Eurographics Symposium on Geometry Processing 2017
J. A. Bærentzen and K. Hildebrandt
(Guest Editors)

Volume 36 (2017), Number 5

Generalized Matryoshka: Computational Design of Nesting Objects

Alec Jacobson, University of Toronto, Canada

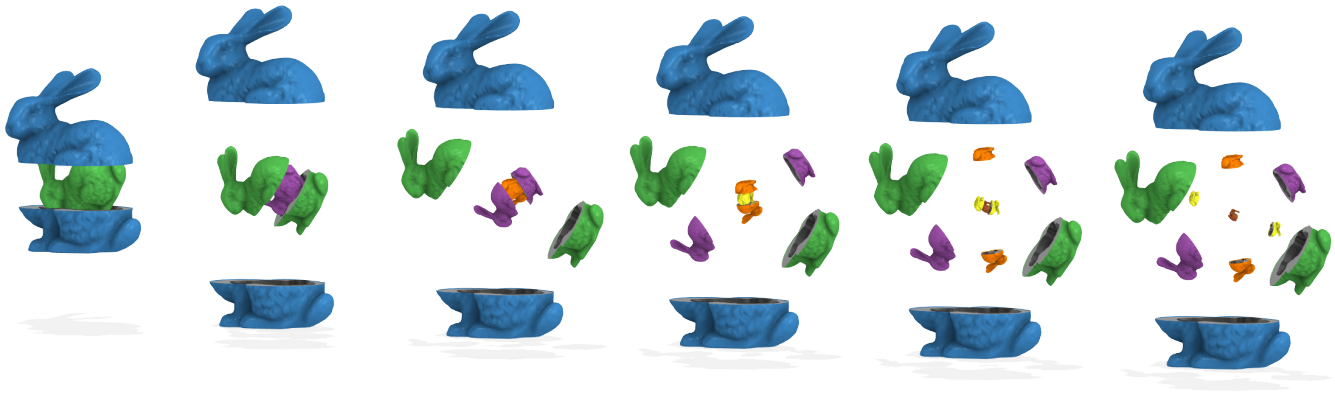


Figure 1: *We generalize the classic* Matryoshka *dolls (also known as Russian nesting dolls) to generic 3D shapes.*

Abstract

This paper generalizes the self-similar nesting of Matryoshka *dolls ("Russian nesting dolls") to arbitrary solid objects. We introduce the problem of finding the largest scale replica of an object that* nests *inside itself. Not only should the nesting object fit inside the larger copy without interpenetration, but also it should be possible to cut the larger copy in two and remove the smaller object without collisions. We present a GPU-accelerated evaluation of nesting feasibility. This test can be conducted at interactive rates, providing feedback during manual design. Further, we may optimize for some or all of the nesting degrees of freedom (e.g., rigid motion of smaller object, cut orientation) to maximize the smaller object's scale while maintaining a feasible nesting. Our formulation and tools robustly handle imperfect geometric representations and generalize to the nesting of dissimilar objects in one another. We explore a variety of applications to aesthetic and functional shape design.*

1. Introduction

Human fascination and satisfaction with snugly fitting geometric shapes has as much to do with pragmatism (e.g., packing belongings efficiently in a storage container) as it does with aesthetics (e.g., calligram poetry). It is no wonder that Russian nesting dolls, *Matryoshka*, quickly became a universally recognized toy since their first incarnation little over a century ago (see Figure 2). Philosophically, the *object-within-itself metaphor* resonates with a broad set of design principles from architecture to software engineering.

In this paper, we consider the geometric problem of placing an object within itself. Our primary motivation is to generalize *Matryoshka* dolls to arbitrary shapes (see Figure 1). We focus on providing a computational design tool to help a user find feasible nesting configurations and maximize the relative scale of the inner object.

Using traditional computational design tools, it is tedious to determine if one object lies entirely inside another. A user must iterate between camera changes to inspect feasibility and perform geometric edits. Designing nesting objects demands *even more* effort since nesting also requires that the inner object can be physically removed without collision when the containing object is cut in half. Further adding to complexity, optimal nesting placements and cut directions may be at unintuitive orientations for non-convex geometries (see Figure 3). Resorting to offline collision detection or, worse, verification via physical fabrication would fragment the design process.

We pose the problem of optimal (self-)nesting as finding the largest scale of an object so that it fits entirely inside a containing object *such that* the containing object can be cut in two and removed without collisions. While solving this problem exactly is intractable, we present various levels of optimization tools to assist a designer. Alternatively, we can operate as a fully automatic generalized *Matryoshka* generator. Specifically, we present highly parallelizable methods to:

- determine whether a given design is a feasible nesting;
- find the maximum scale of the inner object given its placement and cut plane through the containing object; and
- optimize the nesting scale over some or all design parameters.

We demonstrate iterative design scenarios where these methods quickly find a large nesting scale while (optionally) satisfying cus-

Figure 2: *Zvyozdochkin & Malyutin's original* Matryoshka *dolls have a simple, nearly convex shape and packing ratios are roughly 70% to 80%. Derivative designs largely adhere to this formula.*

tom aesthetic or functional constraints from the user (e.g., avoiding cuts through salient regions). Our suite of tools for *self*-nesting extends trivially to nesting one object into a *different* object. As a means of visualization and validation, we fabricate some of our results using commodity 3D printing.

2. Related Work

While to our knowledge no previous work has directly considered the problem of designing and optimizing nesting objects[†], our problem and methodology are similar to previous methods for constructing bounding primitives, designing densely packing shapes and detecting collisions.

Bounding primitives. Constructing hierarchical bounding shapes is a fundamental problem in computer graphics. Traditional methods focus on canonical bounding polytopes such as spheres, boxes, KDOPS, and convex hulls. Bounding polyhedra can be constructed via morphological operations [CB14], and theoretical optimality results exist for very specific shapes (e.g., [Sch11]). However, *bounding* does not ensure rigid, collision-free removal.

Recently, Sacht et al. presented a flow-based method to progressively fit a triangle mesh within a simplification thereof [SVJ15]. These "nested cages" are tightly fitting, but solve a complementary problem: they continuously deform the outer object to fit the inner object as tightly as possible. The desired nested cage is therefore very similar in placement to the original shape, making derivative-based local optimization effective. Instead, our problem treats both objects as rigid up to scale. The optimal result may require a radically different placement of the inner object with respect to the containing object (see Figure 3), implying the need for a global search.

[†] We encountered one example of 3D-printed *Matryoshka* with non-traditional shapes (http://www.thingiverse.com/thing:1032093). This manually designed nesting appears to simply fit a bounding sphere in each shape.

