Structures and Algorithms for the Collaboration between Peers and their Application in *Solipsis*

A fully distributed peer to peer structure based on the 3D Delaunay triangulation

Diplomarbeit

Moritz Steiner
from Juan les Pins

submitted at

Corporate Communications Department
Professor E. Biersack
Institut Eurécom
Sophia-Antipolis
France

Lehrstuhl für Praktische Informatik IV
Professor W. Effelsberg
Fakultät für Mathematik und Informatik
Universität Mannheim
Germany

March 29, 2005

Acknowledgements

The work presented in this thesis could not have been possible without the support of essentially two persons:

Professor Effelsberg, Universität Mannheim, who gave me the idea of writing my *Diplomarbeit* in France, more precisely at the Institut Eurécom in Sophia-Antipolis. He established the contact with Professor Biersack.

Professor Biersack who was willing to supervise my *Diplomarbeit*, who proposed the subject and whose office door was always open to me during my stay here at Eurécom. The cooperation was so excellent that he is going to be the supervisor for my Ph.D. as well, again together with Professor Effelsberg.

I am deeply grateful to my grandfather Werner Hamel, who died in November 2004, when I was still working on this thesis. He financed my studies, from the first semester to their end today.

Abstract

This report proposes a promising solution for constructing scalable p2p networks based on the 3D Delaunay Triangulation (DT). The key idea of the design is to maintain for each node a DT of the neighbour nodes. While demonstrating scalability in a real system is not practical for the current work, we demonstrate the scalability of the 3D DT using simulation. The results obtained indicate that there are upper bounds on the time needed to join and on the average number of neighbours maintained by a peer. Therefore, the amount of bandwidth and processing requirement for each node is bound, independent of the total number of nodes in the system.

Contents

List of Figures

List of Algorithms

1 Introduction

A peer to peer (p2p) system is a network of peers that communicate with each other. A peer is an entity in the system, usually an application running on a device, or the user of such an application. All peers should have the same functionality and be of equivalent importance to the system; no single peer should be critical to its functionality. The characteristics of p2p systems can be summarised as following: [Oram 01]

- Peers should be able to freely offer services to other peers.
- The addressing system should be independent of lower layer network addressing systems.
- Peers should be assumed to be of variable connectivity.

This describes some of the essential features of p2p systems. The peers should have autonomy, i.e. be able to decide services they wish to offer to other peers. Peers should be assumed to have temporary network addresses. They should be recognised and reachable even if their network address has changed. A peer can join and leave the system at its own disposal. The peers cannot necessarily trust each other and rely completely on the behaviour of other peers because of their autonomy. That is why issues of scale and redundancy become much more important than in traditional (centralised or distributed) systems.
Peer to peer systems are constructed by connecting various computers (nodes) in a mesh-like fashion, forming a virtual network on top of the physical Internet. The term "overlay network" is thus often used to describe p2p systems. We will use the term p2p and overlay network interchangeably in this thesis.
Peer to peer networking is gaining more and more importance. Many new applications have ermerged (file sharing [Napster 99, Gnutella 00, eMule 02] and Internet telephony [Skype 03]). There exist two main models of p2p systems:

- The model made popular by Napster [Napster 99], the centralised directory model [Saroiu 02, Lv 02]. The peers of the community connect to a central directory where they publish information about the content they offer for sharing. Upon request from a peer, the central index will match the request with the best peer in its directory that matches the request. The best peer could be the one that is cheapest, fastest, or the most available, depending on the user's needs. Then a file will be transmitted directly between the two peers. This model requires some managed infrastructure, the directory server, which hosts information about all participants in the community. This can cause the model to show scalability limits because it requires larger servers when the number of requests increases, and more storage when the number of users increases. However, Napster's experience showed that – except for legal issues – the model was very strong and efficient.

- The flooding model [Saroiu 02, Lv 02] is different from the centralised directory model. This is a pure p2p model, in which no advertisement of shared resources occurs. Instead, each request from a peer is flooded to directly connected peers, which themselves flood their peers etc. until the request is answered or until the maximum number of flooding steps is reached. This model, which is used by Gnutella [Gnutella 00], requires a lot of network bandwidth and hence does not prove to be very scalable, but it is efficient in limited communities such as company networks. To circumvent this problem *super node* [Yang 03] client software, that concentrates many of the requests, has been developed. This requires less network bandwidth, at the expense of high CPU consumption. Caching of recent search requests and answers is also used to improve scalability.

This thesis proposes a fully distributed p2p architecture which attempts to solve the scalability problem based on the mathematical construct of the 3D Delaunay triangulation. The proposed architecture is called *3D Delaunay Triangulation Overlay Network* ($\mathcal{DTON}$). The main contribution of this thesis is to propose a resource efficient solution which needs no server at all, not even for login.
It can be used as infrastructure for a *Networked Virtual Environment* (NVE), to allow people to interact as they do in the real world, e.g. to speak (broadcast audio or video) to people interested in a common topic. We believe that a massive, persistent 3D virtual environment which allows millions of people to participate simultaneously may eventually happen on the Internet. There are many technical and architectural issues that need to be resolved before such a true cyberspace can be realised. The primary among these needs is a scalable architecture that handles large numbers of simultaneous users. The goal for this thesis is to devise a suitable p2p architecture for constructing a NVE that can scale to millions of users.

This thesis is organised as follows:
Section 2 shortly introduces the notion of NVE. Section 3 deals with partitioning the virtual world; two geometrical constructs are discussed, the Voronoi diagram and the Delaunay triangulation. Section 4 describes the data structures and algorithms developed for segmenting the virtual world. Section 5 explains the network protocol and points out some difficulties using in the 3D Delaunay Triangulation as structure of a fully distributed p2p network. Finally section 6 discusses the implementation of multicast trees using the Delaunay triangulation.

2 Networked Virtual Environments

Networked Virtual Environments (NVEs)[Knutsson 04, Hu 04] are computer-generated, synthetic worlds that allow simultaneous interactions of multiple participants. From the early days of SIMNET [Calvin 93], a joint project of the U.S. Army and Defense Advanced Research Project Agency (DARPA) between 1983 and 1990 for large scale combat simulations, to the recent boom of *Massively Multiplayer Online Games* (MMOG) [Funcom 99, Sony 99, O2OE 98, Blizzard 04], many efforts have been made to allow people to interact in realistic virtual environments. Most of the existing MMOGs are role-playing games, whereas first-person shooter games or real-time strategy games are usually divided into many small isolated game sessions with a handful of players each.
Works of science fiction , such as Neal Stephenson's novel "Snow Crash" [Stephenson 92] and the Matrix movies [Wachowski 99], give an impression of what a 3D environment that is truly consistent, persistent, realistic and immersive could be like. With progression in technology, converging advances in CPU, 3D acceleration and bandwidth may make the vision come true in the near future.
However, to create a large-scale NVE a number of problems must be solved, namely [Knutsson 04, Oram 01]:

- Consistency - For meaningful interactions to happen, all users' perceptions of the virtual world must be consistent. This includes maintaining states and keeping events synchronised. The inherently distributed nature of peer networks makes it difficult to guarantee reliable behaviour. The most widespread solution to ensure consistency across NVEs is to keep redundant information in different peers. For example, in case of processing intensive applications upon a detection of a failure the task can be restarted on other available machines. Alternatively, the same task can be initially assigned to multiple peers. In messaging applications lost messages can be resent or sent along multiple paths simultaneously. Finally, in file sharing applications, data can be replicated across many peers. In all serverless p2p systems, states about neighbours and connections must be kept in different peers to ensure the consistency after the crash of one or more peers.
  Zhou et al. [Zhou 04] classifies NVE event consistency into two main groups:
  - Casual order consistency - Events must happen in the same order as they occur, as humans have deeply-rooted concepts about the logical order of sequence of events.
  - Time-space consistency - In an NVE system, messages are sent to notify for position updates. However, due to network delay and clock asynchrony among computers, it is possible for hosts to receive updates with different delays and thus interpret the order of events differently.

Inconsistencies therefore could occur for entity positions at a given logical time.

Note that casual order consistency and time-space consistency are not necessarily related to each other (i.e. it is possible to preserve casual order consistency but violate time-space consistency). This problem is particularly evident when predications are used to compensate missing updates. On the other hand, in p2p networks the concept of consistency generally refers to what may be called topology consistency, which is whether each node in the p2p system holds consistent views of the parts of the network they share (note that each node only maintains a local view of the complete topology).

- Scalability is usually concerned with the number of simultaneous users in NVE. One important challenge is to allow all willing people to interact in the same environment. This is achieved by developing new systems that do not rely on a centralised server that can fail due to overload. Moreover no investment is needed for servers or the connection relaying them to Internet. The nodes only know the nodes in their *attention radius*. While the virtual world is infinite, in this radius the number of peers is limited. A trade-off has to be made between knowing enough neighbours to interact and not keeping connections open to too many neighbours.

- Reliability and fault resilience is important to make NVE a service with quality. To ensure this point peers may need some redundant information to cope with a crash of one or more neighbours. One of the primary design goals of a p2p system is to avoid a central point of failure. Although most p2p systems (pure p2p) already do this, they nevertheless are faced with failures commonly associated with systems spanning multiple hosts and networks: disconnections / unreachability, partitions, and node failures. These failures may occur more often in some networks (e.g. wireless) than in others (e.g. wired enterprise networks). In addition to these random failures, personal machines are more vulnerable than servers to hacking attacks or viruses. It would be desirable to continue active collaboration among the remaining connected peers in the presence of such failures. An example would be an application like genome@home [Pande 01] executing a partitioned computation distributed over connected peers. Would it be possible to continue the computation if one of the peers disappeared because of a network link failure? If the disconnected peer reappeared, could the completed results (generated during the standalone phase) be integrated into the ongoing computation?

- Performance - NVEs are simulations of the real world. Performance therefore is important to change the virtual world fast enough to allow the impression of reality and to keep the different views consistent. Because of the decentralised nature of these models performance is influenced by three types of resources: processing, storage and networking. In particular, networking delays can be significant in widearea networks. Bandwidth is a major factor when a large number of messages are propagated in the network and large amounts of files are being transferred among many peers. This limits the scalability of the system. Performance in this context cannot be measured in the abstract millisecond level, but rather tries to answer questions of how long it takes to retrieve a file or how much bandwidth a query will consume. These numbers have a direct impact on the usability of a system.

- Security is a big challenge for NVEs, especially if they use a pure p2p infrastructure. Transforming a standard client device into a server poses a number of risks to the system. Only trusted or authenticated peers should have access to information and services provided by a given node. Unfortunately, the security requirement requires either potentially painful intervention from the user, or interaction with a trusted third party. Centralising the task of security is often the only solution even though it voids the p2p benefit of a distributed infrastructure. Another way is to introduce the notion of trust: In the physical world we trust someone who has a good reputation. The concept of reputation can be adopted to the p2p world: you only trust a node you know or a node for which you got a recommendation from a known trusted node.

- Persistency - To create sophisticated contents, certain data such as user profile and valuable virtual objects must be persistently stored and accessed between user sessions. An example of an virtual object may be a bar where people can meet. This information is in most cases stored on a central server, to allow the users to log into the virtual world with the same identity from different computers.

We consider scalability to be the most important issue if we plan to build truly massive worlds and applications, which millions of people can participate in and enjoy. Therefore, this thesis focuses on finding a solution for the scalability problem in NVEs. Existing approaches to improve scalability mainly rely on enhancing server capacity in client-server architecture. Further scalability is achieved by clustering servers and by dividing the game universe into multiple different, or parallel, worlds and spreading the users over them. However, client-server architecture has an inherent upper limit in its available resources (i.e. processing and bandwidth capacity) and is expensive to deploy and to maintain. P2p architecture has emerged in recent years to an alternative that promises scalability and affordability. We attempt to apply p2p architecture to NVE design, in hope that it may solve the scalability problem.

In the following two existing NVE systems without central server or super-nodes are presented.

2.1 Solipsis

Solipsis was developed by Joaquín Keller and Gwendal Simon at France Telecom [Keller 04, Keller 02, Keller 03, Simon 04]. It focusses on the first four requirements mentioned above. Security and persistency are not taken into account, because they can't be solved properly without a centralised instance.
Solipsis intends to be scalable to an unlimited number of users and to be accessible by any computer connected to the Internet. It does not make use of any server and is solely based on a network of peers. In the implemented version the interaction is restricted to chat, but the system could be enhanced, e.g. by audio and video broadcast to the neighbouring peers. Furthermore it allows its virtual coordinates to be changed by *walking* or by using the teleportation mechanism.
Solipsis maintains direct connections among neighbours (latency is thus minimised). Specifically, it requires that each node must be inside a convex hull formed by its neighbours in 2D plane (Figure 1). This way the topology is guaranteed

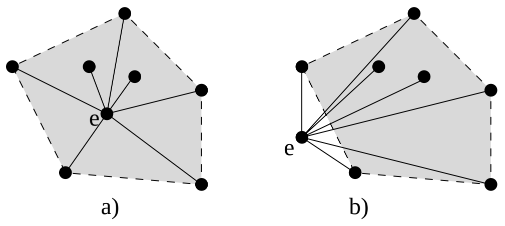

Figure 1: All peers must be inside the convex hull of their neighbours. In situation a) e respects this rule while it does not in b). [Keller 03]

to be fully connected (global connectivity is kept) (Figure 2). Since an incoming node may be unknown to direct neighbours, inconsistent topology may happen during normal operation occasionally. Proper neighbour discovery is thus not guaranteed (local awareness is not kept, see figure 3).
To join in the virtual world of Solipsis, the peer e wishing to join has to know at least one peer e_0 of the virtual world. e_0 routes the join request to the peer e_1, that is its nearest neighbour in the direction of e. The message is recursively routed to the peer e_n that has no neighbour closer to the target location of e than it is itself (Figure 4).
The join procedure then starts the second phase, trying to allocate all peers around the desired location of e. e_n, believing it is the nearest to the desired location of e, routes the message to e_m. If e_m knows a peer nearer to the desired location of e than e_n the first phase starts again. Otherwise the second phase

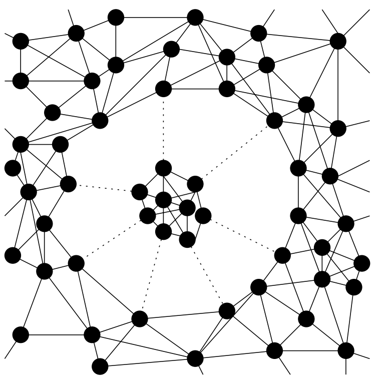

Figure 2: The dotted connections are due to the fact that each node lies inside a convex hull formed by its neighbours. [Keller 03]

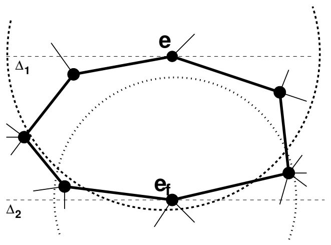

Figure 3: The nodes e and e_f are not connected even though e_f lies inside the attention radius of e. [Simon 04]

continues until a convex polygon with e in its convex hull has been generated. In some cases, proper neighbour discovery could be slow since all neighbours around the desired location are queried (Figure 5).

2.2 SimMud

Knutsson et al. describe p2p support for MMOGs by using Pastry [Rowstron 01] and Scribe [Castro 02], a p2p overlay and its associated simulated multicast [Knutsson 04]. The virtual world is divided into regions of fixed size (Figure 6). Each region is managed by a promoted super-node called a coordinator, which coordinates the shared objects in its region and serves as the root of a multicast tree. Therefore it is not a pure p2p model, but a hybrid one. Users inside the same region subscribe to the same root node to receive updates from other users, so neighbours are discovered via the coordinator. Coordinators maintain links with each other, supporting user transition to other regions. However users cannot perceive users in other regions. If users decide to listen to more than one region, they will receive irrelevant messages. If too many users are inside

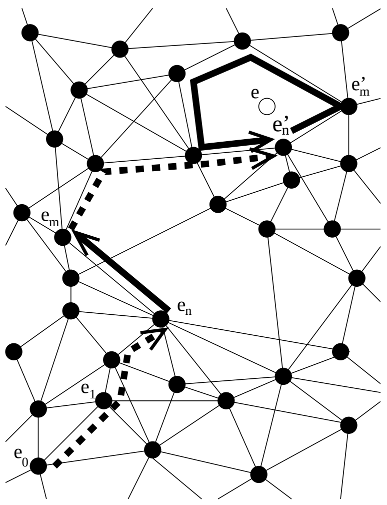

Figure 4: A first phase to reach e_n, an aborted turn around e_m, another phase to reach e'_n and turn around e. [Keller 03]

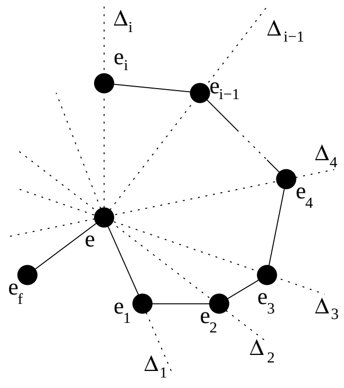

Figure 5: e may need various queries to discover e_i. [Keller 03]

the same region the coordinator may be overloaded. Coordinators are needed to manage persistent objects, therefore there is a backup mechanism to save this data.

