Conformal Geometric Algebra Package

Chaïm Zonnenberg

THESIS

INF_SCR_06_66

Utrecht University
Department of Information & Computing Sciences

July 23, 2007

Universiteit Utrecht

Conformal Geometric Algebra Package

Abstract

This masters thesis describes the research that has been done in the CGA-package-project (Conformal Geometric Algebra Package). In this project a Conformal Geometric Algebra (CGA)-kernel framework has been built for the Computational Geometric Algorithms Library (CGAL). The most important modeling and implementation details will be discussed in detail. Specific research has been done for the use of CGA-flags in high-dimensional polytope algorithms (Convex Hull, Delaunay Triangulation and Voronoi Diagrams). The complete code and documentation of the CGA-package can be found on the internet: `http://www.cs.uu.nl/groups/MG/software`.

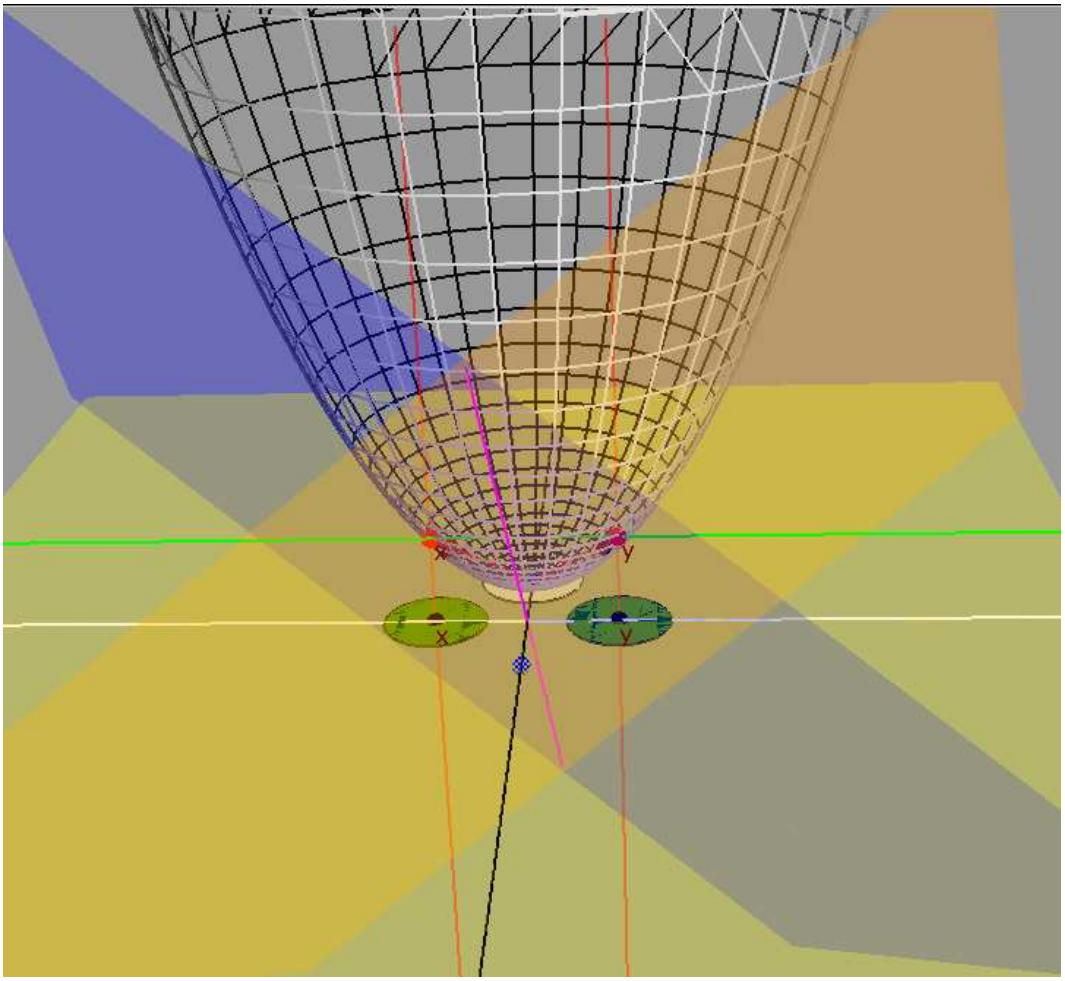


Figure 1: Illustration Edelsbrunner, section 4.3.2

Preface

The default geometric model in Computational Geometry has been Linear Algebra for a long time. The advantage of the linear algebra model is that it is time and space efficient. Nowadays, time and memory space are becoming less important in Computational Geometry, because memory space and processor speed are still increasing rapidly. So, in future geometric models with other advantages for the programmer will become attractive in Computational Geometry as well.

This thesis deals with a relatively new geometric model: Conformal Geometric Algebra (CGA). In CGA the geometrical and mathematical properties are separated, so the programmer does not have to bother about the mathematics that are behind his geometric calculations. CGA would be a very useful model in the implementation of 3D-graphical engines. However, at this moment, few programming toolkits/libraries for this geometric model are available. Therefore this project was initiated: to develop a CGA-library that implements the Computational Geometry-algorithms. The library CGAL was chosen as the foundation of this package, since it is the most-used library for Computational Geometry-research and since it has nice extendability properties. The result of this project is the Conformal Geometric Algebra Package (CGAP).

This thesis describes the different ingredients of CGAP and discusses the most important design and implementation issues of the development. The reader is assumed to be a Computational Geometry-scientist or Computational Geometry-programmer. Knowledge of advanced C++-programming and basic CGAL is required. Besides that some knowledge of the Linear Algebra and Geometric Algebra is also useful. To keep things understandable for reader that are only familiar with Linear Algebra, parallels with Euclidic geometry will be given in several sections. Where further knowledge is needed, literature references will be given.

Extra research for this project has been done on CGA-flags. CGA-flags provide a useful technique to work with high-dimensional polytopes in several algorithms. In this thesis we will show how flags can be used in an algorithm that generates dD-voronoi diagrams.

Several abbreviations will be used in this thesis, like GA, CGA, CGAL and CGAP. The reader should be aware of the fact that those terms look much alike, but do not mean the same things. Therefore a list of most used abbreviations will be given in this thesis.

I want to thank dr. Leo Dorst and drs. Daniël Fontijne of the University of Amsterdam for their advice at the beginning of this project. Besides that, I want to mention Marius D. Zaharia, who has been a pioneer in the research field of CGA-flags. Last but certainly not least, I want to thank

dr. Remco Veltkamp, my thesis supervisor, for his useful input, support, patience and coaching in this master thesis project.

June 2007, Chaïm Zonnenberg

Contents

Notations used in this thesis:

Vectors: $\mathbf{a}, \mathbf{b}, \mathbf{c}$
Unit vectors, CGA-base elements: $\mathbf{e}_1, \mathbf{e}_2$.
Multivectors: A, B, C, D
Set of points: $P_1, P_2, \ldots P_n$.
Points in CGA P, Q, R, S...,
d-dimensional space: $\mathbb{R}^d$
Dual: M^*
Reverse: $M^\dagger$
d-blade of A: $\langle A \rangle_d$
Geometric product, Clifford product: AB
Inner product: $A \cdot B$
Outer product (join): $A \wedge B$
Left contraction: $A \rfloor B$
Right contraction: $A \lfloor B$
Pseudoscalar: I_d
Meet: $\vee$

Most used abbreviations in this thesis

- **CGA** Conformal Geometric Algebra
- **CGAL** Computational Geometric Algorithms Library
- **CGAP** Conformal Geometric Algebra Package
- **GA** Geometric Algebra
- **HGA** Homogeneous Geometric Algebra
- **LA** Linear Algebra

CHAPTER 1

Introduction and Problem Definition

Computer graphics has a lot of applications in 21^{st} century everyday life. In hospitals medical imaging software is used to generate 3D-images from CT-scans and MRI-scans. In the -almost real- world of computer games the gamer has access to real time virtual 3D-environments. Architects use AutoCAD-software to design houses, cars, bridges and microchips. Car navigation systems use geographical information systems to perform their calculations.

All computer applications mentioned above belong to the field of Computer Graphics. Those Computer Graphics applications use software that performs geometric calculations. Geometric models, datastructures and algorithms are used to get the desired results. The subfield of Computer Graphics that researches this is called 'Computational Geometry'.

Within a computational geometry application we can distinguish several levels:

- **applications** The program that interacts with the user and external digital equipment.
- **algorithms and datastructures** The underlying routines and models in the application that make the program work.
- **geometric operations** The geometric operations that are needed by the algorithms. We will give two examples: 1) calculating the intersection of a line and a plane, 2) constructing the circle through the points p, q and r.
- **mathematical models, representations and calculations** The mathematical methods that are used to calculate the desired results of the geometric operations. For example the (mathematical) determinant of three vectors.

Within the field of computational geometry several programming libraries are available to perform geometric operations and underlying mathematical calculations. Such a library is a collection of algorithms, datastructures, geometric operations and mathematical operations, sometimes completed with example applications (demos). The most used C++ Computational Geometry-library is called CGAL (Computational Geometry Algorithms Library). CGAL has been developed by researchers of several universities over the whole world. It is still under development. Many

Computational Geometry-algorithms and -datastructures have been included. CGAL has very useful extendibility properties at all levels mentioned above. The default geometric models that CGAL uses, are Cartesian and homogeneous coordinate systems in combination with standard linear algebra.

A relatively new geometric model is called Conformal Geometric Algebra (CGA). CGA has very useful properties for performing geometric operations, since it separates the mathematical and geometrical perspective of the geometric modeling for the programmer. CGA also has very powerful short formula expressions and is dimension independent. However, nowadays very few libraries that support CGA are available, since linear algebra is the leading model in computational geometry. Therefore the idea of our project arose: designing a CGA-kernel that is suitable for the CGAL-kernel. This project got the name Conformal Geometric Algebra Package (CGAP). In CGAP the advantages of both CGAL and CGA are combined.

Algorithms that work on d-dimensional polytopes and polyhedra are a very good example of the use of CGA within Computational Geometry. So-called 'flags' are useful structures to work with d-dimensional polytopes and infinite polyhedra. Therefore flags were used to show the strengths of CGA with it.

1.1 Problem Definition, scope of the project

In this master's thesis project we have been looking at all different levels of Computer Graphics: Applications, Algorithms, Datastructures, Primitive operations on datastructures and underlying mathematical models/representations. Two concepts are important in this project: modeling and programming.

The goals of this project are:

- The design of a CGA-kernel according to the CGAL design principles.
- The implementation of the most important parts of such a kernel and showing it works in practice. The project has got the name 'Conformal Geometric Algebra Package'.
- As an application and extra research topic: the research how CGA-represented flags can be used in d-dimensional polytope algorithms (Voronoi diagrams and Delaunay triangulations).

The goal of this master's thesis project is not to implement the complete kernel. Many scientists have been working a long time to implement the default CGAL-kernels `Cartesian` and `Homogeneous`. So, implementing a complete CGA-kernel would be a job of many years. The scope of the implementation of this project is therefore limited to the implementation of the kernel components needed to generate the polytope algorithms as mentioned. In this way the combination of strengths of CGAL and CGA becomes visible. However, the kernel model has to have a generic structure, such that new algorithms, objects and functions can be easily integrated.

1.2 Thesis overview

The results of the research part of the project will be given in this thesis and the CGAP-library itself.

In the first chapters (chapters 2-4) we discuss the different building blocks of the package. In those chapters, we limit ourselves to giving the necessary information that is relevant for this project. For more information on a certain topic we will give literature references. A bibliography in the back of this thesis is also given. In the chapters 5-7 we will discuss the details of our own research.

- **Chapter 2 Geometric Algebras**. Here we discuss the geometrical model we are going to use: Conformal Geometric Algebra. Different CGA-objects will be shown and the mathematical model will be introduced. We will also make a comparison with conventional standard vector algebra, to get an impression of the strengths of CGA.
- **Chapter 3. CGAL**. In this chapter we will give the important details of the CGAL-library. The design principles will be given. Different objects and functions of CGAL will be shown.
- **Chapter 4. Voronoi Diagrams, Delaunay Triangulations and Convex Hulls**. We will give a description of the datastructures and algorithms of those polytope algorithms, since the extra research topic of this project has to do with polytope algorithms.
- **Chapter 5. Flags**. Flags are an extension of GA-elements, that can store a whole related sequence of geometrical objects. Flags can be easily used as the building blocks of multi-dimensional polyhedra, for example with the construction of Voronoi Diagrams. All relevant details of flags will be given.
- **Chapter 6. CGAP** In chapter 6 we give a detailed description of the modeling process of the CGA-kernel in CGAL according to the CGAL-design principles. Important design decisions are described. The mappings of CGA-objects on CGAL-objects will be given.
- **Chapter 7. Example application** In chapter 7 everything comes together. We give an application of a dD-Voronoi-algorithm that uses the CGA-kernel in CGAL. This uses the flags-structure to enumerate the edges.
- **Chapter 8. Summary and conclusion** At the end in chapter 8 a short summary and a conclusion is given.

1.3 Project references

The source code of this project can be downloaded from
`http://www.cs.uu.nl/groups/MG/software`. The user& reference manual is also available on this website.

CHAPTER 2

Geometric Algebras

Geometric modeling is the mathematical and geometric foundation of Computer Graphics. A geometric model makes it possible to compute with geometric objects, like points, planes and lines. Several geometric models have been developed, for example: the vector linear algebra with cartesian coordinates and vector algebra with homogeneous coordinates.

Till this day, most modeling is done with vector linear algebra with cartesian coordinates. In every book on linear algebra for beginners this model is explained. Within this model, points are represented by vectors; lines and planes by linear algebra formulas. For translations and rotations we need vector additions and matrix multiplications. Projections are also done by formulas. This model is efficient for the use in computer graphics, because it uses relatively few memory space and processor time.

In this chapter some basic concepts of the Conformal Geometric Algebra-model (CGA-model) are treated. For comparison and familiarity we start with the disadvantages of the Linear Algebra-model. Then we discuss some parts of the mathematical backgrounds of CGA for basic understanding. A definition of the CGA will be given, basic elements and operations are discussed. At the end of this chapter we will give some applications of CGA and discuss the two software packages that handle CGA.

Not all details will be given. This chapter gives a basic introduction to CGA and the concepts that have to be known to understand the CGAP-library, are discussed. For further reading, some references to books and articles are given.

2.1 Disadvantages of Linear Algebra

The *'vector linear algebra with cartesian coordinates'*-model is the most used geometric model nowadays. However, it has some disadvantages:

The first disadvantage of the vector linear algebra model is that it is not always intuitively clear why it works. For example, it is not directly clear why a 3×3-matrix with sine and cosine-functions can represent a rotation in 3D. For people who are accustomed to linear algebra, this might

sound unfamiliar, but the following example looks very complex for beginners in the field of linear algebra.

Example 1 ***Rotation example***

3D-vector linear algebra*: Let α, β and γ denote the rotation angles around resp. the x-, y- and z-axis. Then the rotation matrix equals:*

$$\begin{pmatrix} \cos\gamma\cos\beta & -\sin\gamma\cos\beta & \sin\beta \\ \cos\gamma\sin\beta\sin\alpha+\sin\gamma\cos\alpha & \cos\gamma\cos\alpha-\sin\gamma\sin\beta\sin\alpha & -\cos\beta\sin\alpha \\ \sin\gamma\sin\alpha-\cos\gamma\sin\beta\cos\alpha & \sin\gamma\sin\beta\cos\alpha+\sin\alpha\cos\gamma & \cos\beta\cos\alpha \end{pmatrix}$$

.

Another problem with vector linear algebra is that it is inconsistent in the way it deals with geometric operations. The translation from the geometric operation to the mathematical calculation is not consistent for different geometrical objects. For example: Determining the intersection of a line and a plane in $\mathbb{R}^3$ is done in a very different way, compared to determining the intersections of two non-parallel planes in $\mathbb{R}^3$. In paragraph 2.6 more examples of this kind will be given.

The third difficulty with vector linear algebra is that it is hard to program in. Calculations take many lines of code and bugs are easily introduced. For example, when a minus-sign and a plus-sign are interchanged. Testing and debugging a linear algebra kernel therefore is a difficult and notorious task.

2.2 CGA

In this chapter we introduce another geometric model: the Conformal Geometric Algebra (CGA). This model has some advantages when compared to vector linear algebra. In this model all geometric objects (points, planes, lines, circles) are stored as elements of the same algebra (CGA-elements). Most of the operators have the same meaning for different geometric objects. For example: the meet ($\vee$-operator) does all operations that deal with unions. Rotations and translations are also given by CGA-elements, so a rotation or translation can be performed by a simple CGA-multiplication.

A CGA-model is dimension-independent: Geometrical objects and operations have the same definition and representation in all dimensions. This means, for example, that the definition of a circle in 2D is the same as the definition in 5D. Also operations, like rotations, reflections and joins have the same formulas in all dimensions. When a certain function works in a certain dimension (for example 2D), it can be easily generalized to a higher dimension.

Example 2 ***The CGA-version of rotation example 1)***

CGA*: Let l be a rotation axis in 3D through the origin. This axis defines a Euclidean bivector: $B := l^*$. The rotor performing the rotation is simply:*

$$R = e^{-B\phi/2} = \cos(\phi/2) - B\sin(\phi/2)$$

This rotor R rotates an arbitrary object X with the formula RX/R, i.e. RXR^{-1}, where R^{-1} equals the reverse of R, notation: $R^\dagger$. [1]

In the example we see the advantages of the CGA-model for rotation operations:

[1] This example can be found in [7], p. 29.

- The CGA-formula generalizes to dD with the same short formula. In vector linear algebra, we should expand the matrix to size $d \times d$.
- The CGA-rotation-formula works on arbitrary objects (like planes, lines, spheres, circles etc.). In vector linear algebra only vectors can be rotated in this way.

So the CGA is intuitively more clear and it has a more consistent structure than vector linear algebra.

The CGA-model uses more memory space and most calculations take some more time (both up till a constant factor). But, nowadays efficient handling of memory space and processor time is becoming less important, because processors are getting faster and more memory space is becoming available for a low price. Due to those developments CGA becomes more interesting and usable in computational geometry. Several applications can be found in robotics, computer vision, aerospace technology, neural computing and biomechanics.

2.3 Mathematical backgrounds of CGA

There are two ways to describe the concept 'Geometric Algebra'. It can be viewed from a mathematical perspective and from a geometric perspective. The first deals with the numerical calculations, the second with the geometric objects and geometric operations. A big advantage of GA's is that those two perspectives can be easily separated from each other. For a programmer it is not necessary to know all details of the mathematics behind geometric algebra's. He does not have to think in coordinates and difficult matrices to do his calculations. He just has to know the two important product functions (meet and join) and some things about the geometric interpretations of the objects. This way of splitting the mathematical and geometrical perspectives is called a coordinate free approach.

In the following subparagraphs a mathematical definition of a Conformal Geometric Algebra will be built up. Readers that are unfamiliar with GA or CGA, might read the next section on the geometric perspective first to get a basic understanding of CGA.

The mathematical definition of CGA will be given in four steps. First a basic algebra is defined. Then we extend it to a so called 'Clifford algebra'. The Clifford Algebra is the parent of the 'Geometric Algebra'. The 'Conformal Geometric Algebra' is a special form of the geometric algebra. So every algebra is a specialaziation of its parent algebra.

2.3.1 Basic Algebra

An algebra in general:

- consists of one or more sets (like $\mathbb{R}$ and $\mathbb{R}^3$)
- that are closed under one or more operations (like $+$ and $\cdot$)
- and satisfies basic axioms. (like commutativity and distributivity)

So an algebra can be defined by specifying 1) the set(s) it operates on, 2) the operators it calculates with and 3) the axioms that yield for it. A well-known algebra is the linear algebra.

