Deformation-Driven Element Packing

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

© Reza Adhitya Saputra 2020

Examining Committee Membership

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

Supervisor: Craig S. Kaplan
Associate Professor, School of Computer Science,
University of Waterloo

Internal Member: Christopher Batty
Associate Professor, School of Computer Science,
University of Waterloo

Internal Member: Daniel Vogel
Associate Professor, School of Computer Science,
University of Waterloo

Internal-External Member: Stephen A. Vavasis
Professor,
Department of Combinatorics and Optimization,
University of Waterloo

External Examiner: Karan Singh
Professor, Department of Computer Science,
University of Toronto

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. 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, Dr. Paul Asente, and Dr. Radomír Měch.

FLOWPAK in Chapter 3 was initially conducted with Dr. Paul Asente, and Dr. Radomír Měch at Adobe Research in San Jose, California. FLOWPAK is also discussed 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

A packing is an arrangement of geometric elements within a container region in the plane. Elements are united to communicate the overall container shape, but each is large enough to be appreciated individually. Creating a packing is challenging since an artist should arrange compatible elements so that their boundaries interlock with each other. This thesis presents three packing methods that create element compatibilities using shape deformation. The first method, FLOWPAK, 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, RepulsionPak, utilizes repulsion forces to pack elements, each represented as a mass-spring system, allowing them to deform to achieve a better fit with their neighbours and the container. The last method, AnimationPak, 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 teaching me valuable lessons.
- Bill Cowan, for his advice.
- Christopher Batty, Dan Vogel, for being my PhD committee members.
- Stephen Vavasis, for being my thesis internal-external member.
- Karan Singh, for being my external examiner.
- Matt Thorne, Alex Pytel, and Tyler Nowicki, for all the lunches together.
- Russ Jones, for permission to use the fish art in Figure 1.3d.
- Yusuf Umar, for designing ornamental elements and the box bird in Figure 3.16.
- All my running friends in Kitchener-Waterloo and ultra running community in Ontario, for teaching me on how to keep grinding.

Dedication

For my Mom, Dad, and sister.

Table of Contents

List of Figures **xii**

List of Tables **xvi**

1 Introduction **1**

1.1 Problem Formulation 7

1.2 Thesis Contributions 7

2 Related Work **9**

2.1 Mosaics 10

2.2 2D Packings 14

2.3 Tilings 17

2.4 Discrete Texture Synthesis 19

2.5 3D Packings 21

2.6 Packings on Surfaces 22

3 FLOWPAK: Flow-Based Ornamental Element Packing **24**

3.1 Introduction 24

3.2 Related Work 24

3.3 Approach 25

3.3.1 Target Containers 26

3.3.2 Ornamental Elements and LR functions 28
3.3.3 Creating Vector Fields and Tracing Streamlines 29
3.3.4 Sub-Region Blobs 30
3.3.5 Shape Matching and Deformation 31
3.3.6 Iterative Refinement 32
3.4 Implementation and Results 38
3.5 Conclusions and Future Work 39

4 RepulsionPak: Deformation-Driven Element Packing with Repulsion Forces 44
4.1 Introduction 44
4.2 Related Work 45
4.3 System Overview 46
4.4 Preprocessing 48
4.5 Simulation 50
4.5.1 Element Growth 54
4.5.2 Stopping Criteria 55
4.6 Secondary Elements 55
4.7 Shape Matching 56
4.8 Implementation and Results 61
4.9 Conclusions and Future Work 63

5 AnimationPak: Packing Elements with Scripted Animations 69
5.1 Introduction 69
5.2 Related Work 70
5.3 Animated Elements 74
5.3.1 Spacetime Extrusion 74
5.3.2 Animation 75

5.4 Initial Configuration 77
5.5 Simulation 80
5.5.1 Spatial Queries 82
5.5.2 Slice Constraints 83
5.5.3 Element Growth and Stopping Criteria 86
5.6 Rendering 87
5.7 Implementation and Results 88
5.8 Conclusions and Future Work 93

6 Quantitative Metrics for 2D Packings **99**
6.1 Introduction 99
6.2 Related Work 100
6.3 Quantitative Metrics 101
6.3.1 Spherical Contact Probability 101
6.3.2 Histograms of the Distance Transform 102
6.3.3 The Overlap Function 104
6.4 Comparisons 104
6.5 Conclusions and Future Work 111

7 Conclusions and Future Work **113**
7.1 Conclusions 113
7.2 Future Work 114
7.2.1 Spacious Element Arrangements 114
7.2.2 Element Construction 114
7.2.3 Beyond 2D 116
7.2.4 Collaborations with Artists 117
7.2.5 Packing Evaluation 119

References **120**

APPENDICES **133**

A Animation Videos **134**

List of Figures

1.1 Unilever logo packing and its negative space 2
1.2 Primary and secondary elements 2
1.3 Packings with flow visual styles 5
1.4 A land art packing 6

2.1 A categorization of element arrangements 10
2.2 Three popular methods in mosaicing and packings 11
2.3 An illustration of Lloyd's method. 12
2.4 Stippling artwork and traditional mosaics 13
2.5 Mosaics generated using generalized Lloyd's methods 14
2.6 Vertumnus painting by Giuseppe Arcimboldo 16
2.7 Packings generated by JIM and PAD 16
2.8 An Arcimboldo-inspired 2D packing 17
2.9 A calligraphy packing of an "elephant" 18
2.10 An office layout generated by packing deformable rectangle and polyomino elements 18
2.11 Tilings 20
2.12 A discrete texture 21
2.13 Examples of 3D packings and packings on surfaces 23

3.1 FLOWPAK pipeline 27

3.2 Ornamental elements and their LR functions 29
3.3 The streamline tracing process 30
3.4 Element deformation 32
3.5 Shifting a streamline 35
3.6 Tracing a shortest path 36
3.7 Stretching an element 37
3.8 Rotating an element 37
3.9 Reflecting an element 37
3.10 Packings of lion and unicorn 40
3.11 A packing of a rhinoceros 40
3.12 A packing of a bear with leaf elements 41
3.13 A packing of a cat 41
3.14 A packing of a dog 42
3.15 A packing of a bear with a few elements created from negative space . . . 42
3.16 A packing of a bird 43

4.1 RepulsionPak pipeline 47
4.2 Element deformation 48
4.3 Element discretization 49
4.4 A round element is placed near a sharp convex corner of the container . . . 56
4.5 A local shape descriptor for shape matching 58
4.6 A packing without the use of shape matching 59
4.7 A demonstration of shape matching to place elements 59
4.8 Global and local shape matching 60
4.9 A packing of a cat 63
4.10 Three packings created using RepulsionPak:
Birds, Bats, and Butterflies 64
4.11 A packing that demonstrates torsional forces 65

4.12 A paisley-inspired toroidal packing 65
4.13 Two examples of how extreme parameter values can lead to low-quality results 66
4.14 Artist-made Packings using RepulsionPak 67
4.15 A RepulsionPak packing printed on a t-shirt 68

5.1 An animated packing of copies of Lisa Simpson 71
5.2 A classification of animated packings 72
5.3 The creation of a discretized spacetime element 76
5.4 A spacetime element with a scripted animation 76
5.5 A 2D illustration of a guided element 78
5.6 An illustration of the AnimationPak simulation process 79
5.7 An illustration of the repulsion forces 81
5.8 An illustration of the temporal forces 82
5.9 An envelope and a collision grid 83
5.10 An illustration of an acceptable and unacceptable configurations of an element 84
5.11 Constraints 85
5.12 Element growths 87
5.13 An illustration of rendering spacetime elements 88
5.14 An animated packing of aquatic fauna 90
5.15 An animated packing of a snake chases a bird 91
5.16 An animated packing of the penguins-to-giraffes illusion 91
5.17 A traveling bird passing through a packing of birds 95
5.18 A packing of lion's mane 95
5.19 A comparison between CAVD, The spectral approach, and AnimationPak 96
5.20 A comparison between RepulsionPak and AnimationPak 97
5.21 The effect of adjusting time spring stiffness 97

5.22 A failure case for AnimationPak 98

6.1 An illustration of calculating $Q_s(r)$ 102

6.2 Spherical contact probabilities and distance histograms for reference packings 103

6.3 An example of distance transform of negative space 105

6.4 An illustration of offsetting elements outward 106

6.5 A comparison between a PAD packing and a RepulsionPak packing 108

6.6 A comparison between the artist-made packing and a RepulsionPak packing 109

6.7 A demonstration of the effect of deformation on the evenness of negative space 110

6.8 An example of a medial axis of negative space 112

7.1 Modular elements 115

7.2 Element threading and branching 116

7.3 Element arrangements in bas-reliefs 117

7.4 A doodle mural by Joe Whale 118

List of Tables

4.1 Data and statistics for the RepulsionPak results 62

5.1 Data and statistics for the AnimationPak results 89

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 geometry, or man-made objects. 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-shaped 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. On the other hand, elements can be interpreted as *positive space*. In this thesis, we narrow our focus to packings without element overlaps, which makes it easier to discuss positive and negative space without double counting the positive space.

The evenness of negative space plays an important role in packings. Initially, an artist should arrange elements in such way that their boundaries interlock with each other, causing the separation between neighbouring elements to become roughly the same everywhere. We refer to elements in the resulting packing as *primary elements*. However, the interlocking of primary elements is imperfect, and artists will frequently place smaller *secondary elements* such as triangles or circles to reduce remaining large gaps, as shown in the autumn-themed packing in Figure 1.2.

Design Principles: In studying packing designs, we have identified five high-level principles that are important to the construction of packings. A few of them are adapted from work by Wong et al. [WZS98]. These principles are design considerations that cover

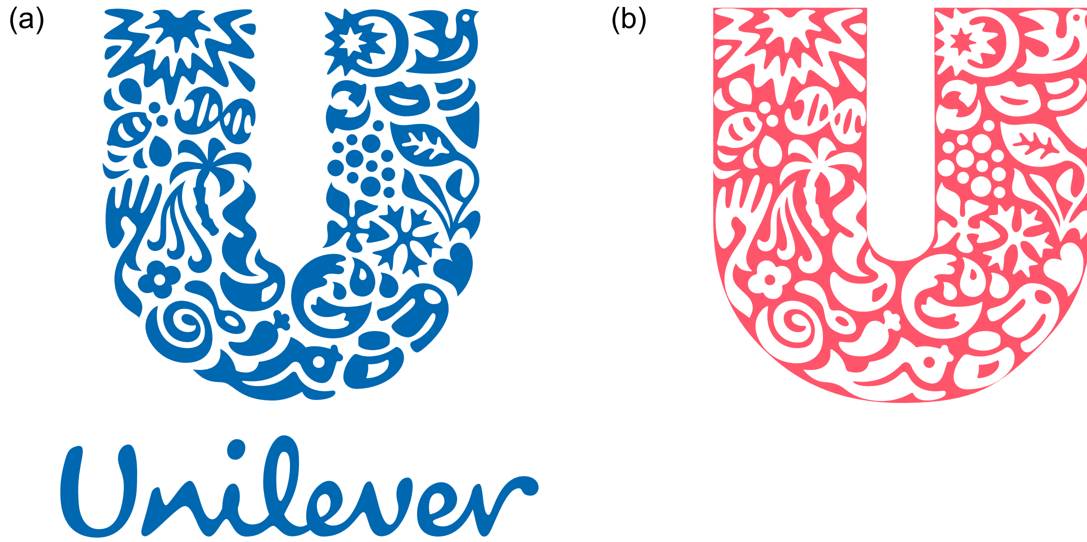


Figure 1.1: (a) The Unilever logo packing consists of 25 elements arranged inside an estimated U-shaped container. (b) The red region, consisting of the remaining area not claimed by the elements, is referred to as negative space.

Figure 1.2: An autumn-themed packing created by Balabolka on Shutterstock. (a) The packing with primary elements only. (b) Secondary elements are added.

a broad range of packing styles. A packing algorithm need not satisfy all five of them. We discuss these principles below:

- **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). The primary elements in Figure 1.2 come in a range of sizes, but none so extreme that they stand out. Additionally, interlocking elements lead to channels of negative space that are roughly constant in width.
- **Flow.** In local parts of a composition, the elements are oriented to communicate a sense of directionality or flow. Examples in Figure 1.3 exhibit some amount of flow. In the dog, many elements appear to flow outward from the flower in the centre of the torso, and then up the neck and down into the legs. The scales and other elements on the fish flow along the length of its body. In the lion and skull, elements flow horizontally outward from a central axis of symmetry, suggesting fur in the case of the lion. Another example is shown in Figure 1.4, which is a packing of leaves that are oriented toward the centre of the container. 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 wrote eloquently on the role of variation in design [Gom84]. In the context of procedural art, Galanter discussed the trade off between order and disorder [Gal03]. The Unilever packing in Figure 1.1 has the elements drawn with a similar curvy design. In Figure 1.2, the elements are autumn-themed shapes and are convex or near convex. In Figure 1.3, the dog's spirals and the fish's scales both obey this principle. The lion and skull do as well, except that half of the elements are reflected copies of the other half, across a vertical line through the centre of the composition. This repetition emphasizes the bilateral symmetry in the design. The land art packing in Figure 1.4 has the most uniformity of these examples: the leaves have similar shapes and the composition conveys the radial symmetry. The artist added some amount of variety by using different leaf sizes and colours.
- **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 1.3 has eyes drawn in as unique elements

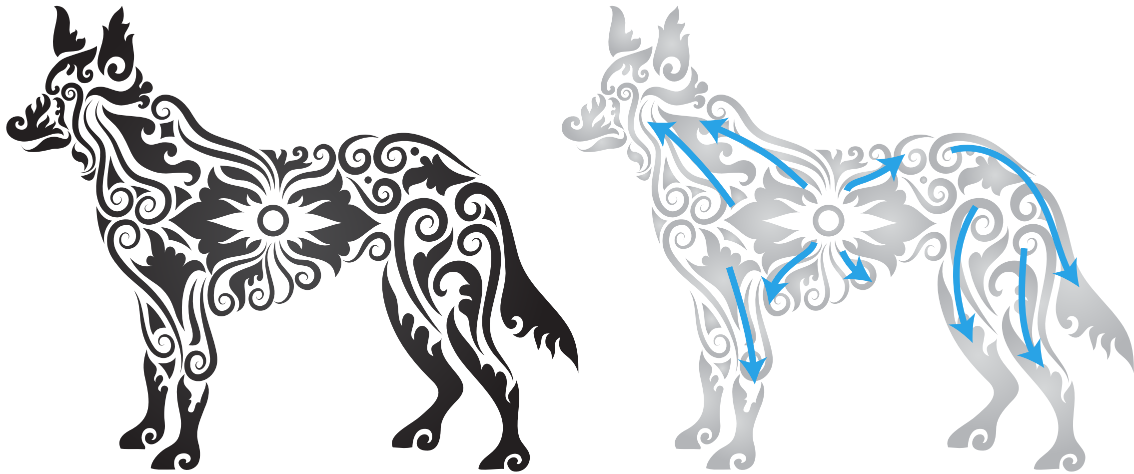
(the dog's eye is expressed via a carefully placed spiral). Other situations that call for specialized shapes include the dog's paws, the fish's teeth and fins, the lion's eyes and nose, and the skull's teeth. Sharp variation in the balance of positive and negative space can also be used to emphasize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

- **Boundaries.** In many ornamental packings, elements are carefully arranged to conform to and emphasize container boundaries. Elements in the Unilever packing (Figure 1.1) and the autumn-themed packing (Figure 1.2) demonstrate this principle most clearly: we can easily fill in the gaps between elements to form a mental image of a continuous outline. In Figure 1.3a, the dog's elements are also well aligned to indicate the container shape. However, this rule is not universal. The lion packing in Figure 1.3b artfully subverts it with elements that flow outward to an indistinct boundary, helping to convey the appearance of fur.

There has been a moderate amount of past research in computer graphics, particularly in the field of non-photorealistic rendering, on the generation of packings, or mosaics. See 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 *create* compatible elements through deformation; these deformed elements can adapt to the shapes of neighbouring 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 require only a small library of element shapes. Deformation-driven methods also allow us to work toward the principle of uniformity amidst variety. We can



(a)

(b) (c) (d)

Figure 1.3: Packings with flow-based visual styles: (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);

Figure 1.4: A land art packing that is composed of leaves, created by James Brunt. Each leaf is oriented toward the centre of the circle.

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.

In this thesis, many of our techniques are inspired by physical simulations. We create an element representation that can be deformed by pseudo-physical forces. Our goal is not to contribute to research on physics-based animation, but to use a physical simulation as an optimization process that allows us to reach a physical equilibrium where the resulting packing has approximately uniform distribution of both positive and negative space.

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 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.

1.1 Problem Formulation

We define the general packing problem to be solved as follows. The user provides several general inputs, although an individual packing method requires additional information from time to time. We discuss these inputs below:

1. A library of elements. Each is a collection of open, closed, or intersecting polygonal paths. An element can contain additional information, such as colours, a preferred orientation, or a scripted animation.
2. One or more target containers. Each container is a closed non-self-intersecting curve to be filled with elements. A container can optionally have internal holes.
3. An optional set of fixed elements that we transfer directly to the result.

Our goal is to fill each container with elements, trying to satisfy several considerations:

1. Balance out the element distribution and make the widths of negative space be as uniform as possible.
2. Deform elements in a controllable way, trading off between the preservation of element shapes and the creation of shape compatibilities.
3. Use the container space efficiently by having as little negative space as possible.
4. Conform to container boundaries.

1.2 Thesis Contributions

In this thesis, we develop three specific deformation-driven packing methods, and present a study to investigate the quantitative evaluation of packing quality in greater detail:

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 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

An element arrangement is defined as a distribution of non-overlapping elements in the plane. Decades of research in non-photorealistic rendering (NPR) have produced many techniques for producing computer-generated element arrangements. Based on their designs, categories of elements arrangements include packings, tilings, and discrete textures (Figure 2.1). The categorization can be further expanded but we only choose ones that are related to this thesis. We further break down the packing category into several subcategories: mosaics, 2D packings, 3D packings, packings on surfaces, and animated packings. This chapter covers the entire categorization except animated packings, which are discussed in Chapter 5. Most packing techniques have the goal to generate overlap-free packings, but in reality, it is a difficult task. A few computer-generated packings shown in this chapter contain overlaps to some degree due to the challenge of finding and aligning compatible elements.

Many packing algorithms can also be grouped according to their main algorithmic techniques (Figure 2.2). The first category, Lloyd's method, iteratively refines an initial element arrangement to generate a more even distribution. The second category, data-driven methods, places elements one by one. For each step, an element is taken from an element library that is compatible with previously placed elements and the container boundary. The last category, deformation-driven methods, attempts to create element compatibilities through shape deformation.

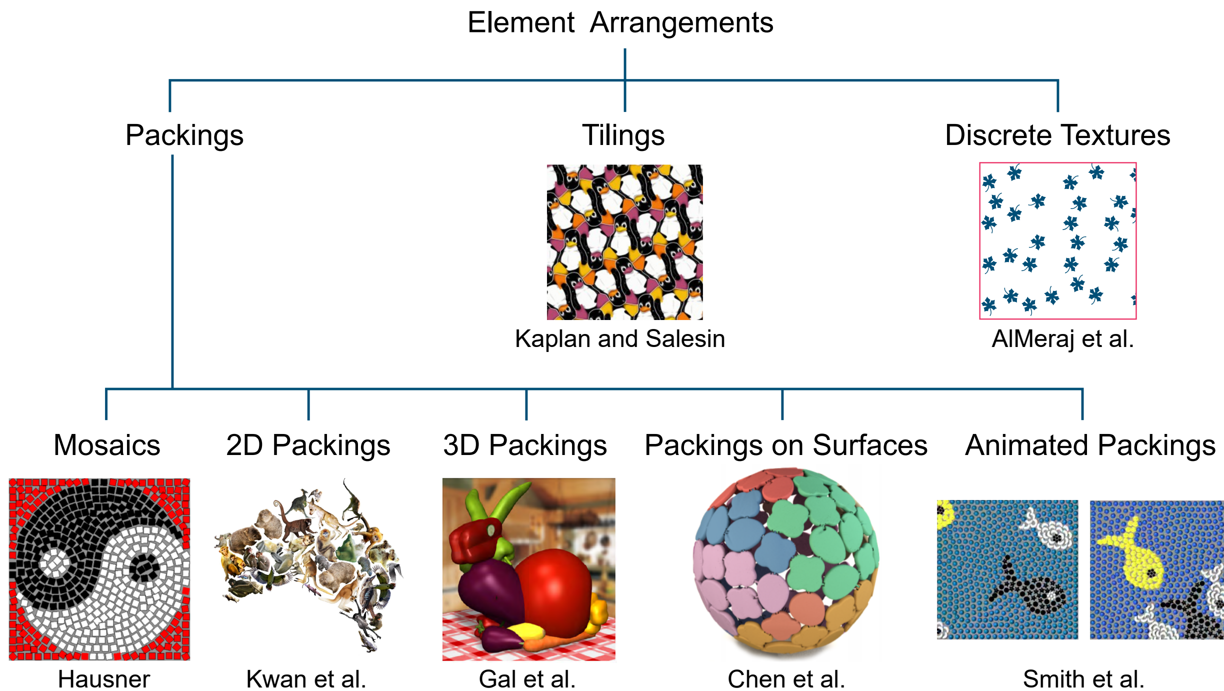


Figure 2.1: A simplified categorization of element arrangements.

2.1 Mosaics

Mosaics are 2D arrangements that are popular for decorating floors or walls. In traditional mosaics, elements are made of stone, ceramics, or glass. The majority of these mosaicing techniques use Lloyd's method to distribute elements via rigid transformations (Figure 2.2a). Originally, Lloyd's method was introduced in computer graphics by McCool and Fiume to distribute point elements [MF92]. It was later generalized to distribute convex and concave elements.

We first discuss the simplest variant of Lloyd's method that distributes point elements, as shown in Figure 2.3. 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, meaning that the point elements are located at the centroids of the Voronoi cells. The final structure is called a *centroidal Voronoi diagram* (CVD).

When computing centroids, a standard CVD assumes the area density is uniform ev-

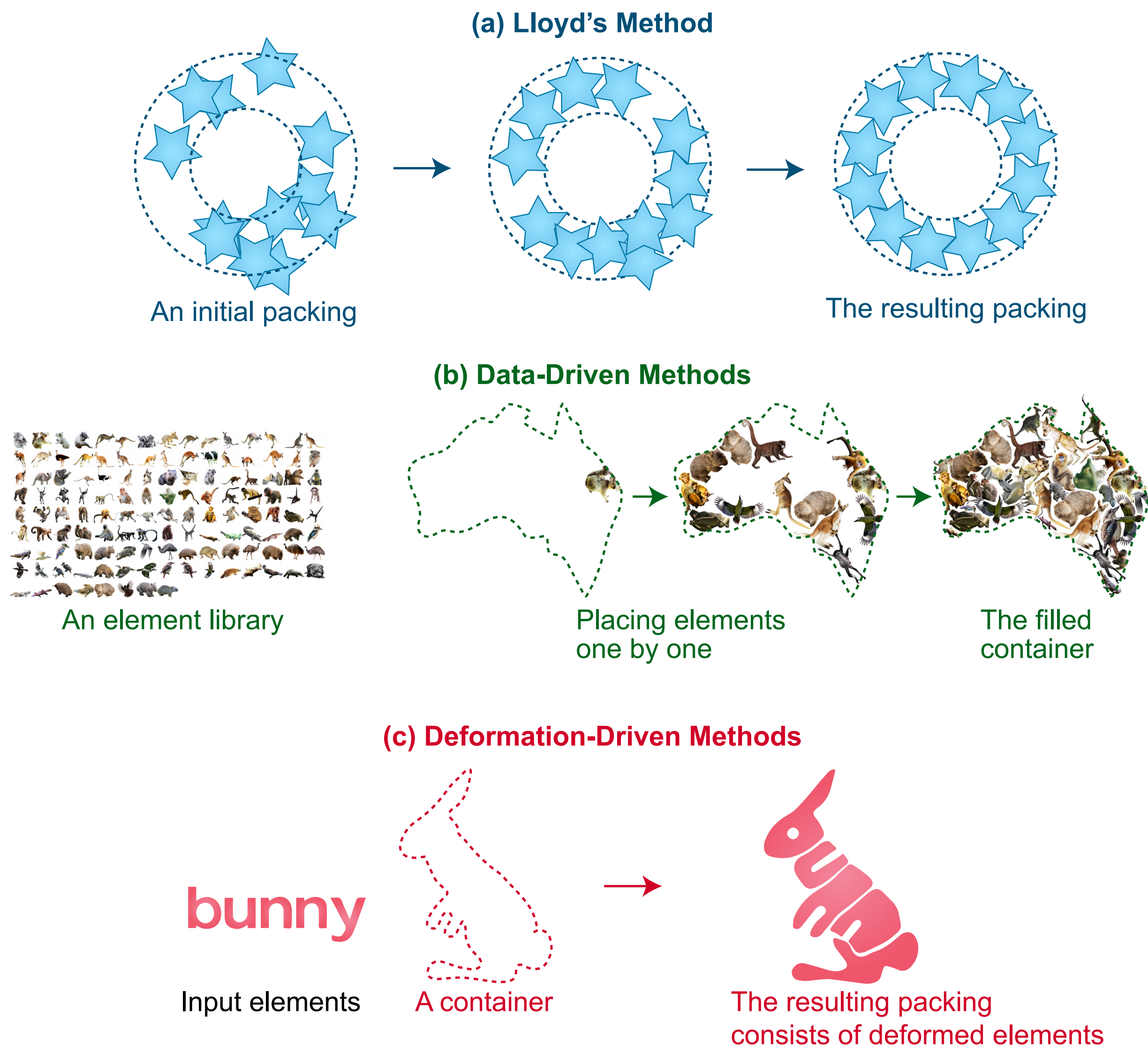


Figure 2.2: Three popular methods in mosaicing and packings. (a) Lloyd's method starts with an initial distribution of elements then iteratively displaces them to generate a more even packing. (b) A data-driven method places elements one by one until the container is filled. For each step, it selects an element from an element library that is compatible with already placed elements and the container. (c) A deformation-driven method attempts to create element compatibility through deformation. The illustrations are adapted from The Spectral Approach [DKLS06], Pyramid of Arclength Descriptors [KSH+16], and Legible Compact Calligrams [ZCR+16].

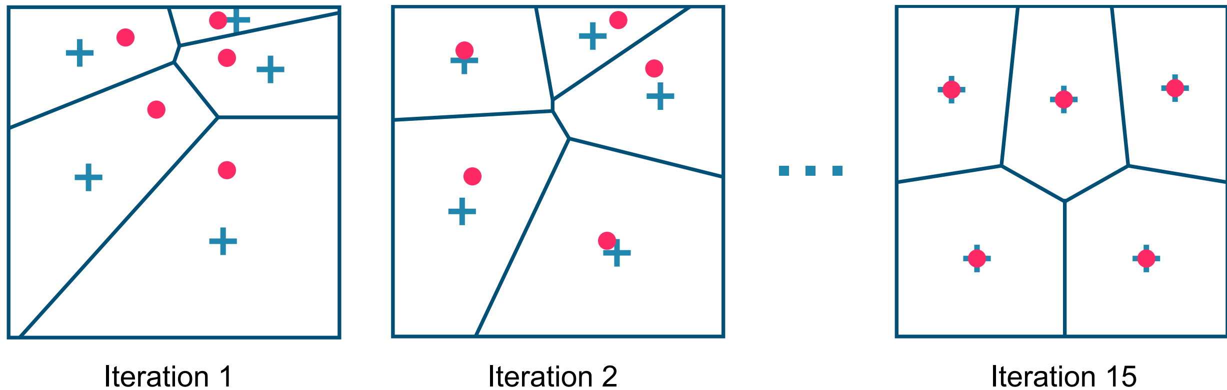


Figure 2.3: An illustration of point-based Lloyd's method. A Voronoi diagram is generated from the input point elements (drawn as red dots). We then move the point elements to the centroids of Voronoi cells (drawn as plus signs). The process is repeated until convergence, producing a centroidal Voronoi diagram. Figure source is Wikipedia, drawn by Dominik Moritz under CC0 1.0.

erywhere. Secord [Sec02] computed a weighted CVD so that the resulting point element distribution resembles a stippling artwork (Figure 2.5a). His method incorporates the pixel intensities of an input image to construct a density field. This alters the centroid calculation so that point elements are attracted to darker regions.

A standard CVD is generated using Euclidean distance, but other metrics lead to different shapes of Voronoi cells. Hausner [Hau01] simulated the appearance of traditional mosaics by using Manhattan distance, producing Voronoi cells that 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 2.5b). However, a standard CVD computes only the positions of the square elements, and Hausner 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 overlaps. More recently, Doyle et al. [DACM19] and Javid et al. [JDM19] generated pebble mosaics using a superpixel image segmentation [ASS+12], which is a variant of Lloyd's method that blends an Euclidean metric with a CIELAB colour metric. They later elongated the Voronoi cells to follow the gradient of the input image, and smoothed out the cell boundaries. The resulting Voronoi cells resemble smooth elongated pebble shapes.