Players in a region of the game space form a multicast group and communicate using the Scribe multicast system. When players move from one region to another they leave the multicast group in the old region and join the multicast group in the new region. A player may be a coordinator for a region and may be playing in another region of the game space at the same time. But still the player has to maintain state and handle communication for that region. Hence in this system players maintain state for regions of game space that are no longer close to them. As the players move in the virtual space, their region of

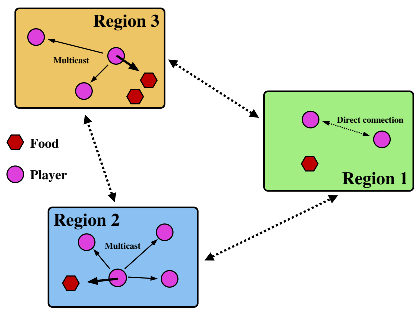

Figure 6: Fixed size regions

influence and interest continuously change, but this system does not take this into consideration.
A more serious problem is the latency penalty incurred by using the p2p overlay. Message updates may need to be relayed by other nodes; less than 4 hops in most cases, but exceeding 50 in other cases — a delay of several seconds. Note that this is ”virtual hops” on the p2p overlay, so physically it translates to more hops.

The next section discusses how to dissect the virtual world in a way that allows efficient methods to find peers and to join the virtual world.

3 Partitioning the virtual world

A main problem in constructing a NVE is to partition the infinite virtual world, so that each peer can easily find all its neighbours and the nearest peer to any coordinate in the world can be found efficiently. Most of the existing NVEs as for example Solipsis are based on a virtual world in two dimensions. This thesis shows how to build a NVE in three dimensions. Therefore a structure must be found to satisfy the specified requirements in 3D. The third dimension is needed to make the virtual world more similar to the real world, as is seen in many MMOGs.

The next paragraph discusses the appropriateness of the Voronoi diagram for these needs.

3.1 The Voronoi diagram

3.1.1 The definition

The Voronoi diagram ($\mathcal{VD}$) was introduced by Georges Voronoï in 1907 [Voronoï 07, Voronoï 08]; its definition in 2D [Aurenhammer 91, Aurenhammer 00]:

> Let S be a set of $n \geq 3$ point sites $p, q, r, \ldots$ in the plane. For points $p = (p_1, p_2)$ and $x = (x_1, x_2)$ let
>
> $$d(p,x) = \sqrt{(p_1 - x_1)^2 + (p_2 - x_2)^2}$$
>
> denote their Euclidean distance. By $\overline{pq}$ we denote the line segment from p to q. For $p, q \in S$ let
>
> $$B(p,q) = \{x | d(p,x) = d(q,x)\}$$
>
> be the bisector of p and q. $B(p,q)$ is the perpendicular line through the center of the line segment $\overline{pq}$. It separates the half-plane
>
> $$D(p,q) = \{x | d(p,x) < d(q,x)\}$$
>
> containing p from the half-plane $D(q,p)$ containing q. We call
>
> $$VR(p,S) = \bigcap_{q \in S, q \neq p} D(p,q)$$
>
> the *Voronoi region* of p with respect to S. By $\overline{A}$ we denote the closure set of A. Finally, the *Voronoi diagram* of S is defined by
>
> $$\mathcal{VD}(S) = \bigcup_{p,q \in S, p \neq q} \overline{VR(p,S)} \cap \overline{VR(q,S)}.$$

> Each Voronoi region $VR(p,S)$ is by definition the intersection of $n-1$ open half-planes containing the site p. Therefore, $VR(p,S)$ is open and convex. Different VRs are disjoint.
> The common boundary of two VRs belongs to $\mathcal{VD}(S)$ and is called a *Voronoi edge*, if it contains more than one point. If the Voronoi edge e borders the regions of p and q then $e \subset B(p,q)$ holds. Endpoints of Voronoi edges are called *Voronoi vertices*; they belong to the common boundary of three or more VRs.

The same definition in a more intuitive way: Given a set S of n points on a plane (each point called a site), $\mathcal{VD}(S)$ is constructed by partitioning the plane into n non-overlapping Voronoi regions that contain exactly one site in each Voronoi region (VR). A VR contains all the points closer to the region's site than to any other site. The entire plane is therefore divided into disjoint regions in a deterministic way.
To obtain the definition in 3D simply replace *3 points* by *4 points*, *line segment* by *plane segment*, *halfplane* by *halfspace*...

In the majority of cases the $\mathcal{VD}$ is introduced by the *post-office problem*: the sites are traditionally viewed as post offices where customers want to post their letters. Then, for any chosen location of a customer, the containing VR makes explicit the post office closest to him/her.
Some of the VRs are necessarily unbounded. They are defined by sites lying on the boundary of the convex hull of S because just for those sites there exist points arbitrarily far away but still closest. No vertices occur, if and only if, all sites in S lie on a single straight line. Such degenerate (collinear) configurations also imply the existence of regions with only one (unbounded) edge [Aurenhammer 91].
This is only the simplest case of $\mathcal{VD}$s, where the objects are points and the distance is given by the usual Euclidean metric; all other cases (see [Aurenhammer 00, Yvinec 98]) have no interest for the project of building a NVE.
The $\mathcal{VD}$ is one of the most fundamental concepts in computational geometry. Geometric properties and algorithms for $\mathcal{VD}$ have been active topics of research for many years (about one out of 16 papers in computational geometry). The $\mathcal{VD}$ has several desired properties that make it very important to applications such as computer graphics, numerical computing and geometric optimisation. It is very important for all these applications that the number of vertices and edges in the $\mathcal{VD}$ only grows in a linear way. For example the nearest neighbour site of any query point can be retrieved in $\mathcal{O}(\log n)$ (the *point-location problem*).

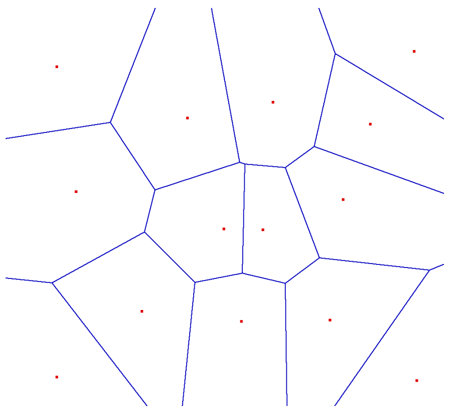

Figure 7: A simple Voronoi diagram in a plane (All 2D $\mathcal{VD}$s and 2D Delaunay triangulations in this paper are generated with VoroGlide [Icking 96]).

Theorem 3.1.1 *The $\mathcal{VD}(S)$ has $\mathcal{O}(n)$ many edges and vertices. The number of vertices in $\mathcal{VD}$ of a set S of n points in the plane is at most $2n-5$ and the number of edges is at most $3n-6$.*

The proof to this Theorem can be found in [Aurenhammer 00, Ottmann 01].

3.1.2 The data structure

Moreover there exists a very simple data structure to efficiently store the $\mathcal{VD}$. It is called *Quad-Edge* and has been developed by Guibas and Stolfi [Guibas 85]. It allows for an edge-to-edge navigation through the mesh by means of its algebraic operations. After the publication it was criticised for its relatively high storage costs, but this is not a problem any more. But there is a simpler structure based on a double connected edge list (DCEL) [Berg 00, Ottmann 01]. The Voronoi edges that the breakpoints trace out, the Voronoi vertices and the pointers between each site and the edges and vertices defining the cell of that site are stored in a DCEL. DCELs are the usual data structures for storing planar subdivisions. A DCEL contains a record for each face, edge and vertex of the subdivision. The different sides of an edge are viewed as two distinct half-edges. The two half-edges for a given edge are called twins. There is a unique next half-edge and previous half-edge for every half-edge, as an origin and destination of every half-edge. Three records are needed: one for the vertices, one for the faces, and the DCEL for the half-edges.

```
vertex{
  Coordinates
  Incident edge
  }
```

```
face{
  OuterComponent
  }
halfedge{
  Origin
  Twin
  IncidentFace
  Next
  Prev
  }
```

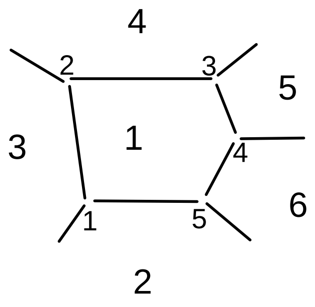

Figure 8: DCEL to store the Voronoi diagram

The $\mathcal{VD}$ in figure 8 is stored in the following way:

```
 vertex1 = { (1,2) | 12 } ...

face1 = { 15 }
...

halfedge54 = { 5 | 45 | 1 | 43 | 15 }
...
```

3.1.3 The algorithm

In two dimensions very efficient algorithms exist to compute the $\mathcal{VD}$. Before looking at the most important ones we try to constitute a lower bound for its computational complexity.

Theorem 3.1.2 *It takes time $\Omega(n \log n)$ to construct $\mathcal{VD}$ of n points $p_1, \ldots, p_n$ whose x-coordinates are strictly increasing.*

Suppose that n real numbers $x_1, \ldots, x_n$ are given. From the $\mathcal{VD}$ of the point set $S = \{p_i = (x_i, x_i^2) | 1 \leq i \leq n\}$ one can derive, in linear time, the vertices of

the convex hull of S, in counterclockwise order. From the leftmost point in S on, this vertex sequence contains all points p_i, sorted by increasing values of x_i. This argument shows that constructing the convex hull and computing the $\mathcal{VD}$, is at least as difficult as sorting n real numbers, which requires $\Theta(n \log n)$ time [Aurenhammer 00].
The most important algorithms are described subsequently:

Divide and Conquer algorithm. Shamos and Hoey [Shamos 75] first presented a worst-case optimal algorithm for computing the $\mathcal{VD}$ in 2D. The set of

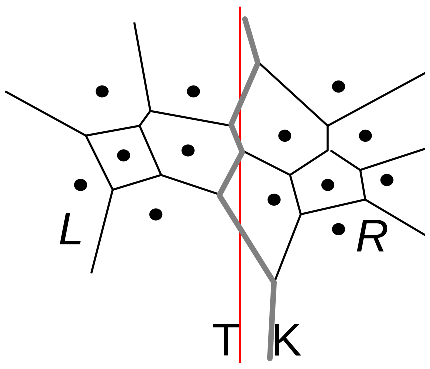

Figure 9: Divide and Conquer algorithm, the subsets L and R as well as the dividing line T and the split line K [Ottmann 01].

point sites S is split by a dividing line T into subsets L and R of about the same sizes. During the recursion, vertical or horizontal split lines can be easily found if the sites in S are sorted by their x- and y-coordinates beforehand. After the split $\mathcal{VD}(L)$ and $\mathcal{VD}(R)$ are computed recursively. The essential part is in finding the split line K, and in merging $\mathcal{VD}(L)$ and $\mathcal{VD}(R)$, to obtain $\mathcal{VD}(S)$. If these tasks can be carried out in time $\mathcal{O}(n)$ then the overall running time is $\mathcal{O}(n \log n)$ and is asymptotically optimal [Aurenhammer 91]. The merge step involves computing the edge course K separating regions of sites in L from regions of sites in R. All the cross edges, composing together the edge course K must cross the dividing line T. This establishes a linear ordering of the cross edges, so it is possible to talk about successive cross edges, the bottom-most cross edge, etc.

Plane sweep algorithm (Fortune's algorithm). Fortune [Fortune 87] first introduced the plane sweep algorithm. Its strategy is to sweep a horizontal line — the *sweep line* — from top to bottom over the plane containing the sites $S = p_1, \ldots, p_n$. While the sweep is performed, information is maintained regarding the structure that one wants to compute. More precisely, information is maintained about the intersection of the structure $\mathcal{VD}$ with

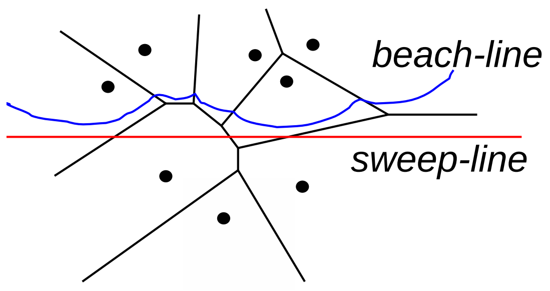

Figure 10: Plane sweep algorithm: the beach- and the sweep line l [Ottmann 01].

the sweep line l. Unfortunately this is not so easy, because the part of $\mathcal{VD}(S)$ above l depends not only on the sites that lie above l but also the part of $\mathcal{VD}(S)$ on sites below l. Stated differently, when l reaches the topmost vertex of $VR(p_i, S)$ has not yet encountered the corresponding site p_i. Therefore, not all information needed to compute the vertex is known. Instead of maintaining the intersection of $\mathcal{VD}$ with l, information is maintained about the part of $\mathcal{VD}$ of the sites above l that cannot be changed by sites below l. Because the bisector of a line, in this case l, and a point is a parabola, the boundary is a connected chain of parabola segments whose top- and bottommost edges tend to infinity. This chain is called the *beach line*. During the sweep, there are two types of events that cause the structure of the beach line w to change: when w hits a new site and when it arrives at the point where its two adjacent spikes intersect. The size of$w$ is $\mathcal{O}(n)$ and the $\mathcal{VD}$ of n points in the plane can be computed within $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space, which is optimal. The proof can be found in [Aurenhammer 91, Aurenhammer 00]. A java applet visualising the algorithm has been developed by Odgaard and Nielsen, it can be tested at [Odgaard 00].

Incremental insertion. This algorithm allows — in contrast to the plane sweep and the divide and conquer algorithms — to update the diagrams as new points are added or deleted. A simple version was proposed by Green and Sibson [Green 78]. The problem is to obtain $\mathcal{VD}(S)$, $S = p_1, \ldots, p_n$, from $\mathcal{VD}(S \setminus \{p_i\})$ by inserting the site p_i. As $VR(p_i)$ can have up to $n-1$ edges, this leads to a runtime of $\mathcal{O}(n^2)$ in the worst case. The insertion process is better described and implemented in the dual environment of $\mathcal{VD}$, the Delaunay triangulation (see 3.2). The advantage over a direct construction of $\mathcal{VD}(S)$ is that Voronoi vertices that appear in intermediate diagrams but not in the final one need not be constructed and stored [Guibas 85, Aurenhammer 00].

Except the *incremental insertion* algorithm, all algorithms need from the beginning as input all points of the diagram they compute. Consider every point of the $\mathcal{VD}$ being a peer in a p2p network or a NVE, peers with a common border are neighbours. Joining or leaving of a peer in the neigbhourhood would involve the complete recomputing of the diagram. Thus we are interested in dynamic algorithms, which allow us to update our diagrams as new points are added or deleted. Therefore we will focus on the *incremental insertion* algorithm.
The join procedure is very short: One peer p of the network has to be known and contacted, then the nearest peer n to the desired location of the peer that wishes to join j has to be found (point location problem). Then $VR(n)$ is divided and the concerned neighbours are informed about the changes. It is very important to notice, for the maintenance of the structure, that the effect of a join remains local (Figure 11) [Araujo 01, Hu 04].

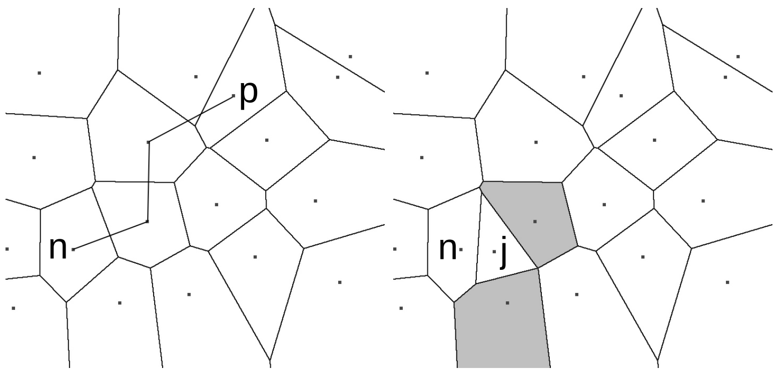

Figure 11: Join procedure. On the left: Forward of join request. The known peer p. The peer n nearest to the desired location. On the right: The new peer j. Shaded regions are neighbours affected by join.

