DOI: 10.1111/cgf.14476
EUROGRAPHICS 2022 / R. Chaine and M. H. Kim
(Guest Editors)

Volume 41 (2022), Number 2

Differentiable 3D CAD Programs for Bidirectional Editing

D. Cascaval[1], M. Shalah[2], P. Quinn[1], R. Bodik[1], M. Agrawala[2] and A. Schulz[1]

[1]University of Washington
[2]Stanford University

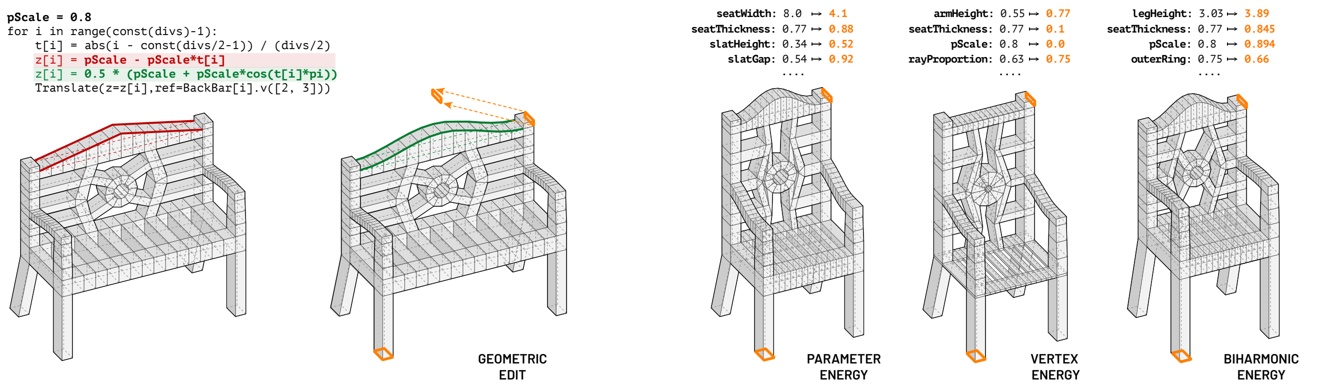

Figure 1: *A user bidirectionally edits a model of a bench. They modify the program to change the sharp backrest (red) to a smooth one (green). Then, the user intends to rapidly change the bench into a chair. To accomplish this the user performs a geometric edit (orange), dragging the top part of the leg up and to the left, while also fixing in place vertices on the opposite leg. Our system optimizes program parameters to achieve this edit according to a variety of energy minimization heuristics (Section 3.3); returning distinct options for the user to select between. Each option changes between 8 to 12 parameters (top) from the original program to satisfy the edit.*

Abstract

Modern CAD tools represent 3D designs not only as geometry, but also as a program composed of geometric operations, each of which depends on a set of parameters. Program representations enable meaningful and controlled shape variations via parameter changes. However, achieving desired modifications solely through parameter editing is challenging when CAD models have not been explicitly authored to expose select degrees of freedom in advance. We introduce a novel bidirectional editing system for 3D CAD programs. In addition to editing the CAD program, users can directly manipulate 3D geometry and our system infers parameter updates to keep both representations in sync. We formulate inverse edits as a set of constrained optimization objectives, returning plausible updates to program parameters that both match user intent and maintain program validity. Our approach implements an automatically differentiable domain-specific language for CAD programs, providing derivatives for this optimization to be performed quickly on any expressed program. Our system enables rapid, interactive exploration of a constrained 3D design space by allowing users to manipulate the program and geometry interchangeably during design iteration. While our approach is not designed to optimize across changes in geometric topology, we show it is expressive and performant enough for users to produce a diverse set of design variants, even when the CAD program contains a relatively large number of parameters.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Shape modeling; Graphics systems and interfaces;***

1. Introduction

Parametric Computer-Aided Design (CAD) models are ubiquitous in engineering. Modern CAD tools (Solidworks, Fusion 360, Onshape, Inventor) represent part geometry using a sequence of parameterized operations, which are executed to produce a 3D shape. Such sequences can be thought of as programs, which offer a form of structured editing: engineers can change operation parameters and the geometry will update without a user having to manually model each component of the output. This is especially powerful when automatically generating repetitive model components such as grids or patterns, and as a module of reuse between different geometric designs. However, manipulating models solely through parameter changes is often cumbersome. Editing a model to achieve a specific design change might require changing many different operation parameters in concert, and real-world manipulations can involve manipulating tens of parameters in a model made up of hundreds of distinct operations.

In contrast, directly manipulating geometry by moving vertices or faces in 3D allows users to specify their desired output without having to translate design intent into a given model's parameter space. The trade-off is that direct manipulation is often imprecise and requires the user to directly model all aspects of a change, which can be both tedious and error-prone, resulting in a model which is inconsistent or physically inaccurate. However, direct manipulation does offer a number of advantages: feedback is immediate, changes are limited to what the user manipulates, and it is clear how to edit any subsection of the model.

Neither editing paradigm is suitable for all manipulations; the right tool for the job depends on the edit itself, which cannot be known in advance. The primary contribution of this work is merging the two paradigms. We extend program-based CAD modeling to allow *bidirectional* editing: users can either adjust program operations, or directly manipulate a chosen subset of the output geometry and obtain corresponding operation parameters. As a result, CAD users and tools can leverage whichever representation is more convenient for a given manipulation. To interleave direct and program edits, it is critical that direct manipulations preserve program structure. Accordingly, in this work we focus on resolving direct manipulations in terms of model parameter changes, without changing program structure, and focus on interactive performance to allow for rapid iterations and geometric adjustments.

The fundamental challenge in designing this bidirectional interface is the gap between tens or hundreds of parameters exposed when composing constructive CAD operations into programs, and the handful of degrees of freedom that users may wish to manipulate during a given edit. While prior work has relied on laborious manual specification of these degrees of freedom and their valid ranges [SXZ*17; MB21] our goal is to enable direct editing through optimization over the constructive parameters themselves. When parameter interdependencies and constraints have not been manually exposed by an expert, there tend to be two main problems: ambiguity, as many plausible solutions can match a direct geometric edit, and safety, as many other parameter combinations result in invalid geometry or execution failure.

To address ambiguity, we formulate the inverse editing problem as a series of constrained optimizations over a set of heuristics, and provide a suggestive interface for selecting the desired solution. Interactive optimization times are challenging to achieve due to the dimensionality of the parameter space and the need to solve many optimization problems in parallel, one for each heuristic. To solve this optimization quickly, we developed a Domain-Specific Language (DSL) for CAD models that is automatically differentiable, aiming to express many common geometric operations while being able to provide derivatives for any expressible program. We limit our language to operators which avoid discontinuities and cannot fail to produce a result.

Since naively calling automatically differentiated CAD kernels is impractical in an editing context due to performance concerns [Ban19], our solution evaluates model code to produce a computation graph that does not refer to the underlying CAD kernel. This in turn enables optimization on the generated code without differentiating the underlying engine, or incurring any of its extra overheads while optimizing (Section 4).

In summary, our primary contribution is **a system for bidirectional editing of 3D CAD models with interactive performance**. We demonstrate how such a system can represent CAD models using a domain-specific language, and describe our solutions to the specific engineering concerns that must be addressed to make it usable in practice. We evaluate our system's editing capability and performance on a series of case study models, demonstrating its efficacy at finding viable parameter combinations to satisfy edits on models with tens of parameters and thousands of vertices.

2. Background and Related Work

Editing program-based models is of interest to the Computer Graphics, CAD, and Programming Languages communities. Each of these fields have contributed their own ideas and work on the problem, and our work draws inspiration from all three.

2.1. Editing CAD Models

Many modern CAD tools (Solidworks, Fusion 360, Inventor, Onshape) use programs in the form of a history of operations that are executed to construct a model. These are called *history-based* systems. To address the challenges of editing such CAD programs, some of these systems incorporate direct edits (e.g. moving a face) by appending operations to the history. Adding operations in this manner is limited to predefined manipulations, and changes the *structure* of the CAD program and types of shape variations it can produce – they may break existing symmetries, and force users to consider the appended operations when making changes to the model.

Another approach is to ignore the CAD program entirely and directly edit the geometry, without preserving any notion of history (e.g. SpaceClaim, KeyCreator, and Rhino). This *history-free* approach loses the benefits provided by the program representation [Alb18; Yar13]. Siemen's Synchronous Technology takes a hybrid approach and integrate history-free geometric editing on top of a program/history [Sie17]. However, by partially disregarding the history, the CAD model can no longer be expressed or analyzed as a single program that defines semantically meaningful shape variations [Str16; Bru14]. In practice, this means that after a model is manipulated directly, existing constraints (such as symmetry and spacing) may be violated, and unexpected breakages may occur

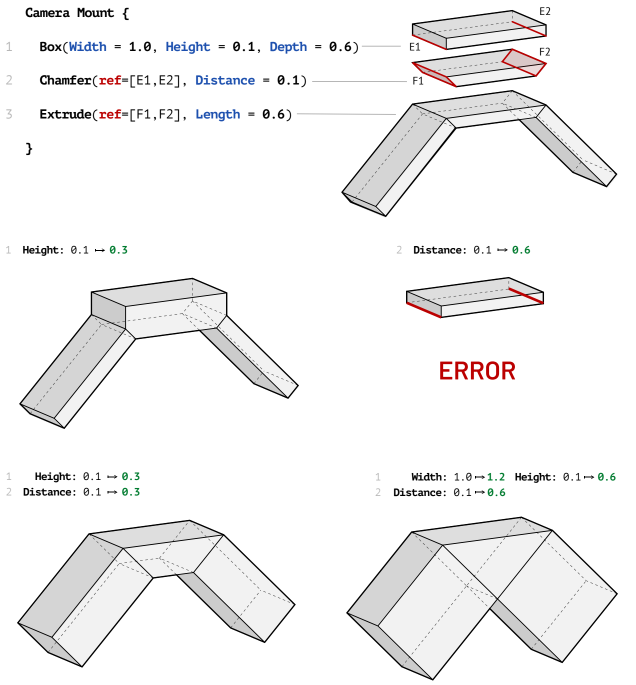

Figure 2: *Attempting to thicken the camera mount uniformly by changing a single parameter (middle left) loses the overall structure of the model as the thickness of the base differs from the thickness of the legs. If we instead modify the chamfer radius (middle right) the program does even run to completion, as it would result in non-manifold geometry. To perform thickening, our tool automatically updates multiple parameters in concert to achieve the desired properties (bottom) while staying within the execution constraints.*

when the part is later updated parametrically and a manipulation cannot be reconciled.

