Master's Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

3D Modeling From Rough Vector Sketches

Yanina Arameleva

Supervisor: prof. Ing. Daniel Sýkora, Ph.D.
Field of study: Open Informatics
Subfield: Computer Graphics
May 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Arameleva Yanina**	Personal ID number:	**483752**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Graphics**		

II. Master’s thesis details

Master’s thesis title in English:

3D Modeling From Rough Vector Sketches

Master’s thesis title in Czech:

3D modelování z hrubých vektorových skic

Guidelines:

Get familiar with the Monster Mash method [1] for 3D modelling from hand-drawn sketches. Extend its user interface to allow the user to freely draw a vector sketch without the need to create continuous and closed regions using a single stroke. Examine algorithms that can simplify rough line drawings into a set of smooth continuous and closed contours [2, 3, 4, 5] and implement one of them. Furthermore, use a deformation model originally developed for 3D meshes [1] to deform simplified 2D curves so that the user can interactively modify their shape at the stage of sketching. Try to integrate the resulting implementation into the existing user interface of the Monster Mash method [1]. Evaluate practical utility of the implemented extension on a variety of hand-drawn sketches.

Bibliography / sources:

[1] Dvorožňák et al.: Monster Mash: A Single-View Approach to Casual 3D Modeling and Animation, ACM Transactions on Graphics 39(6):214, 2020.
[2] Noris et al.: Smart Scribbles for Sketch Segmentation, Computer Graphics Forum 31(8):2516–2527, 2012.
[3] Liu et al.: Closure-aware Sketch Simplification, ACM Transactions on Graphics 34(6):168, 2015.
[4] Liu et al.: StrokeAggregator: Consolidating Raw Sketches Into Artist-Intended Curve Drawings, ACM Transactions on Graphics 37(4):97, 2018.
[5] Mossel et al.: StrokeStrip: Joint Parameterization and Fitting of Stroke Clusters, ACM Transactions on Graphics 40(4):50, 2021.

Name and workplace of master’s thesis supervisor:

prof. Ing. Daniel Sýkora, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **20.02.2023** Deadline for master's thesis submission: **26.05.2023**

Assignment valid until: **16.02.2025**

prof. Ing. Daniel Sýkora, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to express my profound gratitude to my English language tutor, Katherine Traylor, for her invaluable assistance in correcting the language errors in this thesis. My heartfelt thanks go to Marek Dvorožňák, who not only provided me with a simplified version of Monster Mash but also generously helped me understand the code. I cannot overlook the constant guidance and support from my supervisor, Daniel Sýkora, throughout the journey of this thesis. Lastly, I am immensely grateful to all those close to me who offered immeasurable moral support during this process.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guidelines for adhering to ethical principles when elaborating an academic final thesis.

Abstract

This thesis explores the enhancement of the Monster Mash digital drawing tool, focusing on incorporating a sketch simplification algorithm and the As-Rigid-As-Possible (ARAP) deformation algorithm. Based on image segmentation, the simplification algorithm streamlines the drawing process, allowing users to sketch and rapidly simplify their artwork. The ARAP deformation algorithm provides a powerful tool for modifying sketches without extensive redrawing, accelerating the creative process. These additions aim to make digital sketching more enjoyable, efficient, and accessible to a wider range of users.

Keywords: Digital Sketching, Sketch Simplification, As-Rigid-As-Possible (ARAP), Deformation Algorithm, Monster Mash, Image Segmentation, Drawing Tools

Supervisor: prof. Ing. Daniel Sýkora, Ph.D.

Abstrakt

Tato práce se zabývá vylepšením digitálního kreslícího nástroje Monster Mash, s důrazem na začlenění algoritmu pro zjednodušení skic a algoritmu pro deformaci As-Rigid-As-Possible (ARAP). Algoritmus pro zjednodušení skic, založený na segmentaci obrazu, zefektivňuje kreslící proces, čímž umožňuje uživatelům volně skicovat a rychle zjednodušovat svá umělecká díla. ARAP deformace poskytuje mocný nástroj pro úpravy skic bez nutnosti rozsáhlého překreslování, čímž urychluje tvůrčí proces. Tyto přídavky mají za cíl učinit digitální skicování zábavnější, efektivnější a přístupnější širšímu spektru uživatelů.

Klíčová slova: Digitální Skicování, Zjednodušení Skic, As-Rigid-As-Possible (ARAP), Deformační Algoritmus, Monster Mash, Segmentace Obrazu, Kreslící Nástroje

Contents

Figures

Chapter 1

Introduction

1.1 Motivation

Motivation for this thesis comes from my passion for drawing of all kinds and desire to make the process more accessible and enjoyable for users. As avid artist, I understand the frustrations that can arise when using digital drawing tools that are clunky, unintuitive, or lack essential features. I believe that by improving an existing digital drawing tool, we can contribute to the advancement of the digital art industry and help users create their art more efficiently and effectively.

1.2 Thesis Objectives

Sketching is a fundamental tool for artists and designers, enabling them to quickly visualize their ideas and explore different possibilities for their artwork. This research aims to develop an environment for sketching that can be integrated into the framework for sketch-based modeling and animation of 3D shapes, such as the Monster Mash. The objective is to create a sketching tool that is easy and fast to use, but also includes basic drawing tools.

While digital drawing has gained popularity, redrawing sketches can still be a time-consuming task. Despite the benefits of digital drawing, the lack of tools for sketch simplification can make the process more challenging. Hence, there is much room for improvement in this area. By finding an algorithm for simplifying sketches, this research can potentially enhance the time spent drawing.

The second objective is to design and create a graphic tool that can deform 2D sketches. This functionality significantly reduces the time required to make edits, whether big or small, without the need to redraw, thereby increasing productivity. Creating a deformation tool that is user-friendly and easy to learn will make it accessible to a wider range of users, from novice designers to seasoned professionals.

1.3 Structure of the Thesis

The structure of this thesis is arranged in such a way to progressively explore the issues at hand, the approach we adopted, and the resulting outcomes. It is organized into three primary sections, each focusing on a key aspect of the study.

The initial section of the thesis lays down the foundation by introducing the motivation, objective, and need for this research. Following this, the concept of Monster Mash as a digital sketching tool is explored, discussing its model creation process and defining objectives for its enhancement to improve the user experience.

In the second section, encapsulated by chapter 3 and chapter 4, we comprehensively review the existing techniques used for image simplification and deformation. This part is dedicated to designing enhancements for Monster Mash, guided by the insights obtained from the research papers we review.

The third and final section of the thesis, presented in chapter 5 and chapter 6, is dedicated to the implementation and evaluation of the proposed enhancements. chapter 5 takes us through the journey from the beginning of the first version of the drawing part of Monster Mash to its final enhanced form integrated into Monster Mash. It provides an in-depth examination of the coding structure, data structures used, program pipeline, and the challenges that were encountered and subsequently overcome. chapter 6 puts forth a series of experiments to validate the implemented enhancements' effectiveness. This section tests the theoretical and designed concepts in a practical setting, demonstrating their applicability and impact on the user experience in a real-world context.

Chapter 2

Monster Mash

The primary focus of this research is Monster Mash [1], a framework designed for creating and animating 3D shapes through sketching, which offers a more intuitive experience via its 2D interface. In contrast to other sketch-based tools, Monster Mash eliminates the need for a time-consuming 3D modeling workflow that requires explicit rig specifications. The framework combines 3D inflation with a rigidity-preserving approach to generate a smooth 3D mesh that can be animated from a single viewpoint. This method provides a straightforward modeling and animation experience for inexperienced users while also delivering a quick and efficient workspace for professionals.

Monster Mash is divided into three main components: Draw, Inflate, and Animate.

The Draw section of Monster Mash enables users to create their art from scratch by drawing lines. Each stroke represents one shape as well as one layer. Closed region represents characters main body. Users can adjust layer depth, with the last layer being the closest to the user. To adjust layer depth user selects the layer and then press PageUp or PageDown key to put layer above/below neighboring layer. The ordering of layers represents the depth order of the shapes. Unclosed shapes are automatically closed by a closing line. If the closing line intersects the boundary of another shape, it merges the two shapes at the location of the closing line. If there is no layer above or below the closing line, it is treated as a free open boundary, resulting in a hole in the object at that location. In this mode, there is a feature that allows the user to redraw a selected layer. Once the layer is selected, the user can create new strokes, and the selected layer will be replaced with the newly drawn strokes.

In the Inflate section, users can add depth and dimension to their character by inflating the 2D shape into a 3D form. Users can rotate or pan the 3D object as needed.

The Animate section allows users to bring their characters to life by adding movements and expressions. Animation is created by placing control pins on parts of the 3D mesh and moving them around. Users can also move objects without control pins by dragging them. Animation recording and exporting are possible as well.

Figure 2.1 presents an illustration of the drawing part within the Monster Mash program.

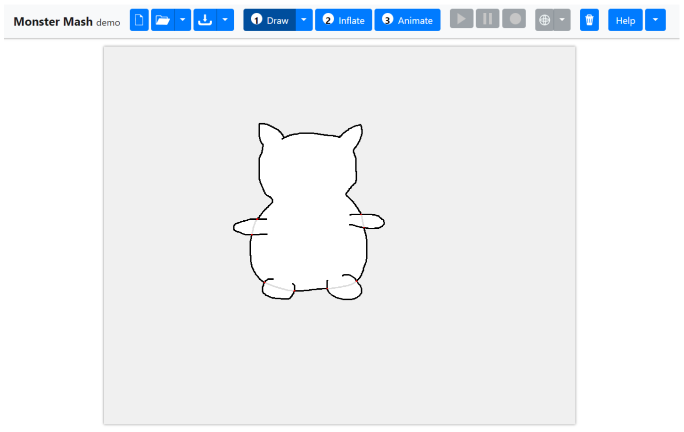


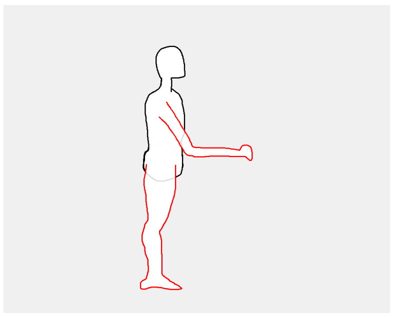
Figure 2.1: Illustration of UI of Monster Mash

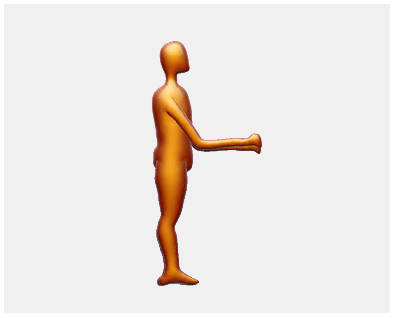
Below, in Figure 2.2, a screenshot of all parts of MM is presented. The red regions in Figure Figure 2.2a represent duplicated regions. The curves in Figures Figure 2.2c and Figure 2.2d are animation curves dictating the character's movement. These illustrations demonstrate the complete process, from character creation to the animation of a drawn character.

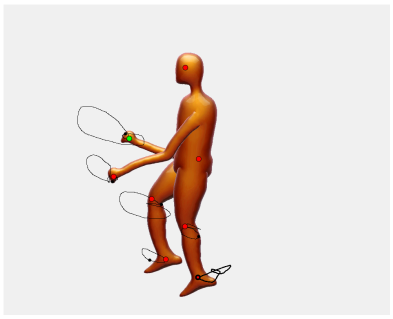
2.1 Model Creation

The construction of the 3D mesh begins in the inflation mode after the sketch is drawn. It starts with creating a "flat" mesh, followed by inflation. Each body part of the model is treated as a separate region (layer). Vertices are added to each region, and Delaunay triangulation [2] is performed to create triangles from the vertices. If one region has an open boundary and another lies above or below it, they are stitched together. A hole is created in the closed boundary region and connected to the second region with an open boundary.

Once the 3D mesh is created, the next step is to inflate it. This process entails generating a height field for each vertex in the mesh, determining the extrusion amount in the z-axis to form the desired 3D shape. The height value of zero is assigned to all vertices along the user-drawn boundaries, as they will serve as the base of the 3D model. For vertices not located along the boundaries, a Poisson equation calculates the height values based on the surrounding vertices.


(a) : Character drawn in drawing part of MM.


(b) : Converted sketch into a 3D model in the inflation part of MM.


(c) : Animated character in animation part of MM.

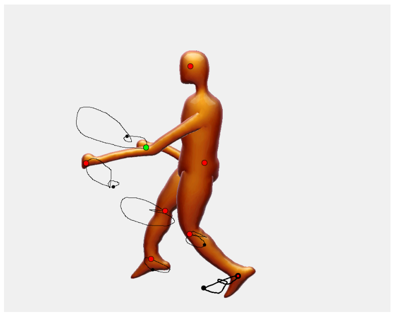

(d) : Animated character in animation part of MM.

Figure 2.2: Examples of character creation in Monster Mash.

For the final stage of the inflation process and for animation, As-Rigid-As-Possible layer preserving deformation (ARAP-L) is employed. It is used to satisfy the depth ordering and positional constraints of the vertices. In contrast to the ARAP method described in section 4.2, ARAP-L takes into account the depth ordering of vertices. It prevents vertices from deeper layers from having higher z coordinates than layers above them and vice versa.

2.2 Enhancement Objectives for Monster Mash

The central aspiration of this thesis revolves around enhancing the user interaction experience with the drawing part of the software application, Monster Mash. The present user interface, albeit functional, offers a rather restrictive environment for users, particularly in terms of the creative freedom typically offered by conventional drawing software.

The current drawing version presented possesses certain limits for the user. As was described above, presently, the Monster Mash allows the user to draw a whole shape by just one stroke. This can be very hard to accomplish because of various reasons. It requires a lot of precision and control to draw the entire shape in a single stroke accurately. Any minor deviation can alter the final shape significantly. The user needs to maintain a steady hand throughout the entire stroke. Shaking, trembling, or involuntary movements can distort the intended design. Drawing a shape in one stroke can be physically tiring. Holding a pencil or brush and maintaining a specific posture for extended periods can result in hand cramping or fatigue. If the shape is complex, capturing all of the details in one stroke can be incredibly difficult. Simpler shapes might be feasible, but more intricate shapes are likely to require multiple strokes to depict them accurately. In order to execute a single-stroke drawing, the user needs to carefully plan their path so that they can complete the drawing without lifting the pen. This requires foresight and the ability to visualize the completed drawing in mind before even starting to draw. If a mistake is made, it is difficult, if not impossible, to correct it without breaking the single-stroke rule. This adds a level of stress that can negatively affect the quality of the drawing.

The program appears to be relatively complex to use. The features provided are not commonly seen in other programs, so learning to use all these features effectively might pose a challenge. The ability to redraw layers by replacing old strokes with new ones could be limiting, particularly if the user wants to retain the old stroke and make minor changes. The current version does not permit mistakes during sketching. This implementation also does not allow users to create objects with holes. Lastly, the Monster Mash drawing interface lacks a comprehensive set of tools that are standard in most drawing software.

The primary objective is thus the augmentation of the application with tools that can address the current limitations of Monster Mash. In this work, we aim to add features that will make the process of sketch creation less stressful. To achieve this, we need to allow users to draw freely, using more strokes as typical drawing software does. We also plan to allow users to erase strokes, a necessary feature that enables them to rectify their mistakes easily. However, this freedom could complicate the subsequent inflation phase of Monster Mash. Therefore, we must introduce a feature that prepares the drawn sketch for the next phase to allow users to draw freely. This entails simplifying the sketch before passing it on to the inflation phase.

To make the user experience even more flexible, we will add a feature that could deform drawn shapes. We believe that this feature can solve problems in already drawn sketches without the need to redraw the shape. This new functionality is expected to provide users with a higher degree of control over their artistic creations. By enabling effortless modifications and manipulations of sketches, the user experience with the application can be significantly enriched.

In summary, this work seeks to tackle the limitations of Monster Mash's current drawing interface by proposing new UI functions. The overarching goal is to provide a more intuitive and flexible interface, thus fostering creativity and enhancing overall user satisfaction.

Chapter 3

Sketch Simplification

Our primary goal in this section is to develop or find an algorithm capable of simplifying sketches, a process that effectively prepares them for the inflation part of MM. Simultaneously, we strive to eliminate any necessity for user intervention during this simplification procedure. The defining metrics for the preferred algorithm are as follows: speed, precision, and regional closure.

Speed is essential, as users cannot be expected to endure protracted waiting periods for the simplification of their sketches. Precision is equally crucial; we aim to avoid any oversimplification resulting from our algorithm's execution. Lastly, regional closure is necessary; the inflation process demands a region devoid of discontinuities.

Initially, we shall explore various image simplification techniques with potential applicability to our project. Subsequently, we will delve into a detailed description of our selected method.

3.1 Related Work

This section delves into algorithms designed for sketch simplification. The goal is to identify a method that can rapidly simplify an image. The outcome of the simplification process should ideally be a closed shape or a collection of closed shape objects. Alternatively, the result should be able to be transformed with relative ease into a closed shape. This requirement is essential for the subsequent phase of Monster Mash, which involves the transformation of a 2D sketch into a 3D object.

3.1.1 Closure-aware Sketch Simplification

Liu et al. [3] propose a stroke similarity metric that incorporates both perceptual and geometric characteristics of strokes. The objective is to cluster perceptually connected strokes together, forming a perceptual region that can be simplified as a single entity. Perceptual strokes are defined as those that are likely part of a closed region or that form the boundary of one. Figure 3.1 demonstrates an instance in which two identical strokes can belong to distinct perceptual regions. Strokes are grouped based on proximity, continuity, and parallelism with respect to nearby regions. Proximity measures how close strokes are to each other, continuity determines whether two strokes have similar directions, and parallelism evaluates how well two strokes align. Regions are constructed based on stroke grouping. Constructing a region requires a set of strokes that is complete (recognized as a perceptual region relative to surrounding strokes) and independent (not dependent on neighboring regions).

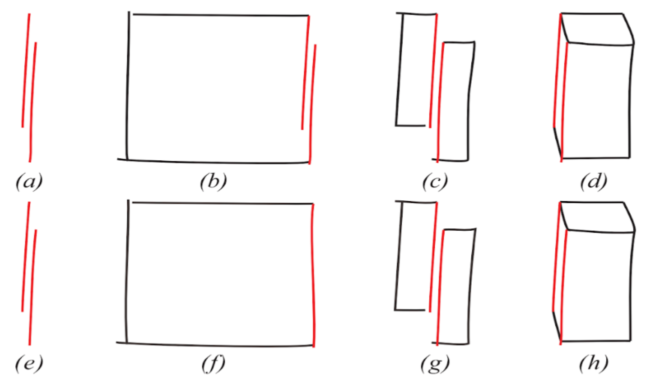


Figure 3.1: An illustration of how identical pairs of strokes can be associated with distinct perceptual regions.

Stroke interpretation is used to determine which strokes belong to the same perceptual stroke and hence belong to the same stroke group An iterative refinement process is applied to refine the initial perceptual regions and strokes. At the start, all strokes are grouped together. In each iteration, the refinement process first identifies the regions based on stroke interpretation and then groups the strokes into new clusters according to this region's interpretation. This refinement process is performed iteratively until no further changes can be made.

The primary benefit of this approach is its ability to distinguish the semantic differences between strokes. Nonetheless, it has certain limitations, such as its inability to handle closed curves and decorative strokes. The iterative refinement process is also computationally expensive. Although the authors do not provide precise time measurements, they do mention a 2.0 minute execution time for a sketch containing 167 strokes, which is considered a high execution time for this amount of strokes. The results of this method can be seen in Figure 3.2.

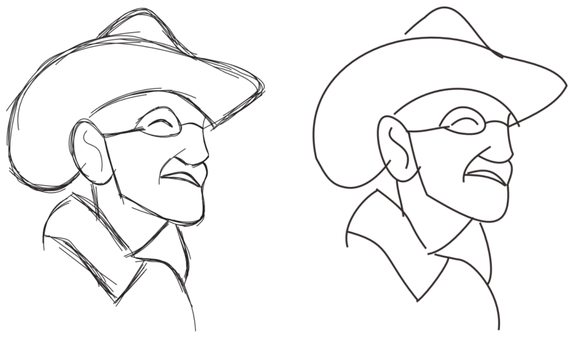


Figure 3.2: The results of simplification using this method. (taken from [3])

3.1.2 StrokeAggregator

Liu et al. proposed StrokeAggregator [4] where the primary approach involves clustering strokes into groups that subsequently define simplified curves. This process leverages principles derived from observations of artistic practices and human perception research.

In this technique, two strokes are compared based on their angular compatibility and relative proximity. Angularly compatible strokes are assigned a score reflecting the degree of compatibility. To compute this score, a common aggregate curve is first created from the pair of strokes, after which tangent angles between the newly created curve and the initial curves are calculated. Strokes are clustered according to their angular scores and then evaluated based on relative proximity. The strokes are considered approximately parallel, so proximity is roughly the distance between two parallel strokes. In the subsequent step, a local cluster refinement is performed, the primary objective of which is to separate clusters that resemble branches. This process involves iteratively subdividing branches into segments while considering factors such as narrowness, denoted by the minimal width-to-length ratio, and uniformity, characterized by strokes exhibiting significantly increased inter-cluster spacing along a substantial segment of their length compared to the intra-cluster spacing within each branch.

