FLOWPAK: Flow-based Ornamental Element Packing

Reza Adhitya Saputra* University of Waterloo
Craig S. Kaplan† University of Waterloo
Paul Asente‡ Adobe Research
Radomír Měch§ Adobe Research

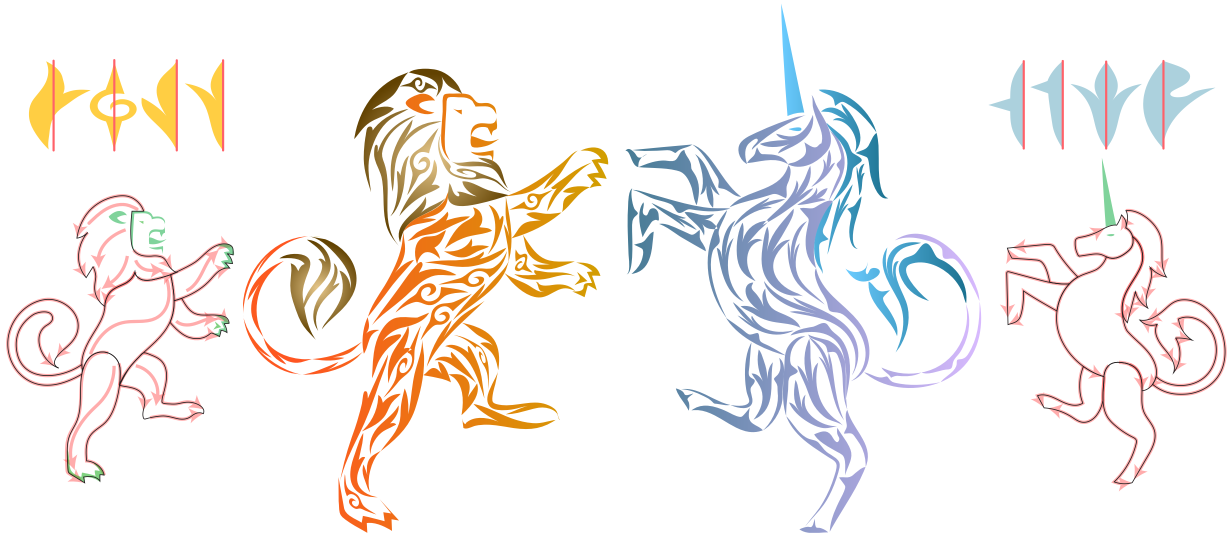


Figure 1: Ornamental packings of a lion and a unicorn. The diagram next to each animal shows a set of four ornamental elements used in the packing (top) and the annotated container regions (bottom). Each ornamental element has a red spine that is used to deform it along a streamline. In the containers, black curves represent boundaries, red curves with arrows represent directional guides, and green curves are fixed elements copied into the final design. The colours in the final rendering were added manually.

Abstract

We present a technique for drawing ornamental designs consisting of placed instances of simple shapes. These shapes, which we call elements, are selected from a small library of templates. The elements are deformed to flow along a direction field interpolated from user-supplied strokes, giving a sense of visual flow to the final composition, and constrained to lie within a container region. Our implementation computes a vector field based on user strokes, constructs streamlines that conform to the vector field, and places an element over each streamline. An iterative refinement process then shifts and stretches the elements to improve the composition.

Index Terms: I.3.3 [Computing Methodologies]: Computer Graphics—Picture/Image Generation; I.3.m [Computing Methodologies]: Computer Graphics—Miscellaneous;

1 Introduction

A popular style of ornamental design involves filling a *container* region with a number of small decorative *elements*. Elements are simple geometric forms, often stylized flora, spirals, or other abstract shapes. The elements are large enough that they can be appreciated individually, but they work together to communicate the overall container shape. Typically, they also form a cohesive stylistic family. Figure 2 shows four examples of these sorts of compositions, which we refer to as *ornamental packings*.

*e-mail: radhitya@uwaterloo.ca
†e-mail: csk@uwaterloo.ca
‡e-mail: asente@adobe.com
§e-mail: rmech@adobe.com

In studying designs like those in Figure 2, we have identified high-level principles that are important to their construction:

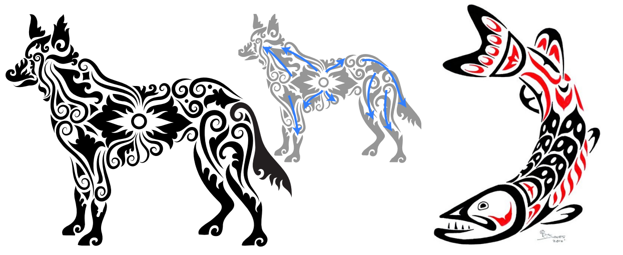
- **Balance.** A composition does not exhibit too much variation in local amounts of positive and negative space. Typically, this goal is accomplished by limiting variation in the diameters of elements (controlling the variation in positive space), and in ensuring that elements are spaced evenly (controlling negative space).
- **Flow.** In local parts of a composition, the elements are oriented to communicate a sense of directionality or flow. All of the examples in Figure 2 exhibit some amount of flow. In the dog, many elements appear to flow outward from the flower in the centre of the torso, and then up the neck and down into the legs. The scales and other elements on the fish flow along the length of its body. In the lion and skull, elements flow horizontally outward from a central axis of symmetry, suggesting fur in the case of the lion. Flow adds visual interest to a composition, engaging the viewer by providing a sense of progression and movement through elements.
- **Uniformity amidst variety.** Repeated elements must balance between two opposing forces. *Uniformity* aims for an overall unity of design; *variety* seeks to break up the monotony of pure repetition. Elements should be permitted to vary in shape, but in a controlled way. We refer to this principle as *uniformity amidst variety*, a term borrowed from philosopher Francis Hutcheson [15]. Gombrich also writes eloquently on the role of variation in design [9]. In our examples, the dog's spirals and the fish's scales both obey this principle. The lion and skull do as well, except that half of the elements are reflected copies of the other half, across a vertical line through the centre of the composition. This repetition emphasizes the bilateral symmetry in the design.
- **Fixed elements.** Compositions use a small number of fixed elements to solve specific design problems or provide focal points. In any figurative drawing, eyes serve as a powerful focal

point; every example in Figure 2 has eyes drawn in as unique elements (the dog's eye is expressed via a carefully placed spiral). Other situations that call for specialized shapes include the dog's paws, the fish's teeth and fins, the lion's eyes and nose, and the skull's teeth. Sharp variation in the balance of positive and negative space can also be used to emphasize a focal point, as in the fish's head and the lion's face that contain considerable amount of empty space.

