A FRAMEWORK FOR SURFACE RECONSTRUCTION OF SPARSELY-SAMPLED OBJECTS

RESEARCH THESIS

Submitted in partial fulfillment of the requirements
for the degree of
Master of Science in Computer Sciences

Alik Zamansky

Submitted to the Senate of
the Technion- Israel Institute of Technology

Tammuz, 5767       Haifa
July 2007
Acknowledgement

THIS RESEARCH THESIS WAS SUPERVISED BY PROF. GILL BAREQUET IN THE
COMPUTER SCIENCE DEPARTMENT

I wish to express my gratitude to my advisor Prof. Gill Barequet for supervising me
as an external student. His guidance and help, especially during unconventional hours, is
highly appreciated. I would also like to thank my parents Rita and Sergey, and my wife
Anna for their love, understanding and encouragement.

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY
ACKNOWLEDGED
## Contents

Acknowledgement ........................................ iii

Abstract ................................................. 1

1 Introduction ........................................... 3
   1.1 Background ........................................... 3
   1.2 Previous Work ........................................ 4
   1.3 Organization of the Dissertation ....................... 5

2 Overview ............................................... 7
   2.1 Problem Definition .................................... 7
   2.2 Motivation ............................................ 8
   2.3 The Proposed Framework ............................... 11
   2.4 Our Case Study ....................................... 15
   2.5 Notation ............................................. 16

3 Contour Correspondence .................................... 19
   3.1 Function Definition and Properties .................... 19
   3.2 Contour Stitching .................................... 20
   3.3 Discussion ........................................... 24

4 Space Partitioning ....................................... 27
   4.1 General Considerations ............................... 27
   4.2 Voronoi Diagram of Polygonal Sites .................. 29
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Vertical Cells</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Extension to Nonvertical Cells</td>
<td>43</td>
</tr>
<tr>
<td>4.5</td>
<td>Shortcomings and Solutions</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>Surface Interpolation</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Translation and Tiling</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>Legal and Illegal Interpolations</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>Dependency Graphs and the Iterative Algorithm</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>The General Three-Dimensional Case</td>
<td>59</td>
</tr>
<tr>
<td>6</td>
<td>Experimental Results</td>
<td>61</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>73</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Correspondence in a severely undersampled input (1) . . . . . . . . . . . . 9
2.2 Correspondence in a severely undersampled input (2) . . . . . . . . . . . . 10
2.3 A simple pipe reconstruction (schematic) . . . . . . . . . . . . . . . . . . 10
2.4 Data flow of the proposed framework . . . . . . . . . . . . . . . . . . . . 12
2.5 A schematic example of using the framework . . . . . . . . . . . . . . . . . 14
2.6 Reconstruction of a synthetic example . . . . . . . . . . . . . . . . . . . . 17
3.1 An example of contour stitching . . . . . . . . . . . . . . . . . . . . . . . 23
3.2 Choosing the shortest vector . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3 An improved contour stitching . . . . . . . . . . . . . . . . . . . . . . . . 25
4.1 Failure to construct a cell for matching contours . . . . . . . . . . . . . . . 28
4.2 Example of a Voronoi diagram of polygonal sites . . . . . . . . . . . . . . 30
4.3 The proof there is always a segment in $V(P)$ between $p \in V(P)$ and $P$ . . 31
4.4 An example of “vertical” cells . . . . . . . . . . . . . . . . . . . . . . . . 32
4.5 Partitioning steps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.6 A complex example for Step 4 . . . . . . . . . . . . . . . . . . . . . . . . 37
4.7 Splitting a cell in $T_r$ by a VDP – example 1 . . . . . . . . . . . . . . . . 38
4.8 Splitting a cell in $T_r$ by a VDP – example 2 . . . . . . . . . . . . . . . . 38
4.9 An example of processing nested contours . . . . . . . . . . . . . . . . . . 42
4.10 An example of “nonvertical” cells . . . . . . . . . . . . . . . . . . . . . . 44
4.11 Shortcomings of the partitioning from Section 4.3 . . . . . . . . . . . . . . 45
4.12 An example of a “tilted branching” case . . . . . . . . . . . . . . . . . . . 46
4.13 Examples of “asymmetric branchings” . . . . . . . . . . . . . . . . . . . . 47
List of Tables

6.1 Testing results and measurements .......................... 70
Abstract

The problem of reconstructing a three-dimensional object from a set of planar parallel slices has been the subject of study for many years. In this problem, each slice consists of simple closed polygonal contours, and contours from adjacent slices should be connected by a surface to reconstruct the original spatial object (so that it can later be visualized and investigated). Several algorithms have been proposed for the reconstruction problem when the distance between every two successive slices is small. However, when the original object is not sampled densely enough, the problem becomes more complex, and the existing algorithms often fail to produce satisfactory results.

In this thesis, a framework is proposed for solving the surface-reconstruction problem for sparsely-sampled objects. The framework consists of several phases which solve different tasks of the reconstruction: obtaining correspondence between contours, resolving branching situations, and producing the actual surface. Different algorithms can be integrated in the framework to construct a correct (non-self-intersecting) surface. Instances of such algorithms are provided as a case study for the framework, and are implemented for testing and evaluation of the proposed method. Experimental results show improvement in quality of the reconstructed surface for sparse inputs, as demonstrated by a few synthetic and real examples.
Chapter 1

Introduction

1.1 Background

The problem of reconstructing and visualizing three-dimensional objects from cross-sectional images, arising primarily in the field of medical imaging (e.g., MRI, CT, and ultrasound), has been studied intensively during the last three decades. Some of the proposed methods produce three-dimensional volumes consisting of voxels. The resulting objects are visualized either by using volume-rendering techniques or by displaying a polygonal surface extracted from the volume data (e.g., by the marching cubes algorithm [17]). Among these methods are shape-based interpolation (see, e.g., [14, 16]), implicit surfaces (e.g., [22, 23]), as well as many others. For more methods and more comprehensive descriptions of various approaches, see [10, 18, 20].

Other techniques, including the one used in this thesis, construct directly a polygonal surface, which connects contours from adjacent cross-sections (slices). The advantage of these surface-interpolation methods is that they build a closed volume bounded by a well-defined surface consisting of a relatively small number of polygons (triangles). The complexity of the reconstructed mesh is comparable to that of the input contours. Such methods usually work faster and consume less memory. However, unlike in some other techniques (e.g., the implicit surfaces), special care must often be taken to avoid self-intersections of the resulting surface. The reason we concentrate on surface-interpolation techniques for
dealing with sparsely-sampled objects is that they seem the most appropriate for the separation we make between the contour correspondence and the actual surface construction.

For surface-interpolation methods, it is usually assumed that a preprocessing step has already constructed closed, possibly nested, but not intersecting contours (polygons) from the raw pixel data in each slice, by applying some image-processing techniques (filtering, edge detection, etc.). The goal is to reconstruct the original object by connecting contours in adjacent slices by a surface.

1.2 Previous Work

Various algorithms for polyhedral surface reconstruction from parallel slices have been suggested. The first known solution to this problem was proposed by Keppel [15]. His work, as well as a few following works dealt with the simple case where each slice contains only one contour and implicitly assumed a high degree of resemblance between the contours. Boissonnat [7] made the first attempt to deal with the contour interpolation problem in its full generality, i.e., making no assumptions about the number, geometry, or nesting hierarchies of contours. His algorithm was based on the Delaunay triangulation of the input slices and worked well except for a few typical cases. Another algorithm, which usually constructs correct three-dimensional surfaces was proposed by Barequet and Sharir [5]; it is based on an analysis of the resemblance between pairs of successive slices.

A few other algorithms that solve the surface-reconstruction problem in its full generality have been proposed. Bajaj et al. [2] proposed a solution quite similar to [5] but much more formal. Barequet et al. [3] proposed an algorithm based on the straight-skeleton technique. Their algorithm analyzes overlapping areas of contours, constructs the straight skeleton of the projection of contour portions that are to be tiled together, and then uses the straight skeleton for tiling and lifting the triangulation back into three dimensions. As a result, the achieved tiling is usually smoother than the one in [2] and [5].

Meyers et al. [19] divided the surface-reconstruction problem into subproblems. The first is the correspondence problem, which involves finding the correct connections between contours from adjacent slices. The second is the tiling problem, which deals with actually connecting two contours on adjacent slices. The third is the branching problem,
which occurs when a contour in one slice corresponds to several contours on the other slice, which significantly complicates the task of tiling. Meyers et al. proposed some partial solutions for correspondence and branching problems without providing a general surface reconstruction algorithm. Haig et al. [12] defined rules for solving the correspondence problem based on contour nesting topologies. However, the reconstruction algorithm they provided is not general enough and cannot handle complicated inputs.

Choi and Park [9] were the first to try solving the correspondence problem separately in a general surface-reconstruction algorithm. Their algorithm decides which pairs of input contours should be connected, leaving the actual connecting to a later stage. They point out the advantage of this attitude: different types of input data may require different heuristics for resolving correspondence, while the tiling and branching problems can be solved in the same manner.

However, their solution of correspondence is based on the overlap of contours’ xy-projections (which is, implicitly, the case for all the general algorithms we have mentioned). While contour-overlap-based approaches are reasonable for densely-sampled objects (when the distance between successive slices is small), they are insufficient when the sampled data are sparse, and more sophisticated solutions for correspondence are required.

1.3 Organization of the Dissertation

Chapter 2 formulates the problem and provides the overview of our method. It introduces the proposed framework and explains our case study. Chapters 3–5 describe the three main components of the framework. In Chapter 3 our contour-matching algorithm is presented. Chapter 4 explains our partition procedure. Some properties of the partition are also discussed and proved in that chapter, and it is explained how the partition shown in our case study can be extended to a more general case. Chapter 5 explains our solution for contour translation and tiling. Chapter 6 contains some experimental results, and Chapter 7 provides some concluding remarks.
Chapter 2

Overview

2.1 Problem Definition

In this thesis we deal with the following problem. We are given a set of consecutive planar parallel cross-sections of an unknown spatial object. Each cross-section consists of multiple non-intersecting, possibly nested contours, each contour represented by a simple polygon. The goal is to reconstruct the boundary of the object, which has to be a non-self-intersecting three-dimensional polygonal surface. The resulting surface has to be visually pleasing and resembling the sampled object as much as possible.

The reconstruction problem, as stated above, is underconstrained by definition, and as such, does not have only one “correct” solution. Instead, many solutions of different quality are possible, depending on the input and on the algorithm being used. In this research we offer a solution for the surface reconstruction problem under the following assumptions:

1. The input data are sparse. A few algorithms, mentioned in the introduction, solve the reconstruction problem efficiently when the distance between a pair of adjacent cross-sections is small. When, for various reasons, this distance is not sufficiently small, surface reconstruction becomes a significantly more complicated task.

2. Despite the above, object topology changes are (usually) not missed by slicing. This assumption is important, because when it does not hold, an object usually cannot be
reconstructed correctly. For example, if the original object consists of two intersecting pipes (like in Figure 2.1(f)) and is sampled above and below the intersection, a topologically-correct reconstruction is not possible.

Note that as in the vast majority of papers on the subject, we do not attempt to estimate the quality of the reconstructed surface by some measure (in addition to comparing the results visually). The citation [4] is a rare exception of this.

### 2.2 Motivation

When input data are sparsely-sampled, we cannot rely on overlap-based correspondence, like in the algorithms of [2, 3, 5, 9], as well as in a few others, because corresponding contours may not overlap in the projection of one slice onto the other. Instead, a more sophisticated procedure needs to be used to determine correspondence between contours in adjacent slices. This procedure should not necessarily be universal, but rather be developed specifically for a given input domain. The second phase of the reconstruction algorithm should be able to construct a non-self-intersecting polygonal surface from the contour-correspondence information found in the first phase.

The ability to use any correspondence procedure can be especially useful when object topology is available. In such a case, the procedure may use this information to produce correct matchings between contours. Another interesting idea (when such information is not available) is to build a learning-based correspondence procedure, i.e., one that is able to learn the topology of objects in a domain from given examples.

The problem is that despite a theoretical distinction between the problem of correspondence and the problems of branching and tiling, in practice all today’s algorithms do not allow to separate between them. The general surface-reconstruction methods of [2, 3, 5] implicitly assume overlay-based correspondence, and cannot be expanded to work with more general contour-matching procedures.

Figure 2.1(a) shows a simple example of sparse input data, containing two slices, each having two circular contours. Figures 2.1(b–f) show (schematically) possible reconstructions, depending on contour matchings obtained by a correspondence procedure. Practically all reconstruction algorithms available today will choose the variant (c), which is
2.2. MOTIVATION

![Input contours](image1)

(a) Input contours

![No matchings](image2)

(b) No matchings

![A_1 - B_1, A_2 - B_2](image3)

(c) $A_1 - B_1, A_2 - B_2$

![A_1 - B_2](image4)

(d) $A_1 - B_2$

![A_1 - (B_1,B_2)](image5)

(e) $A_1 - (B_1,B_2)$

![A_1 - B_2, A_2 - B_1](image6)

(f) $A_1 - B_2, A_2 - B_1$

Figure 2.1: Correspondence in a severely undersampled input (1)

probably the most reasonable correspondence in this particular case if no additional information is provided. However, if additional information is available to a correspondence procedure (e.g., the entire slice data), some other choice may be more reasonable, for instance, (b).