The way in which the neighbour peers are contacted and the regions are divided is described very well by Liebeherr [Liebeherr 02]. Briefly: The peer n contacts its *clockwise* and *counterclockwise* neighbours and informs them about the newly arriving peer j. They contact j and update their respective VR. Liebeherr has also implemented this algorithm and used it to build up a p2p infrastructure called *HyperCast*, which can be downloaded at [Liebeherr 03]. Only the neighbour peers are stored, not the triangles as they can be computed from the list of neighbour peers. Unfortunately his algorithm cannot be extended to $\mathbb{R}^3$, as there the notions of *clockwise* and *counterclockwise* have no meaning in $\mathbb{R}^3$.
Hu and Liao have presented and implemented an algorithm to create a *Scalable Peer-to-Peer Networked Virtual Environment* [Hu 04]. They also use Voronoi diagrams to partition the virtual world. As with the implementation of Liebe-

herr, their algorithm is only applicable to a two dimensional world, as they do not take into account all difficulties arising from switching to 3D.

The process of inserting VRs can be extended to $\mathbb{R}^3$, but the data structure becomes more complicated since the tetrahedra have to be stored as well. It is not possible to reconstruct them from the list of neighbour peers. Regions are convex polyhedra that can be constructed facet by facet by intersecting existing regions with separators that are planes in this case. A VR cannot have more than $n-1$ facets (one for each different site) and thus by Euler's relation, has $\mathcal{O}(n)$ edges and vertices. Therefore, one needs $\mathcal{O}(n+t)$ time per region if t facets, edges, or vertices are deleted during its insertion. Since each component deleted has to be constructed first, an $\mathcal{O}(n^2)$-time algorithm results. This is worst-case optimal since a $\mathcal{VD}$ in $\mathbb{R}^3$ may have $\Theta(n^2)$ number of facets [Aurenhammer 91]. The VRs can become very complicated polyhedra (Figure 12), thus no efficient data structure to store the $\mathcal{VD}$ in $\mathbb{R}^3$ exists, especially the one discussed for $\mathbb{R}^2$ is not applicable.

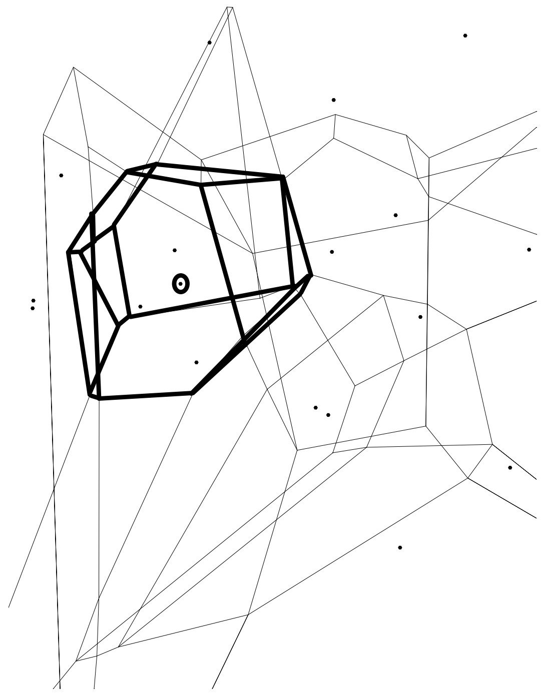

Figure 12: A 3D $\mathcal{VD}$ with only 20 vertices: the highlighted Voronoi region belongs to the encircled point (All the diagrams in 3D are generated with geomview [geomview 92])

It appears that the hardest part of constructing a $\mathcal{VD}$ is the determination of its topological structure: the incidence relation between vertices, edges, and faces. Once the topological properties of the diagram are known, its geometrical properties (coordinates, lengths, angles, etc.) can be computed in time linear in the number of sites. That is why the $\mathcal{VD}$ (especially in $\mathbb{R}^3$) is never calculated directly, but via the Delaunay triangulation which is its dual. Having the Delaunay triangulation, the $\mathcal{VD}$ can be calculated in $\mathcal{O}(n)$. This detour greatly simplifies the computation of the Voronoi diagram [Guibas 85].

3.2 The Delaunay triangulation

3.2.1 The definition

A simplex of dimension k is a k-polotype with $k+1$ vertices, or equivalently the convex hull of $k+1$ affinely independent points. Let A be a set of $k+1$ affinely independent points and S be the k-simplex defined by A. Any subset of $l+1 < k+1$ points in A defines an l-simplex which is a *face* of S. Simplices of dimension 0,1,2, and 3 are respectively called points, segments, triangles, and tetrahedra .
A complex C is a finite set of simplices that satisfy the following properties:

1. any face of a simplex in C is also a simplex in C, and
2. two simplices in C either do not intersect, or their intersection is a simplex of smaller dimension which is their common face of maximal dimension.

The simplices that constitute a complex are called the *faces* of the complex. The faces of dimension 0 are called the *vertices* and the faces of dimension 1 are called the *edges*. In dimension d, the faces of dimension d and $d-1$ are called the *cells* and the *facets*. Two faces of a complex are *incident* if one is included in the other and their dimensions differ by one. Two vertices of a complex are *adjacent* if they share a common edge, and two cells are *adjacent* if they share a common facet [Boissonnat 01, Yvinec 98].

Definition 3.2.1 *Let P be a set of n points in $\mathbb{R}^d$. A triangulation of P is a set of simplices whose vertices are the points of P that satisfy the following two properties:*

1. *two simplices either do not intersect, or their intersection is a simplex of smaller dimension which is their common face of maximal dimension, and*
2. *the simplices tile the convex hull of P.*

The Delaunay triangulation ($\mathcal{DT}$) decomposes the convex hull of a point set S into cells (triangles in $\mathbb{R}^2$ and tetrahedra in $\mathbb{R}^3$) (Figure 13). The $\mathcal{DT}$ is named after Boris Delaunay who introduced it in 1934 [Delaunay 34]. It is the dual of the $\mathcal{VD}$ in a graph-theoretical sense (Figure 14). In fact, a $\mathcal{DT}$ in 3D should be named *Delaunay Tetrahedralization* [Devillers 01]: three points are needed to define a plane (or a triangle, therefore triangulation) in space and four points are needed to define a volume, whereof the simplest one is the tetrahedron. As the literature almost always uses the term *triangulation*, we will do so as well.
The circumsphere $\mathcal{C}_{abcd}$ is the sphere defined by the four vertices *abcd* of a tetrahedron $\mathcal{T}_{abcd}$. The center of the circumsphere $\mathcal{C}_{abcd}$ (of the circumcircle $\mathcal{C}_{abc}$ in $\mathbb{R}^2$) of every cell is a vertex of the $\mathcal{VD}$. If these centers between pairs of adjacent cells are connected, the resulting structure is the $\mathcal{VD}$. The duality immediately

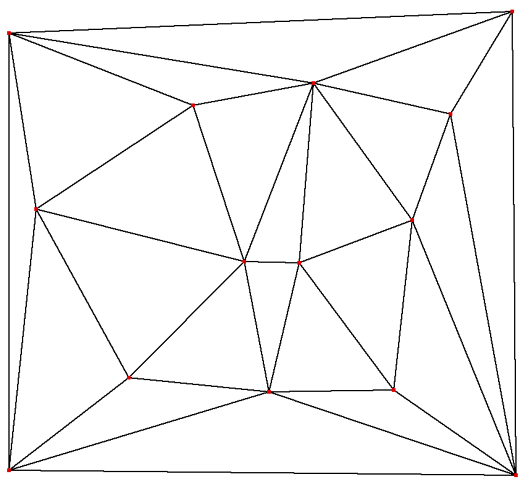

Figure 13: A simple $\mathcal{DT}$ in $\mathbb{R}^2$ (the dual of the $\mathcal{VD}$ in figure 7)

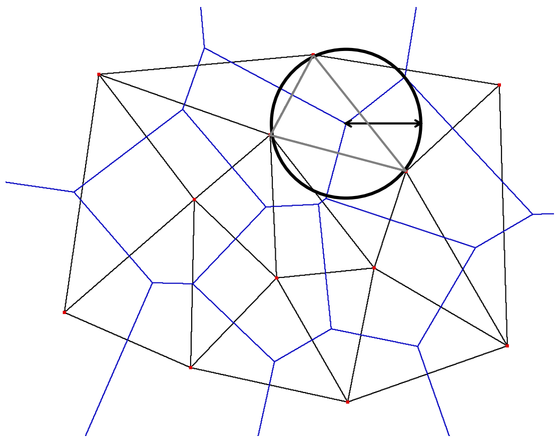

Figure 14: A $\mathcal{VD}$ with its dual the $\mathcal{DT}$. For one triangle the circumcircle is shown: the center of this circle is a *Voronoi vertex*

implies upper bounds of $3n - 6$ and of $2n - 5$ on the number of Delaunay edges and triangles, respectively (see 3.1.1). Whereas in 3D the number of tetrahedra may be $\Theta(n^2)$.
There are only two conditions building the $\mathcal{DT}$:

Definition 3.2.2 *Let P be a set of n points $p_1, \ldots, p_n$ in $\mathbb{R}^2$. The $\mathcal{DT}(P)$ in 2D is a triangulation of P where no point p_i, $1 \leq i \leq n$, lies inside the circumcircle of any triangle.*
Let P be a set of n points $p_1, \ldots, p_n$ in $\mathbb{R}^3$. The $\mathcal{DT}(P)$ in 3D is a triangulation of P where no point p_i, $1 \leq i \leq n$, lies inside the circumsphere of any tetrahedra.

This definition implies that the cells (tetrahedra or triangles, respectively) are not flat, otherwise their circumscribing circle or sphere is not defined. In fact, it is always possible to build the $\mathcal{DT}$ of a point set S without any flat cell [Devillers 03].

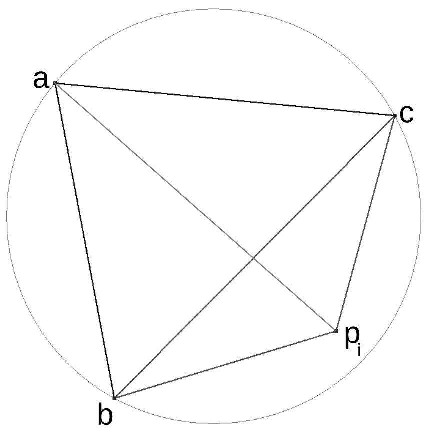

Figure 15: The point p_i is added, it lies inside $\mathcal{C}_{abc}$, therefore the edge $\overline{bc}$ is *flipped* to $\overline{ap_i}$. (After the *flip*, the smallest angle is bigger than before.)

3.2.2 The data structure

The *Quad-Edge* structure (see 3.1.2) offers the advantage of describing, at the same time, a planar graph and its dual (2D), so that it can be used for constructing both the Voronoi diagram and the Delaunay triangulation. From the DCEL of $\mathcal{VD}(S)$ we can derive the set of triangles constituting the $\mathcal{DT}(S)$ in linear time. Reciprocally, from the set of all Delaunay triangles the DCEL of the $\mathcal{VD}$ can be constructed in time $\mathcal{O}(n)$. Therefore, each algorithm for computing one of the two structures can be used for computing the other one, within $\mathcal{O}(n)$ extra time [Guibas 85].
The great advantage of the $\mathcal{DT}$ compared to the $\mathcal{VD}$ is, that it can be stored very easily, even in $\mathbb{R}^3$, because there is only one sort of polyhedra to store (Figure 16): the tetrahedron $\mathcal{T}_{abcd}$ (the triangle $\mathcal{T}_{abc}$ in $\mathbb{R}^2$). The *Quad-Edge* structure was extended by Dobkin and Laszlo [Dobkin 87] to three dimensions. Mücke [Mücke 93] simplified it by removing the information needed for the dual representation. It is based on triangles that can be traversed with a set of functions.

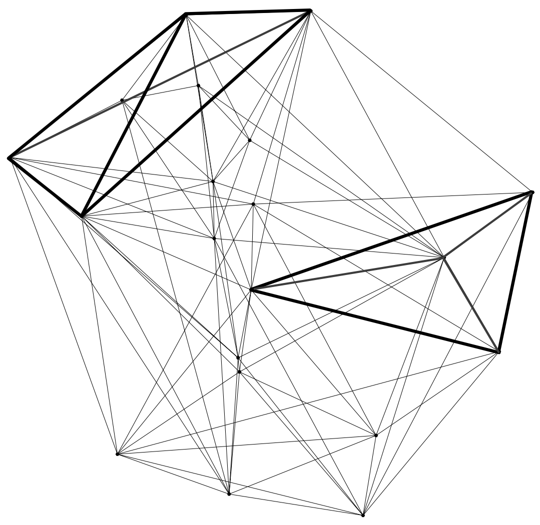

Figure 16: A 3D Delaunay triangulation with only 20 vertices (the dual of figure 12): two tetrahedra are highlighted for better visibility.

3.2.3 The algorithm

The basic component of most algorithms for the construction of the $\mathcal{DT}$ is the *Delaunay diagonal flip*. The triangles $\mathcal{T}_{abc}$ and $\mathcal{T}_{bcp_i}$ in figure 15 form together a convex quadrilateral. A *diagonal flip* replaces $\mathcal{T}_{abc}$ and $\mathcal{T}_{bcp_i}$ with $\mathcal{T}_{abp_i}$ and $\mathcal{T}_{acp_i}$, in effect replacing the diagonal $\overline{bc}$ with the diagonal $\overline{ap_i}$. The *diagonal flip* is a *Delaunay diagonal flip* if p_i is inside $\mathcal{C}_{abc}$ [Mücke 93].

Theorem 3.2.3 *Among all triangulations of a given point set, the $\mathcal{DT}$ has the largest angle vector.*

Flipping the edges whenever possible progressively increases the angle vector of the triangulation. Since there are only a finite number of triangulations, this process eventually reaches a triangulation that has only regular pairs of adjacent cells. This triangulation is then a $\mathcal{DT}$: the most compact triangulation, it is unique. Among all triangulations, the $\mathcal{DT}$ maximises the minimum angle. This implies that $\mathcal{DT}$s tend to avoid skinny cells. The proof can be found in [Yvinec 98].
At most $\mathcal{O}(n^2)$ *Delaunay diagonal flips* can be performed before the triangulation becomes the $\mathcal{DT}$, and no more flip is possible. The proof of the time complexity can be found in [Fortune 92]. The question of finding the three-dimensional analog to the equiangularity property (Theorem 3.2.3) and to the *Delaunay diagonal flip* is still unsettled.
As already seen in section 3.1.3, a dynamic algorithm is needed to build a p2p infrastructure. Therefore, in this section, we will only focus on an *incremental*

algorithm. First, the description is only done for the 2D case (most of the work has already been done by describing the *Delaunay diagonal flip*), then we will discuss the changes needed to extend it to 3D.
To add a site p_i to the $\mathcal{DT}$ the problem is to compute $\mathcal{DT}_i = \mathcal{DT}(\{p_1, \ldots, p_i\})$ from $\mathcal{DT}_{i-1}$ by inserting p_i. Let $\mathcal{T}_{abc}$ be one of the triangles of $\mathcal{DT}_{i-1}$ whose circumcircles $\mathcal{C}_{abc}$ contain the new site p_i and therefore are *in conflict* with p_i. They are not longer Delaunay triangles, according to theorem 3.2.2. Let $\overline{bc}$ be the edge that lies inside the quadrilateral $abcp_i$. A *Delaunay diagonal flip* has to be done, to replace $\mathcal{T}_{abc}$ and $\mathcal{T}_{bcp_i}$ by $\mathcal{T}_{abp_i}$ and $\mathcal{T}_{acp_i}$ (Figure 15). The newly created triangles have to be tested as well, and so on... [Aurenhammer 00]
In case no triangle is in conflict with p_i the convex hull of all sites from $S = p_1, \ldots, p_{i-1}$ needs to be enlarged. First p_i is connected to all sites from S which it *sees*, that means to which there exists an edge that does not cut any edge from $\mathcal{DT}_{i-1}$. All created edges are Delaunay edges to $S \cup p_i$. The opposite edges to p_i have to be checked and flipped if necessary.
The testing is done in a recursive fashion consistent with the incremental nature of the algorithm. When a new node is inserted inside a triangle, three new triangles are created, and three edges need to be tested. When the node falls on an edge, four triangles are created, and four edges are tested. In the case of test failure, a pair of triangles is replaced by the flip operation with another pair, producing two more edges to test.
Heller [Heller 90] was the first to describe how to remove a site from the $\mathcal{DT}$. In his method, the set of neighbouring points of the site p to be deleted are tested, as potential triangles, in anticlockwise order. The triangle with the smallest circumcircle is removed by swapping the edge (the inverse of the insertion algorithm described previously) to reduce the set of neighbours of p by one, and the process is repeated until only three triangles are left. Again as the inverse of the insertion algorithm, p is removed and the three triangles merged into one.
This first algorithm was improved by Mostafavia, Gold and Dakowicz [Mostafavia 03], some triangles may be removed faster, but the main idea remained the same.

