Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Advanced User Interface for 3D Modeling From Hand-Drawn Images

Tomáš Cicvárek

Supervisor: prof. Ing. Daniel Sýkora, Ph.D.
Field of study: Open Informatics
Subfield: Computer Graphics
January 2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Cicvárek** Jméno: **Tomáš** Osobní číslo: **466021**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Pokročilé uživatelské rozhraní pro 3D modelování z ručních kreseb

Název diplomové práce anglicky:

Advanced User Interface for 3D Modeling From Hand-Drawn Images

Pokyny pro vypracování:

Seznamte se technikami pro tvorbu 3D modelů z ručních kreseb [1-5]. Zaměřte se na metodu Monster Mash [3] a rozšiřte její existující uživatelské rozhraní tak, aby umožnilo aplikovat pracovní postup použitý v metodě Ink-and-Ray [1], kde uživatel specifikuje jednotlivé dominantní regiony pomocí hrubých tahů štětcem. Ty slouží jako vstup do algoritmu LazyBrush [5], jenž generuje výslednou podobu segmentace. Pro určení relativní hloubky jednotlivých regionů implementujte algoritmus topologického uspořádání [6], který s pomocí několika uživatelem specifikovaných nerovností stanoví finální pořadí v hloubce. Pro rekonstrukci zakrytých částí segmentů použijte metodu založenou na řešení Laplaceovy rovnice popsanou v článku [1]. Z výsledné množiny 2D regionů seřazených v hloubce vygenerujte 3D model pomocí existující implementace metody Monster Mash [3]. Praktickou použitelnost výsledného uživatelského rozhraní ověřte na sadě ručních kreseb, které dodá vedoucí diplomové práce. Výsledky porovnejte s výstupy konkurenčních metod [1, 4].

Seznam doporučené literatury:

[1] Sýkora et al.: Ink-and-Ray: Bas-Relief Meshes for Adding Global Illumination Effects to Hand-Drawn Characters, ACM Transactions on Graphics 33(2):16, 2014.
[2] Bessmeltsev et al.: Modeling Character Canvases from Cartoon Drawings, ACM Transactions on Graphics 34(5):162, 2015.
[3] Dvorožňák et al.: Monster Mash: A Single-View Approach to Casual 3D Modeling and Animation, ACM Transactions on Graphics 39(6):214, 2020.
[4] Zhang et al.: CreatureShop: Interactive 3D Character Modeling and Texturing from a Single Color Drawing, IEEE Transactions on Visualization and Computer Graphics, 2022.
[5] Sýkora et al.: LazyBrush: Flexible Painting Tool for Hand-drawn Cartoons, Computer Graphics Forum 28(2):599–608, 2009.
[6] Sýkora et al.: Adding Depth to Cartoons Using Sparse Depth (In)equalities, Computer Graphics Forum 29(2):615–623, 2010.

Jméno a pracoviště vedoucí(ho) diplomové práce:

prof. Ing. Daniel Sýkora, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **12.09.2023** Termín odevzdání diplomové práce: **09.01.2024**

Platnost zadání diplomové práce: **16.02.2025**

prof. Ing. Daniel Sýkora, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgements

I would like to give my thanks to my supervisor, prof. Ing. Daniel Sýkora, Ph.D., for his supervision over my thesis and for all information and consultations provided by him. I would like to thank my family as well for their psychical support.

Declaration

I hereby declare that the present master's thesis was composed by myself only and that I specified all used resources in accordance with the Methodical guideline for adhering to ethical principles when I was working on the academic final thesis.
Prague, 9. January 2024

Abstract

There are many tools for creating 3D models. However, for a large number of them, users may be concerned that their interface and controls are too complex. For instance, in situations where they want to create a simple prototype or just study the movement of a 3D model. Monster Mash solved this problem by having the users draw a simple outline of the characters from a profile. The program then creates the resulting model itself. However, if one wants to reconstruct a 3D model from a given drawing in this tool, the drawing must be traced accurately. In addition, the individual parts must be entered in the correct depth order, which can make the process difficult and tedious. In this work, we propose a system that enables users to select the main character components in the figure and determine their relative ordering in a simple way. This information allows us to create their segments and assign them an absolute order in depth automatically. The resulting set of ordered segments can then be uploaded to the Monster Mash tool to create the final 3D model.

Keywords: Monster Mash, image segmentation, occluded shape reconstruction

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

Abstrakt

Existuje mnoho nástrojů pro tvorbu 3D modelů. U velké části z nich se však uživatelé můžou pozastavit nad tím, že jejich rozhraní a ovládání jsou příliš složité. Například v situacích, kdy chtějí vytvořit jednoduchý prototyp nebo pouze studovat pohyb 3D modelu. Monster Mash vyřešil tento problém tím, že uživatelé nakreslí pouhý obrys postavy z profilu. Program pak výsledný model vytvoří sám. Pokud však chceme v tomto nástroji rekonsturovat 3D model ze zadané kresby, je nutno kresbu přesně obkreslit. Jednotlivé části navíc musíme zadat ve správném pořadí v hloubce, což může tento proces činit náročným a zdlouhavým. V této práci navrhujeme systém, který umožňuje uživatelům vybrat hlavní celky postav v obrázku a jednoduše určit jejich relativní uspořádání. Tyto informace umožní automaticky vytvořit jejich segmenty a přiřadit jim absolutní uspořádání v hloubce. Výslednou sadu seřazených segmentů lze následně nahrát do nástroje Monster Mash, který vytvoří výsledný 3D model.

Klíčová slova: Monster Mash, segmentace obrazu, rekonstrukce zakrytých tvarů

Překlad názvu: Pokročilé uživatelské rozhraní pro 3D modelování z ručních kreseb

Contents

1 Introduction **1**
1.1 Segmentation 3
1.2 Depth order 4
1.3 Segment reconstruction 5

2 Related work **7**
2.1 Ink-and-Ray 7
2.2 Monster Mash 8
2.3 CreatureShop 9

3 Analysis **13**
3.1 Segmentation algorithm 14
3.2 Depth order 17
3.2.1 Kahn's algorithm 17
3.2.2 Simplified Kahn's algorithm . 18
3.3 Segment shape approximation . . 21
3.3.1 Salient and illusory surfaces . 21
3.3.2 Occluded shapes estimation . 22

4 Implementation **27**
4.1 User interface 27
4.2 Segmenting algorithm 30
4.3 Depth ordering 36
4.4 Shape approximation 39
4.4.1 The initial guess 41
4.4.2 Distance transform 42
4.4.3 The diffusion and the export 43

5 Results **51**
5.1 Comparison 54
5.2 Limitations 57

6 Conclusion **61**
6.1 Future work 61
6.2 Final words 62

A Bibliography **63**

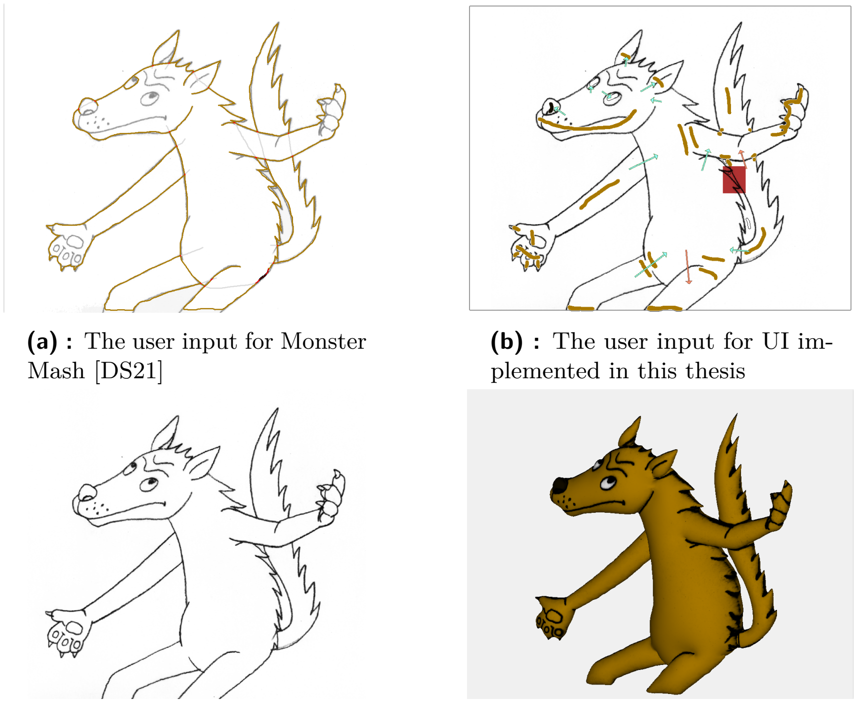
B Image credits **65**

Chapter 1
Introduction

There are currently a number of methods and tools available for 3D modeling, such as Blender [Com23], Autodesk Maya [Aut23b] or Autodesk 3ds Max [Aut23a]. Traditional methods allow users to create a 3D model by forming a 3D mesh made up of polygons. These methods require the manipulation and placement of new polygons, which can be a time-consuming and meticulous process. Thus, the polygons must be accurately positioned in 3D space. If we want to add detail to the model, we can either refine the 3D model even further, which requires even more careful placement of polygons, or apply textures to the model, which involves setting the coordinates for the textures to the 3D model. In order to animate a 3D character, we need to define a skeleton and set it up for skeletal animation. The whole process requires adjusting complex settings that users must learn. However, more intuitive tools and methods are available for newcomers to 3D modeling.

Sketch-based modeling methods [OSSJ09] allow users to draw 2D shapes that are automatically transformed into 3D models, which brings a major difference from traditional methods. While these methods do not allow the creation of complex models, since we have only have a single image, they are more than sufficient for casual modeling, prototyping or creating simpler models. During the whole process, the user does not leave the 2D domain and only modifies a few parameters in the application if necessary. By eliminating the need to inspect the model from all angles, the time required to create a 3D model is reduced. Textures are then applied based on the implementation of the tool used and the user does not have to manipulate complex settings to achieve satisfactory results. This makes this approach much more intuitive for people who are not familiar with 3D modeling. The only input these methods need is the user-made drawings in the tool interface and the uploaded image. We can get various results from a single image because the generated model depends on what tool was just used and how the image and user input is interpreted.

In this work, we propose to extend capabilities of one such tool, namely Monster Mash [DSC$^+$20]. We will do this extension either by improving its web version [DS21], or by creating a new tool. The extension would work



(a) : The user input for Monster Mash [DS21]

(b) : The user input for UI implemented in this thesis

(c) : The original drawing

(d) : The generated 3D model

Figure 1.1: Figure **(a)** demonstrates the amount of manual work that needs to be done in order to reconstruct a 3D model of wolf in Monster Mash **(d)** from an existing drawing **(c)**. Note, how the user needs to precisely trace the region boundaries in the input image including hidden contours and also plan the absolute depth order of individual regions in advance. Using the approach implemented in this thesis **(b)** the user needs to specify only a set of rough scribbles and a few arrows that define relative depth order of individual regions. This enables significant speed up over the original Monster Mash tool. (Wolf image source: ©Anifilm. All rights reserved.)

directly with the original image, which in the current implementation can only be outlined, as shown in Figure 1.1a. Our goal is to intuitively separate the main parts of an image and establish their depth order without having to explicitly outline them. This process should require minimal precision, which would accelerate and simplify the creation of models using Monster Mash for users, especially those unfamiliar with the software or whose equipment does not allow for precise strokes using a computer. The new tool is intended to reduce the need for manual tracing of individual body parts, as shown in Figure 1.1. The process begins with a grayscale image as an input. Users select its parts by providing strokes on the main parts and start the segmentation process. Users then connect the created segments with arrows to create a depth order between the segments. The arrows and strokes, along with an additional input, can be observed in Figures 1.1b. Finally, the occluded segments needed for the 3D model are estimated. At the end of these steps, a project file is generated that can be used to create a 3D model in Monster

Mash, an example is shown in Figure 1.1d.

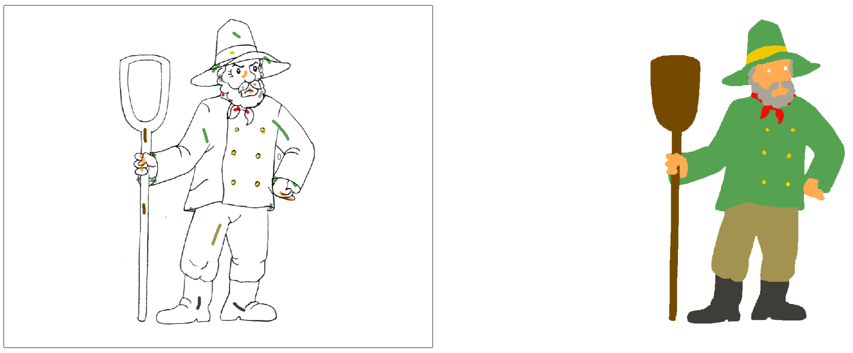
As mentioned, these activities do not usually require high precision. The segmentation algorithm needs only rough scribbles and the depth relations between the separated parts are determined by arrows that can start and end anywhere within the image segments. The overlapping segments, which are necessary in Monster Mash to connect the separated parts, are taken care of by the program. The project archive contains all the files needed to create a 3D model in Monster Mash, including the colored sketch generated during segmentation. In addition, the segmentation results can be improved by blurring or adding contrast to the image. All tools are described in detail in Section 4.1, where the user interface is explained. At saving the project, additional images are created to display the segmentation results. These visualizations are intended as a preview to improve the segmentation.

The procedure proposed in this thesis involves segmentation, depth order assignment, and segment overlap resolution, following the first three steps of the Ink-and-Ray pipeline [SKv$^+$14]: segmentation, completion, and layering.

1.1 Segmentation

In the segmentation step, the image is divided into several parts using the LazyBrush algorithm [SDC09], which is mainly used for grayscale image coloring. The user adds scribbles to the image and the algorithm assigns their color to the corresponding parts, as shown in Figure 1.2. In some cases, it is possible to brush over to other segments without creating artifacts in the resulting segmentation.

The LazyBrush algorithm enables the color to seep into the outlines and light edges in a drawing without bleeding into unwanted areas, even if there are holes in the outlines. In contrast, flood-fill algorithms cause color to leak into to other parts of the image if there is a gap in the outline. Furthermore, flood-fill algorithms are applied to only one color, which is the color of the starting pixel. However, the LazyBrush algorithm covers the entire contour regardless of the intensity in the pixel, resulting in the efficient creation of of single-color regions or segments. As seen in Figure 1.2b, this makes LazyBrush an ideal tool for the image segmentation process. However, these segments cannot be used directly for shape reconstruction, as the segments have user-defined regions that do not overlap. The depth and overlap information necessary to obtain a 3D model is generated in the following two steps.


(a) : User drawn scribbles **(b) :** Created segments

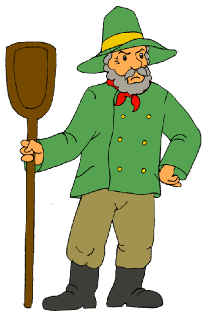
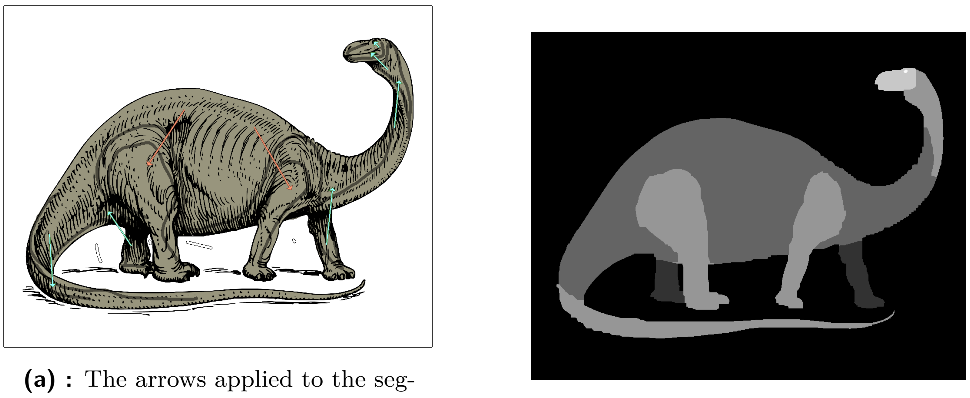

(c) : The segments applied to the original image

Figure 1.2: Figure **(a)** displays the user's input in the form of rough scribbles. In image **(b)** we can see the segmentation results of the original image. Figure **(c)** contains the colorized image by the segmentation result. (Source: ©Anifilm. All right reserved.)

1.2 Depth order

In this work, layering follows the segmentation process. In this step, we determine the relative depth ordering of the previously created segments and identify any overlaps. Algorithms, such as the one proposed in the Ink-and-Ray pipeline [SKv+14], can be used to estimate the depth order. However, instead of relying on approximation algorithms, we can utilize user input by allowing users to add arrows to the image. Adding arrows is an intuitive task that eliminates the need for absolute depth values. Figure 1.3a displays an example of arrows applied to existing segments. To process and validate the depth information, we use the algorithm proposed by Sýkora et al. [SSJ+10], which is further explained in the paper by Kahn [Kah62]. The topological ordering results in an absolute depth, as shown in Figure 1.3b. The white colored segment is the closest to the user and the black color represents the background. The data obtained here will be used in the final step to estimate occluded segments.



(a) : The arrows applied to the segments

(b) : The depth levels

Figure 1.3: The arrows applied by the user are shown in image **(a)**. The depth levels of all segments are shown in the in image **(b)**, with brighter color indicating closer proximity to the user.

1.3 Segment reconstruction

The final step of this project is the reconstruction of the occluded segments and their boundaries. To ensure a smooth continuity of the results, gaps in the contours of the upper segments will mean merging with overlaid segments in the final 3D model. Similarly, some of the occluded segments will be merged with closer segments. Depending on the input images and the user's requirements for the resulting 3D model, manual intervention might be required to set proper merging boundaries. This is necessary in situations where shading, other details, or noise in the image may cause dark pixels to be misinterpreted as contour lines. Such areas can be resolved within the application, as shown in Figure 1.1b, or by adjusting the border lines in a 2D graphics tool after exporting the segments. It is important to note that this problem is not limited to the upper segments. The tool can also modify the merging behavior imposed by the arrows, which is applied globally to the entire segment. However, in some local cases, users may require a different type of behavior to separate or merge two segments.

The segment reconstruction step is equivalent to the completion step in the Ink-and-Ray pipeline [SKv$^+$14]. We use the segments boundaries gained in the segmentation step and the depth levels to determine which segments require estimation of their shape. The algorithm proposed by Jeschke et al. [JCW09] for the approximation quickly provides the desired solution. All shapes, their borders, colored template image, and configuration files are can be then uploaded to Monster Mash to create the final 3D model.

