Animating Still Images

Folding Texture Design and Synthesis

TU Delft

Hao Ren

Animating Still Images

Folding Texture Design and Synthesis

By

Hao Ren

in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

at the Delft University of Technology,
to be defended publicly on September 8, 2023 at 9:00.

Student number:	4512480	
Thesis committee:	Prof. Dr. E. Eisemann,	TU Delft, supervisor
	Dr. Ir. Willem-Paul Brinkman,	TU Delft
	Mathijs Molenaar,	TU Delft

Acknowledge

This master thesis project is conducted in the Computer Graphics and Visualization group at the EEMCS faculty of TU Delft.

I would like to express my thanks to my supervisor Prof. Dr. E. Eisemann for his guidance along the whole journey. His confidence in me finishing this thesis has helped me navigate through this journey, allowing me to overcome many difficulties and complete this work. Also, I want to thank Mathijs for the continuous discussion and feedback.

The journey towards the completion of my thesis has been a challenging yet rewarding one. Especially, I want to express my gratitude to Ms. Inge Grootjans for her unwavering support and encouragement. The empathetic understanding of my struggles in my moments of crisis was vital to me. And finally, many thanks to Dr. Peng Gao, my sambo Linna and my parents. Without you, I won't be able to get this far. Thank you!

Abstract

The phenomenon of one element moving and progressively overlaying another is common in nature, such as waves swashing and backwashing, or eyelids moving over eyeballs while blinking. Folding Texture, which was proposed by Thorben, can simulate this texture “folding” visual effect in real-time without changing geometry.

However, to date, no tool has been developed to assist in the design and synthesis of folding textures. Applications of the technique so far are achieved through manual creation of the folding texture, which is a tedious process.

This thesis explores the problem of folding-texture design and synthesis. A novel approach is proposed for animating still images based on the folding texture technique. The approach uses a semi-automatic, user-assisted method that combines texture editing, motion profile specification, and folding texture synthesis into one seamless process, reducing the need for extensive manual work. It enables novice users to utilize the technique with a fair level of prior knowledge of folding texture.

Keywords: Texture Mapping, Texture Synthesis, Texture Dynamic Sampling, Folding Texture

Table of Contents

List of Figures

1 Introduction

Texture mapping is a popular technique to add additional details to a rendered scene without increasing geometric complexity. The technique uses a texture (also called texture map or image texture) to store for example colors or surface orientation and maps it to the surface of an object. The texture is usually static: its visual appearance does not change over time. In rendering, pixels sample colors from the texture at associated fixed texture coordinates.

Many real-world phenomena are attractive in the way their appearances change over time. However, standard texture mapping alone is not adequate to render these dynamic phenomena. Since texture mapping only specifies how color is sampled in one frame, how pixels sample color from texture in a series of consequent frames must be defined additionally. This can be achieved by either changing pixels' associated texture coordinates or using different textures at different frames.

In this thesis, we focus on dynamic phenomena which share the characteristics that one entity moves and progressively occludes another. Typical examples are waves swashing and backwashing, eyelids moving over eyeballs while blinking, and candle flame waving over a dark background.

The folding-texture technique is a texture-mapping based technique proposed by Thorben in his master thesis [1]. As the name suggests, this technique allows textures to be sampled in a way that gives the effect of it folding over itself as visualized in Figure 1. It can be employed to render phenomena where one entity moves and progressively occludes another. The technique uses three types of texture files. These files as a whole are referred to as folding texture and the technique is referred to as the folding-texture technique in this report.

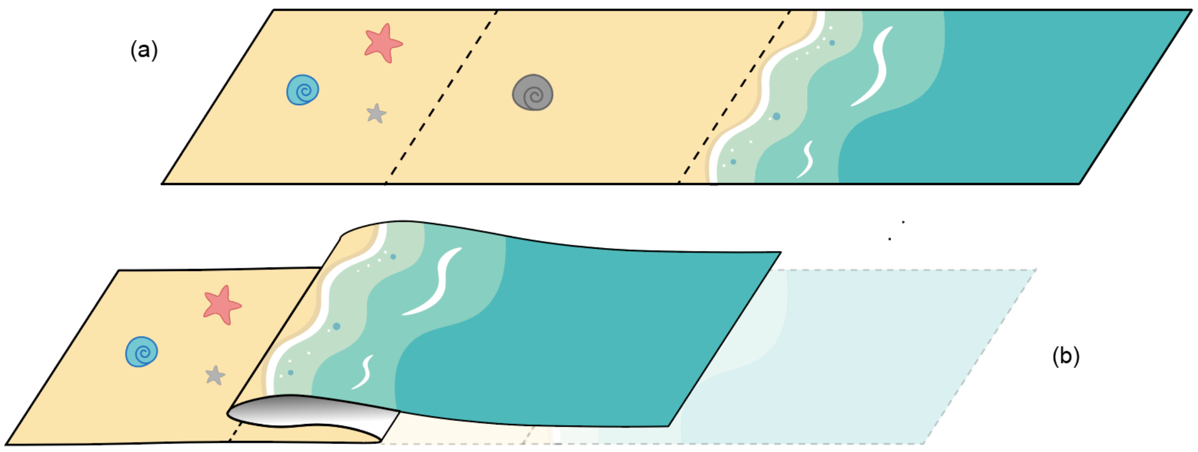


Figure 1. Illustration of the "folding" concept. Texture in (a) is in unfolded status, regions between dotted lines will be folded to create wave motion in (b)

1.1 Motivation

So far, no tool is available for folding texture design and synthesis. Users have to manually calculate and create folding textures with image editing software like Photoshop or GIMP. This primitive way of folding texture creation not only requires users to be familiar with image editing software but also requires users to have in-depth knowledge of the folding texture technique itself. Moreover, a parameterization of each pixel is needed, which is tedious and time-consuming, especially for high-resolution textures.

To pose the problem of folding texture design and synthesis, we propose a semi-automatic, user-assisted approach to facilitate the task. The approach is targeted at folding texture's intrinsic characteristics. It enables users to focus on designing the desired motion patterns without understanding the complex mechanisms underneath.

1.2 Contributions

The thesis work's contributions include,

- a refined definition of the folding texture technique.
- a semi-automatic and user-assisted approach that combines texture editing, motion specification, and folding texture synthesis into one seamless process.
- a ready-to-use software tool as a proof of concept.

1.3 Organization

Following the introduction chapter, the thesis report will be organized as follows. Chapter 2 covers the related techniques and systems. Chapter 3 introduces the folding-texture technique and refines its definition. The design and synthesis approach will be explained in Chapter 4. Chapter 5 covers a few implementation details. Chapter 6 presents a few examples in detail. The conclusion and future work are placed in Chapter 7. The appendix includes a user manual for the implemented software tool.

2 Background and Related Work

The folding-texture technique is based on the texture mapping mechanism. Section 2.1 explains the mechanism of texture mapping in the framework of the computer graphics pipeline. Section 2.2 offers a concise review of the different applications that make use of this mechanism and explains how textures are used in different texture-mapping-based techniques. The folding-texture technique takes still images as input and generates textures with varying appearances. So in Section 2.3, various terminologies related to textures with time-varying appearances are investigated and discussed. Systems that have been employed to animate still images are discussed in Section 2.4.

2.1 Texture Mapping in Computer Graphics Pipeline

As visualized in Figure 2 on next page, the computer graphics pipeline (also referred to as the rendering pipeline [2]) constitutes a series of interconnected stages that facilitate the transformation of digital models or scenes into two-dimensional images or frames. The pipeline can be broadly divided into several stages: vertex processing, primitive processing, rasterization, fragment processing, and screen sample operations.

During the vertex processing stage, input vertex data, including positions, normal vectors, and texture coordinates, are sent to the vertex shader. The vertex shader is responsible for transforming the vertex positions and passing the texture coordinates and other vertex attributes to the next stage. After vertex processing, primitive assembly constructs geometric primitives (triangles, lines, or points) from the processed vertices, and rasterization subsequently converts these primitives into fragments (pixels to be shaded). During rasterization, vertex attributes, including texture coordinates, are interpolated for each fragment. The fragment processing stage uses operations like texturing, lighting, and custom calculations on fragments. Following fragment processing, per-fragment operations, such as depth testing, stencil testing, and blending, are performed to handle overlapping and transparent objects and incorporate additional visual effects.

Several stages are involved in the texture mapping mechanism. In the vertex processing stage, the vertex shader can manipulate associated texture coordinates for specific effects or animation. In the rasterization stage, the texture coordinates for each fragment are calculated by interpolating the texture coordinates of the vertices that make up the primitive being rasterized. The fragment processing stage is where texture mapping primarily takes place. The fragment shader receives the interpolated texture coordinates and employs them to sample the corresponding texture. Built-in OpenGL functions, such as `texture()`, `texture2D()`, or `textureCube()` [2] are used to sample the texture image and determine the final color and other surface properties for each fragment.

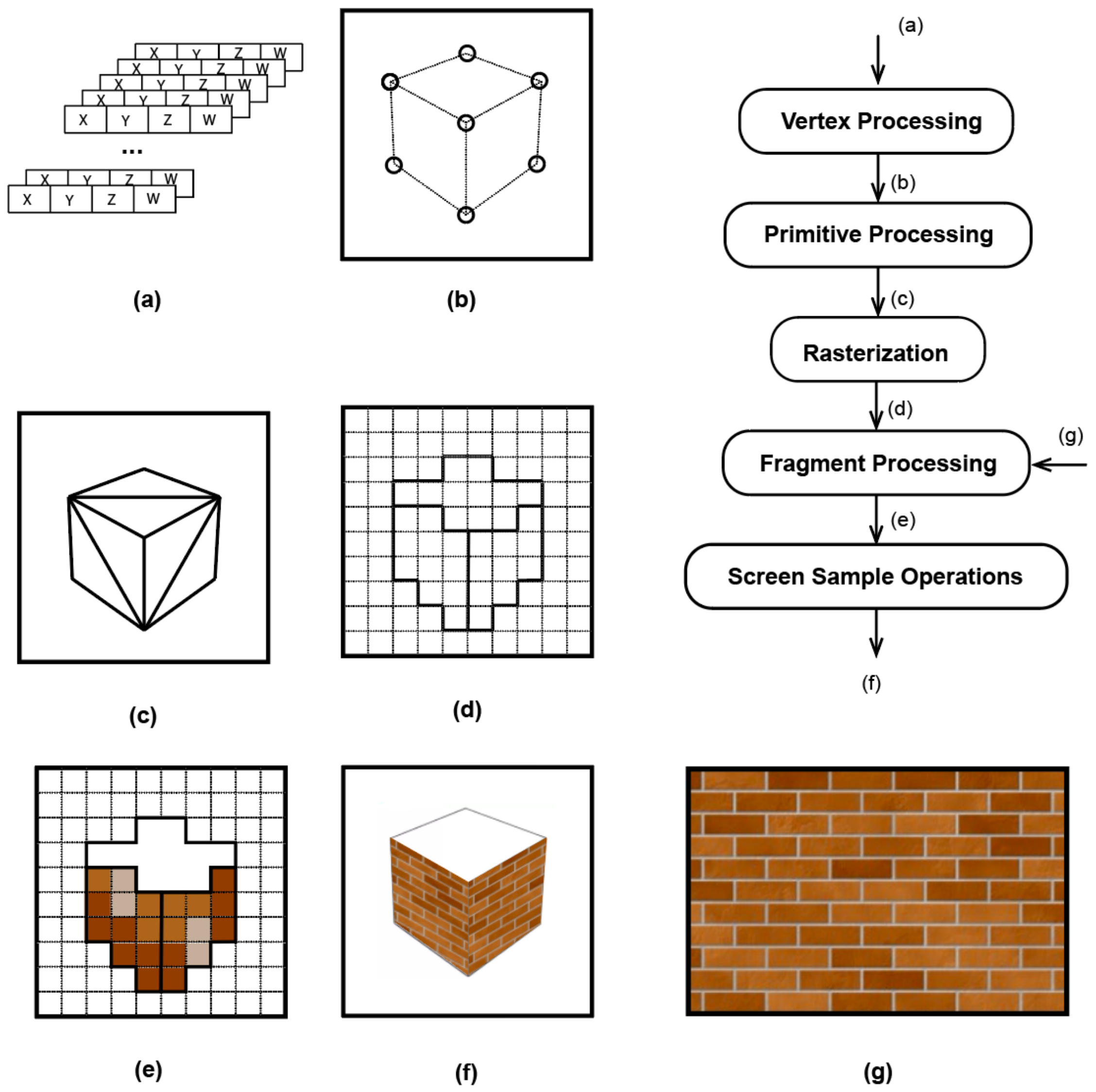


Figure 2. OpenGL rendering pipeline. OpenGL implements a rendering pipeline consisting of a sequence of interconnecting processing stages for converting model geometry data into rendered images. (a) vertex data 3D space (b) vertices in normalized coordinate space (c) triangle primitives assembly (d) rasterized fragments (e) shaded fragments (f) rendered image (g) texture file. Several stages of the modern OpenGL pipeline are omitted.

2.2 Texture as Information Container

Texture mapping refers to the mapping of a texture onto a surface. The word "texture" is used as a rather broad term. It goes beyond its usual meaning as a repetitive pattern and could refer to any multidimensional function saved as an image [3]. Texture mapping techniques use texture files as information containers to store various surface properties other than just color, such as transparency, surface normal, displacement, and specularity, to name a few.

Diffuse mapping is the most frequently used texture mapping technique and the term "texture mapping" originally refers to diffuse mapping. The technique simply maps a texture to a surface. The texture file is usually a bitmap image storing colors. Folding-texture technique also uses one of the texture files to store color.

Displacement mapping uses textures to displace the vertices of a 3D model. The texture file is used to represent the bumps and ridges on the object's surface. It is typically a grayscale image where the brightness of each pixel represents the height or depth of the corresponding point on the object's surface. Bump mapping [4] is a form of displacement mapping that works by perturbing the surface normal of a 3D model. When light interacts with the object's surface, the perturbed surface normals cause variations in the way the light is reflected and refracted, creating the appearance of bumps and ridges on the object's surface.

Normal mapping [5][6] uses a texture to store the object's surface normals. A surface normal is a vector that point directly outward from a surface, and it is used in lighting calculation to determine how light interacts with the surface. Unlike bump mapping that uses grayscale images to permute surface normal, normal mapping uses texture's red, green, and blue color channels to encode a 3D normal vector in tangent space. Folding-texture technique also encodes vectors using textures, but the encoded vectors are 2D instead of 3D vectors.

2.3 Time-varying Textures

The folding-texture technique can be used to generate textures with time-varying appearances. In this section, we briefly explain various techniques and terminologies related to the modeling, representation, and synthesis of time-varying textures. However, it is challenging to draw distinct boundaries between these terms. Sometimes two terms are used interchangeably, whiles in other instances, the same terminology may refer to different concepts.

Temporal Texture

Nelson and Polana [7] used the term temporal textures as a type of motion seen in images that repeats in similar ways across space and over time. The fundamental assumption behind temporal textures is that objects in many natural scenes have characteristic motions, but the space they occupy is indeterminate. This type of motion is common in nature scenes. Typical examples are trees or grass in the wind, ripples on a pool, and water flowing in a stream. The authors extended the basic idea of gray-level texture analysis to dynamic scenes to find structural or statistical patterns in the motion.

Szummer and Picard expanded on the concept of temporal textures and proposed a statistical model to represent and synthesize temporal textures [8][9]. They clarified that temporal textures are a sequence of images with indeterminate spatial and temporal motions. However, they also highlighted that two kinds of motions were not in the applicable scope of temporal textures. One is spatially non-repetitive or temporally non-periodic events such as kicking a ball. The other is temporally periodic but spatial confined motions like walking.

The texture is modeled as the outcome of a dynamic system [10]. They use a linear-temporal autoregressive model to model temporal textures,

$$s(x,y,t) = \sum_{i=1}^{p} \phi s(x+\Delta x_i, y+\Delta y_i, t+\Delta t_i) + a(x,y,t) \tag{2.1}$$

Each pixel is represented as a linear combination of adjacent pixels with space and time difference, where $s(x,y,t)$ is pixel value, and $\Delta x_i, \Delta y_i, \Delta t_i$ are the differences specifying the neighborhood. When the model is used in temporal texture synthesis, it first takes sample temporal textures as input and the parameters are learned by minimizing the conditional least square estimator. Once the parameters are determined, Gaussian random noise is used as new initial conditions and the synthesized texture is computed recursively using the model.

Temporal textures use statistical learning and assume that motion is indeterministic. Its application is limited to a few types of motion seen in nature. The folding-texture technique is intended to generate deterministic motions. For example, motion like a moving car is out of the scope of temporal texture but can be synthesized by the folding-texture technique.

Dynamic Texture

Dynamic texture [11] refers to a sequence of images with stationary motion. The things in the texture move with a consistent pattern and the overall pattern does not change over time. Typical examples include ocean waves, waterfalls, and steam [12].They assumed that a sequence of images are realizations of a stationary stochastic process. Thus, dynamic textures can be modeled as output of a dynamic system. Dynamic texture consists of sequences of images representing temporal and spatial variations in scenes. It can also be viewed as spatial texture's time-domain extension [13]. Dynamic texture is targeted at motions with stationary property and relies on complex mathematical modeling, while folding-texture technique does not have such limitations.

Video Texture

Schödl and Szeliski et al. [15] proposed video textures, which take a short video clip as input and aim to synthesize a new seamlessly looping video sequence, creating the illusion of an infinitely playing video. Video texture conceptualizes videos as textures by drawing the analogy with image textures, but later works on video generation dropped the term "texture" in naming [16]. The proposed technique segments the input video clip, extracts visual and motion features, and matches frames based on extracted features' visual similarity. The visual similarity computation results in a jump table or a transition probability table between frames. Video textures can be applied to represent chaotic and random phenomena, such as the motion of fire, wind, or water, but also be generated to depict objects' motion, including instances such as a person smiling or an individual engaging in physical activity, like running on tracks.

Motion Texture

Motion Texture has several different definitions. Li et al. [17] proposed a new statistical model for synthesizing realistic character motion and named it motion texture. The model consists of motion textons, the basic element in motion texture, and their distributions, and aims to capture both the low-level details and the high-level structure of human motion to generate natural and believable character animations. Enrique et al. [18] tried to differentiate the definition of dynamic texture and motion texture. They argue that previous dynamic texture studies, which focus more on texture's evolvement over time rather than its spatial repetitiveness, should be called motion texture instead. Researchers also use the term motion texture to emphasize their work's difference from video texture, as they use a static image rather than a video clip as input [19]. Motion texture is referred to as a time-varying displacement map generated from a specific physics model.

2.4 Systems for Animating Still Images

Various systems have been proposed to recognize, model, or synthesize time-varying textures. The systems differ in the input and motion specification methods. Some of them are mentioned in the previous section, for example the systems in [10] [15] use video as input, automatically recognize motion and synthesize time-varying textures. The system that will be introduced in this thesis, FoldingGen, uses still images as input and provides an user interface to specify motion. In this section, we will review a few systems used to animate still images and compare them with FoldingGen.

