Direct Manipulation of Procedural Implicit Surfaces

MARZIA RISO, Sapienza University of Rome, Italy and Adobe, France
ÉLIE MICHEL, Adobe, France
AXEL PARIS, Adobe, France
VALENTIN DESCHAINTRE, Adobe, United Kingdom
MATHIEU GAILLARD, Adobe, USA
FABIO PELLACINI, University of Modena and Reggio Emilia, Italy

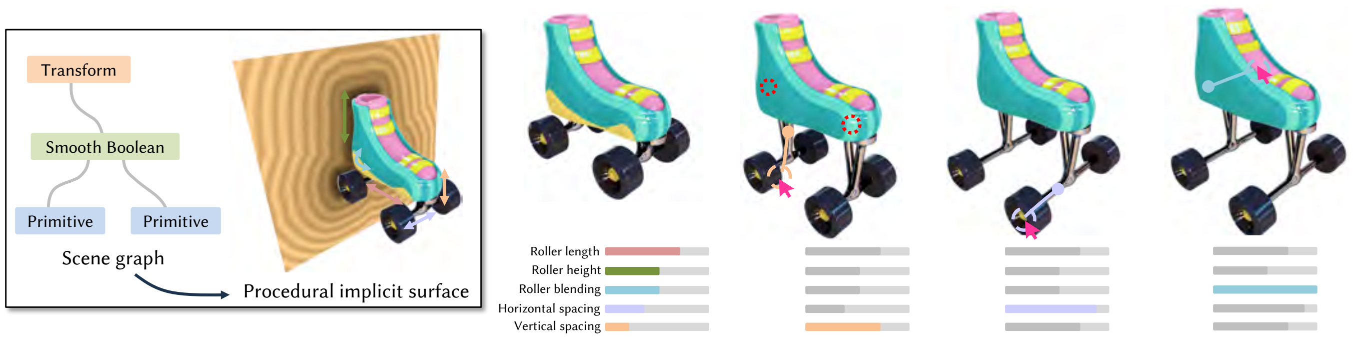


Fig. 1. Our method enables the direct manipulation of procedural implicit surfaces through mouse strokes in the viewport. We estimate an update of the procedural parameters of the implicit surface that matches the user intent thanks to the auto-differentiation of an augmented version of the implicit function (Section 4.2). As opposed to the typical workflow of updating parameters through sliders, our method enables a more direct and intuitive editing process.

Procedural implicit surfaces are a popular representation for shape modeling. They provide a simple framework for complex geometric operations such as Booleans, blending and deformations. However, their editability remains a challenging task: as the definition of the shape is purely implicit, direct manipulation of the shape cannot be performed. Thus, parameters of the model are often exposed through abstract sliders, which have to be non-trivially created by the user and understood by others for each individual model to modify. Further, each of these sliders needs to be set one by one to achieve the desired appearance. To circumvent this laborious process while preserving editability, we propose to directly manipulate the implicit surface in the viewport. We let the user naturally interact with the output shape, leveraging points on a co-parameterization we design specifically for implicit surfaces, to guide the parameter updates and reach the desired appearance faster. We leverage our automatic differentiation of the procedural implicit surface to propagate interactions made by the user in the viewport to the shape parameters themselves. We further design a solver that uses such information to guide an intuitive and smooth user workflow. We demonstrate different editing processes across multiple implicit shapes and parameters that would be tedious by tuning sliders.

ACM Reference Format:
Marzia Riso, Élie Michel, Axel Paris, Valentin Deschaintre, Mathieu Gaillard, and Fabio Pellacini. 2024. Direct Manipulation of Procedural Implicit Surfaces. *ACM Trans. Graph.* 43, 6, Article 238 (December 2024), 12 pages. https://doi.org/10.1145/3687936

1 INTRODUCTION

When creating virtual worlds and prototypes, authoring 3D assets is crucial. In particular, procedural modeling has gained significant traction in the industry in the past decade, relying heavily on implicit surfaces [Mag 2022; Wom 2022; Jeremias and Quilez 2014] – defined as the zero level set of a function. This representation is particularly interesting as it allows for hierarchical combinations of various functions representing primitives (e.g. spheres or boxes) and operators (e.g. Boolean operations, affine transformations or deformations) in a tree or a graph [Reiner et al. 2011; Wyvill et al. 1999]. Each of these operators and primitives comes with its own set of procedural parameters, which can typically be adjusted through sliders for non-destructive authoring. However, editing the shape by adjusting individual sliders requires a comprehensive understanding of its parameterization, as multiple parts can be affected by a single procedural parameter. Conversely, editing one part of a shape may require modifications of several interdependent procedural parameters. To be able to circumvent this tedious process, we propose a direct manipulation approach to editing. This approach allows users to directly interact with the end surface in the viewport and propagating the changes to the relevant procedural parameters. While

Authors' addresses: Marzia Riso, Sapienza University of Rome, Italy and Adobe, France, riso@di.uniroma1.it; Élie Michel, Adobe, France, emichel@adobe.com; Axel Paris, Adobe, France, aparis@adobe.com; Valentin Deschaintre, Adobe, United Kingdom, deschain@adobe.com; Mathieu Gaillard, Adobe, USA, gaillard@adobe.com; Fabio Pellacini, University of Modena and Reggio Emilia, Italy, fabio.pellacini@unimore.it.

this kind of technique saw recent success for mesh-based parametric modeling [Cascaval et al. 2022; Gaillard et al. 2022; Michel and Boubekeur 2021], none of these approaches can be readily applied to implicit surfaces. Tracking of explicit points on the surface during manipulations cannot be achieved easily in implicit surfaces due to the lack of surface parameterization.

Our method allows the user to perform edits by simply selecting and dragging any desired parts on the implicit shape over the 3D viewport. It optionally enables expressing constraints on other patches to remain unchanged throughout the edit, thus increasing expressiveness. We automatically update the procedural parameters of the implicit surface to modify the shape to best match the user manipulation. However, to enable manipulation of a procedural shape we need to be able to characterize an element of surface in a way that is robust to changes in the procedural parameters. As this is not trivially defined for implicit surfaces, we extend Michel and Boubekeur [2021]'s framework of *co-parameterization*, enabling the definition of a point's location and its Jacobian with respect to the shape parameters. We adapt this framework to implicit surfaces and show how it can be used for direct manipulation purposes. We further refine the parameter update through a new solver that exploits the Jacobian computed in an automatic differentiation fashion. We compute the Jacobian for multiple groups of points, each of which represents a patch of the implicit surface. Our framework supports the direct manipulation of procedural parameters for classical implicit primitives combined with complex operators such as smooth Boolean, deformations and affine transformations (Figure 1).

Contributions. We enable in-viewport editing of procedural implicit surfaces thanks to the following contributions:

- A mapping between point positions and unique identifiers for procedural implicit surfaces, allowing the proper tracking of a point during an edit.
- A solver that, given user mouse-strokes and multi-point constraints, interactively updates the values of dozens of procedural parameters to best match the user intent.
- A mean to evaluate the local influence of parameters on individual points of a shape, which could be applied in other optimization pipelines than direct manipulation.

We evaluate our method in terms of editing capacity (e.g. can we reach a desired shape) through a user study and a comparison to existing direct manipulation techniques for analytic implicit surfaces.

2 RELATED WORK

Previous research in the field of direct shape manipulation has tackled either the editing of procedural shapes or the editing of implicit surfaces, as reported below. Our work focuses on the overlap between the two, aiming at directly controlling procedural shapes defined as implicit surfaces. As our contribution is established in the context of procedural shape modeling, we only briefly discuss *discrete* implicit representations (that are not parametric), and focus more specifically on procedural implicit surfaces.

Editing procedural shapes. Directly from screen-space user edits is an active research topic applied for instance to 2D graphics (vector patterns [Riso and Pellacini 2023]), simple joint structures (inverse kinematic [Anjyo and Lewis 2010; Aristidou et al. 2018; Boulic and Mas 1996]), structured 3D shapes encoded as meshes [Bokeloh et al. 2011, 2012], or more general procedural surfaces which are the focus of our work. In the case of procedural 3D meshes, a few papers close to our work proposed techniques that allow in-viewport editing. The pioneer work from Gleicher [1994] investigated graphical interfaces for the direct manipulation of 3D shapes. More recently, Michel and Boubekeur [2021] presented a method to directly modify parametric meshes defined by acyclic graphs (DAGs). Gaillard et al. [2022] introduced an auto-differentiable hierarchical representation of the 3D scene to allow interactive control of procedural models defined by node graphs. Finally, Cascaval et al. [2022] proposed a bidirectional editing interface allowing users to interact with CAD models both by applying changes to the underlying program representation, or by directly manipulating it. A key limitation of the two latter methods is that they respectively rely on bounding boxes and mesh representations, and thus do not support operations that result in topological changes. Yet, in the context of parametric implicit surfaces, the assumption that no change of topologies can occur is impracticable as Constructive Solid Geometry (CSG) operations are widely used to construct complex models. As all these methods rely on the parametrization provided by meshes, they cannot be directly applied to implicits because they have no way to track the evolution of a point on the shape after a change in the procedural parameters. To solve this and enable in-viewport editing of procedural implicit surfaces, we define a bijective mapping between procedural parameters and points on the implicit surface.

