Deformation-Driven Element Packings

by

Reza Adhitya Saputra

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2020

Last edited: Wednesday 29th July, 2020 06:26

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	YYY YYY Professor, Dept. of YYY, University of YYY
Supervisor(s):	Craig S. Kaplan Professor, Dept. of Computer Science, University of Waterloo
Internal Member:	Christopher Batty Professor, Dept. of Computer Science, University of Waterloo
Internal-External Member:	YYY YYY Professor, Dept. of YYY, University of Waterloo
Other Member(s):	Daniel Vogel Professor, Dept. of Computer Science, University of Waterloo

Author's Declaration

This thesis consists of materials which I authored or co-authored: see Statement of Contributions. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

This thesis was built from previous publications authored by myself, Dr. Craig S. Kaplan, and Dr. Paul Asente.

FLOWPAK in Chapter 3 was initially conducted with Dr. Paul Asente, and Dr. Radomír Měch at Adobe Research in San Jose, ~~CA~~. FLOWPAK is also discussed in greater detail in our publication "FLOWPAK: Flow-Based Ornamental Element Packing" [SKAM17] and in our patent "Computerized Generation of Ornamental Designs by Placing Instances of Simple Shapes in Accordance With a Direction Guide" [AKMS20].

RepulsionPak in Chapter 4 and Quantitative Metrics in Chapter 6 were taken from "RepulsionPak: Deformation-Driven Element Packing with Repulsion Forces" [SKA18] and "Improved Deformation-Driven Element Packing with RepulsionPak" [SKA19].

AnimationPak in Chapter 5 was taken from "AnimationPak: Packing Elements with Scripted Animations" [SKA20].

Abstract

We present three methods to fill a container shape with deformable instances of geometric elements, creating an artistic composition called a packing. The first method is FLOWPAK that deforms elements to flow along a vector field interpolated from user-supplied strokes, giving a sense of visual flow to the final composition. The second method is RepulsionPak that utilizes repulsion forces to pack elements, each ~~is~~ represented as a mass-spring system, allowing them to deform to achieve a better fit with their neighbors and the container. The last method is AnimationPak that creates animated packings by arranging animated two-dimensional elements inside a static container. We represent animated elements in a three-dimensional spacetime domain, and view the animated packing problem as a three-dimensional packing in that domain. Finally, we propose statistical methods for measuring the evenness of 2D element distributions, which provide quantitative means of evaluating and comparing packing algorithms.

Acknowledgements

I would like to thank all people who have helped me during my PhD study:

- Craig S. Kaplan and Paul Asente, for being the best teachers.
- Radomír Měch, for the internship and the help with FLOWPAK paper.
- Daichi Ito, for the discussion about design principles and using RepulsionPak to create a packing.
- Dietrich Stoyan, for the discussion about spatial statistics.
- Danny Kaufman, for the discussion about physical simulations.
- Steve Mann, for giving me valuable lessons.
- Bill Cowan, for his advice.
- Christopher Batty and Dan Vogel, for being my PhD committee members.
- Matt Thorne, Alex Pytel, and Tyler Nowicki, for all the lunches together.
- All my running friends in Kitchener-Waterloo and ultra running community in Ontario, who teach me on how to keep grinding.
- Russ Jones, for permission to use the fish art in Figure 3.2d.
- Yusuf Umar, for designing ornamental elements and the box bird in Figure 3.17.

Dedication

For my Mom, Dad, and sister.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A *packing* is an arrangement of geometric *elements* within a *container* region in the plane. As an artistic composition, a packing can communicate a relationship between a whole and the parts that make it up. Elements work together to communicate the overall container shape, but each is large enough to be appreciated individually. Elements are shapes like animals, plants, ~~abstract, geometric,~~ or man-made objects. Elements can also be interpreted as *positive space*. Packings are popular in logo design, graphic design, and art. Figure 1.1a shows the Unilever logo, which is a 2D packing of elements arranged inside a "U" container.

The subset of the container that does not belong to any element is called *negative space* (Figure 1.1b). We can also interpret negative space as separation and gaps between elements. The evenness of negative space plays an important role in packings. The artist should arrange elements in a way that their boundaries interlock with each other, causing the separation between neighboring elements to become roughly the same everywhere. After all, element interlocking is imperfect. It is not uncommon that the artist artfully places small elements such as triangles or circles to reduce remaining large gaps, see two examples in Figure 1.2.

Elements can also be oriented to follow certain directions for aesthetic reasons. Figure 1.2b is an ornamental packing of long thin elements that are bent and oriented outward from the center of the torso to create a flow visual style. Another example is shown in Figure 1.3, which is a packing of leaves, each ~~is~~ oriented toward the center of the container. ~~On~~ both examples, the flow visual style gives an impression of progression and movement~~s~~.

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.

(a)

Unilever

(b)

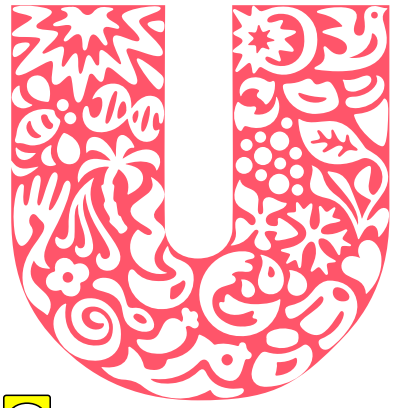


Figure 1.1: (a) The Unilever logo packing that consists of 25 elements arranged inside a "U" container. (b) The red region is called negative space, which is the remaining area that is not claimed by the elements.

(a)

(b)

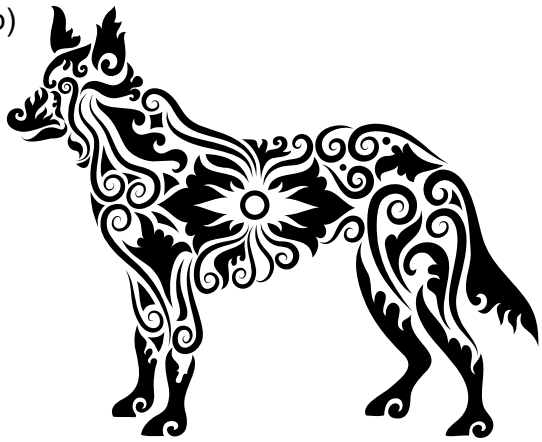


Figure 1.2: Two packing examples in graphic design. (a) A packing of autumn-themed elements, created by Balabolka. (b) A packing of abstract-shaped elements that create a flow visual style, created by ComicVector703.

Figure 1.3: A land art packing that is composed of leaves, created by James Brunt. Each leaf is oriented toward the center of the circle.

See Chapter 2 for specific examples. However, most techniques pack elements via rigid transformations, leading to uneven element distribution and overlaps. Jigsaw Image Mosaics [KP02] and collages based on the Pyramid of Arclength Descriptor [KSH+16] are *data-driven*. These techniques rely on a large library of elements, so that given an area to fill in a partial composition, there is likely to be an element in the library with a compatible shape. The challenge is *finding compatible elements*, which requires designing a shape matching technique that is fast and robust. Additionally, they cannot guarantee to find a compatible element at every iteration, and elements typically do not fit perfectly with each other or the container boundary. The remedy here is providing more data with increased computation time. However, a large library may not be feasible or artistically desirable. If an artist wants a packing of hand-drawn cats, and a data-driven approach cannot find a good result with ten cat shapes, the artist may not want to draw 100 or 1000 cats to ensure a better fit.

In this thesis, we propose *deformation-driven* methods. Instead of finding compatible elements, we ~~intent to *create compatible elements*~~ through deformation ~~that~~ can adapt to the shapes of neighboring elements and the container boundary. We allow elements to deform in a controlled way, to trade off between the evenness of the element distribution and the deformations of the individual elements. By building an algorithm with a controllable deformation model at its core, we achieve a more even distribution of negative space, and

we ~~only require~~ a small library of element shapes.

Deformation-driven methods also allow us to work toward a design principle called *uniformity amidst variety* [Hut29, Gom84]. *Uniformity* aims for an overall unity of design and *variety* seeks to break up the monotony of pure repetition. We can achieve a degree of uniformity by using repeated copies of a small library of elements, but balance that uniformity with variety by deforming those elements. We believe that there is a value in deformation that can generate plausible families of related elements from a single input shape.

The perceived packing quality is closely tied to the evenness of negative space. The more the elements interlock, the more even the negative space. We are interested in quantitative measurements of ~~the~~ evenness in order to evaluate and compare packing algorithms. In this thesis we discuss several possible measurements of evenness based on methods from spatial statistics.

In this thesis, ~~our contribution is developing three deformation-driven packing methods and 2D packing quantitative metrics:~~

1. FLOWPAK is a packing method that deforms long thin elements to follow a user-defined vector field (Chapter 3).

2. RepulsionPak is a packing method that utilizes repulsion forces to distribute and deform elements, each is represented as a mass-spring system (Chapter 4).

3. AnimationPak is a method to pack 2D animated elements inside a static 2D container. Each element is an extruded 3D shape in a spacetime domain and we view the animated packing problem as a 3D packing in that domain. (Chapter 5).

4. Quantitative metrics for measuring the evenness of negative space: spherical contact probabilities, histograms of the distance transforms, and the overlap functions (Chapter 6).

Chapter 2

Related Work

2.1 2D Packings

Lloyd's Method: An approach to generate a packing is to start with an initial configuration and iteratively refine it using Lloyd's method to obtain an even distribution of elements. Figure 2.1 shows an illustration of point-based Lloyd's method. Given N point elements, we first compute a *Voronoi diagram* that partitions the plane into N regions such that all points inside a Voronoi cell are closest to its associated point element. Lloyd's method moves every point element to the centroid of its Voronoi cell, then the Voronoi diagram is recomputed. The process is repeated until the distribution is even, that means all the point elements are located at the centroids of the Voronoi cells. The final structure is called a *centroidal Voronoi diagram* (CVD).

Secord [Sec02] computed a weighted CVD so that the resulting point element distribution resembles a stippling artwork. His method incorporates the gradient of an input image into the centroid calculation, attracting point elements to low-intensity regions.

A standard CVD is generated using Euclidean distance ~~metric~~, but ~~it can be replaced to manipulate the~~ shapes of Voronoi cells. Hausner [Hau01] used Manhattan distance ~~metric so that Voronoi cells~~ resemble squares instead of hexagons. Each Voronoi cell is then replaced with a square element and the resulting arrangement resembles the appearance of traditional mosaics (Figure 2.2). However, ~~Hausner's approach can only position~~ the square elements, and ~~it~~ must incorporate a vector field to modify the orientation of the metric and rotate the square elements. Additionally, the approach is only suitable for square elements as more complicated shapes such as long rectangles would have severe

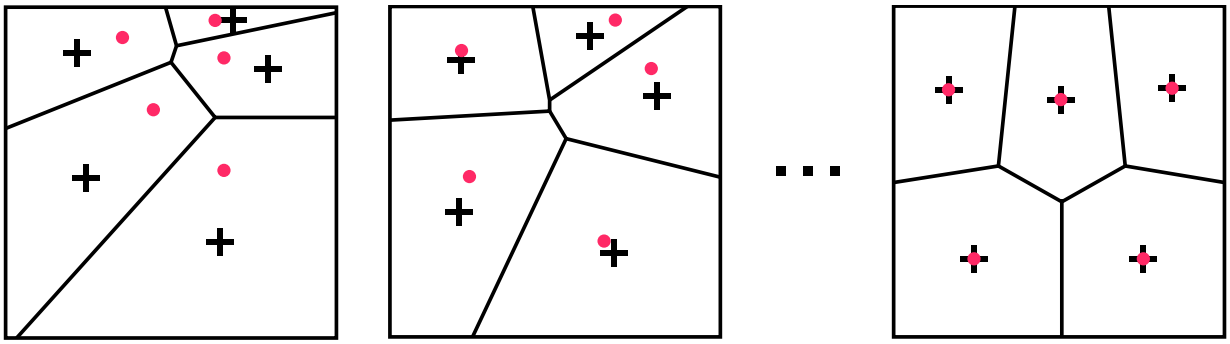


Figure 2.1: An illustration of point-based Lloyd's method. A Voronoi diagram is generated from the input point elements (drawn as red dots). We then move the point elements to the centroids of Voronoi cells (drawn as plus signs). The process is repeated until convergence, producing a centroidal Voronoi diagram. Figure source is Wikipedia, drawn by Dominik Moritz under CC0 1.0.

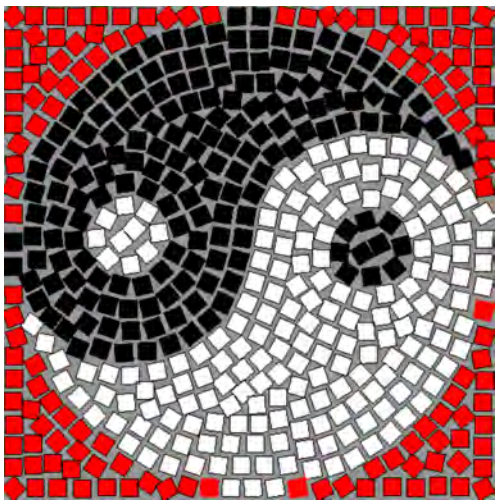


Figure 2.2: Mosaics of square elements generated using Lloyd's method with Manhattan distance [Hau01]

overlaps. More recently, Javid et al. [JDM19] constructed a metric that incorporates a spatial distance, a color distance, and an elongation factor. Their resulting Voronoi cells have elongated shapes, resembling pebble-like mosaics.

Hiller et al. [HHD03] improved upon the Lloyd's method to generate *centroidal area Voronoi diagram* (CAVD), a variant of CVD that is a distribution of polygonal elements. This new extension computes the main inertia axis for each Voronoi cell so that its element can be rotated to get better alignment with the Voronoi cell boundary. In follow up work, Smith et al. [SLK05] generated temporally coherent animated mosaics by utilizing CAVD. Dalal et al. [DKLS06] used an FFT-based image correlation to reposition and rotate elements so that they achieve the best alignments with their Voronoi cell boundaries, which could be seen as making more effective use of negative space, and permitting non-convex elements to interlock more than they did in earlier methods.

Data-Driven Method: A different approach to generate packings is to use a shape matching algorithm (Figure 2.3). This is usually done by placing an element one by one until the container area is filled. For each step, a shape matching algorithm selects an element that has the best compatibility with the previously placed elements or the container boundary. Jigsaw Image Mosaics [KP02] uses a geometric hashing to find a compatible element. JIM places elements using a greedy approach, but is able to backtrack if a previous configuration is more optimal. Pyramid of Arclength Descriptor [KSH+16] is a partial shape matching algorithm that can match subsets of element boundaries.

It is also possible to initially partition the container into smaller segments then independently replace each segment with a matching element. This is desirable if the container has colors or salient parts that need to be emphasized. Huang et al. [HZZ11] produced Arcimboldo-like collages by arranging cutout images collected from the internet. Their container is a larger cutout image which is partitioned into parts using an image segmentation algorithm. Each part is then replaced with a smaller cutout image that has a similar shape and color.

All these data-driven methods require an element library that is big enough, the more elements in the library will increase their chance to find a compatible element boundary. However, bigger database means increased computational time and collecting a large number of elements is also not always feasible.

Deformation-Driven Method: Instead of finding compatible elements, a deformation-driven method allows elements to create compatibilities. As an advantage, a deformation-driven method can rely on a small size element library. Xu and Kaplan [XK07] and Zou et al. [ZCR+16] constructed *calligrams* by filling a container with a small number of deformed letters composing one or two words (Figure 2.4). Their goal was to balance between filling

(a) (b)

Figure 2.3: Data-driven methods: (a) Jigsaw Image Mosaics [KP02]. (b) Pyramid of Arclength Descriptor [KSH+16].

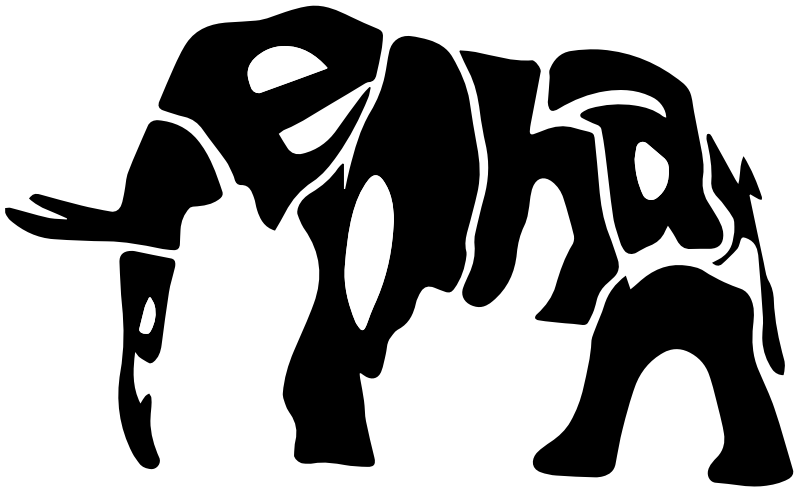


Figure 2.4: A deformation-driven packing of a calligraphic "elephant" [XK07].

the container and preserving readability. Peng et al. [PYW14] computed layouts by packing and deforming simple polygons and polyominoes, although their method cannot handle more complicated shapes. Abdrashitov et al. [AGYS14] developed a sketch-based interface where an artist can draw curves where square-like elements are placed along it create a mosaic arrangement. After the square elements are frozen in place, they are deformed to eliminate overlap, to create shape variations, and to even out the negative space.

2.2 Tilings

As elements reach perfect compatibility, a packing turns to a tiling, ~~see~~ Figure 2.5 ~~for an example~~. The element boundaries interlock, leaving no negative space. Kaplan and Salesin [KS00, KS04] deformed one or two input elements into ~~ones~~ that can tile the plane. Given an input element E, they performed a search in a parameterized tiling space to find its deformed variant T. Due to high frequency deformation, T is often perceptually unrecognizable from its silhouette unless an interior picture is added. Their method can fail to find T if E has deep concavity, for example, a "G" shape. The tile T also cannot fill a container due to its highly structured boundary.

Lin et al. [LML+18] generated ~~ground-reversal~~ compositions that resemble Escher's Sky and Water I, where elements serve as both positive and negative space depending on the viewer's perception. ~~Given t~~wo elements S_1 and S_2~~, each is~~ placed on ~~the~~ opposite side of a rectangular canvas. They are then copied, arranged, and deformed to interlock each other. The farther a copy from the original element, the more deformed it is, creating a "fading effect", S_1 and S_2 spatially fade to nothingness. This work is data-driven, the user needs to provide S_1, and the method tries to find a compatible element S_2 from a library.

2.3 Discrete Texture Synthesis

Some past work has sought to adapt example-based texture synthesis methods from raster images to vector graphics, producing distributions of rigidly transformed elements that mimic the statistics of an *exemplar*. An element is represented as a single point, which is adequate for small near-convex elements. Barla et al. [BBT+06] and Ijiri et al. [IMIM08] used a growth model that copies small neighborhoods from the exemplar into a larger output texture. Hurtut et al. [HLT+09] developed a statistical sampling method based on multitype point processes. AlMeraj et al. [AKA13a] stamped out copies of the exemplar and discarded overlapping elements. Loi et al. [LHVT17] developed a texture synthesis

Figure 2.5: A tiling of cats [KS00], elements perfectly interlock each other and the arrangement does not have negative space.

method that can specify global arrangements, local arrangements, or a blend of multiple arrangements. These techniques are all concerned with replicating the uneven element distribution in the exemplar, without regard for negative space.

For larger concave elements, a single point representation is not enough. Ma et al. [MWT11] used sample-based representation that is created by generating a sparse set of points inside an element. They later distributed these points using a neighborhood metric and an iterative optimization. Unlike previous work, they were able synthesize textures with long deformable elements, for example, spaghetti. Ma et al. later extended their work to accept animated elements [MWLT13], where each sample point has a spatial and time position, turning the problem to spacetime texture synthesis. More recently, Hsu et al. [HWYZ20] adapted the sample-based representation into an interactive ~~app~~ where an artist can initially distribute elements by drawing strokes which are then optimized using a Lloyd-like optimization.

2.4 3D Packings

Collages are arrangements of overlapping elements, similar to portrait paintings by Giuseppe Arcimboldo. Gal et al. [GSP+07] presented a method for constructing 3D collages (Figure 2.6a). They filled a 3D container with overlapping 3D elements using a greedy approach

and a partial shape matching algorithm. Theobalt et al. [TRdAS07] developed a method to generate animated 3D collages. They segmented an animated container into smaller rigid parts, ~~each is replaced with a matching element~~. The final ~~result of~~ animated collage consists of rigid motions of elements. Huang et al. [HWFL14] designed a method to generate mechanical collages, such as giant robots. All these collage methods require a 3D shape database ~~so they are considered as data-driven~~.

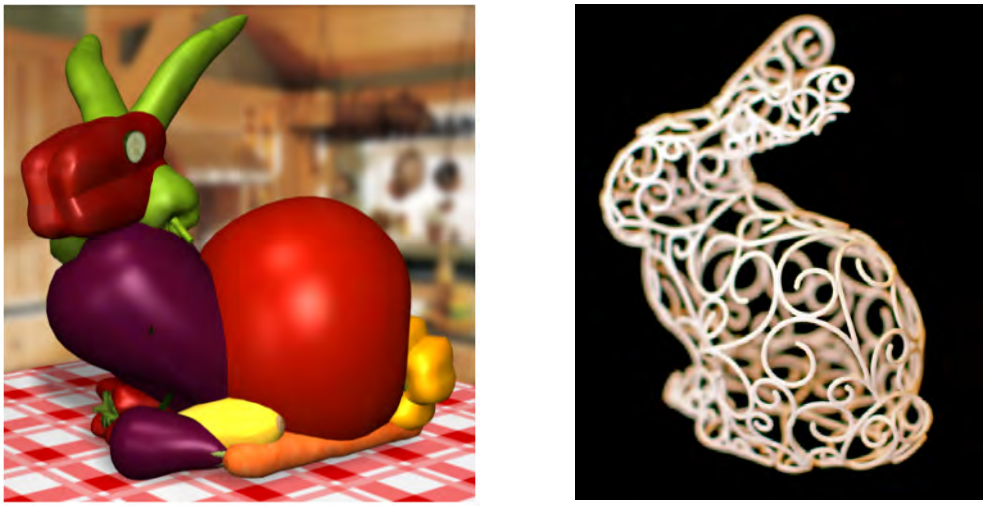
The cutting and packing problem (C&P) is defined as cutting a large object into smaller parts which are then packed inside a container. C&P is popular in manufacturing and 3D printing because objects can be produced with less waste material and packed into a smaller box. A good cutting process is critical in C&P, if it can decompose the input object into simpler parts, then the packing process can be easier. Chernov et al. [CSR10] proposed a method to decompose a large object into smaller ~~phi objects~~ ~~which~~ can be packed more efficiently. A phi object is defined as a shape whose surface boundaries are flat, spherical, cylindrical, or conical. Vanek et al. [VGB+14] introduced PackMerger, a method to pack thin shells which can be assembled together into a larger watertight object. In follow up work, Chen et al. [CZL+15] introduced Dapper, a method to cut and pack volumetric printed objects.

In engineering, packings are useful for a number of applications, such as product packaging, circuit designs, or mechanical layouts. This requires elements to be packed without any overlap. Cagan et al. [CSY02] compiled a survey of 3D packing approaches such as gradient methods, simulated annealing, and genetic algorithms. Byholm et al. [BTW09] developed a method to pack voxelized elements, which are computationally easier for collision detection. Ma et al. [MCHW18] proposed a heuristic method to pack triangular meshes.

2.5 Packings on Surfaces

Some recent work has explored the elaboration of ornamental patterns on surfaces, under constraints imposed by fabrication, such as connectivity. Chen et al. [CZX+16] developed a method to generate a filigree pattern, which is an arrangement of decorative thin rod elements on a surface. Given an initial random configuration of overlapping elements, their method removes overlaps by either deforming or trimming the rods. In similar work, Zehnder et al. [ZCT16] proposed a semi-automated tool for deforming ornamental curves to cover a surface (Figure 2.6b). They started with an initial configuration of scaled down elements ~~that has no overlaps~~. They later grew the elements and avoided overlaps using curve deformation. Bian et al. [BWL18] used Wang tiles made of element parts to generate

filigree patterns. Martínez et al. [MSS+19] developed a CVD-based method to generate star-shaped tiling patterns that are printed onto tracery sheets. Their method works by constructing a star-shaped metric to manipulate the Voronoi cell shapes.



(a) (b)

Figure 2.6: (a) An Arcimboldo-like 3D packing [GSP+07]. (b) Decorative elements that fill a surface [ZCT16].

Chapter 3

FLOWPAK: Flow-Based Ornamental Element Packing

Figure 3.1: 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 colors in the final rendering were added manually.

3.1 Introduction

FLOWPAK ~~[SKAM17]~~ is a technique for filling a container region with elements that are deformed to communicate a sense of directionality or flow. Elements have designs of simple geometric forms, often stylized flora, spirals, or other abstract shapes. Figure 3.2 shows four examples of these sorts of compositions, which we refer to as ornamental packings.

In studying designs like those in Figure 3.2, we have identified five high-level principles that are important to ~~ornamental packing 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 3.2 exhibit some amount of flow. In the dog, many elements appear to flow outward from the flower in the center 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 [Hut29]. Gombrich also writes eloquently on the role of variation in design [Gom84]. 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 center 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 3.2 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

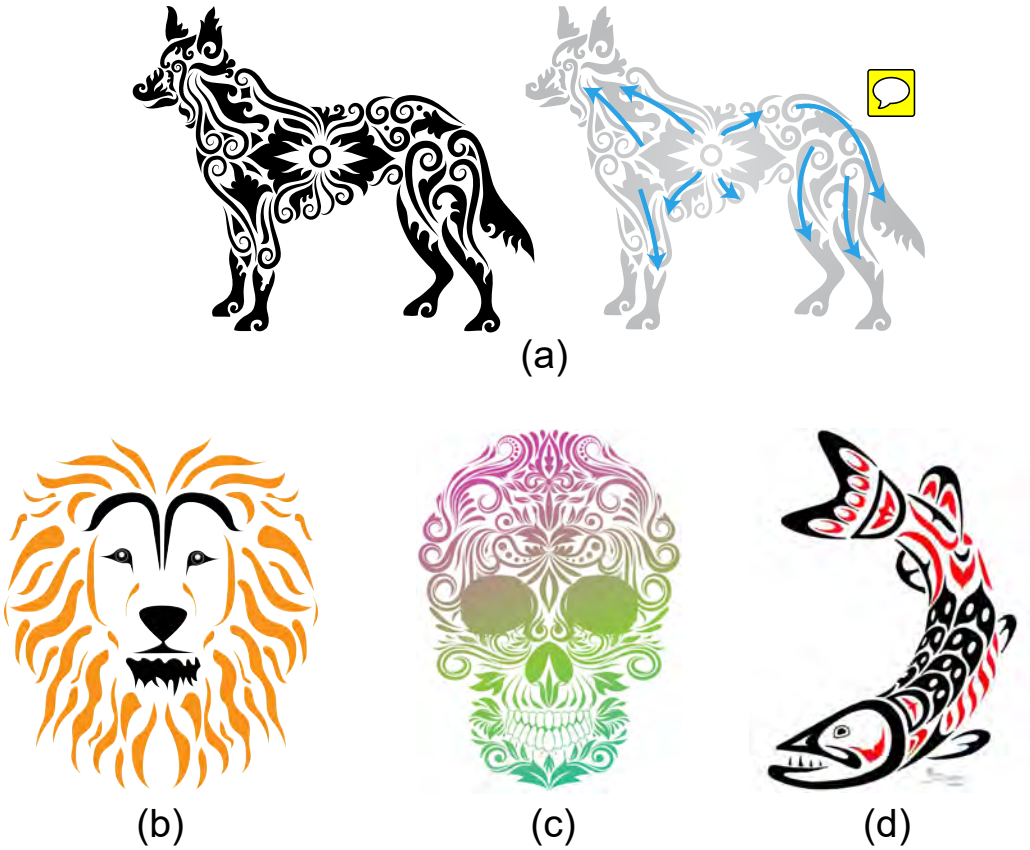


Figure 3.2: Examples of ornamental packings: (a) Dog (by ComicVector703 on Shutterstock), including a visualization showing the flow directions of elements; (b) Lion (from StockUnlimited); (c) Skull (by alitdesign on Shutterstock). (d) Fish in the style of Haida art (by Russ Jones, used with permission);

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.