Chuang et al. [19] developed a system to animate still images by leveraging a physics-based model. The proposed system uses a stochastic model to represent five types of natural phenomena: trees swaying, water rippling, boats bobbing, clouds moving, and no motion. These models are taken from studies in other fields, e.g. structural engineering. As visualized in Figure 3, similar to FoldingGen, the system also requires the user's assistance to select the object that moves. The selected object is moved to a separate layer and a 2D displacement map, named motion texture, is applied to the layer to animate it. The user can control the resulting motion by adjusting the model parameters. The output moving picture is synthesized by compositing all animated layers into one.

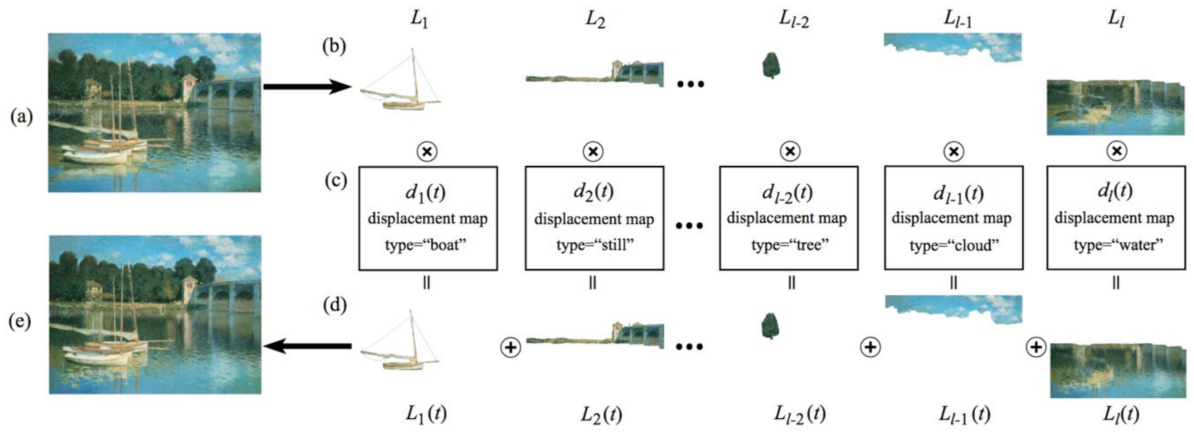

(a)
(b)
(c)
(d)
(e)
L_1
L_2
L_{l-2}
L_{l-1}
L_l
$d_1(t)$ displacement map type="boat"
$d_2(t)$ displacement map type="still"
$d_{l-2}(t)$ displacement map type="tree"
$d_{l-1}(t)$ displacement map type="cloud"
$d_l(t)$ displacement map type="water"
$L_1(t)$
$L_2(t)$
$L_{l-2}(t)$
$L_{l-1}(t)$
$L_l(t)$

Figure 3. System overview. Input image (a) is segmented into different layers by the user in (b). The layers are then animated with a motion texture in c and generated an

animated layer in (d). The final output (e) is assembled by compositing all animated layers. Image taken from [19].

Similar to FoldingGen, the proposed system also uses a semi-automatic user-assisted approach. But it does not provide a graphical user interface and the motions are specified by changing model parameters. In addition, the proposed system is only applicable to a few selected natural phenomena because the stochastic motion texture is based on a physics model which only describes motions caused by natural force. Consequently, the proposed system in [19] can only take real-world pictures as input while FoldingGen can take both real-world pictures and user drawings.

Holynski et al. [20] presented a system to animate still images by leveraging motion information derived from video clips. As visualized in Figure 4, the system has a motion estimation network that is trained with online video clips. When a new image is fed into the estimation network, an Eulerian motion field and displacement fields will be generated for the image sequentially. Then the displacement fields are applied to the encoded input image. Finally, the output is generated from the decoder network.

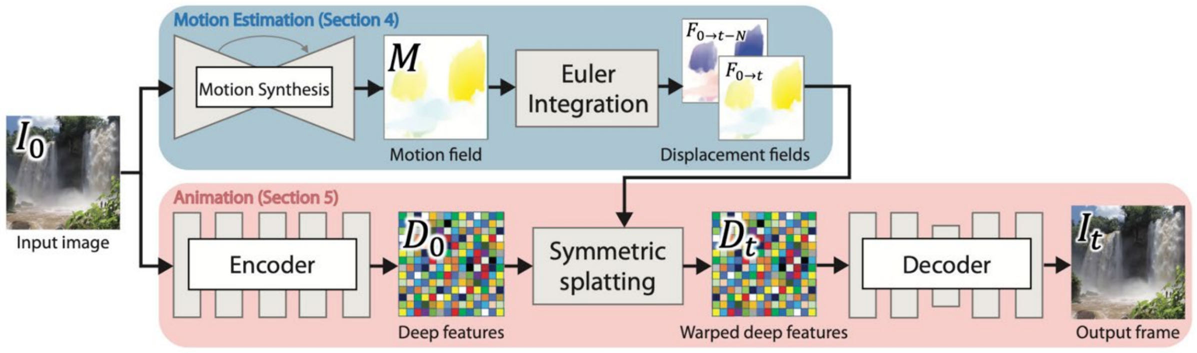


Figure 4. System overview. The input image is fed into two networks, a motion estimation network for a displacement map and a feature encoder network for a feature map. Then the displacement map is applied to the feature map and the output is generated from the decoder. Image taken from [20].

Similar to FoldingGen, the system uses a single static image as input. But the system is designed only for scenes with continuous fluid motion since it needs the motion to be represented by static Eulerian motion. Also, it takes a fully automatic approach so it does not need user interaction and consequently lacks the flexibility for the user to adjust the synthesized motion.

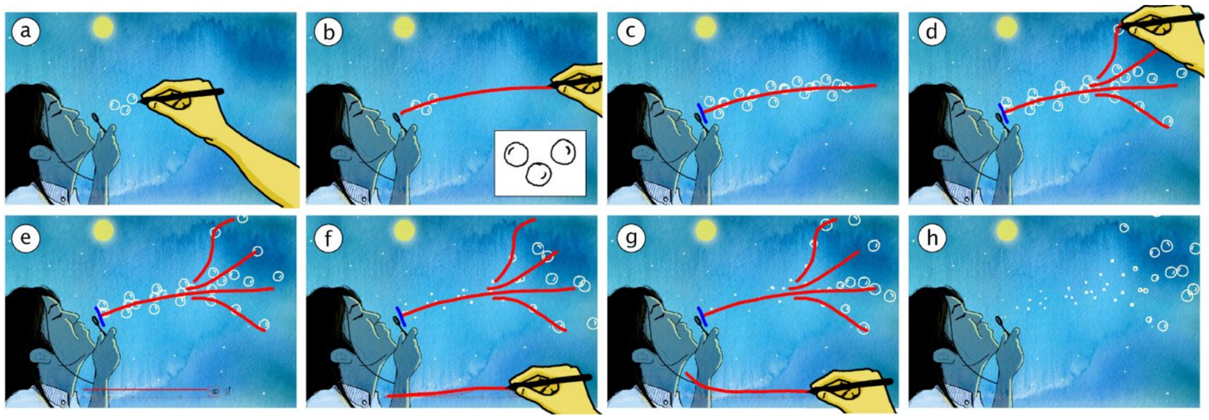


Figure 5. Draco workflow illustration a) User draws objects (b)(c)(d)(e) User specifies motion path (f) User adjusts scale change (g) User specifies velocity change (h) final result. Image is taken from [21].

In the two systems reviewed, motions are described by models and parameters. Systems used in animation authoring, on the other hand, usually provide an interface to define motions interactively. Kazi et al. [21] presented a system named Draco to add motion to still images. The system combines a graphical user interface with a procedural animation engine, enabling users to create and manipulate kinetic textures through a sketch-based interaction. To augment input images with motion in Draco, the user needs to draw patches, which are a collection of representative moving objects (e.g. the bubble in Figure 5a), and specify motion paths via a sketch. The patch and associated motion are named kinetic texture and serve as the basic animation component of the system.

Similar to FoldingGen, Draco provides a user interface and several specialized tools to enable the user to create desired motion with relatively little effort. Also, by involving users in the motion specification, it provides authoring capabilities which are not offered in the previous two systems. However, as can be seen in Figure 5, the moving object is not taken from the input still image whereas in FoldingGen the user can either select the moving parts from the image or draw the moving object in a separate texture. Also, Draco only supports a limited selection of motions common in animation, while the folding-texture technique is a more general way to create motions.

Draco was further extended in a new system named Kitty, which adds interactive behavior to generated dynamic illustrations [22]. Users can specify how moving objects' parameters are related to user interactions. The generated scene can respond to animation viewers' behavior live which is not in the scope of FoldingGen.

3 Folding Texture

To lay the foundation for further design and synthesis process, this chapter explains the folding-texture technique. The technique's origin and definition are discussed in Section 3.1. Then in Section 3.2, we present the dynamic sampling process from a single pixel's point of view and explain how time-varying textures can be represented by folding texture. Next, we explained how the folding motion is achieved in the technique in Section 3.3. Finally, in Section 3.4, we explored how the rendering effect relates to a single pixel's color change and proposed a few possible extensions of the folding-texture technique.

3.1 Definition

When I first read S. Thorben's folding texture thesis [1], my initial question was, "What is folding texture?". I tried to get a straightforward answer from the report. But unfortunately, it does not have a formal and concise definition in the thesis. It is referred to as "a new approach to simulate a kind of folding of the texture" and is presented as a series of procedures that can be implemented in the fragment shader to create the effect of a texture folding over itself.

Thorben used Figure 6 to explain the "folding" concept. The figure shows how a target pixel samples its color from the texture during the folding process. The color bar on top displays the pixel's colors over time and the black bar at the bottom shows the cross-section of the sampled texture. Two points in texture space, u_0 and u_1, mark the target pixel's associated texture coordinates at two different moments t_0 and t_1 (not shown in the image). In the middle and right illustrations, the texture is folded at different degrees. In the part of the texture that is folded, the color value of the pixel is sampled from the texture layered on the top. So in the middle illustration at t_0, the pixel changes its associated texture coordinates from u_0 to u'_0, and at t_1 its associated texture coordinates remain unchanged. While in the right illustration, the pixel samples color value from u'_0 and u'_1 instead of u_0 and u_1 respectively, due to stronger folding intensity.

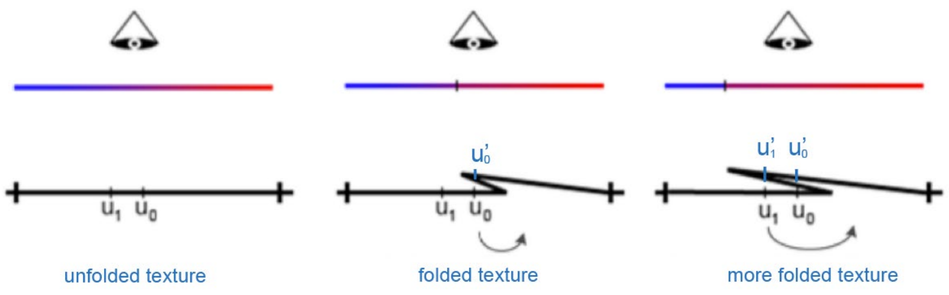


Figure 6. Basic idea of folding texture. The image is taken from [1] and modified (text in blue) for the convenience of explanation.

This folding process was formulated as a procedure consisting of several steps that can be implemented in fragment shader, as listed in Equations (3.1), (3.2), and (3.3).

$$colorValue = c(init(u,v).u_0, init(u,v).v_0) \quad (3.1)$$

$$fold(u,v,t) = c(init(u,v).u_0 + jump(u,v).x(t), init(u,v).v_0 + jump(u,v).y(t)) \quad (3.2)$$

$$\begin{aligned} x(t) &= t * x \\ y(t) &= t * y \end{aligned} \quad (3.3)$$

Each pixel takes its $colorValue$ from its initial texture coordinates at the first frame and changes its associated texture coordinates per predefined functions $(u(t), v(t))$ in the subsequent frames. These two steps are named “initial jump” and “time-dependent jump” in Thorben’s thesis report. In the initial jump, each pixel retrieves its initial texture coordinates $(u_0\, , v_0) \in [0,1]^2$ from a texture file called init-texture and uses them to sample color value from a texture file called color-texture. The notation $init(u,v)$ represents the operation of retrieving initial texture coordinates from init-texture and $c(u,v)$ represents the operation of sampling a color value from the color-texture. In the time-dependent jump, the texture coordinates change at a constant difference in the subsequent frames. The differences are encoded as a vector called jump vector and are stored in a texture file named jump-texture. The jump vector is defined as $(x(t), y(t)) \in [-1,1]^2$ and points from the initial texture coordinates to the final texture coordinates. The vec’or's time dependence is outlined in Equation (3.3), where $t \in [0,1]$, and x and y are the vector’s length in the x-axis and y-axis direction, respectively. When $t = 0.0$, $x(t) = 0.0$ and $y(t) = 0.0$, the time-dependent jump is about to start. And when $t = 1.0$, $x(t) = X$ and $y(t) = Y$, the time-dependent jump occurs at its full length.

As mentioned earlier, the folding-texture technique uses three types of texture files, color texture, init-texture, and jump-texture. The color-texture is an image. It is used as the fundamental texture containing all color information to be sampled. The size of the color texture may be larger or smaller than the texture ultimately applied to the object, depending on whether new content needs to be stored or duplicated portions are displayed on the object. The init-texture stores all pixels’ init jumps and the jump-texture stores all pixels’ jump vectors. The texture format is listed in Table 3.1 below.

Table 3.1 init-texture and jump-texture contents

Channel	Init-texture	Jump-texture
R	u_0	$x(t)$
G	v_0	$y(t)$
B	Δt	t_0
A	f_0	t_1

The “folding” of the texture is achieved by using a specific type of time-dependent jump as visualized in Figure 8. Two threshold parameters, t_0 and t_1, were added to time-dependent jump to have a cutoff version of Equation (3.3). Consequently, the time-dependent jump can happen at any given time whereas the time-dependent jump in Equation (3.3) only happens right after the initial jump is performed.

$$fold\_jump = \begin{cases} f(t), & if\ t_0 < t \leq t_1 \\ 0, & otherwise \end{cases} \quad (3.4)$$

The threshold parameters, t_0 and t_1, corresponds to the time where the function has abrupt vertical uplift or drop in Figure 8. For each pixel, t_0 marks the frame when the pixel starts to sample color value from locations other than the initial texture coordinates and t_0 marks the frame when the pixel samples back to its initial texture coordinates.

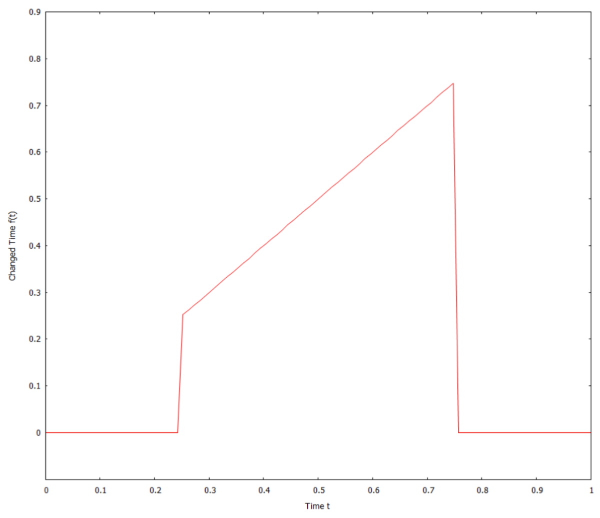


Figure 7. Modification of functions through thresholds. The image is taken from [1]

Refined Definition

The folding-texture technique was presented as a procedure in Thorben's thesis. As explained and summarized in this section so far, it is definitely not a simple procedure that users can quickly learn and apply. The goal of this thesis project is to develop an approach for folding texture design and synthesis so that users can employ the folding-texture technique more easily via our approach. As the first step of the thesis work, we explored the folding texture's definition in more detail and gave it a refined definition.

The folding-texture technique is a **texture-mapping-based** technique that can be employed to represent and synthesize **time-varying textures** with the "folding" effect. The "folding" effect is achieved by adopting **a specific type of texture dynamic sampling**. The technique uses three types of textures**, color-texture, init-texture,** and **jump-texture**, as representation.

The phrases in bold are the building blocks of folding texture's definition. We have explained texture mapping in the context of the rendering pipeline in Section 2.1 and have reviewed a few time-varying texture related techniques in Section 2.3. Next in this chapter, we will explain the rest of these key concepts one by one. Besides, we will also present a general form of the technique which could be studied in the future.

3.2 Texture Dynamic Sampling Process

For standard texture mapping, sampling is done statically. In the rendering pipeline, texture coordinates are pre-calculated for each vertex and stored with geometry. Then geometric primitives are converted into fragments where vertex attributes are interpolated for each fragment in the vertex shader. Then fragment shader is invoked to determine the color value of each fragment. It uses the received interpolated texture coordinates to sample the texture at the corresponding locations. The texture coordinates remain constant throughout the rendering process, resulting in a fixed appearance.

In texture mapping, the texture is an image and a function of spatial coordinates. Time-varying texture extends texture's definition in the time domain. It can be used to render a sequence of images and is a function of both spatial coordinates and time. As a technique that can be used to generate time-varying textures, folding texture needs to define the dynamic sampling process in rendering. Instead of using static texture coordinates, fragment shader retrieves texture coordinates from pre-designed and computed init-texture and jump-texture at runtime, allowing the texture to varying appearance over time.

So we will first review the texture dynamic sampling process in folding texture to understand how motion can be rendered with texture mapping. Texture dynamic sampling in this report refers to the sampling process employed in the folding-texture technique, during which fragment-associated texture coordinates changes between frames. For the convenience of explanation, we define a sampler for each fragment in the texture space. The sampler position represents the target fragment's associated texture coordinates and the target fragment will sample the color value where the sampler is located. Consequently, the trajectory of the sampler represents the whole texture dynamic sampling process.

Figure 8 shows a sampler's linear movement in the texture space, as formulated by Equation (3.3). The sampler's trajectory is called one jump in the folding-texture technique and its path is called jump path. One jump path can be represented as a segment with a predefined length and direction in texture space, which is depicted in Figure 8. During the sampling process, the sampler moves from the init-position to the jump-position along the jump path and samples color for the associated pixel at each frame. The rendering result is visualized in Figure 8a.