Chapter 2

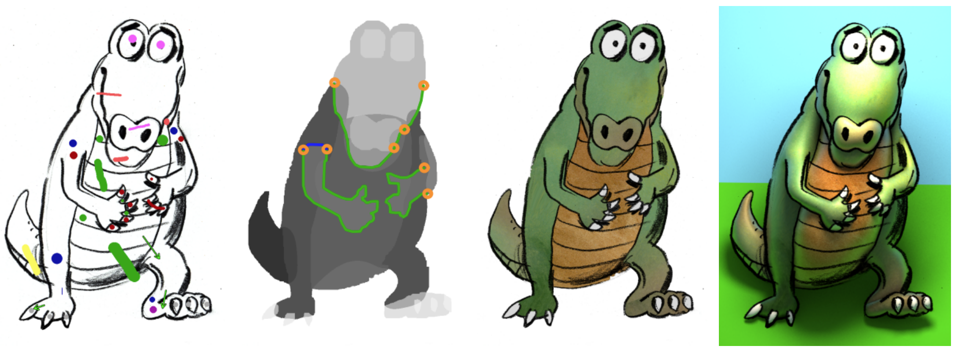
Related work

Creating a 3D model from a hand-drawn sketch has been a long-standing topic. There are many tools and methods to handle the creation of a 3D model from a single image [OSSJ09]. In this chapter, we present a selection of sketch-based modeling methods that are particularly relevant to our application.

2.1 Ink-and-Ray

The first approach is the Ink-and-Ray pipeline [SKv$^+$14] and its user interface is very similar to the one proposed in this work. This approach generates 2.5D objects and creates the illusion that images have a physical form. The pipeline is inspired by bas-relief sculptures, which add depth to the portrayed view. The final results are classified as 2.5D instead of 3D because the models are only visually appealing from a frontal view. The pipeline was designed to add depth only to the images as they are viewed front. From Figure 2.1, it can be seen that strokes must be entered before the segmentation process can begin. The depth order is usually handled automatically here, but inconsistencies can be corrected by the user using an arrow connecting two segments. Boundary conditions can be adjusted to straighten them out in the inflated regions in the 3D model because the default conditions round the segments at the edges. This way, users can suppress rounding in certain boundary areas to maintain a straight and smooth transition at the ends of the segments. To achieve this, the user must click on two locations near the segment boundary in the user interface. The tool then identifies the nearest boundary and determines the path that spans the boundary between the two points.

As can be seen, this work is heavily influenced by the Ink-and-Ray pipeline [SKv$^+$14]. The segmentation algorithm remains the same, with only minor differences in parameterization and options regarding the user interface. The depth relations in this work rely solely on arrows added by the user, as no automated method is planned to solve the segment depth order. The procedure for approximating hidden segments is inspired by the procedure presented in the Ink-and-Ray pipeline [SKv$^+$14] proposed by Geiger et al. [GPR98].



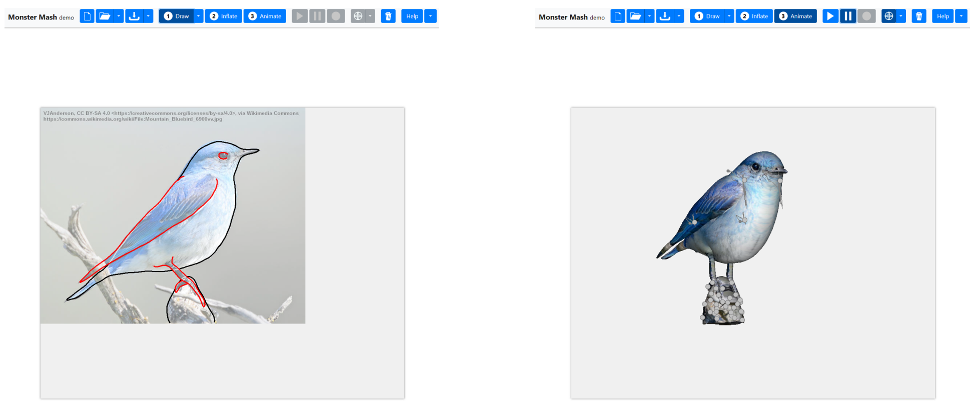
(a) : User input **(b) :** Refined stitches **(c) :** Texture **(d) :** Result

Figure 2.1: The images represent the Ink-and-Ray pipeline [SKv+14]. Figure **(a)** shows the user input. The input consists of the scribbles used in the segmentation process and the optional arrows that specify the depth relations for the program. Figure **(b)** shows us the optional specification of the stitch type along with the estimated overlaps. Figure **(c)** contains the applied texture. The last image **(d)** shows us the rendered result of the provided input. (Source of the images: Ink-and-Ray [SKv+14])

2.2 Monster Mash

Monster Mash is a sketch-based 3D modeling tool [DS21] designed by Dvorožňák et al. [DSC+20]. In this tool, the user remains in the 2D domain during the creation of the 3D model. The input data consists of a set of open and closed strokes. The closed strokes are utilized to create a coherent mesh. On the other hand, we have open strokes that are automatically closed by a line that defines either a free boundary or a merge boundary. These strokes are primarily intended to attach new parts to existing shapes. Thus, all strokes describe individual body part shapes and are stored in layers. These layers are arranged according to the order in which they were added and the shortcuts used. Typically,users draw over existing shapes when adding a shape. One of the shortcuts allows users to draw under other shapes. Double-clicking mirrors the secected shape to the other side of the target object, as shown in Figure 2.2. The tool also allows users to use the image as a template for a future 3D object. The content of the image is drawn and then orthogonally projected into the model as a texture.

The tool also allows users to quickly create looping animations by moving control points and recording their path. The length of each new animation is defined by the first animation created. This feature allows users to calibrate new animations with previously created movements. To ensure that the length of the new animations matches the set duration, the initial portion of the calibrated animation is discarded, allowing the 3D model to be motion tested. The model can also be exported to a more advanced 3D tool for further editing.



(a) : Monster Mash user input **(b) :** Inflated model with control points

Figure 2.2: Both images were captured in the Monster Mash demo tool [DS21]. Figure **(a)** shows us the user input for the Monster Mash in the form of the template image outlined by the user-drawn strokes. The wings, legs and eyes are connected to the main body in the area where their contours are open. The red outlines indicate that the segment will be mirrored to the other side of the segment it is connected to. Figure **(b)** contains the inflated image along with the control points and animation curves.

To obtain a more accurate model using the template in Monster Mash, the existing sketch must be outlined accurately. Additionally, the outlines must be in the correct depth order, as the resulting 3D model depends on them. If there are multiple failed attempts regarding the outlines or their order, the process can become tedious and time-consuming as it may be necessary to start from scratch. This is because the Monster Mash tool only addresses depth order by drawing new parts either above or below the others. The newest outline cannot be drawn between existing overlapping strokes, only over or under them. These limitations slow down the model creation process by forcing the user to either stop and plan the next step or repeat the drawing process.

2.3 CreatureShop

Zhang et al. [ZYC$^+$23] present us with a novel sketch-based modeling method, which they call CreatureShop. The method enables users to create highly attractive 3D models from a single image, and the character can be drawn in almost any viewpoint.

CreatureShop employs four tools to define the geometry of the model, as illustrated in Figure 2.3. The first tool in the user interface allows users to outline the segments of the creature to prepare the individual parts. Although the segmentation process is present, it is handled manually, not by tools such as the LazyBrush. The second tool allows selection of the orientation of the bilaterally symmetric planes. These planes will mirror the assigned segment halves on their other side, allowing the shape and texture of the segment to be mirrored later in the process. The third tool allows the user to find symmetrical landmarks, such as eyes, which can be used to add additional detail to the geometry. The last tool is used to define a plane in the middle of each segment. The middle plane is indicated by a drawn curve that intersects the segment plane. The lines of this tool can also be used to control the shapes of smaller details that cannot be defined using symmetric landmarks.

After the user provides the input, the program creates the individual body parts and connects them into a seamless, watertight mesh. In some cases, mirroring the original image with reflection planes may result in unwanted texture mirroring or textures that are not processed correctly by the framework, as shown in Figure 2.4. Texture inpainting method is used to solve this issue. The algorithm selects the area with the wrong texture and replaces it with a new texture patch sourced from the appropriate area.

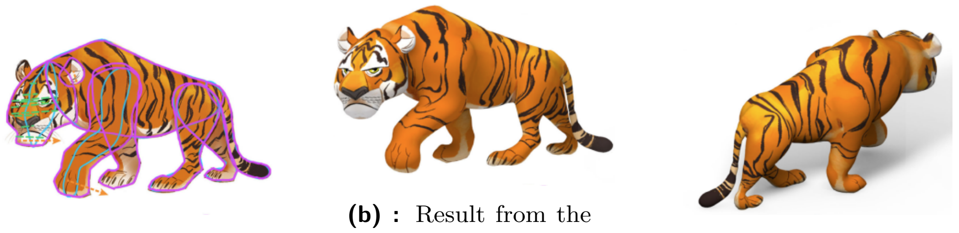
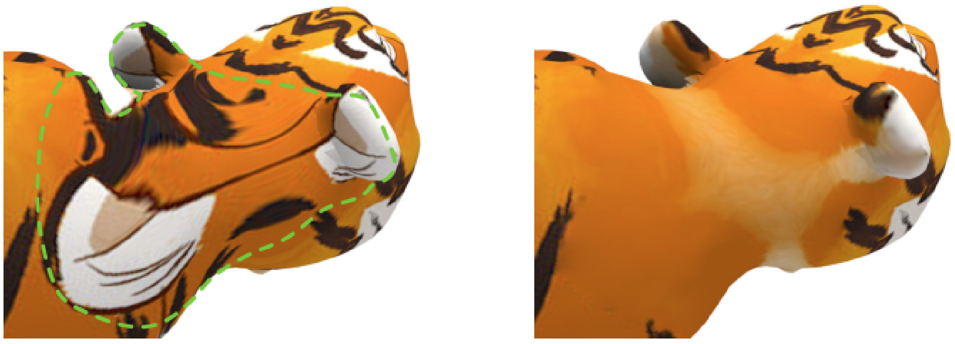

(a) : User input **(b)** : Result from the front **(c)** : Result from the back

Figure 2.3: Figure **(a)** presents the user input. The violet marks indicate the segments of the tiger in the original image. The blue marks represent the midlines that specify the symmetric plane. The orange arrows are used to select the orientation of the symmetric planes. The green dashed lines connect symmetric landmarks on the tiger. Figures **(b)** and **(c)** display the generated 3D model that was enhanced with the inpainting. The source of the images is the CreatureShop [ZYC$^+$23]

In contrast to this approach, the Monster Mash tool primarily processes characters from an orthogonal or frontal view. Arbitrary view can cause artifacts in the models generated by Monster Mash. Although Monster Mash can also generate 3D models from arbitrary view, they are not as accurate. The aim of this work is to simplify the segmentation process, which consists of tracing the segments in Monster Mash. The segmentation process will only require the user to draw scribbles in the segmented areas. The final segments will be determined automatically based on the depth order of the segments.

When comparing the way Monster Mash and CreatureShop work with textures, it is important to note that Monster Mash only applies textures by orthogonal mapping to the front and back of the model. This might result in artifacts such as blurred or elongated textures on the sides of the model,



(a) : Wrong patch **(b) :** Refined stitches

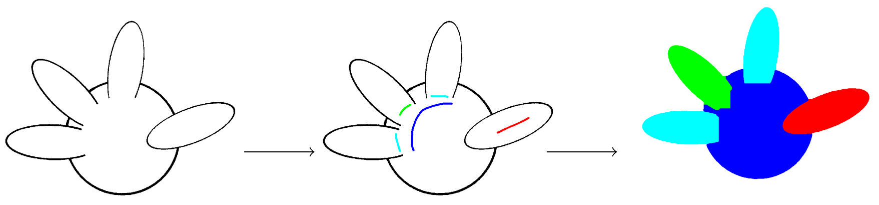
Figure 2.4: The images from CreatureShop [ZYC+23] that show us the application of image inpainting. In image **(a)** the 3D model contains an improperly applied texture. Figure **(b)** shows us the model after the inpainting process.

face or limb textures appearing on the other side of the model, and also copied textures on obscured parts of the model. These artifacts, that might appear in the generated model, cannot be handled by an external tool such as the one proposed in this work. In contrast, CreatureShop mirrors textures locally through the midplanes. In the case of artifacts, inpainting is utilized to provide a suitable texture.

Chapter 3

Analysis

The analysis chapter describes the steps and key algorithms used in this thesis. We focus on the first three steps of the Ink-and-Ray pipeline [SKv+14], which will provide the 2D shapes needed to create the desired 3D model.



(a) : The original image **(b) :** The user input **(c) :** The segmented image

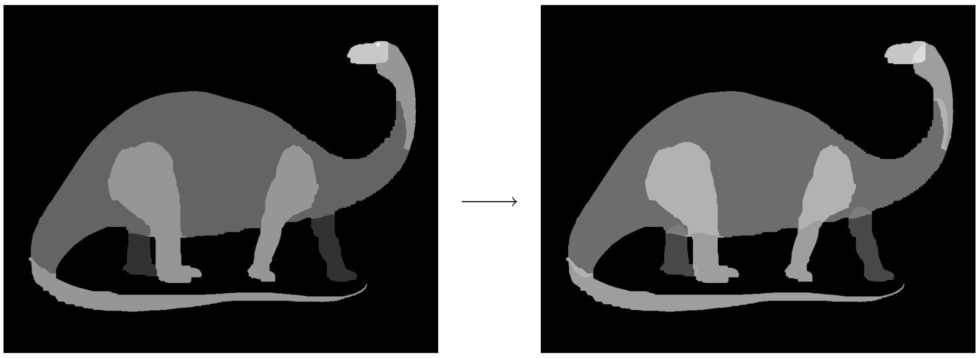
Figure 3.1: In the figure we can see the segmentation process. The user adds scribbles to the image, as can be seen in image **(b)**. The segmentation process then fills the contours with the color of the drawn scribbles.

Segmentation
In the segmentation step, the character or object in the image is divided into several parts (segments). The user creates segments recognizable to the algorithm by drawing strokes, which are then processed by the segmentation algorithm. The segmentation will provide us with a set of regions that will give us basic information about the shape of the character. It also provides us with a color texture for the resulting 3D model.

Layering
In this step, the user establishes the relative depth order between all segments by adding arrows, as shown in Image 1.3a. The layering step then processes the arrows to produce the absolute depth order used in the last step, as displayed in Image 1.3b. The depth order is used to constrain the estimation for visually occluded segments only and when creating the model in the Monster Mash.

Completion
The last step presented in this work is the approximation of the occluded

segment shapes extracted in the segmentation step. An example of the segments can be seen in Image 3.2a. The visible segments are used to complete the occluded regions along with their depth ordering. In the Ink-and-Ray pipeline [SKv$^+$14], the layering and completion steps are not in this order because the dependence of these steps is reversed in the article. Since we do not need to estimate the depth of the segments, the depth order in this work allows us to directly distinguish which shapes need to be processed and which we can omitted entirely. The result of this step is shown in Image 3.2b.



(a) : Segments

(b) : Approximated shapes

Figure 3.2: Image **(a)** displays the segments in their depth layers. The segments are then processed and the occluded ones have their shape approximated, as we can see in image **(b)**. Although some of the shapes may not look right, such as those around the neck, they provide the connection between the segments in the final model and are not visible. We can also see that the eye segment is missing. The eye segment is missing because it is inside the head segment and its boundary does not cross the boundary of any other segment.

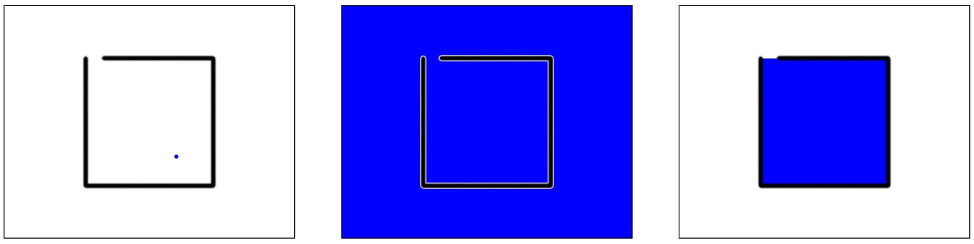
Each step of this work and its results depend on the results of the previous steps and on the user's input. In the following section, we define the problems we need to solve and explain the basic algorithms we need. The implementation of the algorithms is described in the following chapter.

3.1 Segmentation algorithm

The segmentation algorithm is the foundation of our pipeline because it creates the basic structure of the future 3D model. The image containing the character is divided into several parts (segments), which are later sorted and completed to meet the requirements of the Monster Mash tool.

There are several algorithms and methods that allow us to segment an image. In our work, we focus on the LazyBrush algorithm [SDC09] which was introduced as a tool for coloring hand-drawn sketches.

With properly defined parameters, this min-cut algorithm will achieve results that no flood-fill algorithm can achieve. There are certain conditions for flood-fill algorithms to work properly. They require homogeneous regions that must be bounded by a closed boundary. However, these conditions are not always satisfied in the case of sketches, because there may be open contours in the image. In addition, areas along the contours may be blurred, which increases the range of colors present in the segmented area. In the case of scanned images, we have to take into account the noise in the images. Unlike flood-fill algorithms, the LazyBrush algorithm can work with open contours, i.e. holes in the contours, and perserve the color inside them. Depending on the implementation, the LazyBrush algorithm can work well on images that do not have large homogeneous areas, such as the photos in grayscale. However, the configuration required for the sketches allows the color to seep into the outlines even when color gradients are present. A comparison of the results of the flood-fill and min-cut algorithms can be seen in Figure 3.3. The properties of the min-cut algorithms allow us to utilize colored regons as segments that we will need in the next steps.



(a) : An example image with open outline and a user drawn label

(b) : The flood fill algorithm result

(c) : The min-cut algorithm result

Figure 3.3: Image **(a)** shows us an image with an open outline. The scribble is at the location used by both the flood-fill and min-cut algorithms. The background label, used by the LazyBrush algorithm, is located at the corners of the image. The flood-fill algorithm then avoids non-white pixels and ignores the open contour, filling most of the image with blue color, as shown in image **(b)**. In contrast, the min-cut algorithm seeps in the blue color to the contour and stops at the hole in the outline, as shown in image **(c)**.

