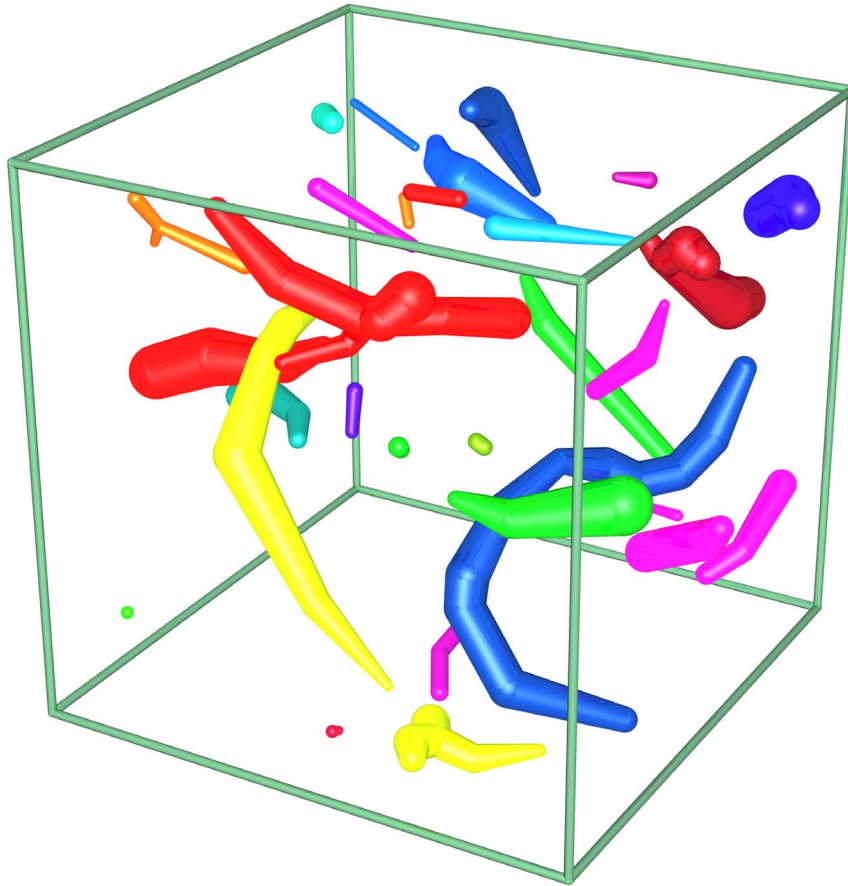


Feature Tracking with Skeleton Graphs



Benjamin Vrolijk

Delft, March 2001

Feature Tracking with Skeleton Graphs

Benjamin Vrolijk

Master's Thesis

Delft University of Technology
Faculty of Information Technology and Systems
Computer Graphics and CAD/CAM Group

Preface

This thesis is the result of my Master's Project. During the past year, I have been working on scientific visualisation at the Computer Graphics and CAD/CAM Group of the Faculty of Information Technology and Systems at the Delft University of Technology.

This work consists of two parts. The first part is a paper, we have submitted for publication in the Proceedings of the Dagstuhl Scientific Visualization Seminar, which are to appear in 2001. The second part describes in more detail the work I have done, more or less as a supplement to the paper.

I wish to thank my supervisors, Frits Post and Freek Reinders, for their support, advice and comments. I also wish to thank the other people of the Computer Graphics and CAD/CAM Group for whatever support they have given me, for creating a nice atmosphere to work in, and of course for their frequent servings of coffee, tea and cake.

Finally, I would like to thank my friends and family, and especially my parents, for all their loving support, during my entire study, in the past years.

Benjamin Vrolijk, March 2001.

Contents

Preface	iii
Contents	v
I Paper	1
II Report	19
1 Introduction	23
2 AVS Modules	27
2.1 Distance Transformation	27
2.2 Skeletonization	28
2.3 SkeletonGraph	28
3 The Skeleton Graph Representation	31
3.1 Data Structure	31
3.2 Attribute Calculation	31
4 Tracking Algorithms	35
4.1 Prediction	35
4.2 Merge	36
4.3 Topology Comparison	36
4.4 Neighbourhood Criterion	40
4.4.1 Skeleton COG distance	40
4.4.2 Node-to-node distance	40
4.4.3 Node-to-edge distance	40
4.4.4 Edge-to-edge distance	41
4.5 Performance	43
5 User Interface and User Interaction	45
5.1 Dual-view principle	45
5.2 Interaction modes	45
5.3 Highlighting	46
5.4 Graph Icons	48
5.5 Feature Icons	49
5.6 Attribute Profiles	49

5.7	User-guided tracking	51
5.8	Implementation	54
6	Conclusions and Further Research	57
A	Tracker Class Structure	59
B	Topology Comparison	61
	Bibliography	65

Part I
Paper

Feature Tracking with Skeleton Graphs

Benjamin Vrolijk Freek Reinders
Frits H. Post
email: {b.vrolijk, k.f.j.reinders, f.h.post}@its.tudelft.nl

March 2001

Abstract

One way to visualise large time-dependent data sets, is by visualisation of the evolution of features in these data. The process consists of four steps: feature extraction, feature tracking, event detection, and visualisation.

In earlier work, we described the execution of the tracking process by means of basic attributes like position and size, gathered in ellipsoid feature descriptions. Although these basic attributes are accurate and provide good tracking results, they provide little shape information. In other work, we presented a way to describe the shape of the features by skeleton attributes.

In this paper, we investigate the role that the skeleton graphs can play in feature tracking and event detection. The extra shape information allows detection of certain events much more accurately, and also allows detection of new types of events: changes in the topology of the feature.

Keywords: Data Visualisation, Feature Extraction, Feature Tracking, Skeleton, Skeleton Graph, Graph Matching, Event Detection.

1 Introduction

The problem of analysing very large scientific data sets originated the field of scientific visualisation in the 1980s. The size of data sets has grown rapidly in the past decade, in particular with data sets generated from simulations of highly dynamic phenomena, such as time-dependent flows. Yet many visualisation techniques, especially global field visualisation techniques such as volume rendering or iso-surfaces, do not scale easily to very large data sets, and thus are not very well suited to the analysis of time-dependent data sets.

One solution to this problem is the approach called *feature extraction*, in which interesting phenomena (features) are automatically detected in large fields, and quantitatively described by computing attribute sets [15]. This results in a quantitative description of the features, which can be used for visualisation and further investigation. Examples of features in flow data are vortices, shock waves, and recirculation zones. An advantage of this approach is that we concentrate on the relevant phenomena and the amount of data to be handled is reduced by a factor of 1000 or more.

Time-dependent data sets are often represented by a number of data fields (*frames*), one for each time step. If features are extracted from each frame separately, a correspondence between features in consecutive frames can be established, using the attribute sets. We call this process *time tracking* [11, 13, 8]. It allows us to visualise the development of these corresponding features over the total time interval of a simulation: motion paths, growth and shape changes can be shown. In these time histories, we can also detect interesting temporal changes (*events*), such as the birth of a new feature, the exit of a feature from the domain, or interactions of multiple features, such as splitting into two or more features, or merging of multiple features into one [9]. Again, these events can be quantitatively described.

For time tracking, we have used primary attribute sets of the features: centre position, volume, size, and orientation [8]. The determination of these attributes is usually based on volume integrals of feature objects, and the accuracy and robustness of these attributes has been demonstrated [10]. The shape of a feature is described by a spatial distribution function, which gives a good estimate of the overall dimensions of a feature, but does not describe the feature's shape in any detail.

For some purposes, the feature shape is important and must be quantified in more detail. In a previous paper [7], we have proposed the use of skeletons and distance transforms for a better quantification of the shape of a class of features. A simplified skeleton can be represented as a graph, which can be used in combination with distance transforms to reconstruct the basic 3D shape of a feature, and can also be used as an iconic object for visualisation.

In this paper we will investigate the role that these skeleton graph models can play in feature tracking and event detection. Generally, the tracking process will be guided by the primary attributes determined by volume integrals, but the skeleton graphs will permit detection of certain events with much higher accuracy. Also, the temporal development of feature shape is quantified in detail, and new types of events, such as changes in the topology of a feature, can be detected.

The structure of the paper is as follows. In Section 2, the feature extraction process is summarised. In Sections 3 and 4, we describe feature tracking and event detection methods using skeleton graphs. In Section 5 some related work is mentioned. We present some results in Section 6, including a comparison of integral properties and skeleton-based properties. We will then summarise and give future research directions in Section 7.

2 Feature extraction

The process of feature extraction can typically be described by the following four steps:

- segmentation or data selection: the parts of the data set are selected that are of interest. This can be done in many different ways, depending on the type of features to be detected. One general technique selects nodes in a grid by applying multiple thresholds to the data defined at each grid point, or quantities derived from the data [15]. In principle however, any segmentation technique that produces a set of spatially located or grouped data items can be used.

- clustering: from the data items, coherent groups can be formed by building coherent clusters from spatially related items. This results in a number of clusters, which can be treated as distinct objects.
- attribute calculation: for each cluster, a number of descriptive attributes are calculated, such as centre position, volume or orientation.
- iconic mapping: the attributes of the clusters are mapped to the parameters of parametric icons, which can be easily visualised.

We have often used volume integrals to compute the attributes of the features. This gives us the volume and the centroid of the features. By computing the eigenvalues and eigenvectors of the covariance matrix of the point positions, we can get a first-order estimation of the size and orientation. These attributes can be used to create an ellipsoid representation of the features, for visualisation. In the rest of this paper, we will also use the terms ellipsoid volume and ellipsoid position, when we mean these attributes, computed by a volume integral.

Because the ellipsoid representation does not give a real description of the shape and orientation of the feature, we have created another attribute set, to describe the shape in more detail.

We have developed an algorithm to create a *skeleton graph representation* of a feature [7]. The algorithm first computes the skeleton of an object. This skeleton consists of the central points of the object, and is obtained by successively removing the outer layers of object points, as long as the topology of the object does not change.

Also, a distance transformation is computed, which stores for each point the minimal distance to the surface of the object.

From these data sets, we create a skeleton graph. The points of the skeleton become the nodes of the graph and the neighbouring nodes are connected by edges. Each node gets two attributes: the position of the node and the distance transform (DT) at that point.

This graph can then be simplified. To describe the topology of the object, we only need a set of special points. We distinguish three different types of topological nodes (see Figure 1a): *end* (E), *junction* (J) and *loop* (L) nodes. End nodes have one edge, junction nodes have more than two edges and loop nodes have a topological edge to themselves.

Besides these topological nodes, we introduce two types of geometric nodes for describing the shape more accurately. The first type is the *curve node* (see Figure 1b). Here, the skeleton is strongly curved. The topology of the feature consists of only one edge (E_1E_2). However, to be able to reconstruct the path of the skeleton more accurately, we have to add at least one node (C) in between. We call this node a curve node. The number of curve nodes added is dependent on a user provided tolerance. This tolerance gives the maximum allowed distance (d) of the skeleton points from the skeleton graph edges.

The second type is the *profile node* (see Figure 1c). Here, the skeleton of the feature is a straight line, but the object surface is curved. Thus, the distance transform varies along the topological edge E_1E_2 . Therefore, to be able to reconstruct the surface of the feature more accurately, we have to add a profile node (P) halfway. The number of profile nodes added is dependent on

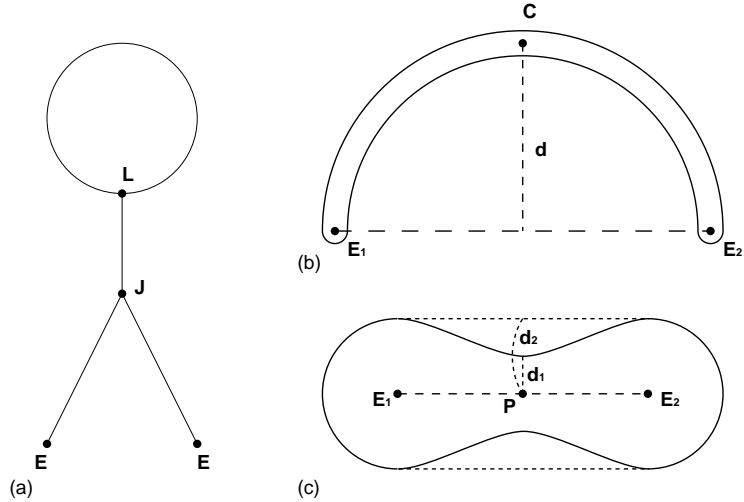


Figure 1: a) The topological node types. b) Adding a curve node. c) Adding a profile node.

a user provided tolerance, which determines the maximum allowed distance of the interpolated DT (d_2) from the real DT (d_1).

We now have a graph, which exactly describes the topology of the object, and approximately describes the shape. We can reconstruct the surface of the object by using the DTs of the nodes. By representing the nodes by spheres with radius DT, and the edges by conical segments, we get an approximate reconstruction of the original shape. This reconstructed shape can be used for visualisation and for attribute calculation.

The reconstructed shape is only an inscribed object of the original shape, because the distance used is the minimal distance to the surface. Therefore, the attribute calculation will not be as accurate as the volume integral calculation, but when the latter is not available, the former can be used instead. We have done a few tests to determine the accuracy of these calculations; the results of these tests will be described in Section 6.

3 Feature tracking

When features have been extracted in every frame of a time-dependent data set, still no motion information is available. To obtain this, we have to establish a frame-to-frame correspondence between features in successive frames. This is called the *correspondence problem* [2]. There are several ways to solve this problem [11, 13, 9]. Our algorithm is based on the assumption that features evolve predictably. Once a part of a path of a feature has been found, we assume we can make a prediction of the path into the next frame. We then compare the prediction with the real features in that next frame and search for a match. If one or more matches are found, we add the best match to the end of the path and continue into the next frame. For the prediction we simply use linear extrapolation. We assume for example, that if a feature moves a distance d from frame $t - 1$ to frame t , it will move the same distance from frame t to

frame $t + 1$. A similar assumption is made for volume, orientation, etc. For the matching of a prediction with a feature, we need a way to determine a correspondence between two features.

We use functions that compute a correspondence factor for each type of attribute. For the volume integral attributes, for instance, we have correspondence functions for the position and the volume (see Table 1).

Corr. criterion	Corr. function	Corr. factor
Position	$\ P_1 - P_2\ \leq T_P$	$C_P = 1 - \frac{\ P_1 - P_2\ }{T_P}$
Volume	$\frac{ V_1 - V_2 }{\max(V_1, V_2)} \leq T_V$	$C_V = 1 - \frac{ V_1 - V_2 }{\max(V_1, V_2) T_V}$

Table 1: Examples of correspondence functions for feature tracking. P is position, V is volume, T is tolerance.