- **Boundaries.** In many ornamental packings, elements are carefully arranged to conform to and emphasize container boundaries. The fish demonstrates this principle most clearly: we can easily fill in the gaps between elements to form a mental image of a continuous outline. The dog's elements are also well aligned to indicate the container shape. However, this rule is not universal. The lion artfully subverts it with elements that flow outward to an indistinct boundary, helping to convey the appearance of fur.

There has been a moderate amount of past research in computer graphics, particularly in the field of non-photorealistic rendering, on the generation of packings or mosaics. See Section 2 for specific examples. However, this past work is not appropriate for creating designs like those of Figure 2. Most techniques pack elements via rigid transformations, leading to high uniformity but insufficient variety. We believe that there is value in articulating systematic modes of geometric deformation that can generate plausible families of related decorative elements from a single input shape. They also focus on packing large numbers of small elements. We are interested in the compositional properties of large, visually distinct elements.

In this paper we present FLOWPAK, a technique for filling a container region with elements that conform to a user-specified flow. Elements can be oriented in the local direction of flow, but can also be deformed to capture changes in flow direction. We express the user's desired flow by placing evenly spaced streamlines inside the container region. Each streamline is then replaced by an element chosen from a pre-drawn set. The element is bent along the streamline to communicate flow, and also deformed to balance the usage of negative space with elements placed on adjacent streamlines. The final designs, such as the lion and unicorn shown in Figure 1, aim to obey the design principles articulated above.


(a) Dog (b) Fish

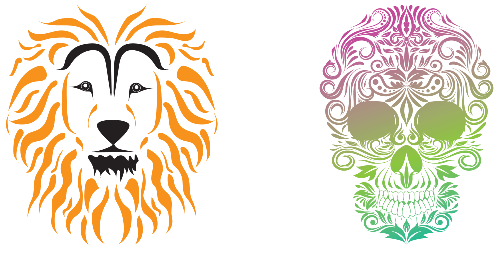

(c) Lion (d) Skull

Figure 2: Examples of ornamental packings: (a) Dog (by ComicVector703 on Shutterstock), including an inset visualizing the flow directions of elements; (b) Fish in the style of Haida art (by Russ Jones, used with permission); (c) Lion (from StockUnlimited); (d) Skull (by alitdesign on Shutterstock).

2 Related Work

Our work is most closely related to past research on distributing small geometric elements to create textures or mosaics. Hausner [11] used a variant of Lloyd's method to distribute square tiles, oriented relative to a vector field, in a simulation of traditional mosaics. Later work extended this approach to distributing distinct element shapes [12], modelling street networks [4], computing layouts by packing and deforming polyforms [26], and incorporating an FFT-based image correlation step to produce more uniform negative space [6].

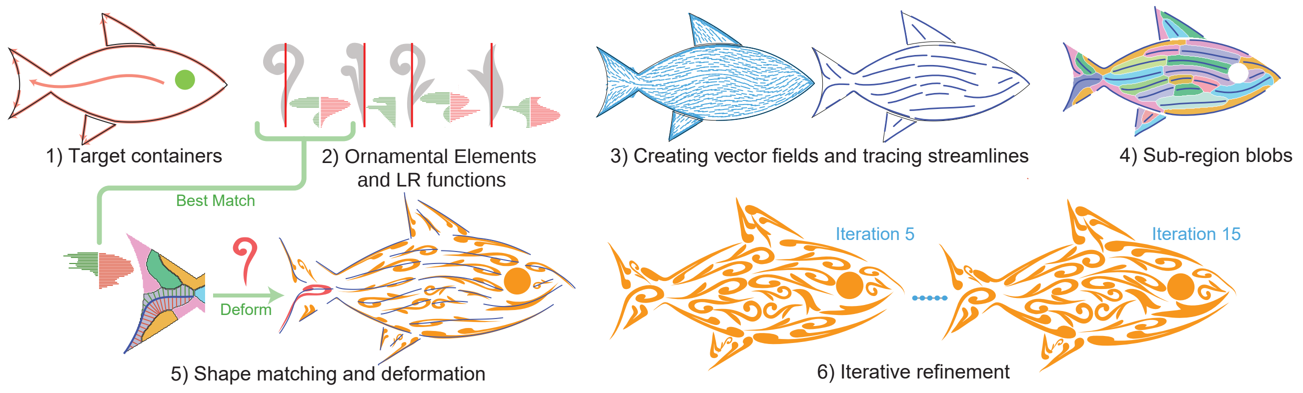
A separate thread of research treats the placement of elements as a form of example-based texture synthesis; see, for example, the work of Hurtut et al. [14] and AlMeraj et al. [1]. In these techniques the goal is to reproduce the statistical properties of an input texture, including the irregular spacing between elements. Ijiri et al. [16] proposed a growth model in which elements could be oriented relative to a vector field and constrained by boundary curves. In these cases, as with the techniques based on Lloyd's method, elements are placed via translation and rotation, with no provision for deformation.

The goal of Jigsaw Image Mosaics (JIMs) [20] was to pack elements into a container region tightly, leaving no negative space at all. JIMs were constructed via an optimization that permitted limited degrees of overlap and deformation. Kwan et al. developed the Pyramid of Arclength Descriptor (PAD) [21] to pack elements tightly by identifying compatible portions of their boundaries. Kaplan and Salesin [19] deformed a single user-supplied input shape into one that could tile the plane. Reinert et al. [27] generated packing layouts by projecting objects from a high dimensional feature space $\mathbb{R}^m$ to $\mathbb{R}^2$ while also inferring users' intentions when manually placing elements. However, these techniques do not consider the design opportunities offered by flow.

