Methods Enabling Interactive Customization of Fabricable Objects by Non-Professionals

by

Maria D. Shugrina
B. A. Computer Science
Boston University, 2007

submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science and Engineering
at the
Massachusetts Institute of Technology

February 2015

Signature redacted

Signature of Author: .....
Department of Electrical Engineering and Computer Science
January 30, 2015

Signature redacted

Certified by: .....
Professor Wojciech Matusik
Associate Professor of Computer Science

Signature redacted

Accepted by: .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Methods Enabling Interactive Customization of Fabricable Objects by Non-Professionals

by

Maria D. Shugrina

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis addresses the problem of allowing casual users to customize the design of 3D printable objects while maintaining their validity. This work defines *Fab Form* as a representation formalizing the requirements for such designs: 1) the user should have a small number of intuitive parameters that allow customization; 2) the designs should maintain their valid state as 3D printable objects; and 3) the customization process should be interactive. To achieve these, my 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. This approach is evaluated by converting several parametric models into corresponding *Fab Forms*.

Thesis Supervisor: Wojciech Matusik
Title: Associate Professor

Acknowledgments

I thank all the people who have helped me on my path both personally and professionally. Most of all, I thank my mother Nadia Shugrina and my step-dad Alex Grabovetsky. I thank my high school math teacher J. Bryan Sullivan, my college advisor and mentor Professor Margrit Betke, and all the other teachers and professors who have helped me gain knowledge or inspiration throughout the years. I also thank my manager and mentor Michiel Bacchiani, and all the other coworkers who taught me good research practices and engineering skills. I also thank all my friends.

For help, discussion and support related to this thesis in particular, I most of all thank my thesis supervisor, Wojciech Matusik, as well as Ariel Shamir, David I. W. Levin, Oleksandr Stubailo, Emma Steinhardt, Desai Chen, Javier Ramos, and Melody Liu. Additionally, I am grateful to the authors of external libraries [30, 38, 7, 9, 35, 28, 16, 50, 21, 41] for making the implementation underlying this thesis possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Consumers today have a desire to move away from mass-manufactured products towards personalized, custom counterparts. Custom objects can provide higher esthetic value, better performance, or more comfort (e.g., shoes, jewelry, prosthetic limbs). Additive manufacturing and 3D printing promise cost-effective fabrication of such custom, personalized products as there is a steady progress in the 3D printing hardware, the range of available materials, as well as a reduction in cost.

Yet, taking advantage of additive manufacturing for customization and personalization requires developing digital models and fabrication applications that can be used reliably by everyone. This task proves to be quite challenging and recent attempts, such as Thingiverse Customizer [25] and custom-built apps for customization of household goods [36, 13], toys [26], and jewelry [29], fall short. Although these applications provide digital models that can be customized, they either restrict customization to very simple designs, or do not provide a guarantee that the customized objects are valid and can truly be fabricated (e.g. 3D-printable, stable, durable). Furthermore, an approach of building a dedicated software application for every customizable design is not scalable, and a more general approach of putting a thin user interface layer over any parametric model, as in [25], results in a frustrating user experience with slow update times and no guarantee on 3D-printability.

This thesis defines a set of requirements for a representation of an end-user-customizable object in the notion of a *Fab Form*, and provides a method for creating

such a representations automatically from a general parametric designs. A *Fab Form* is a parameterized design that lends itself to *interactive* customization by a novice user, while remaining *valid and manufacturable.* Formally, a *Fab Form* supports three main functionalities:

1. **Customization**: provides an intuitive way to navigate the design space.
2. **Validity**: produces only functional fabricable models during customization.
3. **Interactivity**: maintains fast response rates to changing parameter values.

My 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 recent psychological research stating that people like choices, but not too many choices [19]. For example, a *Fab Form* for a shoe could take as input the weight of the individual and 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.

The assumption of the low-dimensional design space allows me to propose mechanisms that guarantee interactive performance, while providing customization and maintaining validity. This is achieved by offloading expensive geometry generation and automatic validity checks into a precomputation stage, where the design space spanned by end-user-visible parameters is adaptively sampled and an approximation for its valid regions is computed. Note that while validity verifications can be computationally expensive, storing their results requires relatively little space. In addition, the geometry for a *Fab Form* is efficiently precomputed by taking advantage of partial geometry evaluation and redundancy of geometric components in the design space. An efficient sampling of the design space is ensured by adaptive subdivision, which takes into consideration changes of the geometry and the boundary of the valid regions in the design space.

1.1 Contributions

This thesis formally defines a *Fab Form* abstraction for representing customizable, manufacturable digital objects that can be evaluated interactively. Coupled with a thin user interface layer, such object representations open up fail-safe customization of manufacturable objects to the novice users, creating higher demand for custom manufacturing and encouraging the growth of the 3D printing industry. This work proposes a set of algorithms and data structures necessary for an efficient implementation of low-parameter *Fab Forms*, as well as a practical method for converting standard parametric models, commonly used in engineering, to *Fab Forms*. The method and data structures presented in this work are used to convert several real world examples into customizable *Fab Forms*, suggesting this approach as an effective way for automatically creating easy-to-use customization applications.

1.2 Thesis Overview

This thesis is organized as follows. Chap. 2 surveys related work, and Chap. 3 presents an overview of the main method proposed. Chap. 4 addresses the *validity* requirement with a sampling-based approach, while Chap. 5 tackles the *interactivity* requirement. Chap. 6 explains how the proposed data structures and algorithm can be used during runtime *customization*, while meeting the full *Fab Form* requirements listed above. Finally, my method is evaluated in Chap. 7 by converting a number of real example designs into *Fab Forms*, from which I automatically construct Web customization applications, showing my method to be practical and effective.

Chapter 2

Related 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 [12, 48]), and have recently also caught the attention of the graphics community with works such as computational structural analysis [51], automatic correction for areas of high stress [40] and semi-automatic correction of model stability [32]. With the exception of fast approximate methods such as [44], comprehensive checks of model durability and functionality typically require costly simulation, making them impractical during interactive customization.

Methods for 3D shape customization have been extensively studied in computer graphics. Fast intuitive shape editing methods [39, 20, 14, 23] as well as assembly-based methods [6, 8] rely on maintaining low-level geometric constraints that produce visually pleasing and intuitive results that can capture the designer's "intent". 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). Script-based shape authoring methods [30, 37, 10] suffer from the same limitation. Thus, typical shape customization loop for fabrication involves running time-consuming checks after every design iteration, and often requires a manual stage

for correcting problems.

In engineering, customizable design is typically represented as a constrained parametric model produced by commercial CAD packages [11, 2]. Although these tools are able to constrain the design according to parametric and geometric constraints set by the designer, parameter manipulation of such customizable 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 [18, 45, 17]. I take a less rigorous and more practical approach, where the the set of checks the model must satisfy goes beyond low-level constraints on design parts. In a similar spirit, some previous work allows setting high-level *topological* constrains on the realized design [46], or defining and maintaining semantic feature information during parameter manipulation [5], albeit these methods heavily rely on their chosen sets of constraints.

Fast integrated design and analysis systems have been proposed for limited design domains, such as furniture [42] and clothing [43]. These systems are able to analyze customized design at interactive rates by using insights into the specifics of the chosen domain. In the case of plank-based furniture design, Umetani et al. [42] are able to provide real-time exploration of the valid design space by operating in the force space, specific to the requirements on furniture designs, because evaluation of appropriate checks for this design domain is fast. I explore the general problem, when no assumptions can be made about the automated checks or the nature of the design.