Definition 1 *A linear algebra is a vector space V combined with an addition operator and a multiplication operator, such that for each vector $\mathbf{a}, \mathbf{b}, \mathbf{c} \in V$ and $r, s \in \mathbb{R}$:* [2]

[2] This definition is given by [13], p. 9.

$(\mathbf{a}+\mathbf{b})+\mathbf{c}=\mathbf{a}+(\mathbf{b}+\mathbf{c})$	*associativity*
$\mathbf{a}+\mathbf{b}=\mathbf{b}+\mathbf{a}$	*commutativity*
$\mathbf{0}+\mathbf{a}=\mathbf{a}$	$\mathbf{0}$ *is the 'additative identity'*
$\mathbf{a}+(-\mathbf{a})=\mathbf{0}$	$-\mathbf{a}$ *additive inverse of* $\mathbf{a}$
$r(\mathbf{a}+\mathbf{b})=r\mathbf{a}+r\mathbf{b}$	*distributivity*
$(r+s)\mathbf{a}=r\mathbf{a}+s\mathbf{a}$	*distributivity*
$r(s\mathbf{a})=(rs)\mathbf{a}$	*associativity*
$1\mathbf{a}=\mathbf{a}$	*the scalar value 1 is the 'multiplicative identity'*

Example 3 *The vector space $\mathbb{R}^2$ with corresponding '+'-operator and scalar-multiplication-operator is an example of a linear algebra.*

Some important terms of basic algebra are: vector, (in)dependence, matrix, subspace, span, dimension, basis. The definition and meaning of those terms can be found in any book that gives an introduction on linear algebra, for example [13] and [20].

2.3.2 Clifford algebra

The Clifford algebra was introduced by William K. Clifford (1845-1879). This algebra has a strong unifying aspect, because it views different algebraic systems, that are related to geometry, as specializations of one 'generic algebra'.

A Clifford algebra C on $\mathbb{R}^d$ uses the default vector base components as generator elements: $\mathbf{e}_1$, $\mathbf{e}_2$, ..., $\mathbf{e}_d$. It is characterized by the so-called quadratic form function $Q : V \to \mathbb{R}$. To generate the algebra, it uses the special Clifford product as multiplication operator. This operator is defined for all elements in C.

Definition 2 *The Clifford product is defined by the following properties:*

- *Signature in quadratic form: for every i: $Q(\mathbf{e}_i)=\mathbf{e}_i\mathbf{e}_i=s_i$, where $s_i \in \{-1,0,1\}$.*
- *Anti-commutativity: $\mathbf{e}_i\mathbf{e}_j=-\mathbf{e}_j\mathbf{e}_i$ when $i \neq j$.*

The several s_i are constants, that can be positive (+1), negative (-1) or zero (0). The set of all s_i form the characteristic of the algebra. This is called the *signature* of the Clifford algebra. An example signature of a Clifford algebra is $\{+1,+1,-1\}$

Example 4 *The Clifford product with algebraic signature $\{+1,+1,-1\}$ generates the following independent elements: $\{1,\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3,\mathbf{e}_1\mathbf{e}_2,\mathbf{e}_1\mathbf{e}_3,\mathbf{e}_2\mathbf{e}_3,\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3\}$. The algebra consists of elements of the form: $\alpha_1+\alpha_2\mathbf{e}_1+\alpha_3\mathbf{e}_2+\alpha_4\mathbf{e}_3+\alpha_5\mathbf{e}_1\mathbf{e}_2+\alpha_6\mathbf{e}_1\mathbf{e}_3+\alpha_7\mathbf{e}_2\mathbf{e}_3+\alpha_8\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3$ for $\alpha_i \in \mathbb{R}$. It has 8 degrees of freedom. So this algebra has dimension 8.*

The dimension of a Clifford algebra can be calculated: a Clifford algebra with base elements $\mathbf{e}_1,\ldots,\mathbf{e}_d$ can generate 2^d independent elements, because every $i \in \{1\ldots d\}$ can be included/not included in the element.

One needs some experience to manipulate Clifford algebra expressions. For more information on Clifford algebras, see [22] and [29].

2.3.3 Geometric Algebra

From a mathematical point of view a Geometric Algebra is a specialization of a Clifford algebra. The Clifford algebra with signature S is extended with additional product functions and other operators. From a geometrical point of view a Geometric Algebra is a Clifford Algebra with a geometric interpretation.

In section 2.4.2 the special 'GA-products' are explained. Every product can be defined in terms of the Clifford product and the algebraic addition operator ($+$). Besides that, it has its own geometric meaning. (See section 2.3.4)

The most used GA-products are:

- the geometric product (identical to the Clifford product in Clifford algebra)
- $\wedge$, outer product
- $\cdot$, inner product
- $\rfloor$, $\lfloor$, resp. left and right contraction
- $\vee$, join function

The addition operator $+$ does not appear often in geometric formulas. Usually a geometric expression consists of one or more products. However, every product function can be rewritten as a function consisting of the addition operator, the dual and the geometric product.

Many algebraic mathematical concepts can be reformulated in GA-language. For example:

- **complex numbers**. $a + b\mathbf{e}_1\mathbf{e}_2 \cong a + bi$. For it yields $(\mathbf{e}_1\mathbf{e}_2)(\mathbf{e}_1\mathbf{e}_2) = -(\mathbf{e}_1\mathbf{e}_2)(\mathbf{e}_2\mathbf{e}_1) = -1$, this is equivalent to the complex multiplication $i^2 = -1$.
- **quaternions**. In the past, geometrists used so-called quaternions to represent 3D euclidean space. They introduced the constants i, j and k, that satisfied: $i^2 = j^2 = k^2 = -1$. Those constants represented base vectors and were anti-commutative: $ij = -ji$, $ik = -ki$ and $jk = -kj$. This system can be simply rewritten as a 3-dimensional GA with signature $\{-1, -1, -1\}$. Since then $\mathbf{e}_1\mathbf{e}_1 = \mathbf{e}_2\mathbf{e}_2 = \mathbf{e}_3\mathbf{e}_3 = -1$ and $\mathbf{e}_1\mathbf{e}_3 = -\mathbf{e}_3\mathbf{e}_1$ etc.
- **conformal geometric algebra's** (see next subsection)

More useful information on geometric algebra's can be found in [25], [8] and [16].

2.3.4 Conformal Geometric Algebra

The base of a d-dimensional Conformal Geometric Algebra (CGA) consists of:

- A d-dimensional GA with positive signature (i.e. all s_i equal 1)
- Two new independent vectors $\mathbf{e}_0$ and $\mathbf{e}_\infty$, that are derived from two additional base vectors: $\mathbf{e}_+$ and $\mathbf{e}_-$. The signature of those newly added base vectors is: $\mathbf{e}_+^2 = 1$ and $\mathbf{e}_-^2 = -1$

Actually, by adding those two vectors, the Euclidean space E^d is extended to the so-called Minkowski-space (see [15]) $\mathbb{R}^{d+1,1}$. Those new vectors $\mathbf{e}_0$ and $\mathbf{e}_\infty$ can be easily derived from the base vectors:

$$\mathbf{e}_0 = \frac{1}{\sqrt{2}}(\mathbf{e}_- - \mathbf{e}_+) \qquad \mathbf{e}_\infty = \frac{1}{\sqrt{2}}(\mathbf{e}_- + \mathbf{e}_+) \tag{2.1}$$

$\mathbf{e}_0$ is called the origin and corresponds to the last base vector in homogeneous vector algebra. $\mathbf{e}_0$ enables us to work projectively. It represents the origin of the subspace and therefore removes the singuarlity of the represented Euclidean space.

$\mathbf{e}_\infty$ encodes the metric of an Euclidean space (projectively represented space). For a geometrical CGA-round point this factor represents the distance from that point to the origin.

$\mathbf{e}_0$ and $\mathbf{e}_\infty$ have the following properties:

$$\mathbf{e}_0^2 = 0 \qquad \mathbf{e}_\infty^2 = 0 \qquad \mathbf{e}_\infty \cdot \mathbf{e}_0 = -1 \tag{2.2}$$

Those properties can be easily derived from equation 2.1. Note that $\cdot$ in the last equation denotes the inner product that will be defined further on in this section. In this case the following yields $\mathbf{e}_+ \cdot \mathbf{e}_- = \mathbf{e}_- \cdot \mathbf{e}_+ = 0$, $\mathbf{e}_0 \cdot \mathbf{e}_0 = \mathbf{e}_0\mathbf{e}_0$ and $\mathbf{e}_\infty \cdot \mathbf{e}_\infty = \mathbf{e}_\infty\mathbf{e}_\infty$.

The most important operator in a CGA is the wedge-operator: $\wedge$. In the notation of the base elements of this algebra this operator is also used. Examples: $\mathbf{e}_1 \wedge \mathbf{e}_3$, $\mathbf{e}_1 \wedge \mathbf{e}_\infty$ and $\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_4 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty$.

To get insight in the elements, we give two examples of what the CGA-elements look like. Normally, for a programmer it is not necessary to think in those coordinates, since CGA is a coordinate free model.

Example 5

Point:	$-0.17\mathbf{e}_1 + 0.10\mathbf{e}_2 - 1.04\mathbf{e}_3 + \mathbf{e}_0 + 0.56\mathbf{e}_\infty$.
Line:	$0.01\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_\infty - 1.83\mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty + 0.98\mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty$ $+1.58\mathbf{e}_1 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty - 0.84\mathbf{e}_2 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty - 1.04\mathbf{e}_3 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty$.

Remark: the scalars are rounded up to 2 decimals for readability

An arbitrary element in a CGA is called a *multivector*. Most geometric CGA-elements consist of the independent base elements that have the same number of base vectors. Such an element is called a *blade*. The number of base vectors in the blade is called the *grade*. In the example above: a point has grade 1 and a line has grade 3. It is also allowed to write: a point is a 1-blade and a line is a 3-blade.

For an arbitrary multivector A, the i-blade can be extracted from it. Notation: $\langle A \rangle_i$.

In a d-dimensional CGA the base element with the maximum grade is called the *pseudoscalar* I_d. It can be calculated as follows: $I_d = \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \ldots \wedge \mathbf{e}_d \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty$.

Additional mathematical information on CGA's can be found in [18] and [27]. In the next subsections we will mention the most important product functions and unary operations within the CGA.

Geometric product

The *geometric product* is equivalent to the basic Clifford multiplication in a Clifford Algebra. Every other product function can be rewritten in this geometric form. The geometric product has no symbol, like most products in other algebras. The product of vectors $\mathbf{a}$ and $\mathbf{b}$ is written as $\mathbf{ab}$.

Inner product

The inner product of vectors is much like the dot product in vector linear algebra. For vectors it can be defined in terms of the geometric product. Given vectors $\mathbf{a}$ and $\mathbf{b}$, the inner product equals: $\mathbf{a} \cdot \mathbf{b} = \frac{1}{2}(\mathbf{ab} + \mathbf{ba})$.

For more complex blades A and B with non-zero grades r and s, the inner product can be defined as: $A \cdot B = \langle AB \rangle_{|r-s|}$. [3]. For arbitrary multivectors M and N yiels: $M \cdot N := \sum_{r,s} \langle \langle M \rangle_r \langle N \rangle_s \rangle_{|r-s|}$.

The inner product has the following properties:

- Grade decreasing. Let A and B be blades with grades respectively r and s, then the grade of $A \cdot B$ is smaller than r and s.
- Symmetric on vectors. Given $\mathbf{a}$, $\mathbf{b}$ arbitrary vectors, then $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$
- Geometric meaning. From a geometric perspective, the inner product deals with orthogonality, metric and projection concepts.

Outer product

For vectors, the outer product can be defined in terms of the geometric product. Given vectors $\mathbf{a}$ and $\mathbf{b}$, the outer product equals: $\mathbf{a} \wedge \mathbf{b} = \frac{1}{2}(\mathbf{ab} - \mathbf{ba})$.

For more blades A and B with non-zero grades r and s, the outer product can be defined as: $A \wedge B = \langle AB \rangle_{r+s}$. [4] This can be generalized to multivectors M and N: $M \wedge N = \sum_{r,s} \langle \langle M \rangle_r \langle N \rangle_s \rangle_{r+s}$.

Definition 3 *Let S_n be the collection of permutations of the numbers $1, 2, \ldots, n$. The outer product of the vectors $\mathbf{a}_1, \ldots, \mathbf{a}_n$ is defined as follows: $\mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \ldots \wedge \mathbf{a}_n := \frac{1}{n!} \sum_{p \in S_n} f(p)$ where $f(p)$ is the term that belongs to permutation p. Let $p = (i_1, \ldots, i_n)$, then this term equals $s\mathbf{a}_{i_1}\mathbf{a}_{i_2} \ldots \mathbf{a}_{i_n}$ where s is the sign of the term, i.e. $s \in \{-1, +1\}$. If p is a positive permutation, s equals 1, otherwise -1.*

Example 6 *Let $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ be three vectors. The outer product $\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}$ equals* $\frac{1}{6}(\mathbf{abc} - \mathbf{acb} + \mathbf{bca} - \mathbf{bac} + \mathbf{cab} - \mathbf{cba})$

The outer product has the following properties:

- Anti-commutativity, for multivectors A and B: $A \wedge B = -B \wedge A$
- Distributivity: $(A + B) \wedge C = (A \wedge C) + (B \wedge C)$
- Grade increasing. Let A and B be blades with grades respectively r and s, then the grade of $A \wedge B$ is bigger than r and s.
- $\mathbf{e}_i \wedge \mathbf{e}_i = 0$. This also yields larger elements, for example: $(\mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_4) \wedge \mathbf{e}_3 = -\mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_3 \wedge \mathbf{e}_4 = 0$

The *outer product* is often used to denote base elements in GA. This in contrast with the Clifford algebra, where the Clifford product is used. From a mathematical perspective this is not a difference, since always yields: $\mathbf{e}_i\mathbf{e}_j = \mathbf{e}_i \wedge \mathbf{e}_j$ for base elements $\mathbf{e}_i$ and $\mathbf{e}_j$ ($i \neq j$).

For vectors the following correspondence of the three products yields: $A = \mathbf{bc} = \mathbf{b} \cdot \mathbf{c} + \mathbf{b} \wedge \mathbf{c}$.

[3] See [25], p. 6.
[4] See [25], p. 6.

Dual and reverse

There exists a duality relation on elements of a CGA. Given a blade B with grade r in a d-dimensional CGA. Then the dual has grade $d-r$. It uses the following symbol: B^*.

Definition 4 *Let B be a multivector. Then the dual B^* is defined as: $B^* := BI_n^{-1}$.*

The inner and outer products are dual to one another:

$$(\mathbf{a} \cdot B)I_n = \mathbf{a} \wedge (BI_n),$$
$$(\mathbf{a} \wedge B)I_n = \mathbf{a} \cdot (BI_n).$$

Another unary operator in CGA is the *reverse operator*. The difference between the reverse and the dual, is that the reverse is grade preserving, whereas duality goes to complementary grade. The reverse is denoted as $B^\dagger$.

Definition 5 *Let B be a multivector. Then the reverse $B^\dagger$ is defined as: $\left\langle B^\dagger \right\rangle_i := (-1)^{\frac{i(i-1)}{2}} \langle B \rangle_i$*

Other products

The CGA also supports other products of interest. In this section we give the mathematical definition. The geometrical meaning will be given in section 2.4.2

Definition 6 *Let A and B be multivectors. The meet $A \vee B$ is defined as: $A \vee B := A^* \cdot B$.*

Definition 7 *The left contraction $\rfloor$can be defined for two multivectors M and N as $M \rfloor N := \sum_{r,s} \langle \langle M \rangle_r \langle N \rangle_s \rangle_{s-r}$.*[5]

The left contraction is a variant of the inner product. Note the slight difference with the standard inner product that has definition $M \cdot N = \sum_{r,s} \langle \langle M \rangle_r \langle N \rangle_s \rangle_{|r-s|}$. The left contraction is used in some cases where the standard inner product is insufficient.

The right contraction works the other way around as the left contraction: $M \lfloor N := N \rfloor M$. It has formula: $M \lfloor N := \sum_{r,s} \langle \langle M \rangle_r \langle N \rangle_s \rangle_{r-s}$

Additional information on mathematical backgrounds of CGA's can be found in [18] and [27].

2.4 Geometrical concepts of CGA

In the previous section we have seen some mathematical backgrounds; now we will look to the CGA from another perspective: geometry.

Basic geometric elements: The basic elements of a CGA are geometric objects like a line, sphere, point pair, hyperplane, circle, etc. All geometric objects are blades. Generally, there exist 4 types of GA-elements[6]:

1. rounds (points, circles, spheres, point pairs).
2. flats (planes, lines) A flat is a round containing the point at infinity $\mathbf{e}_\infty$ in its formula.

[5] The definition originates from [26], p. 1.
[6] See also [7], p.8-15.

3. free blades (free vectors). Those are elements without position. Those blades have no position, that means there is no $\mathbf{e}_0$-component in its formula.

4. tangent blades (for example tangent vector). Those blades do not have an $\mathbf{e}_\infty$ component.

In the next paragraphs the most important geometric objects will be treated one-by-one in detail.

Basic elements: In CGA there are two important elements: $\mathbf{e}_0$ (sometimes denoted as origin) and $\mathbf{e}_\infty$ (infinity). The pseudoscalar I_d also plays an important role (see section 2.3.4).

Operators: The CGA has a lot of geometric product operations like 'meet' for the intersection of two objects, 'join' for the common object that contains two objects, 'left contraction' for projections. In mathematical notation all those operators can be expressed as a formula consisting of the basic product function (binary operator), the dual (unary operator) and the addition operator. Note that the use of the '+'-operator is avoided in CGA; it rarely occurs in geometric expressions.

One of the strengths of CGA is its expression power. Formulas are mostly short and consistent for different geometric objects. This will become clear from the examples in the next paragraphs.

2.4.1 Objects in CGA

Point

In CGA a point can be represented as a round or a flat. The round point is a dual sphere with radius 0. The round point is a blade with grade 1 and the flat point is a blade with grade 2. The CGA-point can be defined from vector coordinates.

Definition 8 *Let p be a Euclidean point in $\mathbb{R}^d$. In Euclidean vector coordinates this can be rewritten as $p = p_1\mathbf{e}_1 + p_2\mathbf{e}_2 + \ldots + p_d\mathbf{e}_d$. The round point R corresponding with p can be defined as $R = p + \mathbf{e}_0 + \frac{1}{2}\|p\|^2\mathbf{e}_\infty$. The flat point F corresponding with p can be defined as $F = (p + \mathbf{e}_0) \wedge \mathbf{e}_\infty$.*

Example 7 *Let $p \in \mathbb{R}^3$ represent a Euclidean 3-dimensional point. Let $p = (a, b, c)$, $a, b, c \in \mathbb{R}^3$.*

The round-point representation R is: $R = a\mathbf{e}_1 + b\mathbf{e}_2 + c\mathbf{e}_3 + \mathbf{e}_0 + z\mathbf{e}_\infty$, where $z = \frac{1}{2}(a^2 + b^2 + c^2)$.

The flat-point representation F is: $F = a\mathbf{e}_1 \wedge \mathbf{e}_\infty + b\mathbf{e}_2 \wedge \mathbf{e}_\infty + c\mathbf{e}_3 \wedge \mathbf{e}_\infty + \mathbf{e}_0 \wedge \mathbf{e}_\infty$

Round points and flat points can be multiplied with an arbitrary scalar factor ($\neq 0$). It then still represents the same point.