A distinct category of past research seeks to develop explicit procedural models for authoring decorative patterns. Wong et al. [29] articulated a set of design principles for decorative art: repetition, balance, and conformation to geometric constraints. They went on to describe a grammar-like system for laying out floral ornament. Beneš et al. [3] developed an interactive interface to guide procedural models in generating decorative elements. Lu et al. developed DecoBrush [22], in which ornamental elements are deformed along line art but are not required to fill containers. The recent PATEX system by Guerrero et al. [10] preserves high-level geometric relationships like symmetry and repetition while ornamental designs are edited.

Our work is also related to research on packing individual letterforms or blocks of text into container regions. Xu and Kaplan [30] and Zou et al. [33] constructed *calligrams* by filling a container with a small number of letters, making up one or two words. The goal is to balance between consuming the container space and preserving legibility. These techniques permit too much deformation (and hence insufficient uniformity) for our ornamental application. Maharik et al. explored Digital Micrography [23], in which lines of small-scale text are deformed to fit along dense streamlines in a container. We take inspiration from their method, but seek to place fewer, larger elements taken from a small library of elements and to use shorter, sparser, and less regular streamlines.

Finally, some recent work has explored the elaboration of ornamental patterns on surfaces, under constraints imposed by fabrication. Chen et al. [5] described a method to synthesize filigree patterns. Zehnder et al. [32] proposed semi-automated tools for deforming ornamental curves to cover a surface. In both cases, the layout of elements must be computed to satisfy both aesthetic and structural goals—most obviously, elements must overlap to produce a connected result that will hold together when 3D printed.



1) Target containers
2) Ornamental Elements and LR functions
3) Creating vector fields and tracing streamlines
4) Sub-region blobs
Best Match
Deform
5) Shape matching and deformation
Iteration 5
Iteration 15
6) Iterative refinement

Figure 3: A visualization of the steps in our ornamental packing algorithm. The input containers are shown as three black outlines in (a): the body and two fins. They are annotated with directional guides in red and fixed elements (in this case, the eye) in green. The steps in the algorithm follow the description at the beginning of Section 4.

3 Problem Formulation

We formally define the problem to be solved as follows. The user provides several pieces of input to our system:

1. A set of target containers. Each container is a closed curve to be filled with ornamental elements.
2. A set of direction guides that guide the placement algorithm, defining the flow of the results. Every target container must have at least one guide, and some or all of the guides typically follow the container boundaries.
3. An optional set of fixed elements that we transfer directly to the result.
4. A set of ornamental elements, each with a spine that will control its deformation.

The first three inputs are combined into a single diagram, where they are distinguished by their colors; see the left and right drawings in Figure 1. The goal of our algorithm is to fill each target container with elements, trying to satisfy several guiding principles:

1. Follow the flow defined by the direction guides.
2. Have as little empty space as possible.
3. Make the spacing between elements be as even as possible.
4. Conform to container boundaries.
5. Vary element width and length to avoid an excessively uniform arrangement.

The next section describes how we achieve these results.

4 Approach

Figure 3 gives an overall view of our system. The numbers in the following steps correspond to parts of the figure.

1. Read the input target containers and copy any fixed elements to the output art (Section 4.1).
2. Analyze the ornamental elements, creating a shape descriptor for each (Section 4.2).
3. Use the direction guides to fill each target container with a vector field then trace streamlines (Section 4.3).
4. Divide the target containers into blobs around the streamlines (Section 4.4).
5. Use the element shape descriptors to determine the best element for each blob. Place the element in the blob, treating it as a skeletal stroke and mapping its spine to the streamline (Section 4.5).
6. Iteratively refine the placement to eliminate empty areas and make the spacing more even (Section 4.6).

4.1 Target containers

The input diagram contains a set of target containers. Each is a single closed curve defining an area to be filled. Most non-trivial examples include more than one target container. For the most part, our algorithm fills each container separately, and so the following explanation is given in terms of a single container. Containers will later be merged in the iterative refinement step.

The artist has the option of including a set of fixed elements that we copy directly into the final result. The following sections include descriptions of how the fixed elements affect the filling algorithm.

We define *input_size* to be the maximum of the combined width or height of all the target containers and fixed elements as laid out by the artist. This value will be used to set various parameters in the synthesis process.

4.2 Ornamental elements and LR functions

An ornamental element is defined as one or more closed curves. Our placement method will eventually deform copies of the element (Section 4.5) using a simple skeletal stroke algorithm [13], so each element must be annotated with a straight spine to guide the deformation. The spine does not need to go through the center of the element—it can be anywhere.

We define two classes of elements: a *full element* extends across both sides of its spine, and a *half element* lies entirely on one side of its spine. Figure 4 shows examples of full and half elements. If the input to our algorithm includes direction guides that coincide with target container boundaries, the placement method will align half elements along these boundaries. If half elements have edges that closely follow their spines, they will visually reinforce container boundaries, as shown in our examples.

We define a simple shape descriptor called an *LR function* that will be used in Section 4.5 to choose which element to place in a particular location. Inspired by the work of Gal et al. [7], we sample the element's spine at n locations and at each location determine how far the ornament extends to the left and right of the spine. The LR function is the set $\{L,R\}$ where $L = \{\ell_1,\ldots,\ell_{n\_f}\}$ is the left function and $R = \{r_1,\ldots,r_{n\_f}\}$ is the right function. The number of samples is denoted by n_f.

The LR function is made scale-invariant by normalizing its domain and range to $[0,1]$. Note that swapping the L and R functions corresponds to reflecting the element across its spine, and reversing each of L and R corresponds to reflecting the element along its spine. We will consider all four combinations of these two reflections when placing an element in a blob (Section 4.4), in order to achieve the best possible fit.

Intuitively, LR functions give an approximate area an ornamental element can claim. Figure 4 shows elements with their left values in green and their right values in red. We have found that $n = 100$ gives sufficient granularity for our algorithm.

