university of groningen

faculty of mathematics and natural sciences

Fast and Robust Extraction of Curve Skeletons from Voxel Models

Bachelor's thesis

2nd July 2012

Student: Christiaan Arnoldus

Primary supervisor: Prof. Dr Alexandru C. Telea

Secondary supervisor: Prof. Dr Michael Biehl

Abstract

There are many algorithms for computing curve skeletons. Most of them have their own notion of what a curve skeleton is and are tailored towards a specific purpose. This makes them inadequate for general use.

Recently, a formal, general definition of curve skeletons was introduced based on the principle of equal-length geodesics. This definition is computable and includes an importance measure to rank the importance of the branches of the curve skeleton. An algorithm computing curve skeletons conforming this definition is already available.

The purpose of this thesis is to put this algorithm to the test and provide background information on skeletonisation where necessary. The quality of the curve skeletons produced by this method is analysed, problems are noted and possible solutions presented. The conclusion is that, while the quality of the curve skeletons computed by this method is usually good, there are border cases in which there is room for improvement.

The performance is also measured and several ways to improve the performance are introduced. This includes a parallel implementation of the algorithm, as the algorithm is found to be very suitable for parallelisation.

Samenvatting

Er zijn vele algoritmen om curve-skeletten te berekenen. De meeste hebben hun eigen notie van wat een curve-skelet precies is en zijn ontworpen voor één specifieke toepassing. Dit maakt deze algoritmen ongeschikt voor algemeen gebruik.

Recentelijk is er een formele, algemene definitie voor curve-skeletten geïntroduceerd gebaseerd op het principe van geodeten met een gelijke lengte. Deze definitie is berekenbaar en omvat ook een metriek die kan worden gebruikt om het belang van de takken van het curve-skelet te rangschikken. Een algoritme om curve-skeletten te berekenen conform deze definitie is beschikbaar.

Het doel van deze scriptie is om dit algoritme te testen en, waar nodig, achtergrondinformatie over skeletonisatie te geven. De kwaliteit van de curve-skeletten geproduceerd door deze methode wordt geanalyseerd, problemen worden geïdentificeerd en mogelijke oplossingen worden aangedragen. De conclusie is dat, hoewel de kwaliteit van de curve-skeletten geproduceerd door deze methode over het algemeen goed is, er randgevallen zijn waar er ruimte is voor verbetering.

Naast dat worden de tijdsprestaties van deze methode gemeten en worden er verschillende manieren aangedragen om deze prestaties te verbeteren. Eén van deze manieren is een geparallelliseerde implementatie van het algoritme, aangezien dit algoritme goed parallelliseerbaar blijkt te zijn.

Contents

1 Introduction **3**
1.1 Curve skeleton properties . . . 3
1.2 Curve skeleton definition . . . 4
1.3 Algorithm for computing curve skeletons . . . 5
1.3.1 Feature transform . . . 5
1.3.2 Shortest-path set . . . 5
1.3.3 Counting boundary components . . . 5
1.3.4 Post-processing . . . 7
1.4 Importance measure . . . 7

2 Related work **9**
2.1 Computation methods . . . 9
2.2 Applications . . . 9

3 Quality analysis **10**
3.1 Quality requirements . . . 10
3.2 Observations, problems and solutions . . . 10
3.2.1 Unwanted branches . . . 10
3.2.2 Disconnected or thick skeletons . . . 13
3.2.3 Unwanted twists . . . 13

4 Performance analysis **15**
4.1 Complexity analysis . . . 15
4.2 Exploiting connectivity . . . 15
4.3 Parallelisation . . . 16
4.3.1 Functional decomposition . . . 16
4.3.2 Data decomposition . . . 16
4.4 Algorithmic enhancements . . . 18
4.4.1 Importance measure early termination . . . 18
4.4.2 Importance measure avoidance . . . 18

5 Discussion **22**
5.1 Skeletonisation of plant roots . . . 22
5.2 Future work . . . 22

Bibliography **24**

List of algorithms

1 Compute boundary graph of Ω . . . 6
2 Compute shortest-path set $\Gamma(p)$ . . . 6
3 Count boundary components of $p \in \Omega$ . . . 7
4 Importance measure of $p \in C(\Omega)$ . . . 19

List of figures

1.1 Bird volume and curve skeleton . . . 3
1.2 Geodesics of curve skeleton points . . . 4
1.3 Algorithm for curve skeleton computation . . . 7

3.1 Chicken volume and curve skeleton . . . 11
3.2 Rocker arm volume and curve skeleton . . . 12
3.3 *T. rex* volume and curve skeleton . . . 12
3.4 Noisy sea mine volume and curve skeleton . . . 14

4.1 Human hand volume and curve skeletons using optimisation techniques . . . 20

5.1 Plant roots volume acquired by scanning device . . . 23

List of tables

4.1 Volumes used for performance measurements . . . 17
4.2 Skeletonization time measurements using parallelisation . . . 17
4.3 Skeletonization time measurements using algorithmic enhancements . . . 18

Chapter 1

Introduction

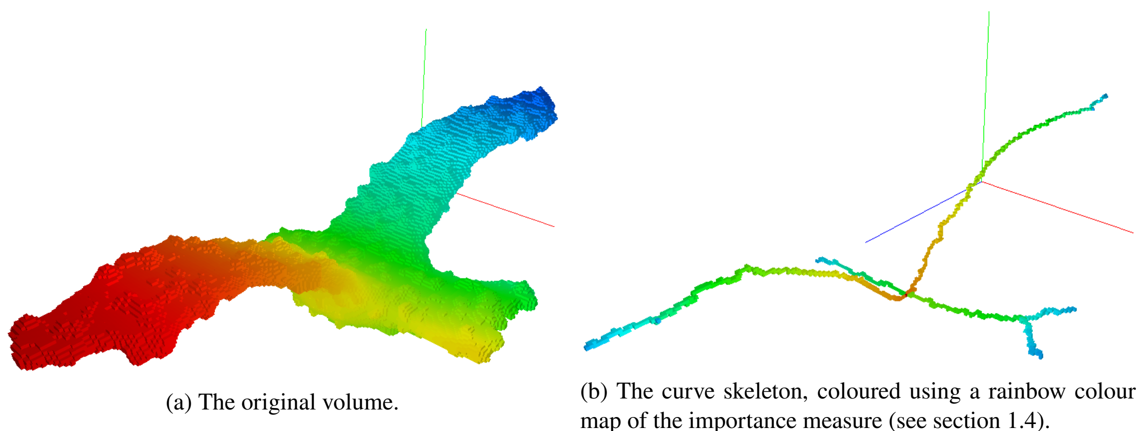
A skeleton is a simple and compact representation of a shape which preserves many of the topological and size characteristics of the original shape. Skeletons can be computed both for 2D and 3D shapes [Telea and van Wijk, 2002]. For 2D shapes, skeletons consist of a set of one-dimensional curves. For 3D shapes, we distinguish both *surface* and *curve skeletons*. Surface skeletons are 2D manifolds, whereas curve skeletons are 1D curves (much like the skeletons of 2D shapes). For the 3D case, shapes and their embedding space are typically represented as *volumes*. These are composed of *voxels* (analogue to pixels in two-dimensions graphics). Volumes usually contain empty voxels, referred to as *background voxels*, which are not considered part of the object that is represented.

This thesis mainly deals with curve skeletons, also known as centrelines, which are curves (that is, one-dimensional, tube-like objects) embedded in a three-dimensional space. Curve skeletons represent the symmetry axes of an object but cannot be used to recreate the original object without additional information [Telea and Vilanova, 2003]. An example of a curve skeleton is given in figure 1.1.

The purpose of this thesis is to study the algorithm for curve skeleton extraction described by Reniers et al. [2007], Reniers and Telea [2008], analyse the quality of the curve skeletons produced by this method (in chapter 3), analyse the time performance of the method (in chapter 4) and propose solutions to identified problems in the area of performance and computational robustness. Related work is mentioned in chapter 2 and possible future work is discussed in chapter 5.

1.1 Curve skeleton properties

Curve skeletons do not have a unanimously accepted formal definition [Reniers et al., 2007]. Rather, curve skeletons are defined by a set of properties which curve skeletons computed by a particular algorithm should ideally achieve. It is widely agreed that these properties include the following [Telea and Vilanova, 2003]:



(a) The original volume.