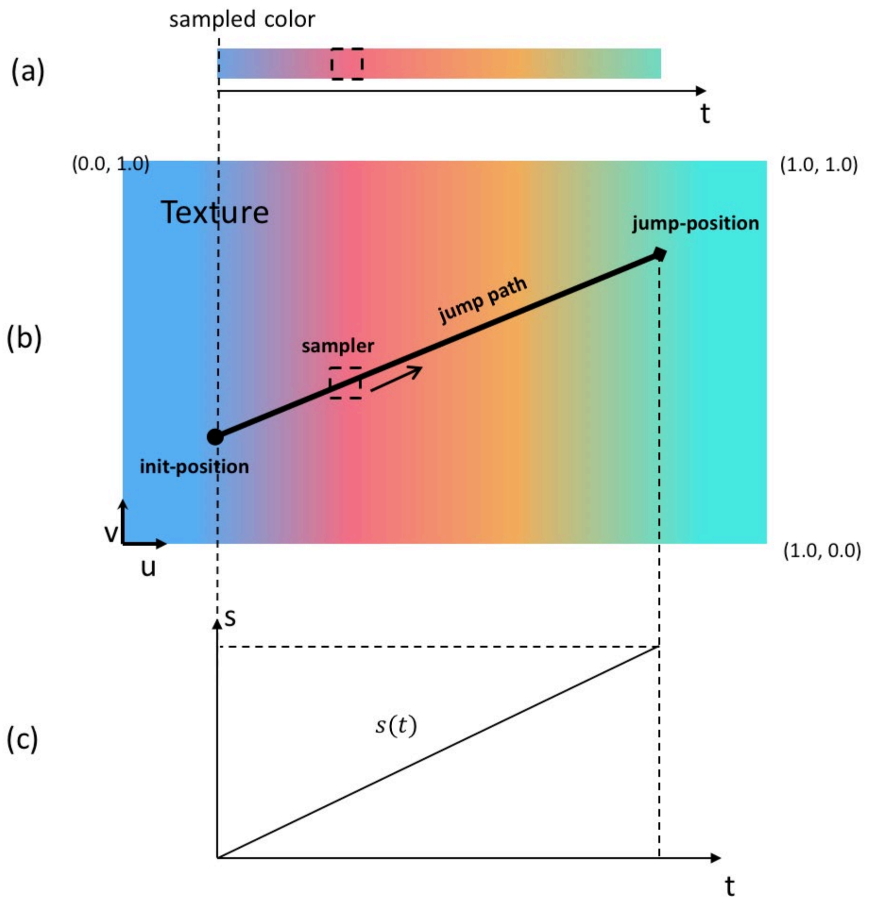


Figure 8. Texture dynamic sampling process. A directed segment, pointing from init-position to jump-position, visualizes the whole sampling path. The sampler (dotted box) moved from init-position to jump-position along the jump path at a constant speed.

For Motions that cannot be represented by a single segment, the sampler's complex movement can be decomposed into a series of interconnecting jump paths as visualized in Figure 9. Each jump path represents a discrete proportion of the sampler's movement, and when combined in sequence, they form the complete, continuous trajectory that the sampler follows during the sampling process. However, in this report, we will limit our scope to single-jump trajectory and leave multiple-jump support for future work.

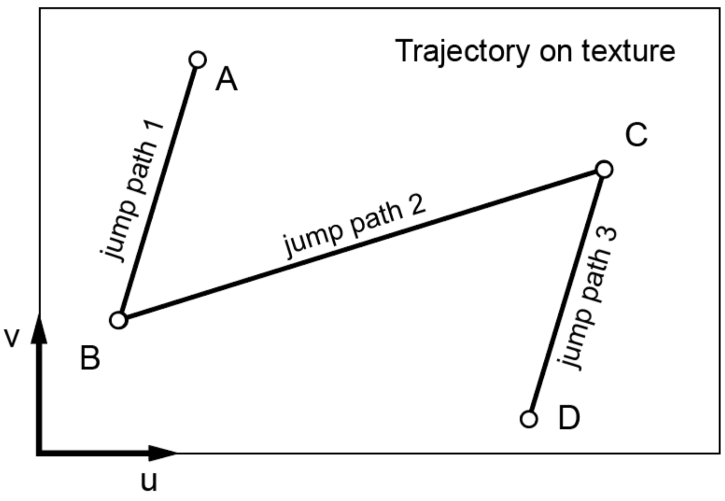


Figure 9. Multi-jump trajectory. A sampler's whole trajectory (A-D) consists of three jump paths (A-B, B-C, C-D).

The sampler in Figure 8 follows a uniform linear movement. But a sampler's movement does not necessarily need to be continuous. For example, the sampler could move back and forth along the jump path to have oscillatory behavior, or it could bypass certain parts due to external constraints. The sampler's movement does not need to be uniform either. It can also move along the jump path at time-varying speeds, for example, accelerating or decorating based on a predefined speed distribution function to create more realistic rendering results. For example, the sampler may move quickly through areas with uniform properties or slow down to capture finer details in regions with complex structures. By controlling the actual movement of the sampler, certain effects can be achieved, and they will be explained in detail later in this chapter. In fact, "folding" is achieved by adopting a type of non-uniform nonlinear movement as shown in Figure 8.

3.3 Folding

Figure 1 illustrates the general folding concept and we extend the illustration to explain the folding progress with more details in Figure 10. The images in the left column show the progress that the texture folds over itself, and the images in the right column display the corresponding rendering result. On the input image in (a), two vertical dotted lines mark the region where folding will happen. The texture starts to fold over itself in (b). In the rendering result on its right, the wave part of the texture starts to shift to the left a bit and occludes part of the beach. When folding is done in (c), the region of the beach with the grey shell is occluded by the waves, and the waves reach their maximum and final position. As visualized by the right column images, we achieve the motion that waves swashing and backwashing by **folding** the input texture like a piece of paper.

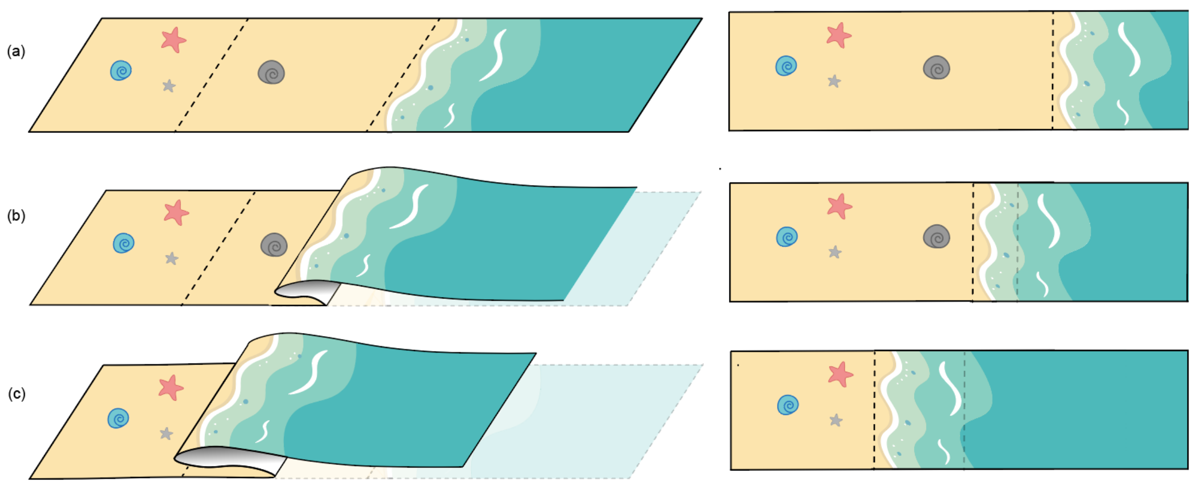


Figure 10. Wave Swashing and Backwashing by Folding. The subplots show the input image (a) before folding (b) folding in progress (c) after folding and corresponding rendering result.

So how can we explain this folding effect in the context of the texture dynamic sampling process introduced in the previous section? We will explain it with a more complex blinking eye example. Figure 11 illustrates how the texture is dynamically sampled around the eye area. The eyeball region is the area where the folding will happen. When the eye closes, the outer boundary of the eyelid 'enters' the eyeball area first, eventually covering most of the eyeball at the end of the movement. For points A, B, and C, the color is sampled along the jump paths, depicted as blue arrows in Figure 11b. Point C is not affected by folding and only follows the linear movement as in Figure 11d. Points B and C are involved in folding. We would like them to maintain their initial color until the eyelid arrives. So for points B and C, a t_0 parameter is introduced to represent the moment time eyelid object first reaches that point. The eyelid starts to appear at point A at time t_a and point B at time t_b, while point C moves across the eyelid. If point A is set to take the initial color prior to t_a, the color value from the eyeball texture will be displayed. The same thing happens at all points initially in the eyeball region, including point B. Without this t_0 parameter, the eyeball will appear squeezed to close instead of progressively occluded by eyelids.

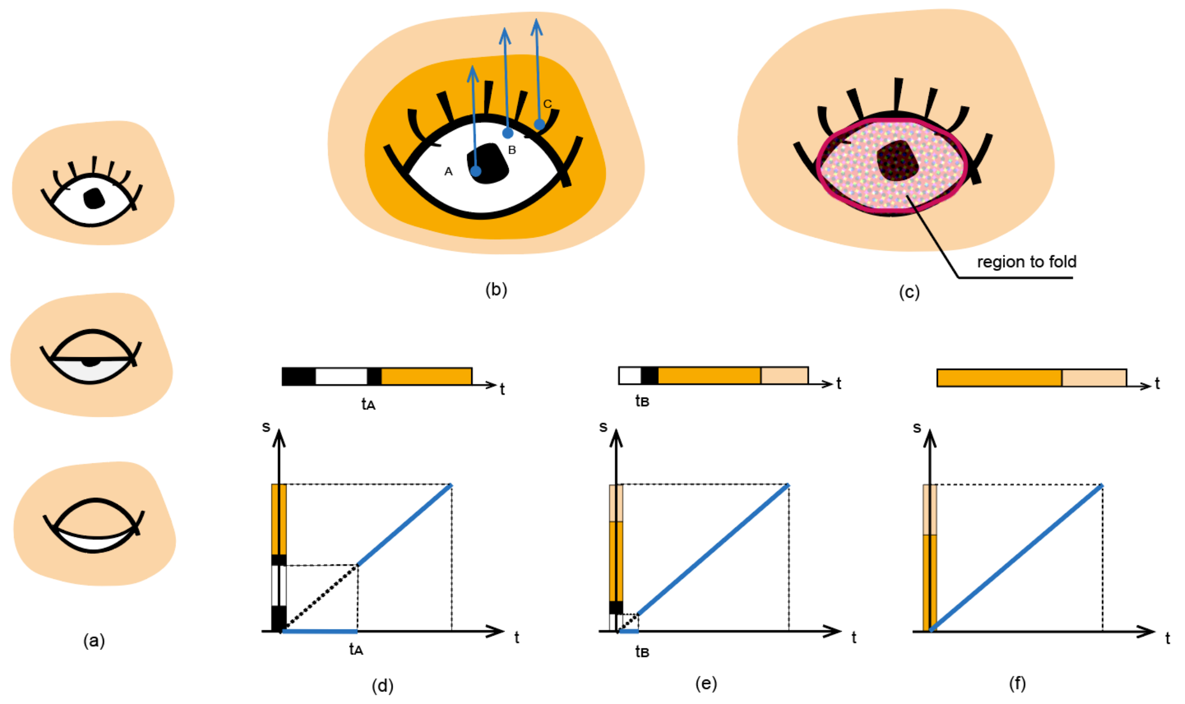


Figure 11. Texture dynamic sampling process of a blinking eye.

The sampler trajectory used in folding can be abstracted as in Figure 12a. By using this type of trajectory in the texture dynamic sampling process, we can fold the texture and achieve the rendering result that one part of the texture is moving and progressively occluding the static part.

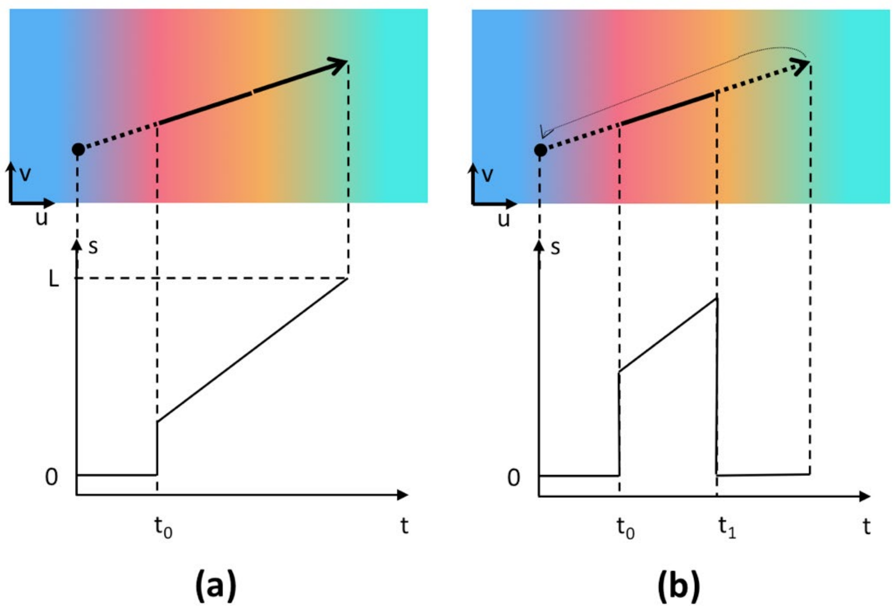


Figure 12. Piecewise linear time-dependent jump used in folding texture. a) is used to create “folding” visual effect (b) can be used for creating “flying over” effect.

Figure 12b shows another type of time-dependent jump used in Thorben's thesis and it leads to a kind of fly-over effect as shown in Figure 13. The points in motion zones maintain their original color first and start to sample color from the vehicle region at t_0, which creates an effect that the vehicle runs into the motion zone region. Then at t_1, motion zone points abruptly change their texture coordinates back to their original position, creating an effect that the car runs out of the motion zone region. The end rendering result looks like the vehicle part of the texture "flying over" the background part.

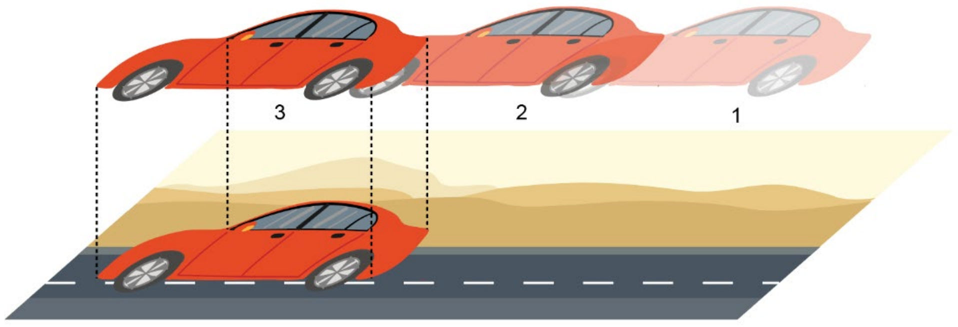


Figure 13. "flying-over" effect.

3.4 More Than Folding

Both Figure 8 and Figure 12 show the sampler's trajectory in the dynamic sampling process. The sampler could have a more complex trajectory than the ones shown in these two figures. Its transient behavior along the straight line path could be described by a pre-defined position function, $s = p(t)$, which represents the sampler's distance from the init-position along the jump path. In fact, with different deliberately designed position functions, we could have granular control over the sampler's motion and achieve various complex visual effects. In this way, we could extend the folding-texture technique by having a more complex sampler trajectory than the ones shown in Figure 12. We will first explore further on a sampler's trajectory's representation.

The sampler position function $p(t)$ can be classified into two categories based on its mathematical definition: linear and nonlinear. Both types of functions can be used to define the motion of the sampler along the jump path in folding texture techniques, and each can result in different visual effects.

3.4.1 Linear Position Function

A linear position function can be formulated as Equation (3.5) and can be represented as a straight line in the two-dimension Cartesian coordinate system.

$$p(t) = a * t + b \tag{3.5}$$

It is a more generalized form of Equation (3.3). As shown in Figure 14, any change in t results in a proportional change in the sampler position. This proportionality is determined by the slope constant a, which means the sampler moves at a constant speed along the jump path, resulting in a uniform distribution of sampled values along the path. Linear position functions can be useful for creating simple, predictable effects and transitions, such as uniform translation.

The y-intercept b represents a shift in the coordinate system. The temporal shift used in [1] was actually implemented via b.

$$b = a * \Delta t \tag{3.6}$$

Where Δt is the temporal shift, and Δt_1 and Δt_2 in Figure 14 are two examples.

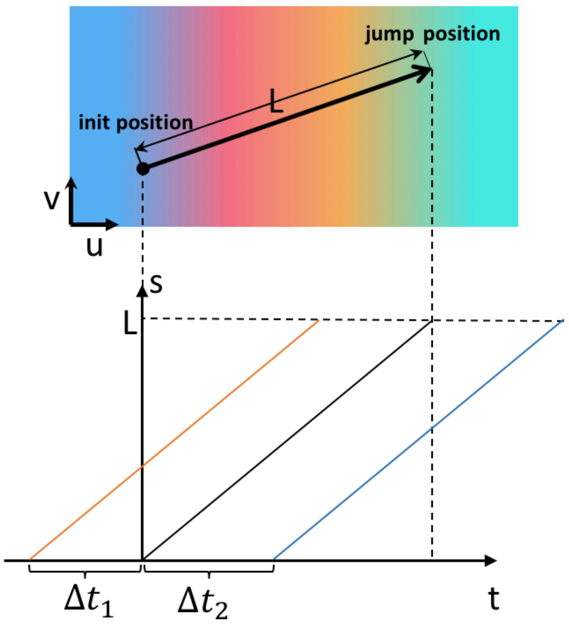


Figure 14. Linear position function.

3.4.2 Nonlinear Position Function

Nonlinear functions can be useful for representing more complex and non-uniform motion in texture dynamic sampling, which can create more natural effects, as well as more complex and diverse visual patterns.

Nonlinear functions like exponential function, sigmoid, and exponential inverse function can be used as position functions. With the exponential or sigmoid function, the samp'er's position changes result in non-uniform acceleration or deceleration. With sine or cosine functions, the sampler creates oscillatory motion along the jump path. This can be useful for simulating wave-like phenomena or repeating patterns, such as wave-like textures, pulse effects, or vibrating objects.

Piecewise linear position function

A piecewise linear position function consists of multiple linear functions over different intervals of its input domain. In other words, the function is composed of several linear segments joined together at specific points, with each segment having its own linear equation. By defining a series of linear functions over different intervals, you can create a path that has abrupt changes in direction, speed, or both at certain points, leading to various visual effects.

Figure 15 shows three examples of piecewise linear positions. In subplot (a), the sampler moves at a constant speed until t_0 and maintains the same color value until t_1, then skips part of the jump path and continues moving to the jump position at a constant speed. If we set $t_0 = 0$ and $p(t_0) = 0$, the position function in Figure 15a becomes the position function used in the folding-texture technique (see Figure 12a on page 16).