Computational design and fabrication. Improved rapid fabrication processses have inspired a wealth of methods for the computational design of interesting objects. While many recent methods share our enthusiasm for computational design of toys [STC*13, TCG*14, BWBSH14, UKSI14, BCT15], most methods do not directly optimize over collision/interpenetration constraints. Similar in spirit to our generalization of *Matryoshka*, Sun & Zheng generalize Rubik's cubes to arbitrary shapes [SZ15]. They manage collisions by a group theoretical reduction to 2D that is not immediately applicable to our scenario.

Perhaps most similar to our problem are those of stackabilization [LAZ*12], Escherization [KS00] and boxelization [ZSMS14]. These methods take arbitrary shapes as input and alter their geometry to conform to the goal of self-stacking in one-direction, tessellating the plane or transforming into a box (resp.). In some sense, our problem is more basic: the relative geometries are fixed. This has a dramatic effect on the types of collision detection and global optimization methods at our disposal.

Improving the rapid fabrication process itself has also led to geometric decomposition and packing methods to meet 3D printer constraints [LBRM12, Att15, CZL*15, HMA15]. The packing methods assume a rectangular build volume and employ bounding primitives to reduce complexity. For example, Vanek et al. pack non-convex objects using a discrete set of rotations by sequentially growing a height field [VGB*14]. Our method nests arbitrary shapes at arbitrary angles and operates directly on the input geometry.

Malomo et al. solve a related problem to construct flexible, cut-away molds for casting-based fabrication of 3D shapes [MPBC16].

While the computational tools for reconfigurables [GJG16] are not a feasible replacement for our fast search, we borrow concepts from this and other (e.g., [UKIG11]) optimization-assisted design tools.

Swept volumes and collision detection. Our methodology for quickly determining nesting feasibility is heavily influenced by "depth peeling"-type methods that have been used for GPU-acceleration of constructive solid geometry visualization [GHF86, KGP*94, HR05], order-independent transparency [Eve01, BCL*07], CNC milling simulation [IO07], and shape diameter estimation [BKR*16]. Similarly, layered depth images [SGHS98] (originally presented for image based rendering) have been leveraged for intersection volume computation [FBAF08], collision detection [MOK95, HTG03, KP03] and swept volumes for minimal

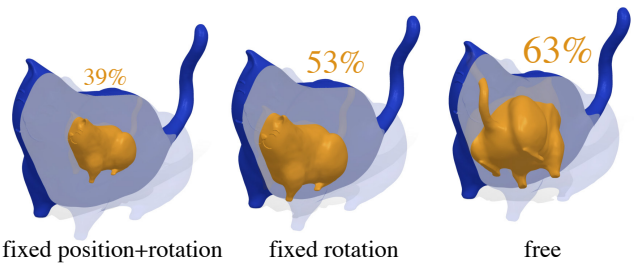


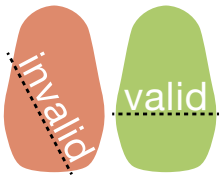
Figure 3: *Left to right: Our optimization leverages positional and rotational degrees of freedom to tightly self-nest* The Fat Cat.

interference-removing translations [KOLM02]. Our method provides the additional application of this approach to early-exiting collision-free translational trajectory verification. Detecting whether an object can be removed intuitively seems *harder* than static collision detection, but leveraging depth-peeling nesting verification often requires fewer tests than static collision detection due to early failures of more strict conditions.

Alternative non-depth-peeling-based methods also leverage the GPU for collision detection, for example by rendering signed distance fields [SGGM06] or via high-throughput bounding-volume hierarchies [LMM10].

3. Nesting

Unlike traditional bounding volumes [SVJ15], we define *valid self-nesting* to occur when a solid shape $\mathcal{A}$ transformed by a non-reflecting similarity transform $\mathbf{T} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ lies strictly inside itself ($\mathbf{T}(\mathcal{A}) \subset \mathcal{A}$) *and* the parts of $\mathcal{A}$ above and below a cut plane $\mathbf{P}$ can be removed by translating linearly along some vectors $\mathbf{a}^+ \in \mathbb{R}^3$ and $\mathbf{a}^- \in \mathbb{R}^3$ (resp.) without colliding into the transformed shape. Let us explore this definition and become familiar with combinations of shapes, transformations and cut planes that admit valid self-nestings.



Perfect self-nesting. We are most interested in the case when the uniform scaling s induced by the transformation $\mathbf{T}$ is as large as possible. It is tempting to claim that for a *convex* shape $\mathcal{A}$ that any transformation $\mathbf{T}$ will produce a valid self-nesting even if the scaling is nearly 100%. While its true that a convex shape scaled by any amount will fit inside itself, it is not true that *any* cut plane $\mathbf{P}$ will allow safe *removal* (see inset). Instead, valid self-nesting depends on the choice of cut plane. We can categorize the set of shapes that admit infinitesimal shrinkages while self-nesting with respect to a chosen cut plane in a manner analgous to the definition of "star-shaped polygons" (see, e.g., [PS85]). An infinitesimal shrinkage of a shape $\mathcal{A}$ admits a valid self-nesting with respect to a cut plane $\mathbf{P}$ if and only if there exists vectors $\mathbf{a}^+$ such that every point on the shape's boundary $\partial\mathcal{A}$ above the plane $\mathbf{P}$ has a clear line of sight along $\mathbf{a}^+$ intersecting $\mathbf{P}$ (and analogously for $\mathbf{a}^-$). In either direction the shape $\mathcal{A}$ must be a (skewed) function graph above the cut plane. This is a severely limited class of shapes.

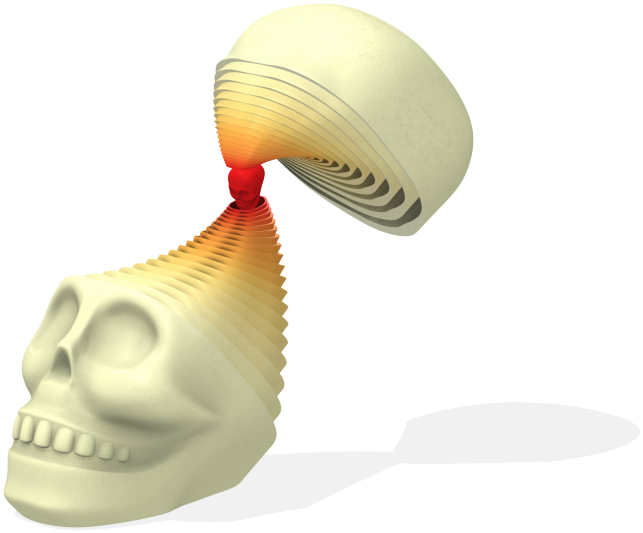


Figure 5: The Calavera *achieves a high self-nesting scale of 91% allowing this 20-layer generalized* Matryoshka.