The main reason why interactive design tools with real-time validity feedback are challenging is that the automated tests, in general, are too computationally expensive for real time performance. Common solutions in such cases are data driven or precomputation-based approaches. For example, Pan et al. [31] use active learning in the configuration space (relative object translation, rotation) for computing inter-penetration depth between two rigid objects. In robotics, a similar problem is encountered in motion planning, when working with the robot configuration space

(C-space) [24]. Although these approaches also rely on sampling, their central motivation is typically finding a path, not exploring the entire space [47]. Precomputing the feasible region of the design space is the approach of my work.

Apart from the costly validity checks, geometry generation during customization can also take a long time, as is the case with e.g. simulation-based shape authoring by [10]. Typical representation of a parametric model is a tree [49, 33] or linear construction history. Lazy re-evaluation of geometric models defined by construction history is a standard feature of commercial CAD packages [11, 2]. In addition, smarter caching techniques with cache attached to the nodes in the tree have been proposed [34]. For the purpose of interactive customization, lazy evaluation may not be sufficient if a single user edit triggers a significant re-evaluation. In a similar vein, Kim et. al. [22] 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, this work proposes using a metric on geometry change to guide the sampling and learning of the feasible space and geometry cache composition.

Chapter 3

Overview

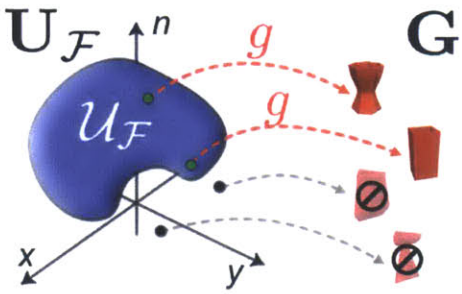
3.1 Method Overview

Constrained parametric design is one of the most common customizable object representations. The input to my method is such a parametric design along with a sub-set 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, my 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 $\mathcal{F}$, $\mathbf{U}_{\mathcal{F}}$ is used to denote the space of end-user-visible parameters, where dimensions may be discrete or continuous. This work relies on the assumption that $\mathbf{U}_{\mathcal{F}}$ is 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 3.1 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 vertical profile curve.

Let us define $g(\mathbf{u}) \in \mathbf{G}$ as the geometric model created by choosing specific parametric values $\mathbf{u} \in \mathbf{U}_{\mathcal{F}}$. Typically, the mapping g is realized as follows: first, a constraint solver assigns values to low-level design parameters, which are then used

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



$$\mathbf{U}_\mathcal{F} = \begin{cases} n \in \mathbb{N}_{>2} \\ x, y \in \mathbb{R}^2 \end{cases}$$

$$W_\mathcal{F} = \begin{cases} \texttt{no\_intersect} \\ \texttt{is\_manifold} \\ \texttt{min\_thickness} > t \end{cases}$$

Table 3.1: A parameterized design $\mathcal{F}$ for a vase annotated with high-level validity tests $W_\mathcal{F}$. The space of end-user-visible parameters $\mathbf{U}_\mathcal{F}$ contains three parameters. The validity 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 geomtry for the given parameter settings), and the right column contains the result of the validity checks.

to generate the actual 3D geometry G (Table 3.1, 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.

To restrict each customizable design only to its valid instantiations, I use the implementation of the high-level tests $W_\mathcal{F} : \mathbf{G} \rightarrow \{true, false\}$ appropriate for this design. It is assumed that a library of such tests is given, and the particular tests implemented for validating this work are listed in Sec. 3.2. These tests implicitly

define the valid region of the design space:

$$\mathcal{U}_{\mathcal{F}} \triangleq \{\mathbf{u} | \mathbf{u} \in \mathbf{U}_{\mathcal{F}}, W_{\mathcal{F}}(g(\mathbf{u})) = true\} \tag{3.1}$$

, where $W_{\mathcal{F}}(g(\mathbf{u}))$ is defined as $false$ at values of $\mathbf{u}$ for which $g(\mathbf{u})$ fails to generate geometry. For example, the vase design in Table 3.1 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.

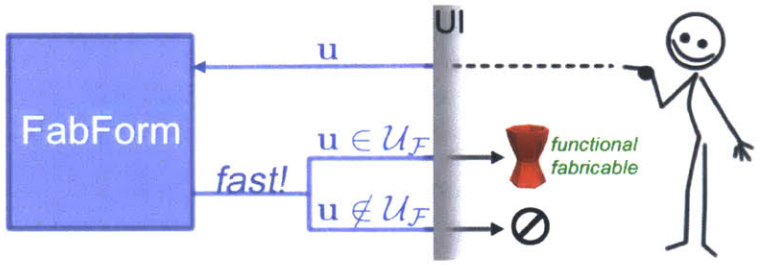


Figure 3-1: A Fab Form is a customizable object representation that ensures final design validity and is suitable for interactive applications. In this representation, a geometric design is only created if the user-specified parameters $\mathbf{u}$ lie in $\mathcal{U}_{\mathcal{F}}$, the valid region of the design space.

Using $\mathcal{W}_{\mathcal{F}}$, my 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. 3-1). 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, the proposed method is split into an expensive offline precomputation step and a fast runtime component.

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

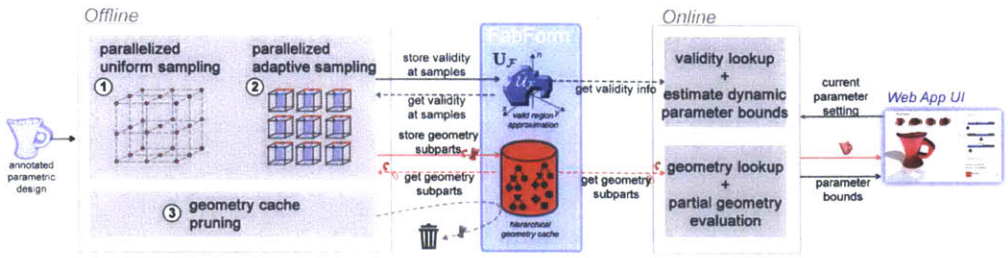


Figure 3-2: **Method Overview:**The space spanned by end-user-visible parameters is sampled in a two-stage parallelized adaptive sampling process. During sampling, a valid region representation and a hierarchical geometry cache are assembled. To conclude the offline processing, the cache is pruned based on running time statistics and frequency of evaluation. Resulting data structures provide valid parameter ranges and fast preview at runtime.

Offline Precompuation (Overview)

Input: parametric design and library of tests $\mathcal{W}_\mathcal{F}$

- Sample designs in $\mathbf{U}_\mathcal{F}$
- Evaluate tests in $\mathcal{W}_\mathcal{F}$ for every sample (Chap. 4)
- Estimate the valid region $\mathcal{U}_\mathcal{F}$ where the tests pass
- Populate and optimize geometry cache (Chap. 5)

At runtime, a customization user interface (UI) is automatically spawned 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. 3-2.

The following section presents the design representation used in this system (Sec. 3.2) that is devised for the caching scheme presented in Chap. 5. Chap. 4 presents my method for sampling-based valid region approximation that applies to a broader range of design representations. Finally, Chap. 6 describes runtime interaction with the pre-computed data structures.