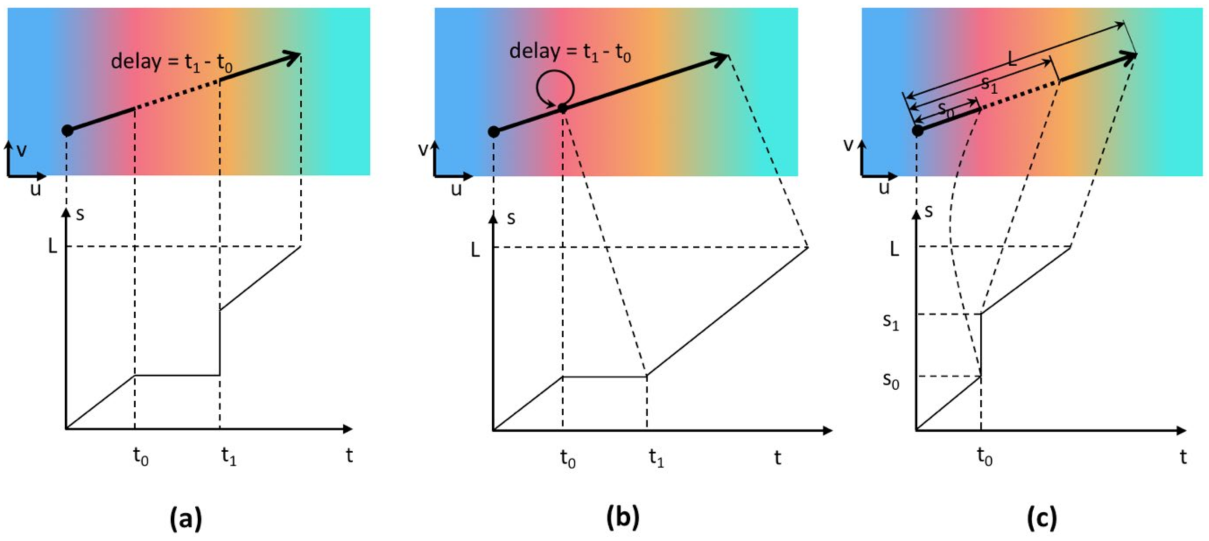


Figure 15. Examples of piecewise linear position functions.

In example (b) sampler moves at a constant speed the whole time except maintaining the same color value at t_0 for a period of time. A good scenario to visualize the rendering effect it could achieve is vehicle runs and stops at a red sign and then continues its journey when the traffic light turns green.

In example (c), we see an abrupt change at t_0. This could be used to create visual effects like teleport, where t_0 represents the teleportation point. Depending on the desired effect, you can adjust the values of s_0, s_1, and t_0 to control the motion of the sampler before and after the teleportation, as well as the teleportation point and the distance between the starting and ending locations.

4 Design and Synthesis

In this chapter, we will first give an overview of the basic workflow of our approach. Then we explain the two concepts introduced in the approach. The introduction of these two concepts will enable users to specify motions on a per-zone basis. We offer several specialized tools for motion specification. We will explain in detail the methods behind them and how they could assist in the design and synthesis of folding textures.

4.1 Workflow Overview

We propose a novel approach that enables users to design and synthesize folding textures effectively. The workflow is demonstrated in Figure 16. Given an input image, the user identifies regions where movement occurs, denoted as motion zones. Then, the user specifies the movements, denoted as motion profile, in each zone using the tools provided by our system. Our system offers various specialized tools and algorithms for folding texture design and synthesis, catering to various needs and preferences. Once the parameters for these tools have been suitably adjusted, fine-tuned, and configured, the folding textures – consisting of color-texture, init-texture, and jump-texture - are generated, providing a comprehensive output for further analysis and application.

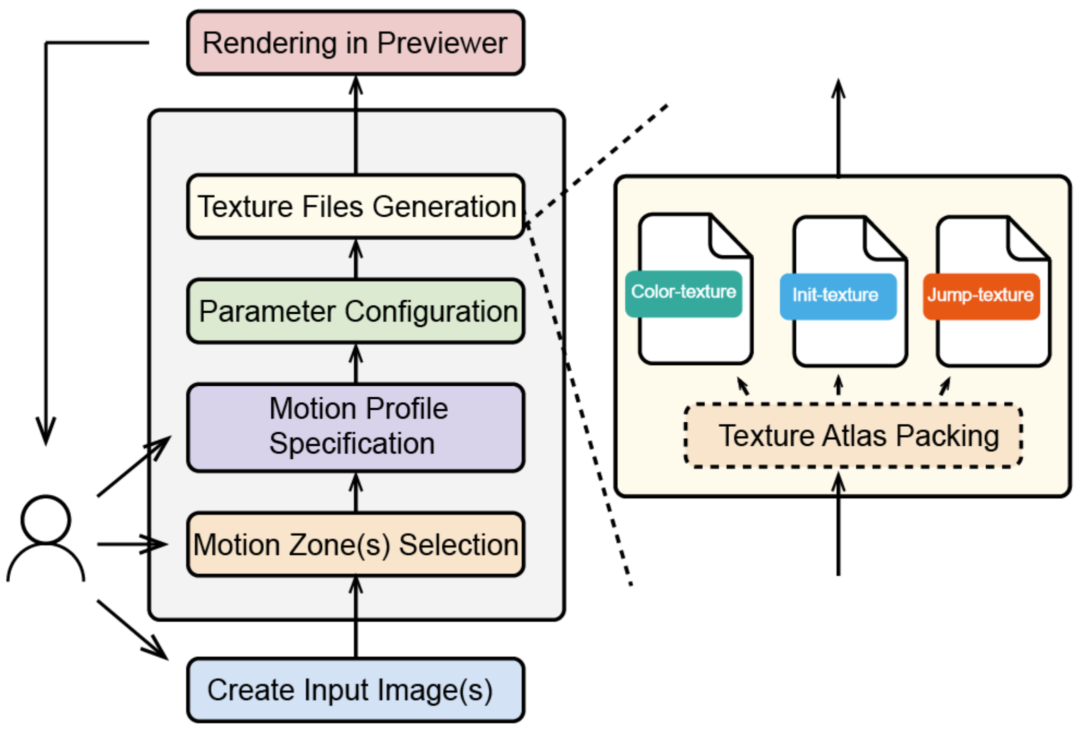


Figure 16. Workflow overview.

Integration of image editing software into the system permits users to directly create input images using programs such as GNU Image Manipulation Program (GIMP), with the system automatically importing the resulting images. Moreover, a simple previewer has been incorporated into the system, enabling users to examine on-the-fly the rendering outcomes of the synthesized folding textures.

4.2 Motion Zone and Motion Profile

In the previous chapter, we have described the sampling process from the angle of a single pixel. The pixel's color changes over time can be represented by its associated sampler's movement in the texture space. So far folding textures were calculated and created manually by specifying these movements for every pixel.

We want to make it easier for the users to use folding-texture via our approach. The notion of motion zone and motion profile are introduced to help user to specify the movements for a collection of pixels instead of for every individual pixel. A user-placed motion zone indicates a region of the output image in which motion takes place.

Figure 17 illustrates the rendering result of the motion zone with specified movements. The 9 pixels in the motion zone have their specified jump paths depicted as black arrows. Their initial color is sampled from their init-position on the jump path. Once the motion zone is drawn on the image, the init-positions and init-texture are determined. These containing pixels' samplers then moves along their jump paths with the default linear position function. The rendering results at each frame are shown in Figure 17 (b), (c), (d), and (e) respectively.

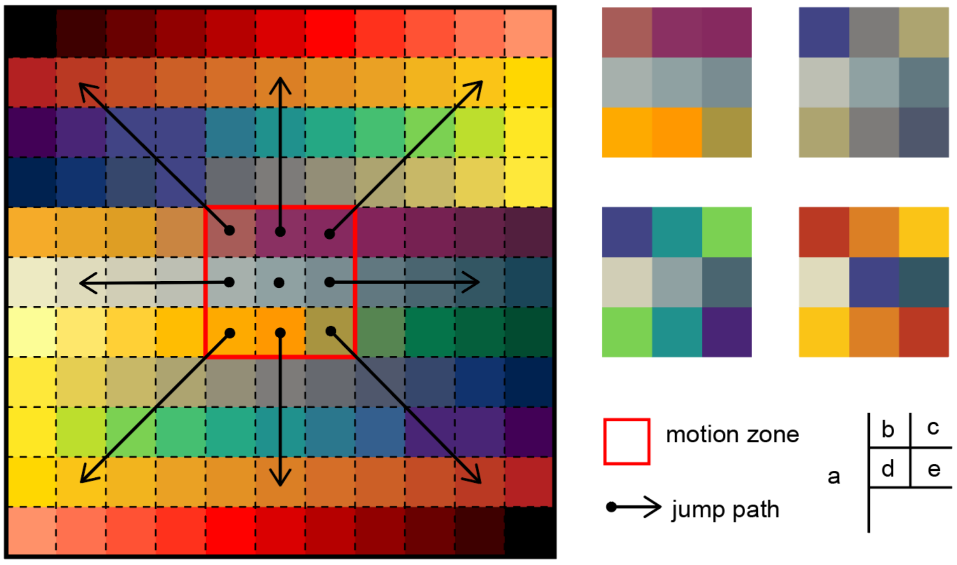


Figure 17. Per-zone view of the sampling process. (a) motion zone and all associated jump paths on the color texture (b)(c)(d)(e) motion zone's rendering result along the time

One motion zone may span the entire input image or a portion thereof. An image could have one or multiple motion zones, with each motion zone potentially incorporating several objects or parts of an object (shown in Figure 18). In practice, it is the motion properties or motion profile of the object or part of the object that a motion zone encapsulates that distinguishes motion zones from each other.

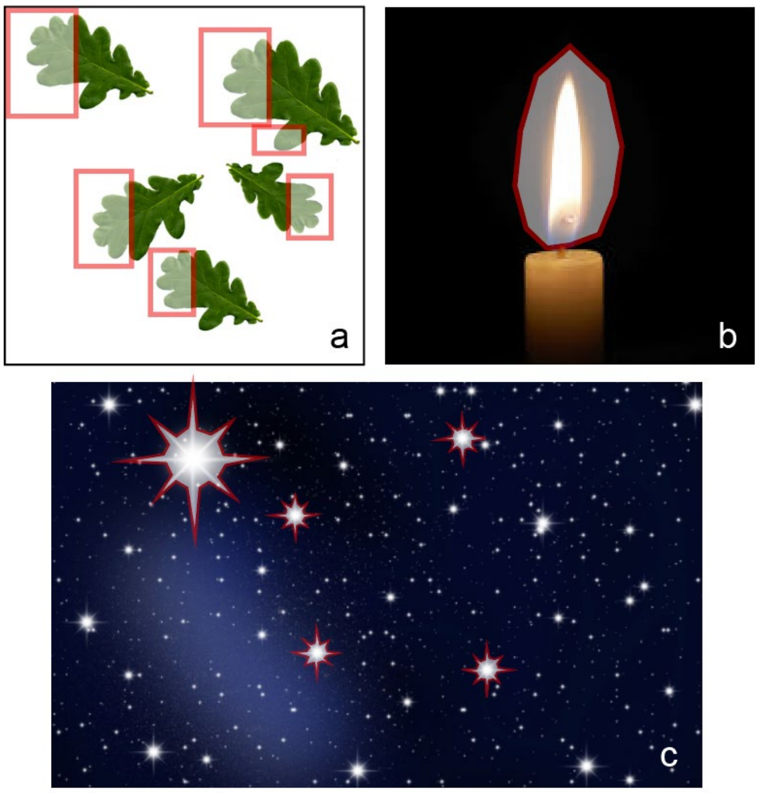


Figure 18. Motion zone examples.

(a) multiple rectangular motion zones (b) single polygonal motion zone (c) multiple polygonal motion zone

The term motion profile is commonly used in motion control systems to describe the desired position, velocity, and acceleration of an object or a set of objects over time. It typically encompasses information about the position, velocity, and other motion-related parameters of the object or system as it traverses through space. In this report, motion profiles are utilized to describe the dynamic movement of samplers in a motion zone. This concept facilitates the creation of realistic movement of objects, enabling them to interact and move with one another in a real-world manner.

As elaborated in the previous chapter, the movement of a sampler is uniquely determined by two elements: its jump path and position function. Consequently, the jump paths and position functions of all pixels within the motion zone form the motion profile.

A jump path can be represented mathematically by its initial position and a jump vector which is pointing to the jump path’s jump-position. It could be expressed either by an init-position/jump-position pair or an init-position/jump vector pair. Once a motion zone is specified on the input image, its containing pixels’ init-positions are implicitly established. Then the user needs to specify both jump paths and position functions to define the desired motion profile for the motion zone.

The representation of the position function exhibits a greater degree of variation, depending on the type of function. For instance, in the case of a linear position function, once a jump path is established, only one additional parameter, sampling step size, needs to be provided. For nonlinear position functions, the function type, and associated parameters must be specified. As for piecewise linear functions, they can be perceived as a combination of linear functions, with each segment defined by the start and end times, as well as speed (the slope of the function).

So far, the user needs to manually create folding texture by specifying movements for every sampler. We introduce motion zones and motion profiles to enable users to specify movements per region instead of per pixel. But users still need to do a lot of manual work to specify the motion profile for a motion zone. How could we further ease this work? Since the motion profile is formed by containing samplers' jump paths and position functions. The problem to address evolves to how could we specify jump path and position function in batch in a motion zone.

We offer a few specialized tools for motion profile specification. The tools can be categorized into two groups, one for the specification of jump vector, and the other for the specification of position function. They will be explained in Sections 4.3 and 4.4 respectively. Users can use both categories of tools in conjunction to describe complex motions.

4.3 Jump Vector Specification

To specify motion profiles for an image containing intricate details, users must carefully specify the jump for sample pixels. As the number of sample points increases, more complex motion effects can be achieved. For instance, creating the motion profile of a smiling face, where the eyes, forehead, mouth, and nose exhibit different motion profiles, requires defining hundreds of sample points. Manually crafting such a texture with high detail and minimal artifacts pixel by pixel can prove to be a demanding task.

So how could we define a bunch of jump vectors at once? We propose two ways to approach this. One option is to specify how jump vectors are spread out within a specific area, determining their local distribution. Another option is to specify a few sample vectors, and all the rest of the jump vectors are calculated from spatial interpolation.

4.3.1 Geometric transformation-based Jump Vector Specification

The adoption of geometric transformation is intuitive. By perceiving a single motion zone as an individual object, one can apply specific types of transformations, such as translation or scaling, to the motion zone. Consequently, it is expected that the jump vectors will undergo a similar transformation, resulting in a particular distribution pattern.

These tools are intended for application on a per-motion-zone basis, implying that when employing this category of tools, points within a given motion zone exhibit identical or highly similar motion profiles.

Consider, for instance, the case of translation transformation applied to a motion zone. In this instance, all points located within the motion zone will share an identical jump vector, with the only difference being their initial positions.

The scale transformation tool is illustrated in Figure 19. When the motion zone (depicted as a black rectangle) is scaled 3 times in the x direction, all its containing pixels' jump vectors have magnitude that equals 3 times the length of its distance to the central symmetric axis. As the result, motion zone's rendering output will change from Figure 19b to Figure 19c.

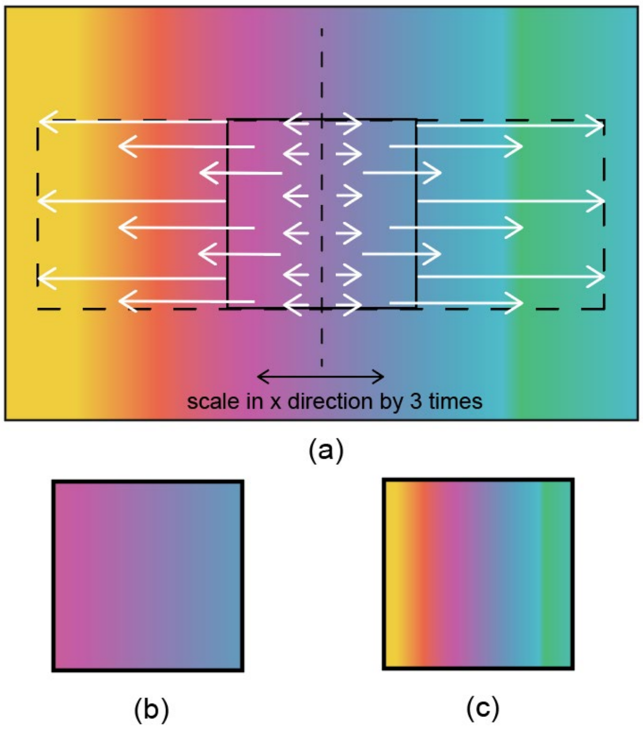


Figure 19. Scale Tool Illustration. (a) Scale tool in action (b)(c) rendering out of the rectangular region

4.3.2 Interpolation-Based Jump Vector Specification

The geometric transformation-based tools are simple and intuitive in nature. In practice, they have demonstrated effectiveness when the motion profile follows a clean and uniform pattern. However, given the finite variety of geometric transformations available, real-world scenarios may exhibit motion profiles that cannot be described by these transformations, resulting in undesired results.

To address this limitation, we proposed a group of tools that employ spatial interpolation for batch specification of jump vectors. Spatial interpolation is a widely used technique in Geographic Information Systems for estimating unknown values at specific points using known points as sample points. In our approach, points with user-specified jump vectors are treated as known points or anchor points. The jump vectors of neighboring points are calculated via interpolation, guided by a few selected parameters.

Various spatial interpolation methods are available, including Shepard[23], radial basis function[24], spline interpolation[25], and as-rigid-as-possible interpolation[26]. We chose to utilize a modified version of Shepard interpolation due to its efficiency and simplicity. It is important to note that extensibility was a key consideration in the design of our approach. The implemented interpolation method could easily be replaced with an alternative algorithm if desired.

Three variants of the Shepard Interpolation method are provided in the implemented proof of concept, the original Shepard Interpolation, the Modified Shepard Method, and Modified Shepard Method with Randomness proposed in this report.

Shepard Interpolation

Shepard Interpolation, also referred to as Inverse Distance Weighting (IDW) interpolation, is a technique employed in spatial interpolation to estimate values at interpolated points based on a set of sample points. Values are interpolated based on the inverse distance to the sample points. This is illustrated in Figure 20a.

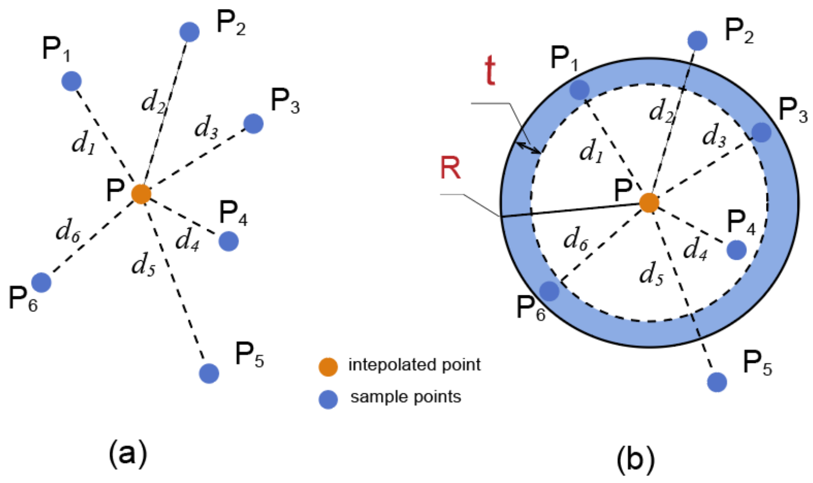


Figure 20. Variants of the Shepard method.