Traditional hand-carved Matryoshka dolls have a distinctive, simple shape. They are convex or nearly convex, almost always split horizontally (perpendicular to gravity), and typically scale by 70-80% between layers (see Figure 2). This *imperfect* nesting scale effectively allows the dolls to have a finite thickness (traditional wooden dolls are a few millimeters thick), but more importantly allows at least *nominal* deviation from the class of perfect nesting shapes (most doll heads bulge slightly above the cut plane).

The goal of this paper is to help a user explore nestings of *arbitrary* solid 3D shapes. Since the scale of the ith level with respect to the original shape will increase exponentially, we strive to help the user find nesting scales comparable to traditional Matryoshka dolls but for much more irregular geometries (see Figure 5).

4. User Interface

In our tool, the user loads the triangle mesh boundary of a solid shape $\mathcal{A}$; positions, rotates and scales a second instance $\mathcal{B}$; and chooses a cut plane $\mathbf{P}$. While the user explores the design, our

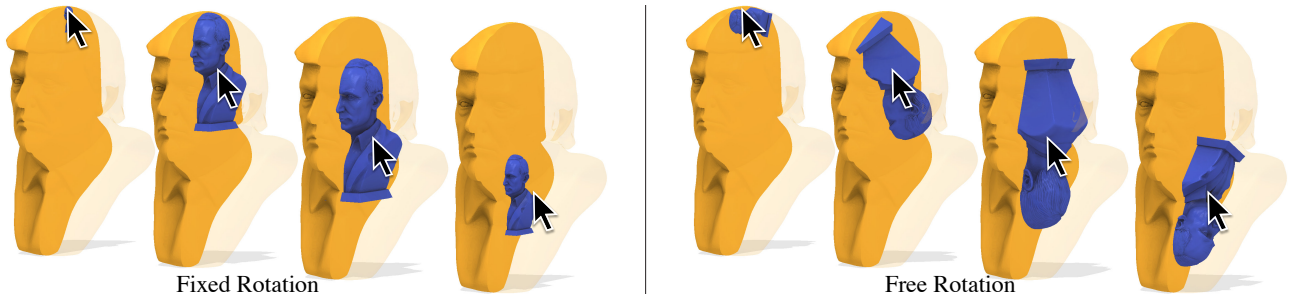


Figure 4: *A user finds nestings for one bust inside another. Our tool optimizes the maximum feasible nesting scale for fixed position and fixed rotation in real-time (∼ 70 fps). The user can also free the rotation resulting in higher nesting scale at the cost of performance (∼ 1 fps).*

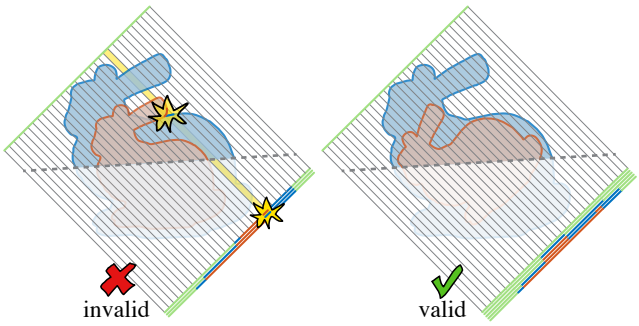


Figure 6: *Outer (blue) and inner (orange) shapes are clipped by the cut plane (dashed) and rendered orthographically along the removal direction (parallel lines). As fragment layers are* peeled *according to depth, we inspect consecutive layers for "bad codes" (yellow). Successfully peeling a pure background layer implies validity.*

program tracks whether the current configuration admits a feasible nesting in real-time (see Section 5). This unconstrained interaction facilitates creativity, yet even with real-time feedback it is often difficult to find a feasible solution.

Similar to the "auto-resolve" of [GJG16], the can user can also directly explore the space of valid self-nestings. While the user rotates and translates the inner shape $\mathcal{B}$ or cut plane, our optimization interactively computes the maximal feasible scale (see Figure 4).

The user can choose which parameters are free during the optimization. If only the scale is free, then we use a fast local search operation (see Section 6). If other parameters are also *free*, we engage a stochastic global optimization (see Section 7). If all are free, the user receives the best possible solution (e.g., as an initial design). This case also implies a fully automatic *Matryoshka* generator tool.

5. Feasibility analysis

The heart of our approach is a fast method for determining if a given configuration is a valid nesting. For this sub-routine we take as input triangle meshes of the boundaries of two solid shapes $\mathcal{A}$ and $\mathcal{B}$ (where $\mathcal{B}$ may be identical to $\mathcal{A}$), a similarity transformation $\mathbf{T} \in \mathbb{R}^{3\times4}$ applied to $\mathcal{B}$, a plane $\mathbf{P}$ given by a point on the plane $\mathbf{p} \in \mathbb{R}^3$ and a normal vector $\mathbf{n} \in \mathbb{R}^3$, and two translational *removal* trajectory vectors $\mathbf{a}^+, \mathbf{a}^- \in \mathbb{R}^3$.

Determining if a configuration is feasible requires answering two questions: Is $\mathbf{T}(\mathcal{B})$ inside of $\mathcal{A}$? Can the two "halves" of $\mathcal{A}$ above and below the plane $\mathbf{P}$ translate along $\mathbf{a}^+$ and $\mathbf{a}^-$ without hitting $\mathcal{B}$?

Drawing from the "free line of sight" intuition of Section 3, we cast these as orthographic visibility queries. Leveraging the direct, real-time rendering pipeline, we construct an orthographic projection so that bounding box of $\mathcal{A} \oplus \mathcal{B}$ fits tightly in the viewing prism. For each removal direction $\mathbf{a} \in \{\mathbf{a}^+, \mathbf{a}^-\}$, we render the meshes of $\mathcal{A}$ and $\mathbf{T}(\mathcal{B})$ *clipped* by the plane $\mathbf{P}$, signed correspondingly to $\mathbf{a}$. If we knew the sorted depth order and identification of all fragments landing at each pixel then we would know we have an *invalid* configuration if:

1. any fragment of $\mathcal{A}$ appears (directly) before a fragment of $\mathbf{T}(\mathcal{B})$,
2. any fragment of $\mathbf{T}(\mathcal{B})$ appears directly before the background, or
3. any fragment of $\mathbf{T}(\mathcal{B})$ appears directly before a front-facing fragment of $\mathcal{A}$.