When using the skeleton graph representations of features for tracking, we can use correspondence functions based on the topology of the skeleton graphs. For this, we could use, for example, a tolerant graph matching algorithm, such as described by B.T. Messmer in [5]. However, because of the high time and/or space complexity of these algorithms, we have created our own, more efficient special-purpose comparison algorithm.

This algorithm chooses one node in the first graph and compares that node with all nodes of the same type in the second graph. In the recursive function that compares both nodes, two arrays are used to keep track of the marked nodes in both graphs. If the number of edges from both nodes is equal, and the number of neighbouring marked nodes is equal, then both nodes are marked. Next, an unmarked neighbouring node in the first graph is chosen and compared with all unmarked neighbouring nodes in the second graph. This process is repeated until either all nodes in both graphs are marked or no more correspondences are found. In the first case, the two graphs are topologically equivalent, in the second case, the topologies differ.

To make the whole comparison algorithm more efficient, we have created two more functions that are run before the recursive algorithm.

The first function simply counts the number of edges and topological nodes of each type in both graphs. If these numbers do not correspond, the graphs will certainly not be topologically equivalent. However, if the numbers are equal, further testing is necessary.

The second function counts the types of the nodes each node is connected to, or equivalently, counts the types of connections (end-end, end-junction, etc.) in both graphs. Again, if the numbers do not correspond, the two graphs have different topologies. Otherwise, the recursive comparison algorithm is run to give the definitive answer.

Note that we only test the topology of the skeleton graphs, not the geometry. Because the geometry of the skeleton graphs is based on tolerances provided by the user during the feature extraction process, it is possible that, for exam-

ple, the number of curve nodes changes, while the shape of the feature hardly changes. The topology of a skeleton graph is more stable than the geometry, therefore we have decided to use only the topological information of the skeleton graphs in these correspondence functions.

Note also, that it is possible that, although the topology of a feature stays the same, the shape of the feature changes very much. For instance, when a junction disappears on one side and at the same time a junction appears on the other side. We do not detect these events, because for that we need a one-to-one mapping of the nodes and edges of two skeleton graphs. Because features evolve (grow/shrink and move) anyway, it is very hard to compute such a correspondence.

We have made one topology correspondence function that returns a boolean, that is, there is no correspondence if the topology changes. A second correspondence function allows a certain number of topological changes, such as the addition of loops or edges.

Because in our test application¹ sometimes the topology of the features changes more than once within a few frames, we have decided not to use the topology information for feature tracking, although it is, of course, still possible.

We have found that it works very well to search for continuing paths, using only the global attributes of the features. Therefore, we have decided to use the position and volume correspondence functions from Table 1 also when using skeleton graph descriptions of features for time tracking. Therefore, we have to compute the position and the volume of a skeleton graph.

Furthermore, we can compute another global attribute of a skeleton graph, namely its length. We compute the total length of a skeleton by adding the individual lengths of all edges of the graph. The correspondence function for the length of a skeleton graph is comparable to the function for the volume.

The volume of a skeleton graph is computed by adding the volumes of all conical segments. The volume of one segment is computed by using the formula for the volume of a conical frustum:

$$V = \frac{1}{3}\pi h(r_1^2 + r_1 r_2 + r_2^2) \quad (1)$$

with h the length of the edge and r_1 and r_2 the DTs of the end nodes of the edge. To make the computation of the total volume more accurate, we add the volume of half a sphere for each end node of the skeleton.

The position of a skeleton graph is defined to be the centre of gravity (COG) of the object, assuming a uniform density throughout the object. We can compute this by calculating the weighted average of the centres of gravity of the individual segments, using the segment volumes as weights. The COG of a conical frustum is located at

$$COG = p_1 + h(p_2 - p_1), \quad (2)$$

with p_1 and p_2 the positions of the end nodes and

$$h = \frac{\frac{1}{2} \sqrt[3]{4r_1^3 + 4r_2^3} - r_1}{r_2 - r_1}, \quad (3)$$

¹We use a 91 frame data set with turbulent vortex structures, obtained from a fluid dynamics simulation. Data courtesy D. Silver and X. Wang of Rutgers University.

the relative position of the COG on the edge from p_1 . For an edge with $r_1 = r_2$, that is a cylinder-shaped edge, the COG is located exactly halfway, so $h = 0.5$.

4 Event detection

When the continuing paths have been found, we can search for events in the evolution of the features. For instance, when a path ends, we would like to know why this happens. Examples of events are the birth or death of a feature, entry into or exit from the domain, and the splitting of one feature or merging of multiple features into one [9]. All these events come in pairs, which are each other's opposites in time. That means that if one event can be found in forward time direction, the other can be found by using the same algorithm in backward time direction. With the detailed shape information that is available with skeletons it is now also possible to detect changes in topology. We call this topological events.

4.1 Terminal events

When a feature decreases in size and disappears, we say a *death event* has occurred. We can detect such an event by comparing the volume of the features in the last frames of the path. When the volume is decreasing and the volume of the prediction is very small or negative, we call it a death event. A *birth event* is the same, only in negative time direction.

When a feature moves to an open boundary of the domain, it is possible that the feature leaves the domain through that boundary, thus ending the path. We can detect such an event by watching the position of the feature in the last frames of the path. When a feature moves towards the boundary and the position of the prediction is close to or beyond the boundary, we say we have found an *exit event*. When the same happens in opposite time direction, we call it an *entry event*.

4.2 Interaction events

When multiple features merge into one, we call this a *merge event*. In the opposite time direction, it is called a *split event*. We search for merge events when there are two or more paths ending in one frame and the ends are unresolved. We then try to merge them into a candidate feature in the next frame, for example the first feature of another path. When we have a number of end nodes in one frame and a candidate feature in the next frame, we first limit the number of end nodes by applying a neighbourhood criterion: the prediction of the end node must be within a certain search space defined around the candidate feature. When two or more end nodes remain after this selection, we try each possible combination of these end nodes and compute a merged feature from these end nodes. Next, we compare the computed merged feature with the candidate feature in the next frame. For this final comparison, the usual continuation criteria can be used. Thus, to be able to search for split and merge events, we need two extra functions: a neighbourhood function and a merge function. For ellipsoid representations, the neighbourhood function computes the distance between the centres of the ellipsoids. The merge function creates a

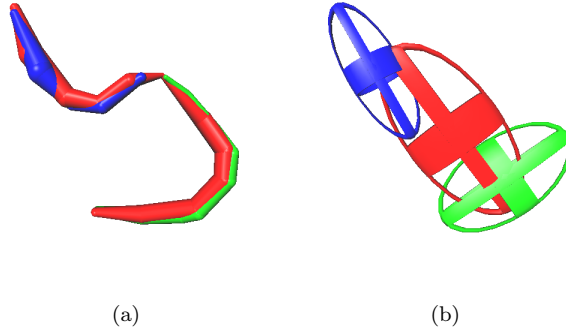


Figure 2: The same split event, using different feature representations. a) skeleton representation, b) ellipsoid representation. The red feature is before split, blue and green are after split.

new hypothetical feature, with a volume equal to the sum of the volumes of the merged features, and a position equal to the weighted average of the individual ellipsoid positions. When using skeleton graph representations, we can compute a much more accurate neighbourhood distance (see Figure 2). The most accurate measure is the edge-to-edge distance, that is, the shortest distance between any two edges of the two skeletons. Two simpler but less accurate measures are the node-to-edge and the node-to-node distances. A fourth distance measure for skeletons is comparable to the measure used for ellipsoids and uses the distance between the centres of gravity of both skeletons. The merge function for two skeletons creates a hypothetical skeleton consisting of all the nodes and edges of the separate skeletons, plus an extra edge connecting the two nearest nodes.

4.3 Topological event

Besides position and size, also the topology of a feature can change during the evolution. When this happens, we call this a *topological event*. We detect this event by using the topology comparison algorithm described earlier. When one of the three tests fails, we know the topology has changed. Next, we try to determine what kind of event has happened. We make a distinction between a *loop event* and a *junction event*. Using Euler's formula

$$V - E + F = C - H \quad (4)$$

it is easy to decide what has happened. In formula (4), V is the number of vertices, E the number of edges and F the number of faces. $C - H$ is called the Euler number, with H the number of holes and C the number of connected objects. Because we are only dealing with connected graphs, F is always 0, and C always 1 [7]. This way formula (4) reduces to:

$$H = 1 - V + E \quad (5)$$

When the value of H changes, we call it a loop event (see Figure 3). If not, but the number of (topological) edges is different, a junction has originated or

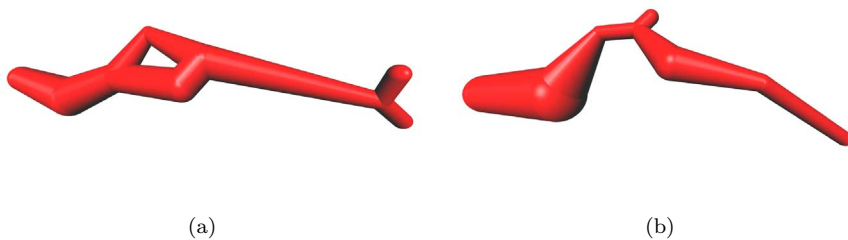


Figure 3: A loop event has occurred. In figure a) the skeleton contains a loop, in figure b), the next frame, the loop has disappeared.

disappeared, therefore we then call it a junction event. If neither of these is true, but the topology has changed in any other way, we simply call it a topological event.

5 Related Work

Solving the correspondence problem is an issue not only in scientific visualisation, but also, for example, in image processing and computer vision. For 3D field data, the matching can be achieved by two methods: region correspondence or attribute correspondence [9].

- With methods based on region correspondence, the matching can be achieved based on spatial overlap [1, 13, 14], or by finding the minimum affine transformation that transforms an object from one frame to the next [3].
- The methods based on attribute correspondence use attributes of the features, such as position and volume, for matching. Examples are the tracking of tokens by using motion smoothness [12], and the tracking of feature evolution [11] for event detection.

Our approach to solving the correspondence problem uses the second method. In the feature extraction process, we compute the attributes of the extracted objects. These attributes are then used for computing the correspondence. The comparison is based on high-level attribute data, not on the original data. Therefore, the comparison is simple and efficient. Furthermore, the correspondence functions can be physics-based, and different, depending on the feature type.

For comparison of graphs, Messmer and Bunke have investigated error-tolerant and error-correcting subgraph isomorphism detection, for example using decision trees [5, 4, 6].

In our application, the basic feature tracking is done using the integral attributes, without detailed comparison of the graphs. Changes in the topology of a feature are detected by the algorithm, described in Section 3. This algorithm is less complex than the ones described by Messmer and Bunke, but suffices for our purposes. Complex graph isomorphism analysis is not necessary for detecting loop and junction events, but it may be useful for more detailed analysis of topological events.

6 Results

Because we can use the reconstructed volume of the skeleton graphs for time tracking, we have performed a number of tests to determine how good the approximation is. We use the volume computed by a volume integral for comparison.

We have tested the influence of the shape of a feature on the volume reconstruction of the skeleton graph description of the feature. For this test, we have generated a number of ellipsoid-shaped features, with different axis ratios. We have three extreme cases:

1. a 'zeppelin-shaped' ellipsoid (axis $R_1 == R_2 \ll R_3$)
2. a 'sphere-shaped' ellipsoid (axis $R_1 == R_2 == R_3$)
3. a 'disc-shaped' ellipsoid (axis $R_1 == R_2 \gg R_3$)

The skeleton of a zeppelin is one topological edge, so the accuracy of the volume reconstruction is dependent on the number of profile nodes in between. The skeleton of a sphere is a single node, so the sphere should be represented very well. The skeleton of a disc is also a single node, but because the DT is very small, the volume estimation of the skeleton graph representation will be very bad.

Figure 4 shows the results of our tests. On the y-axis is the ratio of the skeleton volume and the integral volume, which should ideally be 1. On the x-axis is the ratio of the two equal axes to the third axis. In the middle is the sphere, with ratio 1. To the right the ratio is > 1 , that is the disc shape. To the left the ratio is < 1 , the zeppelin shape. It is easily seen, as expected, that flat objects are not represented very well; the volume ratio decreases very quickly on the right side of the graph. It is also clear from the left side of the graph, that the profile nodes are very important to get a good shape and volume approximation.

As a final test to check the accuracy of the skeleton volume, we have computed the average volume ratio for all features in the complete 91 frame data set of our test application. This average ratio is about 42%.

When comparing the skeleton graph centre of gravity with the volume integral position, we find an average distance of 1.40 voxels in a data set of 128^3 voxels.

Because the volume estimation of the skeletons is not very accurate, tracking features using this attribute will probably not work as well as when using the integral volume. We have performed an experiment tracking the same data set using different sets of correspondence functions. The first set uses the ellipsoid position and ellipsoid volume attributes. Both of these are computed using a volume integral, so both are accurate attributes of the original feature. The second set of functions uses the skeleton graph position (COG) and volume attributes. Again, especially the skeleton graph volume is not very accurate. The third set is similar to the second, except we are also using the skeleton graph length. The fourth set is a combination of the first and second, that is, it uses both the ellipsoid and skeleton position and volume attributes. Finally the fifth set also uses these attributes, plus the skeleton length.

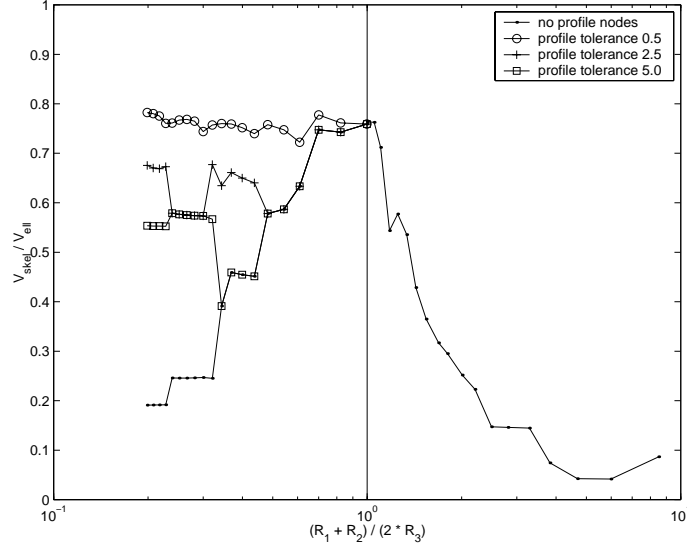


Figure 4: The skeleton volume ratio for different axis ratios.