Editing implicit surfaces. On top of providing control over affine transformations through 3D Gizmos in modeling software [Mag 2022; Wom 2022], previous research on implicit modeling has focused on providing indirect and direct control to the user. The work from Schmidt et al. [2006] shows an interactive modeling application where the user sketches 2D contours that are interpreted as new primitives in a BlobTree model [Wyvill et al. 1999], which can then be combined with CSG operators to create complex shapes.

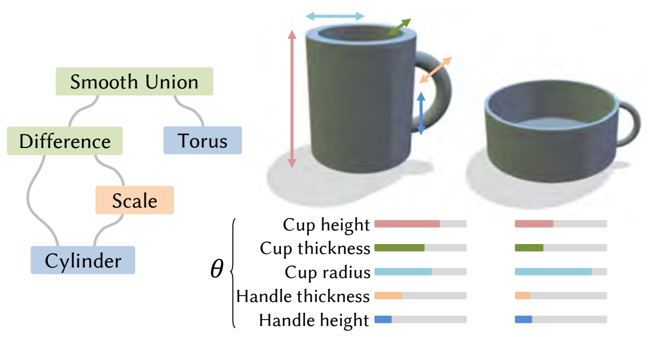


Fig. 2. Our input is a procedural implicit shape represented by a scene graph (left), combining primitives, transformations, Boolean operations, etc. Procedurally-defined shapes allow users to create a large variety of instances $\Phi(\boldsymbol{\theta})$ by tweaking the procedural parameters $\boldsymbol{\theta}$ (right), but this is a non-trivial task as the user must understand the influence of each individual parameter over the model. In each figure of the paper, bidirectional arrows are used as a simplified semantical representation of procedural parameters.

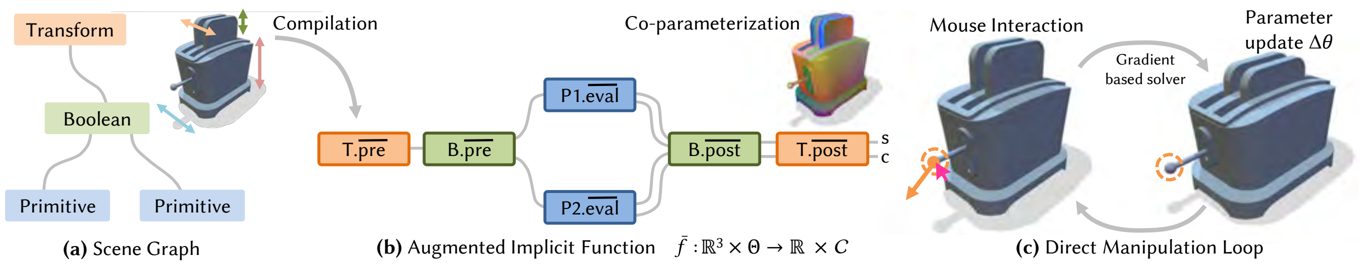


Fig. 3. Starting from a scene graph representation of an implicit surface **(a)**, we augment it so that the resulting implicit function $\bar{f}$ computes both the scalar value s and a *co-parameter* $\mathbf{c}$ that identifies the evaluation point in space **(b)**. We do this by replacing the `eval`, `pre`, `post` functions of the different nodes. This allows estimating the derivative of a position with respect to the procedural parameters, which is then used to modify them to match the user stroke **(c)**.

Despite the system's expressiveness, editing of the parameters is still done indirectly through manual tuning of sliders. Limited to affine transformations, the work of Barbier et al. [2005] shows how to animate a BlobTree model by defining the procedural parameters values as a function of time. Warp curves [Sugihara et al. 2008, 2010] are another alternative for editing implicit surfaces in which the user draws and manipulates polylines which locally deform the implicit surface using variational warping. Modifications are however limited to space-deformations, which are computationally intensive, and are not propagated back to the procedural parameters of the shape. Direct manipulation of implicit surfaces have been investigated for blending operators, with new ones that either improve topological control through new parameters [Zanni et al. 2015], or that match a 2D drawing of the intended blending behavior [Angles et al. 2017]. Research has also been conducted on the editing of *discrete* implicit surfaces encoded as level sets [Museth et al. 2002]. Users may perform direct manipulation operations such as surface smoothing and offsetting using predefined building blocks that modify the stored values of the implicit surface. Preservation of surface details prior to modifications may be done by using particle systems distributed on the surface [Eyiyurekli and Breen 2017], and displaced after modification to retain the details. However, objects in this case are not parametric, and thus the proposed solutions do not easily transpose to the editing of procedural implicit surfaces, which is the subject of our work. Closer to our work is the method exposed in libfive [Keeter 2019], a library for implicit modeling that provides some surface manipulation capabilities. The key difference is that it only allows the user to specify where there must be *some* element of surface, not to specify *which* element exactly. In practice, this limits the user to only inflation and translation operations. Our method is more general; it provides a direct manipulation framework for analytic implicit surfaces that handle more operators, such as affine transformation and complex warping such as twisting.

3 OVERVIEW

We aim at providing direct manipulation tools for procedural implicit surfaces where users interact with the shape itself directly in the viewport, rather than indirectly setting a value by moving an indicator on a track bar, namely a *slider*, as in traditional procedural modeling. Formally, a user edit consists in the selection of multiple points $\mathbf{p}_i$ over the surface, and their expected screen space movement ΔT_i. Typically, ΔT_i matches the movement of the user mouse cursor, actively expressing an edit to be matched. Occasionally, it could also be equal to zero, meaning that the element should not move during the edit, thus representing a constraint. The goal of our solver is to update the shape according to this edit while maintaining its global consistency. Our pipeline handles procedural implicit shapes described by scene graphs, as detailed in the next paragraph.

Background. An implicit surface is defined as the zero level set of a function $f : \mathbb{R}^3 \longrightarrow \mathbb{R}$. A point $\mathbf{p} \in \mathbb{R}^3$ belongs to the implicit surface S if and only if it satisfies $f(\mathbf{p}) = 0$. Representing 3D shapes using implicit surfaces thus inherits from interesting properties of function objects, like their compact analytical representation or the possibility to compose them together. A procedural implicit surface is a generalization of an implicit function f with a second argument from a procedural parameters space $\Theta \subset \mathbb{R}^n$, where n is the number of procedural parameters. These parameters, commonly used in general procedural modeling, are exposed by the designer of the initial shape to its end user. The surface S for a particular value $\boldsymbol{\theta} \in \Theta$ is called an *instance* of the procedural shape Φ:

$$S = \Phi(\boldsymbol{\theta}) = \{\mathbf{p} \in \mathbb{R}^3 \mid f(\mathbf{p}, \boldsymbol{\theta}) = 0\} \tag{1}$$

We support procedural implicit functions that are derived from a *scene graph*, like for instance BlobTrees [Wyvill et al. 1999] or analytical Signed Distance Fields. A scene graph is a directed acyclic graph (or sometimes more simply a tree) whose nodes are either *primitives* such as spheres or boxes, or *operators* such as CSG operators or affine transformations (see Figure 2). Complex implicit shapes arise from the combination of *primitives* via multiple boolean *operators* such as union, intersection of difference. In the implicit domain, a blended variation of regular boolean operations called *smooth boolean* is greatly exploited to create more organic shapes. A formal derivation of the implicit function f from the scene graph is described is Section 3.