The first test catches whether any part of $\mathcal{A}$ would collide with $\mathcal{B}$ as $\mathcal{A}$ travels along the removal direction (away from the camera). The second and third tests catch whether part of $\mathbf{T}(\mathcal{B})$ protrudes out of $\mathcal{A}$. Such a protrusion usually may not trigger the first test in the case that $\mathcal{A}$ is extremely foreshortened near the problem area (i.e., surface normals of $\mathcal{A}$ perpendicular to the view direction).

In practice, we do not gather and sort all fragments landing on each pixel. Instead (inspired by techniques such as "depth peeling" for order-independent transparency, see, e.g., [GHF86, Eve01]), we iteratively *peel* the nearest fragment for each pixel (see Figure 6).

For iteration i we store for each pixel (x,y): the "color" vector $\mathbf{c}^i(x,y)$ and depth scalar $z^i(x,y)$ of the nearest fragment whose depth is strictly greater than the previous iteration's $z^i(x,y) > z^{i-1}(x,y)$. We abuse the color channels in $\mathbf{c}^i$ to store whether the fragment is from $\mathcal{A}$ or $\mathcal{B}$ and whether it is front or back facing.

By our assumption that $\mathcal{A}$ and $\mathcal{B}$ are boundaries of solid shapes (i.e., without gaping open boundaries), the tests above reduce to simply binary logical tests between consecutive iterations i and $i-1$. We can save memory on the GPU, by "ping-ponging" between two sets of color and depth framebuffer-textures (for odd and even iterations).

An occlusion query (`GL_ANY_SAMPLES_PASSED`) is used to determine if enough layers have been peeled: if no fragments are drawn this can be the last iteration. We still need to check feasibility on this layer because a new background pixel on this layer may reveal that part of $\mathcal{B}$ is outside of $\mathcal{A}$.

Per-pixel feasibility tests are conducted by rendering a full-"screen" rectangle, reading from the color and depth buffer textures from the current and previous iteration. Depth buffers are used to reveal whether the pixel belongs to a foreground object ($\mathcal{A}$ or $\mathcal{B}$) or the background ($\varnothing$). We discard all *good* pixels, and render an arbitrary value for bad pixels so that an additional occlusion query (again `GL_ANY_SAMPLES_PASSED`) will trigger.

After checking per-pixel feasibility, if any bad pixels were found we may report invalidity immediately. Otherwise, if no new fragments were peeled on this iteration we may safely report validity. Otherwise, we must continue peeling another layer.

6. Scale search

We now consider the scenario wherein the cut plane and removal directions are fixed along with all rigid degrees of freedom. Maximizing the scale s so that $\mathcal{B}$ *nests* in $\mathcal{A}$ amounts to a feasibility line search. At first glance, this appears very difficult. Feasibility involves detecting collisions and interpenetration, making it non-linear and non-convex with respect to the unknown scale s. In lieu of other information, one might be inclined to opt for a general-purpose global—perhaps stochastic—search method.

Experimentally we have found that a simple geometric convexification of this constraint performs very well. Assume momentarily that both $\mathcal{B}$ and $\mathcal{A}$ are convex. Then feasibility as a function of s is *monotonic*: $s = 0$ trivially produces a valid nesting, and for some

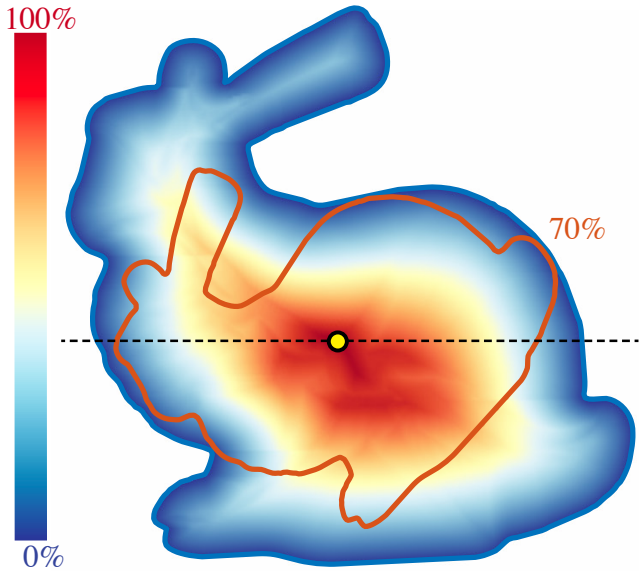


Figure 7: *Nesting scale maximization is a non-linear, non-convex problem. Fixing horizontal cut plane to the inner 2D bunny's centroid, we visualize a pseudo-coloring of the maximum feasible nesting scale over 256 rotations for a grid of* 256^2 *centroid locations.*

$s = s^* > 0$ a collision (either in the rest state or during removal) occurs that further growing will never resolve. Via binary search, we can shrink lower (feasible) and upper (infeasible) bounds around s^* to arbitrary precision with logarithmic computational complexity. For non-convex shapes, conducting this binary search is conservative, but in practice usually optimal.

When manually exploring, the user can choose to have the nesting scale parameter automatically optimized to its largest feasible value while changing the other design parameters. The logarithmic complexity of the search ensures consistent and interactive performance.

7. Global optimization

Let us consider the case that some or all other parameters are free to be optimized. Given a shape $\mathcal{A}$ and another shape $\mathcal{B}$ we seek to find a non-reflective similarity transformation $\mathbf{T}$, cut plane $\mathbf{P}$ and removal directions $\mathbf{a}^+, \mathbf{a}^-$ that maximize the scale s of $\mathcal{B}$ while maintaining a valid nesting (according to the definition in Section 3):

$$\begin{aligned} &\underset{s,\mathbf{R},\mathbf{c},\mathbf{P},\mathbf{a}^+,\mathbf{a}^-}{\text{maximize}} \quad s \\ &\text{such that} \quad \mathbf{T}(\mathcal{B}) \text{ nests in } \mathcal{A} \text{ w.r.t. } \mathbf{P}, \mathbf{a}^+, \mathbf{a}^-, \end{aligned} \tag{1}$$

where we break the transformation $\mathbf{T} \in \mathbb{R}^{3\times 4}$ into a scale $s \in \mathbb{R}$ and rotation $\mathbf{R} \in SO(3)$ about displaced centroid $\mathbf{c} \in \mathbb{R}^2$ of $\mathcal{B}$.