We now discuss the computation of Voronoi diagrams that is generalized to any shapes for which a distance measurement exists. Hiller et al. [HHD03] extended Lloyd's method

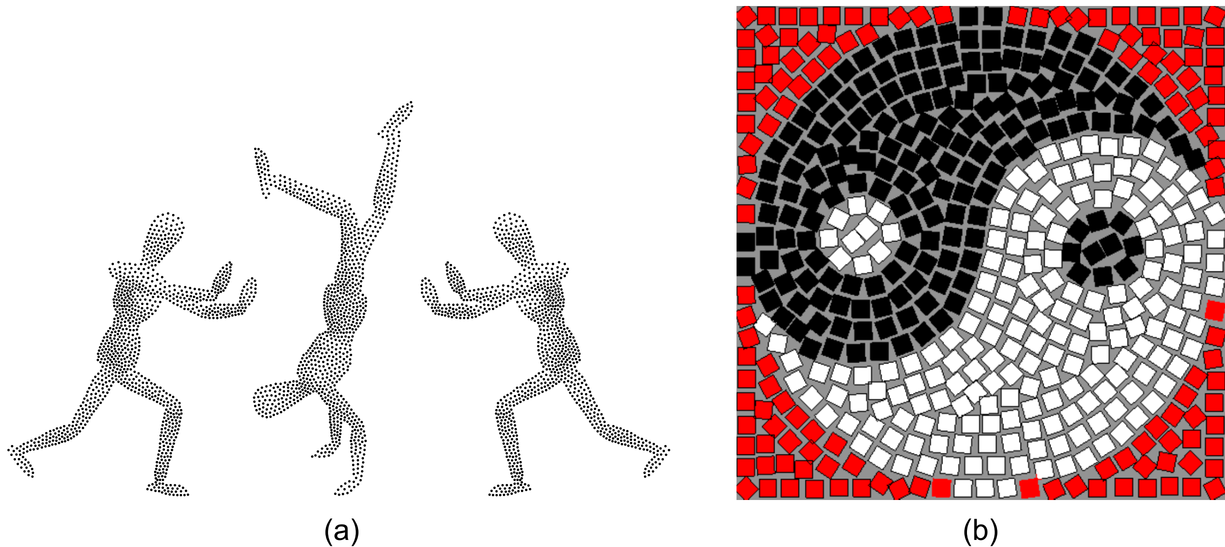


Figure 2.4: (a) Stippling artwork of point elements generated using Lloyd's method that incorporates pixel densities of the input image [Sec02]. (b) Mosaics of square elements generated using Lloyd's method with Manhattan distance [Hau01]

to construct *centroidal area Voronoi diagrams* (CAVDs), a variant of CVDs that yields a distribution of polygonal elements. This new extension computes the main inertial axis for each Voronoi cell so that its element can be rotated to achieve better alignment with the Voronoi cell boundary. In follow up work, Smith et al. [SLK05] developed Animosaics to generate temporally coherent animated mosaics by utilizing CAVDs. Dalal et al. [DKLS06] proposed a spectral approach based on a fast Fourier transform to reposition 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. However, the spectral approach cannot rotate the elements, so it resorts to a brute force approach to find the best orientation. Just like Animosaics, the spectral approach can generate animated mosaics, a topic that we will explore further in Chapter 5.

As an alternative to Lloyd's method, an interactive user interface can also be used to generate mosaics. Abdrashitov et al. [AGYS14] developed a sketch-based mosaic interface where an artist can draw curves to guide the placement of square-like elements. After the square elements are frozen in place, their boundaries are sliced to eliminate overlaps.

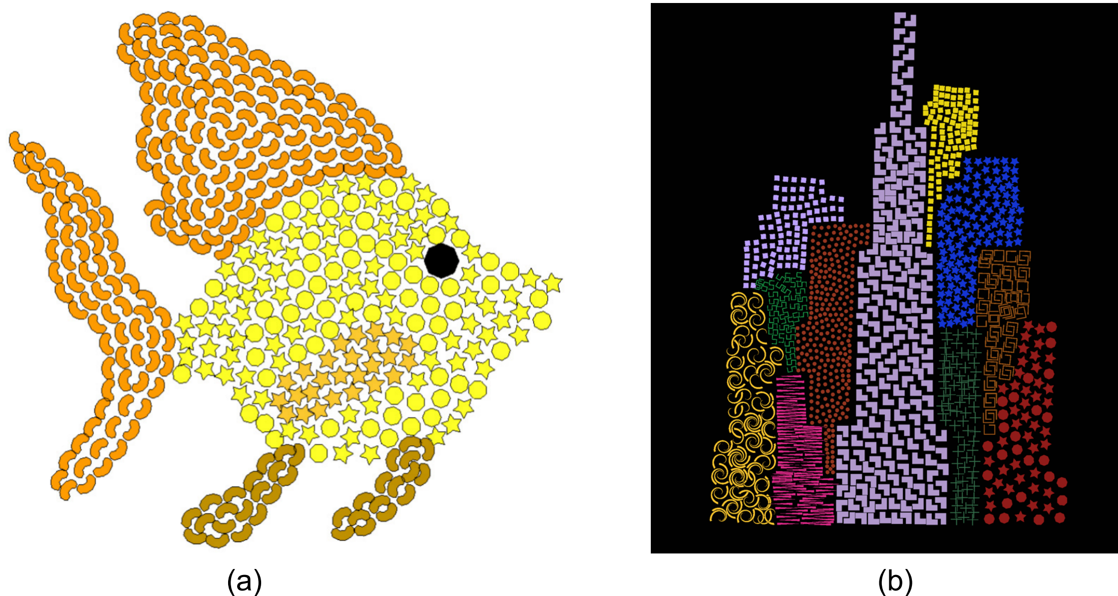


Figure 2.5: Mosaics generated with generalized Voronoi diagrams. (a) Animosaics [SLK05]. (b) A Spectral Approach [DKLS06].

2.2 2D Packings

This section discusses packings composed of elements that resemble man-made objects, geometry, letterforms, plants, or animals. Unlike decorative mosaics, packings are mostly found in logo design and graphic design. Some packing styles are reminiscent to paintings by Giuseppe Arcimboldo, like the one shown in Figure 2.6. Most packing methods drift away from Lloyd's method in favor of data-driven methods (Figure 2.2b) and deformation-driven methods (Figure 2.2c).

A data-driven method requires an input element library to find compatible elements (Figure 2.7). Typically, elements are placed one by one until the container area is filled. A shape matching algorithm selects an element from the library that has the best compatibility with the previously placed elements or the container boundary. Jigsaw Image Mosaics (JIM) [KP02] uses a geometric hashing technique to find compatible elements. JIM places elements using a greedy approach, but is able to backtrack if a previous configuration is more optimal. Pyramid of Arclength Descriptor (PAD) [KSH+16] is a curvature-based shape descriptor that can find new elements that partially match existing element boundaries as a container is being filled.

An alternative approach to data-driven methods is to initially partition a container into smaller segments and independently replace each segment with a matching element taken from the library. This is desirable if the container has colours or salient parts that need to be emphasized. Huang et al. [HZZ11] proposed a data-driven approach that generates 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 colour.

All these data-driven methods require a large element library. The more elements in the library, the better the chances of finding compatible elements. For example, JIM required about 900 elements, PAD used a library with more than 120 elements (see Figure 2.2), and Huang et al. needed about 5000 elements. However, a bigger database means increased computation time and collecting a large number of elements is not always feasible.

The placement process in JIM and PAD is being done without shape deformation. As shown in their results in Figure 2.7, element shapes never have perfectly compatible boundaries, and the packing process allows some overlaps to arise between neighbouring elements. These techniques optionally post-process elements by locally deforming their boundaries to suppress overlaps. While these local adjustments improve compatibility, they cannot influence the placement process, and a large element library is still required to generate satisfactory packings.

Instead of allowing shape compatibilities to arise organically in a large collection of elements, we can deform elements to manufacture those compatibilities. A deformation-driven method can create packings based on much smaller element libraries. Xu and Kaplan [XK07] and Zou et al. [ZCR+16] constructed calligraphic packings by filling a container with a small number of letter elements composing one or two words (Figure 2.2c and 2.9a). Xu and Kaplan's method is able to fill the container but the arrangement and deformation of the letters can make the calligram difficult to read. The improved method by Zou et al. can achieve better readability by computing the medial axis of the container where letter elements are anchored. However, these calligraphic packing methods require significant deformation so they are not able to preserve the characters of the original letterforms. In other work, Peng et al. [PYW14] proposed a method that packs and deforms simple polygon and polyomino elements to generate layouts for urban planning or indoor space, although their method cannot handle more complicated element shapes. An example of a generated indoor layout is shown in Figure 2.10.

These data-driven methods and deformation driven methods inspire us to develop new packing methods. In JIMs and PAD packings, deformation is a post-processing operation

Figure 2.6: *Vertumnus* (ca. 1590), by Giuseppe Arcimboldo, which illustrates a packing of plants, vegetables, and fruits (Skokloster Castle, Skokloster, Sweden).

(a)

(b)

Figure 2.7: Data-driven methods: (a) Jigsaw Image Mosaics [KP02]. (b) Pyramid of Arclength Descriptor [KSH+16].

Figure 2.8: An Arcimboldo-inspired 2D packing of overlapping elements [HZZ11] .

used for local touch-ups. In calligraphic packings and polyomino layouts, the deformation methods are limited to specific element shapes. We would like to use deformation as the core of the packing process and generate packings with more general styles similar to JIMs and PAD packings.

2.3 Tilings

As elements reach perfect compatibility, a packing turns into a tiling, such as the example in Figure 2.11a. The element boundaries interlock, leaving no negative space. Kaplan and Salesin introduced Escherization [KS00, KS04], a technique that deforms one or two input elements into shapes that can tile the plane. The tiles always fill the entire plane, and are not intended for filling a bounded container. Given an input element S, they performed a search in a parameterized tiling space to find its deformed variant T. Due to deformation, T is often perceptually unrecognizable from its silhouette unless an interior picture is added. Kaplan and Salesin's method has a better chance to find T if S has shallow concavities. More recent methods [KS11, NI20] are able to solve the Escherization problem more efficiently and can deal with shapes that have deeper concavities.

Past research has proposed methods to replicate the figure-ground reversal style of Escher's *Sky and Water*, in which elements serve as both positive and negative space

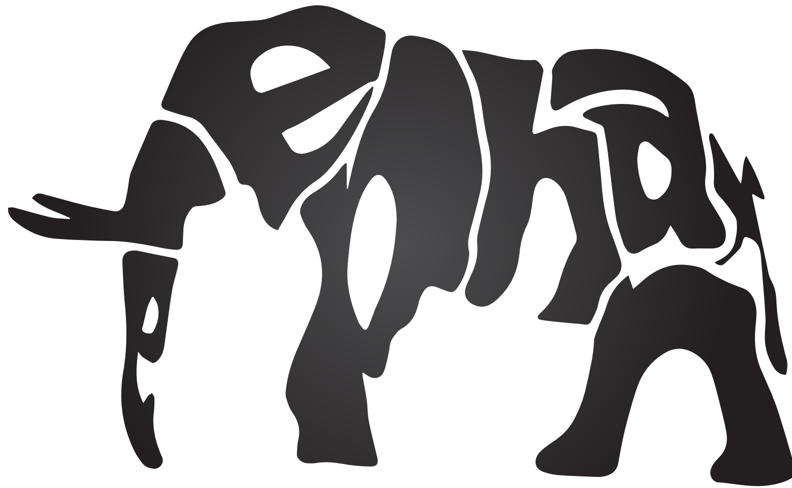


Figure 2.9: A deformation-driven packing of a calligraphic “elephant” [XK07].

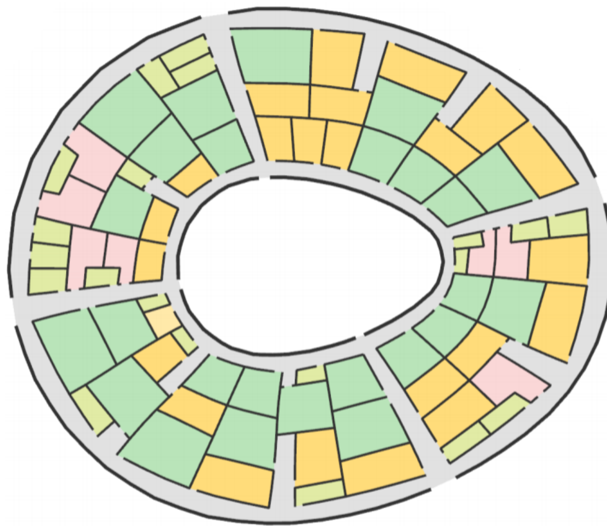


Figure 2.10: An office layout generated by packing deformable rectangle and polyomino elements [PYW14].

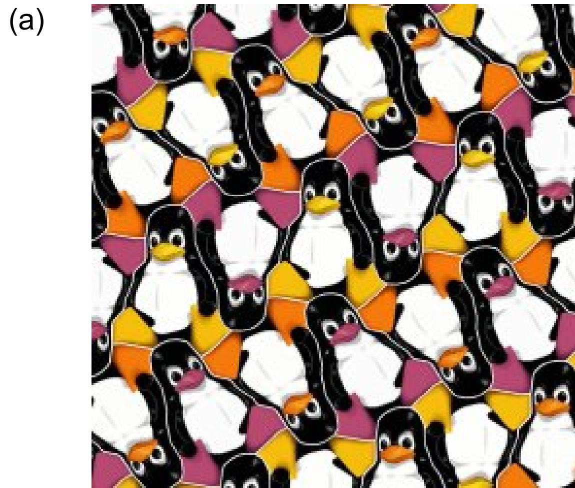
depending on the viewer's perception [KS04, Sug09, LML+18]. Two input elements S_1 and S_2 are placed at opposite ends of a canvas. They are then copied, arranged, and deformed until they meet in a tiling in the middle, with copies of each shape acting as the other's negative space (Figure 2.11b). The gradual evolution of tile shapes can also be found in Parquet Deformations, a graphic design style invented by William Huff [Hof86]. Aesthetically, Parquet Deformations are closely related to Escher's metamorphoses. An example can be seen in Figure 2.11c, where the tiling can be interpreted as a kind of spatial animation. Kaplan [Kap10] proposed several element evolution schemes: a grid-based curve evolution, iterated function systems, and organic labyrinth simulations.

2.4 Discrete Texture Synthesis

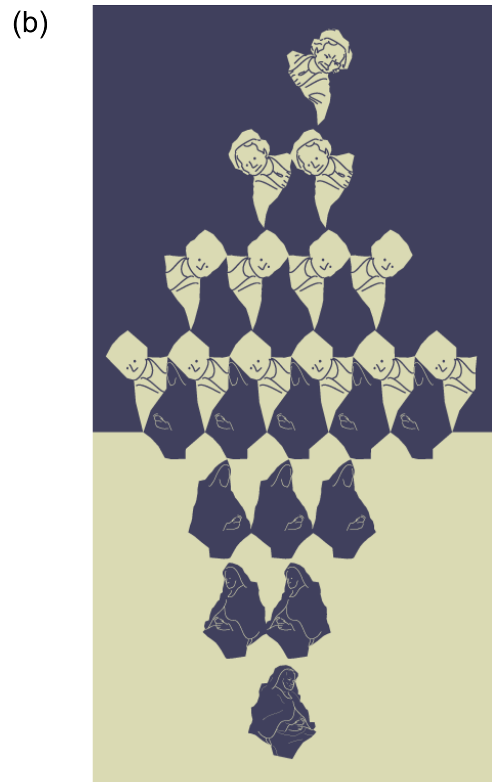
Some past work has sought to adapt example-based texture synthesis methods from raster images to vector graphics, producing distributions of rigidly transformed elements that mimic the statistics of an *exemplar*, an example can be seen in Figure 2.12. For raster-based texture synthesis methods, readers can refer to a survey paper compiled by Wei et al. [WLKT09]. These techniques are all concerned with replicating the distribution of positive space in the exemplar, and less about negative space. 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 neighbourhoods 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 method that can specify global arrangements, local arrangements, or a blend of multiple arrangements.

For larger concave elements, a single point representation is not enough. Ma et al. [MWT11] used a sample-based representation that is created by generating a sparse set of points inside an element. They later distributed these points using a neighbourhood metric and an iterative optimization. Unlike previous work, they were able to 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 tool where an artist can initially distribute elements by drawing strokes which are then optimized using a Lloyd-like optimization.

(a)



(b)



(c)

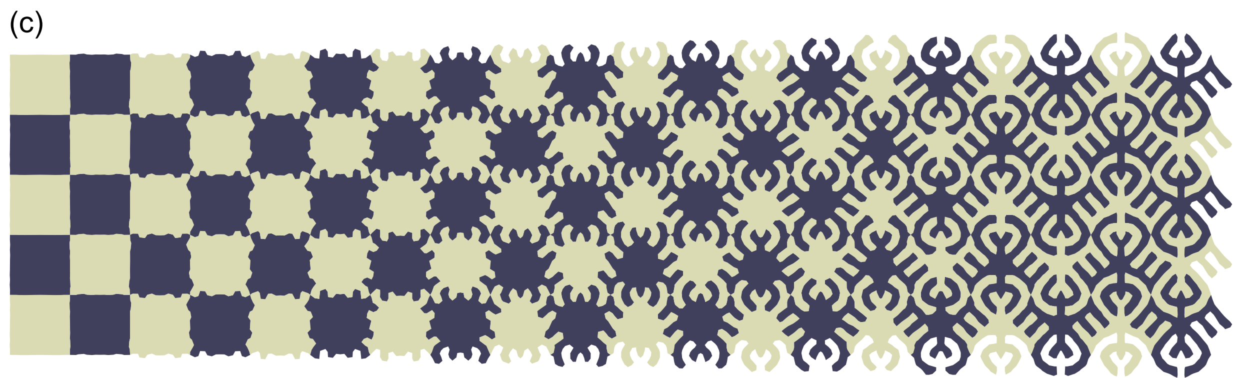


Figure 2.11: (a) A tiling of tux penguins [KS00]. (b) Sky and water tiling style [KS04]. (c) A Parquet Deformation [Kap10].

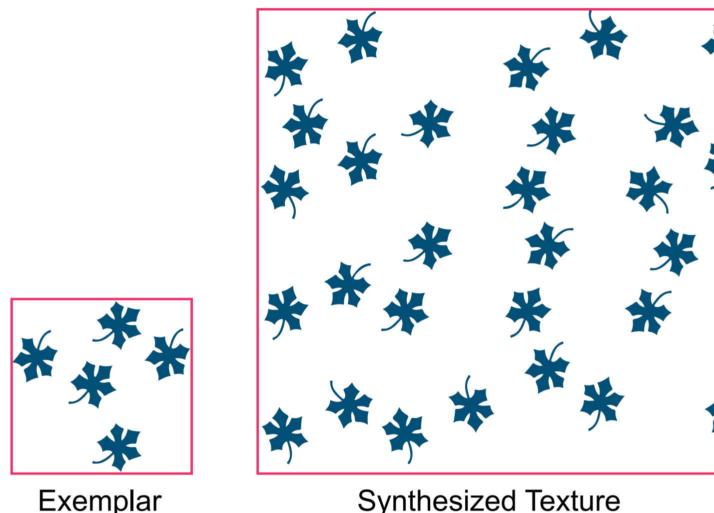


Figure 2.12: An example of discrete texture synthesis [AKA13a].

2.5 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 2.13a). They filled a 3D container with overlapping 3D elements using a greedy approach and a partial shape matching algorithm. This work can be seen as a JIM-like technique in 3D, except that there is more tolerance for overlaps. Similar to the work of Gal et al., Theobalt et al. [TRdAS07] developed a method to generate animated 3D collages, which we will discuss more in Chapter 5. In other work, Huang et al. [HWFL14] designed a method to generate mechanical collages, such as giant robots. All these collage methods require a 3D shape database.

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" that 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

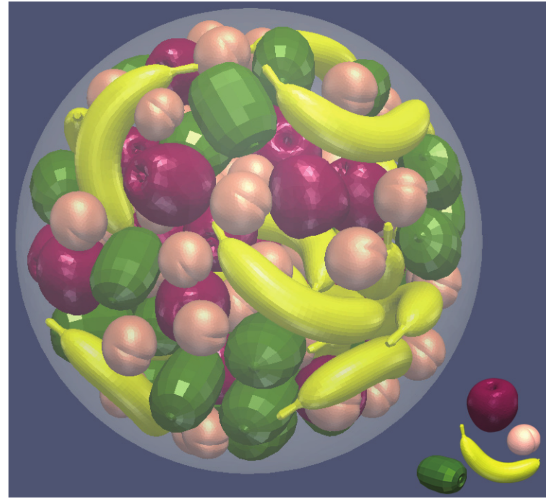
printed objects.

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 non-overlapping triangular meshes (Figure 2.13b).

2.6 Packings on Surfaces

Some recent work has explored the elaboration of ornamental patterns on surfaces, under constraints imposed by fabrication, such as connectivity. Schiftner et al. [SHWP09] introduced Circle Packing meshes, a special kind of triangle mesh, where the incircles of two neighbouring triangles have the same contact point on the shared edge, forming a packing of circles or spheres. 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, as shown in Figure 2.13c. They start with an initial, overlap-free configuration of scaled down elements. They then grow the elements, avoiding overlaps using curve deformation. Chen et al. [CML+17] generated modular surfaces by computing contact point networks of rigid elements, as shown in Figure 2.13d. Bian et al. [BWL18] used Wang tiles made of element parts to generate filigree patterns. Martínez et al. [MSS+19] developed a variant of Lloyd's 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)

(d)

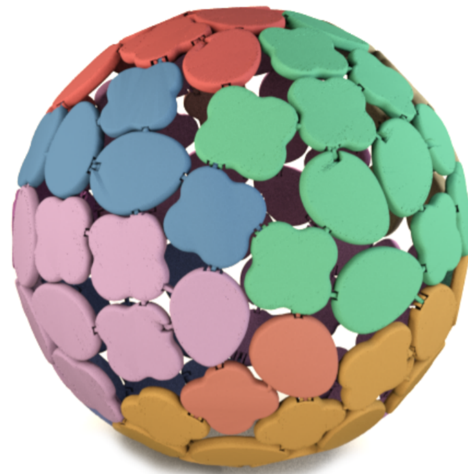

(c)

Figure 2.13: Examples of 3D packings and packings on surfaces. (a) An Arcimboldo-like packing of overlapping 3D elements [GSP+07]. (b) A packing of non-overlapping 3D elements [MCHW18]. (c) Deformed curve elements that fill a surface [ZCT16]. (d) A surface packed with elements connected to each other by hinges [CML+17].

Chapter 3

FLOWPAK: Flow-Based Ornamental Element Packing

3.1 Introduction

FLOWPAK is a technique for filling a container region with elements that are deformed to communicate a sense of directionality or flow. We aim to create packings that obey all design principles articulated in Chapter 1, with a special focus on the flow principle. Elements have simple geometric forms, often stylized flora, spirals, or other abstract geometry. 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.

3.2 Related Work

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], which 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] generated decorative curves by simulating the trajectories of charged particles in a magnetic field. 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 have 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 linked together 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 Approach

The user provides several inputs, similar to those described in Section 1.1 with a few additional details:

1. An input element has a spine that will control its deformation.

2. Each container includes a set of direction guides that influence 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. Fixed elements are also included.

The goal of FLOWPAK is to fill each target container with elements. Figure 3.1 gives an overall view of our system. The drawing in Figure 3.1(1) shows containers, directional guides, and a fixed element. The drawing in Figure 3.1(2) shows elements and their spines. 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 3.3.1).

2. Analyze the ornamental elements, creating a shape descriptor for each (Section 3.3.2).

3. Use the direction guides to fill each target container with a vector field and then trace streamlines (Section 3.3.3).

4. Divide the target containers into Voronoi cells around the streamlines. Each cell is referred to as a *blob*. (Section 3.3.4).

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 3.3.5).

6. Iteratively refine the placement to eliminate empty areas and make the spacing more even (Section 3.3.6).

3.3.1 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.

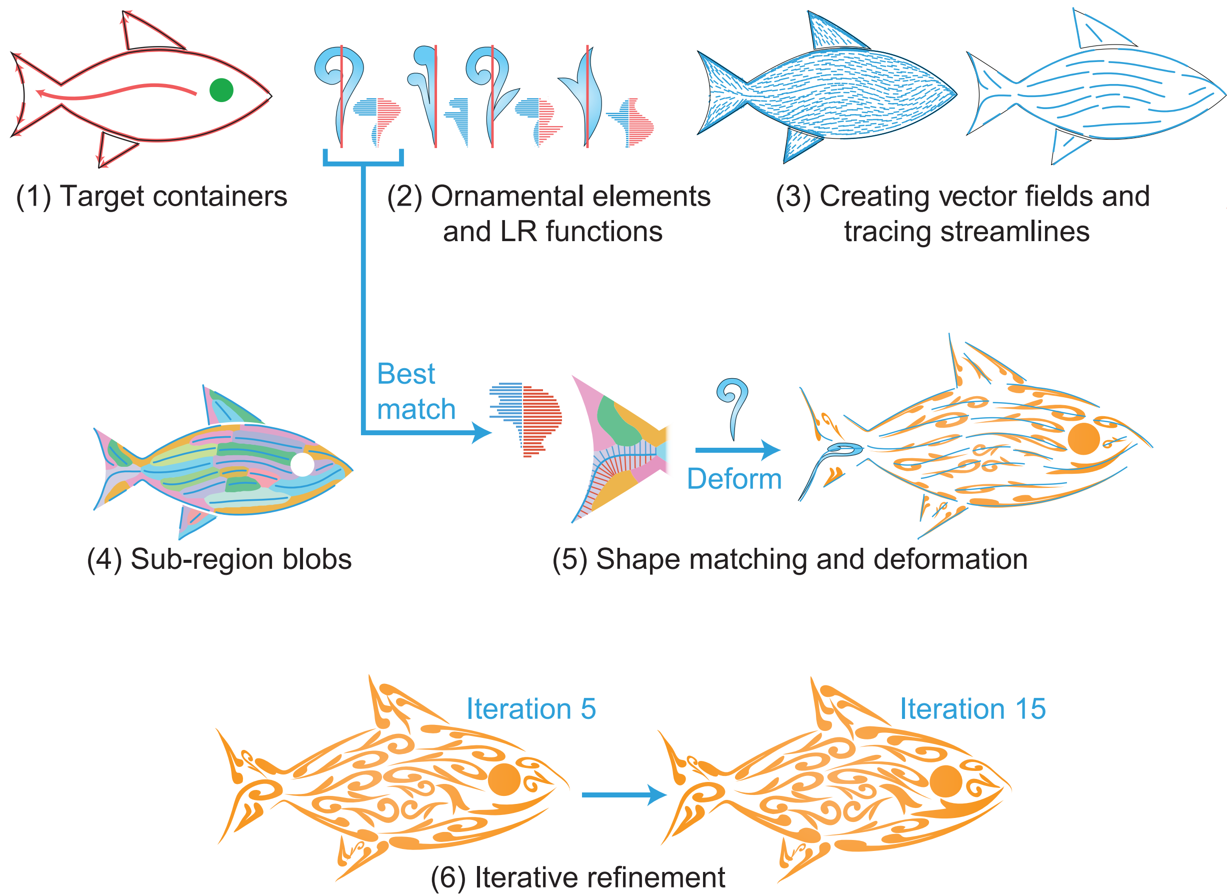


Figure 3.1: A visualization of the steps in our ornamental packing algorithm. The input containers are shown as three black outlines in (1): the body and two fins. They are annotated with directional guides in red and fixed elements (in this case, the eye) in green. The steps in the algorithm follow the description at the beginning of Section 3.3.

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 l_{canvas} to be the canvas size, which is 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.

3.3.2 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 3.3.5) using a simple skeletal stroke 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 centre of the element—it can be anywhere.

We define two classes of elements: a *two-sided element* extends across both sides of its spine, and a *one-sided element* lies entirely on one side of its spine. Figure 3.2 shows examples of two-sided and one-sided elements. If the input to our algorithm includes direction guides that coincide with target container boundaries, the placement method will align one-sided elements along these boundaries. The alignment of one-sided elements will visually reinforce container boundaries, as shown in our results.

We define a simple shape descriptor called an *LR function* that will be used in Section 3.3.5 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 $L = \{\ell_1, \dots, \ell_{n_\text{f}}\}$ is the left function and $R = \{r_1, \dots, r_{n_\text{f}}\}$ 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 3.3.4), in order to achieve the best possible fit.

Intuitively, LR functions give an approximate area an ornamental element can claim. Figure 3.2 shows elements with their left values in blue and their right values in red. We have found that $n_\text{f} = 100$ gives sufficient granularity for our algorithm.

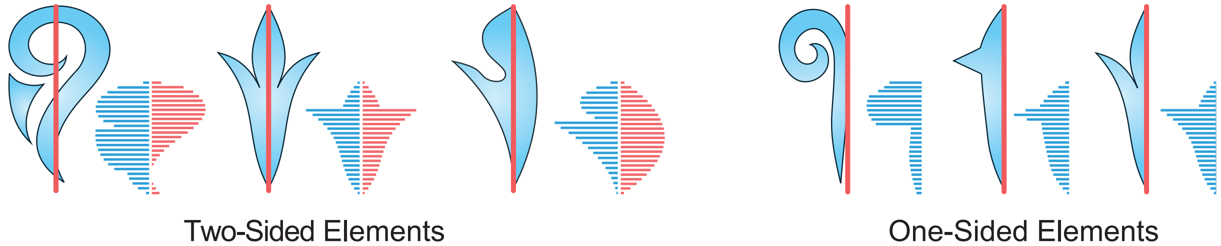


Figure 3.2: Ornamental elements and their LR functions. Two-sided elements have non-empty left and right sides, while one-sided elements have only one non-empty side. We normalize the LR functions to a unit square.

3.3.3 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 $D = \{d_1, d_2, ..., d_{n_\mathrm{d}}\}$ and use the tangent at every sampled point as a directional constraint. We then construct a vector field using the 1-RoSy algorithm of Palacios and Zhang [PZ07]. Note that, as shown in Step 3 of Figure 3.1, 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 $s_\mathrm{stop} = 0.8\, s_\mathrm{gap}$.

We adapt the streamline tracing algorithm of Jobard and Lefer [JL97]. First we generate a set of potential seed points $P = \{\boldsymbol{p_1}, \boldsymbol{p_2}, ..., \boldsymbol{p_{n_\mathrm{p}}}\}$ by densely resampling the target container boundary T and the directional guides in D. We use a sampling distance of $0.005\, l_\mathrm{canvas}$. 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.

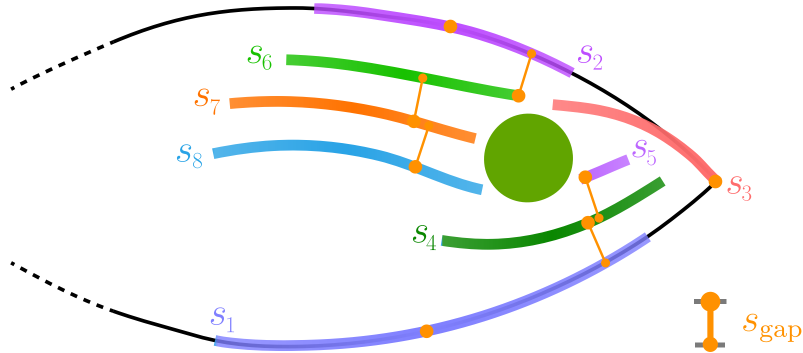


Figure 3.3: The streamline tracing process. The first streamline s_1 always begins on a directional guide or the container boundary. Subsequent streamlines begin on the container boundary, a directional guide, or at a point that is s_{gap} away from a previous streamline.

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 $0.005\, l_{\text{canvas}}$, 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} length test imply that many attempts to form streamlines will stop immediately, especially as the container fills with streamlines.

Figure 3.3 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.

3.3.4 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].