Figure 2.2 presents another simple, yet interesting example. For the input contours shown in Figure 2.2(a), most of today’s reconstruction algorithms produce a surface as in Figure 2.2(b). However, in this example, contour matchings shown in Figures 2.2(c,d) are as reasonable as the overlay-based matchings of 2.2(b). Thus, when dealing with sparse input data in which overlay-based matchings are not necessarily the best choice, the better solution would be to use some “more informed” correspondence procedure to find the contour matchings, and then reconstruct the object according to these matchings.

Another problem with the general reconstruction algorithms that do not use any correspondence procedure (but instead are implicitly based on contour overlay for correspondence) is illustrated in Figure 2.3. The figure shows two overlapping contours, but their overlapping area is small. In this case the algorithms of [2, 3, 5] will connect the corresponding contours, but the resulting tiling will be quite unnatural, as shown in Figure 2.3(a) (a shape which looks almost like a double cone). The algorithm of Choi and Park [9] uses a correspondence procedure of its own, but since it is overlay-based and the overlapping area of the contours is small, their algorithm would decide not to connect the contours.
The goal of this research is to propose a solution for the problems of connecting corresponding contours and of achieving more natural tiling in a general surface reconstruction algorithm. We introduce a general framework for solving the reconstruction problem, in which any correspondence algorithm can be used. Different correspondence algorithms can be developed for different types of sparse input data, so that they can later be integrated into the proposed framework.

However, a practical separation between contour correspondence and the rest of the reconstruction is a very nontrivial task. Supporting any possible correspondence heuristics means we should be able to deal with any, even highly unreasonable, matchings. Even if
a certain correspondence procedure is based on some good and efficient method and finds
the correct matchings most of the time, it can still make mistakes, because for sparse input
data the task of correspondence can be very complicated (as shown in Figures 2.1 and 2.2).

One possible solution would be to assume that the correspondence heuristic method is
“always right.” In this case we should connect the input contours according to the obtained
matchings (while also handling branching situations correctly). These straightforward con-
nections produce intersection conflicts in the reconstructed surface (e.g., in the cases shown
in Figures 2.1(f) and 2.2(d)), and these conflicts should be resolved correctly by some post-
processing procedure.

Naturally, the problem with this solution is that we have to fully rely on some cor-
respondence heuristics, which can sometimes return inaccurate, and even unreasonable
results. Using the second assumption in the problem definition (that topological changes
are not missed by slicing), we can sometimes refrain from unreasonably connecting con-
tours by “overruling” some of the matchings obtained from the correspondence procedure.
However, the problem is then to decide what matchings are unreasonable. In our solu-
tion, we divide the space between every two adjacent slices into disjoint cells. While the
main goal of this partition is to handle branching situations, it is also used for rejecting
unreasonable matchings. The framework presented in the next section is an extension and
systematization of the results in [6].

2.3 The Proposed Framework

The proposed framework makes use of a number of different types of procedures which
interact according to the predefined data flow, as appears in Figure 2.4. Thick arrows
between blocks in the diagram relate to information about all slices, whereas thin arrows
indicate that each time only a pair of successive slices is used. Dashed arrows indicate
optional data flow. Below is the explanation about each block of the diagram:

• **Input Processing:** This block is responsible for preparing the input data for the next
  steps. If input cross-sections are given as gray-level images, polygonal contours must
  be extracted. Also, nesting hierarchy of the polygonal contours should be determined
  in this step.
CHAPTER 2. OVERVIEW

Input: Cross-sections

Input Preprocessing

Contour Correspondence Procedure $C$

Space Partitioning Procedure $P$

Contour Translation and Tiling Procedure $T$

Polygonal Surface Reconstruction Algorithm $A$

Surface Visualization

Output: Polygonal surface

Figure 2.4: Data flow of the proposed framework
2.3. THE PROPOSED FRAMEWORK

- **Contour Correspondence (C):** Establishes correspondence between contours for each pair of successive slices. Receives two slices each time as an input, but can also use the information about all input slices. The output of C is a set of pairs of matching contours, and an additional correspondence information for each such pair.

- **Space Partitioning (P):** Partitions the space between each pair of successive slices into disjoint spatial cells. The cells are created so that parts of two contours are allowed to be inside the same cell only if these contours should be connected according to C. Division of a contour by the cells’ boundaries into several parts is used to provide a reasonable solution for branching (when the split contour corresponds to more than one contour on the adjacent slice).

- **Contour Translation and Tiling (T):** Generates the surface. T translates the contours within their slice’s plane according to the correspondence information obtained from C, and then invokes A (see below) as a “black box” to produce the actual tiling between the (translated) contours or their parts inside each cell. The contours are then returned to their original locations, with the generated surface linearly translated following the contours.

- **Polygonal Surface Reconstruction (A):** A general surface reconstruction algorithm that produces a non-self-intersecting polygonal surface from contours. This can be any method that works well for dense input data (e.g., [2, 3, 5]), depending on whether they meet some specific requirements that would guarantee T’s correctness.

- **Surface Visualization:** Joins together the output for pairs of successive slices and stores the result in a 3D object representation format (e.g., VRML, X3D, etc.).

The dashed arrow from T to P is optional. It refers to situations in which two contours correspond according to C, and P has constructed the cells accordingly, but T has not been able to tile between the two contours. Probably the best (but not necessarily the only) solution for such cases is to reject the correspondence between the two contours as infeasible, and rebuild the cells again.

Figure 2.5 illustrates schematically how the proposed framework can be utilized in practice on a simple example consisting of three contours (2.5(a)). A contour correspondence
CHAPTER 2. OVERVIEW

(a) Input contours

(b) Contour correspondence \((C)\)

(c) Partitioning into cells \((P)\)

(d) Translating contours \((T)\)

(e) Invoking \(A(T)\)

(f) Translating back \((T)\)

(g) The final result

(h) Reconstruction by other algorithms

Figure 2.5: A schematic example of using the framework
procedure $C$ produces matchings as shown in Figure 2.5(b). Two disjoint cells are created by a partitioning procedure $P$ and are shown in Figure 2.5(c). Contours are translated by $T$ according to the correspondence information (2.5(d)), and a surface reconstruction algorithm $A$ is invoked to connect the contours by tiling (2.5(e)). Contours are then translated back to their original locations and the tiling is moved accordingly (2.5(f)). Since the generated surface does not contain any self-intersections (which can be easily verified in this case because tiling in each cell does not cross the cell’s boundaries), this completes the reconstruction process (2.5(g)). Figure 2.5(h) schematically illustrates the surface that would be generated by the reconstruction algorithms mentioned in the introduction [2, 3, 5, 9]. Since these algorithms do not allow for integration of an independent, general, non-overlay-based correspondence procedure, they are unable to connect non-overlapping contours.

2.4 Our Case Study

For practical use of the proposed framework we need to provide specific algorithms for $C$, $P$, and $T$, as well as choose an algorithm $A$. In the following chapters we describe the algorithms we have used in the context of our case study.

The correspondence procedure we use is based on geometric similarity between contours from adjacent slices. It does not make use of any global information (i.e., it does not look at other slices). Our algorithm may decide that two non-overlapping contours correspond, but it requires from them to be reasonably close. Contours that overlap are always considered matching. For a very sparse input this choice can lead to unnecessary connections in the resulting surface, but it is important for the presentation of our case study: this allows us to use a partitioning procedure that builds only “vertical” cells, i.e., cells that create the same partition of the plane on both adjacent slices and for every height between them. Such cells, often referred to as 2.5-dimensional, are simpler for understanding and implementation, and faster to manipulate. It is explained later how they can be easily extended to a more general case of “nonvertical” cells.

Our translation and tiling procedure benefits from the fact that “vertical” cells are faster to manipulate, but the same concept can be easily applied to a more general case. We use the straight-skeleton algorithm of [3] implemented in [24] as the block $A$ to generate
the actual tiling. Our experimental results show that the algorithms we use in our case study (even without extension to a more general case) are efficient for some kinds of input domains, especially those without nested contours.

Figure 2.6 shows the stages of reconstruction by our method for a synthetic example. The input contours are given in Figure 2.6(a), as seen from above. Figure 2.6(b) shows the correspondence information obtained by our correspondence procedure, namely, how the upper (red) contours should be moved to resemble most the lower (blue) contours. Figure 2.6(c) shows the partition of the space between the two slices into six disjoined “vertical” cells. Figure 2.6(d) shows the generated surface in two dimensions after translation, tiling, and translation back within each cell were performed. Finally, the reconstructed three-dimensional surface is shown in Figure 2.6(e).

2.5 Notation

In the described case study we do not make use of any global knowledge, i.e., each pair of adjacent slices is interpolated independently. Therefore, we will only refer to the lower and the upper slices in a given pair, denoted by \( S_0 \) and \( S_1 \) respectively. Each slice \( S_k \) \((k \in \{0, 1\})\) contains some number \( n \) (possibly zero) of contours: \( S_k = \{C_{k0}^0, C_{k1}^0, ..., C_{k(n-1)}^0\} \). We consider the height of the lower and the upper slices as 0.0 and 1.0, respectively.

We use the notation \( c_i^k \) to denote the projection of a contour \( C_i^k \) into the \( xy \)-plane. Thus \( c_i^k \) is a simple two-dimensional polygon. Further, we define two sets of such polygons: \( s_k = \{c_i^k \mid C_i^k \in S_k\} \), one for each slice \((k = 0 \text{ or } 1)\).
2.5. NOTATION

Figure 2.6: Reconstruction of a synthetic example
Chapter 3

Contour Correspondence

3.1 Function Definition and Properties

The goal of this procedure in our framework is to provide a solution for the correspondence problem. The approach we take in the method we use in our case study is based on the following basic observations:

1. For many input domains, even for sparsely-sampled objects, “shapes do not change fast.” This means that a feature of a three-dimensional object cut by a pair of successive slices will result in similarly looking contours in these slices.

2. The existing general interpolation algorithms which can be used in our framework for building the actual surface are overlap-based (as mentioned in Chapter 2). These algorithms perform better tiling of similar contours when similar features of their curves overlap as much as possible.

We describe a function $f_{SIM}$, which, given a pair of contours in adjacent slices, decides whether they should be connected (by a surface-tiling algorithm) and how. Generally speaking, the function gets two simple polygons and determines whether they are similar and in what manner. Given two simple polygons $c_i^0 \in s_0$ and $c_j^1 \in s_1$, the heuristic similarity function $f_{SIM} : s_0 \times s_1 \rightarrow \mathbb{R}^2 \cup \{null\}$ returns either a vector $\vec{v} = (\Delta x, \Delta y)$ by which $c_j^1$ should be translated to maximally resemble $c_i^0$, or $null$ if the contours were not found similar.
We require the function \( f_{\text{SIM}}(c^0_i, c^1_j) \) to have the following properties:

1. Each pair of polygons (contours) gets some similarity score. If it is above some threshold \( T \) (a parameter to the algorithm), \( f_{\text{SIM}} \) returns a translation vector \((\Delta x, \Delta y)\), otherwise \( \text{null} \).

2. The similarity criterion is based on curve matching between \( c^0_i \) and \( c^1_j \), possibly under some translation. This corresponds to the expectation that “shapes do not change fast” between two adjacent slices.

3. The greater the distance between \( c^0_i \) and \( c^1_j \) is, the smaller chance they have to be proclaimed as matching. This reflects the idea that closely-located contours are more likely to correspond.

In addition to these properties, we also require that if \( c^0_i \cap c^1_j \neq \emptyset \) (i.e., the projections of the contours \( C^0_i \) and \( C^1_j \) overlap), then \( c^0_i \) and \( c^1_j \) are considered similar. Though this requirement is somewhat restraining and is not really necessary, we introduce it in our case study in order to be able to use “vertical” cells during the partitioning phase (Chapter 4). If we do not restrain our partitioning to “vertical” cells only (we show in the end of the next chapter how this can be done), no such limitation on the correspondence procedure is necessary.

### 3.2 Contour Stitching

We need a technique to determine for a pair of polygonal contours whether or not they are similar. To be more precise, we should be able to tell whether a contour is similar to another contour or a portion of it. (For the latter case consider a branching situation in which a large contour in one slice corresponds to several smaller contours in the other slice). Such similarity cannot be expressed in terms of rotation or scaling transformations, since there is no restriction on how a shape can change between the two slices. Therefore, we cannot effectively apply the usual object recognition techniques based on matching of features that are invariant under similarity transformations (such as local maxima of curvature). All we can say about the expected similarity of contours is that their boundary curves are often
3.2. CONTOUR STITCHING

geometrically close, possibly under some (usually small) translation. Hence, a distance-based similarity matching technique is required.

First, we explain how translations are handled. Consider the example of a single tilted pipe, as shown in Figure 2.3, and suppose that \( c_0^0 \) and \( c_0^1 \) are the projections of the lower and upper contours, respectively. The contours are (almost) similar, but \( c_0^1 \) is translated by some vector \( \vec{v} \) with respect to \( c_0^0 \). So we can build a list \( L \) of relevant translation vectors, and then check similarity between \( c_0^0 \) in its original position and \( c_0^1 \) translated by each vector from \( L \) (referred to in the sequel as “poses” of the upper contour). This approach is common in object-recognition techniques. Since we want \( L \) to be finite (and relatively short), we must discretize the area of the slices’ bounding rectangle, defining some step \( \Delta \) for both \( x \) and \( y \) directions. Consequently, we can reduce our problem, given a lower contour and a pose of an upper contour, to making a decision about their similarity.