Using these sets of correspondence functions and with the same parameters, we have tracked one data set with a number of iterations and computed what percentage of the total number of features was resolved. During these iterations the tolerances are subsequently raised until the final maximum allowed tolerance. For example, with a tolerance of 10, in 4 iterations, the tolerances used would be respectively 2.5, 5, 7.5 and finally 10. The maximum tolerances we have used are 15 voxels for position and 0.5 (50%) for volume and length. These values have been reached in 10 iterations. The minimal path length allowed is 4 frames. In a final iteration we searched for remaining paths of length 3. The results of this experiment are shown in Figure 5.

In the graph there is an extra line, called 'ell OR skel', which is a special case. For this test, we have alternately used the ellipsoid and skeleton functions, instead of simultaneously. In this case correspondences are found when predictions comply with either the ellipsoid or the skeleton functions. The results are therefore better than the rest. The number of iterations, however, is doubled. The maximum percentage of solved features, achieved in this final test, is 92.7%. The rest of the cases are all very close at the end, ranging from 88.9% to 91.5%. After the first iterations there is a large variation, with the ellipsoid functions scoring the highest percentage. At the end, the skeleton functions score slightly higher than the ellipsoid functions, and the combination of both ('ell+skel') is even better. We have also compared the paths resulting from the ellipsoid and skeleton function sets. They match for 95.70%, that is, in the data set of 20 frames with a total of 668 features, 557 correspondences were found in both cases, 13 correspondences found only using the skeleton set, and 12 correspondences found only by the ellipsoid set.

As seen in Figure 2, the skeleton neighbourhood functions will probably work much better for finding split and merge events, than those based on ellipsoid attributes. To test this hypothesis, we have performed event detection on one

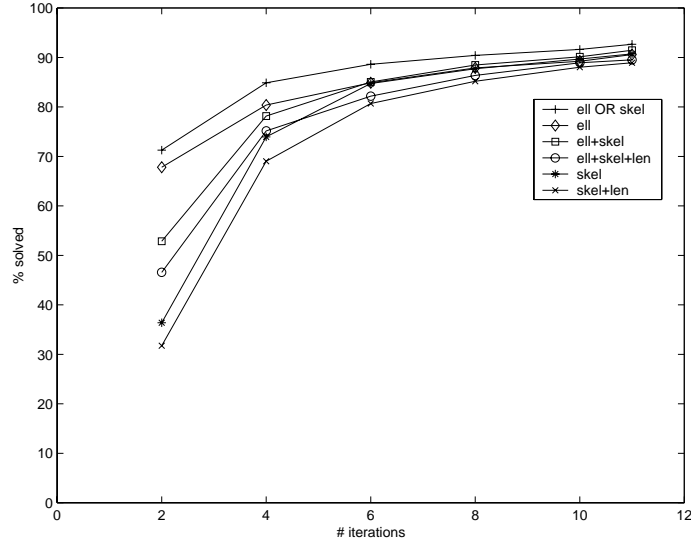


Figure 5: The percentage resolved features for different sets of correspondence functions.

data set with four different neighbourhood functions. We have used the skeleton edge-to-edge and the simpler node-to-edge neighbourhood functions. Furthermore, we have used the skeleton COG and the ellipsoid position neighbourhood functions. Figure 6 shows the relation of the number of split/merge events found and the tolerance needed to detect them, for the four different neighbourhood functions. To find all events, we need a four times larger tolerance with ellipsoids, than with skeletons. This means that a much more restrictive test can be used for the skeletons, and thus the reliability of the results is higher.

Of course, it is important to measure the performance of our algorithms. In a 91 frame data set with 3864 features and 261 paths we searched for split and merge events. On a SGI Octane with a 225 MHz processor and 256 Mb of RAM, it took 19.8 seconds to find 39 events using the edge-to-edge neighbourhood function with a tolerance of 15. When using the ellipsoid position neighbourhood function instead, with the same tolerance, it took 5.6 seconds. However, only 23 events were found. To find 39 events, we needed a tolerance of 24 and it took 25.8 seconds. Thus, the edge-to-edge neighbourhood function is 3 to 4 times slower than the ellipsoid neighbourhood function. However, because a smaller tolerance suffices to find the same number of events, eventually the use of the skeleton neighbourhood function is faster.

In the same data set we searched for topological events, to test the performance of our topology comparison algorithm. It took 0.3 seconds to search the whole data set and find 199 junction and 4 loop events.

As an example, in Figure 7 are a number of frames in the evolution of a single feature. A number of topological and interaction events have been detected, for example, in frame $t = 72$, a merge event has occurred, and in frame $t = 89$, the large feature has split in three.

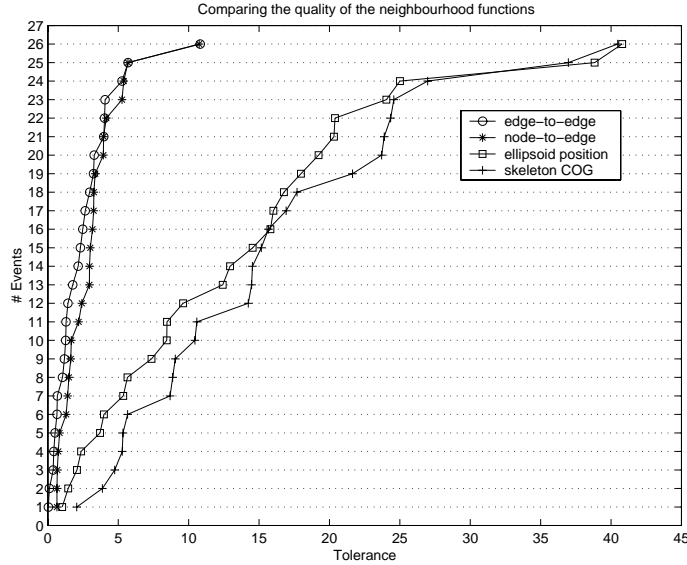


Figure 6: The number of split/merge events found against the tolerance for different skeleton and ellipsoid neighbourhood functions.

7 Conclusions and Further Research

In Figure 5 it is shown that using ellipsoid attributes for tracking continuations gives better results with less iterations. However, when tolerances are increased, the results draw nearer to each other and can even become better when using skeleton attributes or both types of attributes. Furthermore, as shown in Figure 6, the skeleton neighbourhood functions are much better for finding split and merge events, than those based on ellipsoid data. And finally, the skeleton attribute sets give much better shape information, allowing us to look for a completely new type of events: the topological events.

More research could be done to create a more accurate line skeleton representation, in which the contour of the skeleton is represented more accurately. In our skeletons, nodes are represented by spheres, with a radius equal to the nearest distance to the surface, and edges by conical segments connecting these nodes. Therefore, a cross-section of the skeleton, perpendicular to an edge, is always a circle. A more accurate representation could be to compute a complete contour of the feature, more or less simplified. This is, of course, computationally expensive. Much simpler, but still providing a much better volume estimation of the feature than our current skeletons, would be to compute an ellipse fitting of the cross-section, or to use a circle with radius equal to the average distance instead of the minimal distance to the surface.

A next step would be to develop a new representation for surface skeletons. Our line skeletons work well for the test application we have used, with mostly tube-like features. But for applications with features such as shock waves or separation surfaces, this method does not work very well.

Future work could include the development of geometrical events, for detecting all significant shape changes. Also, more precise descriptions of events

could be developed, for example, describing not only the fact that two features merge, but also exactly how the original features are connected.

References

- [1] Y. Arnaud, M. Debois, and J. Maizi. Automatic Tracking and Characterization of African Convective Systems on Meteosat Pictures. *J. of Appl. Meteorology*, 31:443–453, May 1992.
- [2] D.H. Ballard and C.M. Brown. *Computer Vision*. Prentice-Hall, 1982.
- [3] D.S. Kalivas and A.A. Sawchuk. A Region Matching Motion Estimation Algorithm. *CVGIP: Image Understanding*, 54(2):275–288, Sep 1991.
- [4] B.T. Messmer and H. Bunke. Error-correcting graph isomorphism using decision trees. *International Journal of Pattern Recognition and Artificial Intelligence*, 12(6):721–742, 1998.
- [5] B.T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):493–505, May 1998.
- [6] B.T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.
- [7] F. Reinders, M.E.D. Jacobson, and F.H. Post. Skeleton Graph Generation for Feature Shape Description. In W. de Leeuw and R. van Liere, editors, *Data Visualization 2000*, pages 73–82. Springer Verlag, 2000.
- [8] F. Reinders, F.H. Post, and H.J.W. Spoelder. Attribute-Based Feature Tracking. In E. Gröller, H. Löffelmann, and W. Ribarsky, editors, *Data Visualization '99*, pages 63–72. Springer Verlag, 1999.
- [9] F. Reinders, F.H. Post, and H.J.W. Spoelder. Visualization of Time-Dependent Data with Feature Tracking and Event Detection. *The Visual Computer*, 17(1):55–71, January 2001.
- [10] F. Reinders, H.J.W. Spoelder, and F.H. Post. Experiments on the Accuracy of Feature Extraction. In D. Bartz, editor, *Visualization in Scientific Computing '98*, pages 49–58. Springer Verlag, April 1998.
- [11] R. Samtaney, D. Silver, N. Zabusky, and J. Cao. Visualizing Features and Tracking Their Evolution. *Computer*, 27(7):20–27, July 1994.
- [12] I.K. Sethi, N.V. Patel, and J.H. Yoo. A General Approach for Token Correspondence. *Pattern Recognition*, 27(12):1775–1786, Dec 1994.
- [13] D. Silver and X. Wang. Volume Tracking. In R. Yagel and G.M. Nielson, editors, *Proc. Visualization '96*, pages 157–164. IEEE Computer Society Press, 1996.
- [14] D. Silver and X. Wang. Tracking Scalar Features in Unstructured DataSets. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *Proc. Visualization '98*, pages 79–86. IEEE Computer Society Press, 1998.

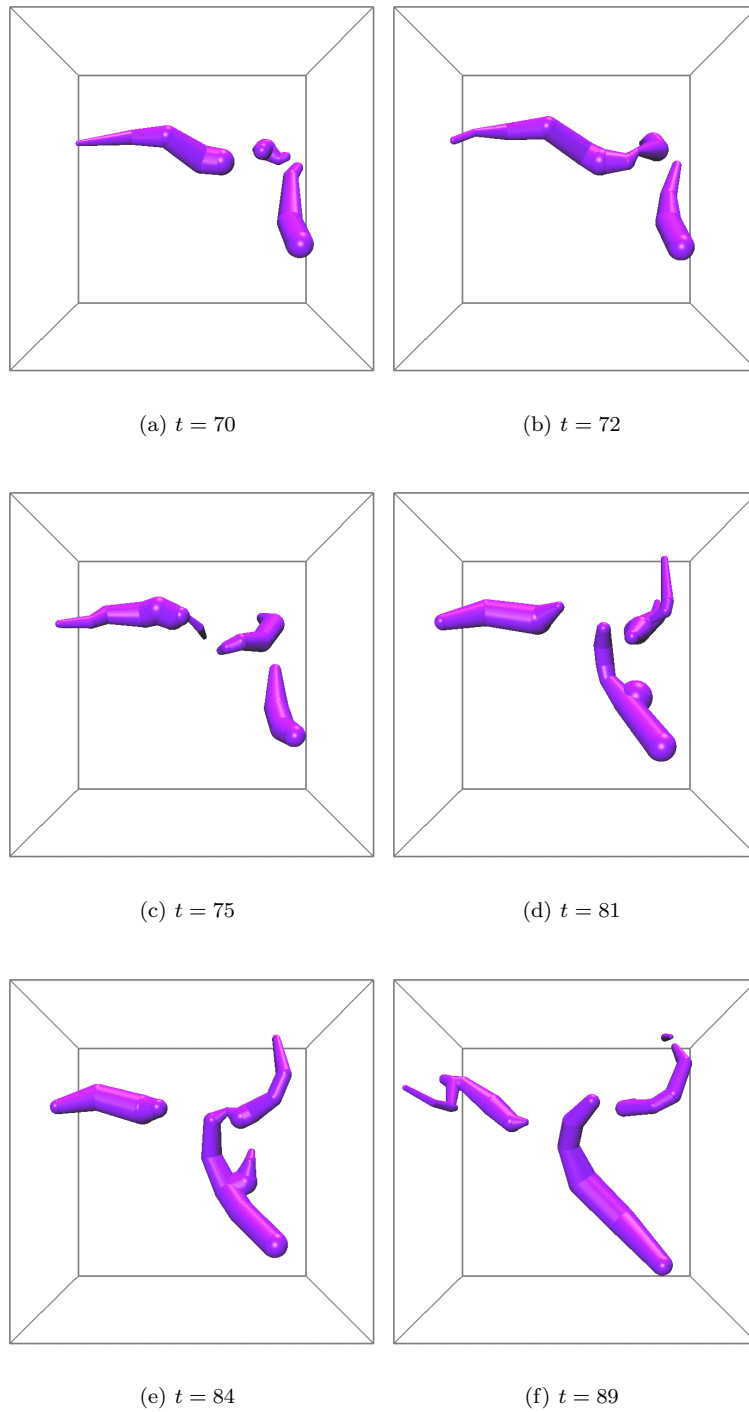


Figure 7: Six frames in the evolution of an object.

- [15] T. van Walsum, F.H. Post, D. Silver, and F.J. Post. Feature Extraction and Iconic Visualization. *IEEE Trans. on Visualization and Computer Graphics*, 2(2):111–119, 1996.

Part II
Report

Abstract

Scientific visualisation often involves working with very large data sets. One way to visualise large time-dependent data sets, is by visualisation of the evolution of characteristic features in these data. These features are structures in the data, which are of interest to the researcher. In time-dependent data sets, the features can be extracted in each time step, and the evolution of the features in time, can be analysed and described by determining the corresponding features in all consecutive time steps of the data set. The process consists of four steps: feature extraction, feature tracking, event detection and visualisation.

In earlier research, feature tracking had been made possible for features, described by volume integral attributes. However, these feature descriptions give only basic shape information. Therefore, skeleton graph descriptions of features have been created, giving detailed shape and topology information of the features.

We have investigated the use of these skeleton graph descriptions for feature tracking and event detection. As a result, detection of certain events has become more accurate, and techniques for detection of a completely new kind of event, the topological event, have been developed.

Also, the visualisation of time-dependent data sets has been much improved, by providing user-interaction capabilities, such as selection and picking, highlighting, attribute plotting and event querying.

Chapter 1

Introduction

Scientific data sets are constantly growing in size. This poses very large problems for the field of scientific visualisation. Many of the standard visualisation techniques, such as volume rendering or iso-surfaces, are not suitable for working with these large data sets.

Time-dependent data sets are often represented by a number of data fields, one for each time step. Analysis and visualisation such time-dependent data sets can be achieved in a number of steps [11, 7].