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
```

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 3.1 shows the blobs of the fish.

We then compute an LR function for each blob as described in Section 3.3.2, 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.

3.3.5 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

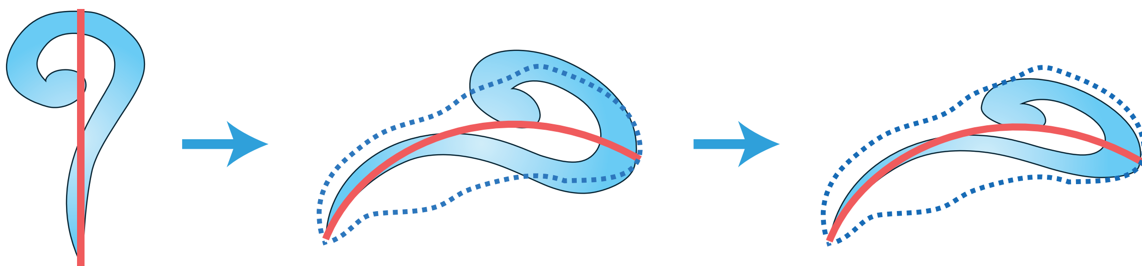


Figure 3.4: The deformation process bends the element along the streamline and scales it to fit inside the blob.

α_l is the element L function;
α_r is the element R function;
β_l is the blob L function; and
β_r is the blob R 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 will naturally place one-sided 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 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 3.4).

3.3.6 Iterative Refinement

The previous steps produce an initial arrangement consisting of a packing of flow-guided elements in each distinct container. We now merge all the containers and all elements

within them, allowing elements from different containers to interact with each other in an iterative manner.

The refinement process aims to reduce the amount of negative space and make it more even by growing and shifting the placed ornamental elements. We improve the placement of each element as much as possible before moving on to the next. Each refinement iteration has two phases. First, we shift the streamlines to more accurately follow the space that is available, as shown in Figure 3.5. 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 3.7.

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

The rest of this section gives the implementation details of the refinement process and Algorithm 2 summarizes the overall method. 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 3.5). It also gives the streamline room to extend if its endpoints were too close to boundaries of other placed elements.

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

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
```

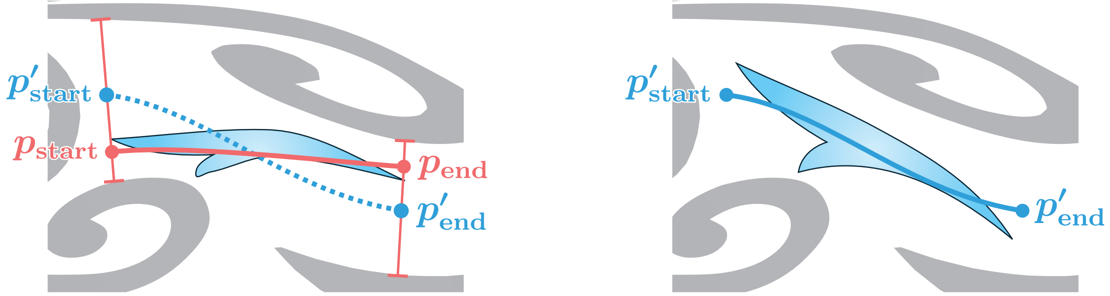


Figure 3.5: Streamline shifting. We move the streamline's start and end points along perpendiculars, stopping before intersecting neighbouring elements.

move more than g_{limit} units, where $g_{\text{limit}} = 0.005\, l_{\text{canvas}}$ (Recall that l_{canvas} is the canvas size of the design as described in Section 3.3.1).

We replace the streamline with a path joining $\boldsymbol{p'_{\text{start}}}$ and $\boldsymbol{p'_{\text{end}}}$. Our goal is to create a path that is smooth and does not deviate too much from the vector field. We calculate the shifted streamline by performing Dijkstra's algorithm on a non-rectangular graph that respects the vector field (Figure 3.6), 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 150 nearest neighbours, considering only neighbours where the angle between the line into the current node and the line to the neighbour form an angle greater than 90°, thereby preventing the streamline from backtracking. The cost of an edge from $\boldsymbol{p_a}$ to $\boldsymbol{p_b}$ is

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

where

f	is $(\boldsymbol{p_b} - \boldsymbol{p_a}) \cdot \boldsymbol{v}$;
$\boldsymbol{v}$	is a sampled unit vector of the vector field at $\boldsymbol{p_a}$;
s_i	is the original streamline; and
$D()$	is a distance function between a polyline and a point.

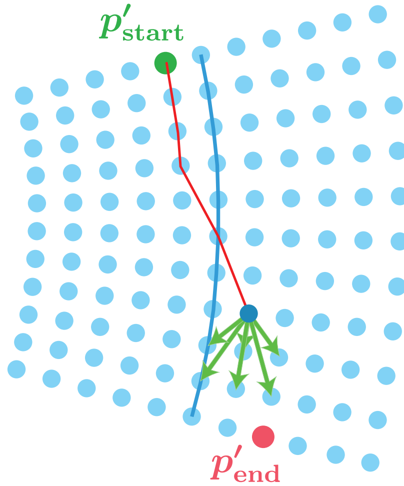


Figure 3.6: Tracing a shortest path using Dijkstra's algorithm. We generate the blue nodes by resampling and offsetting the blue streamline. The search directions at a node are shown with green arrows.

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 blob b'_i by offsetting its boundaries until they intersect other placed elements (Figure 3.7). To enforce gradual growth, the offset cannot be larger than g_{limit}, where $g_{\text{limit}} = 0.005\, l_{\text{canvas}}$.

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 3.8. We generate rotated versions of e_i with varying angles $r_{\text{angle}} = \pm\{1^\circ, 2^\circ, 3^\circ, ..., 10^\circ\}$ and precompute LR functions for each. The shape matching algorithm (Section 3.3.5) automatically choses the best rotation. It can also choose to reflect the element across its spine, along its spine, or both (Figure 3.9).

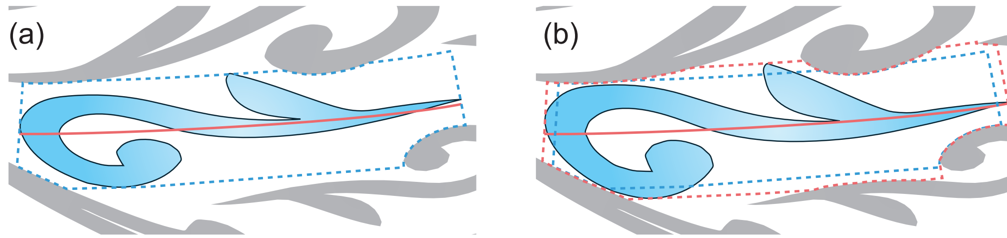


Figure 3.7: (a) An element with its sub-region blob shown in dashed blue line. Note that any blob is constrained by the neighbouring elements. (b) The dashed red line is the grown blob, which accommodates an enlarged element.

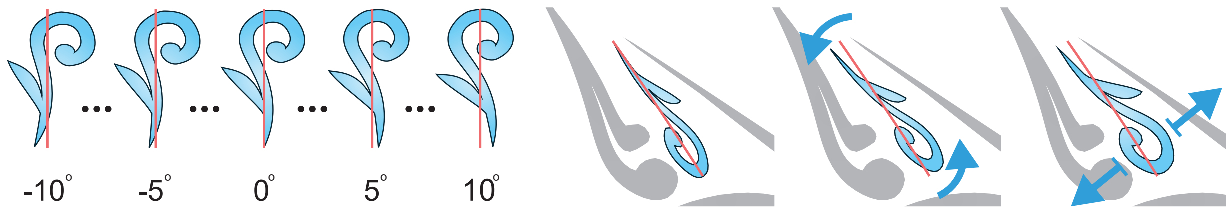


Figure 3.8: Left: rotated versions of the original element. The best rotation angle is chosen via least squares matching. Right: original, rotated, and enlarged versions of an element.

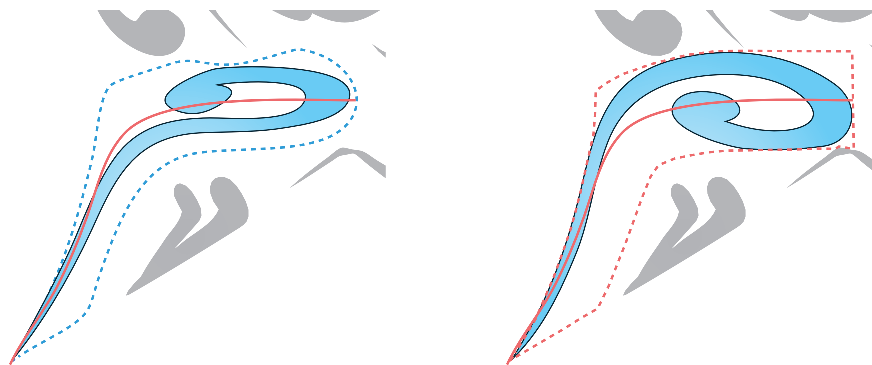


Figure 3.9: An element that reflects across its spine during iterative refinement. LR functions and least squares shape matching allow an element to reflect across its spine, along its spine, or both.

3.4 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 colours and other treatments in an editor.

Our technique is fast except for the iterative refinement process, which considers a large number of variations to the composition via brute-force computation. On a computer with an Intel i7-4790K processor at 4.0 Ghz, 15 iterations of refinement on a packing of 50 elements takes about an hour. Our software is not intended to run interactively; still, we believe the performance could be improved significantly through the use of more sophisticated 2D geometric data structures 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. In Figure 3.10, 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 3.11, 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 3.12, 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 3.13 demonstrates a symmetric packing with a fur contour. We compute only 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 3.14 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 3.14 for the flower petals in the torso and the paws. Second, in Figure 3.15 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 3.16. 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.

3.5 Conclusions and Future Work

We presented FLOWPAK, a method to create ornamental packings in which the elements are 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.

We see two natural opportunities for improving FLOWPAK. First, we would like to generate streamlines with higher curvatures, like u-turns. However, these streamlines could unpleasantly fold the decorative elements. This could be solved with a folding avoidance algorithm [Ase10]. Second, we would like to experiment with a procedure for element placements that can backtrack to previous configurations, as demonstrated by Jigsaw Image Mosaics [KP02]. This way, we can achieve a better configuration with less iterative refinement.

Figure 3.10: 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). The colours in the final rendering were added manually.

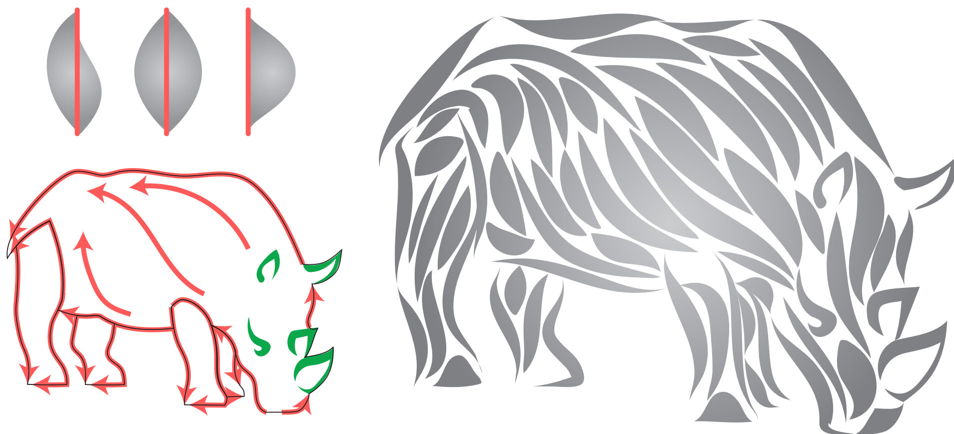


Figure 3.11: A packing of a rhinoceros. Simple teardrop-shaped elements lead to variety in size and curvature.

Figure 3.12: A bear packing with leaf elements. We manually add noise to the elements in the output to create a distressed look.

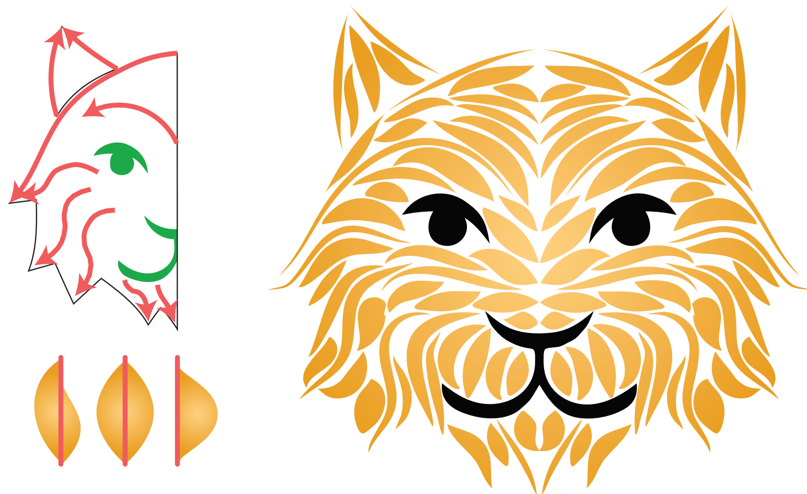


Figure 3.13: A packing with a symmetric layout; we only compute the left half and reflect the result. The elements around the cheeks and the chin are not aligned to the boundary, creating a fur-like effect.

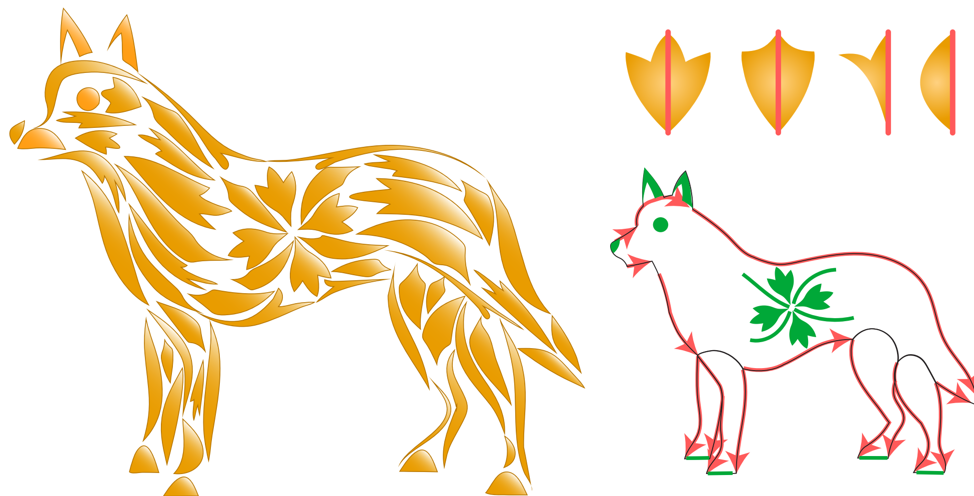


Figure 3.14: A packing of a dog. The fixed elements, shown as green shapes in the diagram, are copied as-is to the output; fixed spines, shown as green paths, force the placement of new elements at the given locations.

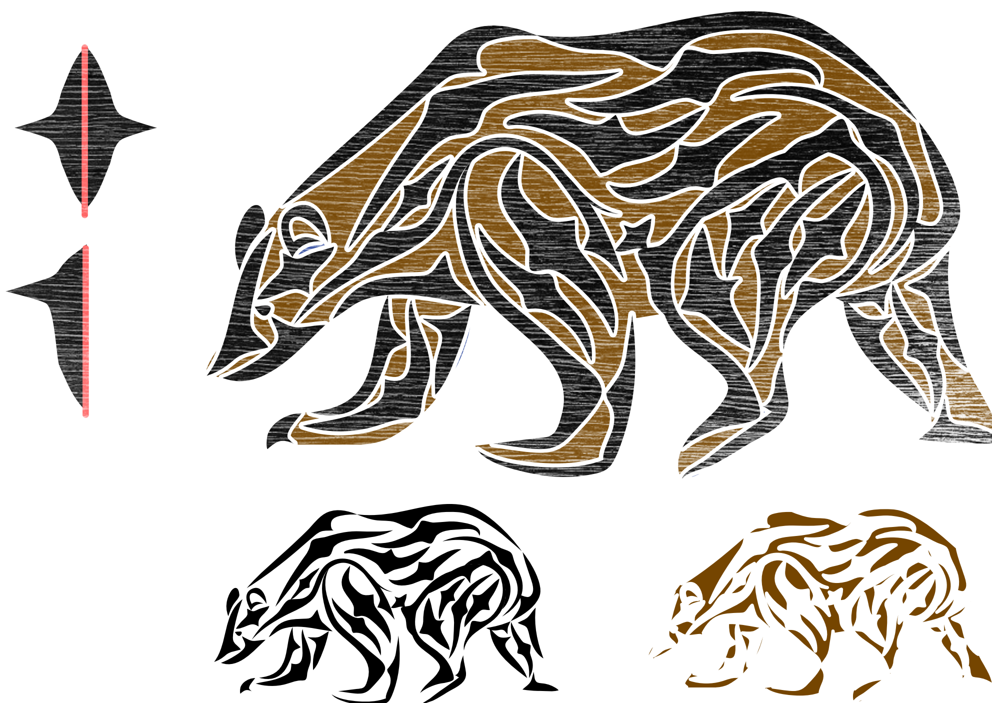


Figure 3.15: A packing of the same container as in Figure 3.12. We place longer and sparser elements and synthesize additional elements from the remaining negative space. The bottom row is the visualization of primary elements in black and negative space elements in brown.

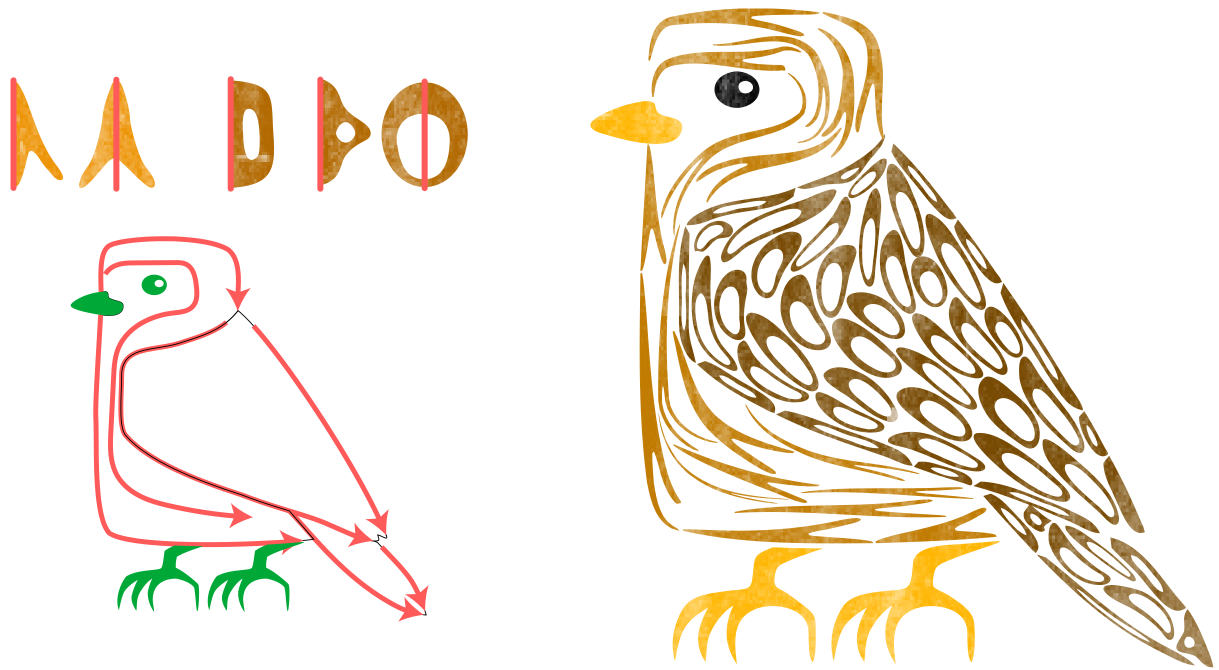


Figure 3.16: A packing of a bird, based on input provided by an artist.

Chapter 4

RepulsionPak: Deformation-Driven Element Packing with Repulsion Forces

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 neighbouring 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 neighbours 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 was shown in Figure 1.2.

Unlike FLOWPAK that is optimized to deform long thin elements, we design RepulsionPak to support arbitrary shaped elements. Elements in FLOWPAK could only bend along their spines. In RepulsionPak, we allow elements to deform in a more open-ended way.

RepulsionPak is intended to fulfill three of five design principles: balance, uniformity amidst variety, and boundaries (Chapter 1). 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.

Chapter 2 discussed the general categorization of mosaicing and packing techniques: Lloyd's method, data-driven methods, and deformation-driven methods. RepulsionPak can be categorized as a hybrid of point-based Lloyd's method and a deformation-driven method. Similar to Lloyd's method, RepulsionPak iteratively refines the element distribution although it does not explicitly compute a Voronoi diagram. Forces do not act directly on elements, but their mesh vertices. During the simulation, these mesh vertices are spread apart but remain connected using elastic springs, effectively creating an element representation that can be packed and deformed. Additionally, unlike previous variants of Lloyd's method, repulsion forces naturally induce rotations in elements, helping them discover greater compatibilities with their neighbours.

4.2 Related Work

Curve Simulations: Pedersen and Singh [PS06] grew curves to create organic labyrinthine paths using a geometric analogue of reaction-diffusion. Their algorithm is related to ours by the use of repulsion forces to maintain even spacing and parallel segments. Yu et al. [YSC20] reformulated repulsion forces as tangent-point energy to pack curves inside a container while avoiding self-intersections and resolving tangling. In fabrication, 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 simulation process is 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. Additionally, the goal of their work (3D fabrication) is quite different from that of RepulsionPak (2D graphic design) and our results appear qualitatively different.

Sample-Based Representation: A few 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.

RepulsionPak uses a similar approach, but we represent an element with denser vertices that are connected with elastic springs.

4.3 System Overview

RepulsionPak requires several inputs, which are described in Section 1.1. Elements are then categorized as primary and optional secondary elements, such as those shown in Figure 4.1a. Additionally, an artist needs to specify 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 seeded with copies of scaled-down elements. This initial element placement can be done in two ways: random seeding (Section 4.4) or a careful placement using a shape matching algorithm (Section 4.7).

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). 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 paths from the element's initial mesh into the deformed mesh produced through simulation. The computation of barycentric coordinates is discussed in the next section.

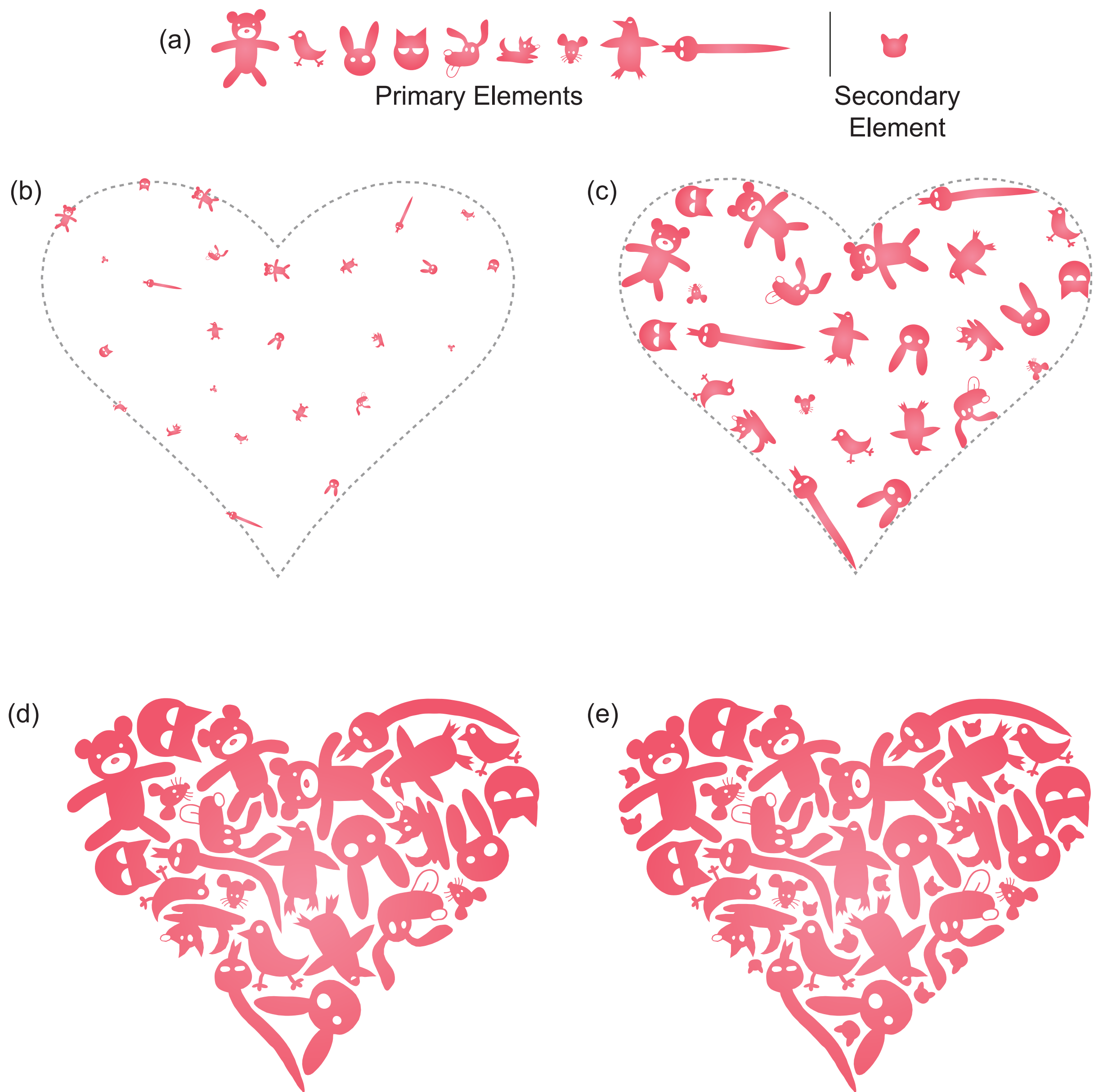


Figure 4.1: The creation of a packing using RepulsionPak. (a) A library of elements, comprising nine primary elements and a single secondary element. (b) A target container with the initial distribution of scaled-down elements. (c) The simulation in progress, showing the elements growing, translating, rotating, and deforming. (d) The resulting packing of primary elements. (e) The final result, after adding secondary elements and allowing them to grow. Figure 4.2 shows the deformations of some of the elements.

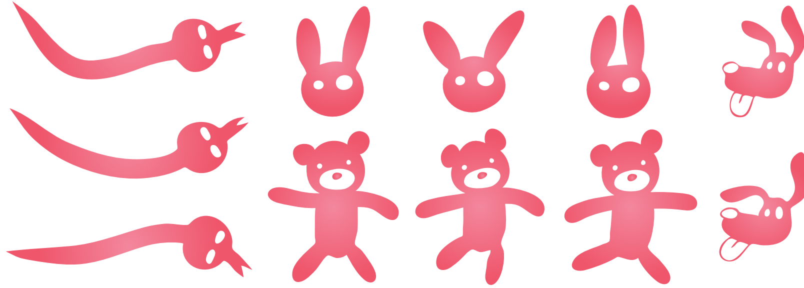


Figure 4.2: Some of the elements in the final packing in Figure 4.1, showing the effect of deformation in our simulation.

4.4 Preprocessing

Given an input element (Figure 4.3a), we generate a skin, which is a simple closed polygon that fully encloses it (Figure 4.3b). We generate the skin by offsetting an element's boundary outward by $d_{\mathrm{gap}}/2$. 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', which is the outer boundary of the mesh (Figure 4.3c). We then construct a Delaunay triangulation of s' (Figure 4.3d). 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 neighbours. We also add extra edges to reduce folding and self-intersections during simulation. First, if two triangles ABC and BCD share an edge BC, then we add a *shear edge* connecting A and D (Figure 4.3e). 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 (Figure 4.3f). These negative space edges are used exclusively for internal bracing. The element's concavities can still be occupied by its neighbours. The construction of negative space edges is a simpler variant of Air Meshes [MCKM15], a technique to detect and resolve collisions.

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

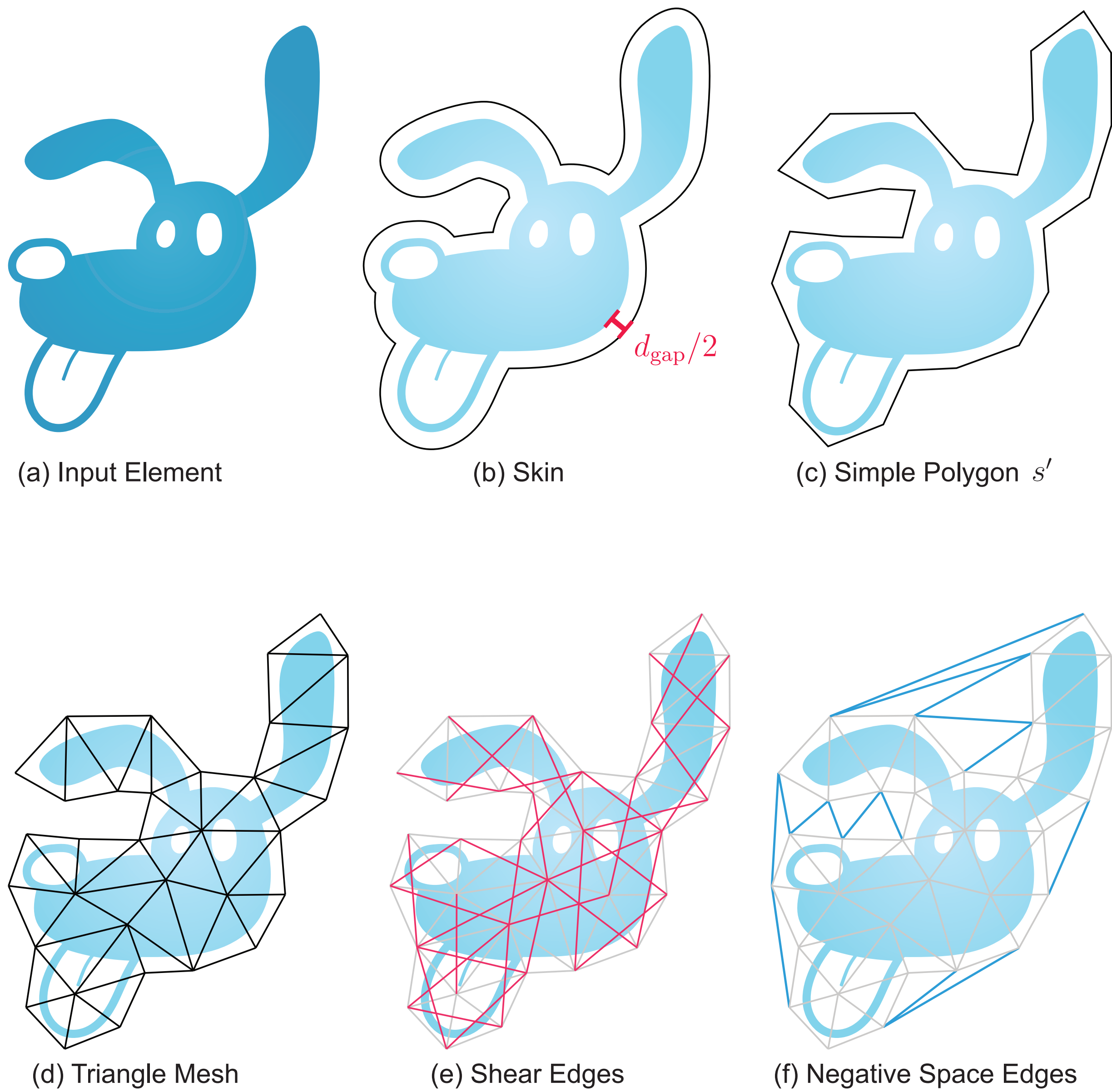


Figure 4.3: An illustration of element discretization for preprocessing.