(b) The curve skeleton, coloured using a rainbow colour map of the importance measure (see section 1.4).

Figure 1.1: A volume of a bird and its (simplified) curve skeleton.

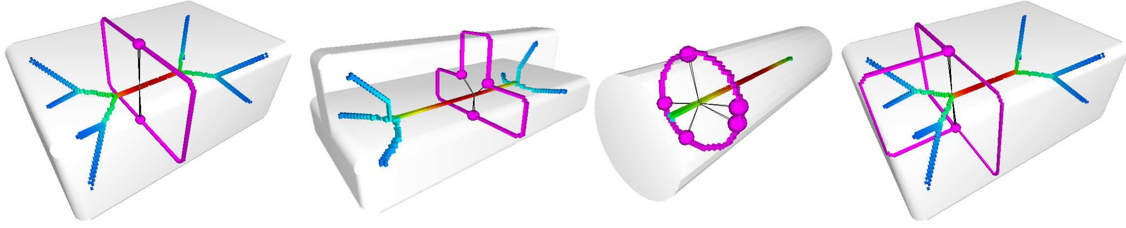


Figure 1.2: Geodesics of curve skeleton points, coloured magenta. Feature points are represented as spheres and connected to the curve skeleton point by lines. Taken from Reniers et al. [2007].

1. The curve skeleton of a connected volumetric object should itself be a connected set of voxels;
2. The curve skeleton of a volumetric object should be centred with respect to the object's boundary;
3. The curve skeleton of a volumetric object should be as thin is possible, ideally not thicker than one voxel;
4. The curve skeleton of a volumetric object should be insensitive to boundary noise. Small details on the object's surface should not create twists in the curve skeleton or introduce extra branches;
5. The curve skeleton of a volumetric should be fast to compute in a robust manner.

Here robustness refers to the invariance to boundary noise. This is necessary in all practical applications, as discretisation inevitably introduces noise [Reniers et al., 2007]. All these properties are in fact desirable for both surface and curve skeletons. Unfortunately, it may not be possible to meet all requirements at once [Telea and Vilanova, 2003]. Additional properties may be required depending on the application.

1.2 Curve skeleton definition

While there is no unanimously accepted formal definition of what constitutes a curve skeleton, Reniers and Telea [2008] include a definition which enforces several of the properties discussed in section 1.1. Before this definition can be presented a few auxiliary definitions have to be introduced.

A *feature point* of a point p of a 3D object Ω with a boundary $\delta\Omega$ is a point on $\delta\Omega$ which is at the minimal Euclidean distance from p to the boundary. A function which calculates feature points is called a *feature transform* $F(p)$:

$$F(p \in \Omega) = \left\{x \in \delta\Omega \mid \mathrm{dist}(p, x) = \min_{k \in \delta\Omega} \mathrm{dist}(p, k)\right\} \tag{1.1}$$

Between two of these feature points, one or more shortest paths exist over the boundary of the object. Such a shortest path over a curved surface is called a *geodesic*. The *shortest-path set* $\Gamma(p)$ contains all shortest paths between all pairs of feature points of a given skeleton point:

$$\Gamma(p) = \left\{\gamma \text{ is a path between } a, b \in F(p) \mid \gamma \text{ is a shortest path between } a, b\right\} \tag{1.2}$$

The union of the shortest-path set forms a curve on the object's boundary. This curve may form a closed loop around the object. If the object has no tunnels (see section 3.2.1), then such a closed loop divides the object's boundary in two or more *boundary components* or *compact boundary pieces* [Reniers and Telea, 2008]. This is possible when the point p has two feature points with two equally long shortest paths between them or when p has more than two geodesics. Multiple possibilities are illustrated in figure 1.2. In these cases the point is a curve skeleton point. This leads to the following definition of a curve skeleton of Ω:

$$C(\Omega) = \left\{p \in \Omega \mid \bigcup_{\gamma \in \Gamma(p)} \gamma \text{ divides } \delta\Omega \text{ in two or more components}\right\} \tag{1.3}$$

The definition described here enforces centredness. Since all curve skeleton points have at least two feature points, they are part of a surface centred in the object (the surface skeleton). When a curve skeleton

point has two geodesics, it is also centred within the surface skeleton, because the two geodesics necessarily have the same length [Reniers and Telea, 2008]. Justifying the definition in the non-generic cases is harder. It has not yet been formally proven, but is supported by empirical evidence [Reniers et al., 2007].

1.3 Algorithm for computing curve skeletons

The algorithm used for computing curve skeletons studied in this thesis is the one described by Reniers and Telea [2008], which is similar to one the one described by Reniers et al. [2007]. This algorithm consists of several steps which are discussed below. The algorithm follows naturally from the definition given in section 1.2, although some problems must be addressed when applying it in discrete space.

1.3.1 Feature transform

In order to calculate the curve skeleton of a volume Ω, first all feature points of the voxels of Ω must be computed, as defined by equation (1.1). Several algorithms for computing feature points exist, whose differences may be quite subtle, as discussed by Reniers and Telea [2007]. The algorithm used here is the one introduced by Danielsson in 1980, because a working implementation was available.

In a continuous space, each curve skeleton point has at least two feature points. When it has two feature points, the geodesics between them are in theory of exact the same length. In discrete space this is not necessarily true. When an object has a thickness of an even number of voxels no voxel in the centre has two feature points. Also, because the length of the geodesics cannot be computed exactly in discrete space, it is possible that only one shortest path is found between two feature voxels while two were expected [Reniers et al., 2007].

These problems are combat as follows. After calculating the feature voxels of all object voxels, the feature sets of each voxel is extended by including all feature voxels of neighbouring voxels as well. The function computing the extended feature sets of each voxel is called the *extended feature transform* (here $F_{\text{ext}}(p)$). By relaxing the definition of a feature point many centrevoxels and shortest paths can be found were otherwise none would have been found.

1.3.2 Shortest-path set

In order to compute the geodesics between the feature voxels of a particular object voxel, the ubiquitous A* algorithm is used on the boundary graph of the object, using Euclidean distance as heuristic. The A* algorithm, introduced by Hart, Nilsson and Raphael in 1968, computes the shortest path between two vertices in a weighted graph. So to apply the A* algorithm on a volumetric object, the object's boundary needs to be represented as a graph, here referred to as the *boundary graph*. This is done by selecting all object voxels which have at least one neighbouring background voxel as vertices of the graph. Edges are constructed between neighbouring boundary voxels. The full algorithm is given by algorithm 1. The weight of the edges is determined by whether the voxels are face, edge or vertex neighbours, as this determines the distance between the centres of the voxels. Because object voxels may have multiple feature voxels in common, it is useful to cache geodesics, in order to prevent their re-computation.

All geodesics form the shortest-path set, as defined by equation (1.2). The full algorithm is given in algorithm 2.

1.3.3 Counting boundary components

In order to determine whether a voxel belongs to the curve skeleton or not, as defined by equation (1.3), the number of boundary components incident to the geodesic curve around the object needs to be counted. This is done by gathering all voxels neighbouring the geodesic curve and determining to how many connected components in the boundary graph these voxels belong. When a geodesic curve has more than two neighbouring boundary components, the voxel under consideration is a curve skeleton voxel.

The above naive approach has a problem related to the discrete nature of volumes. Because the curve around the object is one voxel thick and voxels have up to 26 neighbours, it is likely for an edge to go through the curve, making it impossible to distinguish multiple connected components. This problem is solved by widening the geodesic curve in both directions by a certain amount, called the *dilation distance*.