Problem setting. In our context, a *manipulation* of the procedural implicit surface means a change of the procedural parameters $\boldsymbol{\theta}$ through user interaction. This ensures that the deformed surface globally remains a valid instance of the procedural shape Φ. To comply with the user input, we minimize for each manipulated point the following loss:

$$\mathcal{L}_i := \frac{1}{2}\left\|\Delta \operatorname{Proj}(\mathbf{p}_i) - \Delta T_i\right\|_2^2 \tag{2}$$

where $\Delta \operatorname{Proj}(\mathbf{p}) = \operatorname{Proj}(\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}) - \operatorname{Proj}(\mathbf{p}^{\boldsymbol{\theta}})$ is the effective movement of the point $\mathbf{p}$ after an update $\Delta\boldsymbol{\theta}$ of the procedural parameters, projected by Proj onto the screen. The first problem we address, in Section 4, is the definition of the new position $\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ of the dragged point. Indeed, while the initial position $\mathbf{p}^{\boldsymbol{\theta}}$ is simply found by casting a ray onto the surface, tracking what is semantically the *same element of geometry* after the change of procedural parameters is challenging. We then derive in Section 5 the gradient of $\mathcal{L}_i$, and in particular the Jacobian matrix of each dragged position $\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ with respect to $\Delta\boldsymbol{\theta}$. Lastly, Section 6 details our gradient descent based solver. Our complete pipeline is summarized in Figure 3.

Scene Graph Model. In a typical scene graph used for implicit modeling, an oriented edge is used in two ways: from the root to the leaves, it carries a position $\mathbf{p}$ at which primitives must be evaluated, then from the leaves back to the root, it carries the returned scalar value s. We formalize this by having each primitive provide an `eval` function, which maps a position $\mathbf{p} \in \mathbb{R}^3$ to a scalar value $s \in \mathbb{R}$, and each operator that has m input provide a function `pre` : $\mathbb{R}^3 \rightarrow (\mathbb{R}^3)^m$ that prepares the m positions fed to its inputs and a function `post` : $\mathbb{R}^m \rightarrow \mathbb{R}$ that reduces the m values returned by the inputs. For instance, the `eval` function of a sphere primitive of radius r is `eval` : $\mathbf{p} \mapsto \|\mathbf{p}\| - r$ and here are examples of operators:

Scaling by a factor x	**Union of 2 shapes**
`pre` : $\mathbf{p} \mapsto \mathbf{p}/x$	`pre` : $\mathbf{p} \mapsto (\mathbf{p}, \mathbf{p})$
`post` : $s \mapsto s \cdot x$	`post` : $(s_1, s_2) \mapsto \min(s_1, s_2)$

The final expression of f is obtained by recursively chaining the `pre`, `eval` and `post` expressions as detailed in Algorithm 1. The free variables of the expression – e.g. the scale factor x or the radius r in the examples above – constitute the vector $\boldsymbol{\theta}$ of procedural parameters. In practice there is usually a remapping between the parameters that are publicly exposed to the end user and the low-level parameters of the graph nodes, but we consider without loss of generality that this is part of the `eval`, `pre` and `post` functions.

We assume the resulting function f to be continuous and differentiable around its zero level-set. In order to render shapes using sphere tracing [Hart 1996], we also assume that f is Lipschitz, ie. that there is a bound λ on the magnitude of ∇f, ensuring that $|f(\mathbf{p}, \boldsymbol{\theta})|/\lambda$ is always lower than the distance from $\mathbf{p}$ to the surface. This in turn allows to compute points on the surface using sphere tracing for direct manipulation purposes.

ALGORITHM 1: Derivation of the implicit function from a scene graph. The expression of the implicit function f is derived from a scene graph by recursively compiling its root node into the expression of an evaluation function $\mathbb{R}^3 \mapsto \mathbb{R}$.

```
Input: A node $n$ of the scene graph.
Output: The implicit function represented by the node $n$.
function CompileNode($n$):
    if IsPrimitive($n$)
        return $n$.eval;
    else
        children ← GetChildren($n$);
        return $n$.post ∘ map(CompileNode, children) ∘ $n$.pre;
```

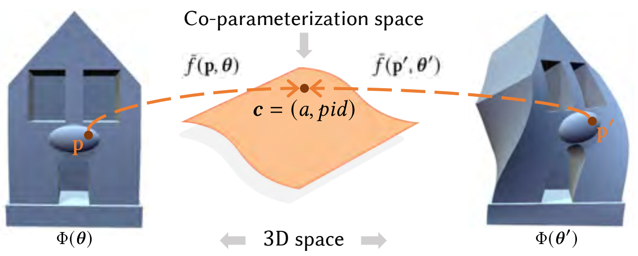

Co-parameterization space
$\bar{f}(\mathbf{p}, \boldsymbol{\theta})$
$\bar{f}(\mathbf{p}', \boldsymbol{\theta}')$
$\boldsymbol{c} = (a, pid)$
p
p′
$\Phi(\boldsymbol{\theta})$
3D space
$\Phi(\boldsymbol{\theta}')$

Fig. 4. Co-parameterization enables the identification of the same point location before and after an edit. The two points $\mathbf{p}$ and $\mathbf{p}'$, selected in different shape instances $\Phi(\boldsymbol{\theta})$ and $\Phi(\boldsymbol{\theta}')$, have two different locations in $\mathbb{R}^3$, but are mapped to the same co-parameter $\boldsymbol{c} \in C$ because they represent the same semantic element of the shape. This enables to define the influence $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ of the procedural parameters at a given point $\mathbf{p}$.

4 COPARAMETERIZATION

To enable direct manipulation, we need to robustly identify the same point location throughout the edit as it allows us to estimate the local influence of the procedural parameters. We formalize $\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ as the position of a point $\mathbf{p}$ after an update $\theta + \Delta\theta$ of the procedural parameters (Section 4.1), and describe how to use the structure of the scene graph to track the identity of manipulated points (Section 4.2).

4.1 Definition

The sole expression of the implicit function f cannot provide the position $\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ of a dragged point for an arbitrary change $\Delta\boldsymbol{\theta}$ of the procedural parameters, because in its compiled form it lacks the semantic awareness of the original scene graph. We propose to define an *augmented* implicit function $\bar{f} : (\mathbf{p}, \boldsymbol{\theta}) \mapsto (s, \boldsymbol{c})$ that not only returns the scalar value $s \in \mathbb{R}$ but also a feature vector $\boldsymbol{c} \in C$ meant to uniquely identify what role the position $\mathbf{p}$ plays in the instance $\Phi(\boldsymbol{\theta})$. This so-called co-parameter $\boldsymbol{c}$ formally characterizes the notion of *same point* in a non-ambiguous way, while maintaining robustness to procedural parameters changes. More formally, $\bar{f}(\mathbf{p}, \boldsymbol{\theta}) = \bar{f}(\mathbf{p}', \boldsymbol{\theta}')$ if and only if $\mathbf{p}$ and $\mathbf{p}'$ are two positions of the same element of geometry under different procedural parameters $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ (Figure 4). Hence $\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ is defined as the only point such that $\bar{f}(\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}, \boldsymbol{\theta} + \Delta\boldsymbol{\theta}) = \bar{f}(\mathbf{p}^{\boldsymbol{\theta}}, \boldsymbol{\theta})$. Note that s is always 0 for surface points, so $\boldsymbol{c}$ is what enforces point identity.

Our co-parameter space $C = \mathbb{R}^3 \times \mathbb{N}$ is detailed in Section 4.2, as well as how we build $\bar{f}$ in practice. Note that contrary to Michel and Boubekeur [2021], we define the co-parameterization on the whole space rather than only on the surface, due to the implicit nature of our shapes.

4.2 Augmented Implicit Function

The scene graph from which our implicit function is derived carries semantic information that also suggests a notion of what *same point* means. This section describes how we modify the construction of the implicit function to encode this extra information as a co-parameterization that we can then use in our solver. Our requirements for the definition of the co-parameterization are **(a)** to uniquely and robustly identify elements of geometry **(b)** to be locally differentiable (Section 5) and **(c)** to be automatically constructed