As a preprocessing step, before the contour-matching algorithm is invoked, we discretize all contours in all slices into a cyclic sequence of points with equally small distance \( \delta \) between each pair of consecutive points (\( \delta \) is a parameter to the algorithm). Thus, the length of the boundary of a contour is roughly proportional to the number of its points (after discretization).

Now we describe how, given two contour projections \( c_0^0 \) and \( c_1^j \), and a translation vector \( \vec{v} \) for \( c_1^j \), we decide whether they are similar under the given translation. Assume that \( c_0 \) is a discretized version of \( c_0^0 \) and \( c_1 \) is a discretized version of \( c_1^j \) translated by \( \vec{v} \). The following function determines whether \( c_0 \) and \( c_1 \) are similar:

\[
\text{AreSimilar}(c_0, c_1) :
\]

1. /* Initialization */
   
   Let \( c_S \) (resp., \( c_L \)) be the smaller (resp., larger) between \( c_0 \) and \( c_1 \)
   
   Let \( P_{\text{start}} \) be an arbitrary point of the circular discretized contour \( c_S \)
   
   Assign \( P_S \leftarrow P_{\text{start}} \) and \( \text{TotalStitchSize} \leftarrow 0 \)

2. /* Synchronization to start a new stitched series */
   
   Find \( P_L \) – a point on \( c_L \) closest to \( P_S \)
   
   Assign \( \text{LEN}_S \leftarrow 0, \text{LEN}_L \leftarrow 0 \)
CHAPTER 3. CONTOUR CORRESPONDENCE

If $\text{Dist}(P_S, P_L) > \varepsilon$ then:

$P_S \leftarrow \text{next}(P_S)$

If $P_S = P_{\text{start}}$ then go to Step 5

Go to Step 2

3. /* Advance on the smaller contour’s curve */

While $\text{Dist}(\text{next}(P_S), P_L) \leq \varepsilon$ do:

$P_S \leftarrow \text{next}(P_S)$

$\text{LEN}_S++$

If $P_S = P_{\text{start}}$ then go to Step 5

4. /* Advance on the larger contour’s curve */

If $\text{Dist}(P_S, \text{next}(P_L)) \leq \varepsilon$ then:

$P_L \leftarrow \text{next}(P_L)$

$\text{LEN}_L++$

Go to Step 3

Else: /* a stitched series ended, consider only if long enough */

$P_S \leftarrow \text{next}(P_S)$

If $P_S = P_{\text{start}}$ then go to Step 5

If $\text{LEN}_S \geq 3$ and $\text{LEN}_L \geq 3$ then $\text{TotalStitchSize} += \text{LEN}_S$

Go to Step 2

5. /* Walking on the smaller contour is complete */

$\text{TotalStitchSize} += \text{LEN}_S$

If $(\text{TotalStitchSize} \mid c_S \mid) \geq T$ then return TRUE

Else return FALSE
3.2. CONTOUR STITCHING

By $|c_S|$ we refer to the total number of points on $c_S$. $\varepsilon$ is a parameter to the similarity matching algorithm that reflects the maximal size of the stitches. (We achieved our best experimental results by setting $\delta < \varepsilon < 2\delta$.) $T$ is the threshold parameter mentioned in the previous section (it is usually chosen between 0.6 and 0.8). The operator $\text{next}(P)$ returns the point following $P$ in a counter-clockwise order. The above function sums up the number of points on the smaller contour that are stitched by matching series of length greater than three to the larger contour, and checks whether this number exceeds the required percentage of all points. This percentage is represented by the threshold $T$. The reason we require the matching series to be at least three points long on each of the contours is to avoid short accidental matches. This also prevents situations in which most points of a very small contour are stitched to a single point on another contour. Figure 3.1 illustrates contour stitching.

The function $\text{AreSimilar}()$ is invoked for each pair of contours, each one belonging to one of the two adjacent slices, for every translation vector $\vec{v} \in L$. Among vectors in $L$, for which similarity was claimed, we prefer the one with the smallest norm. Figure 3.2 shows an example of two possible matchings between a pair of contours; both matchings are above the threshold, and even though a somewhat better stitching can be achieved for the longer vector, the shorter vector is chosen. The reason we prefer the shortest of the vectors that provide good-enough stitching of contours is that closely-located portions of contours are generally more likely to correspond. In order to choose the shortest vector, we simply sort the vectors in $L$ by increasing norm (the zero vector is always the first),

Figure 3.1: An example of contour stitching
CHAPTER 3. CONTOUR CORRESPONDENCE

Figure 3.2: Choosing the shortest vector

and stop checking a pair of contours for matching when \texttt{AreSimilar()} returns TRUE. We also define the maximum allowed translation vector $v_{\text{max}}$ (a parameter that depends on our sampling density), so that longer contour translations will be discarded. The function $f_{\text{SIM}}$ implementing the similarity-matching algorithm is as follows:

\textbf{Function $f_{\text{SIM}}(c^0_i, c^1_j)$:}

1. Construct $L$ – the list of vectors with the discretization step $\Delta$
   - Exclude from $L$ vectors $\vec{v}$ for which $|\vec{v}| > v_{\text{max}}$
   - Sort $L$ by increasing norm

2. For each vector $\vec{v}$ in $L$ do:
   - Let $c_0$ be the discretized version of $c^0_i$
   - Let $c_1$ be the discretized version of $c^1_j$ translated by $\vec{v}$
   - If $\texttt{AreSimilar}(c_0, c_1)$ then return $\vec{v}$

3. If $c^0_i$ and $c^1_j$ overlap then return $(0,0)$, else return $\emptyset$.

Note that fixing the algorithm parameters – $\Delta$, $\delta$, $\varepsilon$, $T$ and $v_{\text{max}}$ is performed once for every input domain, e.g., blood vessels, hip bones, etc.

3.3 Discussion

The similarity-matching method we use in our case study is quite simple and is definitely not the only possible solution. For instance, another possible algorithm could be based on
3.3. DISCUSSION

Figure 3.3: An improved contour stitching

maximum area overlap between two contours as an indicator of their similarity. Even when using the same approach of contour stitching, a few improvements can be proposed. For example, if we want to improve the identification of correspondence in branching situations, when a large contour should be connected to several smaller contours, the following improvement is possible: Label stitched portions of the larger contour’s curve so that they will not be available for stitching with another small contour. This will prevent several small contours from matching the same curve portion of a large contour (which makes sense since each curve portion can be connected by tiling to only one contour on the adjacent slice). For this improvement we also have to make sure that the best-matching smaller contour is connected to any portion of the large contour. Figure 3.3 illustrates this improvement. The contour-matching algorithm, as described above, will proclaim matchings according to Figure 3.3(a), i.e., two small contours are stitched to the same portion of the large contour. With the above improvement, the more appropriate of the two small contours will “reserve” the portion of the large contour to which it was stitched. Stitching of the other small contour to the same portion will not be allowed, thus, in this case, that contour will find another matching with the large contour (Figure 3.3(b)).

If the input contains nested contours, the described procedure still works only on the outermost ones. As already mentioned, the proposed method is less effective for such inputs, especially if most changes between the slices appear in the inner contours. In such cases, other (more complicated) correspondence methods should be implemented. In general, any additional information about the nature of scanned objects can be used to provide a more effective correspondence algorithm.
Chapter 4

Space Partitioning

4.1 General Considerations

In this step we partition the space between a given pair of adjacent slices into disjoint three-dimensional cells. The main goal of this partition is to solve the branching problem, but it also creates simpler conditions for avoiding self-intersections. The idea is to define a cell for each pair of matching contours, so that tiling will be performed independently within each cell. We require that a pair of contours can be tiled successfully only if their common cell could be constructed.

We also expect that when such a cell cannot be constructed for a pair of matching contours, it will happen because the heuristic correspondence procedure was wrong in claiming that the two contours match. The reason two contours cannot be in the same cell might be that there is another cell (or a combination of cells) in the partition which interferes with the connection between them. Figure 4.1 shows an example in which a cell could not be constructed for the contours $C_0^1$ and $C_0^0$ even though they were proclaimed as matching by the correspondence procedure. If we connect $C_0^1$ to $C_0^0$ by tiling, it will probably intersect the tiling between $C_1^1$ and $C_1^0$. Such an intersection would mean that we missed a topological change during sampling, which should not happen according to our second assumption in the problem definition (see Section 2.1). Therefore, the impossibility to construct a cell for some contours can sometimes be used for rejecting incorrect matching calls by the correspondence procedure. At the same time, the more severe problem is refraining from the
The solution we propose for branching is to divide a contour that should be connected to \( m \) contours on the adjacent slice into \( m \) disjoint parts, and then treat these parts as independent contours (i.e., tile each such part separately with the corresponding (part of a) contour in the adjacent slice). We expect our partition to divide such contours into \( m \) parts.

As to avoiding self-intersections in the resulting surface, the following statement trivially holds: if a surface constructed within each of the disjoint cells does not intersect itself and does not cross the boundary of that cell, the total resulting surface is also non-self-intersecting. Hence, we have a sufficient (but not necessary) condition for not having self-intersections in the result.

Let us summarize the desired properties of the partition. Only the first property is strictly defined, others are more intuitive:

1. Two contours \( C_i^0 \) and \( C_j^1 \) that do not match according to the correspondence procedure (and, thus, should not be connected) won’t be inside the same cell.

2. Parts of two contours \( C_i^0 \) and \( C_j^1 \) that match according to the correspondence procedure should be within a common cell.

3. If a contour corresponds to \( m > 1 \) contours on the adjacent slice, it should be divided into \( m \) disjoint parts by the partition. Each part will be in a different cell (with its corresponding contour).

4. Each cell should be as large as possible so as to make it more likely for the tiling to
be bounded by the cell’s boundary. Still, the partition should be “fair,” i.e., each cell should have about the same chance of successful tiling.

The word “should” in properties 2 and 3 means that we expect the created partition to have these properties for reasonable correspondence results (i.e., those that do not imply a missed topology change). The partition we have chosen to implement in our case study, as described in Section 4.3, does not hold these properties in a few typical cases. These cases are explained in Section 4.5 and some solutions are proposed.

4.2 Voronoi Diagram of Polygonal Sites

The central concept we use in the partitioning of the space between slices into disjoint cells is the two-dimensional Voronoi diagram of polygonal sites, defined in this section.

First, let us recall the definition of the distance of a point to a polygon:

**Definition 4.2.1.** The distance from a point $q$ to a polygon $P$ is the distance from $q$ to the closest point to it on $P$: $\text{dist}(q, P) = \min \{\text{dist}(q, p) | p \in P\}$.

Now we can define the Voronoi diagram of polygonal sites:

**Definition 4.2.2.** Given a set $S$ of disjoint simple polygons in the plane, the Voronoi region (or cell) of a polygon $P \in S$ is defined as the set of all points that are closer to $P$ than to any other polygon in $S$. Formally, $V(P) = \{q | \forall P' \in S, P' \neq P : \text{dist}(q, P) < \text{dist}(q, P')\}$. The Voronoi diagram of a given set of polygonal sites is the partition of the plane into Voronoi cells.

To the best of our knowledge, Voronoi diagrams with general simple polygons as sites have not been previously studied or used. Such diagrams are mentioned by Papadopoulou and Lee [21], but their interest was in $L_\infty$ Voronoi diagram of rectilinear polygons (for VLSI applications). Geiger [11] describes a diagram to which he refers as the region Voronoi diagram (RVD). Similarly to the Voronoi diagram of polygonal sites, sites in RVD are simple closed polygonal regions. However, RVD may contain arcs equidistant from segments of the same site. For more information about Voronoi diagrams in general, the reader is referred to [1].
Figure 4.2 shows an example of a Voronoi diagram of five polygonal sites. Note that Voronoi regions can be nested, and that some regions are bounded whereas others are not.

The following two lemmas demonstrate some properties of the Voronoi diagram of polygonal sites:

**Lemma 4.2.3.** The diagram consists of straight line-segments and parabolic arcs, all of which lie in the exterior of the given polygons.

*Proof.* Edges of the Voronoi diagram of polygonal sites consists of points equidistant from line segments that define input polygons. Points equidistant from a point and a line lie on a parabola, and points equidistant from two lines or from two points lie on a line, which is a specific case of a parabola. Therefore, the diagram consists of parabolic arcs.

Any point \( p \) in the interior of a polygon \( P_1 \) is closer to that polygon than to any other polygon \( P_2 \), because the shortest line segment connecting \( p \) to \( P_2 \) crosses a segment of \( P_1 \), and so \( P_1 \) is closer to \( p \) than \( P_2 \). Therefore, points that belong to a Voronoi diagram of polygonal sites lie in the exterior of the given polygons. \( \square \)

**Lemma 4.2.4.** Each polygonal site has one non-empty Voronoi cell; the cell is a contiguous region properly containing its respective polygon, and can be either bounded or unbounded.