Due to the nature of min-cut algorithms, the LazyBrush algorithm itself allows us to solve a binary problem. The task we face is to create a labeling that assigns pixels to two scribbles drawn by the user. We are given a set of labels L and an image containing pixels P in a 4-connected neighborhood N. We want to find a labeling c that assigns label $c_p \in L$ to a pixel $p \in P$. The problem can be formulated as energy function:

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

where p, q are neighboring pixels, $V_{p,q}$ is the color discontinuity energy between them, and D_p represents the color energy assigned to pixel p by the scribble. We can express the term $V_{p,q}$ as the intensity difference between pixels p and q. The term D_p indicates whether pixel p has been covered by one of the scribbles or not. If p has been covered by a scribble, we set the term D_p for that pixel to a high number, otherwise we set this parameter to zero. The prepared data can then be solved with the min-cut algorithm. One of the available algorithms for cutting graphs is the push-relabel algorithm, proposed for example by Timo Stitch [Sti09]. This algorithm incrementally converts the current flow into the maximum flow, instead of sending the flow directly from the source to the sink along paths as it does Ford-Fulkerson algorithm and algorithms derived from it. In our work, we will use a method based on the Boykov-Kolmogorov algorithm [BK04] implemented in the GridCut library [SJ].

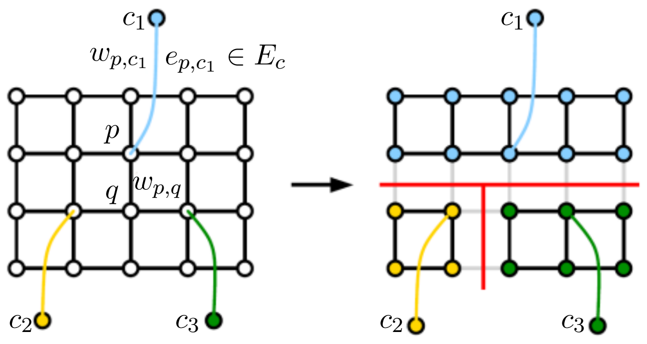


Figure 3.4: Multi-way cut problem example [SDC09]:
On the left we can see the white dots representing pixels from P. The three terminal nodes C are colored. The black lines represent the neighbourhood of every two pixels p and q with the weights between them $w_{p,q}$ which represent the term $V_{p,q}$. The colored edges from pixels $p \in P$ to terminals $c_i \in C$ represent the data term D_p.
In the right image there is a possible multiway cut solution and labeling for all pixels.

If we want to allow the user to add multiple scribbles, the algorithm must solve the problem for each label separately against the other scribbles. In doing so, the area and data are dynamically reduced to avoid unnecessary computation. Figure 3.4 displays a simple setup and the final labeling.

The created segments are then used in the next two steps. In the depth ordering step, they are topologically sorted and assigned to depth layers. In the last step, the missing parts of the segments are estimated to provide the basis for the future 3D model.

3.2 Depth order

To estimate missing contours, a distinction must be made between occluded and occluding segments. The arrows seen in Figure 1.1b are user input defining the relative depth order between all segments. This input indicates possible occlusion caused by closer segments and is crucial in approximating segment boundaries. In that step, only the shape of the occluded segments is reconstructed. These arrows form an oriented graph where the nodes are the segments and the arrows are the edges between them. In order to determine the correct depth order, the generated graph must not contain an oriented loop, since such a situation would lead to an inconsistent depth ordering. We cannot solve the loop problem only before approximating the occluded parts, as there may form several loops and deciding which arrows to delete can be a difficult task. The only solution is to delete the incorrect arrow at the moment it is added to the graph. This problem can be solved by topological sorting with loop detection. Since we test for the presence of a loop every time an arrow is added to the graph, there can exist only one loop, namely the one that was created by the most recent arrow.

Kahn's algorithm [Kah62] presented in the article by Sýkora et al. [SSJ$^+$10] allows us to detect cycles in an oriented graph. Although the original Kahn's algorithm [Kah62] is optimized to work on large-scale networks, the expected input in our case is limited and some possible scenarios in the article do not occur in this work. Here we can work with a simplified version of the algorithm to avoid the sorting steps mentioned in the original article.

3.2.1 Kahn's algorithm

The algorithm utilizes two structures: an event list and a list of activities. At the beginning of the algorithm, the list of activities is sorted by the starting nodes(events) of the edges(activities). In the rest of the explanation of the algorithm, we will refer to nodes as events and edges as activities to maintain consistency with the format of the original article.

Each activity in the list of activities contains a flag that indicates the last element of the group of activities with the same initial event (referred to as the predecessor flag) and the location of the successor activity (referred to as the successor). Each field in the event list contains the number of incoming activities to the event (referred to as the count) and the first position of the activity in the list of activities starting with that event (referred to as the location). The most important features suggested by these two structures are:

- The list of activities is ordered based on the predecessors, which are the initial events of the activities, and thus organized into groups of initial events. Each group can be iterated until the predecessor flag indicates the last element of the group.

- The initial location of the group, or its first activity, is perserved as the location in the event list. If it is a terminal event, a special symbol is used instead of the first edge location.
- The successor in the activity list and the first location in the event list provide communication between the two lists.
- The count in the event list represents the number of incoming edges to the event. It indicates whether the event is ready to be processed or whether it must wait for its predecessors to be processed.

At the beginning of the process, we iterate the event counts and look for possible zeros. If we find a zero indicating the initial network event, we assign it a serial number and set the count to -1. We then reduce the count of all its successors by 1. If any success has the count of 0, we set the flag F to indicate the creation of a new initial event. We do this for all events with a count of 0. After the iteration over the counts is complete, we check the flag F. If it has been set, a new iteration begins because there are additional events to process, and we reset the flag. If the flag F has no been not set, it indicates that:

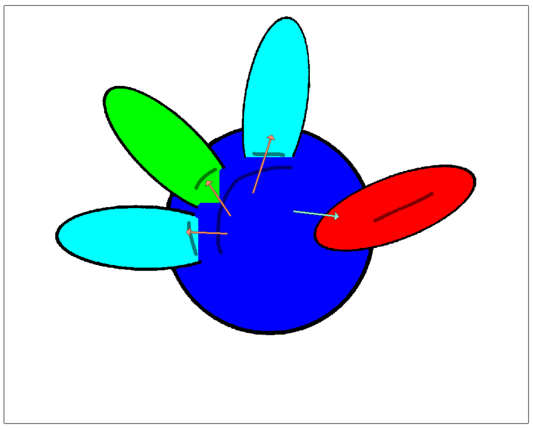
- The topological order has been successfully completed.
- There exists a loop in the network (graph).
- The network is currently undergoing segmentation procedures.

The author further suggests that the need to search for zeros can be avoided by maintaining a separate list of initial events. This list would be updated each time the number of successors of an event reaches 0 and then added to the queue.

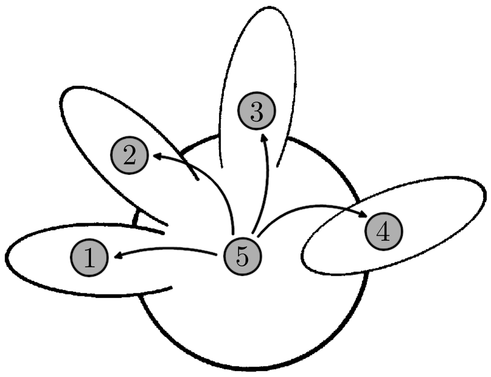
3.2.2 Simplified Kahn's algorithm

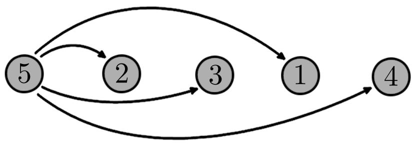
In this case, only the first two results of the algorithm are important: topological order generation and loop detection. In fact, no parallel segmentation processes will be performed in parallel with this algorithm.

The management of structures in this work differs from that in the original article. Actually, our graph is significantly smaller than the graph presented in the article. Furthermore, we try to integrate the algorithm with the rest of the pipeline by using only the core of the algorithm and avoiding the sorting steps. Our graph $G(V, E)$ consists of a set of nodes V that represent segments and oriented edges E between these nodes that represent user-specified arrows. We also utilize an empty list L and a set S containing all initial nodes. The algorithm follows the pseudocode described in Figure 3.6.



(a) : The input with arrows





(b) : The formed graph   **(c) :** The topologically sorted graph

Figure 3.5: To create the topological order of the segments, as shown in image **(c)**, the user must place arrows on the segmented image **(a)**. The arrows form the graph visible in image **(b)**.

There are multiple methods for detecting loops in the graph, depending on the implementation. As the pseudocode in Figure 3.6 shows, we can check if E is empty. If it is not, then the graph contains a loop. However, since we need to perserve the edges, we can use a different approach to detect cycles. One other method compares the number of nodes with the last assigned serial number [Kah62]. Another method is to identify nodes that do not have their count set to -1 as in the original algorithm [Kah62]. If there is a cycle in the graph, nodes in the loop cannot have the count of incoming edges equal to zero or -1 because there exists a path in the loop that leads from a node to its predecessor. Another approach to loop detection is to maintain a flag indicating which nodes have been used and which have not. This flag should initially be set to zero. If a newly visited node has the flag set to 1, this indicates the presence of a cycle in the graph. However, this approach

```
while $S \neq \emptyset$ do
    $S := S - \{n\}$   and   $L := L \cup \{n\}$
    for $\forall m \in V$ having edge $e : n \rightarrow m$ do
        $E := E - \{e\}$
        if $m$ has no other incoming edges then
            $S := S \cup \{m\}$
    endfor
endwhile
if $E \neq \emptyset$ then
    $G$ has at least one oriented cycle else
    $L$ contains topologically sorted nodes.
```

Figure 3.6: The figure illustrates the Kahn's algorithm for creating the topological order, as described by Sýkora et al. [SSJ+10]. G is the processed graph, S indicates the set of the starting nodes, L is a list that contains the topological order, E is the set of the edges, V is the set of the nodes, n is the selected node, m is the neighboring node.

requires the use of an additional flag for each node.

To retrieve the final depth levels of each segment, we must iterate over a topologically sorted list of nodes L. In the list, each node represents a segment. The initial depth level $d(v)$ of all nodes $v \in V$ is equal to 1, 0 is reserved for the background. When we visit the first node v in L, we follow the following steps:

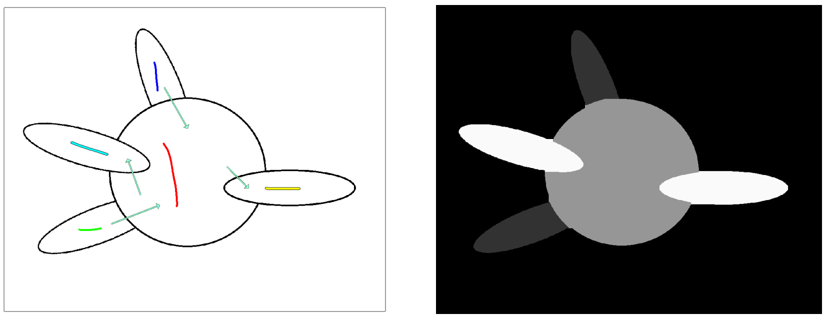
1. Search for all successors s of v.
2. Compute their depth as $d(s) = \max(d(s), d(v)+1)$, finding the maximum of their current depth and the depth determined by the relationship between s and v.
3. If v is the last node of the list, stop the procedure.
4. Select v as the next node in L and continue from the first step.

The depth levels in the output indicate the distance user's the from segments, with 0 representing the farthest segments (i.e. background) and higher numbers indicating increasing proximity.

Both the depth levels and the topological order are used in estimating of occluded segments and extraction of segment boundaries in the following section.

3.3 Segment shape approximation

The data obtained from the previous algorithms allows us to fill in the occluded parts of the segments. An example of the input for this step is displayed in Figure 3.7. We are currently trying to solve a diffusion problem that creates a smooth shape in occluded areas. The diffusion process should create a transition (gradient) between known boundaries that results in a protrusion into the region of closer segments. The diffused boundaries will then be used for the missing part of the occluded segment. The principles depicted in this part of the pipeline correspond to the Completion section of the Ink-and-Ray pipeline [SKv$^+$14].



(a) : Segmented image with the arrows

(b) : Depth layers

Figure 3.7: Picture **(a)** contains the segmentation result and the relative depth order provided by the user. Image **(b)** shows the absolute depth order represented as grayscale values. The white areas correspond to the segments closest to the user, while the black area represents the background.

3.3.1 Salient and illusory surfaces

The article by Geiger et al. [GPR98] serves as the foundation for the shape approximation in the Ink-and-Ray pipeline [SKv$^+$14]. The algorithm proposed in this work is based on the algorithm presented in the referenced article. The article explains how the human visual system perceives illusory surfaces in images and why some are more difficult to perceive than others. They also present an algorithm that allows to estimate the salient and illusory shapes. The process of the surfaces identification consists of several steps:

1. All edges and junctions in the image are located.
2. At each junction, we assign a set of hypotheses for each possible configuration of the junction. These hypotheses interpret the possible configurations of the surfaces in the junctions. Using this set of hypotheses, multiple hypotheses are assigned to several pixels.

3. The hypotheses are diffused by assigning each pixel a probability that indicates whether it belongs to the hypothesis. The diffusion process is blocked by the edges of the drawing.

4. The winning configurations are selected. An example of these hypotheses is shown in the images 3.8b and 3.8c. The optimal configuration is chosen based on two criteria. The first criterion is biased towards the smooth shapes, interpreting the L-shaped junctions as the T-junctions. The algorithm attempts to minimize the L-junctions, but this approach may not always be accurate. To compensate for this bias, the algorithm also considers the entropy of the diffusion as the second criterion. The entropy is measurable solely in the visible pixels and favors diffusions with a probability closer to 1.

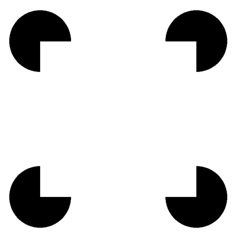
5. Upon receiving the winning hypotheses, we have determined that the surface with the one with the highest entropy per pixel is salient. The probabilities assigning pixels to surfaces distinguish the obscured surface from the one on top. The winning hypotheses result in the original shapes of the salient and illusory surfaces, as depicted in the images 3.8d and 3.8e.

As it has been shown, the algorithm produces both occluded and illusory shapes. Although this work and the Ink-and-Ray pipeline [SKv^{+}14] focul only on salient surfaces, the occlusion problem still needs to be addressed. In this work, we select known and visible regions and make hypotheses based on them. These hypotheses will then be diffused to obtain occluded shapes.

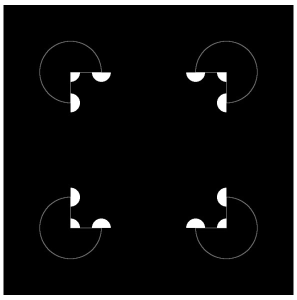
3.3.2 Occluded shapes estimation

In comparison with the article by Geiger et al. [GPR98], our only concern is occluded shapes, which are necessary to create the 3D model. We do not need to calculate which segments are occluded, as this information is provided by the user. It should be mentioned that we do not work with illusory surfaces.

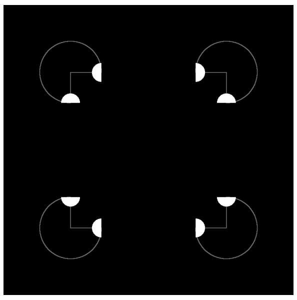
The problem we face is binary, and may seem similar to the segmentation Section 3.1. We have a segment whose boundary we want to recover, along with other segments potentially covering it, which together form a group. To formulate this problem as binary, we will mask this group as a second segment. Following the example in Figure 3.7, we can prepare to recreate the rest of the blue segment. Figure 3.9 displays the blue segment and the boundary conditions that determine the hypotheses. The first condition is white pixels directly bordering black pixels, and the second condition is black pixels bordering gray pixels. The hypotheses will be diffused within the area surrounded by these conditions.
The LazyBrush algorithm was used for image segmentation, but it may not be the best solution for this problem. The grid cutting algorithm produces segments with L-shaped boundaries that fill the gaps in the contours, which



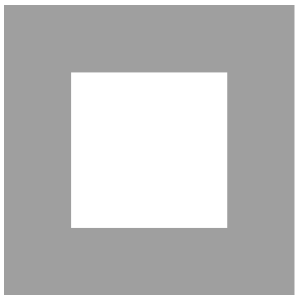
(a) : The Kanizsa square



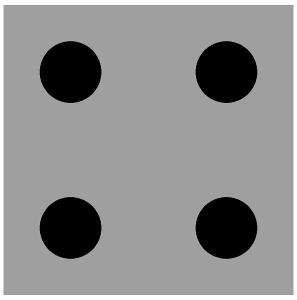
(b) : The hypothesis 1



(c) : The hypothesis 2



(d) : The diffused hypothesis 1 with threshold



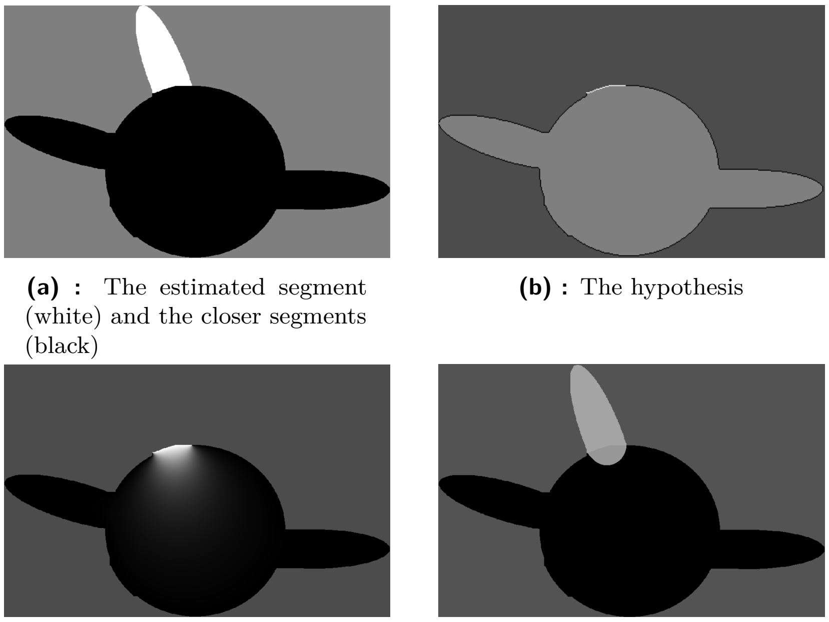
(e) : The diffused hypothesis 2 with threshold

Figure 3.8: Image **(a)** displays the Kanizsa square, which is an illusory white square which occludes four black circles. The images **(b)** and **(c)** show boundaries visible in image **(a)**, along with the two winning hypotheses. These hypotheses are formed at the junction areas and diffused. Notably, there are four L-junctions and eight T-junctions, which were converted from the former L-junctions. The images **(d)** and **(e)** display the result of the threshold of the diffused hypotheses.

makes it unsuitable. Moreover, in some cases, one segment may cover the entire computational space, leaving the other segment with only its own labeled area. To create a smooth boundary, an alternative approach is necessary. The problem we want to solve can be formulated using a different energy function:

$$E = \sum_{\{p,q\} \in N} w_{p,q} |x_p - x_q|^2 \tag{3.2}$$

where E is the total energy, while p, q represent neighboring pixels. N is the set of neighboring pixels that form the 4-neighborhood of any pixel. The weight $w_{p,q}$ between pixels p, q is calculated based on the intensity of the image pixels. The weight value represents contours or other obstacles in the image. In this section, we only encounter obstacles at segment boundaries, since we do not use the original image. As a result we can assign $w_{p,q}$ a value of 1 for all $\{p, q\} \in N$. The assigned labels $x_p, x_q \in \langle 0, 1 \rangle$ represent the probability of assigning a pixel to two segments and are the result of the diffusion process. In contrast to the LazyBrush algorithm, which directly assigns labels 0 and 1 to the pixels, this method assigns the probability of



(a) : The estimated segment (white) and the closer segments (black)

(b) : The hypothesis

(c) : The diffusion result

(d) : The threshold result

Figure 3.9: The task we aim to solve is to recreate shape of the overlapped segment marked by the white area in picture **(a)**. Picture **(b)** displays the boundary conditions that will be diffused. The white and black pixels represent the boundary conditions and the light-gray area is used for computation. The diffusion result for the blue segment can be found in image **(c)**, while the thresholded diffusion results for the overlapped segment are displayed in picture **(d)**.

their assignment to the occluded segment.