There exists a correspondence between a flat point F and a round point R. To convert from round to flat: $F = R \wedge \mathbf{e}_\infty$. To convert from flat to round: $R = ((F\mathbf{e}_0F) \wedge \mathbf{e}_0)\mathbf{e}_\infty((F\mathbf{e}_0F) \wedge \mathbf{e}_0))$

The round point is used more often in geometric expressions than the flat point, since it has nice perpendicularity properties. For example, given the round points P, Q and R, it can be used for calculating the CGA-object that is perpendicular to those three points: $P \wedge Q \wedge R$. This is the circle passing through P, Q and R.

Normalization Round points can be multiplied by scalar factor ($\neq 0$) and then represent the same point. Sometimes a 'unique' or default representation is required for calculations. Therefore the point is normalized by the formula $P' := \frac{P}{-\mathbf{e}_\infty \cdot P}$. This sets the $\mathbf{e}_0$-factor of the point to 1.

Point distance The distance between two points can be calculated by the following formula:

$$-\frac{1}{2}d(P, Q)^2 := \frac{P}{-\mathbf{e}_\infty \cdot P} \cdot \frac{Q}{-\mathbf{e}_\infty \cdot Q} \tag{2.3}$$

When the points are normalized, the distance equals $\sqrt{2P \cdot Q}$.

Line

A line in CGA is represented by a 3 dimensional blade. The internal representation corresponds with the so-called Plücker coordinates.

Definition 9 *Let P and Q be two points in CGA. The line L through P and Q is defined by the formula: $L := P \wedge Q \wedge \mathbf{e}_\infty$.*

Let $\mathbf{p}$ and $\mathbf{q}$ be two points in Euclidean space $\mathbb{R}^d$. The line through $\mathbf{p}$ and $\mathbf{q}$ can be represented by the Plücker coordinates $\mathbf{u} := \mathbf{p} - \mathbf{q}$ and $v := \mathbf{p} \times \mathbf{q}$. The vector $\mathbf{u}$ represents the direction of the line. The vector v is called the moment and represents the shortest cross product vector from the origin to the line.

The CGA-line is represented by the formula: $\mathbf{u} \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty + v \wedge \mathbf{e}_\infty$.

Example 8 *(Correspondence CGA-line and Plücker coordinates)*
Let $p, q \in \mathbb{R}^3$. Then $u = p - q$ and $v = p \times q$ are the Plücker coordinates that represent the line. Note that v can be split as $v_1\mathbf{e}_2\mathbf{e}_3 + v_2\mathbf{e}_1\mathbf{e}_3 + v_3\mathbf{e}_1\mathbf{e}_2$.

The CGA-line through p and q has coordinate representation:
$L := u_1\mathbf{e}_1 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty + u_2\mathbf{e}_2 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty + u_3\mathbf{e}_3 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty + v_1\mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty + v_2\mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty + v_3\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_\infty$

Circle and Sphere

Circles, spheres and hyperspheres are rounds. Circles are blades with grade 3, spheres have grade 4.

Definition 10 *Let P, Q, R be round points in CGA. Then the circle C through those points is defined as $C := P \wedge Q \wedge R$.*

Definition 11 *Let P, Q, R, S be round points in CGA. Then the sphere B through those points is defined as $B := P \wedge Q \wedge R \wedge S$.*

Of course this definition can be generalized for hyperspheres.

The center of a circle can be found by reflecting e_∞ in the circle: $Ce_\infty C$. (See also section 2.4.3) The center of the sphere can be found by reflecting e_∞ in the sphere : $De_\infty D$. This center-formula also yields for higher dimensional hyperspheres.

In the 3-dimensional Euclidean space the dual of a sphere D encodes the center point M and the radius ρ: $D^* = M - \frac{1}{2}\rho^2 e_\infty$. This dual can be multiplied by a non-zero scalar factor. When the radius ρ equals zero, $D^* = M$: the dual represents a round point. This also yields for other dimensions, like circles (2D) and hyperspheres ($d \geq 4$).

Point pair / Line segment

In CGA a point pair and a line segment are represented by the same CGA-element. It is a round blade with grade 2.

Definition 12 *Let P and Q be round points, then $B := P \wedge Q$ is the corresponding point pair.*

A 'point pair' may seem a strange object. The point pair actually is a one-dimensional sphere [7]. A point pair has a center from which the two points on the 1D-sphere have the same distance. Compare this with a circle (i.e. a two-dimensional sphere) and a sphere (i.e. three-dimensional sphere).

[7] See also [24], p. 60.

Plane and hyperplane

Planes and hyperplanes are 'flat' CGA-elements. A 3D-plane is a blade with grade 4. A dD-hyperplane has grade $d+1$.

Definition 13 *Let P, Q and R be round points in CGA. The 3D-plane A has the formula* $A = P \wedge Q \wedge R \wedge \mathbf{e}_\infty$.

The plane with opposite orientation is $-P$. The dual of the plane P has the form: $P^* = \mathbf{n} + d\mathbf{e}_\infty$, where $\mathbf{n}$ is the 3D-tangent vector to the plane with generic formula $\mathbf{n} = a\mathbf{e}_1 + b\mathbf{e}_2 + c\mathbf{e}_3$, and where d is the distance from the plane to the origin point $\mathbf{e}_0$.

In an Euclidean space of arbitrary dimension d, the dD-hyperplane can be defined by d points called $p_1, p_2, \ldots, p_d$ by the formula: $p_1 \wedge p_2 \wedge \ldots \wedge p_d \wedge \mathbf{e}_\infty$.

Vectors

With the word 'vector' in geometry we can mean different things. In vector linear algebra those geometric concepts can be modeled by the same coordinate representation. For example the notation $(5, 3, 2)$ can denote a point, a position vector, a normal vector and a tangent vector. CGA is more closely related to geometry and therefore, those different vector types have different CGA-representations.

In CGA we can distinguish four types of vectors: normal vectors, free vectors (direction vectors), tangent vectors and position vectors. In the table below, the different vector blades are represented. a, b, c, d, e and f denote scalar values in $\mathbb{R}$.

vector type	Formula in 2D Euclidean	Grade
normal vector	$a\mathbf{e}_1 + b\mathbf{e}_2$	1
free vector	$a\mathbf{e}_1 \wedge \mathbf{e}_\infty + b\mathbf{e}_2 \wedge \mathbf{e}_\infty$	2
tangent vector	$a\mathbf{e}_1 \wedge \mathbf{e}_2 + b\mathbf{e}_1 \wedge \mathbf{e}_0 + c\mathbf{e}_2 \wedge \mathbf{e}_0 + d\mathbf{e}_1 \wedge \mathbf{e}_\infty + e\mathbf{e}_2 \wedge \mathbf{e}_\infty + f\mathbf{e}_0 \wedge \mathbf{e}_\infty$	2
position vector	$a\mathbf{e}_1 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty + b\mathbf{e}_2 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty$	3

The normal vector n is the the dual of a flat. It means the vector that is perpendicular to a (hyper)plane. For example, in $\mathbb{R}^3$, it means the dual of a plane.

The free vector is also called a direction vector. A free vector does not have a position. Given the normal vector n, it can be calculated as follows: $n \wedge \mathbf{e}_\infty$.

The tangent vector t has a position and a direction. It can be seen as the formula $\mathbf{e}_0 \wedge \mathbf{n}$, and then translated to the desired location.

The position vector p is a line element from $\mathbf{e}_0$ to pt(p). This corresponds to $\mathbf{e}_0 \wedge n \wedge \mathbf{e}_\infty$, the line through the corresponding point of p and the origin. This vector is freely shiftable along that line.

2.4.2 Operations

In CGA several basic operations are available. In this chapter we discuss the most important operators. Those are also used in the CGAP package.

Intersection

Given two geometrical objects A and B, the intersection C of those objects can be written as $C := A \cap B$ in set notation. When A and B are represented as CGA-blades, the intersection can be easily calculated by the meet-operation: $C := A \vee B$. Note that this is defined as $B^* \cdot A$. The

meet operation also satisfies the 'deMorgan rule' (see [25], p.8): $(A \vee B)^* = A^* \wedge B^*$, under the condition that the join is non-zero.[8]

Example 9 *Let V_1 and V_2 be planes. Then $L := V_1 \vee V_2$ defines the intersection line. When V_1 and V_2 are parallel, L equals 0.*

Example 10 *Let C be the a circle and V be a plane, both in CGA-representation. Then $C \vee V$ defines the point pair that represents the intersection.*

Union

The outer product can be used as a so-called join operator.[9] Given two objects A and B, then $A \wedge B$ represents the smallest subspace that contains A and B. In section 2.4.1 we already saw that this operator is used to construct objects from round points.

Example 11 *Let P_1, P_2, P_3 and P_4 be round points, then:*
The point pair of P_1 and P_2 equals $P_1 \wedge P_2$.
The circle through the three points equals: $P_1 \wedge P_2 \wedge P_3$.
The sphere through the four points equals: $P_1 \wedge P_2 \wedge P_3 \wedge P_4$.

Example 12 *Let C be a circle and P a round point, then $C \wedge P$ equals the sphere through C and P.*
Let L be a line and P a round point, then $L \wedge P$ equals the plane through L and P.

The element $\mathbf{e}_\infty$ is used to join an object with 'infinity':

Example 13 *Let X be a point pair, then $X \wedge \mathbf{e}_\infty$ represents the (infinite) line through the points of X.*
Let C be a circle, then $C \wedge \mathbf{e}_\infty$ represents the plane in which the circle lies.

2.4.3 Rotation and Translation

Translations and rotations can be represented by so called versors. Versors are a special kind of multivectors, that are factorisable to a geometric product of invertable vectors. Versors operate on geometric objects through the product function. Let V be a versor and X a geometric object in CGA, then the translated (or rotated) object X' can be calculated by the following sandwiching formula $X' := VX/V$. The division by V is defined as multiplication with the inverse V^{-1}, so the formula can be rewritten as $X' := VXV^{-1}$.

Definition 14 *Let $\mathbf{v} \wedge \mathbf{e}_\infty$ denote a free vector representing the translation over $\mathbf{v}$. The versor T translating an object over $\mathbf{v}$ equals:*
$T := 1 - \frac{1}{2}\mathbf{v}\mathbf{e}_\infty$.

Definition 15 *Let L define a rotation axis and let ϕ be a rotation angle, then the versor R rotating the object over the axis L is defined by:*
$R := \cos(\frac{1}{2}\phi) - L^* \sin(\frac{1}{2}\phi)$.

[8] This definition of the meet comes from [25], p.8. In this thesis we will use this definition. Others define the meet in a more generalized way and call this case 'in general position'.

[9] In this thesis we will use this definition. Others define the join in a more generalized way and call this case 'in general position'.

Versors can be multiplicated with each other into a new versor. In this way general rigid body motions can be defined in which rotations and translations have been combined.

Example 14 *Let T denote a translation versor and R a rotation versor. Then TR defines the combined translation-rotation-versor that first rotates an object and then translates it.*

When compared to linear algebra this way of rotating and translating elements is much more efficient, intuitive and elegant than vector linear algebra. Simple intuitive formulas can be used in a CGA instead of large complicated error-sensitive matrix-multiplications (linear algebra). Those CGA-formulas are intuitively clear, independent of dimension and independent of geometric object.

Containment and Perpendicularity

The algebraic expressions of CGA can also be used to perform tests on geometric objects. Two common tests are the containment test and the perpendicularity test[10].

- Containment: Let $\mathbf{x}$ be a vector and A a blade (grade(A) $\geq$ 1). When $\mathbf{x}$ is on A, then $\mathbf{x} \wedge A = 0 = \mathbf{x} \cdot A^*$. This also yields for a round point.
- Perpendicularity: Let A, B be blades, such that grade(A)$\leq$ grade(B). Then A and B are perpendicular, if $A \cdot B = 0 = B^* \cdot A^*$.

Those tests can also be used to derive geometrical formulas.

Example 15 *Let P and Q be normalized round points in 3-dimensional Euclidean space. Suppose we want to calculate the midplane V that is perpendicular to the line through P and Q and lies exactly between P and Q. This plane can be given by the formula: $V := (Q - P)^*$.*

This plane is perpendicular to the point pair $P \wedge Q$, since:
$(P \wedge Q) \cdot V =$
$(P \wedge Q) \cdot (Q - P)^* =$
$(P \wedge Q) \cdot Q^* - (P \wedge Q) \cdot P^* =$
$0 - 0 = 0$.

Let A be an arbitrary point on V. According to the containment property: $A \cdot V^* = A \cdot (Q - P) = 0$. Therefore: $A \cdot Q = A \cdot P$ and therefore $\sqrt{2A \cdot Q} = \sqrt{2A \cdot P}$. This means that the distance $d(A, P)$ equals $d(A, Q)$. So every point on V has equal distance to P and Q.

2.4.4 Orientation

Some objects, like spheres, lines and planes have an orientation. The orientation can be positive or negative. The orientation is included in the CGA-representation. Given an oriented CGA-line with representation L. The same line with opposite orientation can be easily calculated by multiplying with -1: so $-L$.

In linear algebra the orientation often refers to the sign of a determinant. In CGA round points also have an orientation, since round points are spheres with radius 0.

[10] See [7], p. 23.

2.5 Applications and Software

2.5.1 Visualizers

For calculating and visualizing CGA two useful tools exist: GAViewer and CLUCalc. Both tools can calculate with CGA-expressions, support programming and offer possibilities for visualization.

GAViewer

The GAViewer (see figure 2.1) has been developed at the University of Amsterdam. It can be used for CGA computations and visualization of GA-elements. It supports scripting. It has a good beginner's tutorial, for people who are starting with CGA. The package contains some examples. It also is a nice tool for making demos in CGA. It supports some interactivity with the user. More details can be found in the manual ([7]).

CLUcalc

CLUCalc is an open source application that was written by Dr. Christian B.U. Perwass of the Kiel University (Germany). It performs GA calculations and visualizes the results of the calculations. CLUCalc can be downloaded at `http://www.clucalc.info`. It supports CGA and scripting (CLUScript).

In figure 2.2 a screenshot is given. The user interface consists of three windows: an editor window, a visualization window and an output window. It also supports animations and has possibilities for user interaction. Besides that it has LaTeX-output possibilities and also can produce output for slideshow presentations.

2.5.2 Applications

Geometric Algebra can be used in all areas of science that use geometric properties. In geometrical mathematics several algebraic systems can be rewritten in GA. For example complex numbers, quaternions, projective algebras, tensors and Minkowsky spaces can be written as a GA.

In physics the fields classical mechanics, quantum mechanics and relativity can be described by GA more compactly and intuitively than standard methods do. The Hestenes spacetime algebra $\mathcal{G}(\mathbb{R}^{3,1})$ is an example of a specific GA that can be used in physics.

In computer vision research is done in the field of geometric algebras. GA's are used for example in 3D-camera calibration problems. In this problem several rigidly transformed cameras take a picture of an object. The problem is to determine the position of the cameras and to give a description of the original object. CGA can be used to describe the problem and the solution algorithm efficiently[11].

Geometric algebra elements can also be used as neurons for neural network computations. The geometric product can be used as a so-called associator. In this way neural networks can be used to solve certain geometric optimization problems.[12]

Geometric Algebra is used in robotics, aerospace technology, and biomechanics. More recent examples of those applications of Geometric Algebra can be found in [21] and [25].

[11] See [26]
[12] See [25], p. 291-334.