*Proof.* It is obvious that each polygon \( P \) has a cell, because points inside \( P \) are closer to \( P \) than to any other polygon (as shown in the proof of the previous lemma). Figure 4.2 provides an example of how Voronoi regions can be bounded or unbounded.
It is left to prove that each polygon has at most one contiguous cell. It is sufficient to show that there is a contiguous path between any two points \( p \) and \( q \) that belong to a Voronoi region of a polygon \( P \), so that the path lies entirely in a Voronoi region of \( P \).

![Diagram](image)

**Figure 4.3:** The proof there is always a segment in \( V(P) \) between \( p \in V(P) \) and \( P \)

First we prove that if \( p \in V(P) \) and \( r \) is the closest point to \( p \) on \( P \), then the segment \( pr \) lies entirely in \( V(P) \). We have to prove that any point \( s \) on the segment \( pr \) is closer to \( P \) than to any other polygon \( P' \) (see Figure 4.3). Assume that \( s \) is closer to some polygon \( P' \) than to \( r \), and \( t \) is the closest point to \( s \) on \( P' \), that is, \( st < sr \). Since \( pst \) is a triangle, \( st + ps > pt \), and so we get \( sr + ps > pt \), i.e., \( pr > pt \), in contradiction to our assumption that \( p \in V(P) \). Hence, we proved that for every point \( p \in V(P) \) there is a point \( r \in P \) so that \( pr \in V(P) \).

Now we build a contiguous path between \( p \) and \( q \) in \( V(P) \): a segment from \( p \) to a point \( r \) on \( P \), then on a boundary of \( P \) to some point \( u \), from which there is a segment \( uq \) that connects \( u \) to \( q \) and lies totally in \( V(P) \). And since there is a contiguous path between any two points in \( V(P) \), \( V(P) \) is a single contiguous region.

For practical reasons, in order to avoid dealing with parabolic arcs, we approximate them by polygonal arcs. We also use a bounding rectangle in order to deal only with finite cells. We denote the bounded, linear approximation of the Voronoi diagram of polygonal sites by \( VDP \), which is a partition of the plane into a set of non-self-intersecting polygons, possibly with holes. A \( VDP \) can, therefore, be represented as a planar polygonal map.

In order to use a \( VDP \) instead of the Voronoi diagram of polygonal sites, we require from it the following properties:

- Topological correctness: A \( VDP \) should have the same topology as the original Voronoi diagram. For example, if a curve in the original diagram lies between two
closely located polygonal sites, the line segment approximation of that curve must not intersect any of these polygons.

- Deterministic construction: For a given set of polygonal sites, the procedure that builds a VDP should always produce the same result.

4.3 Vertical Cells

In this section we explain how the linear approximation of the Voronoi diagram of polygonal sites (VDP) is used to partition the space between a pair of adjacent slices. We assume that the correspondence procedure always matches partially-overlapping contours. This allows us to use the so-called “vertical” cells, which are easier to explain. In the next section we omit this assumption. We also assume that the input does not contain nested contours. The case of nested contours is addressed separately at the end of this section.

By “vertical” cells we refer to 2.5-dimensional cells, i.e., polyhedral cells that create the same two-dimensional partition of the plane on both slices and for every intermediate height. Figure 4.4 provides an example of a partition to such cells. The reason “vertical” cells are easier to manipulate is that they can be treated as two-dimensional cells that define a partition of the plane. In this section we refer by the term “cell” to a two-dimensional cell, unless stated otherwise. Similarly, and with a slight abuse of notation, the term “contour” will refer to the xy-projection of the interior of the contour.

Next we describe the construction steps of the final partition. In the first step, disjoint cells that partition the plane are created, each cell may contain some contours or parts of
4.3. VERTICAL CELLS

contours from different slices. These cells are represented as faces of a polygonal planar map, and belong to one of the three sets:

\( T_m \): The set of cells that contain (parts of) matching contours;

\( T_e \): The set of cells empty of contours;

\( T_r \): The set containing the remaining cells.

In the following steps, some operations that split and unite cells are performed on the planar map, depending on the set to which they belong. These operations replace old cells by new ones, and the latter cells belong to one of the three sets according to their properties. According to the required properties of the partition (see Section 4.1), we expect that at the end of the partitioning process \( T_m \) will contain large cells for tiling between (parts of) matching contours, \( T_r \) will contain small cells around contours that should not be connected to contours on the other slice, and \( T_e \) will be empty (since empty cells are useless).

Before we describe the partitioning steps in detail, we need to define the term connected neighbors:

**Definition 4.3.1.** Two cells \( \sigma_1 \) and \( \sigma_2 \) of a partition \( M \) are called connected neighbors if the following holds:

1. The faces \( \sigma_1 \) and \( \sigma_2 \) are adjacent (have a common boundary with measure—not a point);

2. Both \( \sigma_1 \) and \( \sigma_2 \) contain parts of the same contour \( c \); and

3. There are parts of \( c \) in \( \sigma_1 \) and \( \sigma_2 \) that share a point which lies on the boundary between \( \sigma_1 \) and \( \sigma_2 \).

Figure 4.5 demonstrates the partition steps on a complicated example, where adjacent slices are totally dissimilar (to make sure that all steps are nontrivial). The input is shown in Figure 4.5(a). Assume that according to the correspondence procedure only the overlapping contours match (Figure 4.5(b)).
CHAPTER 4. SPACE PARTITIONING

Figure 4.5: Partitioning steps

(a) Input contours  (b) Found matchings  (c) Step 1
(d) Step 2  (e) Step 3  (f) Step 4(a)
(g) Step 4(b)  (h) Step 4(c)  (i) Step 5 (final partition)
4.3. VERTICAL CELLS

We are now ready to describe the partitioning steps in detail:

**Step 1: Create the initial partition:**
Compute the planar maps \( M_0 = \text{VDP}(s_0) \) and \( M_1 = \text{VDP}(s_1) \), then compute \( M \) – the overlay of \( M_0 \) and \( M_1 \) (see Figure 4.5(c) for an example in which \( M \) contains ten cells).

**Lemma 4.3.2.** There are only three types of cells in \( M \): \( T_0 \) – empty cells that do not contain any parts of contours, \( T_1 \) – cells that contain (parts of) a single contour, and \( T_2 \) – cells that contain (parts of) two contours, one from each slice.

**Proof.** Suppose by contradiction that there exists a cell \( \sigma \) that contains (parts of) two contours from the same slice, i.e., \( \exists i, j, k : (c_i^k \cap \sigma \neq \emptyset) \land (c_j^k \cap \sigma \neq \emptyset) \). But then each of \( c_i^k \) and \( c_j^k \) would have a different Voronoi cell in \( \text{VDP}(s_k) \). The overlay computation between two planar maps does not unite their faces (cells), therefore, it is impossible for (parts of) \( c_i^k \) and \( c_j^k \) to be in the same cell. \( \square \)

The sets \( T_m \), \( T_r \), and \( T_e \) are initialized as follows:

- \( T_m \) contains all cells from \( T_2 \) that contain (parts of) matching contours. (There are four such cells in Figure 4.5(c).)
- \( T_e \) contains all empty cells, i.e., \( T_e = T_0 \). (There is one such cell in Figure 4.5(c).)
- \( T_r \) contains the rest of the cells, i.e., cells that contain (parts of) one or two nonmatching contours. (There are five such cells in Figure 4.5(c).)

**Step 2: Split cells in \( T_r \) into simpler cells:**
Split each cell \( \sigma \in T_r \) which contains \( k > 1 \) parts of contours by the \( \text{VDP} \) of these parts. This replaces each such cell in \( T_r \) by \( k \) new cells in \( T_r \). The planar map is updated accordingly. In Figure 4.5(d) three cells have been split, for two of them \( k = 2 \) and for one \( k = 3 \).

**Lemma 4.3.3.** After performing Step 2, each cell in \( T_r \) contains a single (part of) a contour.

**Proof.** Follows directly from the definition of the step. \( \square \)
CHAPTER 4. SPACE PARTITIONING

Step 3: Unite connected neighbors in $T_r$:

As long as there are pairs of connected neighbors in $T_r$, unite them. As a result of this step, connected parts of contours that belong to multiple connected neighbors are united into one contour part as their respective cells are united. $T_r$ and the planar map are updated. Figure 4.5(e) shows the partition after performing Step 3, with two pairs of cells united.

Lemma 4.3.4. After performing Step 3, each cell in $T_r$ either contains a full contour, or contains a part of a contour and has one or more connected neighbors, all of which are from $T_m$.

Proof. Suppose that there is a cell $\sigma$ in $T_r$ that contains a part of a contour $c$ (according to Lemma 4.3.3 there is only one part of a contour in $\sigma$) and has connected neighbors not from $T_m$. Since $\sigma$ contains a part of a contour, it must have a connected neighbor $\sigma'$ which also contains a part of the contour $c$. Since we assumed that $\sigma'$ cannot be in $T_m$, and it cannot belong to $T_r$ because it contains a part of a contour, $\sigma'$ must be in $T_r$. But then $\sigma$ and $\sigma'$ would be united in Step 3 into a single cell, in contradiction to our assumption. \qed

Step 4: Absorb cells in $T_r$ by cells in $T_m$:

As long as there is a cell $\sigma_0 \in T_r$ which contains a part of a contour $c_0$, calculate $T_{cn}(\sigma_0) = \{\sigma_1, \sigma_2, ..., \sigma_n\}$ – the set of connected neighbors of $\sigma_0$ in $T_m$. If $|T_{cn}(\sigma_0)| = 1$, perform action (a) below, otherwise perform actions (b) and (c), defined as follows:

(a) Unite $\sigma_0$ and $\sigma_1$ into a single cell in $T_m$, remove $\sigma_0$ from $T_r$, and update the planar map. Figure 4.5(f) shows the partition after two cells from $T_r$ were absorbed by their only connected neighbors.

(b) Assume that each cell $\sigma_i \in T_{cn}(\sigma_0)$ contains some part of $c_0$ and (a part of) another contour $c_i$ ($c_0$ and $c_i$ are, of course, from different slices). Split the cell $\sigma_0$ by VDP($\{c_1, c_2, ..., c_n\}$) into new smaller cells. If a new cell contains disconnected parts of $c_0$, split it by VDP of these parts (exactly as in Step 2). Update the planar map. Figure 4.5(g) shows a cell with three connected neighbors split by VDP.
4.3. VERTICAL CELLS

Figure 4.6: A complex example for Step 4

Remarks:

- By splitting a cell by a VDP we refer to computing the intersection of the cell with each of the Voronoi regions. Intersection of two polygons ($\sigma_0$ and the Voronoi region of $c_i$) may result in more than one simple polygon, so the total number of newly created simple polygonal cells may be greater than $n$.

- It is possible for two (or more) cells in $T_{cn}(\sigma_0)$, $\sigma_i$ and $\sigma_j$, to contain parts of the same contour (i.e., $c_i = c_j$ for $i \neq j$). That is, we use less than $n$ contours for the VDP calculation (this happens quite rarely in practice). Figure 4.6 illustrates such a case: cell $\sigma_0$ has three connected neighbors in $T_m (\sigma_1, \sigma_2, \text{and } \sigma_3)$, two of which ($\sigma_1$ and $\sigma_3$) contain parts of the same contour from the adjacent slice.

(c) For each new cell $\sigma$ created by the intersection of $\sigma_0$ with the Voronoi region of $c_i$, check each of the following conditions: (1) If $\sigma$ and $\sigma_i$ are connected neighbors, the two cells are united into a single cell in $T_m$. (2) If $\sigma$ is a connected neighbor of two or more cells that contain the same contour $c_i$ (besides sharing the contour $c_0$), unite $\sigma$ with all these cells into a single $T_m$ cell. (See Figure 4.6 for an example in which the latter condition is satisfied.) Update the sets $T_m, T_r$, and $T_e$, as well as the planar map at the end of this action. Figure 4.5(h) shows the partition after each of the three newly-created cells was united with its connected neighbor.
CHAPTER 4. SPACE PARTITIONING

Remark:

- Not all newly-created cells satisfy one of the two conditions of action (c). A new cell can also be empty of contours (so it will be put in \( T_e \)), or contain a part of \( c_i \), but without being a connected neighbor of \( \sigma_i \) (then it will be added to \( T_r \)). The latter case means that there are more cells in \( T_r \) containing \( c_0 \) and having a connected neighbor in \( T_m \), and the cell added to \( T_r \) will be handled later.

An example of the situation described in the remark for Step 4(c) is shown in Figure 4.7. The partition after the first three steps is shown in Figure 4.7(a). Suppose that the red contour is \( c_0 \) and the cell containing only its part is \( \sigma_0 \); the two connected neighbors of \( \sigma_0 \) are \( \sigma_1 \) and \( \sigma_3 \), containing the contours \( c_1 \) and \( c_3 \), respectively. Figure 4.7(b) shows how the cell \( \sigma_0 \) is split by \( \text{VDP}(\{c_1, c_3\}) \) during Step 4. As a result, we have three new cells, one of which (in the upper right corner) is neither a connected neighbor of \( \sigma_1 \) nor of \( \sigma_3 \).
4.3. VERTICAL CELLS

However, after the other two new cells are united with $\sigma_1$ and $\sigma_3$, the third cell will have a connected neighbor in $T_m$ (the new larger $\sigma_1$), and so it will be handled in the second iteration of Step 4. Figure 4.7(c) shows the final partition.