Unlike these methods, our *bidirectional* editing approach preserves the history as an explicit program and allows the user to seamlessly switch between editing the program structure and parameters, and directly editing its geometry. Procedural modeling as done in software such as Houdini [Sid] and Grasshopper [MA] already operates directly at the program level, and is popular in 3D work outside of CAD. Procedural models rely on users to explicitly expose parameters to edit the generation of scene geometry. Our work can be seen as a way to augment these tools, which already contain the requisite information, with bidirectional functionality.

2.2. Program Editing

While prior work in programming languages has investigated solutions to bi-directional program editing [HSST11] in domains like layout design and SVG, our method differs in how we address *ambiguity* that results from a user manipulation.

Such methods eliminate ambiguities by either restricting user interactions [HBR14] or pre-defining which parameters a manipulation will affect [CHSA16; HLC19]. Our method instead aims to expose more to the user by allowing arbitrary manipulations and returning a variety of different solutions in the case of ambiguity.

There has also been increased effort to reconstruct CAD programs from existing geometry [DIP*18; NWP*18; WPL*20], infer programs to construct assemblies [JBX*20], infer higher-level operations from those programs [JCG*21], and re-write CAD programs to expose meaningful parameters [NWA*20]. While none of these methods propose solutions to interactive manipulation, they are critical for enabling wide application of our work: can be modified to generate and refine a CAD program suitable for input to our system, growing the potential application of bidirectional editing beyond human-written programs.

2.3. Procedural Model Editing

Shape-generating programs are known in the graphics community as *procedural models* [WWSR03; MWH*06; LWW08; BFH05]. Controlling procedural models, however, is a notoriously challenging task [STBB14]. Many approaches on inverse procedural modeling target program generation from input specifications [TLL*11; VGA*12; GJB*20], but do not allow direct editing. While there is some work on local editing from program analysis [JPCS18; LSL*19], none of these methods allow users to control geometric edits directly while maintaining a working program. In concurrent work to ours, Gaillard et al [GKG*22] enable direct editing at the granularity of objects, and directly tackle exploring ambiguous edits made possible in part by a fast objective function. In contrast, our work enables editing at a vertex granularity while relying on heuristics to address ambiguity.

Of particular note is the DAG Amendment method described by Michel et al [MB21], which also allows inverse control of procedural models generated by a DAG of operations, using a brush interaction to address edit ambiguity. This work handles operations that can change the output mesh topology as long as an unambiguous mapping from output to input UV coordinates is maintained. Such generality is achieved by performing differentiation using finite differences, which is performant for models that have been designed to expose a small set of degrees of freedom in advance, and handles a wide variety of operations for which it is unclear how to differentiate automatically.

In contrast, our work pursues the opposite tradeoff: we do not handle operations that can result in topological changes, and instead we apply fully automatic differentiation, allowing our approach to scale up to models of tens or hundreds of parameters where finite differences would be prohibitively expensive. This efficiency allows us to perform a full non-linear optimization interactively when resolving edits. As a result, our work can handle models where parameters have not been manually tuned for editing, models where the valid ranges for parameters vary as functions of other parameters, and edit specifications where some model geometry can be fixed in place for an edit duration while multiple other parts are changed.

2.4. Structure-Driven Geometry Editing

There is a large body of work on structure-preserving geometric edits using an analyze-and-edit approach to constrain the shape without any underlying program [MWZ*14]. Previous work has analyzed salient features [KSSC08], feature curves [GSMC09], relations between shape parts [ZFC*11], replicated patterns [BWKS11; BWSK12], and manufacturability [SSL*14] as methods of describing and preserving shape structure.

Our work builds upon these ideas to define inverse editing as an optimization. However, unlike previous approaches, our method extends geometric analysis with constraints and objectives encoded through a program construction. We show how encoding programs in our differentiable DSL can produce improved results compared to applying traditional geometric optimization followed by parameter synchronization (see Figure 5).

Furthermore, instead of a single optimization, our system allows users to navigate a gallery of results that are generated by prioritizing different geometric properties in the optimization goals.

2.5. Differentiable Programming

Recent years have seen an explosion in the application of differentiable programming techniques in Machine Learning, particularly through automatic differentiation [GW08]. Packages such as PyTorch [PGM*19] provide an interface by which a computation graph is automatically constructed by executing user code, and derivative functions are automatically derived from this graph. Others such as JAX [FJL18] and XLA [Ten20] go a step further, and enable compilation of user functions and their automatic derivatives from high-level dynamic languages into native code for fast optimization and training. Our work takes a similar strategy to these tools, tailored towards the specific optimization problems in our domain. Existing work using automatic differentiation in computer graphics applications focuses on formulating or recreating traditional graphics pipelines (e.g. rasterization, rendering) as differentiable functions [LLGR20; NVZJ19], and using it them to solve inverse problems on procedural models [SLH*20]. Others demonstrate applications that arise from operating directly on structured representations such as vectors or scene primitives, at a higher level than pixel data [RGF*20]. Other work performs finite-element optimization directly on parametric CAD models [HSK*19] by providing differentiable physical simulations. Our work extends both of these lines of work, providing a language composed of differentiable operators that can express a wide range of CAD models, and derivative information for any program in the language. As a result we enable optimization directly on a the structured program representation during the model design and editing loop.

3. Interactive Editing System

Our goal is to enable interactive bidirectional editing of a CAD program and the 3D geometry it generates. When a user edits the program, our system interprets the edited program to generate the corresponding updated geometry. When the user directly edits the geometry by manipulating vertices, edges, or faces, our system finds sets of updated parameters such that the program, when executed with these parameters, produces geometry where the manipulated elements are as close as possible to the user's edit.

To accomplish this, we (1) specify 3D models as programs in a

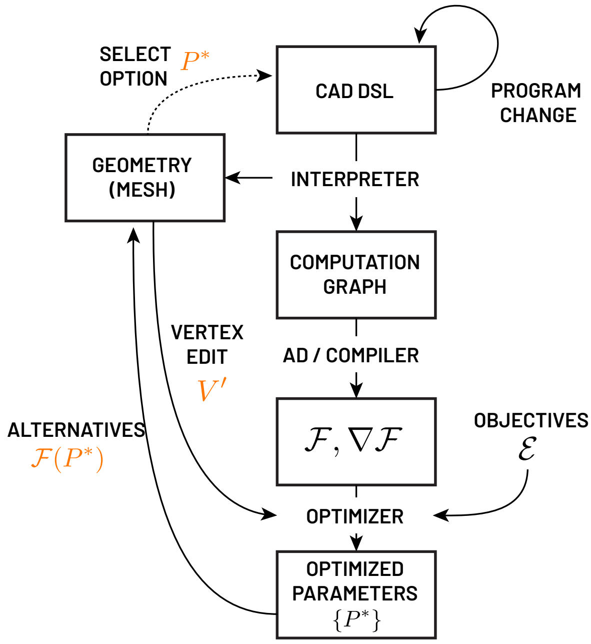

Figure 3: Pipeline. *A program is interpreted to create mesh geometry and a corresponding computation graph. We extract derivatives from this graph and compile a function $\mathcal{F}$ from parameters to vertices and its derivative for use in optimization with several different energy functions. Users can then edit the mesh, and receive updated sets of program parameters P^* corresponding to each energy. Parameters are then used to compute updated mesh vertex positions $\mathcal{F}(P^*)$ matching the user edit. Users choose an option and can continue editing geometrically, as well as update the program with the found parameters.*

CAD language designed to be differentiable; (2) interpret this language to generate both geometry as well as a computation graph representation of the program, which is then automatically differentiated and compiled into native code; (3) optimize over program parameters using a set of heuristic objectives that take into account both program and geometric information, serving as proxies for properties ranging from appearance to physical performance. This pipeline is illustrated in Figure 3.

3.1. Language Design

Our language makes a few key design choices motivated by the domain and specific problem requirements of bidirectional editing.

Explicit Geometric Representation. It is common in the CAD domain for users to reference features of previously constructed geometry, such as edges or faces. This is often necessary in order to concisely describe complex CAD models with a small number of parameters. In our system, we use a polygonal mesh representation, and users can use vertex positions as parameters in subsequent operations, and reference their values when defining constraints. Relying on an explicit mesh representation makes it easier to define a wide variety of optimization objectives that refer to explicit features of output geometry, such as minimizing the distance to vertices manipulated by a user or minimizing the deformation from the starting mesh.

Static Model Topology. However, such an explicit representation comes at a cost. Since we allow user programs to refer directly to output vertices by index, in order to provide differentiability during inverse editing, we must guarantee that a given operation returns the same number of vertices for all parameter combinations – if this number were allowed to change during optimization, it would be unclear which vertices are being referred to by an index reference. The same logic applies to the order of an operation's returned vertices. Both of these cases are discrete discontinuities in the model, and it is not clear how to optimize such a program without differentiating repeatedly during each optimization step. We therefore only allow topological changes to be performed on the program, and not inferred through geometric manipulations.

Our DSL can include any operation which creates, modifies, or removes a statically known amount of vertices and faces. Positions of vertices that are created or modified must be differentiable functions of parameters and prior vertex positions. We guarantee that any point in the parameter space of any program in our language is differentiable. Degenerate or zero-size mesh elements do not pose any issue in this framework, and are thus handled robustly. Working within the limitations of this static topology allows us to compile energy functions ahead of time, drastically increasing performance.