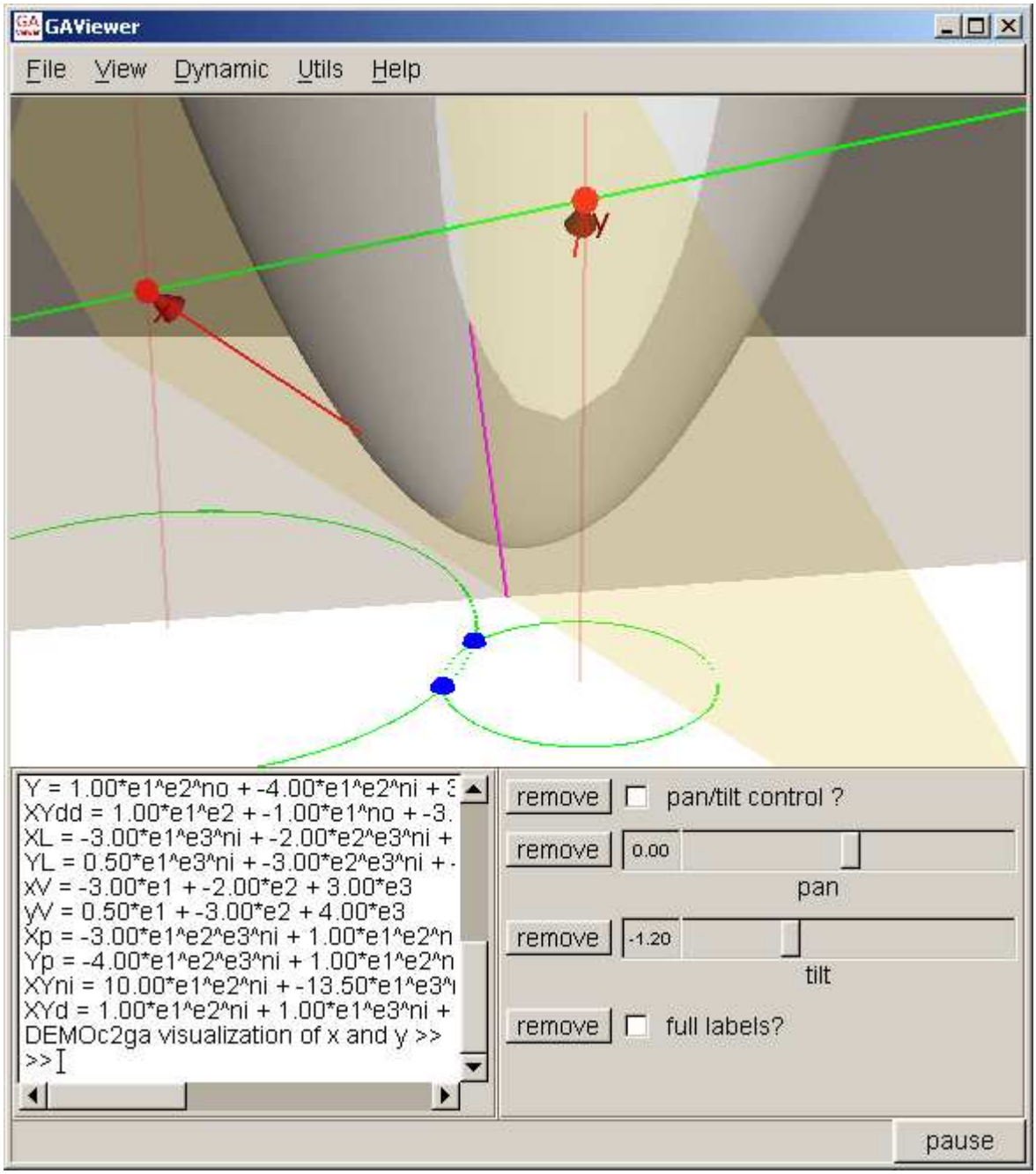


Figure 2.1: Screenshot of GA-viewer

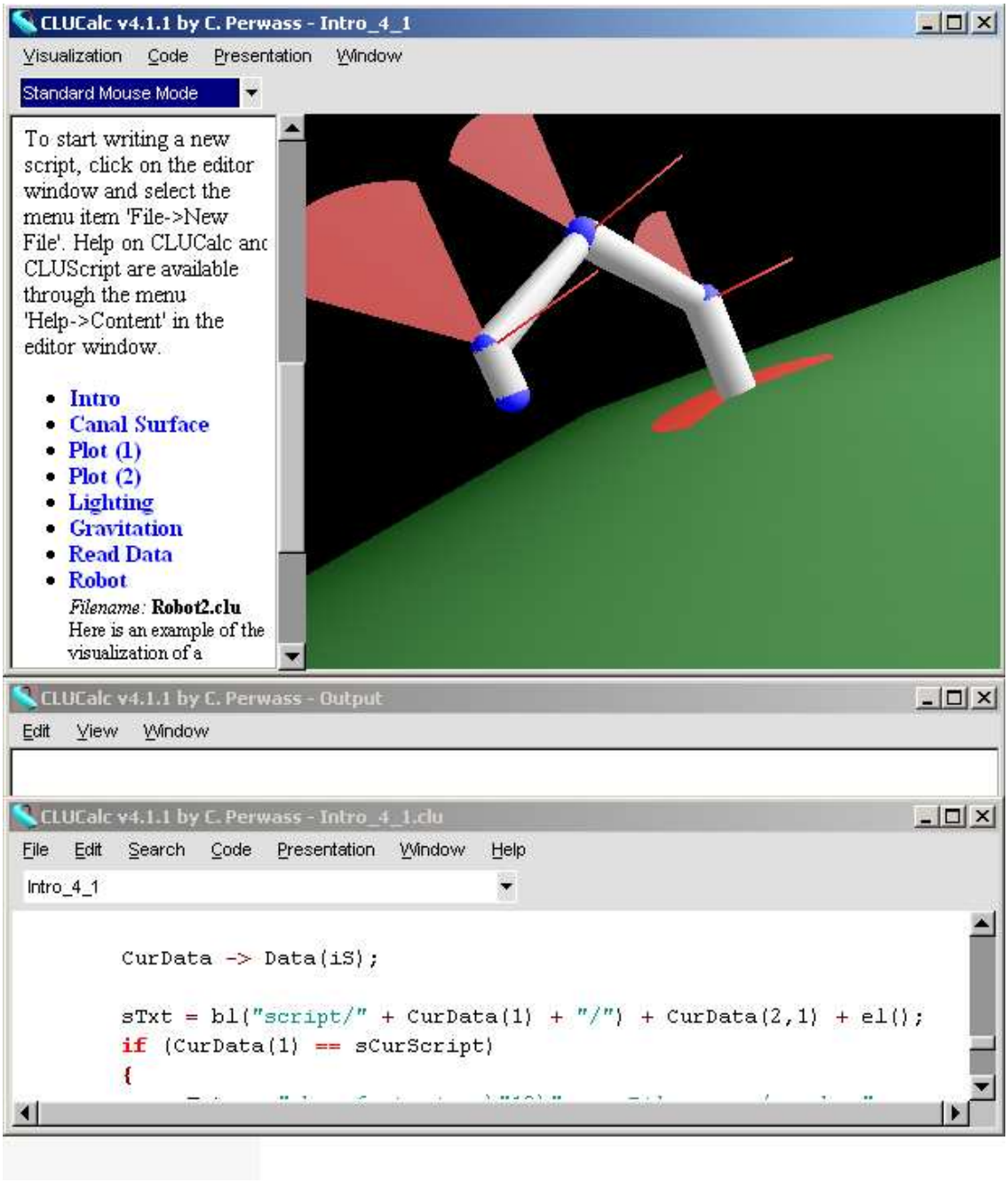


Figure 2.2: Screenshot of CLUCalc

2.6 Discussion: the use of CGA's

In this chapter we discussed the basic concepts of CGA and showed that it is a good alternative to vector linear algebra.

CGA has several advantages, especially when compared to vector linear algebra:

- **coordinate free approach**. The geometric and mathematical perspective of the algebra can be easily separated. Programmers do not have to think in difficult mathematical formulas with matrix multiplications or solutions of linear systems.
- **short formulas**. CGA has a strong expressive power. It is conceptually simple and in this way it avoids programming errors. It deals with geometry in an intuitive way.
- **consistent formulas for different geometrical objects** The same formula for the same operation on different objects. This makes it easier to learn and easier to understand.
- **dimension independent**. When a formula yields in a low dimension, it also works for higher dimensions.

The disadvantages of CGA are:

- **Vector linear algebra is the leading model**. Almost everything in mathematics and physics works with linear algebra. (software, algorithms, books etc.). CGA is relatively unknown and it would require a mind-shift to change this.
- **Experience** Some experience with GA's is needed to perform own calculations. But this also yields for vector linear algebra.
- **More resources**. CGA consumes more memory space and processor time than LA. For every dimension d, this is up till a constant factor in the number of points .

Comparison examples

To illustrate the advantages of CGA, we finish this chapter with some more detailed examples that compare CGA with vector linear algebra.

Example 16 *Representation of a line in 2D*

Let L be a line through the points p and q in $\mathbb{R}^2$. In vector linear algebra there are 5 ways to represent this line:

- *point and direction vector: $l(\alpha) = p + \alpha(q - p)$. For this representation: p and $q - p$ have to be stored.*
- *The equation $\alpha p + (1 - \alpha)q = 0$. For this representation p and q have to be stored.*
- *Using the slope and a point on the line: $l(x) = p + \sin(\alpha)x$. This representation stores a point p and an angle α.*
- *The $\{(x, y)|ax + by = c\}$, with $a, b, c \in \mathbb{R}$ as representation constants.*
- *Plücker coordinates, that uses $p - q$ and $p \times q$ for its representation.*

Which representation is used in the LA-model, depends on the application and the easiest way to perform calculations.

In CGA the blade L always has the same representation, with grade 3. The line can easily be deduced from two points with formula: $p \wedge q \wedge \mathbf{e}_\infty$.

Example 17 *Intersection of plane V and line L in $\mathbb{R}^3$.*

Let V be a plane and L a line in $\mathbb{R}^3$. We want to determine the intersection point P. To solve this problem in vector linear algebra, we need a representation for the plane and the line. However, there are 4 possibilities for the representation of a plane and 5 possibilities for the representation (as shown in the example above). Assume V has equation representation $ax + by + cz = d$ and l has representation $\mathbf{l}(\alpha) = \mathbf{p} + \alpha(\mathbf{q} - \mathbf{p})$.

The intersection point can be calculated as follows:
$a\mathbf{l}(\alpha)_x + b\mathbf{l}(\alpha)_y + c\mathbf{l}(\alpha)_z = d$ *(substitution)*
$a(\mathbf{p}_x + \alpha(\mathbf{q}_x - \mathbf{p}_x)) + b(\mathbf{p}_y + \alpha(\mathbf{q}_y - \mathbf{p}_y)) + c(\mathbf{p}_z + \alpha(\mathbf{q}_z - \mathbf{p}_z)) = d$ *(substitution)*
$\alpha = \frac{d - a\mathbf{p}_x - b\mathbf{p}_y - c\mathbf{p}_z}{a(\mathbf{q}_x - \mathbf{p}_x) + b(\mathbf{q}_y - \mathbf{p}_y) + c(\mathbf{q}_z - \mathbf{p}_z)}$ *under condition:* $a(\mathbf{q}_x - \mathbf{p}_x) + b(\mathbf{q}_y - \mathbf{p}_y) + c(\mathbf{q}_z - \mathbf{p}_z) \neq 0$
The intersection point now equals $P := \mathbf{l}(\alpha)$.

In CGA the representations of the plane and the line are uniquely defined. We just use the intersection formula to determine the intersection point: $P := V^* \cdot l$ (abbreviated as $V \vee l$). When the line and the plane are parallel (no intersection), this formula returns 0.

CHAPTER 3

CGAL

3.1 Introduction

Geometric algorithms are used in computer graphics, virtual reality and robotics. The research on specific geometric problems is done in the research field of Computational Geometry. In the past 25 years, many algorithms have been developed by doing scientific research. The algorithms may look simple, but implementing them in a programming language with some specific needs can be a difficult task.

Many issues have to be taken into account for implementing a framework that uses a geometric algorithm:

- **Data structure**. A useful representation of geometrical objects (points, lines, planes, etc.) should be chosen.
- **Calculations**. A set of operations, that works on the geometrical objects, should be defined and implemented.
- **Output**. The framework should give the results of the algorithm by displaying the points on the screen, or by writing it to a file.
- **Exceptions**. The handling of degenerate cases should be implemented correctly.
- **Precision problems**. Theoretical papers on computational geometry mostly assume exact arithmetic, but in reality rounding errors easily occur and can lead to incorrect results.

Considering those issues, it would take much time for a programmer to implement geometric algorithms from scratch. For this reason several libraries with default geometric algorithms have been developed. The most used C++-library is called CGAL: Computational Geometry Algorithms Library. CGAL contains the building blocks for geometric algorithms, like representations, operations, algorithms and a variety of number types. The building blocks can be used to construct a specific algorithm and can also be easily extended to user specific needs.

CGAL has been developed by researchers of universities in Europe and Israel. The primary goal of CGAL was to make the large amount of geometric algorithms available for industrial application. The library is freeware (for non-commercial use) and open source. Commercial users can buy a license of it[1].

CGAL is still under development. Packages are added regularly. For this CGAP-project we worked with CGAL 3.1. (Release date: December 2004) During this project version 3.2 was released (May 2006). We also tested the implementation with this version.

In this chapter we discuss the basic concepts of CGAL, the design principles, the programming paradigms, and the kernel architecture with the geometrical objects. We focus on the CGAL-concepts that are used in the design and implementation of the CGAP-library. In chapter 6 we will explain how we have been developing the CGAP-library according to the CGAL-design principles. For more information on CGAL, we refer to [3].

3.2 Design principles of CGAL

The CGAL-library has been designed for a wide spectrum of potential users, since Computational Geometry has got many application areas. The library is suited for both scientific research and commercial industry. Therefore the library should be sufficiently generic to be useful for many different applications. In the article 'On the design of CGAL' ([11]) the major design goals of CGAL are explained. Here we will treat those principles in short:

Flexibility

- **Modularity** CGAL has been structured in several modules / building blocks with as few dependencies as possible. For example, the 2D and 3D geometry are separated. Also different algorithms have been structured in different modules.
- **Adaptability** CGAL can be used in another environment/library using its own geometric classes and algorithms. Therefore it might be necessary to adapt some of the existing CGAL-types. For example, in CGAL it is possible to implement user-defined geometric primitives without changing the algorithm itself.
- **Extensibility** The programmer should have the ability to introduce own 'building blocks'. For example a 'point type' that has attached a label (character) to it.
- **Openness** CGAL should be able to work with other libraries like C++ STL, LEDA (number types), GMP Library (multiple precision), OpenGL (displaying 2D/3D-graphics), etc.

Correctness

A library module is correct if it behaves according to its specification. But a module is working correctly only if 1) itself is correct and 2) all modules it depends on are correct. But in CGAL the modules the algorithm depends on, are adaptable. When such a depending module contains an error, the algorithm will fail. Therefore we can never guarantee that a library module is functioning correct.

To work around this problem in a modularized algorithm, it is important to test the modules independently. A way to implement this, is to use assertions of invariants that hold. Those assertions are self-checking functions at runtime. Those assumptions can be programmed in such a way, that they are only checked when producing a debug-version of the program. When the assertion fails, the program terminates immediately with a detailed error message.

[1] See also [11], p. 3.

Example 18 *Correctness example. In the example below a definition of the operator function >> is given. It reads a triangulation from a stream. After the triangulation has been read from the stream, the function performs a check (assertion) whether the triangulation is valid.*

```
template < class Gt, class Tds >
std::istream&
operator>>(std::istream& is, Triangulation_2<Gt, Tds> &tr)
{
  tr.file_input(is);
  CGAL_triangulation_assertion(tr.is_valid());
  return is;
}
```

Robustness

In a theoretical paper exact arithmetic is used, while many implementations use imprecise arithmetic. This can go wrong, because loss of precision can lead to incorrect algorithm decision steps and that can lead to incorrect results. For this reason, mostly an exact computation kernel is available in CGAL for the correctness proof of a theoretical model. Some CGAL-kernels supply the ability to choose the number type: exact, high precision or normal.

Ease of use

The experience of CGAL-programmers ranges from novice to expert level. Learning time and how fast a library gets useful, are important factors for the success of the library.

- **Easy to learn** CGAL users are assumed to have a basic knowledge of C++ and STL (Standard Template Library). CGAL-code is easy to read, because it resembles the programming style of the STL. By giving many slightly adaptable examples and demos in the CGAL-packages, the user should get easily accustomed to CGAL.
- **Uniformity**. For learning and remembering programming concepts, a uniform look-and-feel of design is necessary (naming conventions).
- **Complete and minimal interfaces** CGAL tries to keep interfaces minimal to promote understanding.

For making CGAL easy to learn, CGAL is well-documented. The CGAL 3.1 documentation contains 2312 pages. The documentation is available at different levels: as user manual, reference manual, kernel developer manual, and additional library manual. Besides that the library itself contains many examples and demos to show the strength of the library. In this way the programmer can learn by example.

Efficiency

Wherever possible CGAL uses the most efficient version of an algorithm. Sometimes multiple versions of an algorithm are supplied. The design principle 'efficiency' may conflict with robustness or ease-of-use. In that case, efficiency is sacrificed in favor of those other goals (when it differs a constant factor in computation time).

3.3 Generic programming approach

C++ offers two ways to realize the design principle flexibility: 1) object-oriented programming, that uses inheritance and virtual functions and 2) generic programming, that uses class templates and function templates. Compared to other well-known languages like Java and Delphi, this 'generic programming'-approach is rather unique. In the code-example below both approaches are shown.

Example 19
```
// Object-Oriented Inheritance
class Object2D {};
class Point2D : public Object2D {};

// Generic programming template
class Point3D {};

template <class T>
class LineSegment3D {
    T startpoint, endpoint;
};
```

In the CGAL-design article ([11]) a comparison is given between the two approaches; since this is also an important design issue for our CGAP-package, we summarize this description in this paragraph.

Object Oriented Approach In the OO-approach flexibility is reached in the following way. A virtual base class serves as an interface, and derived classes form the different actual implementations of that interface. Virtual member functions and runtime type information give the user maximum flexibility to choose the actual implementations. Advantages of the OO-approach are the clear definition of the interface and the flexibility at run-time. The OO-approach also has its costs in memory usage and speed: 1) it adds additional memory to each object derived from the base class; 2) an extra call to a virtual function table has to be made when calling a virtual member function; 3) virtual member functions can not be made inline, so code optimization can not be performed within the calling function. In CGAL this would cause a loss in performance, since a lot of small functions are frequently used. Those small functions for example occur in coordinate-access routines and arithmetic on low-dimensional objects.

Generic Programming Approach In the generic programming approach flexibility is reached with template programming in classes and functions. In those templates type parameters are passed symbolically to the function as template arguments. The compiler replaces the templates with the actual types. The compiler only instantiates template member functions that are actually used. This is useful for incremental development. Besides that, it gives the possibility to add extra functionality (with additional requirements on template arguments), that is only compiled when used.
Generic programming offers strong type checking at compile time, so runtime type-errors can not occur. No extra storage is needed and it makes fully use of inline member functions and code optimization techniques. A disadvantage is that it does not have a formal scheme in the language for expressing the requirements of template arguments; this scheme should be described in the documentation.

In many situations the CGAL-developers chose to use the generic programming paradigm to gain flexibility and efficiency. In this way it is compliant with the C++ Standard Template Library (STL). The STL offers powerful concepts, that can be reused in CGAL with only a few additions and refinements. Generic programming also offers the best possibilities for writing fast code and code optimization, which can be an important issue when performing many geometrical/graphical calculations. But besides that in a few places the OO-approach is still used.

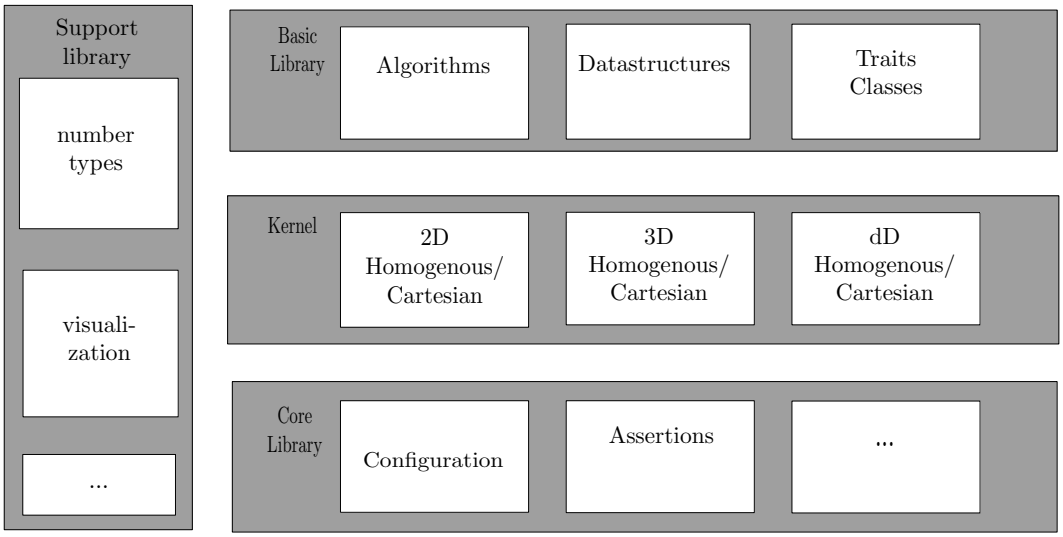


Figure 3.1: CGAL library structure

3.4 CGAL library structure

The CGAL library can be divided into three parts:

1. **The basic library** contains a set of basic algorithms and datastructures. Algorithms are implemented in a generic way, independent of underlying geometric calculations. Traits classes are interfaces between the algorithms / datastructures and the geometric objects in the kernel. The algorithms and datastructures are connected with those traits classes by template parameters (i.e. generic programming)

2. **Kernel** The kernel consists of geometric primitives and operations on these objects. The objects are represented in two ways: 1) as stand-alone classes with the number types (used for calculations) as underlying template parameters; 2) as members of the kernel classes, which allow more flexibility and adaptability of the kernel. In many cases, the kernel classes of CGAL can be used as traits classes for the data structures and algorithms of the basic library. Some kernel classes are by default implemented in CGAL: for several dimensions (2D, 3D and dD) and for several algebraic systems (Homogeneous and Cartesian).

3. **Core library** The basic modules that are needed to make CGAL work, like input/output, randomization, timer, assertion handling, debugging, compiler configuration, etc.

In figure 3.1 the basic structure of CGAL has been modeled. In the figure also the support libraries are visible. Those support libraries, like advanced number types (for example LEDA) and visualization tools (for example OpenGL), can be used with CGAL. CGAL itself contains some classes that handle the communication with those external libraries.

3.5 Kernel architecture

As seen in the previous paragraph, a CGAL kernel consists of geometric primitives and function objects that perform geometric operations on the primitives.

CGAL provides two default kernels: a kernel for cartesian coordinates and a kernel for homogeneous coordinates[2]. Those kernels are available for the dimensions 2, 3 and d ($d \geq 0$). The dimensions 2 and 3 have a rich functionality with lots of available algorithms. The dimension d has less functionality, but is still under development at some parts.

Number types have been made template parameters, such that the kernel offers this flexibility: a field type `Kernel::FT` and a ring type `Kernel::RT`. This refers to the mathematical structures field and ring within the kernel. The ring type only supports +, - and $\times$, the field type also supports $/$.

The CGAL-kernel architecture has been set up in such a way that it can be incrementally developed. A complete functioning kernel would be a complex thing and it would take much time for a programmer to model a complete kernel. In CGAL it is not necessary to implement the whole kernel before using it. We can start from scratch and build in more and more functionality. We just have to implement the functionality that we need. In this way we can do incremental development, and also incremental testing and debugging.

In the next sections we will discuss some basic elements of the kernel in the kernel, since they are needed in the implementation of our CGAP package. In chapter 6, those objects will be integrated with the CGA-geometrical objects.

