# Fab Forms: Customizable Objects for Fabrication with Validity and Geometry Caching

Maria Shugrina[1] Ariel Shamir[2] Wojciech Matusik[1]

[1]MIT CSAIL [2]The Interdisciplinary Center Herzliya

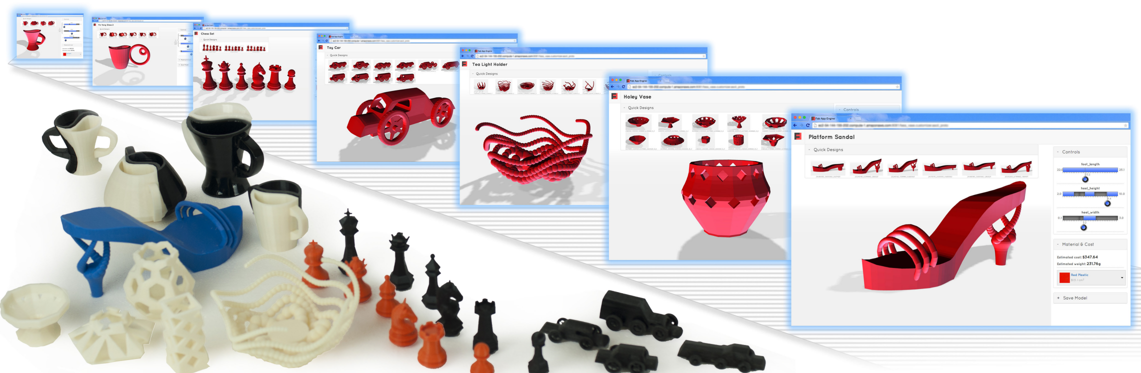


**Figure 1:** *Using offline adaptive sampling, our method converts general parametric designs into Fab Forms, parameterized object representations supporting interactive customization, while ensuring high-level object validity. All realized designs are functional and fabricable and can be previewed in real time. We automatically skin these representations with a Web customization UI intended for casual users.*

## Abstract

We address the problem of allowing casual users to customize parametric models while maintaining their valid state as 3D-printable functional objects. We define *Fab Form* as any design representation that lends itself to *interactive* customization by a novice user, while remaining *valid and manufacturable*. We propose a method to achieve these *Fab Form* requirements for general parametric designs tagged with a general set of automated validity tests and a small number of parameters exposed to the casual user. Our solution separates *Fab Form* evaluation into a precomputation stage and a runtime stage. Parts of the geometry and design validity (such as manufacturability) are evaluated and stored in the precomputation stage by adaptively sampling the design space. At runtime the remainder of the evaluation is performed. This allows interactive navigation in the valid regions of the design space using an automatically generated Web user interface (UI). We evaluate our approach by converting several parametric models into corresponding *Fab Forms*.

**CR Categories:** I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Hierarchy and geometric transformations I.3.8 [Computer Graphics]: Applications—;

**Keywords:** fabrication, customizable design, precomputation, CAD, geometry caching, valid region estimation

## 1 Introduction

Custom products offer inherent advantages over their mass-produced counterparts. Personalized objects can provide more comfort, unique aesthetic appeal, or better performance (e.g. shoes, jewelry, prosthetic limbs). Additive manufacturing promises cost-effective fabrication of such personalized products, as there is a steady progress in the 3D printing hardware and the range of available materials, as well as a reduction in cost and improved availability to consumers (e.g. through 3D printing services).

Yet, taking advantage of additive manufacturing for personalization requires developing customizable digital models and fabrication applications that can be used reliably by everyone. This task proves to be quite challenging, and fabricable end-user-customizable objects available today are very limited. Recent attempts such as one-off apps for customization of household objects [Shapeways; dreamforge], toys [Makieworld Ltd.] and jewelry [Nervous System] either restrict customization to very simple changes, or do not provide a guarantee that the customized objects are valid (e.g. stable, durable) and can truly be fabricated. Furthermore, building a dedicated software application for every customizable design is not scalable. A more general approach of putting a thin UI layer over a parametric model, as in [MakerBot], results in a frustrating user experience with no guarantee on the validity of the resulting model.

In this work, we define the requirements for a representation of an end-user-customizable object in the notion of a *Fab Form*, and provide a method for creating such a representation from a general parametric design. Formally, a *Fab Form* is a 3D design representation supporting:

1. **Customization**: intuitive controls for changing the design.
2. **Validity**: produces only functional fabricable models.
3. **Interactivity**: fast response to changing parameter values.

Our solution for creating *Fab Forms* is driven by one central assumption – the number of parameters exposed to the end-user should generally be low (typically 2-6). This goes in line with re-

cent psychological research stating that people like choices, but not too many choices [Iyengar and Lepper 1999]. For example, a *Fab Form* for a shoe could take as input the weight of the individual and the desired comfort level in addition to the foot dimensions. From the end-user perspective this assumption makes a lot of sense: users want a few intuitive knobs that adjust the corresponding model in a meaningful way. The underlying parametric models can, however, have a few orders of magnitude more degrees of freedom, and arbitrarily complex geometry processing operations.

The assumption of the low-dimensional design space allows us to develop mechanisms that guarantee interactive performance, while providing customization and maintaining validity. We achieve this by offloading expensive geometry generation and automatic validity checks into a precomputation stage, where we adaptively sample the design space spanned by end-user-visible parameters and compute an approximation for its valid regions. Note that while validity verifications can be computationally expensive, storing their results requires relatively little space. In addition, we efficiently precompute the geometry for a *Fab Form* by taking advantage of partial geometry evaluation and redundancy of geometric components in the design space. We ensure an efficient sampling of the design space by adaptive subdivision, taking into consideration changes of the geometry and the boundary of the valid region approximation.

The contributions of this paper can be summarized as follows:

- *Fab Form* abstraction for representing customizable, manufacturable digital objects that can be evaluated interactively.
- Methods to map out and explore valid regions of low-parameter design spaces for arbitrary automated validity tests.
- First hierarchical geometry caching approach optimized for the design space as a whole.
- A method for converting parametric models to *Fab Forms* and evaluation of this method on a number of real world examples.

## 2 Related Work

3D shape customization has been extensively studied in computer graphics. Fast intuitive shape editing methods [Sorkine et al. 2004; Jacobson et al. 2011; Gal et al. 2009; Kraevoy et al. 2008] as well as assembly-based methods [Bokeloh et al. 2012; Chaudhuri et al. 2011] rely on maintaining low-level geometric constraints that produce visually pleasing and intuitive results. However, most of these methods have targeted virtual models for rendering and therefore can produce designs that are not valid for fabrication: deformation methods typically do not protect against self-intersections, and all editing approaches may result in designs that are not functional in a global sense (for instance, too fragile or unstable). Generative shape authoring methods [OpenSCAD; Shapeways 2014; Cutler et al. 2002] suffer from the same limitation. Thus, a typical shape customization loop for fabrication involves running time-consuming checks after every design iteration, and often requires a manual stage for correcting problems. Instead, the aim of our work is to restrict design exploration only to valid designs.

In engineering, customizable design is typically represented as a constrained parametric model produced by commercial CAD packages [Dassault Systèmes; Autodesk]. Although these tools are able to constrain the design according to parametric and geometric constraints set by the designer, parameter manipulation of such models is reserved for experts, who can detect and correct design problems, which range from unsatisfiable constraints to complex functional failures. Simply determining ranges of parameters where constraints are satisfiable is a known hard problem, and has only been addressed analytically for a number of severely limited scenarios [Hoffmann and Kim 2001; Van der Meiden and Bronsvoort 2006; Hidalgo and Joan-Arinyo 2012]. In addition to low-level constraints, some previous work allows setting high-level *topological* constraints on the realized design [van der Meiden and Bronsvoort 2007], or defining and maintaining semantic feature information during parameter manipulation [Bidarra and Bronsvoort 2000], albeit that these methods heavily rely on their chosen sets of constraints. In robotics, a similar problem is encountered in motion planning, when working with the robot configuration space (C-space) [Lozano-Perez 1983]. Although these approaches also rely on sampling, their goal is typically finding a path, not exploring the entire space [Wise and Bowyer 2000], the aim of our work.

A digital design can fail during or after fabrication for a variety of complex reasons. Algorithms for testing design robustness have been developed for decades in the field of engineering (e.g. probabilistic sensitivity analysis [Doubilet et al. 1984; Wu 1994]), and have recently also caught the attention of the graphics community with works such as computational structural analysis [Zhou et al. 2013], automatic correction for areas of high stress [Stava et al. 2012] and semi-automatic correction of model stability [Prévost et al. 2013]. With the exception of fast approximate methods such as [Umetani and Schmidt 2013], comprehensive checks of model durability and functionality typically require costly simulation, making them impractical during interactive customization.

Fast integrated design and analysis systems have been proposed for limited design domains, such as furniture [Umetani et al. 2012] and clothing [Umetani et al. 2011], where insights into the specifics of the chosen domain allow for real-time checks. Common solutions to speed up validity tests that are too computationally expensive often rely on precomputation or data fitting. For example, Pan et al. [2013] train a classifier to quickly estimate inter-penetration depth between two rigid objects, and Umetani et al. [2014] derive a compact model of a glider plane aerodynamic properties from data. Our work is inspired by a more general case, placing no assumptions on the validity tests or the nature of the valid regions.