1. **Feature Extraction.** In each time step (*frame*), the interesting parts (*features*) of the data set are determined. A number of characteristic attributes of these features are calculated, such as position and size. The features can then be described very compactly, by these attributes [12, 8].
2. **Feature Tracking.** The features from consecutive frames are matched by solving the frame-to-frame correspondence problem, using the attribute descriptions of the features. This results in a number of paths, or evolutions, of the features in time [9].
3. **Event Detection.** Certain types of events in the evolutions of the features, are detected and described [10].
4. **Visualisation.** The evolutions of the features are visualised interactively, using compact iconic descriptions of the features [12].

This stepwise approach to analysis and visualisation of time-dependent data is summarised in Figure 1.1.

We use AVS [1] to perform the feature extraction process, but other methods for feature extraction could be used as well. In AVS, the process is split into a number of modules. There are different modules for computing different attribute sets. Melvin Jacobson has created a number of modules for computing skeleton graph attribute sets [3]. The results of the feature extraction process, the attribute descriptions of the features, are written to a *feature file*. The remaining steps (feature tracking, event detection and visualisation) are performed in a separate program (**Tracker**), which has been written by Freek Reinders, during his PhD project. In this program the feature data set is imported from the feature file. Next, feature tracking and event detection can be

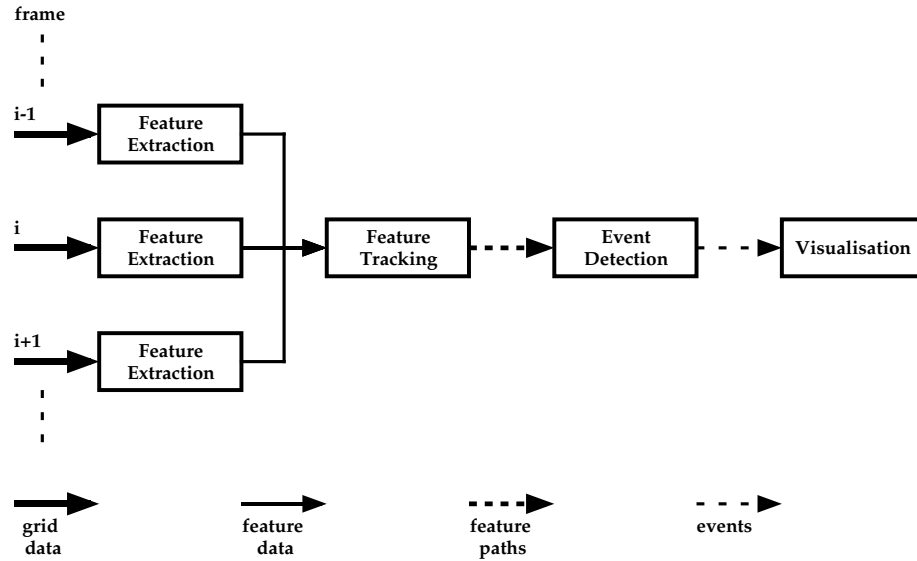


Figure 1.1: The feature tracking pipeline.

performed interactively and with user-provided criteria and tolerances. Finally, the data set can be visualised interactively using two different views.

My work during my Master's project has consisted of the following tasks.

- Making a number of adjustments to the AVS modules, written by Jacobson, for computing the skeleton graph attribute sets of features. Besides correcting a few errors, I have worked on the memory efficiency and user interface of the modules.
- Adapting the **Tracker** program, making it suitable for tracking features using the skeleton graph attribute sets.
- Creating the algorithms for the detection of two new types of events, concerning the topological changes in the skeleton graphs.
- Testing the accuracy of the calculation of the skeleton graph attributes and both the accuracy and the performance of the feature tracking and event detection algorithms.
- Enhancing the visualisation capabilities of the program, by creating better interaction between the two views on the data set, and creating more facilities for user interaction, enabling the user to perform selection and picking, event querying and attribute plotting.

Except for the first and last point, the work is described in outline in the paper in Part I [13]. In the remainder of this thesis, all tasks are described in more detail.

The AVS modules for feature extraction, and my adjustments to these modules are described in Chapter 2. The classes needed to describe the skeleton

graph attribute set in the **Tracker** program, are given in Chapter 3. The algorithms needed to perform feature tracking and event detection on the skeleton graph features are described in Chapter 4, together with a few performance measures. The improvements on the user interface of the program are described in Chapter 5. Finally, the conclusions and recommendations for further research are presented in Chapter 6.

Chapter 2

AVS Modules

Melvin Jacobson created a number of AVS [1] modules for computing the skeleton graph attribute set of a segmented object [3]. I have made some adjustments to these modules, both for better memory efficiency, and easier user interface.

2.1 Distance Transformation

The DT module computes the distance transformation of a segmented volume. The distance transformation of an object computes the shortest distance to the surface of the object. The input to this module consists of clustered data. The integer value at each grid point of this clustered data indicates the number of the cluster to which the grid point belongs, or zero, if it has not been selected [12]. The output of the module contains, at each grid point, the distance to the surface of the cluster.

The distance at each grid point is computed in two passes by adding local distance values to the distance values of the neighbouring grid points. In three dimensions, we can use the 6-, 18- or 26-neighbour distance, and the more accurate chamfer distance. In the user-interface of the module, the user can choose, which measure is to be used.

The ideal chamfer distance measure would have local distance values of 1 for face-connected voxels, $\sqrt{2}$ for edge-connected voxels, and $\sqrt{3}$ for vertex-connected voxels [3, 2].

However, the internal computations of the module were all done with integer values. Also, the output of the module consisted of integer values. Therefore, the ideal values for the chamfer distance had to be approximated. The values used by Jacobson were 3, 4 and 5, respectively. Probably, the results still had to be divided by 3, but that was not done.

I have changed the output of the module to consist of floating point values. The internal computations are, however, still done with integer values. I have changed the values for the chamfer distance into 10, 14 and 17 for face-, edge- and vertex-connected voxels, respectively. Before the values are output, they are divided by 10, giving 1-decimal approximations to the ideal values of 1, $\sqrt{2}$ and $\sqrt{3}$.

2.2 Skeletonization

The Butcher module, created by Jacobson, computes the skeleton voxels of a binary segmented volume. The input to the module consists of clustered data. The output of the module consists of the skeleton voxels of each cluster.

It is possible to select from the user interface of the module, whether a point skeleton, a line skeleton, or a surface skeleton is to be created.

The skeletonization process is performed in a number of steps, each time removing the outer layer of voxels from the clusters. In Jacobson's version of the module, it was possible to perform this process step by step. In each step, both the remaining and the removed voxels were saved, so that it was possible to go one step backwards. This option can be interesting for debugging purposes, but will normally not be used. Therefore, this option has been removed from the module. The entire skeletonization is now performed at once. This saves approximately 8 Mb of memory usage. The final module is now called Skeletonization.

2.3 SkeletonGraph

The MakeGraph module, created by Jacobson, generates a skeleton graph description of clustered data. The input to the module consists of the DT and skeleton data, which are output from the two modules, described in Section 2.1 and Section 2.2. This module combines the skeleton voxels from the Skeletonization module with the distance values from the DT module, to create a voxel graph. The skeleton voxels become the nodes of the voxel graph. Neighbouring skeleton voxels are connected by edges, and the distance value at each voxel is stored as an attribute of each node.

Next, a topological graph is created. This topological graph consists of only those nodes that are necessary to describe the topology of the voxel graph. Three types of nodes remain in this graph: *end* nodes, which have only one edge, *junction* nodes, which have more than two edges, and *loop* nodes, which have an edge to themselves.

Also, a geometrical graph is created. This graph describes the shape of the voxel graph to a certain, user-specified, accuracy. For this, the user can specify two parameters, the curve tolerance and the profile tolerance. These parameters determine the number of *curve* and *profile* nodes, respectively.

The user can choose from the interface of the module, which of the three graphs has to be output. There are now three outputs of the module. The first is a geometry, consisting of tubes and spheres, describing the graph, which can be directly visualised using the geometry viewer of AVS. The second output is a Hermite description of the skeleton graph, which can be visualised using another AVS module (MakeIcon). The third output is an attribute set representation of the skeleton graph, which can be read by another module (CollectAV) to be written to a *feature file*.

My adaptations to this module are as follows. The creation of the Hermite description contained an error, which occurred, when the skeleton graph contained a loop. I have made the memory usage of the module a bit more efficient. Before, the skeleton graph descriptions were written directly to a file. I have removed this function and created an output, which is compatible with

the CollectAV module. Also, another input has been created, which contains the cluster numbers. The skeleton graph descriptions are now sorted according to these cluster numbers. This way, in the module CollectAV, different types of descriptions of the same clusters can be matched and written correctly to a feature file. The final module is now called SkeletonGraph.

Chapter 3

The Skeleton Graph Representation

3.1 Data Structure

The `Tracker` program of Freek Reinders was written in C⁺⁺. This is an object-oriented language. All data types are described in classes, and extensive use is made of inheritance.

In the program, the feature data set is represented by the class `TimeSet`. A `TimeSet` contains a list of `Frames`, a `Frame` contains a list of `Features` and a `Feature` contains a list of `AttributeSets`. An `AttributeSet`, finally, contains a list of `Attributes`. In short:

`TimeSet` → `Frame` → `Feature` → `AttributeSet` → `Attribute`.

The `AttributeSet` class is the super-class of a number of classes, for example, `ObjectFitting`, which is the super-class of `BoxFitting` and `EllipsoidFitting`. This way, functions that are the same for all attribute sets, have to be defined only once. Functions that are identical for both box fittings and ellipsoid fittings, only have to be defined in their common super-class, `ObjectFitting`. For the skeleton graphs, we have created the sub-class `GraphAS`, with its sub-class `Skeleton`. But `GraphAS` also has another sub-class, called `VortexGraph`.

Similarly, the `Attribute` class is the super-class of, among others, the classes `Scalar` and `Vector`. `Scalar` is the super-class of `Volume` and `Mass`. `Vector` is the super-class of `Position`. For the skeleton graphs, we have created the sub-classes `GraphEdgeAttr` and `GraphNodeAttr`. `GraphEdgeAttr` has a sub-class `TopoGraphEdge`. `GraphNodeAttr` has a sub-class `SkeletonNode`, and a sub-class `VortexNode`. See also Table 3.1. The details of the classes are described in Appendix A.

3.2 Attribute Calculation

After all nodes and edges have been read from the input file (see Figure 3.1), the topological edges are determined. The algorithm is as follows. A list of all

```

AttributeSet
→ ObjectFitting
  → EllipsoidFitting
  → BoxFiting
→ GraphAS
  → Skeleton
  → VortexGraph
Attribute
→ Scalar
  → Volume
  → Mass
→ Vector
  → Position
→ GraphEdgeAttr
  → TopoGraphEdge
→ GraphNodeAttr
  → SkeletonNode
  → VortexNode

```

Table 3.1: The class hierarchy of the `Tracker` program, with the relevant classes.

```

1 1: Skeletonization
  5 4   1 END      (85, 12, 106) 4.2
        2 JUNCTION (90, 11, 110) 4.2
        3 END      (106, 8, 101) 1.7
        4 CURVE    (87, 14, 119) 5.4
        5 END      (91, 16, 124) 3.1
        1-2 2-3 2-4 4-5

```

Figure 3.1: An example of a skeleton graph description in a feature file.

topological nodes in the graph is created, by selecting by the field `NodeType`. For each node in this list, all edges are iterated through. Each edge is followed until another topological node is found, that is also in the list. A new topological edge is created, between these two topological nodes. All non-topological nodes found in between, are stored with the topological edge, in the field `IntraNodes`. When all edges from one node have been iterated through, the node is removed from the list, and the next node is picked from the list.

Then the graph attributes are computed. First, all nodes are traversed. If there are no edges, the volume of all nodes (in this case, probably just one) is summed. If there are edges, for each end node, the volume of half a sphere is counted. Next, all edges are traversed. The length, the volume and the weighted centre of gravity of the edge are added to the running totals. The length of an edge is the distance between the end nodes:

$$l = \|p_1 - p_2\|, \quad (3.1)$$

with p_1 and p_2 the positions of the end nodes. The volume of an edge is com-

puted using the formula for the volume of a conical frustum:

$$V = \frac{1}{3}\pi l(r_1^2 + r_1 r_2 + r_2^2), \quad (3.2)$$

with r_1 and r_2 the radii of the end nodes, and l the length of the edge. The centre of gravity of an edge is computed using the following formula:

$$COG = p_1 + h(p_2 - p_1), \quad (3.3)$$

with p_1 and p_2 the positions of the end nodes and

$$h = \frac{\frac{1}{2}\sqrt[3]{4r_1^3 + 4r_2^3} - r_1}{r_2 - r_1}, \quad (3.4)$$

the relative position of the COG on the edge from p_1 . For an edge with $r_1 = r_2$, that is a cylinder-shaped edge, the COG is located exactly halfway, so $h = 0.5$. When all edges have been visited, the centre of gravity of the complete skeleton graph is computed using the sum of weighted centres of gravity of the individual edges:

$$COG_{skel} = \sum_{edges} (COG_{edge} * V_{edge}) / \sum_{edges} (V_{edge}) \quad (3.5)$$

Chapter 4

Tracking Algorithms

4.1 Prediction

Because in general, tracking will be done, based on the aggregate attributes position, volume and length, we want to make sure these attributes are correct in the computed prediction. We use linear extrapolation for computing the predictions:

$$\begin{aligned} F_{n+1}^* &= F_n + \Delta F_{n-1} \\ \Delta F_{n-1} &= F_n - F_{n-1} \end{aligned} \tag{4.1}$$

Each attribute set has functions Add and Subtract, in which the aggregate attributes are added, respectively subtracted. See for example Table 4.1, in which a prediction is computed, given two features. The feature in frame $n - 1$ has volume 6, length 8 and centre of gravity $(6, 5, 4)$. The feature in frame n has volume 9, length 5 and centre of gravity $(1, 2, 3)$. We would then expect that the prediction to frame $n + 1$ has volume 12 and length 2, and that the centre of gravity is at $(-4, -1, 2)$.

Using the above formula, we first compute: $\Delta F_{n-1} = \text{Subtract}(F_n, F_{n-1})$, which has volume $9 - 6 = 3$, length $5 - 8 = -3$, and COG $(1, 2, 3) - (6, 5, 4) = (-5, -3, -1)$.

Of course, this is only an intermediate result, and does not represent a real feature. The length of the skeleton is negative, and so could be the volume. However, the attributes are stored in the Skeleton class, as if it were a real skeleton feature.

The prediction F_{n+1}^* is computed using the formula $\text{Add}(F_n, \Delta F_{n-1})$, and has the following attributes: volume $9 + 3 = 12$, length $5 + -3 = 2$, and centre of gravity $(1, 2, 3) + (-5, -3, -1) = (-4, -1, 2)$. Which is exactly as expected.