The final phase assesses the internal coherence of the computed clusters to address uncertainties and unite clusters exhibiting compatibility in both angular and spatial aspects. This phase reflects the branch segregation process, employing similar criteria and principles. Ultimately, curves are fitted to the resulting clusters.

A limitation of this methodology is its reliance solely on the nearby stroke context. As a result, it might not work well when stroke clues are unreliable. Specifically, it’s not directly suitable for stylized line drawings. Furthermore, the average time required for curve simplification is approximately 2.5 minutes.

The results of this method can be seen in Figure 3.3.

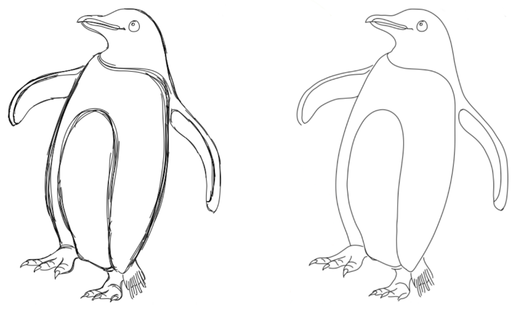


Figure 3.3: The results of simplification using StrokeAggregator method. (taken from [4])

3.1.3 StrokeStrip

An alternative strategy is described in [5]. In the article, the authors solve the sketch simplification problem by introducing a method called StrokeStrip, which focuses on fitting intended curves to vector stroke clusters in a manner consistent with human perception. They achieve this by observing that human viewers perceive stroke clusters as continuous, variably-wide strips whose paths represent the intended curves (see Figure 3.4b). The authors then formulate the curve fitting problem as the joint parameterization of cluster strokes, with a 1D parameterization that is the restriction of the natural arc length parameterization of the perceived strip to the strokes in the cluster (see Figure 3.4c).

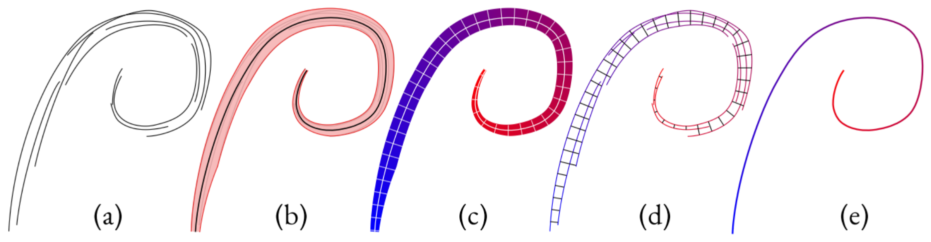


Figure 3.4: Different representations of strokes.

To generate fitting outputs that align with viewer expectations, authors compute a joint cluster parameterization that satisfies several requirements, such as being continuous, arc length preserving, and having isolines that are orthogonal to the strokes. The challenge lies in the fact that the strip geometry is not known a priori, making it difficult to identify points on different strokes that are within the strip (WTS) adjacent. The authors address this challenge by formulating the problem as a constrained variational problem and solving it using a combined discrete-continuous optimization framework. This approach allows them to handle complex cluster configurations robustly, including self-adjacent and self-intersecting clusters. However, there may be limitations to their method, such as its ability to handle extremely intricate or overlapping stroke clusters or potential challenges in processing raster inputs. Additionally, this method exhibits limited efficiency, as it does not permit real-time parameterization. The results of this method can be seen in Figure 3.5.

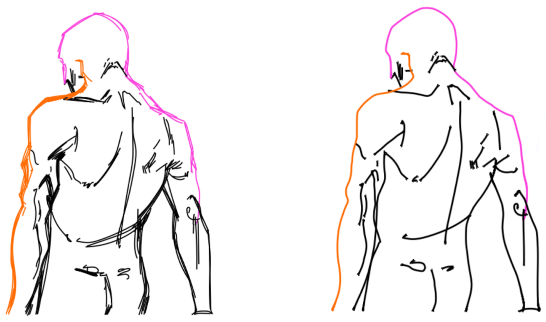


Figure 3.5: The results of simplification using StrokeStrip method. (taken from [5])

3.1.4 Simplifying Sketches with Convolutional Networks

A distinctly different approach, currently gaining popularity, involves the use of neural networks, particularly convolutional neural networks (CNNs) for image processing tasks. CNN is a type of deep learning model primarily used for image processing and recognition tasks. Commonly, CNNs consist of an input and output layer, along with multiple hidden layers including convolutional, pooling, and fully connected layers. Convolutional layers use filters to scan the input data, detecting local features such as edges or textures. Pooling layers reduce data dimensionality and computational complexity, often by retaining the maximum value of a particular feature over a region. Fully connected layers use high-level features from the previous layers to perform classification tasks. CNNs are trained using backpropagation and gradient descent algorithms to minimize the difference between the predicted and actual output.

Simo-Serra et al. [6] developed a method enabling users to handle more general and challenging inputs, such as rough raster sketches obtained from scanned pencil drawings. This technique offers superior performance times, processing most input images in under a second. Their method employs only convolutional layers, excluding the typically utilized fully connected layers. The model they created (see Figure 3.6) consists of three varieties of convolutional layers: down-convolution, which reduces the image size by half; flat-convolution, which retains the image size; and up-convolution, which doubles the image size. The procedure commences with down-convolutions to shrink the image size, subsequently lowering data bandwidth and enhancing spatial support for the following layers. Ultimately, up-convolutions are implemented to reestablish the image's initial dimensions.

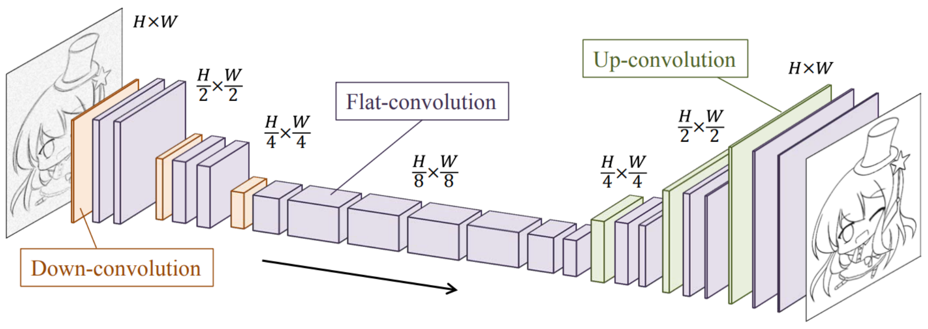

$H \times W$
$\frac{H}{2} \times \frac{W}{2}$
$\frac{H}{4} \times \frac{W}{4}$
Flat-convolution
$\frac{H}{8} \times \frac{W}{8}$
Up-convolution
$\frac{H}{4} \times \frac{W}{4}$
$\frac{H}{2} \times \frac{W}{2}$
$H \times W$
Down-convolution

Figure 3.6: Visulization of CNN (taken from [6])

The dataset is generated using inverse dataset construction, which involves creating rough sketches from clean ones. This method is employed since artists often take creative liberties while developing a clean image, altering various aspects of the original sketch. Such modifications may hinder the model's ability to learn new mappings. The results of this approach are highly satisfactory. Unlike other methods, this technique can even simplify photos of hand-drawn sketches. The authors have developed a website that allows users to test their neural network. Figure 3.7 demonstrates the simplification of a tested sketch using their method.

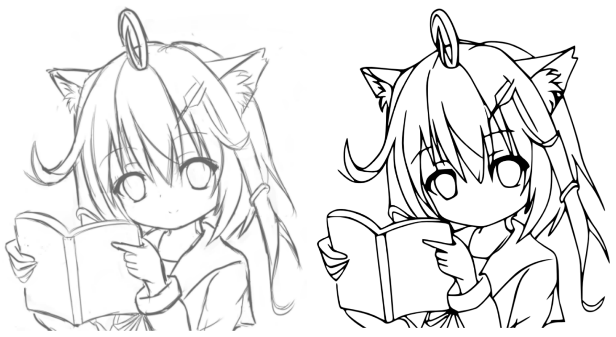


Figure 3.7: The results of simplification using CNN method. (taken from [**NNSimplyfierImg**])

3.1.5 Lazy-Brush

Sýkora et al. proposed Lazy-Brush [7], which is an approach that centers on the application of a technique known as image segmentation for sketch coloring. Image segmentation is a procedure that entails dividing a digital image into numerous sections, also known as sets of pixels or super-pixels. The purpose of segmentation is to simplify or modify the representation of an image into a more meaningful and analyzable format. In practice, segmentation is used primarily to detect objects and boundaries, such as lines and curves, within images.

The proposed algorithm colors areas by applying minor strokes within these segments (see Figure 3.8 and Figure 3.9). The algorithm effectively addresses challenges such as color leakage Figure 3.9b, which can occur when a user attempts to color an unbounded area, as well as the task of manually bridging gaps to prevent this leakage. The algorithm, dubbed Lazy-Brush [7], seeks to colorize as vast an area as possible within an optimal boundary and coloring sections labeled the most by the user, which is a technique referred to as soft scribbles (see Figure 3.9c). The algorithm functions by resolving an optimization problem that assigns a specific color to each pixel.

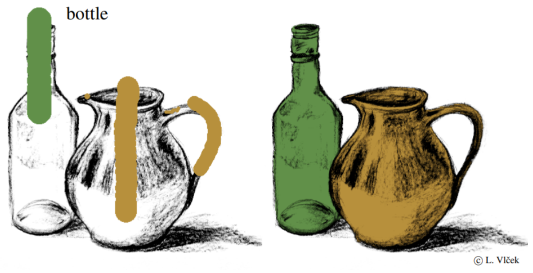


Figure 3.8: The illustration of how Lazy-Brush technique colors hand draw sketches. (taken from [7]

The issue is addressed within the framework of energy minimization, which has been proposed to meet these requirements. The goal is to find labeling for each pixel that minimizes a predefined energy function (see Equation 3.1). This energy function consists of two components: a smoothness term and a data term.

$$E(c) = \sum_{p,q \in \mathcal{N}} V_{p,q}(c_p, c_q) + \sum_{p \in P} D_p\left(c_p\right) \tag{3.1}$$

In the context of image colorization, the color variance between neighboring pixels is encapsulated by the term $V_{p,q}(c_p, c_q)$, known as the smoothness term. It assists in the effective placement of color transitions, ideally within areas with a lower pixel intensity, as illustrated in Fig. 3.9d. Moreover, it automatically adjusts to prevent the creation of unintended shortcuts across regions of white pixels. The mathematical expression for the smoothness term is given by:

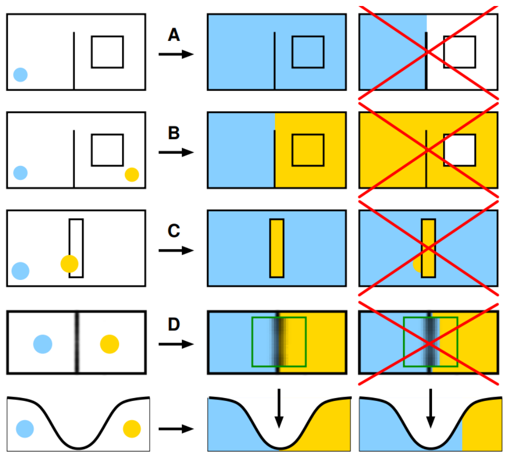


Figure 3.9: The image illustrates an example of how the software functions when users employ labels, represented by colored dots, to specify areas for coloring. (taken from [8]

$$V_{p,q}(c_p, c_q) \propto \begin{cases} I_p & \text{if } c_p \neq c_q, \\ 0 & \text{otherwise,} \end{cases} \tag{3.2}$$

where I_p is the pixel intensity at the location p. The data term, denoted by $D_p(c_p)$, quantifies the energy associated with attributing a particular color c_p to a pixel p. This term is unique in that it is entirely influenced by the user's input, thereby allowing users to flexibly specify the penalty for pixel labeling. Unlike traditional methods that consider all user-annotated constraints as strict rules, this approach offers the flexibility to define softer constraints. The data term is mathematically defined as:

$$D_p(c_p) = \lambda \cdot \mathcal{K}, \tag{3.3}$$

In this equation, λ signifies both the existence and strength of a brush stroke, enabling users to set varying degrees of constraints on the pixel labeling. The variable $\mathcal{K}$ is a constant that represents the discontinuity energy at white pixels, serving as a penalty for color transitions at high-intensity pixel locations.

To address the energy minimization issue, the authors converted it into a multiway cut problem on an undirected graph, which is resolved using a greedy multiway cut algorithm, which capitalizes on a unique graph topology that ensures connected labeling.

One potential application of this algorithm is depicted in the article Smart Scribbles [8], which enables region filling without requiring the user to draw seeds to specify the coloring area manually. In this approach, the sketch itself is appointed as a soft scribble (see Figure 3.10a), eliminating the need for the user to specify one. Following this, borders are incorporated into the image, and these are also marked as a soft scribble but of a distinct color (see Figure 3.10b). This technique enables automatic image filling (see Figure 3.10c).

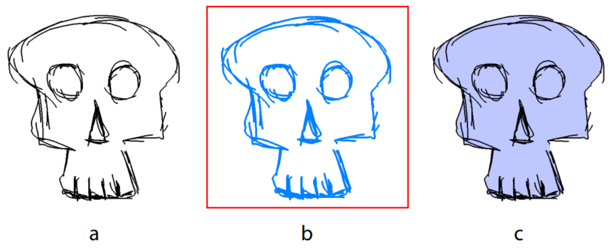


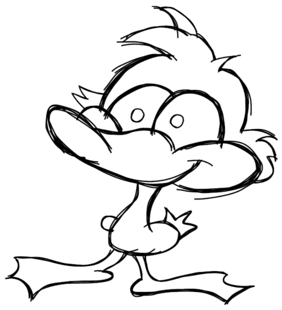
Figure 3.10: The illustration of how Smart Scribble can be used for image coloring. (taken from [8]

3.1.6 Discussion

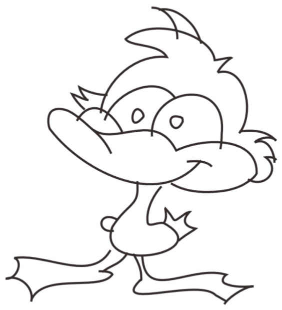
Firstly, we compare the results in Figure 3.11 of all these methods (except for Lazy-Brush) using the same reference image. In the image below, using the example of a duck, we can see how differently these methods simplify the image.

Nearly all methods discussed previously share a common drawback - slow performance time. Although the articles did not provide specific performance measurements, there were occasional references to the time cost in certain examples. This is sufficient to infer that these solutions are too slow for real-time evaluation. The neural network method mentioned in subsection 3.1.4, while fast and yielding favorable results, poses complications for integrating into Monster Mash. Moreover, almost all these methods produce simplified sketches that consist of strokes that do not create closed shapes, they don't take into account the closure of the shape, which is required for our purposes. So to implement these methods into monster mash, there will be a need for an algorithm that will create closed shapes out of the simplified image.

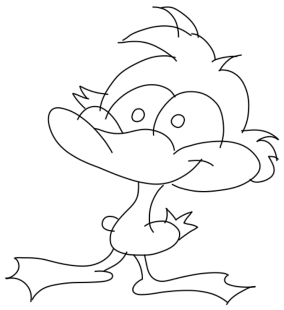
Fortunately, the procedure discussed in subsection 3.1.5, based on the Lazy-Brush algorithm, has the advantage of segmenting images into complete, closed regions. This approach does not suffer from the significant computational time drawback. Given these notable benefits, this method has been selected for implementation.



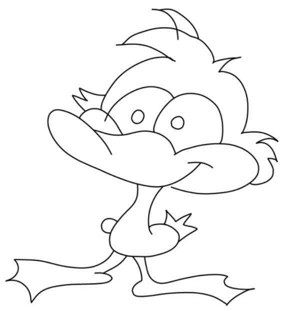
(a) : Original picture of duck.



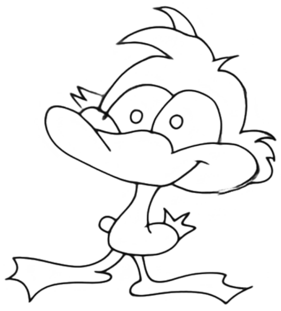
(b) : Duck simplified by Closure-aware method.



(c) : Duck simplified by StrokeStrip method.



(d) : Duck simplified by StrokeAggregator method.



(e) : Duck simplified by Convolutional Networks.

Figure 3.11: The comparison of all methods described above.

3.2 Our Approach

In light of the specified criteria, an appropriate strategy for the aim of this thesis involves the creation of an algorithm that employs image segmentation. The intended outcome of this algorithm is to produce a closed contour outlining the filled shape within a given portion of the sketch. By repeating this process, the final sketch will consist of multiple closed contours. This approach not only ensures computational efficiency but also facilitates seamless integration into Monster Mash while remaining consistent with its underlying principles.

To solve this problem, we implement the same technique as in Smart Scribbles from subsection 3.1.5. Specifically, we use the user-drawn sketch as the primary segmentation marker and the added borders as the secondary segmentation marker, mimicking the dual-marker approach illustrated in Figure 3.10.

The problem at hand can be solved by first addressing a different problem - finding an optimal labeling of a set of pixels to help resolve the initial task. The energy function will be defined to separate the drawing from the background while filling the gaps in the sketch. The outcome of this labeling will be a filled version of the initial sketch Figure 3.12a, as shown in Figure 3.12b. The contour of this shape will then be determined, as depicted in Figure 3.12c.

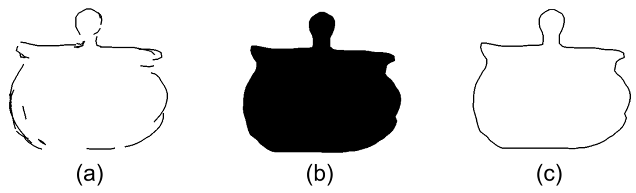


Figure 3.12: Illustration of steps of the image segmentation process.

The optimization problem (see Equation 3.4) seeks the best method to divide an image into two segments by assigning a label of 0 or 1 to each pixel. The objective is to minimize the sum of the weights w_{ij} between neighboring pixels. Neighboring pixels with the same label are not included in the total sum. Sets of pixels that must be labeled 0 and 1 are denoted as S and T, respectively. M represents the set of all pairs of neighboring pixel indices in the image, and N refers to the number of pixels in the image. The weight w_{ij} is set as follows: if at least one pixel is black, the weight is set to a small constant value a; otherwise, it is set to a large value b.

$$\begin{aligned} x^* = \arg\min_x \quad & \sum_{(i,j)\in M} w_{ij}|x_i - x_j| \\ \text{subject to} \quad & x_i = 0, \quad \forall i \in S \\ & x_i = 1, \quad \forall i \in T \\ \text{variables:} \quad & x_{i\in 0\dots N} \in \{0,1\} \end{aligned} \tag{3.4}$$

Values a and b must be chosen carefully. b cannot be too large. Otherwise, it will always be more cost-effective to circumvent the entire stroke, as illustrated in Figure 3.13b, rather than closing it, as depicted in Figure 3.13a.

The input image contains rasterized strokes, resulting in a black-and-white binary image. Firstly, pixels on the image borders are labeled with the value

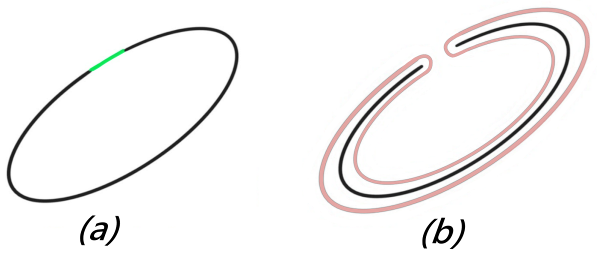
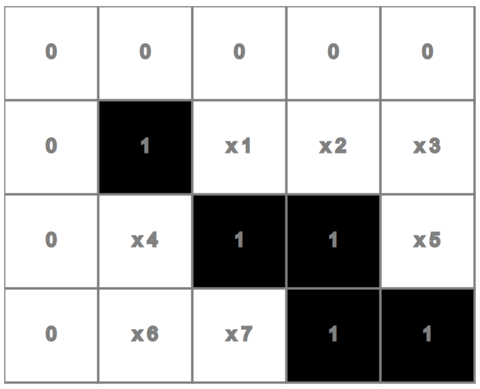


Figure 3.13: Strokes with different weight values.

0, this step is taken to distinguish the foreground (sketch) and background of the image. Following this, pixels that are a part of the strokes are assigned a label 1. The unknown values $x_1 - x_n$ represent the pixels that require labeling, as demonstrated in Figure 3.14.



0	0	0	0	0
0	1	x1	x2	x3
0	x4	1	1	x5
0	x6	x7	1	1

Figure 3.14: Magnified left upper part of an image.