3.5.1 Geometrical objects

Geometric objects in the kernel are written as `Primitive_dim`, where `Primitive` denotes the name of the geometric primitive object and `dim` denotes the dimension. For example: `Point_2`, `Point_d`, `Vector_2`.

We now will give the precise definitions of the primitive objects. Note that the internal representation of the objects is implemented in different ways, depending on the kernel. In chapter 6 the mapping of those objects on CGA-objects to construct the objects of the CGAP-kernel, will be discussed in detail.

Point

A point is a point in the vector space $\mathbb{R}^d$.

Vector

A vector is a difference between two points p_2 and p_1. It denotes the direction and distance from p_1 to p_2 in the vector space $\mathbb{R}^d$.

Direction

A direction is a vector of which the length is not important. So in CGAL, the objects `Vector` and `Direction` are different mathematical concepts.

Line

Lines represent infinite lines. Lines in CGAL have an orientation: this can be seen as the direction the line directs to. [3]

[2] See [2], pg. 9.
[3] See, [3], p. 18.

Ray and Segment

A ray is a semi-finite interval on a line with a starting point and a direction. A segment can be viewed as a line segment. It is a bounded interval on a directed line.

(Hyper)plane

Planes are available in 3D-kernels and hyperplanes in dD-kernels. A plane also has an orientation. Halfspaces can also be represented by (hyper)planes: the plane serves as the border of the halfplane and the orientation determines on which side the halfplane is situated.

Other objects

Besides those basic objects, various other geometric objects are implemented in CGAL. One can think of triangles and tetrahedra. Other examples are iso-rectangles and iso-cuboids.

Representations for polygons and polyhedra are not directly geometric objects in the kernel. They are treated as datastructures in the basic library (not in the kernel classes). Those representations make use of the kernel-primitives, like points and line segments.

Symbolic constants

Besides the primitive objects, also some constants are defined in the kernel classes. The most important are `ORIGIN` and `NULL_VECTOR`. `ORIGIN` is a symbolic point constant that denotes the point at the origin. It is used for example in the conversion from points to vectors. The `NULL_VECTOR` can be used to construct zero length vectors.

Orientation

In CGAL there is no distinction between full-dimensional objects and their boundaries. So in 3D halfspaces and planes represented by the same: `Plane_3`-object, and spheres and balls are also represented by the same `Sphere_3`-object. To indicate the 'correct' side an `Orientation`-enum is used. An `Orientation` can be `ON_NEGATIVE_SIDE`, `ON_POSITIVE_SIDE` and `ON_ORIENTED_BOUNDARY`[4].

3.5.2 Function objects

A function object is a class with a function in it. In some kernels it is possible to call construction/predicate functions directly. Mostly a function object is used for reaching maximum flexibility: the user can replace a function objects by a user-defined version of a function.

CGAL-kernels contain two types of function objects: predicates and constructions. Constructions are function objects that construct a geometrical object. Predicates are function objects that return a value (like boolean, number, enum) or another class (like Orientation, Comparison_Result, Oriented_side). Examples of predicates are: orientation of a set of points, in-circle tests and point comparison.

[4] See [2], p. 14.

3.6 Summary and conclusion

CGAL has several advantages for use in computational geometry programming problems:

- CGAL is a wide standard library , that has been designed for commercial and scientific applications. It is open source and it is freeware for non-commercial use. It is a multiplatform library that works in Linux/Unix gcc, Microsoft Visual Studio and many other C++ compilers. It has been written in C++, the language most used in computer graphic-applications
- CGAL offers a rich variety of algorithms and datastructures.
- CGAL has flexibility as one of its design goals. Number types and kernels can be chosen by the programmer. Therefore it makes extensively use of template programming. Due to its great flexibility, it is a suited tool for doing scientific research.
- CGAL still is still under development. New releases still regularly occur with bug fixes and new functionality is developed as well.

A disadvantage of CGAL in C++ is that when compared to modern programming languages like Java and C#, this C++ CGAL-library has a long compile-debug-test-cycle. C++ syntax offers a rich freedom, but this also increases the error-probability in your code. Besides that, the template programming can generate difficult type-errors during compile time.

CHAPTER 4

Voronoi diagrams, Delaunay triangulations and convex hulls

In this chapter we discuss three geometrical structures that are used as examples and test cases in the CGAP-package: Voronoi diagrams, Delaunay triangulations and convex hulls. All three structures have several applications in biology, geography and robotics. In computer science these structures are used for problems/algorithms like closest-pair, minimum spanning tree and largest empty circle.

We choose for those structures as examples, because:

- The algorithms to generate those structures are already present in the CGAL-package, so we could base it easily on the existing package. In this way we could make fully use of the 'extensibility' design principle of CGAL.
- There are enough geometric formulas used in the algorithms, so we could express the strength of GA in these examples.
- Convex hulls give us a useful example of how flags can be used in certain datastructures in CGAL. See also chapter 7.

The three structures are closely related. A common property is that they all work with convex polyhedral sets. In the first section we will give some definitions in convex-polytope-theory and we build up a mathematical definition of the three structures. In section 4.2 we will discuss some important datastructures for the three structures. Those datastructures are used in the CGAL-implementation or will be used in the CGAP-package. We will also look at the complexity of such a structure. In section 4.3 some algorithms on the structures will be discussed.

Most examples will be given in 2D, but the definitions and datastructures discussed can be generalized to d-D (arbitrary dimension). In this chapter we will only deal with the most important aspects of the three structures, needed for this project. For further information we refer to the literature: [23], [19], [9], [5], [1] and [10].

4.1 Definitions

In this section we will define several geometrical structures. Those definitions are given in a generic way for d-dimensions. Some examples will be given for the 2D and 3D cases. In this chapter d will denote an arbitrary dimension.

4.1.1 Convexity and polytopes

A set $S \subset \mathbb{R}^d$ is called a *closed set* when the complement of S is open, i.e. every point $a \notin S$ has a small neighbourhood that is also not in S. This neighbourhood N can be defined as follows for some small $\epsilon > 0$: $N := \{n \in \mathbb{R}^d \setminus S | \parallel n - a \parallel < \epsilon\}$.

A set $S \subseteq \mathbb{R}^d$ is *convex* if and only if for any two points p and q in the set S the line segment with endpoints p and q is completely contained in S.

A hyperplane in $\mathbb{R}^d$ splits the plane into two 'halves'. Such a halve is called a *half-space*.

Definition 16 *A polyhedral set is the intersection of finitely many closed half-spaces. A polytope is a bounded polyhedral set.*

In polytope-theory some terminology is used like faces, facets, edges, vertices and simplices.

Definition 17 *Let K be a closed convex set in $\mathbb{R}^d$. If H is a hyperplane that supports K, we call $F := K \cap H$ a face of K.*
A k-face of K is a face of dimension k.
A 0-face is called a vertex.
A 1-face is called an edge or line segment.
A $(d-1)$-face is also called a facet.

Definition 18 *A subset P of $\mathbb{R}^d$ is called a k-simplex if it is the convex hull of $k+1$ affinely independent points. It has exactly $k+1$ vertices and $k+1$ facets. Examples of simplexes are: a triangle in $\mathbb{R}^2$ and a tetrahedron in $\mathbb{R}^3$.*

In the next sections we will define the Delaunay triangulation and the Voronoi diagram. Those structures use convex polyhedral sets (or polytopes) to store the internal datastructure. Algorithms that construct the convex hull can be used to determine the Voronoi diagram in dimension $d-1$. See section 4.3.2. More information on polytopes and convexity can be found in [10].

4.1.2 Convex hull

Definition 19 *The convex hull of a set of points P in $\mathbb{R}^d$ is the boundary of the smallest polytope S such that all vertices in P are contained in S. A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P.*

In Figure 4.3 an example of a convex hull in 2D is given. The convex hull of a set of points P can be abbreviated as $CH(P)$.

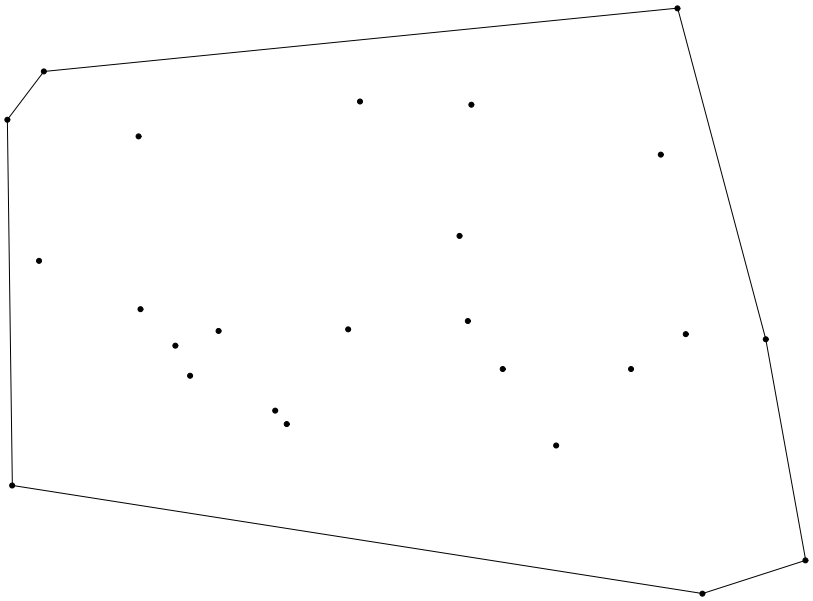


Figure 4.1: Example of convex hull in 2D. The points have been chosen randomly.

4.1.3 Voronoi diagram

The construction process of Voronoi diagrams in 2D can be imagined by growing circles. Imagine an initially number of circles with radius 0 (points), all growing at the same speed. When the circles meet, those pieces of the circles will not grow any further: growing line segments appear between them. When this growing process could be infinitely continued, the final structure would consist of line segments and halflines: the Voronoi diagram. Each grown circle will determine the Voronoi cell of its point of origin. An example of a 2D-Voronoi diagram can be found in figure 4.2.

This intuitive idea can easily be generalized to d dimensions, when we think in spheres/ hyperspheres instead of circles. A Voronoi diagram is also known under the name 'Dirichlet tessellation'. A formal definition louds as follows:

Definition 20 *Let $P = \{p_1, \ldots, p_n\}$ be a point set of n distinct points in $\mathbb{R}^d$ and let I_n be the index set of P. We call the region given by:*

$$V(p_i) = \{x | \, \|x - p_i\| \leq \|x - p_j\| \text{ for all } j \in I_n\} \tag{4.1}$$

the Voronoi cell associated with p_i. So a Voronoi cell is a polyhedral set.

The set given by:

$$\mathscr{V} = \{V(p_1), V(p_2), \ldots, V(p_n)\} \tag{4.2}$$

we call the d-dimensional Voronoi diagram generated by P.

Every vertex q in the Voronoi diagram is on the edge of at least $(d+1)$ different Voronoi cells $V(p_i)$ with center p_i. Due to equation 4.1, the distance from q to every p_i is equal. So we can construct a hypersphere S that goes through all p_i, with center q. This hypersphere S cannot contain any other points from P. (otherwise equation 4.1 would not hold)

There also exists another definition for Voronoi diagrams, based on halfplanes. We define $H(p_i, p_j)$ as the halfplane that contains all points that are closer to p_i than p_j. Let $L(p_i, p_j)$

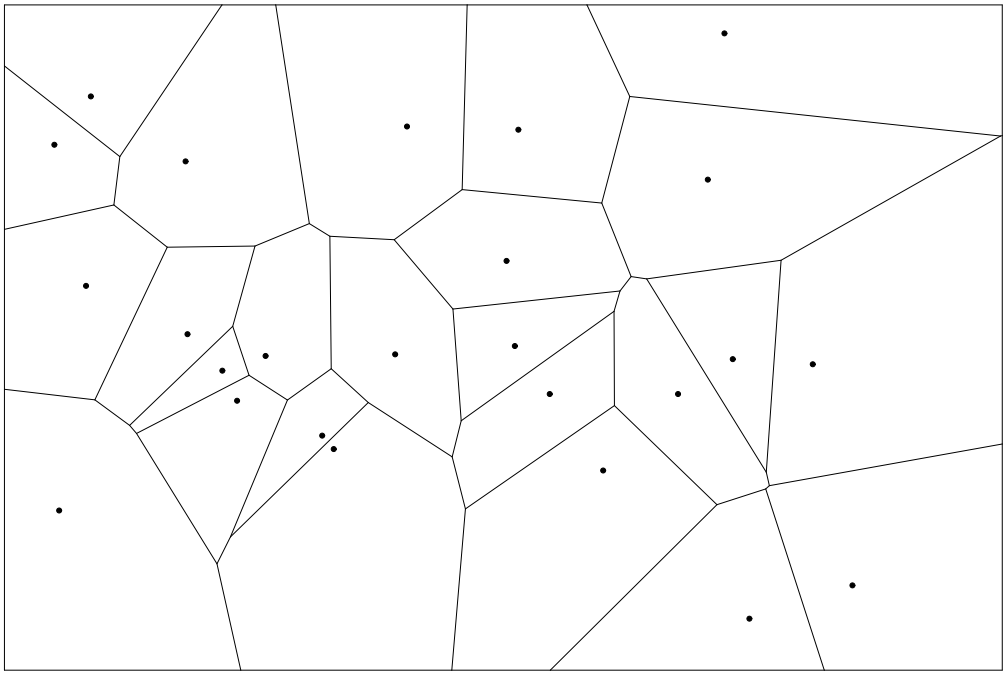


Figure 4.2: An example of a 2-d Voronoi diagram. The same points as figure 4.1 have been used.

be the hyperplane that contains all points that have equal distance to p_i and p_j. This hyperplane $L(p_i, p_j)$ has the following easy formula in CGA: $L = (p_i - p_j)^*$.

Alternative definition 21 *Let $P = \{p_1, \ldots, p_n\}$ be a point set with distinct points in $\mathbb{R}^d$ and let I_n be the index set of P. We call the region:*

$$V(p_i) = \bigcap_{j \in I_n \{i\}} H(p_i, p_j) \tag{4.3}$$

the Voronoi cell associated with p_i. Note that this definition is equal to definition 20.

4.1.4 Delaunay triangulation

In 2D the Delaunay triangulation is a triangulation of the convex hull of the points in the diagram. This triangulation has the property that every circumcircle of a triangle is an empty circle[1]. In figure 4.3 an example is given and in figure 4.4 the empty circles can be seen.

The Delaunay Triangulation of a set op points is *dual* to the Voronoi diagram of those points. That means that every finite line segment in the Voronoi diagram has exactly one corresponding edge in the Delaunay and vice versa. This edge is always perpendicular to the line segment, but it is not necessary that these two intersect.

This description above for 2D can be easily generalized to an arbitrary dimension d. Given a point set P in $\mathbb{R}^d$. The dD Delaunay triangulation is a division of the convex hull of P into d-simplices. For every d-simplex Δ yields the following criterion: the d-dimensional hypersphere through the vertices of Δ does not contain any other point of P.

From the Voronoi diagram book of Boots[2] we cite a formal definition:

[1] See [23], p. 94.
[2] See [23], p. 76.

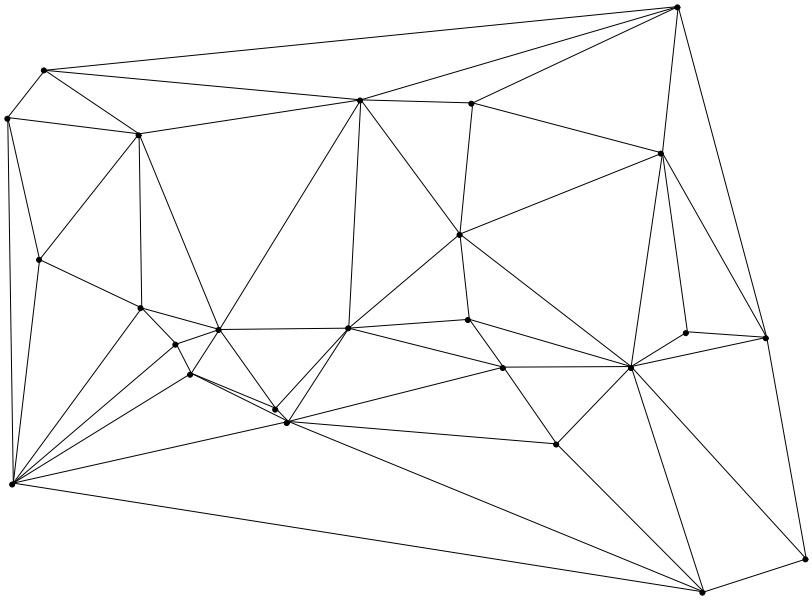


Figure 4.3: An example of a 2-d Delaunay triangulation, based on the points of Figure 4.1 and 4.2

Definition 22 *Let $V(P)$ be a Voronoi diagram generated by the point set $P = \{p_1, \ldots, p_n\}$ in $\mathbb{R}^d$.*

Let $Q = \{q_1, \ldots, q_{n_v}\}$ be the set of Voronoi vertices in V(P). Every $q_i \in Q$ is the center of an empty hypersphere S through $d+1$ points of P. See also section 4.1.3 on the definition of Voronoi diagrams. Those $d+1$ points of the sphere S also span a d-dimensional simplex T_i.

The set $D(P) := \{T_1, \ldots, T_{n_v}\}$ consists of d-dimensional simplices. This $D(P)$ is called the d-dimensional Delaunay triangulation (or tessellation). A simplex in $D(P)$ is called a d-dimensional Delaunay simplex. The union of all those simplices is $CH(P)$.

The structures convex hull, Voronoi diagram and Delaunay triangulation have some other nice properties. We will not discuss them in this thesis, but more information can be found in literature ([23]).

4.2 Datastructures

The datastructures of a Voronoi diagram, a Delaunay triangulation and a convex hull are much alike. In all three structures polytopes play an important role. In this section we will look at a common used datastructure that stores polytopes. This datastructure can also be used for the storage of the three geometrical structures that are the subject of this chapter.

In a datastructure for a convex polygon we need to store two kinds of information:

1. combinatorial information, for example: which points are connected, and which points belong to a line segment or a polygon.
2. the linear structure: the exact location of the points and vector spaces that are spanned by this polytope.

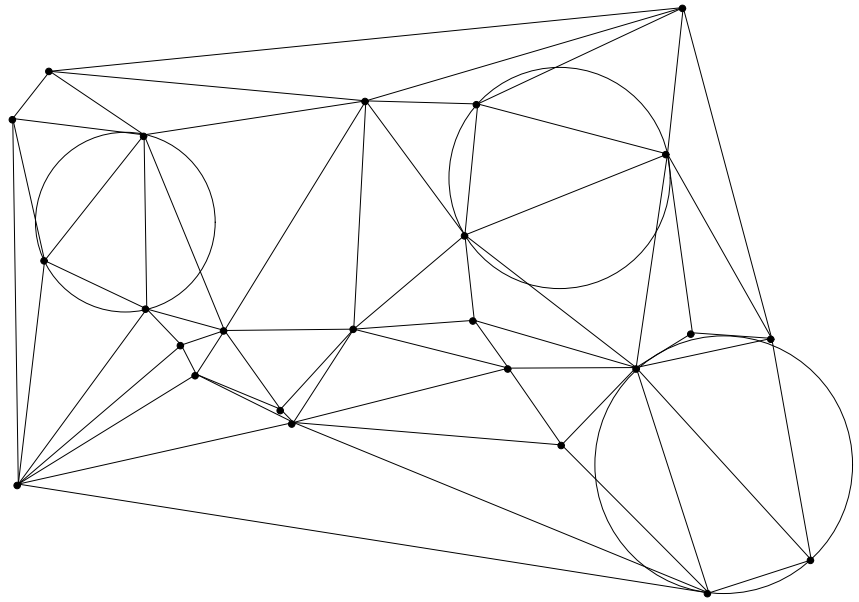


Figure 4.4: Example of three empty circles in the given 2-d Delaunay triangulation, based on the DT of Figure 4.3.