The structure of the predicted skeleton is copied from the last one, so the prediction $Ftr(n+1)$ has the same structure as $Ftr(n)$. The complete skeleton is translated in such a way that the centre of gravity matches with the predicted centre of gravity. The attributes volume and length are adjusted manually. Thus, in the prediction, these attributes do not correspond with the skeleton graph itself. Therefore, also, the icon that is used to visualise the prediction does not exactly match the attributes that are used for the computations.

Feature	Volume	Length	COG
F_{n-1}	6	8	(6, 5, 4)
F_n	9	5	(1, 2, 3)
ΔF_{n-1}	3	-3	(-5, -3, -1)
F_{n+1}^*	12	2	(-4, -1, 2)

Table 4.1: The attributes of two skeleton features and the resulting prediction.

It would be hard to change the skeleton graph in such a way that the size of the skeleton graph would match the volume and length attributes. We would have a structure of the graph, that is copied from the last feature, and we would have to adjust the geometry of the graph, in such a way, that volume and length would also match the computed attributes. The attributes of all graph nodes, both the position and the DT, would have to be adjusted for this.

If the length of the edges is adapted by a certain factor, then of course the volume of the graph changes also, not linearly, but quadratically, and the centre of gravity changes. When the volume has to be corrected, the thickness of the edges, that is, the DT of the nodes, has to be adjusted. Should this adjustment be constant over all nodes, or relative to the current DT? For the conical edges, the volume is proportional to the square of the thickness, but for the end nodes, the volume is proportional to the third power of the DT. By adjusting the volume, the centre of gravity could also change again. Should we go through all this trouble, merely for the visualisation of the prediction? As said before, the computations are done with the correct, computed, values. We have therefore decided not to change the geometry of the skeleton graph.

4.2 Merge

When merging two or more features, the following procedure is executed for making the prediction. From the list of features that have to be merged, the first feature is removed. From the remaining features in the list, it is determined, which is nearest to the first, removed, feature. This feature is also removed from the list, and next, these two features are connected. The resulting feature is added to the list, and the procedure is repeated, until there is only one feature in the list. This feature is then the complete, merged, feature, and this is returned as result from the procedure. See also Figure 4.1. Connecting two features is performed as follows: all nodes in both graphs are traversed and the shortest distance between any two nodes is computed. Between the two nearest nodes, a new edge is added. This edge is also used for computing the skeleton attributes.

4.3 Topology Comparison

A lot of research has been done into graph comparison. We have studied, among others, a number of methods, described by B.T. Messmer and H. Bunke [5, 4, 6], concerning error-tolerant and error-correcting subgraph isomorphism detection, for example, using decision trees. Because of the high time and/or space com-

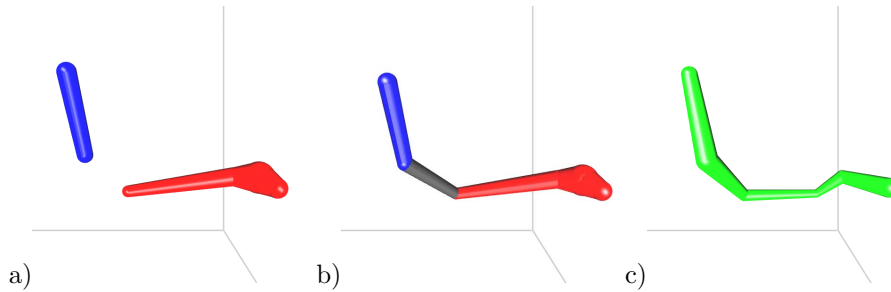


Figure 4.1: a) Two features before the merge event. b) The predicted merged feature, with the extra edge in grey. c) The candidate feature.

plexity of these algorithms, we have created our own, more efficient, special-purpose graph comparison algorithm.

We do not use the position and size of the skeleton graphs in these functions, because we already have correspondence functions for position and size. We do not use the complete graph in the comparison, because the geometrical nodes in the graph depend on tolerances, specified by the user. Because of this, it is possible that, for example, if the curvature of a skeleton changes slightly, a curve node is added or deleted. Therefore, we have decided to use only the topology of the graphs for comparison. We have created two correspondence functions. The first correspondence function is a boolean test, that is, it returns true, if the topologies are the same and false, if they differ. The second function returns the number of topological differences, or more precisely, the change in the number of loops and the number of edges.

In some cases it is very easy to see whether two topologies differ, for example, when the number of topological nodes in both graphs is different. Therefore, we have divided the topology comparison algorithm into three functions. The first two are relative simple functions, which can quickly give a negative result. If neither of these functions can give a final answer, the third function is executed, which will give the definitive and exact answer, by recursively traversing both graphs.

In the first test the number of nodes of each topological type (end, junction, loop), and the number of edges are counted. If these numbers do not match between both graphs, the topologies will most certainly not match either. In most cases, when the topology differs, this test will detect that. However, there are cases where it will not. In Figure 4.2 are two graphs, which have the same number of edges, end nodes, and junction nodes.

However, there is an easy way to make a distinction between the two graphs. In the graph on the left, each junction node is connected to an end node. In the graph on the right, the junction node in the middle is only connected to other junction nodes. The second test is based on this distinction. The test counts the number of connections of each type, such as end–end, end–junction, and so on. But also this test cannot give a definitive answer, whether the topologies are the same. See Figure 4.3 for two graphs that are not distinguished by this

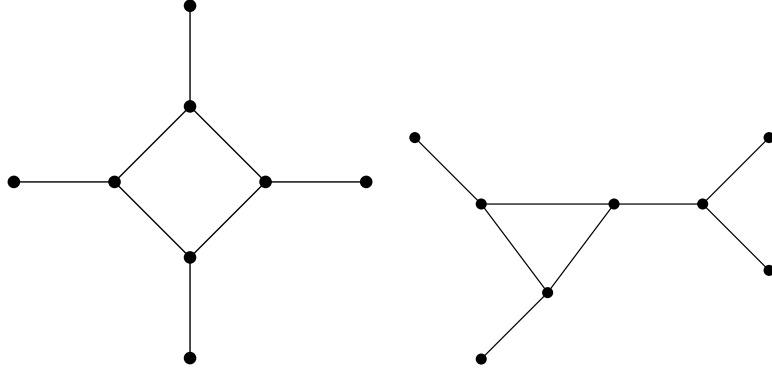


Figure 4.2: Two graphs which can not be differentiated by the first topology test.

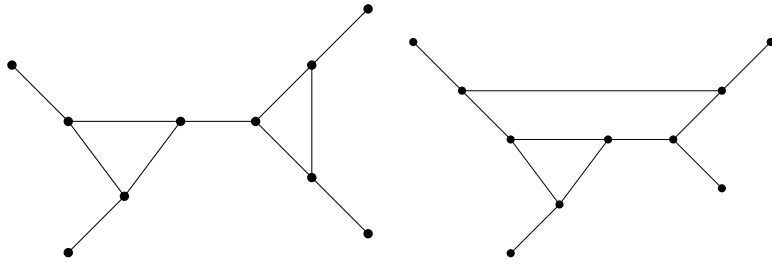


Figure 4.3: Two graphs which can not be differentiated by the second topology test.

second test.

These are, of course, very difficult cases, which may never occur in a real data set. However, the third and final test will always give a definitive answer, also for cases, as difficult as this. This third test recursively traverses all nodes in both graphs, and tries to match for each node, the node type and the number of edges from the node. If these attributes are the same in both graphs, both nodes are marked and a next node is visited. When all nodes have been marked, the graphs are topologically identical.

When two graphs are compared, using this algorithm, these graphs have already passed the first two tests, therefore it is already known that the numbers of nodes and edges, and the types of connections are the same.

First, the numbers of topological nodes of each type are counted, and the node type that occurs least frequently is determined. Next, a node of this type is selected in the first graph.

Now we want to find the corresponding node in the second graph. We try to match each node of the same node type in the second graph, with the selected node in the first graph.

If, on comparison, we find the nodes to be similar, both nodes are marked and a next node is chosen in each graph. Then the comparison is repeated.

When the nodes are compared, not only the node type and the number of edges

are compared, but also the number of marked neighbouring nodes. If all these attributes match, both nodes are marked with the current iteration number. Otherwise, the current iteration is aborted and returns false.

Next, in the first graph, all outgoing edges from the selected node are followed to the next (neighbouring) topological node (outer for-loop). If that node was already marked, the edge is skipped, and the next edge is followed. If there are no more unmarked neighbouring topological nodes, the current iteration is finished and returns true.

If an unmarked neighbouring topological node is found, then also in the second graph an unmarked neighbouring topological node must be found (inner for-loop). When such a node is found in the second graph, the comparison is repeated for these two nodes.

If the result of this comparison is true, that means, the nodes are similar, then the outer for-loop is continued and the next edge in the first graph is followed.

However, if the result of the comparison is false, then the nodes are not similar and the inner for-loop is continued, searching for another unmarked neighbouring topological node.

If no similar node is found in the second graph, both for-loops are terminated, and the current iteration is aborted and returns false. But before this, all nodes in both graphs that have been marked in the current and later iterations, must be unmarked.

The pseude-code for this algorithm is described in Appendix B.

Concluding, we can say the following. When the topologies of two graphs are different, in most cases this will be detected by the first test. Actually, in our data set, all of the more than 200 topological events were found in the first test. But when the topologies of the graphs are equivalent, all three tests have to be run. Most of the time, the topology of the features will not change, so all tests will be run very frequently. Therefore, the tests must be very time efficient. Fortunately, most of the features are topologically very simple, so the tests can be done very quickly.

For detection of topological events, we first use these functions to determine if an event has occurred. If so, we try to determine what type of event has occurred.

Euler's formula,

$$V - E + F = C - H, \quad (4.2)$$

makes it easy to compute the number of loops in a graph. In Equation 4.2, V is the number of nodes, E is the number of edges, and H is the number of holes. F is the number of faces, and can be set to 0. C is the number of connected objects, which is in our case always 1. Substitution gives a formula for the number of holes (or loops) in a graph:

$$H = 1 - V + E \quad (4.3)$$

If the number of loops has changed, we call the event a *loop event*. If not, but the numbers of edges and nodes have changed, the event is called a *junction event*. If neither of these has occurred, the event is simply called a *topo event*. If the graphs in Figure 4.2 would be compared, this would result in a topo event.

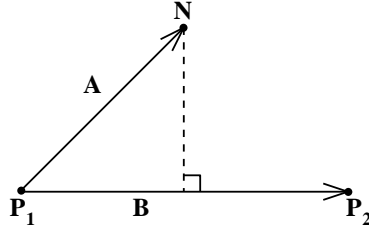


Figure 4.4: Computing the distance from a node to an edge.

4.4 Neighbourhood Criterion

We have created a number of neighbourhood criteria for skeleton graphs. The neighbourhood criteria determine how many features have to be tested for a merge event. We only want to test features that are close to each other, for a possible merge event. But “close” can be defined in a number of ways. With ellipsoid features, described by volume integral attributes, there is not much of a choice, because we only have one position attribute per feature. However, with features described by skeleton graphs, each node has a position attribute, therefore we can determine the distance between two features much more accurately. Not only does this save computing time, because there are less features to be tested, but also the results will be more reliable.

4.4.1 Skeleton COG distance

The easiest and least accurate skeleton neighbourhood distance measure is the skeleton COG distance. This measure returns the distance between the centres of gravity of the two skeleton graphs, and is therefore, in principle, very similar to the ellipsoid position distance measure. However, the volume integral attributes are more accurate than the skeleton graph attributes, therefore the skeleton COG distance will perform less than the ellipsoid distance measure.

4.4.2 Node-to-node distance

The node-to-node distance measure returns the shortest distance between any node in the first graph to any node in the second graph. In formula:

$$D_{n-n} = \min \|N_1 - N_2\| \quad (4.4)$$

4.4.3 Node-to-edge distance

The node-to-edge distance measure returns the shortest distance between a node in one graph to an edge in the other graph. See Figure 4.4. The distance from a node to an edge is computed as follows. The projection of a vector \mathbf{A} onto a vector \mathbf{B} is:

$$\frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{B}\|^2} \mathbf{B} \quad (4.5)$$

The length of this projection is:

$$\frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{B}\|} \quad (4.6)$$

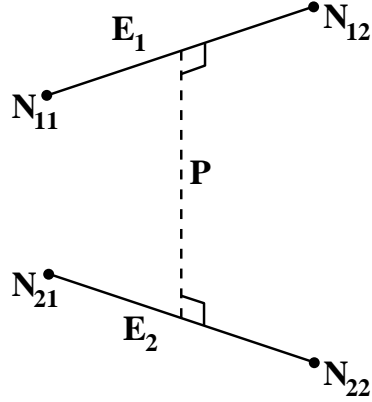


Figure 4.5: Computing the distance between two edges.