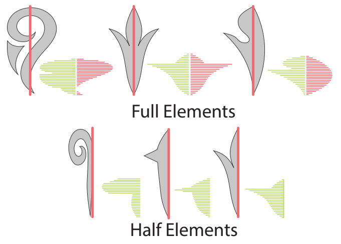


Figure 4: Ornamental elements and their LR functions. Full elements have non-empty left and right sides, while half elements have only one non-empty side. We normalize the LR functions to a unit square.

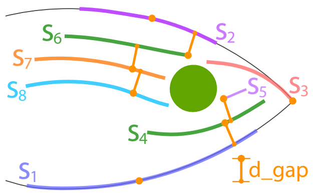


Figure 5: The streamline tracing process. The first streamline s_1 always begins on a directional guide or the container boundary. Subsequent streamlines begin on the container boundary, a directional guide, or at a point that is d_gap away from a previous streamline.

4.3 Creating vector fields and tracing streamlines

To implement the flow principle described in the introduction, we fill each target container with a vector field, constrained by the direction guides in that container.

We sample the directional guides $D = \{d_1, d_2, \ldots, d_{n\_d}\}$ and use the tangent at every sampled point as a directional constraint. We then construct a vector field using the N-RoSy algorithm of Palacios and Zhang [25]. Note that, as shown in Step 3 of Figure 3, fixed elements do not affect the vector field. The artist can include directional guides to guide the vector field around fixed elements if desired.

The next step is to trace streamlines in the vector field, guided by three input parameters:

d_gap	is the desired space between streamlines
s_max	is the maximum desired streamline length
s_min	is the minimum desired streamline length

Because we will ultimately place elements along streamlines without overlap, d_gap determines the approximate width of the placed elements, and s_max the maximum length. We also derive a value d_stop that prevents streamlines from coming too close to each other; in our implementation we compute $d\_stop = 0.8\, d\_gap$.

We adapt the streamline tracing algorithm of Jobard and Lefer [17]. First we generate a set of potential seed points $P = \{p_1, p_2, \ldots, p_{n\_p}\}$ by densely resampling the target container boundary T and the directional guides in D. We use a sampling distance of $0.005\, input\_size$. The first streamline s is generated by randomly removing a seed point from P and following the vector field until one of the following conditions holds:

1. the length of s would exceed s_max.
2. s would come within d_stop of another streamline.
3. s would cross T, leaving the container.
4. s would cross the boundary of a fixed element.

If the length of s is less than s_min, we discard it. Otherwise we sample s, again using $0.005\, input\_size$, and at each point generate two more potential seeds that are d_gap away from s on either side. If a seed is inside the container, we add it to P. The process is repeated until P is empty. Note that the d_stop distance test combined with the s_min length test imply that many attempts to form streamlines will stop immediately, especially as the container fills with streamlines.

Figure 5 shows the creation process, and Algorithm 1 shows the pseudocode. The sort function SORT(P) orders the points in P according to their distance from the boundary T and the directional guides in D, with closer points first and equally distant points ordered randomly. Because the initial points are all on T or on a path in D, their sort value is zero, and they will be processed before any derived points.

Algorithm 1 Tracing streamlines

```
Create a seed list $P = \{p_1, p_2, \ldots, p_{n\_p}\}$ by uniformly resampling
    $T$ and the guides in $D$.
Create an empty set $S$ of streamlines.
Randomly order the elements of $P$.
while $P$ is not empty do
    Generate a new streamline $s$ from $p_1$.
    Remove $p_1$ from $P$.
    if $s$ is longer than $s\_min$ then
        Add $s$ to $S$.
        Create seed points that are $d\_gap$ away from $s$ and
            add them to $P$.
        SORT($P$).
    end if
end while
```

4.4 Sub-region blobs

To assist in choosing which element to place along each streamline, we first subtract the areas of any fixed elements from the target container. We then construct an approximate generalized Voronoi diagram of the interior using the method of Osher and Sethian [24]. The streamlines are then extended at each end, following the vector field, until they encounter the boundaries of their Voronoi regions. We call the area around each streamline a *sub-region blob*. Step 4 of Figure 3 shows the blobs for the sample fish.

We then compute an LR function for each blob as described in Section 4.2, using the streamline as the spine. Because the streamline is not usually straight, we compute the left and right distances along the normals to the streamline. The LR function approximates the blob's shape if the streamline were to be straightened.

4.5 Shape matching and deformation

The next step is to place an ornamental element in each blob. We choose which element to place in the blob by finding the element that minimizes a sum of least squares distance, defined as

$$\sum_{i=1}^{N}(\alpha_{li} - \beta_{li})^2 + \sum_{i=1}^{N}(\alpha_{ri} - \beta_{ri})^2 \qquad (1)$$

where

α_l	is the element left function
α_r	is the element right function
β_l	is the blob left function
β_r	is the blob right function

Every element can be placed in one of four orientations, by optionally incorporating reflections across and along its spine. These reflections correspond, respectively, to swapping the L and R functions and reversing them. When comparing the LR functions for an element and a blob, we compute the least squares distances for all four orientations and choose the orientation with the smallest distance. Note that this matching method automatically places half elements along streamlines that follow container boundaries, visually reinforcing the overall shape.

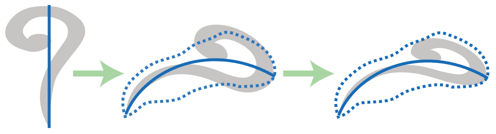


Figure 6: The deformation process bends the element along the streamline and scales it to fit inside the blob.

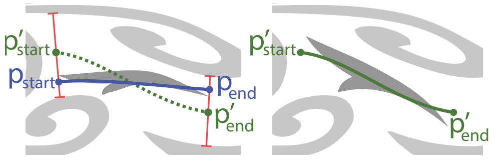


Figure 7: Streamline shifting. We move the streamline's start and end points along perpendiculars, stopping before intersecting neighbouring elements.

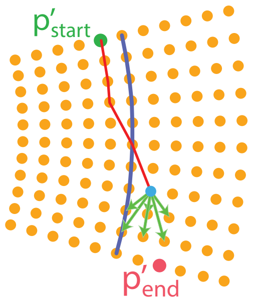


Figure 8: Tracing a shortest path using Dijkstra's algorithm. We generate the orange nodes by resampling and offsetting the original red streamline. The search directions at a node are shown with green arrows.