A common datastructure to model this is a so-called 'multiway tree', which is a directed acyclic graph (DAG).

This multiway tree is based on the following fact: An d-dimensional polytope consists of several $(d-1)$-dimensional faces (also called facets). Those $(d-1)$-dimensional faces consist of some $(d-2)$-dimensional faces. In this way we can continue. The 1-dimensional faces (line segments or halflines) consist of 2 points (0-dimensional faces, those can be finite or infinite). The exact location of those points can also be stored in the datastructure.

In the datastructure every face of every dimension of the polytope is modeled by an instance of the following object.

```
class Face{
  int grade;  // The dimension of the vector space embedding space

  vector<*Face> faceList; // list of adjacent faces with dimension grade-1
                          // when grade=1 (point) this list is empty.
  GeometricalInfo gi; // Contains linear information of this facet
}
```

Every instance of this class contains a pointer-list 'faceList' to the faces of the lower dimension (grade-1) that are connected to it. Also some geometrical information (`gi`) is stored here, like the geometric formula of the plane/line/point it lies in.

A problem with this approach are the infinite points. Every `GeometricalInfo` object needs its own handling of infinite points/halflines, halfplanes etc. In the flag-chapter (chapter 5) we will show that those problems can be easily handled by flags.

4.2.1 The complexity of d-dimensional polytopes

A big disadvantage of this multiway tree approach is that it needs a huge amount of memory space to store all faces. This is due to the fact that the number of faces is enormous. In this subsection we will give an estimation of the number of faces to show the complexity.

Definition 23 *A polytope is simplicial if all its faces (of all dimensions) are simplices.*

If the dimension d is even, all neighborly d-polytopes are simplicial (see [19]). The number of faces of a simplicial neighborly d-polytope with n vertices can be calculated by the following function γ ([19], p. 76):
$\gamma(d,n) = \binom{n-\lfloor(\frac{1}{2}(d+1)\rfloor}{n-d} + \binom{n-\lfloor(\frac{1}{2}(d+2)\rfloor}{n-d}$.

Given a Voronoi diagram with n generators in $\mathbb{R}^d$. Define the function $M_k(d,n)$ as the maximum number of k-faces in a Voronoi polytope. For example: from Euler's theorem we know that $M_0(2,n) = 2n-5$, and $M_1(2,n) = 3n-6$[3].

To get an indication of the complexity of the polytopes, we cite from [19] the following results:

$$M_{d-1}(d,n) = \binom{n}{2} \qquad \text{for } d > 3, \text{ all } n, \tag{4.4}$$

$$\frac{n^r}{r!} \leq M_0(d,n) \leq 2\frac{n^r}{r!} \quad \text{for even } d, \text{ with } r = \tfrac{1}{2}d, \text{for } n \to \infty, \tag{4.5}$$

$$\frac{n^r}{r!}\frac{1}{re} \leq M_0(d,n) \leq \frac{n^r}{r!} \qquad \text{for odd } d, \text{ with } r = \tfrac{1}{2}(d+1), \tag{4.6}$$

$$e = 2.71828\ldots, \text{ for } n \to \infty, \tag{4.7}$$

$$M_k(d,n) = \binom{n}{d-k+1} \qquad \text{for } \lceil\tfrac{d}{2}\rceil \leq k \leq d, \text{ for all } n. \tag{4.8}$$

Example 20 *For example when $d = 10$ and $n = 50$ the total number of faces (with dimension 5 or higher) can be estimated:*

$$\sum_{k=5}^{10} M_k(d,n) \approx 2 \cdot 10^7$$

For example when $d = 10$ and $n = 100$ the total number of faces (with dimension 5 or higher) can also be estimated:

$$\sum_{k=5}^{10} M_k(d,n) \approx 1.3 \cdot 10^9$$

Based on the facts given above, we can conclude that when we want to perform calculations on big Voronoi diagram datastructures and Delaunay triangulation structures in an arbitrary dimension d, it is very inefficient to keep track of the whole structure in memory.

4.3 Algorithms

There are many algorithms that construct a Voronoi diagram, Delaunay triangulation or convex hull from a set of points. These algorithms can be classified by the dimension they work on (2-dimensional or more dimensional) and by the type of algorithm: incremental, divide-and-conquer or plane sweep.

[3] For the proof see [5], p. 150.

- **Incremental method** In this case the points are added one-by-one and the datastructure is updated every time. For example the convex hull incremental algorithm has an average time complexity of $O(n)$ in $\mathbb{R}^2$.

- **Divide-and-conquer method** The problem is recursively divided into smaller parts and the structure is created for that part. Then everything is joined together (see [23], pg. 232-238). For example, the convex hull divide-and-conquer-algorithm in 2D has a time complexity of worst case $O(n \log n)$.

- **Plane sweep method** This method uses the plane sweep algorithm over a dimension-axis. The concept is as follows: points are arranged in increasing order of x-coördinates. This technique is especially useful for low dimensions. ($d = 2, 3$).

- **Lifting algorithms** Besides the three algorithm types above, there are also so-called lifting algorithms that transform a d-dimensional Delaunay triangulation or Voronoi diagram construction problem into a $(d+1)$-dimensional convex hull construction problem or intersection problem.

The lifting algorithms are the most important for this thesis, because they are used in CGAL and in the Voronoi diagram application example of chapter 7. We will discuss the general idea of lifting algorithms, and the special cases for Voronoi diagrams and Delaunay triangulations.

4.3.1 Lifting algorithms

There are lifting algorithms available for creating Delaunay triangulations and Voronoi diagrams. The points in dimension d are 'lifted' to dimension $d + 1$ through a lifting-paraboloid function. In that dimension the problem can be simplified to the creation of a convex hull or an intersection problem.

Lifting function: The mapping is given by the following formula. Given a point $p \in \mathbb{R}^d$ which consists of the vector-coordinates $(p_1, p_2, \ldots, p_d)$. The lifting formula creates the new point $p' \in \mathbb{R}^{d+1}$: $p' := (p_1, p_2, \ldots, p_d, \frac{1}{2}(p_1^2 + p_2^2 + \ldots + p_d^2))$. The point in $\mathbb{R}^d$ is lifted to a virtual paraboloid $f(\vec{\mathbf{x}}) = \frac{1}{2} |\vec{\mathbf{x}}|$ in $\mathbb{R}^{d+1}$.

Lifting in CGA: Remember from section 2.4.1 that a normalized round point in $\mathbb{R}^d$ has formula $P = p_1\mathbf{e}_1 + \ldots p_d\mathbf{e}_d + \mathbf{e}_0 + \frac{1}{2} \|p\|^2 \mathbf{e}_\infty$. So the extra coefficient $\frac{1}{2}(p_1^2 + p_2^2 + \ldots + p_d^2)$ is already in the formula as $\mathbf{e}_\infty$-coefficient. This coefficient can be extracted from the point by taking $c_{d+1} = -\mathbf{e}_0 \cdot P$. So the new point can be easily calculated: Then the lifted $(d+1)$-dimensional point of P equals: $P' = P + c_{d+1}\mathbf{e}_{d+1} + \frac{1}{2}c_{d+1}^2\mathbf{e}_\infty$. So lifting in CGA can be done in two simple formulas.

4.3.2 Edelsbrunner algorithm

One specific lifting algorithm was developed by Edelsbrunner and has been published in 1985 [9]. It generates a Voronoi diagram by using tangent hyperplanes. The tangent hyperplanes are tangent to the paraboloid in the lifted points. It uses the so called 'upper envelope'.

Definition 24 *Let H be a collection of hyperplanes in $\mathbb{R}^{d+1}$. Assume that every hyperplane $T_i \in H$ can be written in the form $y = f_i(\vec{\mathbf{x}})$, in which y equals the $(d+1)$-coordinate and $\vec{\mathbf{x}}$ the d-dimensional vector[4].The upper envelope then is defined as $\{(\vec{\mathbf{x}}, \beta) | \vec{\mathbf{x}} \in \mathbb{R}^d, \beta = \max f(\vec{\mathbf{x}})\}$.*

[4]This is the case when all T_i are tangent planes to the paraboloid.

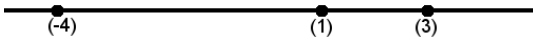


Figure 4.5: 1-D Edelsbrunner algorithm example. The three input points.

The idea of this Edelsbrunner algorithm is that the finite and infinite edges of the $(d+1)$-dimensional upper envelope of the tangent hyperplanes are projected back onto $\mathbb{R}^d$. Those projected edges form the d-dimensional Voronoi diagram.

The upper envelope can be imagined as follows: imagine that every hyperplane T_i has a different color and imagine that the paraboloid is invisible. A viewer is located at position $(\vec{\mathbf{0}}, \infty)$ and looks in direction $(\vec{\mathbf{0}}, -1)$. The colored polyhedron boundary he sees then is the upper envelope. When this upper envelope is projected onto the plane with formula $f(\vec{\mathbf{x}}) = 0$, he sees the colored d-dimensional Voronoi diagram.

The Edelsbrunner algorithm consists of the following steps. Let $P_1, P_2, \ldots, P_n$ be input points in $\mathbb{R}^d$.

1. Lift the points to the paraboloid in $\mathbb{R}^{d+1}$ with CGA-formula $P_i' = P_i + c_i\mathbf{e}_{d+1} + \frac{1}{2}c_i^2\mathbf{e}_\infty$, where $c_i = -\mathbf{e}_0 \cdot P_i$.

2. For each point P_i': define T_i as the tangent hyperplane of the point P_i' with the paraboloid.

3. Let H be the collection of all tangent hyperplanes T_i. Now we calculate the finite and infinite edges of the upper envelope.

4. Those edges are projected back onto $\mathbb{R}^d$.

The result is the Voronoi diagram in $\mathbb{R}^d$. A proof and further explanation of this can be found in [9].

Example 21 *To illustrate the algorithm, we give an example in $\mathbb{R}^1$. Suppose we have got three points: $P_1 = (-4)$, $P_2 = (1)$ and $P_3 = (3)$. The goal is to calculate the 1-dimensional Voronoi diagram. See figure 4.5.*

The first step of the Edelsbrunner algorithm is to lift the points up to the paraboloid $y = \frac{1}{2}x^2$ in $\mathbb{R}^2$. So we get the points $P_1' = (-4, 8)$, $P_2' = (1, \frac{1}{2})$ and $P_3' = (3, 4\frac{1}{2})$. See figure 4.6.

After that we calculate the tangent hyperplanes (lines) of the points with the paraboloid. Those lines have linear formulas $l_1(x) = -4x - 8$, $l_2(x) = x - \frac{1}{2}$ and $l_3(x) = 3x - 4\frac{1}{2}$. So we also get the edges of the upper envelope. The intersection points of the upper envelope are found at $(-1\frac{1}{2}, -2)$ and $(2, 1\frac{1}{2})$. See figure 4.7.

After the points have been projected back on to the 1D axis, it results in the 1D-voronoi diagram. The Voronoi cells are separated by the points $(-1\frac{1}{2})$ and (2). See figure 4.8.

This Edelsbrunner algorithm works for every dimension d. In figure 1 at the beginning of this thesis, another example is given. Here two 2D points are lifted to 3D. The intersection of the two tangent hyperplanes, intersected with the 2D plane results in the 2D Voronoi diagram (i.e. a midline of the two points).

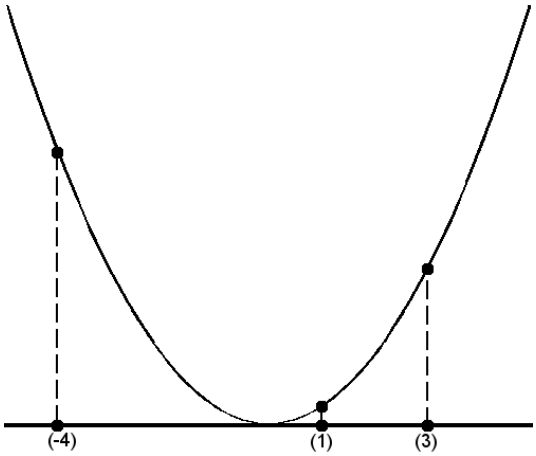


Figure 4.6: 1-D Edelsbrunner algorithm example. The points lifted to the paraboloid.

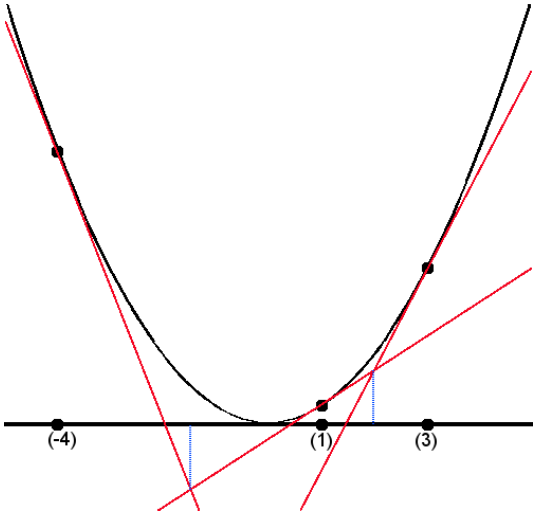


Figure 4.7: 1-D Edelsbrunner algorithm example. The paraboloid with tangent hyperplanes. The upper envelope and the intersection points, projected back on to $\mathbb{R}^1$.

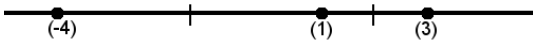


Figure 4.8: 1-D Edelsbrunner algorithm example. The final 1D-Voronoi diagram.

Calculation of the tangent plane

For the Edelsbrunner algorithm a generator point $p \in \mathbb{R}^d$ is lifted to the paraboloid in $\mathbb{R}^{d+1}$. Note: p can written in CGA as $P := p_1\mathbf{e}_1 + p_2\mathbf{e}_2 + \ldots + p_d\mathbf{e}_d + \mathbf{e}_0 + \frac{1}{2}\|p\|^2\mathbf{e}_\infty$. [5] For the algorithm, the tangent plane in the lifted point $p + \frac{1}{2}\|x\|^2\mathbf{e}_{d+1}$ can be calculated in the following way:

Let $Q := \mathbf{e}_{d+1} \cdot P^*$. This Q is the dual of P in the d-dimensional CGA. (note: here we calculate in the $d+1$-dimensional CGA). The tangent plane T_p now equals: $T_p := (Q - \mathbf{e}_{d+1} \wedge (\mathbf{e}_0 \cdot Q)) \wedge \mathbf{e}_\infty$.[6]

In the formula of T_p the expression Q equals the tangent d-vector of the point in the d-dimensional CGA. The expression $-\mathbf{e}_{d+1} \wedge (\mathbf{e}_0 \cdot Q)$ also results in a tangent d-vector and it represents the tangent vector component of the extra dimension $d+1$. Adding those two tangent d-vectors results in the tangent d-vector of the point p. By performing the meet-operation on it ($\wedge$) with the constant $\mathbf{e}_\infty$, the tangent d-vector is transformed into a hyperplane.

Example 22 *Let $p \in \mathbb{R}^2$ be (1,1). Then the corresponding CGA-point P equals: $\mathbf{e}_1+\mathbf{e}_2+\mathbf{e}_0+\mathbf{e}_\infty$. The 2D-dual Q equals:*
$Q := \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_0 - \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_\infty - \mathbf{e}_1 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty + \mathbf{e}_2 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty$.

The plane T_p equals:
$T_p := -\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_\infty + \mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty + \mathbf{e}_1 \wedge \mathbf{e}_3 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty - \mathbf{e}_2 \wedge \mathbf{e}_3 \wedge \mathbf{e}_0 \wedge \mathbf{e}_\infty$.
This plane has characteristics: normal vector $(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}})$ and oriented distance to origin $\frac{1}{\sqrt{3}}$.

Time complexity

Edelsbrunner, O'Rourke and Seidel give an incremental algorithm to compute the arrangement of a set of hyperplanes (including its combinatorial structure)[7]. To analyse the algorithm a so called Zone theorem is used: given an arrangement A of n hyperplanes. Then for every hyperplane H ($H \notin A$), the complexity of the intersection of H with A has order $O(n^{d-1})$. So, adding each hyperplane takes $O(n^{d-1})$: the time to compute the whole arrangement, including its combinatorial structure equals $O(n^d)$. Note that Edelsbrunner works in a lifted (d+1)-dimensional space, so the time complexity for the combinatorial Voronoi diagram then becomes $O(n^{d+1})$.

Another interesting result about the time complexity of the Edelsbrunner algorithm comes from [9]: All 1-level faces (i.e. lines, halflines and line segments) of an arrangement of n hyperplanes in $\mathbb{R}^{d+1}$ can be found in time

$$O(n \log n) \quad \text{for } d = 1, 2 \tag{4.9}$$
$$O(n^{\lceil \frac{1}{2}(d+1) \rceil}) \quad \text{for } d \geq 3. \tag{4.10}$$

And this is worst case optimal for odd d and $d = 2$.

4.3.3 Convex hull in $(d+1)$-D to Delaunay triangulation in d-D

A Delaunay triangulation in $\mathbb{R}^d$ can be computed from a convex hull in $\mathbb{R}^{d+1}$:

[5] Actually, the paraboloid lifting $\frac{1}{2}\|p\|^2$ is already present in the coefficient of the $\mathbf{e}_\infty$-element. However, it is not possible to calculate with the tangent hyperplanes that are producted by this in the `HyperplaneGA`-object of our CGA-package: it uses different intersection formulas. So we had to lift the points to the extra dimension $d+1$, to make it compatible with CGAP.

[6] Those formulas are given for HGA in [28]. As remarked above, we have to add the extra $d+1$th dimension, otherwise the hyperplane would not fit within the `HyperplaneGA`-element.

[7] See [12], p. 16.

1. Lift the points in $\mathbb{R}^d$ to a paraboloid in $\mathbb{R}^{d+1}$.
2. Compute their convex hull with a convex hull algorithm. The set of edges of the lower part of the convex hull is the lifted Delaunay triangulation of the original points.
3. Project the edges of this lower part of the CH on $\mathbb{R}^d$. This forms the Delaunay triangulation.

The explanation of this algorithm can be found in [4].

4.3.4 Delaunay triangulation-comparison: Linear Algebra vs. Conformal Geometric Algebra

To construct a 2D Delaunay triangulation many geometric functions are needed. One of them is to determine whether a point lies in the circumcircle of a triangle.

The function has to test whether the point D lies in the circumcircle of the points A,B and C in 2D. A precondition here is also that the points A,B and C are ordered counterclockwise.

The linear algebra method is by calculating the determinant of:

$$\begin{vmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{vmatrix} \tag{4.11}$$

If this determinant returns a negative value, the point D lies inside the circle. When it returns 0 it is on the circle, otherwise it is outside the circle. The disadvantage of this approach is that it is not intuitively clear why it works. Besides that when we generalize this problem to dimension d, the matrix increases to size $d \times d$.

In CGA: we define the circle S as follows $S = A \wedge B \wedge C$. Condition here is also that the points A,B and C are ordered counterclockwise and that all the points are normalized. For testing whether the point D lies at, inside or outside the circle we calculate the scalar: $\sigma = D \cdot S^*$ (i.e. take the inner product of D and the dual of S). When this scalar is positive, the point D is in the circle, when negative it is outside the circle, when 0 it is on the circle. For higher dimensions (spheres/hyperspheres) we just have to add the extra points to the definition of S. The formulas remain the same.

Proof

We will now prove why this CGA-formula works.