One approach to solve this problem is to transform it into a graph cut problem, which enables the use of efficient algorithms (see Figure 3.15). A graph $G = \{V, E\}$ is created, where each vertex $v \in V$ represents a pixel from the initial image. An edge $e \in E$ connects two vertices if the pixels they represent are neighbors in the image. Additionally, sink and source vertices are added named T and S. Each vertex connects to the source and the sink vertex. Pixels labeled as foreground are connected to the source node with a large maximum terminal value L. Pixels marked as background have their respective nodes connected to the sink node with a high constant maximum terminal value Q. Other vertices have their maximum terminal values for source and sink nodes equal to zero. The minimum cut of the newly created graph forms the desired contour, as demonstrated in Figure 3.16.

Upon creating the graph, the next step is to compute its minimum cut. Several algorithms, such as Ford-Fulkerson [10], Goldberg-Tarjan [11], and Dinic [12], can be used for computing the minimum cut. In this particular case, the GridCut library is employed, which uses an improved Boykov-Kolmogorov

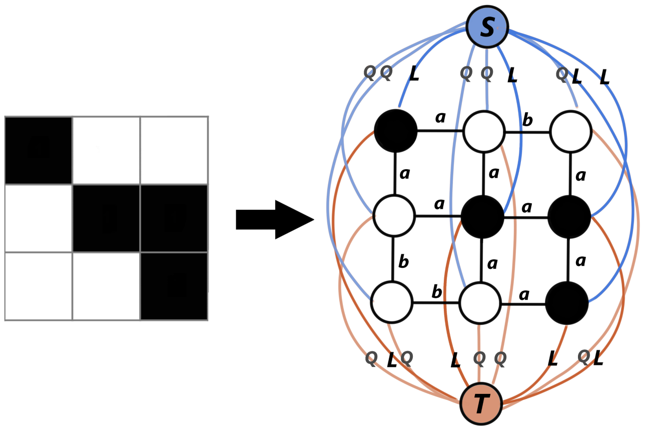


Figure 3.15: A visualization of how the image is transformed into a graph.

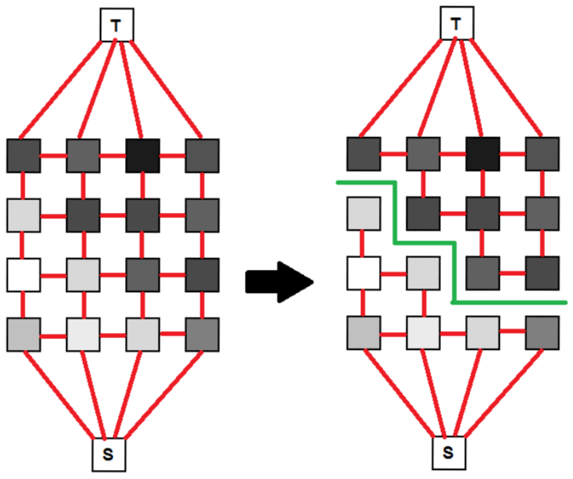


Figure 3.16: A visualization of a cut in the grid-like graph (taken from [9]).

algorithm [13], specifically developed for grid-structured graphs. The basic algorithm is described in detail in subsection 3.2.2.

Computing the minimum cut of the constructed graph results in two disjoint subsets of nodes in the graph. The edges that are part of the minimum cut define the contour of the image. One subset of vertices is colored black, and the other one is colored white, producing a filled sketch. However, the contour of the image still needs to be identified. To find the contour, the STA algorithm described in subsection 3.2.1 is employed. The utilization of the STA algorithm enables us to specify the approximate length of the lines constituting the sketch. Following the successful creation of the contour, we can subsequently generate a simplified version of the sketch.

3.2.1 Square Tracing Algorithm

The Square Tracing Algorithm (STA) [14] is a versatile contour-tracing technique that extracts the boundary of a binary image containing connected regions, such as shapes or objects.

The Square Tracing Algorithm functions by scanning a binary image pixel-by-pixel in search of the first foreground pixel that it encounters. Once identified, the algorithm traces the contour by navigating through the neighboring pixels. In each step, the algorithm executes the following instructions: if it detects a black pixel, it turns left, and if it detects a white pixel, it turns right. This process continues until the algorithm returns to the starting pixel, thereby completing the contour trace. The output is a list of coordinates representing the region's boundary.

3.2.2 Graph Cut Computation

Image segmentation, a fundamental task in the field of computer vision, often relies on efficient graph-based algorithms for solving various problems, such as finding the min-cut/max-flow in a grid-like graph. Numerous approaches have been developed to tackle this challenge; however, conventional algorithms frequently underperform when applied to grid-structured graphs in image segmentation tasks.

Fortunately, there exists a clever min-cut/max-flow Boykov-Kolmogorov [13] algorithm specifically designed to overcome the constraints of conventional augmenting path methods when utilized for image segmentation in grid-like graphs. The proposed algorithm is based on the principle of augmenting paths and employs a distinctive approach of constructing and reusing two search trees, with one originating from the source node s and the other from the sink node t. This method enables a more efficient search and circumvents the costly procedure of reconstructing search trees from the beginning.

The algorithm iteratively performs the following three stages:

1. Growth stage: During the growth stage, two search trees, S and T, rooted at the source node s and the sink node t, respectively, are expanded. Each tree comprises active and passive nodes. Active nodes symbolize the outer border of each tree and can grow by acquiring new children along non-saturated edges from the set of free nodes. In contrast, passive nodes cannot grow, as they are entirely blocked by other nodes from the same tree. The growth stage persists until an active node encounters a neighboring node that belongs to the opposite tree, indicating the discovery of an $s - t$ path. Augmentation stage: Upon identifying the $s - t$ path, the augmentation stage begins. During this stage, the discovered path is expanded, which may cause some edges in the path to become saturated. If a node is saturated,

its connection to its children becomes invalid. Consequently, some nodes in trees S and T transform into "orphans," causing the search trees S and T to fragment into multiple forests.

Adoption stage: The final stage aims to reconstruct the single-tree structure of sets S and T, anchored at the source and sink nodes. Orphans attempt to find new valid parents within their respective sets (S or T) connected via a non-saturated edge. If a suitable parent cannot be located, the orphan is removed from its set and becomes a free node, while its former children are designated as orphans. This stage concludes when all orphans have been addressed, and the single-tree structure of S and T is restored.

The algorithm terminates when the search trees S and T cannot grow, implying that a maximum flow is achieved.

Even though it has potentially worse theoretical complexity than established methods, experimental comparisons presented in the article demonstrate that the algorithm significantly outperforms traditional approaches on typical problem instances in N-D grids with locally connected nodes where many nodes are connected to the terminals.

Chapter 4

Stroke Deformation

Our fundamental objective is to create or find an algorithm for sketch deformation that results in an intuitive outcome for the user. Essential attributes for this sought-after algorithm include swift computational speed and versatility, enabling the user to manipulate global and local deformations.

To begin with, we plan to investigate an array of image deformation techniques that hold promise for our project's requirements. Following this exploration, we will comprehensively discuss the technique we eventually choose to implement, underscoring its features, benefits, and operational mechanics.

4.1 Related Work

This section delves into a thorough examination of algorithms relevant to the field of image deformation. This investigation aims to identify a suitable algorithm, or a blend of them, which can be effectively incorporated into Monster Mash to boost its capabilities. This chapter offers an exhaustive discussion of various methods, meticulously exploring each one, thereby creating a basis for their potential application within Monster Mash.

4.1.1 Affine and Projective Transformations

Affine transformations [15] are a type of linear transformation that preserve collinearity and ratios of distances. This means that straight lines in the original image will still be straight in the transformed image, and parallel lines will remain parallel. Affine transformations include operations such as scaling, rotation, translation, and shear.

Mathematically, an affine transformation can be represented as a matrix multiplication and a vector addition:

$$v' = Av + t$$

Here, v is a vector representing a point in the original image, A is a 2x2 matrix representing the linear transformation (which can include scaling, rotation, and shear), t is a 2D vector representing the translation, and v' is the transformed point.

Projective transformations (also known as homographies) are a more general form of transformation that can represent changes in perspective. This means they can transform parallel lines into lines that intersect at a vanishing point, which is impossible with affine transformations.

In the 2D case, a projective transformation can be represented as a 3x3 matrix multiplication in homogeneous coordinates:

$$v' = Hv$$

Here, v is a 3D vector representing a point in the original image in homogeneous coordinates (i.e., $[x, y, 1]$), H is a 3x3 matrix representing the projective transformation, and v' is the transformed point, also in homogeneous coordinates. To convert v' back to regular 2D coordinates, we divide it by the third component.

Both of these transformations can be applied to a sketch, stroke, or image by manipulating each point within the image (see Figure 4.1). However, as these transformations do not usually map pixels to pixels precisely, a method of interpolation, such as bilinear or bicubic, is generally employed to ascertain the pixel values in the transformed image. It's crucial to note that this method does not preserve the lengths and angles of the original image.

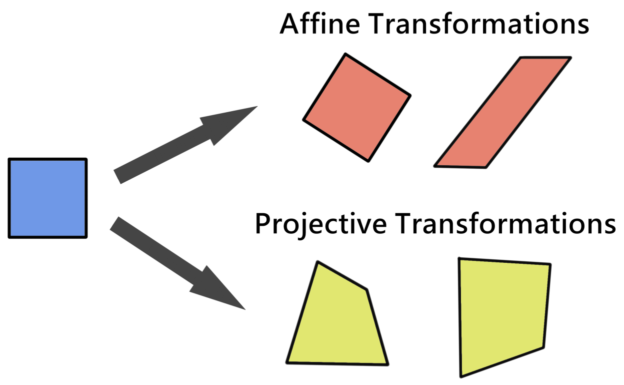


Figure 4.1: Example of affine and projective deformations of a square.

A significant drawback is that these transformations are global, implying that they impact every point identically. To address this issue, these transformations can be paired with other techniques. For instance, one prevalent method for achieving local control involves subdividing the image into smaller segments and conducting the transformation solely on these smaller segments.

4.1.2 B-Splines

B-Splines or Basis Splines [16] provide a flexible method for image and sketch deformation. They are particularly well-suited to image deformation tasks because of their local control and variation-diminishing properties, meaning that changes to the position of one control point affect only a limited region around that point, providing a high level of control over the deformation.

The idea behind B-Splines is to define a smooth curve (or surface in 2D) that is determined by a set of control points. A B-Spline is defined as a piecewise-defined polynomial function, and it is controlled by adjusting the positions of the control points.

In the context of image deformation, a grid of control points is defined over the image, and each control point is associated with a B-Spline. The deformation of a point in the image is calculated as a weighted sum of the displacements of the control points, where the weights are given by the values of the B-Splines.

To apply the B-Spline deformation to the image(see Figure 4.2), the formula is evaluated for each pixel in the image, and the pixel is moved according to the resulting displacement. As with other deformation methods, since the deformation does not generally map pixels to pixels, some form of interpolation is used to determine the pixel values in the deformed image.

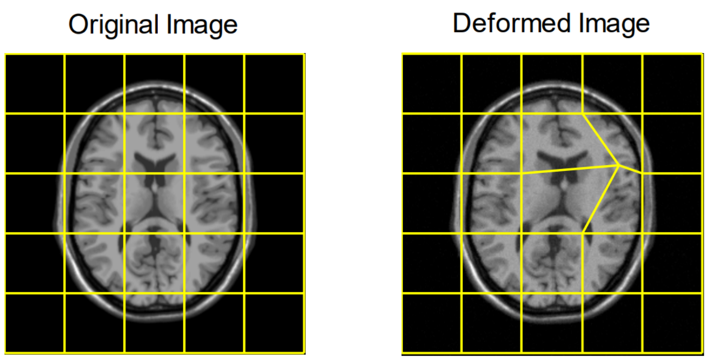


Figure 4.2: Example of B-Spline deformation. (taken from [17]).

The disadvantage of this method is that B-spline deformation does not inherently maintain rigidity. This implies that it can cause significant changes in local shapes or structures during deformation, potentially leading to unrealistic results. Furthermore, B-splines can be complex to comprehend, implement, and manipulate. They require more parameters and computations, which could result in increased computational costs.

4.1.3 As-Rigid-As-Possible

The As-Rigid-As-Possible (ARAP) [18] [19] deformation method is a widely used technique in computer graphics and computer vision for generating smooth and realistic deformations of 2D or 3D objects. The ARAP approach aims to preserve an object's natural shape and geometry while enabling flexible and intuitive deformation. Unlike other deformation techniques, ARAP emphasizes maintaining an object's rigidity by minimizing the strain energy between its vertices. This property makes ARAP particularly suitable for tasks such as shape manipulation, animation, and shape matching. Users can specify a set of constraints that the deformation must satisfy, allowing for precise control over the final deformation result

In the context of image deformation, this method begins by dividing the image into a mesh of triangles or other simple polygons. For each polygon, it then computes a deformation that moves its vertices to their new positions while keeping the polygon as rigid as possible. This is done by minimizing a cost function that measures the deviation of each deformed polygon from a rotation of the original polygon.

The goal is to find the deformed positions p'_i and the rotation matrix R that minimize this cost function. This is typically done using an iterative method, where in each iteration the positions p'_i are updated to minimize the energy function given the current R, and then R is updated to minimize the energy function given the new positions p'_i.

Applying the ARAP deformation to an image involves calculating the deformed positions for each pixel in the image based on the deformations of the polygons in its neighborhood.

ARAP transformations offer advantages such as preserving local rigidity and providing a degree of independent local control for different parts of an object during deformation. However, they come with disadvantages, including the potential for high computational intensity when dealing with large datasets, and a tendency to potentially distort the global shape despite excellent preservation of local structures.

4.1.4 Moving Least Squares Method

Moving Least Squares (MLS) is a technique commonly used in image deformation, particularly when a smooth and flexible deformation is required. Schaefer et al. [20] introduced the technique in a graphics context.

The fundamental idea of MLS is that instead of directly deforming an image, we deform the space in which the image exists. Specifically, we define a set of control points in the original space, and for each control point, we specify a target location to which it should be moved. The deformation of any given point in the image is then determined by the movements of the control points around it. The example of MLS deformation can be seen in Figure 4.3.

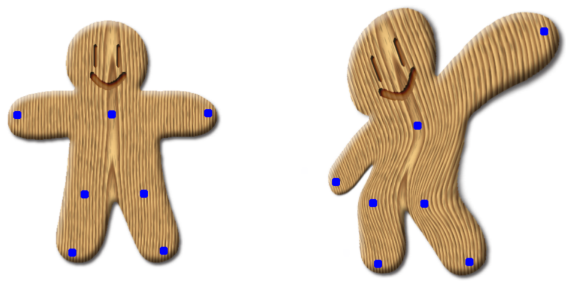


Figure 4.3: Example of MLS deformation. (taken from [20]).

The MLS process consists of two steps: the creation of a local coordinate system relative to the control points and the computation of an affine transformation in this local coordinate system.

Step 1: Define a local coordinate system

Given a set of control points p_i and their corresponding target locations p_i', the weights w_i for each control point are calculated for any given point v in the image. Typically, this computation employs the following formula:

$$w_i = \frac{1}{||v - p_i||^{2d}}$$

where $||v - p_i||$ is the distance from v to p_i and d is a parameter that controls the influence of the control points.

Next, the we centroid p and $p'$is calculated for the original and target locations of the control points, respectively:

$$p = \frac{\sum w_i p_i}{\sum w_i}$$

$$p' = \frac{\sum w_i p_i'}{\sum w_i}$$

Finally, we define a local coordinate system by subtracting the centroid from each control point:

$$q_i = p_i - p$$

$$q'_i = p'_i - p'$$

Step 2: Compute the affine transformation

In the local coordinate system, we compute a 2x2 matrix A that maps each q_i to q'_i. This is done by solving the following least squares problem:

$$\min_A \sum w_i ||Aq_i - q'_i||^2$$

The solution to this problem can be found by singular value decomposition (SVD).

Finally, point v is maped to its deformed location v' using the affine transformation:

$$v' = p' + A(v - p)$$

This process is repeated for every point in the image to produce the deformed image.

This method provides flexible and smooth deformations, allowing for some degree of local control. However, it might not preserve rigidity as effectively as other methods like ARAP, and can be computationally expensive when applied to large datasets

4.1.5 Deep Learning-Based Deformation

Deep learning-based deformation represents a class of methods where a neural network is trained to learn the deformation mapping function. These methods have shown significant promise due to the flexibility and learning capacity of neural networks.

One common approach is to use a type of network known as a Convolutional Neural Network (CNN) [21] [22]. The input to the network is an image or a patch of an image, and the output is the deformation field for that image or patch. The network is trained on a set of images for which the desired deformation fields are known. The results can be seen in **??**

Suppose I is the original image and I' is the deformed image. The network is designed to learn a mapping function f such that:

$$I' = f(I; \theta)$$

Here, f represents the network, and θ are the parameters of the network, which are learned during training. The goal of training is to find the parameters θ

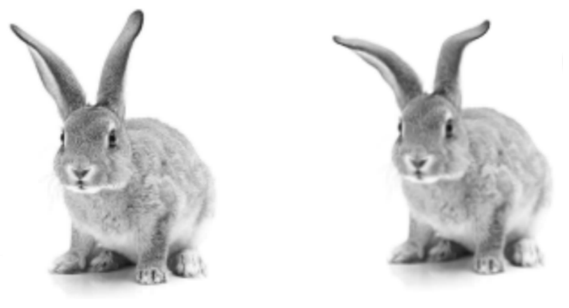


Figure 4.4: Example of CNN deformation. (taken from [21]).

that minimize the difference between the network's output and the desired output for a set of training images. This difference is measured by a loss function L, which in this case might be the mean squared error between the predicted and actual deformed images:

$$L = ||I' - f(I;\theta)||^2$$

The parameters θ are typically learned using a method such as stochastic gradient descent, which iteratively adjusts the parameters to minimize the loss function.

Once the network is trained, it can deform new images by passing them through the network and applying the resulting deformation fields.

Deep Learning-Based Deformations can learn complex deformation patterns from large amounts of data, providing advantages in tasks requiring high-level, nonlinear transformations. However, they require substantial amounts of training data and computational resources. Moreover, despite their powerful capabilities, they may produce less interpretable results compared to more traditional geometric methods.

4.1.6 Discussion

After reviewing all approaches mentioned above, we have chosen an approach that uses the As-Rigid-As-Possible deformation technique. This technique possesses various distinct advantages, rendering it a particularly effective selection for image deformation tasks. Unlike global methodologies such as Affine and Projective Transformations, ARAP enables local manipulation of the deformation, indicating that modifications at a single point do not indiscriminately influence the entirety of the image. Nonetheless, it upholds global coherence, assuring that the overall image maintains its consistency and unity. This feat can be more demanding for local methods such as B-Splines to sustain.

ARAP's primary emphasis lies in preserving the rigidity of local regions within the image during the deformation process. This approach effectively mitigates the risk of unnatural distortions and aids in preserving the original attributes of the image, an accomplishment not always feasible with alternative methods like Moving Least Squares.

Also unlike deep learning-based methods, ARAP does not demand prolonged training periods or large datasets. It is an algorithmic approach that can be directly applied to an image, enhancing its efficiency and immediate applicability. Moreover, the aforementioned technique is employed to deform 3D objects within Monster Mash in the animation part.

4.2 Our Approach

In this research, we utilize the ARAP deformation algorithm, which was discussed in section subsection 4.1.3. The LibIGL library, known for providing mesh deformation functionality for 2D and 3D objects, is used for the implementation phase. The ARAP algorithm within LibIGL library builds upon the approach developed by Sorkine and Alexa [18]. This section elucidates the workings of this approach and how we adapted it for sketch deformation.

Figure 4.5 presents examples of the ARAP technique applied to a 3D cactus model. The red dots denote constrained vertices that remain stationary, while the green dots symbolize constrained vertices that have been moved. By repositioning even a single vertex, the entire mesh adjusts to the most optimal position. These images demonstrate the remarkable capabilities of the ARAP technique.

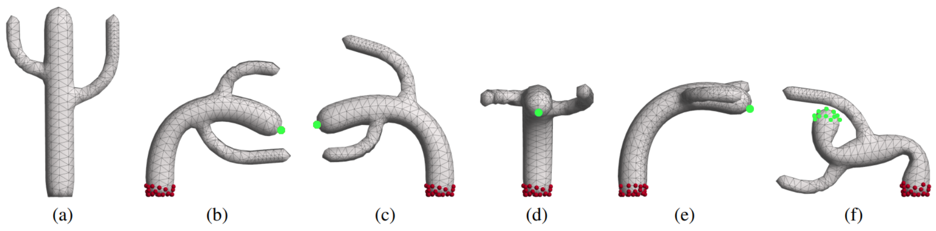


Figure 4.5: Example of ARAP deformation. (taken from [18]).

While the ARAP method is suited for 2D and 3D meshes, our sketches are presented as line sets, which can't be directly employed for deformations. Therefore, a crucial preparatory step involves transforming these outlines into a workable 2D object.

There exist numerous strategies for generating a 2D object from the initial lines. One such approach involves regarding the points forming the shape as the boundary of the 2D object, introducing auxiliary vertices within this boundary, and carrying out a triangulation process, Figure 4.6. This technique develops the object as a filled entity, which imposes certain usage limitations. The filled object approach tends to limit flexibility and user control, particularly since it does not allow for individual line deformations, which are essential for creating intricate designs or making precise adjustments.