Guided exploration of the design space, while fulfilling some constraints, is important even when real-time validity tests are available (e.g. it may take many trials for the user to just find a valid model in the space without any guidance), and has been addressed for some domains. For example, Umetani et al. [2012] are able to provide real-time exploration of only valid plank-based furniture designs by operating in the force space specific to the requirements on furniture designs, where the required checks are fast. Similarly, Yang et al. [2011] propose a mathematical framework for exploring desirable regions of the shape space for deformable meshes under constraints, but their approach is restricted to quad meshes with fixed topology and does not handle computationally intensive constraints. A related problem of design space exploration, guided by soft metrics of desirability or user-imposed preferences, has been addressed for a number of domains [Shapira et al. 2009; Talton et al. 2009; Koyama et al. 2014]. For example, Talton et al. [2009] use crowdsourcing to estimate the density of good 3D models in the design space and help novices explore high-quality designs. Guiding users toward better, more interesting designs is very different from maintaining hard requirements on design validity. We explore the problem of restricting design exploration to the strictly valid region of the design space with no limitations on the nature of the design.

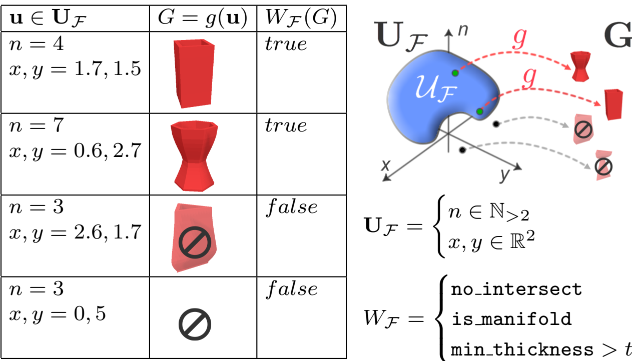
Fast preview is essential for interactive applications, but geometry generation during customization can be computationally intensive. Although fast preview methods are available for some classes of geometric operations such as constructive solid geometry (CSG) operations, others cannot be previewed in real time (e.g. procedural approaches, such as simulation-based shape authoring [Cutler et al. 2002]). Typical representation of a parametric model is a tree [Wyvill et al. 1999; Schmidt and Singh 2008] or linear con-

struction history. Lazy re-evaluation of geometric models is a standard feature of commercial CAD packages [Dassault Systèmes; Autodesk] and smarter node-level caching techniques have been proposed for some classes of geometric operations [Schmidt et al. 2005], but these techniques are not sufficient for interactive applications where a single user edit triggers a significant re-evaluation. We propose the first precomputed hierarchical geometry cache optimized for interactively navigating the entire design space. In a similar spirit, Kim et. al. [2013] precompute simulation-based secondary cloth effects offline to enable high-quality cloth simulation as a character is animated interactively. In order to reduce observable error, that work relies on a metric based on vertex-level cloth deformations. Similarly, we propose using a metric on geometry change to guide the sampling.

## 3 Method Overview

Constrained parametric design is one of the most common customizable object representations. The input to our method is such a parametric design along with a subset of end-user-visible parameters (designated by the designer), and an implementation of any high-level test that is needed to check the validity of the design. Using these, our method converts a design into a *Fab Form* representation that is able to support interactive customization by casual users, while producing only valid manufacturable models.

For a given design $\mathcal{F}$, we denote the space of end-user-visible parameters by $\mathbf{U}_\mathcal{F}$, allowing both discrete and continuous dimensions. We assume $\mathbf{U}_\mathcal{F}$ to be low-dimensional with independent dimensions (e.g., in the presence of constraints), but the underlying parameterized design can have hundreds of interdependent variables, affected by the high-level exposed meta-parameters via constraints. For example, the vase in Table 1 has a three-dimensional $\mathbf{U}_\mathcal{F}$ : a discrete parameter $n$ for the number of sides, and continuous parameters $x$ and $y$ for the coordinates of the middle control point in the profile curve.




| $\mathbf{u} \in \mathbf{U}_\mathcal{F}$ | $G = g(\mathbf{u})$ | $W_\mathcal{F}(G)$ |
|---|---|---|
| $n = 4$<br>$x, y = 1.7, 1.5$ | | *true* |
| $n = 7$<br>$x, y = 0.6, 2.7$ | | *true* |
| $n = 3$<br>$x, y = 2.6, 1.7$ | | *false* |
| $n = 3$<br>$x, y = 0, 5$ | | *false* |

**Table 1:** *A parameterized vase design $\mathcal{F}$ with a three-dimensional space of end-user-visible parameters $\mathbf{U}_\mathcal{F}$ is annotated with high-level validity tests $W_\mathcal{F}$. These tests implicitly define $\mathcal{U}_\mathcal{F}$, the valid region of this design space. Realized designs are shown in the middle column (last design fails to generate geometry for the given parameter settings), and the right column contains the test results.*

We define $g(\mathbf{u}) \in \mathbf{G}$ as the geometric model created for specific parameter values $\mathbf{u} \in \mathbf{U}_\mathcal{F}$. Typically, the mapping $g$ is realized as follows: a constraint solver assigns values to low-level design parameters, which are then used to generate the actual 3D geometry G (Table 1, 2nd column). This process can fail if the constraints are unsatisfiable, or the low-level parameters are assigned invalid values (e.g., negative radius), or if the result is an unprintable geometry (e.g., non-watertight mesh, very thin features). Finally, even manufacturable designs can have structural failures: a shoe may collapse under the weight of the wearer, a cup may not balance on a flat surface. These failures result from complex interplay of variables, constraints and physics, and often cannot be predicted analytically.

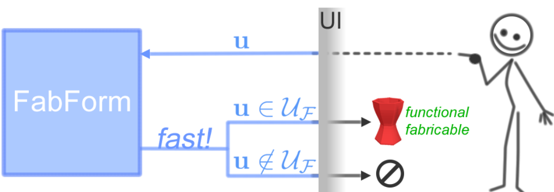



**Figure 2:** *A Fab Form is a customizable object representation that ensures design validity and is suitable for interactive applications. In our representation, a user can only create designs for parameter values* $\mathbf{u}$ *inside* $\mathcal{U}_\mathcal{F}$*, the valid region of the design space.*

To restrict each customizable design only to its valid instantiations, we use the implementation of the high-level tests $W_\mathcal{F} : \mathbf{G} \to \{true, false\}$ appropriate for this design. We assume a library of such tests is given and demonstrate some in Sec. 4. These tests implicitly define the valid region of the design space:

$$\mathcal{U}_\mathcal{F} \triangleq \{\mathbf{u} | \mathbf{u} \in \mathbf{U}_\mathcal{F}, W_\mathcal{F}(g(\mathbf{u})) = true\} \quad (1)$$

where we define $W_\mathcal{F}(g(\mathbf{u}))$ as $false$ at values of $\mathbf{u}$ for which $g(\mathbf{u})$ fails to generate geometry. For example, the vase design in Table 1 is simply tested for 3D-printability (no self-intersections, manifold, with minimum feature thickness above a threshold), but other designs may call for more sophisticated tests such as structural analysis using physical simulation.

Using $\mathcal{W}_\mathcal{F}$, our method analyzes the input design and converts it to an interactive *Fab Form*, that only produces designs in the valid region $\mathcal{U}_\mathcal{F}$ (see Fig. 2). Note that even for moderately complex designs, both geometry generation $g(\mathbf{u})$ and validity tests $\mathcal{W}_\mathcal{F}$ can be computationally expensive, inhibiting interactivity. To address this, we split our method into an expensive offline precomputation step and a fast runtime component.

During offline precomputation, we construct the valid region approximation by running automatic tests over designs sampled over the design space $\mathbf{U}_\mathcal{F}$, and also construct a geometry cache:

**Offline Precomputation (Overview)**

**Input:** parametric design and a set of tests $\mathcal{W}_\mathcal{F}$

- Sample designs in $\mathbf{U}_\mathcal{F}$
- Evaluate tests in $\mathcal{W}_\mathcal{F}$ for every sample (Sec. 5)
- Estimate the valid region $\mathcal{U}_\mathcal{F}$ where the tests pass
- Populate and optimize geometry cache (Sec. 6)

At runtime, we connect an auto-generated customization user interface (UI) over the data structures generated offline. The valid region representation confines the user to the valid region of the parameter space, while the geometry cache enables fast preview:

**Runtime Customization (Overview)**

**Input:** parameter values $\mathbf{u} \in \mathbf{U}_\mathcal{F}$ set by the end user

- Look up parts of the design in the geometry cache
- Evaluate geometry that has not been cached to generate preview
- Dynamically update validity bounds on the exposed parameters

The result is a responsive customizable model that supports an interface protecting the casual user against design failures and providing interactive feedback. See overview in Fig. 3.