Algorithm 2 Iterative refinement

```
Input: $E = \{e_1, e_2, ..., e_{n\_e}\}$ as the ornamental element list.
Input: $S = \{s_1, s_2, ..., s_{n\_s}\}$ as the streamline list.
Input: $B = \{b_1, b_2, ..., b_{n\_b}\}$ as the blob list.
Input: $\alpha$ as the growth tolerance
Input: $t$ as the number of iterations
  for $t$ times do
    Sort $E$, $S$, and $B$ by $\textsc{Area}(e_i)$ (smallest first)
    for Element $e_i$ in $E$ do
      $s_i$ is the corresponding streamline of $e_i$
      Calculate $s'_i$ by shifting $s_i$.
      Recompute the LR function of $b_i$ to give $b'_i$
      Calculate $e'_i$ by placing $e_i$ inside $b'_i$
      if $\textsc{Area}(e'_i) \times \alpha > \textsc{Area}(e_i)$ then
        $s_i \leftarrow s'_i$
        $b_i \leftarrow b'_i$
        $e_i \leftarrow e'_i$
      end if
    end for
    Sort $E$, $S$, and $B$ by $\textsc{Area}(e_i)$ (smallest first)
    for Element $e_i$ in $E$ do
      $b_i$ is the corresponding blob of $e_i$
      Calculate $b'_i$ by growing $b_i$.
      Calculate $s'_i$ based on $b'_i$
      Calculate $e'_i$ by placing $e_i$ inside $b'_i$
      $b_i \leftarrow b'_i$
      if $\textsc{Area}(e'_i) \times \alpha > \textsc{Area}(e_i)$ then
        $s_i \leftarrow s'_i$
        $e_i \leftarrow e'_i$
      end if
    end for
  end for
```

We investigated alternatives for shape matching, using an approach discussed by Gal et al. [8] that tries to fill a sub-region blob as much as possible, with heavy penalties if a part of an element protrudes outside the boundary of the blob. However, we found this computation to be more expensive without providing significant advantages over our LR functions.

Once we have chosen an element, we place it along the streamline using a simple skeletal stroke algorithm [13]. We uniformly scale the element's width to make it as wide as possible while still staying inside the blob (Figure 6).

4.6 Iterative refinement

We now refine the overall composition in an iterative process. We perform this part of the algorithm globally, by merging all containers and allowing the elements within them to interact.

The refinement process aims to reduce the amount of negative space and make it more even by growing and shifting the placed ornamental elements. It would be possible to use a greedy approach, improving the placement of each element as much as possible before moving on to the next, However, we have found that gradually improving the placement of all elements leads to a more even result.

Each refinement iteration has two phases. First, we shift the streamlines to more accurately follow the space that is available, as shown in Figure 7. After shifting, we recalculate the LR function for the blob to reflect the new position, and repeat a variant of the element placement process that allows the elements to rotate slightly in their space. Second, we expand each blob to allow it to use adjacent space that is not filled with another element, as shown in Figure 9.

Each refinement iteration considers the blobs in increasing order of placed element area, allowing smaller elements to grow more. While each step usually results in a larger placed element, some configurations can result in a smaller one. We only accept the new element if its area is no smaller than α times its old area, where α is a growth tolerance that we set to 0.9. Elements therefore have some freedom to grow or shrink, in the search for more globally even spacing.

Algorithm 2 gives the overall method, and the following sections give details. We have found that 15 iterations suffice for most designs. Note that in Algorithm 2, the variable E is the list of placed, distorted ornamental elements, and not the set of prototype elements discussed earlier.

Shifting streamlines. There are two issues that keep the initial placement of elements from being evenly distributed. Our streamline placement method keeps streamlines apart, but they may not be spaced completely evenly. More significantly, the ornamental elements often have unbalanced left and right sides and concavities, leading to extra space on one side or the other. Our refinement process shifts streamlines to address these problems.

The shifting process allows the endpoints of the streamline to move to the left or the right relative to the streamline, depending on which side has more empty space. This allows the streamline's element to become wider and fill more of the space (Figure 7). It also gives the streamline room to extend if its endpoints were too close to boundaries of other placed elements.

Given the endpoints p_{start} and p_{end}, we calculate new endpoints p'_{start} and p'_{end}. We generate perpendicular vectors to the left side and to the right side at each endpoint and construct a line segment joining the points where the vectors intersect other placed elements. We then move the endpoint of the streamline towards the midpoint of this segment. To enforce the principle of gradual refinement, we do not allow the endpoint to move more than g_{limit} units, where $g_{\text{limit}} = 0.005\, input\_size$ (Recall that $input\_size$ is the maximum dimension of the design as described in Section 4.1).

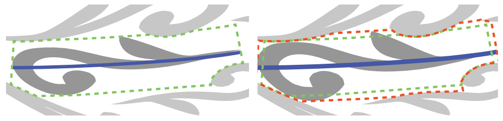


Figure 9: (a) An elementt with its sub-region blob shown in dashed green line. Note that any blob is constrained by the neighboring elements. (b) The dashed red line is the grown blob, which accommodates an enlarged element.

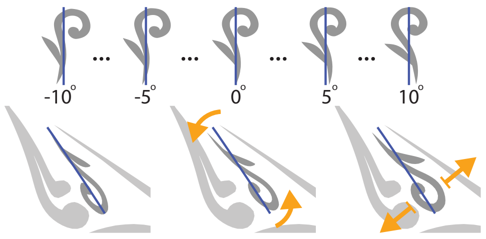


Figure 10: Top row: rotated versions of the original element. The best rotation angle is chosen via least squares matching. Bottom row: original, rotated, and enlarged versions of an element.

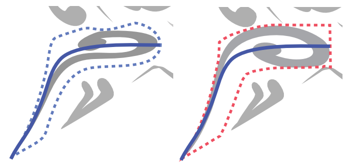


Figure 11: An element that reflects across its spine during iterative refinement. LR functions and least squares shape matching allow an element to reflect across its spine, along its spine, or both.