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 as illustrated in Figure 4.1.

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 4.1b 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

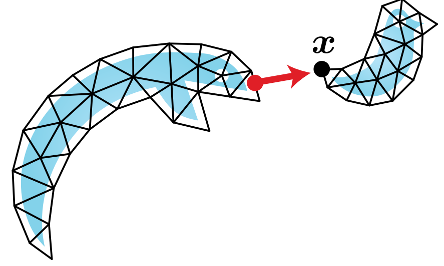
$$\boldsymbol{F}_{\text{total}} = \boldsymbol{F}_{\text{rpl}} + \boldsymbol{F}_{\text{edg}} + \boldsymbol{F}_{\text{bdr}} + \boldsymbol{F}_{\text{ovr}} + \boldsymbol{F}_{\text{tor}} \tag{4.1}$$

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.



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:

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

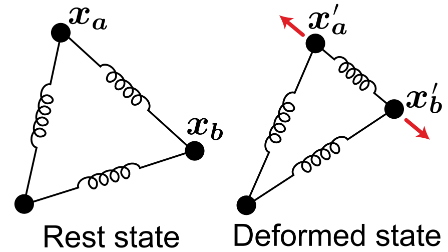
where

k_{rpl} is the strength of the repulsion force relative to other forces in the simulation;
n is the number of nearest neighbouring meshes to $\boldsymbol{x}$;
$\boldsymbol{x_i}$ is the closest point on the skin of the ith neighbour;
$\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{x_i}$; 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 neighbours 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 neighbouring 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 $\boldsymbol{x}'_{\boldsymbol{a}}$ and $\boldsymbol{x}'_{\boldsymbol{b}}$ be deformed mesh vertices connected by a spring. We compute the spring force as follows:



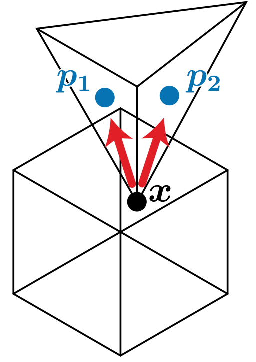
$$\boldsymbol{F}_{\text{edg}} = k_{\text{edg}} \frac{\boldsymbol{u}}{\|\boldsymbol{u}\|} s \left(\|\boldsymbol{u}\| - \ell\right)^2 \tag{4.3}$$

where

k_{edg} is the strength of the edge force relative to other forces;
$\boldsymbol{u} = \boldsymbol{x}'_{\boldsymbol{b}} - \boldsymbol{x}'_{\boldsymbol{a}}$;
$\ell = \|\boldsymbol{x}_{\boldsymbol{b}} - \boldsymbol{x}_{\boldsymbol{a}}\|$ which 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 $\boldsymbol{x}_{\boldsymbol{a}}$ and $-\boldsymbol{F}_{\text{edg}}$ to $\boldsymbol{x}_{\boldsymbol{b}}$. The strengths of the edge forces increase quadratically with displacement. The equation is a modification of Hooke's law, which is defined as $-k_{\text{edg}}(\|\boldsymbol{u}\| - \ell)$. This change allows the meshes to be more tolerant of small displacements and 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 neighbouring 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:



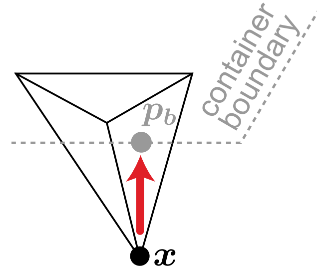
$$\boldsymbol{F}_{\text{ovr}} = k_{\text{ovr}} \sum_{i=1}^{n} (\boldsymbol{p}_{\boldsymbol{i}} - \boldsymbol{x}) \tag{4.4}$$

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
$\boldsymbol{p}_{\boldsymbol{i}}$ 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:

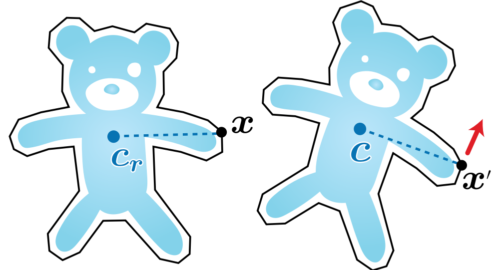


$$\boldsymbol{F}_{\text{bdr}} = k_{\text{bdr}}(\boldsymbol{p}_{\boldsymbol{b}} - \boldsymbol{x}) \tag{4.5}$$

where

k_{bdr} is the relative strength of the boundary force; and
$\boldsymbol{p}_{\boldsymbol{b}}$ 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 centre 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 centre 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 centre 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 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-neighbour 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 neighbours of a vertex $\boldsymbol{x}$ as all vertices

in a 3×3 window of cells centred 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 4.1c-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 neighbouring 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 an element mesh's springs, allowing it to expand as the simulation progresses. Every element mesh M maintains a current scaling factor g. When a mesh is permitted to grow, we increase its g value by $k_g \Delta t_{\text{sim}}$, where k_g is the growth rate and Δt_{sim} is the simulation time step. We then compute the new rest lengths of all springs by scaling the current rest lengths by a factor of g. These new rest lengths are then used as the ℓ values in Equation 4.3 to calculate edge forces. In our implementation, The constant k_g is set to 0.01.

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 of our results have positive space ratios between 35% and 60% depending on the element shapes. For example, the Unilever logo in Figure 1.1 has a positive space ratio of 45%. Additionally, 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

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

We stop iterating when $\mathrm{RMS} < \epsilon$, 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 with 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 $d_{\mathrm{sep}} = 10$.

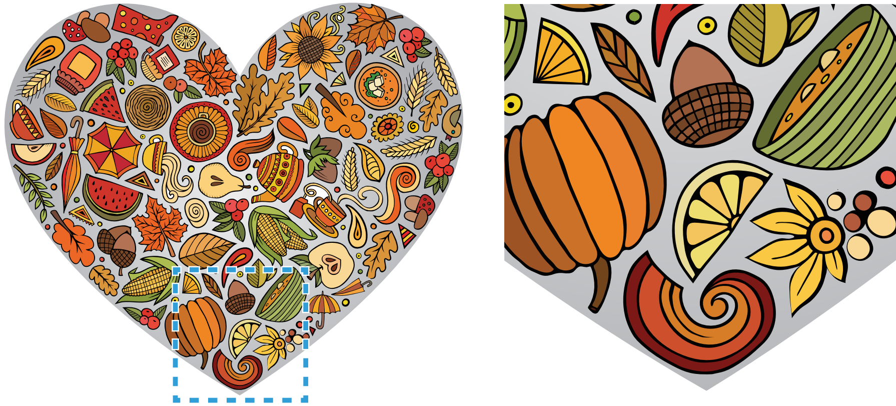


Figure 4.4: A packing generated by RepulsionPak that has a rounded corner. A round element is placed on a sharp convex corner on the bottom of the heart-shaped container.

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", as seen in Figure 4.4. 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 as seen in the rhinoceros's front horn in Figure 4.6.

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 $0.6A_c/n_\mathrm{e}$, 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:

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

In our implementation, we estimated curvature from the discrete sample points using second-order forward finite differences [BL15], although we recently found that second-order central finite differences can give more accurate approximations. Lastly, we 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 $\tau(t_{-1}, t_0) = \tau(t_0, t_1) = \tau_\mathrm{obj}$. 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 4.5). 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 $\tau_\mathrm{obj} = 0.001L_s$; 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.

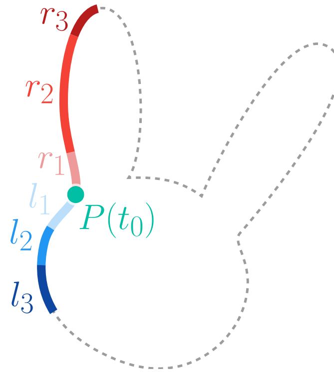


Figure 4.5: An illustration of a local shape descriptor with $n = 3$. These segments have varying arclengths but they all have the same value of τ, the integral of absolute curvature along their lengths.

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 placement (See Figure 4.7a). 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. An example of this problem is illustrated in Figure 4.8. 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

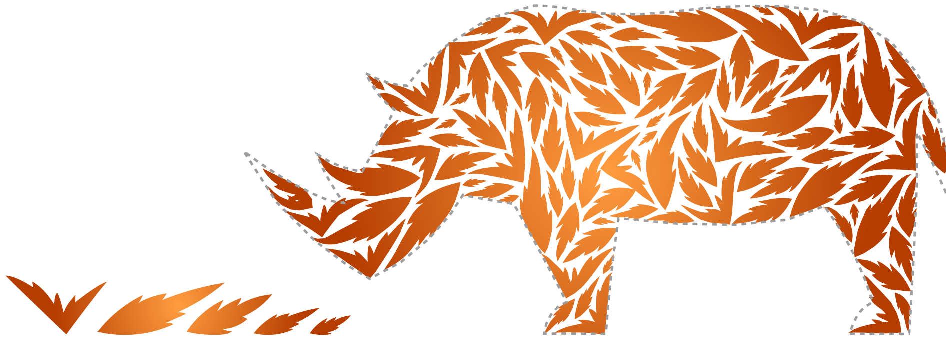


Figure 4.6: A packing without the use of shape matching. Three elements are not aligned properly with convex corners: a long thin element is severely deformed in the front horn, an element cannot emphasize the shape of the back leg properly, and an element cannot extend inside the narrow tail.

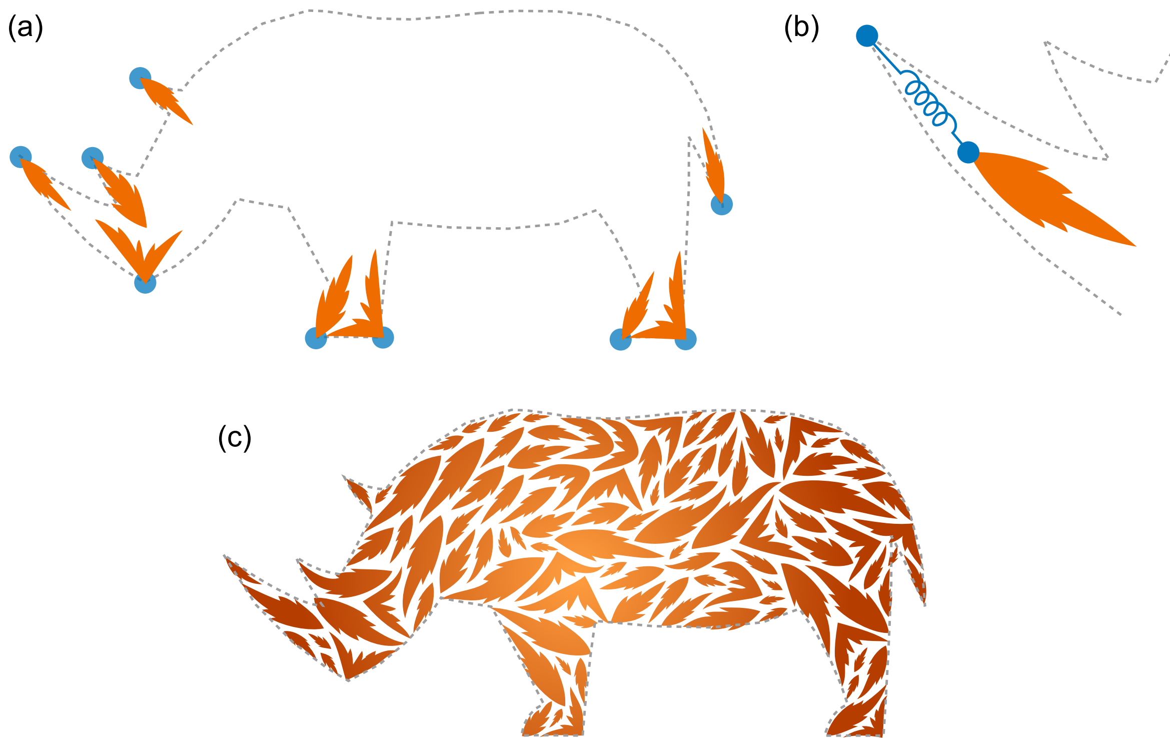


Figure 4.7: A demonstration of shape matching of leaf shapes inside a rhinoceros. (a) We detect nine salient features, namely sharp convex corners, and assign an element to each. (b) A spring holds each element in place. (c) The final result.

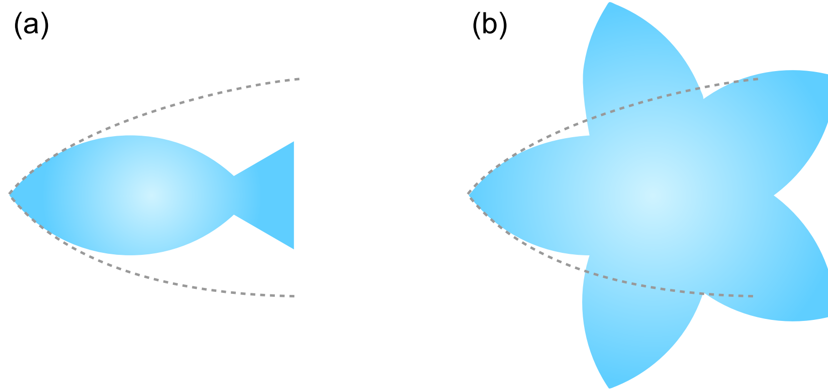


Figure 4.8: Two elements that locally match the container's corner. The element in (a) is completely inside, but the element in (b) has non-local regions that protrude outside the container.

transform. For an element sample point $\boldsymbol{x}$, we then compute a score

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

where

α is a parameter that penalizes elements with parts that protrude outside the container;
β is a parameter that favors elements with parts close to the target container;
$\mu = 0.5\sqrt{2}L_s$ which is half of the diagonal of the container's bounding square.

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. In our implementation we choose $\alpha = 1$ and $\beta = 0.001$. 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:

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

where

n_p is the number of sample points on the element.

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 4.7b, 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 colours 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 4.11, and shape matching is used only in Figure 4.7. We let real-world artists and users to use RepulsionPak for generating packings in Figure 4.14 and 4.15, but we unfortunately were unable to collect their statistics.

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

The packing in Figure 4.9 uses six cat-shaped primary elements and one secondary cat head. RepulsionPak naturally bends legs and tails to fill the container more evenly.

The animal packing in Figure 4.1 has several elements with limbs (the bear, fox, chick, and penguin), extensions (the dog and bunny ears), and long shapes (the snake). Figure 4.2 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 4.10 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.

Table 4.1: Data and statistics for RepulsionPak results. In these results, torsional forces are used only in Figure 4.11 and shape matching is used only in Figure 4.7.

Packing	Primary elements	Secondary elements	Vertices	Primary running time (seconds)	Secondary running time (seconds)	Iteration
Animals (Figure 4.1)	25	14	2412	133	65	16670
Rhino (Figure 4.7)	107	0	4833	237	0	15521
Cats (Figure 4.9)	41	69	3598	185	62	8531
Birds (Figure 4.10a)	43	43	2309	102	54	11571
Bats (Figure 4.10b)	47	22	3048	165	56	13120
Butterflies (Figure 4.10c)	123	135	11916	696	616	14379
Giraffes & Penguins (Figure 4.11)	60	0	2250	163	0	9347
Paisley (Figure 4.12)	162	0	4860	128	0	23040
Circles in Paisleys (Figure 4.12)	144	0	2544	403	0	29584

The bird packing in Figure 4.10 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 4.11 demonstrates torsional forces that gradually turn the elements upside down. The rhinoceros packing in Figure 4.7 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 4.12. We seed a central square with elements, but also place clones of those elements in all the squares of the 8-neighbourhood around the centre. These clones track the transformations 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

Figure 4.9: (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.

shown in Figure 4.13, extreme parameter settings can still produce degenerate results. In Figure 4.13a, 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 4.13b, edge and torsional forces dominate repulsion, producing a packing with stiff, upright elements that do not fill their container effectively.

Two professional artists have used RepulsionPak to create packings shown in Figure 4.14. The skull packing is made of butterfly and plant elements arranged inside multiple containers. The skull's teeth are made of manually placed leaf elements. The background is created by superimposing several packings. The smoothies packing is made of fruit elements arranged inside a smoothie cup. Additionally, a casual user made a packing using RepulsionPak, which was then printed on t-shirts for a fitness club at a tech company, as shown in Figure 4.15.

4.9 Conclusions and Future Work

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

Figure 4.10: 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.

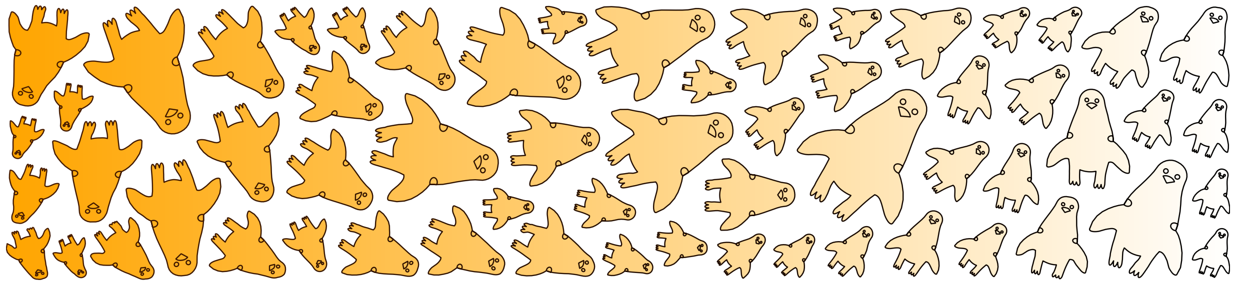


Figure 4.11: A packing that demonstrates torsional forces. The packing uses copies of a single element shape, but every copy is given a rest orientation between 0° and 180°, based on its horizontal position in the container. In the final packing the elements transition from upright to upside-down, recreating an illusion in which giraffe heads become penguins.

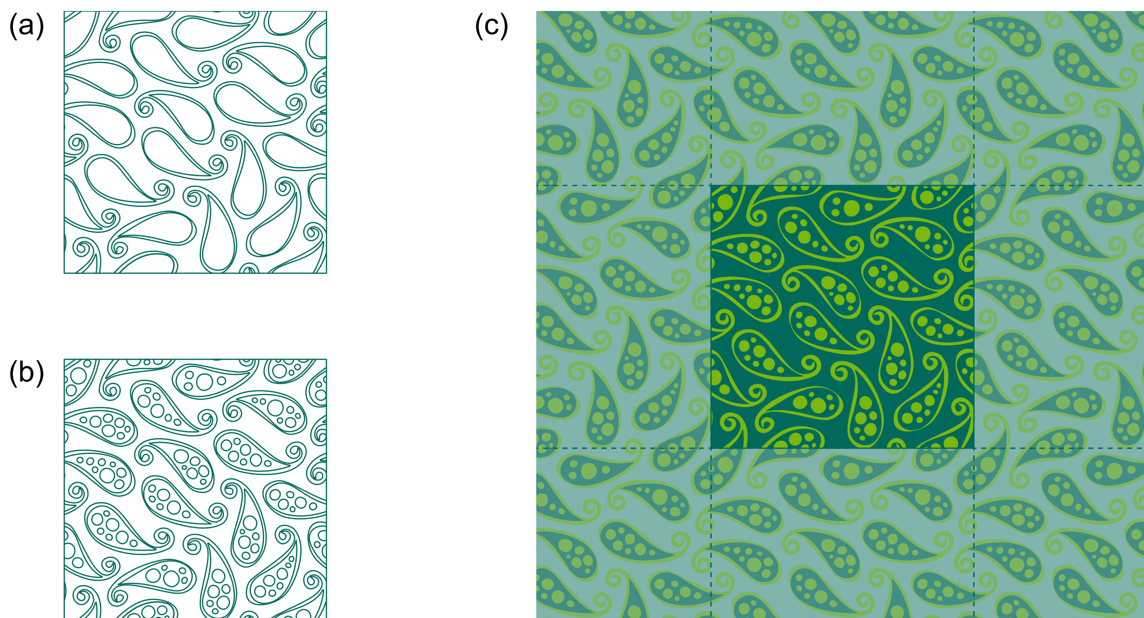


Figure 4.12: A paisley-inspired toroidal packing that can tile the plane. (a) An initial paisley packing. (b) In a separate simulation, we fill each paisley with circles to demonstrate a packing inside a packing. (c) The final result.

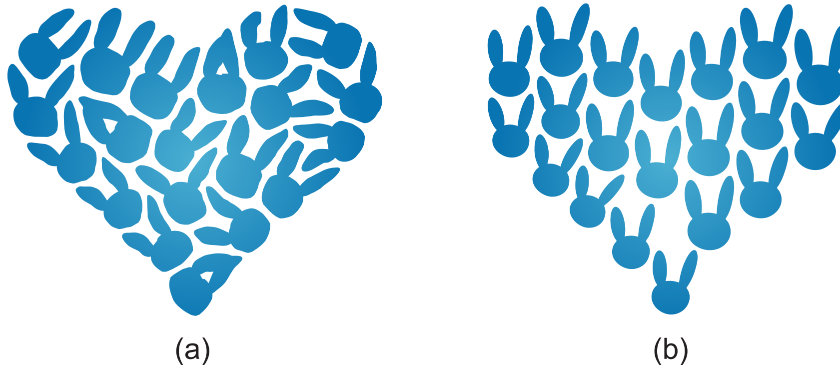


Figure 4.13: Two examples of how extreme parameter values can lead to low-quality results. In (a), we allow repulsion to overwhelm element shape by setting k_{rpl} to 200 and k_{edg} to 20; the resulting packing has even negative space, but elements suffer from high deformation and self-intersections. 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.

with its neighbours 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.

We see several opportunities for further improvements to RepulsionPak. 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 future work should experiment with several to investigate if any offers a suitable trade-off between performance and quality. Additionally, our barycentric warping method can introduce undesirable artifacts in highly deformed elements, as in the swallow tails in the left result of Figure 4.10. 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, for example, work of Jacobson et al. [JBPS11], Liu et al. [LJG14], and James [Jam20].

Figure 4.14: Artist-made packings using RepulsionPak. The top arrangement was made by Paul Trani, consisting of multiple containers filled with butterfly and plant elements. The manually-placed leaves are fixed elements that represent the skull's teeth. The background was made by superimposing several packings. The bottom packing was made by Daichi Ito, where fruit elements are arranged inside a smoothie cup.

Figure 4.15: A RepulsionPak packing printed on a t-shirt, created by Paul Asente.

Chapter 5

AnimationPak: Packing Elements with Scripted Animations

5.1 Introduction

During the development of RepulsionPak, we observed that the simulation process, while intended only as a means to an end, was itself enjoyable to watch. Elements would move and deform in an active, almost organic way as they responded to physical forces. These animations inspired us to explore a method for generating packings in which elements are augmented with scripted animations.

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 to produce 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 be-

cause of the difficulty of creating an animated packing by hand. Finding motivating examples created by artists is difficult. We found only a single animated packing, shown in Figure 5.1, which is an 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

When we consider adding scripted animations to a packing, we must decide separately whether to support animated elements and an animated container shape. Work in this area can then be categorized naturally into a 2×2 grid, as shown in Figure 5.2. We discuss these categories and techniques in greater detail below.

- **Static elements in a static container.** These are standard 2D packing algorithms that produce a single drawing of a static container packed by static elements.
- **Static elements in an animated container.** These algorithms allow the container to move and deform. The elements do not have scripted animations, but they might still animate passively in response to changes in the container.
- **Animated elements in a static container.** The container is fixed, but the elements are endowed with scripted animations. Elements must negotiate for space within the container as they attempt to express their animations.
- **Animated elements in an animated container.** In this category, both elements and the container have scripted animations. We are not aware previous work that falls into this category.

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. Initially, elements are placed using centroidal area Voronoi diagrams (CAVDs). Elements are then advected frame-to-frame using a choice of methods motivated by Gestalt grouping principles, aiming to generate group motions of elements, instead of distracting uncoordinated movements of individual elements. As the container's area changes, elements are added and removed as needed, while attempting to maximize overall temporal coherence. Packings generated by Animosaics fall into the category of static elements in an animated container.

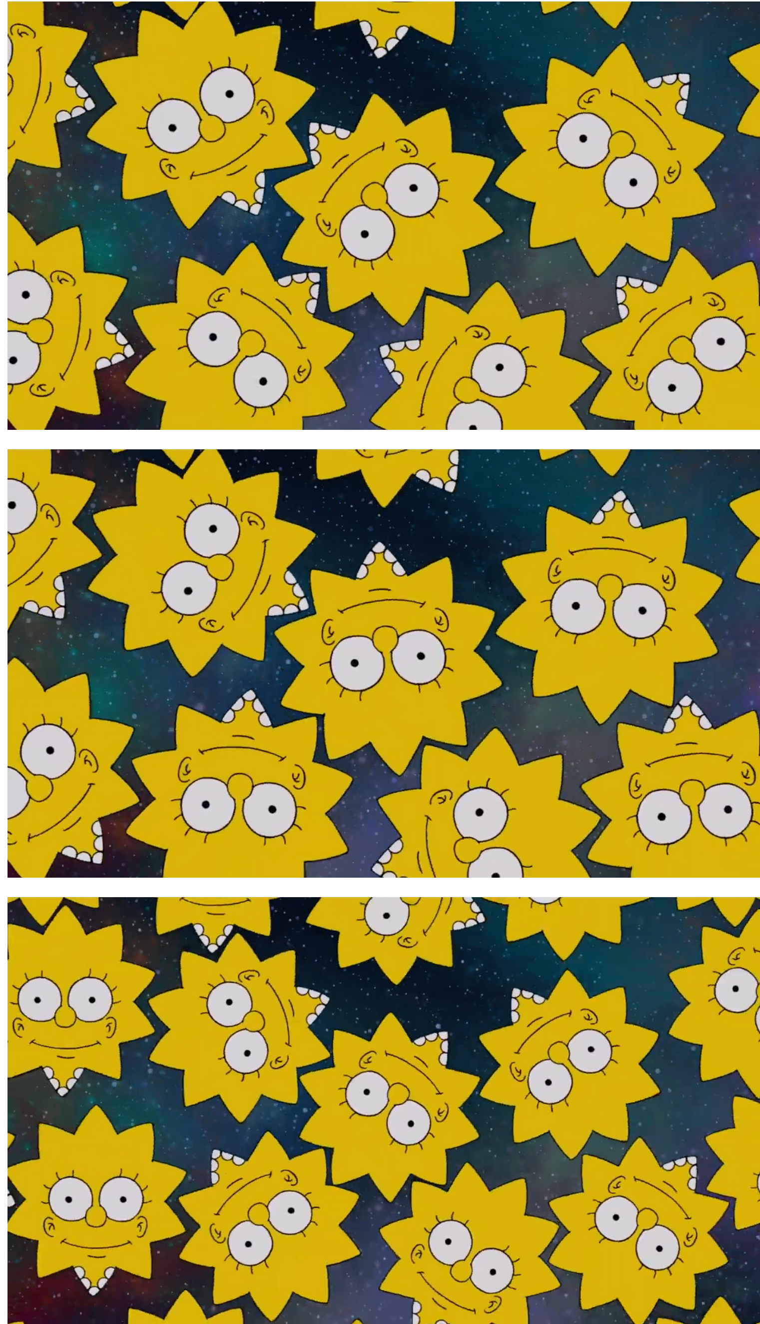


Figure 5.1: Three frames of an animated packing from The Simpsons, Season 31, Episode 19 (© FOX), showing gear-like meshing of oppositely-rotating copies of Lisa Simpson.

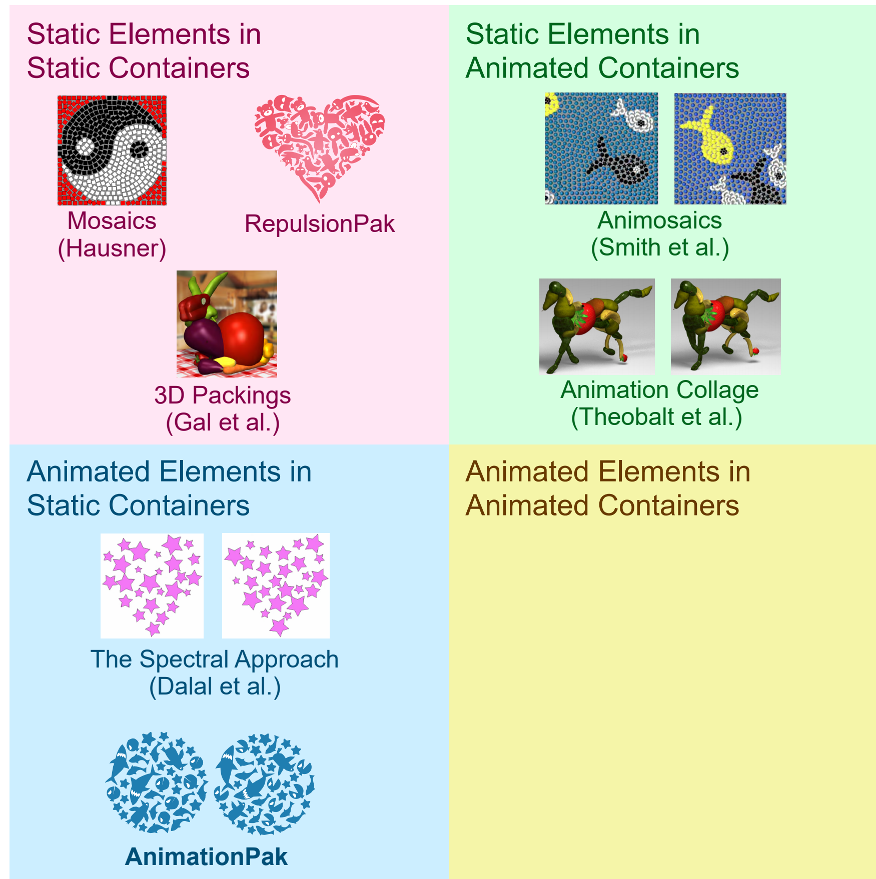


Figure 5.2: A categorization of animated packings.