In FLOWPAK, 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 3.1, aim to obey the design principles articulated above.

3.2 Related Work

Packings: Chapter 2 ~~discusses methods on the generation of packings, or mosaics~~. However, past work is not appropriate for creating designs like those of Figure 3.2. Most techniques pack elements via rigid transformations, leading to high uniformity but insufficient variety. We design FLOWPAK to be a deformation-driven approach so that it can generate plausible families of related decorative elements from a single input shape.

Decorative Ornaments: A distinct category of past research seeks to develop explicit procedural models for authoring decorative patterns. Wong et al. [WZS98] 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. [BvMM11] developed an interactive interface to guide procedural models in generating decorative elements. Guerrero et al. [GBLM16] developed PATEX, a system that preserves high-level geometric relationships like symmetry and repetition while ornamental designs are being edited. Gieseke et al. [GALF17] developed a user interface where an artist can generate a decorative pattern by specifying design principles such as flow, balance, and symmetry. Li et al. [LBMH19] developed a method that can produce a 2D quilting pattern by generating connected stitch paths, each ~~is then~~ decorated by ornamental elements.

Decorative Strokes: FLOWPAK is related to decorative stroke methods since we aim to deform long thin elements. The goal of these methods is to create stylized decorative strokes, and not to fill a container area. Hsu et al. developed Skeletal Strokes [HLW93], a method to warp decorative patterns along a stroke. Asente [Ase10] improved upon Skeletal Strokes by correcting severe deformation on high-curvature strokes. Lu et al. developed DecoBrush [LBW+14] ~~that~~ can join multiple decorative patterns seamlessly.

Vector Field-Guided Compositions: FLOWPAK requires elements to be deformed and oriented according to a vector field. Xu and Mould [XM09] ~~developed a tracing method of particles in a magnetic field to generate decorative curves~~. Li et al. [LBZ+11] developed

a method to decorate a surface using a shape grammar that is guided by a tensor field. Maharik et al. explored Digital Micrography [MBS+11], in which lines of small-scale text are deformed to fit along dense streamlines in a container. These streamlines are traced from a user defined vector field. We take inspiration from Digital Micrography, but seek to place fewer, larger elements taken from a small library of elements and to use shorter, sparser, and less regular streamlines. Lastly, Xu and Mould [XM15] proposed a graph-based tree synthesis method that is guided by vector fields.

Decorative Ornaments on Surfaces: Related methods in fabrication has sought to cover surfaces with arrangements of decorative elements that satisfy manufacturing constraints such as connectivity [CZX+16, ZCT16, BWL18, MSS+19]. Elements must be ~~tightly connected~~ to produce a connected result that will hold together when 3D printed. In the case of FLOWPAK, we seek to distribute disconnected elements inside a container.

3.3 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 3.1. 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.

3.4 Approach

Figure 3.3 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 3.4.1).

2. Analyze the ornamental elements, creating a shape descriptor for each (Section 3.4.2).

3. Use the direction guides to fill each target container with a vector field then trace streamlines (Section 3.4.3).

4. Divide the target containers into blobs around the streamlines (Section 3.4.4).

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 3.4.5).

6. Iteratively refine the placement to eliminate empty areas and make the spacing more even (Section 3.4.6).

3.4.1 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.

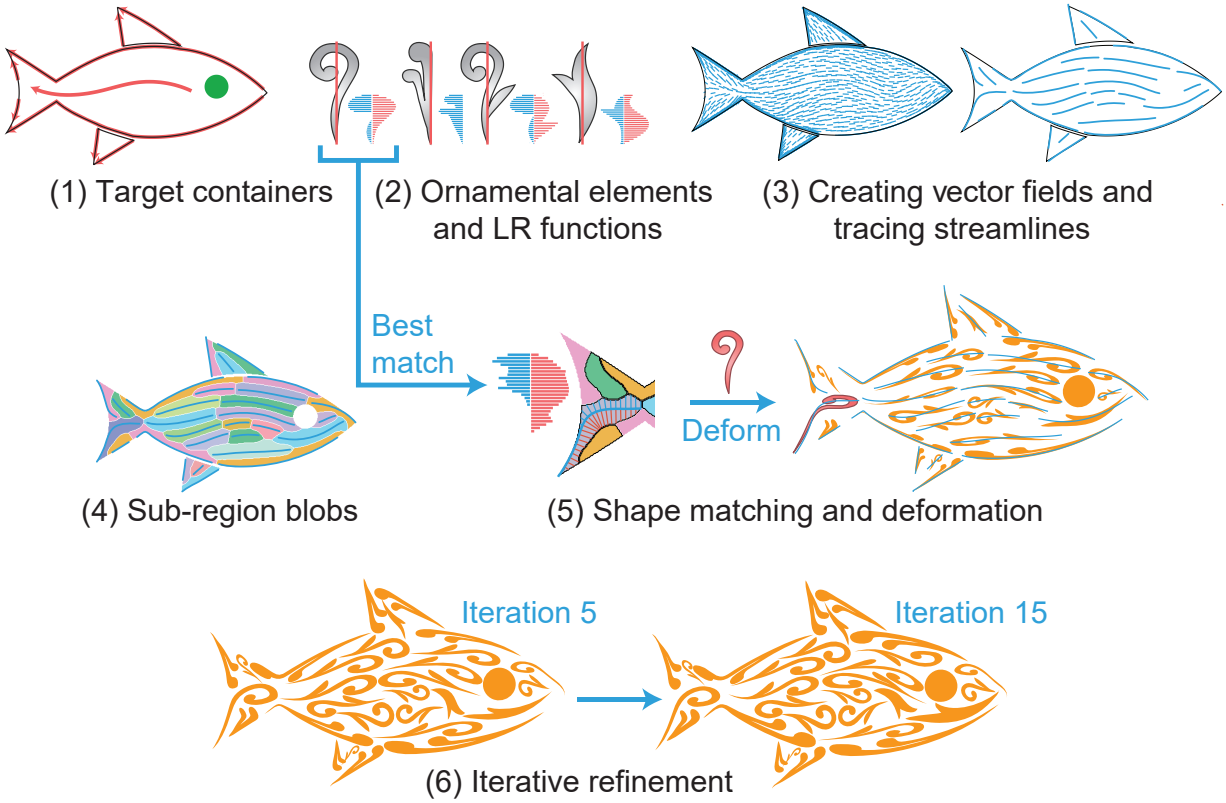


Figure 3.3: A visualization of the steps in our ornamental packing algorithm. The input containers are shown as three black outlines in (a): the body and two fins. They are annotated with directional guides in red and fixed elements (in this case, the eye) in green. The steps in the algorithm follow the description at the beginning of Section 3.4.

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.

3.4.2 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 3.4.5) using a simple skeletal stroke

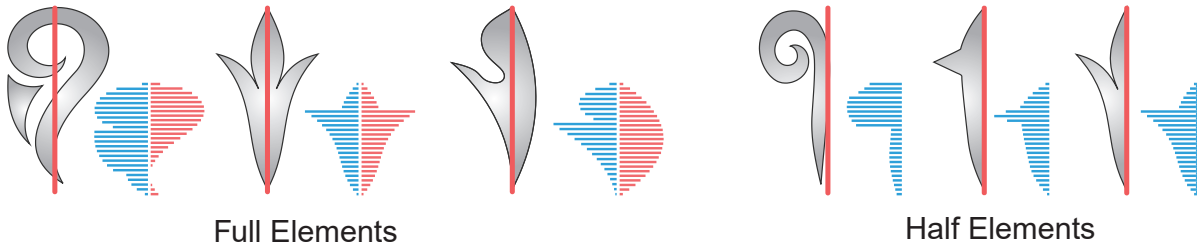


Figure 3.4: Ornamental elements and their LR functions. Full elements have non-empty left and right sides, while half elements have only one non-empty side. We normalize the LR functions to a unit square.

algorithm [HLW93], 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 3.4 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 3.4.5 to choose which element to place in a particular location. Inspired by the work of Gal et al. [GSCO07], 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 3.4.4), in order to achieve the best possible fit.

Intuitively, LR functions give an approximate area an ornamental element can claim. Figure 3.4 shows elements with their left values in blue and their right values in red. We have found that $n\_f = 100$ gives sufficient granularity for our algorithm.

3.4.3 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 $D = \{d_1, d_2, ..., d_{n\_d}\}$ 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[PZ07]. Note that, as shown in Step 3 of Figure 3.3, 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:

s_gap is the desired space between streamlines;
s_max is the maximum desired streamline length; and
s_min is the minimum desired streamline length.

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

We adapt the streamline tracing algorithm of Jobard and Lefer [JL97]. First we generate a set of potential seed points $P = \{\boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_{n\_p}}\}$ 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 *s_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 *s_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 *s_stop* distance test combined with the *s_min*

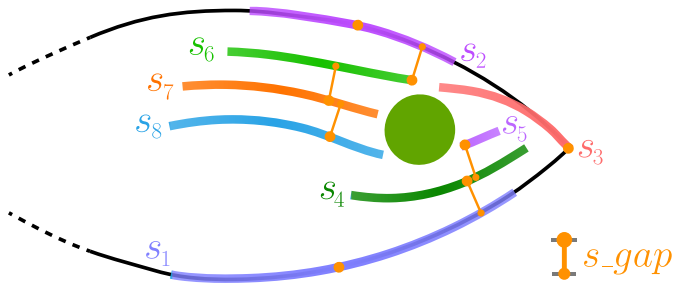


Figure 3.5: The streamline tracing process. The first streamline s_1 always begins on a directional guide or the container boundary. Subsequent streamlines begin on the container boundary, a directional guide, or at a point that is *s_gap* away from a previous streamline.

length test imply that many attempts to form streamlines will stop immediately, especially as the container fills with streamlines.

Figure 3.5 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, ..., 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 s_gap away from s and
      add them to P.
    SORT(P).
  end if
end while
```

3.4.4 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 [OS88]. 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 3.3 shows the blobs of the fish.

We then compute an LR function for each blob as described in Section 3.4.2, 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.

3.4.5 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

$$\sum_{i=1}^{N}(\alpha_{li} - \beta_{li})^2 + \sum_{i=1}^{N}(\alpha_{ri} - \beta_{ri})^2 \tag{3.1}$$

where

α_l is the element left function;
α_r is the element right function;
β_l is the blob left function; and
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.

We investigated alternatives for shape matching, using an approach discussed by Gal et al. [GSP+07] that tries to fill a sub-region blob as much as possible, with heavy penalties

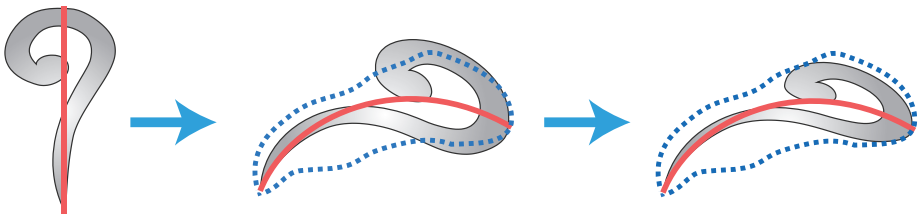


Figure 3.6: The deformation process bends the element along the streamline and scales it to fit inside the blob.

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 [HLW93]. We uniformly scale the element's width to make it as wide as possible while still staying inside the blob (Figure 3.6).

3.4.6 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 3.7. 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 3.9.

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

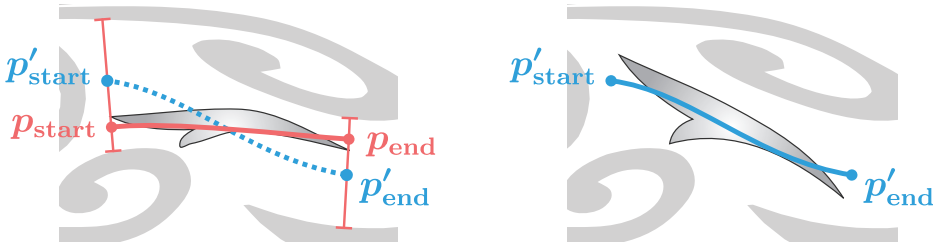


Figure 3.7: Streamline shifting. We move the streamline's start and end points along perpendiculars, stopping before intersecting neighboring elements.

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 3.7). It also gives the streamline room to extend if its endpoints were too close to boundaries of other placed elements.

Given the endpoints $\boldsymbol{p}_{\textbf{start}}$ and $\boldsymbol{p}_{\textbf{end}}$, we calculate new endpoints $\boldsymbol{p}'_{\textbf{start}}$ and $\boldsymbol{p}'_{\textbf{end}}$. 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 3.4.1).

We replace the streamline with a path joining $\boldsymbol{p}'_{\textbf{start}}$ and $\boldsymbol{p}'_{\textbf{end}}$. Our goal is to create a path that is smooth and does not deviate too much from the vector field. We calculate the

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: α as the growth tolerance
Input: t as the number of iterations