Algorithm 1 Compute boundary graph of Ω

Require: voxel set Ω
Ensure: $\delta\Omega$ is the boundary graph of Ω

```
vertices(δΩ) ← ∅
for all x ∈ Ω do
    if x neighbours a background voxel then
        insert x in vertices(δΩ)
    end if
end for
edges(δΩ) ← ∅
for all x ∈ vertices(δΩ) do
    for all k ∈ neighbours(x) do
        if k ∈ vertices(δΩ) then
            insert (x, k) in edges(δΩ)
        end if
    end for
end for
return δΩ
```

Algorithm 2 Compute shortest-path set $\Gamma(p)$

Require: $p \in \Omega$
Ensure: $\Gamma(p)$ contains all shortest paths between $a, b \in F_{\mathrm{ext}}(p)$

```
Γ(p) ← ∅
for all a ∈ F_ext(p) do
    for all b ∈ F_ext(p) \ {a} do
        γ ← shortestpath(a, b)
        insert γ in Γ(p)
    end for
end for
return Γ(p)
```

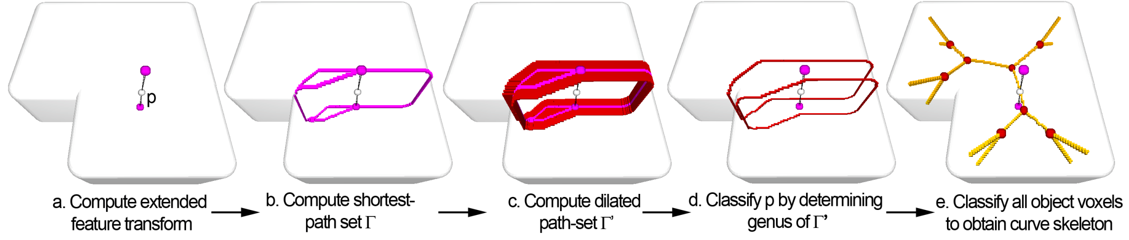


Figure 1.3: Visualization of the algorithm for curve skeleton computation. Taken from Reniers and Telea [2008].

A reasonable value for the dilation distance is 5.0 [Reniers and Telea, 2008], although the ideal value depends on the object. Both a too low and too high value can have severe effects on the quality of the resulting curve skeleton, as discussed in section 3.2. This dilated curve is called the *dilated-path set* or *surface band*.

The algorithm for computing boundary component count is included in algorithm 3, where the voxels of the dilated-path set D can be found using Dijkstra's algorithm. The full algorithm for curve skeleton computation is illustrated in figure 1.3.

Algorithm 3 Count boundary components of $p \in \Omega$

Require: $p \in \Omega$, dilation distance
Ensure: n is the number of boundary components of p

```
n ← 0
V ← ∅ {to contain visited voxels}
D(p) ← {x ∈ δΩ | ∃_{k∈Γ(p)} (dist(x, k) ≤ dilation distance)}
E ← ⋃_{k∈D(p)} (neighbours(k) \ D(p)) {neighbours of surface band}
for all x ∈ E do
  if x ∉ V then
    n ← n + 1
    Q ← empty queue
    insert x in Q
    while |Q| ≠ 0 do
      a ← remove from Q
      for all b ∈ neighbours(a) do
        if b ∉ V and b ∉ E then
          insert b in Q
        end if
      end for
    end while
  end if
end for
return n
```

1.3.4 Post-processing

The algorithm discussed in this section yields the curve skeleton without any post-processing [Reniers and Telea, 2008]. It may nevertheless be desirable to use a post-processing step, for example to remove noise, as discussed in section 1.4. Other possibilities are discussed in chapter 3.

1.4 Importance measure

In order to rank branches of the skeleton, it is useful to be able to give each voxel of the curve skeleton a certain importance value. An application of this importance value is removing noise from a curve skeleton,

as discussed below. A function which calculates such an importance value for each voxel is called an *importance measure*. Reniers et al. [2007], Reniers and Telea [2008] discuss an importance measure based on the area of boundary components, called the *collapse measure* [Reniers et al., 2007] or just importance measure [Reniers and Telea, 2008]. The formula used in this thesis for the importance measure is:

$$\rho(p \in \Omega) = \sqrt{\frac{2}{|\delta\Omega|} \sum_{1 \leq i < k} |C_i(p)|} \tag{1.4}$$

Where k is the number of boundary components of p and $C_i(p)$ is the ith boundary component. The importance measure is normalized to the range $[0, 1]$. The algorithm for computing the importance measure, with some adjustments, is algorithm 4.

This importance measure is monotonic, that is, it decreases when going outward from the centre of the curve skeleton. Again, this is hard to prove, but supported by empirical evidence. Monotonicity ensures that no disconnections are introduced when thresholding to produce a *simplified curve skeleton*. A simplified curve skeleton is a curve skeleton with (most of) its noise removed. This is done by removing all voxels with an importance value lower than a certain value τ:

$$C_\tau(\Omega) = \left\{ p \in C(\Omega) \mid \rho(p) \geq \tau \right\} \tag{1.5}$$

The image of the curve skeleton in figure 1.1 is a simplified curve skeleton. Its colour is a representation of the importance measure as a rainbow colour map, which is referred to as *importance map* henceforth.

Note that, while the importance measure is defined for all object voxels, it makes little sense to compute it for non–curve skeleton voxels, since it will always be zero in that case.

Summary

This thesis deals with the computation of curve skeletons. Curve skeletons are curve-like objects which preserve many properties of the volumetric object from which they were created. A curve skeleton should be connected, centred, thin and fast to compute. To that end this thesis studies an algorithm for curve skeleton computation and tries to improve its performance. The definition of curve skeletons used by this algorithm is that each curve skeleton voxel should have a surface band around its feature voxels, which divide the original object in multiple distinct components. An importance measure is available, which can be used to filter noise from a curve skeleton. Noise is inevitable when working with discrete imaginary.

Chapter 2

Related work

The main purpose of this chapter is to give an insight in the literature on curve skeletons computation available right now. A short introduction on applications of curve skeletons is also given, to give an idea why having a fast, robust and mathematically valid way to compute curve skeletons is important.

2.1 Computation methods

There are many methods of computing curve skeletons (and surface skeletons), including thinning methods, geometric methods, distance transform–based methods and so on [Telea and Vilanova, 2003, Dey and Sun, 2006, Reniers et al., 2007]. While all these methods produce curve skeletons with a set of desirable properties, none of them are completely satisfactory [Dey and Sun, 2006].

Most these methods have their own notion of what a curve skeleton is. These definitions are not given explicitly, but are implied by the algorithm. An explicit, analytical definition is given by Dey and Sun [2006], Reniers et al. [2007] and is the one discussed in section 1.2. This definition not only describes formally what a curve skeleton really is, but it is also computable and gives an importance metric. Therefore, this is the definition of choice in this thesis. This definition has some problems with speed and robustness, but those are dealt with in chapters 3 and 4.

2.2 Applications

Curve skeletons have many application in geometric modelling, computer graphics, visualisation and computer vision [Dey and Sun, 2006]. Several applications of curve skeletons (amongst other things) are discussed by Cornea et al. [2007].

One such application is in *virtual navigation*. Since curve skeletons are centred, they can be used as a collision-free path through an object or scene. Skeletons are also used in animation of 3D models. While traditionally the skeletons are specified by an animator, it would be very useful if those skeleton could be computed automatically from the 3D models instead. Yet another application for curve skeletons is comparing objects, by comparing the curve skeletons instead, which is easier to do because of the reduced dimensionality. This can be applied for *matching*, finding objects similar to a certain object or *registration*, that is, aligning objects which are known to be similar.

Curve skeleton can also be used in *segmentation*, the act of decomposing an object in meaningful components, which is the main aim of Reniers and Telea [2008].

Chapter 3

Quality analysis

In this chapter several quality requirements of curve skeletons are introduced and the algorithm discussed in section 1.3 is reviewed regarding these requirements. Observations and problems are presented, as well as possible solutions to these problems.

3.1 Quality requirements

Section 1.1 introduces several desirable properties a curve skeleton should possess. Here these properties form the basis for several quality requirements which can be used to classify the quality of curve skeletons. These requirements are ordered based on how easy it would be for a post-processing step to improve the quality of a curve skeleton with respect to a requirement, where the first one is the easiest and the last one is the hardest. This is not an absolute ordering; for some skeletons it may be easier to improve the quality than for others.