While the objective function is linear, the constraint is non-linear and non-convex (see Figure 7). It is neither easy nor efficient to measure violation of the constraint in a way that admits differentiation. Because of the removal process, the nesting constraint is strictly more complex than the interpenetration constraints considered in physically based simulation (see, e.g., [BWK03, FBAF08]). Due to the cut plane, the constraint is even more complex than the space-time untangling problem for reconfigurables [GJG16].

Fortunately, we have an extremely fast evaluation for constraint feasibility. This opens the door to stochastic global optimization methods. We experimented with genetic algorithms, simulated annealing, and other Monte Carlo methods, but ultimately found best performance using the particle swarm method [KE95].

7.1. Particle Swarm Optimization

Given an arbitrary objective $g(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ to *maximize*, the basic particle swarm optimization proceeds by assigning k random initial positions (parameter values) $\mathbf{x}_i \in \mathbb{R}^n$ and velocities $\mathbf{v}_i \in \mathbb{R}^n$. At a fixed pace, particles update their positions by stepping along their respective velocities and update their velocities via momentum term and attraction forces toward the current global and local maxima:

$$\mathbf{v}_i \leftarrow \omega \mathbf{v}_i + \varphi_p r_p (\mathbf{x}_i^p - \mathbf{x}_i) + \varphi_g r_g (\mathbf{x}^g - \mathbf{x}_i), \tag{2}$$

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i, \tag{3}$$

where $\omega \in [0,1)$ scales the momentum term, $\varphi_p, \varphi_g \in [0,1)$ scale forces pulling the particle toward its personal best position ever witnessed $\mathbf{x}_i^p \in \mathbb{R}^n$ and current global best over all particles $\mathbf{x}^g \in \mathbb{R}^n$, and r_p, r_g are random variables drawn uniformly from $[0,1)$.

For our problem, we define $\mathbf{x} = [s, \mathbf{R}, \mathbf{c}, \mathbf{P}, \mathbf{a}^+, \mathbf{a}^-]$ and introduce feasibility as a hard constraint:

$$g(\mathbf{x}) = \begin{cases} s & \text{if nests,} \\ \infty & \text{otherwise.} \end{cases} \tag{4}$$

Finally, after each update we project each component onto its respective constraint set: $\mathbf{s} \in [0,1)$, $\mathbf{R} \in SO(3)$, $\mathbf{c} \in$ bounding box$(\mathcal{A})$, $\mathbf{P} \cdot \mathbf{a}^+ > 0$, $\mathbf{P} \cdot \mathbf{a}^- < 0$.

In our interface (see Section 4), a user can fix partial subsets of the degrees of freedom interactively at runtime (see Figure 8).

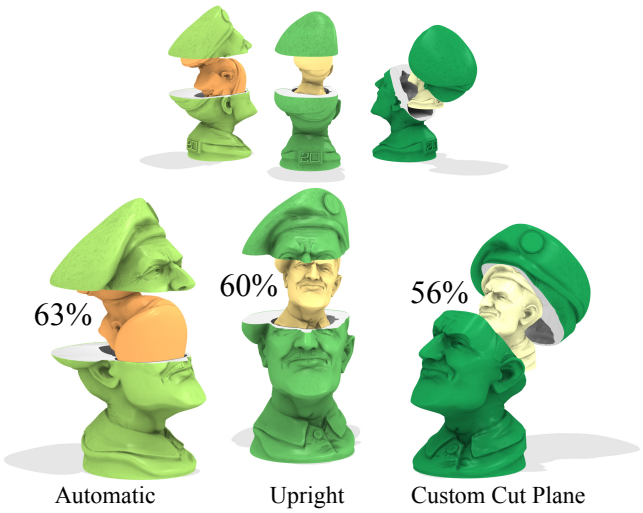


Figure 8: *Left to right: Starting with a fully optimized self-nesting of* The Colonel*, the user can add constraints forcing an upright orientation of the inner object or fixing the cut to a desired plane.*

Figure 9: *Via global optimization,* Nefertiti *self-nests recursively.*

7.2. Accelerations

This basic algorithm performs reasonably efficiently compared to truly brute force approaches such as grid-search or pure random search. However, we observed that this method spends many iterations sampling small scale values deep inside $\mathcal{A}$ that are clearly suboptimal. We now introduce unique accelerations for our problem.

Local scale search. If a particle's position $\mathbf{x}$ has a small scale component s, we can *equivalently* cast the optimization problem in Equation (1) as a maximization of the scale parameter with respect to all other degrees of freedom:

$$\underset{\mathbf{R},\mathbf{c},\mathbf{P},\mathbf{a}^+,\mathbf{a}^-}{\text{maximize}} \quad f(\mathbf{R},\mathbf{c},\mathbf{P},\mathbf{a}^+,\mathbf{a}^-) \tag{5}$$

where

$$f(\mathbf{R},\mathbf{c},\mathbf{P},\mathbf{a}^+,\mathbf{a}^-) = \underset{s}{\text{maximize}} \quad s \tag{6}$$

$$\text{such that} \quad \mathbf{T}(\mathcal{B}) \text{ nests in } \mathcal{A} \text{ w.r.t. } \mathbf{P},\mathbf{a}^+,\mathbf{a}^-, \tag{7}$$

Written in this way we can leverage the binary search approximation of the previous section for maximizing the nesting scale given all other degrees of freedom:

$$f \approx \underset{s}{\text{search}}(\mathbf{R},\mathbf{c},\mathbf{P},\mathbf{a}^+,\mathbf{a}^-). \tag{8}$$

Making this substitution in our particle swarm optimization we remove the scale component from the position vector $\mathbf{x}$ (decrementing the degrees of freedom) and replace the objective function with a search for the maximal scale given all other parameters:

$$g(\mathbf{x}) = \begin{cases} \text{search}_s(\mathbf{R},\mathbf{c},\mathbf{P},\mathbf{a}^+,\mathbf{a}^-) & \text{if nests,} \\ \infty & \text{otherwise.} \end{cases} \tag{9}$$

This places our optimization in a class of hybrid particle swarm optimization methods. In general, hybridizations perform a local optimization (e.g., Newton's method) to race particles upward in their current basin of attraction. Viewed from the original formulation in Equation (1), our hybridization is a (convexified) local optimization along a single coordinate (cf., [WP09]).

Early search abortion. The binary search quickly narrows in on the optimal scale values, given all other degrees of freedom. While logarithmic in performance, the cost to squeeze the bound within 0.1% of the solution is *at best* 10 feasibility tests ($({}^1/_2)^{10} \approx 0.1\%$). During the search iterations, if the upper bound is already less than the particle's best witnessed so far, then the exact value of this search will have no effect on this or any particles' update. Therefore, the binary search can safely be aborted prematurely. This additional hybridization bears some resemblance to branch and bound algorithms found in discrete optimization [LD60].