Given a set of sample points $\{\boldsymbol{p_i}, \boldsymbol{u_i} | for\ \boldsymbol{p_i} \in R^n, \boldsymbol{u_i} \in R\}_{i=1}^{N}$, the interpolated value $u(x)$ is computed as:

$$u(\boldsymbol{p}) = \begin{cases} \dfrac{\sum_{i=1}^{N} w_i(\boldsymbol{p})\boldsymbol{u_i}}{\sum_{i=1}^{N} w_i(\boldsymbol{p})}, & if\ d(\boldsymbol{p}, \boldsymbol{p_i}) \neq 0\ for\ all\ i, \\ \boldsymbol{u_i}, & if\ \ d(\boldsymbol{p}, \boldsymbol{p_i}) = 0\ for\ some\ i \end{cases} \qquad (4.1)$$

where $w_i(\mathbf{x}) = \frac{1}{d(\mathbf{p}, \boldsymbol{p_i})^a}$ is the weights calculated from inverse of the distance.

Modified Shepard Method

Modified Shepard's method (MSM) [27] aims to reduce the bull-eyeing effect seen in the original Shepard interpolation. It reduces the expressive local values that could cause artifacts by only considering sample points within a user-defined radius R (see Figure 20b). The weights are adjusted as follows:

$$w_i(\mathbf{x}) = \left(\frac{\max\,(0,\ \ R - d(\boldsymbol{x}, \boldsymbol{x}_i))}{R \cdot d(\boldsymbol{x},\ \boldsymbol{x}_i)}\right)^2 \qquad (4.2)$$

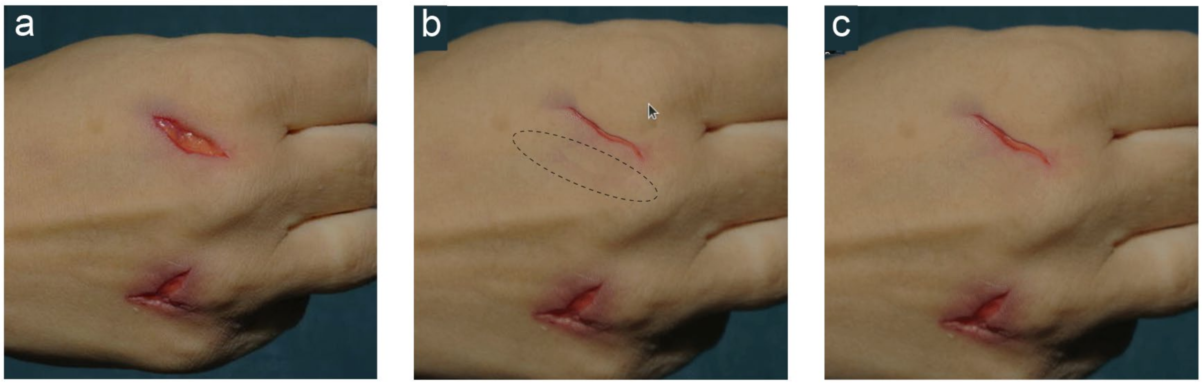


Figure 21. Algorithms comparisons. (a) Input image (b) Modified Shepard Method (c) Modified Shepard Method with Randomness

Modified Shepard Method with Randomness

However, in practice, minor artifacts (see circled region in Figure 21b) are still observed near the R-sphere boundary. To eliminate these artifacts (see Figure 21c) and enhance the realism of the results, we propose a further improvement of the original algorithm, termed Modified Shepard Method with Randomness (MSMR).

Directly inside the boundary R, we introduce a ring-shaped randomness region of width t (depicted blue zone in Figure 20b). In this ring, sample points are randomly selected for inclusion in the computation. The weights are subsequently adjusted as follows:

$$w_i(\mathbf{x}) = \left(\frac{\max\left(0,\ \ k(\boldsymbol{x}_i) \cdot \left(R - d(\boldsymbol{x}, \boldsymbol{x}_i)\right)\right)}{R \cdot d(\boldsymbol{x},\ \boldsymbol{x}_i)}\right)^2$$
$$k(\boldsymbol{x}_i) \begin{cases} \in {}_R S, & if\ |d(\boldsymbol{x}, \boldsymbol{x}_i) - R| < t \\ = 1, & otherwise \end{cases} \qquad (4.3)$$

where $S = \{1, 0\}$.

Based on the interpolation method discussed above, we offer a few specialized tools to facilitate the specification of key points' jump vectors.

Point Mapping

This tool enables users to draw a single jump vector on input images. It is a simple but powerful tool. Theoretically, it can be employed to specify a jump vector for all the points within the motion zone. But as we have mentioned before, this will be tedious work for users. In practice, this tool typically serves as a complementary tool to other tools, adding some "outlier" yet crucial jump vector.

Curve Mapping

Relying solely on point mapping tools is not sufficient. We observed that in practice it is very common for geometrically associated pixels to have similar jump. For instance, the points on the upper eyelid have jump vectors that are perpendicular to the eyelid's boundary.

Orzan proposed that contours can be used to represent and edit digital images [28]. Motion in the image consequently can be represented as the motion of contours in the image. To take advantage of this insight curve mapping tool is introduced to enable users to draw curves to specify jump vectors along contours.

The curve mapping tool is based on the cubic Bezier curve with a set of anchor points as visualized by Figure 22a. With the tool, users can draw the curve on the input image and specify jump vectors ($V_0 \sim V_5$ in Figure 22b) at a few anchor points ($A_0 \sim A_5$ in Figure 22b) along the curve.

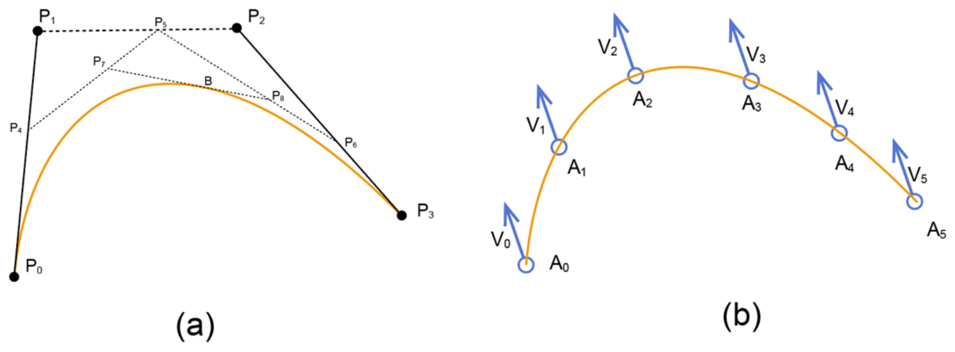


Figure 22. Bezier curve and curve mapping tool.

When the anchor points' jump vectors are set (Figure 23a), interpolation is performed along the curve to generate jump vectors for all points on the curve. Users can set the distribution of jump vectors. We offer a few shortcut options for the users to select. The anchor points serve as sample points for along-the-curve interpolation (Figure 23b). The number and density of anchor points can be configured by the user. In the next step, the neighboring pixels' jump vectors (Figure 23c) are interpolated using vectors on the curve as sample vectors.

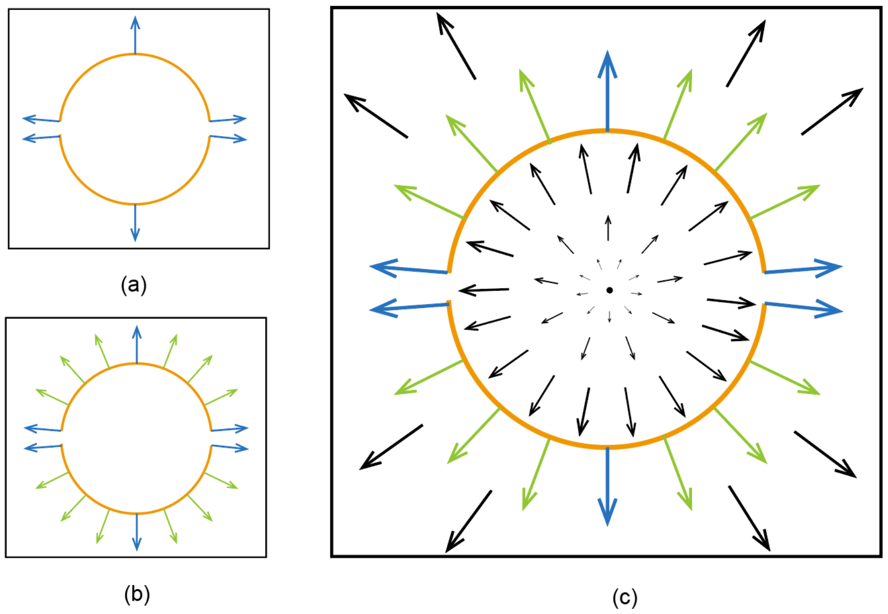


Figure 23. Interpolation steps in curve mapping

Lasso Deform Tool

Inspired by the lasso selection tool found in image editing software, the lasso deform tool enables users to create freehand deformations of motion zones surrounding objects or specific areas within an image.

Vertices of the motion zones polygon (see Figure 24) are translated using the mouse cursor. For each vertex, the jump vector is simply derived from its start- and end position. Jump vectors of pixels inside the motion zone are interpolated from these vertices using the method described above.

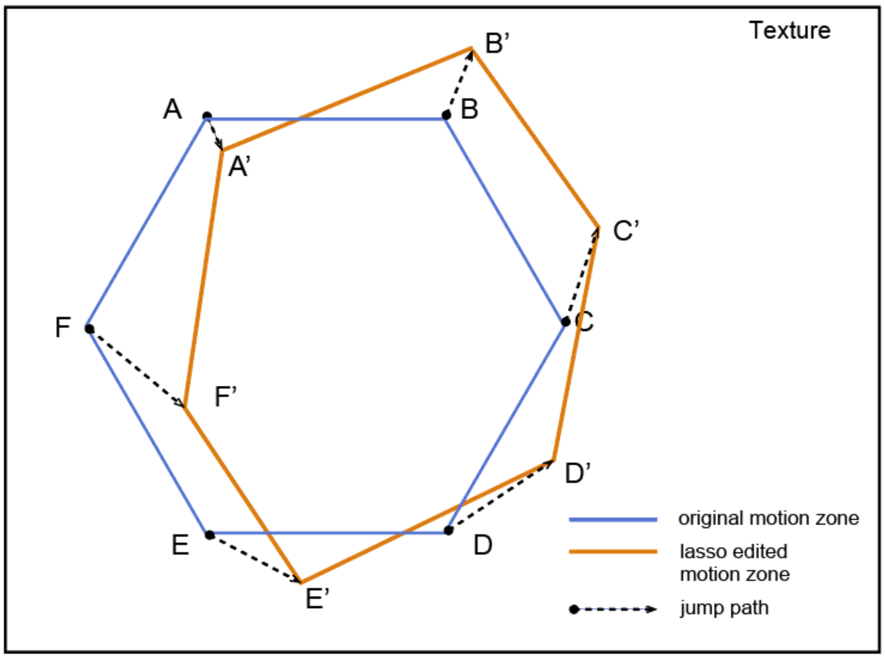


Figure 24. Mechanism of Lasso Deform Tool. With the tool, users can drag and drop the vertices to transform the original polygonal motion zone (blue) to a different shape(orange). Each vertex's original position and new position form its jump vector.

4.4 Position Function Specification

Linear Position Function

The expression for the linear position function is simple (see Equation (3.5)). With the jump path established, only two additional parameters—speed (as in Equation (3.5)) and temporal shift (Δt in Equation (3.6))—need to be specified. The speed will be set as a per-texture parameter, while the temporal shift will be set as a per-motion-zone parameter.

Piecewise Linear Position Function

The folding texture technique employs two types of piecewise linear position functions, as shown in Figure 25. In addition to speed and temporal shift, the frame where a discontinuity occurs must also be specified. These moments are referred to as thresholds and represented as t_0 and/or t_1 parameter in [1].

A Folding Zone tool is introduced to enable the intuitive specification of these parameters. As the name implies, the region marked by the folding zone will be "folded over" at some stage during dynamic sampling.

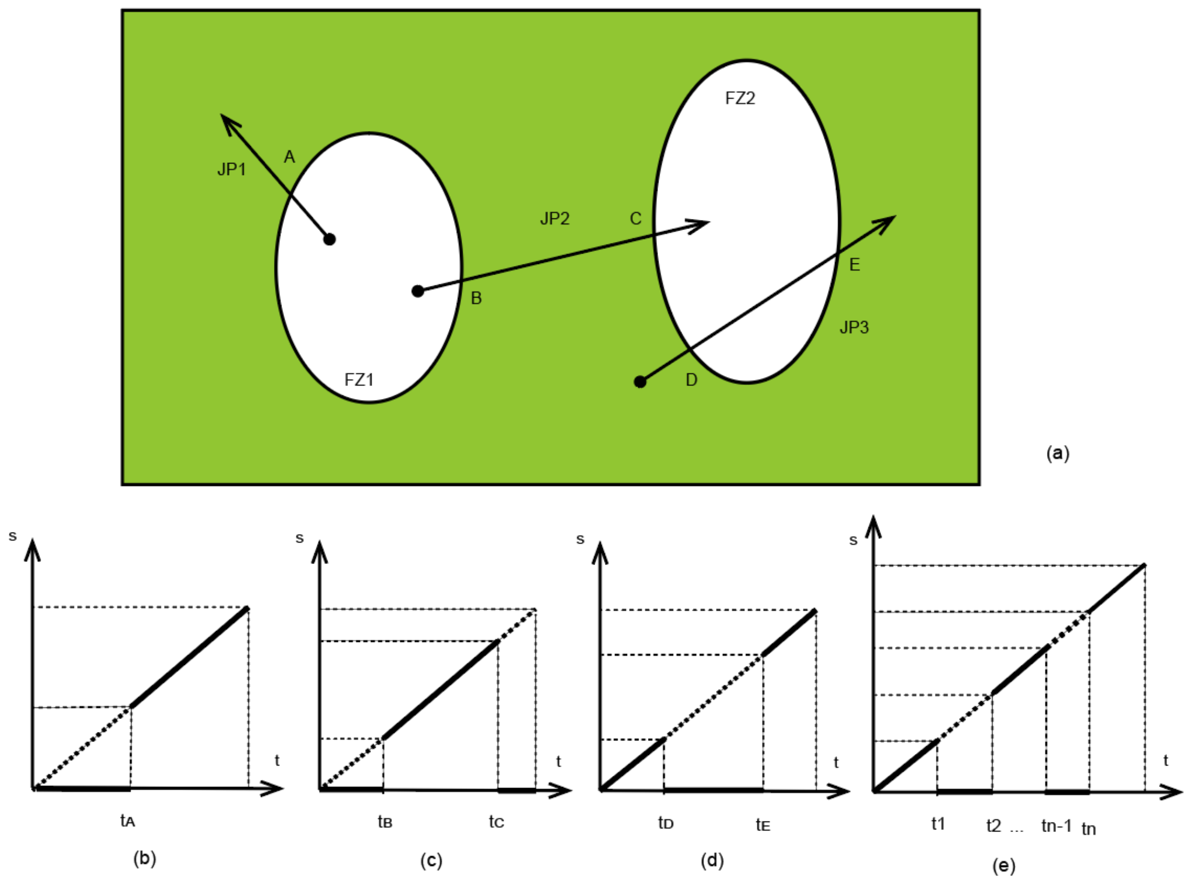


Figure 25. Folding Parameter Computation. two motion zones, FZ1 and FZ2, on the input image. Jump path JP1 intersects with FZ1 at A. JP2 intersects with FZ1 and FZ2 at B and C. JP3 intersects with FZ2 at D and E, respectively

It is essential to note that these parameters may differ for each point. For every point within the folding zone, the parameters are determined by the intersection points between the jump path and the folding zone boundary. Figure 25 demonstrates the relationship between these parameters, jump path, and motion zones. When a jump initiates outside the folding zone and traverses one motion zone, the t_D and t_E parameters in Figure 25d correspond to the two moments when the jump path intersects with the folding zone boundary at. When the jump path originates inside a motion zone and ends at another motion zone, the t_B and t_C parameter in Figure 25c relate to the moment that the sampler reaches B and C. When a jump path starts inside one motion zone and ends outside motion zones, the moment sampler interests with motion boundary is the parameter t_A in Figure 25b.

Theoretically, one jump path could intersect more than two motion zones. Each intersection with a motion zone adds an additional parameter, the position function representation, as shown in Figure 25e. However, this thesis work limited its scope to at most two motion zone interactions. Likewise, other types of piecewise linear position functions (see Figure 15bc) are excluded from the scope of this thesis.

Other Nonlinear Position Function

We offer a limited number of alternatives for other types of nonlinear position functions, such as exponential function, sigmoid, and exponential inverse function. The constant parameter in function expression can be set as an additional parameter, similar to what is done for the speed parameter. These functions will be available in discretized form in the shader. Additionally, these basic nonlinear functions could also serve as a sub-function of a piecewise function, like the linear sub-function in a piecewise linear function. See examples in Chapter 6 for more details.

4.4.1 Alpha Mask as an alternative for folding zone

In our approach, alpha mask can serve as an alternative way to specify regions where “folding” occurs. Alpha blending is a technique for combining or blending two or multiple images based on their respective alpha values.

An additional image needs to be provided to serve as a mask for the original input image. The mask image should be identical to the original image, with the only difference being that alpha values of folding regions are set to zero. As depicted in Figure 26, the sampler switches between the original image and the mask image, taking the color value from the original image when the alpha value is zero and from the mask image when the alpha value is non-zero.

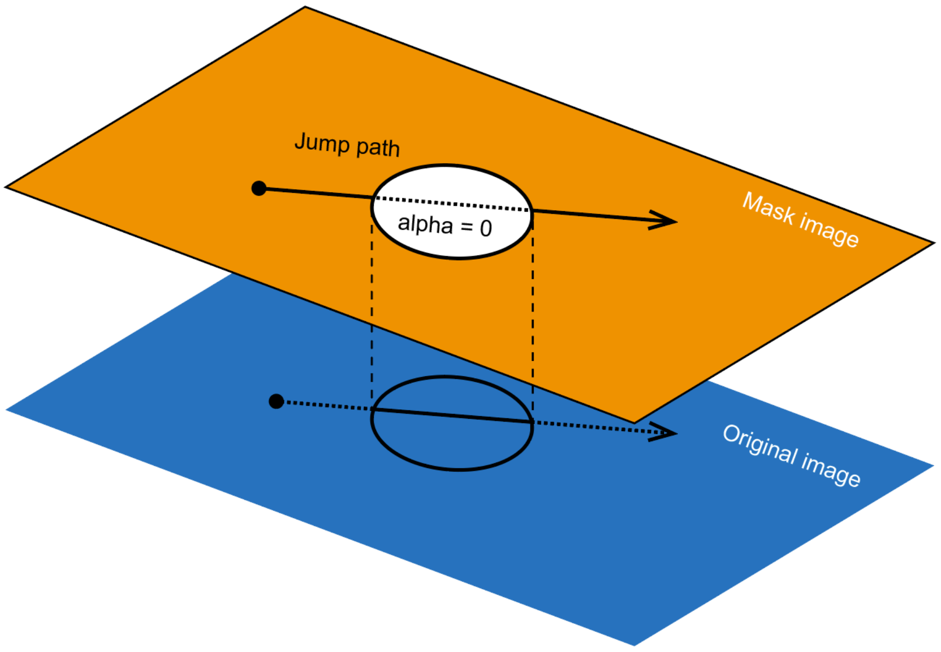


Figure 26 Illustration of alpha blending usage

4.5 Multiple Motion Zone and Multiple Texture Support

Samplers can jump from one image to another in folding. Naturally, our approach supports multiple images as input. When using multiple input images, one is typically selected as the background or scene image, while the others where samplers will jump to are partially rendered over the scene image. The motion zone is specified on the scene image and can be moved inside the same image or to another image and back to the scene image.