The requirements are as follows. A curve skeleton should be:

1. Free of low-importance branches;
2. Free of disconnections;
3. One voxel thick;
4. Free of noise-induced twists.

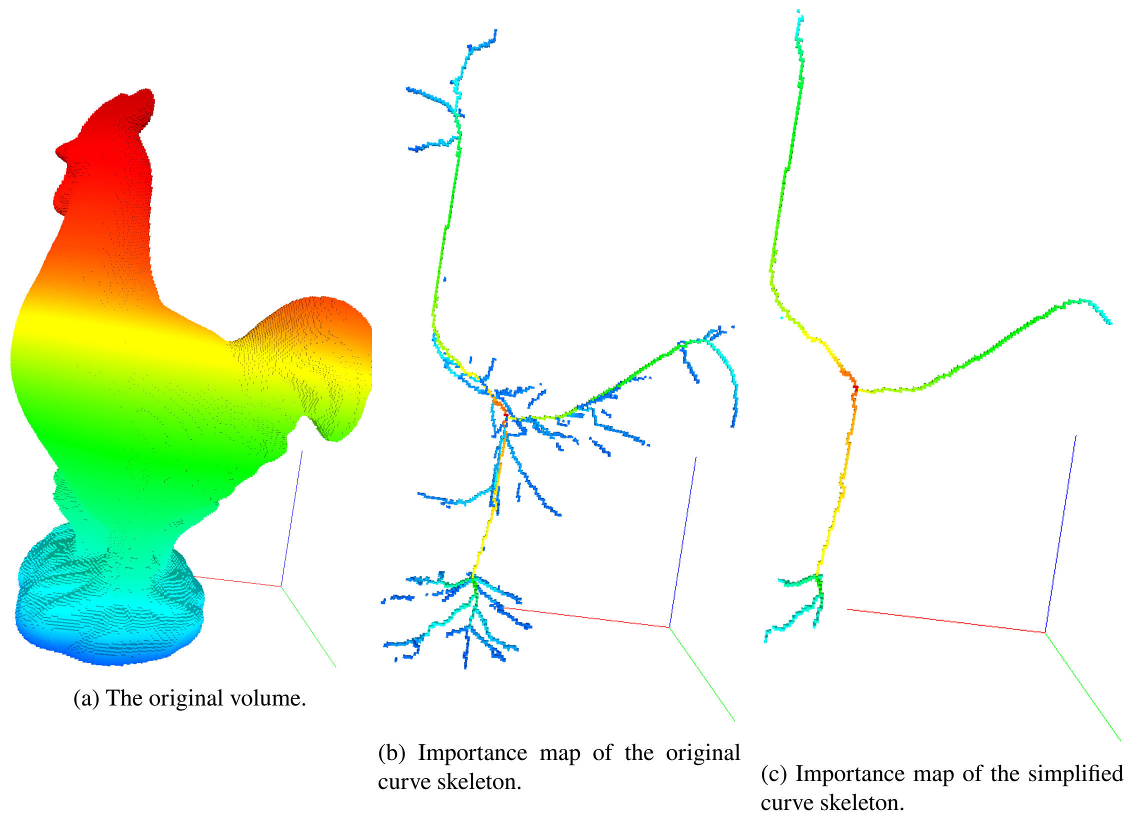
Low-importance branches are easy to eliminate by thresholding using an importance measure, as discussed in sections 1.4 and 3.2.1. Disconnections are harder to eliminate, but it could be done using an algorithm which identifies the endpoints of the curve skeleton and then introduces a connection between endpoints which are close to each other. Making a curve skeleton one voxel thick is not trivial, but it is possible using a thinning algorithm [Telea and Vilanova, 2003]. Removing noise-induced twists, discussed in section 3.2.3, is probably the hardest problem to solve.

3.2 Observations, problems and solutions

In section 3.1 several possible problems and solutions regarding curve skeleton quality are introduced. In this section curve skeletons of several volumes are studied in order to find out whether these problems actually occur in practice and, if they do occur, whether the proposed solutions are satisfying.

3.2.1 Unwanted branches

Figure 3.1 shows a volume whose curve skeleton includes a lot of unwanted branches, caused by minute noise on the object's boundary. Thresholding the curve skeleton using the importance measure nicely removes any unwanted branches with no chance of introducing disconnections (as discussed in section 1.4).



(a) The original volume.

(b) Importance map of the original curve skeleton.

(c) Importance map of the simplified curve skeleton.

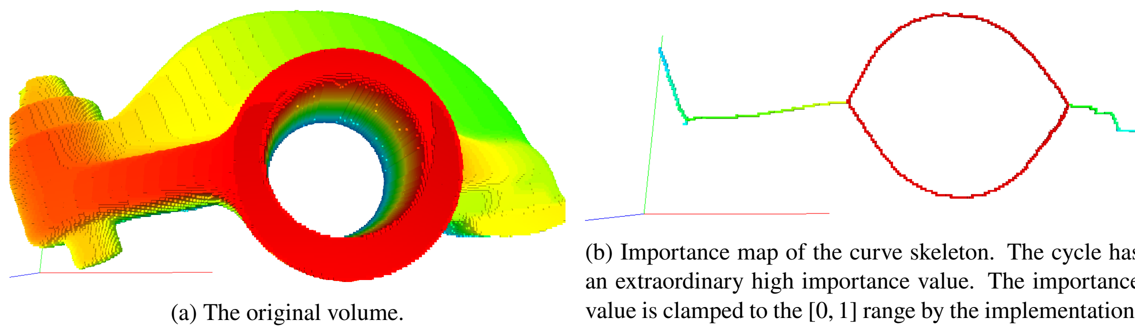
Figure 3.1: A volume of a chicken and its curve skeleton, as well as its simplified curve skeleton.

At first sight the problem of unwanted, noise-induced branches seems easy to solve using the importance measure, but there is a problem when there are cycles present in the curve skeleton. This problem is demonstrated in figure 3.2. The importance measure assumes the combined geodesics of a curve skeleton voxel divide the volume boundary in two or more components. This assumption is problematic if the voxel is part of a curve skeleton cycle, as the components on both sides of the combined geodesics ultimately represent the same surface (which is the complete boundary of the object).

If the importance measure is calculated by naively assuming that all neighbouring components are distinct, the combined area of the neighbouring components is larger than the total surface area of the object (twice the total surface area of the object, when the number of neighbouring components is two). This results in an importance value which is much larger than desired. Alternatively, if the components are treated as being the same component, the number of neighbouring components may be only one. This implies that the voxel is not part of the curve skeleton, as defined by equation (1.3), which is even less satisfactory.

The presence of a cycle in figure 3.2 is not a major problem, as the cycle is the most important part of the skeleton, so it need not be simplified by thresholding. A different situation is given in figure 3.3, where the cycle is located in a less important part of the object, the left-front claw of the dinosaur. As a result, the centre of the object is no longer classified as the sole most important part of the object, violating the monotonicity of the importance measure. When the curve skeleton is simplified by thresholding, the voxels near the cycle remain while the ones slightly father away are removed, introducing disconnections. In this case, the disconnected pieces of the curve skeleton are small and located in the periphery of the object, which means they could be removed by a post-processing filter without damaging the more important parts of the curve skeleton. This may not be possible on more complex objects.