3.2 Design Representation

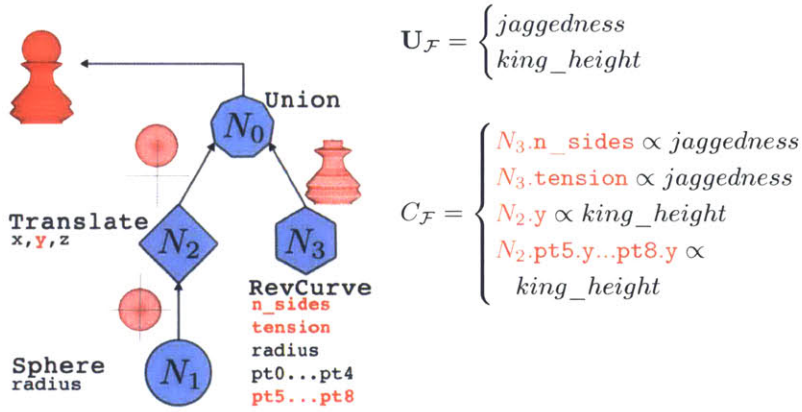


Figure 3-3: The tree representation of a chess pawn (part of the chess set design in Table 7.1). The leaf nodes, N_1 and N_3, generate geometry which is merged and modified by the internal nodes. The value of every node's mutable parameters (marked red) is controlled by the exposed parameters $\mathbf{u} \in \mathbf{U}_\mathcal{F}$ via constraints $C_\mathcal{F}$.

The hierarchical design representation in this thesis is derived from a number of existing approaches, such as CAD models [4], BlobTrees [49], and Surface Trees [33]. A design $\mathcal{F}$ is represented as one or more trees of operation nodes $N_1 \ldots N_k$ (see Fig. 3-3), 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.

Each node type has a number of parameters (boolean, real, or integer), which can be marked as mutable or fixed. Any of the node parameters, as well as any

auxiliary meta-parameters can be linked by linear and non-linear constraints. Only a few of these parameters are marked as end-user-visible, but these typically affect all the mutable parameters in the tree via constraints (e. g. the pawn in Fig. 3-3 has 7 mutable parameters governed by only two exposed meta-parameters via constraints $C_{\mathcal{F}}$). 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 Chap. 5).

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

- satisfiability of constraints,
- watertightness of the geometry for 3D printing,
- object balancing on a flat surface,
- volume of the material (see **Precomputed properties** below.)
- physical simulation to detect significant deformations in an object under load.

Even this small set of tests is quite powerful for manufacturing and customization, but more sophisticated test can also be used with this method (e.g., worst-case structural analysis by Zhou et. al. [51]).

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

Chapter 4

Sampling the Design Space

During the precomputation stage for every design, the design space $\mathbf{U}_{\mathcal{F}}$ is sampled 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 (Chap. 5)

For every sample point, the sampling algorithm runs the constraint solver, evaluates the design tree $\texttt{tree}_{\mathcal{F}}$ to obtain 3D geometry, runs the automated validity tests $\mathcal{W}_{\mathcal{F}}$, and if the design is valid – evaluates properties $\mathcal{P}_{\mathcal{F}}$ and populates the geometry cache with the geometry evaluated at some internal nodes of $\texttt{tree}_{\mathcal{F}}$. See Alg. 1 for formal definition of the **EvaluateSample** procedure.

Adaptive sampling is bootstrapped 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, about 5-7 subdivisions per dimension are sufficient for bootstrapping. Altogether, the offline precomputation runs in three stages (see Fig. 3-2), where steps 1 and 2 are executed in parallel:

Algorithm 1 EvaluateSample(u)

fix exposed parameters at $\mathbf{u}$
`solution` $\leftarrow$ run constraint solver
`properties`[$\mathbf{u}$] $\leftarrow$ {`constraints_ok` : $\exists$ `solution`}

if no `solution` **then**
    `validity`[$\mathbf{u}$] $\leftarrow$ $false$
else
    set parameters in $\texttt{tree}_{\mathcal{F}}$ to `solution`
    load stale subtrees of $\texttt{tree}_{\mathcal{F}}$ from `GeometryCache`
    $G \leftarrow$ evaluate $\texttt{tree}_{\mathcal{F}}$

    `all_tests_passed` $\leftarrow$ $true$
    for test w_i in $\mathcal{W}_{\mathcal{F}}$ **do**
        `passed` $\leftarrow$ $w_i(G)$
        `properties`[$\mathbf{u}$] $\leftarrow$ {w_i.`name` : `passed`}
        `all_tests_passed` $\leftarrow$ `all_tests_passed` $\wedge$ `passed`
    `validity`[$\mathbf{u}$] $\leftarrow$ `all_tests_passed`

    if `all_tests_passed` **then**
        populate `GeometryCache` with subrees of $\texttt{tree}_{\mathcal{F}}$
        for property p_i in $\mathcal{P}_{\mathcal{F}}$ **do**
            `properties`[$\mathbf{u}$] $\leftarrow$ {p_i.`name` : $p_i(G)$}

Offline Precompuation

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 (Chap. 5)

4.1 Valid Region Representation

As the design space $\mathbf{U}_\mathcal{F}$ is sampled, my method constructs a k-d tree-based representation [3] $K_\mathcal{F}$ 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 the tree $K_\mathcal{F}$ corresponds to an undivided hypercube in $\mathbf{U}_\mathcal{F}$, with a sample at every vertex (see Fig. 4-1). At runtime, this representation enables efficient look up of the valid region boundary (Sec. 6.2) and supports fast preview (Sec. 6.3).

In practice, multiple k-d trees are constructed, one for every cell of the uniform sample grid of the bootstrapping. For clarity of further discussion, I will assume that there is a single k-d tree.

4.2 Adaptive Sampling

Euclidean distance in the design space has little correlation with the quantifiable changes in the realized designs (see Fig. 4-2). To overcome this, I propose an adaptive sampling scheme. Following the three objectives of sampling, we would like to sample most densely close to the boundary of $\mathcal{U}_\mathcal{F}$ (**obj1**), in the direction where properties are changing the most (**obj2**), and in the direction where the geometry is changing

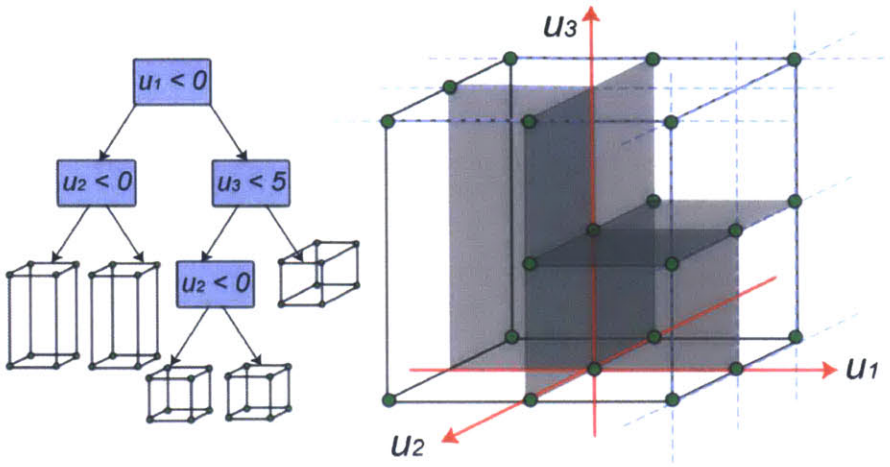


Figure 4-1: A k-d tree $K_{\mathcal{F}}$ used to estimate and represent the valid region of the design space spanned by parameters $\mathbf{u}_1 ... \mathbf{u}_n$. Every leaf is a hypercube with a computed sample (shown in green) at every vertex.

the most (**obj3**).

I propose a scale-invariant metric ΔG 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. The change ΔG between two sample points $\mathbf{u}_1$ and $\mathbf{u}_2$ that evaluate to geometries G_1 and G_2 is defined as:

$$\Delta G(\mathbf{u}_1, \mathbf{u}_2) \triangleq \frac{|\mathcal{V}(G_1) \oplus \mathcal{V}(G_2)|}{|\mathcal{V}(G_1) \bigcup \mathcal{V}(G_2)|} \tag{4.1}$$

where $\oplus$ is the symmetric difference (XOR) in volumes. It can be shown that ΔG is a metric (see Supplemental Material), and its values ranges between 0 and 1. The convention is that samples where any validity check fails have an empty geometry and zero volume. Thus, ΔG between two invalid samples is zero, and ΔG between a valid and invalid samples is 1. Although this metric would be crude for vastly disparate geometries, it tends to quantify geometric changes well for instances of a parametric design (see Fig. 4-2).