The problem was expressed using an energy function. According to the law of conservation of energy, we can set the energy derivative to 0 ($E' = 0$) which leads to the following result:

$$\frac{\partial E}{\partial x_p} = \sum_{i=1}^{4} 2(x_p - x_{q_i}) = 0, \tag{3.3}$$

$$4 * x_p - x_{q_1} - x_{q_2} - x_{q_3} - x_{q_4} = 0, \tag{3.4}$$

$$\Delta x = 0. \tag{3.5}$$

Equation 3.4 expresses the behavior in all pixels and is summarized as a Laplace of an image with equation 3.5. The behavior depicted in equation 3.4 can be observed in Figure 3.10 when multiplying the equation by -1. The sum of the coefficients of all five pixels is zero, including the pixel x_p and its 4-neighborhood, which are the pixels x_{q_1}, x_{q_2}, x_{q_3} and x_{q_4}.

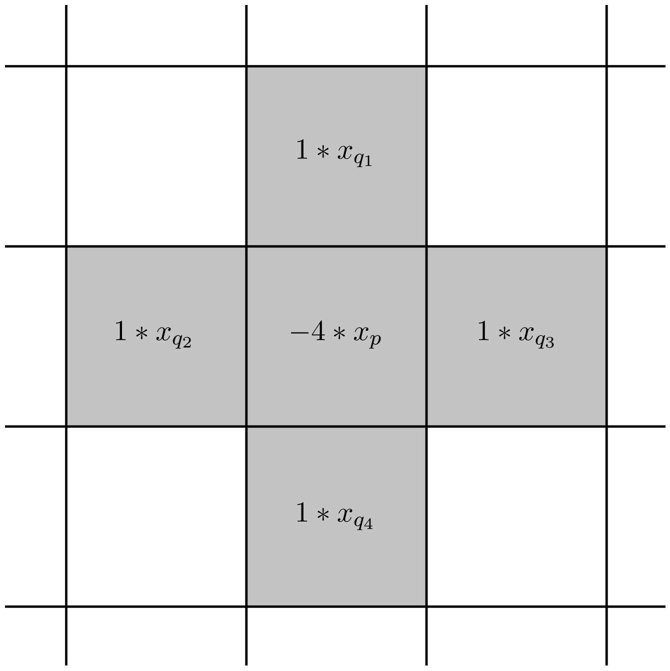


Figure 3.10: The Laplacian of the image is calculated using the 4-neighborhood of the pixels. The sum of the coefficients in all pixels of the 4-neighborhood, including the pixel x_p and its neighboring pixels x_{q_i} where $i \in \{1, 2, 3, 4\}$, is equal to zero.

The initial conditions for this problem are the boundaries of the upper segment group. The first condition is the boundary adjacent to the estimated segment, which is labeled as $x_t = 1$. The second condition is the boundary formed from the upper segments adjacent to the remaining segments, which will be labeled as $x_s = 0$. An example of this configuration is shown in Figure 3.9b. To diffuse the boundary conditions, we compute the Laplacian of the image.

There are several methods for solving the diffusion problem. Indirect methods, such as the Jacobi or Gauss-Seidel iterative methods, are well-known but can be time-consuming for large images. Direct methods that use sparse matrix solvers to compute the diffused pixel values can be more efficient. These methods can be applied to a linear system consisting of a formulated sparse incidence matrix with weights, and with boundary conditions included. However, even these methods can take a long time for larger images.

To improve the computation speed, we use two optimization techniques. The first technique involves constraining the computational space. Initially, a bounding box is created around the current segment and all segments closer to the user. Then, only the necessary pixels are used. Specifically, the boundaries depicting the closer segments define the computation area, as illustrated in Figure 3.9.

The other optimization technique discussed here involves utilizing the variable kernel [JCW09]. To successfully employ this solver, three conditions must be met:

1. creating an accurate initial guess to ensure proper convergence with the variable kernel,
2. determining the distance of each pixel from the boundaries. During the computation, the distances are gradually shortened. Initially, however, it is necessary to transfer information from the farthest possible pixels, which are the segment boundaries, to the currently computed pixel. The algorithm obtains the distances using the distance transform of Felzenwalb et al. [FH12].
3. The data from the previous steps are utilized to obtain the final result during a few dozen or a few hundred iterations instead of hundreds or thousands. Table 5.1 in Chapter 5 provides a comparison to the Gaussian-Seidel method.

The combination of optimization techniques reduces the computation time of the boundary condition diffusion process. The diffused data from Figure 3.9c, which are in the range $[0 - 1]$, are thresholded to obtain the final segment shape shown in Figure 3.9d. The occluded shapes and their boundaries are computed and added to the project file to create the 3D model.

Chapter 4

Implementation

The project was implemented in C++17 in Visual Studio 2019. Its objective is to segment the image, provide depth ordering of the segments and approximate all hidden parts of overlapping segments. At the end, a zip file of the project will be prepared to be used in the Monster Mash software. This will summarize the use of the application followed by a description of the implemented algorithms and structures.

4.1 User interface

The application is controlled by interacting with the image and key-bound commands. The processed image is displayed in a 1000x800 resolution window, following the Monster Mash tool. Any uploaded image is scaled and padded to match this resolution. The workspace does not contain any visual elements other than the image. The console displays information about running processes and other important details. The application has four modes that are essential to create suitable data for Monster Mash:

1. drawing mode,
2. depth mode,
3. segment picking mode, and
4. area selection mode.

The main input data of the application comes from the cursor. The keys control initializing procedures and various optional application controls. In drawing mode, users can draw scribbles on the window, change colors, change the type of scribbles (as discussed in Section 4.2 about LazyBrush implementation), or increase blur and contrast. These features are important because they can suppress or even remove visible image distortions caused by scaling operations, noise, or poor drawing quality. The use of all tools will be described later in this chapter.

Depth mode allows the user to add two types of arrows to the image. These arrows are added by selecting the farther segment with the first click and the closer segment with the second click. Further details on arrows are given in the text below in Section 4.3.

Segment picking mode allows the user to select an existing segment label from the image, similar to the color picking tools found in 2D editing software such as GIMP [dt23] or Photoshop [Inc23]. This allows the user to improve segmentation results by providing additional scribbles of the selected segments. However, the segmentation process must be restarted for the new scribbles to take effect.

The last mode allows to select areas that will be forcibly separated or merged in the final 3D model. To start selecting an area, either the left or right mouse button must be clicked. A rectangular area appears that follows the cursor from the position of the first click. A second click determines whether or not the boundary passing through the selected area merges with another segment. Left-clicking (LMB) on the second click will create normal boundaries, while right-clicking (RMB) will denote merging boundaries in the area. This tool should be used with caution as it affects all segments below it, including directly set boundaries and estimated boundaries of overlapped segments. The advantage of this tool is that it does not alter the shape of the segments, only the type of the boundary. If users are not satisfied with the results of the tool, they can adjust the boundaries either in a 2D editing tool or in the Monster Mash tool.

When we talk about merging boundaries, it is meant that in the final 3D model created using the Monster Mash tool, the segments will be seamlessly connected in areas where merging boundary is defined. Blur and contrast tools can either improve the visual quality of the image where artifacts appear, or they can help in the segmentation process to achieve better results.

The application key bindings are listed below:

- **Q / ESC**
  The application is terminated using this key binding.
- **C**
  The command changes the color of the scribble. Using the RGB/HSV color space element, the desired color can be selected (received by pressing the enter key). If this UI element does not work correctly, the desired color can be entered into the console in RGB format in the range [0-1]. Note that this command is only available in drawing mode.
- **H**
  This command toggles between soft and hard scribbles discussed in Section 4.2 on the implementation of the segmentation algorithm. Please note that this command is only available in drawing mode.
- **D**

Here you can switch to depth mode from any other mode or switch to drawing mode from depth mode.

- **P**
  Users can use this key to enter segment picking mode from any other mode. The drawing mode is accessible from the segment selection mode.
- **V**
  This command can be used to enter selection mode from any other mode. Alternatively, drawing mode can be entered from selection mode.
- **S**
  The progress is saved in a binary file along with the current segmentation layers, scribbles and the modified original image. The original image is not overwritten. Additionally, it saves the screen, the depth order and segments in image files to provide a better overview of the application state.
- **X**
  This key is used to load the saved progress.
- **B**
  The predefined blurring tool can be used to improve visual or segmentation results by blurring the image. Note that this command is only available in draw mode.
- **K**
  This button accesses a tool that increases the contrast by a predetermined value. It is only available in drawing mode.
- **R**
  This command resets the application.
- **M**
  This key starts the segmentation process.
- **O**
  Using this key, the borders of the segments can be approximated to create the Monster Mash project zip file. Depth order is required for this operation to work.
- **A**
  The A key toggles the screen display and can switch between four possible displays: segmented image and user input with original image, segmented result with original image, segmented result alone, and original image with user input.
- **LMB**
  In drawing mode, the LMB is used to draw new scribbles. In depth mode, the first click selects the more distant segment and the second

click selects the closer one. The result is a green arrow. In selection mode, the first click triggers the selection and the second click selects the region in which segment boundaries will not be merging.

- **RMB**
  In drawing mode, the background scribbles are drawn. In selection mode, the boundaries of all segments within the selected area will merge upon the second click.

- **shift+D**
  Using this key to refresh the depth information in the window.

- **shift+V**
  This command resets the selection areas.

- **shift+R**
  Only the blur and contrast are reset using this key.

- **shift+LMB**
  In drawing mode, it is possible to create another scribble of the last segment or the selected segment using the segment picking tool. In depth mode, a red arrow appears upon the second click.

4.2 Segmenting algorithm

The LazyBrush algorithm, explained in Section 3.1, is used to segment the image. Users provide scribbles in the regions they want to segment using the two available types of scribbles listed in the key-bindings in Section 4.1: hard and soft scribbles. Pixels drawn over using hard scribbles are directly assigned to them, while other pixels are labeled depending on the segmentation results. The second type of scribbles follows the majority rule. If a scribble appears in an area where another scribble has more influence, whether it is a soft or hard scribble, the area below that soft scribble will belong to the other scribble. This behavior is useful in situations where there is a risk of accidentally brushing over into other areas. The soft scribble will fill in the intended area, and any area that was accidentally brushed over will still belong to the other scribble, with no artifacts remaining in the resulting segmentation.

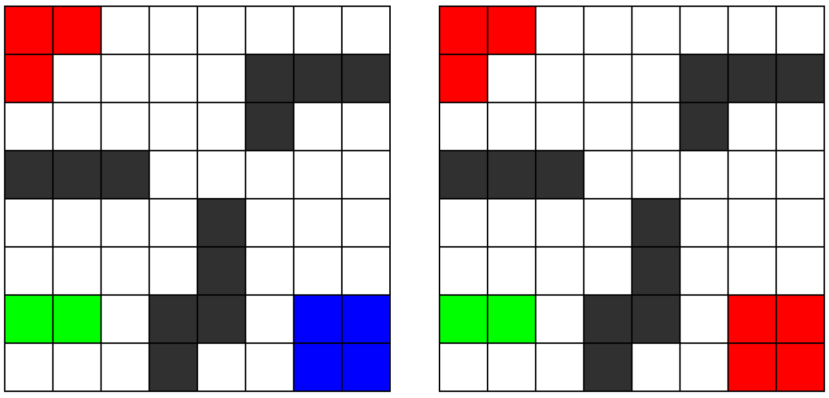
Although the LazyBrush algorithm solves the binary problem of assigning pixels between two labels, the following steps adapt it for multiple labels:

1. Initialize a set of active labels L based on the user's input and a mask M of unlabeled pixels.

2. Identify all unlabeled regions R in M that intersect with scribbles containing only one label l_r. For each found $r \in R$, assign labels in M to l_r. If there are no other regions in M containing scribbles with label l_r, remove l_r from L.

3. If L is empty, stop.
4. Select an arbitrary label $l \in L$.
5. Build a graph G from all unlabeled pixels in M.
6. Connect the pixels labeled with l to the terminal node S, and the pixels seeded with labels from $L - \{l\}$ to the terminal node T.
7. Solve the max-flow/min-cut problem on G using S as the source node and T as the sink node.
8. Set the label in M to l at the pixels, where the corresponding graph nodes (pixels) were assigned to the terminal node S.
9. Remove the label l from L and go to step 2.

To completely understand the process, we can follow the example pipeline that starts in Figure 4.1a. Both types of scribbles can be present in the segmentation process and used for multi-label segmentation. In the segmentation process, we always count one segment against the others and then remove the assigned pixels from the next calculation, as shown in Figure 4.1b.



(a) : Input data **(b) :** Preparation for the first label

Figure 4.1: The figure shows the initialization of the LazyBrush algorithm. The user input is contained in the 8x8 grayscale image **(a)**. The input consists of three labels: blue, red and green. The dark pixels in the image represent contours. Figure **(b)** is used to prepare the input for the computation of the green segment.

The processed image can be viewed as a grid whose nodes are pixels and whose edges are represented by 4-neighborhood of the pixels. The weights between nodes are determined by the smoothness term $V_{p,q}$, as shown in Figure 4.2. The smoothness term $V_{p,q}$, which represents weights/capacities between pixels p and q, is calculated using the weighting function

$$w_{pq} = 1 + K * min(I_p, I_q)^2 \tag{4.1}$$

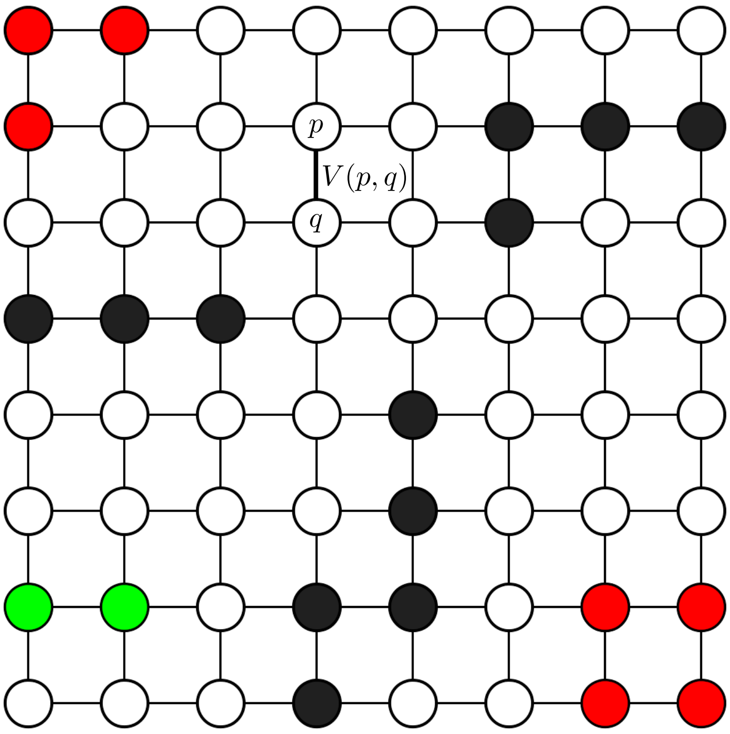


Figure 4.2: The graph in this figure is represents the situation in Figure 4.1b. Each node corresponds to a pixel and edges are only present between neighboring pixels. The weight between two neighboring pixels, p and q, denoted as $V(p, q)$ and is set according to the equation 4.1, which depends on the intensity of the pixels.

where the intensities of pixels p and q are represented by $I_p, I_q \in \langle 0, 1 \rangle$. To increase the contrast, a constant 2 was added using gamma correction, which helps prevent problems with segmentation of light-gray boundaries close to white. The weighting function ensures that the segment boundary seeps into the darkest pixels along the outline, rather than stopping at the edge of the drawn outlines. However, in the case of thick contours, the segment may accidentally bleed into other unintended areas below the outlines. This behavior is inevitable due to the nature of the algorithm, as shown in the first row of Figure 4.5.

To improve the resulting shape and suppress noise in the image, we implemented an optional Gaussian blur to thin out dark areas and create a gradient around them. Using this tool, users can create more appropriately shaped segments, as demonstrated in the second row of Figure 4.5. The resulting shapes then tend to converge towards the center of the contours.

If the image contains faint outlines, it is natural to increase the contrast of these lines so that the algorithm does not accidentally neglect them. Segmen-

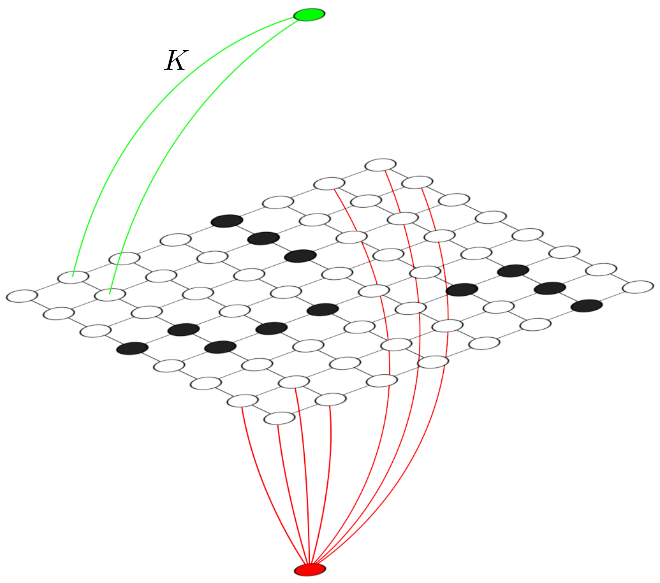


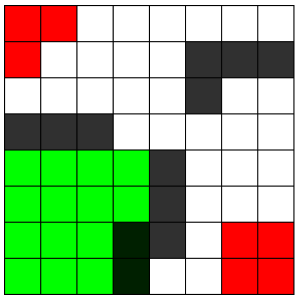
Figure 4.3: The figure illustrates how the assignment of the sink and source pixels are assigned to graph in Figure 4.2. The data term K is assigned to the edges between the source/sink and the scribbled pixels, while the edges between the source/sink and the other pixels have set a value of 0, indicating no connection to the source/sink.