The main problem in 3D is that there are some sets of points that allow different $\mathcal{DT}$s. In 2D a quadrilateral can only be divided into two triangles, and there's only one possible way to flip, whereas in 3D a hexahedra can be decomposed into two or three tetrahedra. A hexahedra (Figure 17) can be separated into two

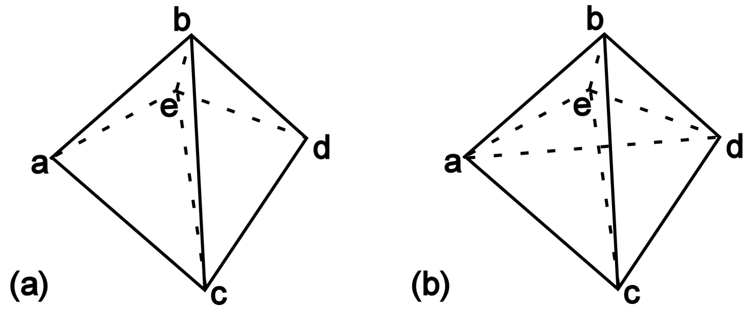

Figure 17: Two ways of triangulating a hexahedra.

tetrahedra $\mathcal{T}_{abce}$ and $\mathcal{T}_{bcde}$ (Figure 17 a) or into three tetrahedra $\mathcal{T}_{acde}$, $\mathcal{T}_{abde}$ and $\mathcal{T}_{abcd}$ (Figure 17 b).

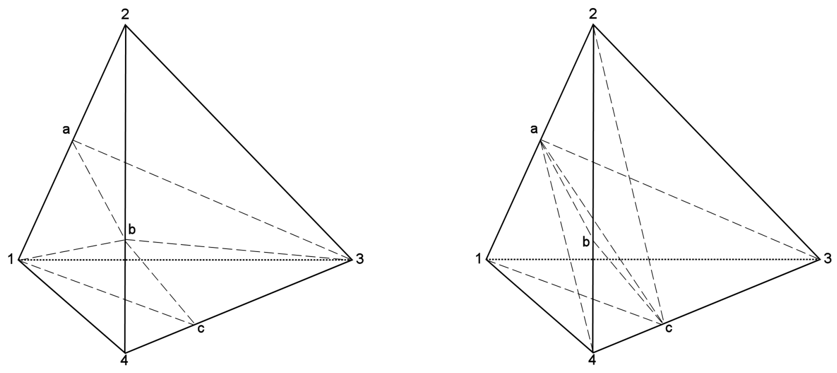

Figure 18: Different $\mathcal{DT}$s for one point set.

Another more complex example (Figure 18) taken from [Yvinec 98]: To $\mathcal{T}_{1234}$ the points a,b and c are added. There are two different ways of adding these points:

$$\mathcal{T}_{1234}\left\{\begin{array}{l}\mathcal{T}_{12b4}\left\{\begin{array}{l}\mathcal{T}_{1ab4}\\ \mathcal{T}_{a2b4}\end{array}\right.\\ \mathcal{T}_{1b34}\left\{\begin{array}{l}\mathcal{T}_{1b3c}\\ \mathcal{T}_{1bc4}\end{array}\right.\end{array}\right. \qquad \mathcal{T}_{1234}\left\{\begin{array}{l}\mathcal{T}_{1a34}\left\{\begin{array}{l}\mathcal{T}_{1a3c}\\ \mathcal{T}_{1ac4}\end{array}\right.\\ \mathcal{T}_{a234}\left\{\begin{array}{l}\mathcal{T}_{a23c}\left\{\begin{array}{l}\mathcal{T}_{a2bc}\\ \mathcal{T}_{ab3c}\end{array}\right.\\ \mathcal{T}_{a2c4}\end{array}\right.\end{array}\right.$$

As with the $\mathcal{VD}$, the $\mathcal{DT}$ also has degenerated cases: three or four more points situated on a line (collinear) or on a plane (coplanar), respectively and four or five points located on a circle in $\mathbb{R}^2$ or on a sphere in $\mathbb{R}^3$ (cospherical), respectively. When degeneration occurs, the $\mathcal{DT}$ in 3D is not unique. In general, any of the possible triangulations of the set of cospherical points can be returned. When a point p is inserted, the set of tetrahedra conflicting with p is determined, and these tetrahedra are deleted from the $\mathcal{DT}$. If the algorithm considers as non-conflicting the tetrahedra whose circumscribing sphere has p on its boundary, and thus if it does not delete these tetrahedra, then we get a unique construction of a $\mathcal{DT}$. This triangulation is unique for a given order of the points, but it depends on the order of insertion of the points. Moreover, the incremental construction does not create any flat tetrahedron: if the point p to be inserted is coplanar with a triangle t that is a facet of the $\mathcal{DT}$, then the two tetrahedra incident to t will have the same conflict status with respect to p, which means that either they will both stay, and t will still be their common facet in the updated triangulation, or they will both be deleted and t will disappear. Thus, p will not form a flat tetrahedron together with t [Devillers 03]. A very fast algorithm to compute the $\mathcal{DT}$ in 3D has been developed by the Computational Geometry Algorithms Library (CGAL) [CGAL 98]. It first computes the triangulation of the convex hull, then adds the internal vertices [Yvinec 98]. This implementation can build the $\mathcal{DT}$ of 100,000 points, randomly distributed, in 12 seconds (on a PIII-500 and 512MB of memory) [Boissonnat 02].

The next section presents the algorithm we developed to compute the $\mathcal{DT}$ in $\mathbb{R}^3$ in a *distributed* manner.

4 A distributed algorithm for the 3D $\mathcal{DT}$

No algorithm exists at present to compute the Delaunay triangulation ($\mathcal{DT}$) in a distributed way. Distributed means that each point of the triangulation is represented by one computer, not being aware of all other points but only of its direct neighbours. The algorithm starts with one single point and step by step all the other points are added and some of the tetrahedra built have to be removed or split. This way of computing the $\mathcal{DT}$signifies much more work than the work of an off-line algorithm. Such an algorithm has to know all the points from the beginning onwards, e.g. it first calculates the triangulation of the convex hull and then proceeds towards the points in the center. Furthermore an off-line algorithm can't add or remove a point, after a change it has always to restart the whole computation from the beginning.
First a simulation of the algorithm was developed; then it was distributed over different computers to form the infrastructure for a real p2p network and later on a NVE.
As programming language Java was chosen, to establish the possibility of running the application on mobile devices such as mobile phones or PDAs.
For the simulation no communication between the nodes representing the points of the triangulation is needed; each node is an object (and not a computer) and each tetrahedron as well. Changes on a node are directly noticed by its neighbours. But the main problem remains: All nodes only know their direct neighbours and not all nodes of the triangulation. Therefore it is very difficult to keep the view of all nodes coherent. In addition no tetrahedra may appear more than once and all neighbouring tetrahedra must know each other as well as their common plane.
To visualise the $\mathcal{DT}$ as well as some other information (e.g. the convex hull) geomview [geomview 92] is used.

First the data structure is described, together with the most important methods to navigate in it, then the two main methods to build and maintain a $\mathcal{DT}$ are discussed: add and remove (join and leave from a points/peers point of view). Following that a major problem in computing triangulations is described: the lack of precision and therefore the possible failure of tests that lead to inconsistent triangulations. Finally some test and validation methods are presented as well as the simulation results.

4.1 The data structure and the basic methods

We do not use any of the data structures presented in section 3.1.2 and section 3.2.2, but one easier to navigate in, even though some information are redundant. There are two main objects in this structure: the `Tetrahedron` and the `Node`. They refer to each other: The tetrahedra have references to the node, they are made of, and the nodes have references to the tetrahedra they are part of. All the other objects have only a supporting function, e.g. the `Vector`s and `Line`s which are basically needed to calculate the center of the circumsphere of a `Tetrahedron` and to represent a `Plane` by its normal vector starting from three `Point`s or `Node`s.

- A `Tetrahedron` is the main object to store the triangulation. It is composed of its four vertices stored as `Node`s and its circumsphere stored as `Sphere` which consists of a `Point` for its center and its radius. Furthermore it knows its four facets stored as `Plane`s and its four neighbour tetrahedra (`Tetrahedron`). This allows the navigation in the triangulation.

- A `Node` knows its own position stored in a `Point` and maintains a list of its neighbour `Node`s and a list of the tetrahedra (`Tetrahedron`) it belongs to. These references allow the navigation in the triangulation.

- A `Plane` represents a 3-dimensional plane. A `Plane` is described by a unit `Vector` and the perpendicular distance of the plane from the origin. The absolute direction of the plane `Vector` *is* important, giving it a direction (back and front faces). Moreover, if the `Plane` was created from three `Node`s, they are stored as well. The boolean `onConvexHull` indicates whether the `Plane` is on the convex hull of the triangulation.

- A `Line` represents a 3-dimensional line. A line is a vector which is located in space. It is described by a unit `Vector` and a `Point` on the line. Any point on the line can be used, and it could change during the existence of a calculation without affecting the integrity of the `Line`, e.g. `Point`$= \{1, 1, 1\}$, `Vector`$= \{1, 0, 0\}$ is the same `Line` as `Point`$= \{2, 1, 1\}$, `Vector`$= \{1, 0, 0\}$. However the absolute direction of `Vector` *is* important, giving the `Line` a direction.

- A `Point` represents a 3-dimensional point. As well as the `Plane` and the `Line` it is one of a set of primitives which can be combined to create and manipulate complex 3-dimensional objects.

- A `Vector` has three components giving it a length and a direction in $\mathbb{R}^3$ (whose sign is important), but no position. `Vector`s are often normalised to unit length. `Vector`s and `Point`s are very closely related and can sometimes be used interchangeably or there are equivalent routines or they can be converted using cross-constructors. A `Point` has a position and cannot be normalised.

Before specifying the two most important procedures, that allow nodes to join and to leave the triangulation, some of the basic helping methods are described. They answer the standard geometrical questions asked during the building of a triangulation, such as

- What is the distance between that line and that point?
- Are these two points on the same side of that plane?
- Do those two tetrahedra have a common plane?
- Where is the center of that tetrahedron?
- Do those two tetrahedra overlap?

Methods of the `Tetrahedron`:

```
Procedure Plane getCommonPlane(Tetrahedron tetra)
  N ← this.nodes
  M ← tetra.nodes
  C ← null  /* variable to store the common points */
  c ← 0  /* number of common points found */
  for i ← 0 to 3 do
      for j ← 0 to 3 do
          if N [i]==M [j] then
              if c == 4 then  Return: error, four common points
              C [c ] ← N [i]
              c ++
          endif
      endfor
  endfor
  if c ≠ 3 then  Return: error, no common plane between the tetrahedra
  Return Plane(C [0],C [1],C [2]);
```

```
Procedure boolean liesInside(Node node)
  N ← this.nodes
  P ← null
  for i ← 0 to 3 do
      P ← this.getPlane(i)
      if not P.isOnSameSide(node out of N that is not on P , node) then
          Return false
      endif
  endfor
  Return true
```

```
Procedure boolean overlap(Tetrahedron tetra)
N ← this.nodes
M ← tetra.nodes
P ← this.getCommonPlane(tetra)
if this==tetra then Return true
if P ≠ null then
    if P.isOnSameSide(node out of N that is not on P, node out of M that
    is not on P) then
        Return true
    endif
    else Return false
endif
else Return false
```

- `Point center()` In all algorithms computing the $\mathcal{DT}$, the *inCircle* test is the most expensive operation. That is why we split it into two parts: first the center and the radius of the circumsphere is computed. This is only done once for each tetrahedron. Every time when it is necessary to check if a node lies inside the sphere, only its distance to the center has to be measured and compared to the radius. This is the second part. For the construction of a $\mathcal{DT}$ of 100 uniformly distributed sites (and approximately 500 tetrahedra), this part is executed a 100,000 times, whereas the first part is executed only 5000 times.

- `boolean equals(Tetrahedron tetra)`

Methods of the `Node`:

```
Procedure Node nearest(Node node)
nearest ← node
distance ← this.getDistance(node)
nearest ← node
foreach n in node.getNeighbourNodes() do
    if n.getDistance(this) < distance then
        distance ← n.getDistance(this)
        nearest ← n
    endif
endfch
Return nearest
```

- `Node[] getNeighbourNodesOfDegree(int n)` This recursive method returns not only the direct neighbours of a `Node`, but also the neighbours of its neighbours and so on...

- `Tetrahedron[] getNeighbourTetrasOfDegree(int n)` This recursive method uses the `getNeighbourNodesOfDegree` method. It returns all `Tetrahedra` to which one of the `Nodes` of step 1 belongs to.

Methods of the `Plane`:

```
Procedure boolean isOnSameSide(Node na, Node nb)
    dista ← this.getDistanceFromNode(na)
    distb ← this.getDistanceFromNode(nb)
    if (dista $\geq 0$ and distb $\geq 0$) or (dista $\leq 0$ and distb $\leq 0$) then Return true
    else Return false
```

- `double getDistanceFromPoint(Point point)`
- `Point getIntersectionWith(Line line)`
- `Point getIntersectionWith(Plane pl2, Plane pl3)`
- `Line getIntersectionWith(Plane pl2)`

Methods of the `Line`:

- `Point getClosestPointTo(Point point)`

Methods of the `Point`:

- `double getDistanceFromPoint(Point point)`

Methods of the `Vector`:

- `Vector cross(Vector vector)`
- `double dot(Vector vector)`
- `Vector normalise()`
- `double getLength()`

There are a lot of other helping methods, but they basically refer to one or more of the shown methods. The documentation of all of them can be found at [Steiner 05].

In the following the methods for joining and leaving the triangulation are described.

4.2 Join the triangulation

Thanks to the helping methods we can now focus on the details arising from the Delaunay triangulation. First of all the algorithm for the join procedure is described in prose, before developing the pseudo code for it and discussing the difficulties experienced.

The node wishing to join needs to know one node that belongs to the $\mathcal{DT}$ to execute the join procedure. The nearest node to the desired location is searched by recursively travelling through the $\mathcal{DT}$. The first node of the triangulation has no neighbour. After the join of the second node, each of them has one neighbour. Not until four nodes are present, is the first tetrahedron built.
For all following nodes it has to be checked if the node lies inside a sphere of at least one existing tetrahedron or if it lies completely outside of the triangulation. In the first case all tetrahedra in whose sphere the joining node lies have to been split:

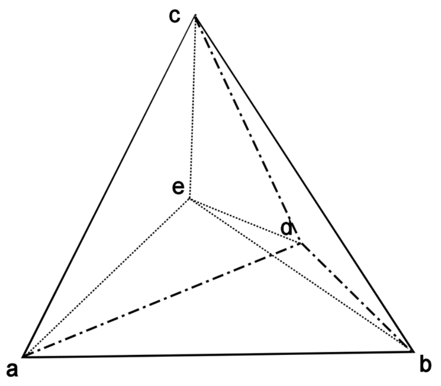

Figure 19: Four new tetrahedra $\mathcal{T}_{abce}$, $\mathcal{T}_{abde}$, $\mathcal{T}_{acde}$, $\mathcal{T}_{bcde}$ resulting from splitting $\mathcal{T}_{abcd}$ with e.

- If the joining node lies inside one tetrahedron (not only inside its sphere) the split creates four new tetrahedra (Figure 19).
- If the new node lies outside the tetrahedra (but inside its sphere) only two or three new tetrahedra are created (Figure 20 and figure 17, see 3.2.3).

In the second case (if the joining node lies outside any existing tetrahedron) the convex hull of all nodes has to be enlarged.