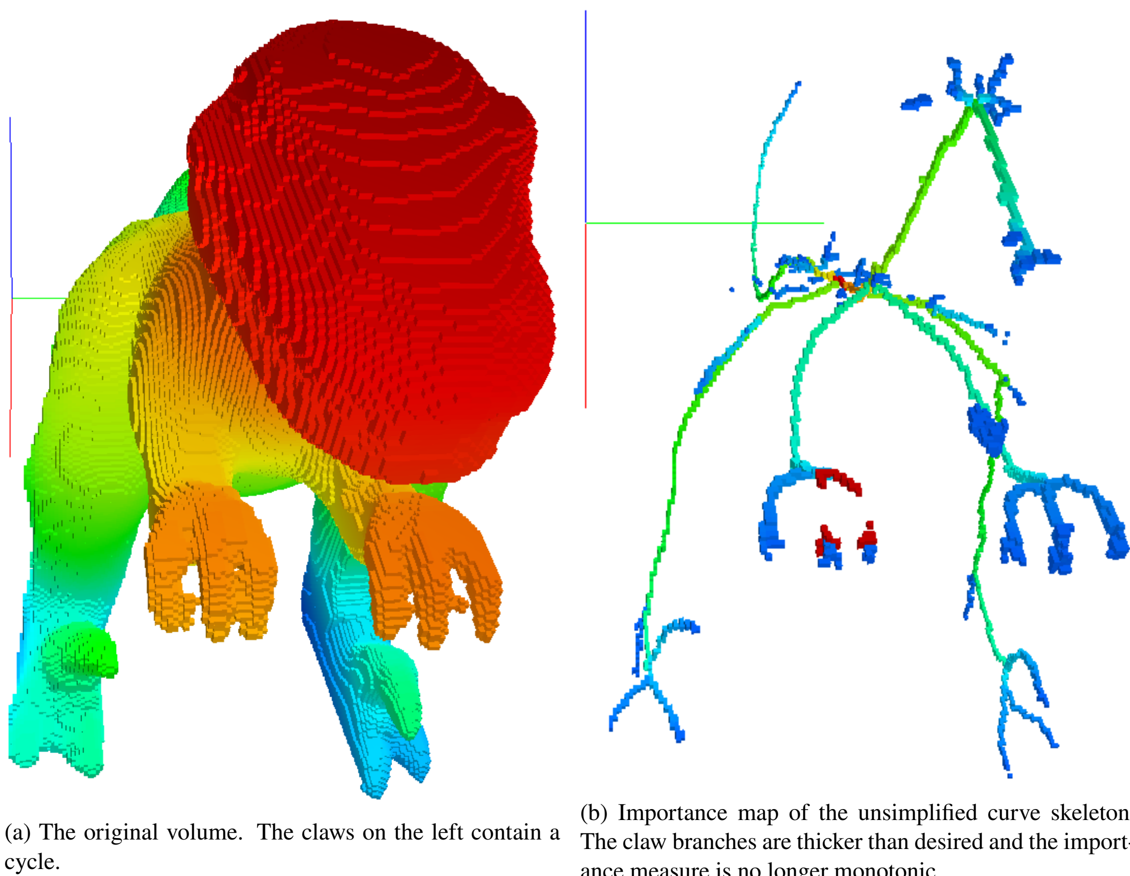
Reniers and Telea [2008] include a discussion about computing curve skeletons of object containing cycles or tunnels, referred to as objects with genus ≥ 1 (objects without cycles are referred to as genus 0). A general solution to this problem is not yet known, the best solution right now seems to be including large cycles in the curve skeleton while elimination small ones.



(a) The original volume.

(b) Importance map of the curve skeleton. The cycle has an extraordinary high importance value. The importance value is clamped to the [0, 1] range by the implementation.

Figure 3.2: A volume of a rocker arm and its curve skeleton.



(a) The original volume. The claws on the left contain a cycle.

(b) Importance map of the unsimplified curve skeleton. The claw branches are thicker than desired and the importance measure is no longer monotonic.

Figure 3.3: A volume of a *Tyrannosaurus rex* and its curve skeleton.

3.2.2 Disconnected or thick skeletons

The algorithm described in section 1.3 usually does not produce curve skeletons which are disconnected or too thick, but those problems can occur due to discretisation issues. When the original volume is thin the algorithms may not be able to find certain feature points or geodesics because the distances between voxels are relatively large (relative to the dimensions of the object). This problem is visible in figure 3.3, in the area around the left claw.

Another problem with the exact opposite symptom can occur when the object thickness is in the same order of magnitude as the dilation distance. In this case, dilating the shortest-path set will always produce a surface band which encloses the object, dividing the object in two components and classifying the voxel as curve skeleton voxel, even though it may not be centred. As a result the resulting curve skeleton may be too thick, which is also evident in figure 3.3 (right claw). Another problem occurs when the surface band covers the tip of the original object, which causes the voxel not to be classified as curve skeleton voxel, even though it may actually *be* centred.

These issues may not be serious problems, if the original object does not contain thin regions. If it does then these regions are probably unimportant outer regions and could be removed from the curve skeleton by simplification or removing small, disconnected curve skeleton pieces. If the thin parts of the object *are* important (or if the whole object is composed of thin structures, for example plant roots, see chapter 5), the best solution would be increasing the resolution of the original object, either by using a more sophisticated scanning technique or algorithmic up-scaling.

When a curve skeleton is too thick it may also be fruitful to use a thinning algorithm. Thinning algorithms are a not very good at producing curve skeletons on their own, as they do not guarantee centredness, but this is not a problem when applying them to curve skeletons since they are already centred [Telea and Vilanova, 2003].

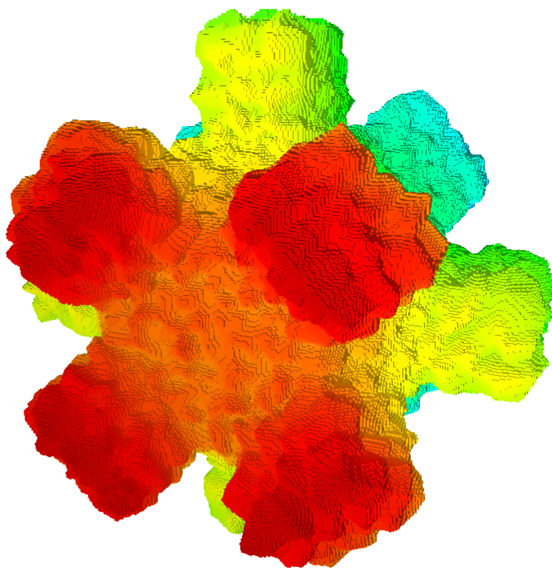
3.2.3 Unwanted twists

Twists may be introduced in the curve skeleton if noise is not uniformly distributed over the surface of the object, as demonstrated in figure 3.4. These twists cannot be removed by thresholding as they are deformations of high-importance parts of the curve skeleton, rather than additional small, low-importance branches, which are assumed to be the major noise-induced artefacts by the skeleton simplifications methods which only remove (low-importance) voxels. Eliminating twists would entail pre-processing the volume to remove noise or maybe employing heuristics to prevent sudden directional changes in the curve skeleton. Neither technique would be trivial to implement.

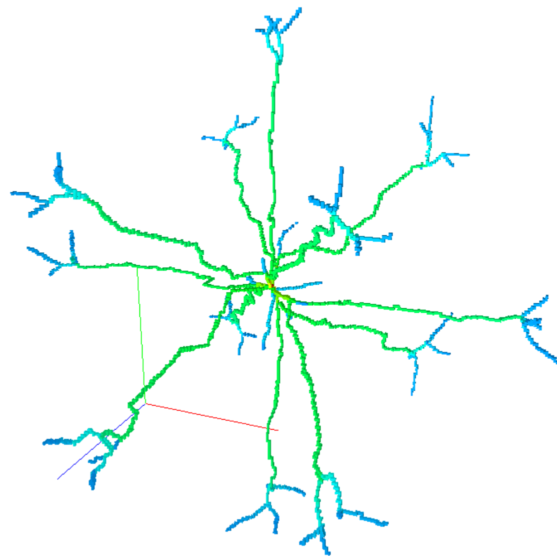
Summary

There are several cases in which the quality of the curve skeletons produced by the algorithm discussed in section 1.3 can be suboptimal. In some cases, like the unwanted branches, the problem is present in most input volumes, but can easily be remedied by post-processing the curve skeleton (in this case by skeleton simplification). Other problems are rare but may be hard to solve, like the presence of unwanted twists in the curve skeleton. All in all, most quality problems are the result of noisy or low-resolution input volumes.

Some solutions to the problems are suggested in this chapter, but the specifics will not be pursued further. Chapter 4 *will* introduce specific ways to improve the algorithm.



(a) The original volume.



(b) Importance map of the curve skeleton. The skeleton contains twists induced by noise on the original volume's boundary.

Figure 3.4: A volume of a sea mine with a noisy boundary and its curve skeleton.

Chapter 4

Performance analysis

Real-world 3D volumes usually contain a large amount of data. It is therefore desirable that the algorithms applied to these data are as efficient as possible. In this chapter the performance of the algorithm introduced in section 1.3 is analysed and performance improvements are suggested, both lossless parallelisation and lossy algorithmic enhancements.

4.1 Complexity analysis

As discussed in section 1.3 the algorithm used here consists of several steps. These steps all have their own time complexity characteristics. Computing the feature transform can be done in $\mathcal{O}(n)$, where $n = |\Omega|$. The time complexity of computing shortest-path set of a single object voxel using the A* algorithm is $\mathcal{O}(b \log b)$, where $b = |\delta\Omega|$. b is usually far smaller than n and, for a single object voxel, the algorithm does not need to visit all boundary voxels, only those part of its shortest-path set (which includes its feature voxels) [Reniers et al., 2007].