Constraints. We also allow direct specification of numerical inequality constraints to further limit the parameter space considered during optimization. Constraints are differentiable functions of model parameters whose result must be kept greater than zero (Section 3.2). Constraints are imposed automatically for some operations, and we expose a mechanism for users to further impose constraints. These constraints are treated as hard requirements, and any result returned by our optimization necessarily respects them. Constraints satisfy three key goals:

Firstly, we would like to avoid parameter changes that lead to non-manifold geometry, which would cause objective functions that are defined on properties such as volume to return inaccurate results. Some constraints to prevent this are automatically generated by the interpreter as part of operation definitions, as we describe in Section 3.2. While not all non-manifold cases can be prevented, applying the automatically-generated constraints rules out the optimization procedure from returning a large class of undesirable parameter combinations by default.

Secondly, users can apply constraints in order to refine the allowable variations in the program; by eliminating self-intersections or other undesirable model variations if these come up in practice.

Using constraints, users can impose additional design objectives such as maintaining relative proportions, aspect ratios, or distances between mesh elements; without changing the structure of their program to enforce these properties by construction. In contrast to other work such as ThingLab [Bor81], users must still directly construct and specify numerical quantities to bound, and the system does not allow equality constraints or deal with constraint conflicts, nor will it prevent the user manually changing parameters in a way that does not satisfy the constraints. Constraints can refer to the positions of vertices in the model, and thus can re-use the computations made in the construction of the model itself.

3.2. Interpretation and Differentiation

Computation Graph. When executing our CAD DSL program to generate output geometry, our algorithm operates as an interpreter. During the execution of each program operation, in addition to computing the result of the operation, our interpreter also keeps track of the individual computations that are performed, maintaining a directed computation graph G (Figure 4).

Each node in G is either a constant, a variable, or a differentiable operator f, (e.g. `+`, `*`, `sin`, `pow`). Each node tracks the temporary value resulting from the execution of its operator, and each graph edge represents a *use* of this value. A vertex coordinate is fully encoded by marking a node in the graph as representing the computation for that coordinate.

Each CAD operation in our program (e.g. `Box`, `Chamfer`, `Extrude`) adds nodes to this graph, often directly referencing the values of vertices output by previous operations as the basis for their computation, as depicted in Figure 4. Each operation also modifies the markers corresponding to vertices, by either adding additional markers for new vertices, moving markers to new nodes to represent vertex transformations, or removing markers (e.g. V_3 of `Chamfer` in Figure 4) to indicate that, while a computation may still be present and referred to, it no longer corresponds to a vertex.

Note that our method only tracks vertex positions, and does not deal with aspects of mesh topology such as edges and faces. These are created and tracked by an underlying CAD system. This process is described in more detail in Section 4. For the purpose of inverse editing this topology is static, and we can fully describe parameter updates by adjusting vertex positions.

Enforcing Constraints. Along with computing and tracing vertex positions, for certain operations our interpreter automatically introduces constraints on the parameters as part of the operation definition. For example, the *length* parameter of an `Extrude` operation is constrained to always remain larger than some positive epsilon, ensuring geometry is extruded outwards along a face's normal. Accordingly, our interpreter encodes the constraint as the value $length - \varepsilon$, which is added to the graph, and marked as a constraint in a fashion identical to vertex coordinates. Similar constraints are generated for `Chamfer`, preventing the radius from exceeding the length of either edge being chamfered (g_2, g_3 in Figure 4).

During optimization, we require nodes marked as constraints to remain positive. As a result, this expression implicitly represents a constraint of the form $g(P_1, \ldots, P_k) \geq 0$. This can refer to vertex coordinates, since every graph node is ultimately a function of the parameters.

In addition to automatically-applied constraints, we expose a `Clamp` operation to the program author, which can be used to set upper and lower bounds on the allowed values of an arbitrary parameter expression, for example, bounding the maximum aspect ratio for the rectangle shown in Figure 4, or bounding the range of a vertex coordinate. When executing `Clamp`$(\alpha, f(P_1, \ldots, P_k), \beta)$, we

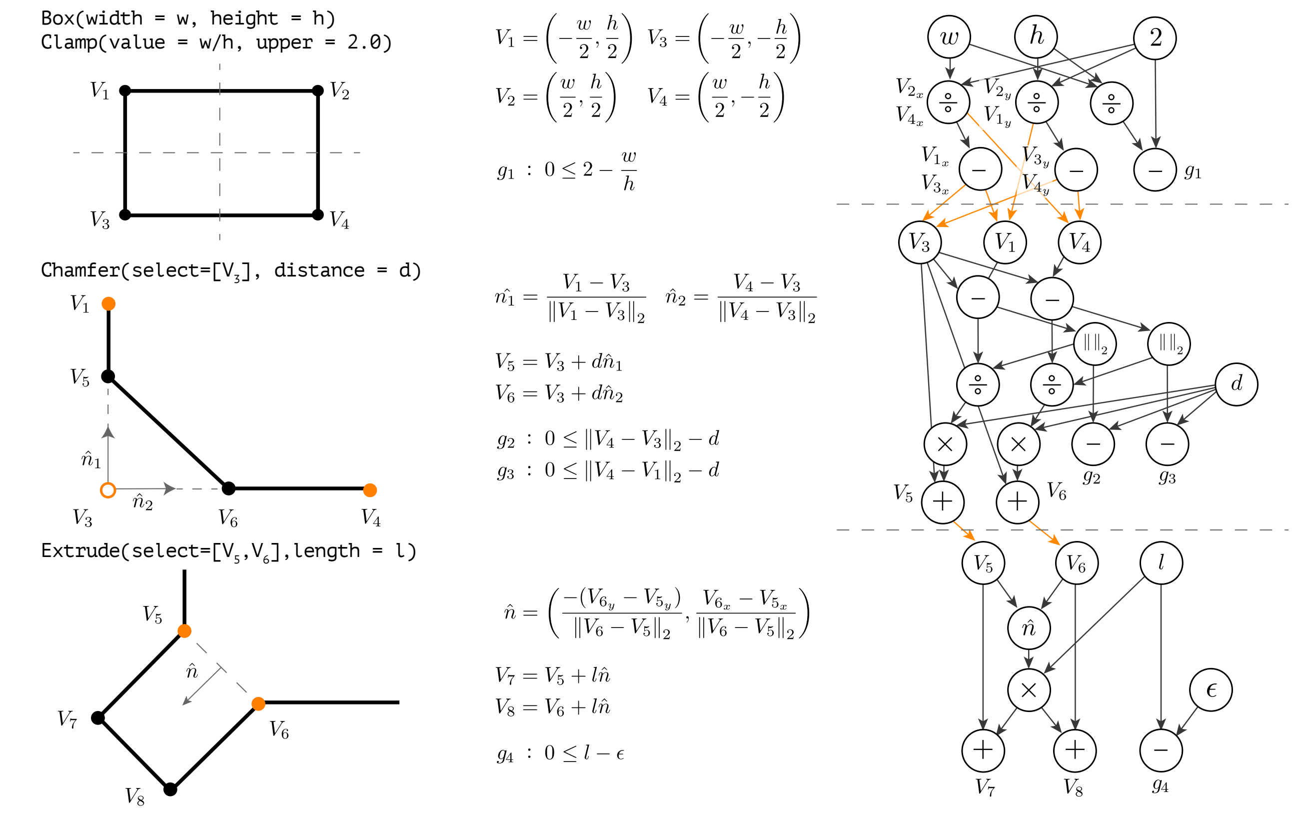

Figure 4: *To construct the computation graph G for this 2D version of our camera mount program (left), we trace the operations in our DSL (e.g.* `Box`*,* `Clamp`*,* `Chamfer`*) to generate expressions that create and modify vertex positions (e.g.* V_1*,* V_2*, ...) based on the operation parameters (middle). We encode these expressions in the computation graph G (right). Each operation* references *previously constructed geometry (orange) and imposes constraints (e.g.* g_1*,* g_2*). We note that, for concision, the nodes in the* `Chamfer` *and* `Extrude` *operations are operating on vertex values, rather than scalars. They can be treated as pairwise operations on the constituent elements.*

generate constraints $f(P_1, \ldots, P_k) - \alpha \geq 0$ and $\beta - f(P_1, \ldots, P_k) \geq 0$, with f representing the parameter expression whose value we are bounding, and α, β representing expressions for lower and upper bounds on f, respectively.

Differentiation and Compilation. G implicitly represents a function $\mathcal{F}$ from our program parameters P to the positions of output vertices V. We can evaluate $\mathcal{F}$ by updating the values of the variable nodes, traversing G to recompute the values for all nodes whose values have been invalidated as a result of a change in an upstream node, and then visiting all of the marked nodes to extract the updated vertex coordinates. Once G is constructed, we extract the gradient $\nabla\mathcal{F}$ through reverse-mode automatic differentiation [Mar19], giving us the gradient vector of program parameters with respect to each vertex coordinate. We can extract the parameter gradient with respect to each constraint as well, which is critical for performant and stable optimization.

As a final step in our interpretation pipeline, we compile $\mathcal{F}$, $\nabla\mathcal{F}$, the constraint functions, and their respective gradients to an explicit native code representation, which is passed off to an optimizing compiler. As a result, we can execute $\mathcal{F}$ and $\nabla\mathcal{F}$ several orders of magnitude more quickly than by traversing G, as all of the overhead of the graph representation has been compiled away. Compilation is necessary to maintain interactivity during program editing and synchronization, which evaluates $\mathcal{F}$ composed with each objective function (Section 3.3) repeatedly. Compilation can take significantly more time than interpretation, so instead of compiling after every program change, we implement this as a separate step that a user performs in order to begin synchronizing the model with geometric edits. Consecutive synchronization steps can re-use a compiled representation until the program is changed, and users only ever need to compile before making a geometric edit immediately after a program edit.