ΔG is used to find the optimal split direction for every leaf hypercube in $K_{\mathcal{F}}$,

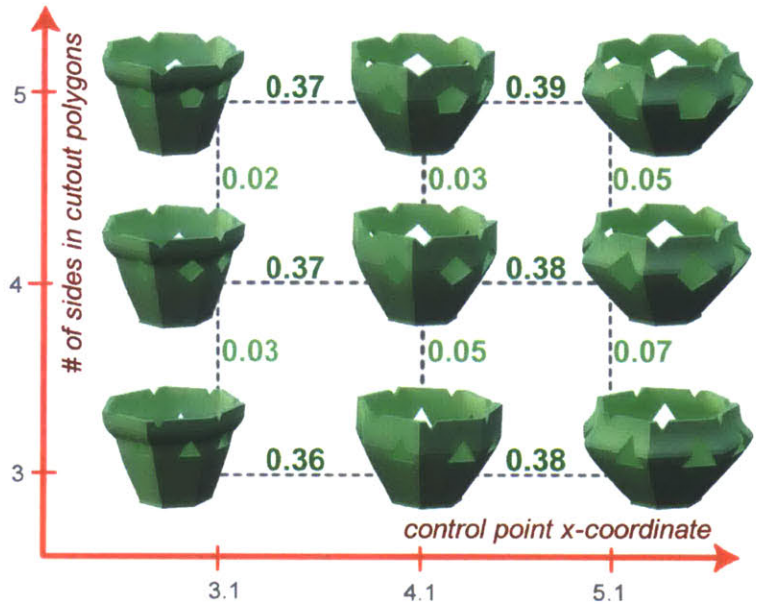
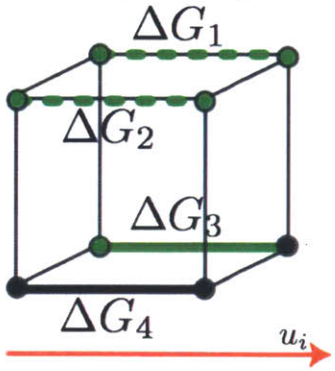


Figure 4-2: **Pitfalls of uniform sampling:** Although the samples are uniformly spaced in this 2-parameter design domain, the design changes much more in the x-direction. The ΔG metric (green) evaluated between adjacent samples quantifies this perception well, motivating the adaptive sampling approach.



$$s(u_i) = \begin{pmatrix} \Delta G_{max} \\ \vdots \\ \Delta G_{2^{d-1}} \\ \vdots \\ \Delta G_{min} \end{pmatrix}$$

Figure 4-3: **Adaptive sampling:** To pick the best split direction, ΔG metric is computed along all edges of a hypercube in design space, and the direction of largest change is picked using a stacked score vector s for every dimension u_i. The ΔG value between valid (green) and invalid (black) samples is set to the maximum value of 1.

and create new samples where the splitting plane cuts the edges. For every possible split direction u_i, ΔG is evaluated on all edges along that directions; these values are sorted and stacked in a vector $s(u_i)$ (see Fig. 4-3). The best direction is picked by sorting vectors $s(u_i)$ in lexicographic order and picking the largest. Subdivision continues until a maximum depth is reached or until maximum ΔG 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, the second ranked dimension is picked for splitting.

Although the running time of split determination is $O(2^n)$, where n is the dimensionality of the design space, experiments (see Chap. 7) show that the running time is feasible under the original assumption that n is small. Because computation of ΔG is quite costly, I cache the results of this computation, and also use ΔG at runtime to define distances inside the design space (see Sec. 6.3).

Chapter 5

Geometry Cache

Generating geometry based on the user-specified parameters can be computationally expensive. Even if expedited preview methods are available for some geometric nodes (e.g. CSG operations can be previewed quickly even when creating the actual mesh is time-consuming), I consider the general case, where any geometry processing operation can occur in the nodes of the geometry tree (Sec. 3.2). In order to provide timely preview during customization, the current parameter values snap to the closest precomputed sample (Sec. 6.3). This allows to pre-populate the geometry cache during sampling (Chap. 4).

5.1 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. I 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, the algorithm liberally stores 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. I next formally set up this problem.

5.2 Cache Optimization Problem Formulation

The geometry output by a node is determined by its subtree, which is uniquely identified by its topology τ, and the values ρ of the low-level parameters in all of its nodes. A subtree keyed by (τ, ρ) can be evaluated for many different values of end-user-visible parameters. The sampling algorithm keeps 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, I 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. 5-1 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, one can 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. Let $n_H(X)$ 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, one can 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

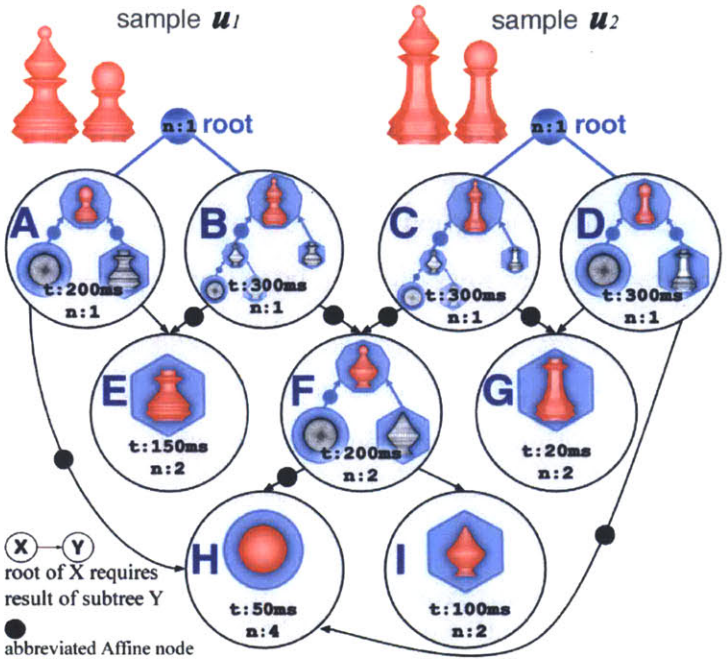


Figure 5-1: Design with two chess figures evaluated at two samples $\mathbf{u}_1$ and $\mathbf{u}_2$, and the resulting dependency graph between geometry-generating subtrees. Each node represents geometry evaluated by the root of its subtree (red), and requires geometry from its children. This graph prevents 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.

redundant.

As a crude approximation to actual probability of reaching a given part of the design space during customization, let us 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:

$$E_H[preview\_time] = \frac{\sum_{r_i \in R} T_H(r_i)}{|R|} = \frac{\sum_{X_i \in N} t_i \cdot n_H(X_i)}{|R|} \tag{5.1}$$

where R are all the samples in the dependency graph, and N are all the nodes. The 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 [15]. As an approximation, I explore a heuristic hill climbing algorithm for

greedily selecting nodes to include in the cache.

5.3 Hill Climbing for Cache Optimization

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, it is better to 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 non-zero amount (see Alg. 2).

Algorithm 2 Cache Optimization

$H \leftarrow \{\ \}$
while not `termination_condition` **do**
  $roots_{max} \leftarrow$ graph roots with maximum $T(.)$
  $X_{max} \leftarrow$ descendant node of $roots_{max}$ with
    maximal $t_i \cdot n_H(X_i)$
  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. 7-1). The space-efficiency of this algorithm gracefully degrades to naive solution and automatically exploits decoupled degrees of freedom in the design. I discuss its strengths and limitations in Chap. 7.