Counting the number of neighbouring components of an object voxel has a time complexity of $\mathcal{O}(b)$, so counting the number of boundary components for all voxels therefore has a time complexity of $\mathcal{O}(b \times n)$. But again, the algorithm rarely (if ever) needs to visit all boundary voxels, only those neighbouring surface band voxels. The actual time complexity will therefore be closer to $\mathcal{O}(n)$.

Computing the importance measure of a single object voxel also has a time complexity of $\mathcal{O}(b)$, but this time the algorithm *will* need to visit all boundary voxels, as the size of such a component needs to be computed. This step needs to visit all curve skeleton voxels, resulting in a worst-case time complexity of $\mathcal{O}(b \times c)$ for the importance measure, where $c = |C(\Omega)|$. When a significant number of object voxels belong to the curve skeleton, this step will dominate the algorithm.

Apparently, computing the importance measure is the most expensive step in this process. While computing the importance measure is, in principle, optional, one usually wants to compute it in order to remove the inevitable noise. But since it is the most expensive step, it would be nice if its computation could be sped up (discussed in section 4.4.1) or avoided (discussed in section 4.4.2) by enhancing the algorithm.

4.2 Exploiting connectivity

Reniers and Telea [2008] discuss a way to speed up curve skeleton computation. Instead of processing all object voxels (possibly in parallel), the process is stopped when a single curve skeleton voxel is encountered, referred to as the *seed* voxel. From that moment on only voxels neighbouring voxels known to be curve skeleton voxels are considered. This works because the curve skeleton is a connected structure (see section 1.1).

This approach does have a few minor problems. If the volume contains multiple objects only one curve skeleton is found. Also, while the curve skeleton is supposed to be a connected structure, it is possible that this is not correctly detected (as discussed in section 3.2.2). This may causes large parts of the curve skeleton to be missing if the process if halted if no neighbour of the known curve skeleton voxels turns out to also be a curve skeleton voxel.

The main reason that this approach was not used here is that is interferes with the ability to parallelise the algorithm (see section 4.3), because it creates a dependency between the computation of individual

curve skeleton voxels (although there would still be a possibility to parallelise the algorithm, for example by processing all neighbours of a curve skeleton voxel in parallel). The average speed-up achieved by this method is about eight times [Reniers and Telea, 2008]. This suggests a parallel algorithm could beat this method if enough processors are available.

4.3 Parallelisation

One way of improving the performance of an algorithm is attempting to utilise multiple processing units, which are common in modern computers, to do computations in parallel which would otherwise have to be done sequentially. This paradigm is called *parallel computing* and the act of making an algorithm suitable for parallel execution is referred to as *parallelisation*. An introduction to parallelisation basics and terminology is given by Barney [2012].

When parallelising an algorithm, the main problem is deciding how to decompose the problem into smaller pieces, which can then be processed in parallel. There are many ways to do this and several of them are applicable to the algorithm at hand. These decomposition methods can be grouped in two, *functional decomposition*, discussed in section 4.3.1 and *data decomposition*, discussed in section 4.3.2.

4.3.1 Functional decomposition

With functional decomposition the algorithm is decomposed into several tasks which can be executed in parallel. In the case of the curve skeletonisation algorithm these tasks could be as described in section 1.3.

Functional decomposition can be implemented using a *pipeline*, which entails that each thread is associated with a specific task. Each voxel (or a small set of voxels) is handed to the first thread in the pipeline, which hands it to the next thread when it finishes processing, accepts new input, and so on. This way tasks can be executed in parallel, even if the input is produced in a sequential manner and order needs to be preserved. An example would be video processing, where frames need to be output in the same order as they were received. The pipeline approach works best when each task takes a similar amount of time, which may not be the case in the skeletonisation case (see section 4.1).

Another way of implementing functional decomposition is by using a *task queue*. In this case threads are not associated with a specific task, but instead take a task (and its input) from the task queue whenever they finished their last task (or just started). This prevents bottlenecks from occurring if one task is significantly more computationally intensive than the others. Tasks can be added to the task queue whenever input becomes available, but the order of the output is not guaranteed. This is usable in an internet server, where the order in which request are processed is not important.

Both of these approaches require synchronisation between threads, which is at best kept minimized since it may cause threads to stall and as a result performance to degrade. Fortunately, in the skeletonisation case all data is available upfront, so functional decomposition is not strictly necessary.

4.3.2 Data decomposition

With data decomposition the data, rather than the algorithm, is decomposed in manageable pieces which can then be processed by several threads in parallel. As mentioned in section 4.2, there is no dependency between voxels when it comes their possible classification as curve skeleton voxel. This suggests the algorithm can be parallelised at will without any communication between threads, making it an *embarrassingly parallel* problem (since it is so easy to parallelise). The performance should scale almost linearly with the number of hardware threads.

This approach is implemented and measured here, for the reason mentioned in the previous paragraph. Each thread implements almost the whole pipeline for a single voxel. While most steps of the algorithm can be executed independently for each voxel, this is not the case for the feature transform. The feature transform is only a small part of the complete process and is therefore left unparallelised. If necessary, better algorithms for computing feature points are available, for example the one introduced by Thanh-Tung Cao et al. [2010], which utilises GPU hardware (more on utilising GPU hardware in chapter 5).

Decomposing data to be distributed of multiple processing threads can be done in two ways. Statically, where the data is divided upfront or dynamically, where data is distributed as threads become available. The dynamic method is preferred when the duration of computation may differ greatly between equally sized

Table 4.1: Volumes used for performance measurements

Volume name	Resolution	Object voxels	Boundary voxels	Skeleton voxels
anvilnoisy	92×127×286	1374088	92760	5404
catpose0	286×135×52	326669	35242	1113
cylinder2boxmultiple	83×200×286	1406426	92585	792
chicken_high	196×129×286	1632661	86518	2504
dinoflat	74×250×285	501422	52882	1422
human_hand	286×121×201	664349	56798	1464
noisybird	155×83×286	177737	34395	1048
pig_highres	161×101×286	1017395	67628	1897
rockerarm	286×147×87	964197	91795	1233
smoothx	80×144×286	1340283	91740	1051
tap_threads	252×71×286	1833517	131106	2772
t-rex_high	286×130×242	1235427	87146	2590

Table 4.2: Skeletonization time measurements using parallelisation

Volume name	FT (ms)	1 thread (ms)	4 threads (ms)	Speed-up	Memory (MB)
anvilnoisy	8693	887301	217716	4.08	2014
catpose0	1977	124792	30437	4.10	561
cylinder2boxmultiple	8269	815037	214574	3.80	2682
chicken_high	10831	565191	141138	4.00	2172
dinoflat	4177	193371	47572	4.06	849
human_hand	4933	262487	63286	4.15	1057
noisybird	1668	100606	25125	4.00	385
pig_highres	6507	367559	89477	4.11	1487
rockerarm	5678	612829	155472	3.94	1665
smoothx	8108	516581	125234	4.12	2104
tap_threads	11536	997645	248266	4.02	2887
t-rex_high	8543	494954	119339	4.15	1780

data units and it is infeasible to predict which are computationally expensive and which are not, but the static method incurs less overhead. The static method is used here, as the results show it is adequate. Static decomposition can, again, be done in multiple ways. Block-wise, where data is is divided in contiguous blocks or cyclic, where every ith data piece is processed by the ith thread. The latter method is used here under the assumption that it would result in a better distribution of the workload.

The parallelisation as outlined in this section is implemented in C++ using an existing framework for skeleton visualization[1]. More explicitly, the implementation is as follows:

1. An initialisation step is done on a single thread, this includes constructing the boundary graph and computing the feature transform;
2. A number (equal to the number of hardware threads or specified by the user) of processing threads are created. Each thread implements the whole remaining skeletonisation pipeline;
3. Each thread processes a number of voxels. The threads know which voxels to process, as the decomposition happens in a regular manner (each ith voxel is processed by the ith thread).

Time measurements were done using the volumes listed in table 4.1 on a computer with an AMD Phenom™ 9650 Quad-Core Processor and 4 GB of RAM. The time measurements are included in table 4.2. Since the program scales almost perfectly, no further adjustments were made to the method of parallelisation.