3.3. Optimization

When users directly edit geometry, they adjust the positions of a set of vertices $V' \subset V$. Unless they manipulate all of the vertices, this is a *partial* specification: the user is providing no information about the vertices $v_i \notin V'$. There are two primary challenges in solving for the program parameters P, given this edit:

- The edit may not be *feasible*, if there does not exist any P such

that the specification above is met while also satisfying the program constraints $g_1, \dots, g_k$.
- There may be many sets of parameters P that satisfy the specification and it is *ambiguous* which one should be returned.

Our solution casts the problem as a constrained optimization. We address feasibility by formulating how close a set of parameters P is to satisfying a vertex edit as a continuous energy:

$$E_{\text{edit}}(P) = \sum_{v_i \in V'} \|V_i(P) - \bar{\mathbf{v}}_\mathbf{i}\|_2^2 \tag{1}$$

where $\bar{\mathbf{v}}_\mathbf{i} \in \mathbb{R}^3$ denotes the spatial position of vertex v_i after the user's edit and $V_i(P)$ denotes the position of v_i when the program is executed with parameters P.

However, the challenge of ambiguity remains. Optimization offers a solution here too: we formulate a number of heuristic objectives that can be combined with the edit energy to preserve characteristics of the model while adhering to the user manipulation. While our heuristics must be defined on the parameters P, by executing and differentiating $\mathcal{F}(P)$ we can define heuristic notions of change over vertex positions and optimize them through parameter changes. This ability to define change over geometry allows us to use any of a large set of well understood heuristics with our system as long as they are differentiable.

We describe a few heuristics included in our system and the characteristics they are designed to preserve. As not all objectives will be useful on all classes of models, we allow users to select a subset of the following objectives to apply. Additionally, we demonstrate that while the heuristics do not comprise a fully automatic solution to the ambiguity issue, they are capable of providing *diverse* solutions (Section 5), allowing users to explore the solution space.

3.3.1. Geometric Objectives

One category of heuristic is based on the assumption that the geometric features that are not manipulated should change as little as possible. One method for minimizing geometric change is to minimize the change in positions of the vertices that were not manipulated by the user, namely, $v_i \notin V'$. The corresponding energy term for this minimization is

$$E_{\text{vtx}}(P) = \sum_{v_i \in V \setminus V'} W_{\text{vtx}}(v_i) ||V_i(P) - V_i(P_0)||_1 \tag{2}$$

$$W_{\text{vtx}}(v_i) = \frac{\Delta(v_i, V')}{\sum_{u_i \in V \setminus V'} \Delta(u_i, V')} \tag{3}$$

where P_0 denotes the original program parameters prior to optimization. We weight the contribution of each vertex to the energy by a *localization* term W_{vtx}. $\Delta(v_i, V')$ is defined as the shortest geodesic distance between v_i and the unmanipulated position of any vertex in V', encoding the notion that users are more likely to want changes closer to the vertices they manipulated; as opposed to changes in geometry further away. We use an L_1 norm to prioritize sparsity, preferring to move a few vertex positions significantly rather than moving many vertices a little, as the L_2 norm would promote.

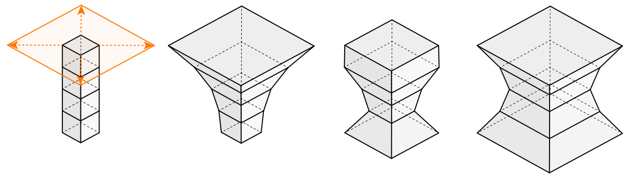

Figure 5: *For the model in (a), the XY coordinates of the vertices at the top and bottom loops are controlled by the same parameter. When the top vertices are edited (orange), (b) shows the result of minimizing the bi-harmonic energy on the vertex positions, disregarding the program. (c) is the result of using the vertex positions in (b) as V' to minimize our edit objective (Equation 1), whereas optimizing directly on the parameters produces the desired smooth result (d).*

Similarly, we can aim to preserve the lengths of edges in the mesh (V, E) not modified by the user, that is, the edges $e_{ij} \notin E'$, where $E' = \{e_{ij} \in E : v_i \in V' \wedge v_j \in V'\}$. The corresponding energy is:

$$E_{\text{edg}}(P) = \sum_{e_{ij} \notin E'} W_{\text{edg}}(e_{ij}) E_{\text{edg}}(e_{ij}) \tag{4}$$

$$E_{\text{edg}}(e_{ij}) = \left(\left\|V_i(P) - V_j(P)\right\|_2 - \left\|V_i(P_0) - V_j(P_0)\right\|_2\right)^2 \tag{5}$$

$$W_{\text{edg}}(e_{ij}) = \max(W_{\text{vtx}}(v_i), W_{\text{vtx}}(v_j)) \tag{6}$$

As in our vertex energy, we include a localization weight W_{edg} for each edge which is the maximum vertex weight of either of the edge's endpoints.

3.3.2. Deformation Objectives

A common approach in interactive surface modeling is to minimize *geometric deformation* for all local areas when the user manipulates a few vertices of a given mesh, potentially moving vertices far away from the user's edit to achieve a smoother overall transform. We demonstrate the application of two well-studied heuristics from the surface modeling literature to accomplish this for our system. The first heuristic is that the deformation should be as smooth as possible, which is achieved by minimizing the bi-harmonic energy [SCL*04]. The second heuristic targets a deformation that is as-rigid-as-possible (ARAP), i.e., it penalizes non-rigid transformations of the model [SA07]. ARAP is a physics-inspired energy that preserves volume and local structure.

A naive approach to minimizing these energies might be to perform the deformation directly on the geometry, and then treat the resulting vertex positions as V' for our edit objective. However, this approach bypasses the constraints of the program and leads to non-smooth results compared to optimizing directly on the parameters, as can be seen in Figure 5.

Instead, we optimize the bi-harmonic and ARAP energies directly on program parameters, which define our degrees of freedom for the deformation, instead of on vertex positions. For both deformation energies we first triangulate the original mesh. This only changes the connectivity, and the vertices can still be represented as functions of the parameters by invoking $\mathcal{F}(P)$. The bi-harmonic energy is then formulated as:

$$E_{bh}(P) = \mathrm{tr}(\mathbf{D}(P)^T \mathbf{Q}\mathbf{D}(P)) \tag{7}$$

where $\mathbf{Q}$ is the bi-laplacian of triangulated mesh and $D(P)$ is the displacement vector defined as $\mathbf{D}(P) = \mathcal{F}(P) - \mathcal{F}(P_0)$.

The ARAP energy can be similarly expressed in terms of the program parameters:

$$E_{ARAP}(P) = \sum_{i=1}^{n} w_i \sum_{j \in \mathcal{N}(i)} w_{ij} \left\| (V_i(P) - V_j(P)) - R_i(V_i(P_0) - V_j(P_0)) \right\|^2 \tag{8}$$

where $\mathcal{N}(i)$ is the one-ring neighborhood of vertex v_i given the triangulated mesh, R_i is the approximate rigid transformation in $\mathcal{N}(i)$, and $w_{ij} = \frac{1}{2}(\cot\alpha_{ij} + \cot\beta_{ij})$, where α_{ij}, β_{ij} are the angles opposite of the mesh edge (v_i, v_j), and $w_i = 1$.

Since the bi-harmonic function can be analytically differentiated with respect to the deformation D, we can further speed up our optimization by evaluating this analytic derivative using a forward automatic differentiation pass for each parameter to create the Jacobian matrix $\frac{\partial \mathbf{D}}{\partial p_i}$. The implementation of ARAP is similar, although with the addition of the local optimization and iteration [SA07]. See Appendix A for more details.

3.3.3. Program-driven Objectives

Optimization towards geometric or deformation objectives produces meshes that geometrically resemble the initial mesh. However, just as with vertex positions and edge lengths, parameter values can also be a proxy for similarity, especially when parameters correspond to semantic attributes of the model, like length, width, or radii. We minimize change in parameter values according to an L_1 energy, effectively focusing changes on a minimal subset of program parameters. This is given by

$$E_{\text{par}}(P) = \sum_{P_i \in P} W_{\text{par}}(P_i) \left\| p_i - p_{0_i} \right\|_1 \tag{9}$$

$$W_{\text{par}}(P_i) = \left(1 - \frac{|V_{P_i} \cap V'|}{|V_{P_i}|}\right)^2 \tag{10}$$

where p_{0_i} is the initial value of parameter P_i at the start of the optimization, and V_{P_i} is the set of vertices whose coordinates in the computation graph G contain P_i as an ancestor. Our localization term $W_{\text{par}}(P_i)$ focuses change in the *parameter* space on parameters that correspond more closely with the vertices affected by a manipulation, even if that change might correspond to changes in many vertices or edge lengths. This objective encodes the hypothesis that a small subset of parameters in a CAD program can correspond to a higher-level design criterion, and focusing variation to just these parameters can result in variation along that design criterion, regardless of the underlying geometry. For example, this objective promotes uniformly widening the elements in a model according to a thickness parameter, even if this results in geometric changes that would be penalized by the purely geometric objectives discussed in Section 3.3.1. An example of this can be seen in Figure 6b, where the parameter energy results in thickening the entire dresser frame.

An outstanding issue with this approach is that one parameter might be much more sensitive than another. Moreover, there is no clear comparison between a parameter that controls rotation angle and one which controls a length – the objective will penalize each change equally, regardless of its effect on the output. In our experiments, we were able to achieve reasonable results with the energy as-is, mainly due to induced sparsity and localization. We propose a more in-depth treatment of parameter normalization as part of future work.

3.3.4. Performance Objectives

We also present objectives that aim to proxy some aspect of physical performance. For example, preserving volume can be used to maintain a fabrication objective such as material usage:

$$E_{vol}(P) = (Vol(P) - Vol(P_0))^2 \tag{11}$$

where $Vol(P)$ represents the volume of the output mesh at parameters P. Another example is preserving the center of mass, which may be essential for stability:

$$E_{\text{cm}}(P) = \|COM(P) - COM(P_0)\|_2 + K_v E_{\text{vtx}}(P) \tag{12}$$

where COM is the center of mass given model parameters, and K_v is a weighting term. Such performance objectives are often inadequate for the goal of resolving ambiguity – while they reduce the result space by maintaining a property, there may still be many valid parameter combinations that satisfy the edit. As a result we find it useful to combine these terms with the other geometric energies, particularly the localized vertex objective, using K_v as 0.01 in our implementation.