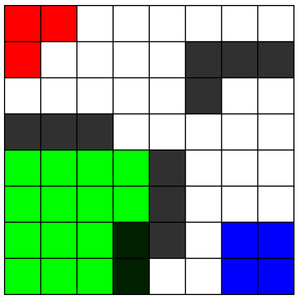
tation is not the only step that requires proper outlines, and as mentioned, the LazyBrush algorithm already handles contrast. The last step of this framework also depends on the intensity of the contour pixels. Therefore, a contrast enhancement tool has been added to the application. However, the contrast enhancement tool also amplifies the noise in the image.

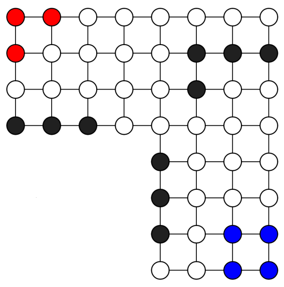
The min-cut algorithm operates with a single sink and one source, both of which are imaginary pixels located either above or below the image. These pixels are connected to the scribbled pixels as shown in Figure 4.3. As mentioned in Section 3.1, the LazyBrush algorithm utilizes the data term D_p. If the user has not brushed over a pixel, the data term for that pixel is set to zero. If a pixel is scribbled over, the data term is set to a high number denoted as K. For hard scribbles, K is proposed to be as high as the image perimeter. In our implementation, we set K to 4000 because our images have a resolution of 1000x800. For soft scribbles, the number is lower and is se to $K/16$. In the example Figure 4.3, we assigned a value of K to the data term between the source and scribbled pixels because we expected hard scribbles on the input.

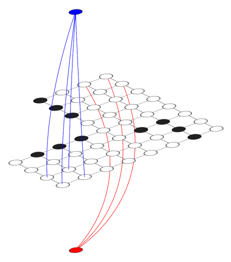
Multi-label segmentation is performed by following the steps described in Section 3.1. The remaining steps can be seen in the example Figure 4.4.

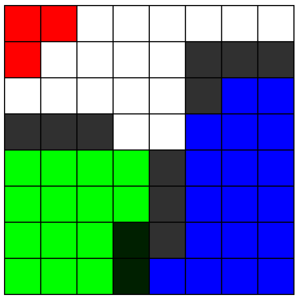
The segmentation result for the green segment is shown in Figure 4.4a. The remaining parts of the image must also be assigned to the appropriate segments, since there is no area covered by only one segment. Then the labels for the next segment were prepared, as shown in Figure 4.4b. The graph produced in this step does not include the area assigned to the green segment in Figure 4.4c. The nodes of the graph are then assigned smoothness and data terms accordingly. We then complete the computation of the min-cut algorithm on the graph as displayed in Figure 4.4d. The result for the green scribble can be seen in Figure 4.4e. Since only one label remains, the remaining area covered by its scribble is assigned the last segment. The final result, shown in Figure 4.4f, was obtained by assigning the last scribble.


(a) : First partial solution


(b) : Preparation to the next label


(c) : Graph representation


(d) : Addition of the label pixels


(e) : Second partial solution

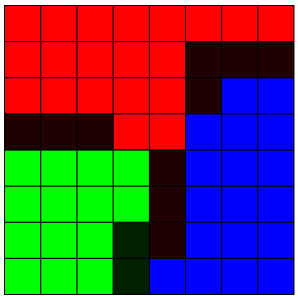

(f) : Final result

Figure 4.4: The figure illustrates the remaining steps in the multi-label segmentation process. Figure **(a)** shows the outcome for the green segment. Computation for the remaining labels is required. The computation for the blue scribble will be performed with the preparation displayed in image **(b)**. The green segment and its scribble will be excluded from further computation, as shown in graph images **(c)** and **(d)**. Once the grid cut algorithm is complete, the area assigned to the blue segment will be determined. Since only one segment remains, the remaining area will be assigned to the red scribble. The resulting segmentation is displayed in image **(f)**.

The GridCut library [SJ] allows us to prepare graphs and calculate min-cuts for splitting the image between two segments. The data term is assigned separately for the sink and source, while the smoothness term is calculated between neighboring pixels in the image.

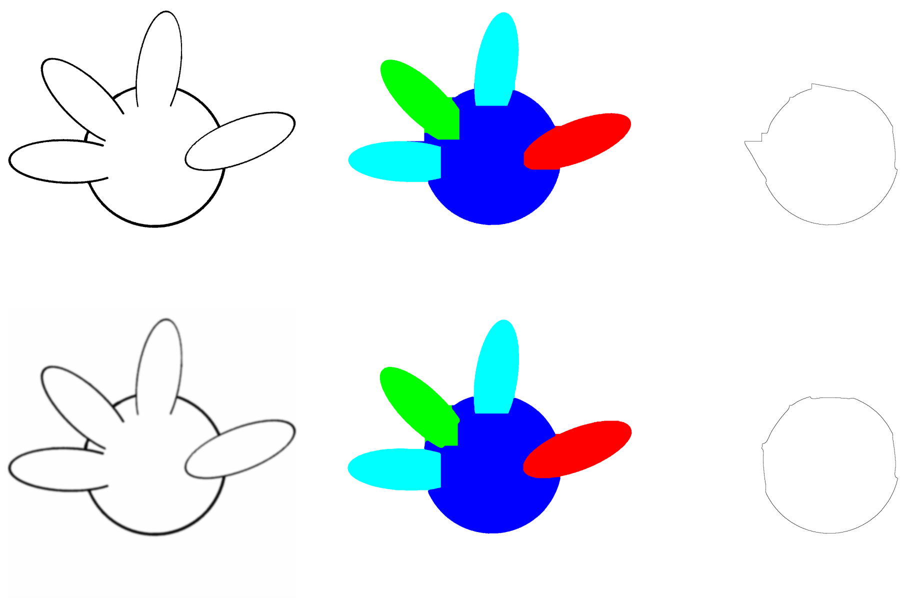


Figure 4.5: The figure illustrates the recreation of the border of the middle segment, which is overlapped by the surrounding segments. The first column contains the input images, the second column shows their segmentation results, and the third column displays the approximated borders. The first row depicts the pipeline of the unprocessed input, while the second row shows the process of the blurred input.
The processed areas for the red and green segments are very similar. There was significant improvement in the area of the cyan segment pointing to the left. The segmentation process produced a large protrusion outside of the middle segment, where the minimal cut was located.

Listing 4.1: The color map structure contains primarily segmentation results and colors mapped as scribble indices.

```
Color Map {
    integer[]: indices of the segments mapped to
               the pixels
    integer: image width
    integer: image height
    byte: active pixel
    integer[2]: amount of the used hard and
                soft scribbles
    RGB[256]: assigment of RGB colors to scribble
              indices
}
```

At the beginning of the program, the framework generates background scribbles around the image. These scribbles allow users to focus only on the drawn area. If additional background scribbles are needed, the user can use RMB as described in the user interface Section 4.1.

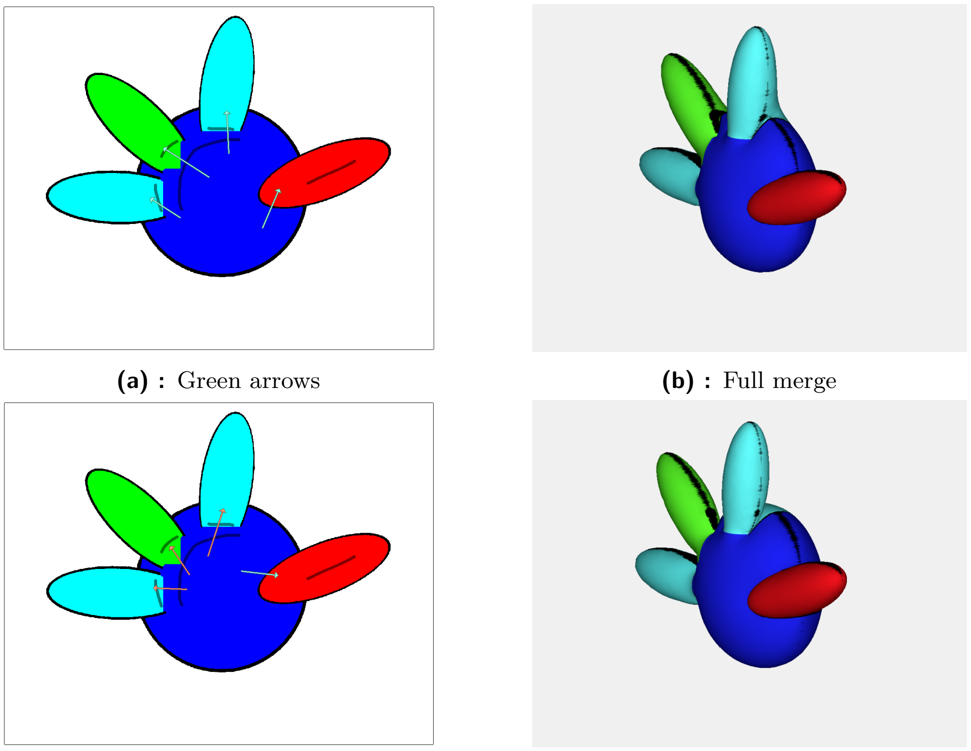
The segmentation process uses several structures. The first stores the scribbles and their intensity in the image. It contains indices of the drawn scribbles in each pixel, which allows us to ignore the color of a segment during the computation so that multiple segments can have the same color. The second structure is the input image converted to a grayscale image, which contains only the intensities of each pixel. The segmentation results and colors mapped to scribble indices are stored in another structure, which is shown in pseudocode 4.1. This structure provides information in other parts of the shape approximation process. The limit of labels is 256, divided between soft and hard scribbles, with 128 for each. This limit is set because it is not anticipated that more scribbles will be needed.

Using the LazyBrush algorithm, we were able to divide the image into multiple segments. These segments require further processing, beginning with their depth order.

4.3 Depth ordering

The relative depth order between segments is manually created by the user, as discussed in Section 4.1 of the user interface. The first selected segment is considered more distant, while the second is considered closer. There are two types of arrows: green arrows indicate the merging boundary of the closer segment in the occluded area, while red arrows keep the segments separated in the resulting 3D model. The use of both types of arrows depends on the intended purpose. Both cases can be seen in Figure 4.6. Similar to our example, the green arrows can be used for example for fingers connected to the palm or for similar situations where we want to completely connect two segments. The red arrows should be used in cases where we want to restrict the connection to the closer segment in the area where there is an opening in the contour or to completely separate the selected segment. In some cases, such as animal ears or limbs that are connected to the body only by the base, it is important to keep the border of the body segment separate from the limbs. The connection should then be controlled solely by the open outline of the closer segments.

When the user adds new arrows to determine the depth order, an oriented loop may be formed. In this context, loops are not desirable because we cannot determine the proximity of segments. To address this issue, we can use an algorithm that checks the topological order and detects the presence of a loop. Kahn's algorithm described in Section 3.2, can help with loop detection. The location of the loop in the graph is unnecessary information. The algorithm implemented in this work removes the newly added arrow and alerts the user if a loop is detected. The user is also informed if the same segment is selected on both clicks or if the background is selected. The displayed alert consists of a short animation that is indicated by a dynamic change in the color of the arrow before it disappears. During the animation, other



(a) : Green arrows

(b) : Full merge

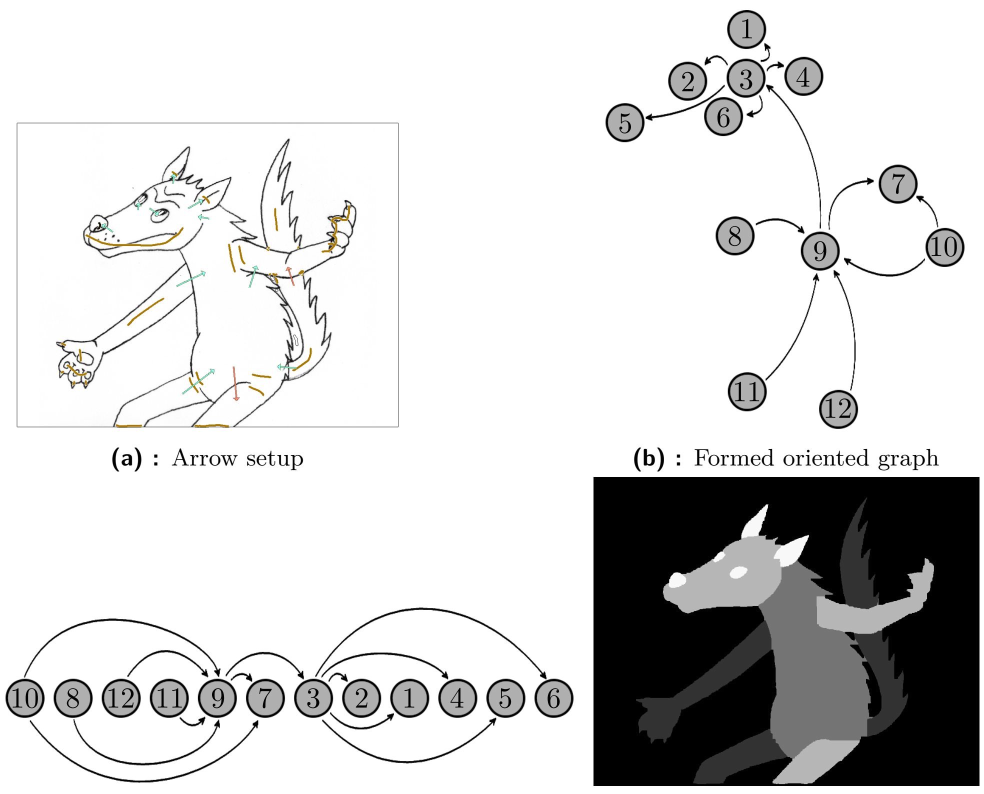
(c) : Red arrows

(d) : Partial merge

Figure 4.6: In the first row, the pictures **(a)** and **(b)** demonstrate the use of green arrows in the demo example. It is important to note that the blue segment area merges with its border to the other segments in the 3D view. Pictures **(c)** and **(d)** in the second row display the difference when red arrows are used. As shown in the image view, the merged area near the holes in the contours of the closer segments remains unchanged. The distinction is in the border of the blue shape, which is separated from the other shapes. In the example, the red segment is always connected by the green arrow. If a red arrow were used instead, the red segment would be completely isolated from the rest of the mesh.

interactions are blocked to to avoid distracting the user from the misplaced arrow. The most important information provided by Kahn's algorithm is the topological order of the segments, which is used to determine the absolute depth level of each segment. As described in Section 3.2, when iterating through the ordered array, each segment pointing arrows to other segments is to the left of the remaining segments in the topologically ordered array. During the iteration, each segment is processed and depth level is updated in its successors. Figure 4.7 displays the steps of the depth level computation.

Our implementation of Kahn's algorithm involves maintaining three lists. The first list contains the nodes, the second contains indices of the initial nodes and serves as a queue. The algorithm runs until this list is empty. The third list maintains the order of nodes and is initially empty. This list also contains segment indices and is used instead of managing sequence numbers.



(a) : Arrow setup **(b) :** Formed oriented graph

(c) : Topologically sorted nodes **(d) :** Formed depth layers

Figure 4.7: Picture **(a)** displays the user input with two types of arrows. Picture **(b)** presents the segments and arrows as an oriented graph. The segments are labeled with numbers from top to bottom and left to right for clarity. Picture **(c)** displays the topological order of the segments, where all arrows point from left to right. Other combinations of the topological order are possible as long as no arrow must point to the left. The final depth levels extracted from the topological order are shown in image **(d)**. (Wolf image source: ©Anifilm. All rights reserved.)

The order kept here is later used to retrieve the depth level of all segments.

Other structures are used in topological sorting. One of them is the structure necessary to track the location of arrows in the image for visualization purposes. The other is used to track edges between segments. The node used in a graph consisting of segments and arrows is explained in pseudocode 4.2. The node contains the list of outgoing edges and thew number of incoming edges. The depth information in the node represents the assigned absolute depth level.

Listing 4.2: The node class used in the graph.

```
Node {
    list of edges: edges outcoming from the node
    integer: number of incoming edges
    integer: the depth level assigned to the node
}
```

Having established the depth order between the segments, we can now concentrate on recovering the occluded shapes. It is important to note that we only need to obtain the depth level information before we begin approximating the occluded segments.

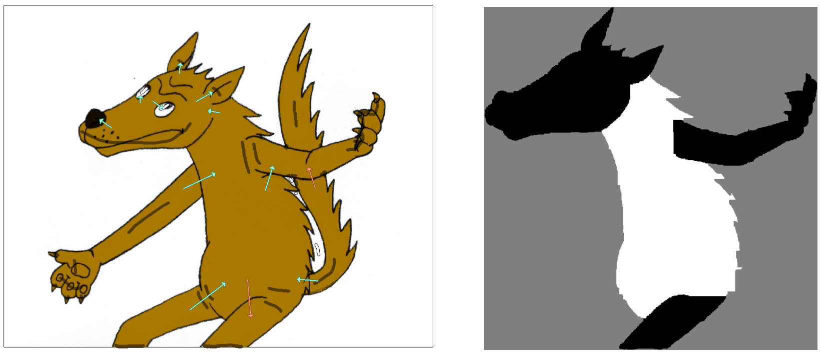
4.4 Shape approximation

The final step of the pipeline is to approximate the occluded shapes using the segmented image and depth data. In the preprocessing step, the image is searched to find the segment boundaries. The neighborhood of each pixel is searched, and if a neighboring pixel belongs to another segment, that pixel is marked as part of its segment boundary.

To create an efficient environment for shape approximation, we iterate over the depth levels of all other segments. This selects segments that are closer to the user than the current segment. Using the topological order, we can navigate from the segment adjacent to the current segment because all previous segments have an equal or lower depth level, as shown in Figure 4.7d. We can further refine the bounding box region around the current segment by determining the minimum and maximum coordinates of all segments involved in the shape approximation process. By creating a computational window in this way, we minimize the required computational space. In this process, we select not only adjacent segments, but also other closer segments, since the occluded segment may intersect them. This section will present the estimation method supplemented with a complete example of the wolf body segment approximation, starting with Figure 4.8.

