110*, 6, 064911.

CLARBERG, P., TOTH, R., HASSELGREN, J., AND AKENINE-MÖLLER, T. 2010. An optimizing compiler for automatic shader bounding. *Computer Graphics Forum 29*, 4, 1259–1268.

COHEN-OR, D., AND KAUFMAN, A. 1995. Fundamentals of surface voxelization. *Graph. Models Image Process. 57*, 6, 453–461.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *Proc. SIGGRAPH*, ACM, New York, NY, USA, 95–102.

COOK, R. L. 1984. Shade trees. In *Proc. SIGGRAPH*, ACM, New York, NY, USA, 223–231.

CUTLER, B., DORSEY, J., MCMILLAN, L., MÜLLER, M., AND JAGNOW, R. 2002. A procedural approach to authoring solid models. In *Proc. SIGGRAPH*, ACM, New York, NY, USA, 302–311.

FLOYD, R., AND STEINBERG, L. 1976. An adaptive algorithm for spatial gray scale. In *Proc. Society of Information Display*, vol. 17/2, 75–77.

FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proc. SIGGRAPH*, ACM, New York, NY, USA, 249–254.

GRITZ, L., 2012. OpenImageIO 1.0. `http://openimageio.org`.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Proc. SIGGRAPH*, ACM, New York, NY, USA, 289–298.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2007. PCU: the programmable culling unit. *ACM Trans. Graph. 26*, 3 (July).

HASSELGREN, J., MUNKBERG, J., AND AKENINE-MÖLLER, T. 2009. Automatic pre-tessellation culling. *ACM Trans. Graph. 28*, 2 (May), 19:1–19:10.

HAŠAN, M., FUCHS, M., MATUSIK, W., PFISTER, H., AND RUSINKIEWICZ, S. 2010. Physical reproduction of materials with specified subsurface scattering. *ACM Trans. Graph. 29* (July), 61:1–61:10.

HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. 1998. Sampling procedural shaders using affine arithmetic. *ACM Trans. Graph. 17*, 3 (July), 158–176.

HEWLETT-PACKARD, 1984. Printer command language.

JACKSON, T. R. 2000. *Analysis of functionally graded material object representation methods*. PhD thesis, Massachusetts Institute of Technology.

LATTNER, C., AND ADVE, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, IEEE Computer Society, Washington, DC, USA, CGO '04.

LIU, H., MAEKAWA, T., PATRIKALAKIS, N., SACHS, E., AND CHO, W. 2004. Methods for feature-based design of heterogeneous solids. *Computer-Aided Design 36*, 12, 1141–1159.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 163–169.

LUO, L., BARAN, I., RUSINKIEWICZ, S., AND MATUSIK, W. 2012. Chopper: partitioning models into 3D-printable parts. *ACM Trans. Graph. 31*, 6 (Nov.), 129:1–129:9.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph. 22*, 3 (July), 896–907.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4, 23–32.

OBJET. Connex 500 multi-material 3D printing system.

PERLIN, K. 1985. An image synthesizer. In *Proc. SIGGRAPH*, ACM, New York, NY, USA, 287–296.

PIXAR. 2005. The RenderMan Interface. Tech. rep., 11.

REISIN, Z. B. 2009. Expanding applications and opportunities with PolyJet^{TM}rapid prototyping technology. Tech. rep., Objet.

SCHWARZ, M., AND SEIDEL, H.-P. 2010. Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics 29*, 6 (Dec.), 179:1–179:10.

SEGAL, M., AND AKELEY, K. 2012. The OpenGL graphics system: A specification, version 4.3. Tech. rep., SGI.

STAVA, O., VANEK, J., BENES, B., CARR, N., AND MĚCH, R. 2012. Stress relief: improving structural strength of 3D printable objects. *ACM Trans. Graph. 31*, 4 (July), 48:1–48:11.

VOXELJET, 2013. VoxelJet VX4000 – the large-format 3D print system.

WANG, L., LAU, J., THOMAS, E. L., AND BOYCE, M. C. 2011. Co-continuous composite materials for stiffness, strength, and energy dissipation. *Advanced Materials 23*, 13, 1524–9.

WEYRICH, T., PEERS, P., MATUSIK, W., AND RUSINKIEWICZ, S. 2009. Fabricating microgeometry for custom surface reflectance. *ACM Transactions on Graphics 28*, 3 (July), 32:1–32:6.

ZHOU, M., XI, J., AND YAN, J. 2004. Modeling and processing of functionally graded materials for rapid prototyping. *Journal of Materials Processing Technology 146*, 3, 396–402.