[1] Skeleton Sandbox by Dennie Reniers (`http://www.cs.rug.nl/svcg/Shapes/Skel3D`)

Table 4.3: Skeletonization time measurements using algorithmic enhancements

	Early termination		Avoidance	
Volume name	Time (ms)	Speed-up	Time (ms)	Speed-up
anvilnoisy	118496	1.84	184369	1.18
catpose0	21323	1.43	23675	1.29
cylinder2boxmultiple	196982	1.09	202433	1.06
chicken_high	95843	1.47	127510	1.11
dinoflat	32561	1.46	39477	1.21
human_hand	46786	1.35	56209	1.13
noisybird	17908	1.40	21293	1.18
pig_highres	65764	1.36	76967	1.16
rockerarm	127048	1.22	140120	1.11
smoothx	109955	1.14	109555	1.14
tap_threads	149568	1.66	212250	1.17
t-rex_high	73222	1.63	103115	1.16

4.4 Algorithmic enhancements

When an algorithm is parallisable large performance gains can be accomplished, without fundamentally changing the algorithm. The performance gain can be as high as n times, where n is the number of available processing units. The fact that the algorithm fundamentally stays the same is a big advantage, as it means the algorithm does not get harder to understand and does not produce output of inferior quality. But on the other hand, it is also a disadvantage, as the speed-up will never exceed n. Much larger performance gains may be possible by altering the algorithm or using a different algorithm altogether.

While this thesis does not introduce a way to reduce the time complexity of the algorithm and accomplish performance gains which exceed the ones acquired by parallelisation, it will introduce ways to attack the identified bottleneck of the algorithm (the importance computation, see section 4.1), in order to achieve a modest speed-up.

4.4.1 Importance measure early termination

The high time complexity of the importance measure stems from the fact that it needs to visit all boundary voxels in order to compute the area of the boundary components, from which in turn the importance is computed (recall section 1.4). This sounds like a waste, since ultimately not the exact importance value is used for simplifying curve skeletons. Only the relation between the importance value and the threshold τ is of importance.

One option for improving the performance of the importance computation is terminating as soon as the importance value is higher than the threshold τ. This is possible because the component areas are computed in an incremental fashion. The full algorithm for importance computation is algorithm 4, with the optimisation included on lines 10–11. This optimisation does not result in a loss of quality, although the curve skeleton needs to be re-computed when one wants to use a higher threshold. The importance values smaller than τ are still computed and can therefore be used to fine-tune the simplified curve skeleton.

The results are included in table 4.3. The speed-up values are relative to the computation time parallelised with four threads and are quite impressive at an average of 1.42. The threshold used is 0.1. This is a fairly low value (the range is $[0, 1]$), but the data show that noise on the curve skeleton has a very low importance value (remember figure 3.1). Choosing an even lower value would improve performance, but a higher value would give more freedom in controlling the simplification process, in exchange for a performance penalty.

A curve skeleton produced by this method is figure 4.1c.

4.4.2 Importance measure avoidance

Since the importance measure is an optional part of the skeletonisation pipeline, another approach at attacking its detrimental effect on performance is omitted it altogether. If it were possible to, in most cases,

Algorithm 4 Importance measure of $p \in C(\Omega)$

```
Require: p ∈ C(Ω), number of neighbouring components n, threshold parameter τ
Ensure: Return value ρ(p) is the importance value of p
 1: for all i ∈ [1, n] do
 2:   s ← seed voxel of the ith component {found with algorithm 3}
 3:   V ← D(p) {visited voxels, exclude dilated-path set}
 4:   insert s in V
 5:   C_i(p) ← ∅
 6:   insert s in C_i(p)
 7:   Q ← empty queue
 8:   insert s in Q
 9:   while |Q| ≠ 0 do
10:     if (2/δΩ)|C_i(p)| ≥ τ^2 then
11:       return ρ(p) = 1.0 {importance is high enough}
12:     else
13:       x ← remove from Q
14:       for all k ∈ neighbours(x) do
15:         if k ∉ V then
16:           insert k in V, C_i(p) and Q
17:         end if
18:       end for
19:     end if
20:   end while
21: end for
22: return ρ(p) calculated with equation (1.4)
```

determine upfront whether a curve skeleton voxel is certainly important or unimportant, that information could be used to decide whether to execute the importance calculation or not. A voxel that is certainly unimportant is ignored and a voxel which is certainly important will be included in the simplified curve skeleton. For the remaining voxels the importance measure is calculated and its inclusion in the simplified curve skeleton is determined based on the threshold.

When looking at the geodesics of a curve skeleton point (see for example figure 1.2), one can see that the (largest) angle between pairs of geodesics is usually very large (in the example 180 degrees). This suggests it may be possible to predict whether a curve skeleton voxel is important by specifying a lower and upper bound for the maximum angle between the geodesics. When the geodesic angle of a curve skeleton is smaller than the lower bound, it is deemed unimportant and assigned an importance of zero. When it is larger than the upper bound it is considered important and given an importance of one. Otherwise the importance is calculated as usual. The angle between geodesics is computed using equation (4.1):

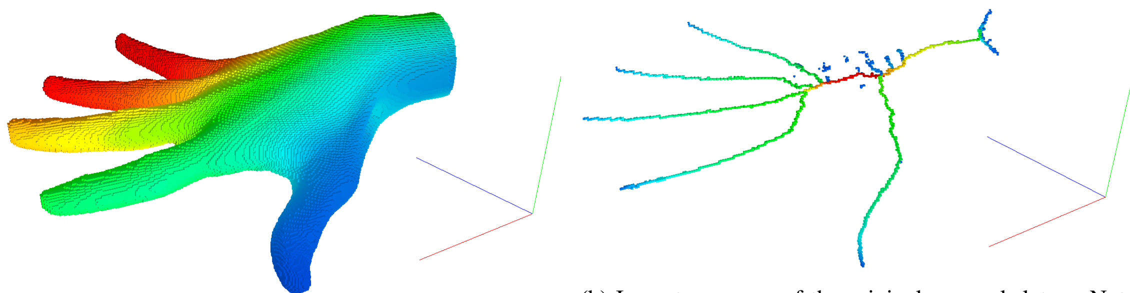
$$\alpha(p) = \max_{a,b \in \Gamma(p)} \arccos(\vec{M}(a) \cdot \vec{M}(b)) \tag{4.1}$$

Where $\alpha(p)$ is the largest angle between the geodesics of p and $\vec{M}(\gamma)$ is the midpoint of γ.

It turn out that the angle between geodesics is not a good indication of a voxel's importance, possible because of discretisation issues. Using it to decide whether to incorporate certain curve skeleton voxels in the simplified curve skeleton may cause certain important voxels to be missing and certain unimportant voxels to be present, even when using conservative values for the angle parameters. This is illustrated in figure 4.1d. Also, the speed-up achieved is not very good, as shown in table 4.3, averaging at 1.16. The lower bound used for the geodesic angle is 60 degrees, the upper bound is 160 degrees. Widening this range would improve performance, narrowing decreases performance.

Summary

Computing curve skeletons can be quite computationally expensive, especially when including the importance measure. This is further amplified by the fact that volumetric datasets are usually very large. Fortu-



(a) The original volume.

(b) Importance map of the original curve skeleton. Note that the noise in the middle part has a low importance value.

(c) Importance map of the curve skeleton, when applying the early termination optimisation. The importance values of the most important part were not fully computed but simply set to one.

(d) Importance map of the curve skeleton, when applying the avoidance optimisation. The curve skeleton voxels for which the importance measure was avoided are scattered all over the skeleton.

Figure 4.1: A volume of a human hand and its curve skeletons, when applying the different optimisation techniques.

nately, large performance gains can be realised as this algorithm is suitable for parallelisation. The importance measure can also be sped up by terminating once the threshold is reached, instead of computing the actual value. None of these methods result in a quality reduction. Attempts to avoid computing the importance by using a heuristic were less successful, as this does result in a reduction of quality and the speed-up is less than stellar.

Chapter 5

Discussion

This chapter introduces a possible application for curve skeletons (in addition to the applications mentioned in section 2.2), which concerns analysing the geometric and topological properties of a volumetric object. Also included is a discussion about possible future work in the area of curve skeletons.