Chapter 6

Runtime Customization

6.1 Automatic Generation of the Customization UI

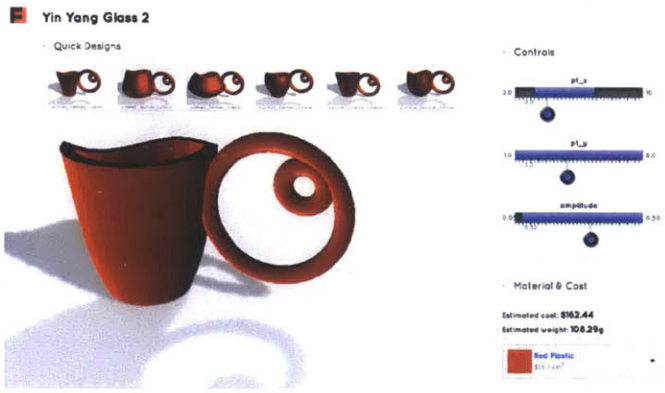


Figure 6-1: An automatically created Web App, supported by the back end Fab Form representation created using my approach. The valid slider intervals (blue) adjust automatically based on the current position $\mathbf{u}$ in the design space, and interactive preview is generated using the geometry cache.

The motivation behind *Fab Forms* is enabling interactive customization applications (Recall Fig. 3-1). To demonstrate the usefulness of the presented data structures in this original context, I have built an engine that automatically creates a customization UI for *Fab Forms* constructed using my system (e.g., Fig. 6-1). Using the k-d tree representation of the design space, the interface confines the user to its valid region (Sec. 6.2). 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. 6.3). The 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 in this setup all precomputed data structures are stored in the cloud, making this setting especially attractive for novice users.

6.2 Local Validity Ranges

To avoid errors, I 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. 4.1) with samples at all vertices marked as *valid* (e.g. Fig. 6-2). Thus, any point $\mathbf{u}$ is considered valid if it lies inside a valid hypercube, or on an all-valid face or edge.

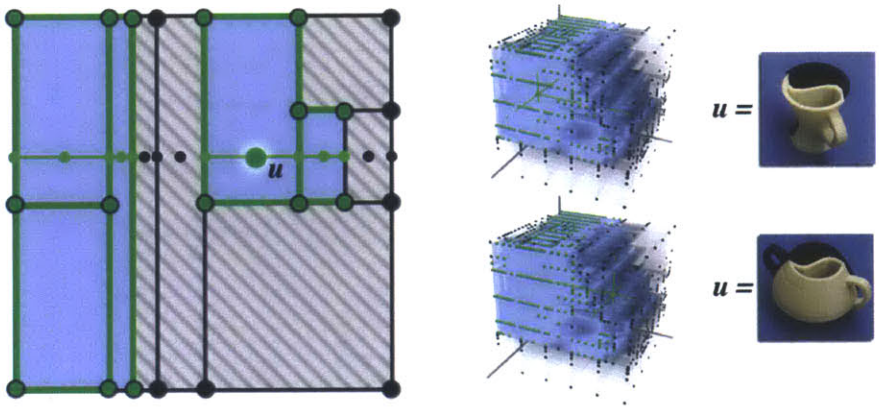


Figure 6-2: Left: finding valid parameter ranges around current point in the design space $\mathbf{u}$, using the k-d tree representation. Right: visualization of the actual estimated valid region (blue) for the Yin Yang Cup 1 (see Table 7.1) and valid intervals for two different values of $\mathbf{u}$.

Given user-provided parameter vector $\mathbf{u}$, the k-d tree is employed to find the valid ranges for all the other parameters in order to restrict the UI to only the valid design regions (acessible slider ranges shown in blue in Fig. 6-1). Note that the valid intervals are different for different values of $\mathbf{u}$. Experimentation showed 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 $\bar{u}_i$, I 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), as well as points between cuts to ensure contiguous valid regions (see Fig. 6-2 for all considered points along a given dimension).

Because validity and ranges 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 this case). I 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.

6.3 3D Model Preview Generation

An interactive preview of the customized object is provided 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. 3.2) and loading cached geometry subtrees to expedite evaluation (Chap. 5). The algorithm first finds the leaf hypercube containing the user-set parameter values $\mathbf{u}$, and then snaps to the closest valid vertex for preview, using a linear approximation of the distance inside the hypercube based on the ΔG 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 Chap. 5). This approach works, because adaptive sampling based on ΔG metric, generates more samples where geometry is changing most. In practice, this preview creates an impression of continuously changing parameters, with only occasional jumps in model appearance. Feedback received during a user study (see Chap. 7) 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.

6.4 Navigating the Valid Region

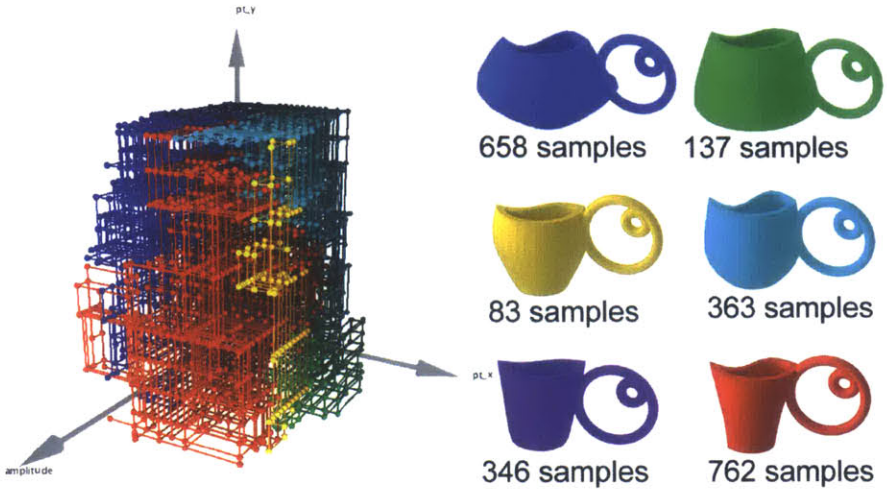


Figure 6-3: Graphs of connected components in the valid region. Centers of the component graphs (right) are provided as starting points for the valid region exploration.

Experimenting with several example designs (Table 7.1) 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. Making sparse, non-convex regions of a multi-dimensional design space navigable is a rich topic of future investigation. To start addressing this problem, I propose generating *customization starting points* that allow user 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. [27], but not with the additional constraint of validity. I suggest a simple way to reuse results of the valid region precomputation to accomplish this without any additional sampling or expensive computation, as in the prior work [27].

In order to generate starting points, I construct a graph from all the valid samples in the k-d tree $K_{\mathcal{F}}$ (Chap. 4), and create an edge between any two valid samples in a leaf hypercube, with edge weight set to ΔG already evaluated during precomputation. For my examples, even spaces with very low valid volume typically result

in graphs with only one large connected component. To get design stating points, I run randomized farthest point sampling using graph distance to segment graph into components (see graph in Fig. 6-3). The graph centroid of each component is then chosen as the design starting point (see design variants in Fig. 6-3 corresponding to subgraphs of the same color). The design variations shown in Table 7.1 are chosen using this method.

