

Direct Manipulation of Procedural Implicit Surfaces

SUPPLEMENTARY MATERIAL

MARZIA RISO, Sapienza University of Rome, Italy and Adobe, France

ÉLIE MICHEL, Adobe, France

AXEL PARIS, Adobe, France

VALENTIN DESCHAINTE, Adobe, United Kingdom

MATHIEU GAILLARD, Adobe, USA

FABIO PELLACINI, University of Modena and Reggio Emilia, Italy

ACM Reference Format:

Marzia Riso, Élie Michel, Axel Paris, Valentin Deschaintre, Mathieu Gaillard, and Fabio Pellacini. 2024. Direct Manipulation of Procedural Implicit Surfaces SUPPLEMENTARY MATERIAL. *ACM Trans. Graph.* 1, 1 (September 2024), 5 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 ADDITIONAL DOCUMENTS

Besides the current document, the supplementary material of our paper consists in the following attachments:

- A detailed description of our **user study** and its results.
- The GLSL **source code** that we used to build our augmented implicit function \tilde{f} .
- A **video**, whose first part gives the overview of our method while the second part showcases interactive results.

2 CO-PARAMETERIZATION FOR PRIMITIVES AND OPERATORS

Our method requires an explicit definition of the co-parameter c for all primitives and operators that we want to support. In other terms, it requires the definition of an `eval` function per primitive, and a `post` function for every operator, each one augmented with the additional co-parameter output. In the current state of our framework we support 29 nodes, split into the following categories: 12 primitives, 11 unary operators, and 6 binary operators. This encompasses all the operations that are classically used in implicit modeling (see Figure 1 for examples deformation operators). Following is the list of primitives and operators our system supports, while the supplemental file *augmented_eval_pre_post.glsl* reports an example implementation in GLSL.

Primitives. Sphere, Box, Halfplane, Cylinder, Ellipsoid, Cone, Capsule, Rhombus, Torus, Hexagonal Prism, Elongated Cylinder, Elongated Ellipsoid.

Authors' addresses: 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Association for Computing Machinery.

0730-0301/2024/9-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Unary Operators. Translate, Rotate, Scale, Bend, Twist, Displace, Mirror X, Mirror Y, Mirror Z, Rounding, Shell.

Binary Operators. Union, Intersection, Difference, Smooth Union, Smooth Intersection, Smooth Difference.

Table 1 and Table 2 reports a the implementation of the `eval`, `pre` and `post` function for some supported primitives and operators.

Table 1. Implementation of the `eval` functions for some supported primitives. Given a 3D point location, each function computes the distance with the implicit surface and its co-parameterization.

```
AugmOutput evalEllipsoid(in vec3 pos, in vec3 radii){
    float l = length(pos/radii);
    float sdf = (l-1.0) * 1/length(pos/(radii*radii));
    vec4 coparam = vec4(pos/radii, 0.0);
    return AugmOutput(sdf, coparam);
}
```

```
AugmOutput evalCone(in vec3 pos, float height, float
    radius) {
    vec2 h = vec2(length(pos.xz), pos.y - height);
    vec2 g = vec2(radius, -height);
    vec2 u = h-g * clamp(dot(g, h) / dot(g, g), 0.0, 1.0);
    vec2 v = h-g * vec2(clamp(h.x / g.x, 0.0, 1.0), 1.0);
    float s = sign(max((h.y - g.y) * sign(g.y), (
        g.y * h.x - g.x * h.y) * sign(g.y)));
    float sdf = s * sqrt(min(dot(u, u), dot(v, v)));
    vec4 coparam = vec4(pos.x / radius, pos.y / height,
        pos.z / radius, 0.0);
    return AugmOutput(sdf, coparam);
}
```

```
AugmOutput evalTorus(in vec3 pos, float radius_major,
    float radius_minor) {
    vec2 q = vec2(length(pos.xz) - radius_major, pos.y);
    float sdf = length(q) - radius_minor;
    vec4 coparam = vec4(atan(pos.y, length(pos.xy) -
        radius_major), atan(pos.z, pos.x), 0.0, 0.0);
    return AugmOutput(sdf, coparam);
}
```

Table 2. Implementation of the $\overline{\text{pre}}$ and $\overline{\text{post}}$ functions for the Boolean Union, Intersection and Difference operations. The $\overline{\text{pre}}$ function duplicate the position to be evaluated by the operands. Oppositely, the $\overline{\text{post}}$ function computes the output distance value according to the applied boolean operation and coherently returns the co-parameterization of the selected branch.