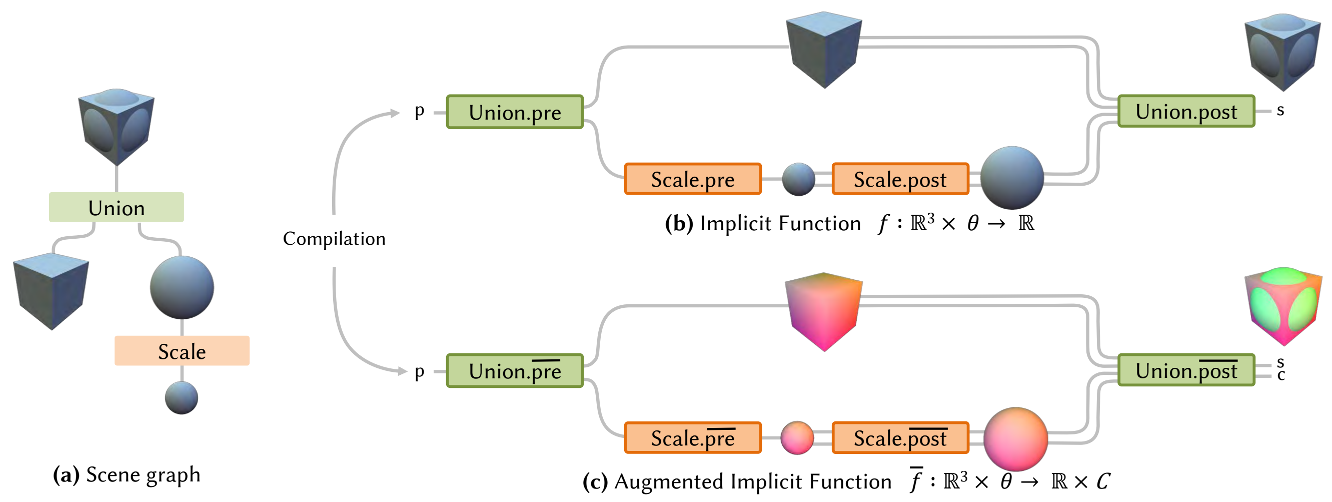

Union
Scale
(a) Scene graph
Compilation
p
Union.pre
Scale.pre
Scale.post
Union.post
s
(b) Implicit Function $f : \mathbb{R}^3 \times \theta \to \mathbb{R}$
p
$\mathrm{Union.}\overline{\mathrm{pre}}$
$\mathrm{Scale.}\overline{\mathrm{pre}}$
$\mathrm{Scale.}\overline{\mathrm{post}}$
$\mathrm{Union.}\overline{\mathrm{post}}$
s
c
(c) Augmented Implicit Function $\bar{f} : \mathbb{R}^3 \times \theta \to \mathbb{R} \times C$

Fig. 5. The implicit function f and the augmented implicit function $\bar{f}$ can be derived from the scene graph **(a)** via a compilation process. While the former **(b)** only provides the distance output for each position **p** it is queried on, the latter **(c)** augments such information with a co-parameter $\boldsymbol{c} = (\boldsymbol{a}, pid)$.

Table 1. Implicit Function f and its augmented counterpart $\bar{f}$ compiled from the scene graph in Figure 5. The latter defines a new output type that carries both the distance and the co-parameters that is computed and propagated by the augmented functions. The second row shows the exact implementation of the `sphereEval()` function in both scenarios, highlighting the co-parameter computation (right). The remaining function implementation is reported in supplemental file *augmented_eval_pre_post.glsl*.

(b) Implicit Function f

```
float f(in vec3 pos, float[5] params){
  // scale = params[0] and so on

  vec3[2] positions = unionPre(pos);
  vec2 scaledPos = scalePre(positions[0], scale);

  float sphere = sphereEval(scaledPos, radius);
  float box = boxEval(positions[1], dims);

  float scaledSphere = scalePost(sphere, scale);
  return unionPost(scaledSphere, box);
}
```

```
float sphereEval(in vec3 pos, float radius){
  float sdf = length(pos) - radius;
  return sdf;
}
```

(c) Augmented Implicit Function $\bar{f}$

```
struct AugmOutput {
    float sdf;
    vec4 coparam;
};

AugmOutput augmentedF(in vec3 pos, float[5] params){
  // scale = params[0] and so on

  vec3[2] positions = unionPre(pos);
  vec2 scaledPos = scalePre(positions[0], scale);

  AugmOutput sphere = sphereEval(scaledPos, radius);
  AugmOutput box = boxEval(positions[1], dims);

  AugmOutput scaledSphere = scalePost(sphere, scale);
  return unionPost(scaledSphere, box);
}
```

```
AugmOutput sphereEval(in vec3 pos, float radius) {
  float sdf = length(pos) - radius;
  vec4 coparam = vec4(pos / radius, 0.0);
  return AugmOutput(sdf, coparam);
}
```

for all of our 3D implicit shapes. To ensure this last point, we build the co-parameterization together with the implicit function f, by defining an *augmented* version of the `eval`, `pre` and `post` functions we described in Section 3:

$$\overline{\texttt{eval}} : \mathbb{R}^3 \longrightarrow \mathbb{R} \times C$$
$$\overline{\texttt{pre}} : \mathbb{R}^3 \longrightarrow \left(\mathbb{R}^3\right)^m \qquad \text{(Unchanged)}$$
$$\overline{\texttt{post}} : \left(\mathbb{R} \times C\right)^m \longrightarrow \mathbb{R} \times C$$

We provide these augmented functions only once for each type of primitive and operator. Algorithm 1 remains the same, but now defines an *augmented* implicit function $\bar{f} : (\mathbf{p}, \boldsymbol{\theta}) \mapsto (s, \boldsymbol{c})$ that returns both the scalar value $s \in \mathbb{R}$ and the co-parameter $\boldsymbol{c} \in C$ at the evaluated position. The co-parameter $\boldsymbol{c} = (\boldsymbol{a}, pid)$ is made of a differentiable part $\boldsymbol{a} \in \mathbb{R}^3$ and a *path index* part $pid \in \mathbb{N}$ that uniquely identifies the path followed in the scene graph during the evaluation of a point. The following paragraphs describe rules of thumb for defining the augmented version of $\overline{\texttt{eval}}$ and $\overline{\texttt{post}}$. Additional material lists formulas for the 29 node types that we support in practice.

Primitive nodes. These represent atomic shapes, and are often derived from a fixed *canonical shape* that is only rigidly transformed by the procedural parameters (e.g., ellipsoid, prism, cylinder, etc.). In this case, we use the position of a point $\mathbf{p}$ in this canonical space as its unique identifier $\boldsymbol{a}$, and the path index pid is always 0 for a primitive. This idea can be generalized to other shapes as illustrated in the additional material, sometimes at the cost of a local discontinuity (e.g., we parameterize the torus as a bent cylinder). While these primitives represent basic shapes, our framework enables highly complex shape design through operator nodes, allowing for vast combination of primitives, as it is typically done in implicit modeling [Wyvill et al. 1999].

Operators nodes. Operator nodes forward the co-parameter received from their input while evaluating a given point position $\mathbf{p}$. While for 1-input operators (e.g., rotation, scaling, bending) the single input co-parameter is simply passed to the next node, in a 2-input operator (e.g., boolean and smooth boolean operations) only a single co-parameter is forwarded alongside the actual operator output. In general operators may introduce ambiguity in the co-parameterization in two ways: by combining multiple inputs, and by duplicating geometry. Both potentially lead to multiple points sharing the same co-parameter, which we avoid by adding the integer part pid of the co-parameterization. A 2-input operator offsets the path index pid that it receives from its second input by 1+ the maximum pid it may receive from its first input. Its output pid is forwarded from either of its inputs depending on its behavior (see proof in additional material). Operators with more than 2 inputs are decomposed into sub steps, and duplication operators are treated as chains of 2-input unions. This allows us to manipulate many different implicit operators (including Boolean and smooth Boolean operators, affine transformations, and warping). In case manipulation occurs at the intersection of two shapes, the co-parameter propagation scheme does not prevent the user from editing smooth Boolean operator parameters (like the smoothness factor, as illustrated in the accompanying video), even with the propagation of a single co-parameter from the inputs. In fact, this is mitigated by the sampling of multiple points in the neighbouring patch of the surface, having some of them falling in each primitive. We discuss in Section 7 more challenging operators that we do not support. Note that the integer part pid is ignored when differentiating the co-parameter, but is used in the solver to control that the dragged point is properly tracked (see Section 6).

Scene graph compilation example. Starting from a scene graph representation, we derive the corresponding implicit function by recursively concatenating the `pre`, `eval` and `post` functions of each node, starting from the root one. This is illustrated in Algorithm 1. The same procedure is adopted to obtain the augmented function $\bar{f}$, using the augmented counterparts of the functions. In this section, we detail step by step a full example of compilation from the input scene graph down to the implicit and augmented implicit functions for the example scene graph shown in Figure 5.

The scene from Figure 5 is made up of 2 primitives, namely the *Sphere* and the *Box*; the former is transformed using the unary operator *Scale* and finally combined with the latter via the binary *Union* operator to create the final shape. By recursively following Algorithm 1, we start from the *Union* node and apply the `CompileNode` function to its children, which are the *Box* primitive and modifier *Scale*, that can be further unrolled until the base case *Sphere* is reached. The implicit function $f : \mathbb{R}^3 \times \Theta \rightarrow \mathbb{R}$ derived from the scene graph is reported in the first column of Table 1.

While the function f only evaluates the distance of a point in space from the implicit shape, we need additional information to allow the direct manipulation of the implicit shape itself. We apply the same Algorithm 1 but replace each node's function with their augmented version $\overline{\texttt{pre}}$, $\overline{\texttt{eval}}$ and $\overline{\texttt{post}}$, thus computing and propagating both the distance and the co-parameter for each evaluated position. The derived augmented implicit function $\bar{f} : \mathbb{R}^3 \times \Theta \rightarrow (\mathbb{R} \times C)$ is reported in the second column of Table 1.