Chapter 7

Evaluation and Discussion

7.1 Designs for Evaluation

To test the presented approach, I designed 7 parameterized models (Table 7.1), and used the method of this work to convert these designs to interactive *Fab Forms*. The design trees had from 7 to 100 geometric nodes (Sec. 3.2). 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), 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
- **Sandal:** physical simulation with weight distributed over the sole of the sandal to detect significant deformations[1]

[1] I used finite element simulation for statics with hexahedral finite elements and Neo-hookean

Design Info	Seed Designs (see 6.4)
CHESS SET 102 tree nodes 56 mutable parameters 55 constraints **2 exposed parameters**	
YIN YANG CUP 1 19 tree nodes 9 mutable parameters 7 constraints **3 exposed parameters**	
YIN YANG CUP 2 20 tree nodes 10 mutable parameters 7 constraints **3 exposed parameters**	
PLATFORM SANDAL 28 tree nodes 68 mutable parameters 65 constraints **3 exposed parameters**	
TOY CAR 20 tree nodes 19 mutable parameters 25 constraints **4 exposed parameters**	
TEA LIGHT HOLDER (TLH) 7 tree nodes 6 mutable parameters 2 constraints **4 exposed parameters**	
HOLEY VASE 17 tree nodes 12 mutable parameters 7 constraints **6 exposed parameters**	

Table 7.1: Parametric designs converted to Fab Forms by my system.

material model.

7.2 Design Space Analysis

7.2.1 Precomputation

For every design, I ran the precomputation stage distributed over 10-40 custom cores on Amazon Web Services (AWS) [1]. Table 7.2 lists CPU time aggregated across all cores. For most models, distributed sampling took less than a day even on the relatively small number of cores, making this approach feasible in our compute-rich world. As expected, the computation time depended most not on the complexity of the model, but on the dimensionality of the design space, and the 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.

$\mathcal{F}$	$\lvert U_{\mathcal{F}} \rvert$	**Samples**	**CPU time**[2]	**% Valid**	**Failure types**
Chess	2	174	11.9 hrs	100%	None
Cup 1	3	2,993	2.6 days	29%	89% unsat. constraints 11% geometric 0% unstable
Cup 2	3	2,774	3.4 days	43%	24% unsat. constraints 11% geometric 64% unstable
Sandal	3	1,996	7.4 days	45%	28% unsat. constraints 49% geometric 23% too much deformation
Car	4	22,147	18.6 days	3.0%	11% unsat. constraints 89% geometric
TLH	4	2,402	22.1 days	100%	None
Vase	6	145,337	138.6 days	6.5%	77% unsat. constraints 23% geometric

Table 7.2: Information about the precomputed valid regions of the design spaces for the examples in Table 7.1. All precomputation times were obtained on *m1.medium* AWS instances, except for the Sandal model which ran on *m3.xlarge* instances due to the demands of physical simulation testing.

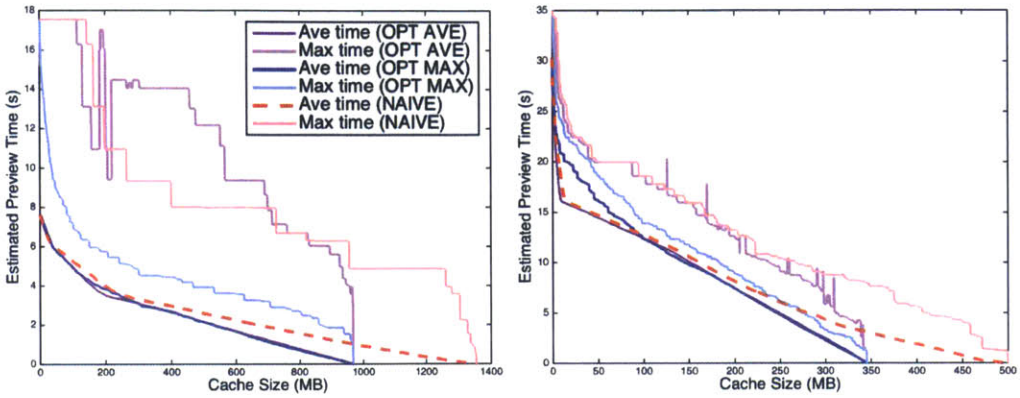
7.2.2 Valid Region

I 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 7.2). This measure is conservative, and even a space with zero valid volume may have large planes of valid samples. Thus, in practice, the user is 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, the method was able to detect unprintable geometry, resulting from geometry processing. Such errors are common when editing a 3D model by hand, and preventing these errors is important for customizing objects for 3D-printing.

Even for similar designs, the distribution of errors varied. For instance, consider the two cup designs, created to investigate how the valid regions change with changes in design. 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, the precomputation algorithm was able to detect configurations of the heel that resulted in too much deformation when bearing a person's weight. Such high-level failures are especially difficult to guard against during customization without an approach such as the one presented in this thesis.

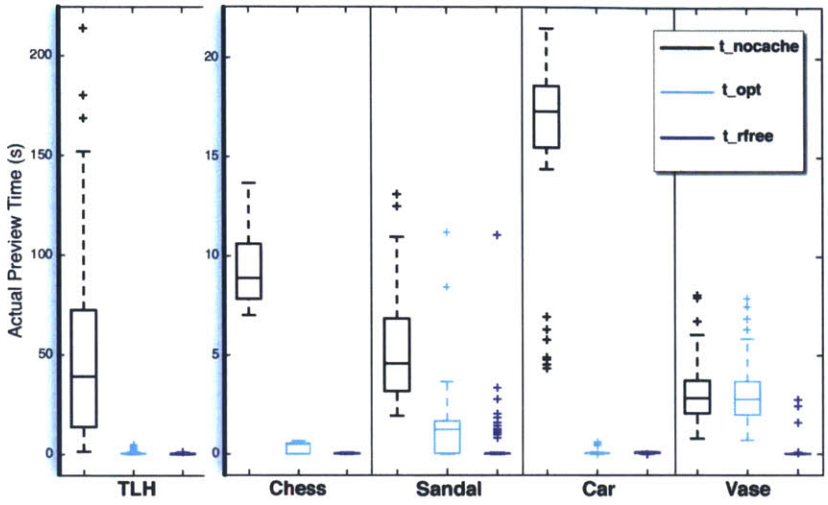
7.2.3 Cache Optimization



(a) Sandal (b) Chess

Figure 7-1: Comparison of 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.

Fig. 7-1 compares three hill-climbing strategies for selecting subtrees to leave in the cache for two of the designs. The plot shows $E_H[preview\_time]$ (Eq. 5.1) and maximum preview time ($T_H^{max}(.)$) across all sample points (see Sec. 5.2). 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. 5.3, and experiments show 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. 7-1), 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.



$\mathcal{F}$	H_{TOP}	H_{opt}	H_{rfree}	$t_{nocache}$	t_{opt}	t_{rfree}
Chess	366MB	330MB	345MB	9.4s	0.330s	0.043s
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 7-2: Actual time to compute preview across 200 design points in each design in Table 7.1: $t_{nocache}$ - 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. The distributions underlying these averages are also plotted as box plots above.

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 OPT-MAX were run 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 and vase show 6x and 75x reduction in storage (see Fig. 7-2). 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 in despite of the redundancy (e.g. in the shoe design the last union node takes as long as several seconds, and its H_{rfree} size is nearly equivalent to H_{TOP}). Given these observations it is likely that to take full advantage of hierarchical caching, the cache must also store precomputed data structures to make upstream nodes faster.