3.3.5. Composed Objective.

Each of the above objectives is then individually combined with our edit objective (Equation 1) to create

$$\mathcal{E}_{\text{obj}} = E_{\text{edit}} + \gamma_{\text{obj}} E_{\text{obj}} \tag{13}$$

where γ_{obj} represents a weighting term for how strongly the optimization should aim to preserve the property represented by E_{obj} as opposed to lowering the weight of the edit energy E_{edit}. These two objectives are often conflicting for geometric objectives, where no parameter combination P that minimizes E_{edit} also minimizes E_{obj}; causing our overall minimization of $\mathcal{E}_{\text{obj}}$ to deviate from satisfying the user's edit. Exposing control of γ_{obj} allows the user to tune how strongly to enforce their edit against an objective. Each $\mathcal{E}_{\text{obj}}$ is treated as a single objective for our optimization step, and the user is given a choice of which objectives to run for a given model. We note that it is also possible to combine multiple different E_{obj} together with E_{edit} to form new ones, as done in Equation 12.

4. Implementation

Underlying CAD System. Our system is implemented as an add-on to Blender [Com18], a popular 3D modeling package. We support all Blender primitives (`Cube`, `Sphere`, etc.) and a number of common CAD operations, which can be composed arbitrarily. A full list of supported operations is included in supplemental materials. Our language is embedded in Python, and uses a subset of Python syntax, using custom interpretation of the Python AST to implement CAD operations and semantics for model parameters. Our full implementation source and blender plugin can be found at https://github.com/paquinn/bimodel_eg22.

It may seem at first that our system could be implemented by naively applying an automatic differentiation tool, however there are several challenges that arise. Implementing on top of an existing CAD system implies that we interface with the underlying geometric kernel during interpretation in order to create and modify geometry. It is neither necessary nor desirable to differentiate kernel calls, nor to execute them during optimization. It is not possible to easily draw this distinction with off-the-shelf operator-overloading based automatic differentiation tools, which do not work across process boundaries and often rely on [PGM*19; FJL18] dynamic language features to evaluate the function with custom datatypes that cannot be passed off to a CAD package. Fully differentiating a CAD kernel as done in [BMA*18; Ban19] instead of providing a host language not only loses the safety guarantees offered by our DSL, but also incurs a 10-40x slowdown on model execution, which is untenable for interactive optimization.

As a result we explicitly construct our computation graph and compilation pass, instead of relying on an existing package to perform automatic differentiation. As mentioned in Section 3.2, our implementation still relies on the underlying CAD system to track mesh topology, and we index Blender vertices along with the vertex markers, updating those vertex coordinates directly with the results of evaluating $\mathcal{F}(P)$. As a result we completely avoid re-executing interpretation for changes that do not affect program structure.

Optimization and Performance. We minimize our objectives using the Sequential Least Squares Quadratic Programming (SLSQP) [Kra*88] optimizer through the SciPy library [VGO*20] with first order derivatives, primarily because of its ability to maintain constraints and make use of first-order information. Initial model interpretation is interactive, with models spanning hundreds of lines of code and over a thousand vertices usually interpreting in less than a second in our tests, running Python on a 2.5GHz Intel Core i7 Processor and 16 GB available RAM (Table 1). These models are comparable in size and complexity to complex CAD parts which are then combined into larger assemblies.

Model	Size: Ops / Verts	Interpret	Compile	Sync (10)	Sync (More)
Mount	221 / 20	0.02s	0.27s	0.08s	–
Dresser	1593 / 468	0.39s	2.06s	0.75s	1.58s
Chair	2534 / 220	0.28s	2.40s	0.69s	0.88s
Castle	11435 / 1481	1.22s	9.99s	1.89s	6.14s
Slipper	14835 / 269	0.45s	3.33s	0.73s	1.22s
Chandelier	19700 / 1554	1.35s	16.26s	1.02s	4.10s
Lamp	29033 / 520	0.55s	14.87s	4.66s	18.34s

Table 1: Performance metrics. *Operations represent arithmetic operations as tracked by our computation graph. Sync represents the total combined time of optimizing 6 objectives: E_{edit}, E_{vtx}, E_{edg}, E_{par}, E_{bh} , and E_{vol}. Note that synchronization time can vary greatly depending on the number and position of vertices selected – we present the mean of optimizing edits of 10 vertices over 10 random edits each; to give a sense of relative performance. Sync (More) contains edits that select more than 50% of the model's vertices, exceeding the amount found in typical edits.*

5. Results

Our results focus on how bidirectional editing can be used to augment program editing through inverse synchronization steps. We present two key results with respect to our optimization procedure:

- When a manipulation is *ambiguous*, minimizing the objectives we have included tends to provide a diverse set of results. This increases the likelihood that one is close to the user's intent, and additionally serves to indicate which areas of the model must be further manipulated to remove ambiguity.
- When a manipulation is *unambiguous*, our objectives tend to converge on extremely similar results, approximating the closest point in the parameter space to that manipulation.

Together, these properties allow users to iterate quickly through a series of geometric edits, ultimately allowing complete control over model parameters through geometry, while maintaining the editing benefits of a program representation.

5.1. Option Gallery

Figure 6 presents a model, an ambiguous manipulation, and the resulting diverse set of options returned by optimization of different objectives. Each objective can be described in terms of the geometric properties it preserves relative to the others.

The **edit energy** tends to move all relevant parameters in the direction of the edit with no other optimization terms. Many parameters tend to change slightly, which can be frustrating when the user is trying to affect only a specific area of the model. We see this most clearly in the castle model, where the edit objective affects the castle (Figure 6d) far away from the edit and shifts it more than any other option.

Minimizing **parameter change** can result in more drastic geometric changes (vertices moved, edges extended) than other options, though this is highly dependent on the program. Such changes can be seen clearly in the dresser and castle models; they are especially prevalent when the parameters are sensitive and small changes in value cause large geometric changes. This energy is most useful when a model directly exposes a subset of parameters that control the aspect the user wants to vary, or when the user intended a large manipulation that other objectives would penalize.

Localized vertex-position objectives can often end up deforming relative proportions of model elements (Lamp, Slipper) because they allow vertices close to the edit to move more than those further away.

As a result we see sharp variation close to the edit, while the rest of the model stays more or less fixed. This can also be seen in the dresser example, where most of the dressers' drawers retain their original height, except the top row of drawers, which stretch vertically in order to satisfy the objective. Moreover, in this example the left half of the model becomes much thinner horizontally than the right half – partially because there are simply more vertices on the right half of the model, so moving them is penalized more.

Localized edge-length preservation in the dresser example shows a similar behavior. By minimizing the L_1 norm of edge length differences we are are encouraging sparsity, so the objective naturally prefers to stretch out the top-most vertical edges, as

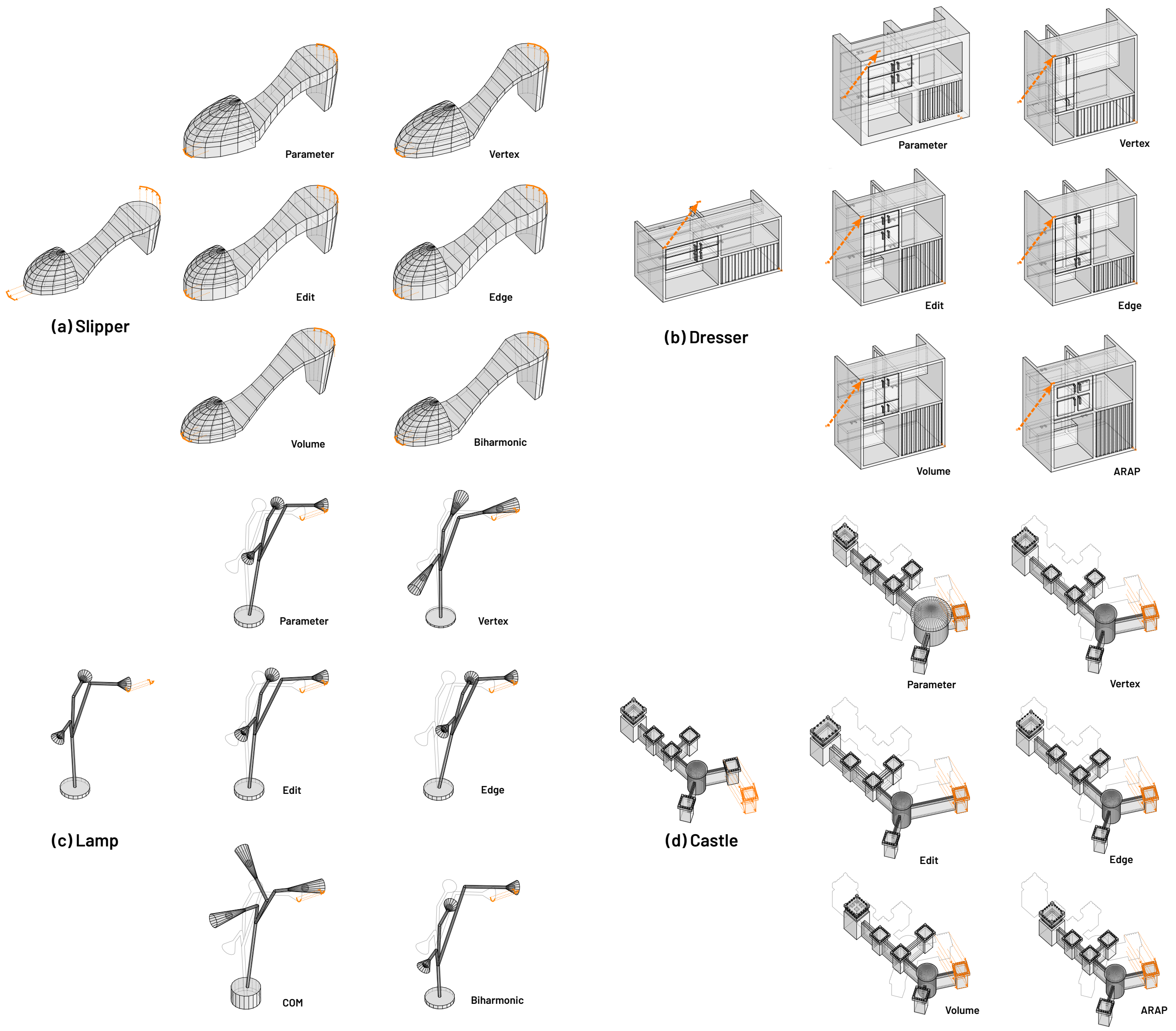

Figure 6: Option Gallery. *Each model is edited (orange), displaying the results our system produces by optimizing each objective. The edit specification is shown overlaid on each result to illustrate how closely the result matches the manipulation.*