First remember from section 2.4.1 that the dual of a circle S can be rewritten as $S^* := \alpha(M - \frac{1}{2}\left\|\rho\right\|^2 \mathbf{e}_\infty)$. This M equals the (normalized) center as round point. ρ equals the radius of the circle. And α equals a non-zero multiplication factor. This α is positive if and only if A, B and C have been oriented counter-clockwise.

Then remember from section 2.4.1 that the inner product of two normalized round points equals $-\frac{1}{2}$ of the squared distance between the points. So $D \cdot M$ equals $-\frac{1}{2}d^2$, where d equals the distance between D and M.

We now can rewrite the formula:

$$\begin{aligned} \sigma &:= D \cdot S^* \\ &= D \cdot \alpha(M - \frac{1}{2}\left|\rho\right|^2 \mathbf{e}_\infty) \\ &= \alpha(D \cdot M - D \cdot \frac{1}{2}\rho^2 \mathbf{e}_\infty) \\ &= \alpha(-\frac{1}{2}d^2 + \frac{1}{2}\rho^2) \qquad \text{(since } \mathbf{e}_0 \cdot \mathbf{e}_\infty = -1, \mathbf{e}_\infty \cdot \mathbf{e}_\infty = 0, \mathbf{e}_i \cdot \mathbf{e}_\infty = 0 \text{ for } i = 1, 2\text{)} \\ \frac{2\sigma}{\alpha} &= \rho^2 - d^2 \end{aligned}$$

Now, when D lies outside the circle, the distance d of the point D to the center of the circle M is larger than the radius, so $d > \rho$. Then $\rho^2 - d^2 < 0$, so $\sigma < 0$ since α is positive.
In the case that D lies in the circle, then $d < \rho$ and then $\sigma > 0$. When D lies on the boundary of the circle then $d = \rho$ and then $\sigma = 0$.

4.4 Summary

In this chapter we gave definitions, datastructures and algorithms for the convex hull, the Voronoi diagram and the Delaunay triangulation. We also have shown that the three structures are closely related. The Delaunay triangulation is the dual of the Voronoi diagram. The Delaunay triangulation and Voronoi diagram in dimension d can be transformed with a lifting algorithm to a convex hull or halfplane problem in dimension $d + 1$.

The underlying datastructure of the three structures are polytopes and polyhedra. We have seen some possibilities to store those polytopes and to perform algorithms on it. We showed that storing the whole structure of a Voronoi diagram or Delaunay triangulation costs a lot of memory space. In chapter 5 we will see that CGA-flags can be used to store geometric information of higher dimensional polytopes. In chapter 7 an example application of d-dimensional Voronoi diagrams is given.

CHAPTER 5

Flags

A 'flag' is an ambiguous concept in geometry. It has a meaning in linear algebra and in polytope theory. Both definitions look like each other. In this chapter we will define a CGA-flag, which more or less combines the two definitions.

Definition 25 *(Linear algebra) A flag is a sequence of subspaces of increasing dimension of a vector space V.*

For example: $V_0 \subset V_1 \subset V_2 \subset \cdots \subset V_k = V$ Let $d_i :=$ dim V_i for every $1 \leq i \leq k$, then: $d_0 < d_1 < d_2 < \cdots < d_k$

So, in linear algebra a flag is a chain of subspaces. When flags in $\mathbb{R}^3$ have a geometric interpretation, one can think of a flag as: $\mathbb{R}^3$ with a plane in it, with a line in it, with a point in it.

Example 23 *An example of a flag in Euclidean space $\mathbb{R}^3$:*
Let $\mathbf{u} := (1,2,3), \mathbf{v} := (0,1,1), \mathbf{w} := (0,0,1)$.
A flag is the sequence: $V_0 := \{\mathbf{0}\}$, $V_1 := \{\alpha\mathbf{u} | \alpha \in \mathbb{R}\}$, $V_2 := \{\alpha\mathbf{u} + \beta\mathbf{v} | \alpha, \beta \in \mathbb{R}\}$, $V_3 := \{\alpha\mathbf{u} + \beta\mathbf{v} + \gamma\mathbf{w} | \alpha, \beta, \gamma \in \mathbb{R}\}$.

Definition 26 *(Polytope theory) A flag is a connected set of elements from each dimensionality of a polyhedron.*

In 2D a flag of a polygon consists of one vertex and one connected edge. In 3D a flag of a polyhedron contains one vertex, one connected edge and one connected 2D-face. In dD a flag of a polytope consists of a 0D-face (vertex), 1D-face (edge), 2D-face, . . . , (d-1)-face.

Conformal Geometric Algebra is very suitable for representing polytope flags. In one CGA-multivector we can store a pseudoscalar (represents $\mathbb{R}^3$), a plane, a line, a point pair, a point (and even a weight), all in just one multivector. Those geometric objects all have different blade dimensions (see paragraph 2.4.1), so we can easily extract each of the objects from the flag by using the grade operator $\langle\cdot\rangle$.

In this chapter we will show that flags can be used to represent parts of convex polyhedral sets. Those sets can be bounded or unbounded. For example, in a 2D Voronoi diagram there are bounded cells that only consist of finite edges. The unbounded cells also consist of infinite edges. In section 5.2 we will see that flags are very useful in algorithms that work with half-planes. In the previous chapter we have seen that convex hulls, Voronoi diagrams and Delaunay triangulations can be constructed with those halfplane algorithms. So CGA-flags are a useful tool for the construction of those geometrical structures.

5.1 CGA-flags

Definition 27 *Let $B_1, B_2, \ldots, B_n$ be a sequence of n CGA-blades, such that for every B_i yields that grade$(B_i) = i$ and $B_i \subset B_{i+1}$ (from a geometric viewpoint). The CGA-flag corresponding to this sequence is defined as the addition of those blades: $B_1 + B_2 + \ldots + B_n$.*

Example 24 *Let P, Q and R be points in 3D CGA.*
Flat point (grade 2) P: $F := P \wedge \mathbf{e}_\infty$
Line through (grade 3) P and Q: $L := P \wedge Q \wedge \mathbf{e}_\infty$
Plane through (grade 4) P, Q and R: $H := P \wedge Q \wedge R \wedge \mathbf{e}_\infty$

The CGA-flag $Z := F + L + H$ represents the point P on the line L, in the plane H.

A CGA-flag consists of blades of different grades, so the different blades can be easily extracted by using the grade-extraction-operator. In the example above, it is easy to extract the line from it by taking $\langle Z \rangle_3$, since the line has grade 3.

In section 2.4.3 we mentioned that rotations and translations can be calculated by using the versor product. The versor product is distributive, so $V(X+Y)/V = VX/V + VY/V$ for multivectors X and Y and versor V. The versor formula works on all geometric objects mentioned in chapter 2. When a CGA-flag is constructed out of such objects, the versor also works on this CGA-flag. So when $VPV^\dagger$ translates and rotates the point P to a new position, then $VZV^\dagger$ rotates the whole flag to its new position.

More properties and facts about CGA-flags can be found in [6]. From now on we will use the word 'flag' to denote a CGA-flag.

A convex polyhedral set can be defined as the intersection of a finite amount of halfspaces (see 4.1.1). Flags can be used to find the finite and infinite edges that form the boundary of the polyhedral set.

5.2 Flags and convex polyhedral sets

In chapter 7 we will build an algorithm for iterating over the edges of a d-dimensional Voronoi diagram. In chapter 4 we have seen the Edelsbrunner algorithm that uses the upper envelope of a set of tangent halfplanes. The algorithm in chapter 7 works with the Edelsbrunner algorithm. To implement this algorithm with CGA, a CGA-flag model is needed and the geometric functions should be defined. Those building bricks will be discussed in this chapter. First we will discuss the datastructure and then we will also give the necessary geometric operations on the flags.

5.2.1 Datastructure

To iterate over the Voronoi edges, the Edelsbrunner algorithm needs a structure to keep track of the current state of the algorithm. The algorithm outputs finite and infinite edges. Let $H := \{T_1, T_2, \ldots, T_{d+2}\}$ be a set of hyperplanes in $\mathbb{R}^{d+1}$ (corresponding to the current state of the outputted edge). To keep track of the state, we need to store:

- the bounding box. This is the subspace the whole structure lies in.
- a d-dimensional hyperplane that represents the hyperplane T_1. This is the subspace the corresponding d-dimensional face lies in.
- a $(d-1)$-dimensional hyperplane that represents the intersection of T_1 and T_2. This is the subspace the corresponding $(d-1)$-dimensional face lies in.
- ...
- a 2-dimensional hyperplane (line) that represents the intersection of $T_1, \ldots, T_d$. This is the subspace the corresponding 2-dimensional face lies in.
- an 1-dimensional edge as:
  - an infinite edge as the possible upper envelope edge of $T_1, \ldots, T_{d+1}$.
  - a finite edge as the possible upper envelope edge of $T_1, \ldots, T_{d+2}$.

As we have seen in section 5.1, the CGA-flag is capable of storing a whole sequence of i-dimensional hyperplanes with $i = \{1, 2, \ldots, d+1\}$ in one CGA-element. So the flag would be a sufficient structure to store the complete set of hyperplanes and the edge. Even the 'bounding box' can be stored in it as the CGA pseudoscalar element I, that has grade $d+2$.

A problem with this approach, however, are the finite and infinite edges. The line segments (finite edges) and rays (infinite edges) can not be dealt with in the same way as hyperplanes. All hyperplanes are flat CGA-elements. A finite edge could be represented by a CGA point pair. But a point pair is a round CGA-element and does not support orientation. And the CGA-formulas of a point pair do not fit within the intersection algorithm. A ray is a tangent CGA-element. Therefore we chose to separate the set of hyperplanes and the one-dimensional edge (finite/infinite) in the datastructure model. Besides that by making a distinction between the hyperplanes and the infinite, it keeps the code more readable and understandable.

So we designed the datastructure below for our algorithm. Some parts of this structure come from the article of M. Zaharia[1].

```
class FlagGA{
  // multivector that contains the whole flag
  GA_Type f;

  // blades, used to store the infinite edge (ray) or the two
  //    intersection points.
  GA_Type mv1, mv2;

  // Indication whether point1 resp. point2 represents an
  //    infinite or a finite point
  bool p1infinite, p2infinite;

  // Determines whether the segment has the same direction as the line <f>3
```

[1] See [28].

```
  bool segmentInLineDirection;

  // Indication whether the edge is valid
  int validity;
}
```

In this class `FlagGA` the multivector flag `f` is used to store the hyperplanes. The multivectors `mv1` and `mv2` can be used to store the finite and infinite edges. An individual i-dimensional hyperplane can be easily extracted from `f` by calculating the $(i+1)$-dimensional blade[2] of it: $\langle f \rangle_{i+1}$.

The integer `validity` denotes whether the edge is valid or not. This will be discussed in section 5.2.2.

Optimizations

The Voronoi algorithm of chapter 7 has to test a lot of times wheter the objects `mv1` and `mv2` are finite or infinite. This could have been implemented by checking the grade of it and seeing wheter it is a point or a ray: a point has grade 1 and a ray has grade 2.

For speeding up the calculations and keeping the code more readable the booleans `p1infinite` and `p2infinite` were added. Those denote that `mv1` respectively `mv2` are finite or infinite (i.e. a point or a ray).

The boolean `segmentInLineDirection` was added for the same reason. It indicates that the point pair `mv1_ mv2` directs in the same direction as the line `<f>3`. It would take a lot more calculation time to check wheter the point pair has the same direction by determining the orientation in CGA-formulas.

5.2.2 Geometric operations with CGA-flags

The Edelsbrunner algorithm needs several operations that manipulate the flag and extract the output from it. Therefore we modeled several functions in the `FlagGA`-class. The Edelsbrunner algorithm needs to add hyperplane intersections to the flag, and should be able to delete hyperplane intersections from the flag. Besides that the infinite and finite edges must be extracted from the flag as output.

Therefore we modeled the following functions into the `FlagGA`-class:

- `int lowestGrade()`. This function returns the lowest grade of the flag. It is used for determining the number of hyperplanes that are present in the current flag.
- `intersectHyperplane(Hyperplane h)`. The function that adds the intersection of a new hyperplane `h` to the flag. An exception has to be made for the intersection with a 2D-line. This is done in the `intersectLine()` function. In the next subsection, those functions will be discussed in detail.
- `clearGradeFromTo(int from, int to)`. Since the flag is used for tracking the internal state of an iterator, it should also be possible to remove hyperplane intersections from the flag. This function removes the grades from `from` to `to` out of the flag. For hyperplanes removing a grade i is simply done by subtracting the i-blade `<f>i` from the flag `f`. In this function the grades 1 and 2 can also denote the segment and the ray respectively.

[2] Remember from section 2.4.1: an i-dimensional hyperplane has grade $i+1$ in CGA. So it can be found in the CGA-flag F with the formula $\langle F \rangle_{i+1}$.

- `factorizeLineOnPoint(Line l, Point p)`. When the line `l` is intersected with a halfplane `h` it results in an intersection point `p`. This function is used to calculate the ray that originates from `p`, has direction `l` and has the same orientation as the halfplane `h`.
- `extractEdge()`. For extracting output edges from the flag this function was introduced. It extracts an output ray or segment from the flag.

The intersectHyperplane-function

Within the upper envelope calculation algorithm, we need several operations to intersect the tangent hyperplanes with each other. We need a function `intersectHyperplane` that adds a new hyperplane T_i to the flag F: it adds the intersection hyperplane V_i to F. Besides that we need to handle the special case of the intersection of a hyperplane with a line and the intersection of a ray with a hyperplane: `intersectLine`.

M. Zaharia describes in his article[3] the hyperplane intersection operation that performs the operations on a Homogeneous Geometric Algebra. It is called the function CUTFLAG. We rewrote the formulas in Conformal Geometric Algebra-language and extended the code for use with CGAP. This resulted in our function `intersectHyperplane`:

```
void intersectHyperplane(const GA_Type& h)
{
  int minGrade = lowestGrade();

  GA_Type obj = base.grade(1 << minGrade);

  if (minGrade==3)
  {
    // obj is a line
    intersectLine(obj, h);
  }
  else
  {
    GA_Type newblade = hip(h.dual(), obj);  // = meet(obj, h)
    base = base + newblade;
  }
}
```

First the lowest grade i of the flag F is determined. The lowest grade object $\langle F \rangle_i$ is extracted. When this $\langle F \rangle_i$ is a line, the `intersectLine`-function is called, otherwise the intersection is added with the formula: $F := F + \langle F \rangle_i \vee h$. (Note: the operator $\vee$ is the meet operation.)

The intersectLine-function

The `intersectLine`-function is called from the `intersectHyperplane`-function, when the flag contains a line (i.e. its lowest grade equals 3). Now, the function has to determine whether another ray has already been stored in the flag (i.e. `p1infinite` or `p2infinite` is true). When that is the case, the hyperplane should be intersected with the ray: the `intersectRay`-function.

Assumed that no ray is present, it first calculates the intersection point P with the formula $P = h \vee l$. Since we are working with halfplanes, we should know whether we should put the point in the direction of the line l (i.e. `mv1`) or in the opposite direction of the line (i.e. `mv2`). This can be

[3] See [28].

done by calculating the orientation of the flat point. The orientation is the $\mathbf{e}_0 \wedge \mathbf{e}_\infty$-component of P and can be extracted with the formula: $(P \cdot \mathbf{e}_0) \cdot \mathbf{e}_\infty$.

Depending on the positive or negative orientation the first or second `mv` is filled with the round point of P and the other `mv` is filled with the corresponding ray.

```
void intersectLine(Line l, Hyperplane h) {
  GA_Type flatPt = hip(l.dual(), h);  // = meet(obj, h)

  // The orientation can be found in the no^ni component of the flatpoint
  FT ori = hip(hip(flatPt, e_no), e_ni).scalar();

  GA_Type roundPt = flatToRoundPoint(flatPt);
  if (p1infinite && p2infinite)
  {
    // Factorize the line into the rays mv1 and mv2.
    factorizeLineOnPoint(l, roundPt);

    if (ori>0)
    {
      p1infinite = false;
      mv1 = roundPt;
      directionLineSegment = false;
    }
    else
    {
      p2infinite = false;
      mv2 = roundPt;
      directionLineSegment = true;
    }
  }
  else
  intersectRay(roundPt, h, ori);
}
```

The intersectRay-function

When the flag contains a ray and is intersected with a hyperplane, it should result in a line segment. However, since we are working with halfplanes, we have check whether the points both lie in the corresponding halfplanes. For the second intersection point there are three possibilities:

Let the fatface $\Rightarrow$ denote the first point and ray with the arrow directing in the direction of the halfplane. Let the normalface $\leftarrow$ or $\rightarrow$ denote the second intersection point with the arrow directing in the direction of the halfplane (orientation). We now can distinguish three situations:

- $\Rightarrow\leftarrow$. This segment is valid, since both points lie in both halfplanes. This situation gives the flag the validity status: `VALID`.
- $\leftarrow\Rightarrow$ or $\Rightarrow\rightarrow$. Those segments are invalid, since the first point of the segment does not lie in the last halfplane. This situation gives the flag the validity status: `INVALID_FIRST_POINT_NOT_IN_LAST_PLANE`.
- $\rightarrow\Rightarrow$. This segment is invalid, since the rays point in the same direction, so the second point does not lie within the first halfplane. This situation gives the flag the validity status: `INVALID_ORIENTATION_EQUAL`.

```
void intersectRay(GA_Type roundPt, GA_Type h, FT ori){
  // Remark: mv1 is in direction of the line, mv2 in opposite direction
  if (!p1infinite)
  {
    validity = VALID;
    p2infinite = false;
    mv2 = roundPt;

    // Check whether the other point lies in the halfplane or not...
    if (!isRoundPointInHalfplane(mv1, h))
        makeInvalid(INVALID_FIRST_POINT_NOT_IN_LAST_PLANE);
    else{
      // The ori should be opposite to the previous ori
      if (ori>0) makeInvalid(INVALID_ORIENTATION_EQUAL);
    }
  }
  else
  {
    validity = VALID;
    p1infinite = false;
    mv1 = roundPt;

    // Check whether the other point lies in the halfplane or not...
    if (!isRoundPointInHalfplane(mv2, h))
        makeInvalid(INVALID_FIRST_POINT_NOT_IN_LAST_PLANE);
    else{
      // The ori should be opposite to the previous ori
      if (ori<0) makeInvalid(INVALID_ORIENTATION_EQUAL);
    }
  }
}
```

5.3 Summary

In this chapter we have given the basic elements of the `FlagGA`-class. We showed how intersection hyperplane additions and deletions can be carried out on the flag. The functions modeled are suited for the Edelsbrunner algorithm. Besides that, they offer possibilities for other halfplane algorithms as well, for example the calculation of convex hulls and Delaunay triangulations.

We end this chapter with an overview of the advantages and disadvantages of the use of flags compared to the use of the default method (standard vector algebra structures).

Advantages:

- CGA-flags offer all advantages of CGA: easy programming, powerful formulas and dimension independence.
- Infinite lines, half-infinite lines and line segments can be stored in the same flag object in the algorithm.
- Mathematical sectioning computations and topology specific combinatorics are separated.
- Rotations and translations can be performed on the whole flag at once.

Disadvantages:

- Flags use more calculation time and memory (both up to a constant factor).
- It takes more time to understand the CGA-concept and the flags-concept, since a higher abstraction level is required as in ordinary polyhedra calculations.

Idea for future research:
Flags can also be used to store rounds, like a circle and a sphere. Think of: a point that is part of a point pair, that is part of a circle, that is part of a sphere, etc. At this moment no literature can be found on this subject, but it might be an interesting field of research.