Although there are many possible segment configurations, it is sufficient to generalize them to two types. In the first type of configurations, a segment is adjacent to two or more segments, including the background. In the first type of configurations, a segment is adjacent to two or more segments, including the background. In the second, a segment is completely surrounded by another. For the latter, then, there are two situations that we have to deal with. The first one describes the outline of the segment as an open or user-set merge area. In this case, we extract the boundary immediately from the previously calculated boundaries and mark the opening in the contour as the merging boundary. Ensure that the selected area described in Section 4.1 of the user interface is considered. However, in the latter situation, the segment cannot be merged with the rest due to the contour separation. The calculation of its boundary will stop because it would not be possible to merge the segment with the 3D model in Monster Mash. In the first image in Figure 4.8, the left eye segment is completely enclosed by the head segment and its outline is closed. Therefore, only its texture will be used in the final 3D model.

As we have already mentioned, some areas are merging because of the open contours. It is important to consider the possibility that the boundary follows the contour caused by the segmentation algorithm, and also the possible presence of small holes in the contour due to noise. These situations are handled by treating contours adjacent to dark pixels as closed.



(a) : Segmentation and depth input

(b) : Body segment preparation

Figure 4.8: Figure **(a)** shows the input for the algorithm. It also exemplifies a scenario where the segment representing the left eye has a closed outline and is fully enclosed by another segment. The image contains the scribbles, the segmentation result based on them and the arrows denoting the topological order of the segments. Figure **(b)** displays the area of a segment with occluded parts (white) and the areas that occlude the body segment (black). These areas are by the segments that are closer to the user than the white segment. (Wolf image source: ©Anifilm. All rights reserved.)

After preprocessing, each segment is processed separately, except for segments with closed contour, which were excluded from the calculation. The boundaries of the closed segments with open contours as well as the segments closest to the user are processed immediately based on the previously extracted segment boundaries.

If we want to estimate the occluded shape of segments using a variable kernel, we follow the following three steps:

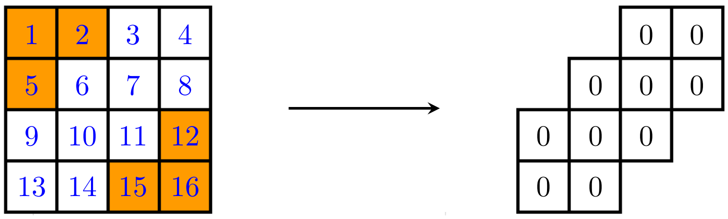
1. find an initial solution guess,
2. find a distance transform from the boundary conditions,
3. estimate the occluded shape of the segment.

4.4.1 The initial guess

Jeskhe et al. [JCW09] state that the initial guess is made by the Voronoi color image. However, our approach differs. First, we obtain a mask of the current segment and other segments closer to the user, as shown in Figure 4.8. We then scale down this mask using the nearest neighbor algorithm to avoid color smearing in the smaller version of the mask. The boundary of the reduced mask is then identified, forming the Dirichlet boundary conditions. These conditions represent the values around the boundary of the segments involved in the computation. The boundary conditions are divided into two types based on color. The first type includes the boundaries of the closer segments in the reduced mask except for the boundary in common with the current segment. These boundaries are indicated by the color black and the number 0. For the second type of boundary conditions, we use the part of the boundary of the currently processed segment shared with the boundary of the closer segments. The boundary is marked with white color and number 1. The boundary conditions for the main body of the wolf can be observed in Figure 4.10a. The computation area is further limited to the area under the overlapping segments because the space where the estimated boundary might be located is limited to the area formed by closer segments. The Laplacian, which represents the diffusion process of the boundary conditions, can be solved by directly computing the sparse matrix from Section 3.3 using a specialized solver such as the Eigen library [GB+21].

In order to obtain the Laplacian image, we need to solve the linear system $A * x = \mathbf{0}$ must be solved. The array x represents the pixels with unknown values, and $\mathbf{0}$ represents the desired Laplacian. The matrix A is sparse and contains in each row the coefficients of the 4-neighborhood of each pixel, except those that convey the boundary conditions, in each row. The main diagonal contains a coefficient of -4 (-3 and -2 in cases of image edges and corners). A value 1 is assigned to the neighboring pixels, specifically to the indices to the left and right of the diagonal, as well as the upper and lower pixels that are further away. To set up the linear system, we remove the rows and columns of the pixels with boundary conditions from the matrix A, as well as their corresponding positions in vectors b and $\mathbf{0}$, since we know their values. The sum along certain rows in matrix A is less than 0 because the boundary pixels have not been included. To account for the boundary pixels, we utilize the vector $\mathbf{0}$ and subtract the value of the boundary pixels from it at the positions of their neighbors. Figure 4.9 shows an example of this setup for a 4x4 image.

To maintain the smoothness of the diffusion, the solution for the scaled-down image, as shown in Figure 4.10b, is scaled back to its original size using the bi-linear scaling algorithm. The image is then used as an initial guess to ensure that no artifacts are present in the final result.



$$\begin{bmatrix} -3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 1 & -2 & 0 & 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ 1 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 1 & 0 & 1 & -3 & 0 & 0 & 0 & \dots & 0 & 0 \\ . & . & . & . & . & . & . & . & \dots & . & . \\ . & . & . & . & . & . & . & . & \dots & . & . \end{bmatrix} * \begin{bmatrix} 3 \\ 4 \\ 6 \\ 7 \\ 8 \\ . \\ . \end{bmatrix} = \begin{bmatrix} 0-2 \\ 0 \\ 0-2-5 \\ 0 \\ 0-12 \\ . \\ . \end{bmatrix}$$

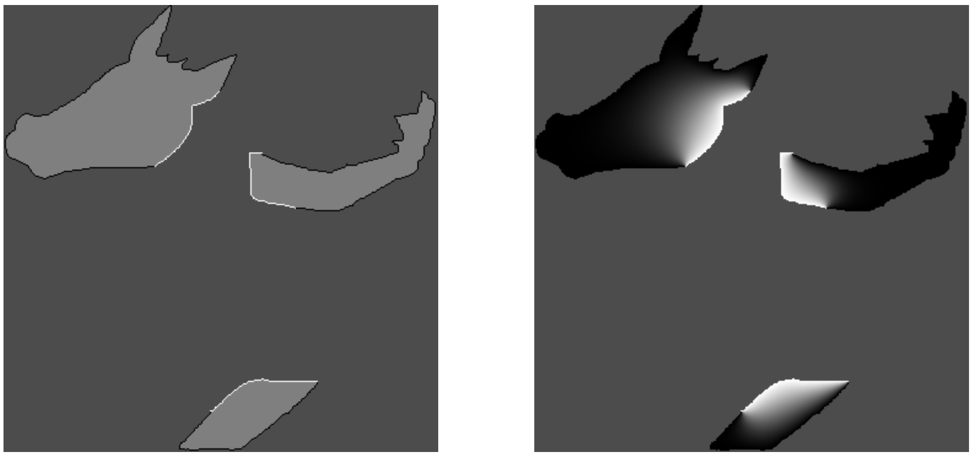
Figure 4.9: The left image displays a 4x4 grid with highlighted pixels at positions 1, 2, 5, 12, 15 and 16, which represent the boundary conditions. The objective is to compute the values of the remaining pixels. The right image shows the desired Laplacian. To solve this problem, we use the linear system $A * x = b$, where A is the matrix that describes the behavior of the Laplace operator, x is the array of pixels with unknown values, and b contains the Laplacian. The array x had the known pixels (boundary conditions) removed and their rows and columns in A that corresponded to those pixels. To include these pixels in the computation, the value they contained was subtracted from the Laplacian of the computed pixels in their neighborhood.

4.4.2 Distance transform

As stated in Section 3.3, the distance to the boundaries is necessary for computing with the variable kernel method. To initialize, we utilize a scan-line algorithm to search the 4-neighborhood of all segments for the borders, as we did for the scaled-down mask. The shape of the boundary is identical to that of the boundary conditions. However, in this case all borders have the same value because we want to find the distance to any border independent of its type. We then follow the Euclidean distance transform algorithm described by Felzenwalb et al. [FH12]. They presented the distance transform of a sampled function (a grid) as a function

$$D_f(p) = \min_{q \in S}(d(p,q) + f(q)) \tag{4.2}$$

where S is a sampled function, $d(p,q)$ is a measure of distance, and $f(q)$ represents an arbitrary function. The proposed algorithm solves a one-dimensional DT problem. For higher dimensions, the algorithm is applied separately to each dimension. The algorithm can be used with different



(a) : Boundary conditions

(b) : Diffused boundary conditions in computed areas

Figure 4.10: Figure **(a)** provides an example of boundary conditions. The white border belongs to the overlapped segment, while the black border belongs to the closer segments. The light-gray area denotes the space where the computation will take place. The dark-gray area is ignored as there was no segment that overlapped the current segment. Figure **(b)** displays the result we obtained from the matrix solver. The result will be scaled up using a bi-linear filter to maintain the smoothness and will serve as the initial guess for the diffusion algorithm.

measures of distance, including the L1 (Manhattan) and Euclidean distances. The Euclidean distance transform (EDT) utilizes the min-convolution of the input image (function f) and a parabola. The calculation of EDT is represented by the equation

$$D_f(p) = \min_{q \in S}((p-q)^2 + f(q)) \tag{4.3}$$

where S is a one-dimensional grid, $f : S \to \mathbb{R}$ represents a function on the grid, which defines the height of the root of a parabola in each pixel q. The EDT is defined by the lower envelope formed by all parabolas.

The algorithm's separability allows for independent processing of each dimension. Figure 4.11 displays the result of the distance transform based on the boundary conditions from Figure 4.10. The distance transform is created only in the desired areas from Figure 4.10. The DT is computed only in the selected areas, as the visible area of the body does not require estimation. Only its border parts are used.

4.4.3 The diffusion and the export

The third step in this pipeline involves applying the varying kernel to the initial guess. The initial guess, acquired from the Section 4.4.1, is then converged to the Laplacian of the image, except for the Dirichlet boundary

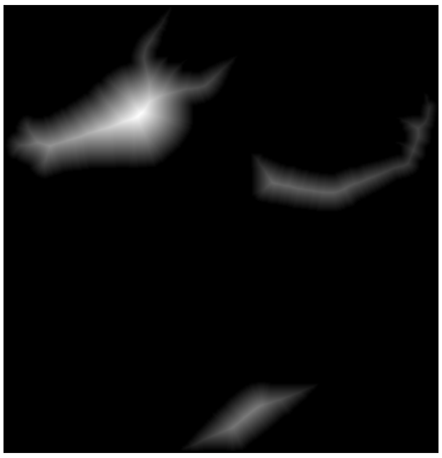


Figure 4.11: The Euclidean distance transform used for the reconstruction of the occluded body parts.

conditions [JCW09]. As previously mentioned, the computation will take place in a limited space denoted by the boundary conditions. The variable kernel method solves the main issue with the Gaussian-Seidel and Jacobi iteration methods, which is the reliance on information from one side of the image to another. Jeschke et al. [JCW09] have proven that the Jacobi method is guaranteed to converge for a typical Laplacian kernel. Even if the kernel is modified by moving the kernel members further away, the Laplacian will always be equal to zero. However, the array of unknown x will be different if the kernel size remains constant during the computation compared, compared to using a commonly used kernel with 4-neighborhood. To modify matrix A according to the kernel, the ones are moved further away from the diagonal by h for the left and right neighbors. The neighboring pixels above and below the currently processed pixel are moved away by a distance equal to distance equal to the image width multiplied by h. The value of h, which is selected for the initial iteration steps is the distance to the boundary pixels and is equal to the distance gained from the distance transform at each pixel. The modified kernel can be observed in Figure 4.12. It is reasonable to expect that the Gaussian-Seidel iteration method will behave similarly to the Jacobi method. Both methods share foundational principles and maintain properties that lead to convergence.

In this work, we use the Gaussian-Seidel iteration method expanded with the variable kernel. The speedup of this algorithm is shown in Section 5 when compared to the classical Gaussian-Seidel iteration method. In each iteration, we update the entire, similarly to the original Gaussian-Seidel iteration. The basic algorithm employs the 4-neighborhood of a pixel, as shown in Figure 3.10. However, in our case, we set up the kernel differently, as described in the article by Jeskhe et al. [JCW09].

In each step, a scale factor of $1 - \frac{i}{n}$ is kept, where n is the number of iterations and i is the current iteration number. By multiplying the distance of the pixel to the border in the distance transform with the scaling factor, the offset to all directions from the pixel in current iteration can be obtained. In Figure 4.12, which was previously mentioned, the distance from the computed pixel was set to three pixels using this method. The same approach as in the Gaussian-Seidel iteration is then employed, following the equation

$$x_p = (x_{q_1} + x_{q_2} + x_{q_3} + x_{q_4})/4 \tag{4.4}$$

where the $x_{q_i}, i \in 1, 2, 3, 4$ represent the pixels denoted by the offset from the current pixel. When using this technique, we can employ one of two strategies mentioned in the article [JCW09]: the shrink always strategy or the shrink half strategy.

The shrink always strategy reduces the kernel size in each iteration by the scaling factor of $1 - \frac{i}{n}$, where n is the number of iterations and i is the current iteration index. The shrink half strategy keeps the kernel radius maximal in the first half of the iterations, then continues as the former strategy. In our case, we used the shrink half strategy as it should converge faster than the shrink always strategy. In the final steps of both methods, the distance from the current pixel is reduced to 1, converting the variable kernel method back to the original Gaussian-Seidel iteration. However, it is important to note that the shrink half strategy requires at least twelve to fourteen iterations to converge correctly. To address this issue, we have set the minimum number of iterations to forty, resulting in satisfying outcomes. However, this approach may not be always sufficient as the results may not be fully converged. The number of iterations depends on the size and shape of the computed area, making it impossible to set an absolute number of iterations. Instead, we can define the maximum difference in a pixel before and after each iteration. Therefore, we use the variable kernel method for a fixed number of iterations and then continue with the basic Gaussian-Seidel iteration method until the change in all pixels is less than the specified difference.

The image diffusion is similar to that shown in Figure 4.10. To finalize the output, the image values are thresholded. The boundary conditions are colored white for 1 and black for 0. Pixels with the value 0.5 and higher are set to 1, and the rest are set to 0 creating the desired segment area. The remaining task is to set the segment boundary.

As previously mentioned, the type of boundary determines whether the part will be merged with the rest of the model or not. The problem of overlapped segments was mostly solved by using arrows. In cases where there are issues with occluded regions, selection areas were proposed to cover unintentionally merged or separated areas. It is also possible to encounter merging issues in segments that are on top. Not all dark areas are intended to define outlines, such as noise, shading, or other details. These issues can also be addressed by the selection tool.

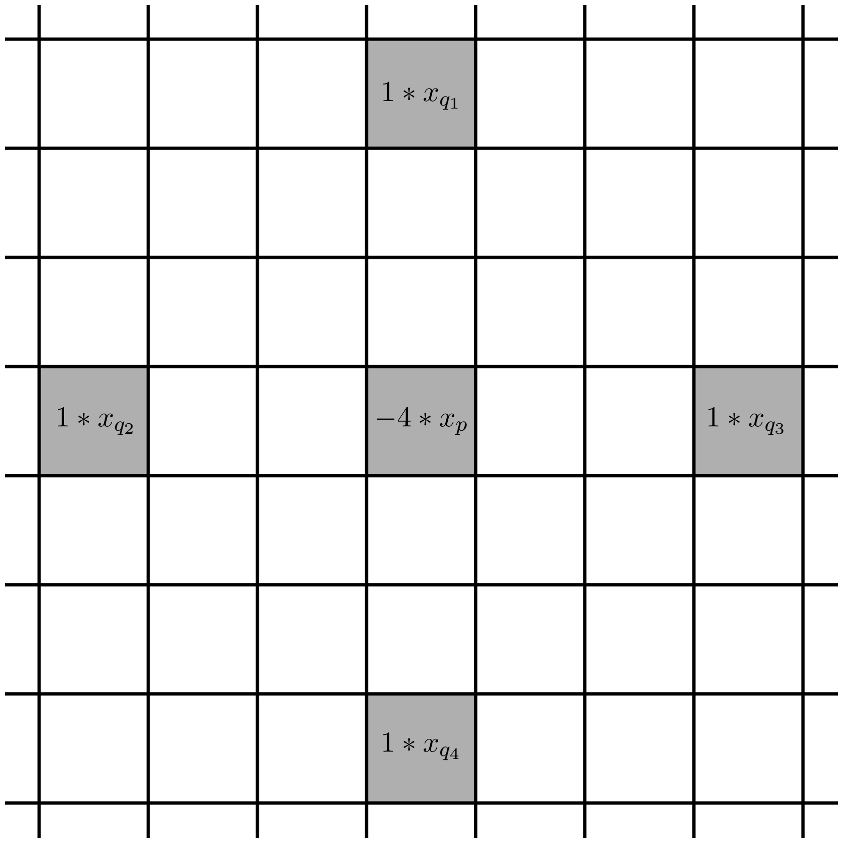
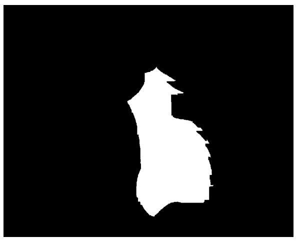
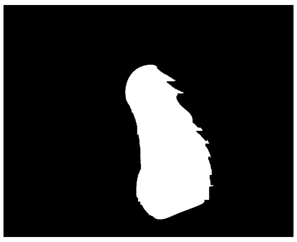


Figure 4.12: The image is an example of a kernel used during the variable kernel iteration method for pixel x_p. The offset has been set to 3 by multiplying the scale factor with the distance transform. It is important to notice that the pixels between the current pixel and the offset pixels are not used.

The optional user selection is the first tool applied because it is considered the ground truth solution. It indicates the type of border and is applied to both overlapping and overlapped segments. The mechanics for the border type of the overlapping and overlapped segments differ from this point out.

For the overlapped segment, we use the newly computed segment as a mask to set its border against the initial segments. Then, we refer to the color map containing the segmentation results. If a pixel on the mask boundary belongs to a different segment in the color map than the mask, we check the type of arrow leading from the current segment to the other segment. If the arrow is green, we set the border in that pixel to apply merging in Monster Mash. The second arrow indicates separation. In cases where the arrow is absent but the depth levels indicate occlusion, we applied a heuristic to separate the overlapped segment from the closer segments, which was done if the difference in depth levels was greater than 1. Figure 4.13 illustrates how the body of the wolf was refined.


(a) : Body segment created by LazyBrush


(b) : Body segment with its occluded parts estimated

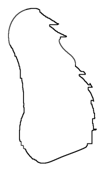

(c) : New border of the body segment

Figure 4.13: Figure **(a)** shows the segmentation result of the body that requires refinement. Figure **(b)** displays the approximated body segment with its overlapped parts. Figure **(c)** contains the border of the approximated segment. The occluded areas were determined based on the arrows as no user selection was present.

The borders of the overlapping segments are established to merge when there is an open contour connecting this segment with a segment of lower depth level. If the neighbor is the background or when the intensity in the original image is low, i.e. the outlines are present, the border is set to separate the segments.