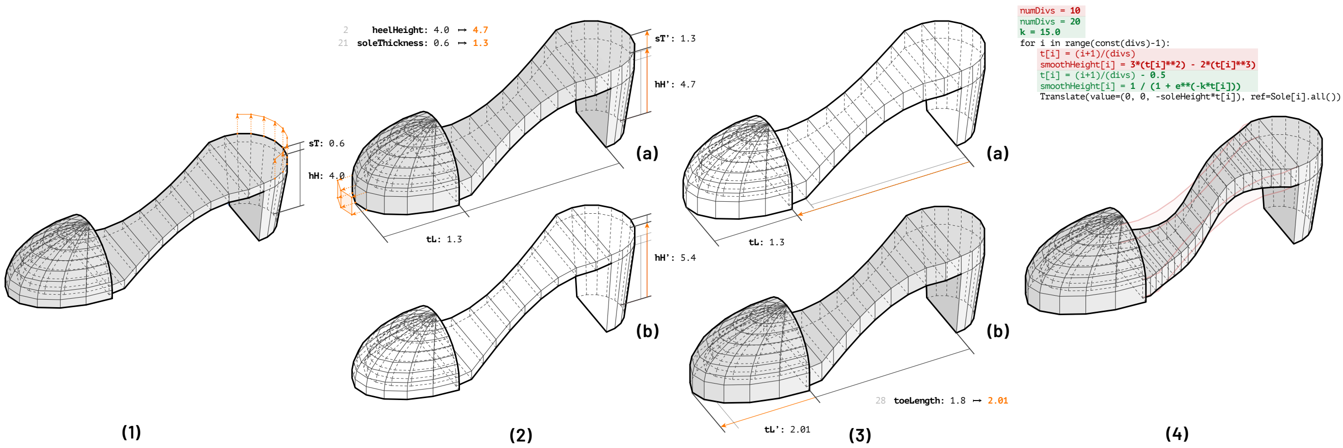

Figure 7: Slipper workflow. *A user iterates on a design for a freeform slipper. Vertices on the heel of the shoe are dragged up (1) and an option is selected, and subsequently vertices are dragged outwards on the toe (2a). After an option is selected for the toe edit (3b), a program manipulation tweaking the sole curvature and mesh discretization is performed (4).*

localization penalizes this change the least. In the slipper model, the edge objective prefers moving the toe forward to stretching it, because this results in overall less change in edge lengths, even when localizing the effects.

Deformation objectives do well at maintaining the overall geometric shape of the model, as they minimize the rate of local changes to the mesh imposed by satisfying the edit. This propagates changes smoothly from the edited region into the remainder of the model, resulting in a looser, but smoother, localization than the vertex and edge objectives (Slipper, Lamp).

Objectives preserving **global properties**, such as Volume and Center of Mass, result in more non-local changes than the other objectives as they update parts of the model outside of the user's specified edit to maintain the overall property. This can be seen in the positions of un-manipulated arms in the lamp model, and the width of the bottom of the heel in the slipper model. Penalizing vertex movements prevents the changes from being extremely drastic.

Our bidirectional editing approach also addresses two common pain points in CAD modeling. Firstly, models encode constraints such as proportions, patterns, and bounds on the positions of elements. When editing CAD programs directly, users must reason about which parameters need to change together, and how to maintain explicitly-defined constraints. Our inverse editing step operates agnostically to how these constraints and patterns happen to be imposed in a particular program, finding a solution that preserves the encoded constraints regardless of how many parameters need to be adjusted. Patterns, such as the vertical stripes at the bottom right of the dresser (6b) or the crenelations on the castle towers (6d), maintain their structure through arbitrary parameter changes. Our method allows both painless editing of patterns and preservation of desirable invariants, both common challenges in CAD modeling.

Secondly, designed objects can have semantic properties that cannot be encoded purely in their geometry — for example, a lamp must be stable and upright. It is also difficult to construct a program that limits the space of possible designs to exactly those that will be stable. Instead, optimizations such as Center of Mass (Figure 6c) can proxy the desired property and stay in the range of stable solutions without the user ever having to specify exactly how such limits are implemented.

5.2. Workflows

Figure 7 illustrates how a user might edit a free-form slipper design to better match a specific target shape they have in mind. In the first manipulation (1), the user wants to raise the heel and thicken the sole. To indicate this, she raises a few vertices on the heel of the shoe. But, the edit is ambiguous – geometrically, the edit could be achieved by either raising the heel, thickening the sole, or both; our objectives converge to two options that either increase a combination of both heel height and sole thickness (2a); or raise the heel alone (2b). Since the user desires to increase both of these aspects, they choose the first option.

The user then wants to extend the toe, and indicates this by dragging a few vertices out. Our system converges to two options that lengthen the sole (3a) or lengthen the toe (3b) respectively. Since the model happens to expose `toeLength` explicitly as a parameter, the user chooses the option minimizing parameter energy (3b) which matches their edit and focuses change on the `toeLength` parameter, accomplishing their goal.

In the final step, our user wants to impose steeper curvature of the sole from toe to heel. To accomplish this she makes a program edit, changing the function controlling the curvature from a polynomial basis to a logistic one, and adjusts the number of divisions sampled to make the discretization finer.

A similar process occurs in Figure 8. We note that for this example, previous work on editing man-made models fail to expose

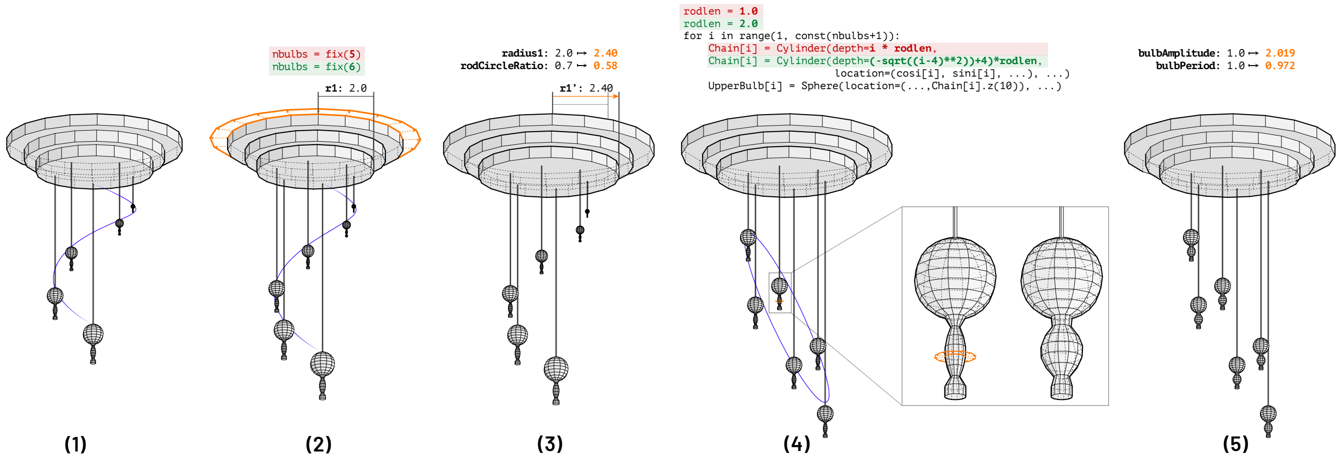

Figure 8: Chandelier workflow. *A user explores design variations for a chandelier model. She performs a program manipulation on the original, 5-bulb model (1) to achieve a variant with 6 bulbs (2), and scale out a ring of vertices around the top of the model to uniformly widen the base (3) via the biharmonic energy. Another program change is performed to rearrange the bulb pattern (4), and a final geometric edit widens a single bulb with the ARAP energy, resulting in all of the bulbs updating to match (5).*

relationships between the radii of the three bases of the chandelier. In particular, iWires [GSMC09] does not maintain a program representation and does not find any wires for the bulbs, which makes it impossible to achieve the last edit in the workflow, while [BWSK12] does not support rotational patterns. Neither method can generate a chandelier with more or less bulbs while preserving the arrangement.

5.3. Initial User Feedback

We gathered preliminary qualitative feedback on our interface and usability from domain experts. Four users with extensive background in CAD modeling across different domains (Engineering, Architecture, and Visual Arts) were introduced to our tool for 15 minutes. Users were then asked to use our tool's inverse editing features alone to a) match a 3D model to a target geometry overlaid in the interface; and b) manipulate a model to achieve a design variation, over three different models in 1.5 hours. We used a think-aloud protocol and asked participants to say what they were thinking as they worked with our tool. User testing highlighted the importance of promoting *diversity* in our results, as this allows users to understand how to adjust their manipulations even if the tool does not provide any correct result.

We observed that users initially perform several geometric edits when encountering a new model for the first time, intuiting a sense of how a model varies. When precise dimensions are not of concern, users quickly generate plausible design variations using geometric edits that change tens of parameters, and rarely consider more than two of the returned options. Informally, users preferred the Vertex and Parameter energies, but there were also edits where the Edge length and ARAP energies returned better results.