We replace the streamline with a path joining p'_{start} and p'_{end}. Our goal is to create a path that is smooth and does not deviate too much from the vector field. We calculate the shifted streamline by performing Dijkstra's algorithm on a non-rectangular graph that respects the vector field (Figure 8), using a method similar to one by Xu and Mould [31] for pathfinding in a vector field.

To construct the graph, we begin by densely sampling the original streamline with a distance of $0.25\,g_{\text{limit}}$. We then duplicate the points, offset to the left and right, again using $0.25\,g_{\text{limit}}$. The duplication is repeated until the graph extends to the left and right of the streamline by a distance equal to the maximum left and right widths of the blob (i.e., the maximum values in an unnormalized version of the blob's LR function).

For a node, we check its $N = 150$ nearest neighbors, considering only neighbors where the angle between the line into the current node and the line to the neighbor form an angle greater than 90°, thereby preventing the streamline from backtracking. The cost of an edge from n_a to n_b is

$$w = w_f(1 - f^p) + w_d\,D(s_i, n_a) \tag{2}$$

where

f	is $(n_b - n_a) \cdot \vec{v}$
$\vec{v}$	is a sampled vector of the vector field
s_i	is the original streamline
$D()$	is a distance function between a polyline and a point

In practice, we set $w_d = 0.1$, $w_f = 1$, and $p = 3$.

After finding a set of points, we fit cubic Bézier curves using a method devised by Schneider [28] and extend the path at both ends by following the vector field until it intersects the edges of its blob.

Growing blobs. The growth process tries to enlarge each sub-region blob to claim empty space. Given a blob b_i, we calculate a larger blog b'_i by offsetting its boundaries until they intersect other placed elements (Figure 9). To enforce gradual growth, the offset cannot be larger than g_{limit}, where $g_{\text{limit}} = 0.005\,\mathit{input\_size}$.

The value g_{limit} used in growing blobs and shifting streamlines limits the speed of the refinement. Making it larger would require fewer iterations to fill the available space, but at a cost of elements growing less evenly.

Element placement. In the refinement process, we allow a more flexible element placement so that the elements can fill more of their blobs. We allow the element to rotate by a small amount, up to ten degrees, before placing it, as shown in Figure 10. We generate rotated versions of e_i with varying angles $r_{\text{angle}} = 1°, 2°, 3°, ..., 10°$ and precompute LR functions for each. The shape matching algorithm (Section 4.5) automatically choses the best rotation. It can also choose to reflect the element across its spine, along its spine, or both (Figure 11).

5 Implementation and Results

We design our containers and decorative elements in a vector graphics editor, and then use them as input to a C++ program that outputs final placed elements in an SVG file. We use the Clipper library [18] for calculation of LR functions and for testing polygon intersections during deformation and growth. As a postprocess, we optionally smooth outlines and replace polygonal paths with Bézier curves. Finally, we apply colours and other treatments in an editor.

Our technique is fast except for the iterative refinement process, which considers a large number of variations to the composition via brute-force computation. On a computer with an Intel i7-4790K processor at 4.0 Ghz, 15 iterations of refinement on a packing of 50 elements takes about an hour. Our software is not intended to run interactively; still, we believe the performance could be improved significantly through the use of more sophisticated 2D geometric data structures like quadtrees.

We tested our approach using a variety of container shapes, based mostly on animals, and many different ornamental elements with varying amounts of geometric complexity. In Figure 12, we show a packing of a rhinoceros with simple teardrop elements that demonstrates the variety we achieve in shape and curvature. We use more complex leaf elements on the bear in Figure 13, and adjust the tracing parameters to obtain shorter placed elements. We also process the placed elements to create a distressed look.

The packing of a cat in Figure 14 demonstrates a symmetric packing with a fur contour inspired by Figure 2c. We only compute the left part and reflect the result. The elements around the cheeks and the chin extend outward, not following the boundary, and creating the appearance of fur.

We experimented with two extensions to our pipeline, which could enhance its aesthetic value and flexibility. First, in Figure 15 we allow the user to draw *fixed spines* in addition to fixed elements. These fixed spines act like pre-placed streamlines, which will be assigned blobs and then elements. However, they are not required to follow the surrounding vector field, and are not shifted during the refinement process. Fixed spines are used in Figure 15 for the flower petals in the torso and the paws. Second, in Figure 16 we construct explicit new shapes (drawn in brown) to fill the negative space between placed elements (in black), by computing offset polygons from the negative space between elements. The result is a distinct and appealing style.

Finally, we asked an artist to draw containers and a decorative elements. The result is the bird design shown in Figure 17. The artist requested that different elements and densities be used in different container regions; the result has sparse "Y" elements in the breast and head, and denser "O" elements in the wings. The artist was pleased with the results.

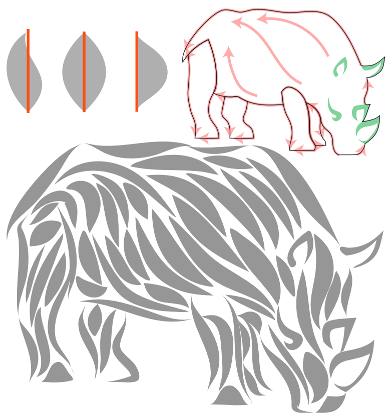


Figure 12: A packing of a rhinoceros. Simple teardrop-shaped elements lead to variety in size and curvature.

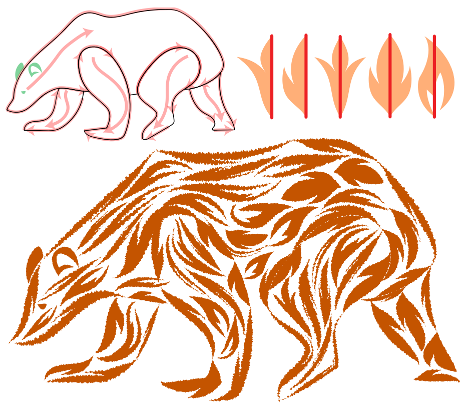


Figure 13: A bear packed with leaf elements. We manually add noise to the elements in the output to create a distressed look.