```
for t times do
    Sort E, S, and B by 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 AREA(e'_i) × α > AREA(e_i) then
            s_i ← s'_i
            b_i ← b'_i
            e_i ← e'_i
        end if
    end for
    Sort E, S, and B by 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 ← b'_i
        if AREA(e'_i) × α > AREA(e_i) then
            s_i ← s'_i
            e_i ← e'_i
        end if
    end for
end for
```

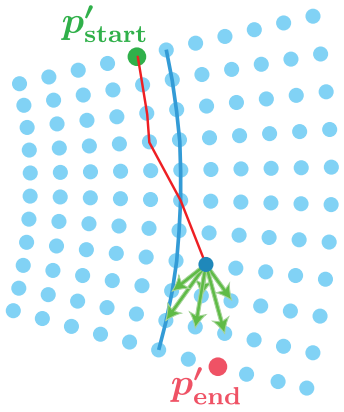


Figure 3.8: Tracing a shortest path using Dijkstra's algorithm. We generate the blue nodes by resampling and offsetting the blue streamline. The search directions at a node are shown with green arrows.

shifted streamline by performing Dijkstra's algorithm on a non-rectangular graph that respects the vector field (Figure 3.8), using a method similar to one by Xu and Mould [XM15] 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 $\boldsymbol{n_a}$ to $\boldsymbol{n_b}$ is

$$w = w_f(1 - f^p) + w_d\, D(s_i, \boldsymbol{n_a}) \tag{3.2}$$

where

f	is $(\boldsymbol{n_b} - \boldsymbol{n_a}) \cdot \boldsymbol{v}$;
$\boldsymbol{v}$	is a sampled vector of the vector field;
s_i	is the original streamline; and
$D()$	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 [Sch90] 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 3.9). To enforce gradual growth, the offset cannot be larger than g_{limit}, where $g_{\text{limit}} = 0.005\, input\_size$.

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 3.10. 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 3.4.5) automatically choses the best rotation. It can also choose to reflect the element across its spine, along its spine, or both (Figure 3.11).

3.5 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 [Joh14] 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 colors 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 such as uniform grids or 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.

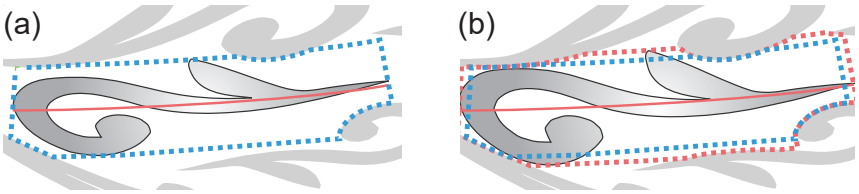


Figure 3.9: (a) An element with its sub-region blob shown in dashed blue line. Note that any blob is constrained by the neighboring elements. (b) The dashed red line is the ~~grown~~ blob, which accommodates an enlarged element.

Figure 3.10: Left: rotated versions of the original element. The best rotation angle is chosen via least squares matching. Right: original, rotated, and enlarged versions of an element.

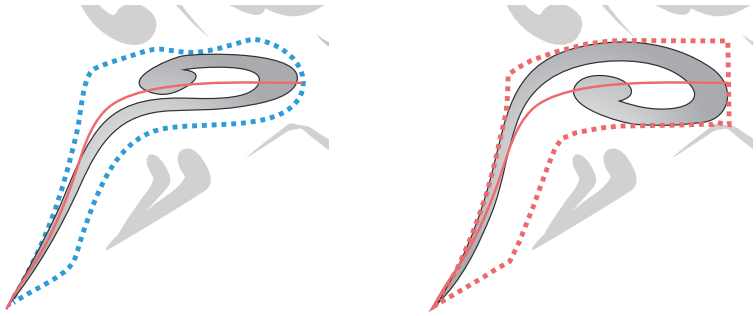


Figure 3.11: An element that reflects across its spine during iterative refinement. LR functions and least squares shape matching allow an element to reflect across its spine, along its spine, or both.

In Figure 3.1, we show two packings of a lion and a unicorn. Each packing is generated with only a set of four elements. The packings demonstrate that FLOWPAK is able to pack and deform elements inside the containers. In Figure 3.12, 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 3.13, 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 3.14 demonstrates a symmetric packing with a fur contour. 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 3.15 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 3.15 for the flower petals in the torso and the paws. Second, in Figure 3.16 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 decorative elements. The result is the bird design shown in Figure 3.17. 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 result.

3.6 Conclusions

We presented FLOWPAK, a method to create ornamental packings in which the elements were oriented and deformed to give a sense of visual flow to the final composition. Our implementation computed a vector field based on user strokes, constructed streamlines that conform to the vector field, and placed an element over each streamline. An iterative refinement process then shifted and stretched the elements to improve the composition.

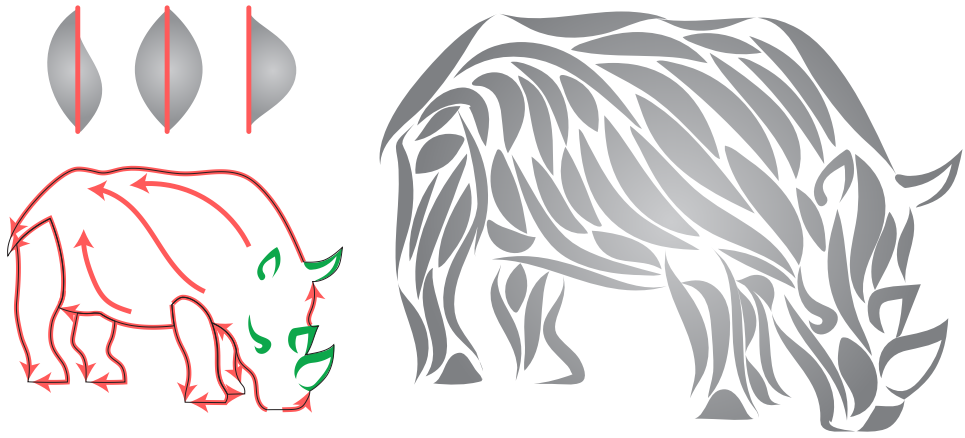


Figure 3.12: A packing of a rhinoceros. Simple teardrop-shaped elements lead to variety in size and curvature.

Figure 3.13: A bear packing with leaf elements. We manually add noise to the elements in the output to create a distressed look.

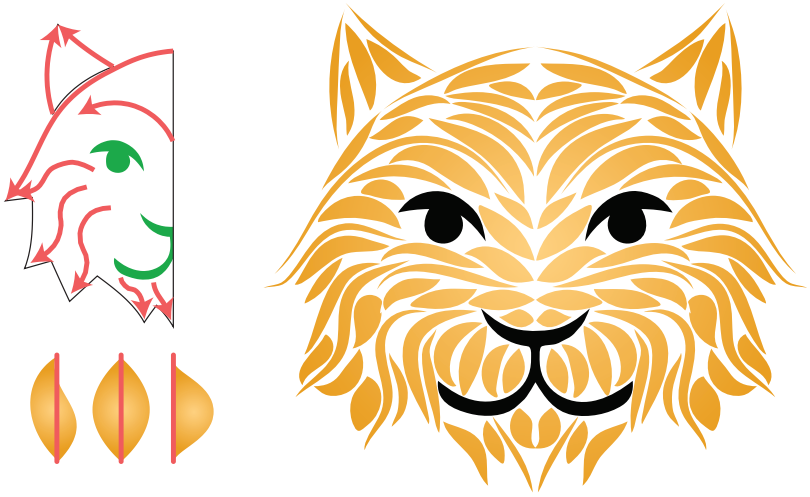


Figure 3.14: A packing with a symmetric layout; we only compute the left half and reflect the result. The elements around the cheeks and the chin are not aligned to the boundary, creating a fur-like effect.

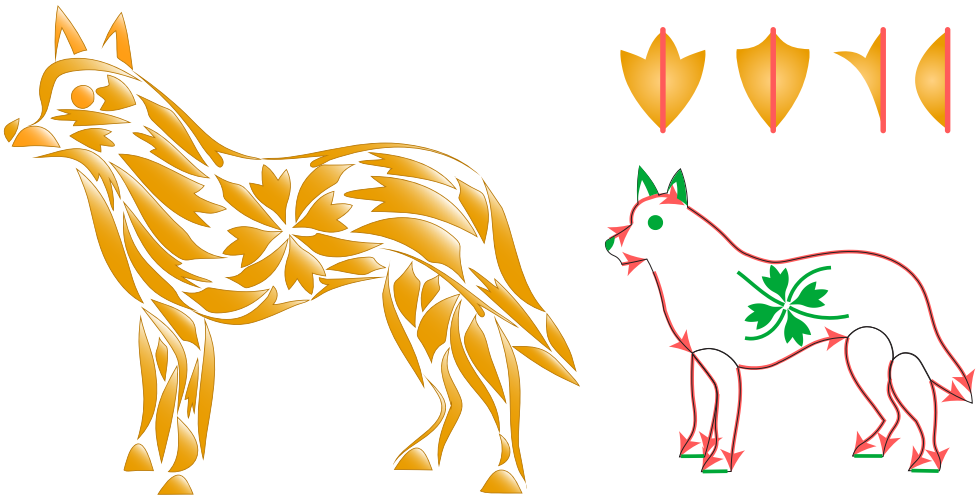


Figure 3.15: A packing of a dog. The fixed elements, shown as green shapes in the diagram, are copied as-is to the output; fixed spines, shown as green paths, force the placement of new elements at the given locations.

Figure 3.16: A packing of the same container as in Figure 3.13. We place longer and sparser elements and synthesize additional forms to fill negative space.

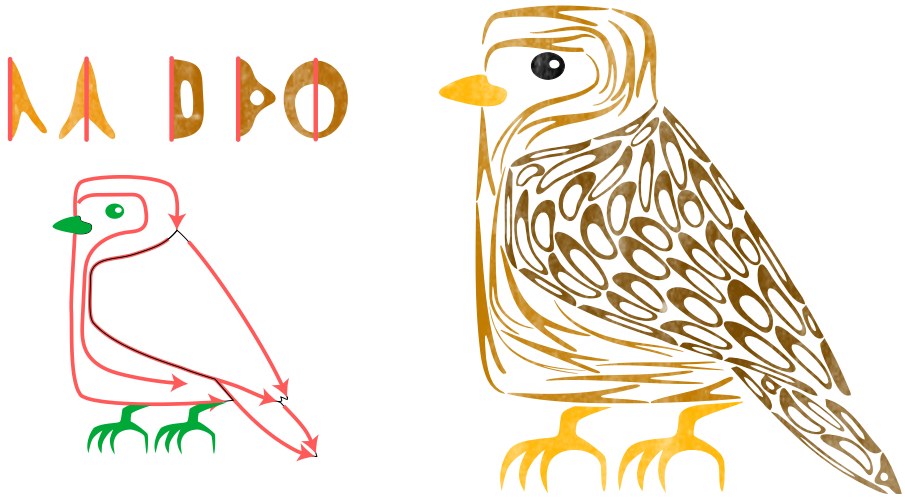


Figure 3.17: A packing of a bird, based on input provided by an artist.

Chapter 4

RepulsionPak: Deformation-Driven Element Packing with Repulsion Forces

Figure 4.1: (a) A packing of six cat elements inside a fish-shaped target container. (b) Input elements. (c) lined up deformed elements in the packing. Controllable deformation and repulsion forces allow the elements to deform, efficiently filling the container and creating a uniform distribution of negative space. We then reduce the remaining negative space by placing smaller cat heads. The gradient fill was added as a post-process.

4.1 Introduction

RepulsionPak is a technique to pack elements using a physical simulation, in which each element is represented by a mass-spring system called an *element mesh*. *Repulsion forces* between neighboring meshes work to even out the negative space, inducing displacements in mesh springs. These displacements translate, rotate, and deform the elements as they gradually adapt to the shapes of their neighbors and the container boundary. To control the amount of deformation, we use *spring forces* within a mesh to preserve element shapes.

Most of the elements in a packing are large shapes of real-world objects like animals, plants, or man-made objects. We refer to these as *primary elements*. An artist distributes primary elements so that they communicate the shape of the container, while attempting as much as possible to ensure an even distribution of negative space. When the primary elements leave behind large pockets of negative space, the artist typically fills those pockets with small *secondary elements*, often simple abstract shapes like circles or triangles. A packing example with primary and secondary elements is shown in Figure 4.2.

Unlike FLOWPAK ~~that~~ is optimized to deform long thin elements, we design RepulsionPak to support arbitrary shaped elements. RepulsionPak is also intended to fulfill three of five design principles: balance, uniformity amidst variety, and boundaries (Section 3.). We achieve a more even distribution of negative space by building an algorithm with a controllable deformation model at its core. We are able to use repeated copies taken from a small library of elements but their final shapes are varied thanks to element deformation. We also show that by carefully placing elements in the initial distribution using a shape matching algorithm, the shape of the container boundary can be ~~more~~ emphasized.

4.2 Related Work

Lloyd's Method: Variants of Lloyd's method have been proposed by Hausner [Hau01], Hiller et al. [HHD03], Smith et al. [SLK05], and Dalal et al. [DKLS06]. Although these approaches can generate appealing results, they can only accept convex or near convex elements. RepulsionPak can be categorized as a variant of point-based Lloyd's method without explicitly computing a Voronoi diagram. Forces do not act directly onto elements, but their mesh vertices. During the simulation, these mesh vertices are spread apart but still connected using elastic springs, effectively creating an element representation that can be packed and deformed.

Data-driven Methods: Data-driven methods include Image Mosaics (JIM) [KP02], the Pyramid of Arclength Descriptor (PAD) [KSH+16]. JIM is able to pack nearly-convex

Figure 4.2: A packing created by Balabolka. (a) The packing with primary elements only. (b) Secondary elements are added.

elements tightly by placing one element at a time and backtracking as needed. As a shape descriptor, PAD can find new elements that partially match existing element boundaries as a container ~~was~~ being filled. Both methods permit some elements to overlap. While they can both correct gaps and overlaps using deformation, the deformation is applied locally near edges in a post-processing step after elements ~~were~~ frozen in place. 2D Arcimboldo Collage [HZZ11] and 3D Arcimboldo Collage [GSP+07] are also data-driven, but they permit large overlaps ~~to resemble~~ the aesthetic style of Arcimboldo's artwork.

Deformation-Driven Methods: Xu and Kaplan [XK07] and Zou et al. [ZCR+16] deformed letters of one or two words to fit into a container. Because the order of the letterforms was defined by the text, their solutions require significant distortion of the individual letters. Peng et al. [PYW14] computed layouts by packing and deforming simple polygons and polyominoes. Their method cannot handle more complicated shapes, making it unsuitable for our style of packings.

Zehnder et al. [ZCT16] proposed a method to cover 3D surfaces with deformed ornamental elastic curves. Our method has some similarities to theirs in that both start with scaled-down copies of elements and grow them, but the growth process is quite different. Unlike their approach, our elements exert forces on each other throughout the growth process, allowing them a greater opportunity to translate, rotate, and deform in search of more even negative space. Furthermore, the goal of their work (3D fabrication) is quite different from ours (2D ornamentation) and our results appear qualitatively different.

Physically-Based Methods for Deformable Objects: Contemporary research has yielded many more sophisticated physical simulation methods. the Finite Element Method (FEM) is accurate and realistic but notoriously slow. Position Based Dynamics (PBD) [MHHR07] is simple, robust, and efficient ~~to be used~~ in real-time applications, but

suffers from inaccuracy. More recently, Projective Dynamics [BML+14] bridges a gap between FEM and PBD. Projective dynamics uses a variational implicit time integration and optimize by alternating a local constraint step and a global solve step.

Sample-Based Representation: A few of texture synthesis methods represent an element as a collection of sparse points [MWT11, MWLT13, HWYZ20]. This representation allows them to synthesize textures with elements that have large concave shapes. Our RepulsionPak has a similar approach, but we represent an element with denser vertices that are connected with elastic springs.

Curve Simulation: Pedersen and Singh [PS06] grew curves to create organic labyrinthine paths. Their algorithm is related to ours by the use of repulsion forces to maintain even spacing and parallel segments. More recently, Yu et al. [YSC20] used tangent-point energy to pack curves inside a container while avoiding self-intersections.

4.3 System Overview

Our system requires three main pieces of input:

- A library of primary and optional secondary elements, such as those shown in Figure 4.3a. Each element is a collection of open or closed polygonal paths—any curves must be flattened ahead of time.
- One or more closed polygonal target containers, such as the heart in Figure 4.3b. Target containers can optionally have internal holes.
- The desired element spacing distance $d_{\text{gap}} > 0$.

RepulsionPak starts by preprocessing the elements, creating additional space around each to enforce the spacing distance, and fitting a triangle mesh over each element. The containers are then seeded with randomly placed copies of scaled-down elements (Section 4.4).

RepulsionPak then performs a physics simulation on the meshes, making them simultaneously grow and repel each other. As a proof of concept, we implement a spring-based simulation, which yields satisfactory results despite its simplicity; many alternatives are possible (see Section 4.9). Forces in the simulation push mesh vertices away from vertices in other meshes, attempt to keep the meshes from undergoing excessive deformation, and resolve places where meshes overlap or vertices move outside target containers (Section 4.5).

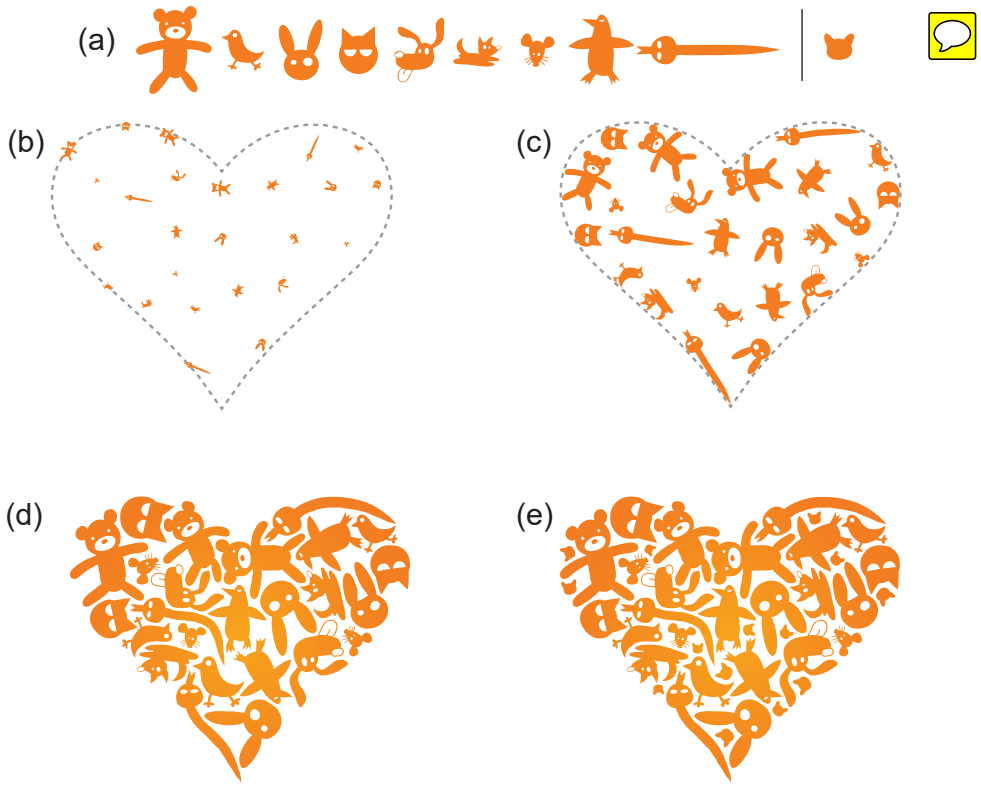


Figure 4.3: The creation of a packing using RepulsionPak. (a) A library of elements, comprising nine primary elements and a single secondary element. (b) A target container with the initial distribution of scaled-down elements. (c) The simulation in progress, showing the elements growing, translating, rotating, and deforming. (d) The resulting packing of primary elements. (e) The final result, after adding secondary elements and allowing them to grow. Figure 4.4 shows the deformations of some of the elements.

After each iteration of the simulation, meshes grow into adjacent space if possible, so that they gradually consume the negative space in the container.

The simulation concludes either when the elements occupy a sufficient proportion of the container area, or when some number of simulation steps fail to significantly reduce the negative space (Section 4.5.2).

An optional second simulation further reduces and evens out the negative space. It begins by placing small secondary elements in large pockets of negative space. This simulation is the same as the first, except that vertices of primary element meshes are not allowed to move (Section 4.6).

Final SVG output is created by using barycentric coordinates to map each element's

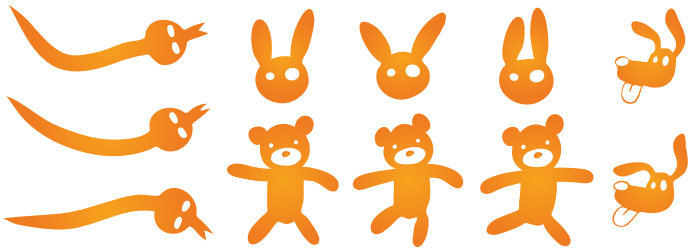


Figure 4.4: Some of the elements in the final packing in Figure 4.3, showing the effect of deformation in our simulation.

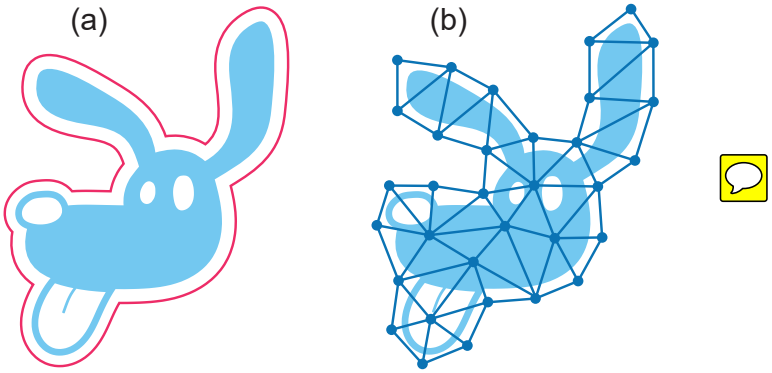


Figure 4.5: An illustration of element discretization for preprocessing. (a) An element with its boundary displaced to create a skin, drawn in red. (b) A triangle mesh with boundary vertices on the skin.

paths from the element's initial mesh into the deformed mesh produced through simulation.

4.4 Preprocessing

The *skin* of an element is a simple closed polygon that fully encloses it, as in the red shape in Figure 4.5(a). We generate the skin by offsetting an element's boundary outward by $d_{\text{gap}}/2$. The simulation aims to produce an approximate tessellation of the target container by deformed skins, thereby achieving the desired element spacing and suppressing overlaps.

We triangulate the element skin to obtain a triangle mesh. To create the mesh we uniformly sample the skin polygon s, with samples spaced apart by distance d_{gap}, to obtain a simpler polygon s' (the outer boundary of the mesh). We then construct a Delaunay

triangulation of s'. The vertices and edges of this mesh become unit masses and longitudinal springs in a physical simulation, allowing elements to deform in response to their neighbors. We also add extra edges to prevent folding and self-intersections during simulation (Figure 4.6b). First, if two triangles ABC and BCD share an edge BC, then we add a *shear edge* connecting A and D. Second, we triangulate the negative space inside the convex hull of the original Delaunay triangulation, and create new *negative space edges* corresponding to the newly created triangulation edges. These negative space edges are used exclusively for internal bracing. The element's concavities can still be occupied by its neighbors. The construction of negative space edges is a simpler variant of Air Meshes [MCKM15], a technique to detect and resolve collisions.

Due to discretization, a low mesh resolution does not guarantee a separation of d_{gap}. Increasing the mesh resolution produces a more precise result at the expense of greater running time.

Barycentric Coordinates: The simulation operates on meshes, not element geometry. In the final rendering phase, we will redraw an element relative to a deformed copy of its mesh. To do so, we first re-express every vertex of an element path in terms of the mesh triangles. Every element vertex lies either inside a mesh triangle or just beyond a border edge. We encode each vertex in barycentric coordinates relative to its enclosing or nearest triangle.

Initial Element Placement: We prepare our simulation by randomly placing non-overlapping elements.

1. Generate random points $P = \{\boldsymbol{p}_1, \boldsymbol{p}_2, ..., \boldsymbol{p}_n\}$ inside the target container via blue noise sampling [Bri07]. The user controls the number of points; using more points gives results with smaller elements. We can automatically estimate n by dividing the container area by the desired average area of the element skins.

2. Cycle through the primary elements, assigning each element to a random unused p_i with a random orientation, repeating until every point has an element.

3. Shrink all the elements so that they do not overlap and occupy only a small fraction of the container's area; in our implementation we have found that $5 - 10\%$ of the area gives good results. Making them larger would speed up the simulation but does not allow enough freedom of movement to generate successful packings. Figure 4.3b shows an initial placement.

4.5 Simulation

We design a simulation in which we generate forces that pack and deform elements by transforming their meshes. Let $\boldsymbol{x} = (x, y)$ be a vertex of an element mesh. The total force $\boldsymbol{F}_{\text{total}}$ applied to $\boldsymbol{x}$ is

$$\boldsymbol{F}_{\text{total}} = \boldsymbol{F}_{\text{rpl}} + \boldsymbol{F}_{\text{edg}} + \boldsymbol{F}_{\text{bdr}} + \boldsymbol{F}_{\text{ovr}} + \boldsymbol{F}_{\text{tor}} \tag{4.1}$$

where

$\boldsymbol{F}_{\text{rpl}}$ is the repulsion force;
$\boldsymbol{F}_{\text{edg}}$ is the edge force;
$\boldsymbol{F}_{\text{bdr}}$ is the boundary force;
$\boldsymbol{F}_{\text{ovr}}$ is the overlap force; and
$\boldsymbol{F}_{\text{tor}}$ is the torsional force.

These forces combine with the growth process, described in Section 4.5.1, to completely fill the target container.

Repulsion Force: The repulsion force tries to push element meshes apart when they approach each other, with the goal of making them transform to align their boundaries (Figure 4.6a).

The vertex $\boldsymbol{x}$ will experience an inverse square repulsive force, inspired by Coulomb's law, from all nearby meshes. We use the following formula:

$$\boldsymbol{F}_{\text{rpl}} = k_{\text{rpl}} \sum_{i=1}^{n} \frac{\boldsymbol{u}}{\|\boldsymbol{u}\|} \frac{1}{\varsigma + \|\boldsymbol{u}\|^2} \tag{4.2}$$

where

k_{rpl} is the strength of the repulsion force relative to other forces in the simulation;
n is the number of nearest neighboring meshes to $\boldsymbol{x}$;
$\boldsymbol{x_i}$ is the closest point on the skin of the ith neighbor;
$\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{x_i}$; and
ς is a *soft parameter*; it places an upper bound on the magnitude of $\boldsymbol{F}_{\text{rpl}}$, avoiding explosive instability when $\|\boldsymbol{u}\|$ is very small.

An imbalance in the repulsion forces across a mesh's vertices will naturally induce translation and rotation in meshes, helping their boundaries discover compatible segments and consuming more of the remaining negative space.

If $\boldsymbol{x}$ lies inside of another element's mesh, then the aggregate repulsion force from other neighbors can push $\boldsymbol{x}$ further inside and worsen the overlap. If we discover such an overlap, we set $\boldsymbol{F}_{\text{rpl}}$ to 0 and use the overlap force $\boldsymbol{F}_{\text{ovr}}$, discussed below, to correct it.

Edge Force: A mesh's edges are treated as longitudinal *springs*; displacements of these springs allow a mesh to deform in response to repulsion forces by neighboring meshes. An undeformed element mesh provides the rest lengths for all of its springs; as mesh vertices move relative to each other, the springs attempt to restore these rest lengths. Let $\boldsymbol{x_a}$ and $\boldsymbol{x_b}$ be mesh vertices connected by a spring. We compute the spring force as follows:

$$\boldsymbol{F}_{\text{edg}} = k_{\text{edg}} \frac{\boldsymbol{u}}{\|\boldsymbol{u}\|} s \, (\|\boldsymbol{u}\| - \ell)^2 \tag{4.3}$$

where

k_{edg} is the strength of the edge force relative to other forces;
$\boldsymbol{u} = \boldsymbol{x_b} - \boldsymbol{x_a}$;
ℓ is the rest length of the spring; and
s is +1 or -1, according to whether $(\|\boldsymbol{u}\| - \ell)$ is positive or negative.

We apply $\boldsymbol{F}_{\text{edg}}$ to $\boldsymbol{x_a}$ and $-\boldsymbol{F}_{\text{edg}}$ to $\boldsymbol{x_b}$. The equation is a modification of Hooke's law in which the strength of the force increases quadratically with displacement. This change allows the meshes to resist severe deformations when subjected to powerful forces.

Overlap Force: Occasionally, a vertex $\boldsymbol{x}$ from one mesh can be pushed inside the skin of a neighboring mesh. In such cases, we temporarily disable the repulsion force on this vertex by setting it to 0, and instead apply an overlap force that attempts to eject the intruding vertex. In particular, every mesh triangle having $\boldsymbol{x}$ as a vertex will pull $\boldsymbol{x}$ in the direction of its centroid. The overlap force is thus given by:

$$\boldsymbol{F}_{\text{ovr}} = k_{\text{ovr}} \sum_{i=1}^{n} (\boldsymbol{p_i} - \boldsymbol{x}) \tag{4.4}$$

where

k_{ovr} is the relative strength of the overlap force;
n is the number of mesh triangles that have $\boldsymbol{x}$ as a vertex; and
$\boldsymbol{p_i}$ is the centroid of the ith triangle incident on $\boldsymbol{x}$.

The overlap force is zero for vertices that are not within another mesh.