A texture atlas is a large image that combines multiple smaller textures into a single image, as visualized by Figure 27. This technique is widely used in computer graphics and game development to reduce the number of draw calls and improve rendering performance. In color-texture assembling, we crop parts of the input images that contribute to the rendering result and assemble these pieces together into a texture atlas.

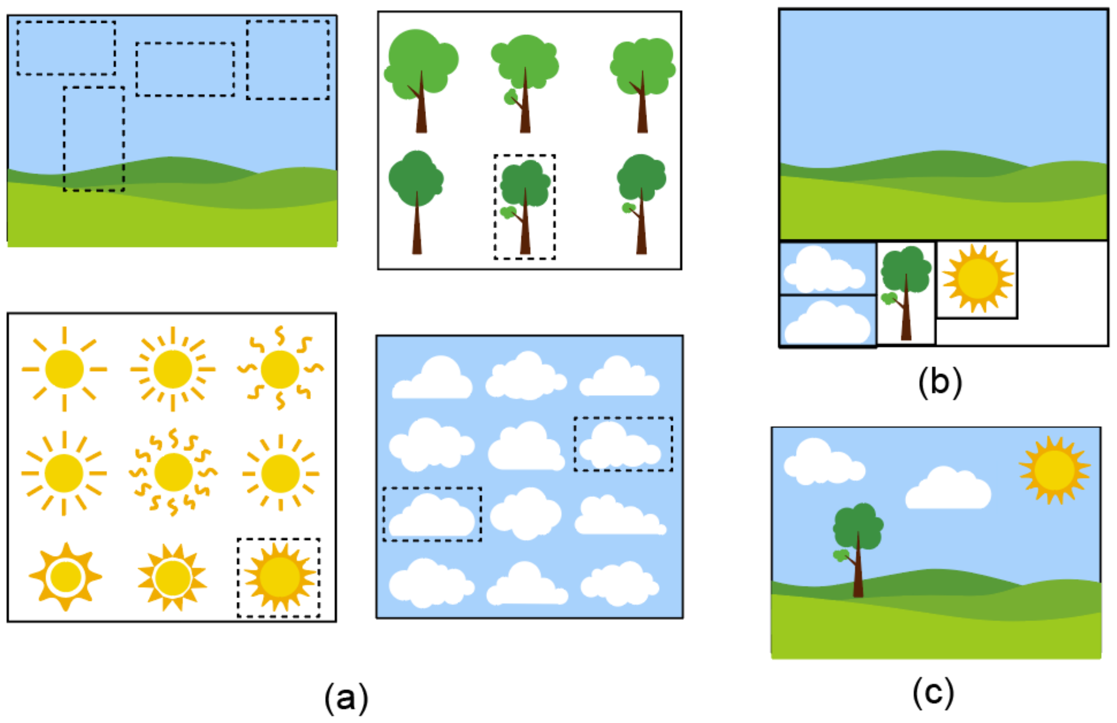


Figure 27. Illustration of color texture assembling process. (a) input images (b) assembled color texture (c) one frame from the rendering result

By combining multiple input images into a single image and cutting off the unused regions, the overall memory usage can be reduced. Also, by using a single texture file, the number of texture switches and draw calls required during rendering are minimized, which can improve runtime performance.

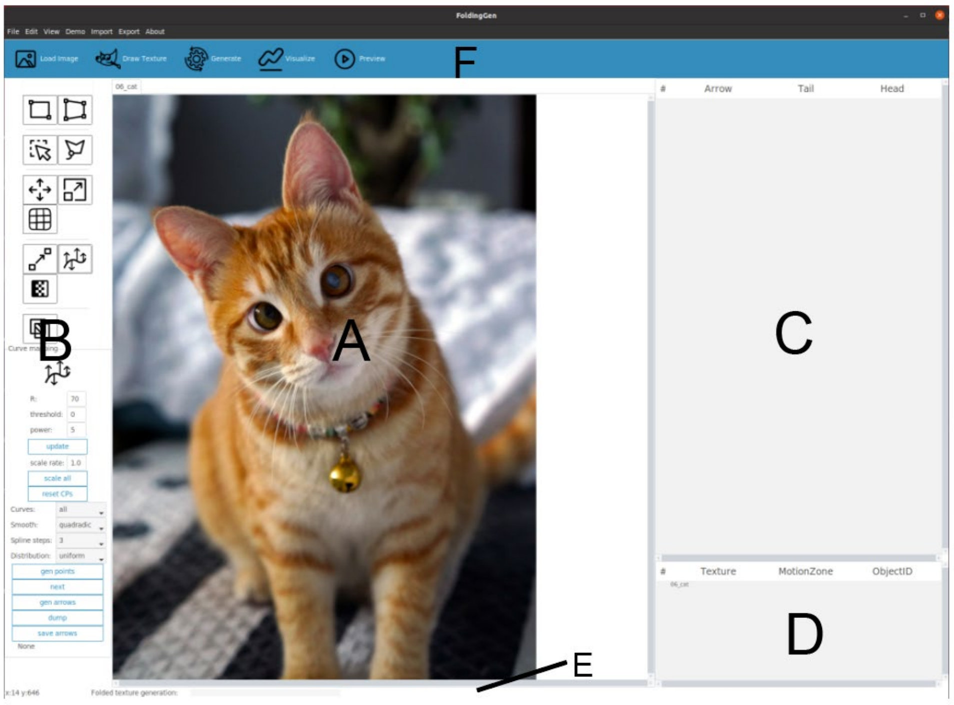
5 Implementation

In this chapter, we dive into the implementation details of the algorithms and methods used. We have developed FoldingGen as proof of concept. The software features a graphical user interface with specialized tools, an integrated previewer to verify rendering results, and an integrated visualizer to show the distribution of generated jump vectors.

5.1 FoldingGen

FoldingGen is developed in Python and C++, utilizing various open source modules and frameworks to implement its features and functionalities. The interface is built using the standard Python GUI (Graphical User Interface) library Tkinter, facilitating the creation of customizable and platform-independent user interfaces. Tkinter integrates seamlessly with the Tcl/Tk library, providing a wide range of widgets and tools for crafting intuitive GUIs.

The FoldingGen interface layout is designed for efficient folding texture creation. The workspace is divided into several panels and toolbars. As shown in Figure 28 the canvas occupies the central area of the interface where input images are displayed, and motion zones and motion profiles are specified. The toolbox panel, located on the left side of the interface, contains various tools for drawing and transforming motion zones, such as rectangle, polygon, translation, and scale tools. It also includes tools for motion profile specification and displays an optional property panel for the current tool in use.



R: 70
threshold: 0
power: 5
scale rate: 1.0
Curves: all
Smooth: quadradic
Spline steps: 3
Distribution: uniform
None
x:14 y:646
Folded texture generation:

Figure 28. FoldingGen interface layout. The interface consists of (A) Canvas(with input image imported), (B) Toolbox Panel, (C) Jump Path Tree, (D) Motion Zone Panel, (E) Status Bar, and (F) Workflow Panel.

On the upper right side of the interface, the jump path tree displays a list of all the jump paths, grouped by motion zone. Users can add, delete, or modify the init-position and jump-position of each jump path. Below the Jump path tree, the Motion Zone Panel lists all the input images and motion zone a in hierarchical structure. Users can move motion zones between input images, which is essential for multiple input image support. At the top of the interface, the menu bar provides access to various commands and options, such as Demo, Import, Export, and Help. These menus allow users to perform tasks like Import/Export Motion Zone, Import/Export Motion Profile, and accessing help documentation. Below the menu bar, the Workflow Panel contains buttons that trigger the stages in the proposed workflow, as described in Section 4.1. Located at the bottom of the interface, the Status Bar displays information about the current cursor position, texture synthesis progress, and other relevant details.

The ability to save work progress and resume later is a crucial feature for software usability. Saving work progress ensures that any changes made during a work session are preserved. FoldingGen enables users to save drawn texture, specified motion zone, and motion profiles for later import. This allows users to work on a project at their own pace, taking breaks or dividing their work into smaller sessions as needed. When exporting motion zones and motion profiles, the data is saved in human-readable JSON format, and the software’s internal running status is saved in a text-based dump file, which has proven quite useful in debugging and reproducing examples.

5.2 Texture Synthesis Algorithms

As illustrated in Figure 16, folding texture consists of three components: color-texture, init-texture, and jump-texture. Color-texture serves as the fundamental texture, storing all the color information required for rendering. Like conventional texture mapping, we store the color texture in a commonly used file format.

As mentioned in Section 4.5, we improve memory usage by packing the input images into a single texture atlas. Algorithm 1 outlines the color texture assembling process.

ALGORITHM 1: ALGORITHM TO PACK COLOR TEXTURE

```
     Input: input images, motion zones, and motion profiles
     Output: color texture file F
1    Create output file F
2    Initialize image_list
3    for each input_image
4        for each motion_zone
5            cropped_image = crop(input image, motion zone, motion profile)
6            image_list.append(cropped_image)
7        end for
8    end for
9    atlas_texture = atlas_packing(image_list, packing_config)
10   write atlas_texture to F
11   return F
```

A jump path can be represented mathematically by its initial position and a jump vector which is pointing to the jump path's end position. So the init-texture stores init positions and the jump-texture stores the jump vectors.

The init-texture holds motion zone related information, containing texture coordinates of all points within a motion zone's axis-aligned bounding box. The last channel is used to mask pixels outside of the motion zone. The synthesis algorithm is shown in Algorithm 2.

ALGORITHM 2: ALGORITHM TO GENERATE INIT-TEXTURE FILE

```
     Input: input images metadata, color texture, and motion zone data
     Output: init-texture file $F$
1    Create output file $F$
2    for point $(x,y)$ contained in the motion zone's rectangular bounding box
3        $X = (x + atlas\_offset\_x) / image\_width$
4        $Y = ( y + atlas\_offset\_y ) / image\_height$
5        if $(x,y)$ is in the motion zone then
6            $z = 1$
7        else
8            $z = 0$
9        end if
10       write $X, Y, z$ to file $F$
11   end for
12   return $F$
```

Jump-texture stores the jump vectors for points within the motion zone. When using interpolation-based motion profile specification, Algorithm 3 is used. The `calculate_threshold()` method in the algorithm determines the start and end time of the sampler's movement along the jump path, taking `folding_zones` as a parameter, which contains a list of the polygon representation of the folding zone.

ALGORITHM 3: ALGORITHM TO GENERATE JUMP-TEXTURE FILE

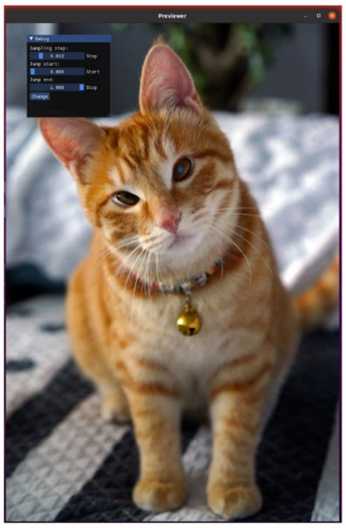
```
     Input: input images metadata, color texture and motion zone data, key jump vectors
     Output: jump-texture file $F$
1    Create output file $F$
2    for point $(x,y)$ contained in the motion zone's rectangular bounding box
3        $interpolated\_jump\_vec$ = interpolate ($x$, $y$, $key\_jump\_vecs$)
4        if folding_zones is specified then
5            $t_0$, $t_1$ = calculate_threshold ($interpolated\_jump\_vec$, $folding\_zones$)
6        else
7            $t_0 = 0, t_1 = 1$
8        end if
9        $jump\_vec$ = normalize ($interpolated\_jump\_vec$, $image\_data$, $offsets$)
10       $X = jump\_vec.x$, $Y = jump\_vec.y$
11       write $X$, $Y$, $t_0$, $t_1$ to file $F$
```

```
12  end for
13  return $F$
```

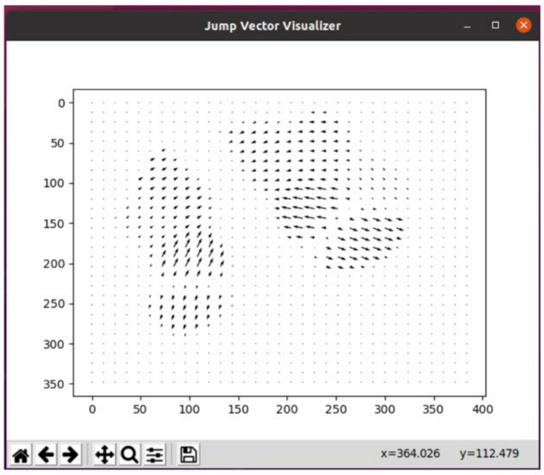
Thorben used an additional texture named function texture to store nonlinear position functions like exponential or sigmoid functions. We dropped the usage of function texture and have a discrete version of these functions available in the previewer instead. Currently the previewer only supports one function type for all motion zones. But this could be easily extended by adding one parameter for each motion zone.

5.3 Visualizer and Previewer

Figure 29a shows the integrated previewer, designed to facilitate user inspection of rendering outcomes of synthesized folding textures, and provide immediate feedback on the design. The integrated previewer, developed with OpenGL in C++, offers a debug menu where users can change parameters, such as position function, speed, and time offset, in real-time to adjust the rendering result.



Sampling step: 0.010 Step
Jump start: 0.000 Start
Jump end: 1.000 Stop



(a) (b)

Figure 29. Integrated previewer and visualizer. (a) simple previewer to render synthesized folding texture on a 2D geometry. (b) integrated visualizer to sample and visualize one motion zone's jump vectors as a vector field.

Additionally, an integrated visualizer (Figure 29b) displays the distribution of jump vectors in the form of a vector field. When dealing with high-resolution input images, displaying all jump vectors in one graph can be inconvenient. Users can set a down-sampling scale to visualize a selection of the jump vectors.

5.4 Misc

The software is developed following a set of open-source project best practices to maintain high code quality and ensure portability. These practices contribute to robust, reliable, and maintainable software, which have proven to be crucial to the program when the code base grows beyond a specific size (see Table 5.1).

Table 5.1 Repository Status

Category	Description	Lines of codes
Product codes	FoldingGen, Previewer and Visualizer	13522
Test codes	Tests guarding algorithms and interaction	3121
Documentation	User guide	743
Total		17386

Code quality

- Linting tools are set up to continuously guarantee that code is in a healthy state. pylint, pycodestyle, and mypy are used for Python code. Clang-tidy, Clang-format, and cpplint are used for C++ code.
- Use case level, module level, and unit tests are added as pre-commit tests to ensure software behaves as expected.
- Comprehensive user documentation that covers installation, configuration, and usage is provided.

Portability

- Currently, the system only runs on Linux (Ubuntu 20.04LTS) and is integrated with GIMP as the default image editing tool. However, from the beginning of implementation, the system's support for other platforms has been considered. The majority of the codebase is platform agonistic. Only some migration work is needed to run on Windows or OSX.
- Bazel is selected as a dependency management and build/test tool. Most of the dependent third-party libraries and modules are also managed in a hermetic way. The system only depends on a minimum set of packages installed on the system, making it easier to port and deploy.
- Tkinter is chosen as the GUI framework for quick prototyping due to its low learning curve. However, FoldingGen is implemented in the MVC design pattern, which makes it easier to switch to a different framework if needed.

6 Results

In this chapter, we first present the design process we used in practice. Then with a few examples, we showcase how we designed and synthesized folding texture with the proposed system, which in turn proves the validity of the proposed approach. The content is focused only on the specific use case, and a comprehensive user guide of the software can be found in the appendix section.

Rendering results are presented as a sprites sheet. Video captures can be found in the supplementary materials and on GitHub pages[1].

6.1 Design Process

From the applications and use cases we have explored, a general process can be abstracted and followed to create a desired visual effect, including but not limited to “folding”, “flying-over”, translation, etc. Most steps of this process adhere to the workflow proposed in Section 4.1.

Study the object

As with other creative processes, the first step is to study real-world examples to understand how different entities or parts of entities interact with each other. We recommend starting by gathering reference materials to guide the development of a design. Materials can include high-quality images, videos, or real-life objects that showcase various aspects of the subject. Consider various angles, lighting conditions, and environments to capture a wide range of possible appearances. These materials can be found through online searches, books, or firsthand observations.

During and after collecting reference materials, study the subject you aim to create or render. This might involve researching the appearance, structure, behavior, or other characteristics of the subject to gain a thorough understanding of its properties and nuances.

Create input images

Once you have studied the references and have a general target result in mind, you can start working within our framework. Draw from scratch or combine available works to prepare input images.

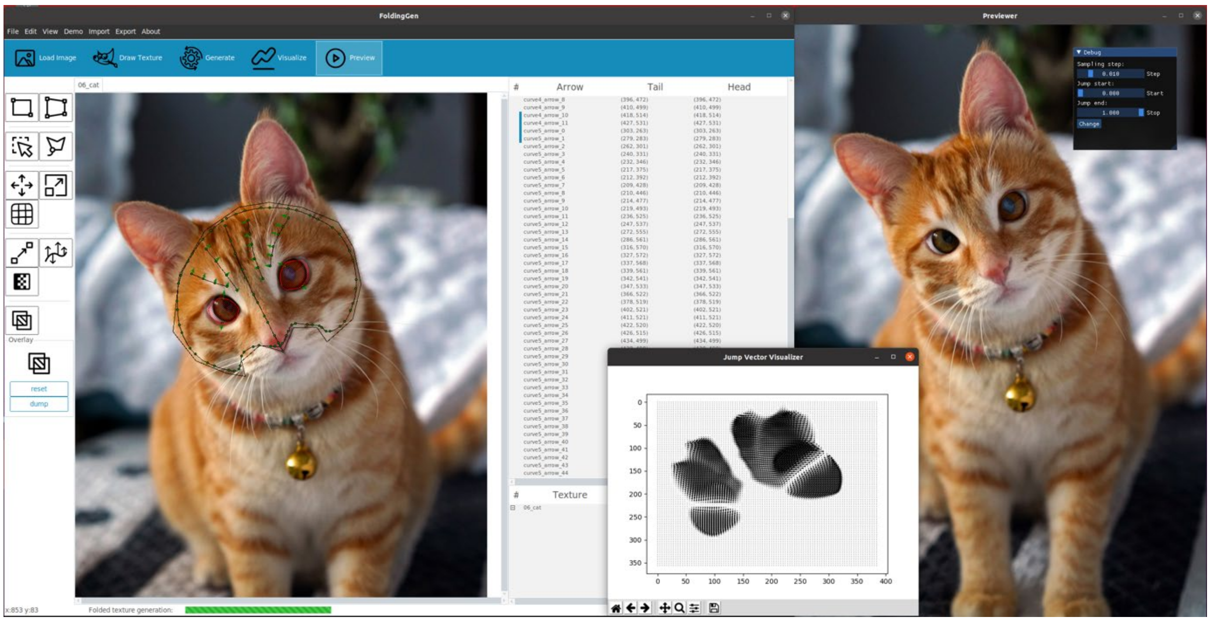
Motion Zone and Motion Profile Specification

Identify key motions observed in reference materials. Specify them in the form of motion zone and associated motion profiles, as explained in Section 4.2.