Let \mathbf{P}_1 and \mathbf{P}_2 be the end nodes of the edge, and \mathbf{N} the single node. Then, the node-to-edge distance can be calculated by choosing:

$$\begin{aligned}\mathbf{A} &= \mathbf{N} - \mathbf{P}_1 \\ \mathbf{B} &= \mathbf{P}_2 - \mathbf{P}_1\end{aligned}\quad (4.7)$$

The relative position of the projection on the edge is:

$$f = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{B}\|^2}\quad (4.8)$$

However, we have to make sure the projection is on the edge:

$$\begin{aligned}\text{if } f < 0 \text{ then } f &= 0 \\ \text{if } f > 1 \text{ then } f &= 1\end{aligned}\quad (4.9)$$

Then, the distance from the node to the edge is:

$$D_{n-e} = \|\mathbf{A} - f \mathbf{B}\|\quad (4.10)$$

4.4.4 Edge-to-edge distance

The edge-to-edge distance measure returns the minimal distance between any two edges from two skeleton graphs. See Figure 4.5. The distance between two edges is computed as follows. Let the positions of the nodes of the first edge be \mathbf{N}_{11} and \mathbf{N}_{12} and the positions of the nodes of the second edge \mathbf{N}_{21} and \mathbf{N}_{22} . Then,

$$\begin{aligned}\mathbf{E}_1 &= \mathbf{N}_{12} - \mathbf{N}_{11} \\ \mathbf{E}_2 &= \mathbf{N}_{22} - \mathbf{N}_{21}\end{aligned}\quad (4.11)$$

with \mathbf{E}_1 and \mathbf{E}_2 indicating the two edges. Compute the vector \mathbf{P}_\perp perpendicular to both edges, using the cross-product:

$$\mathbf{P}_\perp = \mathbf{E}_1 \times \mathbf{E}_2\quad (4.12)$$

If \mathbf{E}_1 and \mathbf{E}_2 are parallel, the cross-product \mathbf{P}_\perp will be $\mathbf{0}$. In that case, we choose a random vector perpendicular to \mathbf{E}_1 :

$$\mathbf{P}_\perp = \begin{bmatrix} -\mathbf{E}_{1,y} \\ \mathbf{E}_{1,x} \\ 0 \end{bmatrix} \quad (4.13)$$

We create a parameter description of both edges and the connecting line by:

$$\begin{aligned} \mathbf{L}_1 &= \mathbf{N}_{11} + s \mathbf{E}_1 \\ \mathbf{L}_2 &= \mathbf{N}_{21} + t \mathbf{E}_2 \\ \mathbf{L}_c &= \mathbf{L}_2 - \mathbf{L}_1 = \mathbf{N}_{21} - \mathbf{N}_{11} + t \mathbf{E}_2 - s \mathbf{E}_1 \end{aligned} \quad (4.14)$$

The line \mathbf{L}_c should be parallel to the vector \mathbf{P}_\perp , so we can solve s and t from these equations by assuming that

$$\frac{\mathbf{L}_{c,x}}{\mathbf{P}_{\perp,x}} = \frac{\mathbf{L}_{c,y}}{\mathbf{P}_{\perp,y}} = \frac{\mathbf{L}_{c,z}}{\mathbf{P}_{\perp,z}} \quad (4.15)$$

We need two equations to solve s and t . If there are no zeros in the vector \mathbf{P}_\perp , we can use two of the following three equations:

$$\begin{aligned} \mathbf{L}_{c,x}/\mathbf{P}_{\perp,x} &= \mathbf{L}_{c,y}/\mathbf{P}_{\perp,y} \\ \mathbf{L}_{c,x}/\mathbf{P}_{\perp,x} &= \mathbf{L}_{c,z}/\mathbf{P}_{\perp,z} \\ \mathbf{L}_{c,y}/\mathbf{P}_{\perp,y} &= \mathbf{L}_{c,z}/\mathbf{P}_{\perp,z} \end{aligned} \quad (4.16)$$

If there are zeros in the vector \mathbf{P}_\perp , we can use one or two of the following equations:

$$\begin{aligned} \mathbf{L}_{c,x} &= 0 && \text{(if } \mathbf{P}_{\perp,x} = 0) \\ \mathbf{L}_{c,y} &= 0 && \text{(if } \mathbf{P}_{\perp,y} = 0) \\ \mathbf{L}_{c,z} &= 0 && \text{(if } \mathbf{P}_{\perp,z} = 0) \end{aligned} \quad (4.17)$$

We write these equations in another form, to be able to solve s and t . For example, the first equation in Equation 4.16 can be written as:

$$\begin{aligned} s \left(\frac{\mathbf{E}_{1,y}}{\mathbf{P}_{\perp,y}} - \frac{\mathbf{E}_{1,x}}{\mathbf{P}_{\perp,x}} \right) + t \left(\frac{\mathbf{E}_{2,x}}{\mathbf{P}_{\perp,x}} - \frac{\mathbf{E}_{2,y}}{\mathbf{P}_{\perp,y}} \right) + \\ \left(\frac{\mathbf{N}_{21,x} - \mathbf{N}_{11,x}}{\mathbf{P}_{\perp,x}} - \frac{\mathbf{N}_{21,y} - \mathbf{N}_{11,y}}{\mathbf{P}_{\perp,y}} \right) = 0 \end{aligned} \quad (4.18)$$

The two equations that are chosen from Equation 4.16 and Equation 4.17 can be rewritten in the following form:

$$\begin{bmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} = \mathbf{0} \quad (4.19)$$

To solve this, we use again the cross-product:

$$\mathbf{sol} = \begin{bmatrix} v_{11} \\ v_{12} \\ v_{13} \end{bmatrix} \times \begin{bmatrix} v_{21} \\ v_{22} \\ v_{23} \end{bmatrix} \quad (4.20)$$

and return to homogeneous coordinates:

$$\mathbf{sol}_h = \begin{bmatrix} \mathbf{sol}_x/\mathbf{sol}_z \\ \mathbf{sol}_y/\mathbf{sol}_z \\ 1 \end{bmatrix} \quad (4.21)$$

Now we have:

$$\mathbf{sol}_h = \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} \quad (4.22)$$

However, we have to make sure that the resulting line connects two points on the edges, therefore

$$\begin{aligned} \text{if } s < 0 \text{ then } s &= 0 \\ \text{if } s > 1 \text{ then } s &= 1 \\ \text{if } t < 0 \text{ then } t &= 0 \\ \text{if } t > 1 \text{ then } t &= 1 \end{aligned} \quad (4.23)$$

Finally, we compute the nearest points \mathbf{NP} and the minimal distance D_{e-e} between two edges:

$$\begin{aligned} \mathbf{NP}_1 &= \mathbf{P}_{11} + s \mathbf{E}_1 \\ \mathbf{NP}_2 &= \mathbf{P}_{21} + t \mathbf{E}_2 \\ D_{e-e} &= \|\mathbf{NP}_2 - \mathbf{NP}_1\| \end{aligned} \quad (4.24)$$

4.5 Performance

I have done a number of tests to measure the performance of the tracking algorithm, using the different attribute sets. In particular, I have compared tracking with the ellipsoid feature descriptions, to tracking with the skeleton graph feature descriptions.

On a SGI Octane with one 225 MHz R10000 processor and 768 Mb of RAM, I have tracked the complete data set, consisting of 91 frames in a number of iterations, using either the ellipsoid feature descriptions or the skeleton graph feature descriptions. The parameters used are the following:

Minimum path length: 4

Position tolerance: 15, weight 2

Volume tolerance: 0.5, weight 1

The times needed to track the entire data set, are in Table 4.2. Tracking with the skeleton attributes takes about 6 times longer than with the ellipsoid attributes. In all cases the percentages of solved features are very close to each other, varying from 88.46% to 89.05%.

The differences in the times needed to track the data set, cannot be explained by saying that the skeleton graph attribute set is more complicated than the ellipsoid attribute set. When features have multiple attribute sets, predictions are calculated for all attribute sets. In this data set, the features have both ellipsoid and skeleton graph attributes. All calculations are therefore performed on both types of attributes, so that should not result in the difference in tracking time.

	Ellipsoid attributes	Skeleton attributes
5 iterations	45.2	255.1
10 iterations	115.8	732.1

Table 4.2: The time needed to track the entire 91 frame data set, in seconds.

However, the differences can be explained by examining the tracking algorithm more closely. In Figure 5 of the paper in Part 1 [13], it can be seen, that tracking with the skeleton graph attributes, performs worse than tracking with ellipsoid attributes, when using small tolerances. This is mainly because the skeleton graph reconstructed volume is not very accurate.

Normally, when searching for correspondences, features that are already resolved, are not tested again. This means, when less features are resolved, more features remain to be resolved, therefore, more correspondences have to be tested. This explains the above results.

We can test this theory by setting an option in the tracking algorithm to always test all correspondences, even with features that have already been resolved. This takes, of course, much longer, but at least, the times are almost identical for both the ellipsoid and the skeleton graph attributes.

Chapter 5

User Interface and User Interaction

5.1 Dual-view principle

To visualise the data in our program, we make use of two linked and synchronised views of the data set. The first view, the *feature viewer*, is a 3D interactive view of the iconic representations (see Section 5.5) of the features in one frame of the data set. It is possible to rotate and zoom the view, and to play through the frames. See for example the bottom right of Figure 5.1. The second view, the *graph viewer*, gives an abstract 2D view of the complete data set. The data set is represented by a graph, the *event graph* [10], in which the features are represented by nodes (see Section 5.4), and connections between features are represented by edges between the nodes. The frame numbers are on the x-axis, and the feature numbers are on the y-axis. The evolution of a feature through time is therefore represented by a path in the graph from left to right. See for example the left of Figure 5.1.

5.2 Interaction modes

There are two modes of interaction with the viewers, *play mode* and *selection mode*. In play mode, using the control panel, the user can play or skip through the frames, or jump to a specific frame. In the feature viewer, only the icons of the current frame are displayed. In the graph viewer, a box is drawn around the current frame, and the view is centred on that frame. Thus, when playing, it seems as if the graph slides through the view from right to left. It is also possible to jump to a specific frame, by clicking on the corresponding frame number on the x-axis of the graph viewer. In Figure 5.1 is an example of the two viewers in play mode. From the graph viewer on the left, and from the control panel on the top right, it appears that the features, which are visualised in the feature viewer (bottom right), are from frame 10. Naturally, in both viewers, the same colours are used for the same features.

In selection mode, the user can select certain features. By clicking on the feature icons in the feature viewer or on the nodes in the graph viewer, the cor-

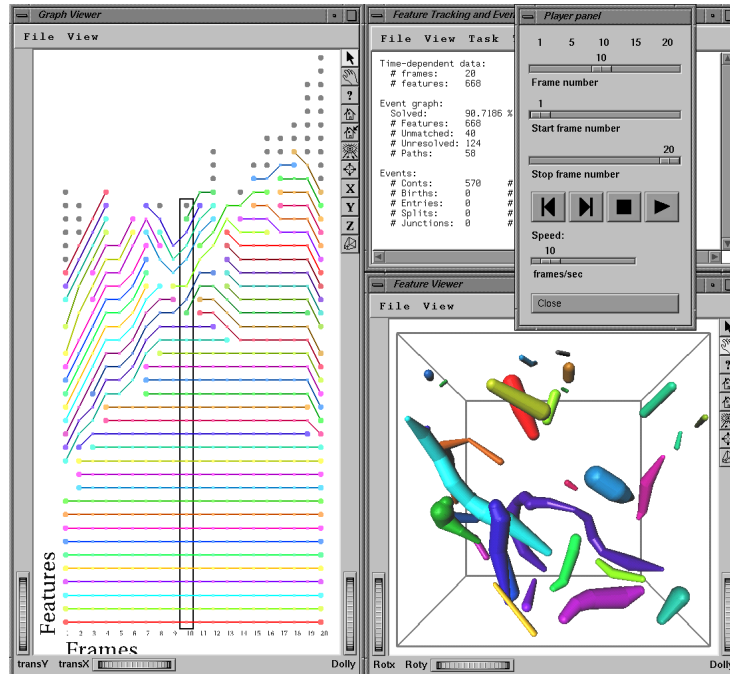


Figure 5.1: Playing through the data set.

responding features are selected. By clicking on an edge in the graph viewer, the entire corresponding path is selected. When a feature is selected, the attributes of the feature are printed to the screen and the feature is highlighted in both viewers. When a path is selected, the attributes of the path are printed, and all features in the path are highlighted. In Figure 5.2, a number of features and one entire path have been selected. The features have been highlighted in both viewers and the attributes have been printed to the screen. For each individually selected feature, the complete attribute set is printed. For a selected path, only the frame and feature numbers of the individual features are printed.

Selection of a single path gives the user another very useful tool, because the entire evolution of one object can be viewed in one image. See Figure 5.3 for an example.

In selection mode, all selected features are highlighted in the feature viewer. But when a selection has been made and play mode is chosen again, then only features from the current frame are displayed, with the selected features in that frame, if any, highlighted. An example is shown in Figure 5.4. Of the selected paths, only the features from the current frame are shown (and highlighted) in the feature viewer. Also the individually selected features from the current frame are highlighted. All other features from the current frame are not highlighted.

5.3 Highlighting

Highlighting a feature can be implemented in a number of ways. In the graph viewer, the selected features and paths are displayed in colour, while the rest

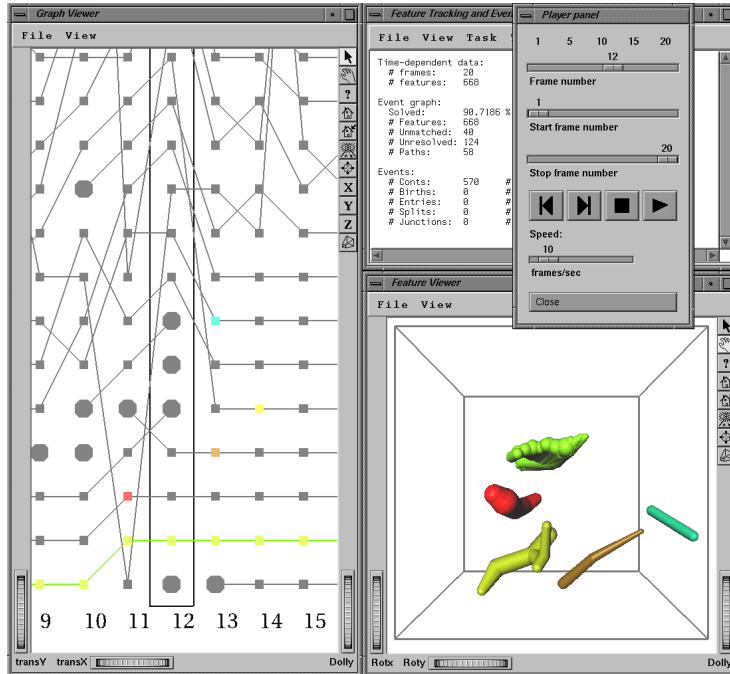


Figure 5.2: Selection of a number of features.

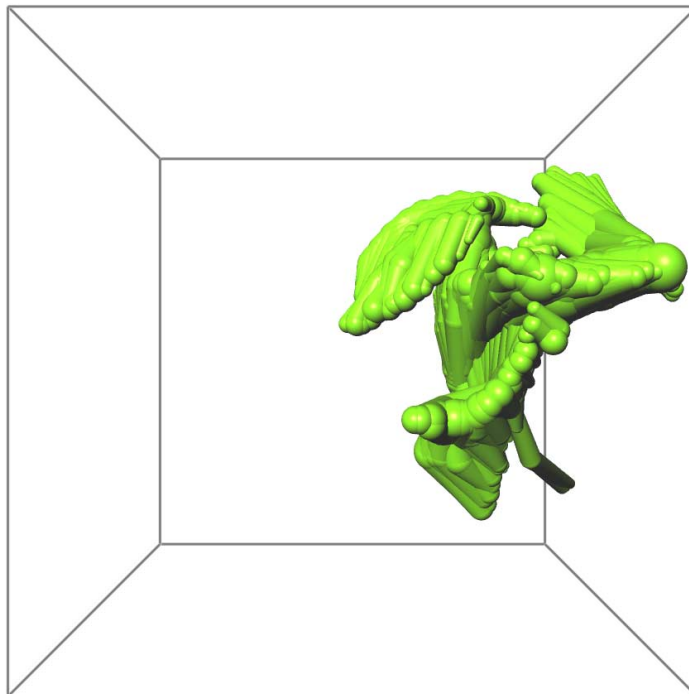


Figure 5.3: Highlighting an entire path.