Boundary Force: The boundary force causes element meshes to stay inside the target container and conform to its boundary. It applies to any vertex $\boldsymbol{x}$ that is outside the

container, and moves the vertex towards the closest point on the container's boundary, by an amount proportional to the distance to the boundary:

$$\boldsymbol{F}_{\text{bdr}} = k_{\text{bdr}}(\boldsymbol{p_b} - \boldsymbol{x}) \tag{4.5}$$

where

k_{bdr} is the relative strength of the boundary force; and
$\boldsymbol{p_b}$ is the closest point on the target container to $\boldsymbol{x}$.

The boundary force is zero for any point inside the container.

Torsional Force: As forces propagate through an element mesh, the aggregate velocity vectors of the vertices can induce a rotation of the entire element. However, some elements may have a preferred orientation, either for aesthetic reasons or because the shape is comprehensible only at certain orientations. We introduce a torsional force that penalizes individual vertices for which the orientation, relative to their element's center of mass, drifts too far from its initial orientation.

Consider a vertex $\boldsymbol{x}$ belonging to an element, and let $\boldsymbol{c}_r$ be the element's center of mass in its undeformed state. We may define the "rest orientation" of $\boldsymbol{x}$ as the orientation of the vector $\boldsymbol{u}_r = \boldsymbol{x} - \boldsymbol{c}_r$. During simulation we compute the current center of mass $\boldsymbol{c}$ of the element and let $\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{c}$. Then the torsional force is

$$\boldsymbol{F}_{\text{tor}} = \begin{cases} k_{\text{tor}}\boldsymbol{u}^{\perp}, & \theta > 0 \\ -k_{\text{tor}}\boldsymbol{u}^{\perp}, & \theta < 0 \end{cases} \tag{4.6}$$

where

k_{tor} is the relative strength of the torsional force;
θ is the signed angle between $\boldsymbol{u}_r$ and $\boldsymbol{u}$; and
$\boldsymbol{u}^{\perp}$ is a unit vector rotated 90° counterclockwise relative to $\boldsymbol{u}$;

Using the equation above, $\boldsymbol{F}_{\text{tor}}$ is always perpendicular to $\boldsymbol{u}$ and the direction of $\boldsymbol{F}_{\text{tor}}$ points to the left or to the right depending on θ. Unlike the first four force types, the torsional force is optional.

Simulation Details: We use explicit Euler integration to simulate the motions of the mesh vertices under the forces described above. Every vertex has a position and a velocity vector; in every iteration, we update velocities using forces, and update positions using velocities. These updates are scaled by a simulation time step Δt_{sim}, typically chosen from the range $[0.01, 0.1]$. A smaller time step results in a more stable simulation at the cost

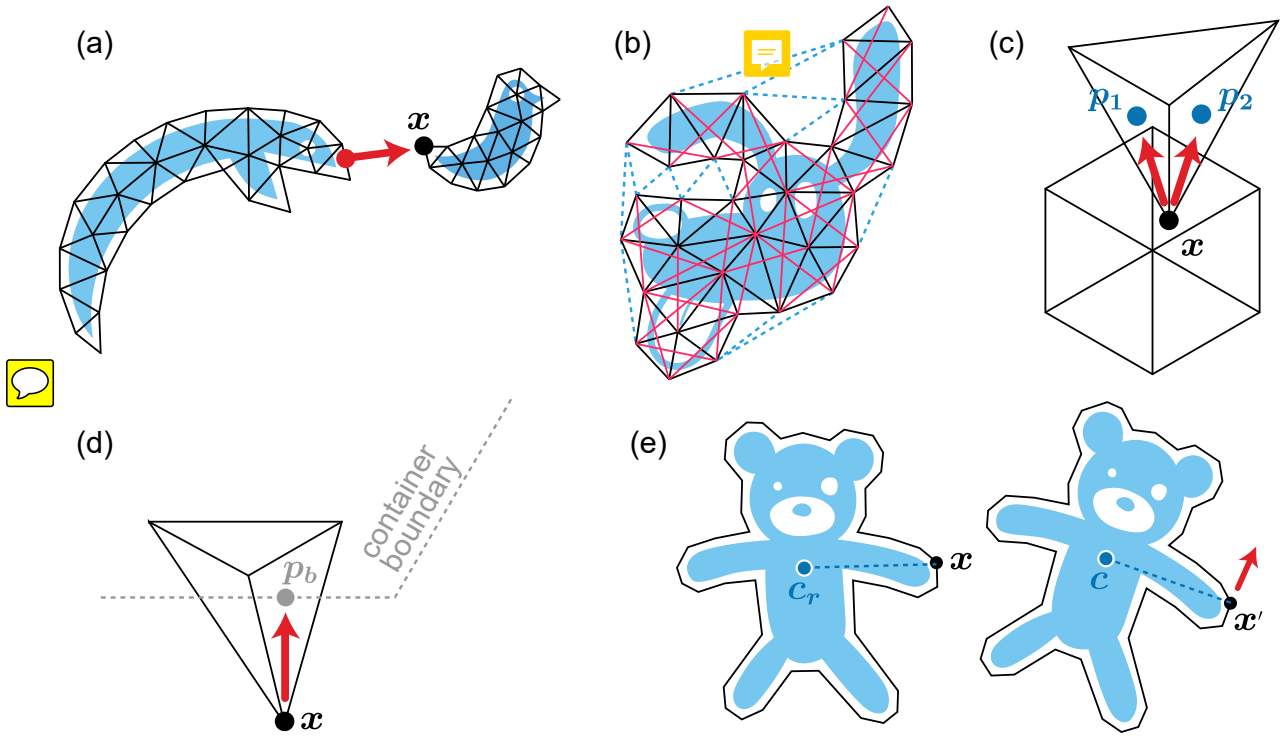


Figure 4.6: Illustrations of the forces in our simulation. **(a) Repulsion force**: The closest point on the snake's mesh repels a vertex $\boldsymbol{x}$ in the bird mesh. **(b) Edge force**: We generate edge forces using edge springs (black), shear springs (red), and negative-space springs (dashed blue). **(c) Overlap force**: Centers of triangles $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ attract a vertex $\boldsymbol{x}$ that lies in the interior of another mesh. **(d) Boundary force**: A vertex $\boldsymbol{x}$ moves toward $\boldsymbol{p_b}$, the closest point on a target container, when it is outside the container. **(e) Torsional force**: We restrict the orientation of the element by generating a torsional force at every boundary vertex $\boldsymbol{x}$ to restore its angular position relative to $\boldsymbol{c}$, the center of mass of the element.

of additional running time. We cap velocities at $5\Delta t_{\text{sim}}$ to dissipate extra energy from the system.

The repulsion and overlap forces rely on nearest-neighbor queries on the set of all vertices. We accelerate these queries by storing vertices in a uniform spatial subdivision grid that covers the container. In our implementation, cell width and height are $6 - 10\%$ of the larger dimension of the grid. We define the neighbors of a vertex $\boldsymbol{x}$ as all vertices in a 3×3 window of cells centered on the cell containing $\boldsymbol{x}$. This approximation ignores the negligible interactions between distant vertices.

The constants k_{rpl}, k_{ovr}, k_{bdr} k_{edg}, and k_{tor} control the relative strengths of the five forces in the simulation. They must also be chosen relative to the time step Δt_{sim} and the overall width and height of the container. We find that our simulation produces satisfactory results when $k_{\text{rpl}} \approx k_{\text{ovr}} \approx k_{\text{bdr}} \geq k_{\text{edg}} > k_{\text{tor}}$. For example, if the container's bounding box is approximately 1000×1000, we set $k_{\text{rpl}} = k_{\text{ovr}} = k_{\text{bdr}} = 80$, and $k_{\text{tor}} = 1$. We set $k_{\text{edg}} = 40$ for edge springs and shear springs, and $k_{\text{edg}} = 10$ for negative-space springs, since weaker negative-space springs are sufficient to avoid self-intersections. We also set $\varsigma = 1$ to avoid explosive repulsion forces. Increasing k_{edg} relative to the other forces suppresses deformation, yielding a close approximation of packing with rigid elements.

4.5.1 Element Growth

RepulsionPak starts with small initial elements to avoid intersections, and gradually enlarges them until they tightly fill the target container. Figure 4.3c-d shows elements growing and gradually consuming negative space. Elements have different intrinsic sizes, which are respected in the initial placement. Because they all grow at roughly the same rate, their relative sizes tend to be maintained.

After each iteration of the physics simulation, the element meshes undergo a growth step. If an element mesh has no vertices that lie inside of neighboring meshes, it is permitted to grow in this iteration. A mesh with overlaps may still grow in subsequent iterations, if local changes to the packing open up more negative space. This approach produces slight variations in skin offsets in the output packing but the effect is negligible.

We implement growth in the context of the physics simulation by scaling the rest lengths of a mesh's springs, allowing it to expand as the simulation progresses. Every mesh M has a counter n_M that records the number of times it has grown. When a mesh is permitted to grow, we add 1 to the counter. Then, if L_i is the rest length of the ith spring in the original undeformed mesh, we increase its rest length to $(1 + n_M k_g \Delta t_{\text{sim}}) L_i$. The constant k_g, usually 0.01 in our system, controls the growth rate.

4.5.2 Stopping Criteria

We choose one of two criteria to stop the simulation. First, the artist specifies the desired positive space ratio at which the simulation immediately terminates. The ratio should not be set too high, and we find that most packings have positive space ratios between 45% and 65% depending on the element shapes. For example, concave elements with long extensions are more difficult to pack, so a lower positive space ratio is recommended to generate a satisfactory packing. Second, we stop the simulation when the element meshes are no longer able to maneuver enough to consume the remaining negative space. After each iteration, we compute an *area fraction* A, defined to be the fraction of the container area taken up by element meshes. We then compute a measurement of the recent change in area fraction in a sliding window that covers the w most recent iterations of the system; we use $w = 100$. If $A_0, \ldots, A_w$ are the area fractions in the $w + 1$ iterations up to the current one, then we define

$$\text{RMS} = \sqrt{\frac{1}{w} \sum_{i=1}^{w} (A_i - A_{i-1})^2} \tag{4.7}$$

We stop iterating when RMS $< \epsilon$, where ϵ is 0.01 in our system.

4.6 Secondary Elements

The iteration process described above can leave behind isolated pockets of empty space, which will be visible in the final composition. We imitate the approach taken by human artists by filling these pockets will small, usually simple secondary elements.

We seed the container with secondary elements by finding points that are far from any existing element mesh. Specifically, we compute a discrete approximation of the distance transform of the negative space. We then create an initial candidate list of all points for which the distance value is above a threshold $d_{\min}$, sorted by decreasing distance. We consider each of these candidates in turn, adding it to a final list of seed locations provided that no previously chosen seed is within distance d_{sep} of the candidate. In our implementation, if the distance transform is computed on a 1000×1000 grid fit to the container's bounds, then we typically set $5 \leq d_{\min} \leq 10$ and $d_{\text{sep}} = 10$.

Next, we assign random secondary elements to these chosen seed points, scaled down as before to avoid overlaps. We then run the simulation and growth process again, but freeze

the primary elements: they exert repulsion forces on secondary elements and can induce overlaps, but primary mesh vertices cannot move. The secondary elements gradually grow to consume some of the remaining negative space until the packing satisfies the same stopping criteria described above.

4.7 Shape Matching

The motions and deformations of elements, as described in the previous sections, give them an opportunity to conform to each other and to the target container. However, in some cases the random seeding may position some elements in such a way that the simulation process will still leave undesirable artifacts. In particular, when a round element is placed near a sharp convex corner of the container, it cannot deform enough to extend into the corner, but cannot yield its position to a pointy element that offers a better fit. In the final packing, the sharp corners of the containers will appear "rounded off". Another problem occurs when a long narrow element is initially placed diagonally across a corner, in which case the simulation pushes the element's middle into the corner, causing extreme deformation.

To overcome these deficiencies, we optionally perform an initial fit-guided placement pass before seeding the rest of the container at random. Here we take inspiration from existing rigid packing algorithms [KSH+16], which are driven primarily by shape compatibility. We use a simplified shape descriptor; we can tolerate a less perfect initial fit, with the expectation that deformation will improve the quality later.

We begin by building shape descriptors for the elements. Each element is first scaled to have area $0.6A_c/n_e$, where A_c is the container area and n_e is the number of elements. This step resizes the element to a rough estimate of its final size in the packing, an approximation that is adequate for our fit-based placement. We then sample the target container and the boundaries of the elements, with adjacent samples separated by distance δ. We set $\delta = 0.002L_s$, where L_s is the side length of the container's bounding square.

We define a local descriptor based on integral of absolute curvature [CFH+09, KSH+16]. Let $P(t)$ be an arclength-parameterized 2D curve, and let $\kappa(t)$ represent the curvature of the curve at $P(t)$. For a given interval $[s, t]$ within the curve's domain, we may define the integral of absolute curvature:

$$\tau(s,t) = \int_s^t |\kappa(x)|dx. \tag{4.8}$$

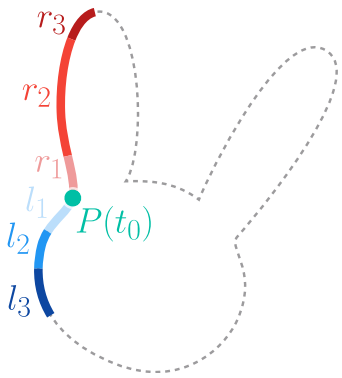


Figure 4.7: An illustration of a local shape descriptor with $n = 3$. These segments have varying arclengths but they all have the same value of τ, the integral of absolute curvature along their lengths.

In practice, we estimate curvature from the discrete sample points using second-order forward finite differences [BL15] and compute τ by summing curvature estimates using the trapezoid rule.

Let an objective τ_{obj} be a positive real number and $P(t_0)$ be a given point on a curve. If τ_{obj} is sufficiently small, then as we traverse the curve on either side of $P(t_0)$ we will eventually reach points $P(t_{-1})$ and $P(t_1)$ such that $\tau(t_{-1}, t_0) = \tau(t_0, t_1) = \tau_{\text{obj}}$. We let $l_1 = t_0 - t_{-1}$ and $r_1 = t_1 - t_0$ be the arclengths that produce these integrals. We can continue this process for any number of steps, walking in both directions along the curve away from previous sample points until we reach τ_{obj}, yielding new arclengths l_k and r_k (see Figure 4.7). Finally, for a given number of steps n we define the shape descriptor at $P(t_0)$ as

$$(l_n, l_{n-1}, \ldots, l_2, l_1, r_1, r_2, \ldots, r_{n-1}, r_n) \tag{4.9}$$

Like PAD [KSH+16], our shape descriptor is rotation invariant. Effectively it is one level of a PAD, which suffices because we do not require scale invariance. Descriptors can be compared using simple Euclidean distance, accelerated by storing them in a k-D tree. In our implementation we set $\tau_{\text{obj}} = 0.001 L_s$; the dependence on L_s makes the measurement robust against changes in absolute container size, and the factor of 0.001 was determined through experimentation. We further choose $n = 5$, yielding a 10-dimensional descriptor. We compute this descriptor for all the container and element samples defined above.

Based on these descriptors, we now use a simple greedy heuristic to identify salient container features where shape matching will be used. Here we restrict our attention to convex protrusions with high curvature, which benefit the most from careful element

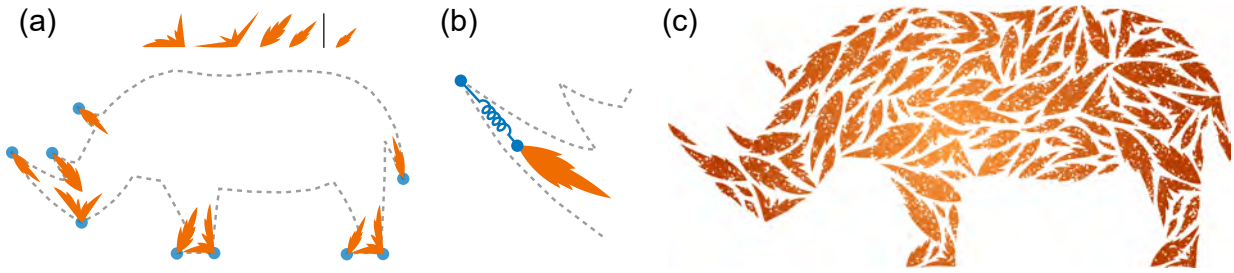


Figure 4.8: A demonstration of shape matching of leaf shapes inside a rhinoceros. (a) We detect nine salient features, namely sharp convex corners, and assign an element to each. (b) A spring holds each element in place. (c) The final result.

placement (See Figure 4.8a). Given a shape descriptor, we define its total length to be $\sum_{i=1}^{n} l_i + r_i$. We iterate over all sample points on the container boundary in increasing order by the total lengths of their descriptors. For each sample point $P(t_0)$, we add it to a list of salient features under two conditions. First, we require that the angle formed by $P(t_0)$ with two samples to either side of it be sufficiently acute; we use a threshold of 0.3 radians to ensure convex-to-convex matching. Second, we ensure that salient features are not too close to each other by requiring that every new sample point be separated by a distance of at least $0.2L_s$ from all previously identified salient features.

Even when an element's descriptor is a close match to a segment of target container, it may still not be safe to place that element at a given location. For example, one part of an element may extend into a corner of the container, while a different part, outside the purview of the local shape descriptor, could protrude outside the container entirely. We augment our descriptor-based fit calculation with an additional score adapted from Gal et al. [GSP+07] in order to ensure a more global element fit. Let $d(\boldsymbol{x})$ be a signed distance function for the container, with negative values inside the container and positive outside. In practice we superimpose a 1000×1000 grid over the container's bounding square and compute a discrete approximation of the distance transform. For an element sample point $\boldsymbol{x}$, we then compute a score

$$q(\boldsymbol{x}) = \begin{cases} e^{-d(\boldsymbol{x})^2/\mu^2\beta}, & \text{if } d(\boldsymbol{x}) < 0 \\ 1, & \text{if } d(\boldsymbol{x}) = 0 \\ -\alpha d(\boldsymbol{x}), & \text{if } d(\boldsymbol{x}) > 0 \end{cases} \tag{4.10}$$

With this scoring function, a sample point that lies on the container boundary will have a score of 1, the maximum for $q(\boldsymbol{x})$. Scores decay exponentially towards 0 for points

farther inside the container. Sample points outside the container are penalized by assigning a negative score proportional to distance.

We then define a quality measurement for an entire element by summing over all of its sample points:

$$Q = \sum_{i=1}^{n_p} q(\boldsymbol{x}_i) \tag{4.11}$$

The parameter α penalizes elements with parts that protrude outside the container; β favors elements with parts close to the target container; $\mu = 0.5\sqrt{2}L_s$ is half of the diagonal of the container's bounding square; and n_p is the number of sample points on the element. In our implementation we choose $\alpha = 1$ and $\beta = 0.001$. For every salient container feature, we obtain the 10 closest descriptors from the k-D tree, and choose one that has the highest Q. Finally, we place the selected element in the container by aligning the endpoints of the element's and container's matching shape descriptors.

The shape matching process yields a set of elements that should be attached to particular locations on the target container. However, during simulation they could wander away from these initial positions, under the influence of the many other forces at play. As shown in Figure 4.8b, we encourage these elements to remain in place by attaching them to the salient container points with additional springs.

4.8 Implementation and Results

Our software was written in C++, and reads in text files describing elements and containers; we prepared these files using Adobe Illustrator. We ran our software on a computer with a 2.4 GHz Intel i7-4700HQ processor and 16 GB of RAM. As a post-process, we optionally read packings back into Illustrator, fit smooth curves to polygonal paths, and apply colors and other visual effects. Table 4.1 shows statistics for the results in this chapter. All results in this chapter use $\Delta t_{\text{sim}} = 0.1$. Torsional forces are used only in Figure 4.10, and shape matching is used only in Figure 4.8.

The supplemental materials include movies that visualize the simulation process[1]. These movies make it clear that elements can jostle each other around, inducing translation, rotation, and deformation throughout the simulation.

The packing in Figure 4.1 uses six cat-shaped primary elements and one secondary cat head. RepulsionPak naturally bends legs and tails to fill the container more evenly.

[1] https://cs.uwaterloo.ca/~radhitya/repulsionpak/supplemental_material.zip

Table 4.1: Data and statistics for the RepulsionPak results. The table shows the numbers of primary and secondary elements (n_p, n_s), the number of vertices (v), the running times of the primary and the secondary simulations (t_p, t_s) in seconds, and the number of iterations (i). In these results, torsional forces are used only in Figure 4.10 and shape matching is used only in Figure 4.8.

Packing	n_p	n_s	v	t_p	t_s	i
Cats (Figure 4.1)	41	69	3598	185	62	8531
Animals (Figure 4.3)	25	14	2412	133	65	16670
Rhino (Figure 4.8)	107	0	4833	237	0	15521
Birds (Figure 4.9a)	43	43	2309	102	54	11571
Bats (Figure 4.9b)	47	22	3048	165	56	13120
Butterflies (Figure 4.9c)	123	135	11916	696	616	14379
Giraffes & Penguins (Figure 4.10)	60	0	2250	163	0	9347
Paisley (Figure 4.11)	162	0	4860	128	0	23040
Circles in Paisleys (Figure 4.11)	144	0	2544	403	0	29584

The animal packing in Figure 4.3 has several elements with limbs (the bear, fox, chick, and penguin), extensions (the dog and bunny ears), and long shapes (the snake). Figure 4.4 highlights the deformations for some of these elements; they are noticeably more deformed than nearly convex elements like the cat and mouse.

The butterfly packing in Figure 4.9 is an attempt to reproduce the visual style of a dense packing (or tessellation), similar to Jigsaw Image Mosaics [KP02] or the "Butterflies in Butterfly" example from the Pyramid of Arclength Descriptor paper [KSH+16]. The target container is made from two regions, one with internal holes. The resulting packing is tight but overlap-free.

The bird packing in Figure 4.9 exhibits significant deformation in the wings and the tails of the birds. In particular, the thin tails of the swallows have some unaesthetic sharp bends. We would like to investigate ways to ensure these bends are smoother.

The packing in Figure 4.10 demonstrates torsional forces that gradually turn the elements upside down. The rhinoceros packing in Figure 4.8 demonstrates initial placement via shape matching. The entire shape matching process took 922 milliseconds with a 4-element library.

We have also experimented with creating tileable packings, as shown in Figure 4.11. We seed a central square with elements, but also place clones of those elements in all the squares of the 8-neighborhood around the center. These clones track the transformations

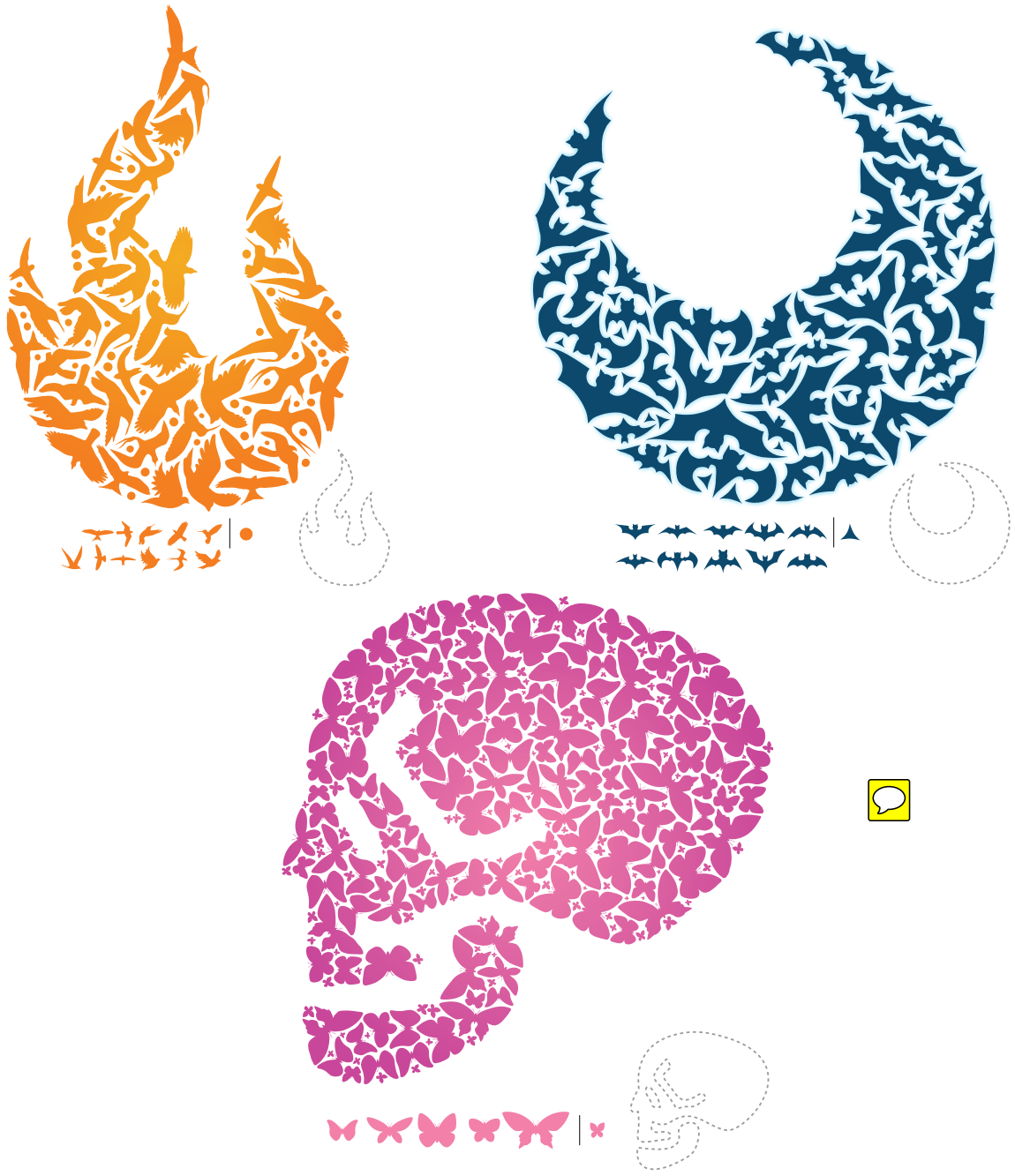


Figure 4.9: Three packings created using RepulsionPak: Birds, Bats, and Butterflies. The results are visually appealing overall, though some birds' tails suffer from excessive deformation in the packing on the left.

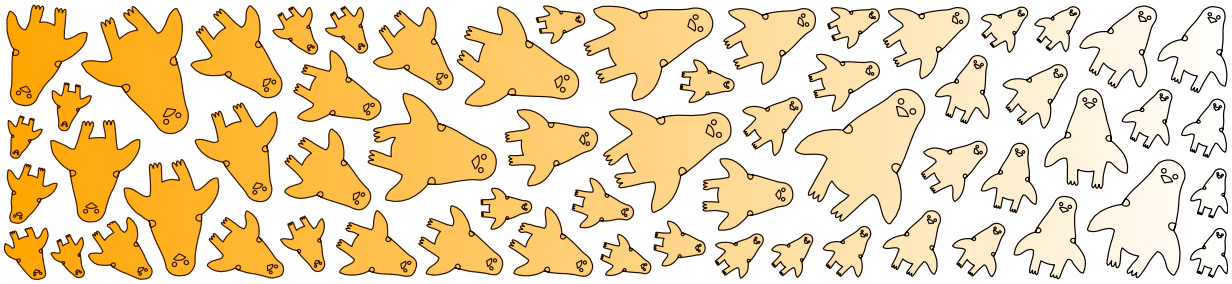


Figure 4.10: A packing that demonstrates torsional forces. The packing uses copies of a single element shape, but every copy is given a rest orientation between $0°$ and $180°$, based on its horizontal position in the container. In the final packing the elements transition from upright to upside-down, recreating an illusion in which giraffe heads become penguins.

and deformations of their originals, but also exert forces on them during simulation, leading to an even, seamless packing in a toroidal domain.

We have found that RepulsionPak is robust to parameter variation, and produces predictable, high quality results without the need for fine-grained adjustments. However, as shown in Figure 4.12, extreme parameter settings can still produce degenerate results. In Figure 4.12a, the repulsion force is made much stronger than the edge force, leading to excessive element deformation and self-intersections in the pursuit of even negative space. In Figure 4.12b, edge and torsional forces dominate repulsion, producing a packing with stiff, upright elements that do not fill their container effectively.