8. Experiments and results

We implemented our method in C++ using OpenGL and GLSL shaders. We tested our implementation on an Intel Xeon 3.5GHz CPU with 64GB of RAM and an NVidia GeForce GTX 1080 GPU. After initial experimentation, we fixed the optimization parameters to $k = 200, \omega = 0.98, \varphi_p = 0.01, \varphi_g = 0.01$. During scale search, we stop when the bound extent is less than 0.01% and return the lower (feasible) bound. We run 500 update iterations during the particle swarm optimization; improvements were occasionally witnessed past this point, but returns were usually marginal. For all examples in the paper, we use 512×512 viewports to render along $\mathbf{a}^+$ and $\mathbf{a}^-$ simultaneously during depth peeling (i.e., into a 1024×512 buffer). Increasing beyond this resolution should improve accuracy, but in practice had no effect for our examples.

Most non-linear, non-convex global optimizations cannot guarantee finding a global optimum: ours is no exception. Compared to

Models	$\lvert\mathcal{A}\rvert + \lvert\mathcal{B}\rvert$	Feas.	Search	Opt.	s^*
Falcon+Key	26K	0.4ms	10ms	18s	90%
Calavera+self	53K	0.6ms	7ms	22s	90%
Orange+blue bust	79K	1.0ms	14ms	27s	68%
Bunny+self	140K	1.7ms	32ms	135s	61%
Colonel+self	145K	1.6ms	52ms	123s	63%
Fat Cat+self	150K	1.7ms	33ms	122s	63%

Table 1: *We report timings and optimized scales (assuming all parameters free) for some of the models in this paper.*

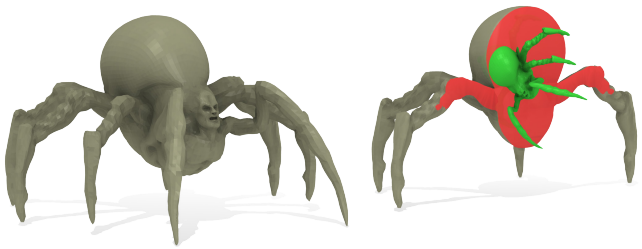


Figure 10: *The* Spider Man *has a very non-convex shape, but finds a suitable nesting via an intuitive rotation.*

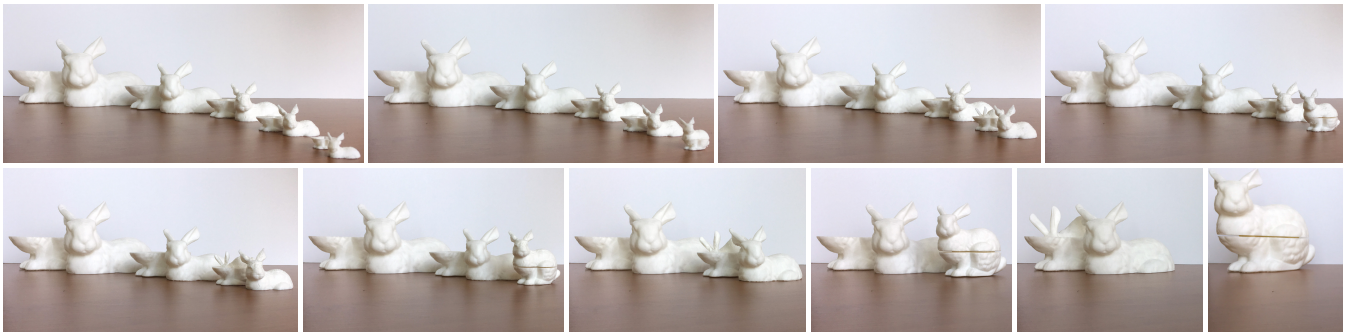


Figure 11: *3D printed bunnies nest recursively inside a 16cm tall bunny.*

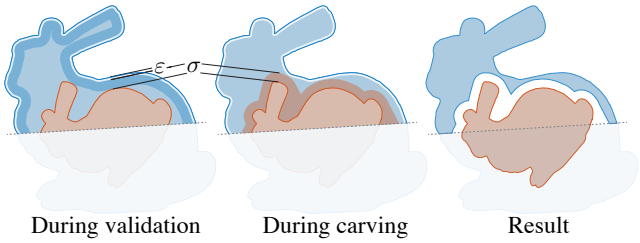


Figure 12: *Our model accommodates engineering tolerances. Left to right: We conservatively validate nesting up to printer accuracy (σ) plus minimum wall thickness (ε). When carving, we inflate the swept volume by σ, resulting in physically realizable nesting.*

brute force, our accelerated particle swarm optimization converges to a satisfying solution much faster. For example, the brute-force grid search to visualize the energy landscape for the simplified 2D problem in Figure 7 took eight days to compute.

For all meshes in this paper (25,000 to 150,000 triangles), testing nesting feasibility runs at over 500 frames per second (fps). Fixing all other parameters, finding the maximum scale via binary search to a tolerance of 0.1% runs interactively at around 30 fps (see Table 1). Freeing all parameters, our global optimization inherits this high performance: typically the particle swarm optimization can conduct its 100,000 searches in 30 to 300 seconds. The variance in times is due to the success of the early search abortion acceleration. In particular, for near-perfect nesting shapes optimization finishes very quickly: the optimal nesting of the *Calaveras* in Figure 5 is found in less than half a minute. The *Bunny* in Figure 1 is a more complex shape and correspondingly takes longer to find a fully automatic, optimal nesting. The number of parameters freed to the optimization amounts to a trade-off between interactivity (both in terms of control and performance) and optimality. In Figure 8, the fully automatic solution initially requires nearly two minutes of search time. Adding the upright constraint reduces the degrees of freedom and correspondingly the search time to under a minute. Finally, constraining the cut plane reduces the search to 20 seconds.

Figure 9 shows a self-nesting of the *Nefertiti* bust, where the cut plane has been placed behind the face. The total design time was less than five minutes.