To obtain the shapes and borders of the 3D model, we created an accurate initial guess, found a distance transform, and used the variable kernel method to compute the Laplace of the image. Our tools helped us define the model's structure and depth layout, with refinement in the stitched areas resulting in a more appealing result.

The borders and segment masks are saved to a zip archive along with the colored image as a template. Additionally, two files are included: a configuration file for Monster Mash and a file with the topological order of all segments. It is important to include these files to avoid any undesirable situations or results when using the Monster Mash tool. The order of the segments is determined by their depth order. The segments that we have ignored will be missing in the list. The configuration file for the Monster Mash tool specifies the tool's preferences upon loading, including mode (drawing, inflation or animation), template image display, and other settings. These settings can always be modified within the Monster Mash tool.

The implementation section guided us through the process of creating a 3D model from a single picture. The final Figure 4.14 in this section provides a visual representations of our accomplishments. We began by applying scribbles intended for the LazyBrush algorithm and the segmentation process. We then used the arrows to sort the segments based on proximity. We applied selection areas to define the properties of the borders. The estimation process then filled in the missing parts using the variable kernel method. Subsequently, the borders were extracted from the new segment masks and the specified user data. The resulting project file in the form of a zip archive was uploaded to Monster Mash to create the desired 3D model. An overview of several 3D models can be found in the following chapter.

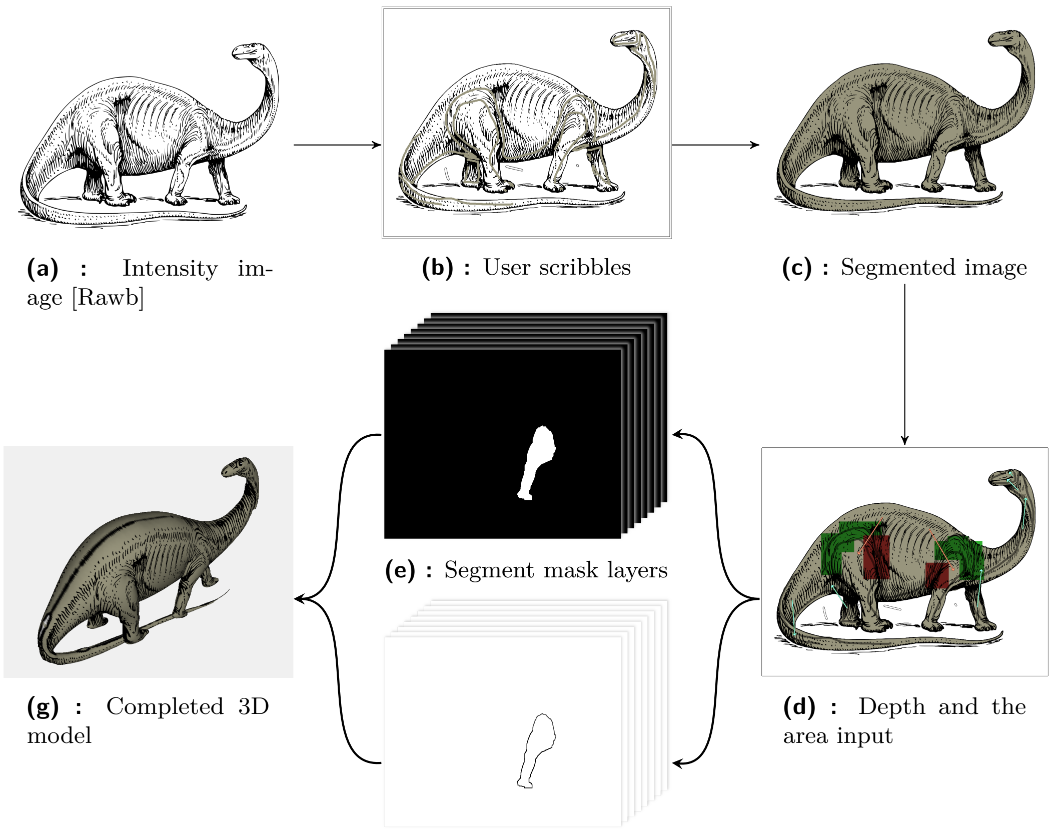

(a) : Intensity image [Rawb]
(b) : User scribbles
(c) : Segmented image
(e) : Segment mask layers
(g) : Completed 3D model
(d) : Depth and the area input
(f) : Segment border layers

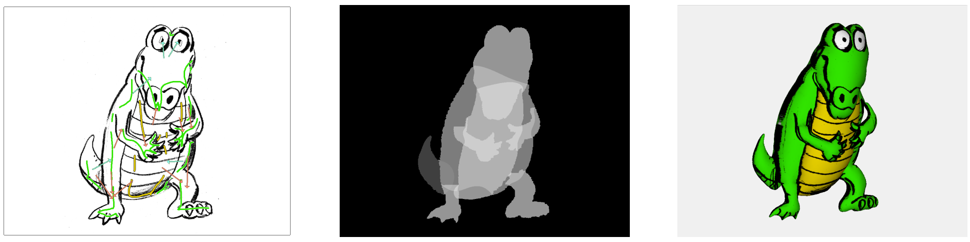
Figure 4.14: The chart displays the primary steps in the pipeline described in this thesis. The process begins with image **(a)** which serves as an intensity image with black represented as 0 and white as 1. The user then applies scribbles, as shown in image **(b)**, to separate the character into parts. The parts can have the same color, and in some cases, separate scribbles can belong to the same segment. The image segmentation, as demonstrated in image **(c)**, is followed by the user adding input in the form of arrows and the area tool, as displayed in image **(d)**. This enables the framework to estimate occluded segments, shown in image **(e)**, and create their borders in image **(f)**. These are then added together with the colored image and two predefined configuration files to a project file, which can be uploaded to the Monster Mash tool. Figure **(g)** shows the final 3D model that was generated.

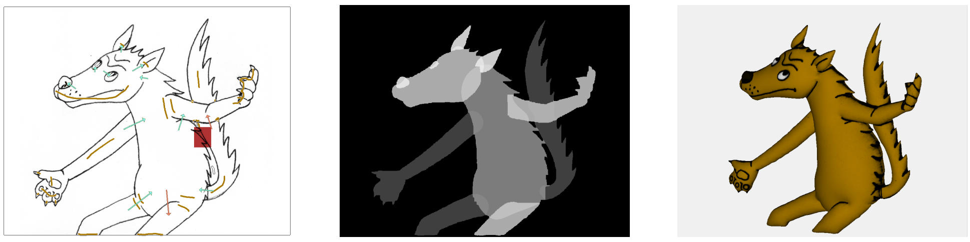
Chapter 5
Results

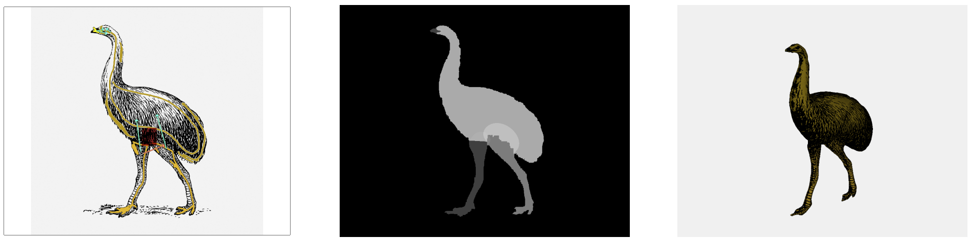
This framework allows users to process hand-made drawings and experiment with the 3D models that are generated from them using Monster Mash. The tool allows us to segment the image, including the texture, sort the segments created and specify the the of border if necessary. These steps require minimal precision and allow for additional inputs that enhance the results. The objective is to create a smooth boundary in overlapping areas, allowing them to merge in the final 3D model. This was achieved by diffusing the boundary conditions in the specified image parts of the image.

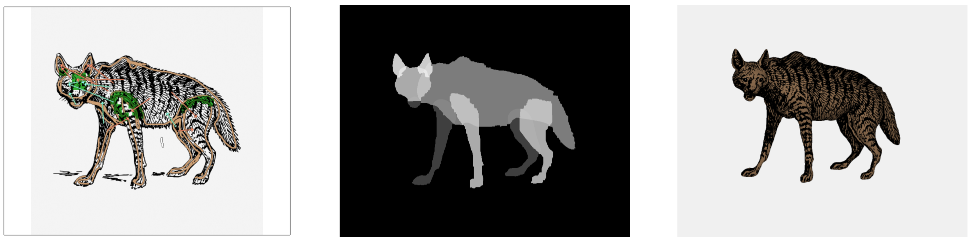
The tool described in this work was tested on several test images. Figures 5.1 and 5.1 display a selection of images that led to successful results. The first column of the figures shows the intermediate steps consisting of the provided user data, while the second column represents the resulting generated segments in the overlay. The depth order of the segments depends on the color intensity, with lighter colors indicating closer proximity to the user. The third column shows the 3D models created using the Monster Mash tool. The types of overlapping boundaries vary depending on the arrows and user specifications to accurately merge the segments in the resulting mesh.

Section 4.4.3 mentions the speedup of the variable kernel method compared to the Gaussian-Seidel iteration. Table 5.1 demonstrates the difference between the two methods when calculating the parts of the Dino Figure 5.1a. Both algorithms ran until the iteration produced a difference of less than 0.00001 for each pixel. The variable kernel method brought significant speedup and reduced the number of iteration cycles by at least one order of magnitude. The computation time was reduced from units of seconds to hundreds of milliseconds, resulting in almost immediate results. The algorithms were executed on a computer with Intel Core i7-10750H, 2.6 GHz, 16GB RAM, and the Windows 10 operating system.


(a) : Dino results


(b) : Wolf results


(c) : Big bird [Rawa] results


(d) : Hyena [Rawf] results

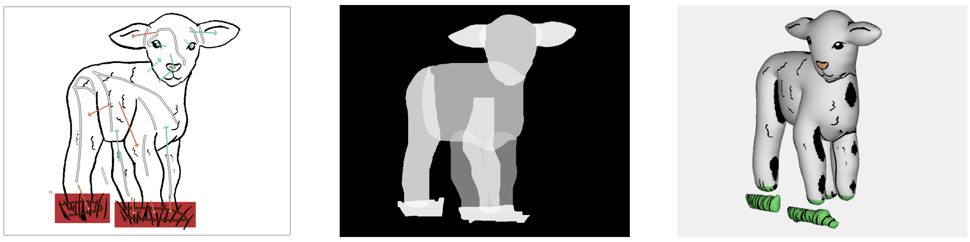
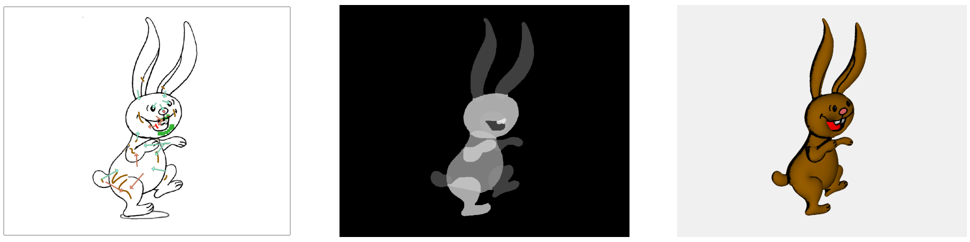
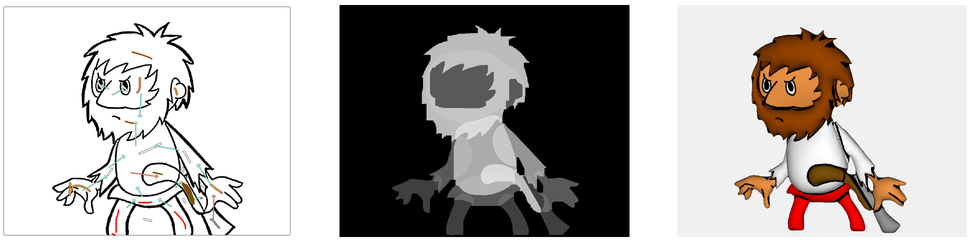
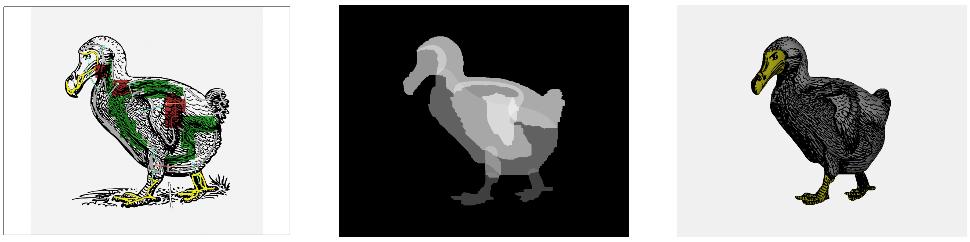
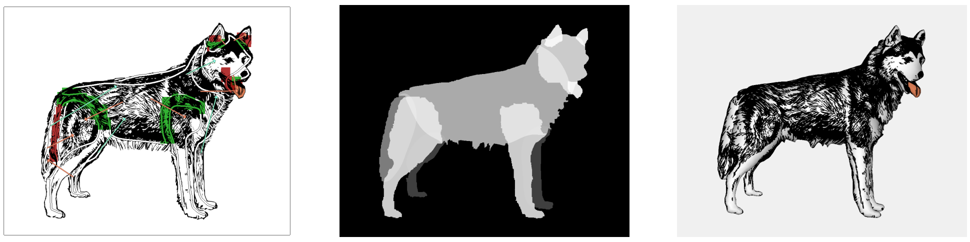

(e) : Sheep results

Figure 5.1: These are the testing images for this tool. The left column displays the input images with the user input applied. The middle column shows the created segments, with lighter colors indicating closer proximity to the user. The right column displays 3D models created in Monster Mash. (Wolf, Dino sources: ©Anifilm. All rights reserved. The sheep image is provided by the supervisor)


(a) : Rabbit results


(b) : Robber results


(c) : Dodo [Rawc] results


(d) : Dog [Rawd] results

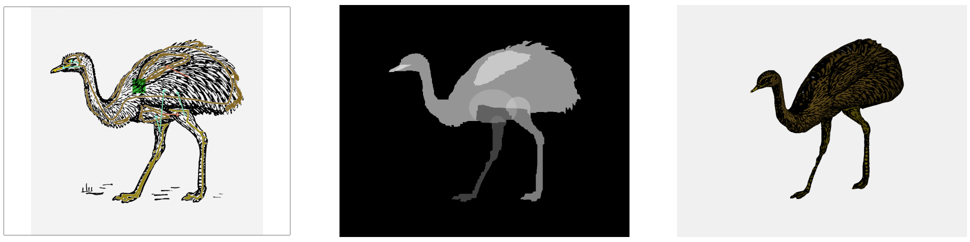

(e) : Emu [Rawe] results

Figure 5.2: The figure contains the second set of test images for the tool. The left column shows the input images with user input applied, including colored areas by the segmentation algorithm. The middle column displays the created segments, with lighter the color indicating closer proximity to the user. The right column displays 3D models created in Monster Mash. (sources: robber ©UPP. rabbit ©Anifilm. All rights reserved.)

	Variable kernel		Gauss-Seidel	
Segment	Iterations	Time [ms]	Iterations	Time [ms]
tail	468	582	8 663	10 180
main body	693	705	19 920	18 294
belly	541	354	10 589	6 445
head	348	37	1 871	187

Table 5.1: The table contains the variable kernel method and the Gauss-Seidel method in terms of speed. The number of iteration cycles and required time are in the first and second column, respectively.

5.1 Comparison

Monster Mash
Compared to the segmentation process in Monster Mash, which is based on tracing the template drawing, the LazyBrush algorithm described in Section 4.2 requires only rough scribbles. As a result, in most cases, users do not have to concentrate as much on placing scribbles in the segmented area. If placement is inefficient, an additional scribble can be added, reducing the need for repeated redrawing. If the user needs to dispose of drawn scribbles, repositioning scribbles should be faster and more flexible than manual tracing.

For a closer comparison, we can evaluate the performance of the implemented tool and the time required to trace the image in Monster Mash. Table 5.2 shows a comparison of the approaches including the time required to create 3D models, potentially redraw, and importing into Monster Mash.

character	Monster Mash	This tool
brontosaurus	$4'05''$	$3'20''$
wolf	$4'15''$	$2'15''$
big bird	$1'45''$	$1'33''$
sheep	$1'58''$	$1'40''$
hyena	$2'16''$	$9'03''$

Table 5.2: A comparison of the performance of this tool and Monster Mash was conducted. The time required for creating the 3D model is faster or similar to Monster Mash in images with larger homogeneous areas and minimal segment overlap in one place. However, in scenarios where there are too many merging borders at once, a more detailed and precise setup is required, resulting in higher time consumption, as demonstrated in the time comparison for the hyena image. In the ear area, there are overlaps between the ear and the body segments that need to be merged with the head segment. It is also important to consider the head border behind the ear, which may be affected by the selection tool. Properly setting took more than nine minutes to solve in this case.

The depth order in this tool is determined solely by the orientation of the arrows. A comparison between the depth ordering in this tool and that implemented in Monster Mash reveals a notable difference in approach. In Monster Mash, the depth order is based on the order and placement of the drawn segments. The order of segments can be determined through interaction with them. The gray lines indicate the borders of the occluded segments, as shown in Figure 5.3a. However, when the segments are not selected, the borders may not be visible, as in the case of the head in Figure 5.3. In contrast, the Monster Mash tool displays the entire contour of the selected segments. In this framework, the arrows in this approach indicate the depth order directly. Since this work does not permit additional interaction with the estimated segments, the estimation results are not visible in the tool window. However, the segments created using the LazyBrush algorithm are visible only due to the color difference, image outlines, or in detail upon saving the current work. The segments are saved in separate images, which could be misleading or difficult to process in more detailed work.