6 Conclusion and Future Work

We have demonstrated a method to create ornamental packings in which vector fields are used to provide a sense of visual flow. We achieve a degree of uniformity by using repeated copies of a small set of initial decorative elements, but balance that uniformity with variety by deforming those elements. An iterative refinement process improves upon an initial placement of elements, arriving at a final composition with a more even use of negative space.

We see many possibilities for further improvements to our algorithm and future research on ornamental packing.

- In our current algorithm, elements must completely fill streamlines. It would be worthwhile to investigate whether multiple shorter elements could be threaded along streamlines.
- Our results do not have significant high-curvature streamlines like u-turns, since they could unpleasantly fold the decorative elements. This could be solved with a folding avoidance algorithm [2].
- Our iterative refinement process uses a greedy approach, in which we iterate over all placed elements in a fixed order from smallest to largest. We would like to investigate global optimizations that could be applied to improve the overall composition in an order-independent way. A natural first choice would be an approach based on simulated annealing, although performance could become a more serious issue in that case.
- As with many research projects in non-photorealistic rendering, this work raises deep questions about the aesthetics of ornamental packings. What compositions are most appealing? What is the most effective way to distribute negative space? Some

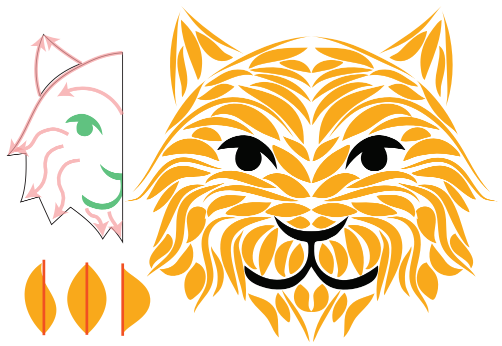


Figure 14: A packing with a symmetric layout; we only compute the left half and reflect the result. The elements around the cheeks and the chin are not aligned to the boundary, creating a fur-like effect.

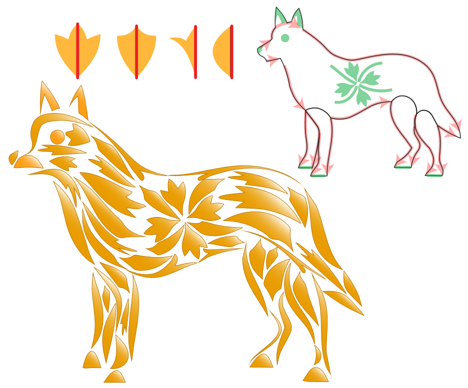


Figure 15: A packing of a dog. The fixed elements, shown as green shapes in the diagram, are copied as-is to the output; fixed spines, shown as green paths, force the placement of new elements at the given locations.

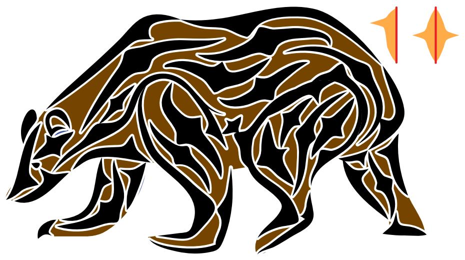


Figure 16: A packing of the same container as in Figure 13. We place longer and sparser elements and synthesize additional forms to fill negative space.

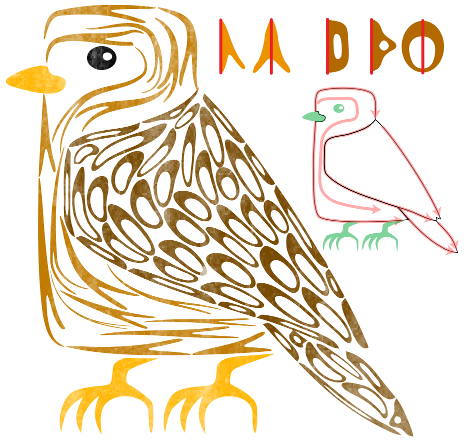


Figure 17: A packing of a bird, based on input provided by an artist.

hand-drawn compositions insert additional small elements, like circles and squares, to break up large areas of negative space; an automated simulation of this process would be helpful.

- We would like to explore the automatic creation and placement of the fixed elements, perhaps by discovering them as salient regions in source photographs, and extracting and vectorizing them. This extraction must be carried out carefully, yielding enough fixed elements to communicate a container clearly without disrupting the uniformity of the design.

Acknowledgments

We thank the reviewers for helpful feedback. Thanks to Russ Jones for permission to use the fish art in Figure 2. Yusuf Umar designed elements used in Figure 17 and Daichi Ito gave us early feedback on creating good ornamental artwork. This research began as an internship at Adobe Research and was sponsored in part by generous gifts from Adobe Research and NSERC.

References

[1] Z. AlMeraj, C. S. Kaplan, and P. Asente. Patch-based geometric texture synthesis. In *Proceedings of the Symposium on Computational Aesthetics*, CAE '13, pp. 15–19. ACM, New York, NY, USA, 2013. doi: 10.1145/2487276.2487278

[2] P. J. Asente. Folding Avoidance in Skeletal Strokes. In M. Alexa and E. Y.-L. Do, eds., *Eurographics Workshop on Sketch-Based Interfaces and Modeling*. The Eurographics Association, 2010. doi: 10.2312/SBM/SBM10/033-040

[3] B. Beneš, O. Št'ava, R. Měch, and G. Miller. Guided procedural modeling. *Computer Graphics Forum*, 30(2):325–334, 2011. doi: 10.1111/j.1467-8659.2011.01886.x

[4] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang. Interactive procedural street modeling. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pp. 103:1–103:10. ACM, New York, NY, USA, 2008. doi: 10.1145/1399504.1360702

[5] W. Chen, X. Zhang, S. Xin, Y. Xia, S. Lefebvre, and W. Wang. Synthesis of filigrees for digital fabrication. *ACM Trans. Graph.*, 35(4):98:1–98:13, July 2016. doi: 10.1145/2897824.2925911

[6] K. Dalal, A. W. Klein, Y. Liu, and K. Smith. A spectral approach to NPR packing. In *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '06, pp. 71–78. ACM, New York, NY, USA, 2006. doi: 10.1145/1124728.1124741

[7] R. Gal, A. Shamir, and D. Cohen-Or. Pose-oblivious shape signature. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):261–271, Mar. 2007. doi: 10.1109/TVCG.2007.45

[8] R. Gal, O. Sorkine, T. Popa, A. Sheffer, and D. Cohen-Or. 3D collage: Expressive non-realistic modeling. In *Proceedings of the 5th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '07, pp. 7–14. ACM, New York, NY, USA, 2007. doi: 10.1145/1274871.1274873

[9] E. H. Gombrich. *The Sense of Order: A Study in the Psychology of Decorative Art*. Phaidon Press Limited, 1984.