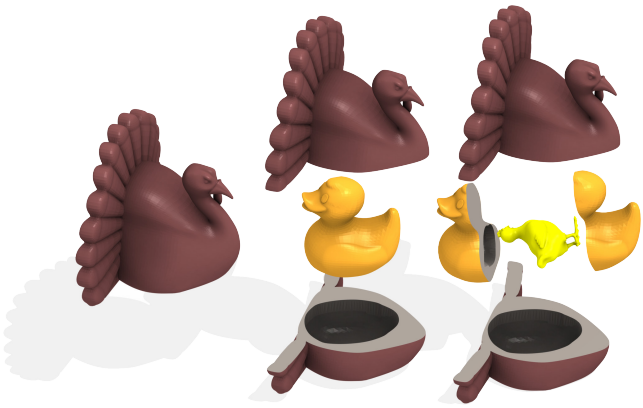


Figure 13: *Our self-nesting tools generalize to nesting one shape inside a different shape inside yet another different shape.*

Robustness. The input to our method is a *solid* triangle mesh (def. [ZGZJ16]). However, our fragment shader based approach is far less sensitive to poor meshing and surface artifacts than traditional collision detection (e.g., [Lin93]) or boolean operations (e.g., [ZGZJ16]). In general, as long as the model is consistently oriented and free of spurious internal faces our method will work well without the strict requirement on mesh quality [CGA17] or winding numbers [JKSH13,ZGZJ16]. *The Spider Man* in Figure 10 has small defects (open boundaries, non-manifold edges, self-intersections), these are invisible during rendering and invisible to our method.

Fabrication. When the end goal is to 3D print the resulting nesting shapes, we take into account the printer accuracy σ and minimal wall thickness/filament size ε, by replacing $\mathcal{A}$ with the inward offset surface at negative signed distance $-(\sigma+\varepsilon)$ during validity checking and optimization. When carving the swept-volume of $\mathcal{B}$ relative to the removal trajectories from either half of A, we replace $\mathcal{B}$ with the outward offset surface at positive signed distance σ (see Figure 12). For our 3D printed results, we used a MAKERBOT REPLICATOR Z18 and empirically determined these parameter values ($\sigma = 0.7\text{mm}, \varepsilon = 0.5\text{mm}$).

In our results, we *fill* the space between the inner and outer objects. Besides creating a satisfyingly tight fit, this sometimes helps find

Figure 14: *3D printed* Calaveras *nest recursively. Each orientation and cut plane is determine automatically giving slight variations between layers. Our methods accommodate for printer tolerances, so nesting scale is determined* at scale. *Compared to the* virtual result *in Figure 5, an interesting polarity shift happens here: for smaller sizes, it is more efficient to nest upside down.*

Figure 15: *A "Glass Key" nests along an optimal removal direction within a "Maltese Falcon."*

unintuitive rotations when nesting. It would also be possible to print thin, loose shells at least as thin as the minimal wall thickness ε.

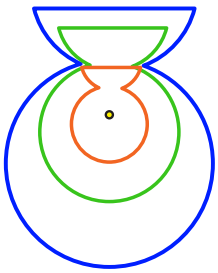
Besides a motivating application, fabrication acts as third party verification of our collision handling (see Figure 11). We use CGAL [CGA17] and LIBIGL [JP*13] to generate signed distance offset surfaces and use the robust boolean operations of [ZGZJ16] in LIBIGL to cut the outer layer in half and carve out the interior. For fabrication, boolean operations are the computational bottleneck (taking roughly four times as long as our optimization in the fully automatic case), and, of course, fabrication is the real bottleneck. The largest bunny in Figure 11 took 31 hours on our MAKERBOT.

Disparate nesting. Nothing in our formulation of the nesting problem or our proposed tools requires that the inner object $\mathcal{B}$ is the same as the outer object $\mathcal{A}$. Our method trivially generalizes to nesting disparate shapes (see Figure 4). In Figure 13, a chicken nests inside a duck inside of a turkey, at 50% and 77% respectively. Total design time is under two minutes, not including finding models online.

The fabrication process above already takes advantage of disparate nesting. This generalization is crucial for fabricating multi-layer *self*-nesting shapes since scales decrease exponentially but printer tolerances remain constant (see Figure 14). For optimal fabricated results, each nesting must be computed independently.

In Figure 15, a high-genus key is nested inside of a falcon statue. Fabrication tolerances are incorporated during optimization so that the result may be 3D printed.

9. Limitations and Future Work



Most non-convex global optimizations suffer from local minima and lack formal guarantees; ours is no exception. For non-convex shapes, feasibility monotonicity is not guaranteed: configurations exist where scaling a non-convex shape creates a collision that is resolved after further scaling (see inset). These situations seem to be rare in practice.

In future work we would like to improve the performance of our optimization by further exploiting the parallelism of the particle swarm optimization. We model removal strictly as translation, however, for rigid objects we could potentially achieve even larger nesting ratios if we also consider rotations. This greatly complicates evaluation of feasibility and is an exciting direction for future work. We also assume that the input shapes are strictly rigid and delegate shape editing operations to the user. Rigid nesting is limited by the available volume within a shape: high genus or thin non-convex shapes will not nest well (see inset). We would also like to consider deformations *during* the optimization (à la freeform modeling) and the nesting of deformables made of soft materials.

Acknowledgements

This work is funded in part by NSERC Discovery Grants (RGPIN-2017-05235 & RGPAS-2017-507938), the Connaught Fund (NR-2016-17), and a gift by Adobe Systems Inc. Thank you to David Levin for illuminating discussions and Kevin Gibson, Masha Shugrina, Michael Tao, and Alex Tessier for early draft reviews.

References

[Att15] ATTENE M.: Shapes in a box: disassembling 3d objects for efficient packing and fabrication. In *Comput. Graph. Forum* (2015). 2

[BCL*07] BAVOIL L., CALLAHAN S. P., LEFOHN A., COMBA J. A. L. D., SILVA C. T.: Multi-fragment effects on the gpu using the k-buffer. In *Proc. I3D* (2007). 2

[BCT15] BÄCHER M., COROS S., THOMASZEWSKI B.: Linkedit: interactive linkage editing using symbolic kinematics. *ACM Trans. Graph.* (2015). 2

[BKR*16] BALDACCI A., KAMENICKÝ R., RIEČICKÝ A., CIGNONI P., DURIKOVIČ R., SCOPIGNO R., MADARAS M.: Gpu-based approaches for shape diameter function computation and its applications focused on skeleton extraction. *Comput. Graph. Forum* (2016). 2

[BWBSH14] BÄCHER M., WHITING E., BICKEL B., SORKINE-HORNUNG O.: Spin-it: Optimizing moment of inertia for spinnable objects. *ACM Trans. Graph.* (2014). 2

[BWK03] BARAFF D., WITKIN A., KASS M.: Untangling cloth. *ACM Trans. Graph. 22*, 3 (2003), 862–870. 5

[CB14] CALDERON S., BOUBEKEUR T.: Point morphology. *ACM Transactions on Graphics (Proc. SIGGRAPH 2014)* (2014). 2

[CGA17] CGAL: CGAL, Computational Geometry Algorithms Library, 2017. http://www.cgal.org. 7, 8