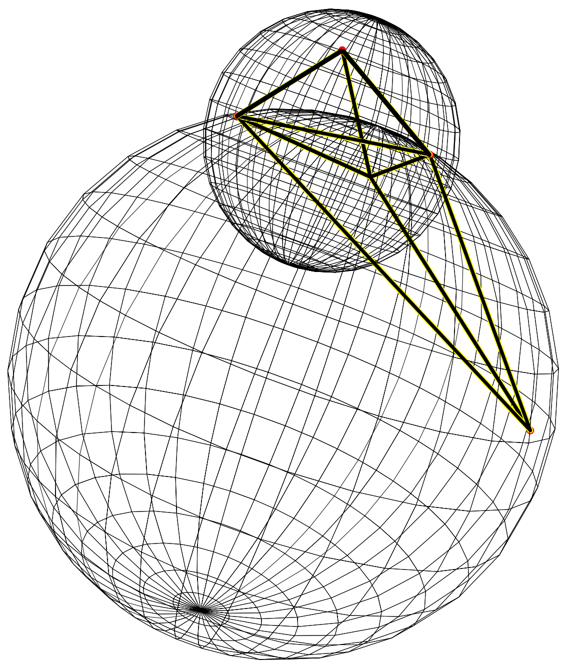

Figure 20: Split with two resulting tetrahedra: three points lie on both spheres, the two remaining ones lie only on one sphere each.

In order to keep the join procedure short, first the *split* and the *enlarge convex hull* procedure are defined.
The *split* procedure belongs to the class `Tetrahedron` and splits an existing tetrahedron by adding a new point inside its circumsphere.
The *enlarge convex hull* method belongs to the class `Node` and enlarges the convex hull of the triangulation by adding a new point that does not lie inside any point's circumsphere.
Thanks to those two major procedures the whole join procedure is very short. It uses a recursive approach to search the node nearest to the desired position, therefore the argument `node` is the only node of the $\mathcal{DT}$ known before the first execution of the procedure. In the following recursive calls `node` is the respective nearest neighbour to the desired location of the argument `node` from the previous call (see procedure Node nearest).

Procedure Tetrahedron[] split(*Node newNode*)

```
N ← this.nodes
tetras ← null  /* variable for the min. 2 and max. 4 tetrahedra resulting
from the split */
for i ← 0 to 3 do
    tetras [i] ← Tetrahedron(N [i%4],N [(i+1)%4],N [(i+2)%4],newNode)
    if the points are collinear then
        tetras [i] ← null
    endif
    if not newNode.liesInside(this) then
        if N [(i+3)%4].liesInside(this.getSphere) then
            tetras [i] ← null
        endif
    endif
endfor
Update the neighbour relationship between the 5 nodes (the four nodes of
this and newNode)
Update the list of tetrahedra of the 5 nodes
Update the tetrahedra neighbourship relations between tetras
for i ← 0 to 3 do
    if this.getPlane(i).isOnConvexHull() then
        for j ← 0 to 3 do
            for k ← 0 to 3 do
                if tetras [j].getPlane(k)==this.getPlane(i) then
                    tetras [j].getPlane(k).setOnConvexHull(true)
                endif
            endfor
        endfor
    endif
endfor
Return tetras
```

There are two main difference between the 2D algorithm presented in section 3.2.3 and the one we developed for the 3D case:

- In 2D a recursion is needed that checks if the triangles, created by the Delaunay flips, are in conflict with any point. In 3D the recursion is not needed: all the tetrahedra of the neighbour nodes are checked, but the error does not propagate as it does in 2D. Many tests showed this fact, but unfortunately we cannot prove it.
- In 3D the order of inserting plays a role, as the $\mathcal{DT}$ is not unique for different orders of insertion (Figure 18). Therefore, if two new nodes arrive at about the same time, the order in which these events are noticed by the concerned nodes must always be the same (see 5.1).

Because there is no recursion in 3D it is very easy to prove that the algorithm halts. In the main loop only a finite number of tetrahedra is chosen to be checked; of those some are destroyed. The new point is inserted and the $\mathcal{DT}$ is repaired, and at this stage the join algorithm stops. At the worst case — all points are situated on the surface of a sphere and the new points lies exactly in the middle of that sphere — all points of the $\mathcal{DT}$are concerned and therefore all tetrahedra are checked and destroyed. But also in this case it is obvious that the loop halts. Consider for the pseudo code of the join procedure that the $\mathcal{DT}$ contains more than four nodes.

```
Procedure void enlargeConvexHull(Node nearestNode,
Tetrahedron[] tetras)
  this /* is the new node */
  nearestNode /* is the node nearest to this */
  tetras /* are the concerned tetrahedra of the $\mathcal{DT}$ */
  newlyCreated ← null /* variable for the created tetrahedra (needed to
  update the relations between them) */
  tetra /* the added tetrahedron */
  for $i \leftarrow 0$ to tetras.length do
      for $j \leftarrow 0$ to 3 do
          if tetras [i].getPlane(j).isOnConvexHull() then
              if not tetras [i].getPlane(j).contains(this) then
                  tetra ← Tetrahedron(tetras [i].getPlane($j$), this)
                  if inCircleTest(tetra,nearestNode.getNeighbourNodes())
                  then
                      if inCircleTest(tetra,nearestNode) then
                          if not tetra.overlap(tetras) then
                              Update the neighbour relationship between
                              tetra.getNodes()
                              tetra.addNeighbour(tetras [i]) /* convex hull is
                              updated as well */
                              tetras [i].addNeighbour(tetra)
                              updateNeighbours(newlyCreated,tetra) /*
                              updates the tetrahedra relationship between all
                              combinations of elements of newlyCreated and
                              tetra. In case some of them overlap the older one
                              is deleted */
                              newlyCreated.add(tetra)
                          endif
                      endif
                  endif
              endif
          endif
      endfor
  endfor
```

Procedure `void join(Node node)`

```
/* this is the node wishing to join */
nearest ← this.nearest(node)
if nearest==node then
    tetras==node.getNeighbourTetrasOfDegree(2) /* returns the
    tetrahedra of node and of its neighbour nodes */
    liesInside ← false /* true if this lies at least inside one sphere */
    newlyCreated ← null /* variable for all the newly created tetrahedra
    by the execution of the split and enlargeConvexHull methods. */
    for i ← 0 to tetras.length do
        if this lies inside sphere of tetras[i] then
            oldNeighbourTetras ← tetras[i].getNeighbours()
            oldNeighbourNodes ←
            tetras[i].getNeighbourNodesOfDegree(2)
            removeTetra(tetras[i])
            newtetras ← tetras[i].split(node)
            check(newTetras,oldNeighbourNodes) /* performs the
            inCircle test with all combinations of elements of newTetras
            and oldNeighbourNodes. If it fails the affected tetrahedra is
            deleted. */
            check(oldNeighbourTetras, this)
            updateNeighbours(oldNeighbourTetras, newTetras) /*
            updates the tetrahedra relationship between all combinations of
            elements of oldNeighbourTetras and newTetras. In case
            some of them overlap the older one is deleted */
            updateNeighbours(newTetras, newlyCreated)
            newlyCreated.add(newTetras)
            liesInside ← true
        endif
    endfor
    if not liesInside then
        enlargeConvexHull(node,
        node.getNeighbourTetrasOfDegree(2))
    endif
endif
else
    join(nearest)
endif
```

4.3 Leave the triangulation

The leaving node creates a hole in the triangulation. This hole has to be filled with tetrahedra composed of the neighbours of the leaving node [Guibas 04, Mostafavia 03]. The leaving node informs all its neighbours and computes the needed tetrahedra. These are sent to its neighbours. They have already deleted the tetrahedra which are no longer needed, so that the $\mathcal{DT}$ is once again in proper style.
The hole is enclosed by triangles, the remaining parts of the destroyed tetrahedra. Unfortunately not all newly created tetrahedra contain one of these triangles, therefore the inside of the hole must be triangulated without any help from the existing structure.

Procedure `void leave`

```
foreach n in this.getNeighbourNodes() do
    newlyCreated ← null
    n.removeNeighbourNodes(this)
    foreach tetrahedron t in n.getNeighbourTetras() that contains
    this do removeTetra(t)
    common ← common neighbours of this and node
    forall the combinations n1,n2,n3 of nodes out of common do
        newTetra ← new Tetrahedron(n1, n2, n3, n)
        if check(newTetra, node.getNeighbourNodes()) then
            if check(newTetra, this.getNeighbourNodes()) then
                if not newTetra.overlap(n.getNeighbourTetras())
                then
                    Add the neighbour relations between the nodes of
                    newtetra
                    Add newtetra to its nodes n1,n2,n3,n
                    newlyCreated.add(newTetra)
                endif
            endif
        endif
    endfall
    Update the neighbour relations between newlyCreated
    updateNeighbours(node.getNeighbourTetras(), newlyCreated)
endfch
```

There are mainly to options to deal with the crash of a peer c. In both of them the peer l who notices first the crash takes the leadership, and uses the leave procedure to repair the $\mathcal{DT}$ at the place of c. Therefore l has to know all the neighbours of c.

- In the first option l asks for them, that can take some time, because the peers that were neighbours to c can be several hops away from l. Then l repairs the $\mathcal{DT}$.
- In the second option the neighbour lists are regularly exchanged, thus the $\mathcal{DT}$ can be rebuild directly.

For the implementation we chose the second option, because we wanted to minimise the time needed to repair the $\mathcal{DT}$(see 5.1).

4.4 Precision of computing

One major problem that appears during the execution of methods like `isOnSameSide` of `Plane` or `liesInside` of `Sphere` is the precision. In some cases its very difficult to say if a point lies inside or outside or on the surface of a sphere. However this is crucial to the construction of the triangulation. A single wrong decision makes all the building process obsolete [Shewchuk 97]. In the literature these two methods are called $Orient(a, b, c, d)$ — returns a positive value if d lies below the oriented plane passing through a, b, c — and $InSphere(a, b, c, d, e)$ — returns a positive value if e lies inside $\mathcal{C}_{abcd}$. — they may be implemented as matrix determinants:

$$Orient(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}$$

$$InSphere(a,b,c,d,e) = \begin{vmatrix} a_x & a_y & a_z & a_x^2 + a_y^2 + a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2 + b_y^2 + b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2 + c_y^2 + c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2 + d_y^2 + d_z^2 & 1 \\ e_x & e_y & e_z & e_x^2 + e_y^2 + e_z^2 & 1 \end{vmatrix}$$

$\mathcal{DTON}$ uses Java doubles, based on the IEEE 754 double precision numbers, to calculate these tests. During the tests in nearly all cases their precision is sufficient to make the right decision, while in the other cases, even if the test fails, always the same (wrong) answer is returned. Therefore the $\mathcal{DT}$ has this single error only, at least, but it doesn't propagate itself; the triangulation is coherent and can be used for further calculations, even if it is not a $\mathcal{DT}$.

A geometric algorithm is *exact* if it is guaranteed to produce a correct result when given an exact input. An algorithm is *robust* if it always produces the correct output under the real RAM model, and under approximate arithmetic always produces an output that is consistent with some hypothetical input that is a perturbation of the true input [Fortune 92]. Exact arithmetic is attractive when it is applicable, because it can be employed without the time consuming need for careful analysis of a particular algorithm's behaviour when faced with imprecision. On some implementations more than half the developers' time is spent solving problems of roundoff error and degeneracy [Shewchuk 97].

4.5 Tests and validation

To validate the topological consistency of $\mathcal{DTON}$, it is necessary to check for all nodes if they do not lie inside the sphere of a tetrahedron. Moreover, all neighbourship relations between the tetrahedra are checked: Two tetrahedra having a common plane must cross-reference each another.
To validate the results, the output was compared to the results of the CGAL $\mathcal{DT}$ algorithm [CGAL 98, Boissonnat 99], which was the only one (beside qhull [Qhull 95, Barber 96]) to compute the $\mathcal{DT}$ in $\mathbb{R}^3$. CGAL was mainly chosen because the input and output format are easy to handle.
Several different scenarios have been tested to validate the algorithm:

1. Up to 10,000 uniformly distributed nodes.

2. Up to 1000 nodes uniformly distributed on the surface of a sphere, with and without points in the inside. It is not trivial to get a random point on the surface of a sphere [Marsaglia 72].
   Set up a coordinate system (z, ϕ) where z is an arbitrary axis. ($z = -r \ldots r$, where r is the sphere's radius), and where ϕ is the longitude, which runs between 0 and 2π. To generate a random point on the sphere, it is necessary only to generate two random numbers, z and ϕ, each with a uniform distribution. To find the latitude θ of this point, note that $z = r\sin(\theta)$, so $\theta = \arcsin(\frac{z}{r})$; its longitude is simply ϕ. In rectilinear coordinates, $x = r\cos(\theta)\cos(\phi)$, $y = r\cos(\theta)\sin(\phi)$, $z = r\sin(\theta) = z$.
   (x, y, z) are not independent but constrained by $x^2 + y^2 + z^2 = r^2$.
   This case is interesting because it can help to improve the p2p networks based on a two-dimensional structure. The 3D $\mathcal{DT}$ of such a set of points builds some tetrahedra with edges going through the sphere. These connections, called *wormholes* or *highways*, allow fast traveling to far away nodes, only reachable with many hops without these *highways* (Figure 21 and 22).

3. Up to 1000 nodes distributed in clusters with the Lévy Flight. Lévy introduced in 1937 the so called Lévy distribution [Lévy 37]. The Cauchy

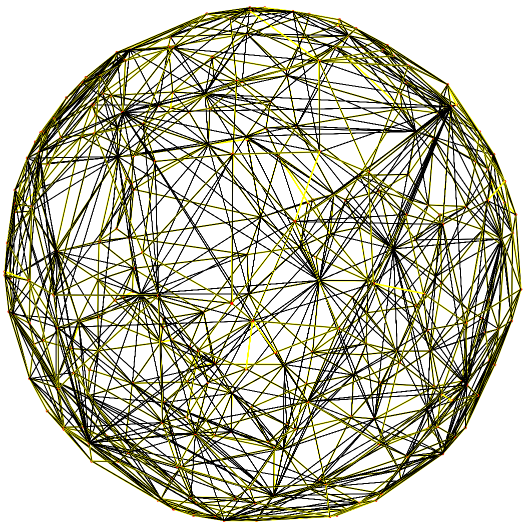

Figure 21: $\mathcal{DT}$ of a set of points situated on a sphere

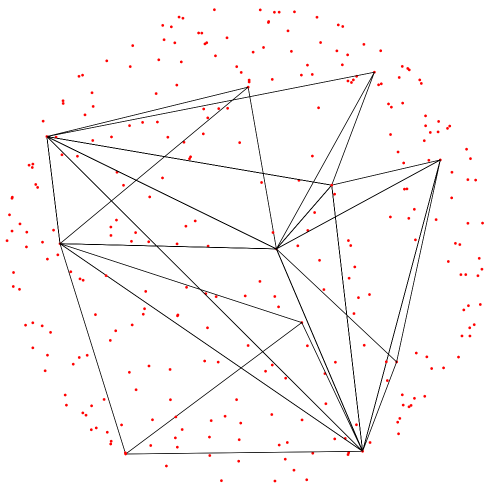

Figure 22: $\mathcal{DT}$ of a set of points situated on a sphere, only the tetrahedra containing *wormholes* are plotted.

and the Gauss normal distribution are special cases of it. Based on the Lévy distribution the Lévy Flight was developed. It produces a random walk through the plane or the space, making many small steps and some big ones (Figure 23). Note that the characteristic size of the system is the size of the largest step and that the flight is self similar at higher magni-

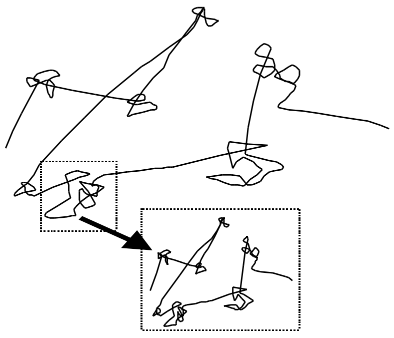

Figure 23: A characteristic Lévy Flight

fications. Removing the path and looking only at the turning points, we get a distribution of points having smaller and bigger clusters (Figure 24). This distribution is more appropriate to represent the real world, than the

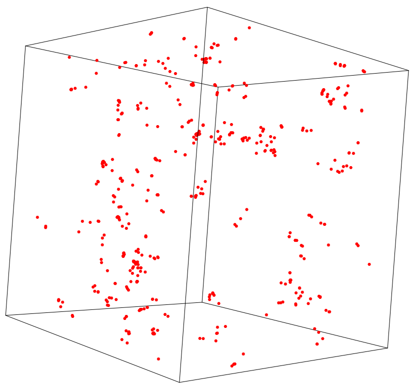

Figure 24: Distribution containing 500 points with clusters resulting from a Lévy Flight (somehow difficult to see where projected to 2D)

uniform distribution of points. The population will be always concentrated around recently found extremum, and in the same time always few population members will explore more distant regions of search space.
Lévy Flights have mathematical properties that discourage a physical approach. They have infinite variance and an analytical form known only for a few special cases [Gutowski 01, Gupta 99]. Therefore the implemented algorithm is much easier: it is based on a simple exponential distribution. The angles are randomly chosen, as described in the part on the uniform distribution on a sphere.