Initially, Elements are placed using centroidal area Voronoi diagrams (CAVDs). Elements are then advected frame-to-frame using a choice of methods motivated by Gestalt grouping principles, that aim to generate group motions of elements, instead of distracting uncoordinated movements of individual elements.

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. 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. Animated packings generated by the spectral approach fall into the category of animated elements in a static

container.

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. Both techniques fall into the category of static elements in an animated container.

Animated 3D Packings: Theobalt et al. [TRdAS07] developed a data-driven method to generate animated 3D collages that resemble Arcimboldo paintings by arranging static 3D elements into an animated 3D container. They partitioned the input animated container by identifying rigid parts, each of which 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 with some degree of deformation to improve the fit with the container. This work falls into the category of static elements inside an animated container.

Animated tilings: We discussed metamorphosis tilings in Section 2.3, consisting of slowly evolving interlocked elements. Although the entire composition of a metamorphosis tiling is static, the evolving elements can be interpreted as a spatial animation. More recently, 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.

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 a school of fish 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. Inspired by DET, Kazi et al. [KCG+14] developed an interactive user interface where an artist can draw curves to guide the motions of animated elements. Both techniques by Ma et al. and Kazi et al. do not belong to any category in the 2×2 grid above, as their arrangements do not have containers and elements can move freely. 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 inputs to AnimationPak are similar to those described in Section 1.1, with the addition of a scripted animation for each element. 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 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. As with 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 an even distribution of negative space, we first offset the shape's paths by a distance of $d_{\text{gap}}/2$, with the goal of maintaining a separation of d_{gap} between elements in the resulting packing. (Figure 5.3a). In our system we scale the shape to fit a unit square and set $d_{\text{gap}} = 0.08$.

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 $\{s_1, \ldots, s_{n_{\text{s}}}\}$, with slice s_i being placed at time $(i-1)/(n_{\text{s}}-1)$. Higher temporal resolution will produce a smoother final animation at the expense of more computation. In our examples, we set $n_{\text{s}} = 100$. Figure 5.3d shows a set of time slices, with $n_{\text{s}} = 5$ 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}, 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 $n_{\text{s}} = 5$.

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.

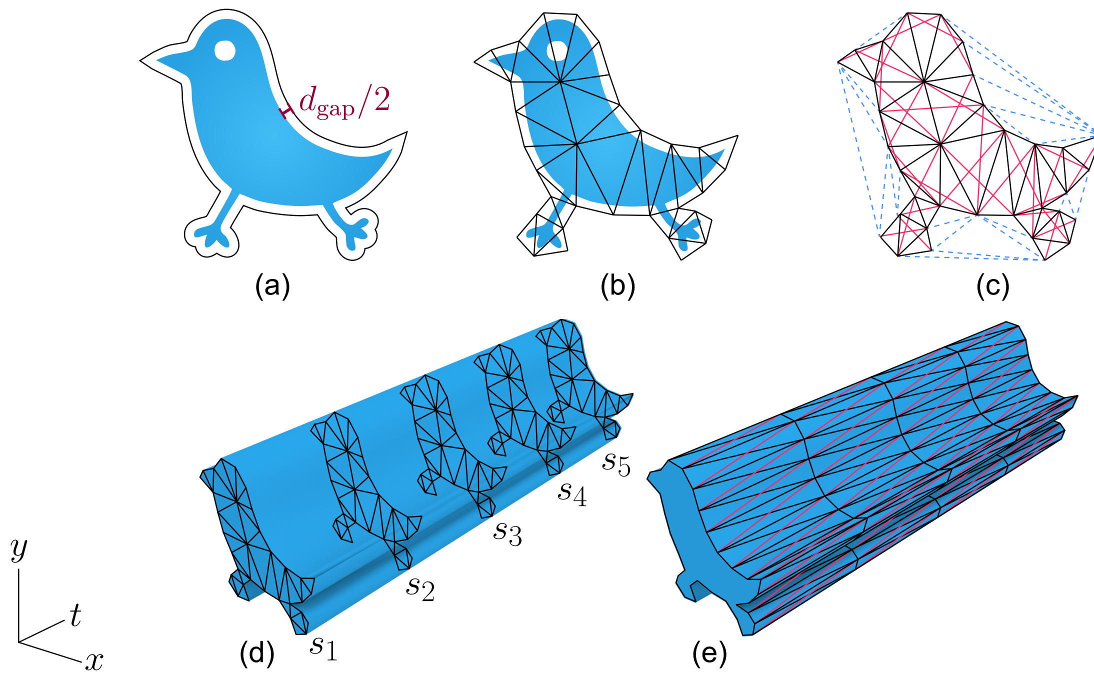


Figure 5.3: The creation of a discretized spacetime element. (a) A 2D element shape offset by $d_{\text{gap}}/2$. (b) A single triangle mesh slice. (c) Shear edges (red) and negative space edges (blue). (d) A set of five slices placed along the time axis. (e) The vertices on the boundaries of the slices 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 than in (d) and (e).

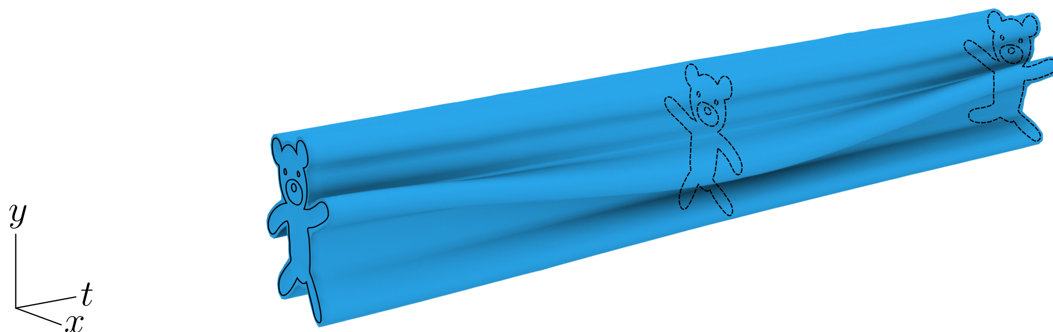


Figure 5.4: A spacetime element with a scripted animation.

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 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.

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

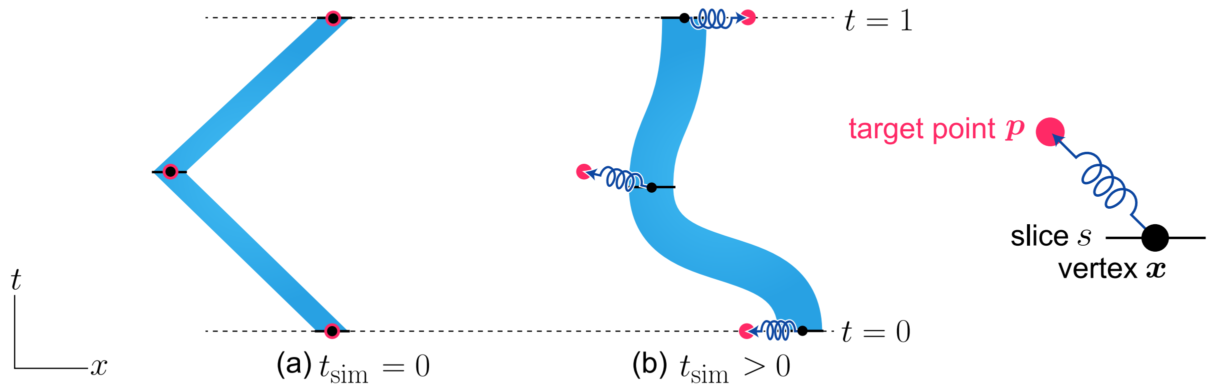


Figure 5.5: A 2D illustration of a guided element. Slices are depicted as black lines and slice vertices as black dots. A spring connects the centremost vertex $\boldsymbol{x}$ of a slice s to a target point p. (a) The initial shape of a guided element is a polygonal extrusion. (b) The spacetime element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

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. As seen in Figure 5.6a, 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. In practice, we shrink these elements to 5–10% of their original size.

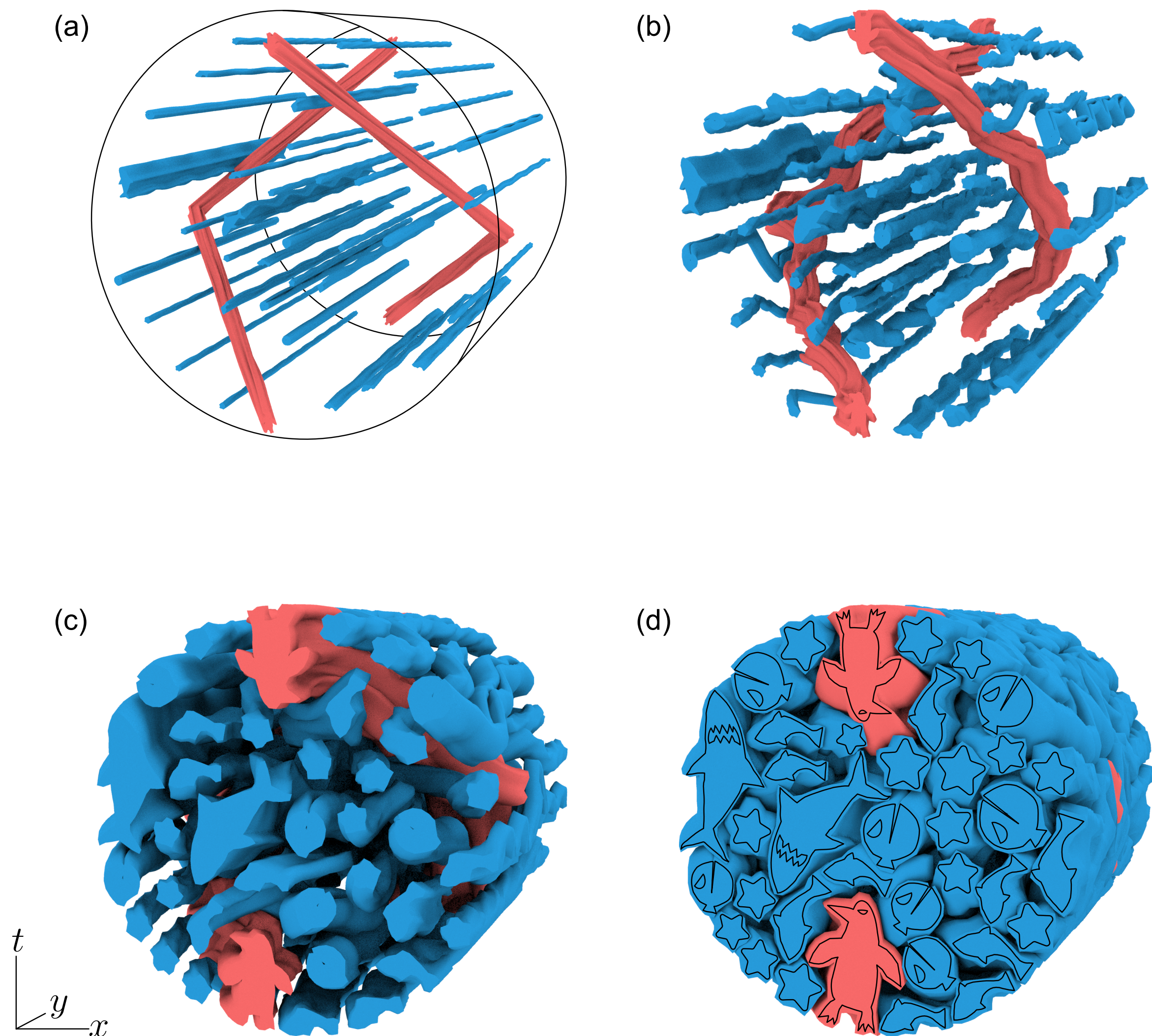


Figure 5.6: The simulation process. (a) Initial placement of shrunken spacetime elements inside a static 2D disc, extruded into a cylindrical spacetime domain. Guided elements are shown in red and unguided elements in blue. (b) A physics simulation causes the spacetime elements to bend. They also grow 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.

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 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

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

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; readers can refer back to Section 4.5 for equations used to generate them.

Repulsion forces allow elements to push away vertices of neighbouring 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, a given vertex $\boldsymbol{x}$ accumulates repulsion forces from points at different time positions on neighbouring elements. To locate these points on neighbouring 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. The edge forces are calculated using a quadratic version of Hooke's law, allowing more tolerance to small displacements while avoiding severe deformation. 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 spring type has a different relative strength.

Overlap forces resolve a vertex penetrating a neighbouring spacetime element. Overlaps can occur later in the simulation when negative space is limited. Once we detect a

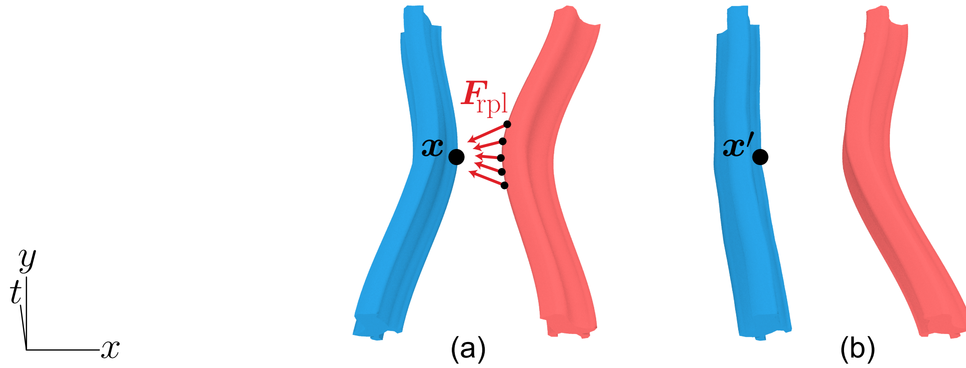


Figure 5.7: Repulsion forces applied to a vertex $\boldsymbol{x}$, allowing the element to deform and move away from a neighbouring element.

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 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 (Figure 5.8). Drifting slices can cause two problems. First, they could cause unexpected accelerations and decelerations in the final animation. Second, the deformation of the spacetime element would not be monotonic, and this would cause rendering artifacts as illustrated in Figure 5.10. 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

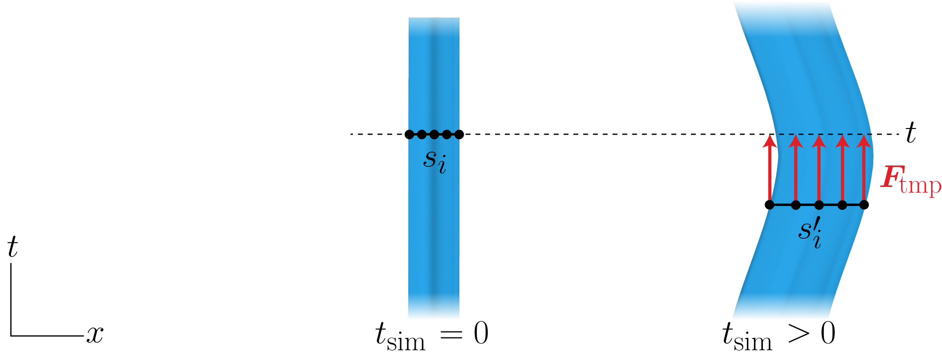


Figure 5.8: An illustration of the temporal force. The vertices in slice s'_i are drawn back towards time t.

velocities. These updates are scaled by a time step Δt_{sim} that we set to 0.01. We cap velocities at $10\Delta t_{sim}$ 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 bounding box is resized to a unit cube, we set $k_{rpl} = 10$, $k_{ovr} = k_{bdr} = 5$, $k_{tor} = k_{tmp} = 1$. We set $k_{edg} = 0.01$ for time springs, $k_{edg} = 0.1$ for negative-space springs, and $k_{edg} = 10$ 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 neighbouring 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.

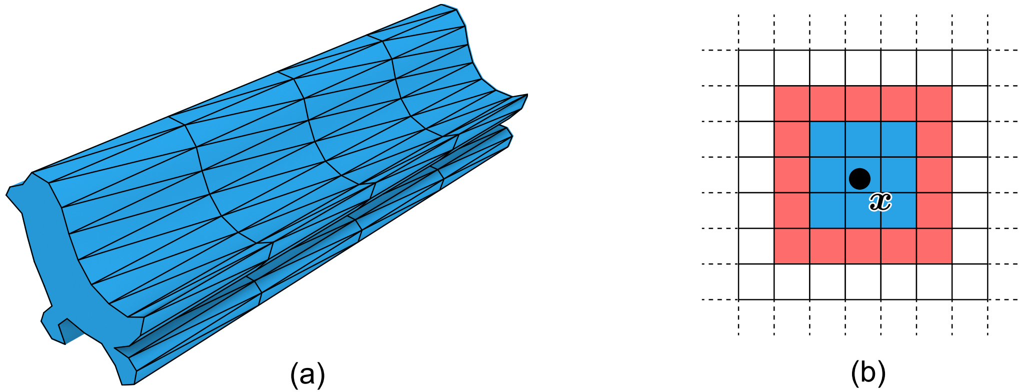


Figure 5.9: (a) The triangles that connect consecutive slices define the envelope of the element. The midpoints of these triangles are stored in a collision grid. (b) A 2D visualization of the region of collision grid cells around a query point $\boldsymbol{x}$ in which repulsion and overlap forces will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

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 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

Our system can provide robust simulations that arrange and deform spacetime extrusions. However, to eliminate potential problems during the rendering, we have to enforce two additional conditions. First, all spacetime elements must be present at any time position. Second, element slices have to be perpendicular to the time axis. In Figure 5.10, we show examples of acceptable and unacceptable configurations of a spacetime element.

We enforce two mandatory geometric constraints and an optional constraint on element

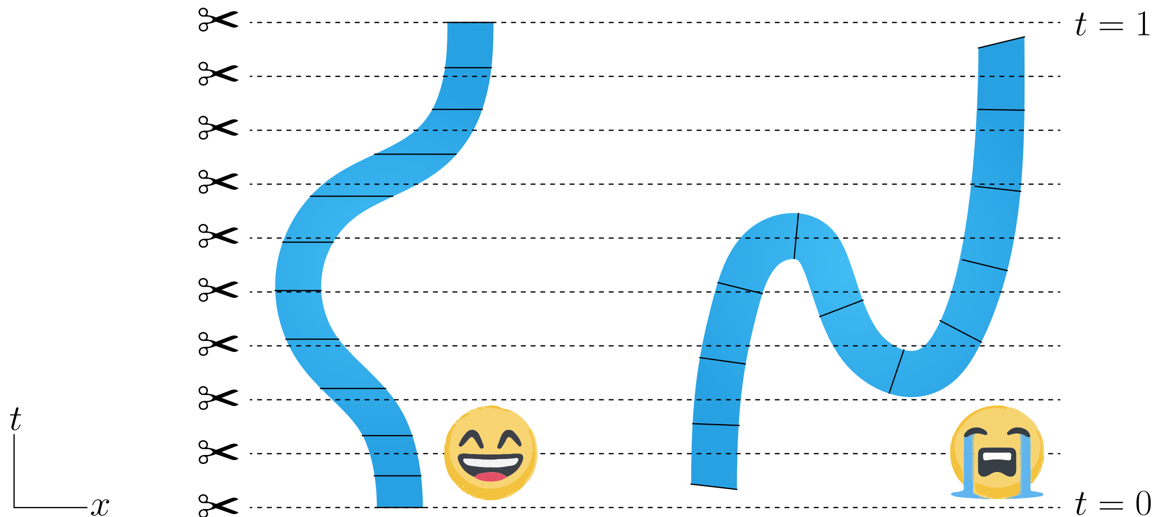


Figure 5.10: The left element shows an example of a desired configuration, while the right one will cause rendering artifacts since it does not have any of three required conditions: monotonic deformation, full extent from $t = 0$ to $t = 1$, and slices perpendicular to the time axis.

slices. Each of the following constraints is reapplied after every 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 5.11a shows an illustration of the end-to-end constraint. This constraint only performs localized adjustments on the first and last slices. Over time, these adjustments will spread throughout the elements as edge forces strive for equilibrium.

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. 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 5.11b shows an illustration of the simultaneity constraint.

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

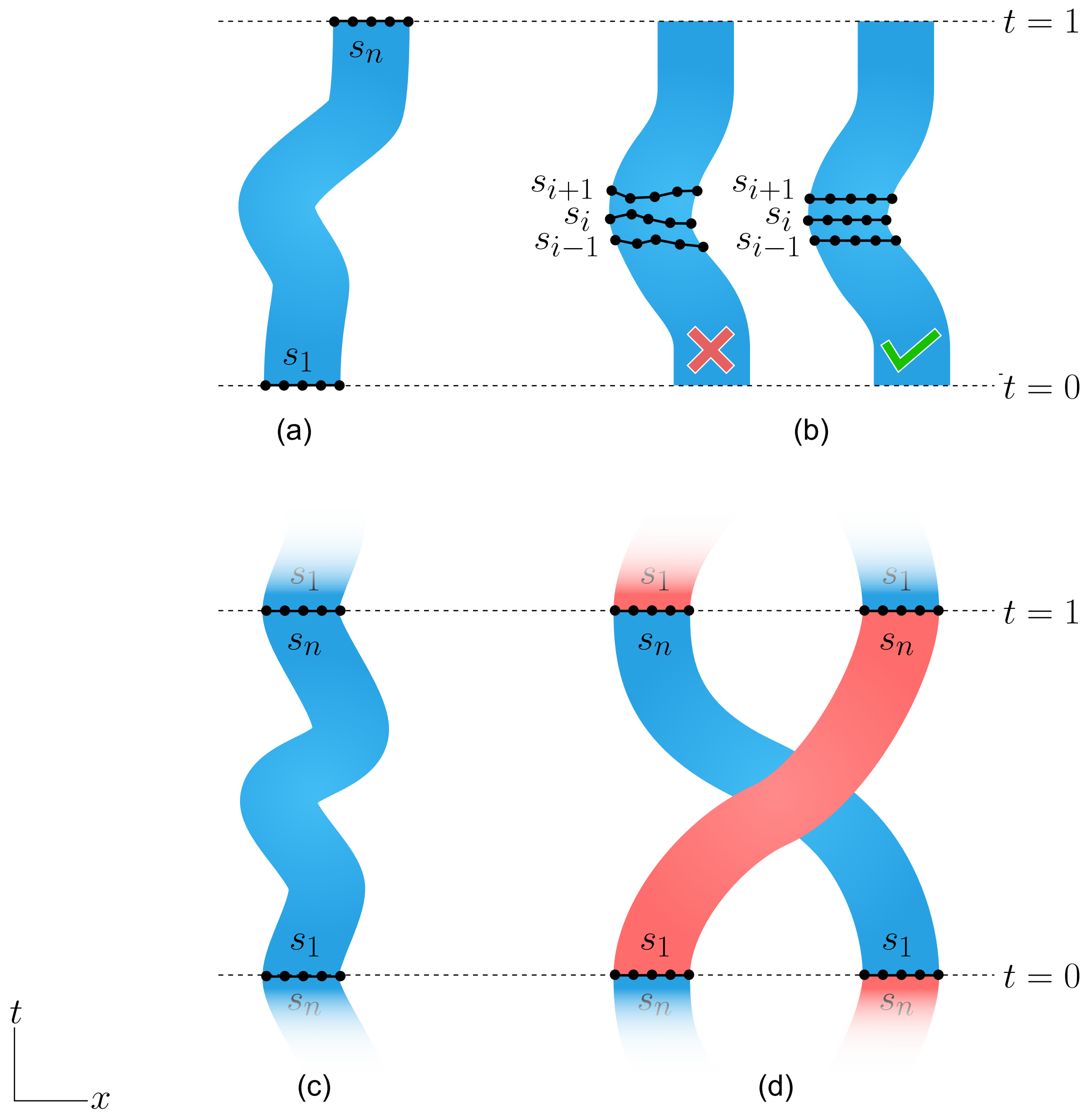


Figure 5.11: a) End-to-end constraint: slice s_1 and s_n, located at $t = 0$ and $t = 1$, should never change their t positions but can change their x, y positions. b) Simultaneity constraint: all vertices on the same slice should have the same t position. c) Loop constraint with a single element: the 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.

the $t = 0$ slice of *some* element e_2. We can have $e_1 = e_2$ (Figure 5.11c), but more general loops are possible in which the elements arrive at a permutation of their original configuration (Figure 5.11d). 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 5.12a,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 5.12).

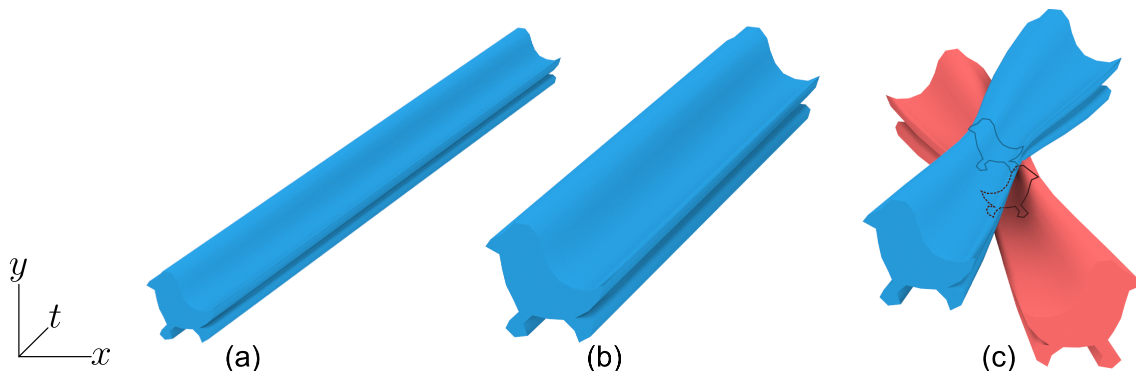


Figure 5.12: A spacetime element shown (a) shrunken at the beginning of the simulation, and (b) grown later in the simulation. (c) When two elements overlap somewhere along their lengths, they are temporarily prohibited from growing there.

Stopping Criteria: We halt the simulation when the space between neighbouring elements drops below a threshold. When calculating repulsion forces, we find the distance from every slice vertex to the closest point in a neighbouring element. The minimum of these distances over all vertices in an element slice determines that slice's closest distance to neighbouring 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 5.13. 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

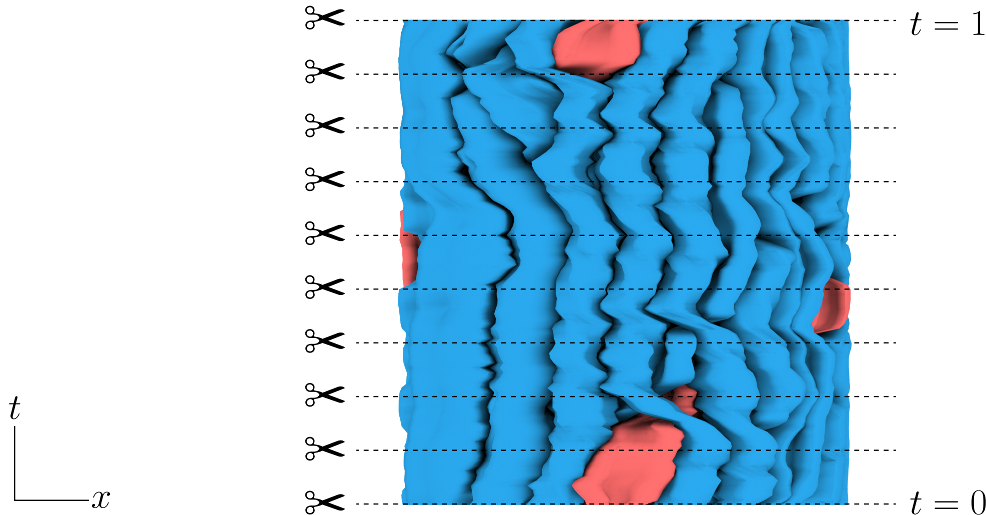


Figure 5.13: An illustration of rendering spacetime elements to generate 2D frames by cutting them at evenly spaced time values from $t = 0$ to $t = 1$.

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 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 neighbouring 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

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 5.14)	37	97,800	623,634	106,000	01:06:35
Snake and bird (Figure 5.15)	37	58,700	370,571	58,700	01:01:32
Penguin to giraffe (Figure 5.16)	33	124,300	824,164	143,000	01:19:50
Traveling bird (Figure 5.17)	32	63,700	389,373	70,100	00:19:24
Lion (Figure 5.18b)	16	39,400	236,086	41,800	00:41:56
Animals (Figure 5.20b)	34	69,600	444,337	69,800	01:00:19
Heart stars (Figure 5.19c)	26	85,200	598,218	85,800	00:23:08

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.

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. The processor is a quad core CPU with hyper-threading; 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.

Figure 5.14 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 5.14a. 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.