Figure 4.8 shows a very similar case, in which, however, Step 4 works a little differently. The partition after Steps 1–3 is shown in Figure 4.8(a), and one can notice the difference in cells compared to Figure 4.7(a). The cell $\sigma_0$ is split by VDP during Step 4(b) into simple polygonal cells, and this time only two such cells are created (rather than three, as in the previous example). But since one of the newly-created cells contains two disconnected parts of a contour, it is further split between these parts, and the first iteration of Step 4(b) ends with three newly-created cells again. From this point, the example is processed exactly as the previous one, and the final partition is shown in Figure 4.8(c).

**Lemma 4.3.5.** After performing Step 4, each cell in $T_r$ contains a full contour.

**Proof.** We have to prove that Step 4 terminates, and that when it does, there are no cells that contain multiple contours or a part of a single contour.

It is easy to show that a cell in $T_r$ cannot contain (parts of) multiple contours after Step 4: After Step 3, a cell in $T_r$ contains (parts of) a single contour. During actions (a) and (c) of Step 4, cells in $T_r$ can only disappear, and during action (b) cells in $T_r$ are split into smaller cells, so these new cells cannot contain (parts of) two different contours.

Next, we prove that Step 4 terminates. Assume, without loss of generality, that the cell $\sigma_0$ we split during Step 4 contains a part of a contour $c_0$ which belongs to the upper slice. First, let us see how the cell is created. The VDP of all contours from the lower slice at Step 1 divides the plane into regions; suppose that parts of $c_0$ appear inside $k$ regions. Some of these Voronoi regions become cells in $T_m$ after Step 1, namely, those containing (a part of) a contour from the lower slice, that corresponds to $c_0$, when the parts of these matching contours are not split into different cells by the map overlay. Suppose that $m$ such cells ($m \leq k$) are created. Parts of $c_0$ for which it is not the case find themselves alone in a $T_r$ cell after Step 2. Connected neighbors of this kind can be united into bigger $T_r$ cells in Step 3.

Suppose that before Step 4, the cell $\sigma_0$ has $n$ connected neighbors from $T_m$, denote them by $\sigma_1, \sigma_2, ..., \sigma_n$ ($1 < n \leq m$). Suppose also that during Step 4(b) we split $\sigma_0$ by
VDP of $k \leq n$ contours ($k$ can be smaller than $n$ only if $\sigma_0$ has two connected neighbors containing the same contour). An intersection of $\sigma_0$ with the Voronoi region of some contour $c_i$ from the neighbor $\sigma_i$ may result in several simple polygonal cells (e.g., as shown in Figure 4.7(b)). Also, if a newly-created cell contains two or more disconnected parts of a contour, it is split into several cells (e.g., as shown in Figure 4.8(b)). Either way, one of the newly created cells will always be a connected neighbor of $\sigma_i$ through the common boundary of $\sigma_i$ and $\sigma_0$, so it will be united with $\sigma_i$ during Step 4(c). But other newly-created cells may not be connected neighbors of $\sigma_i$. Since these cells contain a part of $c_0$, at least one of them (suppose, $\sigma$) must have a connected neighbor in $T_m$ after Step 4(c), implying that $\sigma$ will undergo another iteration of Step 4.

In the next iteration of Step 4, if $\sigma$ has a single connected neighbor, it will be absorbed by that neighbor at action (a). If it has more than one connected neighbors, it will be split again by the VDP of the contours inside these neighbors. However, since $\sigma_i$ is not a connected neighbor of $\sigma$, $c_i$ will not participate in the VDP this time. Hence, every iteration of Step 4 on a part of an original $T_r$ cell $\sigma_0$ has at least one contour less participating in the VDP, so eventually the iterations must end when $\sigma$ is united with a single connected neighbor at action (a) (e.g., as in the examples in Figures 4.7 and 4.8).

Now it is left to show that after all iterations of Step 4, no cells in $T_r$ contain only a part of a contour. Suppose such cell, $\sigma'$, exists. Since it contains only a part of a contour, it must have a connected neighbor, say, $\sigma''$. Obviously, $\sigma''$ cannot be from $T_m$, otherwise more iterations of Step 4 on $\sigma'$ would be required. So $\sigma'' \in T_r$. According to Lemma 4.3.4, two connected neighbors in $T_r$ are not possible after Step 3, so it is left to show that such cells cannot exist after Step 4. Step 4 operates on a $T_r$ cell $\sigma_0$ containing a contour $c_0$ that has connected neighbors from $T_m$. Therefore, after each iteration, no matter how the contour $c_0$ is divided between newly-created cells, at least one of these cells always has a connected neighbor in $T_m$. □

Step 5: Discard empty cells:

The goal is to take advantage of useless space occupied by empty cells to enlarge cells in $T_m$, where it can help in the tiling stage. Hence, as long as there is a cell $\sigma$ in $T_e$ which has neighbors (adjacent faces) from $T_m$, unite $\sigma$ with the neighbor from $T_m$ with which
it has the longest common boundary. Unite the rest of the empty cells with cells from $T_r$ in the same manner. Remove the empty cells from $T_e$, update $T_m$, $T_r$ and the planar map accordingly. Figure 4.5(i) shows the final partition of this example, after the current step.

Remark:

- The more “fair” but unreasonably expensive implementation for Step 5 would be to split empty cells by the VDP of the contours inside their neighbors from $T_m$. Since each such neighbor contains (parts of) two contours, both contours can be treated as a Voronoi site (or, in other words, their Voronoi regions can be united). This way contours that are closer to the empty cell will get more of its space.

**Lemma 4.3.6.** *After Step 5, the set $T_e$ is empty of cells.*

**Proof.** Follows trivially from the definition of this step. □

We can see intuitively that the described steps correspond to the required properties of the partition stated in Section 4.1. Formally, we can prove the following:

**Proposition 4.3.7.** *After Steps 1–5, the partition consists of two types of cells: cells that contain (parts of) two matching contours, and cells that contain a single full contour.*

**Proof.** We have to prove three claims:

1. There are no cells empty of contours or containing a part of a contour:
   This follows directly from Lemmas 4.3.5 and 4.3.6.

2. There are no cells that contain parts of more than two contours:
   According to Lemma 4.3.5, $T_r$ contains cells with a single contour, so we only need to prove that cells with more than two contours do not exist in $T_m$. In Step 1, $T_m$ is initialized to contain cells with (parts of) two contours (Lemma 4.3.2). Steps 2 and 3 do not handle cells from $T_m$. Step 4 unites cells from $T_m$ with their connected neighbors from $T_r$ (possibly split into smaller cells), thus, it adds parts of one of the contours which are already in the cell. The situation in which two cells from $T_m$ are united is possible at action (c) of the step, but it happens only if the two cells contain
parts of the same two contours, so their union creates a $T_m$ cell with exactly two contours. Step 5 handles empty cells, and, thus, it cannot add another contour to a cell from $T_m$. Therefore, cells in $T_m$ contain (parts of) only two contours, and those are the contours with which the cells were initialized (before they absorbed other cells).

3. If a cell contains (parts of) two contours, these contours match (according to the correspondence procedure):

   It has been shown above that in the final partition, cells from $T_m$ contain (parts of) only those contours, with which they were initialized. According to the definition of $T_m$’s initialization, those contours match.

Note that the space partitioning between adjacent slices presented in this case study is efficient mostly for data that do not contain nested contours. The described partitioning procedure (with a slight extension) can easily handle nested contours, too, but not necessarily in the way we would expect. In order to be able to work with any input, even containing nesting hierarchies, the partitioning steps should be based only on the outer contours. When the partition is complete, previously inner contours may become outer (consider a hierarchy of at least three contours split by the common boundary of two cells, as shown in Figure 4.9). However, it does not affect a general interpolation algorithm, which will perform surface tiling inside each cell, regardless of how many contours there are.
4.4 Extension to Nonvertical Cells

We would often like to allow our correspondence procedure to consider overlapping contours as not corresponding. In such a case using only “vertical” cells, as described in the previous section, is not sufficient, because such cells do not allow us to separate between overlapping contours. In this section we show a simple extension to the partitioning procedure described above which allows such separation.

When dealing with “nonvertical” cells, we can no longer refer to them as two-dimensional, because we have to consider the height. However, we can deal with what we refer to as “almost-vertical” cells. These cells are created by intersection and union operations on “vertical” cells whose height may be either the whole interval between the slices (as before) or half of it (either the upper or the lower). The new partitioning procedure consists of exactly the same steps, but these steps are now applied on the “almost-vertical” cells. The VDP computation, defined only for two dimensions, is still applied to the contours’ projections, but the result is vertically lifted into 3D (so after Step 1 all cells are still “vertical”). The definition of connected neighbors remains exactly the same. The only difference is that a cell with height 1 can now be divided into two distinct cells by the plane \( z = 0.5 \) to separate between nonmatching overlapping contours during Step 2. In the subsequent steps, these cells are treated as totally independent, and can be independently united with their neighbors. It is easy to see that Proposition 4.3.7 holds for “almost-vertical” cells, too.

Figure 4.10 shows an example of partitioning by “almost-vertical” cells. The contours and the partition are shown in both top and front views. Figure 4.10(a) shows the input contours and their correspondence. Note that there are two overlapping contours that do not match (according to the correspondence procedure). Figure 4.10(b) shows the partition into three “vertical” cells, created by applying VDP on each slice, calculating the overlay of the two planar maps, and extending the result into three dimensions (Step 1). The middle cell, which contains two nonmatching contours, belongs to \( T_r \), and the other two belong to \( T_m \). Figure 4.10(c) shows the partition after Step 2; the cell in the middle was split horizontally into two distinct cells. Now we have two cells in \( T_m \) and two cells in \( T_r \). Step 3 has no effect because there are no connected neighbors from \( T_r \) to be united. In
Step 4 we unite the connected neighbors from $T_m$ and $T_r$. This leaves us with only two larger cells, both in $T_m$. Step 5 has no effect in this example, so the partition after Step 4, shown in Figure 4.10(d), is also the final partition.

### 4.5 Shortcomings and Solutions

In this section we discuss the shortcomings of the partitioning procedure described in Section 4.3 and explain how some of them can be overcome. While the procedure guarantees that a common cell will not be constructed for a pair of nonmatching contours, it is possible that matching contours will not be in a common cell in the final partition. The translation
4.5. SHORTCOMINGS AND SOLUTIONS

Figure 4.11: Shortcomings of the partitioning from Section 4.3

and tiling procedure (see Chapter 5) does not connect such contours by tiling, which effectively rejects the matching claimed by the correspondence heuristics. We can point out a few typical cases, in which the partitioning does not produce the expected results and reasonable matchings can be rejected.

The first type of the typical cases is characterized by the combination of two factors: the distance between matched contours is large in comparison to the size of the contours, and some contours are matched with more distant contours rather than with closer ones. Figure 4.11 shows two examples which demonstrate the problem. In Figure 4.11(a), the two rectangular contours match, as well as the two circular contours. The partition is shown before Step 5, i.e., before the empty cell is discarded. In the final partition of this example, there will be four cells in $T_r$ instead of two cells in $T_m$ (as expected), and as a result, the matching contours will not be connected. In Figure 4.11(b), the large rectangular contour matches two small rectangular contours, and the final partition is shown. The partition consists of four cells in $T_r$ instead of two cells in $T_m$ and one in $T_r$, as expected.

Most cases of this type can be resolved by adding a few enhancements to the partitioning procedure, as is explained below. These enhancements were not included in the procedure in Section 4.3 in order to simplify the presentation of the partitioning phase. From the practical point of view, although matchings characterized by the two mentioned-above factors can be considered reasonable, they imply that the input is too sparse. We have not encountered such situations while experimenting with real data, and resolving them correctly did not seem significant to us for showing the effectiveness of our method.
The main changes which should be introduced to the partitioning procedure from Section 4.3 to resolve the above situations are as follows. First, cells in $T_r$ should in some cases be treated as cells in $T_m$ (in particular, two cells from $T_r$ containing matching contours should be united). This can be done through defining the term *matching neighbors* similarly to *connected neighbors*, but for neighboring cells that contain matching contours. (For example, in Figure 4.11(a), $\sigma_2$ and $\sigma_3$ are matching neighbors, and in Figure 4.11(b), $\sigma_2$ is a matching neighbor of both $\sigma_0$ and $\sigma_1$.) Step 4 should process matching neighbors similarly to connected neighbors. Second, empty cells should be used during the partitioning process to help non-neighboring cells in $T_r$ which contain matching contours to become one large cell in $T_m$. For instance, in the example shown in Figure 4.11(a), $\sigma_0$, $\sigma_1$, and $\sigma_4$ should become one cell in $T_m$. In order to do so, empty cells should be united with cells in $T_r$ at the end of Step 2, as well as during Step 4 (in which new empty cells may appear).

The second type of typical cases in which the produced result differs from the expected one is shown in Figure 4.12, and is referred to as a “tilted branching” case. In order to solve such situations, we can look at *sets of contours* that match another contour or set of contours under (almost) the same translation vector. These matching sets can be determined either at the correspondence phase, or in a preprocessing step of the partitioning phase, by analyzing similar translation vectors of matching contours. At the beginning of the partitioning phase, after a VDP for each slice is calculated and before the two maps are overlayed, neighboring cells which belong to contours from the same set should be united. In the subsequent steps of the partitioning, such contours should be treated as a single contour. During the translation and tiling phase, contours from the same set should be translated simultaneously (provided that these contours share a common cell in the final partition).