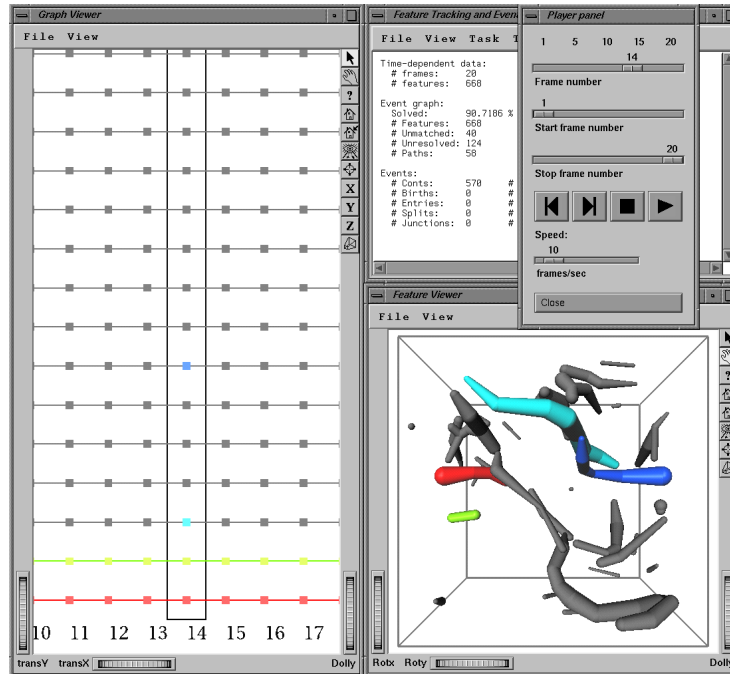


Figure 5.4: A selection of features during playing.

of the graph is displayed in grey. In the feature viewer, different highlighting methods are used, depending on the current interaction mode. In selection mode, only the selected features are displayed, which means that all features that are not selected, are made invisible. In play mode, however, only the features from the current frame are displayed. Selected features in that frame are then displayed in colour, all other features in the frame are displayed in grey. Other methods could, of course, also be used. Features that are not highlighted could be made invisible, displayed in grey, or displayed transparently.

5.4 Graph Icons

All nodes in the graph viewer are visualised by small icons (see Figure 5.5). These icons represent the types of incoming and outgoing events for the corresponding feature. For example, when an entry event has occurred, that is, a feature has entered the data set through the boundary, an icon with a small arrow, pointing inwards, is displayed.

The user can inspect all events of a certain type. All event types are listed in the menu of the graph viewer. When one event type is selected from this menu, all icons of the corresponding event type are highlighted in the event graph, and the rest of the graph is greyed out. This way, it is easy to select a number of features before or after the event, which are then highlighted in the feature viewer, and to check how well the event detection has been performed.



Figure 5.5: The icons for different events.

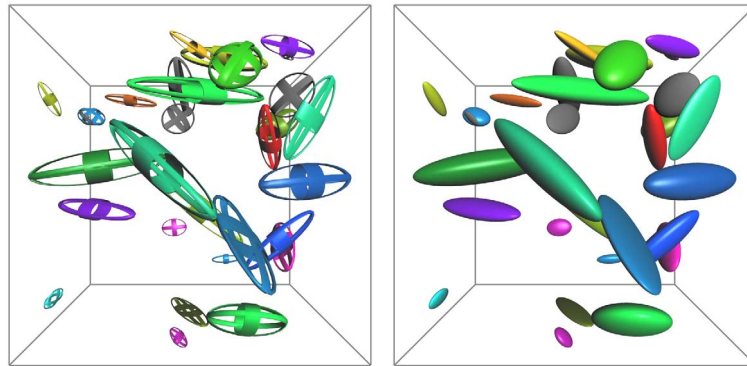
5.5 Feature Icons

For visualisation of the features in the feature viewer, we use parametric icons. The attributes of the features are mapped onto the parameters of the icons. Often, different types of icons can be used for visualising an attribute set. Furthermore, when a feature is described by a number of attribute sets, the icons for each attribute set can be used. For example, when a feature data set consists of both volume integral descriptions and skeleton graph descriptions, then ellipsoid icons can be used to map the volume integral attributes and skeleton icons can be used to map the skeleton graph attributes. While visualising the data set, it is possible to switch between the different iconic representations. In Figure 5.6, four different representations of the same data set are shown, two with ellipsoid icons and two with skeleton icons.

5.6 Attribute Profiles

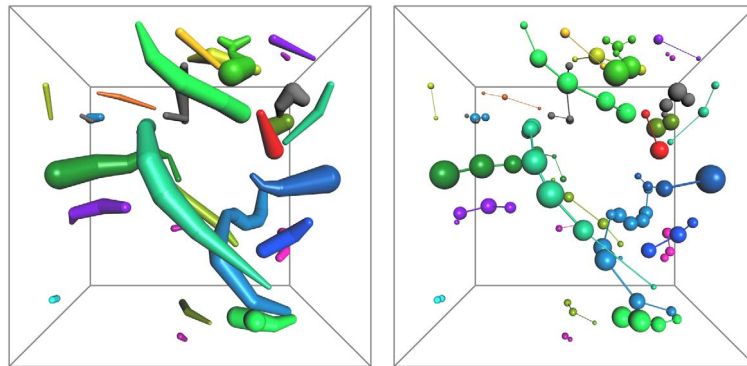
To test how well the feature tracking has been performed, or to investigate the variation of the features' attributes in time, it can be very useful to display attribute profiles in the graph viewer. Normally, the y-axis of the graph viewer has no direct relation to any feature attribute. But we can display the graph viewer, with on the y-axis of the graph, any scalar attribute of the features. So far, we have created the appropriate functions for the volume integral and skeleton graph attribute sets. With the volume integral attribute set, we can plot the X, Y and Z coordinates of the position, as well as the length of the position vector, and the volume. For the skeleton graph attribute set, we can plot the COG X, Y and Z coordinates, the length of the COG vector, the reconstructed skeleton volume and skeleton length, and the number of graph nodes, graph edges and topological graph edges.

We can determine from these graphs, whether the evolution of a feature is more or less continuous in its attributes. In Figure 5.7 is an example of an attribute profile with on the y-axis the length of the ellipsoid position vector of



(a) The 'banded ellipsoid' icon.

(b) The 'solid ellipsoid' icon.



(c) The 'tube skeleton' icon.

(d) The 'sphere skeleton' icon.

Figure 5.6: Four different iconic representations of the same data set.

the features.

When a path in the graph viewer has been selected when choosing to create an attribute plot, the attributes that are plotted for the selected path are also printed to the screen. This makes it easy to copy the exact values of the attributes into a suitable program, to create a graph for a single path.

In Figure 5.8 is an example of such a graph. The X, Y and Z coordinates of an ellipsoid fitting are plotted on the y-axis of the graph. On the x-axis are the frame numbers. In frame 12, the objects splits in two parts. In Figure 5.9 is another example of the same path. Here, the reconstructed skeleton volume is plotted against the frames. We can clearly see that the total volume before and after the split event is more or less continuous.

5.7 User-guided tracking

In our program, feature tracking can be done in three different modes.

- Automatic mode
- Semi-automatic mode
- Stepwise mode

In the automatic mode, all paths are searched without any user interaction. First, the parameters have to be specified. But when the tracking process has been started, the algorithm continues until no more paths can be found.

In semi-automatic mode, the algorithm is paused after each path that is found. The newly found path is then displayed in both the graph and the feature viewer. At that time it is possible to change the tracking parameters for the rest of the process.

In the stepwise mode, the algorithm pauses for each connection that is tried. First, the path is shown with the prediction to the next frame plus all possible candidates in that next frame (see Figure 5.10).

When a matching candidate feature is found, this feature is added to the path and the resulting path is displayed in the viewers. When no matching candidate is found, the path has apparently ended and the complete path is displayed in both viewers.

While tracking, the attribute plotting capabilities of the graph viewer can be very useful (see Figure 5.7). In the attribute plots, it can be easily seen, whether the paths that are found, are more or less continuous in their attributes. Furthermore, when features in the graph appear to belong to a path, but are not found by the algorithm, it is possible to visually inspect the features and print their attributes, to see why a correspondence has not been found. This way, the user can determine what tracking parameters should be used, for the correspondence to be found, whether the tolerances have to be raised, or perhaps another correspondence function has to be used.

For example, the cyan path in the bottom centre of Figure 5.7 seems to have missed a number of features at the beginning. When we zoom in on the box in the figure (see Figure 5.11), and view the 3D icons of the relevant features (Figure 5.12), we see that they should all belong to the path. However, when we plot the skeleton graph volume for these features (Figure 5.13a), it becomes

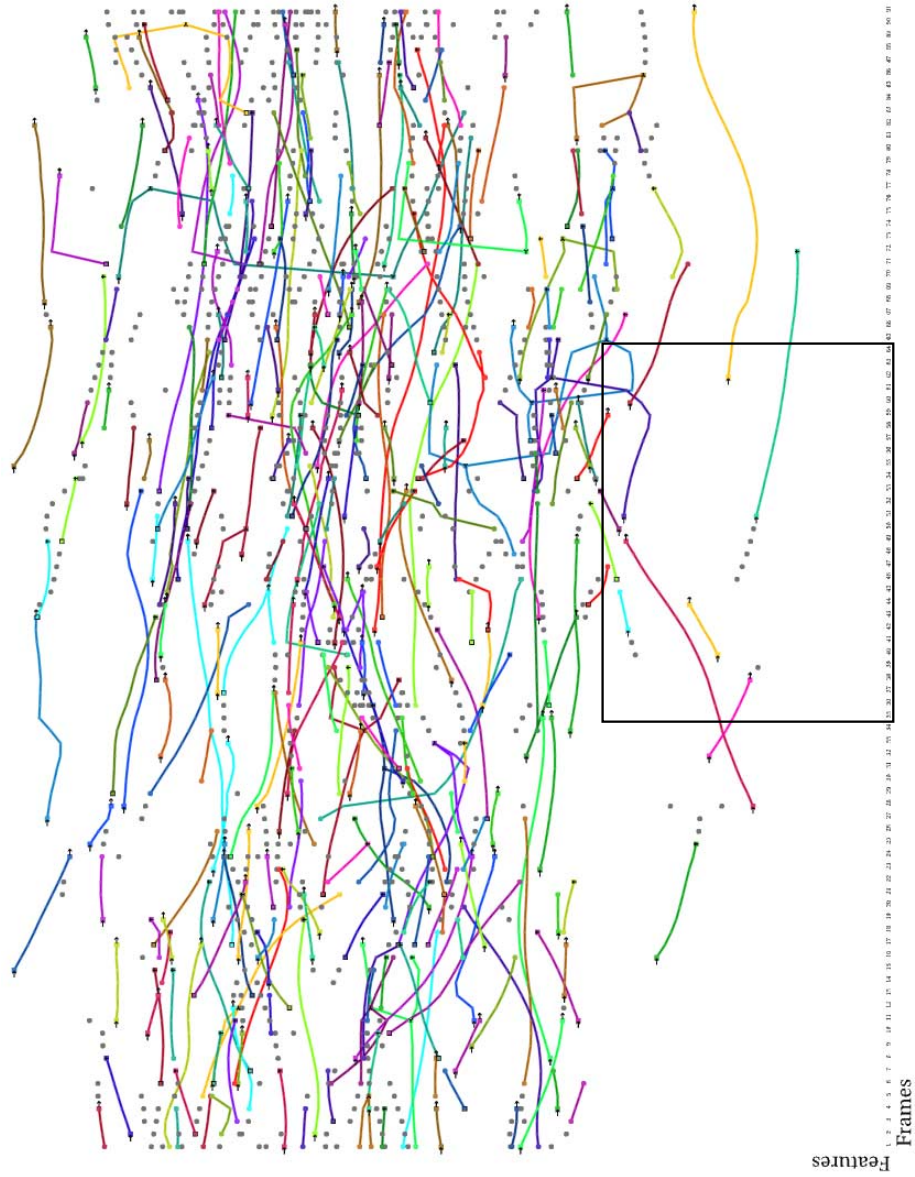


Figure 5.7: An attribute profile with the length of the position vector on the y-axis.

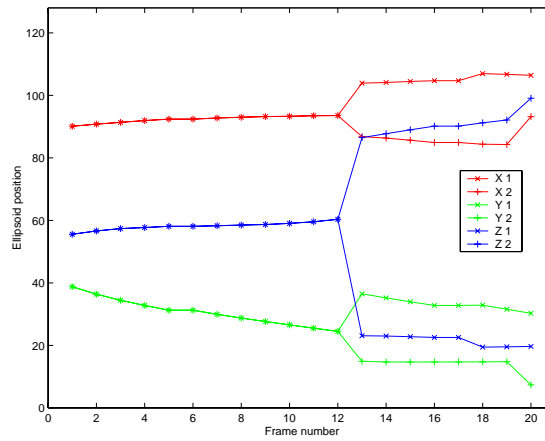


Figure 5.8: The coordinates of an ellipsoid fitting of an object, that splits in frame 12.

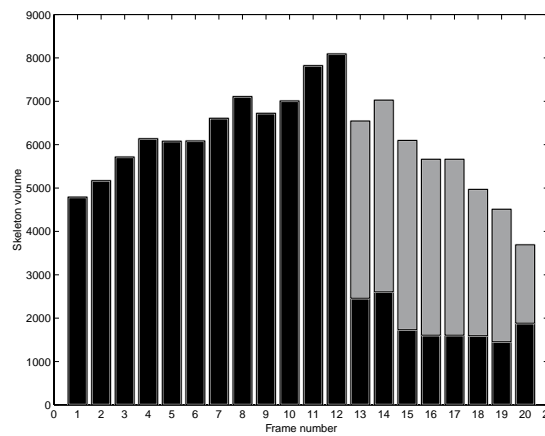


Figure 5.9: The reconstructed skeleton volume of an object, that splits in frame 12.

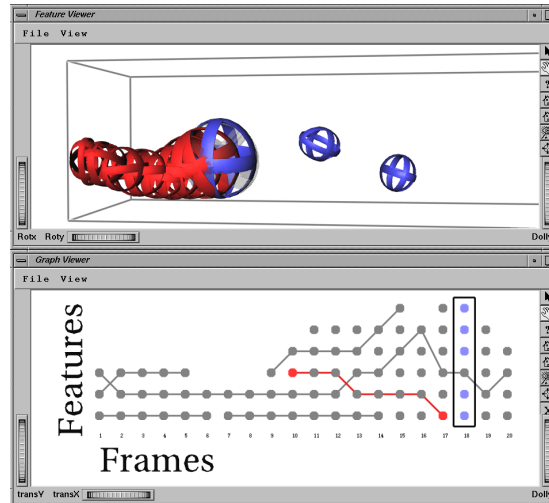


Figure 5.10: One step in the tracking process. The path, found so far, is in red, the possible candidates are in blue, and the prediction is transparent grey.

clear, why the correspondences have not been found. On the other hand, had we used the ellipsoid volume for tracking (Figure 5.13b), the entire path would have been found.