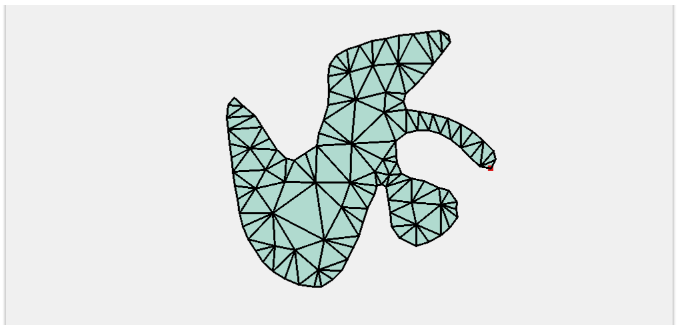


Figure 4.6: The example of an alternative way to create 2D object out of contour.

An alternative solution is to construct a 2D strip from the lines. In this process, every line generates four triangles - two positioned above the line and two below it. Consequently, for each vertex v_i present on the sketch's borders, two additional vertices are required to be computed. The first additional vertex is regarded as "above" and denoted as v_i', while the second is considered "below" and labeled v_i''.

To calculate the coordinates of vertex v_i', we consider two lines, l_{i-1} and l_i, that intersect at this point. We then compute their respective normals n_{i-1} and n_i and normalize these values. Following this, we generate a vector

$$d_i = \frac{(n_{i-1} + n_i)}{|n_{i-1} + n_i|} * l,$$

where $l = |d|$. Given the method employed to create vector d_i, every vector d_i will possess a length of l. The positions of the two new vertices can then be calculated as $v_i' = v_i + d_i$, and $v_i'' = v_i - d$ (see Figure 4.7).

Figure Figure 4.8) provides a visual representation of a portion of the resultant strip.

Having successfully calculated vertices of the 2D object, we can proceed to generate the object itself. 2D object can be represented by a mesh S with m triangles and n vertices. The mesh S is composed of a vertex matrix V and a face matrix F. Each row of the matrix V encapsulates the coordinates of a single vertex, resulting in a structure with two columns and n rows. During the creation of this matrix, vertices that form part of the initial mesh are positioned first, followed by vertices v', and finally, vertices v''. The symbol s represents the number of vertices in the initial sketch, hence $n = 3s$. Therefore, the vertex matrix can be expressed as follows:

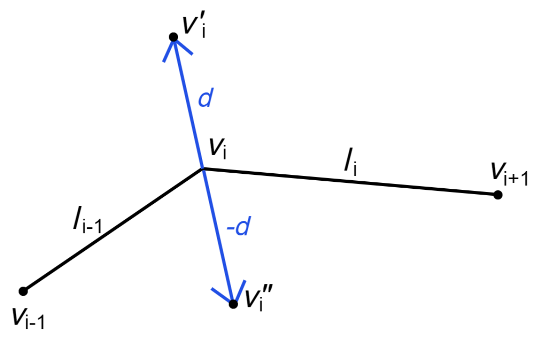


Figure 4.7: Example of created vertices v'_i and v''_i for lines l_{i-1} and l_i.

$$V = [v_0, ..., v_s, v'_0, ..., v'_s, v''_0, ..., v''_s]^T \tag{4.1}$$

Each row of the matrix F contains three indices of vertices from the array V that construct a single triangle and m rows, where $m = 4s$. It is necessary to generate four triangles for every line of the initial sketch shape. $F(i)$ corresponds to one row of the matrix. This is achieved by iterating through $i \in 0...s$, and creating triangles for each index i (see Figure 4.8).

$$\begin{aligned}
F(i) &= [i, \quad i+s, \ i+1] \\
F(i+s) &= [i+s, i+1, \ i+s+1] \\
F(i+2s) &= [i, \quad i+2s, i+2s+1] \\
F(i+3s) &= [i, \quad i+1, \ i+2s+1]
\end{aligned}$$

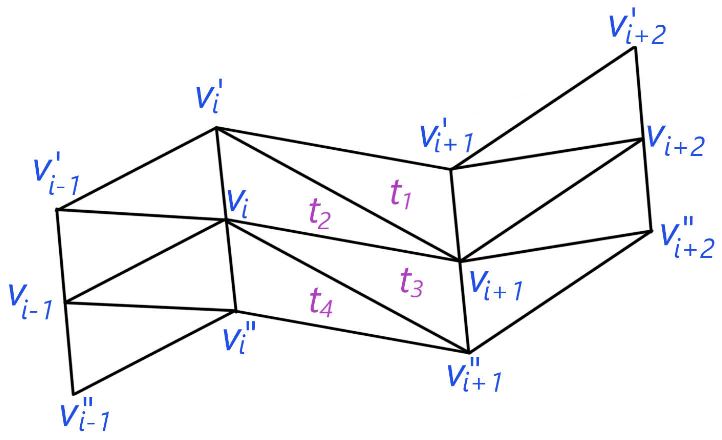


Figure 4.8: A part of the strip.

The deformed mesh, denoted as S', maintains the same connections but has different vertex positions. To analyze and process the deformation, the mesh is divided into cells C_i based on vertices. Every individual cell composes of a vertex along with the triangles that are interconnected to the said vertex. $\mathcal{N}(i)$ represents the set of vertices connected to vertex i.

$$E\left(\mathcal{S}'\right) = \sum_{i=1}^{n} w_i E\left(\mathcal{C}_i, \mathcal{C}_i'\right) = \sum_{i=1}^{n} w_i \sum_{j \in \mathcal{N}(i)} w_{ij} \left\| \left(\mathbf{p}_i' - \mathbf{p}_j'\right) - \mathbf{R}_i \left(\mathbf{p}_i - \mathbf{p}_j\right) \right\|^2 \tag{4.2}$$

Value w_i is set to 1 and cotangent weight formulas define values w_{ij}. α_{ij} and β_{ij} represent the angle opposite to the mesh edge (i, j).

$$w_{ij} = \frac{1}{2}\left(\cot \alpha_{ij} + \cot \beta_{ij}\right) \tag{4.3}$$

The ARAP method employs an iterative solver that alternates between optimizing positions p' and rotation matrices R_i. During each iteration, the method updates the surface vertices and computes all R_i that aligns the surface with the input points. This procedure is repeated until a local energy minimum is achieved.

Rotation Matrix Ri

To determine the optimal rotation matrix R_i that represents the deformation between the original cell C_i and its transformed counterpart C_i', a weighted least squares method is utilized. This method minimizes an energy function E that measures the difference between the cells in Equation 4.4.

$$E\left(\mathcal{C}_i, \mathcal{C}_i'\right) = \sum_{j \in \mathcal{N}(i)} w_{ij} \left\| \left(\mathbf{p}_i' - \mathbf{p}_j'\right) - \mathbf{R}_i \left(\mathbf{p}_i - \mathbf{p}_j\right) \right\|^2 \tag{4.4}$$

After denoting $e_{ij} := p_i - p_j$, algebraic adjustments yield:

$$E\left(\mathcal{C}_i, \mathcal{C}_i'\right) = \sum_{j \in \mathcal{N}(i)} w_{ij} \left(\mathbf{e}_{ij}'^T \mathbf{e}_{ij}' - 2\mathbf{e}_{ij}'^T \mathbf{R}_i \mathbf{e}_{ij} + \mathbf{e}_{ij}^T \mathbf{e}_{ij}\right). \tag{4.5}$$

Since we want to minimize this energy function, the part that does not contain a rotation matrix R_i can be omitted. This will create Equation 4.6:

$$E\left(\mathcal{C}_i, \mathcal{C}_i'\right) = \underset{\mathbf{R}_i}{\operatorname{argmax}} \operatorname{Tr}\left(\mathbf{R}_i \sum_{j} w_{ij} \mathbf{e}_{ij} \mathbf{e}_{ij}'^T\right). \tag{4.6}$$

Next, S_i is denoted as a weighted sum of the edge products, and Singular Value Decomposition is performed in Equation 4.7.

$$\mathbf{S}_i = \sum_{j \in \mathcal{N}(i)} w_{ij} \mathbf{e}_{ij} \mathbf{e}'^T_{ij} = \mathbf{U}_i \Sigma_i \mathbf{V}_i^T \tag{4.7}$$

R_i can then be calculated from the SVD of S_i, given that the maximum of $Tr(R_i S_i)$ over all orthogonal rotation matrices R_i is achieved when:

$$\mathbf{R}_i = \mathbf{V}_i \mathbf{U}_i^T \tag{4.8}$$

Positions p'_i

To compute the optimal vertex positions from the given rotations, the gradient of $E(S')$ with respect to the positions p' is calculated. Partial derivatives are computed with respect to p'_i, and by setting these partial derivatives to zero, a sparse linear system of equations is obtained. After performing algebraic adjustments, the following is obtained:

$$\sum_{j \in \mathcal{N}(i)} w_{ij} \left(\mathbf{p}'_i - \mathbf{p}'_j \right) = \sum_{j \in \mathcal{N}(i)} \frac{w_{ij}}{2} \left(\mathbf{R}_i + \mathbf{R}_j \right) \left(\mathbf{p}_i - \mathbf{p}_j \right) \tag{4.9}$$

The left side of this equation represents the discrete Laplace-Beltrami operator, which is defined as the divergence of the surface gradient of a function, acting on p'. Therefore, the system of equations can be expressed in a straightforward form:

$$\mathbf{L}\mathbf{p}' = \mathbf{b} \tag{4.10}$$

Incorporating user-defined modeling constraints c_k into this system can be accomplished by removing respective rows and columns from L and updating the right-hand side with the values c_k.

Update

Upon the mesh S achieving a state of local energy, an update is conducted, updating the current mesh S with the values from the new mesh S'. This involves using vertices that were originally a part of the contour. Given the specific structure of the vertex array in Equation 4.1, it is sufficient to create a new contour from the first third of the array for implementing updates to the original contour.

Chapter 5

Implementation

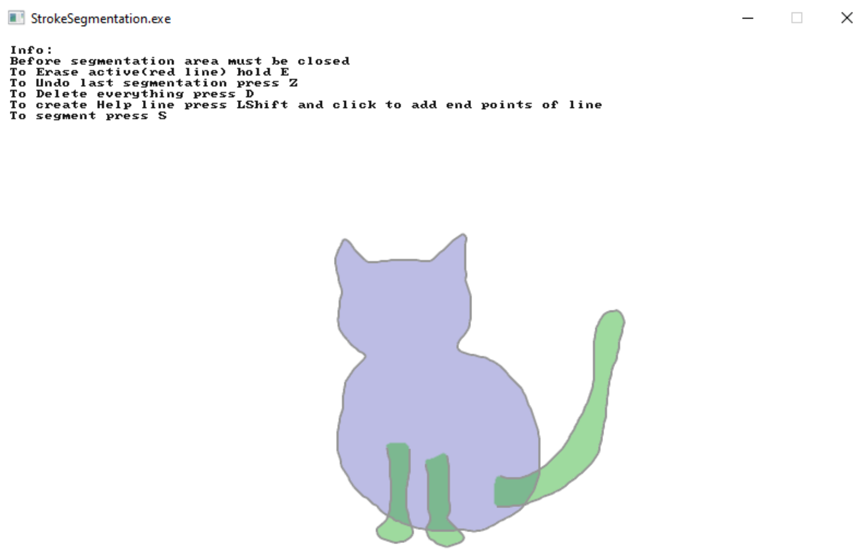
This chapter elaborates on the key aspects of image segmentation and ARAP deformation implementations within the MM project. Initially, it delves into the implementation details of image segmentation. Subsequently, the chapter describes the ARAP deformation implementation, with the help of the LibIGL library, explaining the creation of 2D objects from contours. The final part of the chapter talks about the challenges faced and the solutions found during the process.

5.1 First Trial Version

Before the introduction of new features into the MM program, test programs were developed. The primary aim was to experiment with and validate the functionality and applicability of various methods and libraries intended for integration into the main program. This preparatory testing proved crucial in preventing potential issues and perfecting the techniques for the best possible result.

The initial program in Figure 5.1 was specifically designed for the examination of image segmentation. It offered capabilities such as drawing, erasing, and the addition of helplines, thus facilitating a comprehensive testing environment. Crafted in C++, the program employed the Allegro library. Allegro provided an assortment of functions, including graphics, keyboard input, and timers, thereby creating a robust framework for the program. The segmentation class, developed during this phase, was later adapted for use in the MM program with minor modifications. The software lacked a user interface; instead, various functions were activated through designated keys. Essential information and certain program states were displayed directly on the drawing canvas to provide users with basic assistance.

Despite its modest size, the second program was instrumental in evaluating the LibIGL ARAP functionality. This examination was pivotal in gaining



```
Info:
Before segmentation area must be closed
To Erase active(red line) hold E
To Undo last segmentation press Z
To Delete everything press D
To create Help line press LShift and click to add end points of line
To segment press S
```

Figure 5.1: Screenshot of the trial image segmentation program.

an understanding of the behavior and limitations of this specific library. By embedding it into a more manageable test program, its performance could be monitored in isolation. This insight was invaluable when integrating this functionality into the broader framework of the MM program. The program rendered a simple 2D strip. The user could move specific vertices using keyboard keys and observe the resulting deformations, Figure 5.2.

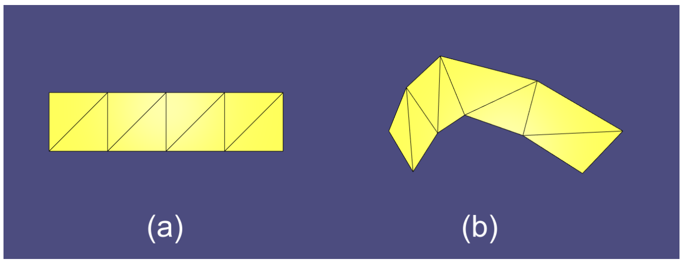


Figure 5.2: Screenshot of the trial deformation program.

In conclusion, these test programs were not merely a preliminary step in the development process but also a significant contributor to the MM. They served as a tool for identifying potential pitfalls, comprehending the performance of third-party libraries, and refining the code, thereby resulting in a more robust and efficient final product.

5.2 Monster Mash in Code

In this section, we will delve deeper into the implementation of Monster Mash. Its code is written in C++ combined with JavaScript, HTML, and

CSS for the user interface. To compile the code, emscripten is used, which is a toolchain that enables developers to compile C/C++ code into JavaScript so that it can be run in a web browser. The toolchain works by converting LLVM bitcode, which is an intermediate representation of compiled C/C++ code, into WebAssembly (wasm). WebAssembly is binary format code that is not intended to be written by hand but instead is intended to be generated by compilers from higher-level languages. Once the bitcode is compiled to wasm, Emscripten can generate a wasm module, which is a binary file containing the compiled code. The wasm module can then be loaded into a web page and executed in a web browser using a JavaScript. This allows developers to write C/C++ code that can run in a web browser without needing to rewrite it in JavaScript. Additionally, because wasm is a binary format, it can be smaller and faster than equivalent JavaScript code.

The code is designed to be versatile, allowing for direct compilation into a desktop version using C++ code. This desktop version, primarily intended for testing, lacks a graphical user interface; the required operations are triggered via specific keys. This configuration is ideal for testing, as the compiled wasm code does not support C++ debugging. However, despite these differences, both the desktop and WebAssembly versions function identically in all other aspects, save for the user interface. Thus, users could expect the exact features across both platforms.

For the purposes of this project, I was provided with a partial segment of Monster Mash that exclusively includes the first component of the program - the drawing part. Although the program appears identical to the original version, it lacks the requisite code for the inflation, deformation, and animation functionalities. Consequently, when launched, the aforementioned components of the program do not yield any output.

The implementation uses OpenGL for rendering and relies on various data structures and libraries to manage pictures and 3D models. Some key components include: Image handling using the `Imguc` class, a custom image class that stores image data and provides functions for manipulation. The Eigen library for linear algebra operations is useful for handling transformations and camera operations, and the Libigl library for geometry processing.

5.2.1 Project Structure

The project comprises several C++ classes, as well as `main.html`, `main.css`, and `main.js` files. The `main.html` file structures the web page, while the `main.css` file defines styles for various elements such as buttons, images, and containers. The JavaScript file, `main.js`, provides frontend functionality to the web page. For instance, it adds event listeners to the buttons and invokes functions defined in the C++ classes.

To facilitate the communication between the JavaScript and C++, we define functions in `main.cpp`. These functions correspond to operations like saving/loading a project or switching modes, and will be invoked from the `main.js` file. To ensure these functions are not omitted during the optimization stage of compilation, we prefix them with the EMSCRIPTEN_KEEPALIVE keyword.

The `main.cpp` file is then compiled into a `main.wasm` file using Emscripten. This process also generates a `main.js` file responsible for loading the WebAssembly module.

Here's an example of a function definition in `main.cpp`:

```
EMSCRIPTEN_KEEPALIVE
void saveProject() {
  drawWindow.saveProject("/tmp/mm_project.zip");
}
```

Listing 5.1: Example of function definiton from main.cpp

The generated `main.js` file should be included in the `main.html` file. This inclusion ensures the JavaScript file handles loading the corresponding `main.wasm` file.

Next, in **texttt.js**, we invoke the C++ function via the Module object:

```
$('#buttonSaveProject').click(function() {
  Module._saveProject();
});
```

Listing 5.2: Example of call function defined in C++

This way, we can call C++ functions from web pages through JavaScript.

The C++ part consists of several classes worth mentioning:

- `drawWindow`: This class extends MyWindow and represents the main window of a drawing part of Monster Mash.
- `loadSave`: This class provides functions for saving and loading data.
- `main`: A main class that defines functions for JS code.
- `myPainter`: A utility for drawing graphical primitives. It serves as an abstraction layer for painting operations, offering functionality to draw various shapes such as lines, rectangles, ellipses, and arrows, as well as handling color and thickness settings.
- `myWindow`: This class is designed to manage window rendering, specifically using the Simple DirectMedia Layer (SDL) library and OpenGL for rendering.

- `commonStructs`: Important structures and macros are defined here.

Going forward, unless specified otherwise, any references to algorithms, structures, and other elements will implicitly pertain to those from the `drawWindow` class.

5.2.2 Data Structures

During the model creation process, various images are stored. As the user draws a stroke, it is rasterized onto the currently active layer (`currOutlineImg`). Each drawn stroke in Figure 5.3a results in the creation of a new layer. To store a single layer, the software generates two images within the `Imguc` structure. The first image consists of the outline (see Figure 5.3b), while the second contains the region image (see Figure 5.3c), in which the drawn region appears in white and the background in black. If the drawn stroke does not form a closed region, the program connects the first and last points of the stroke, using a different color for the connected portion in the outline image. Although gray is the default color used for the closing region line, it has been intentionally colored green in the example for improved clarity. Outline and region images are stored in their corresponding `deques` (`outlineImgs` and `regionImgs`), which now represent a single layer of the artwork. An auxiliary `deque` is employed (called `layers`) to facilitate layer addressing, as user-deleted layers can complicate addressing layers. Before rendering, all outline and region layers are merged into `mergedOutlinesImg` and `mergedRegionsImg`. All aforementioned objects, along with many others, are contained within the `ImgData` structure, which maintains images and the current drawing state (e.g., the selected layer).

```
struct ImgData = {
        deque<Imguc> outlineImgs, regionImgs;
        deque<int> layers;
        Imguc mergedOutlinesImg;
        Imguc mergedRegionsImg;
        Imguc currOutlineImg;
        int selectedLayer;
        ...

};
```

5.2.3 Program Pipeline

The pipeline of the drawing part of the current MM program proceeds as follows:

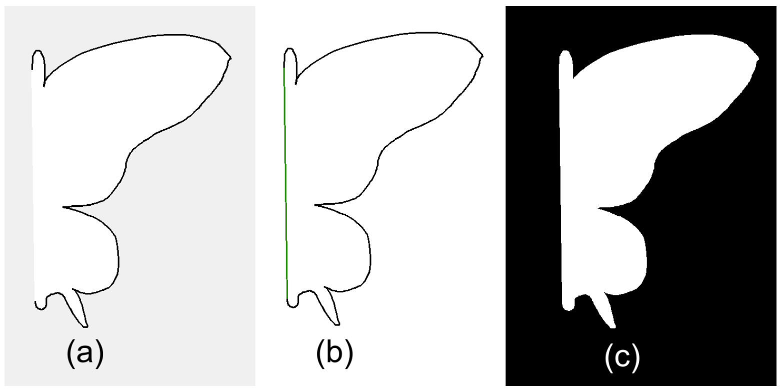


Figure 5.3: Image (a) shows drawn art in the MM, (b) and (c) are stored outline and region layers.

1. The user launches the application, and the `drawWindow` is created with UI components initialized.

2. The user begins drawing a stroke on the canvas.

3. `mousePressEvent()` is triggered, drawing a line (a segment of the stroke) from the previous coordinates to the current mouse coordinates.

4. As the user moves the mouse, `mouseMoveEvent()` is invoked, similarly drawing a line on the current canvas.

5. When the user releases the mouse button, drawing of the current stroke ceases, triggering `mouseReleaseEvent()`. This method's functionality varies depending on other keys pressed and the duration of the pressed button. If the button is pressed briefly, it is considered a short click, leading to the selection or deselection of a layer. If a double click is detected, the method duplicates the selected region, provided it has not already been duplicated. If neither of the above occurs, the method stores the newly created region from the stroke within `ImgData`. To create a filled region, MM uses flood fill algorithm [23].