[CZL*15] CHEN X., ZHANG H., LIN J., HU R., LU L., HUANG Q., BENES B., COHEN-OR D., CHEN B.: Dapper: Decompose-and-pack for 3d printing. *ACM Trans. Graph.* (2015). 2

[Eve01] EVERITT C.: Interactive order-independent transparency. *White paper, nVIDIA 2*, 6 (2001), 7. 2, 4

[FBAF08] FAURE F., BARBIER S., ALLARD J., FALIPOU F.: Image-based collision detection and response between arbitrary volume objects. In *Proc. SCA* (2008). 2, 5

[GHF86] GOLDFEATHER J., HULTQUIST J. P. M., FUCHS H.: Fast constructive-solid geometry display in the pixel-powers graphics system. In *Proc. SIGGRAPH* (1986). 2, 4

[GJG16] GARG A., JACOBSON A., GRINSPUN E.: Computational design of reconfigurables. *ACM Trans. Graph.* (2016). 2, 4, 5

[HMA15] HERHOLZ P., MATUSIK W., ALEXA M.: Approximating free-form geometry with height fields for manufacturing. *Comput. Graph. Forum* (2015). 2

[HR05] HABLE J., ROSSIGNAC J.: Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. *ACM Trans. Graph.* (2005). 2

[HTG03] HEIDELBERGER B., TESCHNER M., GROSS M.: Real–time volumetric intersections of deforming objects. In *Vision, Modeling, and Visualization: Proceedings* (2003), AKA, p. 461. 2

[IO07] INUI M., OHTA A.: Using a gpu to accelerate die and mold fabrication. *IEEE Comput. Graph. Appl.* (2007). 2

[JKSH13] JACOBSON A., KAVAN L., , SORKINE-HORNUNG O.: Robust inside-outside segmentation using generalized winding numbers. *ACM Trans. Graph. 32*, 4 (2013), 33:1–33:12. 7

[JP*13] JACOBSON A., PANOZZO D., ET AL.: libigl: A simple C++ geometry processing library, 2013. http://igl.ethz.ch/projects/libigl/. 8

[KE95] KENNEDY J., EBERHART R.: Particle swarm optimization. In *Proc. Neural Networks* (1995). 5

[KGP*94] KELLEY M., GOULD K., PEASE B., WINNER S., YEN A.: Hardware accelerated rendering of csg and transparency. In *Proc. SIGGRAPH* (1994). 2

[KOLM02] KIM Y. J., OTADUY M. A., LIN M. C., MANOCHA D.: Fast penetration depth computation for physically-based animation. In *Proc. SCA* (2002). 3

[KP03] KNOTT D., PAI D. K.: Cinder: Collision and interference detection in real-time using graphics hardware. In *Proc. GI* (2003). 2

[KS00] KAPLAN C. S., SALESIN D. H.: Escherization. In *Proc. SIGGRAPH* (2000). 2

[LAZ*12] LI H., ALHASHIM I., ZHANG H., SHAMIR A., COHEN-OR D.: Stackabilization. *ACM Trans. Graph.* (2012). 2

[LBRM12] LUO L., BARAN I., RUSINKIEWICZ S., MATUSIK W.: Chopper: Partitioning models into 3d-printable parts. *ACM Trans. Graph.* (2012). 2

[LD60] LAND A. H., DOIG A. G.: An automatic method for solving discrete programming problems. *ECONOMETRICA* (1960). 6

[Lin93] LIN M. C.: *Efficient collision detection for animation and robotics*. PhD thesis, UC Berkeley, 1993. 7

[LMM10] LAUTERBACH C., MO Q., MANOCHA D.: gProximity: Hierarchical gpu-based operations for collision and distance queries. In *Computer Graphics Forum* (2010). 3

[MOK95] MYSZKOWSKI K., OKUNEV O. G., KUNII T. L.: Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer* (1995). 2

[MPBC16] MALOMO L., PIETRONI N., BICKEL B., CIGNONI P.: Flexmolds: Automatic design of flexible shells for molding. *ACM Trans. Graph.* (2016). 2

[PS85] PREPARATA F. P., SHAMOS M. I.: *Computational Geometry: An Introduction*. 1985. 3

[Sch11] SCHRAMM O.: *How to cage an egg*. 2011, pp. 87–104. 2

[SGGM06] SUD A., GOVINDARAJU N., GAYLE R., MANOCHA D.: Interactive 3d distance field computation using linear factorization. In *Proc. I3D* (2006), I3D '06. 3

[SGHS98] SHADE J., GORTLER S., HE L.-W., SZELISKI R.: Layered depth images. In *Proc. SIGGRAPH* (1998). 2

[STC*13] SKOURAS M., THOMASZEWSKI B., COROS S., BICKEL B., GROSS M.: Computational design of actuated deformable characters. *ACM Trans. Graph.* (2013). 2

[SVJ15] SACHT L., VOUGA E., JACOBSON A.: Nested cages. *ACM Trans. Graph. 34*, 6 (2015). 2, 3

[SZ15] SUN T., ZHENG C.: Computational design of twisty joints and puzzles. *ACM Trans. Graph.* (2015). 2

[TCG*14] THOMASZEWSKI B., COROS S., GAUGE D., MEGARO V., GRINSPUN E., GROSS M.: Computational design of linkage-based characters. *ACM Trans. Graph.* (2014). 2

[UKIG11] UMETANI N., KAUFMAN D. M., IGARASHI T., GRINSPUN E.: Sensitive couture for interactive garment editing and modeling. *ACM Trans. Graph.* (2011). 2

[UKSI14] UMETANI N., KOYAMA Y., SCHMIDT R., IGARASHI T.: Pteromys: Interactive design and optimization of free-formed free-flight model airplanes. *ACM Trans. Graph.* (2014). 2

[VGB*14] VANEK J., GALICIA J. A. G., BENES B., MĚCH R., CARR N., STAVA O., MILLER G. S.: Packmerger: A 3d print volume optimizer. *Comput. Graph. Forum* (2014). 2

[WP09] WAMPLER K., POPOVIĆ Z.: Optimal gait and form for animal locomotion. *ACM Trans. Graph.* (2009). 6

[ZGZJ16] ZHOU Q., GRINSPUN E., ZORIN D., JACOBSON A.: Mesh arrangements for solid geometry. *ACM Trans. Graph.* (2016). 7, 8

[ZSMS14] ZHOU Y., SUEDA S., MATUSIK W., SHAMIR A.: Boxelization: Folding 3d objects into boxes. *ACM Trans. Graph.* (2014). 2