7.2.4 Performance Evaluation

Due to the nature of the experimental designs, and the cost of the implemented nodes it was difficult to achieve sub-second preview times without caching the full H_{rfree}, but I also experimented with setting the memory threshold lower (see H_{opt} in Fig 7-2). I computed actual time to get preview for 200 samples randomly drawn from the valid region of each design for no cache, H_{opt} and H_{rfree}, obtaining average times $t_{nocache}$, t_{opt} and t_{rfree}, respectively (see Fig. 7-2). To visualize the spread of preview times, the times for the most computationally intensive designs are graphed as box plots for all three cache configurations (Fig. 7-2). 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 be blamed on the 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 Tea Light Holder involves hunderds of procedurally generated CSG operations, but one could also create geometry using simulation and other computationally expensive operations. The precomputation approach is agnostic of the geometry generation procedure used, and results in sub-500ms preview time with H_{rfree}, which was found quite sufficient for customization interfaces by running a user study.

7.3 Evaluation of the Resulting User Interface

To show that the *Fab Form* representation can support a practical interactive interface, I 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 7.1 and a printout of a random *goal* model for each design, randomly sampled from its valid region. Each participant was given a 30-second primer on the automatically generated interface (Sec. 6.1), and I 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 thad releasing it. The failed task occurred on the 6-dimensional Vase example, where the design space was harder to navigate with sliders.

Figure 7-3: Models customized by the participants during the freeform part of the user study.

In addition, each user was given a choice of a third design and asked to create a model they would most like to print (see results in Fig. 7-3). I 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 helpfed 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. Only very minor comments about ocasionally large jumps in the design appearance and other artifacts of snap-based preview (Sec. 6.3) were received. It is, therefore, a viable way to provide fast feedback for designs requiring precomputation.

7.4 3D Prints

To fully validate the system, I 3D-printed a number of objects customized using this system (see Fig. 7-4). No additional tests were performed before forwarding the models to the printer.

(a) CS

(b) YY1

(c) TC

(d) TLH (and shadows)

(e) HV

Figure 7-4: Sample 3D prints for designs customized with this system.

Conclusion

We presented a practical approach for constructing end-user-customizable 3D-printable object representations from constrained parametric designs, and have shown that it is, indeed, feasible. My approach precomputes Fab Form representations that enable automatic creation of easy customization applications.

Appendix A

Implementation Details

The operation nodes in this system fall into the following categories:

- `Create Shape` (all leaves have this category), e. g. Box
- `Modify Shape`, e. g. Scale
- `Combine Shapes`, e. g. CSG Union
- `Conditional`
- `Loop`

where bolded categories are abstract C++ classes that can have any number of concrete implementations. The data structures are flexible enough to work with arbitrary node implementations, simply linked into the binary. To demonstrate my method, I implemented `Combine` nodes for CSG operations and several custom `Create` nodes, including a `CreateOpenScad` node that can evaluate arbitrary [30] scripts. The implementations rely on Triangle [38], [7], [9], and, of course, [30].

Each node implementation is governed by a unique set of parameters from a small number of types (`int`, `bool`, `double`, `vector`).[1] To deal with such different sets of parameters in a consistent way, I use protocol buffers [16] that enable one to get and set parameters by type and name.

In addition, protocol buffers provide a handy way to read and write trees of operations to and from file (i.e. using protobuf `MessageSet`), and to save and recombine

[1]For some node implementations the set of parameters varies among instances of the same implementation (e.g. `CreateOpenScad` nodes have parameters that depend on the input script).

information about the valid region after parallel precomputation. Formatted protocol buffers with fixed double precision are also used as keys to the `GeometryCache`.

Likewise, precomputation is implemented in a fully general way to accept designs of variable parameter types (`bool`, `double`, `int`) and dimensionality of the parameter space. In order to deal with continuous and discrete dimensions smoothly, a base C++ class `Dimension` is used. Subclasses of `Dimension` keep track of cuts along their dimension and report whether or not a new subdivision can be made. Each sample point is treated as a vector of doubles, but internally stores the double, integer and boolean values.

The Gecode library [35] is used to solve for linear and non-linear constraints, and OpenVDB [28] is relied on for volumetric operations needed for the tests and ΔG computation.

Appendix B

ΔG as a Metric Space

Lemma: ΔG is a metric, where ΔG is defined as the symmetric difference (XOR) of the volumes of two shapes A and B, normalized by the volume of their union:

$$\Delta G(A, B) \triangleq \frac{|A \oplus B|}{|A \cup B|}$$

Proof:

Non-negativity, symmetry: hold trivially.

Coincidence: $\Delta G(A, B) = 0 \Leftrightarrow A = B$. ΔG is only zero if the symmetric difference of two shapes is zero, and that holds if and only if two volumes are identical.

Triangle inequality: $\Delta G(A, C) \leq \Delta G(A, B) + \Delta G(B, C)$ for all A, B, C. For any A, B, C, w. l. g. let the space be partitioned according to which of the three shapes overlap in that region of space, as shown in the Fig. B-1, where each variable refers to the volume of the corresponding region. Then, we can trivially state the following

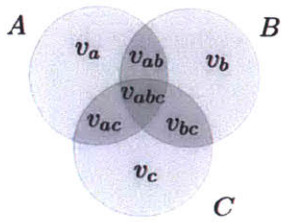


Figure B-1: Space partitioned by three volumetric shapes, with regions of overlap (potentially empty) labeled by variables.

identities:

$$|A \oplus C| = |A \oplus B| + |B \oplus C| - (2v_b + 2v_{ac}) \tag{B.1}$$

$$|A \cup B| = |A \cup C| + v_b - v_c \tag{B.2}$$

$$|B \cup C| = |A \cup C| + v_b - v_a \tag{B.3}$$

Consider the expanded triangle inequality:

$$\Delta G(A,C) \stackrel{?}{\leq} \Delta G(A,B) + \Delta G(B,C)$$
$$\frac{|A \oplus C|}{|A \cup C|} \stackrel{?}{\leq} \frac{|A \oplus B|}{|A \cup B|} + \frac{|B \oplus C|}{|B \cup C|}$$

Then, applying identities B.1-B.3 we obtain:

$$\frac{|A \oplus B|}{|A \cup C|} + \frac{|B \oplus C|}{|A \cup C|} - \frac{2(v_b + v_{ac})}{|A \cup C|} \stackrel{?}{\leq}$$
$$\frac{|A \oplus B|}{|A \cup C| + v_b - v_c} + \frac{|B \oplus C|}{|A \cup C| + v_b - v_a}$$

Then carrying over and combinding terms:

$$\frac{-2(v_b + v_{ac})}{|A \cup C|} \stackrel{?}{\leq}$$
$$\frac{(v_c - v_b) \cdot |A \oplus B|}{|A \cup C| \cdot (|A \cup C| + v_b - v_c)} + \frac{(v_a - v_b) \cdot |B \oplus C|}{|A \cup C| \cdot (|A \cup C| + v_b - v_a)}$$

Then simplifying and reapplying identities B.2 and B.3 in reverse:

$$-2(v_b + v_{ac}) \stackrel{?}{\leq} \frac{(v_c - v_b) \cdot |A \oplus B|}{|A \cup B|} + \frac{(v_a - v_b) \cdot |B \oplus C|}{|B \cup C|}$$

Then, reversing the inequality and collapsing the terms back:

$$2(v_b + v_{ac}) \stackrel{?}{\geq} (v_b - v_c) \cdot \Delta G(A,B) + (v_b - v_c) \cdot \Delta G(B,C)$$

It is sufficient to prove that the left expression is greater than or equal to the maximum

possible value of the right expression. This happens when both terms on the right are positive, i. e. when $v_b > v_c$ and $v_b > v_a$, and when both ΔG terms attain the maximum possible value of 1. Thus it remains to show:

$$2v_b + 2v_{ac} \stackrel{?}{\geq} 2v_b - v_c - v_a$$

$$2v_{ac} \geq -(v_c + v_a)$$

This is trivially true, because all the volumes are nonnegative. $\square$

Bibliography

[1] Amazon. Amazon web services. http://aws.amazon.com/.