Users were critical about difficulty in predicting how a model will change in response to an edit, and required a few test manipulations on any given model before it could be productively edited. Additionally, users occasionally encountered situations where the model program was constructed in a way that made an attempted edit infeasible. Users in this situation were usually uncertain what had occurred, and would benefit significantly from some form of automated explanation.

When browsing options generated in response to their edits, users usually only go through more than two alternatives if neither came close to satisfying their intent. Critically, if no alternatives correspond to user intent, users are almost instantly able to identify necessary changes to make to their edit to disambiguate by comparing the differences between results. Accordingly, users typically perform several iterations of geometric edits to achieve the target manipulations, changing tens of parameters in the process of doing so. All users found inverse editing with our tool significantly easier to use for creating design variations than the analogous CAD tools or program edits.

In conclusion, our method exposes a trade-off between giving users more options, and the time it takes to evaluate each option, versus the user identifying the ambiguity adjusting the manipulation to clarify intent. Interactive optimization performance is critical to bidirectional editing being useful, as users must be able to iterate quickly and adjust their manipulations.

6. Limitations and Future Work

The primary limitation of our approach is that while topological changes can be achieved by editing the program, our synchronization step by definition always preserves mesh topology; as it requires the function generating the mesh to be continuous, as well differentiable on most of its domain for the technique to work well. Michel et al [MB21] illustrate another path forward where the gradient is not evaluated throughout the optimization and topological changes can be supported as long as UV coordinates are maintained. We see potential to merge such methods and accommodate a greater class of operators with higher performance, and note that the question of optimizing edits across topological change is unsolved in general.

Particularly, integer parameters such as loop bounds—which don't contribute to the gradient of a vertex and whose variation would create topologically different meshes—are always fixed in our system (Section 3.1). Future work on optimizing through discrete changes would enable us to expand our solution to include arbitrary topological changes. Existing methods are beginning to allow differentiation through such discontinuities [BMM*21] via careful language design. One possible approach in this vein is to extend our current language to define references, user edits, and objective functions on collections of high-level elements of variable arity rather than explicitly over a fixed number of vertices.

We address ambiguity in a user manipulation by using heuristics in the form of different objective functions and energies to allow the user to manually disambiguate their intent if needed. This approach is not automatic, and when a given user intent is not within the program's output space, many objectives tend to converge to similar results while still differing from a user's intent. To address this, future work can use existing approaches to refining discrete parameter spaces, including design galleries [MAB*97; SFP*20] and user-in-the loop optimization [KSG20], in combination with program synthesis as a possible avenue for exploration. We suggest an approach where users can select an area of interest and a system can synthesize possible parameterizations that contain the desired output, which can then be explored using bidirectional editing.

Finally, an exciting avenue of future work revolves around the use of additional program information in geometry processing techniques. Geometry processing has traditionally been focused around polygonal meshes, and our method enables the augmentation of this mesh with a wealth of additional information garnered from analyzing the generating program – we propose a sensitivity analysis of vertices with respect to parameters, methods for geometric constraint inference and maintenance, and program-aided design optimization integrated with the model editing process, to name a few. This would not only allow geometry processing techniques to be applied directly to CAD models, skipping often expensive translation steps, but also enable more powerful and versatile editing, analysis and optimization.

7. Conclusion

3D CAD software is used by almost every designer in the world to create most of the virtual and man-made physical models we encounter. Moreover, direct editing of CAD programs is a challenge designers have been facing for multiple decades. This work argues that a combination of geometry, optimization, and programming languages methods can be intersected to devise new solutions that significantly reduce the existing burdens around CAD editing, allowing easier design iteration and customization. As an initial step, we described an implementation of *bidirectional* editing, where users leverage both programmatic expressivity as well as direct geometric manipulation. This approach realizes a significant advancement over the current state of the art of editing in CAD tools, ultimately allowing designers and engineers to work more quickly on a wide range of 3D design tasks.

References

[Alb18] Alba, Michael. *What's the Difference Between Parametric and Direct Modeling?* https://www.engineering.com/DesignSoftware/DesignSoftwareArticles/ArticleID/16587/Whats-the-Difference-Between-Parametric-and-Direct-Modeling.aspx. (Accessed on 09/06/2019). Mar. 2018 2.

[Ban19] Banovic, Mladen. "Efficient algorithmic differentiation of CAD frameworks". *Dissertation, Paderborn University* (2019) 2, 9.

[BFH05] Berndt, Rene, Fellner, Dieter W., and Havemann, Sven. "Generative 3D Models: A Key to More Information Within Less Bandwidth at Higher Quality". *Proceedings of the Tenth International Conference on 3D Web Technology*. Web3D '05. ACM, 2005, 111–121 3.

[BMA*18] Banovic, Mladen, Mykhaskiv, Orest, Auriemma, Salvatore, et al. "Algorithmic differentiation of the Open CASCADE Technology CAD kernel and its coupling with an adjoint CFD solver". *Optimization Methods and Software* 33.4-6 (2018), 813–828. doi: 10.1080/10556788.2018.1431235. eprint: https://doi.org/10.1080/10556788.2018.1431235. url: https://doi.org/10.1080/10556788.2018.1431235 9.

[BMM*21] Bangaru, Sai, Michel, Jesse, Mu, Kevin, et al. "Systematically Differentiating Parametric Discontinuities". *ACM Trans. Graph.* 40.107 (2021), 107:1–107:17 13.

[Bor81] Borning, Alan. "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory". *ACM Trans. Program. Lang. Syst.* 3.4 (Oct. 1981), 353–387. issn: 0164-0925. doi: 10.1145/357146.357147. url: https://doi.org/10.1145/357146.357147 5.

[Bru14] Brunelli, Mark. *Parametric vs. Direct Modeling | PTC*. https://www.ptc.com/en/cad-software-blog/parametric-vs-direct-modeling-which-side-are-you-on. (Accessed on 09/08/2019). Apr. 2014 2.

[BWKS11] Bokeloh, Martin, Wand, Michael, Koltun, Vladlen, and Seidel, Hans-Peter. "Pattern-aware shape deformation using sliding dockers". *ACM Transactions on Graphics (TOG)*. Vol. 30. 6. ACM. 2011, 123 4.

[BWSK12] Bokeloh, Martin, Wand, Michael, Seidel, Hans-Peter, and Koltun, Vladlen. "An algebraic model for parameterized shape editing". *ACM Transactions on Graphics (TOG)* 31.4 (2012), 78 4, 12.

[CHSA16] Chugh, Ravi, Hempel, Brian, Spradlin, Mitchell, and Albers, Jacob. "Programmatic and Direct Manipulation, Together at Last". *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. ACM. Santa Barbara, CA, USA, 2016, 341–354 3.

[Com18] Community, Blender Online. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. url: http://www.blender.org 8.

[DIP*18] Du, Tao, Inala, Jeevana Priya, Pu, Yewen, et al. "Inversecsg: Automatic conversion of 3d models to csg trees". *ACM Transactions on Graphics (TOG)* 37.6 (2018), 1–16 3.

[FJL18] Frostig, Roy, Johnson, Matthew James, and Leary, Chris. "Compiling machine learning programs via high-level tracing". *Systems for Machine Learning* (2018) 4, 9.

[GJB*20] Guo, Jianwei, Jiang, Haiyong, Benes, Bedrich, et al. "Inverse Procedural Modeling of Branching Structures by Inferring L-Systems". *ACM Transactions on Graphics (TOG)* 39.5 (2020), 1–13 3.

[GKG*22] GAILLARD, MATHIEU, KRS, VOJTECH, GORI, GIORGIO, et al. "Automatic Differentiable Procedural Modeling". Vol. 41. 2. Wiley Online Library. 2022 3.

[GSMC09] GAL, RAN, SORKINE, OLGA, MITRA, NILOY J., and COHEN-OR, DANIEL. "IWIRES: an analyze-and-edit approach to shape manipulation". *ACM Transactions on Graphics* 28.3 (2009) 4, 12.

[GW08] GRIEWANK, A. and WALTHER, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2008. ISBN: 9780898717761. URL: https://books.google.com/books?id=xoiiLaRxcbEC 4.

[HBR14] HOTTELIER, THIBAUD, BODI'K, RAS, and RYOKAI, KIMIKO. "Programming by Manipulation for Layout". *The 27th Annual ACM Symposium on User Interface Software and Technology UIST '14*. 2014, 231–241 3.

[HLC19] HEMPEL, BRIAN, LUBIN, JUSTIN, and CHUGH, RAVI. "Sketch-n-Sketch: Output-Directed Programming for SVG". *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 2019, 281–292 3.

[HSK*19] HAFNER, CHRISTIAN, SCHUMACHER, CHRISTIAN, KNOOP, ESPEN, et al. "X-CAD: optimizing CAD models with extended finite elements". *ACM Transactions on Graphics (TOG)* 38.6 (2019), 1–15 4.

[HSST11] HU, ZHENJIANG, SCHURR, ANDY, STEVENS, PERDITA, and TERWILLIGER, JAMES F. "Dagstuhl Seminar on Bidirectional Transformations (BX)". *Proceedings of the 2011 International Conference on Management of Data*. Vol. 30. SIGMOD '11 1. ACM. Athens, Greece, 2011, 35–39 3.

[JBX*20] JONES, R. KENNY, BARTON, THERESA, XU, XIANGHAO, et al. "ShapeAssembly: Learning to Generate Programs for 3D Shape Structure Synthesis". *ACM Transactions on Graphics (TOG), Siggraph Asia 2020* 39.6 (2020), Article 234 3.

[JCG*21] JONES, R. KENNY, CHARATAN, DAVID, GUERRERO, PAUL, et al. "ShapeMOD: Macro Operation Discovery for 3D Shape Programs". *ACM Transactions on Graphics (TOG), Siggraph 2021* 40.4 (2021), Article 153 3.

[JPCS18] JESUS, DIEGO, PATOW, GUSTAVO, COELHO, ANTÓNIO, and SOUSA, ANTONIO AUGUSTO. "Generalized selections for direct control in procedural buildings". *Computers & Graphics* 72 (2018), 106–121 3.