4.6 Simulation results

We run our simulation on a Pentium 4 2.8 GHz and 1GB RAM equipped with RedHat Linux and Java 1.4.2. We want to demonstrate the scalability of $\mathcal{DTON}$. For this reason we chose the following metrics:

- the minimum, maximum and average number of neighbour nodes
- the maximum number of hops between to nodes
- the average computing time needed to join a node
- the number of tetrahedra created
- the number of InCircle tests performed to join a node

All these metrics were used with all three scenarios (*uniform*, *sphere* and *levy*) described above. The cube containing the points has an edge length of 1000 units, the points have a resolution of $\frac{1}{100}$ unit. The variations in the graphs are due to the fact, that the node locations are generated using random numbers. The highest variations occur at the *levy* scenario, due to the clustering of the nodes.
The distribution of the number of neighbours shows clearly the differences between the different scenarios. In scenario *sphere*, half of the nodes have only 8 neighbours or less and only 10% have more than 17 neighbours. This is more than in 2D, where uniformly distributed nodes have on average 6 neighbours in a $\mathcal{DT}$. This is due to the longer and shorter *highways*.
In the scenario *levy* the effect of the clusters is clearly visible, more nodes have fewer neighbours than in *uniform* — the ones on the border of a cluster — but also more nodes have more neighbours than in *uniform* — the ones inside a cluster (Figure 25 and figure 26). These results are important because the performance of the algorithm basically depends on the number of neighbour nodes.
The average number of neighbour nodes is 15 for the scenarios *uniform* and *levy* and 10 for *sphere* (Figure 27). Therefore we did not implement an *attention radius* as does Solipsis (see 2.1) for example. In fact with the structure used it would be easy to do so: instead of opening connections to the direct neighbours only, connections are opened also to the neighbours of second degree (the neighbours of the neighbours), or to the nodes contained by all the neighbour tetrahedra of the tetrahedra containing the considered node.
The main loop of the join procedure finds, among all possibly concerned tetrahedra, those to destroy. Similar to the number of neighbour nodes, the number of tetrahedra destroyed varies for the different scenarios (Figure 28). The number of tetrahedra created is also lower for scenario *sphere* than for the other ones. This is due to the fact that nearly all tetrahedra are created on the surface on the sphere and only very few inside it (Figure 22).
Because of the clusters in the *levy* scenario, the computation of the $\mathcal{DT}$ needs

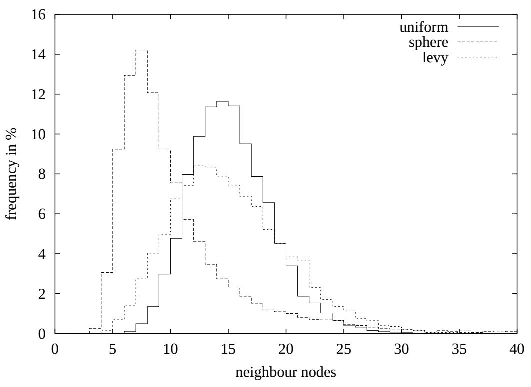

Figure 25: The distribution of the number of neighbours in a $\mathcal{DT}$ of 1000 nodes for the different scenarios. (averages of 10 runs).

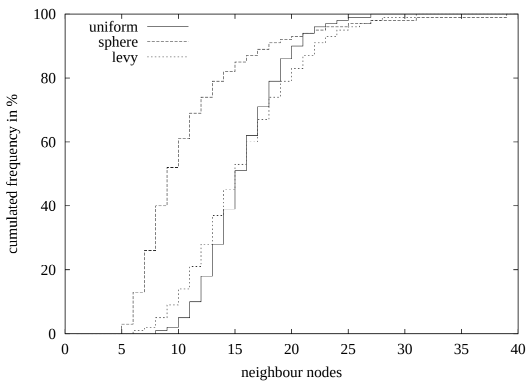

Figure 26: The cumulated distribution of the number of neighbours in a $\mathcal{DT}$ of 1000 nodes for the different scenarios. (averages of 10 runs).

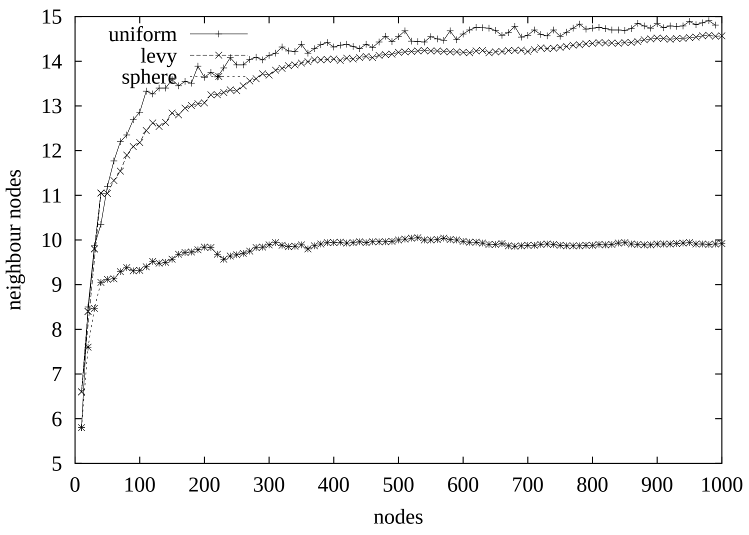

Figure 27: The average number of neighbour nodes depending on the number of nodes in the $\mathcal{DT}$ for the different scenarios.

more inCircle tests (Figure 29). Inside a cluster the nodes are very dense, hence at the border of a cluster the circumspheres of the tetrahedra can be very large, which results in the destruction of many tetrahedra.
The maximum number of hops between two nodes is very low and grows logarithmically with the number of nodes. It is nearly identical for all scenarios (Figure 30, note the logarithmic scale).

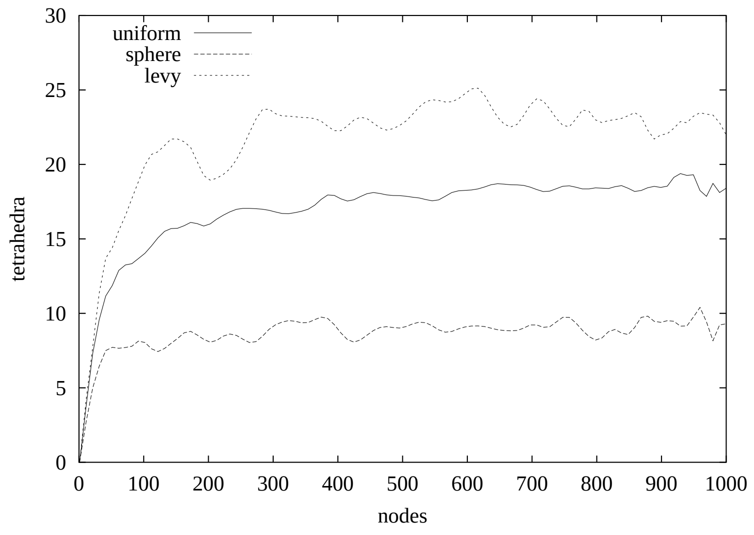

Figure 28: The number of tetrahedra destroyed to join one node to the $\mathcal{DT}$ depending on the number of nodes in the $\mathcal{DT}$ for the different scenarios. (averages of 10 runs)

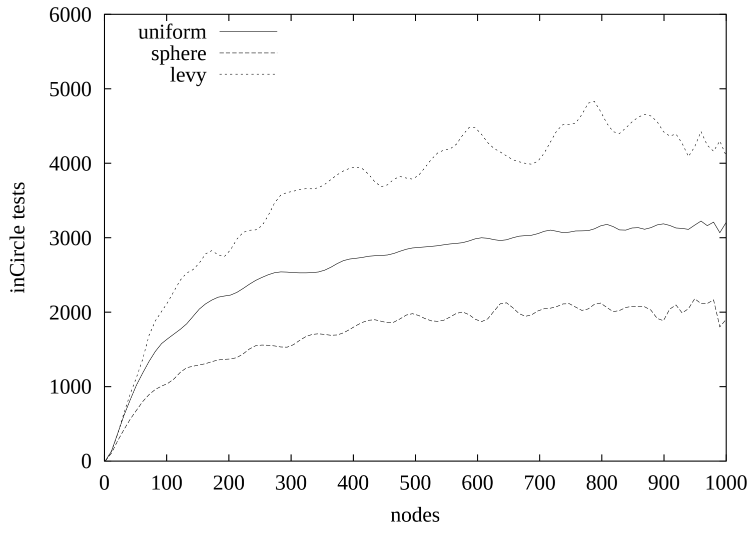

Figure 29: The number of inCircle tests achieved to join one node to the $\mathcal{DT}$ depending on the number of nodes in the $\mathcal{DT}$ for the different scenarios. (averages of 10 runs)

The time to join a node to the $\mathcal{DT}$ is bounded: as one can clearly see, the join of a node has only a local effect (Figure 31). The scalability of $\mathcal{DTON}$ is therefore ensured.

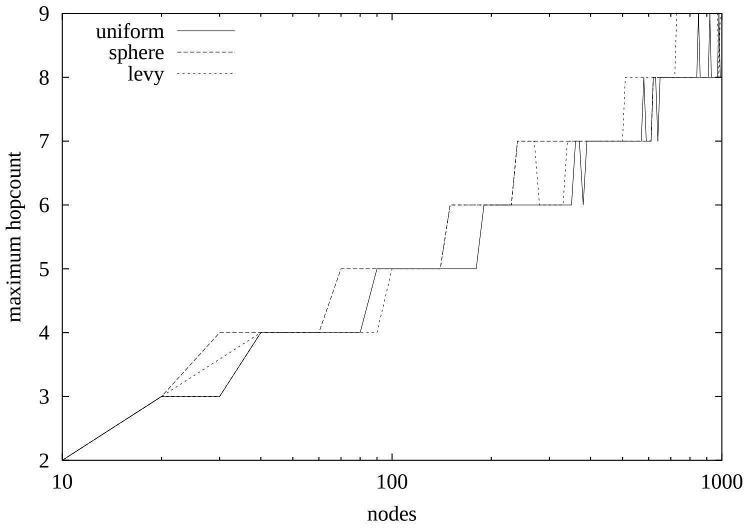

Figure 30: The maximum hopcount depending on the number of nodes in the $\mathcal{DT}$ for the different scenarios.

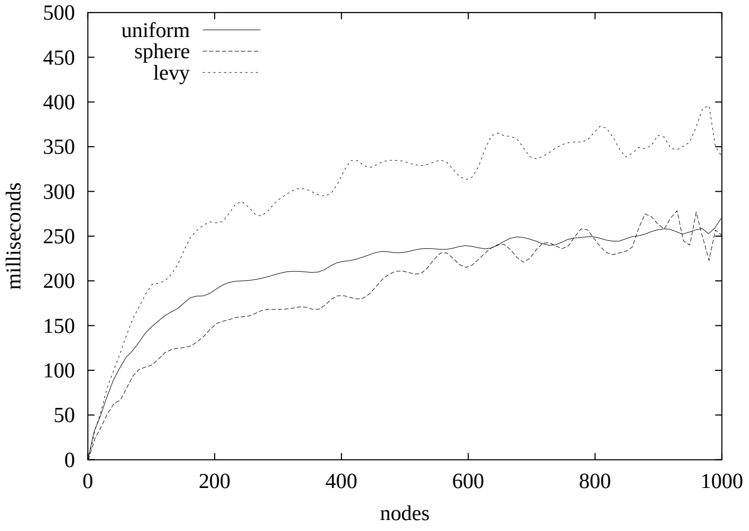

Figure 31: The time to join one node to the $\mathcal{DT}$ depending on the number of nodes in the $\mathcal{DT}$ for the different scenarios. (averages of 10 runs)

The next section presents a fully distributed p2p structure based on the specified algorithm and data structures.

5 A fully distributed p2p structure based on the 3D $\mathcal{DT}$

The algorithm described in the previous chapter provides the logic needed to build a Delaunay triangulation in 3D with the computers participating in the peer to peer network acting as points. In order to distribute it on different computers, and not only to simulate that distribution, as in the last section, a communication layer has to be added.
First the protocol of this layer is described with its different message types, then two major problems are discussed: the coherence of the triangulation during and after the concurrent execution of more than one task and the crash of a peer and the involved repair of the triangulation.

5.1 The protocol

The protocol has 16 different message types to inform neighbour peers about changes or to gather information.

JOIN Requests a `Peer` object starting from an IP address and a port. This is always the first message from a peer aiming to join the networks.

NEIGHBOURNODES Requests the list of the neighbour nodes of the receiver.

NEIGHBOURTETRAS Requests the list of the neighbour tetrahedra of the receiver.

ADDNEIGHBOURNODE Adds a neighbour node to the receiver.

ADDNEIGHBOURTETRA Adds a neighbour tetrahedron to the receiver.

REMOVENEIGBOURNODE Removes a neighbour node from the receiver.

REMOVENEIGBOURTETRA Removes a neighbour tetrahedron from the receiver.

FINISH Only for debugging: prints all neighbour nodes and tetrahedra to a file.

LOCK Locks the receiver for other operations.

JOINED Asks if the join procedure of the receiver is completed.

CLEARNEIGBHOURNODES Deletes all neighbour nodes from the receiver.

SEARCH Searches (recursively) a node closest to a given position. If the receiver is closer to the given position than any other peer the search stops.

HEARTBEAT Sends a heartbeat message.

MESSAGE Sends a simple text message.

UPDATENEIGHBOURNODE Sends the list of all neighbour nodes to all neighbour nodes after a change. This ensures absolute coherence needed in case of a crash, but slows down all operations.

UPDATENEIGHBOURTETRA Sends the list of all neighbour tetrahedra to all neighbour nodes after a change.

It is crucial for the coherence of the virtual world, to locally accept only one alteration at a time. For example two peers may join or leave the network at the same time only if they do not alter the same nodes or tetrahedra. Otherwise the two (or more) peers would create or destroy tetrahedra or peer relationships without knowing about the actions of the other one. That would inevitably lead to an inconsistent triangulation. To avoid this all peers involved in a join or leave procedure are locked for other procedures. They can still perform actions, e.g. send messages, but they cannot alter the triangulation. This approach is comparable to the lock mechanisms implemented in database systems to allow transactions to conform with the ACID paradigm.
The peer that wants to join tries to lock all involved nodes. If it doesn't succeed, it unlocks them and retries after a randomly chosen time out of a defined interval to avoid deadlocks. Furthermore it has to wait until all involved nodes have finished their own join procedure and are a part of the triangulation. After the successful join, it unlocks the locked nodes.
In the case of a crash of one of the peers, the other peers have to take notice of it. Therefore heartbeat messages are sent to the neighbours. If one peer notices that it has not received a heartbeat message from one of its neighbours for a certain time, it tries to reestablish the connection; if this fails it takes over the leadership of the involved peers. The peer locks all the neighbours of the crashed peer and executes the leaving procedure (see 4.3) for the crashed peer. The former neighbours are disconnected from the crashed peer and the $\mathcal{DT}$ is rebuilt. Then the locked peers are unlocked again.
In 2D this precaution is unnecessary since there exists only one unique $\mathcal{DT}$, so the order the peers connect to the network does not play a role (see 3.2.3).

5.2 Tests

The results of the application could not be checked in the same way as the simulation's results, because there is no global view of all nodes and all tetrahedra. Each peer has its own view of the virtual world containing its neighbours (Figure 32 and 33).
To obtain a global view, all peers write the tetrahedra they know in a common file (Figure 34). This file is then compared with the output of the simulation or the CGAL output.

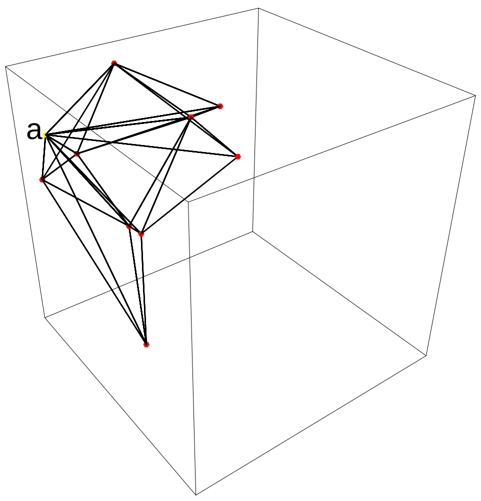

Figure 32: Local view of the $\mathcal{DT}$ by peer a.