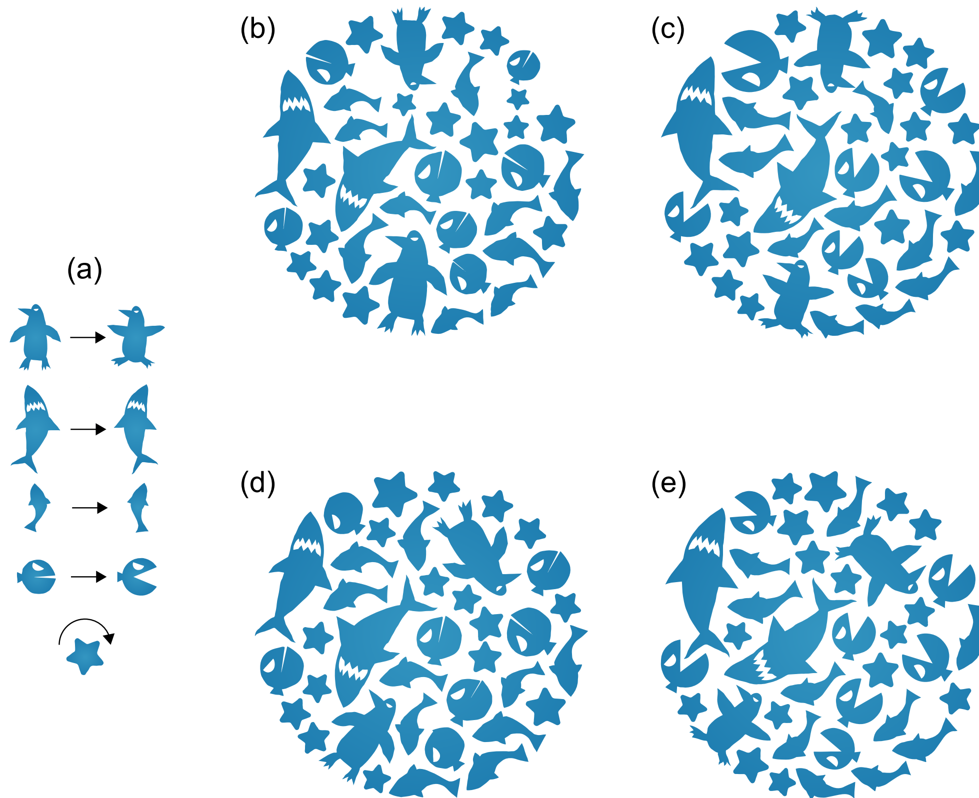


Figure 5.14: (a) Input animated elements, each with its own animation: swimming penguins, swimming sharks and fish, Pac-Man fish that open or close their mouths, and rotating stars. (b-e) four selected frames from an animated packing.

A snake chases a bird around an annular container in Figure 5.15, demonstrating a container with a hole and giving a simple example of the narrative potential of animated packings. Figure 5.16 animates the giraffe-to-penguin illusion shown as a RepulsionPak static packing shown in Figure 4.11. This example uses torsional forces to control slice orientations.

Figure 5.17 shows an example where a guided element is not constrained to the inside of a container. The traveling bird moves from left to right, passing through a packing of birds. As some slices of the traveling bird take up container space, the other birds have to stop some of their slice growth prematurely. This is a demonstration of the benefit of the non-uniform growth in element slices, allowing elements to adapt to changes in available container area. In the rendered 2D animation, the elements have a subtle shrinking-and-

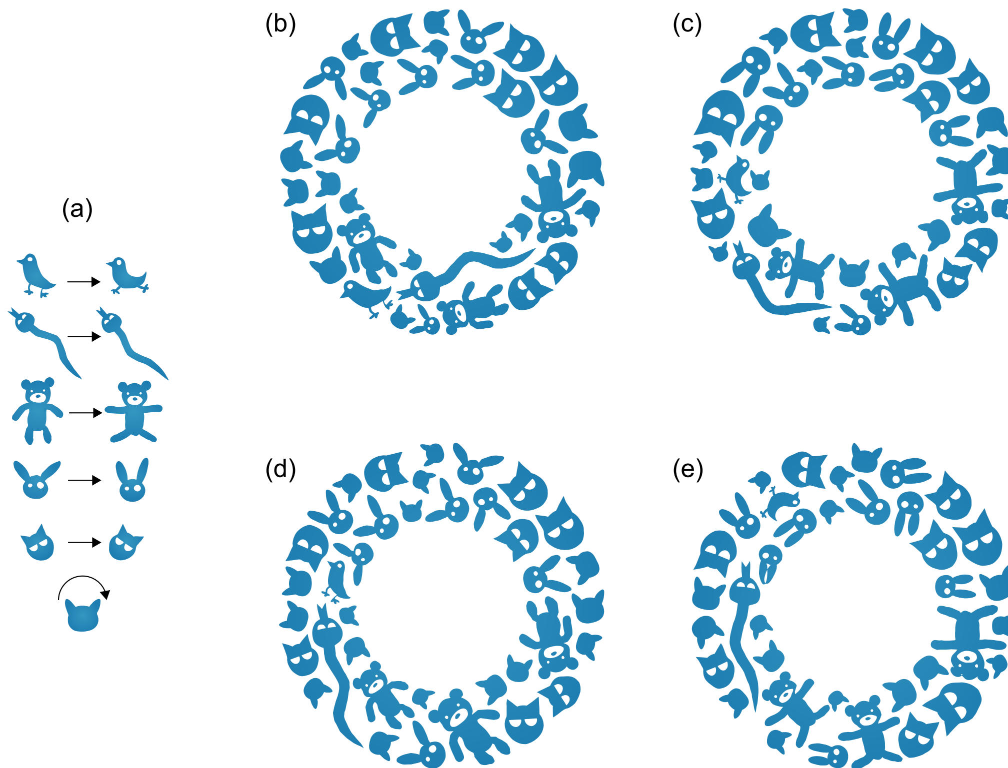


Figure 5.15: A snake chasing a bird through a packing of animals. The snake and bird are both guided elements that move clockwise around the annular container.

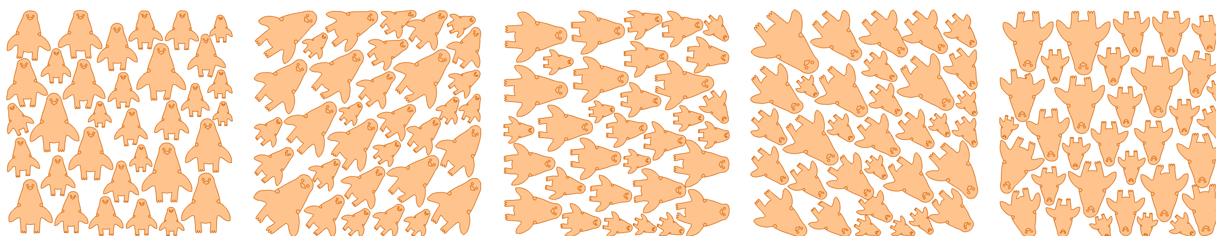


Figure 5.16: Penguins turning into giraffes. The penguins animate by rotating in place. Torsional forces are used to preserve element orientations. Frames are taken at $t = 0$, $t = 0.125$, $t = 0.25$, $t = 0.375$, and $t = 0.5$.

expanding effect. The composition can also tile horizontally, which can be seen in the supplemental video.

Figure 5.18a is a static packing of a lion's mane created by an artist and used as an example of flow-based packings (Chapter 1) Figure 1.3. In Figure 5.18b, 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. We call the slow motions of the lion's mane as ambient animation, a topic that we would like to pursue as future work.

Figure 5.19 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 counter-rotating stars.

Figure 5.20 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 5.21 emphasizes the trade-off between temporal coherence and evenness of negative space by creating two animations with different time spring 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. The visualization in (c) shows the trajectories of two identical elements in both cases, which makes it clear that weak time springs cause more frantic movements.

Figure 5.22 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 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 and Future Work

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.

We see a number of opportunities for improvements and extensions to AnimationPak:

- **Motion Paths:** Element motions in animated packings can look chaotic due to the interaction between time springs and repulsion forces. It may be possible to replace time springs with motion paths that connect slices together. The challenge is to generate plausible motion paths that can be modulated by other forces. We would also need to detect motion extrema of an animation, which are then used as constraints when generating a motion path. We would like to investigate motion path editing techniques such as work by Gleicher [Gle01] and Lockwood and Singh [LS11].

- **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, 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. As shown in the 2×2 classification of algorithms (Figure 5.2), we are not aware any previous method that can pack animated elements inside an animated container. AnimationPak has the potential to support animated containers, as demonstrated by the packing with a loose bird in Figure 5.17, in which elements can vary their slice sizes to adapt to changes in container area. As a further extension we could consider dynamically adding and removing elements during the animation. This would certainly complicate the process of giving every element an initial placement that is fully inside the container and disjoint from other elements.

- **Cinemagraphs**: The lion's mane packing shows the potential of AnimationPak to generate a cinemagraph, a composition with repeating slow ambient animations combined with static elements. There are many image processing techniques to generate cinemagraphs, for example, Video Textures [SSSE00] and Cliplets [JMD+12]. It would be interesting to the construction of illustrated cinemagraphs using animated element arrangements.

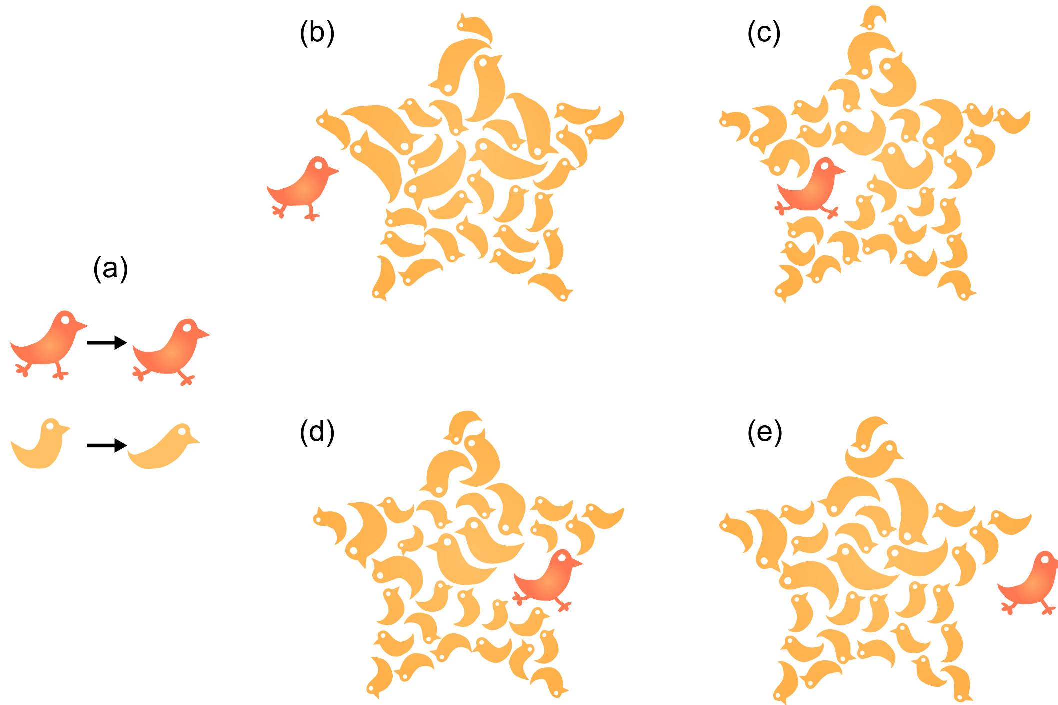


Figure 5.17: An animation of a traveling bird that comes in and out a packing of birds. The other birds adapt to the reduced space by prematurely stopping some of their slices' growths.

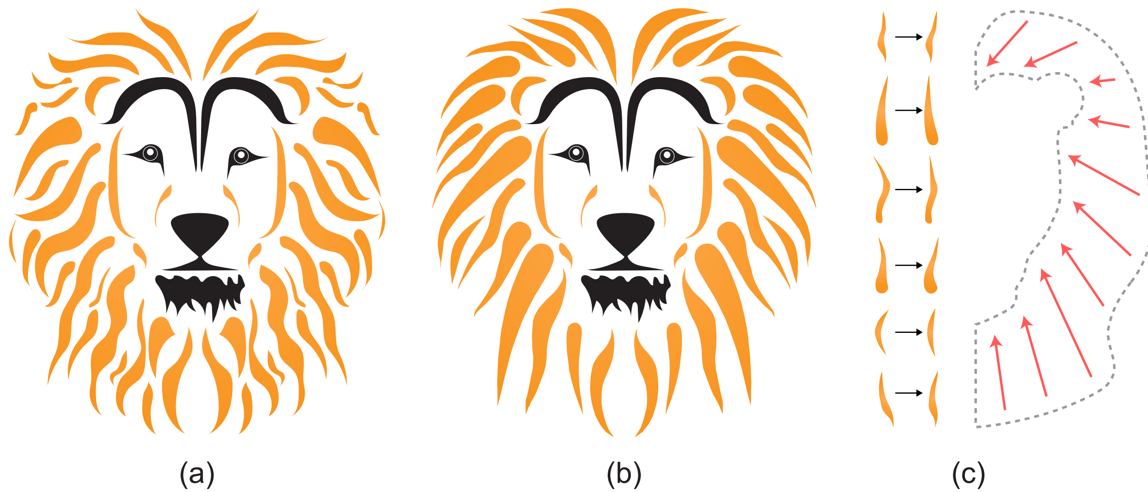
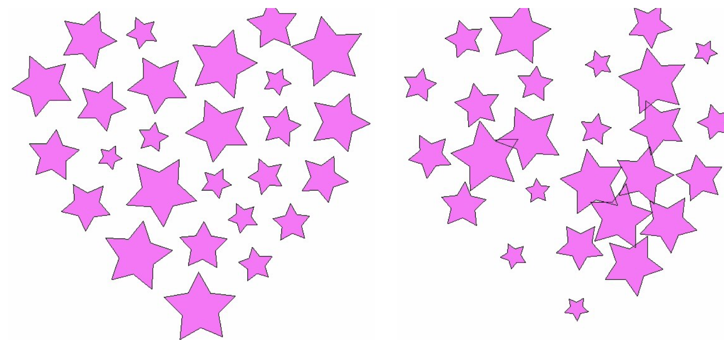
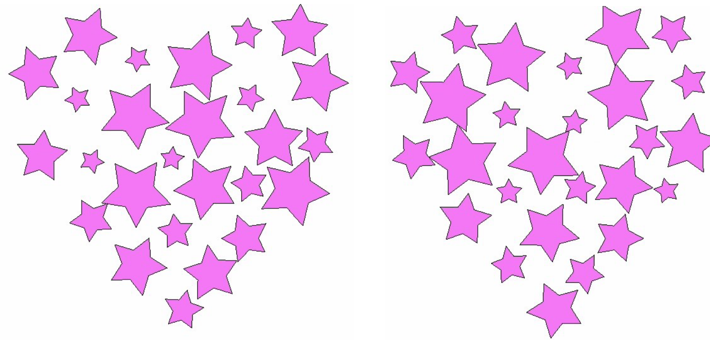


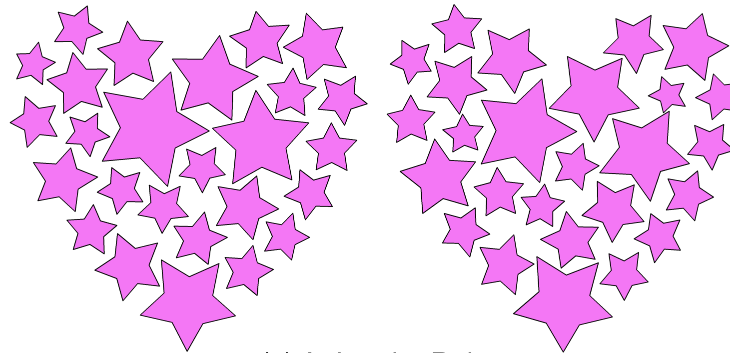
Figure 5.18: (a) A static packing made by an artist, taken from StockUnlimited. (b) The first frame from an AnimationPak packing. (c) The input animated elements and the container shape with a vector field. Torsional forces keep elements oriented in the direction of the vector field. We simulate half of the lion's mane and render the other half using a reflection, and add the facial features by hand.



(a) CAVD



(b) The Spectral Approach



(c) AnimationPak

Figure 5.19: 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.

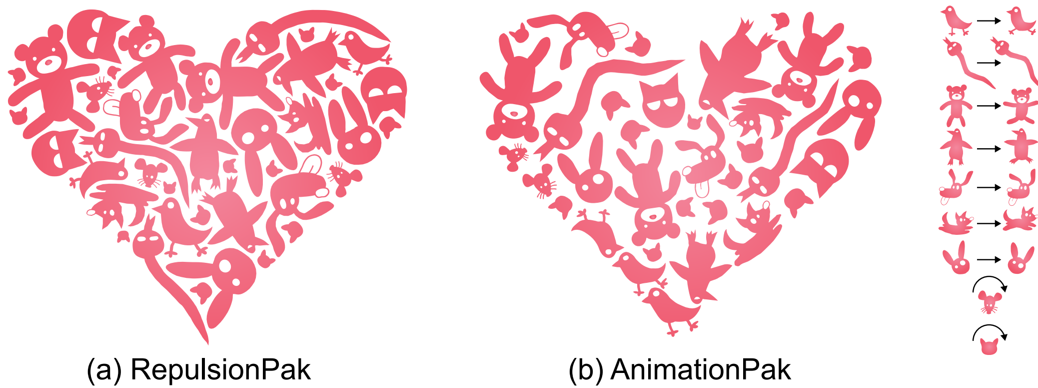


Figure 5.20: (a) A static packing created with RepulsionPak. (b) The first frame of a comparable AnimationPak packing. The input spacetime elements are shown on the right. The AnimationPak packing has more negative space because we must tradeoff between temporal coherence and packing density.

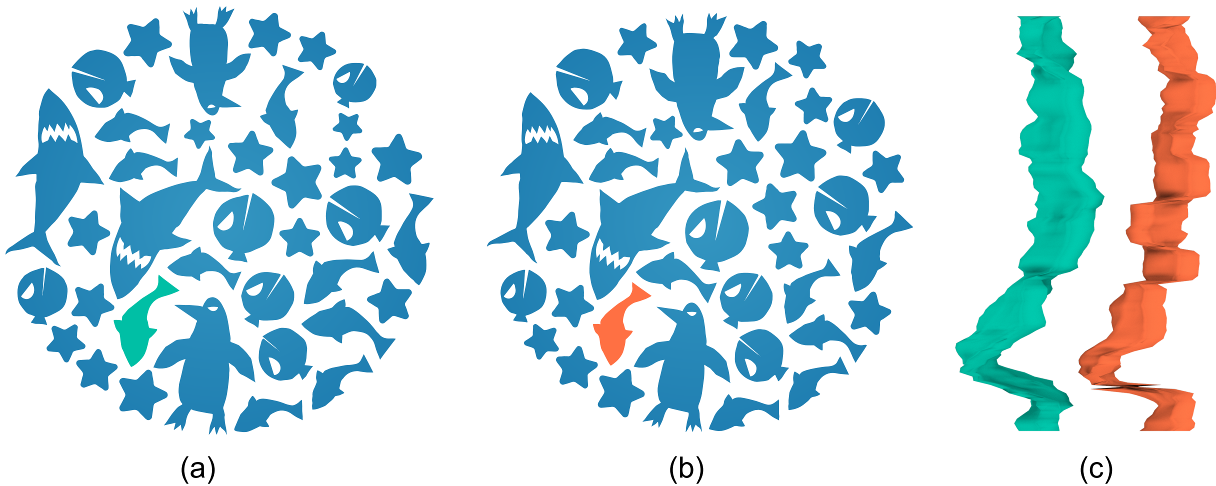


Figure 5.21: (a) One frame from Figure 5.14. (b) The same packing with time springs that are 1% as stiff. Reducing the stiffness of time springs leads to a more even packing with less negative space, but the animated elements must move frantically to preserve packing density. The spacetime trajectories 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.

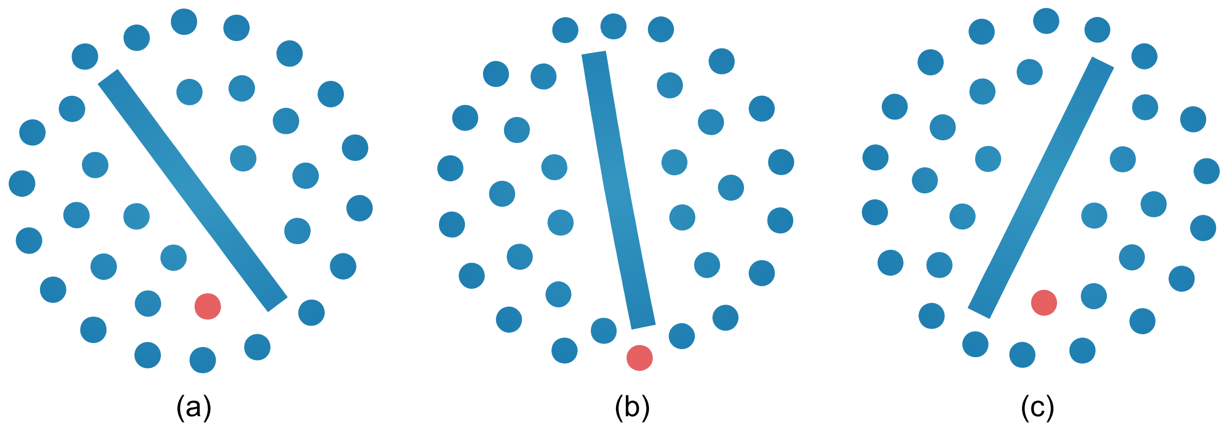


Figure 5.22: A failure case for AnimationPak, consisting of a rotating beam and a number of small circles. Instead of being dragged around by the beam, the circles dodge it entirely by sneaking through the gap between the beam and the container. The red circle demonstrates one such maneuver.

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 packing is a challenge. Understanding packing aesthetics is important, since it allows us to create better and more elegant packings. There are two ways to approach a packing evaluation. First, we could perform a qualitative evaluation by conducting user studies. Second, we could develop quantitative metrics to measure statistical and geometric features of packings. This chapter chooses the latter option by developing metrics that are useful to evaluate packing algorithms, with the goal of measuring progress in packing research. We see these metrics as a preliminary attempt to quantify packing aesthetics, and we hope that our work will inspire new research in packing evaluation.

The visual appeal of packings follows in part from aligning neighbouring 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 neighbouring elements should be roughly the same everywhere. In this chapter, we have developed several measurements of evenness, which allow us to examine packings made by real-world artists and to compare RepulsionPak with other packing algorithms. Packings generated by FLOWPAK and AnimationPak are not included in this chapter. FLOWPAK prioritizes the flow principle over the evenness of negative space, and AnimationPak's spacetime packings are best left until we have developed a greater understanding of the static 2D case.

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 measure the similarity between exemplars and synthesized textures. They conducted two user studies: a pile-sorting study and a pairwise comparison study. These user studies were concerned with the distribution of elements, without regard for 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 participants to compare packings based on their 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) in each of three categories: their preference, stylishness, interlocking. In their study, PAD received the highest scores in all three categories. We see the user study could be expanded to discuss more categories, such as design principles discussed in Chapter 1. Additionally, their results included occasional element overlaps, which they did not discuss (Section 6.4).

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 their energy measurement 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, and 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 computed the spatial relationships of these centroids, 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

We develop metrics that summarize the distribution of negative space as a data visualization consisting of 2D functions. By generating and displaying these functions, we can compare multiple packings. These metrics do not apply to tilings, in which elements interlock perfectly. They also do not attempt to quantify the aesthetics of discrete textures since they do not analyze the positive space distribution.

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. Figure 6.1 shows an example of a typical $Q_s(r)$, showing it to be a decreasing monotonic function. In order to interpret the SCP, it is helpful first to examine a "packing" with perfectly even negative space (Figure 6.2a). 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, $Q_s(0) = d_{\mathrm{gap}}/(d_{\mathrm{gap}} + d_{\mathrm{s}})$; it is also clear that $Q_s(d_{\mathrm{gap}}/2) = 0$, 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 6.2b), the intersections of horizontal and vertical channels produce pockets of negative space that can accommodate balls of radius $d_{\mathrm{gap}}\sqrt{2}/2$. 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 $r = d_{\mathrm{gap}}/2$, 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. 6.2c,d), the negative space will be narrower in some places and wider in others, recognizable as a shallower SCP with a longer tail.

The SCP is expressed in terms of probabilities as it originates from spatial statistics. However, it also has a simple geometric interpretation that leads naturally to a simpler and

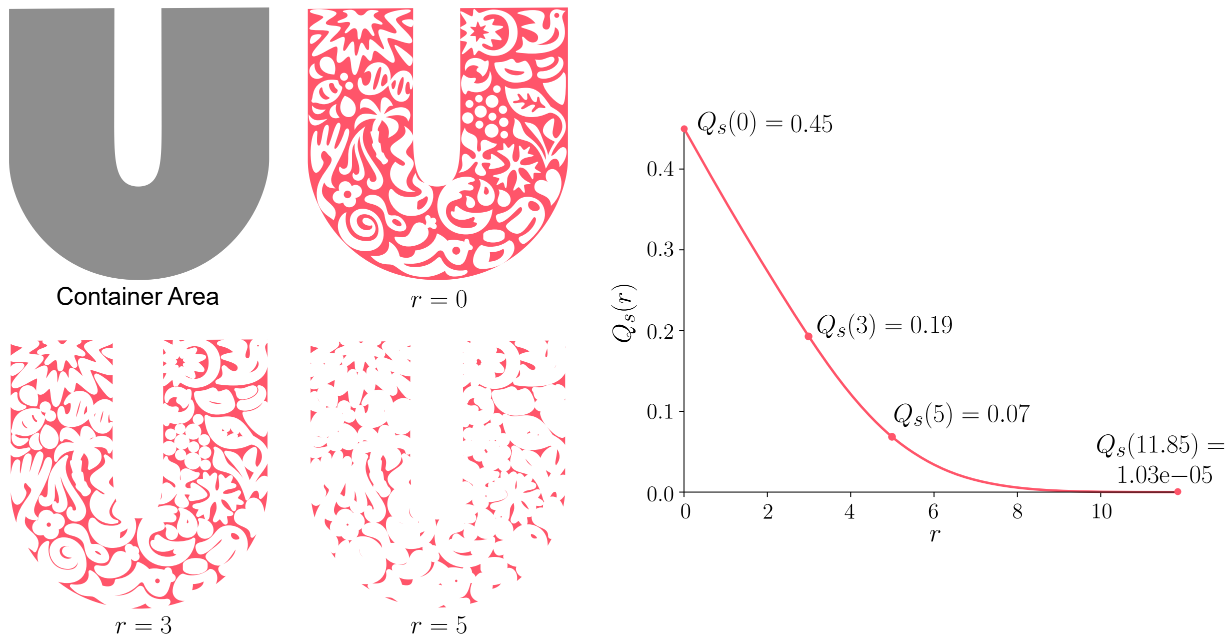


Figure 6.1: The spherical contact probability $Q_s(r)$ is defined as the probability that a disc of radius r, chosen uniformly at random within the container region, lies entirely within the packing's negative space. Geometrically, we compute the value of the SCP as the ratio between the inward-offset negative space area (drawn in red above) with the container area (drawn in grey). We show several offsets of the negative space with discs of radius $r = 0$, $r = 3$, and $r = 5$. On the right, we show a plot of $Q_s(r)$. We can observe that the positive space ratio is $Q_s(0) = 45\%$ and the largest disc that can be inserted into the negative space has radius $r = 11.85$.

more precise computation. We geometrically compute the 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. Figure 6.1 shows the illustration of the geometric interpretation of the SCP.

6.3.2 Histograms of the Distance Transform

In the context of 2D packings, a distance transform is a scalar field located in the negative space where every point contains the distance r value to the nearest point on the boundary

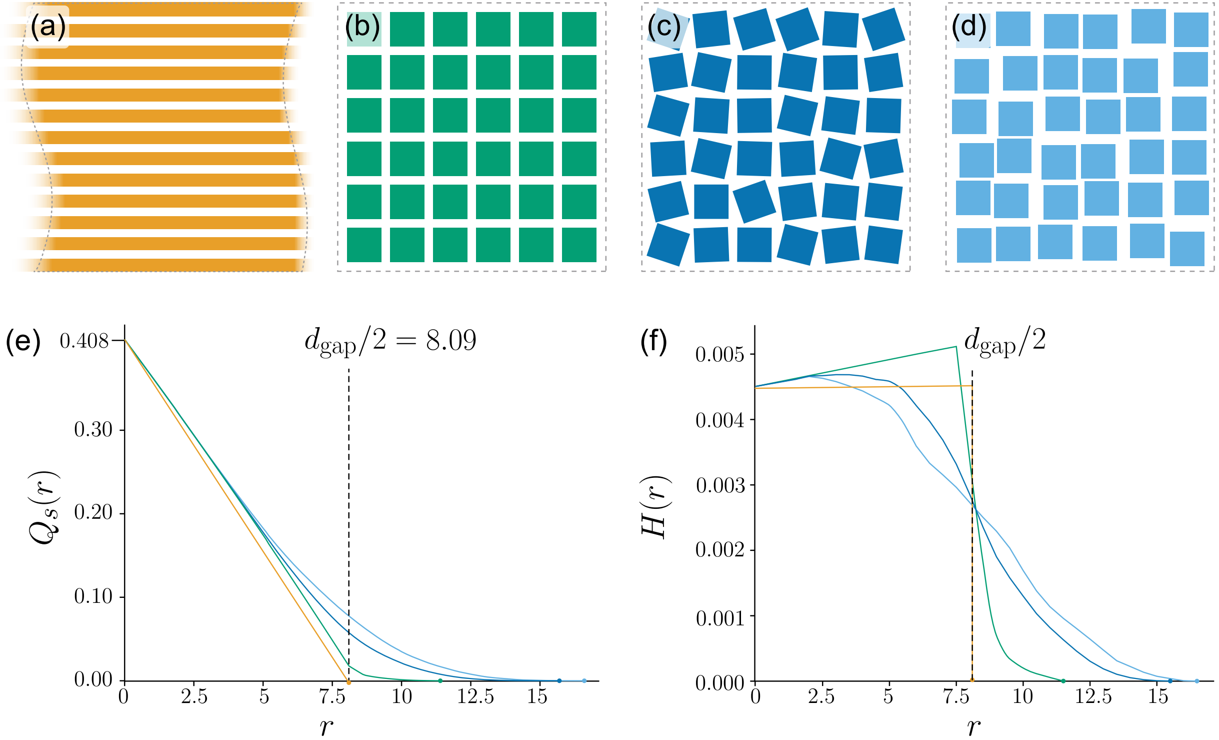


Figure 6.2: Spherical contact probabilities and distance histograms for reference packings. A "perfect packing" of infinite stripes is shown in (a), followed by a square packing with the same area fraction and negative space width d_{gap} in (b). The square packing is then perturbed with random rotations in (c) and translations in (d). The corresponding SCP functions and histograms are plotted in (e) and (f).

of an element. Note that the value r can also be interpreted as the disc radius in SCP. In practice, we construct a discrete approximation of the distance transform in a uniform 2D grid by computing this distance for the centre of every grid cell. The computation can be accelerated by storing the element boundaries inside a quadtree. An alternative approach is a level-set method that propagates the fronts of the element boundaries [OS88]. Figure 6.3 shows an example of a discrete distance transform, where lighter pixel colours represent higher r values.