In the following section (Sec. 4), we present the design representation used in our system and compatible with our caching scheme

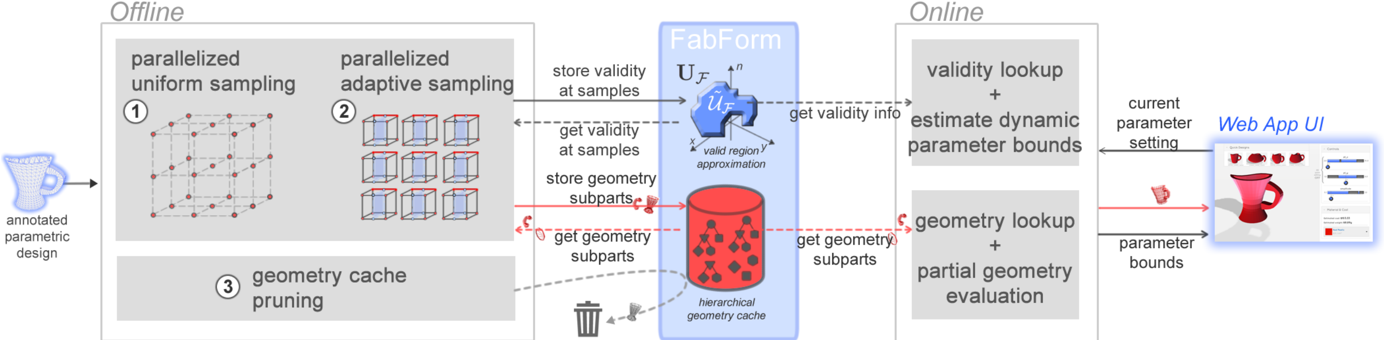
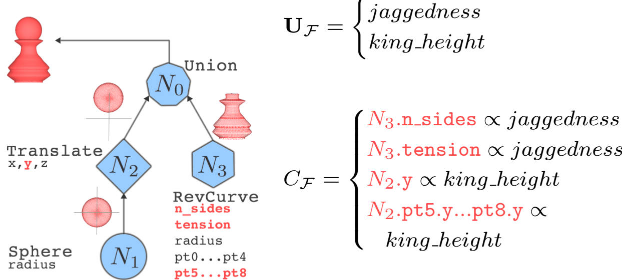



**Figure 3:** ***Method Overview:*** *We sample the space spanned by end-user-visible parameters in a two-stage parallelized adaptive sampling process. During sampling, we assemble a valid region representation and a hierarchical geometry cache. To conclude the offline processing, the cache is pruned based on running time statistics and frequency of evaluation. Resulting data structures provide valid parameter ranges and fast preview at runtime.*

(Sec. 6). In Sec. 5, we present our method for sampling-based valid region approximation that applies to a broader range of design representations. Finally, Sec. 7 describes runtime interaction with our precomputed data structures.

## 4 Design Representation

Our hierarchical design representation is derived from a number of existing approaches, such as CAD models [Bettig and Hoffmann 2011], BlobTrees [Wyvill et al. 1999], and Surface Trees [Schmidt and Singh 2008]. We represent a design $\mathcal{F}$ as one or more trees of operation nodes $N_1 \dots N_k$ (see Fig. 4), where the leaves of the trees generate geometry, and every internal (and root) node encapsulates some parameterized geometry-processing operation acting on the output of its child nodes. The end result of a tree evaluation is the geometric model produced by the root node.




$$\mathbf{U}_\mathcal{F} = \begin{cases} jaggedness \\ king_height \end{cases}$$

$$C_\mathcal{F} = \begin{cases} N_3.\texttt{n_sides} \propto jaggedness \\ N_3.\texttt{tension} \propto jaggedness \\ N_2.\texttt{y} \propto king_height \\ N_2.\texttt{pt5.y...pt8.y} \propto \\ \quad king_height \end{cases}$$

**Figure 4:** *The tree representation of a chess pawn in our system (part of the chess set in Table 2). The leaf nodes $N_1$ and $N_3$ generate geometry which is merged and modified by the internal nodes. The value of every node's mutable parameters (red) is controlled by the exposed parameters $\mathbf{u} \in \mathbf{U}_\mathcal{F}$ via constraints $C_\mathcal{F}$.*

Each node has a number of parameters (boolean, real, or integer), marked as mutable or fixed. We allow any of the parameters, including auxiliary meta-parameters, to be linked by linear and nonlinear constraints. Only a few of these parameters are marked as end-user-visible, but these typically affect all the mutable parameters via constraints (e.g. the pawn in Fig. 4 has 7 mutable parameters governed by only two exposed meta-parameters). Each node re-evaluates its geometry only if some parameters in its subtree have changed. This allows caching of geometry at the subtree level (see Sec. 6).

Each design is annotated with automated tests $\mathcal{W}_\mathcal{F}$. A number of methods for testing a digital design against failure have been suggested (see Sec. 2). We experimented with the following tests:

- satisfiability of constraints,
- watertightness of the geometry for 3D printing,
- object balancing on a flat surface,
- no excessively thin features,
- volume of the material (see **Precomputed properties** below),
- physical simulation to detect significant deformations in an object under load.

We found that even this small set of tests is quite powerful for manufacturing and customization, but more sophisticated tests can also be used with our method (e.g. [Zhou et al. 2013]).

**Precomputed properties:** Some of the tests may depend on thresholds provided only at runtime. For example, we would like to allow users to set a limit on the maximum material volume (and consequently, the printing cost) at runtime, or to display these estimates during customization. To this end, we annotate each design with properties $\mathcal{P}_\mathcal{F}$ to be precomputed (in our example, material volume), in addition to the boolean tests.

## 5 Sampling the Design Space

During the precomputation stage for every design, we sample the design space $\mathbf{U}_\mathcal{F}$ with three goals in mind:

**obj1** Estimate the shape of the valid region $\mathcal{U}_\mathcal{F}$
**obj2** Precompute design properties $\mathcal{P}_\mathcal{F}$
**obj3** Build up the geometry cache (Sec. 6)

For every sample point, we run the constraint solver, evaluate the design tree $\texttt{tree}_\mathcal{F}$ to obtain 3D geometry, run the automated validity tests $\mathcal{W}_\mathcal{F}$, and if the design is valid – evaluate properties $\mathcal{P}_\mathcal{F}$ and populate the geometry cache with the geometry evaluated at some nodes of $\texttt{tree}_\mathcal{F}$. See Alg. 2 in the Appendix for formal definition of the **EvaluateSample** procedure.

We bootstrap adaptive sampling by uniformly sampling $\mathbf{U}_\mathcal{F}$ inside the region bounded by the rough ranges set on the exposed parameters by the designer. This step is necessary because the initial rough bounds can fall outside of the valid region, causing adaptive sampling to return an empty approximation. In practice, we found that about 5-7 subdivisions per dimension are sufficient for bootstrapping. Altogether, the offline precomputation runs in three stages (see Fig. 3), where steps 1 and 2 are executed in parallel:

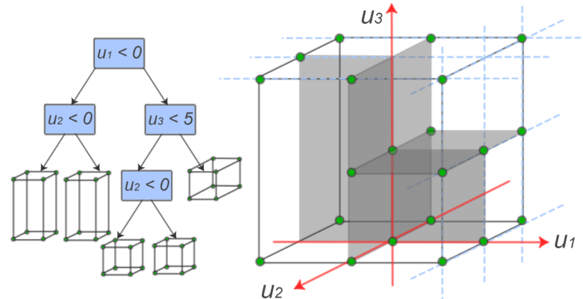



**Figure 5:** *A k-d tree $K_{\mathcal{F}}$ used to estimate and represent the valid region of the design space spanned by parameters $\mathbf{u}_1 \ldots \mathbf{u}_n$. Every leaf is a hypercube with a computed sample (shown in green) at every vertex.*

**Offline Precomputation**

**Input:** a parametric design annotated with tests $\mathcal{W}_{\mathcal{F}}$
**Outputs:** a k-d tree for the valid region $\mathcal{U}_{\mathcal{F}}$ and a geometry cache

**1.** Uniform sampling:
- sample $\mathbf{U}_{\mathcal{F}}$ over a uniform grid
- run **EvaluateSample** on every sample

**2.** Adaptive sampling:
- adaptively subdivide each grid cell
- run **EvaluateSample** on every sample

**3.** Optimize the geometry cache (Sec. 6)

### 5.1 Valid Region Representation

As we sample the design space $\mathbf{U}_{\mathcal{F}}$, we construct a k-d tree-based representation [Bentley 1975] $K_{\mathcal{F}}$ that approximates the valid region $\mathcal{U}_{\mathcal{F}}$ of this space. In contrast to a classic k-d tree, where each leaf holds a number of existing samples, every leaf of our tree $K_{\mathcal{F}}$ corresponds to an undivided hypercube in $\mathbf{U}_{\mathcal{F}}$, with a sample at every vertex (see Fig. 5). At runtime, this representation enables efficient lookup of the valid region boundary (Sec. 7.2) and supports fast preview (Sec. 7.3).

In practice, we construct multiple k-d trees, one for every cell of the uniform sample grid used for bootstrapping. For clarity of discussion, we will assume a single k-d tree.

### 5.2 Adaptive Sampling

Euclidean distance in the design space has little correlation with the quantifiable changes in the realized designs (see Fig. 6). To overcome this, we propose an adaptive sampling scheme. Following the three objectives of sampling, we would like to sample most densely close to the boundary of the current valid region approximation (**obj1**), in the direction where properties are changing the most (**obj2**), and in the direction where the geometry is changing the most (**obj3**).