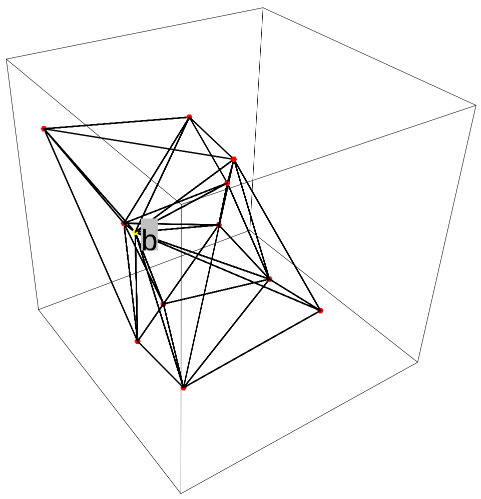

Figure 33: Local view of the $\mathcal{DT}$ by peer b.

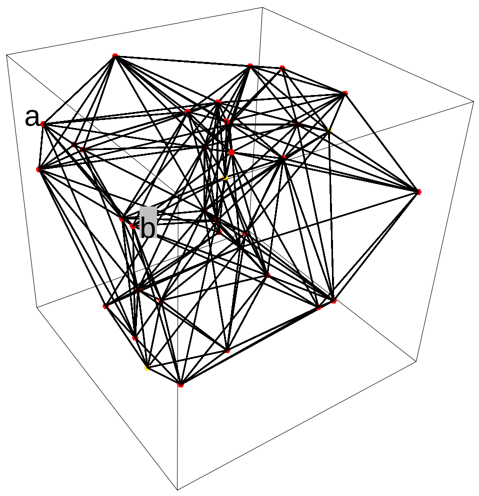

Figure 34: Global view of the $\mathcal{DT}$ obtained by superposing all local views.

The network protocol was tested on up to 40 computers in the Eurécom student computer pool. The most important measurements have already been done on the simulation, except for one that cannot be done without real communication between the peers: the overhead measurement. The peers in the simulation are only Java objects, not independent applications, hence they do not need a protocol to communicate and the protocol overhead cannot be measured.

The size of the protocol messages is around 500 bytes. The number of messages sent increases with the number of neighbours. As shown in section 4.6 the

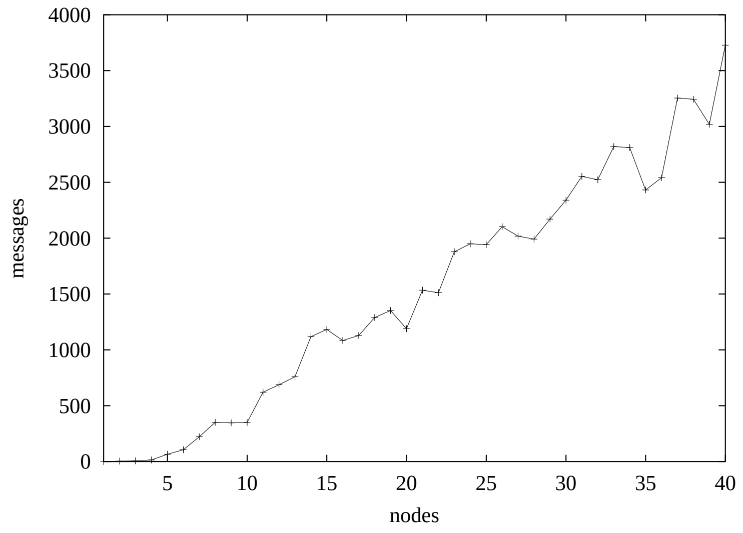

Figure 35: The number of messages sent to join a peer, depending on the number of peers present in the network.

average number of neighbours is 15. This number is only reached if more than 100 peers are connected. Therefore the number of messages does not tend to an upper bound with only 40 peers (Figure 35).

5.3 A comparison to other fully distributed p2p structures

In terms of design concepts, $\mathcal{DTON}$ is most similar to Solipsis, where each node maintains a certain number of neighbouring nodes (according to some rules), and neighbour discovery is done by mutual collaboration. Both make direct connections among peers, which makes message transmission efficient. Both are fully distributed, so do not need to worry about super-node failure or overloading a particular node. The key difference between $\mathcal{DTON}$ and Solipsis lies in which information about neighbouring nodes are maintained. In Solipsis the rule is that each node must be contained within a convex hull formed by its neighbours (refer back to figure 1). However, there are cases where Solipsis may not discover a neighbour properly (refer back to figure 3), yet the same scenario would not happen for $\mathcal{DTON}$. The join procedure of $\mathcal{DTON}$ is much shorter than that one of Solipsis; the nearest peer to the desired location from the node that wishes to join is found in a few steps (Figure 30), whereas Solipsis needs many queries (see 2.1 and especially figure 4).

SimMud [Knutsson 04] is also a scalable system. However, its main problem is the additional latency introduced by the super-nodes (coordinators). In SimMud all messages must first be sent to the coordinator before the coordinator dispatches them to the affected nodes individually. This results in increased loading for coordinator nodes and increased latency due to relay by the coordinator nodes. Furthermore back-up mechanisms for the coordinators must be provided in case that they fail. Whereas $\mathcal{DTON}$ would not overload any particular node, as the system is fully distributed, so no single node bears more responsibility and work than another node. Latency is minimised as all peers may direct connect to each other, without any message relay. Also, because there is no super-node, no special back-up or recovery mechanism is needed. However, by centralising certain aspects of message delivery, SimMud is able to leverage message compression and aggregation techniques to reduce bandwidth consumption. Furthermore SimMud deals with persistent data, a feature that neither $\mathcal{DTON}$ nor Solipsis have, which is the main reason for the complicated backup mechanisms.

The next section discusses how to efficiently build multicast trees using the $\mathcal{DT}$ structure.

6 Multicast Trees

Multicast transmission of data eliminates duplicate data packet copies that would otherwise traverse those links that are common to two or more of the source to receiver shortest paths. Therefore a multicast tree is needed. In this section we show how to efficiently calculate a multicast tree using the structure of the $\mathcal{DT}$. To do so, first two basic notions are explained: *Minimum spanning trees* and *Overlay Multicast*.

6.1 Minimum spanning trees

Given a connected, undirected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a weight to each edge, which is a number representing how unfavourable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree.
A *minimum spanning tree* or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. One famous algorithm to compute a minimum spanning tree for a connected weighted graph was developed by Prim [Prim 57]. It works as follows [Cormen 01]:

```
create a tree $T$ containing a single vertex, chosen arbitrarily from the graph
create a set $S$ containing all the edges in the graph
while not every edge in $S$ connects two vertices in $T$ do
    remove from $S$ an edge $e$ with minimum weight that connects a vertex
    in $T$ with a vertex not in $T$
    add $e$ to $T$
endw
```

Shamos and Hoey [Shamos 75] recognised that the edges of a minimum spanning tree must be Delaunay edges (Figure 36). This interesting and important property still holds. That means if the minimum spanning tree of a point set S is built, all the edges of the tree are edges of $\mathcal{DT}(S)$. In fact, $\mathcal{O}(n)$ time suffices to derive the minimum spanning tree from the $\mathcal{DT}$ [Preparata 85]. This gives, for the 2D case, an $\mathcal{O}(n \log n)$-time algorithm, which is optimal by reduction to sorting n real numbers [Aurenhammer 91].

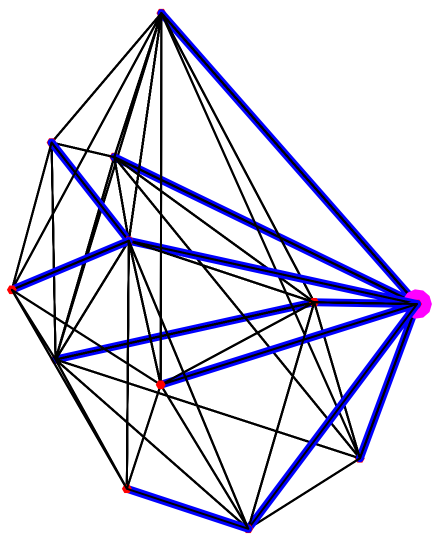

Figure 36: A multicast tree based on the $\mathcal{DT}$ including the tetrahedra. Notice that every edge of the tree is an edge of a tetrahedron of the $\mathcal{DT}$.

6.2 Overlay Multicast

IP Multicast is a technique that can transmit one copy of data traffic to multiple receivers at one time. Because it can greatly save network bandwidth consumption and because many applications are inherently multicast-based, it has been a research focus since the idea was proposed. However, for many reasons, such as management, security, and inter-domain routing it has not been widely deployed. Recently, many researchers have put their research focus on Overlay Multicast where the data replication, multicast routing, group management, and other functions are all achieved at application layer. Since Overlay Multicast does not require changing the current Internet infrastructure, it can easily be deployed. In dynamic node-based Overlay Multicast, the group members are self-organised into an overlay multicast tree. In large multicast groups, frequent joining or leaving events will occur. How to adapt to these changes is one of the main issues considered. How to scalably form an efficient multicast tree is another important issue [Wei 94, Liebeherr 02].

6.3 Multicast tree based on the $\mathcal{DT}$

Assuming the nodes of the $\mathcal{DT}$ are uniformly distributed (scenario *uniform*) Prim's algorithm can be used almost instantly. All neighbour nodes of the source node belong to the tree. All their neighbours are added to the tree if they are outside the tree or if the new edge is cheaper (our cost function is the Euclidean

distance) than the old one (in this case the old edge is removed).
It gets more complicated if the nodes are distributed in clusters (scenario *levy*). First the nodes must be partitioned into clusters. Thereafter the tree of the clusters must be computed.
Since there exist powerful clustering algorithms [Aurenhammer 91] we ignored that problem and take the cluster allocation from the algorithm, which computes the nodes distribution.
The data structure used for the implementation is rather simple: The main class is the `TreeNode`; it has a reference to its ancestor, a `TreeNode`, and to its leaves, `TreeNode`s. Furthermore it has a reference to the `Node` of the $\mathcal{DT}$ to which it belongs and its weight (distance of the edges connecting it to the source) is stored in `distance`. The tree itself is represented by the class `SpanningTree`; it mainly consists of a reference to the root `TreeNode` of the tree. The `Cluster`, if it is connected to the tree, has references to the `TreeNode` outside the cluster where the connections come from `connectedFrom` and to the `TreeNode` inside the cluster where the connection goes to `connectedTo`. Moreover its weight is stored in `distance`.
There are two different approaches to computing the tree:

- The source builds the tree to a set of nodes.
- An interested node searches a path to the source. If it encounters a node already in the tree, it stops its search. All the paths together form the tree.

First the procedures for building the tree from the *source* are described. The procedure `spanningTreeActive` takes as argument the maximum hopcount from the source to a receiver. Subsequently the `spanningTreeInt` method is recursively executed for each neighbour node until the allowed path length is reached. Each called node checks if its cluster is already connected to the tree. If it is not, the node is added to the tree. If the cluster is connected, it checks if the path including the new edge is shorter than the existing connection to the cluster. In this case the former path is removed and replaced by the new one.

```
Procedure SpanningTree spanningTreeActive(int n)
  tree ← new SpanningTree(this)  /* this is the source */
  foreach node of this.getNeighbourNode() do
    if not node.getCluster.isConnected() then
      TreeNode leaf ← new TreeNode(node)
      tree.getRoot().addLeaf(leaf)
      node.getCluster().setConnected(tree.getRoot(), leaf) /*
      the arguments are connectedFrom and connectedTo */
    endif
  endfch
  spanningTreeInt(tree.getRoot(), n-1)
  Return tree
```

Procedure SpanningTree spanningTreeInt(*TreeNode node, int n*)

```
if n > 0 then
    foreach treeNode of node.getLeaves() do
        foreach node of treeNode.getNode().getNeighbourNode() do
            if not node.getCluster().isConnected() then
                TreeNode newLeaf ← new TreeNode(node)
                treeNode.addLeaf(newLeaf)
                node.getCluster().setConnected(treeNode, newLeaf)
            endif
            else
                if tree.contains(node) then
                    /* cluster connected and node already in the tree */
                    if ancestor of node not equals treeNode then
                        if treeNode.getNode().getCluster() not equals
                        node.getCluster() then
                            /* ancestor is not in the same cluster than node
                            */
                            if (treeNode.getDistance() + distance from
                            node.getCluster().getConnectedTo() to
                            treeNode ) < node.getCluster().getDistance()
                            then
                                /* the new connection to the cluster is
                                shorter */
                                node.getCluster().getConnectedFrom().
                                removeLeaf(node.getCluster().getConnectedTo())
                                treeNode.addLeaf(node.getCluster().getConnectedTo())
                                node.getCluster().setConnected(treeNode,
                                node.getCluster().getConnectedTo())
                            endif
                        endif
                    endif
                endif
                else
                    /* cluster connected but node not inserted in the tree */
                    node.getCluster().getConnectedTo().addLeaf(new
                    TreeNode(node))
                endif
            endif
        endfch
        if n > 1 then
            spanningTreeInt(treeNode, n-1)
        endif
    endfch
endif
```

The clusters only have one incoming connection. All the nodes inside a cluster are connected to the node `connectedTo` (Figure 37).

In the *receiver* option to build a multicast tree, the source is passive and the receivers search a way to the source (Figure 38). If they encounter a node that is already in the tree they stop their search. In the case the cluster of the receiver is already connected, it simply opens a connection to the node of its cluster that is connected to the outside.

```
Procedure void joinTree(Node node)
    /* node is the source */
  if this is not yet in tree then
      if this's cluster is not yet in the tree then
          nearest ← neighbour of this nearest to node
          if node == this then
              node.treeNode.addLeaf(this)
          endif
          else
              if nearest is not yet in tree then
                    /* join nearest first */
                  nearest.joinTree(node);
              endif
              nearest.treeNode.addLeaf(this)
          endif
      endif
      else this.getCluster.getConnectedTo.addLeaf(this)
  endif
  else  /* this is already in tree */
```

The *source* method builds a Shortest Path Tree. It is composed of the shortest paths between the source and each of the receivers. Such a tree does not minimise the total cost (length) of the tree. However the cost caused by the *receiver* method are much higher, because once a cluster is connected to the tree no cheaper connection is searched.
On figure 37 and 38 the two methods have been used to compute multicast trees on the same set of 100 nodes of scenario *levy*. Executing both algorithms 10 times on a $\mathcal{DT}$ of 100 nodes of the *levy* scenario inside a cube with an edge length of 100 units, showed that the tree computed by the *source* algorithm has an average total length of 1169 units, whereas the tree computed by the *receiver* algorithm has an average total length of 1309 units.

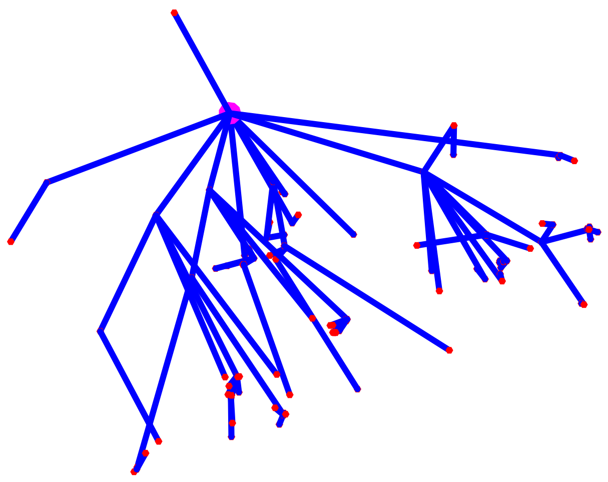

Figure 37: Multicast tree computed by the source, notice the clusters. The tetrahedra are not plotted for better visibility.

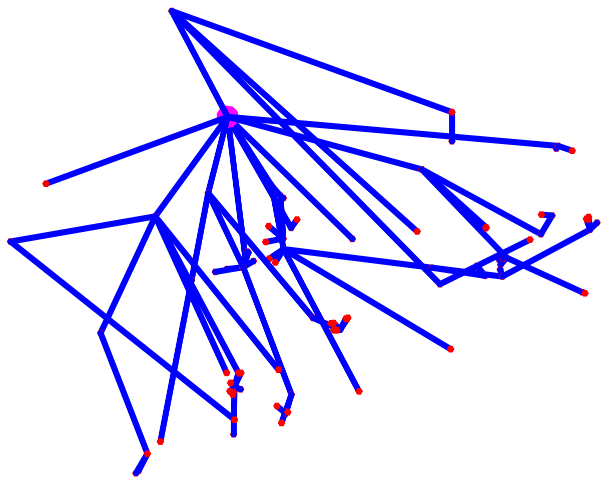

Figure 38: Multicast tree computed by the receivers, notice the clusters. The tetrahedra are not plotted for better visibility.