Preview as feedback

[1] https://mcoderh.github.io/foldingGen/

Analyze the result in the integrated previewer (shown in Figure 30). Use it as feedback and change configuration or input images accordingly in the next iteration.



#	Arrow	Tail	Head
	curve4_arrow_8	(396, 472)	(396, 472)
	curve4_arrow_9	(410, 499)	(410, 499)
	curve4_arrow_10	(418, 514)	(418, 514)
	curve4_arrow_11	(427, 531)	(427, 531)
	curve5_arrow_0	(303, 263)	(303, 263)
	curve5_arrow_1	(279, 283)	(279, 283)
	curve5_arrow_2	(262, 301)	(262, 301)
	curve5_arrow_3	(240, 331)	(240, 331)
	curve5_arrow_4	(232, 346)	(232, 346)
	curve5_arrow_5	(217, 375)	(217, 375)
	curve5_arrow_6	(212, 392)	(212, 392)
	curve5_arrow_7	(209, 428)	(209, 428)
	curve5_arrow_8	(210, 446)	(210, 446)
	curve5_arrow_9	(214, 477)	(214, 477)
	curve5_arrow_10	(219, 493)	(219, 493)
	curve5_arrow_11	(236, 525)	(236, 525)
	curve5_arrow_12	(247, 537)	(247, 537)
	curve5_arrow_13	(272, 555)	(272, 555)
	curve5_arrow_14	(286, 561)	(286, 561)
	curve5_arrow_15	(316, 570)	(316, 570)
	curve5_arrow_16	(327, 572)	(327, 572)
	curve5_arrow_17	(337, 568)	(337, 568)
	curve5_arrow_18	(339, 561)	(339, 561)
	curve5_arrow_19	(342, 541)	(342, 541)
	curve5_arrow_20	(347, 533)	(347, 533)
	curve5_arrow_21	(366, 522)	(366, 522)
	curve5_arrow_22	(378, 519)	(378, 519)
	curve5_arrow_23	(402, 521)	(402, 521)
	curve5_arrow_24	(411, 521)	(411, 521)
	curve5_arrow_25	(422, 520)	(422, 520)
	curve5_arrow_26	(426, 515)	(426, 515)
	curve5_arrow_27	(434, 499)	(434, 499)
	curve5_arrow_28		
	curve5_arrow_29		
	curve5_arrow_30		
	curve5_arrow_31		
	curve5_arrow_32		
	curve5_arrow_33		
	curve5_arrow_34		
	curve5_arrow_35		
	curve5_arrow_36		
	curve5_arrow_37		
	curve5_arrow_38		
	curve5_arrow_39		
	curve5_arrow_40		
	curve5_arrow_41		
	curve5_arrow_42		
	curve5_arrow_43		
	curve5_arrow_44		

#	Texture
⊟	06_cat

```
▼ Debug
Sampling step:
  0.010   Step
Jump start:
  0.000   Start
Jump end:
  1.000   Stop
Change
```

Figure 30. FoldingGen in action. Screenshots are taken when designing one folding texture.

6.2 Applications

6.2.1 Hand Knife-cut Wound

In this example, we aim to render a knife-cut wound on a hand. From the videos and photographs we have searched and reviewed; we made a few observations. When a wound occurs, the skin experiences displacement and deformation in different regions around the wound. In the immediate area of the wound, the skin may be pushed apart or separated, creating a visible opening. The skin around the wound may be stretched or compressed due to the tension created by the wound opening.

The background image (excluding the annotation on the image) in Figure 31 is used as the input image. Considering the observed motion, the motion zone should contain a region that includes not just the wound but also the areas where the skin deforms. So, regions A to E should at least be included in one of the motion zones. Region A is the place where "folding" will happen, meaning skin texture will progressively overlay region A in the texture dynamic sampling process. Regions B and D are perpendicular to the cut length. The skin in these two regions tends to be pushed apart or separated as a direct result of the cutting force. In regions C and E, along the length of the cut, the skin may experience less dramatic displacement compared to the area perpendicular to the cut.

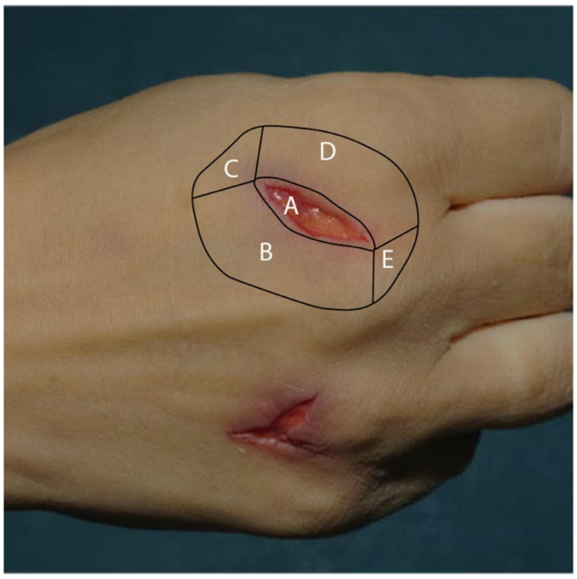


Figure 31. Hand-wound example input image. Different areas of an open wound are marked as regions A ~ E.

Based on the analysis, the motion zone and motion profile are specified in Figure 32a. Motion zone MZ covers an area of the hand where skin deformation can be observed. Right outside region A, two curves, C2 and C3, are drawn with the Curve Mapping tool to specify the most evident skin deformation. The exposed tissue in region A is selected as a folding zone since it is where folding will happen. An additional curve C1 is drawn to mark the boundary where deformation decreases to zero. The curve intentionally takes a path that is closer to the radially outer boundary of regions B and D and to the inner boundary of regions C and E, ensuring deformation is more visible in regions perpendicular to the cut length.

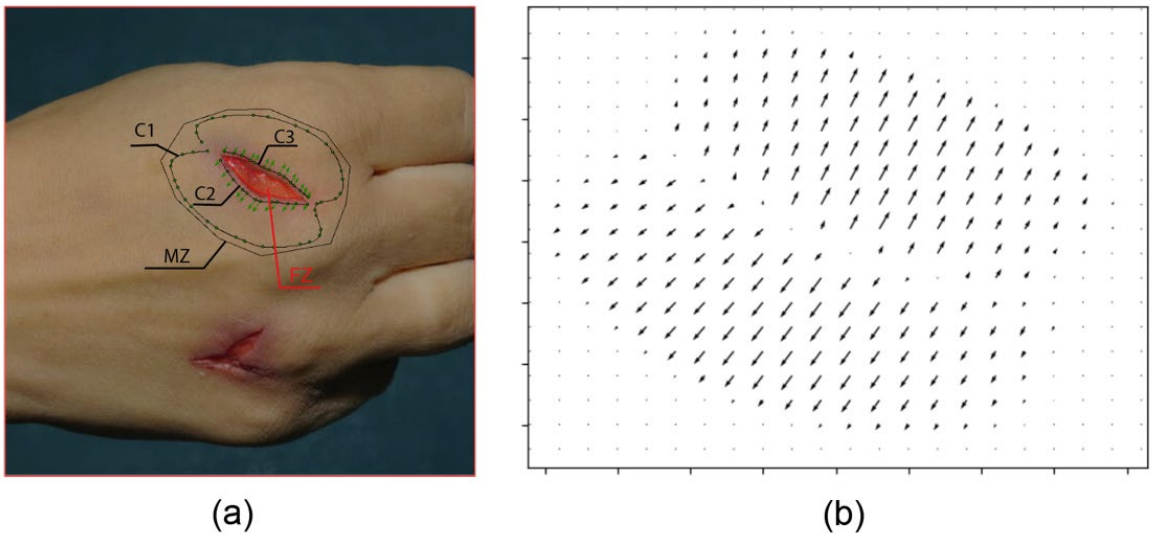


Figure 32. Motion profiles of hand-wound example. (a) annotated screenshot of the motion profile specified in FoldingGen. The motion zone is marked as MZ. The folding zone is marked as FZ. C1 to C3 are curves drawn with the Curve Mapping

tool. The green dots or arrows are anchor point jump vectors. (b) vector field visualization of the generated jump vectors.

The computed jump vectors are visualized as a vector field in Figure 32b. The vector field is displayed at a lower resolution to improve visibility. As shown, the wound center and curve C1 contain zero vectors, indicating a lack of motion at these pixels. In the direction perpendicular to the cut length, the jump vector's magnitude gradually increases and then decreases as we get further away from the wound center.

Skin is flexible in nature, which we emulate by using a sine function to control the motion. Using 62 anchor points (37 still) from 3 curves, one motion zone, and a sine position function with one threshold parameter, we achieve rendering results as in Figure 33.

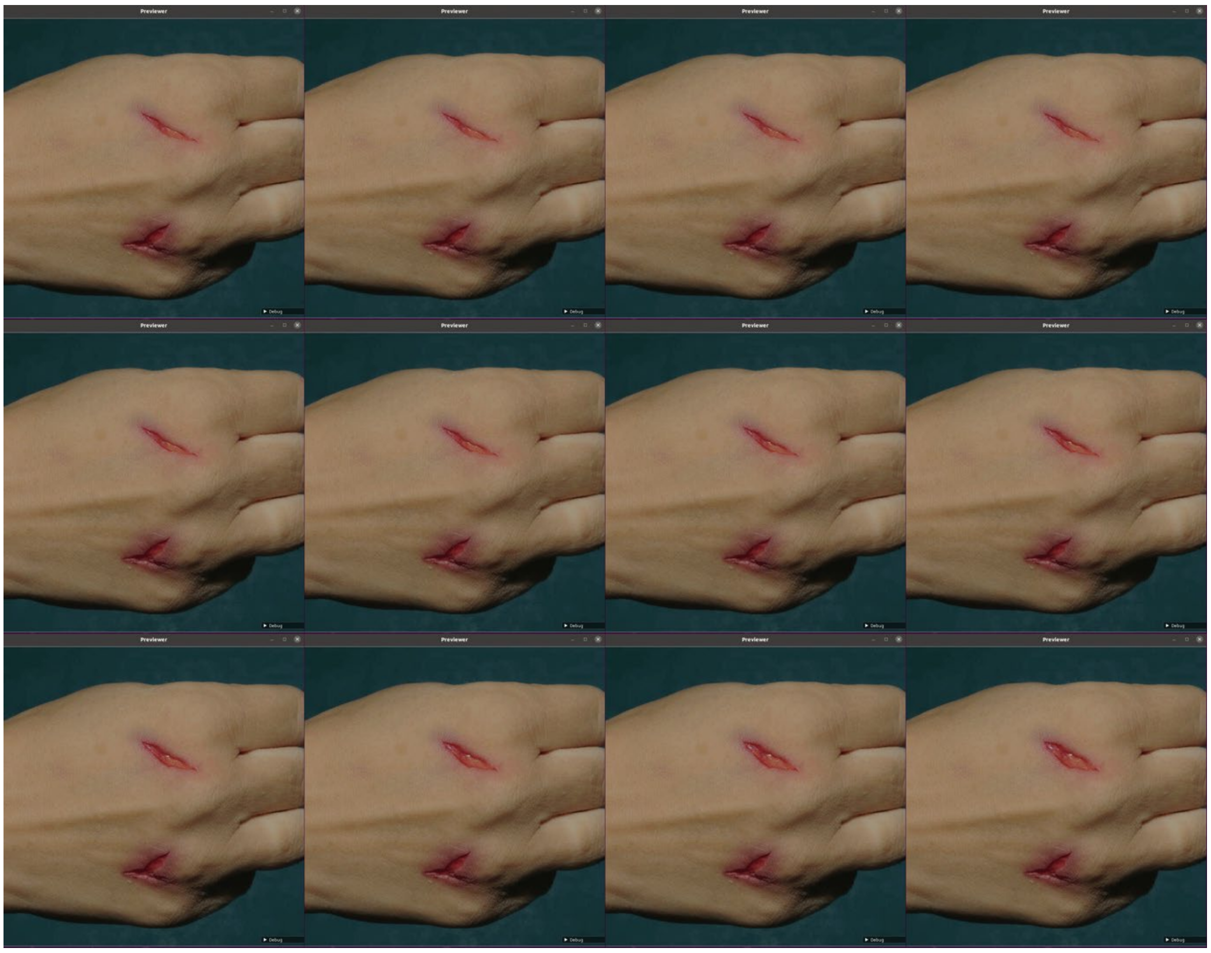


Figure 33. Hand Skin Cut Rendering Result.

6.2.2 Squinting Cat

From the reference material analyzed, a few observations have been made about cats' facial expressions when they squint or slowly blink. The most noticeable change occurs in the eyelids, with the upper eyelid lowering and the lower eyelid raising, resulting in partially closed eyes. This creates a squinting appearance. The muscles around the eyes, cheeks, and forehand also relax, displaying subtle changes.

As shown in Figure 34 most of the cat's face should be included in a single motion zone, excluding the mouth area where no motion occurs. Region A circles the area around the eyes and cheeks, where the most evident motion will happen. Region D is fully contained inside Region A. This is the region in which the "folding" happens: the eyelid will progressively overlay the eyeball. In the forehead region B, the cat's facial muscle relaxes and extends the fur from the center to both sides, while fur in region C along the face's central line stays still due to the face's symmetric nature.

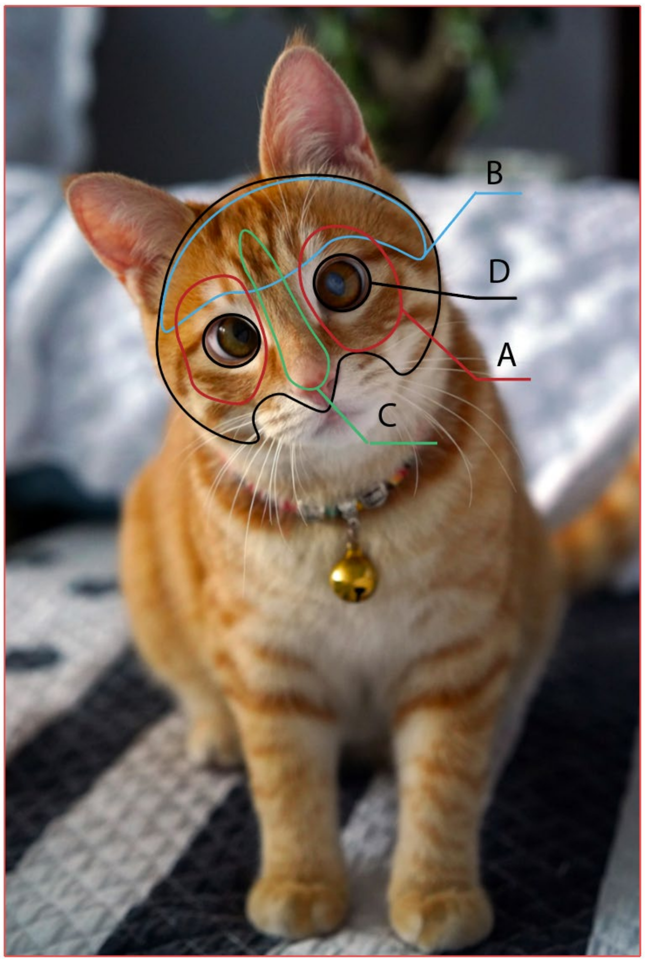


Figure 34. Squinting cat example input image. Representative areas on the cat's face are marked as regions A~D.

Based on this analysis, the motion zone and motion profile are specified in Figure 35. Motion zone MZ covers the cat's face where motion is observed. Close to MZ's boundary, curve C6 indicates where motion diminishes to zero. In region B, four curves C1 through C4 are drawn using Curve Mapping Tool to specify the muscle movement on the forehead. On the upper and lower eyelid, the Point Mapping tool is used to specify movements in the eye and cheek area. Both eyes are marked as folding zone and an additional curve C5 is drawn to mark the still line along the face's symmetric axis.

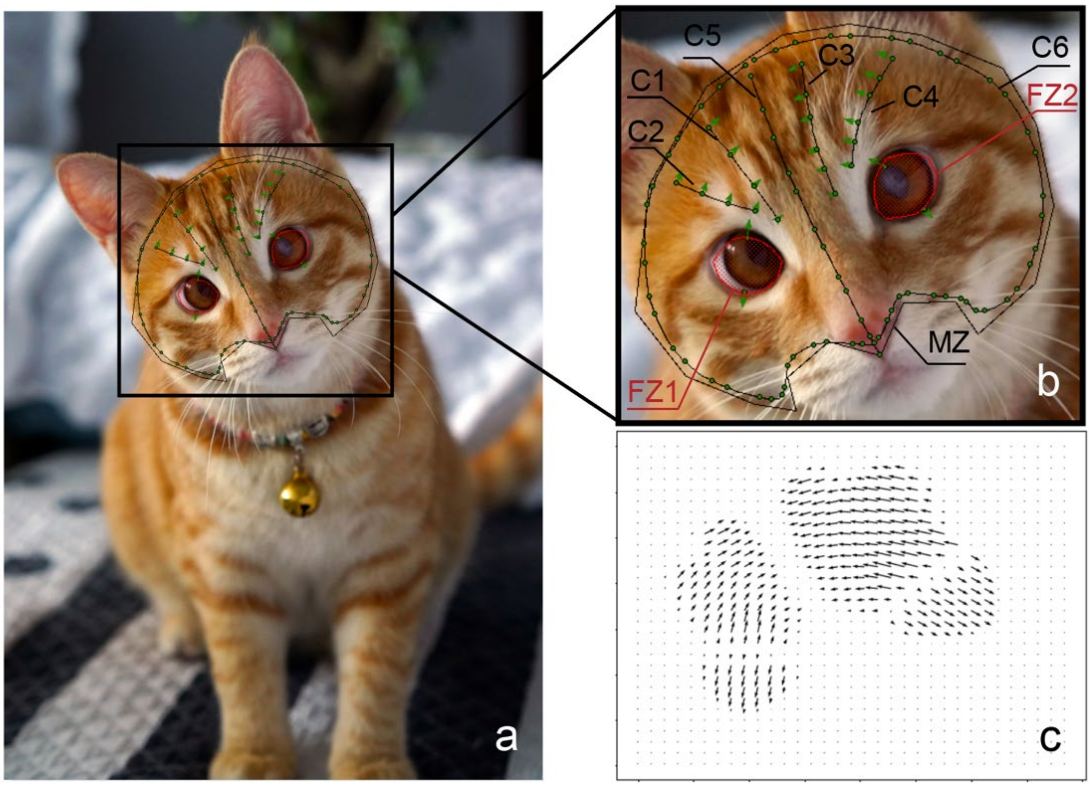


Figure 35. Motion Profile of squinting cat example. (a) screenshot of motion profile specified in FoldingGen (b) zoom-in view of the cat's face area (c) visualizer output

Figure 35c visualizes the computed jump vectors as a vector field. The distribution of the computed jump vectors aligns with the intended motions. Similar to the hand-wound example, we use a piecewise function with linear and sine sub-functions. With 94(68 of them marks no motion) sample jump paths in the motion zone, visually convincing rendering results are achieved, as shown in Figure 36.