Finally, the third type of problematic cases is illustrated in Figure 4.13, and is referred
to as “asymmetric branching.” Such branchings between the base contour and \( m \) contours of the adjacent slice occur when some of these \( m \) contours are located significantly farther than others from the base contour. These branchings cannot be resolved well by applying a VDP directly to the original \( m \) contours, because some edges (as in Figure 4.13(a)) or vertices (as in Figure 4.13(b)) of the Voronoi diagram may miss the base contour. The question whether such branchings are “reasonable” from the correspondence point of view is arguable. In any case, if “asymmetric branchings” are to be handled, it will require combining the Voronoi diagrams with some other approach to dividing contours. We are currently unable to handle properly such cases.

Figure 4.13: Examples of “asymmetric branchings”

(a) Branching from one to two contours  
(b) Branching from one to four contours
Chapter 5

Surface Interpolation

5.1 Translation and Tiling

Now we want to apply an existing general surface-interpolation algorithm to tile (parts of) contours for each cell of the partition separately. The first three sections of this chapter assume we are dealing with “vertical” cells, as it is our main case study of the proposed framework. The last section discusses how translation and tiling is performed for “nonvertical” cells, described in Section 4.4.

We require the following properties from the surface-interpolation algorithm we intend to use as a “black box”:

**Property 5.1.1.** The algorithm creates a non-self-intersecting surface.

**Property 5.1.2.** The xy-projection of the constructed tiling of the given contours lies fully inside the symmetric difference of their projections.

**Property 5.1.3.** The algorithm connects every pair of contours from adjacent slices that overlap (in their xy-projections).

The last property is important for connecting matching contours, but is not required for the proof of correctness. The algorithms of [2, 3], and (with a slight modification) [5] have all the three properties, and thus can be used in our framework. Denote the chosen algorithm by $A$. 

49
CHAPTER 5. SURFACE INTERPOLATION

Figure 5.1: Tiling by a general reconstruction algorithm

Figure 5.1 shows an example of a surface reconstructed by the straight-skeleton-based algorithm of [3]. We can see that all the three properties hold, especially note Property 5.1.2. Note also that the heights of some of the vertices are different from 0 and 1. The straight-skeleton algorithm is used both for surface tiling in our case study, and as an example of an overlay-based reconstruction method.

The algorithm of [3] makes use of the straight skeleton structure, calculated for polygons that form the symmetric difference of a successive pair of input slices. The straight skeleton is defined by a continuous shrinking process, in which the edges of a polygon are moved inwards parallel to themselves at a constant speed. As the edges move in this way, the vertices (shared by pairs of edges) also move, at speeds that depend on the angle of the vertex. Neighboring vertices can meet in the course of this process, in which case the edge bounded by these vertices disappears. If one of the moving vertices collides with a nonadjacent edge, the polygon is split in two by the collision, and the process continues in each part. The straight skeleton is the set of curves traced out by the moving vertices in this process. Figure 5.2 shows the straight skeleton of a simple polygon, and Figure 5.3(a) shows
the straight skeletons of five simple polygons after they were triangulated by a straight-skeleton-based reconstruction algorithm. More details on the straight skeleton of polygons can be found in [3, 24]. For more information on the straight skeleton of straight line graphs, the reader is referred to [1].

Given a cell that contains some parts of contours $C_i^0$ and $C_j^1$ (assume that their heights are 0 and 1, respectively), and a translation vector $\vec{v} = f_{SIM}(c_i^0, c_j^1)$, the interpolation inside the cell is performed as follows:

1. Translate all parts of $C_j^1$ that lie within the cell by $\vec{v}$. If $\vec{v} = (\Delta x, \Delta y)$, then we translate all its points by $(\Delta x, \Delta y, 0)$.

2. Invoke the interpolation algorithm $A$ on the parts of $C_i^0$ inside the cell and the translated parts of $C_j^1$ to generate the tiling.

3. Return the translated parts of contours to their original locations. Each point that defines edges of the tiling is translated by $(-\Delta x \cdot h, -\Delta y \cdot h, 0)$, where $h$ is its height between 0 and 1.

Figures 2.5(d) and 5.3(b) show the results of interpolations of two synthetic examples in two dimensions, and figures 2.5(e) and 5.3(d) show their respective reconstructed spatial surfaces.

Note that some parts of the tiling generated by $A$ may not connect between contours on adjacent slices, but rather belong to a single contour. We require (for the cells of $T_m$ only) that such parts of the tiling lie entirely in the contour’s plane. This solves two problems: First, it prevents these parts of the tiling from being translated, which would be non-intuitive; Second, this way we avoid “pseudo-branchings,” i.e., unnatural connections between parts of a split contour along the common boundary of their respective cells.

For example, consider the lower contour in Figure 5.3(b) which is split into three parts. Each part of the contour belongs to a different cell, and is treated by $A$ as an independent contour. Without the above requirement, connections between these parts would look like usual branchings. The original contour has a long strip that cannot be connected to contours on the upper slice; clearly, branchings along this strip would look very unnatural.

While the algorithm of [5] always meets the above requirement, other algorithms do not generate tiling triangles that lie within a single slice. Using such algorithms requires a
post-processing procedure that “suppresses the height” of such triangles (at the end of the second step, i.e., before the contour is moved back). Since we used the straight-skeleton algorithm \cite{3} for interpolation, we implemented a simple “height suppression” procedure. In Figure 5.3(b) segments of tiling that lie within the lower slice are colored in blue, whereas the rest of the tiling segments are purple. Cells of $T_i$ do not require such a post-processing, as can be seen from the same figure. In more realistic examples, however, the need for “height suppression” is relatively rare.

The three-step translation and tiling procedure performed for each cell, as described above, does not guarantee that the created surface will not intersect itself. Section 5.2 discusses why self-intersections are possible and how we can avoid them, and Section 5.3 describes an iterative algorithm which uses the three-step procedure to always construct a non-self-intersecting surface.
5.2 Legal and Illegal Interpolations

We start with the following definition of a *legal interpolation* by the three-step procedure from the previous section:

**Definition 5.2.1.** Surface interpolation inside a cell is legal if all points defining the resulting tiling lie within the cell’s boundary. Otherwise, the interpolation is illegal.

Using the above definition, we can formulate the two following lemmas:

**Lemma 5.2.2.** If a legal interpolation is created for all cells of the partition by an algorithm $A$ having Property 5.1.1, then the resulting surface does not intersect itself.

Proof. Property 5.1.1 states that the surface created by the algorithm $A$ does not intersect itself. After the surface has been created in the second step, all parts of the upper contour from the cell are translated back to their original locations, together with the generated tiling. The transformation applied to the tiling is a *three-dimensional shear transformation along the z-axis* which can be expressed as $x' = x + az$, $y' = y + bz$, $z' = z$, where $(a, b)$ is the contour’s translation vector in its supporting plane, and $0 \leq z \leq 1$. And since shear is an *affine transformation*, which preserves *collinearity*, it cannot add self-intersections. Therefore, if Property 5.1.1 is held by $A$, the resulting interpolation in each cell cannot contain self-intersections.

Since it is given that the interpolation in each cell is also *legal*, the generated non-self-intersecting surface does not cross the cell’s boundary. Furthermore, since all cells are disjoint, surfaces from different cells cannot intersect either. Therefore, the resulting surface (for the given pair of adjacent slices) contains no self-intersections. ☐

**Lemma 5.2.3.** If some interpolation algorithm $A$ has Property 5.1.2, then the interpolation will be legal for any cell under the zero (identity) translation vector.

Proof. The three-step interpolation inside each cell is performed on the (parts of) contours that lie inside the cell. According to Property 5.1.2, the $xy$-projection of the constructed tiling will lie inside the symmetric difference of the $xy$-projections of (the parts of) the contours, and hence, inside the polygon which is an $xy$-projection of the “vertical” cell. ☐
According to Lemma 5.2.2, after performing the three-step interpolation, we can test for each cell whether the generated tiling crosses the cell’s boundary, and if all cells are found legal, we know for sure that the surface is non-self-intersecting. However, the problem is what happens when the interpolation in one or more cells is illegal.

Figure 5.3(b) shows how such situation is possible: when a contour is moved before tiling and then moved back after $A$ is invoked, Property 5.1.2 does not hold any longer for the transformed tiling. And when the property does not hold, the tiling may cross the boundary of its cell, as in the case of two cells in Figure 5.3(b).

Two possible solutions can be offered. The first solution is to make use of the area of a neighboring cell when that area is no longer required by the neighbor. In the example in Figure 5.3(b) the two cells with illegal interpolation make use of an unneeded area of another cell. However, this method is sometimes insufficient to solve the problem, since neighbors do not always have an unneeded area that can be used.

Another method of dealing with illegal interpolation inside a cell is by choosing a less optimal translation vector. For example, if according to $f_{SIM}$ the optimal vector is $\vec{v}$, and the interpolation under this vector is illegal, we can try, for example, the vectors $\frac{3}{4}\vec{v}$, $\frac{1}{2}\vec{v}$, $\frac{1}{4}\vec{v}$, and, finally, $(0,0)$. The latter guarantees a legal interpolation (according to Lemma 5.2.3), however, non-overlapping parts of contours will not be connected. In general, choosing less optimal vectors leads to less optimal tiling or even correspondence. Yet, a shorter translation vector in the optimal direction is preferable to the zero vector.

In the next section we present an algorithm that generates a geometrically correct (non-self-intersecting) surface for a given pair of adjacent slices. It makes use of the three-step interpolation procedure to construct the surface, and of both methods described above to solve the problem of illegal interpolations.

## 5.3 Dependency Graphs and the Iterative Algorithm

The algorithm presented in this section works in iterations on a so-called dependency graph. All geometrical operations performed by the algorithm (like testing for intersections, etc.) are applied to two-dimensional data, for efficiency and simplicity. Hence, the term tiling in this section refers to the $xy$-projection of the three-dimensional tiling.
5.3. DEPENDENCY GRAPHS AND THE ITERATIVE ALGORITHM

Definition 5.3.1. A cell $\sigma_i$ depends on a cell $\sigma_j$ (denoted by $\sigma_i \rightarrow \sigma_j$) under a given interpolation of $\sigma_i$ if the resulting tiling crosses the boundary of $\sigma_j$.

Using the above definition, we define a dependency graph for a given pair of adjacent slices as a directed graph whose vertices are cells, and there is a directed edge from a vertex $\sigma_i$ to a vertex $\sigma_j$ if $\sigma_i \rightarrow \sigma_j$ under the current interpolation of $\sigma_i$. Note that dependency graphs may contain cycles. In particular, it is possible that both $\sigma_i \rightarrow \sigma_j$ and $\sigma_j \rightarrow \sigma_i$ are edges of the graph. Note also that $\sigma_i \rightarrow \sigma_j$ does not necessarily imply that $\sigma_i$ and $\sigma_j$ are neighbors.

We define three possible colors for vertices: white, black, and gray. A vertex is white if the tiling constructed for its corresponding cell is guaranteed not to conflict (intersect) with the tiling of any other cell. A vertex is black if the tiling constructed for its corresponding cell crosses the cells’ boundary and may conflict with the tilings of other cells. A vertex can be gray only in intermediate steps of the current iteration of the algorithm. Gray vertices are those that start the current iteration as black, but may become white at the end of it.

Beside color and incident edges, cells (vertices) have a few other attributes:

- **Self region**: The polygonal region occupied by the cell;
- **Self tiling region**: The minimal polygon that contains the current tiling for the cell (current, because cells can be interpolated several times);
- **Others’ tiling region**: The polygonal region inside the cell permanently occupied by other cells’ tilings (may consist of several simple polygons);
- **Others’ tiling temporary region**: The polygonal region inside the cell temporarily occupied by other cells’ tilings;
- **Current translation vector**: The fraction of the optimal translation vector that will be used for the next tiling. The current vector can be between zero and the cell’s optimal vector, always in the optimal direction.
We are ready to describe the iterative reconstruction algorithm:

1. **Initialization**: The dependency graph contains all cells as vertices. All vertices are black, and no edges exist yet. All regions of all cells are empty. The translation vector of each cell is its optimal vector.

2. **Interpolating black cells**: For each black cell $b$, invoke the three-step interpolation procedure, update the self tiling region, and test whether it crosses the cell’s boundary. If not, change the color of $b$ to white; otherwise, add a directed edge from $b$ to each cell it depends on (according to the definition of the dependency relation).

3. **Testing black cells for tiling conflicts**: For each black cell $b$, test whether it can be tiled using unneeded regions of cells it depends on. This is done by testing whether $b$’s self tiling region overlaps with either self or others’ tiling regions of the cells it depends on. If no conflicts were determined, then do the following: If $b$ depends on at least one black or gray cell, change $b$’s color to gray and update the others’ tiling temporary region of every cell $b$ depends on. Otherwise (if $b$ depends only on white cells), update the others’ tiling regions of these cells and change $b$’s color to white.

4. **Choosing less optimal vectors for black cells**: Cells that are still black during this step cannot be tiled without conflicts. Thus, a less optimal translation vector should be attempted for them. Each black cell is assigned a vector whose norm is smaller than the one used in the previous attempt. The number of steps between the optimal vector and the zero vector is a small parameter specified by the user.