6. Each time `mouseReleaseEvent()` ends, `recreateMergedImgs()` method is invoked as the final step. This method performs the task of merging all the outline and region layers into two separate layers. One layer contains all the regions, while the other layer comprises the merged outlines. Subsequently, these layers are pushed into OpenGL, enabling their display on the screen.

7. Once all the desired layers have been created, the sketch reaches its final form. Subsequently, the program proceeds to the next phase of the drawing pipeline, specifically the inflation and animation stage, which represents the culmination of the drawing pipeline.

5.3 Proposed User Design

User design is important when adding new features to a program, as it helps make the software easy to use and understand. A good user design helps users navigate the program and enjoy their experience with it. When updating an existing program, it is critical to ensure that new functions will align with older functionality and users' needs.

In the current MM version, the user can only draw strokes, where each stroke represents one shape. There is also the possibility of redrawing selected shapes from scratch. This design does not allow users to create complicated shapes, and moreover, it does not allow for errors. If the user makes a mistake, they have to redraw the shape from scratch.

With future features in mind, we came up with a design similar to the current MM but with more of the necessary features used in sketching software. The proposed UI will have features typical for this type of software, like drawing strokes, erasing strokes, deforming strokes, and an undo button. There will also be a function that typical drawing programs rarely have: sketch simplification. This design needs to overcome the issue in the previous version of each region allowing only one stroke, but it also needs to produce output in the same format as the current version of MM so that the next part of the program, the inflation, will work correctly.

The proposed design can be seen in Figure 5.4. Unlike the current MM, the new design includes a toolbar (marked with a green box) with the following functions:

(1) Normal drawing: Users can draw lines on the canvas.

(2) Helpline mode: Users can draw lines that will not be visible in the final result, but that can be used to close open shapes.

(3) Normal eraser: Upon clicking, erases both help and normal lines.

(4) Simplification: The function automatically simplifies the currently drawn sketch.

(5) Undo button: Reverts the last sketch simplification step.

5.3.1 Expected Use

The process of creating artwork goes like this: First, the user draws a shape they like and can erase any mistakes while drawing. If the shape isn't fully enclosed, the user will close it using help lines. Next, they use the "Simplify

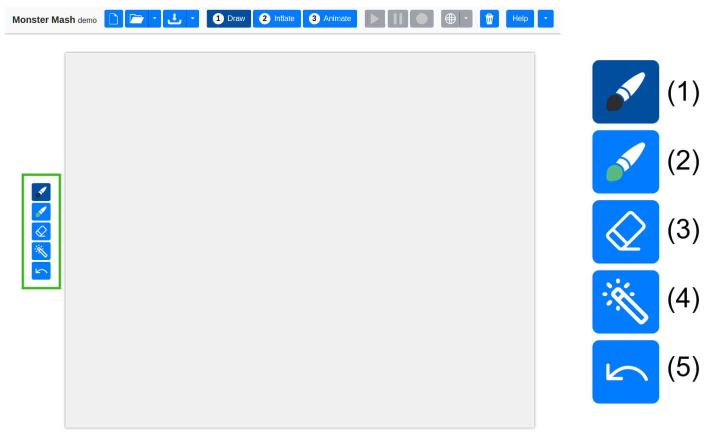


Figure 5.4: New design for Monster Mash.

Image" function to simplify the shape. If the simplified result isn't satisfactory, they can undo this process and adjust the sketch.

After the sketch is simplified, the user can change the shapes. This can be done by either dragging the shape's borders or by using control points, which are added by a quick click on the desired edge of the shape.

Engaging solely with the drawing component of the software posed specific difficulties. Inflation and animation testing was not feasible within the new version because only the drawing part was provided. To solve this problem, test projects were exported to the original version of MM. First, the project was saved. Following this, the stored project was loaded into the original MM. Additional features of MM, such as inflation and animation functionalities, then become accessible. This procedure enabled comprehensive testing of the entire MM pipeline.

5.4 Integration into Monster Mash

In the realm of software development, the task of adding new features to an existing, well-functioning program is more than meets the eye. It's not simply a matter of developing a fresh component and seamlessly integrating it into the system.

The complexity of this task arises from the fact that a program isn't a loose collection of features, but an intricately interconnected network. Each element within the program is interdependent, working in harmony with others to ensure overall functionality. When we introduce a new feature, it's like inserting a fresh piece into an already completed puzzle. This task requires meticulous precision and often necessitates altering the existing pieces to accommodate the new ones.

This process may require significant changes or even complete rewrites some parts of the original code, making it a challenging task. While it's true that software should ideally be designed for extensibility from the start, the realities of shifting deadlines, evolving project requirements, and an increasing understanding of the problem space often complicate this objective.

Before introducing a new feature to MM, it was necessary to determine what output was anticipated in the next phase of MM: inflation. Over time, it became apparent that the only crucial elements for the subsequent part were the outline and region layers, which remained unchanged. As the gradual implementation of new features into MM progressed, a majority of the code underwent modifications.

5.4.1 Modifications to Monster Mash

This section elucidates the modifications made to the MM program to incorporate new features. Specifically, two new classes have been introduced for managing image simplification and deformation, which are detailed in subsection 5.4.2 and subsection 5.4.3.

Storing Lines

The initial modification in the `drawWindow` function involved storing lines in a structure. Previously, lines were not stored. Each line is now represented by the following class:

```
class Line {
int x1;
int x2;
int y1;
int y2;
bool isHelpLine;
}
```

Distinct structures were used for lines belonging to the current sketch and for those from rasterized layers. This was because the current sketch lines, being deletable by the user, needed unique, easily removable indices, leading to their storage in a map.

```
std::map<int, Line> current_strokes;
```

Contrarily, segmented strokes didn't require a unique index, so they were stored in an array for each layer. For every layer containing a simplified sketch, a `DeformCurve` object was created. This object stores the structures required for ARAP deformation, such as the 2D strip. These structures were also placed in a `deque` for unified access, similar to MM's image-storing approach.

```
deque<vector<Line>> segmented_strokes;
deque<DeformCurve> deform_curves;
```

Program States

In the original `drawWindow` function, there were only two states: *DRAWING* and *REDRAWING*. To accommodate the new features, additional states were introduced, enabling the user to switch between them efficiently. The *DRAW_MODE* allows users to sketch new lines freely on the current layer and is active at the program start. *ERASING_MODE* facilitates the deletion of lines from the current layer. *DEFORM_CURVE_MODE*, activated post-image simplification, lets users deform the simplified lines, but not the current sketch lines. *HELP_LINE_MODE* enables users to add lines to the current sketch.

Line Erasure

A key feature is the ability to erase unwanted lines. An erase map structure was created to facilitate this. Initially, the concept was to develop a 2D array where each pixel represented the line index, but multiple strokes can overlap in the same location. Consequently, the final erase map consists of three nested arrays, with the first two representing the image and the last containing drawn line indices.

```
vector<vector<vector<int>>> erase_map;
```

To erase pixel at coordinates $[x, y]$, the function finds all IDs in these coordinates (`erase_map`$[x][y]$) and removes lines with these IDs from the `current_stroke` structure. The eraser is supposed to delete lines in a broader range. To accomplish that, we do the same routine through intervals $(x - s, x + s)$ and $(y - s, y + s)$ for x and y coordinates, where $s/2$ represents the eraser size.

Undo Function

The incorporation of an undo function significantly enhances user experience and flexibility within Monster Mash. During the process of image segmentation, results may occasionally deviate from the user's expectations or intentions, as detailed in subsection 5.4.2. To tackle this issue, an undo function was implemented, allowing users to revert to the last segmentation step immediately after its execution.

The undo function stores the current sketch's associated state before the simplification operation. This state includes all the critical details of the sketch, such array of `strokes`, `erase_map`, etc. In the event of an undo operation, this state is retrieved, and the current sketch state is replaced with the stored state, effectively reverting the sketch to its pre-segmentation state.

Area Fill

In order to enhance the performance of the MM, a change was made to the area filling process. Previously, the fill algorithm was applied to the exterior area, which is typically significantly larger than the area of a drawn layer, resulting in longer execution times. However, to color the interior region of a given layer, it is first necessary to locate a seed pixel that is undeniably within the confines of this area. While the positions of lines change as users deform them, the orientation of each line's left and right sides remains constant. This means that if the left area was initially the interior area, it will remain so even after deformation.

To determine the orientation, the exterior area is initially filled in black using the flood fill algorithm [23]. A line from the sketch's contour is then selected, and its midpoint pixel is located. Given the midpoint is inside the line, there will surely be a black pixel in these coordinates. From this pixel, the algorithm proceeds in both directions along the line's normals, as illustrated in an Figure 5.5. During this process, the pixel color is checked in both directions. The first side to encounter a white pixel is deemed the interior side because the exterior area is filled with black. This side is then labeled as the interior side.

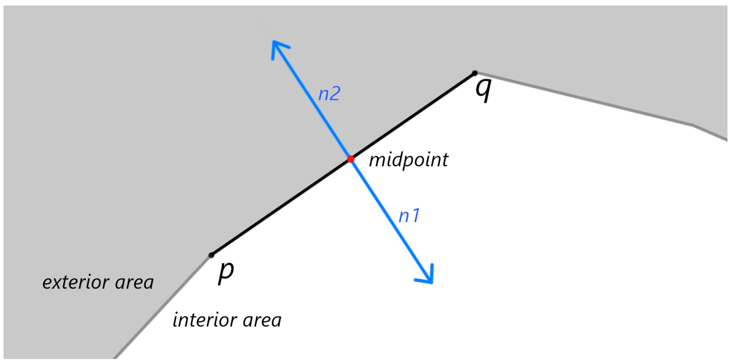


Figure 5.5: Illustration of determining the correct side of the line.

Once the correct side has been identified, it becomes easier to rapidly locate an interior pixel. When updating layers, a white image with a black contour is provided, and the previous procedure is repeated. The line's center is calculated, and the algorithm proceeds in the direction of the normal in previously identified side. The first white pixel encountered is the desired seed. It's crucial to emphasize that the sequence of all lines, as well as the sequence of the points that constitute a line, remain unchanged. These rules ensure that the search will always be conducted on the correct side of the line.

UI Adaptations

Adapting the new features to fit within the user interface of both web and desktop versions of Monster Mash presented unique challenges and considerations. Given the differences in UI paradigms between web and desktop environments, the integration of new features required distinct approaches for each platform.

For the desktop version, the integration primarily involved the addition of new key listeners. These listeners were designed to respond to specific user key inputs, triggering the appropriate functions in response. The design process of these listeners had to take into account potential conflicts with pre-existing key assignments. This was necessary as many keys, like "E" (potentially for erasure) and "S" (potentially for segmentation), were already associated with other functions.

The web version, however, required a more complex adaptation process. New UI elements, including buttons, had to be created first using HTML. These elements were then made functional by attaching JavaScript listeners to them. Additionally, these new UI elements required styling to ensure consistency with the existing web UI design. The integration of these new features into the web version also involved providing communication between the JavaScript code and the underlying C++ functions through the `main.cpp` file.

Auxiliary Libraries

The implementation of the new features in Monster Mash involved the use of auxiliary libraries to enhance development efficiency and facilitate specific functionalities. One such library was EasyBMP, which was utilized during the development process.

EasyBMP is a simple, cross-platform, open-source C++ library designed for easy reading, writing, and manipulation of bitmap (BMP) image files. In the context of this project, it was employed to facilitate easy image saving, which proved to be invaluable, particularly during the testing and debugging phases. By leveraging EasyBMP, it was possible to quickly and conveniently save images representing intermediate steps of sketch segmentation. It also served as a way to visually document the procedure, which could be useful for presentations, reports, or future reference.

5.4.2 Image Segmentation Implementation

The class in focus is named `segmentation`. This class emerged during the implementation of a trial program for segmentation testing. The class leans more towards a procedural base rather than object-orientation, largely because its primary function is to transform one type of data into another. The main public function is illustrated as follows:

```
vector<Line> segment_drawing(Imguc &image,
                             vector<Line> &help_lines);
```

The function accepts as input parameters the current layer which contains the drawn sketch and an array of help lines. The function returns an array of the segmented sketch. Instead of a list of strokes, which would require rasterization for image segmentation, it directly takes an image. Since the MM already performs rasterization, this approach eliminates unnecessary rasterization within the segmentation process.

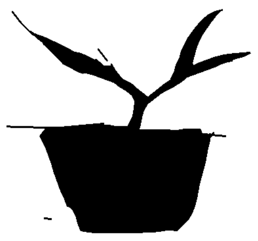
To execute image segmentation, the algorithm outlined in section 3.2 is utilized. The GridCut library was employed to solve the min-cut problem. This library provides a min-cut solver for grid-structured graphs. Following the initialization of the grid and setting weight to the edges, all that remains is a call the `compute_maxflow()`. Subsequently, the library separates all nodes into two sets. Pixels belonging to the first set are colored white, while those in the other set are colored black, thus creating a new image.

Due to the requirement of the deformation function to create a 2D strip out of the lines, exceedingly sharp or thin parts of the sketch can produce errors. The mesh vertices could overlap at these parts, potentially causing the ARAP function to fail. To mitigate this, the segmentation result undergoes two morphological operations, specifically erosion followed by dilation. A 3x3 matrix, filled exclusively with ones, was employed for both morphological operations.

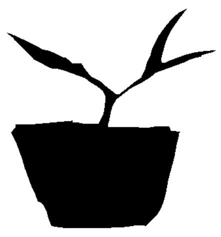
Dilation and erosion are morphological operations widely used in image processing and computer vision. Dilation is used to expand the boundaries or regions of objects in a binary image. This operation places the given mask over each pixel in the image and compares it with the corresponding pixels. If any "on" pixels (designated by "1") in the mask overlap with an "on" pixel in the image (stroke pixels), the resulting pixel in the dilated image is set to "on". For the erosion, the rule is: if even a single "off" pixel (designated by "0") in the mask aligns with an "on" pixel in the image (stroke pixels), the corresponding pixel in the eroded image is converted to "off". This action effectively shrinks the image objects by trimming their edges.

Erosion, when applied to the image, causes all thin parts of the edge to disappear, but this also reduces the overall size of the object. For restoration, dilation is utilized, which returns the shape of the area to its previous state.

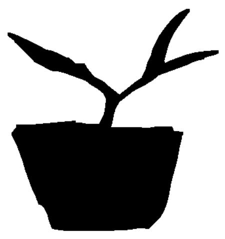
Figure 5.6d illustrates two layers, Figure 5.6a and Figure 5.6c, with the blend mode "difference" applied. This demonstrates that only the thin parts of the image are eliminated, while the rest remains unchanged.



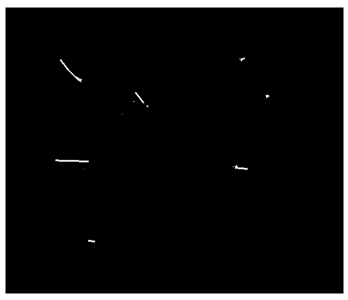
(a) : Picture after image segmentation.



(b) : Picture after erosion is applied.



(c) : Picture after dilation is applied.



(d) : Picture **(a)** and **(b)** processed with the difference blend mode.

Figure 5.6: Example of how an image is processed using the application of erosion and diffusion techniques.

STA Algorithm

The STA algorithm, as described in subsection 3.2.1, is employed to identify the contour. This algorithm, sourced from the MM project, has been slightly adapted. The variation in this implementation lies in the need to separate the traced contour into two categories: normal lines and help lines, and to split the contour into a set of lines.

As lines are added, the algorithm traces along the edges of the filled image. The current length of the traced line is calculated during this process. If the line exceeds a predefined constant MIN_LINE_LEN, it is added to the array of lines. However, before this addition occurs, it must be determined whether the line qualifies as a help line. If any help lines added by the user form part of the traced contour, they will be positioned identically to the traced contour. Therefore, while tracing a line, it is possible to ascertain if a part of the line intersects a help line. Encountering a single pixel containing a help line is sufficient to label the line as a help line.

To verify if an intersection with a previously drawn help line has occurred, an auxiliary image is generated, where only help lines are placed. The lines are rasterized using the Bresenham algorithm [24], whose implementation was sourced from [25]. The choice of this algorithm is driven by the class's design intent of minimal dependence on multiple libraries within the MM. This intent was further strengthened by the fact that the class was developed during the implementation of a trial program for segmentation testing. Following to rasterization, dilation is performed with the mask

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

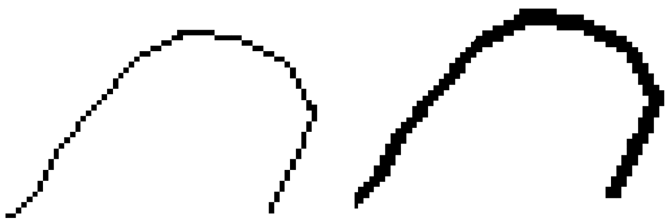


Figure 5.7: Example of a line that has undergone the process of dilation

The impact of this function on the image can be observed in Figure 5.7. The dilation operation has enlarged the boundaries of the original image's line by one pixel in all directions, resulting in broader and larger regions. This assists in labeling help lines more accurately, as even a 1-pixel shift in the contour could otherwise result in a line being labeled as a normal line.

Setting Values

Determining the optimal constants for edge weights proved to be a challenge. Ultimately, the weights for nodes connected to the terminals were set to 10000. The weights between nodes in the grid (representing pixels) were determined by the following function:

```
short weight_func(int p1, int p2) {
  short cap;
  // case 1: one of the pixels is black
  if (p1 == 0 || p2 == 0 ) {
    cap = 3;
  // case 2: both pixels are white
  } else {
    cap = 17;
  }
  return cap;
}
```

Identifying the correct values was a tough task. For instance, increasing the weights for case 1 enabled the image to close larger holes, a potentially beneficial feature, yet it simultaneously made the image more tolerant to closing unwanted holes, as illustrated in Figure 5.8.

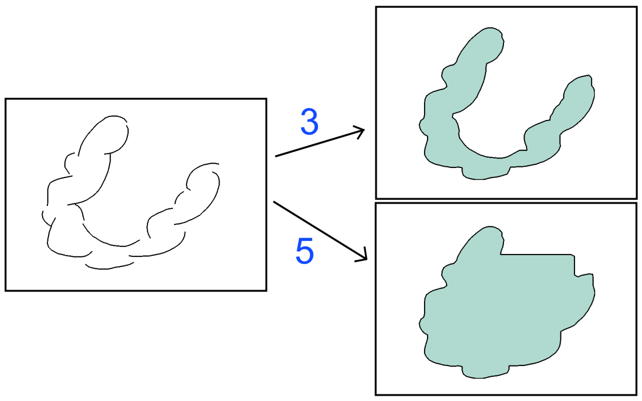


Figure 5.8: Example of how increased variable `cap` can change segmentation computation.

5.4.3 As-Rigid-As-Possible Deformation Implementation

In this project, the implementation of ARAP was enabled by the LibIGL library. LibIGL provides a robust and efficient ARAP implementation for the deformation of 2D objects. The advantage of the LibIGL implementation of ARAP is that it's optimized for performance, allowing for real-time interactions. The choice to use LibIGL over Monster Mash (MM) also arose due to MM's inherent constraint of 3D object implementation.

The primary function for deformation, facilitated by LibIGL's ARAP implementation, is invoked from the `drawWindow` class. The necessity for this approach is due to the structural design of LibIGL's ARAP, which allows only a single instance creation.

To transform given contour into a 2D object and to generate the necessary structures, the `DeformCurve` class was designed. Each instance of the `DeformCurve` class represents a single closed contour, which corresponds to one layer of the sketch. The main data structures associated with this class are:

```
  Eigen::MatrixXd bc;
  Eigen::MatrixXd strokeStripV;
  Eigen::MatrixXi strokeStripF;
  Eigen::VectorXi strokeConstraints;
```

The `strokeStripV` matrix represents the vertices of the 2D object, where each row corresponds to a unique vertex. The `strokeStripF` matrix contains the faces of the 2D object, with each row representing a distinct face. The `strokeConstraints` vector is a list of constraints added by the user, containing the IDs of constrained vertices. The `bc` matrix is akin to `strokeConstraints`, but instead of holding vertex IDs, it stores the coordinates of the corresponding vertices.

Upon initialization, the `DeformCurve` class populates `strokeStripV` and `strokeStripF` with the appropriate values. The `bc` matrix is updated just before executing the ARAP deformation. The technique for creating vertex coordinates is elaborated in section 4.2.

The `strokeConstraints` vector is updated when the user adds new constraints. When a user attempts to create a constraint, the program identifies the closest vertex to the queried click coordinates. If the computed distance exceeds a predefined threshold, the constraint creation is aborted. This design choice is intended to maintain the integrity of the deformation by preventing the imposition of constraints that could potentially distort the contour. Constraints can be categorized into two types - temporary and permanent. Permanent constraints are established with a short click, whereas temporary constraints are generated when a user drags the contour without any prior constraint creation at the specific location. A temporary constraint is created at the `mousePress()` event and is subsequently deleted at the `mouseRelease()` event.