<pre>vec3[2] unionPre(in vec3 pos){ return vec3[2](pos, pos); }</pre>	<pre>AugmOutput unionPost(in AugmOutput from_inputs[2], int pid_offset) { float sdf = min(from_inputs[0].sdf, from_inputs[1].sdf); vec4 coparam = mix(from_inputs[0].coparam, from_inputs[1].coparam + vec4(0.0, 0.0, 0.0, float(pid_offset)), step(from_inputs[1].sdf, from_inputs[0].sdf)); return AugmOutput(sdf, coparam); }</pre>
<pre>vec3[2] intersectionPre(in vec3 pos){ return vec3[2](pos, pos); }</pre>	<pre>AugmOutput intersectionPost(in AugmOutput from_inputs[2], int pid_offset) { float sdf = max(from_inputs[0].sdf, from_inputs[1].sdf); vec4 coparam = mix(from_inputs[0].coparam, from_inputs[1].coparam + vec4(0.0, 0.0, 0.0, float(pid_offset)), step(from_inputs[0].sdf, from_inputs[1].sdf)); return AugmOutput(sdf, coparam); }</pre>
<pre>vec3[2] differencePre(in vec3 pos){ return vec3[2](pos, pos); }</pre>	<pre>AugmOutput differencePost(in AugmOutput from_inputs[2], int pid_offset) { float sdf = max(from_inputs[0].sdf, -from_inputs[1].sdf); vec4 coparam = mix(from_inputs[0].coparam, from_inputs[1].coparam + vec4(0.0, 0.0, 0.0, float(pid_offset)), step(from_inputs[0].sdf, -from_inputs[1].sdf)); return AugmOutput(sdf, coparam); }</pre>

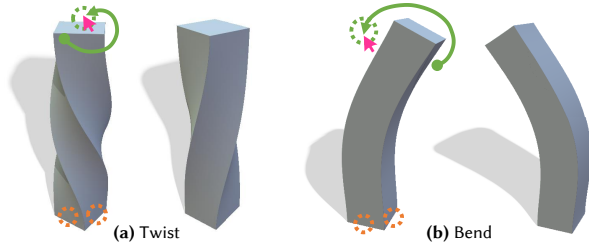


Fig. 1. We support a large set of primitives and operators, including smooth and hard Boolean operations, affine transformations, and deformations. The latter include twists (a) and bends (b) for which the corresponding angle parameters can be edited. Fixed point constraints are circled in orange.

3 JACOBIAN VISUALIZATION

Throughout the paper and in the accompanying video, we show the Jacobians of the position \mathbf{p} with respect to the procedural parameters θ as a set of colored arrows pointing out of sampled positions on the implicit surface. Each arrow represents the influence of the procedural parameters whose slider has the corresponding color. In Figure 2, we better highlight the correspondence between the visualization we provide and the actual Jacobian matrix computed for a

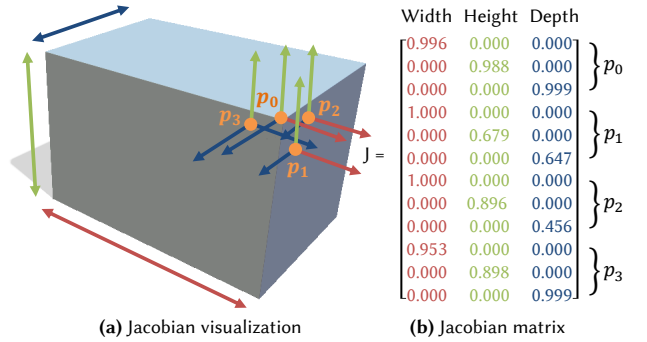


Fig. 2. Jacobians are visualized as arrows pointing out of each sample on the implicit shape (a). Jacobians with respect to the procedural parameters for each point are stored in a $3 \times n$ submatrix of the complete Jacobian one (b).

patch of the surface encompassing 4 points. This visualization also gives an indication to the user over which procedural parameters are going to be modified by the edit, and to which extent.