We propose a scale-invariant metric $\Delta G$ that satisfies all three objectives, under the assumption that the rate of change of properties is correlated with the change in geometry. Let $\mathcal{V}(G)$ map the geometry $G$ to its volumetric representation, and let $|\mathcal{V}(G)|$ denote the scalar volume measure of this representation. We define the change $\Delta G$ between two sample points $\mathbf{u}_1$ and $\mathbf{u}_2$ that evaluate to geometries $G_1$ and $G_2$ as:

$$\Delta G(\mathbf{u}_1, \mathbf{u}_2) \triangleq \frac{|\mathcal{V}(G_1) \oplus \mathcal{V}(G_2)|}{|\mathcal{V}(G_1) \bigcup \mathcal{V}(G_2)|} \tag{2}$$

where $\oplus$ is the symmetric difference (XOR) in volumes. It can be shown that $\Delta G$ is a metric (see Supplemental Material), with values ranging between 0 and 1. Our convention is that samples where

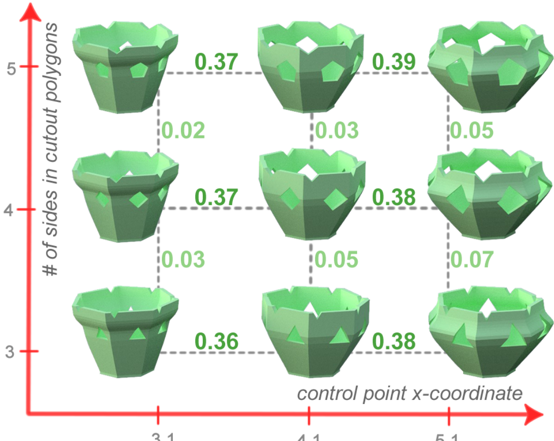



**Figure 6:** ***Pitfalls of uniform sampling:*** *Although the samples are uniformly spaced in this 2-parameter design domain, the design changes much more in the x-direction. The $\Delta G$ metric (green) evaluated between adjacent samples quantifies this perception well, motivating our adaptive sampling approach.*

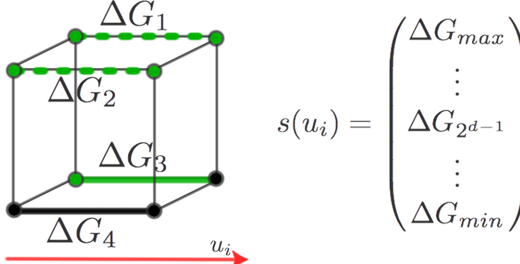



**Figure 7:** ***Adaptive sampling:*** *To pick the best split direction, we compute $\Delta G$ metric along all edges of a hypercube in the design space, picking the direction of largest change using a stacked score vector $s$ for every dimension $u_i$.*

any validity check fails have zero volume. Thus, $\Delta G$ between two invalid samples is zero, and $\Delta G$ between a valid and an invalid sample is 1. Although this metric would be crude for vastly disparate geometries, we find that it quantifies geometric changes well for instances of a parametric design (see Fig. 6).

We use $\Delta G$ to find the optimal split direction for every leaf hypercube in $K_{\mathcal{F}}$, and create new samples where the splitting plane cuts the edges. For every possible split direction $u_i$, we evaluate $\Delta G$ on all edges along that directions, sort these values and stack them in a vector $s(u_i)$ (see Fig. 7). We pick the best direction by sorting vectors $s(u_i)$ (breaking ties using lexicographic order). Subdivision continues until a maximum depth is reached or until the maximum $\Delta G$ in a hypercube is below a certain threshold. Not all dimensions of $\mathbf{U}_{\mathcal{F}}$ are continuous, and special care needs to be taken during subdivision. In particular, if a discrete dimension cannot be split further, we split along the second ranked dimension.

Although the running time of split determination is $O(2^n)$, where $n$ is the dimensionality of the design space, we found that the running time is feasible under our original assumption that $n$ is small. Because computation of $\Delta G$ is quite costly, we cache the results of this computation, and also use $\Delta G$ at runtime to define distances inside the design space (see Sec. 7.3).

## 6 Geometry Cache

Generating geometry based on the user-specified parameters can be computationally expensive. Even if expedited preview meth-

ods are available for some geometric nodes (e.g. CSG operations can be previewed quickly even when creating the actual mesh is time-consuming), we consider the general case, where any geometry processing operation can occur in the nodes of the geometry tree (e.g. the box example in Table 2 uses high-resolution fluid simulation for generating part of the geometry). In order to provide timely preview during customization, we snap to the precomputed sample closest to the current parameter values (Sec. 7.3). This allows us to pre-populate the geometry cache during sampling (Sec. 5).

### 6.1 Design-Space Caching

A naive cache would simply store top-level geometry for every valid sample. However, offline sampling of the entire design space creates a unique opportunity to optimize geometry generation for the *design space as a whole*. There are many approaches that could be applied, such as rearranging the trees of operations, as well as library-specific optimizations and parallelization. We consider a simple approach of caching 3D geometry at the subtree level. The intuition is that certain subtrees of the design are reused throughout the design space and 3D geometry produced by them can be cached.

During sampling, we liberally store the 3D geometry produced by all subtrees where the root node took non-negligible time to compute (otherwise, we could just cache the children). The goal of cache optimization is to select a subset of these subtrees to optimize the time to generate preview across the entire design space under some constraint on the memory, or the converse. We next formally set up this problem.

### 6.2 Cache Optimization Problem Formulation

The geometry output by a node is determined by its subtree, which is uniquely identified by its topology $\tau$, and the values $\rho$ of the low-level parameters in all of its nodes. A subtree keyed by $(\tau, \rho)$ can be evaluated for many different values of end-user-visible parameters. During sampling, we keep track of the average computation time $t_i$ required by the root of every subtree $X_i$ (this is different from the average compute time at every unique *node* of the design tree, as compute time in the parent node often depends on the results of its children).

When the sampling is complete, we construct a dependency graph between all the subtrees evaluated during sampling, where an edge from node $X$ to node $Y$ signifies that root of subtree $X$ requires the result of subtree $Y$. Fig. 8 shows such a dependency graph for a 2-piece chess design evaluated at two samples. Each evaluated sample corresponds to a special *root* node of the graph.

This graph representation prevents us from introducing redundancies into the subtree cache. In addition, it allows easy estimates of the expected and maximum time to evaluate a sample given different cache configurations $H$. To find the total computation time $T_H(r)$ for a sample corresponding to the root $r$ of the graph, we simply sum up $t$ values while running a breadth-first traversal of the children starting at $r$, where traversal terminates at the nodes that are in $H$. We use $n_H(X)$ to denote the number of evaluations for the root of subtree $X$ across *all the samples in the design space*, given cache $H$. To find $n_H(X)$ for a node $X$, we sum up recursively computed $n_H(p_i)$ for all of its parents $p_i$ that are not in $H$, where $n$ at the roots of the graph is set to 1. Any node with all of its parents in the cache is *redundant*.

As a crude approximation to actual probability of reaching a given part of the design space during customization, we assume that snapping to any of the precomputed samples for preview during customization is equally likely. Thus, for a given cache $H$, the expected preview time is:

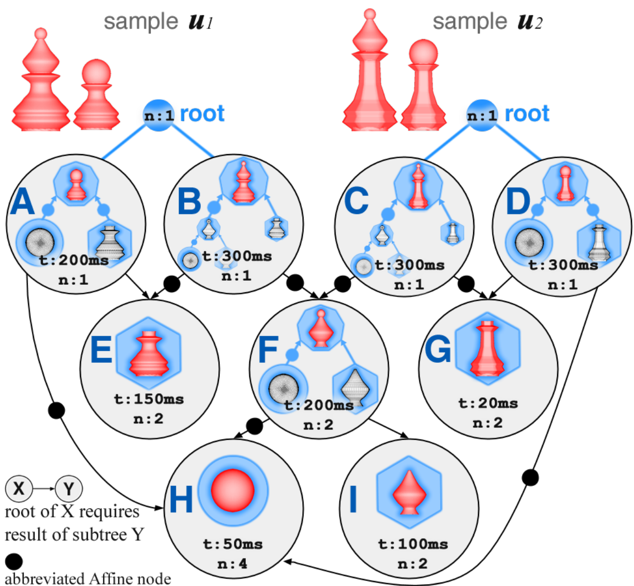



**Figure 8:** *Design with two chess figures evaluated at two samples $\mathbf{u}_1$ and $\mathbf{u}_2$, and the resulting dependency graph between geometry-generating subtrees. Each node represents geometry evaluated by the root of its subtree (red), and requires geometry from its children. This graph prevents us from introducing redundancies into the optimized cache. Notice that the subtree $E$ is evaluated twice for sample $\mathbf{u}_1$, and subtree $H$ is evaluated four times overall.*

$$E_H[preview_time] = \frac{\sum_{r_i \in R} T_H(r_i)}{|R|} = \frac{\sum_{X_i \in N} t_i \cdot n_H(X_i)}{|R|} \quad (3)$$

where $R$ is the set of all samples in the dependency graph, and $N$ is all the nodes. Our approach could be modified to use actual usage statistics. Finding the set of cached nodes to minimize e.g. $E_H[preview_time]$ across all roots given a constraint on memory can be formulated as an integer linear program, which is known to be NP-hard [Garey and Johnson 1979]. As an approximation, we explore a heuristic hill climbing algorithm for greedily selecting nodes to include in the cache.