After the `strokeConstraints` vector has been updated, it is essential to notify the ARAP instance. This can be accomplished by executing the following:

```
arap_precomputation(strokeStripV, strokeStripF,
                    strokeStripV.cols(), strokeConstraints,
                    arap_data);
```

The `arap_data` object is an instance of the ARAP Data struct, which is used to store information required for ARAP deformation. When user interaction involves moving the line, the moved vertex is updated in the `strokeConstraints` structure, and the `bc` array is refreshed. Subsequently, the function to compute the new position is invoked.

```
igl::arap_solve(bc, arap_data, strokeStripV);
```

This function updates the data in `strokeStripV`, which will be rendered in the `drawWindow`. Notably, only the first third of the stroke strip array is rendered because it contains required vertices that are a part of the contour.

5.4.4 Challenges and Solutions

The challenge of delving into the MM code proved formidable, requiring a significant amount of time to understand the diverse naming conventions, the inherent logic within the code, and the structural organization. The task of compiling a C++ project laden with numerous dependencies presented additional hurdles.

Initial attempts at building the project in Windows resulted in impossible obstacles, necessitating a shift to Linux. Following the successful build in this alternative environment, a new task emerged: the integration of the project into a development environment. This step was crucial to facilitate debugging and simplify the start-up process. Fortunately, the situation improved as an instruction manual for building MM was provided to me by the creator of MM.

The introduction of the deformation feature demanded significant changes in the program pipeline. With this feature, users can deform the layer by dragging its contour, implying that all layers must be re-rendered after each mouse movement due to changes in layer intersections and overlaps. As explained in subsection 5.2.3, the `recreateMergedImgs()` function was previously called after each `mouseRelease()` event. However, calling this function after every `mouseMove()` event significantly decreased performance due to slow memory access in the `Imguc` structure. In light of many MM libraries depending on this structure, changing it would be overly complex.

Fortunately, improvements such as invoking `recreateMergedImgs()` only when the user moves the selected region at least one pixel and computing minimum and maximum coordinates for each layer to render only the essential screen portions enhanced the program's speed. The most significant enhancement in speed was achieved by utilizing a faster version of the flood fill algorithm, which refrains from the use of a stack structure in favor of relying on the call stack for recursion. Furthermore, this optimized approach focuses on filling the interior area rather than the exterior, a process detailed in Section 'Area Fill' (refer to subsubsection 5.4.1).

Interestingly, deformation in the web version works faster than the desktop one, but the desktop version provides smoother drawings.

Chapter 6

Experiments

This chapter presents a series of comprehensive experiments designed to evaluate the real-world performance and effectiveness of the advanced features incorporated into the Monster Mash tool. The principal purpose of these experiments is to critically assess the enhancements brought about by the newly integrated As-Rigid-As-Possible (ARAP) deformation algorithm and the sketch simplification functionality, which is achieved through image segmentation.

While the focus primarily lies on ARAP deformation and sketch simplification, an erasing function has also been added to the tool. Although this function may not be the central subject of the study, its importance in contributing to a complete user experience is recognized.

The initial series of experiments concentrate on the sketching part of the workflow, particularly the interactive process involving the deformation and simplification of sketches. This investigation aims to discern how these additions could potentially revolutionize the creative process in a practical setting, using the newly modified version of Monster Mash as the testing platform.

Subsequent tests aim to evaluate the enhancements to the sketching tool in the context of the complete workflow. This involves importing sketches crafted with the modified version into the current Monster Mash version. The primary objective is to test the compatibility of the new sketching features with the pre-existing inflation and animation functionalities of Monster Mash.

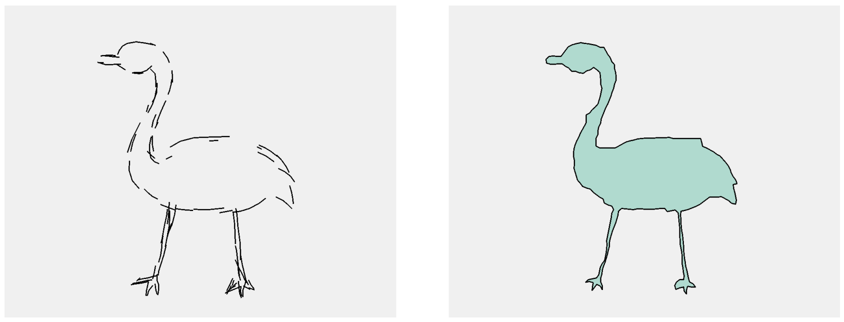
These experiments aim to provide an evaluation of the implementation, shedding light on its merits and areas that could be improved. The conclusions derived from these findings will not only offer a robust examination of the enhancements but also guide future improvements in this area.

To facilitate simpler testing, the compiled project was hosted on a web server, enabling convenient access. As a result, the entire testing process could be performed across merely two pages - the original MM and MM featuring new enhancements.

6.1 Sketch Simplification

This section is dedicated to the examination of the sketch simplification tool's implementation through various test scenarios.

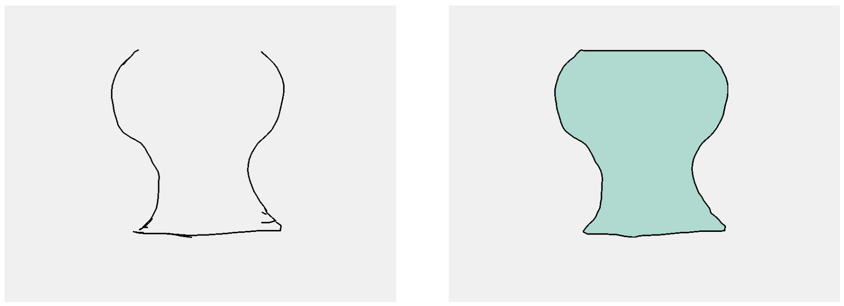
In our first test case, we inspected the tool's ability to fill holes in sketches, as exemplified in Figure 6.1. The image segmentation process proved capable of filling numerous gaps in the sketch. However, a key drawback became evident: the process fills the holes using lines parallel to the x and y-axis. This outcome can be attributed to the grid-structured graph employed by the tool, which leans on Manhattan distances instead of Euclidean ones.



(a): **(b):**

Figure 6.1: Results of simplifying the goose image.

The subsequent sketch in Figure 6.2 contains a considerable hole. In this instance, the algorithm was successful in closing the hole agilely. However, Figure 6.3 introduces sketches featuring holes requiring closure through parallel lines, representing the worst-case scenario for this algorithm.



(a): **(b):**

Figure 6.2: Results of simplifying the vase with a large hole.

Additional testing was performed on a sketch abundant with strokes inside and some noise outside, Figure 6.4. The results were quite satisfactory: the

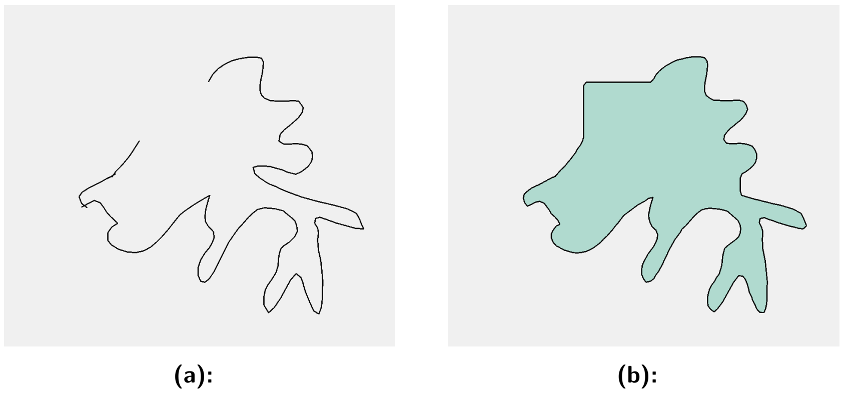


Figure 6.3: Results of simplifying the random shape with a diagonal hole

presence of background noise failed to interfere with the segmentation process, revealing the robustness of the tool. Furthermore, even with a large number of strokes, the tool managed to maintain its performance without encountering any issues or significant changes in execution time.

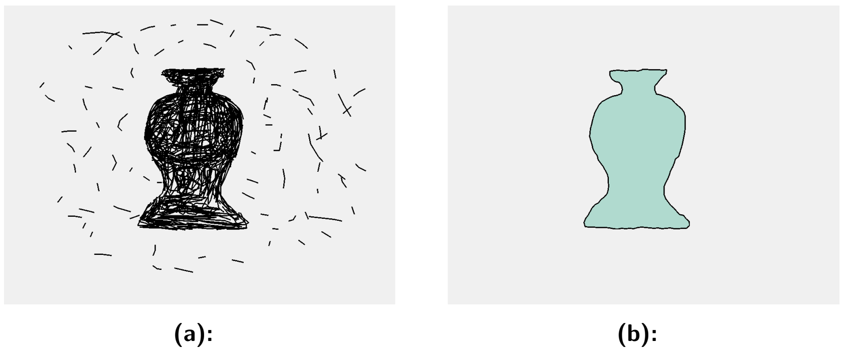


Figure 6.4: Results of simplifying the vase with a lot of noise.

On the other hand, it's worth noting that this approach tends to prioritize the shortest paths between strokes. Consequently, there are instances where the segmentation result may appear as depicted in Figure 6.5. (To aid in creating this figure, a supplementary image was imported and traced taken from [26].)

This test aimed to investigate whether the algorithm consistently recognizes help lines. The region produced from these help lines, albeit without borders, reflects their shape. Our testing shows that the tool can competently detect segments of help lines, as demonstrated in Figure 6.6.

Below are other results of the more typical drawing process recorded step by step. The result can be seen from Figure 6.7 through to to Figure 6.11.

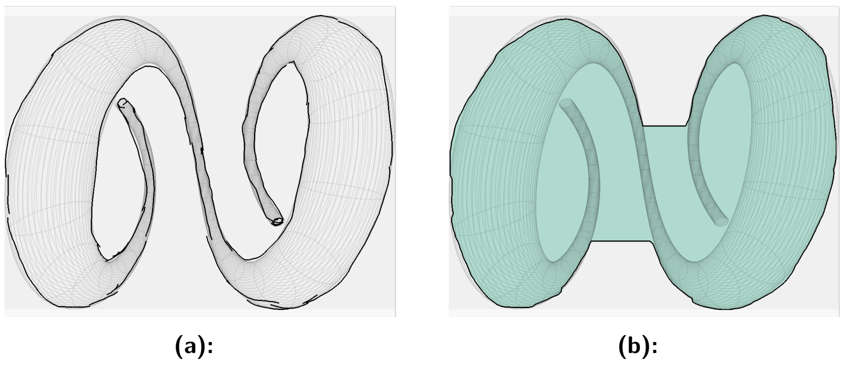


Figure 6.5: Results of simplifying the image with a spiral.

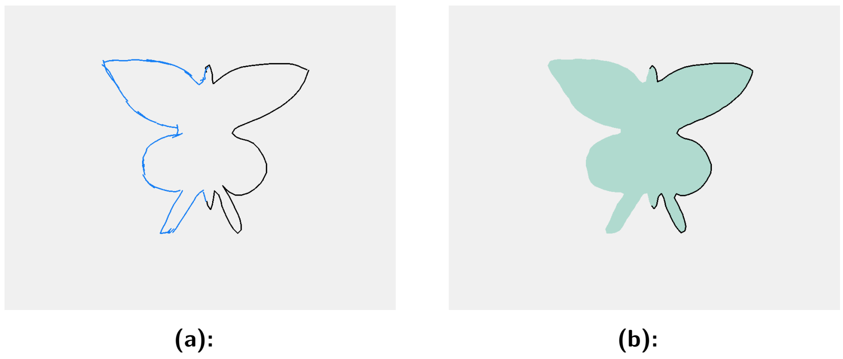


Figure 6.6: Results of simplifying the image of a butterfly using help lines.

6.2 Sketch Deformation

In this section, attention is paid to the nuances of movement and manipulation in both 2D and 3D models. The depiction of these movements presents a challenge due to their dynamic nature. To overcome this, in some cases, the approach taken involves the incorporation of vectors that represent mouse movements, providing a visual representation of the transformation process.

In the Figure 6.12, the moved image is color-coded in purple for easier distinction. This color distinction provides a clear contrast and makes possible a more comprehensive understanding of the changes being made during the deformation process. The process of manipulating the object is compared to manipulating thin wire. It is observed that using only one control point for manipulation tends to move the entire sketch, while simultaneously causing deformation in certain areas. The most significant deformation is seen in the part that is being dragged.

As more control points are introduced, the object begins to deform in a manner that aligns with expectations. For instance, a larger deformation is applied as demonstrated in Figure 6.13, while minor deformations are specifically targeted towards the tail and hands of the monster in Figure 6.14.

The substantial changes were more challenging to accomplish. Occasionally, some unexpected components shifted, and a significant number of control points were utilized to prevent this.

A comparison between the ARAP deformation in the new version to the 3D ARAP deformation in the original version of MM is concluded. When these deformations are executed in MM (see Figure 6.15b, Figure 6.16b), results similar to those seen in Figure 6.15a and Figure 6.16a are achieved. It is observed that making larger changes is considerably simpler with a 3D ARAP deformation than with a 2D one. Minor changes, however, present a similar level of complexity in both dimensions. A noteworthy observation is that the 3D version appears to provide a more stable version for manipulation and deformation.

In this comparison, represented by Figure 6.17, the symbol "S" is deformed in a similar manner in both versions. While the results appear similar, the 3D version presents more stable outcomes. Some parts are seen to crisscross in the 2D version, preventing it from filling the area.

In this test, a deformation that widens the symbol "S" is applied. As shown in Figure 6.18, the 3D version does not have the capability to deform the image in the same manner as the 2D deformation. This observation underscores the differences in deformation capabilities between the two versions

The deformation tool can prove exceptionally beneficial in instances requiring the sketch to possess multiple similarly-shaped layers, such as in the case of an octopus sketch. To illustrate this, consider an example where only the initial two tentacles are drawn, even separately from the main sketch (see Figure 6.19).

Upon the completion of two tentacles of the octopus, we duplicate its appendages. The remaining task is achieved by utilizing the deformation tool to reshape the copied tentacles and reposition them appropriately. Consequently, the entire procedure resembles an artistic approach akin to the sculpting of a statue.

The next section will provide examples of a more typical deforming process. The results can be seen from Figure 6.20 through to Figure 6.25.

6.3 End-to-End Pipeline Scenarios

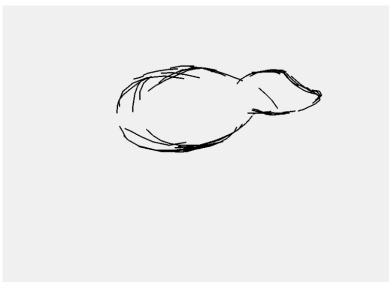
Figure Figure 6.26 showcases an example of the full pipeline's application, starting from sketch creation in the new version of MM to the 3D animation in the original version of MM.

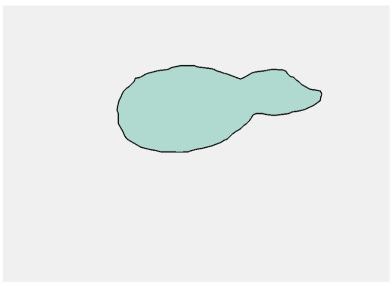
6.4 Exploring New Possibilities

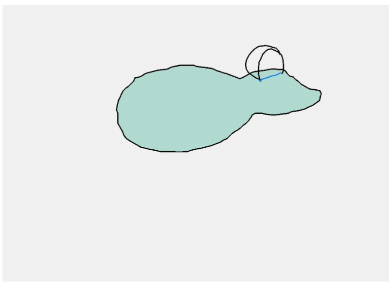
The final test was dedicated to demonstrating the new capabilities of MM. There are certain shapes that could not be created in the older version of MM, but are now feasible with the new features.

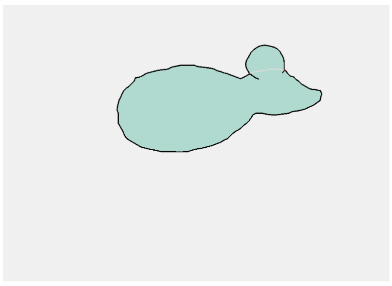
The first example involves the addition of new layers containing holes. Shapes such as a half-torus can be created near the main body. After loading the sketch into the original MM and inflating it, shapes with holes are produced (see Figure 6.27). This was impossible in the previous version because this type of shape requires at least two strokes to draw.

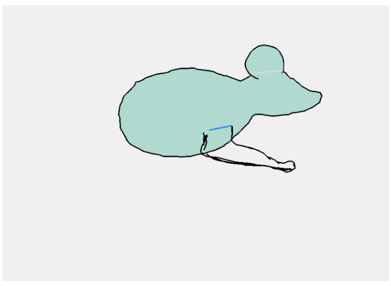
The second example involves the creation of a main object that contains a hole. Even though the image simplification process will always fill any holes within it, this can be circumvented by creating two separate objects and positioning them near each other. This test also evaluates how the inflation algorithm reacts to such an object. The aim is to create a torus. Initially, one half-circle is created (see Figure 6.28a), followed by the creation of the other half (see Figure 6.28b). They are then positioned near each other using the deformation tool and loaded into the original MM. The image demonstrates that while some artifacts appear in the stitching part, the objects hold together as one entity. The control points (red dots) in Figure 6.28d are stretching the circle in an attempt to separate it, but the object continues to behave like a single entity.

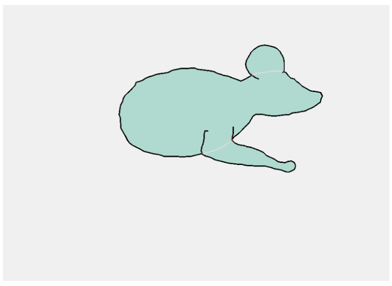

(a) : First layer before simplification.

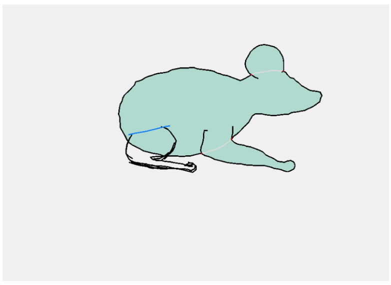

(b) : First layer after simplification.

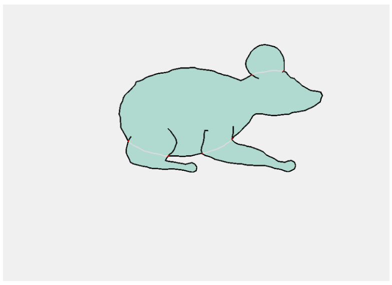

(c) : Second layer before simplification.

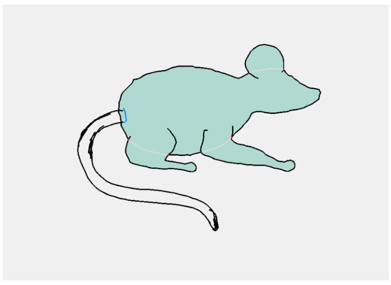

(d) : Second layer after simplification.


(e) : Third layer before simplification.


(f) : Third layer after simplification.


(g) : Fourth layer before simplification..


(h) : Fourth layer after simplification.


(i) : Fifth layer after simplification.

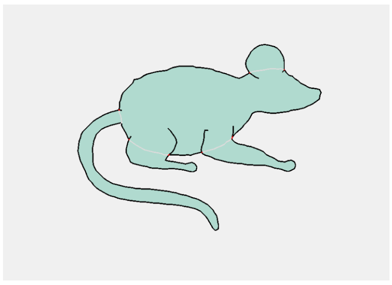
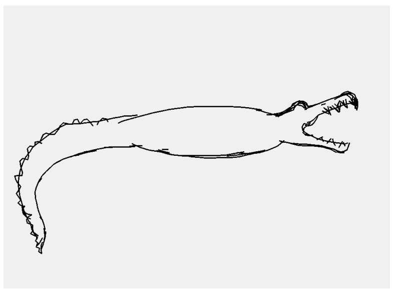
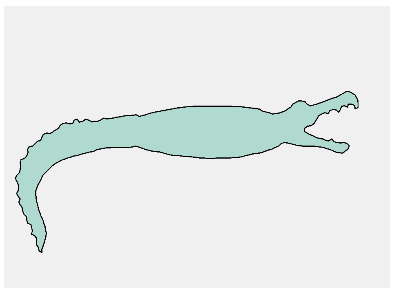
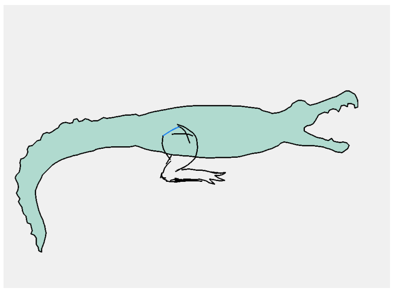
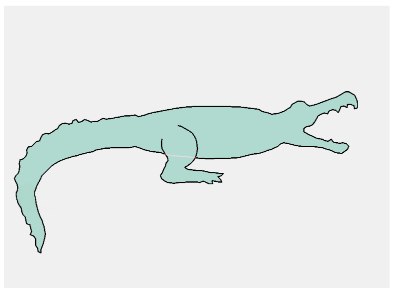

(j) : Complete sketch of the mouse.