[2] Autodesk. Autocad. http://www.autodesk.com/products/autodesk-autocad.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] Bernhard Bettig and Christoph M Hoffmann. Geometric constraint solving in parametric computer-aided design. *Journal of computing and information science in engineering*, 11(2):021001, 2011.

[5] Rafael Bidarra and Willem F Bronsvoort. Semantic feature modelling. *Computer-Aided Design*, 32(3):201–225, 2000.

[6] Martin Bokeloh, Michael Wand, Hans-Peter Seidel, and Vladlen Koltun. An algebraic model for parameterized shape editing. *ACM Transactions on Graphics (TOG)*, 31(4):78, 2012.

[7] Carve csg. http://carve-csg.com.

[8] Siddhartha Chaudhuri, Evangelos Kalogerakis, Leonidas Guibas, and Vladlen Koltun. Probabilistic reasoning for assembly-based 3d modeling. In *ACM Transactions on Graphics (TOG)*, volume 30, page 35. ACM, 2011.

[9] Clipper. http://www.angusj.com/delphi/clipper.php.

[10] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller, and Robert Jagnow. A procedural approach to authoring solid models. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 302–311. ACM, 2002.

[11] Dassault Systèmes. Solidworks. http://www.solidworks.com/.

[12] Peter Doubilet, Colin B Begg, Milton C Weinstein, Peter Braun, and Barbara J McNeil. Probabilistic sensitivity analysis using monte carlo simulation. a practical approach. *Medical decision making: an international journal of the Society for Medical Decision Making*, 5(2):157–177, 1984.

[13] dreamforge. Cookie caster. http://cookiecaster.com.

[14] Ran Gal, Olga Sorkine, Niloy J Mitra, and Daniel Cohen-Or. iwires: an analyze-and-edit approach to shape manipulation. In *ACM Transactions on Graphics (TOG)*, volume 28, page 33. ACM, 2009.

[15] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of books in the mathematical sciences. W. H. Freeman, 1979.

[16] Google. Protocol buffers. http://code.google.com/apis/protocolbuffers/.

[17] Marta Hidalgo and Robert Joan-Arinyo. Computing parameter ranges in constructive geometric constraint solving: Implementation and correctness proof. *Computer-Aided Design*, 44(7):709–720, 2012.

[18] Christoph M Hoffmann and K-J Kim. Towards valid parametric cad models. *Computer-Aided Design*, 33(1):81–90, 2001.

[19] Sheena S. Iyengar and Mark R Lepper. Rethinking the value of choice: A cultural perspective on intrinsic motivation. *Journal of Personality and Social Psychology*, 76(3):349–366, 1999.

[20] Alec Jacobson, Ilya Baran, Jovan Popovic, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.*, 30(4):78, 2011.

[21] The jQuery Foundation. jquery. http://jquery.com/.

[22] Doyub Kim, Woojong Koh, Rahul Narain, Kayvon Fatahalian, Adrien Treuille, and James F O'Brien. Near-exhaustive precomputation of secondary cloth effects. *ACM Transactions on Graphics (TOG)*, 32(4):87, 2013.

[23] Vladislav Kraevoy, Alla Sheffer, Ariel Shamir, and Daniel Cohen-Or. Non-homogeneous resizing of complex models. In *ACM Transactions on Graphics (TOG)*, volume 27, page 111. ACM, 2008.

[24] Tomas Lozano-Perez. Spatial planning: A configuration space approach. *Computers, IEEE Transactions on*, 100(2):108–120, 1983.

[25] MakerBot. Thingiverse customizer. http://www.thingiverse.com/apps/customizer.

[26] Makieworld Ltd. Makies: create your own doll! http://makie.me.

[27] Joe Marks, Brad Andalman, Paul A Beardsley, William Freeman, Sarah Gibson, Jessica Hodgins, Thomas Kang, Brian Mirtich, Hanspeter Pfister, Wheeler Ruml, et al. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 389–400. ACM Press/Addison-Wesley Publishing Co., 1997.

[28] Ken Museth. Vdb: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics (TOG)*, 32(3):27, 2013.

[29] Nervous System. Radiolaria: bio-inspired design app. http://n-e-r-v-o-u-s.com/radiolaria.

[30] Openscad. http://www.openscad.org.

[31] Jia Pan, Xinyu Zhang, and Dinesh Manocha. Efficient penetration depth approximation using active learning. *ACM TRANSACTIONS ON GRAPHICS*, 32(6), 2013.

[32] Romain Prévost, Emily Whiting, Sylvain Lefebvre, and Olga Sorkine-Hornung. Make it stand: balancing shapes for 3d fabrication. *ACM Transactions on Graphics (TOG)*, 32(4):81, 2013.

[33] Ryan Schmidt and Karan Singh. Sketch-based procedural surface modeling and compositing using surface trees. In *Computer Graphics Forum*, volume 27, pages 321–330. Wiley Online Library, 2008.

[34] Ryan Schmidt, Brian Wyvill, and Eric Galin. Interactive implicit modeling with hierarchical spatial caching. In *Shape Modeling and Applications, 2005 International Conference*, pages 104–113. IEEE, 2005.

[35] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with gecode, 2010. http://www.gecode.org/.

[36] Shapeways. Sake set creator. https://www.shapeways.com/creator/sake-set.

[37] Shapeways. Shapejs, 2014. http://shapejs.shapeways.com/.

[38] Jonathan Richard Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Applied computational geometry towards geometric engineering*, pages 203–222. Springer, 1996.

[39] Olga Sorkine, Daniel Cohen-Or, Yaron Lipman, Marc Alexa, Christian Rössl, and H-P Seidel. Laplacian surface editing. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 175–184. ACM, 2004.

[40] Ondrej Stava, Juraj Vanek, Bedrich Benes, Nathan Carr, and Radomír Měch. Stress relief: improving structural strength of 3d printable objects. *ACM Transactions on Graphics (TOG)*, 31(4):48, 2012.

[41] three.js. http://threejs.org/.

[42] Nobuyuki Umetani, Takeo Igarashi, and Niloy J Mitra. Guided exploration of physically valid shapes for furniture design. *ACM Trans. Graph.*, 31(4):86, 2012.

[43] Nobuyuki Umetani, Danny M Kaufman, Takeo Igarashi, and Eitan Grinspun. Sensitive couture for interactive garment modeling and editing. *ACM Trans. Graph.*, 30(4):90, 2011.

[44] Nobuyuki Umetani and Ryan Schmidt. Cross-sectional structural analysis for 3d printing optimization. In *SIGGRAPH Asia 2013 Technical Briefs*, SA '13, pages 5:1–5:4, New York, NY, USA, 2013. ACM.

[45] Hilderick A Van der Meiden and Willem F Bronsvoort. A constructive approach to calculate parameter ranges for systems of geometric constraints. *Computer-Aided Design*, 38(4):275–283, 2006.

[46] Hilderick A van der Meiden and Willem F Bronsvoort. Solving topological constraints for declarative families of objects. *Computer-Aided Design*, 39(8):652–662, 2007.

[47] Kevin D Wise and Adrian Bowyer. 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, 2000.

[48] Y-T Wu. Computational methods for efficient structural reliability and reliability sensitivity analysis. *AIAA journal*, 32(8):1717–1723, 1994.

[49] Brian Wyvill, Andrew Guy, and Eric Galin. Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system. In *Computer Graphics Forum*, volume 18, pages 149–158. Wiley Online Library, 1999.

[50] Zaphoyd Studios. Websocket++. http://www.zaphoyd.com/websocketpp.

[51] Qingnan Zhou, Julian Panetta, and Denis Zorin. Worst-case structural analysis. *ACM Transactions on Graphics (TOG)*, 32(4):137, 2013.