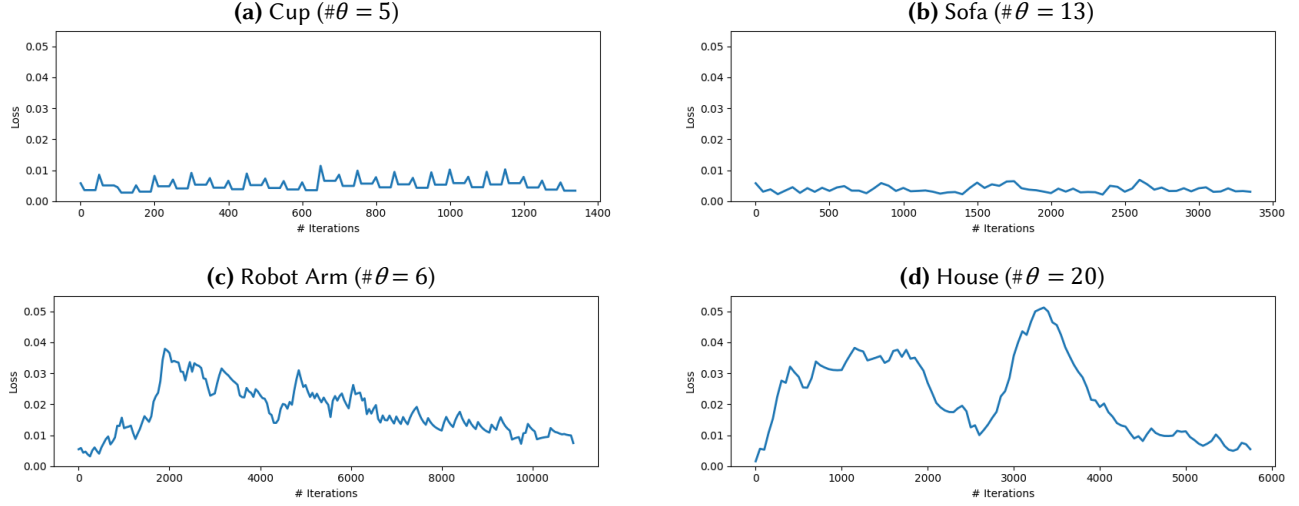


Fig. 3. Loss function variation over the entire optimization process, from the mouse click to its release. The plot refers to the first edit shown in the result Figure 9 in the paper and whose timings are reported in Table ?? . For a small number of nodes in the scene graphs and a reduced number of parameters involved through the edit, the loss function is low and stable, reaching a good convergence at each step (a-b). Conversely, spikes are visible as the number of nodes or parameters involved increases. However, loss values are always relatively small and the optimization still reaches a good convergence.

4 GRADIENT DESCENT DETAILS

The approach we propose computes the procedural parameters update $\Delta\theta$ by minimizing the loss function \mathcal{L} using a gradient descent approach. At each mouse movement event, we refine the currently computed update $\Delta\theta$ by a quantity $\eta\nabla\mathcal{L}/m$, where η is a global learning rate, m is the normalization factor vector, and $\nabla\mathcal{L}$ is the gradient of the loss function estimated using the Jacobians computed from the augmented implicit function \tilde{f} . Every refinement step computes 50 iterations of gradient descent at most, without using any adaptive estimation or momentum. For four tested models, Figure 3 shows the loss function computed while performing the edit reported on the first column of Fig. 9 of the paper, over the entire event until the mouse is released. In both (a) and (b), the visible spikes correspond to the beginning of each mouse move event and its relative optimization loop. In both cases, the loss function does not show a great variation throughout the edit, suggesting that the solver is usually close to the actual solution. Being efficient at computing the function \tilde{f} fastens the entire optimization loop and prevents the target of two consecutive optimization events to be too distant in space, thus often being in a good interval around the solution. On the contrary, examples (c) and (d) refer to shapes defined by 243 and 244 nodes respectively, whose functions are slower to evaluate and whose procedural parameters are intertwined in a complex way. Although in both cases some unregular spikes are visible throughout the edit, mainly in the first half of it, the loss value is relatively low and the optimization shows a good convergence in the last iterations.