Exploiting the information propagated by the augmented output makes it possible to track the same point during an edit, and allows estimating the influence of the procedural parameters over them.

5 EVALUATION AND NORMALIZATION

The computation of Jacobians with respect to procedural parameters is a core step in the optimization process. Here we define how to compute and refine them accordingly to the user selection (Figure 7).

5.1 Jacobian Evaluation

Minimizing the direct manipulation loss $\mathcal{L}_i$ from Equation 2 requires to evaluate at $\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ the Jacobian $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ of the position $\mathbf{p}$ of a point with respect to procedural parameters $\boldsymbol{\theta}$, at a fixed point identity $\boldsymbol{c}$, as described in Sec. 6. For each position $\mathbf{p}$ of a point, the Jacobian is a $3 \times n$ matrix, where n is the number of procedural parameters. Fortunately, this matrix can be derived from the Jacobian of $\bar{f} : (\mathbf{p}, \boldsymbol{\theta}) \mapsto (s, \boldsymbol{c})$ by applying the implicit function theorem:

$$\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}} = -\left(\frac{\partial \bar{f}}{\partial \mathbf{p}}\right)^{+} \cdot \frac{\partial \bar{f}}{\partial \boldsymbol{\theta}} \tag{3}$$

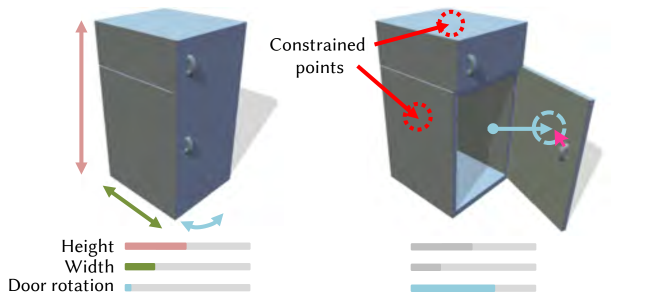


Fig. 6. To reduce ambiguities during an edit, our framework enables multi-point constraints over parts of the shape. This guides the optimization towards a procedural parameter update that does not affect the constrained areas, thus increasing the framework expressiveness.

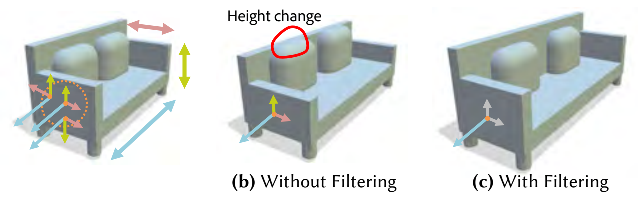


Fig. 7. We automatically filter out some columns of the Jacobian matrix to deactivate procedural parameters that are deemed less relevant to the edit. Here, a selection on the side of the sofa suggests an edit involving the width rather than the height or depth. Without filtering **(b)**, all dimensions are updated, resulting in undesirable changes in height and depth, while our filtering discards the height and depth dimensions, focusing on width **(c)**.

where $(\cdot)^+$ denotes the pseudo-inverse of a matrix. We use automatic differentiation to evaluate the Jacobian of all outputs of $\bar{f}$ with respect to both the procedural parameters $\boldsymbol{\theta}$ and the position $\mathbf{p}$.

However, simply evaluating the point-wise Jacobian of our procedural shape has two major drawbacks that we need to address: first, the different columns of the Jacobian – which relate to different procedural parameters – are *not homogeneous* in terms of scales and units. For this, we perform a normalization step that we detail in Section 5.2. Second, the differential information is only valid for a single point. When selecting a patch of the surface where many points are involved, each of them may be influenced by different procedural parameters. This may result in the simultaneous update of several different parameters, ending in a less controllable interaction as the user performs the edit. There is thus a need for a filtering step where we zero the values that relates to procedural parameters that we do not aim to modify (Section 5.3).

5.2 Jacobian Normalization

Switching e.g., a length parameter from meters to millimeters divides by a factor 1000 the corresponding columns of the Jacobian $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ and thus leads to gradient descent updates 1000 times slower only for this parameter. To prevent this, we estimate a normalization factor for each procedural parameter of the model. First, as a preprocessing step, we randomly sample 50 ray directions from the six viewpoints aligned with the canonical axes and evaluate the Jacobian $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ at each intersection between a traced ray and the implicit shape. The normalization factor m_i of the i-th procedural parameter is then defined as the maximum magnitude of the i-th column of the Jacobian over all samples. During editing, we similarly update the normalization factors m_i after each user edit and each change of viewpoint. Normalization factors are exploited in the Jacobian filtering process (Section 5.3), enabling a direct and robust comparison between parameters, and are also used to scale the gradient of the loss function $\nabla\mathcal{L}$ during the optimization (Section 6).

5.3 Jacobian Reduction and Filtering

To increase the robustness of the Jacobian, we estimate it for a neighboring patch of surface rather than at a single point. We evaluate $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ at 16 sample points within a screen-space disk centered on the user's mouse cursor and reduce them to their average $3 \times n$ matrix.

We then filter this patch-wise Jacobian matrix, noted $J(\mathbf{p}, \boldsymbol{\theta})$, by cancelling out procedural parameters that do not influence enough the position of the patch. The process of filtering the procedural parameters does not require any input from the user, and is automatically performed by extracting information from the Jacobian matrices themselves. We filter the i-th column of the Jacobian by estimating the influence of the i-th procedural parameter on the geometry. To be preserved, each Jacobian column $\boldsymbol{j}_i$ has to respect at least two of the three following conditions. **(a)** Its normalized magnitude $\frac{\|j_i\|}{m_i}$ must be higher than an empirical threshold $\lambda_{mag} = 0.35$, aiming to keep dimensions with large impact on the shape geometry. **(b)** The standard deviation of $\boldsymbol{j}_i$ across all selected points must be lower than an empirical threshold $\lambda_{std} = 0.2$, aiming to keep dimensions that behaves similarly across the patch. **(c)** The angular distance $d_v \cdot \frac{j_i}{\|j_i\|}$, where d_v denotes the view direction, must be lower than an empirical threshold $\lambda_{view} = 0.4$ to foster dimensions whose impact is orthogonal to the view direction.

6 SOLVING

Thanks to the evaluation of the filtered and reduced Jacobian $J(\mathbf{p}, \boldsymbol{\theta})$ of $\mathbf{p}$ with respect to $\boldsymbol{\theta}$, we can integrate the manipulated shape in generic continuous optimization frameworks. At each frame of the interaction, we use a few steps of gradient descent to minimize the following multi-point manipulation loss:

$$\mathcal{L} = \sum_i \mathcal{L}_i + \lambda \mathcal{L}_{reg} \tag{4}$$

where $\mathcal{L}_{reg} = \|\Delta\boldsymbol{\theta}\|_2$ is a regularization term that prevents sudden changes in procedural parameters and $\lambda = 0.2$. The gradient of $\mathcal{L}_i$ with respect to $\Delta\boldsymbol{\theta}$ is:

$$\nabla\mathcal{L}_i = J_{\text{Proj}} \cdot J(\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}, \boldsymbol{\theta} + \Delta\boldsymbol{\theta}) \tag{5}$$

When updating $\boldsymbol{\theta}$ at each step of the gradient descent, we divide coefficient-wise the gradient $\nabla\mathcal{L}$ by the vector m of normalization factors (Section 5.2). Since the surface is only implicit, the updated positions $\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ are estimated as part of the optimization (Equation 8), and the confidence of this estimation is measured by the difference $\|\Delta c\| = \|\Delta a\|_2 + \delta_{\Delta pid,0}$ between the initial co-parameter of $\mathbf{p}_i^{\boldsymbol{\theta}}$ and the one evaluated at $\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$. When this error exceeds a

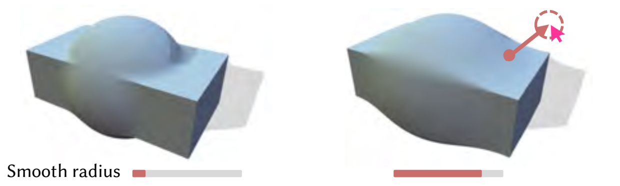


Fig. 8. Smooth Booleans are key operators of implicit shape modeling, as they blend shapes together in a more organic way than hard Booleans (left). Our framework naturally supports updating the smoothness parameter, by dragging a point from the smooth junction between the two shapes (right).