### 6.3 Hill Climbing for Cache Optimization

We found that a hill climbing approach reducing expected preview time alone results in a sub-optimal user experience, as few very slow updates completely destroy the effect of interactivity. Instead, we employ a hill climbing approach that selects the next node to cache, such that it most reduces the *average* preview time, while also reducing the *worst* update time by any amount (see Alg. 1).

**Algorithm 1** Cache Optimization

```
H ← { }
while not termination_condition do
    roots_max ← graph roots with maximum T(.)
    X_max ← descendant node of roots_max with
            maximal t_i · n_H(X_i)
    add X_max to H
    remove any children of X_max that have become redundant
```

In practice, this results in the same average preview time for a given cache size, while preferentially reducing the worst preview time

(see Fig. 13). The space-efficiency of our algorithm gracefully degrades to naive solution and automatically exploits decoupled degrees of freedom in the design. We discuss its strengths and limitations in Sec. 8.

# 7 Runtime Customization

## 7.1 Automatic Generation of the Customization UI

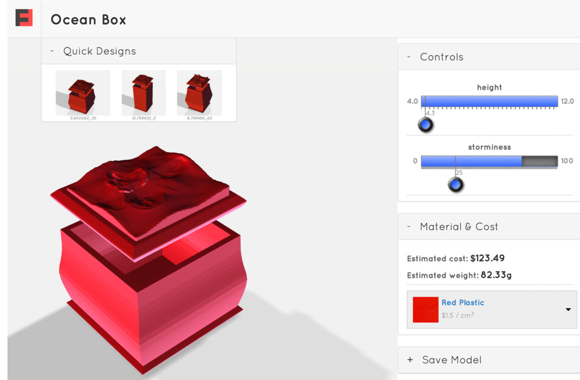



**Figure 9:** *An automatically created Web App, supported by the back end Fab Form representation created using our approach. The valid slider intervals (blue) adjust automatically based on the current position* $\mathbf{u}$ *in the design space, and interactive preview is generated using the geometry cache (this example requires costly fluid simulation to create surfaces of different "storminess", but not all settings result in printable geometry).*

The motivation behind *Fab Forms* is to enable interactive customization applications (recall Fig. 2). To demonstrate the usefulness of our data structures in this original context, we have built an engine that automatically creates a customization UI for *Fab Forms* constructed using our system (e.g., Fig. 9). Using the k-d tree representation of the design space, the interface confines the user to its valid region (Sec. 7.2). When a parameter value is changed, the valid ranges of the other parameters adjust automatically providing a guarantee on design validity. The geometry cache is used to generate fast preview (Sec. 7.3). Our UI engine is implemented as a Web App in HTML/javascript with WebGL support, and a C++ back end that performs validity lookup and geometry generation. Note that with this architecture all precomputed data structures are stored in the cloud, making this setting especially attractive for novice users.

## 7.2 Local Validity Ranges

We take a conservative approach and define the valid region $\mathcal{U}_\mathcal{F}$ of the design space as the union of all leaf hypercubes in the k-d tree (see Sec. 5.1) with samples at all vertices marked as *valid* (e.g. Fig. 10). Thus, any point $\mathbf{u}$ is considered valid if it lies inside a valid hypercube, or on an all-valid face or edge.

Given user-provided parameter vector $\mathbf{u}$, we use the k-d tree to find the valid ranges for all the other parameters in order to restrict the UI to only the valid design regions (accessible slider ranges shown in blue in Fig. 9). Note that the valid intervals are different for different values of $\mathbf{u}$. We also observed that simply finding the maximum and minimum bounds on each parameter around $\mathbf{u}$ is insufficient, as the shape of the valid region is often non-convex. To find valid intervals along a given design dimension $\bar{u}_i$, we trace a line through $\mathbf{u}$ along $\bar{u}_i$ and evaluate validity of all the cuts made along that direction during sampling (a small number in practice),

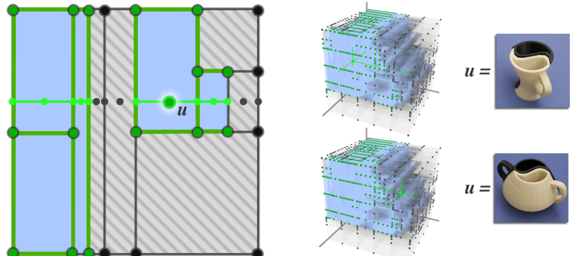



**Figure 10:** *Left: finding valid parameter ranges around current point in the design space* $\mathbf{u}$*, using our k-d tree representation. Right: visualization of the actual estimated valid region (blue) for the Yin Yang Cup 1 (see Table 2) and valid intervals for two different values of* $\mathbf{u}$.

as well as points between cuts to ensure contiguous valid regions (see Fig. 10 for all considered points along a given dimension).

Because validity lookup operates on the raw k-d tree of samples with no post-processing of the valid regions, the shape of the valid region can be changed quickly at runtime if the user sets new thresholds on some of the computed properties $\mathcal{P}_\mathcal{F}$ (e.g. material volume in our case). We reserve exploration of this for future work, but use precomputed material volume property to provide live updates on the model cost and weight in different materials.

## 7.3 3D Model Preview Generation

We provide an interactive preview of the customized object by mapping the current point in the design space to the closest valid sample, running constraint solver (fast) to set low-level parameters of the design tree (Sec. 4) and loading cached geometry subtrees to expedite evaluation (Sec. 6). We first find the leaf hypercube containing the user-set parameter values $\mathbf{u}$, and then snap to the closest valid vertex for preview, using a linear approximation of the distance inside the hypercube based on the $\Delta G$ values computed along all of its edges during sampling. Snapping to a pre-computed vertex ensures high number of cache hits, resulting in faster rendering (see Sec. 6). This approach works, because adaptive sampling based on $\Delta G$ metric, generates more samples where geometry is changing the most. In practice, this preview creates an impression of continuously changing parameters, with only occasional jumps in model appearance. Feedback we received during our user study (see Sec. 8) indicated this as only a minor problem. Of course, if exact specification is important, the final model for fabrication can be computed at non-interactive rates using exact parameter values.

## 7.4 Navigating the Valid Region

Experimenting with several example designs (Table 2) revealed that it is quite easy to create designs where the volume of the valid region $\mathcal{U}_\mathcal{F}$ is very small, for instance, by setting overly strict constraints. When this happens, it may be difficult to reach all the valid segments of the design space using simple sliders (e.g. Fig. 11). Making sparse, non-convex regions of a multi-dimensional design space navigable is a rich topic of future investigation. To start addressing this problem, we generate *customization starting points* that allow users to jump to several points in $\mathcal{U}_\mathcal{F}$ by clicking on a preview, without the use of sliders. The problem of constructing exemplars for navigating parameter spaces has been addressed by Marks et. al. [1997], but not with the additional constraint of validity. We suggest a simple way to reuse results of our precomputation to accomplish this without any additional sampling or expensive computation, as in the prior work [Marks et al. 1997].

In order to generate starting points, we construct a graph from all

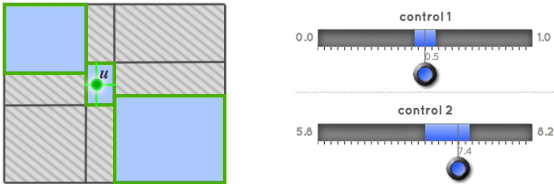



**Figure 11:** ***Example of difficult navigation:*** *For some examples, even large valid regions(blue) may be difficult to reach using valid sliders around the current parameter setting* ***u****.*

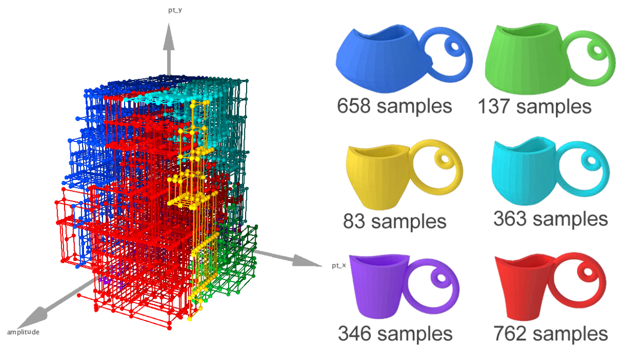



**Figure 12:** ***Design starting points:*** *Valid graph components obtained using our method. Centers of the component graphs (right) are provided as starting points for the valid region exploration.*

the valid samples in the k-d tree $K_{\mathcal{F}}$ (Sec. 5), and create an edge between any two valid samples in a leaf hypercube, with edge weight set to $\Delta G$ already evaluated during precomputation. For our examples, even spaces with very low valid volume typically result in graphs with only one large connected component. To get design starting points, we run randomized farthest point sampling using graph distance to segment graph into components (see graph in Fig. 12). The graph centroid of each component is then chosen as the design starting point (see design variants in Fig. 12 corresponding to subgraphs of the same color). The design variations shown in Table 2 are chosen using this method.