7 Conclusion

7.1 A possible solution to the scalability problem

We have presented a promising solution for constructing scalable p2p networks based on the 3D Delaunay triangulation. The key idea of the design is for each node to maintain a $\mathcal{DT}$ of the neighbour nodes. Although demonstrating scalability in a real system is not practical for the current work (tests could only be run on 40 computers), we have shown the scalability potential of 3D $\mathcal{DT}$ with simulation results.
In the simulation, it is shown that there are upper bounds to the time needed to join and the number of average neighbours maintained by a peer. This indicates that the amount of bandwidth and processing requirement for each node is bound, independent of the total number of nodes in the system. From this it follows that the system is scalable.

7.2 Future work

Many fields of interest have been touched upon by this thesis but could not be elaborated.

- The problem of precision (see section 4.4) is not really solved. If too many nodes are in the virtual space close by, the numbers of Java are not precise enough. However, tests with up to 10,000 nodes, having a resolution of $\frac{1}{100}$ unit in a cube having an edge length of 100 units, were always successful.

- The movement of nodes can only be simulated by executing the leave and join procedures, hence it is possible to move nodes within a certain area without destroying the $\mathcal{DT}$. If the node has to be moved farther, some algorithms can be developed that are more efficient than the combination of the leave and join method.

- As described in section 5.1, to keep the $\mathcal{DT}$ coherent during modifications, the involved nodes have to be locked. This entails the sending of many messages and consumes most of the time of the overall process. Future research may find a way to keep the coherence without locking the nodes.

- Most of the data exchanged are serialised Java objects. Sending the data in a smarter format could drastically reduce the average message size.

- The algorithms developed are not yet integrated into Solipsis. At the moment the research group around Keller and Simon is working on a second version, which will be completely unitised. Thus the integration into the new version is going to be much easier than to change the current Solipsis code.

- The algorithm for building multicast trees does not take into consideration the crash of a node. Therefore heartbeat messages have to be exchanged; if a node notices that its ancestor is crashed, it has to execute the join procedure again to be reconnected to the tree. Its children are not affected.

References

[Araujo 01] F. Araujo and L. Rodriguez, “GeoPeer: A Location-Aware Peer-to-Peer System”, , Faculdade de Ciências da Universidade de Lisboa, Portugal, 2001.

[Aurenhammer 00] F. Aurenhammer and R. Klein, *Handbook of Computational Geometry*, chapter 18, pp. 201–290, Elsevier Science Publishers, 2000.

[Aurenhammer 91] F. Aurenhammer, “Voronoi diagrams — a survey of a fundamental geometric data structure”, *ACM Computing Surveys (CSUR)*, 23(3):345–405, September 1991.

[Barber 96] C. B. Barber, D. P. Dobkin and H. Huhdanpaa, “The Quickhull Algorithm for Convex Hulls”, *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[Berg 00] M. D. Berg, M. V. Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 2nd edition, 2000.

[Blizzard 04] Blizzard, “The website of World of Warcraft”, http://www.worldofwarcraft.com, 2004.

[Boissonnat 01] J.-D. Boissonnat, “Diagrammes de Voronoï, Triangulations et Surfaces”, 2001.

[Boissonnat 02] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud and M. Yvinec, “Triangulations in CGAL”, *Computational Geomety: Theory and Applications*, 22:5–19, 2002.

[Boissonnat 99] J.-D. Boissonnat, F. Cazals, F. Da, O. Devillers, S. Pion, F. Rebufat, M. Teillaud and M. Yvinec, “Programming with CGAL: the example of triangulations”, *Proceedings of the fifteenth annual symposium on Computational geometry*, pp. 421–422, 1999.

[Calvin 93] J. Calvin, A. Dickens, B. Gaines, P. Metzger, D. Miller and D. Owen, “The SIMNET virtual world architecture”, *Virtual Reality Annual International Symposium IEEE*, pp. 450–455, September 1993.

[Castro 02] M. Castro, P. Druschel, A.-M. Kermarrec and A. Rowstron, “SCRIBE: A large-scale and decentralized application-level multicast infrastructure”, *IEEE Journal on Selected Areas in communications*, 2002.

[CGAL 98] CGAL, "The CGAL website", http://www.cgal.org, 1998.

[Cormen 01] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, MIT Press and McGraw-Hill, 2nd edition, 2001.

[Delaunay 34] B. Delaunay, "Sur la sphère vide. A la mémoire de Georges Voronoï.", *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskih i Estestvennyh Nauk*, 7:793–800, 1934.

[Devillers 01] O. Devillers, S. Pion and M. Teillaud, "Walking in a triangulation", *Proceedings of the seventeenth annual symposium on Computational geometry*, pp. 106–114, 2001.

[Devillers 03] O. Devillers and M. Teillaud, "Perturbations and vertex removal in a 3D Delaunay triangulation", *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 313–319, 2003.

[Dobkin 87] D. P. Dobkin and M. J. Laszlo, "Primitives for the manipulation of three-dimensional subdivisions", *Proceedings of the third annual symposium on Computational geometry*, pp. 86–99, 1987.

[eMule 02] eMule, "The website of Emule", http://www.emule-project.net, 2002.

[Fortune 87] S. Fortune, "A sweepline algorithm for Voronoi diagrams", *Proceedings of the second annual symposium on Computational geometry*, pp. 313–322, 1987.

[Fortune 92] S. Fortune, "Numerical stability of algorithms for 2D Delaunay triangulations", *Proceedings of the eighth annual symposium on Computational geometry*, pp. 83–92, 1992.

[Funcom 99] Funcom, "The website of the Anarchy Online game", http://www.anarchy-online.com, 1999.

[geomview 92] geomview, "The Geomview website", http://www.geomview.org, 1992.

[Gnutella 00] Gnutella, "The website of GNUTELLA", http://www.gnutella.com, 2000.

[Green 78] P. J. Green and R. R. Sibson, "Computing Dirichlet tessellations in the plane", *Computer Journal*, 21:168–173, 1978.

[Guibas 04] L. Guibas and D. Russel, "An Empirical Comparison of Techniques for Updating Delaunay Triangulations", *Symposium on Computational Geometry*, June 2004.

[Guibas 85] L. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi", *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[Gupta 99] H. M. Gupta and J. R. Campanha, "The gradually truncated Lévy Flight for systems with power-law distributions", *Physica A*, 268:231–239, 1999.

[Gutowski 01] M. Gutowski, "Lévy flights as an underlying mechanism for global optimization algorithms", *ArXiv Mathematical Physics e-prints*, June 2001.

[Heller 90] M. Heller, "Triangulation algorithms for adaptive terrain modelling", *Proceedings of the fourth International Symposium on Spatial Data Handling*, pp. 163–174, 1990.

[Hu 04] S.-Y. Hu and G.-M. Liao, "Scalable Peer-to-Peer Networked Virtual Environment", *SIGCOMM Workshops*, August 2004.

[Icking 96] C. Icking, R. Klein, P. Köllner and L. Ma, "The website of VoroGlide at the FernUniversität Hagen", http://www.pi6.fernuni-hagen.de/GeomLab/VoroGlide, 1996.

[Keller 02] J. Keller and G. Simon, "Toward a Peer-to-Peer Shared Virtual Reality", *IEEE workshop on resource sharing in massively distributed systems*, July 2002.

[Keller 03] J. Keller and G. Simon, "SOLIPSIS: A Massively Multi-Participant Virtual World", *International Conference on Parallel and Distributed Techniques and Applications*, 2003.

[Keller 04] J. Keller and G. Simon, "The website of solipsis", http://solipsis.netofpeers.net, 2004.

[Knutsson 04] B. Knutsson, H. Lu, W. Xu and B. Hopkins, "Peer-to-Peer Support for Massively Multiplayer Games", *INFOCOM*, March 2004.

[Liebeherr 02] J. Liebeherr and M. Nahas, "Application-layer multicasting with Delaunay triangulation overlays", *IEEE Journal on Selected Areas in Communications*, 20(8):1472–1488, October 2002.

[Liebeherr 03]	J. Liebeherr, G. Dong, H. Lu, J. Wang and G. Zhang, "The website of HyperCast", http://www.cs.virginia.edu/∼mngroup/hypercast/, 2003.
[Lv 02]	Q. Lv, P. Cao, E. Cohen, K. Li and S. Shenker, "Search and replication in unstructured peer-to-peer networks", *International Conference on Supercomputing, New York, USA*, pp. 84–95, 2002.
[Lévy 37]	P. Lévy, *Théorie de l'Addition des Variables Aléatories*, Gauthier-Villars, Paris, 1937.
[Marsaglia 72]	G. Marsaglia, "Choosing a point from the surface of a sphere", *The Annals of Mathematical Statistics*, 43:645–646, 1972.
[Mücke 93]	E. Mücke, *Shapes and Implementations in Three-Dimensional Geometry*, Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1993.
[Mesa3d 99]	Mesa3d, "The Mesa3d website", http://www.mesa3d.org, 1999.
[Mostafavia 03]	M. A. Mostafavia, C. Gold and M. Dakowicz, "Delete and insert operations in Voronoi/Delaunay methods and applications", *Computers & Geosciences*, volume 29, pp. 523–530, 2003.
[Napster 99]	Napster, "The website of Napster", http://www.napster.com, 1999.
[O2OE 98]	O2OE, "The website of the Mankind game", http://www.mankind.net, 1998.
[Odgaard 00]	A. Odgaard and B. K. Nielsen, "The website of a java-applet demonstrating Fortune's sweep line algorithm", http://www.diku.dk/hjemmesider/studerende/duff/Fortune, 2000.
[OpenMotif 00]	OpenMotif, "The OpenMotif website", http://www.openmotif.com, 2000.
[Oram 01]	A. Oram, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, O'Reilly, 2001.
[Ottmann 01]	T. Ottmann, "Lecture on computational geometry: the Voronoi Diagram", 2001.

[Pande 01]	V. Pande, “The website of GENOME@HOME”, http://www.stanford.edu/group/pandegroup/genome/, 2001.
[Preparata 85]	F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, Springer, 1985.
[Prim 57]	R. Prim, “Shortest connection networks and some generalizations”, *Bell System Technical Journal*, 268:231–239, 1957.
[Qhull 95]	Qhull, “The qhull website”, http://www.qhull.org, 1995.
[Rowstron 01]	A. Rowstron and P. Druschel, “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”, *Lecture Notes in Computer Science*, 2218:329+, 2001.
[Saroiu 02]	S. Saroiu, P. K. Gummadi and S. D. Gribble, “A measurement study of peer-to-peer file sharing systems”, *Multimedia Computing and Networking (MMCN)*, 2002.
[Shamos 75]	M. I. Shamos and D. Hoey, “Closest-point problems”, *16th IEEE Annual Symposium on the Foundations of Computer Science*, pp. 151–162, October 1975.
[Shewchuk 97]	J. R. Shewchuk, “Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates”, *Discrete and Computational Geometry*, 18(3):305–363, October 1997.
[Simon 04]	G. Simon, *Conception et réalisation d’un système pour environnement virtuel massivement partagé*, Ph.D. Thesis, University of Rennes, 2004.
[Skype 03]	Skype, “The website of Skype”, http://www.skype.com, 2003.
[Sony 99]	Sony, “The website of the Everquest game”, http://everquest.station.sony.com, 1999.
[Steiner 05]	Steiner, “The website of 3D Delaunay Triangulation Overlay Network”, http://www.eurecom.fr/∼steiner/dton, 2005.
[Stephenson 92]	N. Stephenson, *Snow Crash*, 1992.

[Voronoï 07] G. Voronoï, "Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Premier Mémoire: Sur quelques propriétés des formes quadritiques positives parfaites.", *Journal für die Reine und Angewandte Mathematik*, 133:97–178, 1907.

[Voronoï 08] G. Voronoï, "Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième Mémoire: Recherches sur les parallelloèdres primitifs.", *Journal für die Reine und Angewandte Mathematik*, 134:198–287, 1908.

[Wachowski 99] L. Wachowski and A. Wachowski, "The Matrix", Movie, 1999.

[Wei 94] L. Wei and D. Estrin, "A comparison of multicast tree algorithms", *IEEE INFOCOM - The Conference on Computer Communications*, 1994.

[Yang 03] B. Yang and H. Garcia-Molina, "Designing a Super-peer Network", *19th International Conference on Data Engineering (ICDE), Bangalore, India*, March 2003.

[Yvinec 98] M. Yvinec and J.-D. Boissonnat, *Algorithmic Geometry*, Cambridge University Press, 1998.

[Zhou 04] S. Zhou, W. Cai, B.-S. Lee and S. J. Turner, "Time-space consistency in large-scale distributed virtual environments", *ACM Transactions on Modeling and Computer Simulation*, 14(1):31–47, 2004.

Index

A Installation instructions

This section explains how to configure and install all the software. The directory `/homes/steiner` is of course only an example.

A.1 geomview

Geomview [geomview 92] is needed to visualise the triangulation. It is very helpful for debugging to see the tetrahedra and the convex hull. Geomview does not run under any version of windows. To install it on linux OpenMotif [OpenMotif 00] and Mesa[Mesa3d 99] have to be installed.
First download and install the packages:

```
$ tar -xzvf openMotif-2.2.3.tar.gz
$ rm openMotif-2.2.3.tar.gz

$ tar -zxvf MesaLib-6.1.tar.gz
$ rm MesaLib-6.1.tar.gz
```

Then configure and install them:

```
$ cd openMotif-2.2.3
$ ./configure --prefix=/homes/steiner/openMotif
$ make
$ make install
$ cd ..

$ rm -rv openMotif-2.2.3
$ cd Mesa-6.1/
$ make linux-x86
$ cd ..
```

The actual version 1.8.1 (from March 2001) on the geomview website [geomview 92] is not compilable on actual linux version (8 or 9). Therefore the beta-version 1.8.2 is needed. The easiest way to get it, is to google for `geomview-snapshot-2004-02-21.tar.gz`. After downloading and unpacking it, configure and install it like this:

```
$ ./configure --prefix=/homes/steiner/geomview
--with-motif=/homes/steiner/openMotif/
--with-opengl=/homes/steiner/Mesa-6.1/
$ make
$ make install
```

Finally the directory `/homes/steiner/geomview/bin/` must be added to the PATH.

The simulation of $\mathcal{DTON}$ outputs files that can be displayed by geomview, to do so enter: `geomview list.LIST`

A.2 CGAL

According to the people working on CGAL is the only bugfree and working algorithm to compute the $\mathcal{DT}$. Therefore (and because because the input and output format is easier to handle than that one of qhull [Qhull 95]) it is used to verify the results of $\mathcal{DTON}$.
The installation procedure is well described on the CGAL website [CGAL 98].
A simple C++ program (`cgal.C`) that runs the CGAL triangulation algorithm with points read from a file and outputs the resulting tetrahedra in a format readable by $\mathcal{DTON}$ can be downloaded at [Steiner 05]. The input file with the coordinates has to be name `points.out`, the generated output file with the tetrahedra is named `tetras.out`.

A.3 The simulation of $\mathcal{DTON}$

The simulation of $\mathcal{DTON}$ can be downloaded at [Steiner 05]. Java 1.4 has to be installed on the running machine. To start the simulation enter: `java -classpath /homes/steiner/Delaunay3dDistributedSimulation/classes DtonSim`
The nodes for the computation of the $\mathcal{DT}$ can be randomly chosen or read out of a file. The class `LevyFlight` can be used to create node distributions according to scenario 2 or 3 (see 4.5). The output can be compared to the result of another algorithm (e.g. CGAL), to do so the results of this other algorithm have to be stored in a file `tetras.out`. All the parameters have to be changed directly in the main class `DtonSim`.

A.4 The distributed $\mathcal{DTON}$algorithm

The distributed algorithm of $\mathcal{DTON}$ itself can be downloaded, as well as the simulation, at [Steiner 05]. To start it enter: `java -classpath /homes/steiner/Delaunay3dDistributedSimulation/classes Dton number`. `number` is needed for debug purposes. For the execution of $\mathcal{DTON}$ on many machines at once, there can be found a script named `testdisb` on the website. It reads file with the list of IPs on which $\mathcal{DTON}$ should be executed.

B Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Sophia-Antipolis, den 29. März 2005 Moritz Steiner