4.9 Conclusions

We presented RepulsionPak, a method to create packings with deformable elements. Each element is represented as a mass-spring system, allowing it to deform to achieve a better fit with its neighbors and the container. The combination of repulsion forces and controlled deformation allows RepulsionPak to create shape compatibilities that eliminate the need for a large element library and fill the target container effectively.

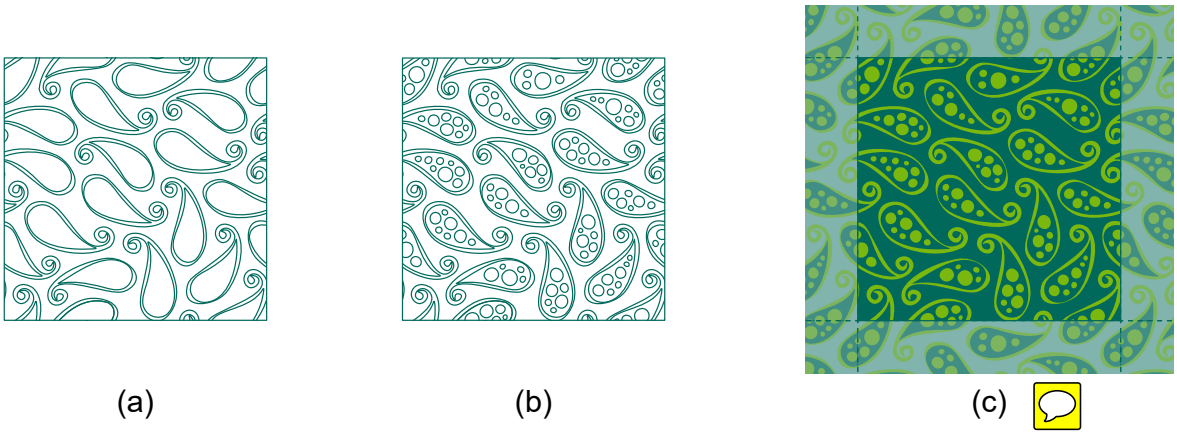


Figure 4.11: A paisley-inspired toroidal packing that can tile the plane. (a) An initial paisley packing. (b) In a separate simulation, we fill each paisley with circles to demonstrate a packing inside a packing. (c) The final result.

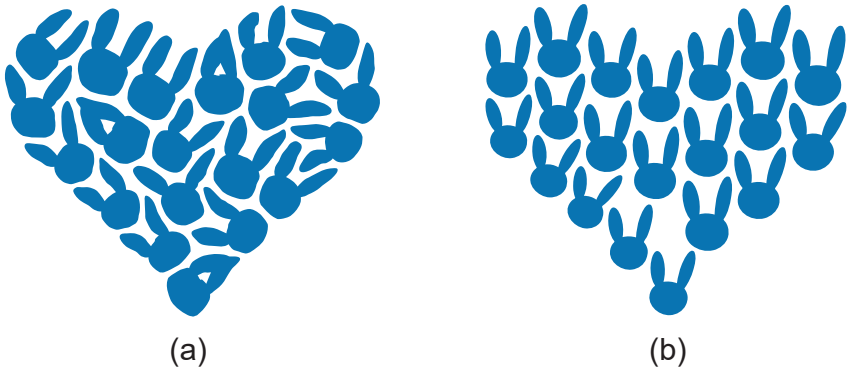


Figure 4.12: Two examples of how extreme parameter values can lead to low-quality results. In (a), we allow repulsion to overwhelm element shape by setting k_{rpl} to 200 and k_{edg} to 20; the resulting packing has even negative space, but elements suffer from high deformation and self-intersections. In (b), we minimize repulsion and prioritize orientation by setting k_{edg}, k_{tor}, and k_{rpl} to 200, 100, and 50, respectively. The elements deviate minimally from their original shapes and orientations, but cannot fill the container effectively.

Chapter 5

AnimationPak: Packing Elements with Scripted Animations

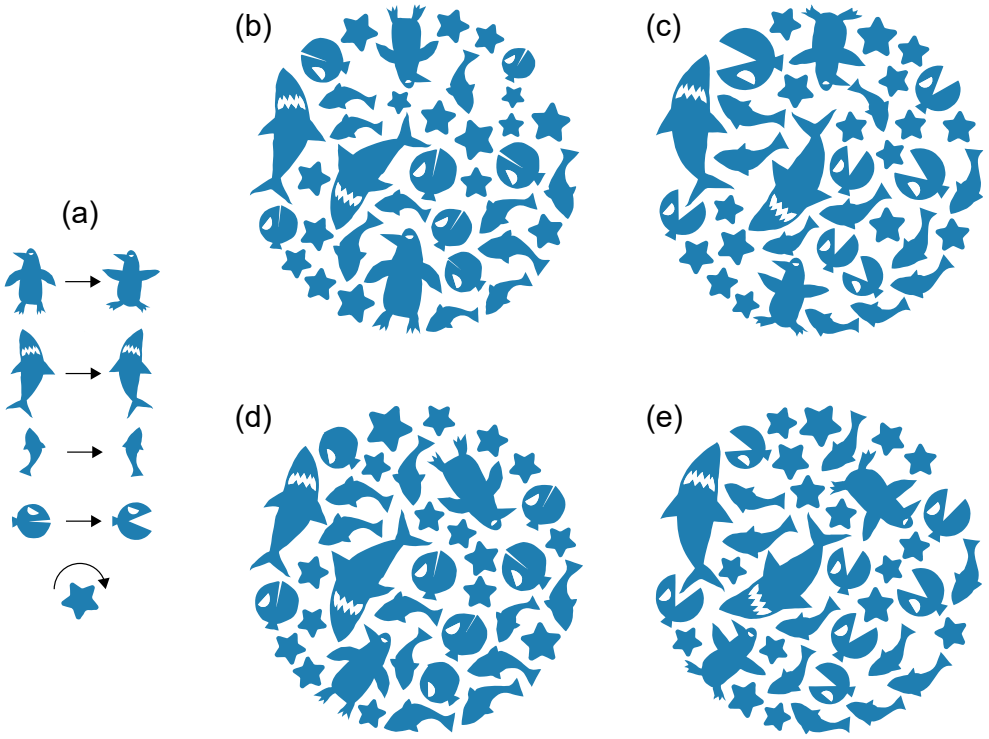


Figure 5.1: (a) Input animated elements, each with its own animation: swimming penguins, swimming sharks and fish, Pac-Man fish that open ~~or~~ close their mouths, and rotating stars. (b-e) four selected frames from an animated packing.

5.1 Introduction

AnimationPak is a method to pack elements with scripted animations. An element can have an animated deformation, such as a slithering snake or a dancing bear. It can also have an animated transformation, giving a changing position, size, and orientation within the container. Our goal is ~~producing~~ an *animated packing*, with elements playing out their animations while simultaneously filling the container shape evenly. A successful animated packing should balance among the evenness of the negative space, the preservation of element shapes, and the comprehensibility of their scripted animations.

We consider an animated element to be a geometric extrusion along a time axis, a three-dimensional object that we call a "spacetime element". We use a three-dimensional physical simulation similar to RepulsionPak to pack spacetime elements into a volume created by extruding a static container shape. The animated packing emerges from this three-dimensional volume by rendering cross sections perpendicular to the time axis.

Animated packings are a largely unexplored style of motion graphics, presumably because of the difficulty of creating an animated packing by hand. Finding motivating examples created by artists is difficult, and we only found a single animated packing shown in Figure 5.2, which is animated gear-like meshing of copies of Lisa Simpson's spiky head. The discussion in Section 5.2 shows there is also limited past research on animated packings.

5.2 Related Work

Packings and Mosaics: Researchers have explored many approaches to creating 2D packings and simulated mosaics, including using Centroidal Area Voronoi Diagrams (CAVDs) to position elements [Hau01, HHD03, SLK05], spectral approaches to create even negative space [DKLS06], energy minimization [KP02], and shape descriptors [KSH+16].

Animated Packings and Tilings: Animosaics by Smith et al. [SLK05] constructs animations in which static elements without scripted animations follow the motion of an animated container. Elements are placed using CAVDs, and advected frame-to-frame using a choice of methods motivated by Gestalt grouping principles. As the container's area changes, elements are added and removed as needed, while attempting to maximize overall temporal coherence. Dalal et al. [DKLS06] showed how the spectral approach they introduced for 2D packings could be extended to pack animated elements in a static container. Like us, they recast the problem in terms of three-dimensional spacetime; they computed optimal element placement using discrete samples over time and orientation.

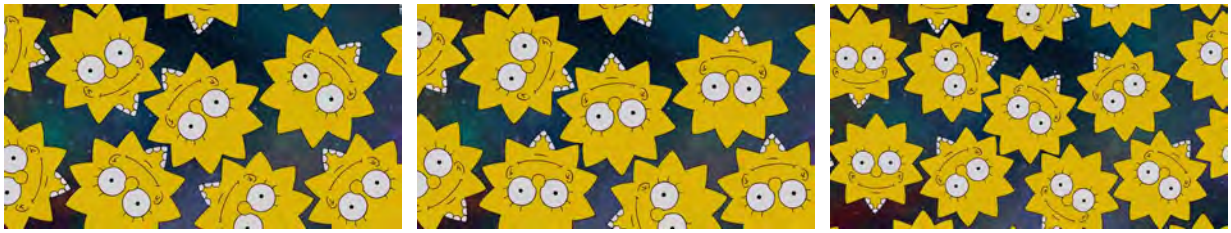


Figure 5.2: Three frames of an animated packing from The Simpsons Season 31, Episode 19 (© FOX), showing gear-like meshing of oppositely-rotating copies of Lisa Simpson.

However, their spacetime elements have fixed shapes and are made to fit together using only translation and rotation, limiting their ability to consume the container's negative space.

Liu and Veksler created animated decorative mosaics from video input [LV09]. Their technique combines vision-based motion segmentation with a packing step similar to Animosaics. Kang et al. [KOHY11] extracted edges from video and then oriented rectangular tesserae relative to edge directions.

Kaplan [Kap19] explored animations of simple tilings of the plane from copies of a single shape. Elements in a tiling fit together by construction, and therefore always consume all the negative space in the animation.

3D Packings: AnimationPak fills a 3D container with 3D elements, and is therefore related to approaches that construct 3D packings, although the time axis in the spacetime domain behaves differently than the third spatial axis. Related work includes 3D collages methods [GSP+07, TRdAS07, HWFL14], cutting and packing methods [CSR10, VGB+14, CZL+15], non-overlapping 3D element packing methods [BTW09, MCHW18], and packing methods for engineering purposes [CSY02].

Animated 3D Collages: Theobalt et al. [TRdAS07] developed a data-driven method to generate animated 3D collages that resemble an Arcimboldo's paintings by arranging static 3D elements into an animated 3D container. They partitioned the input animated container by identifying rigid parts, each ~~is~~ replaced by a matching element taken from a library. The result is a 3D collage with overlapping elements, and the animation is created from rigid transformations of the elements.

Animated Textures: Ma et al. [MWLT13] developed Dynamic Element Textures (DET), a method to synthesize an animated texture of a group of identical elements, such as shells or particles. For each animated element in an exemplar, DET generates sample points both in spatial and temporal domains. DET synthesizes a larger texture by copying

and distributing the sample points, which are then repositioned to resemble the original distribution in the exemplar. Similar to DET, AnimationPak represents an animated element as a collection of discrete vertices that reside in a spacetime domain. However, as a texture synthesis method, the goal of DET is to create an element arrangement that resembles an exemplar, and not fill a container tightly.

Derived Animations: AnimationPak falls into the category of systems that create a derived animation based on some input animation. This problem, which requires preserving the visual character of the input, is a longstanding one in computer graphics research. Spacetime constraints [WK88, Coh92] allow an animator to specify an object's constraints and goals, and then calculates the object's trajectory via spacetime optimization. Motion warping [WP95] is a method that deforms an existing motion curve to meet user-specified constraints. Bruderlin and Williams [BW95] used signal processing techniques to modify motion curves. Gleicher [Gle01] developed a motion path editing method that allows user to modify the traveling path of a walking character.

Previous work has also investigated geometric deformation of animations. Ho et al. [HKT10] encoded spatial joint relationships using tetrahedral meshes, and applied as-rigid-as-possible shape deformation to the mesh to retarget animation to new characters. Choi et al. [CKHL11] developed a method to deform character motion to allow characters to navigate tight passages. Masaki [Osh17] developed a motion editing tool that deformed 3D lattice proxies of a character's joints. Dalstein et al. [DRvdP15] presented a data structure to animate vector graphics with complex topological changes. Kim et al. [KHHL12] explored a packing algorithm to avoid collisions in a crowd of moving characters. They defined a motion patch containing temporal trajectories of interacting characters, and arranged deformed patches to prevent collisions between characters.

5.3 Animated Elements

The input to AnimationPak is a library of animated elements and a fixed container shape. AnimationPak currently supports two kinds of animation: the user can animate the shape of each individual element and can also give elements trajectories that animate their position within the container. This section explains how we animate the element shapes using as-rigid-as-possible deformation, and then construct spacetime-extruded objects that form the basis of our packing algorithm. These elements animate "in place": they change shape without translating. The next section describes how these elements can be given ~~transformation~~ trajectories within the container. Size and orientation of an element can be

animated either way; they can be specified as an animation of the element's shape, or they can be part of the transformation trajectory.

5.3.1 Spacetime Extrusion

Each element begins life as a static shape defined using vector paths. ~~Similar to~~ RepulsionPak, we construct a discrete geometric proxy of the element that will interact with other proxies in a physical simulation. The construction of this proxy for a single shape is shown in Figure 5.3, and the individual steps are explained in greater detail below.

In order to produce a packing with even distribution of negative space, we first offset the shape's paths by a distance Δs, leaving the shape surrounded by a channel of negative space (Figure 5.3a). In our system we scale the shape to fit a unit square and set $\Delta s = 0.04$.

Next, we place evenly-spaced samples around the outer boundary of the offset path and construct a Delaunay triangulation of the samples (Figure 5.3b). We will later treat the edges of the triangulation as springs, allowing the element to deform in response to forces in the simulation. We also follow RepulsionPak by adding shear edges to prevent folding and negative space edges to avoid self-intersections during simulation (Figure 5.3c).

We refer to the augmented triangulation shown in Figure 5.3c as a *slice*. The entire spacetime packing process operates on slices. However, we will eventually need to compute deformed copies of the element's original vector paths when rendering a final animation (Section 5.6). To that end, we re-express all path information relative to the slice triangulation: every path control point is represented using barycentric coordinates within one triangle.

To extend the element into the time dimension, we now position evenly-spaced copies of the slice along the time axis. Assuming that the animation will run over the time interval $[0, 1]$, we choose a number of slices n_s and place slices $\{s_1, \ldots, s_{n_s}\}$, with slice s_i being placed at time $(i-1)/(n_s-1)$. Higher temporal resolution will produce a smoother final animation at the expense of more computation. In our examples, we set $n_s = 100$. Figure 5.3d shows a set of time slices, with $n_s = 5$ for visualization purposes.

To complete the construction of a spacetime element without animation, we stitch the slices together into a single 3D object. Let s_j and s_{j+1} be consecutive slices constructed above. The outer boundaries of the element triangulations are congruent polygons offset in the time axis. We stitch the two polygons together using a new set of *time edges*: if AB is an edge on the boundary of s_j and CD is the corresponding edge on the boundary of s_{j+1},

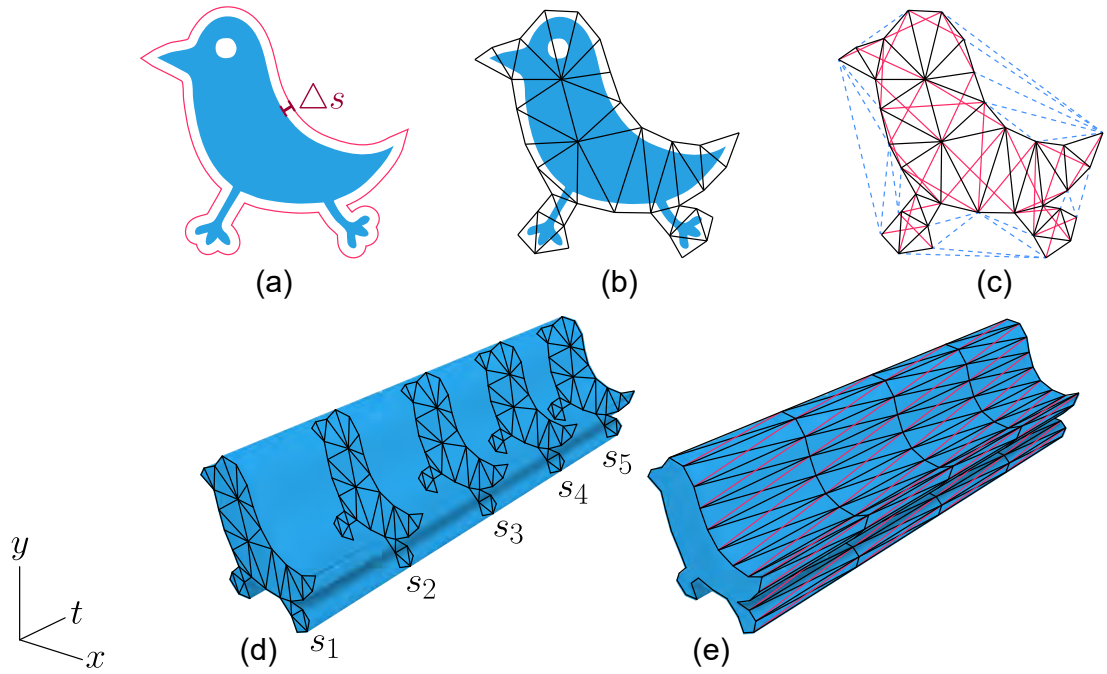


Figure 5.3: The creation of a discretized spacetime element. (a) A 2D element shape offset by Δs. (b) A single triangle mesh slice. (c) Shear edges (red) and negative space edges (blue). (d) A set of five slices placed along the time axis. (e) The vertices on the boundaries of the slices are joined by time edges. The black edges in (e) define a triangle mesh called the envelope of the element. In practice we use a larger number of slices in (d) and (e).

then we add time edges AC, AD, and BC. During simulation, time edges will transmit forces backwards and forwards in time, maintaining temporal coherence by smoothing out deformation and transformations. Figure 5.3e shows time edges for $n_s = 5$.

5.3.2 Animation

The 3D information constructed above is a parallel extrusion of a slice along the time axis, representing a shape with no scripted animation. We created a simple interactive application for adding animation to spacetime elements, inspired by as-rigid-as-possible shape manipulation [IMH05]. The artist first designates a subset of the slices as keyframes. They can then interactively manipulate any triangulation vertex of a keyframe slice. Any vertex that has been positioned manually has its entire trajectory through the animation

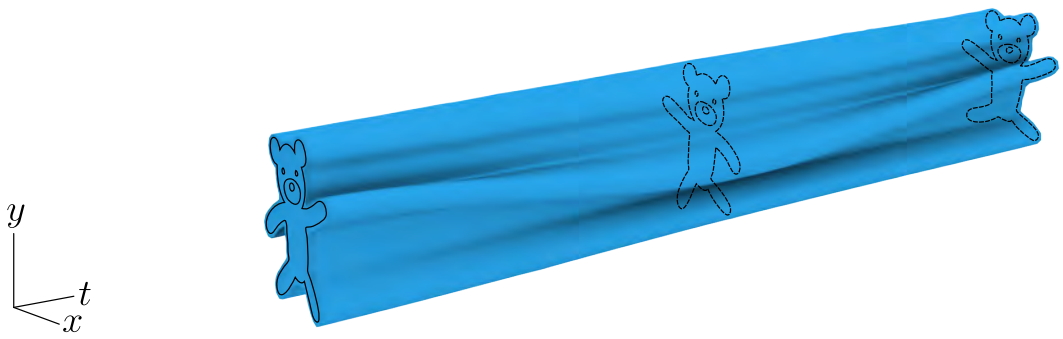


Figure 5.4: A spacetime element with a scripted animation.

computed using spline interpolation. Then, at any other slice, the positions of all other vertices can be interpolated using the as-rigid-as-possible technique. The result is a smoothly animated spacetime volume like the one visualized in Figure 5.4.

Unlike data-driven packing methods, AnimationPak allows distortions so it does not require a large library of distinct elements to generate successful packings. The results in this chapter all use fewer than ten input elements, and some use only one. The physical simulation induces deformation to enhance the compatibility of nearby shapes in the final animation.

5.4 Initial Configuration

We begin the packing process by constructing a 3D spacetime volume for the container by extruding its static shape in the time direction. The container is permitted to have internal holes, which are also extruded. The resulting volume is scaled to fit a unit cube. We also shrink each of the spacetime elements, in the spatial dimensions only, to 5–10% of its original size. These shrunken elements are thin enough that we can place them in the container without overlaps.

The artist can optionally specify trajectories for a subset of the elements, which we call *guided elements*. A guided element attempts to pass through a sequence of fixed target points in the container, imbuing the animation with a degree of intention and narrative structure. To define a guided element, we designate the triangulation vertex closest to its centroid to be the anchor point for the element. The artist then chooses a set of spacetime target points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$, with $\boldsymbol{p}_i = (x_i, y_i, t_i)$, that the anchor should pass through during the animation. In our interface, the artist uses a slider to choose the time t_i for a target

point, and clicks in the container to specify the spatial position (x_i, y_i). The artist can also optionally specify scale and orientation at the target points. We require $t_1 = 0$ and $t_n = 1$, fixing the initial and final positions of the guided element. We then linearly interpolate the anchor position for each slice based on the target points, and translate the slice so that its anchor lies at the desired position. The red extrusions in Figure 5.6a are guided elements.

If the artist wishes to create a looping animation, the (x_i, y_i) position for target points $\boldsymbol{p}_1$ and $\boldsymbol{p}_n$ must match up, either for a single guided element or across elements. In Figure 5.6 the two guided elements form a connected loop; (x_1, y_1) for each one matches (x_n, y_n) for the other.

In this initial configuration, the guided elements abruptly change direction at target points. However, because the slices are connected by springs, the trajectories will smooth out as the simulation runs. Also, the simulation is not constrained to reach each target position exactly. Instead, we attach the anchor to the target using a *target-point spring* that attempts to draw the element towards it while balancing against the other physical forces in play (Figure 5.6b). The strength of these springs determines how closely the element will follow the trajectory.

We then seed the container with an initial packing of non-guided spacetime elements. We generate points within the container at random, using blue-noise sampling [Bri07] to prevent points from being too close together, and assign a spacetime element to each seed point, selecting elements randomly from the input library. Depending upon the desired effect, we either randomize their orientations or give them preferred orientations. We reject any candidate seed point that would cause an unguided element's volume to intersect a guided element's volume.

Finally we shrink each element, guided and unguided, uniformly in the spatial dimension towards its centroid. These shrunken elements are guaranteed not to intersect one another; as the simulation runs, they will grow and consume the container's negative space, while avoiding collisions. The blue extrusions in Figure 5.6a show an initial placement of spacetime elements.

5.5 Simulation

We perform a physics simulation on the spacetime elements and the container. Elements are subjected to a number of forces that cause them to simultaneously grow, deform, and repel each other (Figure 5.6). In Section 5.5.2 we introduce some new hard constraints that must be applied after every time step. Note that we must distinguish two notions of

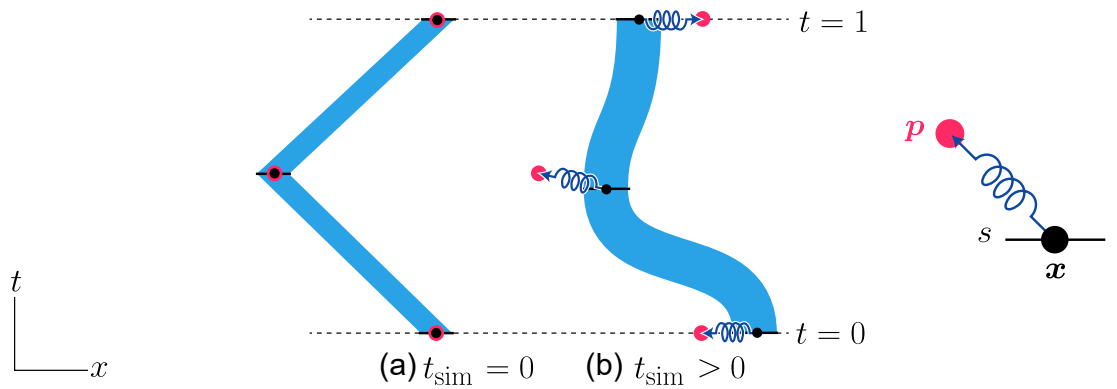


Figure 5.5: A 2D illustration of a guided element. Slices are depicted as black lines and slice vertices as black dots. A spring connects the centermost vertex $\boldsymbol{x}$ of a slice s to a target point p. (a) The initial shape of a guided element is a polygonal extrusion. (b) The spacetime element deforms but the springs pull it back towards the target points.

time in this simulation. We use t to refer to the time axis of our spacetime volume, which will become the time dimension of the final animation, and t_{sim} to refer to the time domain in which the simulation is taking place.

Let $\boldsymbol{x} = (x, y, t)$ be a vertex of a slice. The total force $\boldsymbol{F}_{\text{total}}$ applied to $\boldsymbol{x}$ is

$$\boldsymbol{F}_{\text{total}} = \boldsymbol{F}_{\text{rpl}} + \boldsymbol{F}_{\text{edg}} + \boldsymbol{F}_{\text{bdr}} + \boldsymbol{F}_{\text{ovr}} + \boldsymbol{F}_{\text{tor}} + \boldsymbol{F}_{\text{tmp}} \tag{5.1}$$

where

- $\boldsymbol{F}_{\text{rpl}}$ is the repulsion force;
- $\boldsymbol{F}_{\text{edg}}$ is the edge force;
- $\boldsymbol{F}_{\text{bdr}}$ is the boundary force;
- $\boldsymbol{F}_{\text{ovr}}$ is the overlap force;
- $\boldsymbol{F}_{\text{tor}}$ is the torsional force; and
- $\boldsymbol{F}_{\text{tmp}}$ is the temporal force.

These forces are the spacetime analogues of the ones used in RepulsionPak, except the new temporal force. In this section, we explain the first five forces briefly and readers can refer back to Section 4.5 for equations used to generate them.

Repulsion forces allow elements to push away vertices of neighboring elements, inducing deformations and transformations that lead to an even distribution of elements within the container (Figure 5.7). Since the simulation operates in the spacetime domain, vertex $\boldsymbol{x}$ accumulates repulsion forces from points at various time positions. To locate

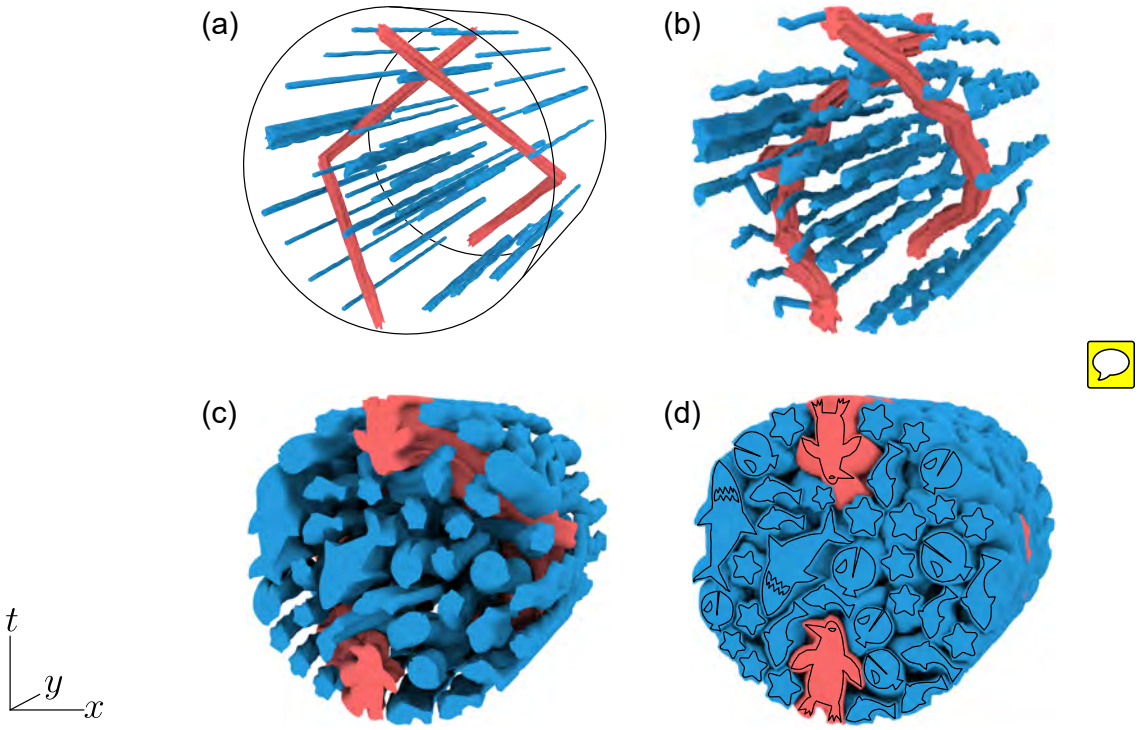


Figure 5.6: The simulation process. (a) Initial placement of shrunken spacetime elements inside a static 2D disc, extruded into a cylindrical spacetime domain. Guided elements are shown in red and unguided elements in blue. (b) A physics simulation causes the spacetime elements to bend. They also grow gradually. (c) The spacetime elements occupy the container space. (d) The simulation stops when elements do not have sufficient negative space in which to grow, or have reached their target sizes.

these points on neighboring elements that are considered nearest, we use a collision grid data structure, described in greater detail in Section 5.5.1.

Edge forces allow elements to deform in response to repulsion forces. As discussed in Section 5.3.1 and Section 5.4, we have five types of springs: edge springs, shear springs, negative-space springs, time springs, and target-point springs. Each ~~of the~~ spring type has a different relative strength.

Overlap forces resolve a vertex penetrating a neighboring spacetime element. Overlaps can occur later in the simulation when negative space is limited. Once we detect a penetration, we temporarily disable the repulsion force on vertex $\boldsymbol{x}$, and apply an overlap force $\boldsymbol{F}_{\text{ovr}}$ to push it out.

Boundary forces keep vertices inside the container. If an element vertex $\boldsymbol{x}$ is outside

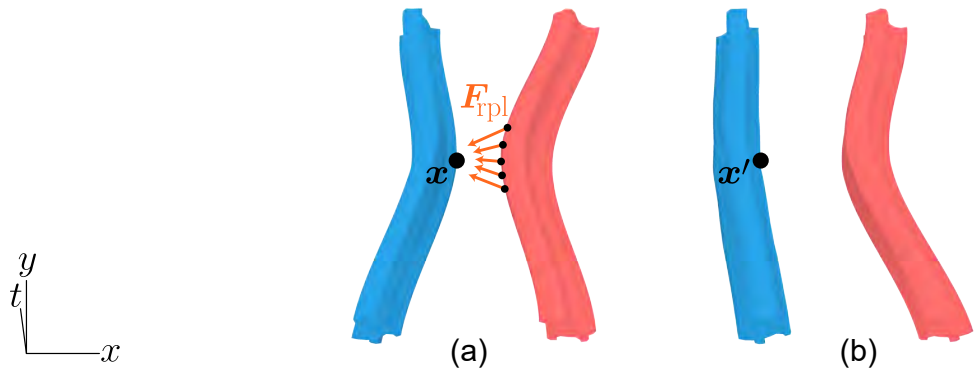


Figure 5.7: Repulsion forces applied to a vertex $\boldsymbol{x}$, allowing the element to deform and move away from a neighboring element.

the container, the boundary force $\boldsymbol{F}_{\text{bdr}}$ moves it towards the closest point on the container's boundary by an amount proportional to the distance to the boundary.

Torsional forces allow an element's slices to be given preferred orientations, to which they attempt to return.

Temporal forces prevent slices from drifting too far from their original positions along the time axis ~~positions~~ (Figure 5.8), which could cause unexpected accelerations and decelerations in the final animation. For every vertex, we compute the temporal force $\boldsymbol{F}_{\text{tmp}}$ as