## 8 Results and Discussion

**Designs:** To test our approach, we designed 8 parameterized models (Table 2), and used our method to convert these designs to interactive *Fab Forms*. Our design trees had from 7 to 100 geometric nodes (Sec. 4). The nodes included CSG operations (all models), basic 3D primitives (most models), OpenScad scripts (wheel and handle in car and cup 2, respectively), surfaces of "revolution" with a polygonal base (chess, vase), extruded curves (sandal), a C++ procedure for generating arc-length-parameterized tentacles of unioned spheres (tea light holder, sandal), high-resolution fluid simulation followed by thickening and meshing (box), and imported meshes (horse head in chess set). Although the number of exposed parameters varied from 2 (chess) to 6 (vase), the total number of mutable parameters in the designs ranged between 6 and 68, and was controlled by 2 to 55 linear and non-linear constraints. Each parameterized design was annotated with a set of automatic tests, and properties to be computed:

- **All designs:** constraint satisfiability, watertight geometry, and material volume property
- **Cup 1, Cup 2:** balancing on a flat surface
- **Box:** no very thin parts[1]
- **Sandal:** physical simulation with weight distributed over the sole of the sandal to detect significant deformations[2]

| Design Info | Design Starting Points (see 7.4) |
|---|---|
| CHESS SET<br>102 tree nodes<br>56 mutable parameters<br>55 constraints<br>**2 exposed parameters** | |
| OCEAN BOX<br>14 tree nodes<br>10 mutable parameters<br>8 constraints<br>**2 exposed parameters** | |
| YIN YANG CUP 1<br>19 tree nodes<br>9 mutable parameters<br>7 constraints<br>**3 exposed parameters** | |
| YIN YANG CUP 2<br>20 tree nodes<br>10 mutable parameters<br>7 constraints<br>**3 exposed parameters** | |
| PLATFORM SANDAL<br>28 tree nodes<br>68 mutable parameters<br>65 constraints<br>**3 exposed parameters** | |
| TOY CAR<br>20 tree nodes<br>19 mutable parameters<br>25 constraints<br>**4 exposed parameters** | |
| TEA LIGHT HOLDER (TLH)<br>7 tree nodes<br>6 mutable parameters<br>2 constraints<br>**4 exposed parameters** | |
| HOLEY VASE<br>17 tree nodes<br>12 mutable parameters<br>7 constraints<br>**6 exposed parameters** | |

**Table 2:** *Parametric designs converted to Fab Forms by our system.*

**Precomputation:** For every design, we ran the precomputation stage distributed over 10-40 custom cores on Amazon Web Services (AWS) [Amazon]. Table 3 lists CPU time aggregated across all cores. Even for this relatively small number of cores, distributed sampling took less than a day for most models. This is roughly the time (and price) of 3D printing a few examples, a small cost, given the benefits of allowing thousands of users to customize these models. As expected, the computation time depended most not on the complexity of the model, but on the dimensionality of the de-

[1] Approximated using shape diameter function [Shapira et al. 2008]

[2] We used finite element simulation for statics with hexahedral finite elements and Neo-hookean material model.

sign space, and our 6 and 4-dimensional designs took the longest to compute. An area of future work is coming up with a sampling approach that is not limited to low-dimensional parametric designs.

| $\mathcal{F}$ | $\lvert\mathbf{U}_{\mathcal{F}}\rvert$ | **Samples** | **CPU time** | **% Valid** | **Failure types** |
|---|---|---|---|---|---|
| Chess | 2 | 174 | 11.9 hrs | 100% | None |
| Box | 2 | 420 | 1.0 days | 74% | 2% geometric<br>98% thin parts |
| Cup 1 | 3 | 2,993 | 2.6 days | 29% | 89% unsat. constraints<br>11% geometric<br>0% unstable |
| Cup 2 | 3 | 2,774 | 3.4 days | 43% | 24% unsat. constraints<br>11% geometric<br>64% unstable |
| Sandal | 3 | 1,996 | 7.4 days | 45% | 28% unsat. constraints<br>49% geometric<br>23% too much deformation |
| Car | 4 | 22,147 | 18.6 days | 3.0% | 11% unsat. constraints<br>89% geometric |
| TLH | 4 | 2,402 | 22.1 days | 100% | None |
| Vase | 6 | 145,337 | 138.6 days | 6.5% | 77% unsat. constraints<br>23% geometric |

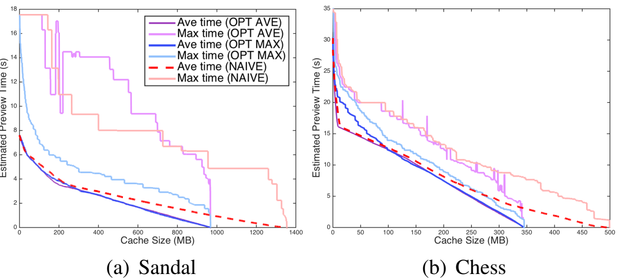
**Table 3:** *Information about the precomputed valid regions of the design spaces for the examples in Table 2. All precomputation times were obtained on $m1.medium$ AWS instances, except for the sandal and box models which ran on $m3.xlarge$ instances due to the demands of physical simulation.*

**Valid Region Analysis:** We compute *valid volume percent* of the design space as the $n$-dimensional volume (product of lengths in all dimensions) of all hypercubes with *all* vertices marked as valid divided by the total volume of the bounded space (see Table 3). This measure is conservative, and even a space with zero valid volume may have large planes of valid samples. Thus, in practice, we are able to explore large regions of the Car and Vase design using sliders, although their valid region volumes are only 3.0% and 6.5%, respectively.

For some designs (chess and TLH), the entire design space was valid, but for other designs automatic testing during precomputation was able to detect a large number of failures that would have made customization cumbersome, at best. For most models, large percentage of the failures resulted from regions of the space having unsatisfiable constraints. This is a common problem faced by mechanical engineers using CAD software, so there is no surprise that these kinds of errors would surface after methodical sampling of the entire design space. Furthermore, we were able to detect unprintable geometry, resulting from geometry processing. Such errors are also common when editing a 3D model by hand, and preventing these errors is important for customizing objects for 3D-printing.

We found that even for similar designs, the distribution of errors varied. For instance, we created two cup designs to investigate how the valid regions change. In Cup 1, there were several tricky and occasionally conflicting constraints governing the angle and position of the handle, which resulted in 89% of all failures. On the other hand, Cup 2 had a large handle that made the stability test dominate the failure cases. These distributions can also give the designer insight into his design, possibly prompting changes. In the Sandal design, we were able to detect configurations of the heel that resulted in too much deformation when bearing a person's weight. We printed one such failed design and confirmed that it breaks under normal usage conditions. Such high-level failures are especially difficult to guard against during customization without our approach.

**Cache Optimization:** In Fig. 13, we compare three hill-climbing strategies for selecting subtrees to leave in the cache for two of our designs. We plot $E_H[preview_time]$ (Eq. 3) and maximum preview time ($T_H^{max}(.)$) across all sample points (see Sec. 6.2). The naive approach (red dash) simply picks nodes with the largest $t_i \cdot n$, where $n$ is the number of subtree evaluations during sampling. Because it does not take into account dependencies between subtree evaluations, the naive approach can introduce redundancies into the cache, and tends to provide the worst improvement in $E_H[preview_time]$ for a given cache size. The OPT-AVE approach, picks subtrees with the highest $t_i \cdot n_H(Xi)$, where $n_H(X_i)$, the evaluation count of the subtree, is updated based on the current state of the cache, and all redundant subtrees are removed as soon as they become redundant. The OPT-MAX approach is described in Sec. 6.3, and we found that while it reduces $E_H[preview_time]$ at about the same rate relative to the cache size as OPT-AVE (compare blue and purple curves in Fig. 13), it also preferentially reduces the maximum preview time (notice that light blue curve for $T_H^{max}(.)$ using OPT-MAX is much lower than the lilac curve of the OPT-AVE approach). This results in less variation in preview time, and better perceived user experience.