threshold $e_c = 0.7$, we divide the gradient by $||\Delta c||$ to slow down the drift. The scaled gradient is then multiplied by the global learning rate η. We then update the estimate $\mathbf{p}_i$ of the 3D position $\mathbf{p}_i^{\theta+\Delta\theta}$ of the dragged point:

$$\Delta\boldsymbol{\theta} := -\eta\nabla\mathcal{L}/m \tag{6}$$

$$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta} \tag{7}$$

$$\mathbf{p}_i \longleftarrow \mathbf{p}_i + J(\mathbf{p}_i, \boldsymbol{\theta}) \cdot \Delta\boldsymbol{\theta} \qquad \forall i \tag{8}$$

7 RESULTS

We implemented our method as well as our implicit primitives and operators in C++/GLSL. We use libfive trees [Keeter 2019] as target to the compilation of our scene graph representation (Algorithm 1), which provides us with symbolic expression optimization and numeric automatic differentiation. All models shown throughout this paper (Figure 1, 6, 7, 10, 11) were rendered using sphere tracing [Hart 1996] in a standalone application (see accompanying video). Experiments were performed on a desktop computer equipped with AMD® Ryzen 5 clocked at 3.6 GHz with 32 GB of RAM, and an NVIDIA GTX 1050 graphics card. Statistics for the models shown throughout this paper and performances for the different steps of the pipeline are reported in Table 2.

Performance. As illustrated in the accompanying video, our method runs at interactive framerates (including manipulation and rendering), enabling direct manipulation by the user without the need to wait for any sort of loading. The most computationally intensive part is the solving (Table 2), where the maximum number of gradient descent iterations is set to 50, which we found to be a good trade off between quality and performance (see additional material for further discussion). To achieve interactivity, we optimize the expression of $\bar{f}$ using libfive expression optimization feature at initialization. As a further improvement in speed, the Jacobians used during solving are updated every 4 frames.

Control. We illustrate our direct manipulation tool on a set of 11 procedural implicit surfaces with varying complexity and topology. Scene graph complexity spans from a minimum of 19 nodes (Figure 6) to 265 nodes for the roller model in Figure 1, with a number of procedural parameters ranging from 4 to 45. Figure 11 shows editing sessions with three successive edits to these models. A typical workflow in our framework involves fixing some parts of the implicit surface while dragging some other parts, as highlighted in Figure 1, 6, and 11. On top of our Jacobian filtering (Section 5.3), this helps the solver disambiguate the procedural parameter update and allows for more intuitive edits, even when warping and affine transformations are involved. For instance, the House model combines a bend and a twist, that can both be manipulated separately when the right fixed constraints are provided. Another example is the Robot arm, which involves chained rotations that can be controlled independently by fixing points on the different joints.

Our framework is also resilient to changes in topology induced by CSG operators (union, intersection, and difference). This represents an important feature, since the ease with which one can create varying topology is one of the key strengths of implicit modeling. So, the manipulation of shapes whose topology varies when altering its procedural parameters is supported by the presence of a unique path index (*pid*) in the co-parameter that identifies the dragged points. The support to topology changes is highlighted in Figure 9 as well as by edits performed on the pipe and cheese models (Figure 11).

Table 2. Performance for the different scenes, with the amount #*nodes* of nodes in the scene graph, the #$\boldsymbol{\theta}$ of procedural parameters and the #$\boldsymbol{\theta}_e$ changed during the edit. We report the execution time for the different steps of an edit, namely the co-parameter sampling time t_c, the Jacobian evaluation time t_j and the average solving time t_s for the 50 optimization iterations. All timings are in ms.

Scene	Fig.	#$\boldsymbol{\theta}$	#$\boldsymbol{\theta}_e$	#*nodes*	t_c	t_j	t_s
Roller	1	8	1	265	1.524	2.365	9.903
Cup	2	5	3	39	2.523	0.151	0.548
Fridge	6	3	1	19	2.325	0.214	0.381
Sofa	7	13	1	77	2.319	0.678	1.652
Webcam	10	5	1	206	1.628	0.935	3.503
Cheese	11a	27	2	158	1.897	1.932	3.145
Robot Arm	11b	6	1	243	2.061	0.732	2.831
Pipes	11c	6	2	249	1.559	1.510	4.152
Toaster	11d	4	1	256	1.904	1.126	4.811
House	11e	20	4	244	2.137	0.999	2.313
Rabbit	11f	45	5	188	1.484	3.980	5.928

Comparison with other techniques. Our method is focused on the direct manipulation of analytic implicit surfaces defined as procedural scene graphs. We support the classical operators of implicit modeling, such as CSG operators that changes the topology of the model, morphological operators (e.g., dilatation) and smooth Booleans (see Figure 8), bringing the power of direct manipulation techniques performed on meshes [Cascaval et al. 2022; Gaillard et al. 2022; Michel and Boubekeur 2021] to procedural implicit surfaces.

Close to our method is the direct manipulation solution exposed in libfive [Keeter 2019], which allows manipulation of analytic implicit surfaces. Their solver tries to find a procedural parameter update such that *some* part of the surface passes by the new mouse position, back-projected into the 3D space along the normal of the base position on the model. However, it has no way to identify *which* part of the shape the user intends to modify. Although libfive's heuristic behaves well for procedural parameters that move elements of surface along their normal, it struggles with tangential movements, due to its lack of awareness of a point's identity. In

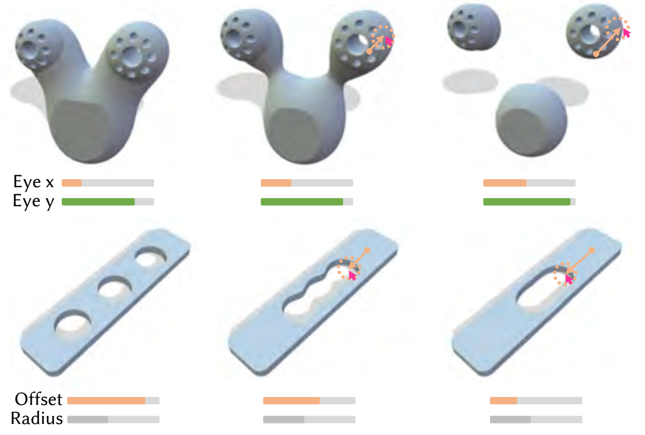


Fig. 9. Examples featuring changes of topology. Our method naturally inherits from the ability of procedural implicit surfaces to represent objects of varying topology, be it through additive or subtractive, smooth or hard Boolean operations.

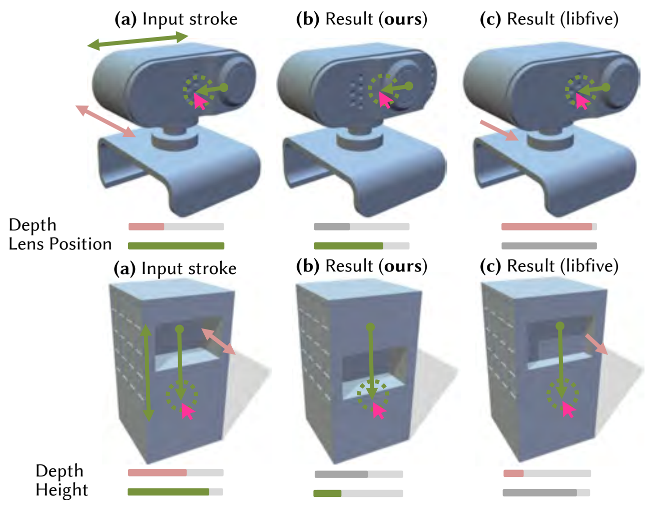


Fig. 10. When the user intends to drag points in a direction significantly different than the local surface normal **(a)**, our direct manipulation approach keeps track of the dragged point **(b)** while libfive's solver only constrains that the overall surface passes by the new mouse position **(c)**. In top example, libfive is not able to affect the lens position parameter, while in bottom example the opening depth and not the position is affected when solving for the edit.

contrast, our solution appropriately evaluates the local influence of a procedural parameter in all directions equally.

Figure 10 shows two cases where libfive's normal-aligned paradigm fails in updating the procedural parameters while matching the user edit. In top row example, the user manages to modify the lens position of the camera using our approach, while using libfive's solver it only manages to increase the camera depth, which is in fact aligned to the dragged point normal. Similarly in bottom row, the vertical user edit is correctly mapped to a change in the opening height using our method, while it is interpreted as a opening depth update using libfive's solver. A comparison of the behaviour of both solvers on a sample 2D case is reported in the additional material.