$$\boldsymbol{F}_{\text{tmp}} = k_{\text{tmp}} \boldsymbol{u}^t (t - t') \tag{5.2}$$

where

k_{tmp} is the relative strength of the temporal force;
t is the initial time of the slice to which the vertex belongs;
t' is the current time value of the vertex; and
$\boldsymbol{u}^t = (0, 0, 1)$.

Simulation Details: We use explicit Euler integration to simulate the motions of the mesh vertices under the forces described above. Every vertex has a position and a velocity vector; in every iteration, we update velocities using forces, and update positions using velocities. These updates are scaled by a time step Δt_{sim} that we set to 0.01. We cap velocities at $10\Delta t_{\text{sim}}$ to dissipate extra energy from the simulation.

The constants k_{rpl}, k_{ovr}, k_{bdr} k_{edg}, k_{tor} and k_{tmp} control the relative strengths of repulsion, overlap, boundary, edge, torsional, and temporal forces, respectively. They must be adjusted relative to the time step Δt_{sim} and the size of the container. Since container's

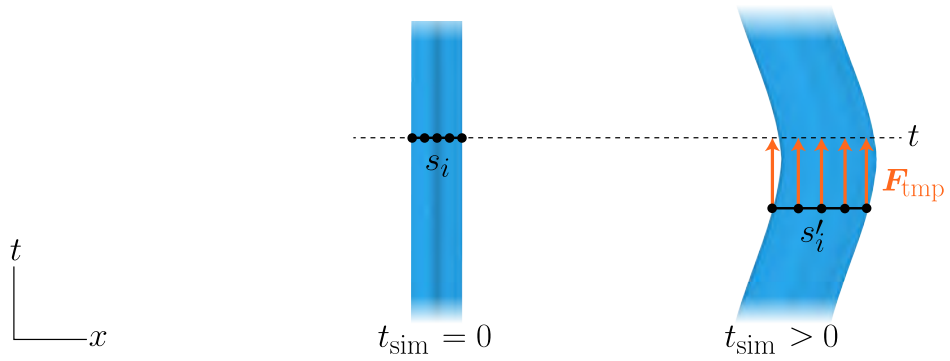


Figure 5.8: An illustration of the temporal force. The vertices in slice s_i are drawn back towards time t.

bounding box is resized to a unit cube, we set $k_{\text{rpl}} = 10$, $k_{\text{ovr}} = k_{\text{bdr}} = 5$, $k_{\text{tor}} = k_{\text{tmp}} = 1$. We set $k_{\text{edg}} = 0.01$ for time springs, $k_{\text{edg}} = 0.1$ for negative-space springs, and $k_{\text{edg}} = 10$ for edge springs, shear springs, and target-point springs.

5.5.1 Spatial Queries

Repulsion and overlap forces rely on being able to find points on neighboring elements that are close to a given query vertex. To find these points, we use each element's *envelope*, a triangle mesh implied by the construction in Section 5.3.1. Each triangle of the envelope is made from two time edges and one edge of a slice boundary, as shown in Figure 5.9a. Given a query vertex $\boldsymbol{x}$, we need to find nearby envelope triangles that belong to other elements.

To accelerate this computation, we first compute and store the centroids of every element's envelope triangles in a uniformly subdivided 3D grid that surrounds the spacetime volume of the animation. In using this data structure, we make two simplifying assumptions; first, that because envelope triangles are small, their centroids are adequate for finding triangles near a given query point; and second, that the repulsion force from a more distant triangle is well approximated by a force from its centroid.

Given a query vertex $\boldsymbol{x}$, we first find all envelope triangle centroids in nearby grid cells that belong to other elements. For each centroid, we use a method described by Ericson [Eri05] to find the point on its triangle closest to $\boldsymbol{x}$ and include that point in the list of points in Eq. (1). These nearby triangles will also be used to test for interpenetration of elements. We then find centroids in more distant grid cells, and add those centroids

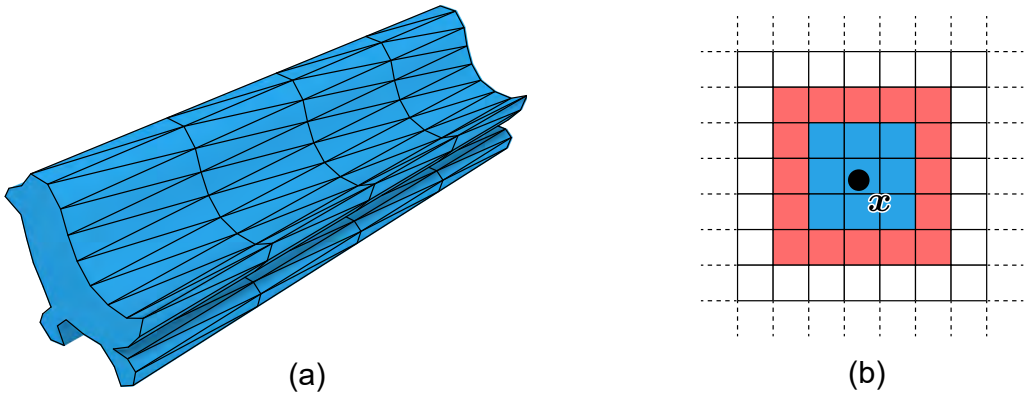


Figure 5.9: (a) The triangles that connect consecutive slices define the envelope of the element. The midpoints of these triangles are stored in a collision grid. (b) A 2D visualization of the region of collision grid cells around a query point $\boldsymbol{x}$ in which repulsion and overlap forces will be computed. In the central blue region, we check overlaps and compute exact repulsion forces relative to closest points on triangles of neighboring elements; in the peripheral red region we do not compute overlaps, and repulsion forces are approximated using triangle midpoints only.

directly to the Eq. (1) list, skipping the closest point computation. In our system we set the cell size to 0.04, giving a $25 \times 25 \times 25$ grid around the simulation volume. A query point's nearby grid cells are the 27 cells making up a $3 \times 3 \times 3$ block around the cell containing the point; the more distant cells are the 98 that make up the outer shell of the $5 \times 5 \times 5$ block around that (Figure 5.9).

5.5.2 Slice Constraints

There are three hard geometric constraints on the configuration of slices, which must be enforced throughout the simulation. Each of the following constraints is reapplied after each physical simulation step described above.

1. **End-To-End Constraint:** A spacetime element must be present for the full length of the animation from $t = 0$ to $t = 1$. After every simulation step, every vertex belonging to an element's first slice has its t value set to 0, and every vertex of the last slice has its t value set to 1 (Figure 5.10a).
2. **Simultaneity Constraint:** During simulation, the vertices of a slice can drift away from each other in time, which could lead to rendering artifacts in the animation.

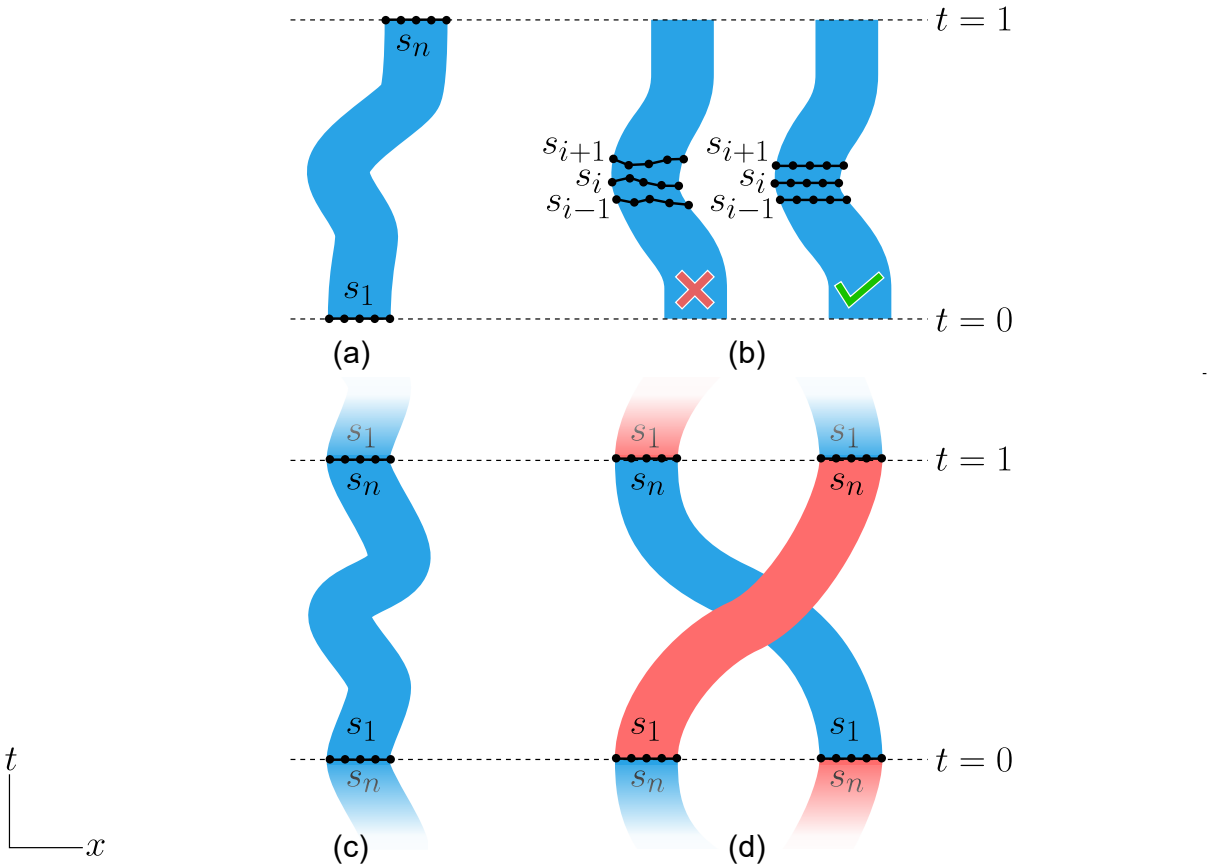


Figure 5.10: a) End-to-end constraint: slice s_1 and s_n, located at $t = 0$ and $t = 1$, should never change their t positions but can change their x, y positions. b) Simultaneity constraint: all vertices on the same slice should have the same t position. c) Loop constraint with a single element: the x, y positions for s_1 and s_n must match. d) Loop constraint with two elements: the x, y position for s_1 for one element matches the x, y position for s_n of the other.

After every simulation step, we compute the average t value of all vertices belonging to each slice other than the first and last slices, and snap all the slice's vertices to that t value (Figure 5.10b).

3. **Loop Constraint:** AnimationPak optionally supports looping animations. When looping is enabled, we must ensure that the $t = 0$ and $t = 1$ planes of the spacetime container are identical. The $t = 1$ slice of every element e_1 must then coincide with the $t = 0$ slice of *some* element e_2. We can have $e_1 = e_2$ (Figure 5.10c), but more general loops are possible in which the elements arrive at a permutation of their

original configuration (Figure 5.10d). We require only that there is a one-to-one correspondence between the vertices of the $t = 1$ slice of e_1 and the $t = 0$ slice of e_2. If $\boldsymbol{p}_1 = (x_1, y_1, 1) \in e_1$ and $\boldsymbol{p}_2 = (x_2, y_2, 0) \in e_2$ are in correspondence, then after every simulation step we move $\boldsymbol{p}_1$ to $(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}, 1)$ and $\boldsymbol{p}_2$ to $(\frac{x_1+x_1}{2}, \frac{y_2+y_2}{2}, 0)$.

5.5.3 Element Growth and Stopping Criteria

We begin the spacetime packing process with all element slices scaled down in x and y, guaranteeing that elements do not overlap. As the simulation progresses we gradually grow the slices, consuming the negative space around them (Figure 5.11a,b). A perfect packing would fill the spacetime container completely with the elements. Because each element wraps the underlying animated shape with a narrow channel of negative space, this would yield an even distribution of shapes in the resulting animation. For real-world elements, the goal of minimizing deformation of irregular element shapes will lead to imperfect packings with additional pockets of negative space.

Element Growth: We induce elements to grow spatially by gradually increasing the rest lengths of their springs. The initial rest length of each spring is determined by the vertex positions in the shrunken version of the spacetime element constructed in Section 5.4. We allow an element's slices to grow independently of each other, which complicates the calculation of new rest lengths for time springs. Therefore, we create a duplicate of every shrunken spacetime element in the container, with a straight extrusion for unguided elements, and a polygonal extrusion for guided elements. This duplicate is not part of the simulation; it serves as a reference. Every element slice maintains a current scaling factor g. When we wish to grow the slice, we increase its g value. We can compute new rest lengths for all springs by scaling every slice of the reference element by a factor of g relative to the slice's centroid, and measuring distances between the scaled vertex positions. These new rest lengths are then used as the ℓ values in Equation 4.3 to calculate edge forces.

Every element slice has its g value initialized to 1. After every simulation step, if none of the slice's vertices were found to overlap other elements we increase that slice's g by $0.001\Delta t_{\text{sim}}$, where Δt_{sim} is the simulation time step. If any overlaps are found, then that slice's growth is instead paused to allow overlap and repulsion forces to give it more room to grow in later iterations. This approach can cause elements to fluctuate in size during the course of an animation, as slices compete for shifting negative space (Figure 5.11).

Stopping Criteria: We halt the simulation when the space between neighboring elements drops below a threshold. When calculating repulsion forces, we find the distance

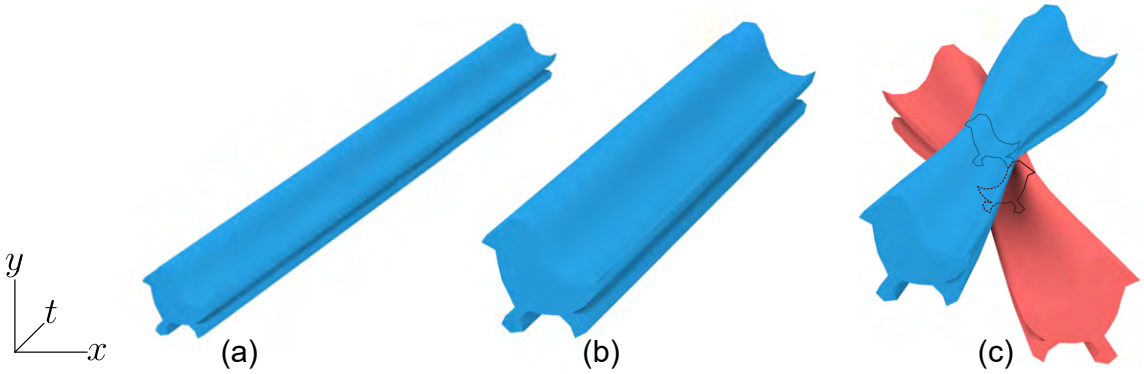


Figure 5.11: A spacetime element shown (a) shrunken at the beginning of the simulation, and (b) grown later in the simulation. (c) When two elements overlap somewhere along their lengths, they are temporarily prohibited from growing there.

from every slice vertex to the closest point in a neighboring element. The minimum of these distances over all vertices in an element slice determines that slice's closest distance to neighboring elements. We halt the simulation when the maximum per-slice distance falls below 0.006 (relative to a normalized container size of 1). That is, we stop when every slice is touching (or nearly touching) at least one other element.

In some cases it can be useful to stop early based on cumulative element growth. In that case, we set a separate threshold for the slice scaling factors g described above, and stop when the g values of all slices exceed that threshold.

5.6 Rendering

The result of the simulation described previously is a packing of spacetime elements within a spacetime container. We can render an animation frame-by-frame by cutting through this volume at evenly spaced t values from $t = 0$ to $t = 1$, as shown in Figure 5.12. For our results, we typically render 500-frame animations.

During simulation, a given spacetime element's slices may drift from their original creation times. However, time springs keep the sequence monotonic, and the simultaneity constraint ensures that every slice is fixed to one t value. To render this element at an arbitrary frame time $t_f \in [0, 1]$, we find the two consecutive slices whose time values bound the interval containing t_f and linearly interpolate the vertex positions of the triangulations at those two slices to obtain a new triangulation at t_f. We can then compute a deformed

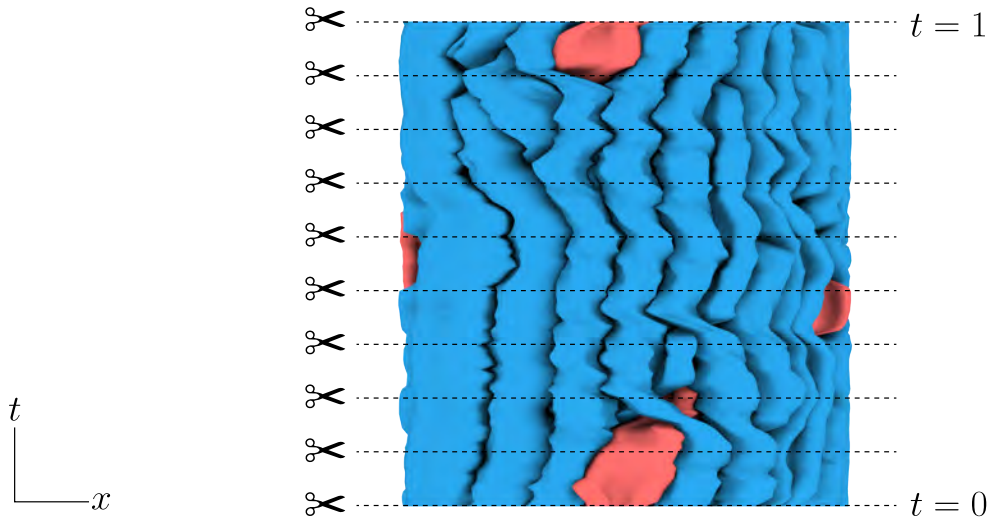


Figure 5.12: An illustration of rendering spacetime elements to generate 2D frames by cutting them at evenly spaced time values from $t = 0$ to $t = 1$.

copy of the original element paths by "replaying" the barycentric coordinates computed in Section 5.3.1 relative to the displaced triangulation vertices. We repeat this process for every spacetime element to obtain a rendering of the frame at t_f.

This interpolation process can occasionally lead to small artifacts in the animation. A rendered frame can fall between the discretely sampled slices for two elements at an intermediate time where physical forces were not computed explicitly. It is therefore possible for neighboring elements to overlap briefly during such intervals.

5.7 Implementation and Results

The core AnimationPak algorithm consists of a C++ program that reads in text files describing the spacetime elements and the container, and outputs raster images of animation frames.

Large parts of AnimationPak can benefit from parallelism. In our implementation we update the cells of the collision grid (Section 5.5.1) in parallel by distributing them across a pool of threads. When the updated collision grid is ready, we distribute the spacetime elements over threads. We calculate forces, perform numerical integration, and apply the end-to-end and simultaneity constraints for each element in parallel. We must process any loop constraints afterwards, as they can affect vertices in two separate elements.

Table 5.1: Data and statistics for the AnimationPak results. The table shows the number of elements, the number of vertices, the number of springs, the number of envelope triangles, and the running time of the simulation in hours, minutes, and seconds.

Packing	Elements	Vertices	Springs	Triangles	Time
Aquatic fauna (Figure 5.1)	37	97,800	623,634	106,000	01:06:35
Snake and bird (Figure 5.13)	37	58,700	370,571	58,700	01:01:32
Penguin to giraffe (Figure 5.14)	33	124,300	824,164	143,000	01:19:50
Traveling bird (Figure 5.15)	32	63,700	389,373	70,100	00:19:24
Lion (Figure 5.16b)	16	39,400	236,086	41,800	00:41:56
Animals (Figure 5.17b)	34	69,600	444,337	69,800	01:00:19
Heart stars (Figure 5.18c)	26	85,200	598,218	85,800	00:23:08

We created the results in this chapter using a Windows PC with a 3.60 GHz Intel i7-4790 processor and 16 GB of RAM. We used a pool of eight threads, corresponding to the number of logical CPU cores. Table 5.1 shows statistics for our results. Each packing has tens of thousands of vertices and hundreds of thousands of springs, and requires about an hour to complete. We enable the loop constraint in all results. This chapter shows selected frames from the results; see supplemental videos for full animations[1].

Figure 5.1 is an animation of aquatic fauna featuring two penguins as guided elements. During one loop the penguins move clockwise around the container, swapping positions at the top and the bottom. Each ends at the other's starting point, demonstrating a loop constraint between distinct elements. All elements are animated, as shown in Figure 5.1a. Note the coupling between the Pac-Man fish's mouth and the shark's tail on the left side of the second and fourth frames.

A snake chases a bird around an annular container in Figure 5.13, demonstrating a container with a hole and giving a simple example of the narrative potential of animated packings. Figure 5.14 animates the giraffe-to-penguin illusion shown as a static packing in

[1] https://cs.uwaterloo.ca/~radhitya/animationpak/videos.zip

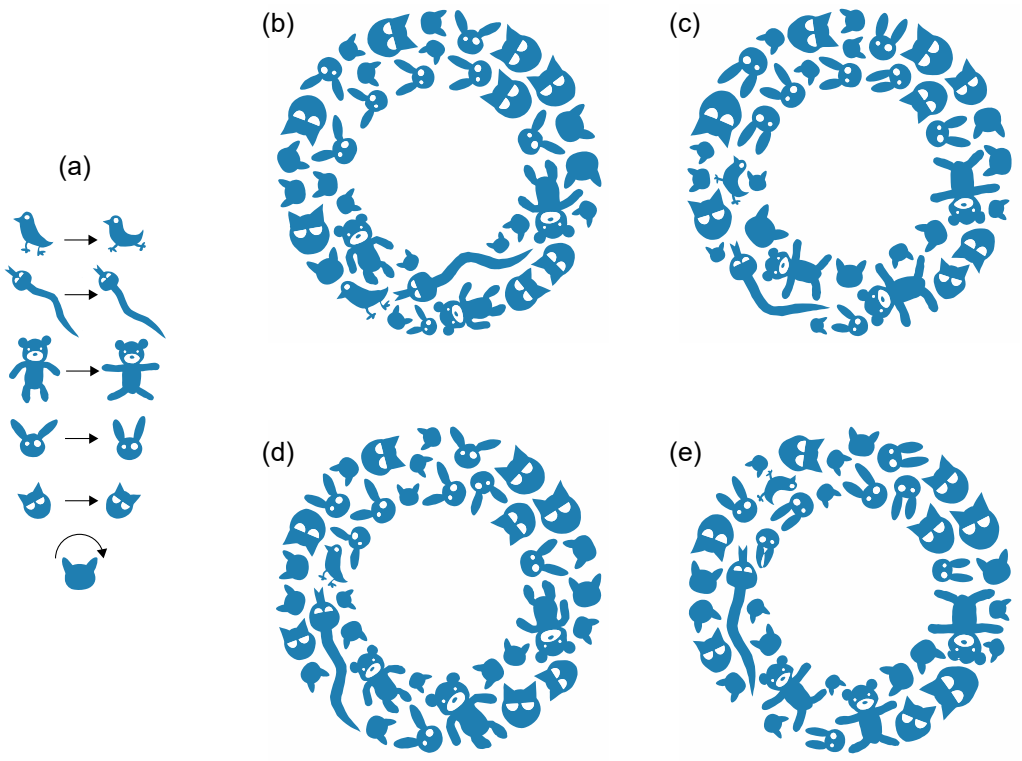


Figure 5.13: A snake chasing a bird through a packing of animals. The snake and bird are both guided elements that move clockwise around the annular container.

RepulsionPa This example uses torsional forces to control slice orientations.

Figure 5.15 shows ~~a guided element as a traveling bird which is not constricted inside a container.~~ The traveling bird moves from left to right, passing through a packing of birds. As some slices of the traveling bird taking up container space, the other birds have to stop some of their slice growths prematurely. This is a demonstration of the benefit of ~~the non-uniform slice growths so that elements can~~ adapt to changes in container area. In the rendered 2D animation, the elements have a subtle shrinking-and-expanding effect. The composition can also tile horizontally, which can be seen in the supplemental video.

Figure 5.16a is a static packing of lion's mane created by an artist and used as an example in FLOWPAK (Chapter 3). In Figure 5.16b, we reproduce it with animated elements for the mane. The orientations of elements follow a vector field inside the container, and are maintained during the animation by torsional forces. We simulate only half of the packing and reflect it to create the other half. The facial features were added manually in a post-processing step.

Figure 5.14: Penguins turning into giraffes. The penguins animate by rotating in place. Torsional forces are used to preserve element orientations. Frames are taken at $t = 0$, $t = 0.125$, $t = 0.25$, $t = 0.375$, and $t = 0.5$.

Figure 5.17 compares a static 2D packing created by RepulsionPak with a frame from an animated packing created by AnimationPak. The extra negative space in AnimationPak comes partly from the trade-off between temporal coherence and tight packing, and partly from the lack of secondary elements, which were used in a second pass in RepulsionPak to fill pockets of negative space.

Figure 5.18 offers a direct comparison between packings computed using Centroidal Area Voronoi Diagrams (CAVD) [SLK05], the spectral approach [DKLS06], and AnimationPak. These packings use stars that rotate and pulsate. For each method we show the initial frame ($t = 0$) and the halfway point ($t = 0.5$). The CAVD approach produces a satisfactory—albeit loosely coupled—packing for the first frame, but because the algorithm was not intended to work on animated elements, the evenness of the packing quickly degrades in later frames. The spectral approach is much better than CAVD, but their animated elements still have fixed spacetime shapes and can only translate and rotate to improve their fit. Repulsion forces and deformation allow AnimationPak to achieve a tighter packing that persists across the animation, including gear-like meshing of ~~oppositely-rotating~~ stars.

Figure 5.19 emphasizes the trade-off between temporal coherence and evenness of negative space by creating two animations with different time springs stiffness. In (a), the time springs are 100 times stronger than in (b). The resulting packing has larger pockets of negative space, but the accompanying video shows that the animation is smoother. The packing in (b) is tighter, but the elements must move frantically to maintain that tightness.

Figure 5.20 is a failed attempt to animate a "blender". The packing has a beam that rotates clockwise and a number of small unguided circles. In a standard physics simulation we might expect the beam to push the circles around the container, giving each one a helical spacetime trajectory. Instead, as elements grow, repulsion forces cause circles to explore the container boundary, where they discover the lower-energy solution of slipping past the edge of the beam as it sweeps past. If we extend the beam to the full diameter

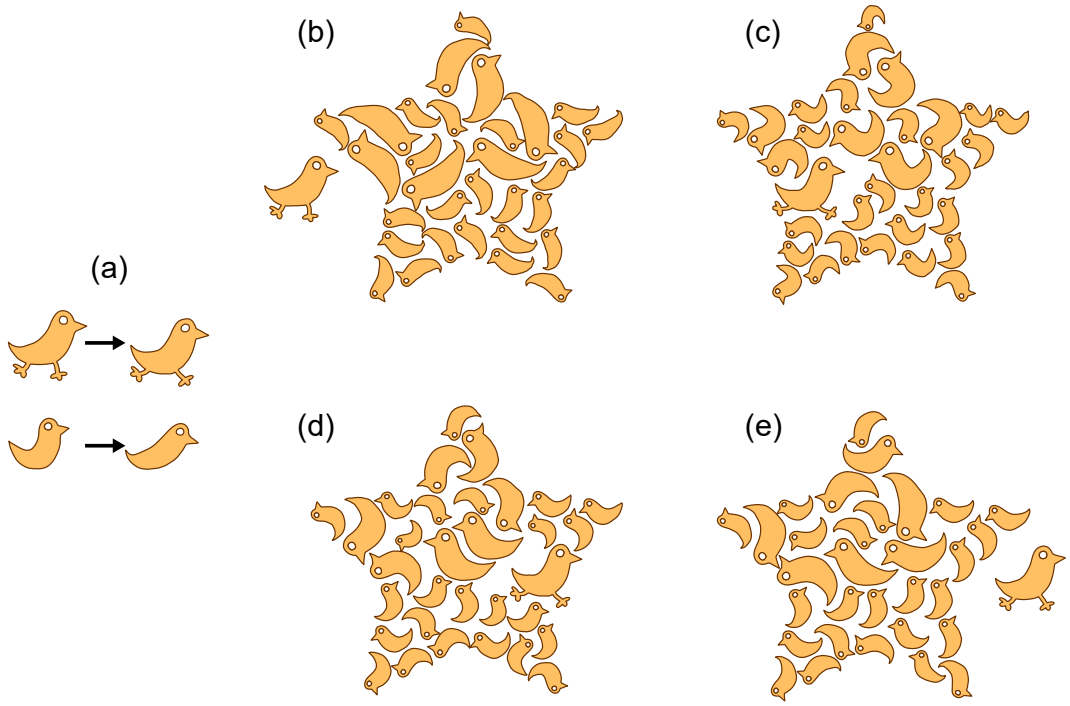


Figure 5.15: An animation of a traveling bird that comes in and out a packing of birds. The other birds adapt to the reduced space by prematurely stopping some of their slices' growths.

of the container, consecutive slices simply teleport across the beam, hiding the moment of overlap in the brief time interval where physical forces were not computed. AnimationPak is not directly comparable to a 3D physics simulation; it is better suited to improving the packing quality of an animation that has already been blocked out at a high level.

5.8 Conclusions

We introduced AnimationPak, a system for generating animated packings by filling a static container with animated elements. Every animated 2D element is represented by an extruded spacetime tube. We discretize elements into triangle mesh slices connected by time edges, and deform element shapes and animations using a spacetime physical simulation. The result is a temporally coherent 2D animation of elements that attempt both to perform their scripted motions and consume the negative space of the container. We show a variety of results where 2D elements move around inside the container.

Figure 5.16: (a) A static packing made by an artist, taken from StockUnlimited. (b) The first frame from an AnimationPak packing. (c) The input animated elements and the container shape with a vector field. Torsional forces keep elements oriented in the direction of the vector field. We simulate half of the lion's mane and render the other half using a reflection, and add the facial features by hand.

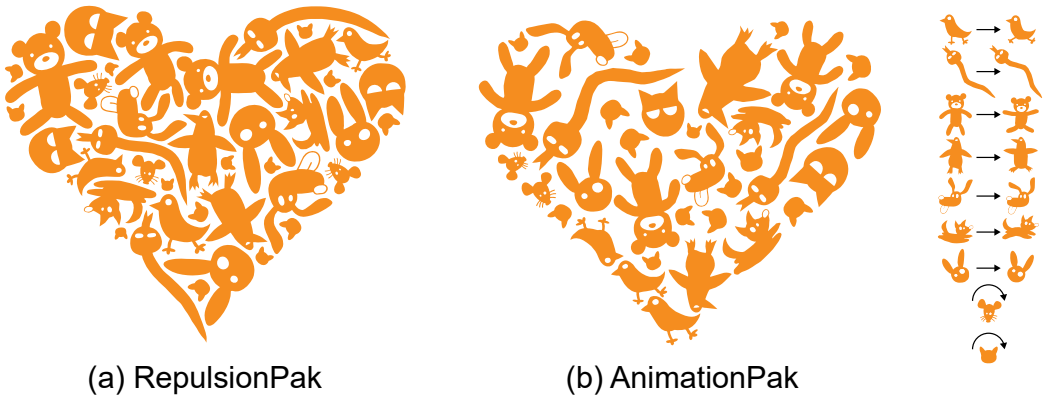
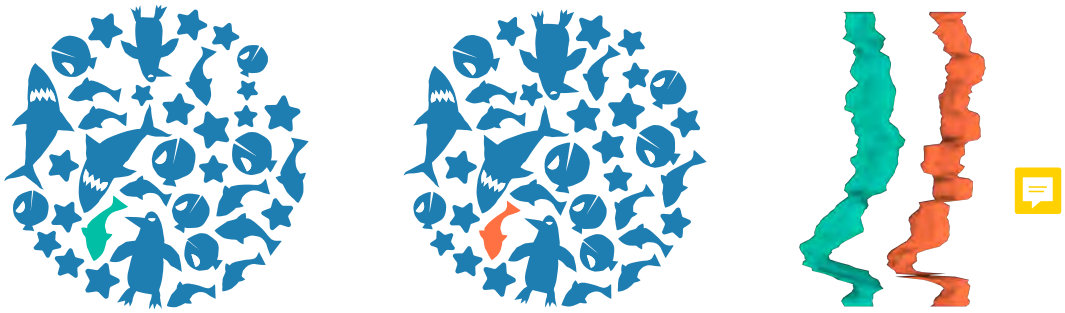


Figure 5.17: (a) A static packing created with RepulsionPak. (b) The first frame of a comparable AnimationPak packing. The input spacetime elements are shown on the right. The AnimationPak packing has more negative space because we must ~~tradeoff~~ between temporal coherence and packing density.

(a) CAVD (b) The spectral approach (c) AnimationPak

Figure 5.18: A comparison of (a) Centroidal Area Voronoi Diagrams (CAVDs) [SLK05], (b) spectral packing [DKLS06], and (c) AnimationPak. We show two frames for each method, taken at $t = 0$ and $t = 0.5$. The CAVD packing starts with evenly distributed elements but the packing degrades as the animation progresses. The spectral approach improves upon CAVD with better consistency, but still leaves significant pockets of negative space. The AnimationPak packing has less negative space that is more even.



(a) (b) (c)

Figure 5.19: (a) One frame from Figure 5.1. (b) The same packing with time springs that are 1% as stiff. Reducing the stiffness of time springs leads to a more even packing with less negative space, but the animated elements must move frantically to preserve packing density. The spacetime trajectories of the highlighted fish in (a) and (b) are shown in (c). The orange fish in (b) exhibits more high frequency fluctuation in its position.

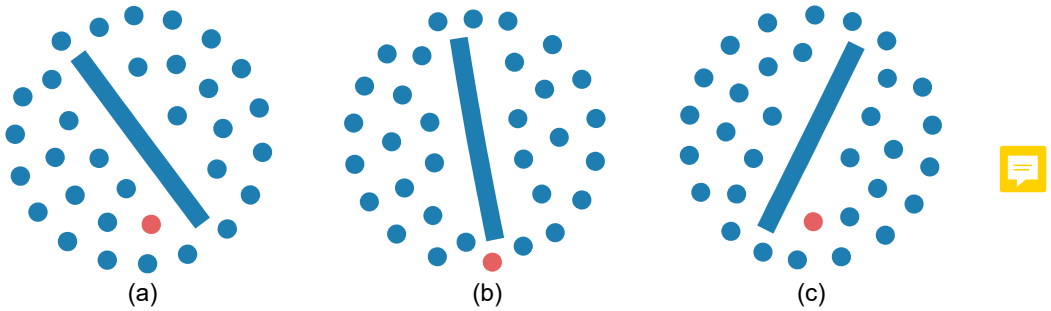


Figure 5.20: A failure case for AnimationPak, consisting of a rotating beam and a number of small circles. Instead of being dragged around by the beam, the circles dodge it entirely by sneaking through the gap between the beam and the container. The red circle demonstrates one such maneuver.

Chapter 6

Quantitative Metrics for 2D Packings

6.1 Introduction

As with many techniques in artistic applications of computer graphics, evaluating the quality of a computer-generated element packing is a challenge. The visual appeal of packings follows in part from aligning neighboring elements along compatible segments of their boundaries, suggesting that they interlock by design. We believe that the evenness of negative space is an indicator of the quality of a packing. The separation between neighboring elements should be roughly the same everywhere. In this chapter, we have developed several measurements of evenness, which allow us to examine the behavior of manually constructed reference packings and to compare RepulsionPak with other packing algorithms.