5. **Ruling out problematic gray cells**: As long as there are gray cells that depend on at least one black cell, change their color from gray to black. Since it is not known in this step what the unused regions of black cells would be at the end of the algorithm, gray cells cannot count on them.
6. *Turning gray cells into white*: Change the color of each remaining gray cell \( g \) into white. For each cell on which \( g \) depends, its others’ tiling temporary region becomes part of its others’ tiling (permanent) region.

7. *Testing whether more iterations are needed*: If all the remaining cells are white, the algorithm terminates. Otherwise, clear the others’ tiling temporary regions of all cells and go back to Step 2 (and start a new iteration).

**Lemma 5.3.2.** The algorithm terminates.

**Proof.** Before the application of Step 4 of the algorithm, there are two possible options: either there is at least one black cell, or all cells are gray or white. If there are no black cells, the current iteration is also the last one since all gray cells will become white in Step 6. If there is a black cell, a shorter vector will be chosen for it. The number of cells in a graph is finite and constant, and the number of vectors chosen for each cell before using the zero vector is also fixed. From Lemma 5.2.3 we know that the interpolation under the zero vector is always legal, and, hence, a cell with a zero vector becomes white in Step 2. Therefore, after a finite number of steps all cells become white and the algorithm terminates. 

The initial graph for the example shown in Figure 5.3(b) consists of four vertices, two are black and two are white, and the two black cells depend on the same white cell. The example is solved in a single iteration of the algorithm, as the black cells turn white during Step 3.

The next proposition provides the main result about the presented case study:

**Proposition 5.3.3.** The algorithm always constructs a non-self-intersecting surface if \( A \) has Properties 5.1.1 and 5.1.2.

**Proof.** We already proved that the algorithm terminates (because of Lemma 5.2.3 which is based on Property 5.1.2). We have to prove now that upon the termination of the algorithm, the generated surface does not intersect itself. According to the definition of Step 7, the algorithm terminates only when all cells are white. A cell can become white only during
Steps 2, 3, and 6. Such an event during Step 2 means that the interpolation inside the cell is legal and (by Property 5.1.1) contains no self-intersections. Such an event during Step 3 means that the cell depends only on cells that are already white and that its tiling does not intersect their tilings. In fact, the algorithm tests that there is no intersection in the projections of the tiling in two dimensions, which guarantees that the respective surfaces do not intersect either.

Thus, it is left to show that the tiling of a cell that turns from gray to white in Step 6 does not intersect the tilings of other white cells (some of which may become white at the same step). Gray cells are cells created in Step 3, whose tiling uses the unneeded area of their black neighbors without conflicts. Such cells cannot become white already in Step 3 because black cells will change their tiling in the next iteration, which may cause a tiling conflict. However, Step 5 of the algorithm guarantees that no gray cell depends on a black cell at the end of this step, so it is safe to turn all gray cells into white in Step 6. There are no tiling conflicts between gray cells and their neighbors (by the definition of Step 3), and new conflicts cannot arise because gray cells have no black neighbors after Step 5, so the surfaces generated in gray cells that turn white in Step 6 cannot intersect either.

The question left to resolve is what happens when (parts of) matching contours could not be connected by tiling. This may happen when a less optimal vector is used (or even when the declared “optimal” vector chosen by the correspondence procedure is not so good). The problem is that a part of a contour, which was “reserved” by the partition for a connection with a certain contour from the adjacent slice, was not connected at the end.

Consider the example in Figure 5.4(a): The correspondence procedure decided that the upper (red) contour should be connected to each of the three (blue) contours in the lower slice. The created partition splits the upper contour into three parts, according to the required branching. Now suppose that, for some reason, a lower contour in one of the cells could not be connected by tiling to the corresponding part of the upper contour. As a result, this part remains disconnected from contours in the other slice, which will make the branching look quite unnatural (even despite the “height suppression”—see Section 5.1).

Although such situations are quite rare, we might want to resolve them by rejecting the unsuccessful match between the contours that could not be tiled together, and rebuilding the
5.4. **THE GENERAL THREE-DIMENSIONAL CASE**

When dealing with “nonvertical” cells, the main difference is that Lemma 5.2.3 does not hold. We can see in the example shown in Figure 4.10(d) how an interpolation under the zero vector can intersect the boundary of a cell. Therefore, the algorithm described in the previous section should be modified in order to always terminate. The modification is the following: if a cell could not turn white under the zero translation vector (a phenomenon which was previously impossible), (the parts of) the two contours in that cell should be tiled separately in the next iteration of the algorithm. The effect of this action is the same as if the cell was split horizontally by the plane \( z = 0 \), i.e., the contours were not connected between them by tiling. Such interpolation if always legal, provided that the algorithm \( A \) does not build a surface higher than 0.5 when interpolating a single contour.

Another modification of the algorithm is required when testing for conflicts. For “vertical” cells we check that there is no intersection in tiling projected onto the plane, because this condition is sufficient to ensure that the surface does not intersect either. In the general case we should check for surface intersections in three dimensions, which is significantly more time-consuming. Nevertheless, this is necessary, as is shown in Figure 4.10(d): if we test only the \( xy \)-projections for intersections, the two cells in the figure will not turn gray in

![Figure 5.4: Recalculating the partition for a failed tiling](image)

(a) The initially-calculated partition  
(b) The recalculated partition

Figure 5.4: Recalculating the partition for a failed tiling

partition again (as explained in the framework description, see Figure 2.4). Figure 5.4(b) shows the recalculated partition after a match between two contours was rejected. With the new partition the branching in the final surface will look much more natural.

5.4 **The General Three-Dimensional Case**
Step 3 (and later white in Step 6), but will rather remain black, and at the end the matching contours will not be connected by tiling.
Chapter 6

Experimental Results

An integral part of this thesis is an object-reconstruction system based on the algorithms from the described case study of the presented framework. It was implemented in C++ in the Visual Studio 2003 environment under the Windows XP operating system on a PC machine. The source code consists of about 8000 lines of code. We used CGAL [8] (a geometric algorithm library) for geometric data structures and operations. The VRONI software package [13] was used for the construction of linear approximations of Voronoi diagrams of line segments, and the implementation of the straight-skeleton-based reconstruction algorithm of [24] was used for surface tiling.\(^1\) The system’s three-dimensional output was constructed in VRML, while intermediate two-dimensional results were provided using the WinFIG shareware and its file format.

The implemented object-reconstruction system contains the contour matching algorithm described in Chapter 3. The translation and tiling were implemented according to the description in Chapter 5. The algorithm of space partitioning corresponds to the description in Section 4.3, i.e., all steps are implemented for “vertical” cells and non-nested contours. Linear approximations of Voronoi diagrams of polygonal sites (VDPs) were computed using the approximations of line segment Voronoi diagrams (LSVD), obtained by VRONI. Arcs of LSVD (approximated by line segments) that are equidistant from points or segments belonging to the same polygonal sites are removed from the (linearly-approximated) LSVD to obtain the resulting VDP. The experimental results provided below demonstrate

---

\(^1\) We are grateful to M. Held and E. Yakersberg for providing us their respective software packages.
the behavior of our framework-based reconstruction system for different types of inputs.

Figure 6.1 shows a synthetic example in which the advantages of the proposed method can be clearly noticed. The input, shown in Figure 6.1(a), consists of six slices, each slice represented by a different color. The object was reconstructed by our system twice: when contour translation is allowed, and when it is not. In the latter case, the system behaves as a contour-overlap-based method. Tiling of one pair of slices for this case is shown in Figure 6.1(b). Figure 6.1(c) shows tiling of the same pair of slices when contour translation is enabled. Clearly, the tiling in Figure 6.1(b) suffers from the “double cone” problem which is shown in Figure 2.3(a).

Figures 6.1(d) and 6.1(e) show the reconstructed surface for the cases when contour translation is first disabled, and then enabled, respectively. One can observe that in Figure 6.1(d) non-overlapping contours are not connected, leaving gaps in the reconstructed shape. Even when contours from adjacent slices are connected by tiling, the quality of tiling between these contours is significantly better when contour translation is enabled.

Figures 6.2 and 6.3 demonstrate the behavior of our system in complicated synthetic examples. Figure 6.2 shows the reconstruction for the example shown in Figure 4.6. This example contains a complicated branching situation, in which a contour in one slice should be split and connected to three contour parts in the other slice, two of which belonging to the same contour. Figure 6.2(a) shows the final partition and tiling in two dimensions, and Figure 6.2(b) shows the reconstructed spatial surface.

The robust behavior of our system is shown in the example in Figure 6.3. The example contains eight contours in the lower slice and 16 contours in the upper slice. Some upper contours match some lower contours, but most contours in the example are “randomly” distributed in their respective slices. The contours are of different sizes and forms: some contours are small and simple, whereas others are large and complex. Figure 6.3(a) shows the final partition containing 20 cells, as well as the xy-projection of the tiling. The reconstructed surface is shown in Figure 6.3(b).

Figure 6.4 shows a fully reconstructed amacrine cell of a monkey’s eye retina by our framework-based system. Only one of each three originally-sampled slices were used to make the input data sparse enough. Figure 6.5 compares a fragment of the amacrine cell
(a) Six input slices, each slice shown in a different color

(b) Tiling a pair of slices, no translation
(c) Tiling a pair of slices, with translation

(d) Reconstruction without translation
(e) Reconstruction with translation

Figure 6.1: Reconstruction of a synthetic example with and without contour translation
(a) The final partition and tiling  (b) Visualization of the surface in 3D

Figure 6.2: Reconstruction of a synthetic example from Fig.4.6

(a) The final partition and tiling  (b) Visualization of the surface in 3D

Figure 6.3: Reconstruction of another synthetic example
Figure 6.4: Reconstruction of the amacrine cell by the framework-based system

(a) By the straight-skeleton algorithm
(b) By the framework-based system

Figure 6.5: A fragment of the amacrine cell reconstruction
CHAPTER 6. EXPERIMENTAL RESULTS

reconstructed by the straight-skeleton algorithm (Figure 6.5(a)) and by the framework-based system (Figure 6.5(b)). While there is no difference in correspondence between the two reconstructions, there is a notable difference in the quality of tiling. The tiling in Figure 6.5(b) is smoother because the reconstruction in Figure 6.5(a) suffers from the “double cone” problem.

Figure 6.6 compares the reconstructed fragments of blood vessels from sparse input data. In this example, the straight-skeleton algorithm produces a poor surface in terms of correspondence (shown in Figure 6.6(a)) compared to the surface reconstructed by our framework-based system (Figure 6.6(b)). Figure 6.7 illustrates an intermediate as well as the final partition of a pair of slices taken from these data. Figure 6.7(a) shows the partition after Step 2. As one can notice, many small input contours lead to a large amount of empty cells in the intermediate partition, which do not exist in the final partition shown in Figure 6.7(b). These empty cells emerge during the first step of the partitioning when the overlay of the slices’ maps is calculated.

Figure 6.8 shows the reconstruction of human lungs and heart by the framework-based system. The inputs of both objects consist mostly of large, overlapping contours, so one cannot expect any significant improvement when the proposed framework-based method is used. Only slight improvement could be noticed in the trachea area of the lungs and in the major arteries of the heart, where there are small and not completely overlapping contours. Figures 6.8(a,b) show the reconstructed surface of the two organs. Figures 6.8(c,d) demonstrate the final partition and tiling of a pair of slices of the lungs and the heart, in two dimensions.

Table 6.1 shows some statistics of object reconstruction by our framework-based system. It provides the results for three models: the amacrine cell of the retina shown in Figure 6.4, the lungs shown in Figure 6.8(a), and the blood vessels shown in Figure 6.6(b). The results and measurements are given by framework phases. Each model is referred to by three columns in the table—“total”, “avg.”, and “max”. The “total” column contains the total measurement per model, whereas “avg.” and “max” columns contain measurements either for a single slice, or for a pair of slices, depending on the framework phase. The tests were executed on a PC with 2GHz Pentium processor and 512MB of RAM. All time measurements are given in seconds.
(a) By the straight-skeleton algorithm

(b) By the framework-based system

Figure 6.6: Reconstruction of blood vessels from sparse input

(a) Partition after Step 2 (26 cells)  
(b) The final partition (10 cells)

Figure 6.7: Reconstruction of blood vessels—partitioning of a pair of slices
(a) Lungs

(b) Heart

(c) A pair of slices of the lungs

(d) A pair of slices of the heart

Figure 6.8: Reconstruction of human organs
From the input data section of the table, one can observe that while the retina example contains more slices than the two other examples, blood vessels have more contours and more vertices in a slice, both in average and in the most “loaded” slice. It can be seen that the contour matching procedure and per-slice VDP calculations take much less time than the subsequent phases. Note that VDP calculation was described in this thesis as part of Step 1 of the partitioning. In practice, since VDP calculation is performed per slice rather than per a pair of slices, it is calculated before the rest of the partitioning. Step 1 of the partitioning, therefore, contains only the construction of cells by calculating the overlay of slices’ maps.

Measurements of the partitioning phase are given by steps. For each step it is shown how many cells existed at the end of it, and how much time it took. It can be seen in all the three models that Step 1 takes most of the partitioning time. As expected, the vessels example requires the largest average time for a pair of slices, since it contains the largest number of contours per slice. Overall, the partitioning procedure takes about 90% of the entire running time.