User Study. We evaluated the effectiveness of our method via a user study involving 20 subjects, with well-spread background in 3D editing programs. The majority of them confirmed they use 3D editing programs from time to time, but we also recorded the activity of real novice users up to proficient ones. A similar distribution can be observed for users' affinity with procedural modeling, with only a few of them being total novices to the concept.

The user study starts with an hands-on session, where the users could familiarize with both slider-based and direct manipulation interactions tools. Then, a more comprehensive editing session of five tasks is performed. Each task requires the user to reach a target shape configuration shown, in a provided on-screen reference image. Users are guided to reach the target using slider-only interaction in the first task and direct manipulation for the remaining ones. Before moving to the next task in the editing session, users are required to provide feedback about their ability to reach the provided target. After the complete editing session, they were also provided a more detailed questionnaire.

Users found interacting with a direct manipulation tool quite easy, overall being able to directly edit a desired parameter. Specifically, it emerged that 95% (all but 1 subject) found the direct manipulation tool to be reactive to their inputs, and the 55% assessed that they rarely or never find themselves preferring to use sliders over direct manipulation.

Out of the 100 tests collectively performed by the users, they admitted not being able to reach the target only eight times, and this has often happened in the presence of models involving deformations, as also reported by users in the final questionnaire. Some of them admitted to having experienced frustration while interacting with our tool, with the common cause being having multiple parameters change at the same time, providing feedback like *"It is sometimes difficult to isolate the exact parameter you want to tune"*. This may suggest more focus on the parameter reduction strategies, which are already being considered for future exploration. They reported to not be able to precisely locate the part of the model that needed to be selected to perform a change in parameters, which could be mitigated by visualizing more information directly on the shape.

However, when compared to slider-based editing, the majority of the users admitted to prefer direct manipulation to adjust parameter values and, moreover, all subjects admitted they would interact through direct manipulation if another scene was presented. All the subjects regularly using 3D editing programs confirmed that they would frequently use a direct manipulation tool if integrated in their favorite program. Further details regarding the user study setting and the questionnaire answers are reported in the additional materials.

Limitations and Future Work. Our method allows the direct manipulation of analytic implicit surfaces, but does not come without limitations. First, an explicit co-parameter must be derived for primitives and operators that we aim at editing, which may not be trivial.

Moreover, co-parameterization must remain injective, which is not fulfilled in some edge cases, e.g., when the size of a box becomes 0. In this case, multiple points end up at the same position.

While our framework supports many different primitives and operators (see supplemental material for the derivation of all nodes), some remain to be integrated. For instance, we do not support domain repetition operators, which requires to discriminate points belonging to different instances by producing a different *pid* for each one. Another example of unsupported operator is morph between two shaped. Although co-parameter is coherently defined for each shape, our definition does not ensure consistent interpolation. This may end in the morphing operator leading to an unintuitive manipulation (see supplemental for an example).

Second limitation is related to the different nature of procedural parameters user while modeling. Regarding discrete procedural parameters, our system could support the ones that can be defined as rounded versions of underlying continuous ones (e.g., the repetition of a shape in a radial or axis-aligned arrangement, usually described as a modulus operation between a position and a distance parameter). Oppositely, a parameter that is fundamentally discrete, e.g., a "primitive type" parameter that switches between two completely different primitives, is not supported as it is hard to even qualitatively define what the user would expect while changing it.

We also found that our Jacobian filtering (Section 5.3) does not perform as well for models with too many procedural parameters influencing the same patch. This is partially solved by using multiple fixed constraints for editing, but future work may investigate more advanced filtering and reduction strategies, as also suggested by the results of our user study. Finally, our current gradient descent optimization (Section 6) could benefit from more advanced techniques, such as ADAM optimization, or even Natural Gradient Descent as our total number of parameters remains small.

8 CONCLUSION

In this work, we proposed a direct manipulation approach for procedural implicit surfaces. We automatically augment the implicit function to output a *co-parameter*, allowing to robustly track the same point location throughout an edit. We leverage this to enable users to directly drag parts of the shape in the viewport, as opposed to tediously editing sliders, and more generally open the opportunity to evaluate the local influence of each procedural parameter on individual points of a shape. Our framework supports the direct manipulation of implicits made of a large set of primitives and complex operators, including warping and affine transformations.

REFERENCES

2022. MagicaCSG. http://ephtracy.github.io/index.html?page=magicacsg.

2022. Womp 3D Inc. https://womp.com/.

Baptiste Angles, Marco Tarini, Brian Wyvill, Loïc Barthe, and Andrea Tagliasacchi. 2017. Sketch-Based Implicit Blending. *ACM Trans. Graph.* 36, 6 (2017), 13 pages.

K. Anjyo and J. Lewis. 2010. Direct Manipulation Blendshapes. *IEEE Computer Graphics and Applications* 30, 04 (jul 2010), 42–50. https://doi.org/10.1109/MCG.2010.41

A. Aristidou, J. Lasenby, Y. Chrysanthou, and A. Shamir. 2018. Inverse Kinematics Techniques in Computer Graphics: A Survey. *Computer Graphics Forum* 37, 6 (2018), 35–58. https://doi.org/10.1111/cgf.13310

Aurélien Barbier, Eric Galin, and Samir Akkouche. 2005. A framework for modeling, animating, and morphing textured implicit models. *Graphical Models* 67, 3 (2005), 166–188.

Martin Bokeloh, Michael Wand, Vladlen Koltun, and Hans-Peter Seidel. 2011. Pattern-aware shape deformation using sliding dockers. *ACM Trans. Graph.* 30, 6 (2011), 1–10.

Martin Bokeloh, Michael Wand, Hans-Peter Seidel, and Vladlen Koltun. 2012. An algebraic model for parameterized shape editing. *ACM Trans. Graph.* 31, 4 (jul 2012), 10 pages.

Ronan Boulic and Ramon Mas. 1996. *Hierarchical Kinematic Behaviors for Complex Articulated Figures.* Prentice-Hall, Inc., USA, 40–70.

D. Cascaval, M. Shalah, P. Quinn, R. Bodik, M. Agrawala, and A. Schulz. 2022. Differentiable 3D CAD Programs for Bidirectional Editing. *Computer Graphics Forum* 41, 2 (2022), 309–323. https://doi.org/10.1111/cgf.14476 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14476

Manolya Eyiyurekli and David E. Breen. 2017. Detail-preserving level set surface editing and geometric texture transfer. *Graphical Models* 93 (2017), 39–52.

Mathieu Gaillard, Vojtech Krs, Giorgio Gori, Radomir Mech, and Bedrich Benes. 2022. Automatic Differentiable Procedural Modeling. *Computer Graphics Forum* (2022).

Michael Lee Gleicher. 1994. *A differential approach to graphical interaction.* Carnegie Mellon University.

John C. Hart. 1996. Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces. *The Visual Computer* 12, 10 (1996), 527–545.

Pol Jeremias and Inigo Quilez. 2014. ShaderToy. https://www.shadertoy.com/.

Matt Keeter. 2019. libfive: Infrastructure for solid modeling. https://libfive.com/.

Élie Michel and Tamy Boubekeur. 2021. DAG Amendment for Inverse Control of Parametric Shapes. *ACM Transactions on Graphics* 40, 4 (2021), 173:1–173:14.

Ken Museth, David Breen, Ross Whitaker, and Alan Barr. 2002. Level Set Surface Editing Operators. *Level Set Surface Editing Operators, SIGGRAPH* 21 (05 2002).

Tim Reiner, Gregor Mückl, and Carsten Dachsbacher. 2011. Interactive Modeling of Implicit Surfaces Using a Direct Visualization Approach with Signed Distance Functions. *Computer & Graphics, Proceedings of Shape Modeling International* 35, 3 (2011), 596–603.