6.2 Related Work

In this section, we discuss a few related methods for evaluating discrete texture synthesis, stippling, or packings. For more general discussion, Isenberg compiled a survey of evaluation methods in NPR research [Ise13].

Qualitative Evaluation: AlMeraj et al. [AKA13b] proposed qualitative evaluations to ~~find~~ the similarity between exemplars and synthesized textures. They conducted two user studies: a pile-sorting study and a pairwise comparison study. These user studies ~~concerned about~~ small-sized element distributions, but not element interlocking or the

evenness of negative space, so their work is not suitable for our packing evaluation. Kwan et al. performed a simple user study of 13 participant to compare Pyramid of Arclength Descriptor (PAD) [KSH+16] with previous packing methods. A participant was randomly shown a computer-generated packing and they had to give ratings from 1 (worst) to 6 (best) ~~to~~ each of three categories: their preference, stylishness, interlocking. The study revealed that PAD received the highest scores on all three categories. However, we argue that the first two categories are ambiguous and do not ~~give better understanding to the quality of a packing.~~

Quantitative Evaluation: In Jigsaw Image Mosaics (JIM) [KP02], Kim et al. used an energy minimization approach that penalizes a packing if it has too much negative space, too many overlaps, or severely deformed elements. They did not further discuss whether ~~it~~ can be used for a quantitative evaluation, but we are inspired by their work to develop a metric based on shape overlaps (Section 6.3.3). Maciejewski et al. [MIA+08] compared computer-generated stippling artwork with artist-made stippling artwork using the Gray-Level Co-Occurrence Matrix (GLCM), which measures spatial relationships between pixel intensities. GLCM works best for analyzing dense point distributions, but it unfortunately cannot be used to analyze arrangements with larger shapes. In medical research, Aliy et al. [ASF+13] proposed a method to analyze the arrangement of small biological objects in a histology image. They treated these objects as a point distribution by computing their centroids and encoded their spatial relationships, so their work is unsuitable for our research. In fabrication, the packing quality is dictated by manufacturing constraints, such as connectivity, so that printed objects can resist from breaking [CZX+16, ZCT16, MSS+19]. However, our objective is more about the evenness of negative space and less about connectivity strengths.

6.3 Quantitative Metrics

Calibration: Two packings must be calibrated to each other before our measurements can be compared meaningfully. We can compare packings of different sizes by normalizing their containers to have unit area. We must also arrange for the packings to have the same *negative space ratio* (the overall amount of negative space as a fraction of container area), so that our measurements can focus on the distribution of negative space and not just the amount. Given two calibrated packings, we examine three main evenness metrics: spherical contact probabilities, distance histograms, and overlap of offset elements.

6.3.1 Spherical Contact Probability

The spherical contact probability (SCP) is the probability that a disc of radius r, chosen uniformly at random within the container region, lies entirely within the packing's negative space [CSKM13]. The SCP can be summarized via a function $Q_s(r)$ that gives this probability for each radius r. In order to interpret the SCP, it is helpful first to examine a "packing" with perfectly even negative space (Figure 6.1a). Consider a pattern of infinite horizontal stripes of width d_s, separated from each other by negative space of width d_{gap}. For this pattern, $Q_s(0) = d_{gap}/(d_{gap} + d_s)$; it is also clear that $Q_s(d_{gap}/2) = 0$, because no disc of diameter greater than d_{gap} can fit in the negative space (our d_{gap} is twice the radius of the ball). Furthermore, $Q_s(r)$ will decrease linearly between these two points, and remain at zero thereafter; its graph will consist of a tilted line segment connected to a horizontal ray.

No real-world packing exhibits this SCP. Even in a perfect arrangement of squares (Figure 6.1b), the intersections of horizontal and vertical channels produce pockets of negative space that can accommodate balls of radius $d_{gap}\sqrt{2}/2$. These pockets tend to raise the SCP slightly everywhere, and cause it to bend into a small tail that approaches zero gradually. For a given set of elements in a container, the best packings will have a steeply-decreasing SCP that stays close to the idealized stripe function most of the way down, has a low value at $r = d_{gap}/2$, and then bends towards horizontal near that value. Note that there is always a largest disc that can fit in a packing's negative space, and hence a largest r for which $Q_s(r) > 0$. In our graphs, we plot $Q_s(r)$ only until this point, allowing us to compare the largest empty gaps of two packings. In less effective packings (Figs. 6.1c,d), the negative space will be narrower in some places and wider in others, recognizable as a shallower SCP with a longer tail.

We geometrically compute SCP by offsetting the negative space inward. Let N be the shape of the negative space (essentially the container region with holes where elements are). For a given radius r, we compute $N(r)$, the Minkowski difference of N with a disc of radius r. Then $Q_s(r)$ is simply the ratio of the area of $N(r)$ to the area of the container.

6.3.2 Histograms of the Distance Transform

The distance transform of the negative space provides insights into how negative space varies. In the context of 2D packings, a distance transform is a 2D image where each pixel contains the radius r value to the nearest element. Figure 6.2 shows an example for a distance transform, lighter a color of a pixel, higher the r value. We investigated

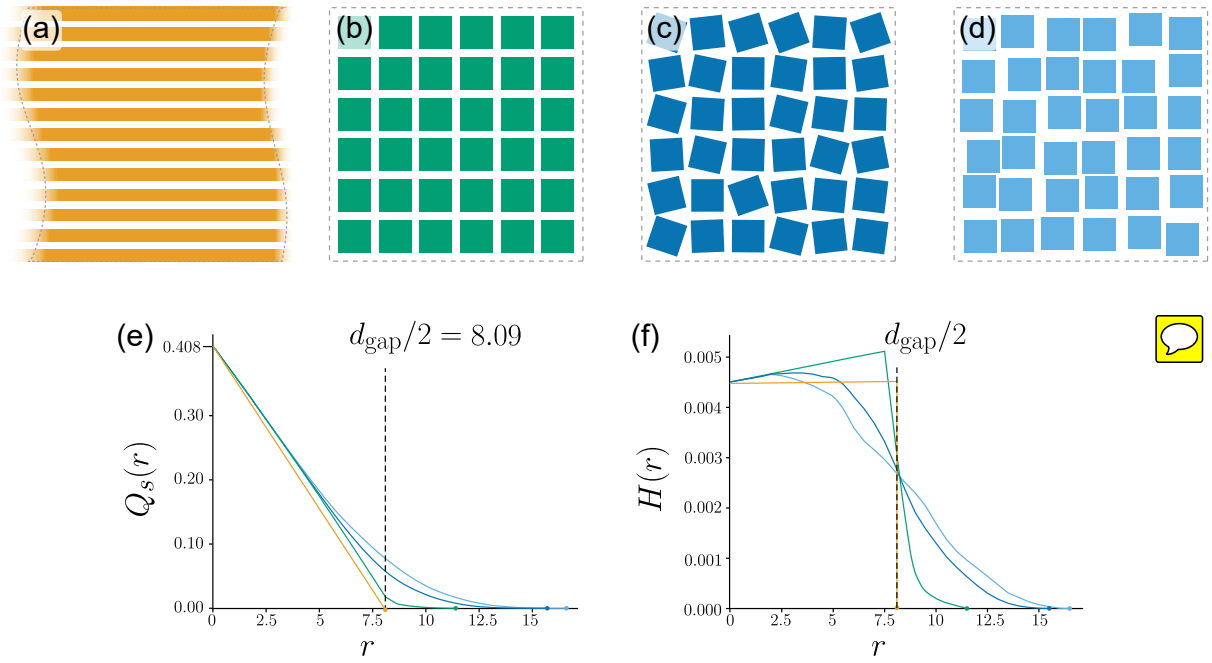


Figure 6.1: Spherical contact probabilities and distance histograms for reference packings. A "perfect packing" of infinite stripes is shown in (a), followed by a square packing with the same area fraction and negative space width d_{gap} in (b). The square packing is then perturbed with random rotations in (c) and translations in (d). The corresponding SCP functions and histograms are plotted in (e) and (f).

the idea of analyzing r values only on a medial axis since it bisects the gaps of negative space. However, the challenge is to prune a significant number of problematic medial axis's branches that are either too long or too short, as shown in Figure 6.3.

As creating a high quality medial axis is difficult, we decide to analyze the entire negative space by computing the histogram of the distance transform. However, this would require quantizing r values into bins. Instead, note that the SCP $Q_s(r)$ is precisely the normalized area of negative space for which the distance transform is at least r. Looked at another way, if we offset each element by a radius r, then $1 - Q_s(r)$ must be the normalized area of the union of these offset elements. This area can then be interpreted as a cumulative distribution function of distance. From this observation we can compute a continuous variant of the distance histogram as a probability density function via the derivative of the SCP: $H(r) = -Q_s'(r)$.

Given two calibrated packings, the areas under their distance histograms are the same.

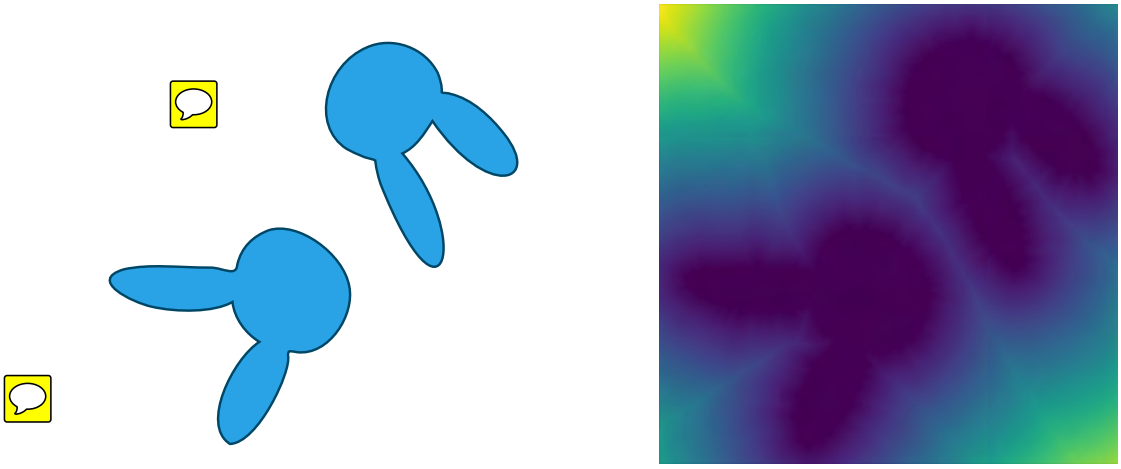


Figure 6.2: The image on the right is a distance transform generated from negative space around two rabbits. The brighter the color of a pixel, the farther it is from the nearest boundary.

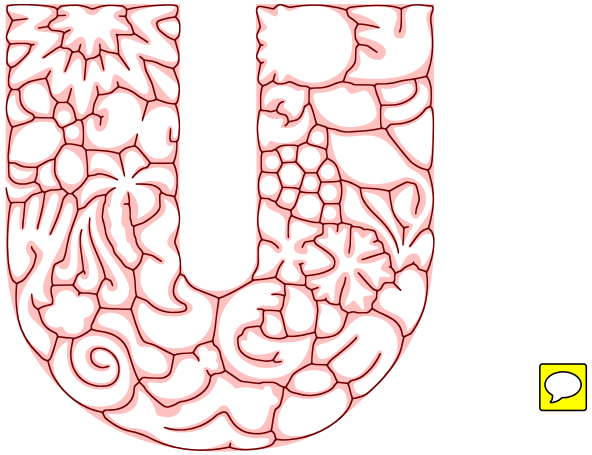


Figure 6.3: A medial axis generated from the negative space of the Unilever packing using Zhang-Suen thinning algorithm [ZS84]. The medial axis contains unnecessarily long branches that touch the element boundaries and inconsequential short branches that should be deleted.

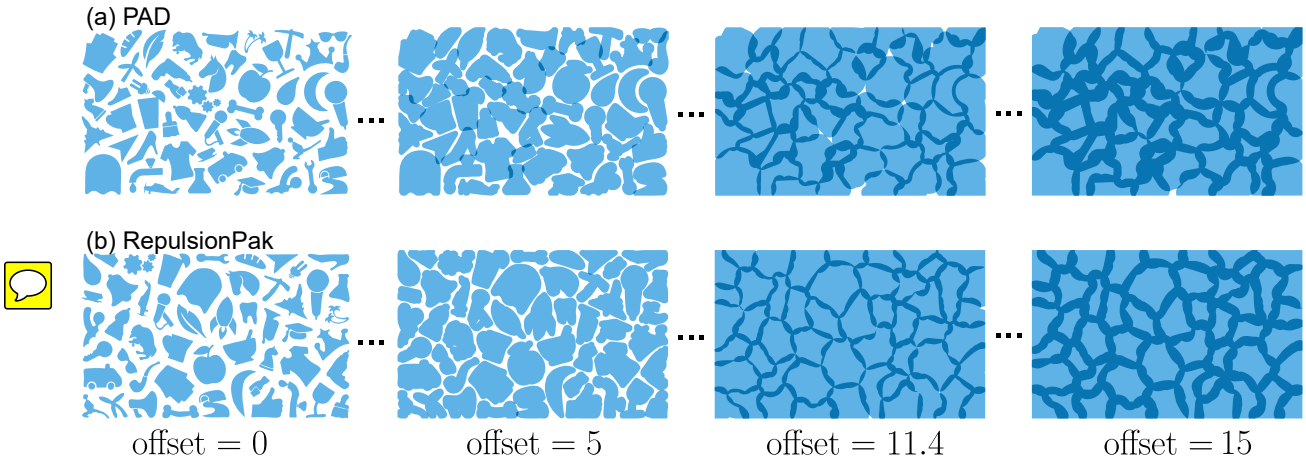


Figure 6.4: An illustration of offsetting elements outward. The packings have $d_{\text{gap}}/2 = 5.7$. At an offset of 5, which is slightly less than $d_{\text{gap}}/2$, the overlap for the PAD packing is 1.462% of the total area, while the overlap for our packing is only 0.039%. At an offset of 11.4, which equals d_{gap}, the PAD packing shows more empty space (1.05%) than RepulsionPak (0.07%). As the offset is increased, overlaps in the PAD packing create channels with uneven widths, whereas ours are more uniform.

But a more even packing will have a shorter tail, indicating a tighter upper bound on gap size, and it will have a larger concentration of density around $d_{\text{gap}}/2$. In Figure 6.1, the histogram for the perfect stripe pattern is a step function that drops to zero at $d_{\text{gap}}/2$. The ideal square packing has a histogram that climbs gently until around $d_{\text{gap}}/2$ before dropping steeply. The other two packings have shallower, smoother histograms. Note also that high values of the distance histogram near $d_{\text{gap}}/2$ correspond to a rapid negative change in the SCP, suggesting a more even packing.

6.3.3 The Overlap Function

The overlap function is a function of a non-negative offset amount r. For any given r, we offset every element by computing its Minkowski sum with a disc of radius r. As r grows, elements will start overlapping; the overlap function measures the total area of these overlaps, normalized by container area, as a function of r. We can also visualize these overlapping areas directly as in Figure 6.4. In a perfect packing, we would expect no overlaps until $r = d_{\text{gap}}/2$, our desired gap distance, at which point overlapping areas would start to grow into channels of roughly even width.

6.4 Comparisons

For all comparisons below, we estimate ~~$d_{\text{gap}}/2$ as~~ the average of skin widths of all elements in a RepulsionPak packing after the simulation stops. We assume the other non-RepulsionPak packing has approximately the same d_{gap}~~/2~~ value ~~since both packings have the identical negative space ratio due to the calibration.~~

Comparison to PAD: Figure 6.5 compares RepulsionPak and PAD [KSH+16]. Packing (a) is a result from the PAD paper; Packing (b) was created with RepulsionPak using the same elements, and calibrated to have the same negative space as (a). Note that the PAD packing actually has several overlapping elements (for example, the tooth and the horse), but white haloes around elements artfully conceal overlaps with little degradation in visual quality. Our packing avoids overlaps by design. The SCP plot in (c) shows that our packing has a lower value at $d_{\text{gap}}/2$, indicating more even negative space, and has a shorter tail, indicating fewer large empty areas. Our result also has a histogram bump around $d_{\text{gap}}/2$, and a lower overlap function.

Comparison to an Artist-Made Packing: In Figure 6.6 we show a RepulsionPak result created using the elements from the artist-made packing. Our packing was calibrated to match the artist's. Looking closely, the artist's packing has a few elements separated by narrow gaps, such as the cherries and the corn on the top left. Our result has fewer large empty gaps, as indicated by a short tail in its SCP. Our result also has a histogram bump around $d_{\text{gap}}/2$, and a lower overlap function. The result shows the effectiveness of the repulsion forces in successfully discovering compatibilities in the element boundaries and filling the space effectively.

Comparison to Rigid Packings: To evaluate the effect of deformation on negative space, we compute all three metrics under increasing values of k_{edg}, the edge force relative strength. Note that we only modify the strengths of edge springs and shear springs, ~~and~~ the strength of negative space springs ~~are~~ unchanged. Increasing k_{edg} allows the element meshes to resist deformation, ultimately approximating a rigid packing algorithm. We created 15 packings, five for each of three values of k_{edg} (5, 250, and 1000). Each packing used 25 elements chosen at random from a library of 60, with no secondary elements. All packings are partly calibrated: they all have the same negative space ratio, but elements are chosen at random and have some variation in their sizes. As shown in Figure 6.7, a low value of k_{edg} leads to greater deformation and steeper SCPs. The more pronounced histogram bump at $r = d_{\text{gap}}/2$ suggests more even negative space. The lower overlap functions indicate fewer narrow gaps between elements.

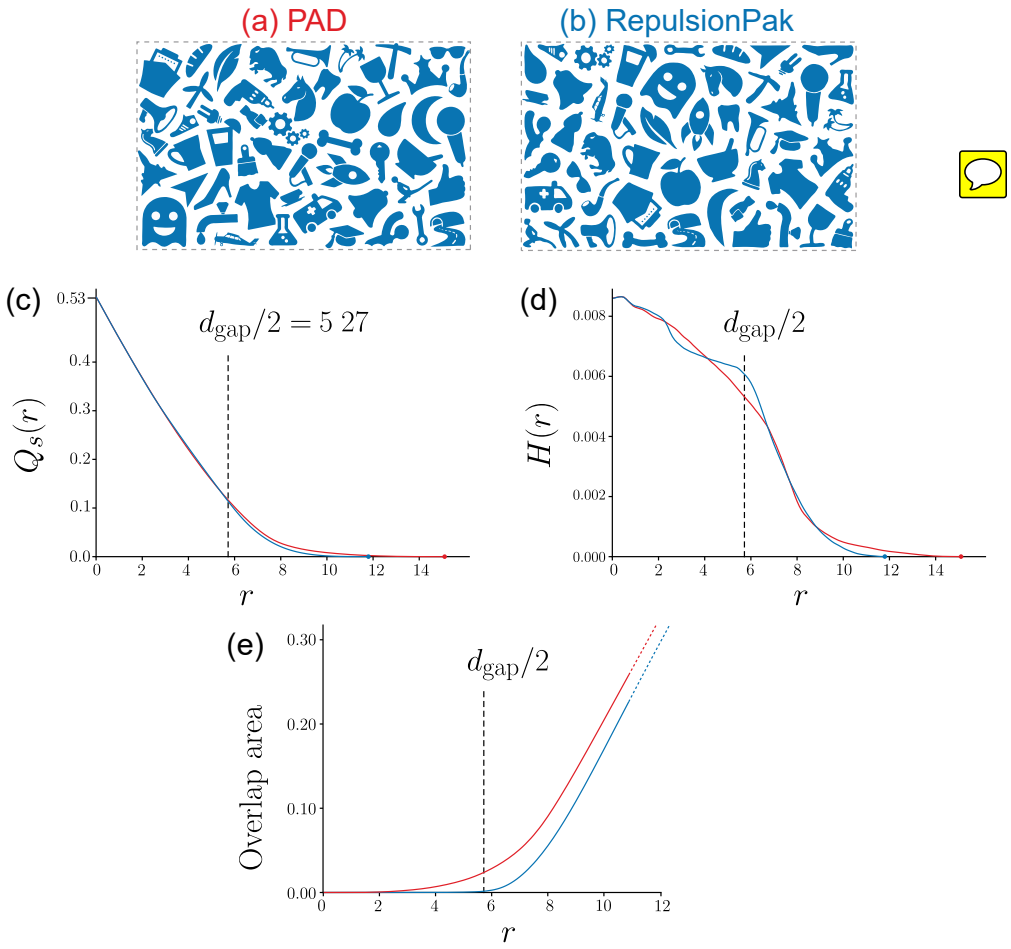


Figure 6.5: A comparison between a PAD packing shown in (a) and a RepulsionPak packing in (b) with their corresponding SCPs (c), distance histograms (d), and overlap functions (e). The PAD and RepulsionPak packings are calibrated to have the same negative space ratio. Our SCP is lower and shorter than the PAD's result, our histogram shows higher concentration around $d_{\text{gap}}/2$, indicating more even negative space, and our packing has a lower overlap function.

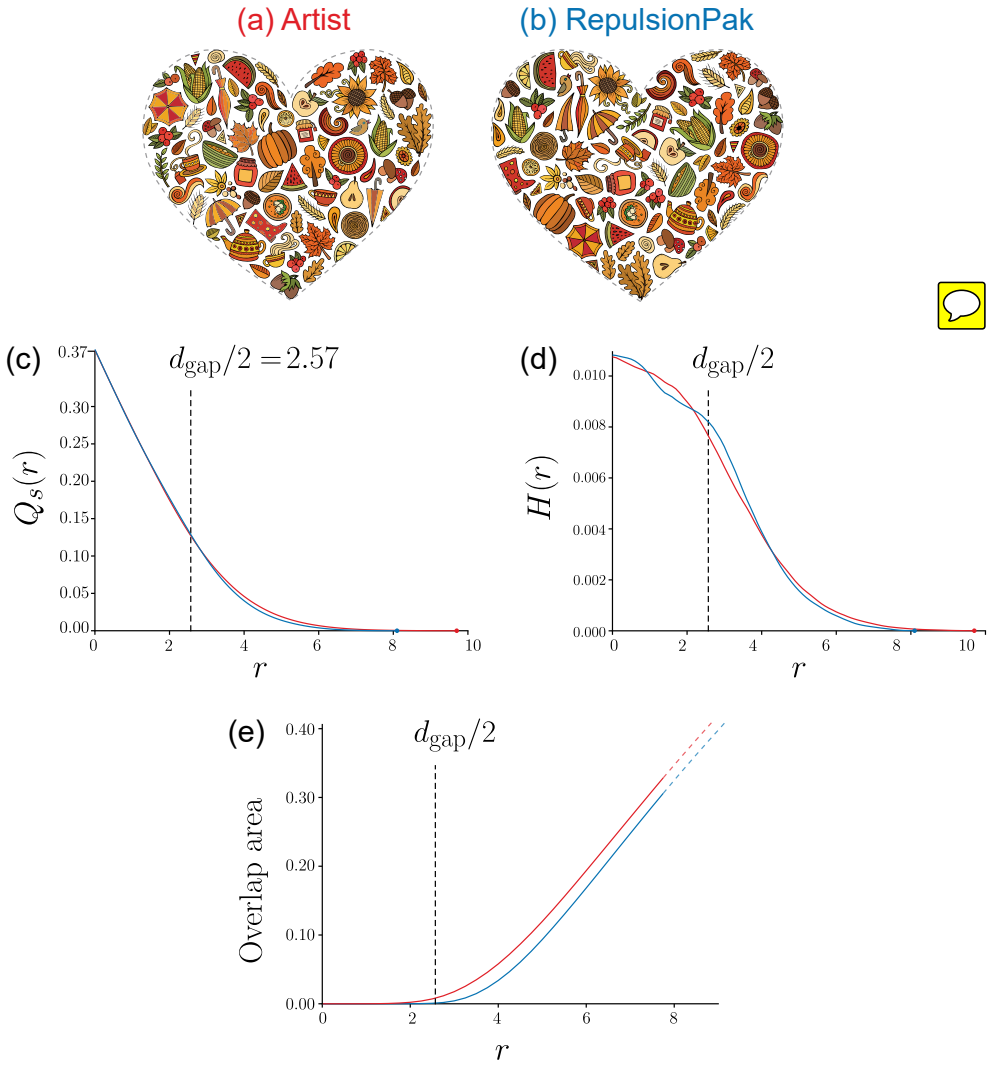


Figure 6.6: A comparison between the artist-made packing (Artist: Balabolka on Shutterstock) shown in (a), and a RepulsionPak packing with the same elements in (b). We plot the corresponding SCPs (c), distance histograms (d), and overlap functions (e). For comparison purposes we remove secondary elements from the artist's packing. Our SCP is lower and shorter than the artist's result, our histogram shows more concentration around $d_{\text{gap}}/2$, and RepulsionPak also has a lower overlap function.

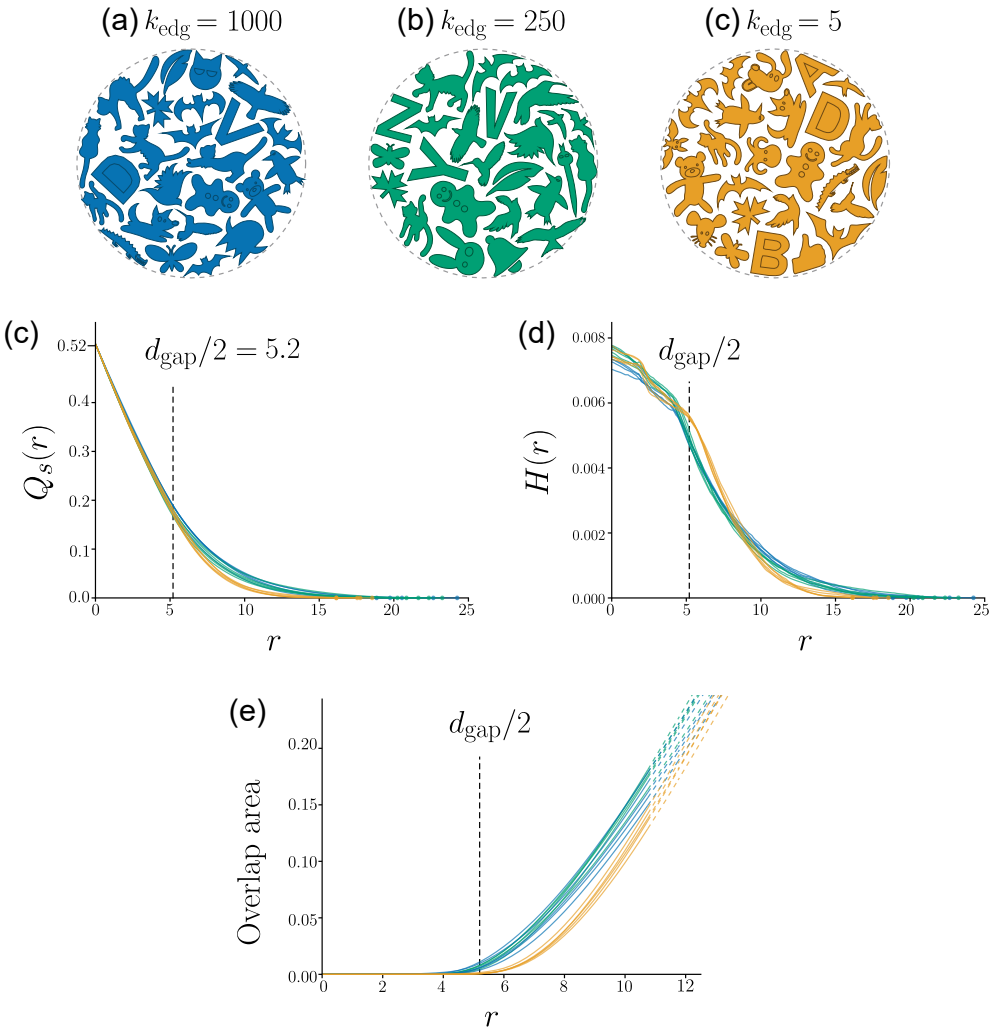


Figure 6.7: A demonstration of the effect of deformation on the evenness of negative space. The packings in (a), (b) and (c) are representative results using three values of the edge force strength k_{edg}, from rigid ($k_{\mathrm{edg}} = 1000$) to moderate ($k_{\mathrm{edg}} = 250$) to deformable ($k_{\mathrm{edg}} = 5$). We construct five random packings for each value of k_{edg}, and plot their SCPs (d), histograms (e), and overlap functions (f). Packings with the most deformation have steeper and shorter SCPs, more histogram concentrations around $d_{\mathrm{gap}}/2$, and lower overlap functions.

6.5 Conclusions

The evenness of negative space is an indicator of the quality of a packing. The more the elements interlock, the more even the negative space. We evaluated the evenness of negative space using spherical contact probabilities, histograms of distance transform, and overlap functions. A packing with a more even negative space has a steeper and shorter SCP, more histogram concentrations around $d_{\text{gap}}/2$, and a lower overlap function.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We presented three deformation-driven packing methods. FLOWPAK is a method to deform and pack long thin elements to follow a vector field. RepulsionPak is a method that uses repulsion forces to pack and deform elements that are represented as mass-spring systems. AnimationPak is an extension of RepulsionPak that packs animated 2D elements, each ~~is~~ an extruded 3D shape in a spacetime domain.

Given a small size element library, we demonstrated that deformation-driven methods can create element compatibilities and fill the container effectively. Repeated elements give a sense of uniformity but deformation creates a sense of variety.

We discussed ~~that~~ the evenness of negative space ~~is~~ an indicator of the quality of a packing. We measured the evenness using three statistical metrics: spherical contact probabilities, histograms of distance transforms, and overlap functions.

7.2 Future Work

We see many possibilities for further improvements to our methods and packing research.

7.2.1 FLOWPAK

- **Element Threading:** In FLOWPAK, an element must completely use up its streamline, causing a few elements to stretch too long. We would like to combine multiple shorter elements by threading them along streamlines, creating a longer *compound element* (Figure 7.1a). First, we need a way to connect the overlapping segments of the elements together to create smooth and plausible transitions. This possibly leads us to a deeper research topic: how to develop a content-aware blending method on vector graphics patterns. Second, we should carefully decide on the selection of elements we have to thread and the number of elements we need. We would like to investigate a greedy approach with backtracking [KP02].

- **Element Branching:** We also would like to support branching structures to ~~resemble~~ floral ornaments (Figure 7.1b). We start ~~with~~ tracing a long streamline that follows the vector field, and then generate shorter streamline branches, which produce blobs that partly overlap the main blob. Similarly, we also need to connect and blend element segments at every intersection. We also should consider the semantics of the element shapes. Not every part of an element can support branching; for example, a leaf cannot be connected with twigs. In another example, a stem or a trunk is an ideal segment where we can glue other elements.

- **High Curvature Streamlines:** FLOWPAK 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 [Ase10].

- **Better Iterative Refinement:** The 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. Another idea is to use RepulsionPak to refine the packing ~~but we~~ need to enforce a constraint on elements' spines to ensure they adhere to the vector field.

- **Automatic Fixed Element Creation:** 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.

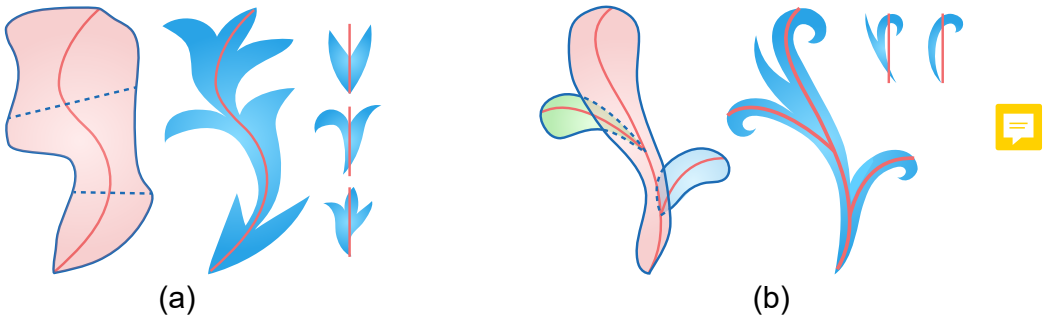


Figure 7.1: (a) Thread multiple elements on a single streamline. (b) Combine multiple streamlines to create branching.

7.2.2 RepulsionPak

- **Improving the Numerical Integrator:** We deliberately chose a simple simulation model based on springs and forward Euler integration, because our main goal was to demonstrate the validity of a deformation-driven approach, and not to contribute a new physical simulation method. Contemporary research has yielded many more sophisticated physical simulation methods, such as Position Based Dynamics [MHHR07], Projective Dynamics [BML+14], and the Finite Element Method. No one method is obviously best suited to this problem, and we intend to experiment with several to investigate if any offers a suitable trade-off between performance and quality.

- **RepulsionPak for Fabrication:** We would like to explore the use of RepulsionPak in a fabrication context. For example, our boundary compatibilities might be used to create a connected object. Alternatively, it would be interesting to 3D print the negative space, which is already connected, leaving holes that surround the element shapes.

- **Vector Graphics Warping:** Our barycentric warping method can introduce undesirable artifacts in highly deformed elements, as in the swallow tails in the left result of Figure 4.9. We would like to explore other methods for warping an element's geometry based on the correspondences between the triangles of its original mesh and the deformed meshes in the final packing, based for example on the work of Jacobson et al. [JBPS11] and Liu et al. [LJG14].

- **Interactive packings:** We would like to develop human-in-the-loop interactions, in which the user and the computer work together to create a desired composition.

Examples of recent work in this style include research by Zehnder et al. [ZCT16], Gieseke et al. [GALF17], and Hsu et al. [HWYZ20], which let the user directly place and manipulate elements while a composition is being created. It would also be interesting to display an interactive packing on a large touch screen, and let people ~~to~~ move elements around ~~by touching and dragging gestures~~.

7.2.3 AnimationPak

- **2D Vector Graphics Interpolation:** Because we use linear interpolation to render an element's shape between slices, we require elements not to undergo changes in topology. More sophisticated representations of vector shapes, such as that of Dalstein et al. [DRvdP15], could support interpolations between slices with complex topological changes. We would also need to synthesize a watertight envelope around the animating element in order to compute overlap and repulsion forces.

- **Dynamic Mesh Resolution:** We would like to improve the performance of the physical simulation. One option may be to increase the resolution of element meshes progressively during simulation. Early in the process, elements are small and distant from each other, so lower-resolution meshes may suffice for computing repulsion forces.

- **Continuous Collision Detection:** Our discrete simulation can miss element overlaps that occur between slices. A more robust continuous collision detection (CCD) algorithm such as that of Brochu et al. [BEB12] could help us find all collisions between the envelopes of spacetime elements.

- **Secondary Elements:** In RepulsionPak ~~[SKA18]~~, an additional pass with small secondary elements had a significant positive effect on the distribution of negative space in the final packing. It may be possible to identify stretches of unused spacetime that can be filled opportunistically with additional elements. The challenge would be to locate tubes of empty space that run the full duration of the animation and have sufficient diameters to accommodate added elements.