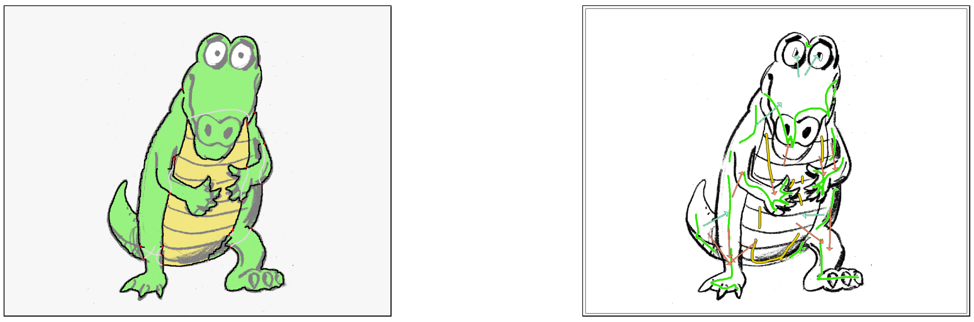
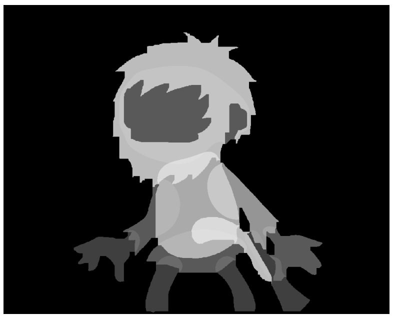

(a) : Monster Mash visuals **(b) :** Visuals of this work

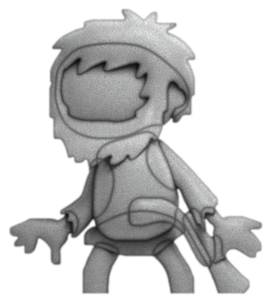
Figure 5.3: The figure illustrates the depth order of both Monster Mash and the current framework for comparison. (Dino source: ©Anifilm. All rights reserved.)

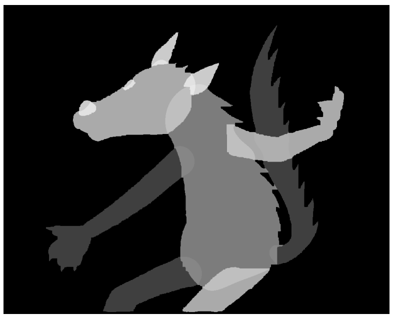
Ink-and-Ray

One of the articles that will be compared with this work is Ink-and-Ray [SKv+14]. This work is essentially the implementation of the Ink-and-Ray pipeline [SKv+14] adapted for the Monster Mash tool. Therefore, most of the results should be very similar to that tool. Pipeline [SKv+14] produces images that are converted to the 2.5D domain. We will not compare the 3D model results because Monster Mash works differently and was not part of the implementation of this work. Figure 5.4 shows a comparison of the generated layers using both frameworks. The main difference can be seen in the Rumcajs figure **??**, where the body and head segments overlap in this implementation, while they only touch in the other implementation. Additionally, there is a difference at the ear, where the head segment turns sharper than in the Ink-and-Ray implementation. There may be other differences, but some of these may be due to the result of segmentation.

The results for the wolf example are very similar. The most obvious similarities can be observed on the back of the wolf and in the area where the body segment overlaps with the closer leg. The ears are segmented separately, consistent with the choice of segmentation. The most noticeable difference is the absence of the left eye in this implementation, which remains in the Ink-and-Ray pipeline [SKv+14]. The eye is absent here because its segment shares no visible boundaries with any other segment. This configuration was chosen because Monster Mash would create separate a 3D mesh for it.


(a) : Rumcajs reconstruction in this work


(b) : Rumcajs reconstruction in Ink-and-Ray [SKv+14]


(c) : Wolf reconstruction in this work

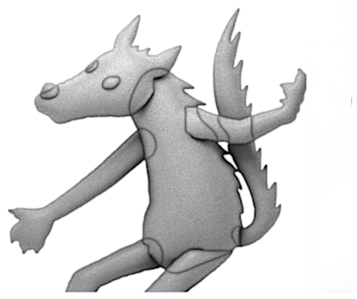
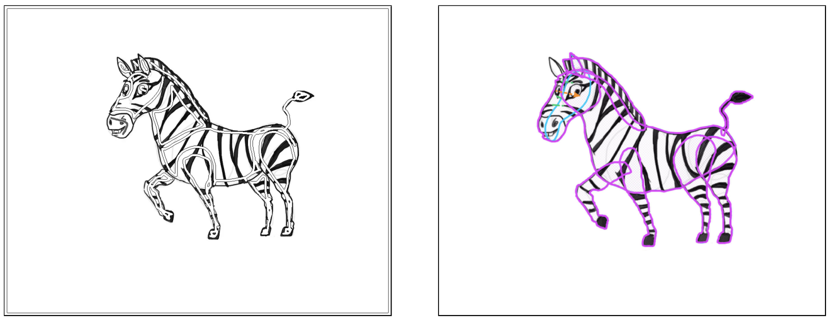

(d) : Wolf reconstruction in Ink-and-Ray [SKv+14]

Figure 5.4: The figure presents results of this work in the first column and those from the Ink-and-Ray pipeline [SKv+14] in the second column.

CreatureShop

CreatureShop will be used as another point of comparison. Both tools share the segmentation process, but their workflows for image processing differ significantly. The segmentation process in CreatureShop is manual and requires the user to trace the hidden parts of the segment. Repeating failed attempts can lengthen the segmentation process unless a tool is available to erase incorrectly drawn lines. If the user is experienced, the tracing can be more accurate and meet the user's requirements. This can reduce the time required compared to using this pipeline. In our work, we compute the Laplace of the image and add the estimated part to the segment already created by LazyBrush. The estimated part is based on the depth order of the segments. The segmentation process allows us to place the scribbles anywhere in the region of the future segment. It is assumed that the user may not be able to draw precisely and may be limited by their equipment. However, even if the user's input is not precise enough, this tool can provide accurate results. Additional scribbles can be added to problem areas to refine the shape of the segment. It is important to note that this tool converts all loaded colored images to grayscale because the segmentation tool relies on pixel intensity. On the other hand, CreatureShop can work directly with color images.



(a) : Segmentation input of the zebra image [Med]

(b) : User input for the zebra image by CreatureShop [ZYC+23]

Figure 5.5: The image of zebra segmented using both the CreatureShop and this tool. In the CreatureShop, the user outlines the boundary of each segment using the purple brush. In this work we added scribbles for the LazyBrush algorithm. Due to the stripes, more scribbles are required in this tool. For more information, see the segmentation limits in Section 5.2.

5.2 Limitations

The tool described in this work has several limitations. The first issue that may arise is the presence of dark areas in the images that do not depict the outlines. These regions can lead to improper segmentation results due to the inability of LazyBrush to create the correct segmentation regions within them. Figure 5.6 illustrates a case where images require more detailed processing. Zebra stripes can cause distortions during segmentation. To address this problem, additional scribbles need to be drawn over the dark areas and near the segment outlines. While precision may be more necessary in cases with such details, it is not always necessary to be completely precise.

Another problem concerns the diffusion process. In some cases, it may be necessary to estimate part of a relatively narrow segment. However, if a significant part of the segment is covered, the resulting estimate might become fragmented. In these cases, the boundary conditions prepared for the diffusion process are set so that the covered segments do not meet. It is important to note that this limitation is not exclusive to narrow segments. In some cases, the arrangement in the drawing may prevent the estimated segments from joining in the 3D model as desired. For example, when examining the elephant in Figure 5.7, only the two left limbs require estimation. While the

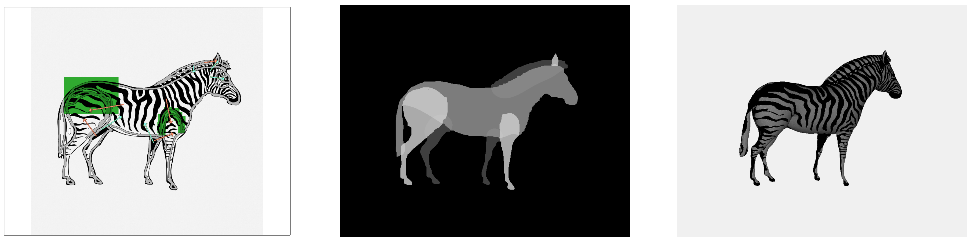

(a) : User input **(b) :** Estimated segments **(c) :** 3D result

Figure 5.6: The image of the zebra [Rawg] demonstrates the challenge of managing dark pixel areas. The creation of the model is still manageable.

front leg is adjacent to the main body, the hind leg does not share a common boundary with the elephant's trunk. Thus, the leg is drawn too far from the main body, preventing it from connecting to the main mesh in the Monster Mash tool. Instead, a separate mesh is created, as shown in Figure 5.7c. Unfortunately, this tool cannot solve this problem directly. Therefore, the segment must be redrawn in a painting tool to connect it to the main body segment, including the segment layer as well as its border.

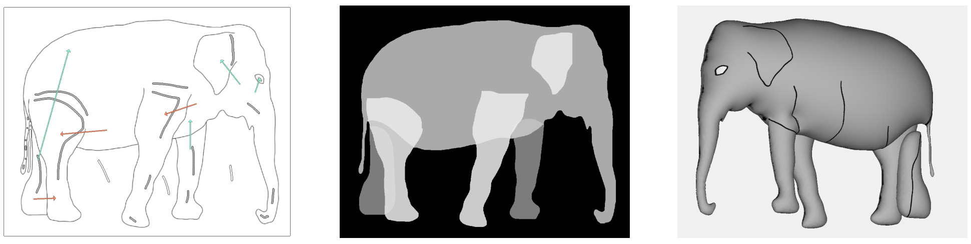

(a) : The user input **(b) :** Estimated segments **(c) :** 3D result

Figure 5.7: The elephant in image **a** is prepared for estimation of all hidden parts. As shown in image **b**, the hidden leg part is estimated in a way that it does not meet the body segment, preventing it from merging with the mesh of the 3D model displayed in image **c**. (The elephant image was provided by the thesis supervisor.)

The third limitation is related to the interpretation of the generated data by the Monster Mash tool [DS21]. As already mentioned, there are two types of borders: one allows merging, while the other directly outlines the segments. The issue arises when two or more merging boundaries intersect in an uploaded Monster Mash project. In such situations, Monster Mash cannot create the desired 3D model. The only solution to this problem is to detect problematic boundaries, preferably by removing segments in Monster Mash. The segments can then be adjusted by selecting tool for selecting areas in this work or by rearranging the segments in a different way. No other solution to this problem has yet been found.

The fourth issue is related to segment boundaries. It occurs when segment boundaries rapidly change types rapidly at small intervals. In this setting, Monster Mash is unable to produce a 3D model. This situation occurs when shading, noise, or other dark pixels cross the top segment boundary. The only solution in this framework is to use a selection tool that assigns type directly to the problematic boundary segments. This scenario is solved for both the hyena in Figure 5.1d and the brontosaurus in Figure 4.14. It is important to note that the selection areas tool modifies all borders it covers.

In this work, we should consider creating more detailed models with more overlapping segments. When merging segments, we must be careful not to merge unwanted segments. For instance, we can refer to the Dino Figure 5.1a. Although in our case the claws are part of the hand, if we wanted to create a more detailed model and separate them from the hand, the Monster Mash tool would merge the claw mesh not only with the hand, but also with the abdomen below it. This problem is mentioned in this section because it can occur frequently.

The limitations outlined in this chapter suggest opportunities for further improvements to the framework. In the next chapter, improvements and extensions will be proposed to address all the relevant issues identified in this chapter.

Chapter 6

Conclusion

The aim of this project was to develop an advanced user interface for Monster Mash. The drawn image serves as a basis for obtaining the data needed to create the 3D model. Although we did not integrate this framework into Monster Mash due to time constraints, we created a separate program to generate project files that can be uploaded to the Monster Mash tool [DS21].

The process of creating data for Monster Mash involved three steps. First, the image was segmented. Second, the depth order between all segments was established. Finally, the occluded parts were estimated to determine the final shape of the model parts. The depth order and selection tool implemented in this work determined which parts of the segments would be merged with other segments in the final model.

6.1 Future work

This tool was designed as an extension of the Monster Mash tool. Integrating this framework into Monster Mash would greatly speed up the creation of a model from a single image.

Since this work has a simple user interface, it could be improved by introducing a more visually pleasing interface that is not based on shortcuts only. The new interface could include a status window and a panel with elements for manipulating color selection, brush size and other parameters. These improvements would make it more intuitive and attractive for users, and could even allow the original image to be used as a template texture.

The comparison of the acceleration in Table 5.1 reveals that the head segment has also been calculated. However, the head segment has only two possible neighbors for shape estimation, namely the eyes. However, these segments are completely surrounded by the head segment. It is suggested that regions where such segments exist should be automatically assigned to a lower segment (the head in the case of the dino), thus reducing the required time needed for estimation. In addition, segments that are not part of a contiguous group of segments do not contribute to the estimation and should also be removed from computation.

Although it is not possible to solve all the limitations described in Section 5.2 using this tool alone, it is possible to improve the framework. Specifically, the selection areas currently overwrite the entire area below them, which should be changed so that only problematic boundaries are selected instead. This issue is related to the segment visualization problem. Currently, image segments can only be perceived after the work is saved and completed segments can only be accessed from the project file generated for Monster Mash. Therefore, the tool should be improved to allow visualization of both types of segments including segmentation and diffusion results.

The fundamental problem that needs to be addressed is the constraint based on segment shapes. Before the diffusion process of each segment, users should be able to adjust the threshold of the Laplacian result by setting its parameter. Another way to address issues during diffusion is to allow users to draw additional conditions in the computed area to help indicate the shape of the occluded segments. These user-drawn scribbles should belong only to the currently processed segment or the upper segments. The proposed method is designed to connect thin parts of the segments and enlarge the occluded parts which would otherwise be separated from the rest of the model.

6.2 Final words

While writing this thesis and working on the tool described here, I gained a deeper understanding of 2D and 3D graphics. The framework discussed in this text provides insight into the possibilities of sketch-based modeling, including its optimized speed and possible applications in graphics programs. The methods explained in this thesis can be used to add depth to illustrations using the Ink-and-Ray pipeline [SKv+14], create prototypes and create casual 3D models using the Monster Mash tool [DSC+20], and even create precise 3D models suitable for 3D printing using CreatureShop [ZYC+23]. There may also be previously undiscovered methods that could make sketch-based modeling more efficient and accurate.

Although the topic of sketch-based modeling methods is not widely known, at least to my knowledge, these methods can be very useful for beginners in creating 3D models and understanding the basics of 3D modeling. This is especially true when implementing and directly using these methods. The discussion deals with segmentation algorithms, flow algorithms, a topological sorting method, and diffusion methods. These topics were relatively new to me and provided and valuable experience for my future work.

Appendix A

Bibliography

[Aut23a] Autodesk, Autodesk 3ds max, 2023. Available at https://www.autodesk.com/products/3ds-max.

[Aut23b] Autodesk, Autodesk maya, 2023. Available at https://www.autodesk.com/products/maya.

[BK04] Y. Boykov and V. Kolmogorov, An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **26** no. 9 (2004), 1124–1137.

[Com23] B. O. Community, Blender, 2023. Available at https://www.blender.org.

[DSC+20] M. Dvorožňák, D. Sýkora, C. Curtis, B. Curless, O. Sorkine-Hornung, and D. Salesin, Monster Mash: A single-view approach to casual 3D modeling and animation, *ACM Transactions on Graphics* **39** no. 6 (2020), 214.

[DS21] M. Dvorožňák and D. Sýkora, Monster mash demo, June 2021. Available at https://github.com/google/monster-mash.

[FH12] P. F. Felzenszwalb and D. P. Huttenlocher, Distance transforms of sampled functions, *Theory of Computing* **8** no. 19 (2012), 415–428.

[GPR98] D. Geiger, H. Pao, and N. Rubin, Salient and multiple illusory surfaces, in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1998, pp. 118–124.

[GB+21] G. Guennebaud, J. Benoît, and others, Eigen, 2021. Available at https://eigen.tuxfamily.org.

[Inc23] A. Inc., Adobe photoshop, 2023. Available at https://www.adobe.com/cz/products/photoshop.html.

[JCW09] S. Jeschke, D. Cline, and P. Wonka, A GPU laplacian solver for diffusion curves and Poisson image editing, *ACM Transactions on Graphics* **28** no. 5 (2009), 116.

[Kah62] A. B. Kahn, Topological sorting of large networks, *Communications of the ACM* **5** no. 11 (1962), 558—562.

[OSSJ09] L. Olsen, F. F. Samavati, M. C. Sousa, and J. A. Jorge, Sketch-based modeling: A survey, *Computers & Graphics* **33** no. 1 (2009), 85–103.

[Sti09] T. Stitch, Graph cuts with cuda, in *GPU Technology Conference*, Nvidia Corporation, 2009. Available at `https://www.nvidia.com/content/GTC/documents/1060_GTC09.pdf`.

[SDC09] D. Sýkora, J. Dingliana, and S. Collins, LazyBrush: Flexible painting tool for hand-drawn cartoons, *Computer Graphics Forum* **28** no. 2 (2009), 599–608.

[SKv+14] D. Sýkora, L. Kavan, M. Čadík, O. Jamriška, A. Jacobson, B. Whited, M. Simmons, and O. Sorkine-Hornung, Ink-and-Ray: Bas-relief meshes for adding global illumination effects to hand-drawn characters, *ACM Transactions on Graphics* **33** no. 2 (2014), 16.

[SSJ+10] D. Sýkora, D. Sedláček, S. Jinchao, J. Dingliana, and S. Collins, Adding depth to cartoons using sparse depth (in)equalities, *Computer Graphics Forum* **29** no. 2 (2010), 615–623.

[SJ] D. Sýkora and O. Jamriška, Gridcut. Available at `https://gridcut.com/`.

[dt23] T. G. development team, Gnu image manipulation program, 2023. Available at `https://www.gimp.org/`.

[ZYC+23] C. Zhang, L. Yang, N. Chen, N. Vining, A. Sheffer, F. M. Lau, G. Wang, and W. Wang, CreatureShop: Interactive 3D character modeling and texturing from a single color drawing, *IEEE Transactions on Visualization and Computer Graphics* **29** no. 12 (2023), 4874–4890.

Appendix B

Image credits

[srcMed] K. Medford, cartoon zebra, Modified version of the image provided by wikiHow; License: CC BY-NC-SA 3.0 DEED. Available at https://www.wikihow.com/Draw-a-Zebra.

[srcRawa] Rawpixel, big bird, Image source: Public domain. Available at https://www.rawpixel.com/image/6289252/psd-sticker-vintage-public-domain.

[srcRawb] Rawpixel, brontosurus, Image source: Public domain. Available at https://www.rawpixel.com/image/6327123/png-sticker-public-domain.

[srcRawc] Rawpixel, dodo, Image source: Public domain. Available at https://www.rawpixel.com/image/6327530/psd-sticker-public-domain-vintage-illustrations.

[srcRawd] Rawpixel, dog, Image source: Public domain. Available at https://www.rawpixel.com/image/6261350/png-vintage-public-domain.

[srcRawe] Rawpixel, emu, Image source: Public domain. Available at https://www.rawpixel.com/image/6256901/psd-sticker-vintage-public-domain.

[srcRawf] Rawpixel, hyena, Image source: Public domain. Available at https://www.rawpixel.com/image/6536525/image-vintage-public-domain-black.

[srcRawg] Rawpixel, zebra, Image source: Public domain. Available at https://www.rawpixel.com/image/6305642/psd-sticker-vintage-public-domain.