Marzia Riso and Fabio Pellacini. 2023. pEt: Direct Manipulation of Differentiable Vector Patterns. In *Eurographics Symposium on Rendering.*

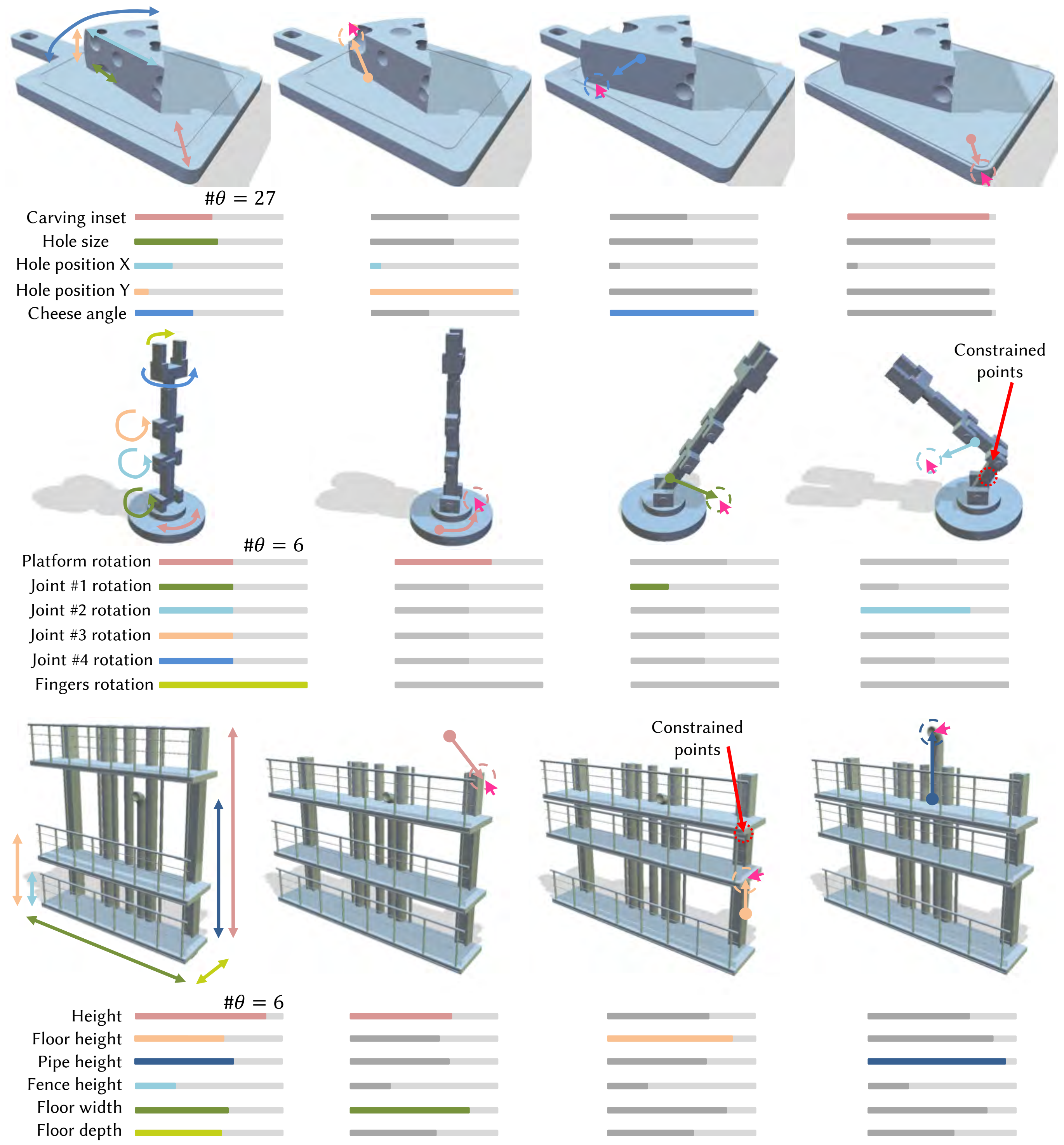
R. Schmidt, B. Wyvill, M. C. Sousa, and J. A. Jorge. 2006. ShapeShop: Sketch-Based Solid Modeling with BlobTrees *(SIGGRAPH '06).*

Masamichi Sugihara, Erwin de Groot, Brian Wyvill, and Ryan Schmidt. 2008. A Sketch-Based Method to Control Deformation in a Skeletal Implicit Surface Modeler. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling.*

Masamichi Sugihara, Brian Wyvill, and Ryan Schmidt. 2010. WarpCurves: A tool for explicit manipulation of implicit surfaces. *Computers & Graphics* 34, 3 (2010), 282–291.

Brian Wyvill, Andrew Guy, and Éric Galin. 1999. Extending the CSG Tree - Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *Computer Graphics Forum* 18, 2 (1999), 149–158.

C. Zanni, M. Gleicher, and M.-P. Cani. 2015. N-Ary Implicit Blends with Topology Control. *Comput. Graph.* 46 (2015), 1–13.


$\#\theta = 27$
Carving inset
Hole size
Hole position X
Hole position Y
Cheese angle
Constrained points
$\#\theta = 6$
Platform rotation
Joint #1 rotation
Joint #2 rotation
Joint #3 rotation
Joint #4 rotation
Fingers rotation
Constrained points
$\#\theta = 6$
Height
Floor height
Pipe height
Fence height
Floor width
Floor depth

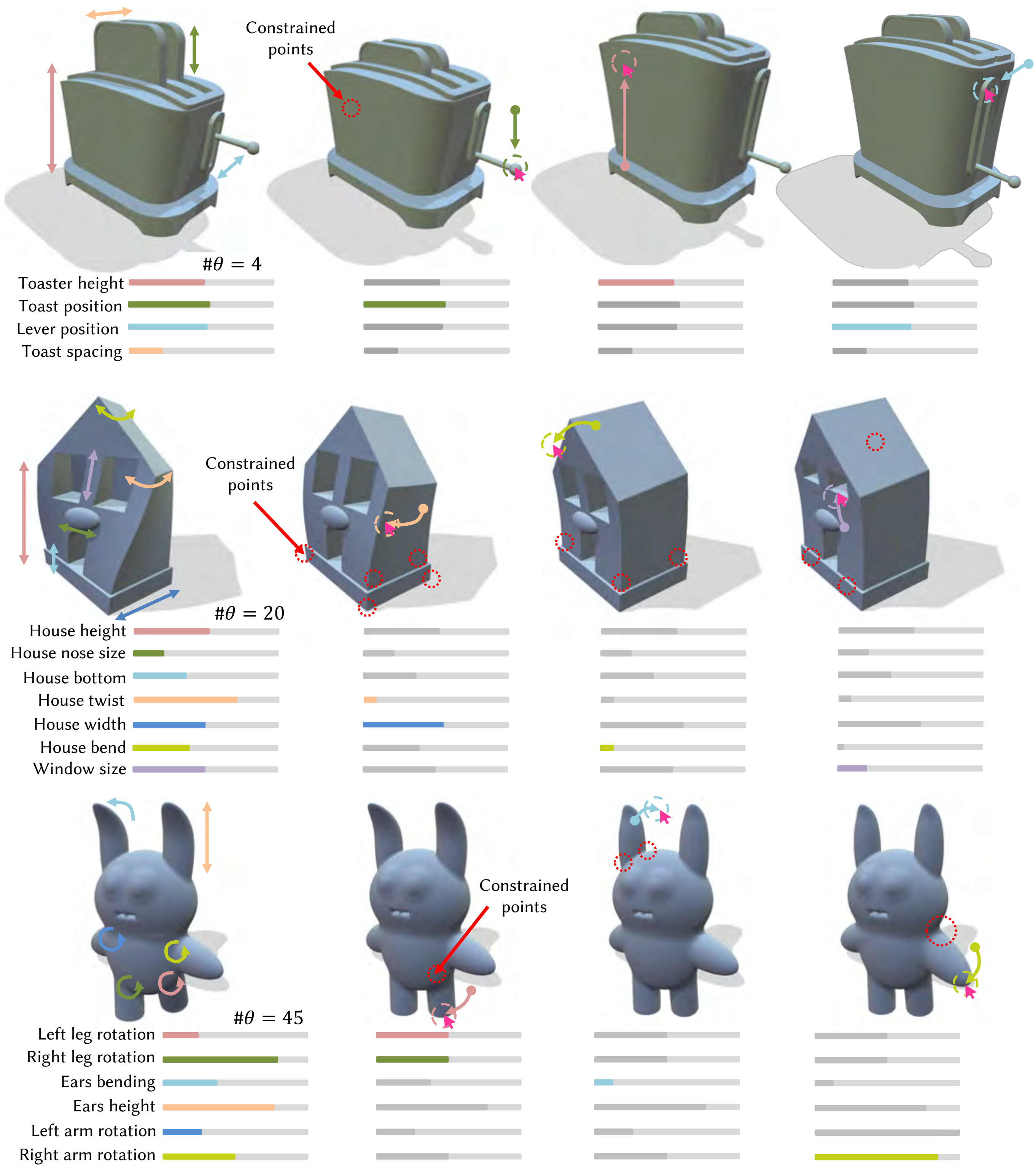


Fig. 11. Editing sessions performed using our framework. The first image represents the original procedural implicit shape with a simplified semantical representation of its procedural parameters. The remaining images show three consecutive edits that are performed on the implicit shape, including both constrained and unconstrained manipulations. For each edit, we report the selected points and the mouse trajectory, highlighting the procedural parameter update in the underlying sliders.