5 EXPLICIT SOLVER COMPARISON

One of the advantages of our approach comes from the fact that the estimation of the Jacobian of a position with respect to the

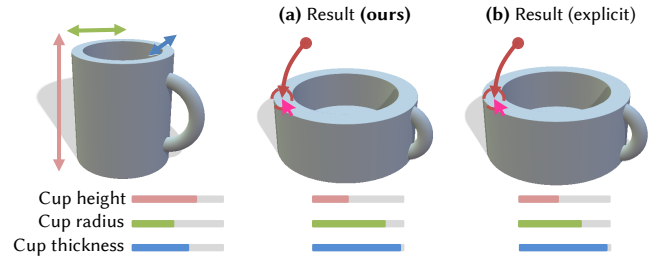


Fig. 4. We directly compare our proposed solver to the one based on the explicit function \mathcal{E} function. Starting from the same configuration (a) and using the exact same sampled points and mouse trajectory, the results obtained using our solver (b) and the explicit function based one are visually similar. Numerically, the L_2 distance between the procedural parameters values is 0.033.

procedural parameters can be done directly through the $\tilde{f} : \mathbb{R}^3 \times \Theta \rightarrow \mathbb{R} \times C$ function, without needing to find an explicit expression that links together the positions and the procedural parameters. This is particularly interesting as finding an explicit relation is complex in general, in particular when the scene graph involves operators that affect the output distance values rather than on input position transformation, such as Smooth Booleans, or the *Shell* and *Round* operators. Nevertheless, we can compare for a subset of scenes that do not use such operators our fully implicit approach with one that consists in explicitly building the expression of $\mathcal{E} : (c, \theta) \in C \times \Theta \mapsto \mathbf{p} \in \mathbb{R}^3$. When possible, we derive this function by assembling through an algorithm close in spirit to Algorithm 1 atomic functions `eval_pos` associated to primitives and operators. In such a case, the minimized loss for each patch of manipulated

surface can be expressed as follows:

$$\mathcal{L}_i := \frac{1}{2} \|\Delta \text{Proj}(\mathcal{E}(c_i)) - \Delta T_i\|_2^2 \quad (1)$$

where c_i is the co-parameter evaluated at position p_i at the beginning of the edit.

In Figure 4 we show a direct comparison between our proposed implicit solver and the one based on the explicit function \mathcal{E} . Using the same selected patch of the surface and the same mouse movement, both solvers reach a visually similar outcome with a numerical L_2 distance between the updated procedural parameter sets equal to 0.033. This validates that our solver behaves well on cases where an explicitation, while it is at the same time able to generalize to more complex setups.

6 COMPARISON WITH LIBFIVE

Figure 10 in the main paper illustrates how the results of our method differs from the direct manipulation solver provided with libfive [Keeter 2019]. Figure 5 breaks down on a simplified 2D case the behavior of both solvers. The key difference is that contrary to ours, libfive's solver has no awareness of the identity of the dragged point (i.e., its co-parameter), therefore it may only solve for *some* element of surface passing by the target point whereas we solve for a specific element of surface to end up at the target location. This makes a big difference when manipulating parameters that moves points tangentially to the surface: libfive balances its lack of co-parameter by the extra assumption that the user intends to move points along the normal of the surface, as much as possible. When this assumption is true, libfive's solver behaves well (also because it reuses this hypothesis when unprojecting the mouse into the 3D space). However, it is unable to affect parameters that only relate to tangential movement.

7 A FAILURE CASE

We show here an example of implicit modeling operator that we were not able to fit in our framework, i.e., for which we could not define an augmented $\overline{\text{post}}$ such that the resulting co-parameter would match the intuition of users.

The *Morph* operator consists in blending the signed distances returned by two implicit functions. Formally, it is defined as follows (where mix corresponds to the GLSL function):

$$\begin{aligned} &\textbf{Morphing by a factor } t \in (0, 1) \\ &\text{pre} : p \mapsto (p, p) \\ &\text{post} : (s_1, s_2) \mapsto \text{mix}(s_1, s_2, t) \end{aligned}$$

Figure 6 shows the result of morphing a cube primitive with a sphere (middle) or a torus (right). A naive solution for augmenting this operator consists in having post return a similar mix of the co-parameters $c_1 = (a_1 \text{pid}_1)$ and $c_2 = (a_2 \text{pid}_2)$.