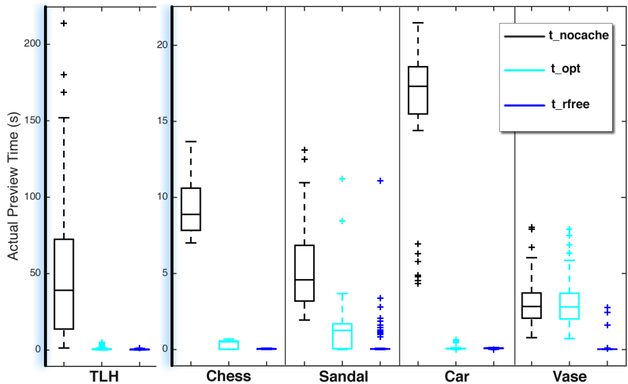



(a) Sandal (b) Chess

**Figure 13:** *We compare maximum (red, purple, blue) and average (pink, lilac, light blue) estimated preview times for cache configurations selected using three different hill-climbing strategies for chess and sandal designs.*

**Subtree cache vs. top-level geometry cache:** The motivation for caching subtrees as opposed to final design geometry at every valid sample (we refer to this as *top-level* cache $H_{TOP}$) is the intuition that certain computationally expensive geometric parts are reused throughout the design space. Thus, this strategy is most space-efficient when the design has decoupled degrees of freedom, and in the worst case approaches the memory footprint of the *top-level cache*. Indeed, even if we run OPT-MAX until all non-redundant subtrees have been cached, this full redundancy-free cache configuration $H_{rfree}$ is only a fraction of $H_{TOP}$ for some of the designs, in particular car, box and vase show 6x, 10x and 75x reduction in storage, respectively (see Fig. 14). In some of the other examples, even though there is redundancy in the internal subtree nodes, top level nodes themselves are computationally intensive and result in caching of top-level geometry despite the redundancy (e.g. in the shoe design the last union node takes as long as several seconds, and thus its $H_{rfree}$ size is nearly equivalent to $H_{TOP}$). We hypothesize that to take full advantage of hierarchical caching, the cache must also store precomputed data structures to make upstream nodes faster.

**Performance Evaluation:** Due to the nature of our designs, and cost of our nodes we found that it was difficult to achieve sub-second preview times without caching the full $H_{rfree}$, but we also experimented with setting the memory threshold lower (see $H_{opt}$




| $\mathcal{F}$ | $H_{TOP}$ | $H_{opt}$ | $H_{rfree}$ | $t_{nocache}$ | $t_{opt}$ | $t_{rfree}$ |
|---|---|---|---|---|---|---|
| Chess | 366MB | 330MB | 345MB | 9.4s | 0.330s | 0.043s |
| Box | 1.48GB | 100MB | 145MB | 10.6s | 0.280s | 0.013s |
| Cup 1 | 178MB | 40MB | 45MB | 0.520s | 0.420s | 0.420s |
| Cup 2 | 663MB | 400MB | 645MB | 0.820s | 0.335s | 0.011s |
| Sandal | 980MB | 650MB | 968MB | 5.2s | 1.2s | 0.210s |
| Car | 17.1GB | 770MB | 2.6GB | 17s | 0.080s | 0.086s |
| TLH | 17.0GB | 13GB | 13.6GB | 50s | 1.0s | 0.413s |
| Vase | 14.8GB | 150MB | 196MB | 3.0s | 3.0s | 0.058s |

**Figure 14:** *Actual time to compute preview across 200 design points in each design in Table 2: $t_{nocache}$ - time when there is no cache, $t_{rfree}$ - time with full redundancy-free cache $H_{rfree}$, $t_{opt}$ - time using slightly smaller cache $H_{opt}$. The $H_{TOP}$ reflects the storage requirements for storing just the top level geometry at every design sample, without the subtree-based caching scheme. We also plot distributions underlying these averages as box plots above.*

in Fig 14). We computed actual time to get preview for 200 samples randomly drawn from the valid region of each design in three regimes – without cache, with $H_{opt}$ and with $H_{rfree}$, obtaining average times $t_{nocache}$, $t_{opt}$ and $t_{rfree}$, respectively (see Fig. 14). To visualize the spread of preview times, we also graph the times for the most computationally intensive designs as box plots for all three cache configurations (Fig. 14). It is clear that without caching interactive customization of most of these designs is not possible. For example, TLH takes on average 50 seconds to generate, and toy car takes 17 seconds. Such long preview times without caching could partially be blamed on our choice of geometry processing libraries that are not fully optimal. However, even for the most optimized geometry processing implementations there are many examples of models that cannot be evaluated in real time. For instance, the box example incorporates a high-resolution fluid simulation to generate the geometry. Our caching and preview approach is agnostic to the geometry generation procedure used, and therefore works as well on these examples as on any other. Our method results in sub-500ms preview time with $H_{rfree}$, which we found quite sufficient for customization interfaces in our user study.

**User Study:** To show that our *Fab Form* representation can support a practical interactive interface, we conducted a small-scale user study with 12 participants, 6 of whom have never prepared a model for 3D-printing. Each participant was assigned two random designs chosen from Table 2 and a printout of a random *goal* model for each design, randomly sampled from its valid region. We gave each participant a 30-second primer on our automatically generated interface (Sec. 7.1), and timed how long it took them to reach a design closely resembling the goal. For 23 out of 24 task-based trials, the users succeeded in reaching the goal within a mean of 2 minutes and 2 seconds (stdev=82 seconds), and performed on average 12 slider manipulations (stdev=7.3), defined as moving the slider followed by releasing it. The failed task occurred on the 6-dimensional Vase example, where the design space was harder to navigate with sliders.

**Figure 15:** *Models customized by our participants during freeform part of the user study.*

In addition, we gave each user a choice of a third design and asked them to create a model they would most like to print (see results in Fig. 15). We aggregated a post-study survey and subjective comments. Most users found sliders with black regions easy to interpret and work with, but a few were puzzled about the cause of the failures and expected more feedback. For higher dimensional spaces, users reported that it is sometimes difficult to navigate the space, such as when a particular desired value of one parameter is not reachable for the values of the other currently set parameters, and reported that quick designs helped navigate the space of possibilities. Several users suggested that quicker preview would be better, but most agreed that the current speed did not hinder interaction. We only received very minor comments about occasionally large jumps in the design appearance and other artifacts of snap-based preview (Sec. 7.3). It is, therefore, a viable way to provide fast feedback for designs requiring precomputation.

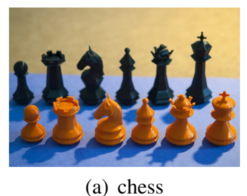
For completeness, we ran a qualitative test to compare against Thingiverse Customizer [MakerBot] by uploading leaf OpenSCAD nodes from our examples (wheel from the car and tentacle from TLH). Customizer provides a thin Web UI layer for [OpenSCAD] scripts, relying on fast preview available for some classes of operations, and does not support constraints or validity checks. Despite these missing features, interactive Customizer experience was significantly inferior to our prototype: no continuous model updates (update only after slider release with 1-3s delay), no printing cost/weight estimates, no interactive viewing at any angle, and there was a long wait on obtaining the actual 3D model.

**3D Prints**: To fully validate our system, we 3D-printed a number of objects customized using our system (Fig. 16, Fig. 1). No additional tests were performed before forwarding the models to the printer.

**Limitations and Future Work:** Our method samples low-dimensional design spaces to construct a valid region approximation, assuming that it can be mapped through sampling. Pathological cases could be constructed, where this assumption fails (e.g. if the result of validity test is a flip of a coin). In future work, we would like to develop methods for estimating confidence in our valid region approximation, and to extend its mapping to higher-dimensional design spaces. Our caching approach is agnostic to fast preview available for certain geometry processing operations, and a more space efficient approach treating preview time and geometry generation time differently could be developed. Our offline method is currently used as a post-processing step after designing a traditional parameterized model. Tools for directly authoring customizable designs constitute a rich and challenging research topic, where we hope our work might be useful.

## Conclusion

We presented an approach for converting any parametric design with a small number of exposed parameters into an end-user-customizable representation, allowing for interactive customization while guaranteeing validity. Our method supports *any* parameterized geometry generation methods, and allows guarding the user in


(a) chess

(b) cup 1

(c) car

(d) TLH (and shadows)

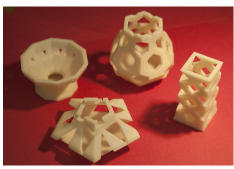

(e) vase

**Figure 16:** *Sample 3D prints for designs customized with our system.*

real time against *any* kind of high-level failure that can be tested for automatically, regardless of the computational cost. We have shown this representation to be practical for automatically creating customization applications for casual users.

## Acknowledgements

We would like to thank Oleksandr Stubailo for the first UI prototype, Desai Chen for the simulation-based tests, David I. W. Levin, Etienne Vouga, and Javier Ramos for fruitful discussion, and Emma Steinhardt, Melody Liu and Lily Zhou for help with design and 3D-printing. We are also grateful to the authors of external libraries [OpenSCAD; Shewchuk 1996; Carve CSG; Clipper; Pfaff and Thuerey 2013; Schulte et al. 2010; Museth 2013; Google; Zaphoyd Studios; jQuery Foundation; threejs] used in our implementation. This research was funded by NSF grant 1138967. Ariel Shamir is partly supported by the Israel Science Foundation (grant no. 324/11).

## References

AMAZON. Amazon web services. http://aws.amazon.com/.

AUTODESK. Autocad. http://www.autodesk.com/products/autodesk-autocad.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9, 509–517.

BETTIG, B., AND HOFFMANN, C. M. 2011. Geometric constraint solving in parametric computer-aided design. *Journal of computing and information science in engineering 11*, 2.

BIDARRA, R., AND BRONSVOORT, W. F. 2000. Semantic feature modelling. *Computer-Aided Design 32*, 3, 201–225.

BOKELOH, M., WAND, M., SEIDEL, H.-P., AND KOLTUN, V. 2012. An algebraic model for parameterized shape editing. *ACM Trans. Graph. 31*, 4, 78:1–78:10.