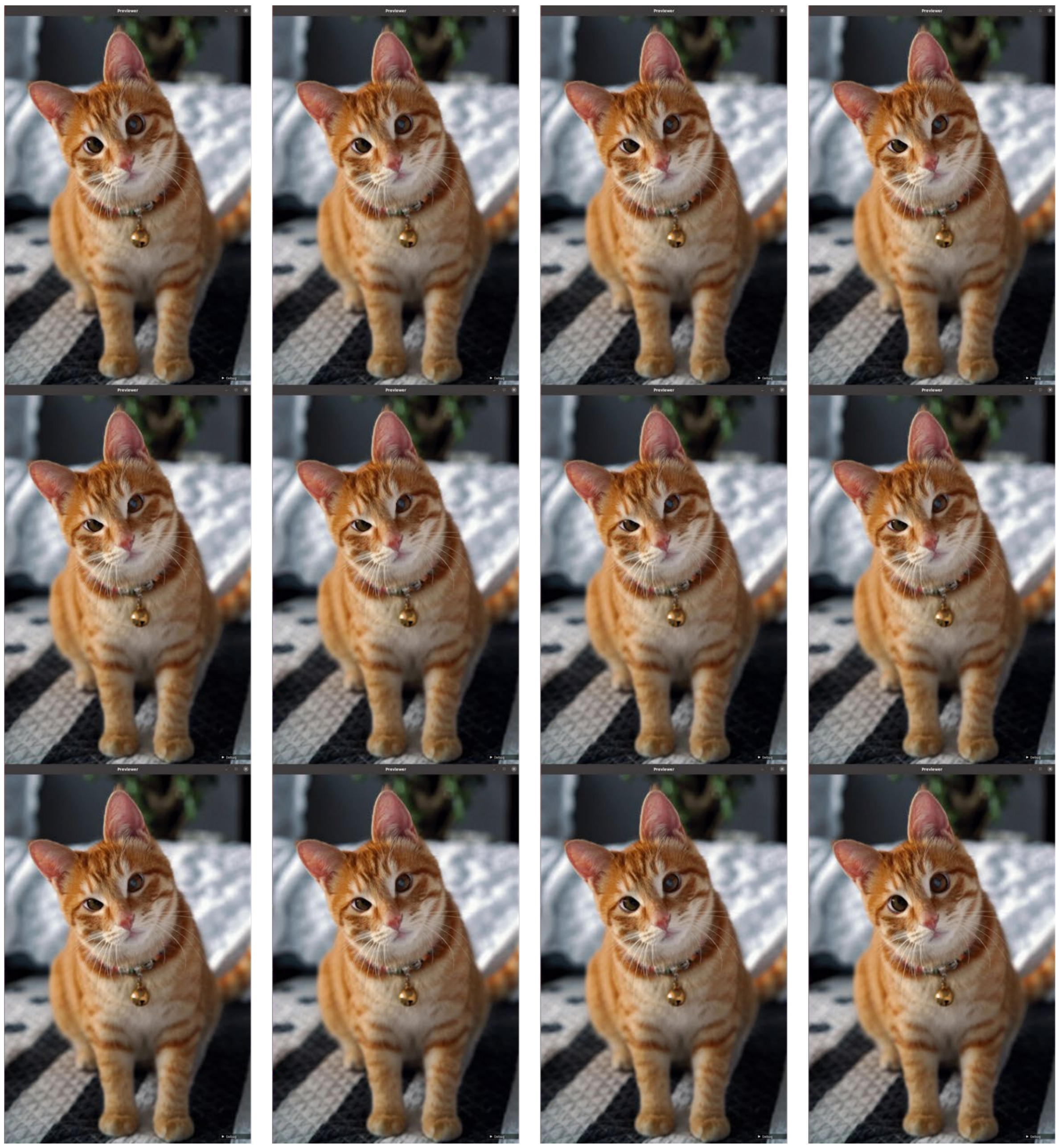


Figure 36. Squinting cat example rendering result.

6.2.3 Bouncing Pecs

The term "bouncing pecs" refers to the voluntary contraction and relaxation of the chest muscles, which creates a visible movement in the chest area. This rapid change in muscle tension creates the visual effect of the chest muscles bouncing.

When the clavicular head of the pectoralis major contracts, it pulls the pectoralis major upward, causing a rising movement of the chest area. As the clavicular head relaxes, the pectoralis major falls to its natural position, causing a dropping movement of the chest area and completing one “bounce”.

A few observations are made in reviewing searched reference videos. As illustrated in Figure 37 the motion zone should cover regions A through C. The most evident muscle displacements are observed in region B. In region C, motion gradually decreases to zero. A and B are separated by the muscle’s lower boundary, where we see the most evident “bouncing” effect. At the same time, most parts of region A are “fixed”, with no motion visible. Region A is crucial to achieving the “bouncing” effect. Thus the motionless region is included in the specified motion zone.

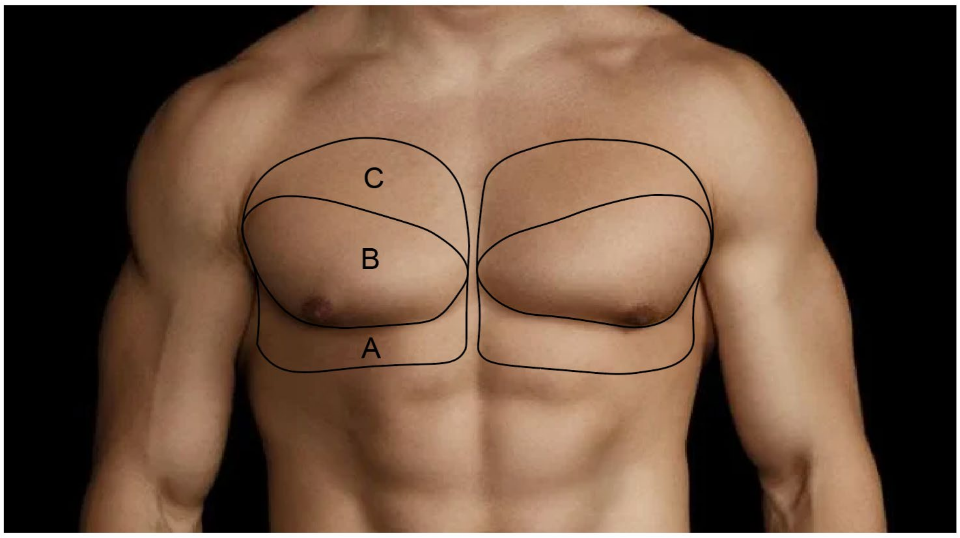


Figure 37. Bouncing Pecs Example Input Image.

Based on the analysis, the motion zone and motion profile are specified in Figure 38a. Two motion zones each cover half of the chest and both cover the regions that contribute to the bouncing motion. In region B, a few relatively longer jump paths are drawn with the Point Mapping tool along the muscle fiber direction, aiming to generate evident displacement. In regions A and C, a few still sample points are added to mark where motion terminates. Along the muscle lower boundary between regions A and B, a few relatively small jump paths are added to ensure motion diminishes rapidly around this boundary, thus achieving the bouncing effect.

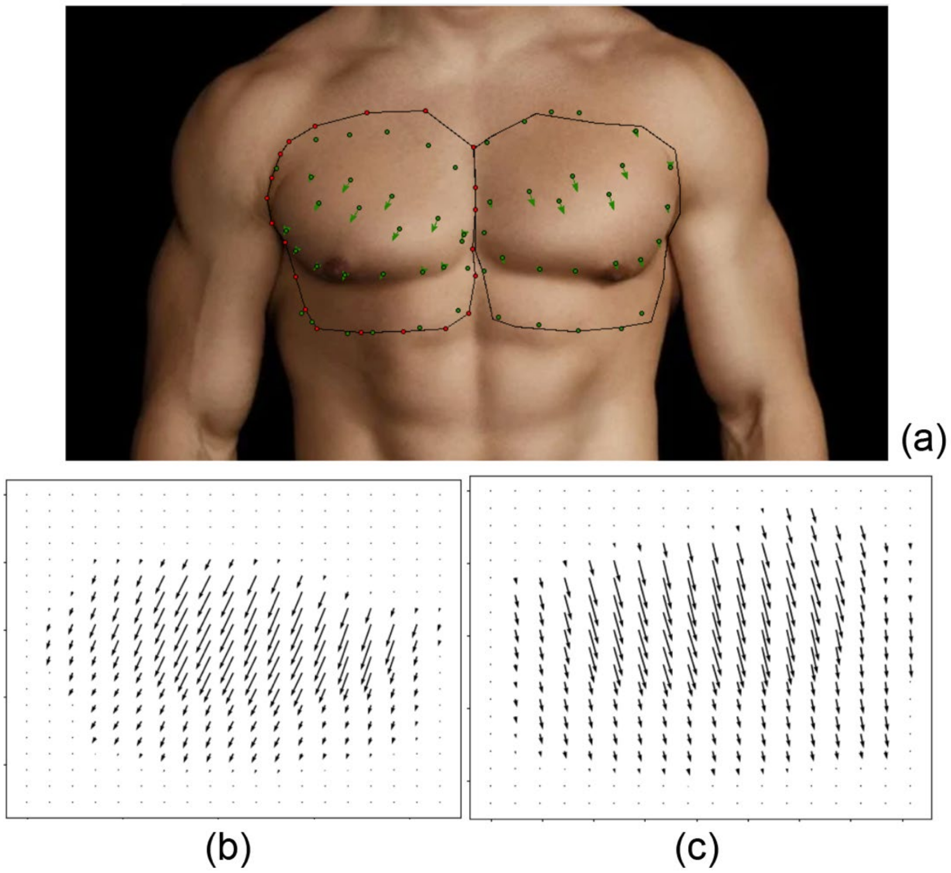


Figure 38. Motion profiles of bouncing pecks example. Screenshots of motion profile specified in FoldingGen (b) and (c) vector field visualization of jump vectors in left and right motion zones, respectively.

As visualized in Figure 38b, the computed jump vectors' distribution aligns with our intentions. Similar to the hand-wound example, we use a piecewise function with linear and sine sub-functions. With 28(13 still) sample jump paths in each motion zone, we achieve a visually convincing rendering result in Figure 39.

Figure 39. Bouncing Pecs rendering result.

Unfortunately, the sprite sheet generated from the recording cannot effectively visualize the results. You may review the supplementary material or access it directly online for a more vivid representation.

6.3 Summary

By presenting the design process and rendering results of the hand-wound, squinting-cat, and bouncing-pecs examples, we proved the validity of our proposed approach. We have tried several other examples, you may find them on our GitHub page. Normally, we would ask people to try our method, but the implementation work was already proven to be substantial and time constraints was tight. We decided to leave the user study out of this thesis work’s scope.

We used our approach to reproduce the examples used in Thorben’s thesis. All the examples can be reproduced between 5 minutes and 45 minutes. Unfortunately, Thorben did not record how much time he took to manually create those examples. I tried to manually create a few of them myself. I found that users need to consider so many small details at the same time while creating folding textures. It took me hours to adjust and fine-tune one texture. It is worth mentioning that I was already quite familiar with the folding-texture technique and we already knew a lot about how to use it. I would expect a new user would feel way more challenged than me. So according to our firsthand experience, we did observe improvements in efficiency and usability.

7 Conclusions

In conclusion, we addressed the problem of designing and synthesizing folding textures. We have presented a semi-automatic user-assisted approach that combines texture editing, motion profile specification, and texture synthesis into one seamless process, making the folding-texture technique more accessible. As proof of concept, we have implemented a ready-to-use software tool. We have successfully demonstrated the validity of our proposed approach through several examples using the implemented software.

There are multiple directions for future research:

Common motion profile abstraction: Many phenomena share similar motion fields and have been studied extensively. By leveraging the existing models and expressing them in the form of folding texture representation, we can provide motion profile templates in FoldingGen, thus enhancing design efficiency.

Automatic reference material analysis: Currently, users need to learn and identify motion profiles from reference materials. It would be beneficial if we could apply computer vision techniques, such as optical flow, to automatically analyze sample videos to generate jump vector fields and present them to the user as references.

User interface improvement: Additionally, refining and optimizing the software tool could lead to an even more seamless and efficient user experience, making it even more accessible to designers and artists.

Appendix A

FoldingGen User Guide

Available at https://mcoderh.github.io/foldingGen/user_guide.html

Bibliography

[1] S. Thorben, “Folding Texture,” Master Thesis, Technische Universität Braunschweig, 2012.

[2] J. M. Kessenich, G. Sellers, and D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, version 4.5 with SPIR-V*, Ninth edition. Boston, MA: Addison-Wesley, 2017.

[3] P. S. Heckbert, “Survey of texture mapping,” *IEEE Comput. Graph. Appl.*, vol. 6, no. 11, pp. 56–67, 1986.

[4] J. F. Blinn and M. E. Newell, “Texture and Reflection in Computer Generated Images,” *Commun. ACM*, vol. 19, no. 10, pp. 542–547, Oct. 1976, doi: 10.1145/360349.360353.

[5] J. Cohen, M. Olano, and D. Manocha, “Appearance-preserving simplification,” in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 1998, pp. 115–122.

[6] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno, “A general method for preserving attribute values on simplified meshes,” in *Proceedings Visualization’98 (Cat. No. 98CB36276)*, IEEE, 1998, pp. 59–66.

[7] R. C. Nelson and R. Polana, “Qualitative recognition of motion using temporal texture,” *CVGIP Image Underst.*, vol. 56, no. 1, pp. 78–89, 1992.

[8] M. O. Szummer, “Temporal texture modeling,” Master Thesis, Massachusetts Institute of Technology, USA, 1995.

[9] M. Szummer and R. W. Picard, “Temporal texture modeling,” in *Proceedings of 3rd IEEE International Conference on Image Processing*, Sep. 1996, pp. 823–826 vol.3. doi: 10.1109/ICIP.1996.560871.

[10] L.-Y. Wei, S. Lefebvre, V. Kwatra, and G. Turk, “State of the Art in Example-based Texture Synthesis,” presented at the Eurographics 2009, State of the Art Report, EG-STAR, Eurographics Association, Mar. 2009, p. 93. Accessed: Mar. 20, 2023. [Online]. Available: https://hal.inria.fr/inria-00606853

[11] S. Soatto, G. Doretto, and Y. N. Wu, “Dynamic textures,” in *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, Jul. 2001, pp. 439–446 vol.2. doi: 10.1109/ICCV.2001.937658.

[12] P. Saisan, G. Doretto, Y. N. Wu, and S. Soatto, “Dynamic texture recognition,” in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, IEEE, 2001, p. II–II.

[13] R. Péteri, S. Fazekas, and M. J. Huiskes, “DynTex: A comprehensive database of dynamic textures,” *Pattern Recognit. Lett.*, vol. 31, no. 12, pp. 1627–1632, 2010.

[14] J. Rhoades, G. Turk, A. Bell, A. State, U. Neumann, and A. Varshney, “Real-time procedural textures,” in *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, 1992, pp. 95–100.

[15] A. Schödl, R. Szeliski, D. H. Salesin, and I. Essa, “Video textures,” in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, in SIGGRAPH ’00. USA: ACM Press/Addison-Wesley Publishing Co., Jul. 2000, pp. 489–498. doi: 10.1145/344779.345012.

[16] T. Brooks *et al.*, “Generating Long Videos of Dynamic Scenes.” arXiv, Jun. 09, 2022. doi: 10.48550/arXiv.2206.03429.

[17] Y. Li, T. Wang, and H.-Y. Shum, “Motion texture: a two-level statistical model for character motion synthesis,” in *Proceedings of the 29th annual conference on Computer Graphics and Interactive Techniques*, in SIGGRAPH ’02. New York, NY,

USA: Association for Computing Machinery, Jul. 2002, pp. 465–472. doi: 10.1145/566570.566604.

[18] S. Enrique, M. Koudelka, P. Belhumeur, J. Dorsey, S. Nayar, and R. Ramamoorthi, “Time-Varying Textures,” Jan. 2005.

[19] Y.-Y. Chuang, D. B. Goldman, K. C. Zheng, B. Curless, D. H. Salesin, and R. Szeliski, “Animating pictures with stochastic motion textures,” in *ACM SIGGRAPH 2005 Papers*, in SIGGRAPH ’05. New York, NY, USA: Association for Computing Machinery, Jul. 2005, pp. 853–860. doi: 10.1145/1186822.1073273.

[20] A. Holynski, B. L. Curless, S. M. Seitz, and R. Szeliski, “Animating Pictures With Eulerian Motion Fields,” presented at the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 5810–5819.

[21] R. H. Kazi, F. Chevalier, T. Grossman, S. Zhao, and G. Fitzmaurice, “Draco: bringing life to illustrations with kinetic textures,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, in CHI ’14. New York, NY, USA: Association for Computing Machinery, Apr. 2014, pp. 351–360. doi: 10.1145/2556288.2556987.

[22] R. H. Kazi, F. Chevalier, T. Grossman, and G. Fitzmaurice, “Kitty: sketching dynamic and interactive illustrations,” in *Proceedings of the 27th annual ACM symposium on User interface software and technology*, in UIST ’14. New York, NY, USA: Association for Computing Machinery, Oct. 2014, pp. 395–405. doi: 10.1145/2642918.2647375.

[23] D. Shepard, “A two-dimensional interpolation function for irregularly-spaced data,” in *Proceedings of the 1968 23rd ACM national conference*, 1968, pp. 517–524.

[24] B. Baxter, “The Interpolation Theory of Radial Basis Functions,” Ph.D Thesis, University of Cambridge, 1992.

[25] S. McKinley and M. Levine, “Cubic spline interpolation,” *Coll. Redw.*, vol. 45, no. 1, pp. 1049–1060, 1998.

[26] M. Alexa, D. Cohen-Or, and D. Levin, “As-rigid-as-possible shape interpolation,” in *Proceedings of the 27th annual conference on Computer Graphics and Interactive Techniques*, 2000, pp. 157–164.

[27] R. Franke and G. Nielson, “Smooth interpolation of large sets of scattered data,” *Int. J. Numer. Methods Eng.*, vol. 15, no. 11, pp. 1691–1704, 1980.

[28] A. Orzan, “Contour-based Images: Representation, Creation and Manipulation,” Ph.D Thesis, Institut National Polytechnique de Grenoble, 2009. [Online]. Available: https://theses.hal.science/tel-00528871