The histogram of a distance transform provides insights into how negative space varies. However, this would require quantizing distance 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 packings with the same negative space area, the areas under their distance histograms are the same. 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 6.2, 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

This section compares various 2D packings using three metrics discussed previously. Our comparisons are limited to static 2D packings that have clear boundaries between positive space and negative space.

Calibration: Two or more packings must be calibrated to each other before they can be compared meaningfully. The calibration process requires packings to satisfy these conditions:

1. All packings should use an identical container shape. We then normalize the container to have unit area.

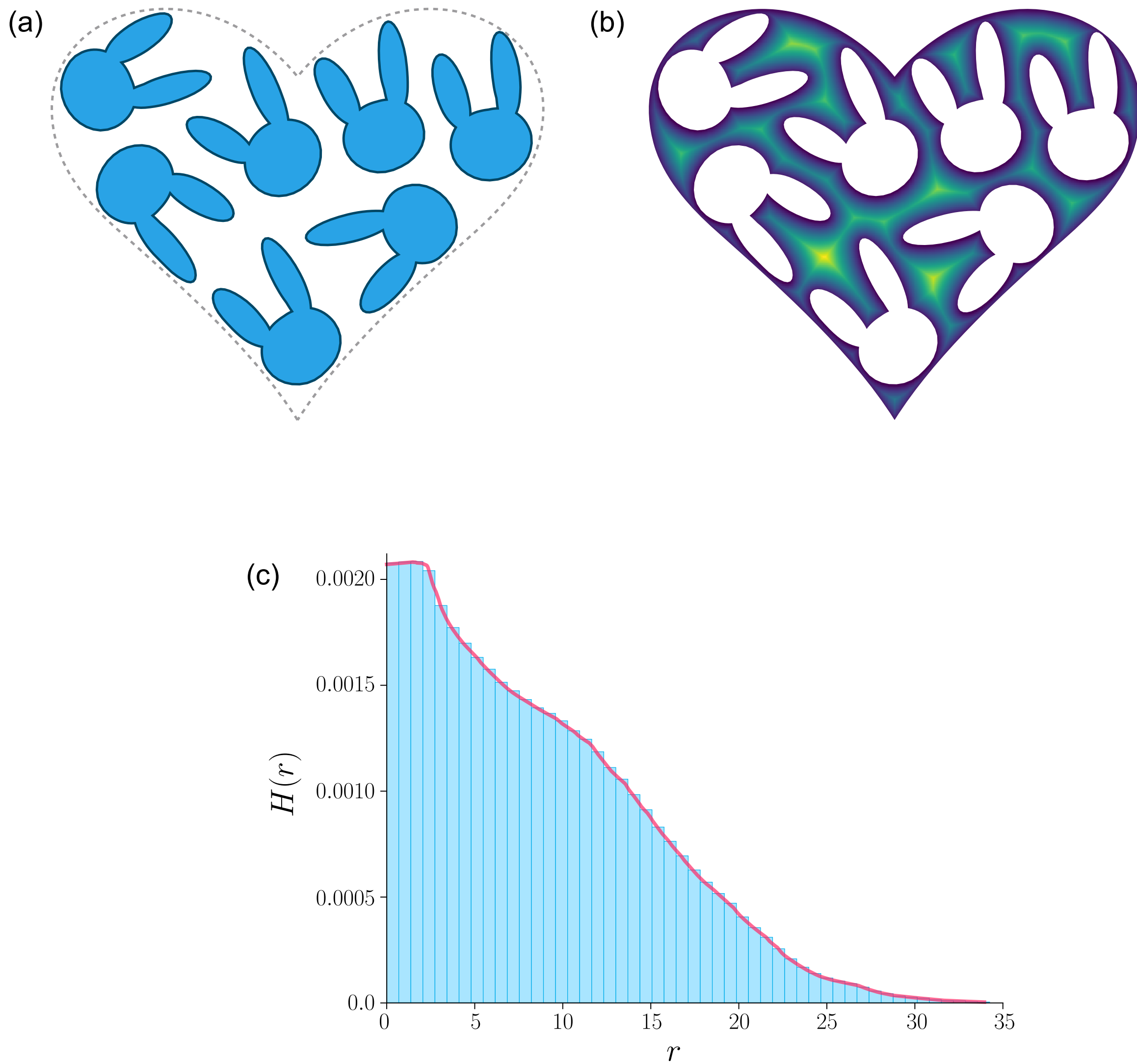


Figure 6.3: (a) An arrangement of rabbits. (b) The distance transform of the negative space, where lighter pixel colours represent higher r values. (c) Two visualizations of the distance transform; the continuous histogram $H(r)$ drawn as a red curve and the discrete histogram with 50 bins drawn as blue rectangles.

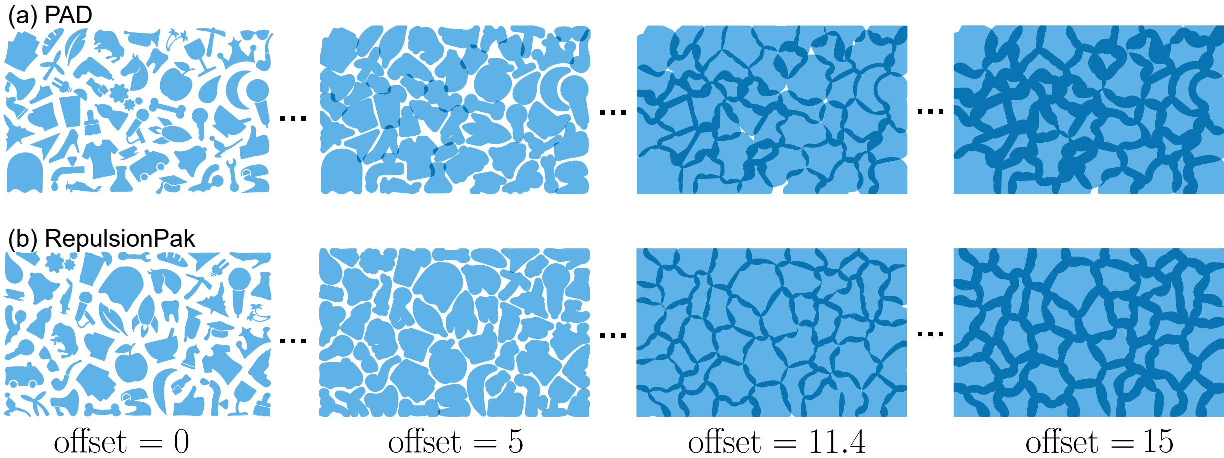


Figure 6.4: An illustration of offsetting elements outward. The packings have $d_{\text{gap}}/2 = 5.7$. At an offset of 5, which is slightly less than $d_{\text{gap}}/2$, the overlap for the PAD packing is 1.462% of the total area, while the overlap for our packing is only 0.039%. At an offset of 11.4, 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. The corresponding overlap function is shown in Figure 6.5e.

2. We must arrange for the packings to have the same *negative space ratio* (the overall amount of negative space as a fraction of container area), in order to permit fair comparisons of the distributions of that negative space.

3. All packings should have the same set of elements, ensuring the difficulty of the packing process remains the same.

4. All packings should have the same element relative sizes, as changing the size of an element can make it harder or easier to interlock with other elements.

In RepulsionPak simulations, we offset the elements outward by $d_{\text{gap}}/2$ to create element skins (Section 4.4). This allows us to estimate the expected final width of negative space as d_{gap}. Note that calibrated packings have the same negative space area so they should have the same expected of d_{gap}. In a good packing, the gaps of negative space would be close to d_{gap}. On the other hand, the gaps in a bad packings would have variations in widths which are smaller or bigger than d_{gap}.

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 (Section 4.5). Note that we only modify the strengths of edge springs and shear springs, leaving the strengths of negative space springs 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. This comparison corresponds to our intuition that edge forces allow us to trade off between element deformation and evenness of negative space.

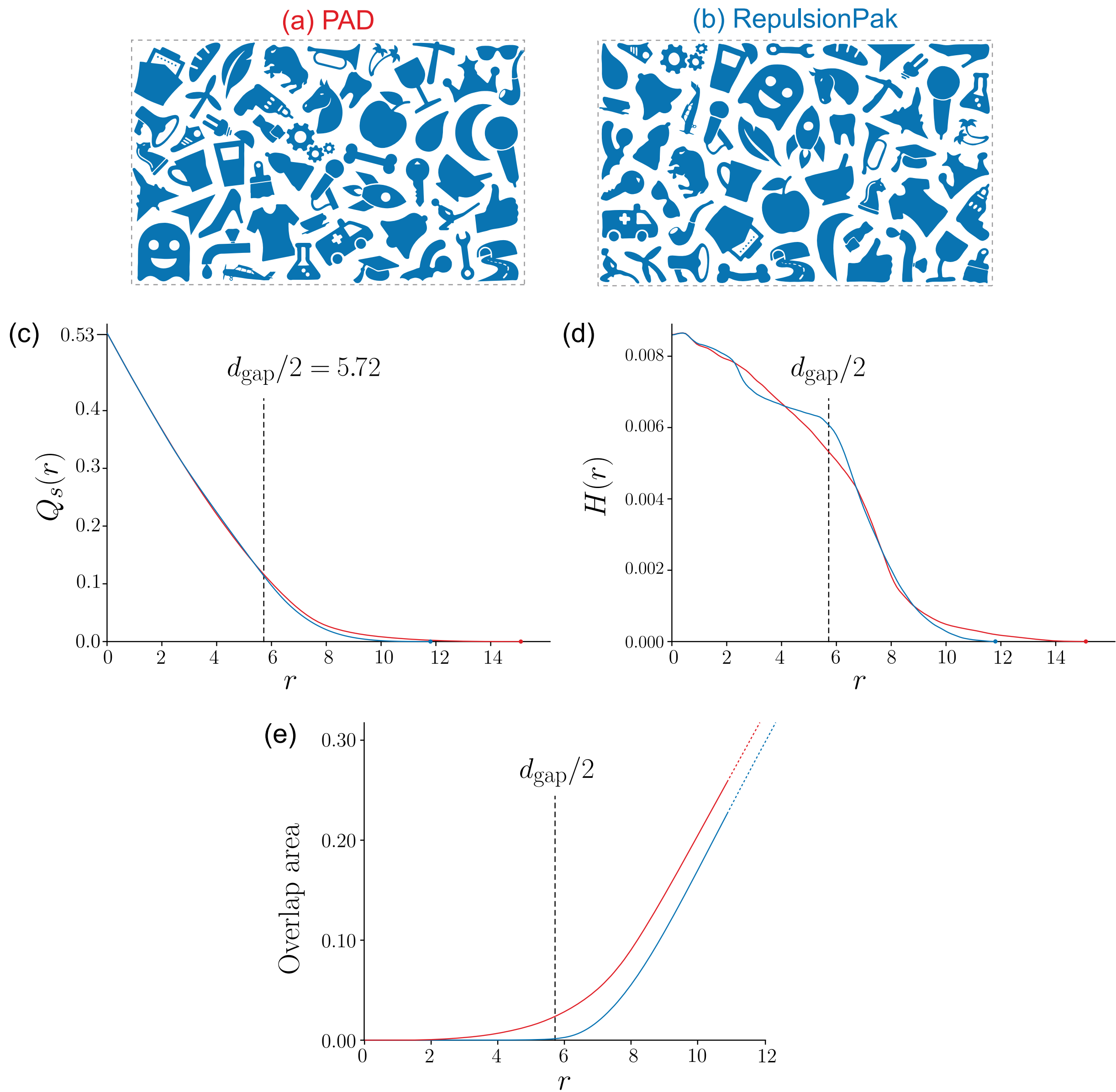


Figure 6.5: A comparison between a PAD packing shown in (a) and a RepulsionPak packing in (b) with their corresponding SCPs (c), distance histograms (d), and overlap functions (e). The PAD and RepulsionPak packings are calibrated to have the same negative space ratio. We visualize the statistics of the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

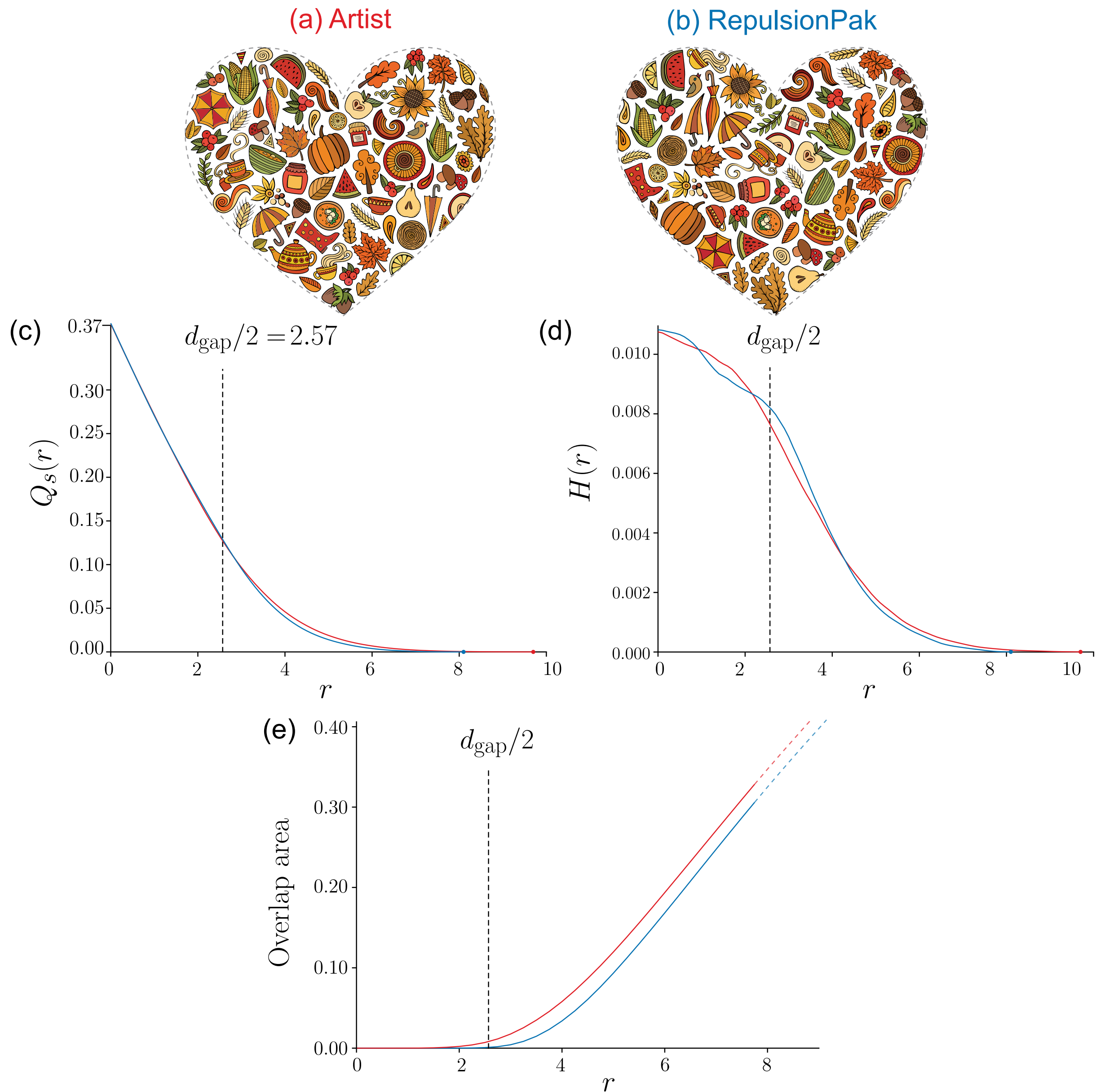


Figure 6.6: A comparison between the artist-made packing (Artist: Balabolka on Shutterstock) shown in (a), and a RepulsionPak packing with the same elements in (b). We plot the corresponding SCPs (c), distance histograms (d), and overlap functions (e). For comparison purposes we remove secondary elements 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.

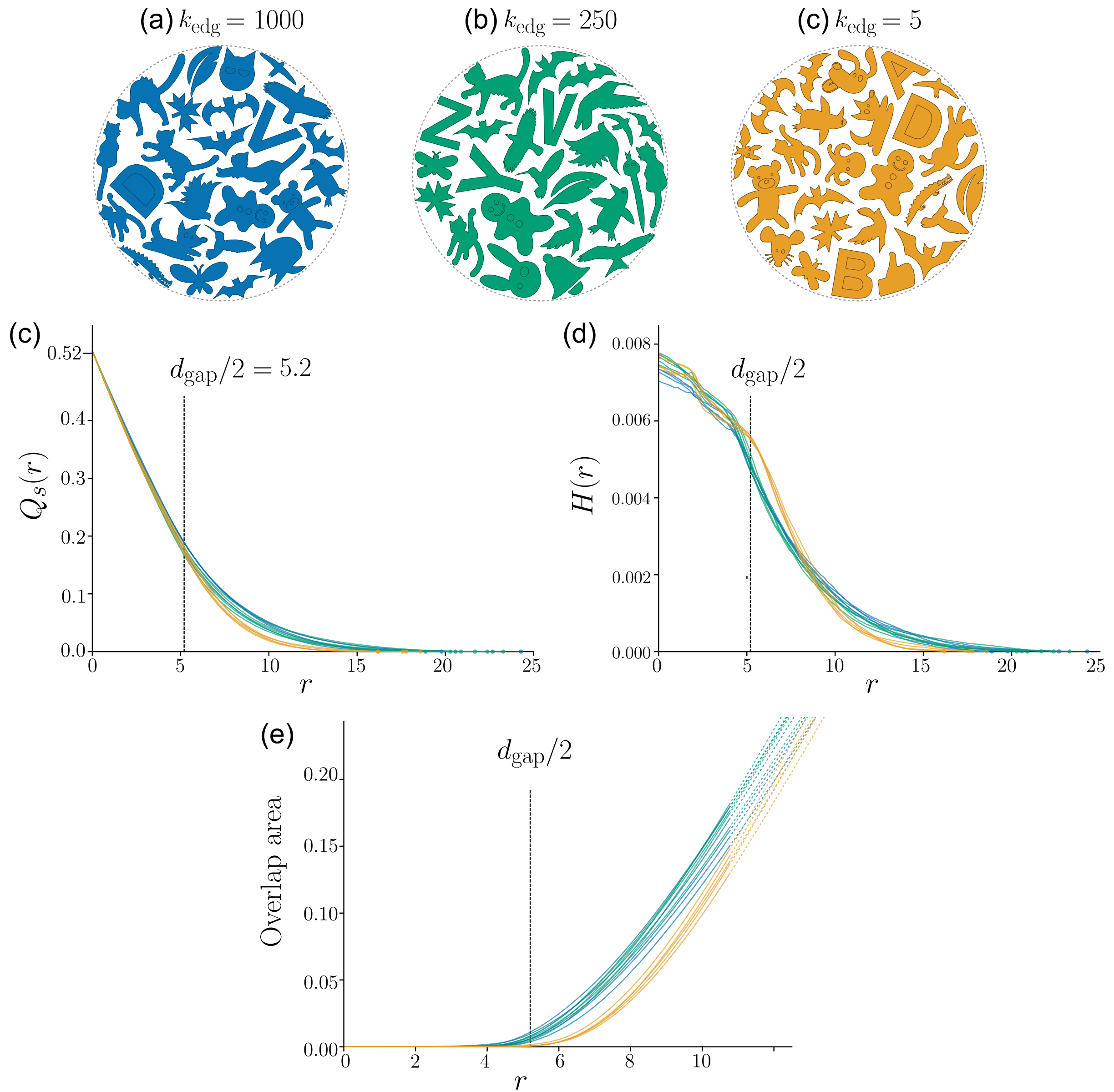


Figure 6.7: A demonstration of the effect of deformation on the evenness of negative space. The packings in (a), (b) and (c) are representative results using three values of the edge force strength k_{edg}, from rigid ($k_{\text{edg}} = 1000$) to moderate ($k_{\text{edg}} = 250$) to deformable ($k_{\text{edg}} = 5$). 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 $d_{\text{gap}}/2$, and lower overlap functions.

6.5 Conclusions and Future Work

We believe that 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.

There are a few ideas that we would like to explore to improve packing evaluation:

- Due to its random initial placement, the evenness of negative space of RepulsionPak results can vary slightly, as shown in Figure 6.7. For each RepulsionPak result in Figure 6.5 and 6.6, we run the simulation a few times and choose one that we like. Although we believe that RepulsionPak generates consistent results, we would like to improve the comparisons by aggregating data from multiple simulations.

- 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.

- We would like to investigate better visualizations for the SCP. When comparing calibrated packings, SCPs communicate differences in the evenness of negative space, but the differences between SCPs can be subtle. A better visualization might amplify these differences to make evaluation easier.

- Our metrics are all based on Euclidean distance which is equivalent to Minkowski sums with discs. With this model of distance, offsets of sharp corners are rounded off. Yet, when visually evaluating the evenness of negative space, we might implicitly visualize mitered corners as being even, as in the square grid of Figure 6.2b. It would be worthwhile to investigate statistical measures based on pseudodistance metrics that treat mitering as perceptually even.

- In a sense, the local maxima of the distance transform are the points that yield the most information about the distribution of negative space: any non-maximal point is subordinate to some nearby maximum. During this work, we investigated whether we could limit the computation of the distance transform to its local maxima. The

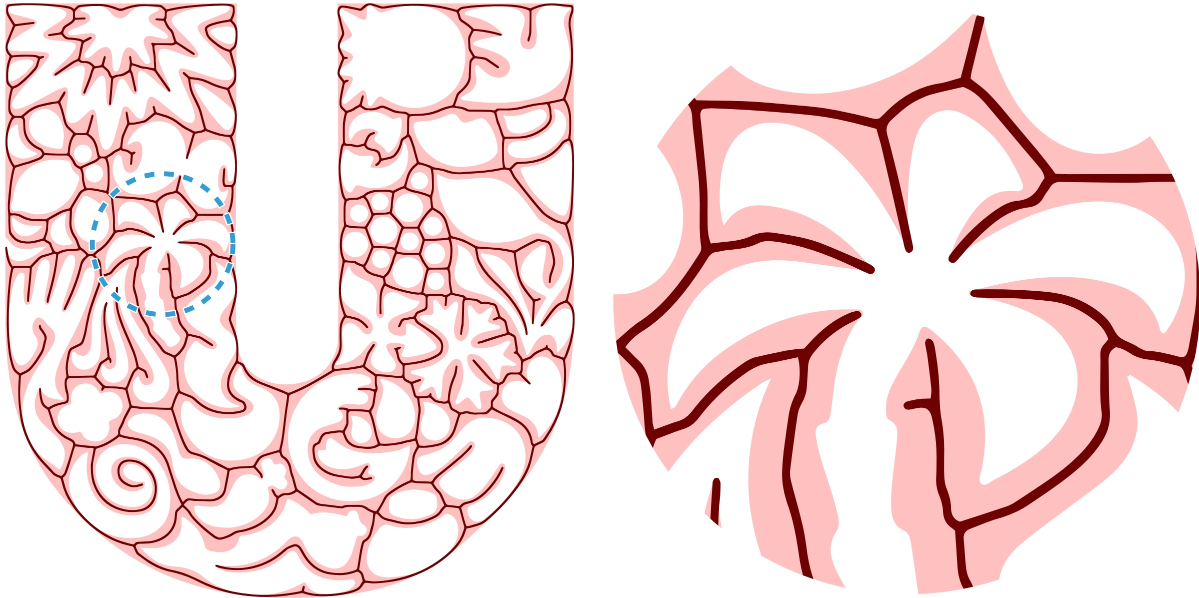


Figure 6.8: A medial axis generated from the negative space of the Unilever packing using the Zhang-Suen thinning algorithm [ZS84]. The medial axis structure can be used to identify local maxima of the distance transform. However, branches can touch or nearly touch sharp concave corners, and do not correspond to human judgment of negative space.

obvious choice of the medial axis is error-prone, and produces branches in corners that can misrepresent the perception of negative space (Figure 6.8). The area Voronoi diagram of the elements is promising, but ignores channels bounded by distant parts of the same element. More work is needed to identify a geometric skeleton that can serve as a scaffolding for evaluating the distance transform.

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 an extruded 3D shape in a spacetime domain.

We discussed the evenness of negative space as 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.

Given a small 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.

Informally, the packings our methods have generated received a lot of positive feedback. In particular, FLOWPAK and RepulsionPak are sufficiently stable and automatic that they could be used in a general graphic design context to create simple packings. Of course, this level of automation will never fully capture the semantic intent of a design, for example, the placement of an animal form's eye. A real-world tool will have to integrate our simulations into the larger context of manual drawing and editing. Lastly, the animations produced by AnimationPak are a promising first step, but will require further refinement before they can be deployed in motion graphics.

7.2 Future Work

This thesis is only the tip of the iceberg of packing research and we see many possibilities for future work.

7.2.1 Spacious Element Arrangements

We would like to create element arrangements with varying and wider negative space gaps. Evenness of negative space is only one of many design considerations to create aesthetically pleasing element packings. As discussed in Chapter 1, variations in the balance of positive and negative space can be used to emphasize focal points. It would be interesting to incorporate a density field, similar to stippling artwork by Secord [Sec02]. In a low density area, elements are smaller and negative space is wider. Conversely, a high density area can be represented by bigger elements and narrower negative space.

Another potential extension to our deformation-driven methods is synthesizing discrete textures, where element distributions are more important than element interlocking. Because of the generous negative space, we could run RepulsionPak without the growth process. However, the main challenge is to modify our simulation to incorporate inter-element spatial relationships extracted from an exemplar. Additionally, we can adapt AnimationPak to generate animated discrete textures, in a similar style to work by Ma et al. [MWLT13]. Lastly, generating a pleasing arrangement from a fully automated method is difficult, so it also would be interesting to incorporate an interactive tool by Reinert et al. [RRS13] that can infer an artist's intention while creating an artistic arrangement.

7.2.2 Element Construction

We would like to explore techniques to assist users in constructing element shapes. A simple example can be seen in a FLOWPAK packing, where we allow the user to construct additional motifs automatically by tracing the shapes of pockets of negative space (Figure 3.15). Another approach may be to identify salient shapes in a source photograph to construct fixed elements. We first perform an image segmentation algorithm [CP02, ASS+12] and the artist can select regions they think are important. Each region is then vectorized to create a new fixed element.

Earlier data-driven packing techniques achieve variety by assembling a large library of element shapes. In FLOWPAK and RepulsionPak, we used deformation to amplify

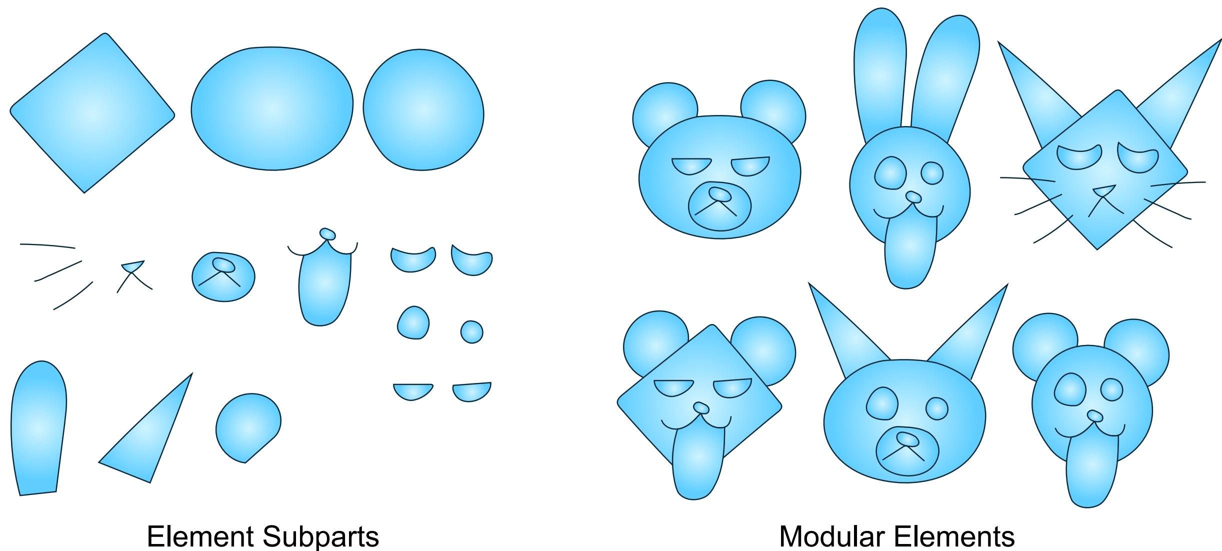


Figure 7.1: Examples of modular elements assembled from smaller parts.

the variety of a much smaller element library. Other approaches may allow us to generate variety in different ways. For example, we could construct *modular* elements by assembling them from smaller subparts (Figure 7.1). Past research has explored example-based and part-based assembly of modular objects [BA06, RHDG10, KCKK12, HL12], but more work is needed to make these tools practical and useful in the context of packings. Modularity is itself a kind of discrete mode of deformation, and we could mix-and-match parts during packing to improve packing quality. When fitting elements to the container boundary (Section 4.7) we could begin by choosing subparts that optimize the fit, and then build elements that contain those subparts. Of course, compatibility could be further improved by deforming modular elements using the techniques in this thesis.

In the context of FLOWPAK, we are interested in exploring a simpler form of modularity, in which short subparts are assembled into larger shapes by threading them along curves, as illustrated in Figure 7.2a. This approach to modularity might adapt itself particularly well to floral ornamentation. In that case, it would be natural to explore extensions to FLOWPAK that support branching structures instead of isolated curves, as in Figure 7.2b. A challenge here is to identify pairs of subparts, and points on them, that support high-quality attachments. This quality can be measured geometrically, possibly aided by new algorithms for content-aware blending of vector shapes. It also has a semantic component; for example, a leaf shape can be attached to a stem, but not directly to a twig. Past research on ornament synthesis, such as DecoBrush [LBW+14] may be relevant here.