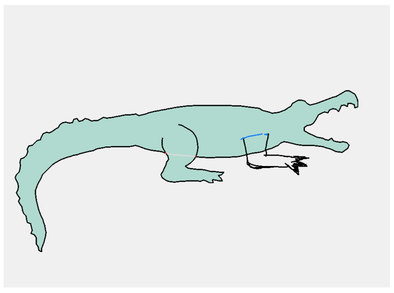
Figure 6.7: Process of creating a mouse sketch.


(a) : First layer before simplification.


(b) : First layer after simplification.


(c) : Second layer before simplification.


(d) : Second layer after simplification.


(e) : Third layer before simplification.

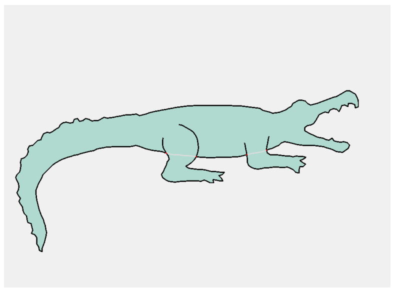
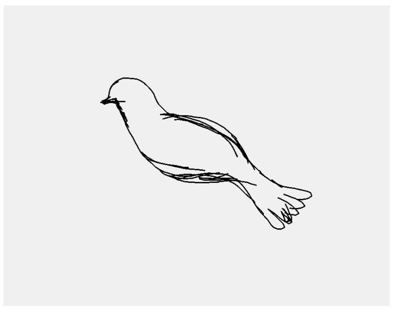
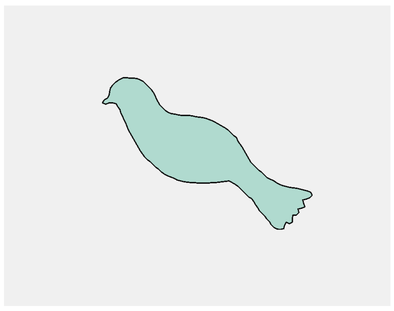
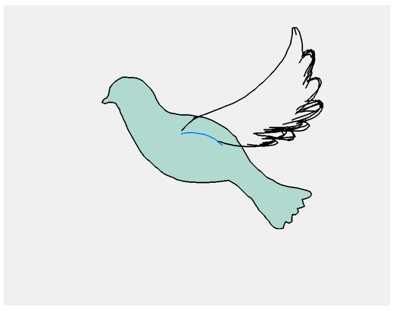

(f) : Complete sketch of the alligator.

Figure 6.8: Process of creating an alligator sketch.


(a) : First layer before simplification.


(b) : First layer after simplification.


(c) : Second layer before simplification.

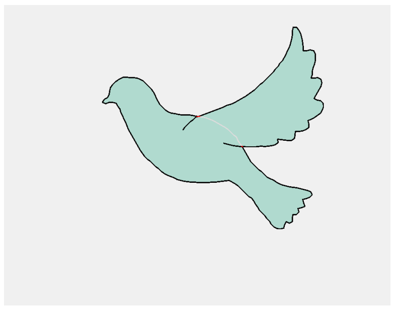
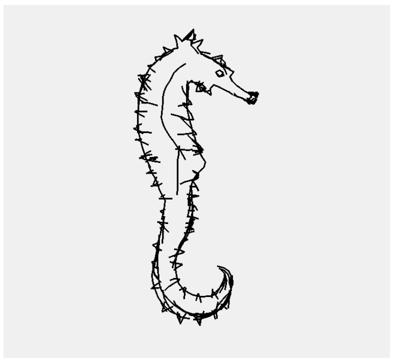

(d) : Complete sketch of the dove.

Figure 6.9: Process of creating a dove sketch.


(a) : First layer before simplification.

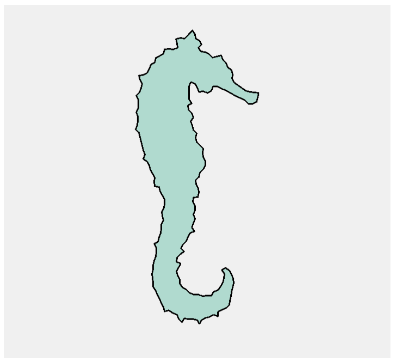
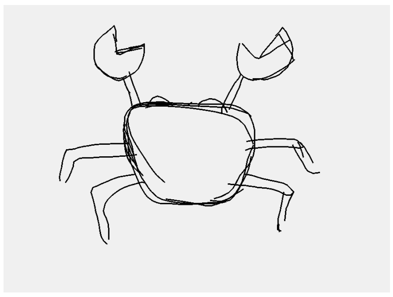

(b) : First layer after simplification.

Figure 6.10: Process of creating a seahorse sketch.


(a) : First layer before simplification.

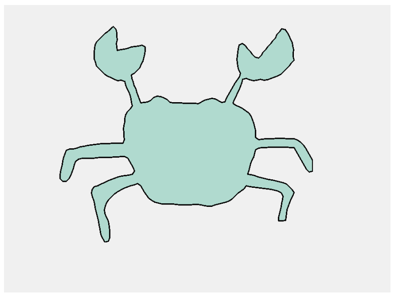

(b) : First layer after simplification.

Figure 6.11: Process of creating a crab sketch.

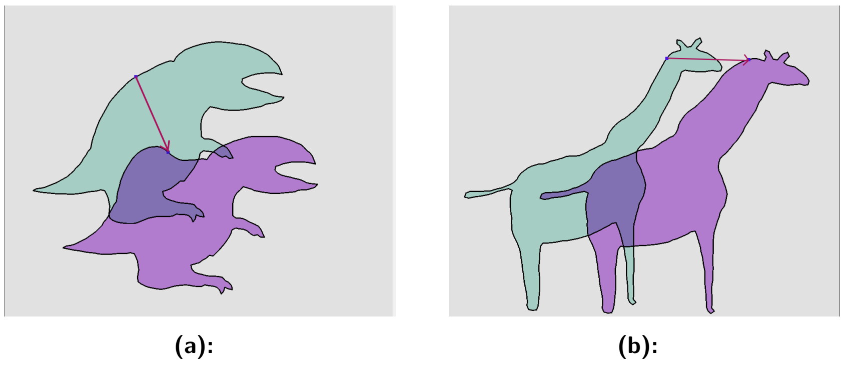


Figure 6.12: Results of deforming the images by one control point.

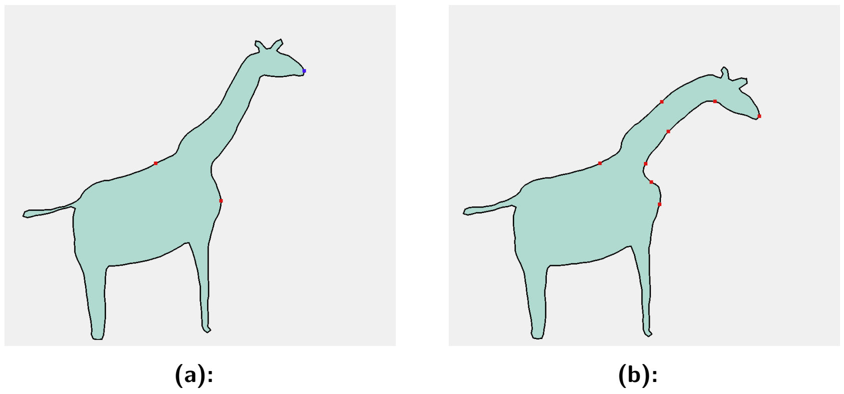


Figure 6.13: Results of significant deformation of the giraffe.

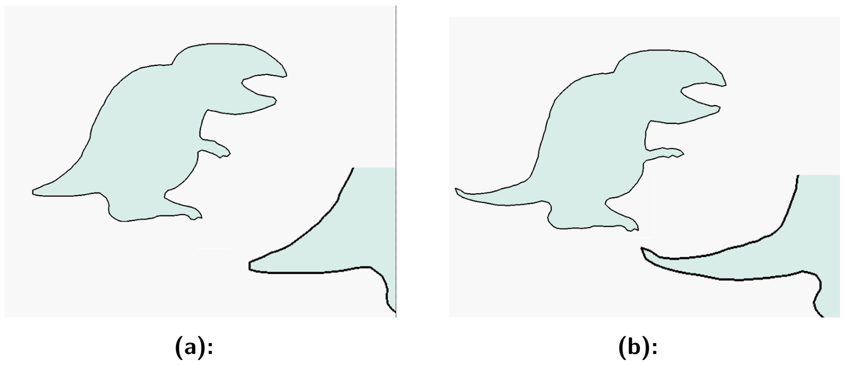
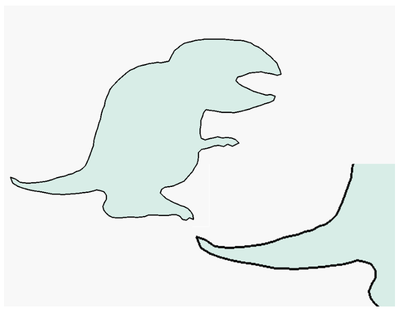


Figure 6.14: Results of minor deformation of the monstr.


(a):

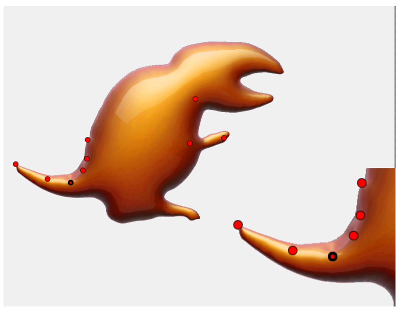
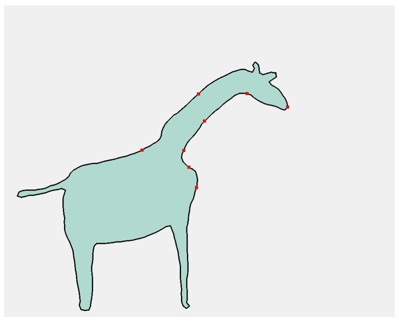

(b):

Figure 6.15: Comparison of 2D and 3D ARAP deformations of a monster.


(a):

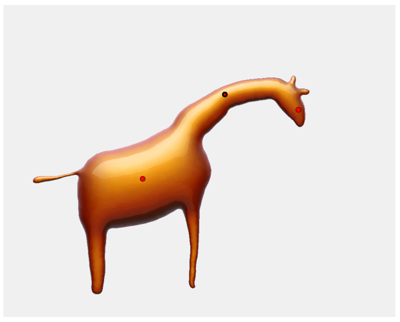

(b):

Figure 6.16: Comparison of 2D and 3D ARAP deformation of a giraffe.

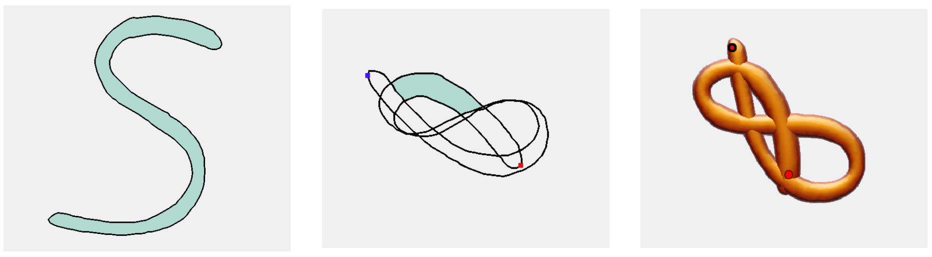


Figure 6.17: Comparison of 2D and 3D ARAP deformation of a symbol "S".

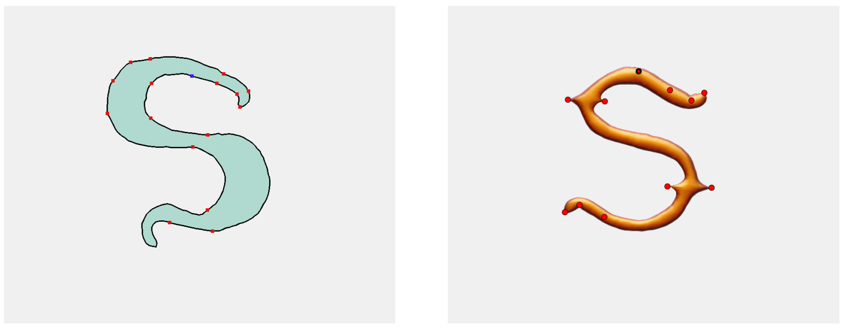
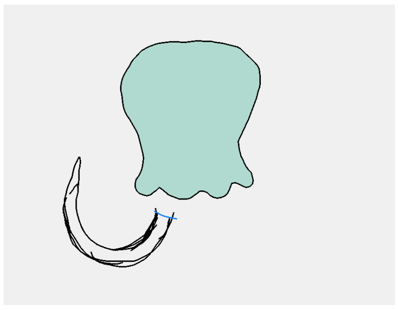
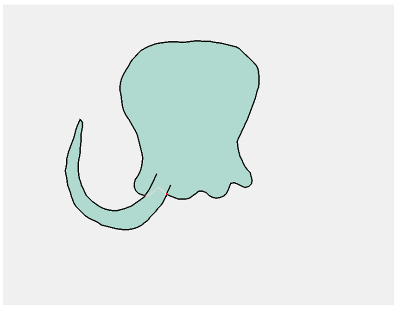
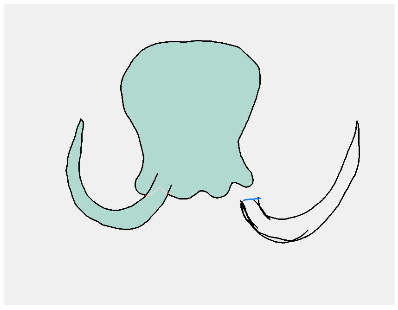
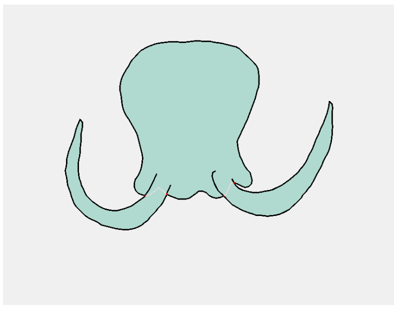


Figure 6.18: Second comparison of 2D and 3D ARAP deformation of a symbol "S".


(a) : Octopus with the main body and one unfinished tentacle.


(b) : Octopus with one simplified tentacle.


(c) : Addition of the second tentacle.


(d) : Simplification of the tentacle.

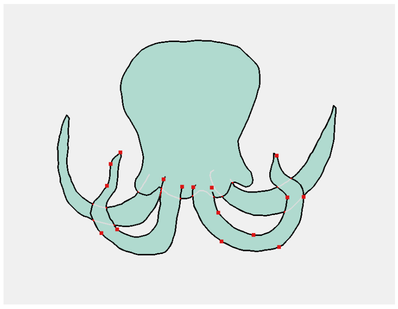
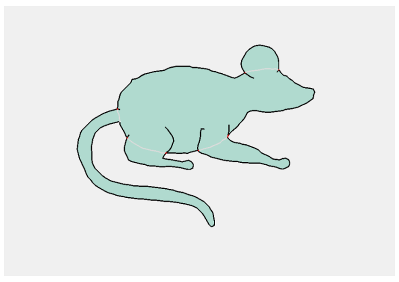
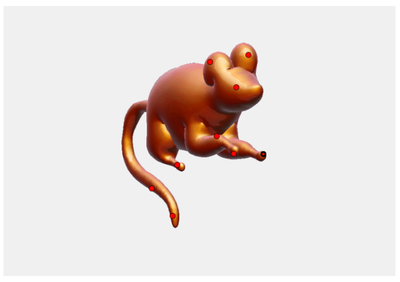
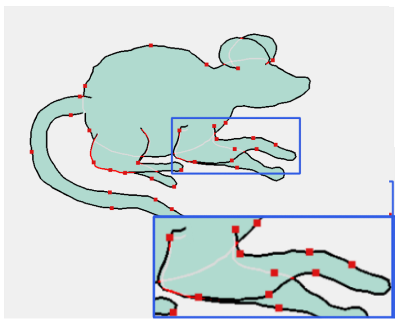

(e) : Result after copying and deforming 2 tentacles.

Figure 6.19: The result of the octopus created with the deformation tool.


(a) : Simplified mouse.


(b) : Inflated mouse.


(c) : Mouse deformed by 2D ARAP.

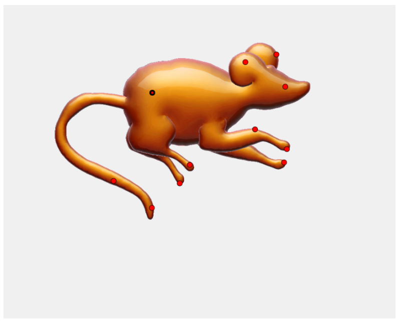

(d) : Mouse deformed by 3D ARAP.

Figure 6.20: The result of the mouse modified by the deformation tool.

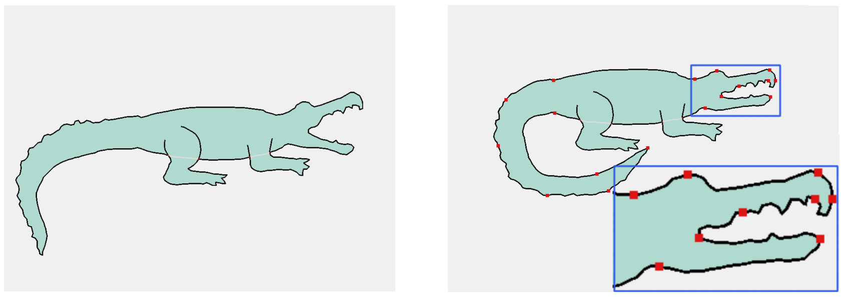


Figure 6.21: The result of the alligator modified by the deformation tool.

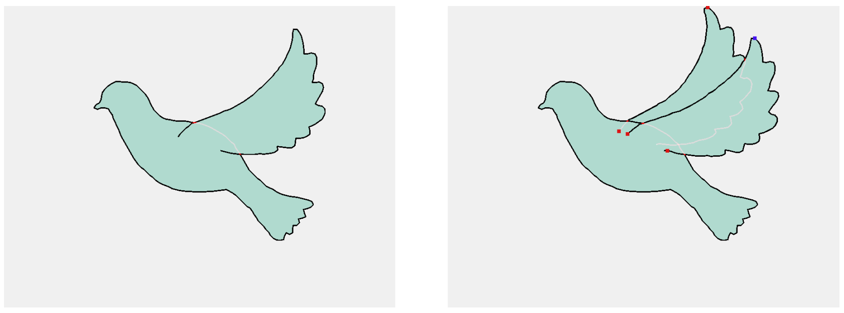


Figure 6.22: The result of the dove's wings modified by the deformation tool.

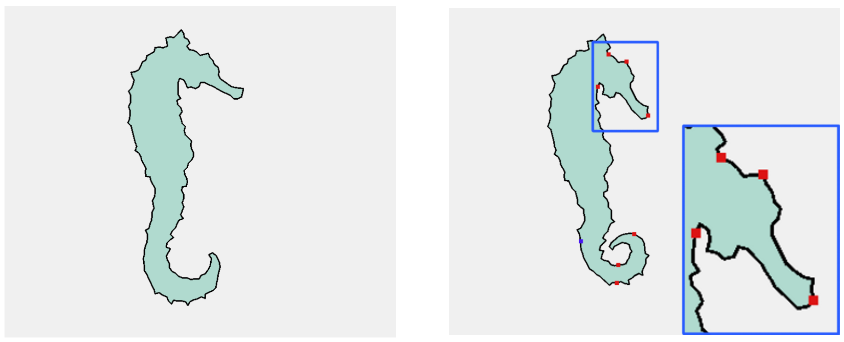


Figure 6.23: The result of the seahorse modified by the deformation tool.