The translation and tiling procedure takes much less time than the partitioning, and the time required to build a VRML file is negligible. The overall reconstruction time is less than a minute and a half in the largest example (the retina). This performance is reasonable, though the existing overlap-based methods are significantly faster.

A significant improvement in the running time can be achieved by implementing the system as a multithreaded application, and running it on a multicore processor or a multiprocessor computer. Such implementation is possible because each pair of successive slices can be processed independently from the others by a different thread. For example, a thread which has finished processing a pair of slices can choose another pair, which has not yet been processed. Running such an application on \( n \) processors or an \( n \)-core processor will reduce running time almost by \( n \) (assuming that the total number of slices is large compared to \( n \)).
### Table 6.1: Testing results and measurements

<table>
<thead>
<tr>
<th>Model</th>
<th>Retina</th>
<th></th>
<th>Lungs</th>
<th></th>
<th>Vessels</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>avg.</td>
<td>max</td>
<td>total</td>
<td>avg.</td>
<td>max</td>
</tr>
</tbody>
</table>

#### Input data
- Slices: 96, 33, 16
- Contours: 230, 2.4, 6
- Vertices: 2801, 29, 79

#### Contour matching (on pairs)
- Time: 2.859, 0.030, 0.130
- Percent: 3.2%, 6.0%, 0.3%

#### VDP calculation (on slices)
- Time: 3.422, 0.036, 0.109
- Percent: 3.8%, 2.4%, 2.1%

#### Partitioning (on pairs)
- Cells (1): 548, 5.8, 18
- Time (1): 53.550, 0.564, 2.891
- Cells (2): 553, 5.8, 18
- Time (2): 1.749, 0.018, 0.422
- Cells (3): 548, 5.8, 18
- Time (3): 0.561, 0.006, 0.125
- Cells (4): 533, 5.6, 18
- Time (4): 1.125, 0.012, 0.125
- Cells (5): 254, 2.7, 7
- Time (5): 19.420, 0.204, 1.172
- Tot. time: 76.405, 0.804, 4.203
- Percent: 85.9%, 87.2%, 93.9%

#### Translation and tiling (on pairs)
- Time: 6.094, 0.064, 0.281
- Percent: 6.9%, 4.1%, 3.4%

#### VRML object construction
- Time: 0.218, 0.250, 0.125
- Percent: 0.2%, 0.3%, 0.3%

#### Output 3D model
- Time: 88.998, 79.734, 48.063
- Vertices: 10406, 11096, 5710
- Faces: 18965, 20615, 10786
Chapter 7

Conclusion

In this research we have proposed a framework for surface reconstruction from cross-sections obtained by sparse sampling of spatial objects. Unlike existing algorithms, our method attempts to separate the correspondence problem from the branching and tiling problems, while the reconstructed surface is guaranteed not to intersect itself for any possible correspondence graph.

The proposed framework consists of several phases that solve different reconstruction subtasks: contour correspondence, branching, and tiling. We have designed and implemented some algorithms as a case study of the framework in order to investigate and test the effectiveness of our approach. Though these algorithms can be further improved (as explained in Sections 3.3, 4.4, 4.5, and 5.4), the experimental results show that our approach is effective for reconstructing sparsely-sampled objects.

Surface reconstruction from sparse input data requires a high degree of separation between the contour correspondence problem and the tasks of branching and tiling. Such separation allows us to use different correspondence heuristics for different types of input data. This separation is not complete in the algorithms we provided as a case study, because the partitioning into cells somewhat restrains the correspondence heuristics. However, we believe that our approach to branching by splitting contours by a Voronoi diagram of polygonal sites, and to surface tiling by translating contours is effective, and should be used to further develop the idea of independent contour correspondence.

From the practical point of view, the framework-based system, implemented as part of
this thesis, showed improvement for certain types of sparse input data. For other examples there was no notable improvement relative to overlap-based reconstruction methods. The running times of the system are reasonable, but can be further reduced, as explained in Chapter 6. In addition, an application based on the idea of independent correspondence may allow the user to correct computer-proposed matchings by simply drag-and-dropping contours, thus, specifying the optimal translation vector. Such flexibility in user intervention is not possible in existing surface reconstruction methods.

Future research is needed to further develop the idea of independent correspondence based on the proposed framework. Improvement is required in handling data with complicated contour-nesting hierarchies, e.g., by independently matching and moving inner contours. In addition, it is interesting to try to improve the processing of branchings, such as those shown in Figures 4.12 and 4.13, by considering translated locations of multiple target contours rather than calculating their Voronoi diagram in given locations. Another interesting challenge is to improve the handling of artifacts caused by “height-suppression” during the translation and tiling phase, possibly by restoring the height of suppressed features in a postprocessing procedure.
Bibliography


ממסגרת לשחזור מ箏ה של אובייקטים דוגמנים בדילנות

חיבור על מחקר

לשם מילוי חלקי שלHDRישה ל الزمن תואר
מגיסטר למגעי במדעי המחשב

אליק זונסקי

הוגש לטכני הטכניון – מכון טכנולוגי לישראל
תנומי תשס”ז
חיפה
יולי 2007
המחבר נוגב בהנחייתו של פרופ' גֶּל בֶּרֶקֶת בפקולטה להנדסת מחשבים.

הנני אסף תודד להנעה של פרוֹפֶּר גֶּל בֶּרֶקֶת על הסכמה להנחת חמ窀 חנוכי להמחור.

חנני, אני מעוניין לאות את עורתי ו применения בכל שלבי המחקר.

אני מבקשום ממקים במקהל כל תכוף המחקר והחוויתם לשימור.

ול bütün זאת, לאשתי האהובה הנא על תרומתה להכיתת העבודה ועל סבלתה.

אני מודה לעכביים על התמיכה *************************************************************************************

אני מודד להנחייתו שלスーパיז תודד הහנפה אוניברסיטה בהשכיתון.
# תוכן עניינים

1. מבוא..............................................................................................................
   1.1 רקע........................................................................................................
   1.2 שבדות קדומות...................................................................................
   1.3 ארוגות התיה......................................................................................

2.傾ך הכלל......................................................................................................
   2.1 הגדרת הנדסת תביעה........................................................................
   2.2 מצבי התור.......................................................................................
   2.3 המסרת המפורט................................................................................
   2.4 סימונים............................................................................................
   2.5 האתריון החקר..............................................................................

3. והתאמת קונטרירם........................................................................................
   3.1 הגדרת התאמת פונקציה...................................................................
   3.2 תיירים קונטרירם...............................................................................
   3.3 דיני....................................................................................................

4. ת👋 לומת רבה.................................................................................................
   4.1 שיקולים כלל......................................................................................
   4.2 הציורים הורנלי של אטריים פולונליים...........................................
   4.3 תיאום אטריים...................................................................................
   4.4 להחבת לתאום שיאומי אטריים........................................................
   4.5 תسودת פתרונות..............................................................................
5. אינטרפולציה של משטח

5.1 הווה ויריעות

5.2 אינטרפולציה של שניים ו.ones חלקיים

5.3 גרף תלויות ואלאגוריתמים האנטרטרביס

5.4 המקרה הכללי לתלולית-ממדים

6. תוצאת ניסיון

7. סיכום

7.1 רשימה מקדימה

7.2 תקציר לפיו
רשימת אורות

8.............................................................................................................................. 2.1
9.............................................................................................................................. 2.2
10............................................................................................................................. 2.3
12.................................................................................................................................. 2.4
14.................................................................................................................................. 2.5
17.................................................................................................................................. 2.6
23.................................................................................................................................. 3.1
24.................................................................................................................................. 3.2
25.................................................................................................................................. 3.3
28.................................................................................................................................. 4.1
30.................................................................................................................................. 4.2
31.................................................................................................................................. 4.3
32.................................................................................................................................. 4.4
34.................................................................................................................................. 4.5
37.................................................................................................................................. 4.6
38.................................................................................................................................. 4.7
38.................................................................................................................................. 4.8
42.................................................................................................................................. 4.9
44.................................................................................................................................. 4.10
45.................................................................................................................................. 4.11
46.................................................................................................................................. 4.12
47.................................................................................................................................. 4.13
50.................................................................................................................................. 5.1
50.................................................................................................................................. 5.2
52.................................................................................................................................. 5.3
59.................................................................................................................................. 5.4
63.................................................................................................................................. 6.1
64.................................................................................................................................. 6.2
64.................................................................................................................................. 6.3
65.................................................................................................................................. 6.4
65.................................................................................................................................. 6.5
67.................................................................................................................................. 6.6
67.................................................................................................................................. 6.7
68.................................................................................................................................. 6.8
רשימה טבלאות

6.9 תוצאות מדידות

70
תקציר

בעית החזור של אובייקטים תלות-ממידיים ממק桉 תמני חתך כשווה חברה ייעור צדדים שונים של תחביר כבר白沙 לשלים. כל מפת תמיות הקימוט לפתרון בעיות פיתוח של תمواد של המתחברות [16,14], המתחברות במחלק סריקה לעיר מנומר, שיתוף רגתי, לכל ולמתואר. במבקש הניחים, מפותה שתשתיhabi מכונטרים (מיטרהוף) פוליגוליארי ממק桉 תמני ררחום הא/popper.

שיטות אלה בוחנות פוליגוליארי תלות-ממידי בא際וע מחברין יבר מכולת למעבדים הסמותרים. החיבור בישועת פיתוי של הוא שחר מחקרי חיו ישוב משטח מורכבים מספרים קור קולウィיט של פאתי (משולשים), גודל הפצל בידי התאサーテ לבול חוכל. זה על את, משיטואו של ילאן זה כל התחום של הש.Minute המתחברות יהודה ניב מיביות רמאתיות, לכל, שלח ייחודי את עמוד.


במקורה חכול. לכל השיתוא הקימוט כים, אשר פוזורח את ביית השחראי במקורה חכול, ההמאתה בקונטרוים

נמצאת על פיצול של הגאוגרפי שלד. בצומת חברון, שיש קונטרוים חזרה רכשו וייש חנות (ול

התקלח) בין הגאוגרפי שלד על המשורר. יעשו או מחזירים את קונטרוים בחירה,_lite משטח

ככל, היום סבירי לאשה מובאת שמיירѕך בכי שני דגניית (כלומר, ביני שיניים סמיניסיים) קוף

משפץ. לעומת, כאשר מקומאות הש Loving ילא, באישום חבטות להביך טциально לב

מהות, או איזו שליש במקורות רבי. במקומאות, קונטרוים אחרלי שלחוב עני משטח

על חלופי בקונטרוים קים המשו בכל השיתוא הקימוט: פ춤, שישואו אלא אינן מפרידות ביעיית ההמאתה לכל ענייה הريطון וההמשתעת, והשחראי מתבצעים בשתייה מברכת אהוב.
מה מתבצע הוא הריצוף של חלקים, או בכמה מקרים, שלב שבו נוכלقلق סמלים של לוגי, ואפשר לתפך את ניתן להעתקה באמצעות תומרת המ отлича לארבעת בסיסים. ביצירת האלגוריתמים הקבילים (ובמ TokenType מ Bison) 

המשולשים של הסטרצים, הם מתאימים למצב של חתכים אחרים ואלגוריתמים אחרים. קודה ושנים

או לשון שניים - תומכים

חיתוך

שניים

במוסדות

ול التعاون

מדגים

ركة

למציא

בtrer

שלבים

שלבים

שלים

שלים

שלים

שלים

שלים

שלים

שלים

שלים

שלים

שלים

שלים

שלים

שלים

שלים

שלים

shallower layers. The deeper layers represent more complex features:

• The initial ConvNet is: All ConvNet layers are set to zero.
• The last layers of the ConvNet are: The ConvNet will be used to determine the final output.

The network will be trained to learn the features from the input data. The features will be extracted from the output of each layer, and the weights of each layer will be updated to maximize the difference between the predicted and the actual output. This process is repeated for multiple iterations, until the network converges to a solution that minimizes the difference between the predicted and the actual output.

The network architecture is designed to have a modular and flexible structure, allowing it to be used for various tasks. The ConvNet is trained for different tasks, such as image classification, object detection, and natural language processing. The ConvNet is also used as a pre-trained model, which can be fine-tuned for specific tasks.

The ConvNet is a powerful tool for feature extraction and classification, and it has been successfully applied to a wide range of tasks. The ConvNet has been shown to achieve state-of-the-art performance in many vision and natural language processing tasks.
In Chapter 5, the process and the division of the stage is described, the division throughout which were created the tiles, one in each stage. The divisor was a graph on the "Graphs Ties". In the division created the tiles are the graph codes, the links and others. Each cell in a specific area in the filling tiles is named. The algorithm of iteration in each step is added more tiles in a final stage that the surface to guarantee iteratively in the algorithm is used, that is, an algorithm that is called directed graph on "Directed Graphs".

In the division created the tiles are the graph codes, the links and others. The algorithm that is used is the correct surface that is built geometrically).

The implementation of the sections is calculated on the basis of the graphics that are part of the C++ libraries CGAL [8] geometric and others. The software consists 8000 lines, Chapter 6 deals with examples of the software. Chapter 7 summarizes the comments that contain the research results and includes examples. Chapter 6 contains examples of the calculations. Chapter 7 contains comments and includes examples.