Carve CSG. http://carve-csg.com.

CHAUDHURI, S., KALOGERAKIS, E., GUIBAS, L., AND KOLTUN, V. 2011. Probabilistic reasoning for assembly-based 3d modeling. *ACM Trans. Graph. 30*, 4, 35:1–35:10.

Clipper. http://www.angusj.com/delphi/clipper.php.

CUTLER, B., DORSEY, J., MCMILLAN, L., MÜLLER, M., AND JAGNOW, R. 2002. A procedural approach to authoring solid models. *ACM Trans. Graph. 21*, 3, 302–311.

DASSAULT SYSTÈMES. Solidworks. http://www.solidworks.com/.

DOUBILET, P., BEGG, C. B., WEINSTEIN, M. C., BRAUN, P., AND MCNEIL, B. J. 1984. Probabilistic sensitivity analysis using monte carlo simulation. A practical approach. *Medical decision making 5*, 2, 157–177.

DREAMFORGE. Cookie caster. http://cookiecaster.com.

GAL, R., SORKINE, O., MITRA, N. J., AND COHEN-OR, D. 2009. iWIRES: an analyze-and-edit approach to shape manipulation. *ACM Trans. Graph. 28*, 3, 33:1–33:10.

GAREY, M., AND JOHNSON, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of books in the mathematical sciences. W. H. Freeman.

GOOGLE. Protocol buffers. http://code.google.com/apis/protocolbuffers/.

HIDALGO, M., AND JOAN-ARINYO, R. 2012. Computing parameter ranges in constructive geometric constraint solving: Implementation and correctness proof. *Computer-Aided Design 44*, 7, 709–720.

HOFFMANN, C. M., AND KIM, K.-J. 2001. Towards valid parametric cad models. *Computer-Aided Design 33*, 1, 81–90.

IYENGAR, S. S., AND LEPPER, M. R. 1999. Rethinking the value of choice: A cultural perspective on intrinsic motivation. *Journal of Personality and Social Psychology 76*, 3, 349–366.

JACOBSON, A., BARAN, I., POPOVIC, J., AND SORKINE, O. 2011. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph. 30*, 4, 78:1–78:8.

JQUERY FOUNDATION, T. jQuery. http://jquery.com/.

KIM, D., KOH, W., NARAIN, R., FATAHALIAN, K., TREUILLE, A., AND O'BRIEN, J. F. 2013. Near-exhaustive precomputation of secondary cloth effects. *ACM Trans. Graph. 32*, 4, 87:1–87:8.

KOYAMA, Y., SAKAMOTO, D., AND IGARASHI, T. 2014. Crowd-powered parameter analysis for visual design exploration. In *Proc. UIST '13*, ACM, 65–74.

KRAEVOY, V., SHEFFER, A., SHAMIR, A., AND COHEN-OR, D. 2008. Non-homogeneous resizing of complex models. *ACM Trans. Graph. 27*, 5, 111:1–111:9.

LOZANO-PEREZ, T. 1983. Spatial planning: A configuration space approach. *IEEE Trans. on Computers 100*, 2, 108–120.

MAKERBOT. Thingiverse customizer. http://www.thingiverse.com/apps/customizer.

MAKIEWORLD LTD. Makies: create your own doll! http://makie.me.

MARKS, J., ANDALMAN, B., BEARDSLEY, P. A., FREEMAN, W., GIBSON, S., HODGINS, J., KANG, T., MIRTICH, B., PFISTER, H., RUML, W., ET AL. 1997. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proc. SIGGRAPH*, ACM, 389–400.

MUSETH, K. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph. 32*, 3, 27:1–27:22.

NERVOUS SYSTEM. Radiolaria: bio-inspired design app. http://n-e-r-v-o-u-s.com/radiolaria.

OpenSCAD. http://www.openscad.org.

PAN, J., ZHANG, X., AND MANOCHA, D. 2013. Efficient penetration depth approximation using active learning. *ACM Trans. Graph. 32*, 6, 191:1–191:12.

PFAFF, T., AND THUEREY, N., 2013. MantaFlow. http://mantaflow.com.

PRÉVOST, R., WHITING, E., LEFEBVRE, S., AND SORKINE-HORNUNG, O. 2013. Make it stand: balancing shapes for 3d fabrication. *ACM Trans. Graph. 32*, 4, 81:1–81:10.

SCHMIDT, R., AND SINGH, K. 2008. Sketch-based procedural surface modeling and compositing using surface trees. *Comp. Graph. Forum 27*, 2, 321–330.

SCHMIDT, R., WYVILL, B., AND GALIN, E. 2005. Interactive implicit modeling with hierarchical spatial caching. In *International Conference on Shape Modeling and Applications*, IEEE, 104–113.

SCHULTE, C., TACK, G., AND LAGERKVIST, M. Z., 2010. Modeling and programming with gecode. http://www.gecode.org/.

SHAPEWAYS. Sake set creator. https://www.shapeways.com/creator/sake-set.

SHAPEWAYS, 2014. ShapeJS. http://shapejs.shapeways.com/.

SHAPIRA, L., SHAMIR, A., AND COHEN-OR, D. 2008. Consistent mesh partitioning and skeletonisation using the shape diameter function. *The Visual Computer 24*, 4, 249–259.

SHAPIRA, L., SHAMIR, A., AND COHEN-OR, D. 2009. Image appearance exploration by model-based navigation. *Comp. Graph. Forum 28*, 2, 629–638.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Applied computational geometry towards geometric engineering*. Springer, 203–222.

SORKINE, O., COHEN-OR, D., LIPMAN, Y., ALEXA, M., RÖSSL, C., AND SEIDEL, H.-P. 2004. Laplacian surface editing. In *Proc. of Eurographics*, ACM, 175–184.

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, 48:1–48:11.

TALTON, J. O., GIBSON, D., YANG, L., HANRAHAN, P., AND KOLTUN, V. 2009. Exploratory modeling with collaborative design spaces. *ACM Trans. Graph. 28*, 5, 167:1–167:10.

three.js. http://threejs.org/.

UMETANI, N., AND SCHMIDT, R. 2013. Cross-sectional structural analysis for 3d printing optimization. In *SIGGRAPH Asia Technical Briefs*, ACM, 5:1–5:4.

UMETANI, N., KAUFMAN, D. M., IGARASHI, T., AND GRINSPUN, E. 2011. Sensitive couture for interactive garment modeling and editing. *ACM Trans. Graph. 30*, 4, 90:1–90:12.

UMETANI, N., IGARASHI, T., AND MITRA, N. J. 2012. Guided exploration of physically valid shapes for furniture design. *ACM Trans. Graph. 31*, 4, 86:1–86:11.

UMETANI, N., KOYAMA, Y., SCHMIDT, R., AND IGARASHI, T. 2014. Pteromys: interactive design and optimization of free-formed free-flight model airplanes. *ACM Trans. Graph. 33*, 4, 65:1–65:10.

VAN DER MEIDEN, H. A., AND BRONSVOORT, W. F. 2006. A constructive approach to calculate parameter ranges for systems of geometric constraints. *Computer-Aided Design 38*, 4, 275–283.

VAN DER MEIDEN, H. A., AND BRONSVOORT, W. F. 2007. Solving topological constraints for declarative families of objects. *Computer-Aided Design 39*, 8, 652–662.

WISE, K. D., AND BOWYER, A. 2000. A survey of global configuration-space mapping techniques for a single robot in a static environment. *The International Journal of Robotics Research 19*, 8, 762–779.

WU, Y. 1994. Computational methods for efficient structural reliability and reliability sensitivity analysis. *AIAA Journal 32*, 8, 1717–1723.

WYVILL, B., GUY, A., AND GALIN, E. 1999. Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system. *Comp. Graph. Forum 18*, 2, 149–158.

YANG, Y.-L., YANG, Y.-J., POTTMANN, H., AND MITRA, N. J. 2011. Shape space exploration of constrained meshes. *ACM Trans. Graph. 30*, 6, 124:1–124:12.

ZAPHOYD STUDIOS. Websocket++.

ZHOU, Q., PANETTA, J., AND ZORIN, D. 2013. Worst-case structural analysis. *ACM Trans. Graph. 32*, 4, 137:1–137:12.

# Appendix

## Evaluate Sample Procedure

The procedure executed for every evaluated sample during precomputation (Sec. 5) is listed in Alg 2.

**Algorithm 2 EvaluateSample**($\mathbf{u}$)

```
1: fix exposed parameters at u
2: solution ← run constraint solver
3: properties[u] ← {constraints_ok : ∃ solution}

4: if no solution then
5:     validity[u] ← false
6: else
7:     set parameters in tree_F to solution
8:     load stale subtrees of tree_F from GeometryCache
9:     G ← evaluate tree_F

10:    all_tests_passed ← true
11:    for test w_i in W_F do
12:        passed ← w_i(G)
13:        properties[u] ← {w_i.name : passed}
14:        all_tests_passed ← all_tests_passed ∧ passed
15:    validity[u] ← all_tests_passed

16:    if all_tests_passed then
17:        populate GeometryCache with subrees of tree_F
18:        for property p_i in P_F do
19:            properties[u] ← {p_i.name : p_i(G)}
```