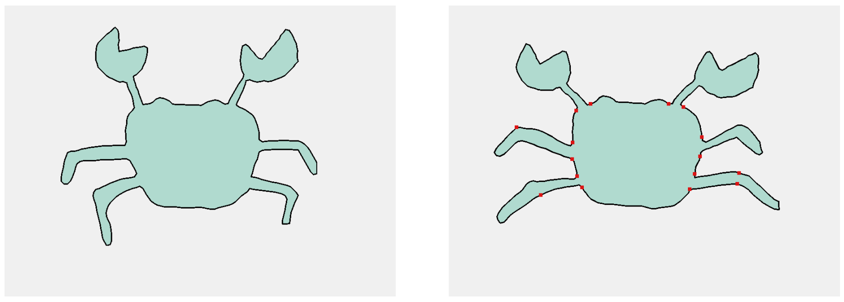
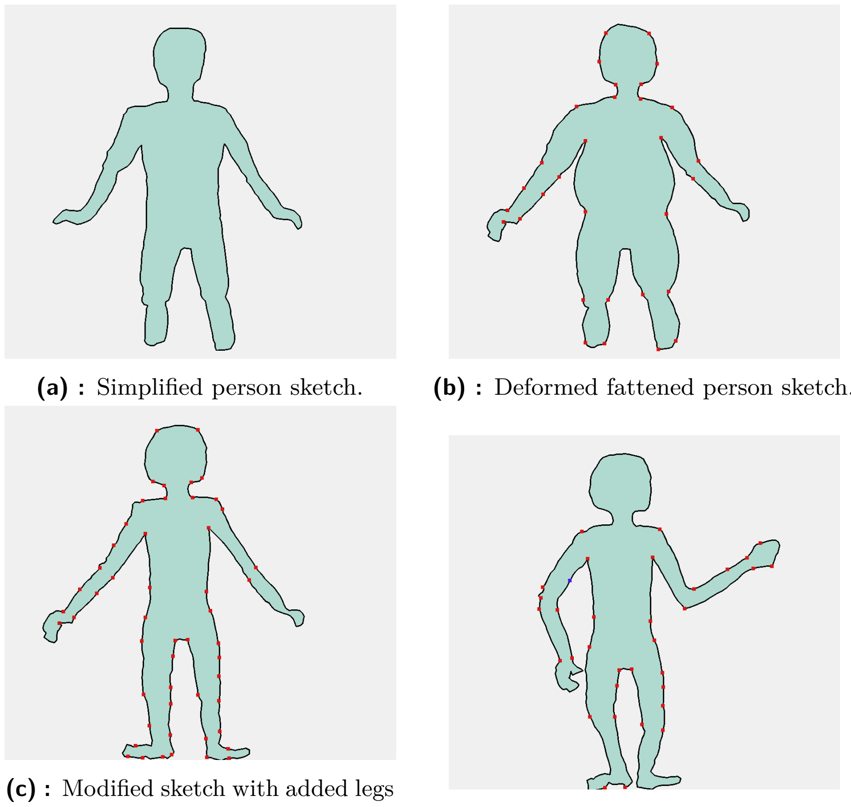


Figure 6.24: The result of the crab modified by the deformation tool.



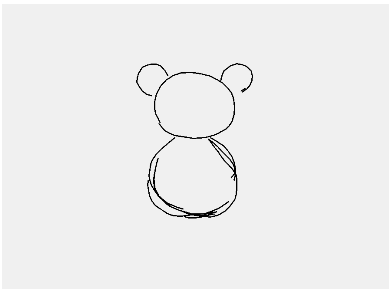
(a) : Simplified person sketch.

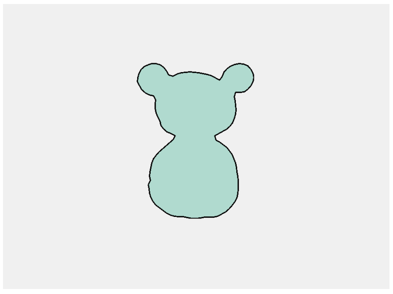
(b) : Deformed fattened person sketch.

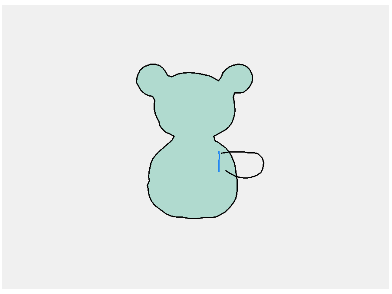
(c) : Modified sketch with added legs using deformation tool.

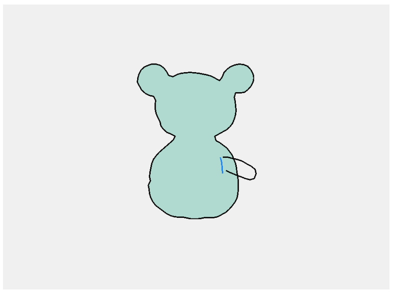
(d) : Modified limbs in person sketch

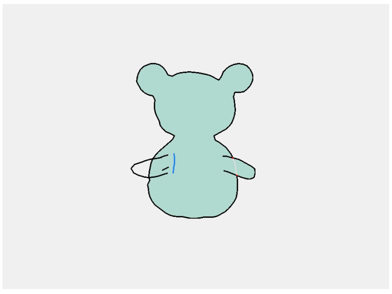
Figure 6.25: The results of the person sketch, modified by the deformation tool.

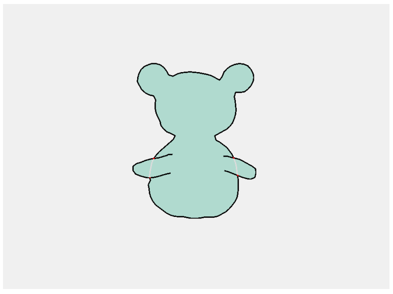

(a) : First layer before simplification.

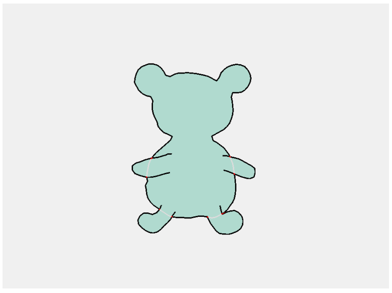

(b) : First layer after simplification.

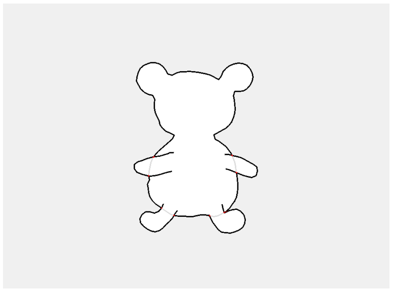

(c) : Second layer before simplification.

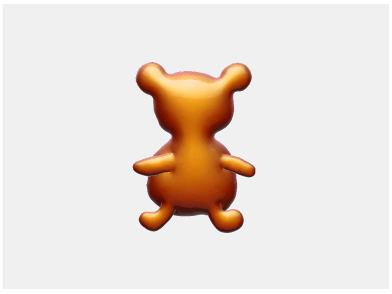

(d) : Erased and redrawn second layer.


(e) : Third layer before simplification.


(f) : Third layer after simplification.


(g) : Thr sketch with both legs added.


(h) : Loaded project in the original MM.


(i) : Inflated bear.

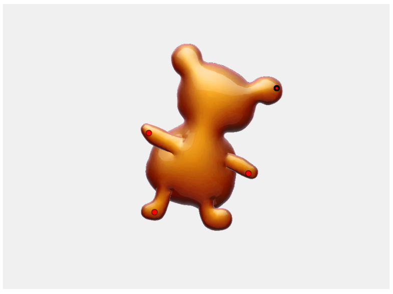

(j) : Deformed bear in the animation mode of MM.

Figure 6.26: Process of creating a bear sketch and loading it into MM.

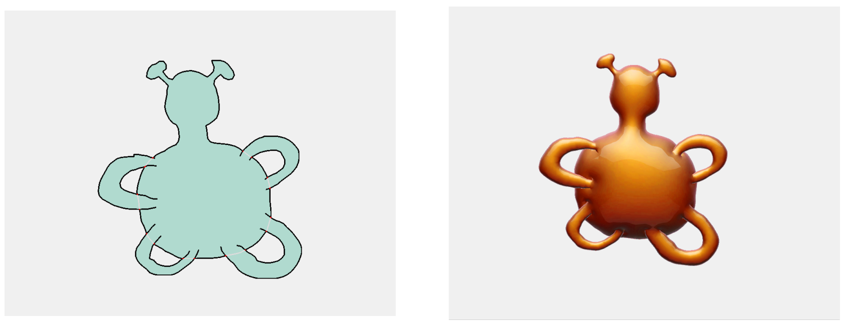
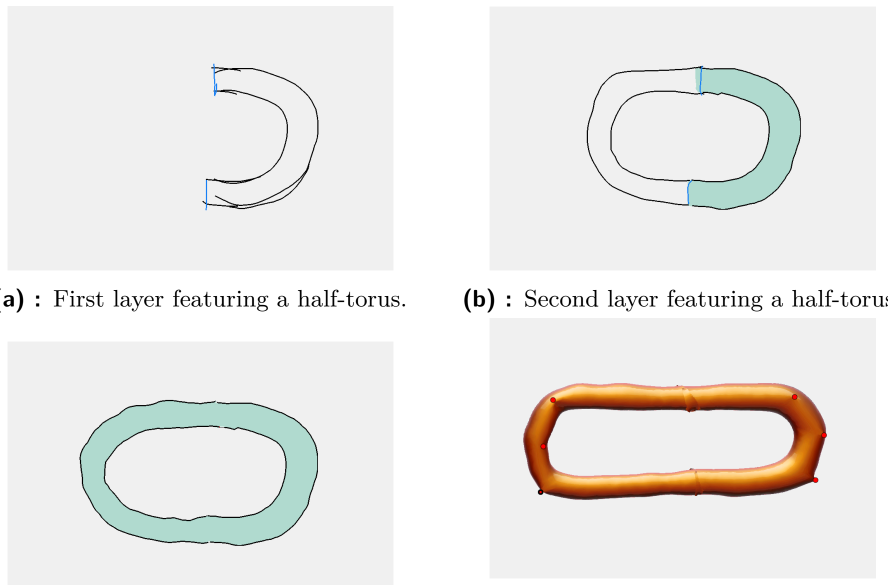


Figure 6.27: Result of segmentation featuring layers with holes.


(a) : First layer featuring a half-torus.

(b) : Second layer featuring a half-torus.

(c) : Result of positioning layers.

(d) : 3D model of torus imported to original MM.

Figure 6.28: Example of creating torus in MM.

Chapter 7

Conclusion

This research aimed to enhance the Monster Mash drawing tool, driven by a passion for art and a desire to augment digital drawing tools, thereby contributing to the advancement of the digital art industry. The focus was on incorporating sketch simplification and the As-Rigid-As-Possible deformation algorithms into the tool. This work has resulted in significant improvements, offering users a more enjoyable and efficient sketching experience.

The implementation of a sketch simplification algorithm has simplified the drawing process, making possible freer sketching and fast simplification. This feature offers potential benefits for both professional artists and hobbyists, making the drawing process less stressful and prone to errors.

The ARAP deformation algorithm provided a robust tool for altering sketches without requiring extensive redrawing. This function allows the creative process, permitting users to focus more on the conceptual aspects of their work. It allows for modifications in the sketch that would otherwise be more challenging to accomplish via redrawing.

A series of rigorous experiments were conducted to assess these newly integrated features. Their effectiveness within the interactive sketching process and their compatibility with the existing animation and inflation functions of Monster Mash were tested. The results offer a valuable evaluation of the tool's enhancements, highlighting their advantages and areas for further refinement.

7.0.1 Pros and Cons

In the experiments, the previous version of the tool was compared to the new implementation. It was found that the drawing process has become more straightforward and less stressful due to the tool's new functionalities. The drawing experience has been significantly enhanced, primarily due to the simplification tool's ability to fill a substantial amount of holes in sketches, despite occasionally overfilling certain parts.

The deformation tool addition facilitated the strokes edition. It proved most effective when correcting minor mistakes and changing positions of the protrusions of the object, while controlling larger changes proved more challenging. During testing, it was observed that there is a need for an undo button for the deformation tool, allowing users to reverse unwanted deformations, especially when users mistakenly deform the sketch.

From a user interface perspective, the mode buttons were easily understandable and user-friendly. With the current minimalistic button design, users could quickly understand each button's function and experiment freely. Shortcuts for the new buttons were also quite useful, enhancing the efficiency of mode switching.

7.0.2 Possible Improvements

In terms of sketch simplification, conducting more in-depth user testing could potentially be beneficial to estimate user preferences between a tool capable of filling larger holes or one that can close smaller holes but overfill less.

It became evident during the application testing that an undo function for the last drawn strokes or deformation would be desirable. Therefore, expanding the undo button's functionality could enhance the user experience, allowing it to cover a broader range of actions, similar to conventional drawing software.

Considering Monster Mash uses its own ARAP implementation, incorporating this implementation into the drawing part instead of the 2D ARAP deformation could potentially yield more stable results. It would furthermore be an efficient reuse of existing code.

In conclusion, it is relevant to mention that the data structures currently employed for the storage of images could be enhanced for efficiency as the current implementation appears to display suboptimal performance speed.

While there are areas for future development, this thesis has successfully enhanced an existing digital drawing tool. The improvements in sketch simplification and deformation features have certainly enhanced the user experience of the Monster Mash drawing tool.

Appendix A

List of Attachments

`MastersThesis.pdf`	The Master's Thesis text in PDF
`MastersThesis.zip`	The Master's Thesis latex source code
`MonsterMash.zip`	Monster Mash application
|- `src/`	Source code
|- `Build/`	Folder with executables
|- |- `Desktop/monstermash`	Desktop executable
|- |- `wasm/index.html`	Web executable
`Segmentation.zip`	Trial segmentation application
|- `src/segmentation.sln`	Project for VS
`StripARAP.zip`	Trial deformation aplication
|- `StripARAP/example`	Executable

Note: The web version of MM needs to run a server to work. This can be done by executing the command `python -m http.server` in the location `Build/Desktop/wasm/`.

Appendix B

List of Shortcuts

Shortcut	Meaning
1D	One-dimensional space
2D	Two-dimensional space
3D	Three-dimensional space
N-D	N-dimensional
NP	nondeterministic polynomial time
SVD	Singular value decomposition
STA	Square Tracing Algorithm
WTS	Within The Strip
ARAP	As-Rigid-As-Possible
LLVM	Low-Level Virtual Machine
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
JS	JavaScript
MM	Monster Mash
UI	User Interface
MLS	Moving Least Squares method
wasm	WebAssembly

Appendix C

Bibliography

[1] Marek Dvorožňák et al. “Monster mash: a single-view approach to casual 3D modeling and animation”. In: *ACM Transactions on Graphics* 39 (Nov. 2020), pp. 1–12. DOI: 10.1145/3414685.3417805.

[2] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Monographs in Computer Science. Springer New York, 2012. ISBN: 9781461210986. URL: https://books.google.cz/books?id=%5C_p3eBwAAQBAJ.

[3] Xueting Liu, Tien-Tsin Wong, and Pheng-Ann Heng. “Closure-aware Sketch Simplification”. In: *ACM Transactions on Graphics* 34 (Oct. 2015), pp. 1–10. DOI: 10.1145/2816795.2818067.

[4] Chenxi Liu, Enrique Rosales, and Alla Sheffer. “StrokeAggregator: Consolidating raw sketches into artist-intended curve drawings”. In: *ACM Transactions on Graphics* 37 (July 2018), pp. 1–15. DOI: 10.1145/3197517.3201314.

[5] Dave Mossel et al. “StrokeStrip: Joint parameterization and fitting of stroke clusters”. In: *ACM Transactions on Graphics* 40 (Aug. 2021), pp. 1–18. DOI: 10.1145/3450626.3459777.

[6] Edgar Simo-Serra et al. “Learning to simplify: fully convolutional networks for rough sketch cleanup”. In: *ACM Transactions on Graphics* 35 (July 2016), pp. 1–11. DOI: 10.1145/2897824.2925972.

[7] Daniel Sýkora, John Dingliana, and Steven Collins. “LazyBrush: Flexible Painting Tool for Hand-drawn Cartoons”. In: *Comput. Graph. Forum* 28 (Apr. 2009), pp. 599–608. DOI: 10.1111/j.1467-8659.2009.01400.x.

[8] G. Noris et al. “Smart Scribbles for Sketch Segmentation”. In: *Computer Graphics Forum* 31.8 (2012), pp. 2516–2527. DOI: 10.1111/j.1467-8659.2012.03224.x.

[10] Lester Randolph Ford and Delbert Ray Fulkerson. “Maximal Flow Through a Network”. In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404. DOI: doi:10.4153/CJM-1956-045-5.

[11] Uthpala Ekanayake, Wasantha Daundasekera, and S. Perera. “New Approach to Obtain the Maximum Flow in a Network and Optimal Solution for the Transportation Problems”. In: *Modern Applied Science* 16 (Jan. 2022), p. 30. DOI: 10.5539/mas.v16n1p30.

[12] Yefim Dinitz. “Dinitz’ Algorithm: The Original Version and Even’s Version”. In: Jan. 2006, pp. 218–240. ISBN: 978-3-540-32880-3. DOI: 10.1007/11685654_10.

[13] Y. Boykov and V. Kolmogorov. “An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.9 (2004), pp. 1124–1137. DOI: 10.1109/TPAMI.2004.60.

[14] T. Pavlidis. *Algorithms for Graphics and Image Processing*. Digital system design series. Computer Science Press, 1982. ISBN: 9780914894650. URL: https://books.google.cz/books?id=86VRAAAAMAAJ.

[15] Richard Szeliski. *Computer Vision: Algorithms and Applications*. 1st. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN: 1848829345.

[16] Michael Unser. “Splines: A perfect fit for signal and image processing”. In: *IEEE Signal processing magazine* 16.6 (1999), pp. 22–38.

[18] Olga Sorkine and Marc Alexa. “As-Rigid-As-Possible Surface Modeling”. In: Jan. 2007, pp. 109–116. DOI: 10.1145/1281991.1282006.

[19] Takeo Igarashi, Tomer Moscovich, and John Hughes. “As-Rigid-As-Possible shape manipulation”. In: *ACM Trans. Graph.* 24 (July 2005), pp. 1134–1141. DOI: 10.1145/1186822.1073323.

[20] Scott Schaefer, Travis McPhail, and Joe Warren. “Image deformation using Moving Least Squares”. In: *ACM Trans. Graph.* 25 (July 2006), pp. 533–540. DOI: 10.1145/1179352.1141920.

[21] Ho Law et al. “Quasiconformal model with CNN features for large deformation image registration”. In: *Inverse Problems and Imaging* 16.4 (2022), p. 1019. DOI: 10.3934/ipi.2022010.

[22] Guha Balakrishnan et al. “VoxelMorph: A Learning Framework for Deformable Medical Image Registration”. In: *IEEE Transactions on Medical Imaging* PP (Feb. 2019), pp. 1–1. DOI: 10.1109/TMI.2019.2897538.

[23] Bryan D. Agkland and Neil H. Weste. “The edge flag algorithm — A fill method for raster scan displays”. In: *IEEE Transactions on Computers* C-30.1 (1981), pp. 41–48. DOI: 10.1109/TC.1981.6312155.

[24] J. E. Bresenham. “Algorithm for computer control of a digital plotter”. In: *IBM Systems Journal* 4.1 (1965), pp. 25–30. DOI: 10.1147/sj.41.0025.

[25] Shivam Pradhan. *Bresenham’s Line Generation Algorithm*. 2023. URL: https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/.

Appendix D

Sources - Images

[3] Xueting Liu, Tien-Tsin Wong, and Pheng-Ann Heng. “Closure-aware Sketch Simplification”. In: *ACM Transactions on Graphics* 34 (Oct. 2015), pp. 1–10. DOI: 10.1145/2816795.2818067.

[4] Chenxi Liu, Enrique Rosales, and Alla Sheffer. “StrokeAggregator: Consolidating raw sketches into artist-intended curve drawings”. In: *ACM Transactions on Graphics* 37 (July 2018), pp. 1–15. DOI: 10.1145/3197517.3201314.

[5] Dave Mossel et al. “StrokeStrip: Joint parameterization and fitting of stroke clusters”. In: *ACM Transactions on Graphics* 40 (Aug. 2021), pp. 1–18. DOI: 10.1145/3450626.3459777.

[6] Edgar Simo-Serra et al. “Learning to simplify: fully convolutional networks for rough sketch cleanup”. In: *ACM Transactions on Graphics* 35 (July 2016), pp. 1–11. DOI: 10.1145/2897824.2925972.

[7] Daniel Sýkora, John Dingliana, and Steven Collins. “LazyBrush: Flexible Painting Tool for Hand-drawn Cartoons”. In: *Comput. Graph. Forum* 28 (Apr. 2009), pp. 599–608. DOI: 10.1111/j.1467-8659.2009.01400.x.

[8] G. Noris et al. “Smart Scribbles for Sketch Segmentation”. In: *Computer Graphics Forum* 31.8 (2012), pp. 2516–2527. DOI: 10.1111/j.1467-8659.2012.03224.x.

[9] Sagi Eppel. “Tracing liquid level and material boundaries in transparent vessels using the graph cut computer vision approach”. In: *CoRR* abs/1602.00177 (2016). arXiv: 1602.00177. URL: http://arxiv.org/abs/1602.00177.

[17] L Yin et al. “Complexity and accuracy of image registration methods in SPECT-guided radiation therapy”. In: *Physics in medicine and biology* 55 (Jan. 2010), pp. 237–46. DOI: 10.1088/0031-9155/55/1/014.

[18] Olga Sorkine and Marc Alexa. “As-Rigid-As-Possible Surface Modeling”. In: Jan. 2007, pp. 109–116. DOI: 10.1145/1281991.1282006.

[20] Scott Schaefer, Travis McPhail, and Joe Warren. “Image deformation using Moving Least Squares”. In: *ACM Trans. Graph.* 25 (July 2006), pp. 533–540. DOI: 10.1145/1179352.1141920.

[21] Ho Law et al. “Quasiconformal model with CNN features for large deformation image registration”. In: *Inverse Problems and Imaging* 16.4 (2022), p. 1019. DOI: 10.3934/ipi.2022010.

[26] Gordon Johnson. *Spiral Tubes Mesh 3D Abstrac.* [Online; accessed May 19, 2023]. July 11, 2022. URL: https://pixabay.com/vectors/spiral-tubes-mesh-3d-abstract-7313878/.