However, this fails at grasping the intent of the user for both components of the co-parameter. Mixing path indices pid_1 and pid_2 leads to an index that is not integer, and that even rounded to the closest integer may not correspond to an existing path in the scene graph. Furthermore, even when these pid are identical, like in our simple example from Figure 6, mixing leaf parameters a_1 and a_2 might have no sense: different primitives may use very different patterns/frames to define their parameterization (we do not assume

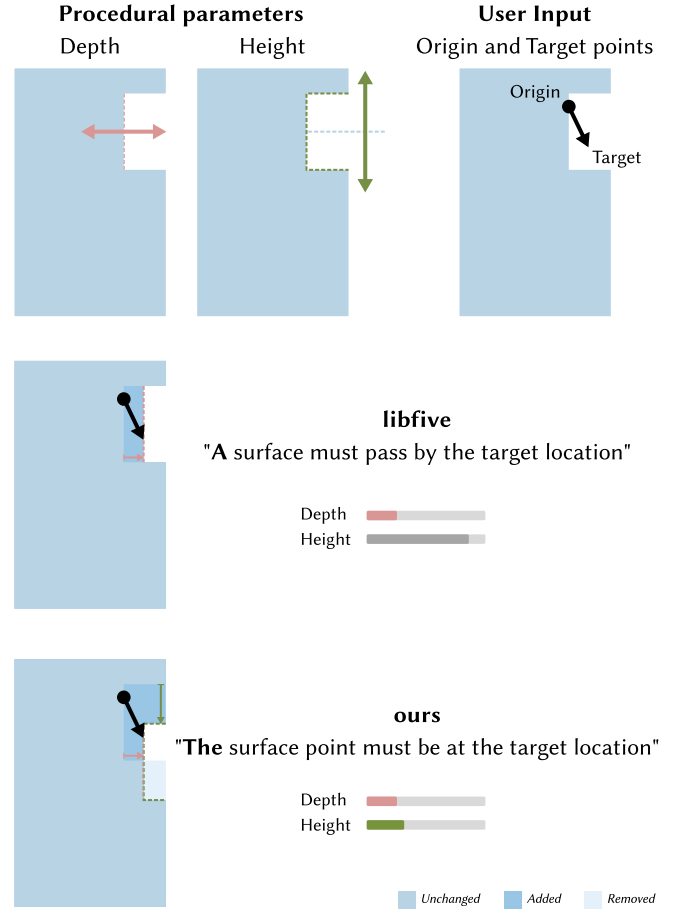


Fig. 5. This figure illustrates the difference of behavior of our approach compared to the one of libfive on a simplified 2D example inspired from Figure 10 bottom row. The first row describes the input, namely a parametric shape with 2 procedural parameters and the user input during a direct edit, namely the point originally clicked on and the target position where we expect it to move. Contrary to 3D examples, the position of the target is fully known (there is no ambiguous unprojection). The second row shows the parameter update found by libfive solver, which ensures that the surface passes by the target point, but ignores the identity of the origin of the gesture. The last row shows the parameter update found by our solver, which also ensures that the surface element located at the target position has the same identity as the origin.

anything about it). Although in our case it works reasonably to mix a cube and a sphere, multiple mixes of a co-parameter coming from a cube and a torus may lead to the same result while coming from different points.

For all these reasons, although our system can try reacting to user input even in this case, this naive choice of $\overline{\text{post}}$ leads to unintuitive manipulation, as illustrated in both Figure 6 and the end of the accompanying video.

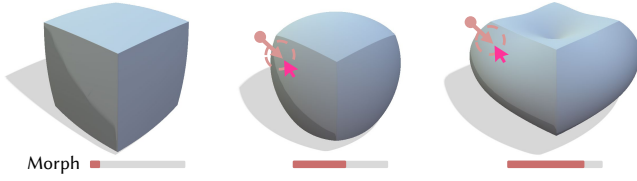


Fig. 6. Morphing of a cube (left) with either a sphere (middle) or a torus (right). Our naive attempt at augmenting the *post* function of the *Morph* operator performs well on a cube-sphere morphing because the cube and the sphere are consistently parameterized, but edition becomes unintuitive in the case of a box-torus morphing because overall our method does not require co-parameterizations to be consistent.