The user is not yet, but he should be in the future, able to select the features in the graph viewer, and manually create a connection between them. Then, of course, the user should also be able to manually delete a connection, if, on visual inspection, it appears that a correspondence has been found incorrectly.

5.8 Implementation

Both viewers were made using Open Inventor [14]. In the Feature Viewer the scene graph consists of a node “FrameIcons” and a node “Selection”. In the node “FrameIcons” the icons of the features in the current frame are stored. The icons are updated, each time another frame is viewed. Each icon has a unique name, “Feature:x,y”, with x and y the frame and feature numbers, so that the features can be identified upon selection. When a selection is made, the icons of the selected features are copied to the node “Selection”. Furthermore, the node “FrameIcons”, with all its children, is made invisible.

In the Graph Viewer, the scene graph is constructed similarly. There are a few extra nodes in the scene graph, for the axes and the text. But there is also a node “Selection”, and when a path or feature is selected in the graph, the selected node is copied here, and the rest of the graph is made grey. While playing through the frames, in the Feature Viewer, only the features in the current frame are displayed. Also when a selection has been made, the selected features in the current frame (if any), are displayed in colour, and the rest of the features are displayed in grey. This means the default colours of the features have to be overridden. This can be done by inserting a grey material as first node in the “FrameIcons”. By using the method “setOverride(TRUE)” on this node, all material nodes below this one in the scene graph are overridden.

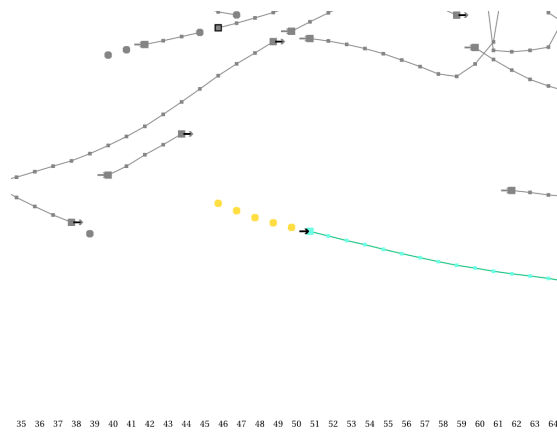


Figure 5.11: Zoomed in on the box in Figure 5.7.

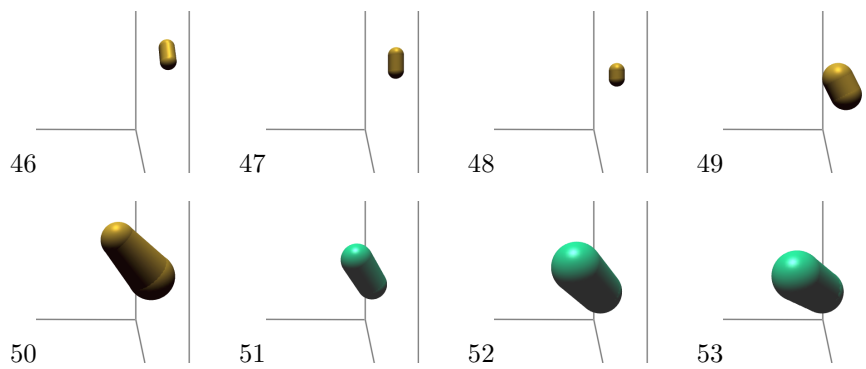


Figure 5.12: The relevant features, viewed in the 3D viewer.

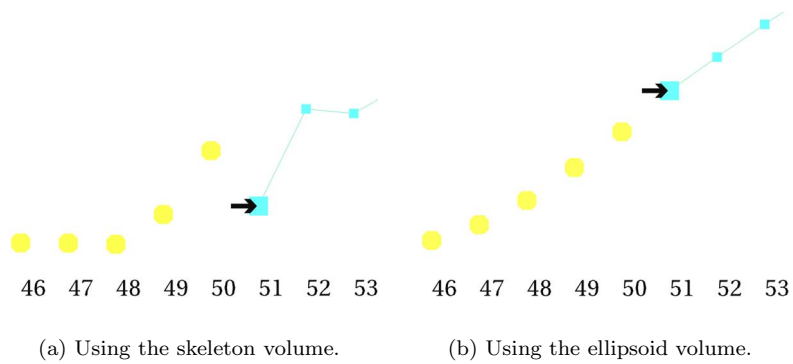


Figure 5.13: Two different plots of the volume of the selected features.

Chapter 6

Conclusions and Further Research

Skeleton graph descriptions of features have appeared to be very useful. We have developed the methods for the tracking of features, using the skeleton graph attributes. The results are very good. However, when volume integral attributes are available, these are more accurate, and have better continuity, and tracking with these attributes will give even better results. Especially, the reconstructed volume of the skeleton graphs can be very inaccurate. The volume of the skeleton graphs is computed using the DT, which is the minimal distance to the surface. Therefore, the volume reconstruction is inaccurate, for example, for flat shapes. Tracking on the topology is also possible, when it is known, that the topology does not change (much) during evolution.

The skeleton graph descriptions are also very useful for detecting split and merge events. Much more accurate neighbourhood criteria can be defined, when using skeleton graph attributes, than with volume integral attributes. Therefore, fewer features must be tested for possible events. Not only does this make the algorithm more efficient, but also the results are much more reliable.

Furthermore, the skeleton graph descriptions enable us to search for a completely new kind of event, the topological event. We have made it possible to detect loop and junction events.

Also, for visualisation, the skeleton graphs icons are very helpful, as they give much more shape information than the ellipsoid icons for the volume integral attributes.

The visualisation of time-dependent data sets has been improved. Initial experiments have shown the use of two simultaneous views on the data set to be very useful. Selection and picking, highlighting of selected objects in both views at the same time, and attribute profiles have appeared to be very valuable tools. For example, the attribute profiles have shown that the length of the position vector could very well be used for feature tracking.

A very important improvement to the current skeleton graph feature extraction, would be to compute a more accurate distance transformation. Currently, the minimal distance to the surface is computed, giving an inscribed circular cross-

section of the object, and resulting in an under-estimation of the volume, which can sometimes be substantial. Instead of the minimal distance, for example, the average distance could be used. Another possibility would be to use two or more distances to the surface, to create a more accurate contour description, such as an ellipse or polygon, instead of just one (minimal) distance.

User interaction capabilities could be further improved. A very useful function for user-guided tracking would be the possibility of correspondence editing. In the graph viewer, it should be possible to add correspondences (edges) between features, and to remove correspondences that have been found incorrectly.

It could be interesting, to develop algorithms for a more precise description of events, when using skeleton graph attributes. Not only topological events could perhaps be described more accurately, but also split and merge events could be refined, for example, by describing where a feature splits, or how two features merge.

Finally, another subject for further research would be the creation of another type of skeletons. For features such as shock waves or separation surfaces, the current skeleton graph description cannot be used. A completely new kind of feature description has to be developed for these surface skeletons.

Appendix A

Tracker Class Structure

The class GraphAS contains the following fields:

The total volume and the total length of the graph are stored in floating point variables. The centre of gravity of the graph, is stored in a floating point vector. The number of nodes and edges in the graph are stored as integers, as well as the number of topological edges.

```
float Length;           // Total length
float Volume;          // Total volume
floatVector COG;       // Centre of gravity
int NrNodes;           // Number of nodes
int NrEdges;           // Number of edges
int NrTopoEdges;      // Number of topological edges
```

In the list of Attributes are stored respectively:

- NrNodes times a GraphNodeAttr;
- NrEdges times a GraphEdgeAttr;
- NrTopoEdges times a TopoGraphEdge.

In the class GraphNodeAttr the following fields are stored:

The number and type of the node, and a list of edges to or from this node.

```
int NodeNr;            // The number of the node
list<GraphEdgeAttr*> Edges; // The edges to or from this node
NODETYPE NodeType;    // The type of node (END, REGULAR,
// JUNCTION, LOOP, CURVE, PROFILE)
```

The class SkeletonNode contains besides these attributes also a position and the value of the distance transformation at that position.

```
floatVector Pos;       // The position of the node
float DT;              // The DT of the node
```

The class `GraphEdgeAttr` contains the number of the edge and two pointers to the nodes of the edge:

```
int EdgeNr;  
GraphNodeAttr* Node1;  
GraphNodeAttr* Node2;
```

The class `TopoGraphEdge` has, besides the attributes of the `GraphEdgeAttr`, also a list of `GraphNodeAttr`. In this list are the nodes of the graph that are between the end nodes of this topological edge. All nodes in this list are therefore regular, not topological, nodes.

```
list<GraphNodeAttr*> IntraNodes;
```

Appendix B

Topology Comparison

The topology comparison algorithm, which was described in Section 4.3, is described here in pseudo-code. The function `TestGraphTopology` is called with the two graphs, which have to be compared. In this function, two similar nodes are selected and passed on to the function `RecursiveGraphCompare`, which recursively traverses all nodes in both graphs.

```
bool TestGraphTopology(graph1, graph2)
{
    // Number of nodes in each graph
    int NrNodes1 = graph1->NrOfNodes(),
        NrNodes2 = graph2->NrOfNodes()
    // Create an array for each graph to hold
    // the marked state of each node
    int Marked1[NrNodes1],
        Marked2[NrNodes2]
    GraphNode *node1, *node2
    int Iteration

    // Count the number of nodes of each type
    numEnd = graph1->CountNodesOfType(END)
    numJunction = graph1->CountNodesOfType(JUNCTION)
    numLoop = graph1->CountNodesOfType(LOOP)

    minNum = numEnd
    minType = END
    if (numJunction > 0 && numJunction < minNum) {
        minNum = numJunction
        minType = JUNCTION
    }
    if (numLoop > 0 && numLoop < minNum) {
        minNum = numLoop
        minType = LOOP
    }

    result = FALSE
}
```

```

foreach node1 in graph1->Nodes {
  if (node1->NodeType == minType)
    break
}
foreach node2 in graph2->Nodes {
  if (node2->NodeType == minType)
    if (RecursiveGraphCompare(node1, node2,
      Marked1, Marked2, NrNodes1, NrNodes2, 1) {
      result = TRUE
      break
    }
}
return result
}

bool RecursiveGraphCompare(node1, node2, Marked1, Marked2,
  NrNodes1, NrNodes2, Iteration)
{
  // If the nodes are similar
  if ((node1->NodeType == node2->NodeType) &&
    (node1->NumEdges == node2->NumEdges)) {
    if (node1->NumMarkedNeighbours() ==
      node2->NumMarkedNeighbours()) {
      // Mark the nodes with the current iteration number
      Marked1[node1->NodeNr] = Marked2[node2->NodeNr] = Iteration
      Found1 = FALSE
      // Traverse all outgoing edges
      foreach edge1 in node1->Edges {
        nextnode1 = edge1->NextTopoNode()
        if (Marked1[nextnode1->NodeNr])
          next edge1
        Found1 = TRUE
        Found2 = FALSE
        // Traverse all outgoing edges
        foreach edge2 in node2->Edges {
          nextnode2 = edge2->NextTopoNode()
          if (Marked2[nextnode2->NodeNr])
            next edge2
          Found2 = TRUE
          // Compare the next nodes
          if (RecursiveGraphCompare(nextnode1, nextnode2,
            Marked1, Marked2, NrNodes1, NrNodes2,
            Iteration + 1)) {
            result = TRUE
            next edge1
          }
        }
      }
    }
    else {
      result = FALSE
      next edge2
    }
  }
}
}

```

```
        if (Found2 == FALSE)
            result = FALSE
        if (result == FALSE)
            break
    }
    if (Found1 == FALSE)
        result = TRUE
}
else
    result = FALSE
}
else
    result = FALSE
// Unmark all nodes, marked in this and later iterations
if (result == FALSE) {
    UnMark(Marked1, Iteration)
    UnMark(Marked2, Iteration)
}
}
```


Bibliography

- [1] AVS. *AVS Release 5 User's Guide*. Advanced Visual Systems Inc., 1993.
- [2] G. Borgefors. Distance transformations in arbitrary dimensions. *Computer Vision, Graphics, and Image Processing*, 27(3):321–345, September 1984.
- [3] M. E. D. Jacobson. Generation of Skeleton Graphs for Shape Description in Feature Visualization. Master's thesis, Delft University of Technology, april 2000.
- [4] B.T. Messmer and H. Bunke. Error-correcting graph isomorphism using decision trees. *International Journal of Pattern Recognition and Artificial Intelligence*, 12(6):721–742, 1998.
- [5] B.T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):493–505, May 1998.
- [6] B.T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.
- [7] F. Reinders. *Feature-Based Visualization of Time-Dependent Data*. PhD thesis, Delft University of Technology, The Netherlands, 2001.
- [8] F. Reinders, M.E.D. Jacobson, and F.H. Post. Skeleton Graph Generation for Feature Shape Description. In W. de Leeuw and R. van Liere, editors, *Data Visualization 2000*, pages 73–82. Springer Verlag, 2000.
- [9] F. Reinders, F.H. Post, and H.J.W. Spoelder. Attribute-Based Feature Tracking. In E. Gröller, H. Löffelmann, and W. Ribarsky, editors, *Data Visualization '99*, pages 63–72. Springer Verlag, 1999.
- [10] F. Reinders, F.H. Post, and H.J.W. Spoelder. Visualization of Time-Dependent Data with Feature Tracking and Event Detection. *The Visual Computer*, 17(1):55–71, January 2001.
- [11] T. van Walsum. *Selective Visualization on Curvilinear Grids*. PhD thesis, Delft University of Technology, The Netherlands, 1995.
- [12] T. van Walsum, F.H. Post, D. Silver, and F.J. Post. Feature Extraction and Iconic Visualization. *IEEE Trans. on Visualization and Computer Graphics*, 2(2):111–119, 1996.

- [13] B. Vrolijk, F. Reinders, and F.H. Post. Feature Tracking with Skeleton Graphs. In F.H. Post, G.P. Bonneau, and G.M. Nielsen, editors, *Proceedings of the Dagstuhl Scientific Visualization Seminar, 22-26 May 2000*. Kluwer Academic Publishers, to appear in 2001. Submitted for publication.
- [14] J. Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*. Addison-Wesley Publishing Company, 1993. Open Inventor Architecture Group.