- **Animated Containers:** Like the spectral method [DKLS06], and unlike Animosaics [SLK05], AnimationPak can pack animated elements into a static container. We would like to extend our work to also handle animated containers. As demonstrated by the packing with a loose bird in Figure 5.15, elements can vary their slice sizes to adapt to changes in container area. A more challenging solution is adding and

removing elements, as this would certainly affect the initial element placement, which would need to ensure that elements are placed fully inside the spacetime volume of the container.

- **Deformation-Driven 3D Packings:** AnimationPak implements forces and constraints geared towards spacetime animation, but many of the same ideas could be adapted to develop a deformation-driven method for packing purely spatial 3D objects into a 3D container. We would like to evaluate the expressivity and visual quality of deformation-driven 3D packings in comparison to other 3D packing techniques.

- **User Study with Artists:** There are many examples of static two-dimensional packings created by artists, which can serve as inspiration for an algorithm like RepulsionPak. We were only able to find a single example of animated packing from The Simpsons (Figure 5.2). We think its unpopularity is because it is difficult and time-consuming to create by hand. Therefore, we would like to engage with artists to understand the aesthetic value and limitations of AnimationPak.

7.2.4 Packing Evaluation

- **Human Perception of ~~the~~ Evenness:** We would like to conduct experiments that investigate the extent to which quantitative measurements of the evenness of negative space in a packing correlate with the human perception of a packing's quality. In informal evaluations, some viewers found that the packing in Figure 6.6b, created with RepulsionPak, was packed more tightly than the artist's packing in Figure 6.6a, even though both have the same total amount of negative space.

- **Measuring Other Design Principles:** We would like to develop additional metrics to evaluate how well an ornamental design fulfills other design principles. A measure of element deformation in a composition would permit a comparison against future deformation-driven techniques. In Chapter 3, we argue that visual flow and "uniformity amidst variety" are important ~~to~~ attractive packings. In another study, Wong et al. [WZS98] describe basic design principles for decorative arts: repetition, balance, and conformation to geometric constraints. The rigorous expression of aesthetic principles is a fascinating area for future research.

- **Different Offsetting for SCP:** Our validation metrics are all based on Minkowski sums or differences with discs, corresponding to a form of shape offsetting where corners become round. The result of this rounding is visible in our graphs, for

example in the gradual flattening of the SCP for the squares in Figure 6.1e. It would be worthwhile to repeat these measurements using mitered offsetting, and to evaluate whether rounded or mitered offsetting is a closer match to human perceptual judgment of evenness.

- **Better Visualization for SCP:** When comparing calibrated packings, SCPs communicate differences in the evenness of negative space, but the differences between SCPs can be subtle. In addition to distance histograms, we would like to investigate other visualizations of this information that might amplify these differences to make evaluation easier.

- **Evaluating 2D Animated Packings:** All three metrics are only for evaluating 2D packings. While they extend naturally to three purely spatial dimensions, it is not clear whether they can be adapted to the spacetime context. We would like to investigate spatial statistics for the quality of animated packings created by AnimationPak that correlate with human perceptual judgments.

References

[AGYS14] Rinat Abdrashitov, Emilie Guy, JiaXian Yao, and Karan Singh. Mosaic: Sketch-Based Interface for Creating Digital Decorative Mosaics. In *Proceedings of the 4th Joint Symposium on Computational Aesthetics, Non-Photorealistic Animation and Rendering, and Sketch-Based Interfaces and Modeling*, SBIM '14, pages 5–10, New York, NY, USA, 2014. ACM.

[AKA13a] Zainab AlMeraj, Craig S. Kaplan, and Paul Asente. Patch-Based Geometric Texture Synthesis. In *Proceedings of the Symposium on Computational Aesthetics*, CAE '13, pages 15–19, New York, NY, USA, 2013. ACM.

[AKA13b] Zainab AlMeraj, Craig S. Kaplan, and Paul Asente. Towards Effective Evaluation of Geometric Texture Synthesis Algorithms. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, NPAR 13, page 514, New York, NY, USA, 2013. Association for Computing Machinery.

[AKMS20] Paul Asente, Craig Kaplan, Radomír Měch, and Reza Adhitya Saputra. Computerized Generation of Ornamental Designs by Placing Instances of Simple Shapes in Accordance With a Direction Guide, February 11 2020. US Patent 10,559,061.

[Ase10] Paul Asente. Folding Avoidance in Skeletal Strokes. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*. The Eurographics Association, 2010.

[ASF+13] Maya Alsheh Aliy, Johanne Seguin, Aurélie Fischer, Nathalie Mignet, Laurent Wendling, and Thomas Hurtut. Comparison of the Spatial Organization in Colorectal Tumors using Second-Order Statistics and Functional ANOVA. In *2013 8th International Symposium on Image and Signal Processing and Analysis (ISPA)*, pages 257–261, 2013.

[BBT+06] Pascal Barla, Simon Breslav, Joëlle Thollot, François Sillion, and Lee Markosian. Stroke Pattern Analysis and Synthesis. In *Computer Graphics Forum (Proc. of Eurographics 2006)*, volume 25, 2006.

[BEB12] Tyson Brochu, Essex Edwards, and Robert Bridson. Efficient Geometrically Exact Continuous Collision Detection. *ACM Trans. Graph.*, 31(4), July 2012.

[BL15] Thomas F. Banchoff and Stephen T. Lovett. *Differential Geometry of Curves and Surfaces*. Chapman and Hall/CRC, 2015.

[BML+14] Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph.*, 33(4):154:1–154:11, July 2014.

[Bri07] Robert Bridson. Fast Poisson Disk Sampling in Arbitrary Dimensions. In *ACM SIGGRAPH 2007 Sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[BTW09] Thomas Byholm, Martti Toivakka, and Jan Westerholm. Effective Packing of 3-Dimensional Voxel-Based Arbitrarily Shaped Particles. *Powder Technology*, 196(2):139–146, 2009.

[BvMM11] B. Beneš, O. Šťava, R. Měch, and G. Miller. Guided Procedural Modeling. *Computer Graphics Forum*, 30(2):325–334, 2011.

[BW95] Armin Bruderlin and Lance Williams. Motion Signal Processing. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 95, pages 97–104, New York, NY, USA, 1995. Association for Computing Machinery.

[BWL18] Xiaojun Bian, Li-Yi Wei, and Sylvain Lefebvre. Tile-Based Pattern Design with Topology Control. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(1), July 2018.

[CFH+09] M. Cui, J. Femiani, J. Hu, P. Wonka, and A. Razdan. Curve Matching for Open 2D Curves. *Pattern Recogn. Lett.*, 30(1):1–10, January 2009.

[CKHL11] Myung Geol Choi, Manmyung Kim, Kyung Lyul Hyun, and Jehee Lee. Deformable Motion: Squeezing into Cluttered Environments. *Computer Graphics Forum*, 30(2):445–453, 2011.

[Coh92] Michael F. Cohen. Interactive Spacetime Control for Animation. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 92, pages 293–302, New York, NY, USA, 1992. Association for Computing Machinery.

[CSKM13] Sung Nok Chiu, Dietrich Stoyan, Wilfrid S. Kendall, and Joseph Mecke. *Stochastic Geometry and Its Applications*. Wiley Series in Probability and Statistics. Wiley, 2013.

[CSR10] N. Chernov, Yu. Stoyan, and T. Romanova. Mathematical Model and Efficient Algorithms for Object Packing Problem. *Computational Geometry*, 43(5):535 – 553, 2010.

[CSY02] Jonathan Cagan, Kenji Shimada, and Sun Yin. A Survey of Computational Approaches to Three-Dimensional Layout Problems. *Computer-Aided Design*, 34(8):597–611, 2002.

[CZL+15] Xuelin Chen, Hao Zhang, Jinjie Lin, Ruizhen Hu, Lin Lu, Qixing Huang, Bedrich Benes, Daniel Cohen-Or, and Baoquan Chen. Dapper: Decompose-and-Pack for 3D Printing. *ACM Trans. Graph.*, 34(6), October 2015.

[CZX+16] Weikai Chen, Xiaolong Zhang, Shiqing Xin, Yang Xia, Sylvain Lefebvre, and Wenping Wang. Synthesis of Filigrees for Digital Fabrication. *ACM Trans. Graph.*, 35(4):98:1–98:13, July 2016.

[DKLS06] Ketan Dalal, Allison W. Klein, Yunjun Liu, and Kaleigh Smith. A Spectral Approach to NPR Packing. In *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '06, pages 71–78, New York, NY, USA, 2006. ACM.

[DRvdP15] Boris Dalstein, Rémi Ronfard, and Michiel van de Panne. Vector Graphics Animation with Time-Varying Topology. *ACM Trans. Graph.*, 34(4), July 2015.

[Eri05] Christer Ericson. Chapter 5: Basic Primitive Tests. In Christer Ericson, editor, *Real-Time Collision Detection*, The Morgan Kaufmann Series in Interactive 3D Technology, pages 125–233. Morgan Kaufmann, San Francisco, 2005.

[GALF17] Lena Gieseke, Paul Asente, Jingwan Lu, and Martin Fuchs. Organized Order in Ornamentation. In *Proceedings of the Symposium on Computational Aesthetics*, CAE '17, pages 4:1–4:9, New York, NY, USA, 2017. ACM.

[GBLM16] Paul Guerrero, Gilbert Bernstein, Wilmot Li, and Niloy J. Mitra. PATEX: Exploring Pattern Variations. *ACM Trans. Graph.*, 35(4):48:1–48:13, July 2016.

[Gle01] Michael Gleicher. Motion Path Editing. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D 01, pages 195–202, New York, NY, USA, 2001. Association for Computing Machinery.

[Gom84] Ernst Hans Gombrich. *The Sense of Order: A Study in the Psychology of Decorative Art*. Phaidon Press Limited, 1984.

[GSCO07] Ran Gal, Ariel Shamir, and Daniel Cohen-Or. Pose-Oblivious Shape Signature. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):261–271, March 2007.

[GSP+07] Ran Gal, Olga Sorkine, Tiberiu Popa, Alla Sheffer, and Daniel Cohen-Or. 3D Collage: Expressive Non-Realistic Modeling. In *Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '07, pages 7–14, New York, NY, USA, 2007. ACM.

[Hau01] Alejo Hausner. Simulating Decorative Mosaics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 573–580, New York, NY, USA, 2001. ACM.

[HHD03] Stefan Hiller, Heino Hellwig, and Oliver Deussen. Beyond Stippling - Methods for Distributing Objects on the Plane. *Computer Graphics Forum*, 2003.

[HKT10] Edmond S. L. Ho, Taku Komura, and Chiew-Lan Tai. Spatial Relationship Preserving Character Motion Adaptation. *ACM Trans. Graph.*, 29(4), July 2010.

[HLT+09] Thomas Hurtut, Pierre-Edouard Landes, Joëlle. Thollot, Yann Gousseau, Remy Drouillhet, and Jean-François Coeurjolly. Appearance-Guided Synthesis of Element Arrangements by Example. In *Proceedings of the 7th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '09, pages 51–60, New York, NY, USA, 2009. ACM.

[HLW93] 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, pages 197–206, New York, NY, USA, 1993. ACM.

[Hut29] Francis Hutcheson. *An Inquiry Into the Original of Our Ideas of Beauty and Virtue*. J. and J. Knapton and others, 1729.

[HWFL14] Zhe Huang, Jiang Wang, Hongbo Fu, and Rynson W.H. Lau. Structured Mechanical Collage. *IEEE Transactions on Visualization and Computer Graphics*, 20(7):1076–1082, 2014.

[HWYZ20] Chen-Yuan Hsu, Li-Yi Wei, Lihua You, and Jian Jun Zhang. Autocomplete Element Fields. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI 20, New York, NY, USA, 2020. Association for Computing Machinery.

[HZZ11] Hua Huang, Lei Zhang, and Hong-Chao Zhang. Arcimboldo-Like Collage Using Internet Images. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 155:1–155:8, New York, NY, USA, 2011. ACM.

[IMH05] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. As-Rigid-As-Possible Shape Manipulation. In *SIGGRAPH '05*, SIGGRAPH '05, pages 1134–1141, New York, NY, USA, 2005. ACM.

[IMIM08] Takashi Ijiri, Radomír Měch, Takeo Igarashi, and Gavin S. P. Miller. An Example-Based Procedural System for Element Arrangement. *Computer Graphics Forum*, 27:429–436, 2008.

[Ise13] Tobias Isenberg. Evaluating and Validating Non-Photorealistic and Illustrative Rendering. In Paul Rosin and John Collomosse, editors, *Image and Video based Artistic Stylisation*, volume 42 of *Computational Imaging and Vision*, chapter 15, pages 311–331. Springer, London, Heidelberg, 2013.

[JBPS11] Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. Bounded Biharmonic Weights for Real-Time Deformation. *ACM Trans. Graph.*, 30(4):78:1–78:8, July 2011.

[JDM19] Ali Sattari Javid, Lars Doyle, and David Mould. Irregular Pebble Mosaics with Sub-Pebble Detail. In *ACM/EG Expressive Symposium*. The Eurographics Association, 2019.

[JL97] Bruno Jobard and Wilfrid Lefer. Creating Evenly-Spaced Streamlines of Arbitrary Density. In Wilfrid Lefer and Michel Grave, editors, *Visualization in Scientific Computing '97: Proceedings of the Eurographics Workshop in*

Boulogne-sur-Mer France, April 28–30, 1997, pages 43–55, Vienna, 1997. Springer Vienna.

[Joh14] Angus Johnson. Clipper—An Open Source Freeware Library for Clipping and Offsetting Lines and Polygons. http://www.angusj.com/delphi/clipper.php, 2014.

[Kap19] Craig S. Kaplan. Animated Isohedral Tilings. In *Proceedings of Bridges 2019: Mathematics, Art, Music, Architecture, Education, Culture*, pages 99–106, Phoenix, Arizona, 2019. Tessellations Publishing. Available online at http://archive.bridgesmathart.org/2019/bridges2019-99.pdf.

[KHHL12] Manmyung Kim, Youngseok Hwang, Kyunglyul Hyun, and Jehee Lee. Tiling Motion Patches. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA 12, pages 117–126, Goslar, DEU, 2012. Eurographics Association.

[KOHY11] Dongwann Kang, Yong-Jin Ohn, Myoung-Hun Han, and Kyung-Hyun Yoon. Animation for Ancient Tile Mosaics. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, NPAR '11, pages 157–166, New York, NY, USA, 2011. ACM.

[KP02] Junhwan Kim and Fabio Pellacini. Jigsaw Image Mosaics. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 657–664, New York, NY, USA, 2002. ACM.

[KS00] Craig S. Kaplan and David H. Salesin. Escherization. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 499–510, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[KS04] Craig S. Kaplan and David H. Salesin. Dihedral Escherization. In *Proceedings of Graphics Interface 2004*, GI '04, pages 255–262, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine.

[KSH+16] Kin Chung Kwan, Lok Tsun Sinn, Chu Han, Tien-Tsin Wong, and Chi-Wing Fu. Pyramid of Arclength Descriptor for Generating Collage of Shapes. *ACM Trans. Graph.*, 35(6):229:1–229:12, November 2016.

[LBMH19] Yifei Li, David E. Breen, James McCann, and Jessica Hodgins. Algorithmic Quilting Pattern Generation for Pieced Quilts. In *Proceedings of Graphics Interface 2019*, GI 2019. Canadian Information Processing Society, 2019.

[LBW+14] Jingwan Lu, Connelly Barnes, Connie Wan, Paul Asente, Radomír Mech, and Adam Finkelstein. DecoBrush: Drawing Structured Decorative Patterns by Example. *ACM Trans. Graph.*, 33(4):90:1–90:9, July 2014.

[LBZ+11] Yuanyuan Li, Fan Bao, Eugene Zhang, Yoshihiro Kobayashi, and Peter Wonka. Geometry Synthesis on Surfaces Using Field-Guided Shape Grammars. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):231243, February 2011.

[LHVT17] Hugo Loi, Thomas Hurtut, Romain Vergne, and Joelle Thollot. Programmable 2D Arrangements for Element Texture Design. *ACM Trans. Graph.*, 36(4), May 2017.

[LJG14] Songrun Liu, Alec Jacobson, and Yotam Gingold. Skinning Cubic Bézier Splines and Catmull-Clark Subdivision Surfaces. *ACM Trans. Graph.*, 33(6):190:1–190:9, November 2014.

[LML+18] Shih-Syun Lin, Charles C. Morace, Chao-Hung Lin, Li-Fong Hsu, and Tong-Yee Lee. Generation of Escher Arts with Dual Perception. *IEEE Transactions on Visualization and Computer Graphics*, 24(2):1103–1113, 2018.

[LV09] Yu Liu and Olga Veksler. Animated Classic Mosaics from Video. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II*, ISVC '09, pages 1085–1096, Berlin, Heidelberg, 2009. Springer-Verlag.

[MBS+11] Ron Maharik, Mikhail Bessmeltsev, Alla Sheffer, Ariel Shamir, and Nathan Carr. Digital Micrography. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 100:1–100:12, New York, NY, USA, 2011. ACM.

[MCHW18] Y. Ma, Z. Chen, W. Hu, and W. Wang. Packing Irregular Objects in 3D Space via Hybrid Optimization. *Computer Graphics Forum*, 37(5):49–59, 2018.

[MCKM15] Matthias Müller, Nuttapong Chentanez, Tae-Yong Kim, and Miles Macklin. Air Meshes for Robust Collision Handling. *ACM Trans. Graph.*, 34(4):133:1–133:9, July 2015.

[MHHR07] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position Based Dynamics. *J. Vis. Comun. Image Represent.*, 18(2):109–118, April 2007.

[MIA+08] Ross Maciejewski, Tobias Isenberg, William M. Andrews, David S. Ebert, Mario Costa Sousa, and Wei Chen. Measuring Stipple Aesthetics in Hand-Drawn and Computer-Generated Images. *IEEE Computer Graphics and Applications*, 28(2):62–74, March/April 2008.

[MSS+19] Jonàs Martínez, Mélina Skouras, Christian Schumacher, Samuel Hornus, Sylvain Lefebvre, and Bernhard Thomaszewski. Star-Shaped Metrics for Mechanical Metamaterial Design. *ACM Trans. Graph.*, 38(4), July 2019.

[MWLT13] Chongyang Ma, Li-Yi Wei, Sylvain Lefebvre, and Xin Tong. Dynamic Element Textures. *ACM Trans. Graph.*, 32(4), July 2013.

[MWT11] Chongyang Ma, Li-Yi Wei, and Xin Tong. Discrete Element Textures. In *SIGGRAPH '11*, SIGGRAPH '11, pages 62:1–62:10, New York, NY, USA, 2011. ACM.

[OS88] Stanley Osher and James A. Sethian. Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *J. Comput. Phys.*, 79(1):12–49, November 1988.

[Osh17] Masaki Oshita. Lattice-Guided Human Motion Deformation for Collision Avoidance. In *Proceedings of the Tenth International Conference on Motion in Games*, MIG 17, New York, NY, USA, 2017. Association for Computing Machinery.

[PS06] Hans Pedersen and Karan Singh. Organic Labyrinths and Mazes. In *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '06, pages 79–86, New York, NY, USA, 2006. ACM.

[PYW14] Chi-Han Peng, Yong-Liang Yang, and Peter Wonka. Computing Layouts with Deformable Templates. *ACM Trans. Graph.*, 33(4):99:1–99:11, July 2014.

[PZ07] Jonathan Palacios and Eugene Zhang. Rotational Symmetry Field Design on Surfaces. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[Sch90] Philip J. Schneider. An Algorithm for Automatically Fitting Digitized Curves. In Andrew S. Glassner, editor, *Graphics Gems*, pages 612–626. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[Sec02] Adrian Secord. Weighted Voronoi Stippling. In *Proceedings of the 2nd International Symposium on Non-Photorealistic Animation and Rendering*, NPAR 02, page 3743, New York, NY, USA, 2002. Association for Computing Machinery.

[SKA18] Reza Adhitya Saputra, Craig S. Kaplan, and Paul Asente. RepulsionPak: Deformation-Driven Element Packing with Repulsion Forces. In *Proceedings of the 44th Graphics Interface Conference*, GI '18. Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine, 2018.

[SKA19] Reza Adhitya Saputra, Craig S. Kaplan, and Paul Asente. Improved Deformation-Driven Element Packing with RepulsionPak. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–1, 2019.

[SKA20] Reza Adhitya Saputra, Craig S. Kaplan, and Paul Asente. AnimationPak: Packing Elements with Scripted Animations. In *Proceedings of the 46th Graphics Interface Conference*, GI '20. Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine, 2020.

[SKAM17] Reza Adhitya Saputra, Craig S. Kaplan, Paul Asente, and Radomír Měch. FLOWPAK: Flow-Based Ornamental Element Packing. In *Proceedings of the 43rd Graphics Interface Conference*, GI '17, pages 8–15. Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine, 2017.

[SLK05] Kaleigh Smith, Yunjun Liu, and Allison Klein. Animosaics. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, pages 201–208, New York, NY, USA, 2005. ACM.

[TRdAS07] Christian Theobalt, Christian Rössl, Edilson de Aguiar, and Hans-Peter Seidel. Animation Collage. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA 07, page 271280, Goslar, DEU, 2007. Eurographics Association.

[VGB+14] J. Vanek, J. A. Garcia Galicia, B. Benes, R. Měch, N. Carr, O. Šťava, and G. S. Miller. PackMerger: A 3D Print Volume Optimizer. *Computer Graphics Forum*, 33(6):322332, September 2014.

[WK88] Andrew Witkin and Michael Kass. Spacetime Constraints. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 88, pages 159–168, New York, NY, USA, 1988. Association for Computing Machinery.

[WP95] Andrew Witkin and Zoran Popović. Motion Warping. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 95, pages 105–108, New York, NY, USA, 1995. Association for Computing Machinery.

[WZS98] Michael T. Wong, Douglas E. Zongker, and David H. Salesin. Computer-Generated Floral Ornament. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 423–434, New York, NY, USA, 1998. ACM.

[XK07] Jie Xu and Craig S. Kaplan. Calligraphic Packing. In *Proceedings of Graphics Interface 2007*, GI '07, pages 43–50, New York, NY, USA, 2007. ACM.

[XM09] Ling Xu and David Mould. Magnetic Curves: Curvature-Controlled Aesthetic Curves Using Magnetic Fields. In *Proceedings of the Fifth Eurographics Conference on Computational Aesthetics in Graphics, Visualization and Imaging*, Computational Aesthetics'09, pages 1–8, Aire-la-Ville, Switzerland, Switzerland, 2009. Eurographics Association.

[XM15] Ling Xu and David Mould. Procedural Tree Modeling with Guiding Vectors. *Computer Graphics Forum*, 34(7):47–56, October 2015.

[YSC20] Christopher Yu, Henrik Schumacher, and Keenan Crane. Repulsive Curves, 2020.

[ZCR+16] Changqing Zou, Junjie Cao, Warunika Ranaweera, Ibraheem Alhashim, Ping Tan, Alla Sheffer, and Hao Zhang. Legible Compact Calligrams. *ACM Trans. Graph.*, 35(4):122:1–122:12, July 2016.

[ZCT16] Jonas Zehnder, Stelian Coros, and Bernhard Thomaszewski. Designing Structurally-Sound Ornamental Curve Networks. *ACM Trans. Graph.*, 35(4):99:1–99:10, July 2016.

[ZS84] TY Zhang and Ching Y. Suen. A Fast Parallel Algorithm for Thinning Digital Patterns. *Communications of the ACM*, 27(3):236–239, 1984.