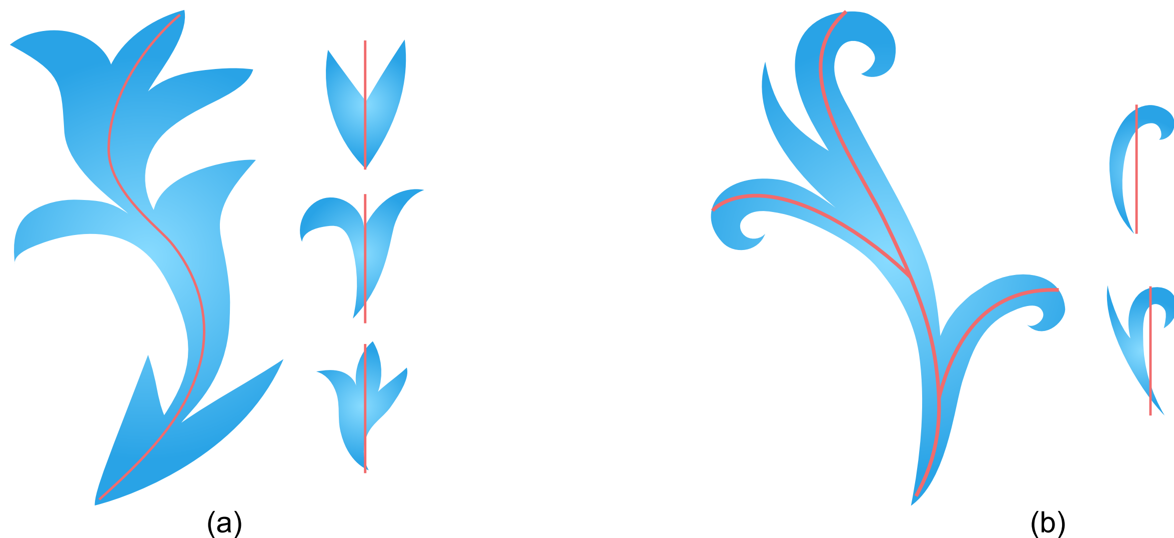


Figure 7.2: Plant-style modular elements. (a) A larger element is constructed by threading multiple subparts along a curve. (b) Multiple streamlines are joined in a branching structure.

7.2.3 Beyond 2D

Inspired by the 3D shaded skull packing in Figure 4.14, we would like to explore the use of deformation-driven methods to create packings on surfaces. To realize this idea, it should be possible to adapt RepulsionPak by computing geodesic distances to generate physical forces. Packings on surfaces are also useful in a fabrication context, for example, creating printed objects in the style of previous work like that of Chen et al. [CML+17]. We could exploit boundary compatibilities to discover ideal locations to connect neighbouring elements. Alternatively, it would be interesting to 3D print the negative space, which is already connected due to the absence of overlaps. The resulting packing would be more like a filigree [CZX+16], with thin wires bounding holes in the shapes of the elements.

Many of the techniques in this thesis could be adapted to develop a deformation-driven method for packing purely spatial 3D objects in 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. One potential issue is that elements in the middle of the container may be fully occluded by their neighbours, which would render them useless as they cannot be viewed. This problem will arise especially in a dense 3D packing of small elements.

It may also be possible to construct interesting packings of 3D objects constrained to

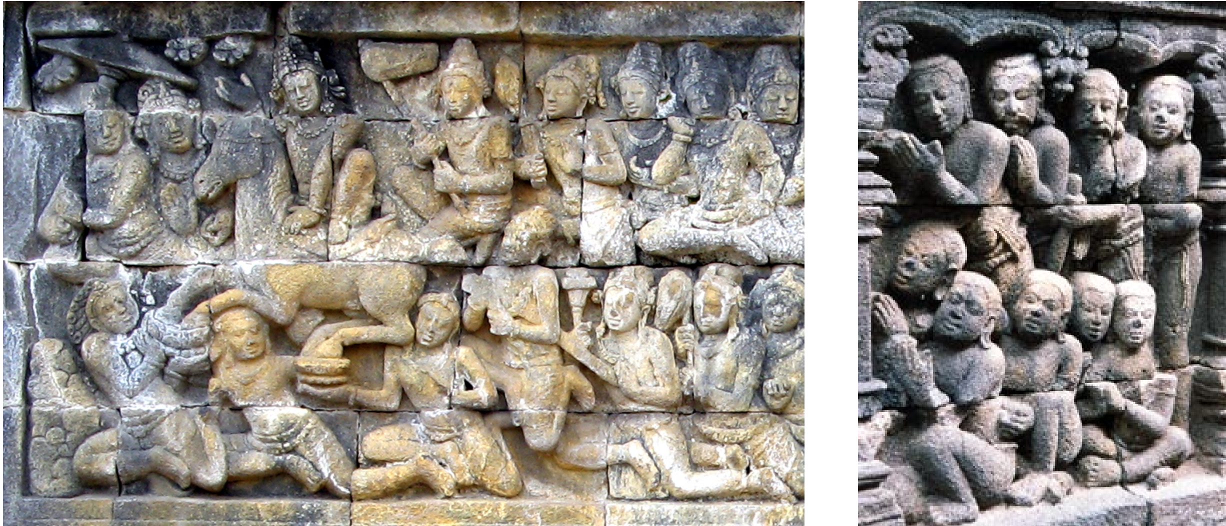


Figure 7.3: Bas-reliefs from Borobudur temple. These bas-reliefs give us an idea to generate a packing of 3D objects on a surface. Individual element can still rotate in 3D to align itself with its neighbours. The photographer of the left image was Michael Gunther, and the right image was taken by Michel Estermann.

lie near a surface. Figure 7.3 shows two stone bas-reliefs from the Borobudur temple on the island of Java. Each relief consists of a tight, albeit non-interlocking, arrangement of human shapes. We are interested in constructing similar arrangements using deformation-driven packing, with elements that are required to lie on a shared surface. The interactions between elements during the packing process may cause them to occlude each other but without interpenetration. We are curious to know how occlusions affect aesthetics in the context of bas-reliefs like these.

7.2.4 Collaborations with Artists

Collaborations with artists can provide us invaluable feedback for our deformation-driven methods. Two artists have used RepulsionPak to generate packings in Figure 4.14, although we did not collect formal feedback or data on their experiences. After RepulsionPak was demonstrated at a large public event, we also received expressions of interest and enthusiasm from online graphic design communities.

We would like to develop an interactive user interface for our packing methods. Examples of recent work in this style include research by Zehnder et al. [ZCT16], Gieseke et

Figure 7.4: A young artist, Joe Whale, is drawing a doodle mural. The photograph is taken from instagram.com/thedoodleboy.co.uk.

al. [GALF17], and Hsu et al. [HWYZ20], which let the user directly place and manipulate elements while a composition is being created. Inspired by doodle murals by Joe Whale (Figure 7.4), it would be interesting to develop an interactive installation in which users could interact with a live packing on a large touch-screen display.

It would also be interesting to create a system to assist artists in creating arrangements of physical objects. One approach could make use of a camera-equipped augmented reality headset. We use the camera to scan elements, which are then processed by the system. The headset displays a suggestion of an element arrangement which the artist can follow. As the artist moves the elements around, the system rescans the current element placement and recomputes a new suggestion.

In the context of animated packings, we were only able to find a single example from The Simpsons (Figure 5.1). We believe these designs are rare because they have been so difficult to create using conventional software. We would like to engage with artists to understand the aesthetic value and limitations of AnimationPak.

7.2.5 Packing Evaluation

We would like to develop other quantitative metrics to evaluate how well an ornamental design fulfills other design principles. In Chapter 3, we argue that visual flow and "uniformity amidst variety" are important in attractive packings. In another study, Wong et al. [WZS98] describe basic design principles for decorative arts: repetition, balance, and conformation to geometric constraints.

Currently, we measure only the evenness of negative space. A measure of element deformation in a composition would permit a comparison against future deformation-driven techniques. In RepulsionPak, we can measure a packing's potential energy at equilibrium. However, there are some questions to answer. Can we measure how deformations are perceived? How much they affect the aesthetics of packings? Such measurements could help us develop better deformation algorithms in general, not just in the context of packings.

All three metrics described in Chapter 6 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.

More broadly, although our algorithms and measurements are derived from careful study of packings created by artists, there is no guarantee that a statistic like the evenness of negative space is a suitable proxy for human aesthetic judgment. Future work should seek to articulate mathematical criteria that correlate with qualitative packing quality, using perceptual studies on human subjects.

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 2014, pages 5–10, 2014. ACM, doi:10.1145/2630407.2630409.

[AKA13a] Zainab AlMeraj, Craig S. Kaplan, and Paul Asente. Patch-Based Geometric Texture Synthesis. In *Proceedings of the Symposium on Computational Aesthetics*, CAE 2013, pages 15–19, 2013. ACM, doi:10.1145/2487276.2487278.

[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 2013, page 514, 2013. ACM, doi:10.1145/2486042.2486043.

[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*, 2010. The Eurographics Association, doi:10.2312/SBM/SBM10/033-040.

[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. ISPA 2013, pages 257–261, 2013, doi:10.1109/ispa.2013.6703749.

[ASS+12] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. SLIC Superpixels Compared to State-of-the-Art Superpixel Methods. *IEEE TPAMI*, 34(11):2274–2282, 2012, doi:10.1109/tpami.2012.120.

[BA06] William Baxter and Ken-ichi Anjyo. Latent Doodle Space. *Computer Graphics Forum*, 2006, doi:10.1111/j.1467-8659.2006.00967.x.

[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, doi:10.1111/j.1467-8659.2006.00986.x.

[BEB12] Tyson Brochu, Essex Edwards, and Robert Bridson. Efficient Geometrically Exact Continuous Collision Detection. *ACM Transactions on Graphics*, 31(4), July 2012, doi:10.1145/2185520.2185592.

[BL15] Thomas F. Banchoff and Stephen T. Lovett. *Differential Geometry of Curves and Surfaces*. Chapman and Hall/CRC, 2015, doi:10.1201/b18913.

[BML+14] Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Transactions on Graphics*, 33(4):154:1–154:11, July 2014, doi:10.1145/2601097.2601116.

[Bri07] Robert Bridson. Fast Poisson Disk Sampling in Arbitrary Dimensions. SIGGRAPH 2007 Sketches, 2007. ACM, doi:10.1145/1278780.1278807.

[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, doi:10.1016/j.powtec.2009.07.013.

[BvMM11] B. Beneš, O. Št'ava, R. Měch, and G. Miller. Guided Procedural Modeling. *Computer Graphics Forum*, 30(2):325–334, 2011, doi:10.1111/j.1467-8659.2011.01886.x.

[BW95] Armin Bruderlin and Lance Williams. Motion Signal Processing. SIGGRAPH 1995, pages 97–104, 1995. ACM, doi:10.1145/218380.218421.

[BWL18] Xiaojun Bian, Li-Yi Wei, and Sylvain Lefebvre. Tile-Based Pattern Design with Topology Control. *ACM on Computer Graphics and Interactive Techniques*, 1(1), July 2018, doi:10.1145/3203204.

[CFH+09] M. Cui, J. Femiani, J. Hu, P. Wonka, and A. Razdan. Curve Matching for Open 2D Curves. *Pattern Recognition Letters*, 30(1):1–10, January 2009, doi:10.1016/j.patrec.2008.08.013.

[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, doi:10.1111/j.1467-8659.2011.01889.x.

[CML+17] Weikai Chen, Yuexin Ma, Sylvain Lefebvre, Shiqing Xin, Jonàs Martínez, and Wenping Wang. Fabricable Tile Decors. *ACM Transactions on Graphics*, 36(6):175:1–175:15, November 2017, doi:10.1145/3130800.3130817.

[Coh92] Michael F. Cohen. Interactive Spacetime Control for Animation. SIGGRAPH 1992, pages 293–302, 1992. ACM, doi:10.1145/133994.134083.

[CP02] Dorin. Comaniciu and Meer. Peter. Mean Shift: A Robust Approach Toward Feature Space Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25:281–288, 01 2002, doi:10.1109/34.1000236.

[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, doi:10.1002/9781118658222.

[CSR10] N. Chernov, Yu. Stoyan, and T. Romanova. Mathematical Model and Efficient Algorithms for Object Packing Problem. *Computational Geometry*, 43(5):535 – 553, 2010, doi:https://doi.org/10.1016/j.comgeo.2009.12.003.

[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, doi:10.1016/s0010-4485(01)00109-9.

[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 Transactions on Graphics*, 34(6), October 2015, doi:10.1145/2816795.2818087.

[CZX+16] Weikai Chen, Xiaolong Zhang, Shiqing Xin, Yang Xia, Sylvain Lefebvre, and Wenping Wang. Synthesis of Filigrees for Digital Fabrication. *ACM Transactions on Graphics*, 35(4):98:1–98:13, July 2016, doi:10.1145/2897824.2925911.

[DACM19] Lars Doyle, Forest Anderson, Ehren Choy, and David Mould. Automated Pebble Mosaic Stylization of Images. *Computational Visual Media*, 5(1):33–44, March 2019, doi:10.1007/s41095-019-0129-0.

[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 2006, pages 71–78, 2006. ACM, doi:10.1145/1124728.1124741.

[DRvdP15] Boris Dalstein, Rémi Ronfard, and Michiel van de Panne. Vector Graphics Animation with Time-Varying Topology. *ACM Transactions on Graphics*, 34(4), July 2015, doi:10.1145/2766913.

[Eri05] Christer Ericson. Chapter 5: Basic Primitive Tests. In *Real-Time Collision Detection*, The Morgan Kaufmann Series in Interactive 3D Technology, pages 125–233. Morgan Kaufmann, 2005, doi:https://doi.org/10.1016/B978-1-55860-732-3.50010-3.

[Gal03] Philip Galanter. What Is Generative Art? Complexity Theory as a Context for Art Theory, 01 2003. Available online at https://www.philipgalanter.com/downloads/ga2003_paper.pdf.

[GALF17] Lena Gieseke, Paul Asente, Jingwan Lu, and Martin Fuchs. Organized Order in Ornamentation. In *Proceedings of the Symposium on Computational Aesthetics*, CAE 2017, pages 4:1–4:9, 2017. ACM, doi:10.1145/3092912.3092913.

[GBLM16] Paul Guerrero, Gilbert Bernstein, Wilmot Li, and Niloy J. Mitra. PATEX: Exploring Pattern Variations. *ACM Transactions on Graphics*, 35(4):48:1–48:13, July 2016, doi:10.1145/2897824.2925950.

[Gle01] Michael Gleicher. Motion Path Editing. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D 01, pages 195–202, 2001. ACM, doi:10.1145/364338.364400.

[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, doi:10.1109/TVCG.2007.45.

[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 2007, pages 7–14, 2007. ACM, doi:10.1145/1274871.1274873.

[Hau01] Alejo Hausner. Simulating Decorative Mosaics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 2001, pages 573–580, 2001. ACM, doi:10.1145/383259.383327.

[HHD03] Stefan Hiller, Heino Hellwig, and Oliver Deussen. Beyond Stippling - Methods for Distributing Objects on the Plane. *Computer Graphics Forum*, 2003, doi:10.1111/1467-8659.00699.

[HKT10] Edmond S. L. Ho, Taku Komura, and Chiew-Lan Tai. Spatial Relationship Preserving Character Motion Adaptation. *ACM Transactions on Graphics*, 29(4), July 2010, doi:10.1145/1778765.1778770.

[HL12] Thomas Hurtut and Pierre-Edouard Landes. Synthesizing Structured Doodle Hybrids. SIGGRAPH Asia 2012 Posters, 2012. ACM, doi:10.1145/2407156.2407204.

[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 2009, pages 51–60, 2009. ACM, doi:10.1145/1572614.1572623.

[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 1993, pages 197–206, 1993. ACM, doi:10.1145/168642.168662.

[Hof86] Douglas R. Hofstadter. *Metamagical Themas: Questing for the Essence of Mind and Pattern*. Penguin Science. Penguin Books, 1986.

[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, doi:10.1109/tvcg.2014.2303087.

[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, 2020. ACM, doi:10.1145/3313831.3376248.

[HZZ11] Hua Huang, Lei Zhang, and Hong-Chao Zhang. Arcimboldo-Like Collage Using Internet Images. SIGGRAPH Asia 2011, pages 155:1–155:8, 2011. ACM, doi:10.1145/2024156.2024189.

[IMH05] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. As-Rigid-As-Possible Shape Manipulation. SIGGRAPH 2005, pages 1134–1141, 2005. ACM, doi:10.1145/1186822.1073323.

[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, doi:10.1111/j.1467-8659.2008.01140.x.

[Ise13] Tobias Isenberg. Evaluating and Validating Non-Photorealistic and Illustrative Rendering. In *Image and Video based Artistic Stylisation*, volume 42 of *Computational Imaging and Vision*, chapter 15, pages 311–331. Springer, 2013, doi:10.1007/978-1-4471-4519-6_15.

[Jam20] Doug L. James. Phong Deformation: A Better C^0 Interpolant for Embedded Deformation. *ACM Trans. Graph.*, 39(4), July 2020, doi:10.1145/3386569.3392371.

[JBPS11] Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. Bounded Biharmonic Weights for Real-Time Deformation. *ACM Transactions on Graphics*, 30(4):78:1–78:8, July 2011, doi:10.1145/2010324.1964973.

[JDM19] Ali Sattari Javid, Lars Doyle, and David Mould. Irregular Pebble Mosaics with Sub-Pebble Detail. In *ACM/EG Expressive Symposium*, 2019. The Eurographics Association, doi:10.2312/exp.20191084.

[JL97] Bruno Jobard and Wilfrid Lefer. Creating Evenly-Spaced Streamlines of Arbitrary Density. In *Visualization in Scientific Computing 1997: Proceedings of the Eurographics Workshop in Boulogne-sur-Mer France, April 28–30, 1997*, pages 43–55, 1997. Springer Vienna, doi:10.1007/978-3-7091-6876-9_5.

[JMD+12] Neel Joshi, Sisil Mehta, Steven Drucker, Eric Stollnitz, Hugues Hoppe, Matt Uyttendaele, and Michael Cohen. Cliplets: Juxtaposing Still and Dynamic

Imagery. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST 12, pages 251–260, 2012. ACM, doi:10.1145/2380116.2380149.

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

[Kap10] Craig S. Kaplan. Curve Evolution Schemes for Parquet Deformations. In *Proceedings of Bridges 2010: Mathematics, Music, Art, Architecture, Culture*, pages 95–102, 2010. Tessellations Publishing. Available online at http://bridgesmathart.org/2010/cdrom/proceedings/55/paper_55.pdf.

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

[KCG+14] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. Draco: Bringing Life to Illustrations with Kinetic Textures. CHI 2014, page 351360, 2014. ACM, doi:10.1145/2556288.2556987.

[KCKK12] Evangelos Kalogerakis, Siddhartha Chaudhuri, Daphne Koller, and Vladlen Koltun. A Probabilistic Model of Component-Based Shape Synthesis. *ACM Transactions on Graphics*, 31(4), 2012, doi:10.1145/2185520.2185551.

[KHHL12] Manmyung Kim, Youngseok Hwang, Kyunglyul Hyun, and Jehee Lee. Tiling Motion Patches. SCA 2012, pages 117–126, 2012. Eurographics Association, doi:10.1109/tvcg.2013.80.

[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 2011, pages 157–166, 2011. ACM, doi:10.1145/2024676.2024701.

[KP02] Junhwan Kim and Fabio Pellacini. Jigsaw Image Mosaics. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 2002, pages 657–664, 2002. ACM, doi:10.1145/566570.566633.

[KS00] Craig S. Kaplan and David H. Salesin. Escherization. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 2000, pages 499–510, 2000, doi:10.1145/344779.345022.

[KS04] Craig S. Kaplan and David H. Salesin. Dihedral Escherization. In *Proceedings of Graphics Interface 2004*, GI 2004, pages 255–262, 2004. Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine. Available online at https://graphicsinterface.org/proceedings/gi2004/gi2004-31/.

[KS11] Hiroshi Koizumi and Kokichi Sugihara. Maximum Eigenvalue Problem for Escherization. *Graphs and Combinatorics*, 27(3):431, Mar 2011, doi:10.1007/s00373-011-1022-5.

[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 Transactions on Graphics*, 35(6):229:1–229:12, November 2016, doi:10.1145/2980179.2980234.

[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, 2019. Canadian Information Processing Society, doi:10.20380/GI2019.13.

[LBW+14] Jingwan Lu, Connelly Barnes, Connie Wan, Paul Asente, Radomír Měch, and Adam Finkelstein. DecoBrush: Drawing Structured Decorative Patterns by Example. *ACM Transactions on Graphics*, 33(4):90:1–90:9, July 2014, doi:10.1145/2601097.2601190.

[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):231–243, February 2011, doi:10.1109/TVCG.2010.36.

[LHVT17] Hugo Loi, Thomas Hurtut, Romain Vergne, and Joelle Thollot. Programmable 2D Arrangements for Element Texture Design. *ACM Transactions on Graphics*, 36(4), May 2017, doi:10.1145/3072959.2983617.

[LJG14] Songrun Liu, Alec Jacobson, and Yotam Gingold. Skinning Cubic Bézier Splines and Catmull-Clark Subdivision Surfaces. *ACM Transactions on*

Graphics, 33(6):190:1–190:9, November 2014, doi:10.1145/2661229.2661270.

[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, doi:10.1109/tvcg.2017.2660488.

[LS11] Noah Lockwood and Karan Singh. Biomechanically-Inspired Motion Path Editing. SCA 2011, page 267276, 2011. Association for Computing Machinery, doi:10.1145/2019406.2019442.

[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 2009, pages 1085–1096, 2009. Springer-Verlag, doi:10.1007/978-3-642-10520-3_104.

[MBS+11] Ron Maharik, Mikhail Bessmeltsev, Alla Sheffer, Ariel Shamir, and Nathan Carr. Digital Micrography. SIGGRAPH 2011, pages 100:1–100:12, 2011. ACM, doi:10.1145/1964921.1964995.

[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, doi:10.1111/cgf.13490.

[MCKM15] Matthias Müller, Nuttapong Chentanez, Tae-Yong Kim, and Miles Macklin. Air Meshes for Robust Collision Handling. *ACM Transactions on Graphics*, 34(4):133:1–133:9, July 2015, doi:10.1145/2766907.

[MF92] Michael McCool and Eugene Fiume. Hierarchical Poisson Disk Sampling Distributions. In *Proceedings of the Conference on Graphics Interface 1992*, pages 94–105, 1992. Morgan Kaufmann Publishers Inc., doi:10.20380/GI1992.12.

[MHHR07] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position Based Dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, April 2007, doi:10.1016/j.jvcir.2007.01.005.

[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, doi:10.1109/MCG.2008.35.

[MSS+19] Jonàs Martínez, Mélina Skouras, Christian Schumacher, Samuel Hornus, Sylvain Lefebvre, and Bernhard Thomaszewski. Star-Shaped Metrics for Mechanical Metamaterial Design. *ACM Transactions on Graphics*, 38(4), July 2019, doi:10.1145/3306346.3322989.

[MWLT13] Chongyang Ma, Li-Yi Wei, Sylvain Lefebvre, and Xin Tong. Dynamic Element Textures. *ACM Transactions on Graphics*, 32(4), July 2013, doi:10.1145/2461912.2461921.

[MWT11] Chongyang Ma, Li-Yi Wei, and Xin Tong. Discrete Element Textures. SIGGRAPH 2011, pages 62:1–62:10, 2011. ACM, doi:10.1145/1964921.1964957.

[NI20] Yuichi Nagata and Shinji Imahori. An Efficient Exhaustive Search Algorithm for the Escherization Problem. *Algorithmica*, 82(9):2502–2534, Sep 2020, doi:10.1007/s00453-020-00695-6.

[OS88] Stanley Osher and James A. Sethian. Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *Journal of Computational Physics*, 79(1):12–49, November 1988, doi:10.1016/0021-9991(88)90002-2.

[Osh17] Masaki Oshita. Lattice-Guided Human Motion Deformation for Collision Avoidance. In *Proceedings of the Tenth International Conference on Motion in Games*, MIG 17, 2017. ACM, doi:10.1145/3136457.3136475.

[PS06] Hans Pedersen and Karan Singh. Organic Labyrinths and Mazes. In *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR 2006, pages 79–86, 2006. ACM, doi:10.1145/1124728.1124742.

[PYW14] Chi-Han Peng, Yong-Liang Yang, and Peter Wonka. Computing Layouts with Deformable Templates. *ACM Transactions on Graphics*, 33(4):99:1–99:11, July 2014, doi:10.1145/2601097.2601164.

[PZ07] Jonathan Palacios and Eugene Zhang. Rotational Symmetry Field Design on Surfaces. SIGGRAPH 2007, 2007. ACM, doi:10.1145/1275808.1276446.

[RHDG10] Eric Risser, Charles Han, Rozenn Dahyot, and Eitan Grinspun. Synthesizing Structured Image Hybrids. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)*, 29(4):85:1–85:6, 2010, doi:10.1145/1833349.1778822.

[RRS13] Bernhard Reinert, Tobias Ritschel, and Hans-Peter Seidel. Interactive By-Example Design of Artistic Packing Layouts. *ACM Transactions on Graphics*, 32(6):218:1–218:7, November 2013, doi:10.1145/2508363.2508409.

[Sch90] Philip J. Schneider. An Algorithm for Automatically Fitting Digitized Curves. In *Graphics Gems*, pages 612–626. Academic Press Professional, Inc., 1990.

[Sec02] Adrian Secord. Weighted Voronoi Stippling. In *Proceedings of the 2nd International Symposium on Non-Photorealistic Animation and Rendering*, NPAR 2002, pages 37–43, 2002. ACM, doi:10.1145/508530.508537.

[SHWP09] Alexander Schiftner, Mathias Höbinger, Johannes Wallner, and Helmut Pottmann. Packing Circles and Spheres on Surfaces. *ACM Transactions on Graphics*, 28(5):1–8, December 2009, doi:10.1145/1618452.1618485.

[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 2018, 2018. Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine, doi:10.20380/GI2018.03.

[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, doi:10.1109/TVCG.2019.2950235.

[SKA20] Reza Adhitya Saputra, Craig S. Kaplan, and Paul Asente. AnimationPak: Packing Elements with Scripted Animations. In *Proceedings of Graphics Interface 2020*, GI 2020, pages 393 – 403, 2020. Canadian Human-Computer Communications Society / Societe canadienne du dialogue humain-machine, doi:10.20380/GI2020.39.

[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 2017, pages 8–15, 2017. Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine, doi:10.20380/GI2017.02.

[SLK05] Kaleigh Smith, Yunjun Liu, and Allison Klein. Animosaics. SCA 2005, pages 201–208, 2005. ACM, doi:10.1145/1073368.1073397.

[SSSE00] Arno Schödl, Richard Szeliski, David H. Salesin, and Irfan Essa. Video Textures. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 2000, pages 489–498, 2000. ACM, doi:10.1145/344779.345012.

[Sug09] Kokichi Sugihara. Computer-Aided Generation of Escher-like Sky and Water Tiling Patterns. *Journal of Mathematics and the Arts*, 3(4):195–207, 2009, doi:10.1080/17513470903185626.

[TRdAS07] Christian Theobalt, Christian Rössl, Edilson de Aguiar, and Hans-Peter Seidel. Animation Collage. SCA 2007, pages 271–280, 2007. Eurographics Association, doi:10.2312/SCA/SCA07/271-280.

[VGB+14] J. Vanek, J. A. Garcia Galicia, B. Benes, R. Měch, N. Carr, O. Šť'ava, and G. S. Miller. PackMerger: A 3D Print Volume Optimizer. *Computer Graphics Forum*, 33(6):322–332, September 2014, doi:10.1111/cgf.12353.

[WK88] Andrew Witkin and Michael Kass. Spacetime Constraints. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 1988, pages 159–168, 1988. ACM, doi:10.1145/54852.378507.

[WLKT09] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the Art in Example-based Texture Synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*, pages 93–117, March 2009. Eurographics Association, doi:10.2312/egst.20091063.

[WP95] Andrew Witkin and Zoran Popović. Motion Warping. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 1995, pages 105–108, 1995. ACM, doi:10.1145/218380.218422.

[WZS98] Michael T. Wong, Douglas E. Zongker, and David H. Salesin. Computer-Generated Floral Ornament. SIGGRAPH 1998, pages 423–434, 1998. ACM, doi:10.1145/280814.280948.

[XK07] Jie Xu and Craig S. Kaplan. Calligraphic Packing. In *Proceedings of Graphics Interface 2007*, GI 2007, pages 43–50, 2007. ACM, doi:10.1145/1268517.1268527.

[XM09] Ling Xu and David Mould. Magnetic Curves: Curvature-Controlled Aesthetic Curves Using Magnetic Fields. CAE 2009, pages 1–8, 2009. Eurographics Association, doi:10.2312/COMPAESTH/COMPAESTH09/001-008.

[XM15] Ling Xu and David Mould. Procedural Tree Modeling with Guiding Vectors. *Computer Graphics Forum*, 34(7):47–56, October 2015, doi:10.1111/cgf.12744.

[YSC20] Christopher Yu, Henrik Schumacher, and Keenan Crane. Repulsive Curves, 2020, arXiv:2006.07859.

[ZCR+16] Changqing Zou, Junjie Cao, Warunika Ranaweera, Ibraheem Alhashim, Ping Tan, Alla Sheffer, and Hao Zhang. Legible Compact Calligrams. *ACM Transactions on Graphics*, 35(4):122:1–122:12, July 2016, doi:10.1145/2897824.2925887.

[ZCT16] Jonas Zehnder, Stelian Coros, and Bernhard Thomaszewski. Designing Structurally-Sound Ornamental Curve Networks. *ACM Transactions on Graphics*, 35(4):99:1–99:10, July 2016, doi:10.1145/2897824.2925888.

[ZS84] TY Zhang and Ching Y. Suen. A Fast Parallel Algorithm for Thinning Digital Patterns. *Communications of the ACM*, 27(3):236–239, 1984, doi:10.1145/357994.358023.

APPENDICES

Appendix A

Animation Videos

This appendix consists of animation videos that correspond to several figures in this thesis.

The file names of the videos are:

- fig_3.1_iterative_refinement.mp4
- fig_3.10_lion_iterative_refinement.mp4
- fig_3.10_unicorn_iterative_refinement.mp4
- fig_4.1_animals.mp4
- fig_4.7_rhinoceros.mp4
- fig_4.9_cats.mp4
- fig_4.10_butterflies.mp4
- fig_4.11_giraffes_penguins.mp4
- fig_5.6_simulation_process.mp4
- fig_5.14_aquatic fauna.mp4
- fig_5.15_snake_bird.mp4
- fig_5.16_penguins_to_giraffes.mp4

- fig_5.17_traveling_bird.mp4
- fig_5.18_lions_mane.mp4
- fig_5.19_animals.mp4
- fig_5.20_cavd_vs_spectral_vs_animationpak.mp4
- fig_5.21_time_spring_stiffness_comparison.mp4
- fig_5.22_beam_and_circles.mp4
- fig_6.4_overlap_viz.mp4
- fig_6.6_overlap_viz.mp4

If you accessed this thesis from a source other than the University of Waterloo, you may not have access to these files. You may access them by searching for this thesis at http://uwspace.uwaterloo.ca.