5.1 Skeletonisation of plant roots

On of the original aims of this project was to develop a program which could compute various metrics of the structure of plant roots, by analysing volumetric scans of plant roots by using curve skeleton technology. An example of such a scanned dataset in shown in figure 5.1. This did not quite succeed, as the high-quality curve skeletons required for such a task are not yet available. The main problem is that the plant roots consist of thin structures, for which scans of relatively low resolution may be inadequate when computing curve skeletons, as discussed in chapter 3. Increasing the resolution of the scanner would introduce a new problem, in the area of performance, which resulted in the performance analysis of chapter 4.

While an application for plant root visualisation was not built, it is nevertheless an interesting case study. Such an application would use graph-traversal algorithms to calculate various metrics such as the average branch length, the number of ramifications, variation in thickness and so on. One would construct a graph from a one-voxel-thin curve skeleton by turning the voxels in graph vertices and constructing edges of the appropriate length between them. The graph-traversal algorithm could be made more efficient by replacing curve skeleton arcs by straight lines (a subject briefly touched upon by Cornea et al. [2007]). Finally, the application would have to visualise these data in a user-friendly manner.

5.2 Future work

The introduction of a formal definition for curve skeletons by Dey and Sun [2006], Reniers et al. [2007] is hardly the end of the research on curve skeletons. The results of chapter 3 show that the current implementation does not produce perfect curve skeletons, so smart modifications and additions will have to be developed before a *generic* curve skeleton computation implementation can be applied on real-world problems. Especially the problem of irregular twists in the curve skeleton seems to be a major problem.

Performance may be become a problem when the size of the datasets increase. Chapter 4 shows that the problem of computing curve skeletons is massively parallel, which is good news, considering the increased focus on increasing the number of processors in present-day computers, rather than making the individual processors faster. This also suggest curve skeleton computation could benefit from being moved from the CPU to the GPU, as modern GPUs consist of numerous processing units. Emerging general-purpose GPU programming languages may make this more feasible, but right now it would be hard to do. Even though each voxel can be processed individually, the algorithms require branching logic and complex data structures, which cannot be implemented efficiently on GPU hardware right now. GPU can already be used for computing distance transforms [Thanh-Tung Cao et al., 2010] and surface skeletons [van Dortmont et al., 2006].

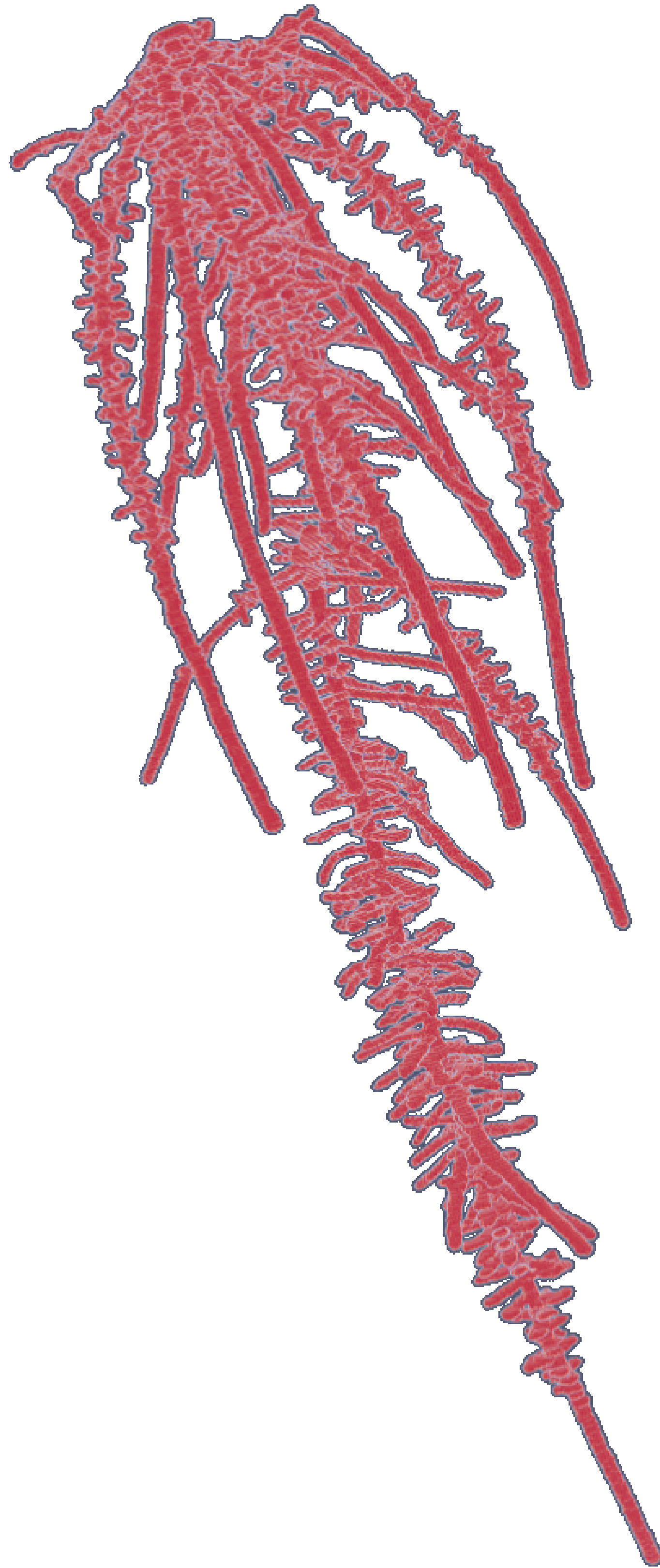


Figure 5.1: A volume of plant roots, acquired by a scanning device.

Bibliography

Blaise Barney. Introduction to Parallel Computing. Technical report, Lawrence Livermore National Laboratory, 2012. URL `https://computing.llnl.gov/tutorials/parallel_comp`.

Nicu D. Cornea, Deborah Silver, and Patrick Min. Curve-Skeleton Properties, Applications, and Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 13:530–548, 2007. URL `http://coewww.rutgers.edu/www2/vizlab/NicuCornea/publication/1stpaper_download/tvcg06.pdf`.

Tamal K. Dey and Jian Sun. Defining and Computing Curve-skeletons with Medial Geodesic Function. In *Proceedings Eurographics Symposium on Geometry Processing*, 2006. URL `http://www.cse.ohio-state.edu/~tamaldey/paper/curve-skeleton/cskel.pdf`.

Dennie Reniers and Alexandru Telea. Tolerance-Based Feature Transforms. *Advances in Computer Graphics and Computer Vision and Communications in Computer and Information Science*, 2007. URL `http://www.cs.rug.nl/~alext/PAPERS/CGVTA07/chapter.pdf`.

Dennie Reniers and Alexandru Telea. Hierarchical Part-type Segmentation using Voxel-based Curve Skeletons. *The Visual Computer*, 2008. URL `http://www.cs.rug.nl/~alext/PAPERS/VIS_COMPUTER08/paper.pdf`.

Dennie Reniers, Jarke J. van Wijk, and Alexandru Telea. Computing Multiscale Curve and Surface Skeletons of Genus 0 Shapes Using a Global Importance Measure. *IEEE Transactions on Visualization and Computer Graphics*, 2007. URL `http://www.cs.rug.nl/~alext/PAPERS/TVCG07/paper.pdf`.

Alexandru Telea and Jarke J. van Wijk. An Augmented Fast Marching Method for Computing Skeletons and Centerlines. In *Proceedings Eurographics–IEEE TCVG Symposium on Visualization*. ACM Press, 2002. URL `http://www.cs.rug.nl/~alext/PAPERS/VisSym02/dskel.pdf`.

Alexandru Telea and Anna Vilanova. A Robust Level-Set Algorithm for Centerline Extraction. In *Proceedings Eurographics–IEEE TCVG Symposium on Visualization*. ACM Press, 2003. URL `http://www.cs.rug.nl/~alext/PAPERS/VisSym03/paper.pdf`.

Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. Parallel Banding Algorithm to Compute Exact Distance Transform with the GPU. In *The Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2010. URL `http://www.comp.nus.edu.sg/~tants/pba_files/pba.pdf`.

M.A.M.M. van Dortmont, H.M.M. van de Wetering, and A.C. Telea. Skeletonization and Distance Transforms of 3D Volumes Using Graphics Hardware. In *Proceedings DGCI*. Springer, 2006. URL `http://www.cs.rug.nl/~alext/PAPERS/DGCI06/paper.pdf`.