8 COMBINING PATH INDICES IN OPERATOR NODES

Section 5.2 in the main paper introduces the *Augmented Implicit Function* by rewriting the *eval*, *pre* and *post* functions for each node type in the scene graph. We show in this section that the *pid* that results from this augmentation uniquely identifies a path between a primitive and the root output of the scene graph.

We **recall** that in our construction a primitive node always returns $pid = 0$ and an operator forwards the path index from one of its inputs by offsetting it by $1 +$ the maximum path index may flow through previous inputs. For instance, a *pid* transmitted from the first input is left as is, a *pid* transmitted from the second input is offset by $1 +$ the maximum path index that may flow from the first input, etc.

We formally **define** here the *path* of a point \mathbf{p} as a trace of the evaluation of $\tilde{f}(\mathbf{p}, \theta)$. For each call to a *post* function in the evaluation of \tilde{f} as described in Algorithm 1 (in the main paper), the returned co-parameter \mathbf{c} is assumed to be derived from exactly one of the m inputs. The path of \mathbf{p} in \tilde{f} is defined as the sequence (k, \dots) whose head k is the index of this input ($1 < k \leq m$) for the outermost *post*, and whose tail is recursively defined as the path of \mathbf{p} in the k -th input of this *post*. If there is no call to a *post* function in the expression of \tilde{f} (which means that the shape is reduced to a single primitive) then the path is defined as the empty sequence (there is only one path).

We now show by **strong induction over the number $N \geq 1$ of nodes** in the scene graph that with $(s, (a, pid)) = \tilde{f}(\mathbf{p}, \theta)$ and $(s', (a', pid')) = \tilde{f}(\mathbf{p}', \theta)$, we always have $pid = pid' \Leftrightarrow \mathbf{p}$ and \mathbf{p}' have the same path in \tilde{f} .

If the scene graph is made of a single node ($N = 1$), then this node is a primitive and there is a single path (the empty sequence), therefore \mathbf{p} and \mathbf{p}' always have the same path, which proves the left-to-right implication. Moreover, we set by construction the *pid* to 0 for all primitive nodes, which makes the right-to-left implication true as well and thus **proves the base case** of our mathematical induction.

Let us now assume that our statement is true for any number N' of nodes such that $1 \leq N' < N$, and show that it then is true for N as well. Since there are $N > 1$ nodes in \tilde{f} , then the latter has the form $\text{post}(\tilde{g}_1, \dots, \tilde{g}_m)$ where the \tilde{g}_i are sub-expressions. The presence of this *post* allows us to write (k, \dots) (resp. (k', \dots)) the path of \mathbf{p} (resp. \mathbf{p}') in \tilde{f} .

If \mathbf{p} and \mathbf{p}' have the same path in \tilde{f} , then $k = k'$ and path tails are identical as well. Since $\tilde{g}_k = \tilde{g}_{k'}$ has at most $N - 1$, our induction hypothesis ensures that \tilde{g}_k returns the same path index for both \mathbf{p} and \mathbf{p}' . By assumption, the offset that the *post* function adds to the *pid* returned by \tilde{g}_k may only depend on the index $k = k'$, so we can conclude that $pid = pid'$, which **proves the left-to-right implication**.

Reciprocally, we now assume that $pid = pid'$. We note $\max^{(1)}$ the maximum *pid* that may flow from the first input of the outermost *post* function. If $0 \leq pid \leq \max^{(1)}$, we know by construction that *pid* was forwarded from the first input, i.e. $k = 1$. Otherwise, if $1 + \max^{(1)} \leq pid \leq \max^{(2)}$ then $k = 2$, etc. Since $pid = pid'$ and $\max^{(i)}$ does not depend on \mathbf{p} , we have $k' = k$. In other terms, both paths share the same head, so $\tilde{g}_k = \tilde{g}_{k'}$. Furthermore, the path index that is forwarded from \tilde{g}_k is the same for both \mathbf{p} and \mathbf{p}' . Since \tilde{g}_k has at most $N - 1$, our induction hypothesis applies and ensures that the \mathbf{p} and \mathbf{p}' have the very same tail. This **proves the right-to-left implication**, and thus conclude our mathematical induction.

REFERENCES

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