[10] P. Guerrero, G. Bernstein, W. Li, and N. J. Mitra. PATEX: Exploring pattern variations. *ACM Trans. Graph.*, 35(4):48:1–48:13, July 2016. doi: 10.1145/2897824.2925950

[11] A. Hausner. Simulating decorative mosaics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pp. 573–580. ACM, New York, NY, USA, 2001. doi: 10.1145/383259.383327

[12] S. Hiller, H. Hellwig, and O. Deussen. Beyond Stippling - Methods for Distributing Objects on the Plane. *Computer Graphics Forum*, 2003. doi: 10.1111/1467-8659.00699

[13] S. C. Hsu, I. H. H. Lee, and N. E. Wiseman. Skeletal strokes. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, UIST '93, pp. 197–206. ACM, New York, NY, USA, 1993. doi: 10.1145/168642.168662

[14] T. Hurtut, P.-E. Landes, J. Thollot, Y. Gousseau, R. Drouillhet, and J.-F. Coeurjolly. Appearance-guided synthesis of element arrangements by example. In *Proceedings of the 7th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '09, pp. 51–60. ACM, New York, NY, USA, 2009. doi: 10.1145/1572614.1572623

[15] F. Hutcheson. *An Inquiry Into the Original of Our Ideas of Beauty and Virtue*. J. and J. Knapton and others, 1729.

[16] T. Ijiri, R. Měch, T. Igarashi, and G. S. P. Miller. An example-based procedural system for element arrangement. *Comput. Graph. Forum*, 27:429–436, 2008.

[17] B. Jobard and W. Lefer. Creating evenly-spaced streamlines of arbitrary density. In W. Lefer and M. Grave, eds., *Visualization in Scientific Computing '97: Proceedings of the Eurographics Workshop in Boulogne-sur-Mer France, April 28–30, 1997*, pp. 43–55. Springer Vienna, Vienna, 1997. doi: 10.1007/978-3-7091-6876-9_5

[18] A. Johnson. Clipper—an open source freeware library for clipping and offsetting lines and polygons. `http://www.angusj.com/delphi/clipper.php`, 2014.

[19] C. S. Kaplan and D. H. Salesin. Escherization. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pp. 499–510. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. doi: 10.1145/344779.345022

[20] J. Kim and F. Pellacini. Jigsaw image mosaics. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pp. 657–664. ACM, New York, NY, USA, 2002. doi: 10.1145/566570.566633

[21] K. C. Kwan, L. T. Sinn, C. Han, T.-T. Wong, and C.-W. Fu. Pyramid of arclength descriptor for generating collage of shapes. *ACM Trans. Graph.*, 35(6):229:1–229:12, Nov. 2016. doi: 10.1145/2980179.2980234

[22] J. Lu, C. Barnes, C. Wan, P. Asente, R. Měch, and A. Finkelstein. DecoBrush: Drawing structured decorative patterns by example. *ACM Trans. Graph.*, 33(4):90:1–90:9, July 2014. doi: 10.1145/2601097.2601190

[23] R. Maharik, M. Bessmeltsev, A. Sheffer, A. Shamir, and N. Carr. Digital micrography. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pp. 100:1–100:12. ACM, New York, NY, USA, 2011. doi: 10.1145/1964921.1964995

[24] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on hamilton-jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, Nov. 1988. doi: 10.1016/0021-9991(88)90002-2

[25] J. Palacios and E. Zhang. Rotational symmetry field design on surfaces. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07. ACM, New York, NY, USA, 2007. doi: 10.1145/1275808.1276446

[26] C.-H. Peng, Y.-L. Yang, and P. Wonka. Computing layouts with deformable templates. *ACM Trans. Graph.*, 33(4):99:1–99:11, July 2014. doi: 10.1145/2601097.2601164

[27] B. Reinert, T. Ritschel, and H.-P. Seidel. Interactive by-example design of artistic packing layouts. *ACM Trans. Graph.*, 32(6):218:1–218:7, Nov. 2013. doi: 10.1145/2508363.2508409

[28] P. J. Schneider. An algorithm for automatically fitting digitized curves. In A. S. Glassner, ed., *Graphics Gems*, pp. 612–626. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[29] M. T. Wong, D. E. Zongker, and D. H. Salesin. Computer-generated floral ornament. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pp. 423–434. ACM, New York, NY, USA, 1998. doi: 10.1145/280814.280948

[30] J. Xu and C. S. Kaplan. Calligraphic packing. In *Proceedings of Graphics Interface 2007*, GI '07, pp. 43–50. ACM, New York, NY, USA, 2007. doi: 10.1145/1268517.1268527

[31] L. Xu and D. Mould. Procedural tree modeling with guiding vectors. *Comput. Graph. Forum*, 34(7):47–56, Oct. 2015. doi: 10.1111/cgf.12744

[32] J. Zehnder, S. Coros, and B. Thomaszewski. Designing structurally-sound ornamental curve networks. *ACM Trans. Graph.*, 35(4):99:1–99:10, July 2016. doi: 10.1145/2897824.2925888

[33] C. Zou, J. Cao, W. Ranaweera, I. Alhashim, P. Tan, A. Sheffer, and H. Zhang. Legible compact calligrams. *ACM Trans. Graph.*, 35(4):122:1–122:12, July 2016. doi: 10.1145/2897824.2925887