[Kra*88] KRAFT, DIETER et al. "A software package for sequential quadratic programming". (1988) 9.

[KSG20] KOYAMA, YUKI, SATO, ISSEI, and GOTO, MASATAKA. "Sequential gallery for interactive visual design optimization". *ACM Transactions on Graphics (TOG)* 39.4 (2020), 88–1 13.

[KSSC08] KRAEVOY, VLADISLAV, SHEFFER, ALLA, SHAMIR, ARIEL, and COHEN-OR, DANIEL. "Non-homogeneous resizing of complex models". *ACM Transactions on Graphics (TOG)*. Vol. 27. 5. ACM. 2008, 111 4.

[LLGR20] LI, TZU-MAO, LUKÁČ, MICHAL, GHARBI, MICHAËL, and RAGAN-KELLEY, JONATHAN. "Differentiable vector graphics rasterization for editing and learning". *ACM Transactions on Graphics (TOG)* 39.6 (2020), 1–15 4.

[LSL*19] LIPP, M., SPECHT, M., LAU, C., et al. "Local Editing of Procedural Models". *Computer Graphics Forum* 38 (May 2019), 13–25. DOI: 10.1111/cgf.13615 3.

[LWW08] LIPP, MARKUS, WONKA, PETER, and WIMMER, MICHAEL. "Interactive Visual Editing of Grammars for Procedural Architecture". *ACM Transactions on Graphics* 27.3 (Aug. 2008), 35:1–35:9 3.

[MA] MCNEEL, ROBERT and ASSOCIATES. *Grasshopper3D*. URL: https://www.grasshopper3d.com/ 3.

[MAB*97] MARKS, JOE, ANDALMAN, BRAD, BEARDSLEY, PAUL A, et al. "Design galleries: A general approach to setting parameters for computer graphics and animation". *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, 389–400 13.

[Mar19] MARGOSSIAN, CHARLES C. "A review of automatic differentiation and its efficient implementation". *WIREs Data Mining and Knowledge Discovery* 9.4 (2019), e1305. DOI: https://doi.org/10.1002/widm.1305 6.

[MB21] MICHEL, ELIE and BOUBEKEUR, TAMY. "DAG Amendment for Inverse Control of Parametric Shapes". *ACM Transactions on Graphics* 40.4 (2021), 173:1–173:14 2, 3, 12.

[MWH*06] MÜLLER, PASCAL, WONKA, PETER, HAEGLER, SIMON, et al. "Procedural Modeling of Buildings". *ACM Transactions on Graphics* 25.3 (July 2006) 3.

[MWZ*14] MITRA, NILOY J, WAND, MICHAEL, ZHANG, HAO, et al. "Structure-aware shape processing". *ACM SIGGRAPH 2014 Courses*. ACM. 2014, 13 4.

[NVZJ19] NIMIER-DAVID, MERLIN, VICINI, DELIO, ZELTNER, TIZIAN, and JAKOB, WENZEL. "Mitsuba 2: A retargetable forward and inverse renderer". *ACM Transactions on Graphics (TOG)* 38.6 (2019), 1–17 4.

[NWA*20] NANDI, CHANDRAKANA, WILLSEY, MAX, ANDERSON, ADAM, et al. "Synthesizing structured CAD models with equality saturation and inverse transformations". *vertex* 9.15 (2020) 3.

[NWP*18] NANDI, CHANDRAKANA, WILCOX, JAMES R, PANCHEKHA, PAVEL, et al. "Functional programming for compiling and decompiling computer-aided design". *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), 1–31 3.

[PGM*19] PASZKE, ADAM, GROSS, SAM, MASSA, FRANCISCO, et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". *Advances in Neural Information Processing Systems 32*. Ed. by WALLACH, H., LAROCHELLE, H., BEYGELZIMER, A., et al. Curran Associates, Inc., 2019, 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf 4, 9.

[RGF*20] REDDY, PRADYUMNA, GUERRERO, PAUL, FISHER, MATT, et al. "Discovering pattern structure using differentiable compositing". *ACM Transactions on Graphics (TOG)* 39.6 (2020), 1–15 4.

[SA07] SORKINE, OLGA and ALEXA, MARC. "As-rigid-as-possible surface modeling". *Symposium on Geometry processing*. Vol. 4. 2007, 109–116 7, 8, 15.

[SCL*04] SORKINE, OLGA, COHEN-OR, DANIEL, LIPMAN, YARON, et al. "Laplacian surface editing". *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. 2004, 175–184 7.

[SFP*20] SHIMIZU, EVAN, FISHER, MATTHEW, PARIS, SYLVAIN, et al. "Design Adjectives: A Framework for Interactive Model-Guided Exploration of Parameterized Design Spaces". *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 2020, 261–278 13.

[Sid] SIDEFX. *Houdini*. URL: https://www.sidefx.com/ 3.

[Sie17] SIEMENS. *Synchronous Technology - Going beyond traditional modeling approaches to solve design challenges*. 2017. URL: https://www.plm.automation.siemens.com/media/global/en/Siemens-PLM-Synchronous-technology-going-beyond-mi-63435_tcm27-28553.pdf 2.

[SLH*20] SHI, LIANG, LI, BEICHEN, HAŠAN, MILOŠ, et al. "MATch: Differentiable Material Graphs for Procedural Material Capture". *ACM Trans. Graph.* 39.6 (Dec. 2020), 1–15 4.

[SSL*14] SCHULZ, ADRIANA, SHAMIR, ARIEL, LEVIN, DAVID IW, et al. "Design and fabrication by example". *ACM Transactions on Graphics (TOG)* 33.4 (2014), 62 4.

[STBB14] SMELIK, RUBEN M, TUTENEL, TIM, BIDARRA, RAFAEL, and BENES, BEDRICH. "A survey on procedural modelling for virtual worlds". *Computer Graphics Forum*. Vol. 33. 6. Wiley Online Library. 2014, 31–50 3.

[Str16] STRATER, JEFF. *Solved: Re: Direct vs History based modelling - Page 2 - Autodesk Community- Fusion 360*. `https://forums.autodesk.com/t5/fusion-360-design-validate/direct-vs-history-based-modelling/m-p/6717750#M84483`. (Accessed on 09/06/2019). Nov. 2016 2.

[SXZ*17] SCHULZ, ADRIANA, XU, JIE, ZHU, BO, et al. "Interactive Design Space Exploration and Optimization for CAD Models". *ACM Transactions on Graphics* 36.4 (July 2017) 2.

[Ten20] TENSORFLOW. *XLA Architecture*. `https://www.tensorflow.org/xla/architecture`. (Accessed on 01/26/2021). Nov. 2020 4.

[TLL*11] TALTON, JERRY O, LOU, YU, LESSER, STEVE, et al. "Metropolis procedural modeling". *ACM Transactions on Graphics (TOG)* 30.2 (2011), 11 3.

[VGA*12] VANEGAS, CARLOS A, GARCIA-DORADO, IGNACIO, ALIAGA, DANIEL G, et al. "Inverse design of urban procedural models". *ACM Transactions on Graphics (TOG)* 31.6 (2012), 168 3.

[VGO*20] VIRTANEN, PAULI, GOMMERS, RALF, OLIPHANT, TRAVIS E., et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". *Nature Methods* 17 (2020), 261–272. DOI: `10.1038/s41592-019-0686-2` 9.

[WPL*20] WILLIS, KARL DD, PU, YEWEN, LUO, JIELIANG, et al. "Fusion 360 Gallery: A Dataset and Environment for Programmatic CAD Reconstruction". *arXiv preprint arXiv:2010.02392* (2020) 3.

[WWSR03] WONKA, PETER, WIMMER, MICHAEL, SILLION, FRANÇOIS, and RIBARSKY, WILLIAM. "Instant Architecture". *ACM Transactions on Graphics* 22.3 (July 2003), 669–677 3.

[Yar13] YARES, EVAN. *The failed promise of parametric CAD part 1: From the beginning*. `https://www.3dcadworld.com/the-failed-promise-of-parametric-cad/`. (Accessed on 09/06/2019). June 2013 2.

[ZFC*11] ZHENG, YOUYI, FU, HONGBO, COHEN-OR, DANIEL, et al. "Component-wise Controllers for Structure-Preserving Shape Manipulation". *Computer Graphics Forum* 30.2 (2011), 563–572 4.

Appendix A: Derivations

We describe how we compute the gradients $\frac{\partial E_{bh}(P)}{\partial p_i}$ and $\frac{\partial E_{ARAP}(P)}{\partial p_i}$, for a parameter $p_i \in P$. Let $E(P) = E_{bh}(P) = tr(D(P)^T QD(P)$. Then,

$$\frac{\partial E(P)}{\partial p_i} = \left(QD(P) + Q^T D(P)\right) \frac{\partial\left(D(P)^T QD(P)\right)}{\partial p_i} =$$
$$\left(QD(P) + Q^T D(P)\right)(Q + Q^T)D(P)\frac{\partial D(P)}{\partial p_i}$$

Recall that $D(P) = \mathcal{F}(P) - \mathcal{F}(P_0)$, and we compute $\frac{\partial D(P)}{\partial p_i} = \frac{\partial \mathcal{F}(P)}{\partial p_i}$ using forward-mode automatic differentiation.

Similarly, let $E(P) = E_{ARAP}(P)$. We use the same alternating minimization strategy as in [SA07]: for a given fixed set of rigid transformations $\{R_i\}$, we find parameters P' that minimize $E(P')$. Then, we find the rigid transformations $\{R_i\}$ that minimize $E(P)$ for the given set of parameters P. $\{R_i\}$ is computed using SVD (Equation (6) [SA07]) of the covariance matrix S_i (Equation (5) [SA07]). Then, P' is computed by solving Equation (9) [SA07], with the only difference that we compute $\frac{\partial E(P')}{\partial p'_i} = \frac{\partial E(P')}{\partial V_i(P')}\frac{\partial V_i(P')}{\partial p'_i}$.