CHAPTER 6

CGA package in CGAL

The goal of this master thesis project is to make an implementation of a Conformal Geometric Algebra library in CGAL. In this chapter we will discuss the most important design issues. We will give a justification of the design decisions that form the foundation of the library: we will discuss the Object Oriented model, the geometric primitives and related questions. At the end of this chapter an analysis will be given by comparing the CGAP Object Oriented-model with the design principles of CGAL (as discussed in chapter 3) and the strengths of CGA (as discussed in chapter 2).

6.1 Assumptions and starting points for the OO-model

The Conformal Geometric Algebra is used as a mathematical framework for this library. CGA is a Geometric Algebra that offers the possibility to represent a variety of geometric primitives in one CGA-element: points, lines, planes, circles, spheres, etc. Those geometric objects are also needed in a CGAL kernel.

CGA adds two extra dimensions to the original Euclidean space. Another option for the mathematical framework would have been the Homogeneous Geometric Algebra (HGA). The HGA just adds one extra dimension. Objects in an HGA-kernel in CGAL would consume less memory space and calculation time. However a CGA-kernel is able to represent more objects than an HGA-kernel. For example, it is not possible to store rounds (circles, point pairs and spheres) in HGA.

For the design of the OO-model of the kernel two kinds of requirements have to be taken into account:

1. **CGA-requirements**. As discussed in chapter 2, CGA has some useful properties for programmers, like dimension-independence, short formulas and its coordinate free approach of geometry. In the CGAP package design those properties have to be visible, such that it shows the full strength of CGA.

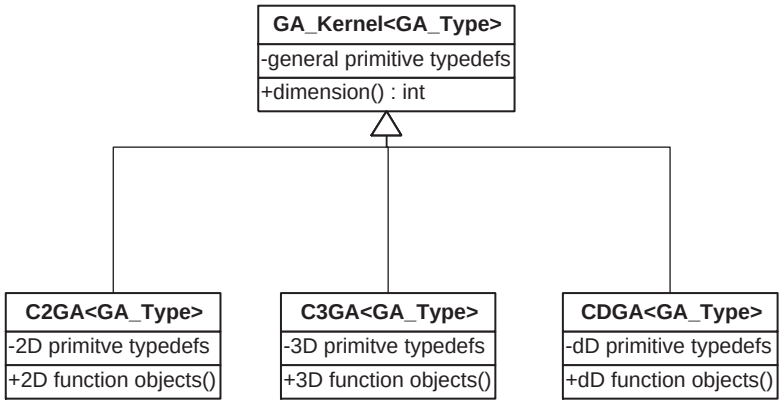


Figure 6.1: The OO-model of the kernels

2. **CGAL-kernel implementation requirements**. The CGAP-package has to be implemented according to the CGAL design principles: flexibility, correctness, ease-of-use, efficiency and robustness. A detailed description of how a specific kernel can be designed and implemented in CGAL, is given in [14]. Some of the ideas and tips of this article have been used in the design process of the CGAP-kernel.

For some issues it is not possible to design the model according to both requirements, due to conflicting approaches. In such a case a choice has to be made between one of two. In the following paragraphs, those choices will be discussed.

It would take too much time to implement a complete CGA-kernel for this master thesis project. A nice property of CGAL-kernels is that they are extensible. Therefore the design of CGAP has been made extensible. Other developers have the ability to extend the kernel in an easy way. In this chapter some recommendations for future work will be given.

6.2 Kernel design

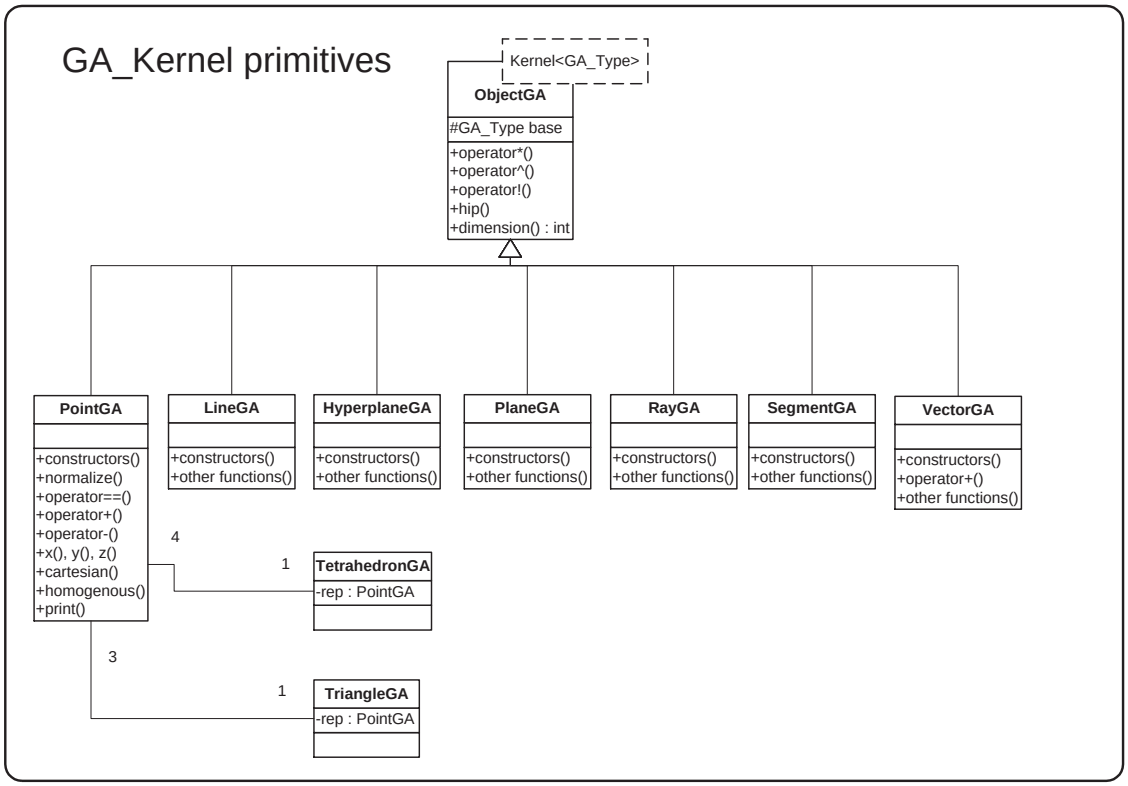
6.2.1 Object Oriented Model

To design and implement geometric kernels in CGAL, there are three parts of interest:

1. The kernel-classes
2. The kernel objects (geometric primitives, like plane, point, line, etc.)
3. The function objects, in which all predicates and constructions are given. Those function objects perform the operations on the geometric primitives in the kernel.

In figure 6.1 a class-diagram of the final kernel design is given. The object model of the CGAP kernel consists of one main-class: `GA_Kernel` and for every dimension type a subclass: `C2GA`, `C3GA` and `CDGA`.

In figure 6.2.1 the final class-structure of the primitives can be found. In the following sections this design will be discussed.



Dimensionality:

The division in classes and subclasses of the CGAP-kernel is mainly based on the different dimensions it can work on. **CGAL** distinguishes 3 types of dimensions: 2D, 3D and dD (i.e. arbitrary dimension). The kernel-objects can support more of those dimension types. For example, the `Cartesian_base` kernel can be used for 2D and 3D objects and operations. The kernel primitives can only have one dimension type, for example `Point_2`, `Line_3`, `Hyperplane_d`.

The **Conformal Geometric Algebra** objects and operations are dimension independent (see also section 2.2). This is one of the advantages of CGA when compared to vector algebra. To show the strength of CGA, we wanted to implement this property in CGAP. For every geometrical object (kernel primitive), we therefore implemented one class, instead of three. For example: we did not implement `Point_2`, `Point_3` and `Point_d`, but only `PointGA`.

Initially all CGA-calculations were implemented with the GAIGEN-library (see also appendix A). GAIGEN is a code-generator and it implements the CGA for one specific dimension, including optimizations. In GAIGEN it is not possible to specify the dimension at run-time. This makes it hard to make one generic CGA-kernel-class (like `Cartesian_base`). For that we should include the generated GAIGEN code of all possible dimensions in the kernel. To keep things small and easily separated, we therefore decided to implement a generic CGA-kernel: `GA_Kernel` and made the different dimensions (2, 3 and d) subclasses of it: `C2GA`, `C3GA` and `CDGA`. The GAIGEN-implementation could be given by the generic programming approach.

Coordinate free?

One of the strengths of CGA is its coordinate free approach of geometry. The programmer does not have to code at the low coordinate level, but can abstract from it by using CGA. In this way complex matrix computations and bug-sensitive formulas are avoided.

We tried to make CGAP as coordinate free as possible. Some standard CGAL-traits classes and algorithms however, have to use x- and y-coordinates directly for their calculations (for example: sorting-algorithms). Of course, we wanted to make our kernel suitable and compatible with those algorithms and traits classes. Therefore we implemented functions in it, such that the points could be transformed to their Cartesian equivalent. Those functions are called `x()`, `y()` and `z()` for 2D and 3D. In dD the function `cartesian(int unitVector)` returns it for an arbitrary dimension.

CGA Number type

The standard CGAL Cartesian kernel supports 2 number types: the field type (FT) and the ring type (RT). The field type is used for all standard arithmetic operations $(+, -, *, /)$, the ring type does not support division. By choosing the number type the programmer can improve the output precision or the calculation speed. In this way also exact arithmetic is possible.

In CGAP we also wanted to reach maximum flexibility. Therefore we implemented the number type of the CGA-kernels as a template parameter.

By default the automatically code-generated GAIGEN kernels are used. They support all needed GA-operations. The internal number type of GAIGEN is also adaptable. By default this type is set to the C++-type `float`. The programmer can specify it in the GAIGEN-code generator and it also has an option to make this number type a template in code.

A disadvantage of GAIGEN is that its GA-dimension is limited to 8. This means that we can represent Euclidean spaces up to dimension 6 with it (CGA uses 2 extra dimensions for $\mathbf{e}_0$ and $\mathbf{e}_\infty$). By using a user defined GA number type, higher dimensionality can be reached. In this user defined type all necessary operators should be implemented (like join, meet, dual, inner product, geometric product etc.)

Adaptable Kernel

One of the design principles of CGAL is its flexibility: it should be easy to extend. (see section 3.2). For the realization of this principle, it should be easy to add new geometric objects and functions to the kernel. Besides that it should be possible to inherit own subkernels of the existing kernels.

Since CGAL uses the generic programming approach, those goals can not be reached by standard OO-modeling. Hert describes in an article [1] a solution to this problem for general programming. For our CGAP-kernels her solution looks as follows in code:

```
// Base kernel definition
template < typename K_, typename GA_Type_>
struct GA_Kernel_base {

  typedef PointGA<K_>      Point;
  typedef VectorGA<K_>     Vector;
...
}

// Sub kernel definition
template <typename GA_Type_>
struct C2GA : public GA_Kernel_base< C2GA<GA_Type_>, GA_Type_> {

  typedef GA_Kernel_base< C2GA<GA_Type_>, GA_Type_>  ParentK_;
```

[1] See [14], p.4

```
public:
  typedef typename ParentK_::Point Point_2;
  typedef typename ParentK_::Vector     Vector_2;
...
}

// Instantiation in the user application that uses the kernel:
typedef c2ga_element GA_Type_2;
typedef CGAL::C2GA<GA_Type_2> FK_2;
class K_2 : public FK_2 {};
```

This way of modeling is also used in the CGAL Cartesian kernel. At first sight, this code looks rather complex. The idea is as follows: the geometric object classes can use the kernel as a template parameter, so the distinct geometric objects can determine the right types of the other objects and operations.

6.3 Objects and operations

In the previous section we discussed the design issues that play a role at kernel level. In this section we will zoom in at the building bricks of the kernel: the objects and operations that are inside the kernel. Therefore we recall the requirements of CGAL with respect to all important objects and operations and we will combine that with the possibilities CGA-geometry. In this way we will account for the choices made.

In the standard CGAL kernels there exist different classes for every dimension. For example: `Point_2` and `Point_3` in `Cartesian` consist of different classes and also predicates and constructions work on different classes. To show the strength of CGA, we decided to use one class for a geometric object, independent of its dimension (2/3/d). So a `Point_2` in `CG2A` and a `Point_3` in `C3GA` represent the same type: `PointGA`.

This also yields for function objects, like `construct_line_x`, since the CGA-formulas are the same for every dimension.

ObjectGA
We also wanted the possibility to use operators on the geometric objects directly. So we could write formulas like `circle = point1 ∧ point2 ∧ point3` instead of `circle.base = point1.base ∧ point2.base ∧ point3.base` , where `base` refers to the underlaying `GA_Type`. This gives our library a lot of expression power. To make this possible we gave all geometric objects one common superclass: `ObjectGA`. In this superclass all standard GA-operators were implemented and connected to their `GA_Type`-equivalents.

Note: In paragraph 3.3 we mentioned that standard CGAL avoids the use of inheritance. Wherever possible the generic programming approach is used instead of object inheritance, since it improves performance. In CGAP we also used this generic programming approach, but for the `ObjectGA`-class we used the object oriented approach. CGA is not meant to be a fast calculation platform, so speed is not an issue here and the inheritance can be used. However, in the `ObjectGA`-class the slow 'virtual inheritance' is used as few as possible.

6.3.1 Kernel primitives in CGAP

PointGA

In CGA a point can be represented in two ways: by a flat point or a round point (see section 2.4.1). For the internal representation of the PointGA primitive the round point was used for the following reasons:

1. All CGA literature that has been used for this thesis, uses mostly round points to represent points.
2. Round points have easier formulas for containment and orthogonality statements.
3. The conversion formula from round point to flat point ($f = r \wedge \mathbf{e}_\infty$) is easier than from flat point to round point ($r = ((f\mathbf{e}_0 f) \wedge \mathbf{e}_0)\mathbf{e}_\infty((f\mathbf{e}_0 f) \wedge \mathbf{e}_0)$), so the conversion step will take a small amount of time.

Although CGA is a coordinate free algebra, the functions `x()`, `y()` and `z()` had to be implemented, because some CGAL algorithms needed these in the point primitives. In section 6.3.3 this will be discussed in detail.

VectorGA and RayGA

In CGA there exist 4 different representations of a vector (see also section 2.4.1): normal vector, tangent vector, free vector and position vector.

In the CGAL-kernel-manual[2] a `Vector` is defined as the difference of two points p_1 and p_2 and denotes the direction and distance from p_1 to p_2. Therefore we represented the `VectorGA` as a free vector, i.e. a vector without a position. Example in $\mathbb{R}^3$: $f = (ae_1 \wedge e_\infty + be_2 \wedge e_\infty + ce_3 \wedge e_\infty)$

A `Ray` r is defined[3] as a directed straight ray. It starts in a point called the source of r and goes to infinity. For the representation of this we choose the tangent vector, which has a position and a direction.

Examples
The `operator+` function of `pointGA` and `vectorGA`. One of the CGAP demo-algorithms requires a function that adds a vector to a point. Let p be the point and f the free vector. The translation-versor T corresponding to f is (see section 2.4.3):
$T = 1 + \frac{1}{2}f$ The new point is given by the formula $p' = TpT^\dagger$.
In C++ code the calculation only takes two simple lines:

```
Point operator+(const Point& p) const
{
     //Translator: 1 + base/2
     GA_Type &t = FT(1.0f) + base/FT(2.0f);

     //Add this vector to the point: versor product
     GA_Type &result = t * p.base * t.reverse();

     return Point(result);
}
```

[2] See [2], p. 85.
[3] See [2], p. 78.

A vectorGA-instance can be created by two pointGA instances p en q. The instance should represent the vector from p to q. When p and q are normalized, this vector can be calculated by the following formula: $(q - p) \wedge e_\infty$. In C++-code:

```
  VectorGA(const Point &p, const Point &q)
  {
 GA_Type & pp = p.normalize();
 GA_Type & qq = q.normalize();
 base = (qq-pp) ^ e_ni;
  }
```

HyperplaneGA and PlaneGA

The 3D-kernel defines a `Plane_3`-class and the dD-kernel a `Hyperplane_d`-class. In 3D this plane represents a flat 2-dimensional surface in Euclidean space. In dD the hyperplane is defined as a flat $(d-1)$-dimensional surface.

As we have seen in section 2.4.1, a (hyper)plane can be defined by d independent points that lie on the hyperplane. Planes can also be constructed by other elements. To show the power of simple CGA-formulas, we give some constructors of `PlaneGA`:

```
 PlaneGA(const Point &p, const Point &q, const Point &r){
   REP = p ^ q ^ r ^ e_ni;}

 PlaneGA(const Line &l, const Point &p){
   REP = l ^ p;}

 PlaneGA(const Segment &s, const Point &p){
   REP = s ^ p ^ e_ni;}

 PlaneGA(const Ray &r, const Point &p){
   REP = r ^ p ^ e_ni;}
```

Compare this to the CGAL cartesian-kernel function that is called by the first constructor of its `Plane_3`-class:

```
plane_from_pointsC3(const FT &px, const FT &py, const FT &pz,
                    const FT &qx, const FT &qy, const FT &qz,
                    const FT &rx, const FT &ry, const FT &rz,
    FT &pa, FT &pb, FT &pc, FT &pd)
{
  FT rpx = px-rx;
  FT rpy = py-ry;
  FT rpz = pz-rz;
  FT rqx = qx-rx;
  FT rqy = qy-ry;
  FT rqz = qz-rz;
  // Cross product rp * rq
  pa = rpy*rqz - rqy*rpz;
  pb = rpz*rqx - rqz*rpx;
  pc = rpx*rqy - rqx*rpy;
  pd = - pa*rx - pb*ry - pc*rz;
}
```

LineGA and SegmentGA

For representing the `SegmentGA` two options were possible: 1) using a tuple of two `PointGA`-objects or 2) by using the point pair GA-element. The first method is used in the standard CGAL Cartesian kernel. For this CGA primitive we choose to use the second: it is consistent with the other GA-primitives that also use GA-elements and it is possible to use GA-formulas on it (like translation, scaling and rotation).

As discussed in section 2.4.1, a point pair can be seen as a 1-dimensional round blade. Getting the two distinct points out of the point pair is done by a so called dissection formula.

Let A denote the point pair GA-element. The two points p_1 and p_2 it contains can be subtracted by the following dissection formula[4]:

$$p_1 = A \cdot (\mathbf{e}_\infty \wedge A) - \|A\|^2 \mathbf{e}_\infty + \|A\|(\mathbf{e}_\infty \cdot A)$$

$$p_2 = A \cdot (\mathbf{e}_\infty \wedge A) - \|A\|^2 \mathbf{e}_\infty - \|A\|(\mathbf{e}_\infty \cdot A)$$

For representing the `LineGA`-class in CGA we used the line CGA element.

TetrahedronGA and TriangleGA

For the implementation of the 2D and 3D Delaunay triangulation algorithms, it was necessary to implement the `triangleGA` and `tetrahedronGA` classes. CGA can not represent simplices (like triangle and tetrahedron) directly in one GA-element. Therefore for those objects we took the same approach as the CGAL Cartesian kernel: a triangle instance is represented by a 3-tuple of points, and a tetrahedron instance by a 4-tuple of points.

Future extensions

The standard CGAL Cartesian and Homogeneous kernels also contain some other geometric primitives, like direction, circle and sphere. Besides that, they have objects for rotating, scaling and transforming objects. In this implementation of CGAP those objects have not been implemented, because this would take too much time and they were not needed for the implementation of the algorithms of chapter 4.

In future these primitives can be implemented in the following way: The `DirectionGA` has a direction vector as internal representation (a direction does not have a length nor position).
The `CircleGA` and the `SphereGA` also have easy representations. Given the round points a, b, c and d, we can define the circle through a, b and c as $a \wedge b \wedge c$ and the sphere through all points as $a \wedge b \wedge c \wedge d$.

The transformation objects (rotation, scale, and translation) can be represented by the objects `Rotation_rep_GA`, `Scaling_rep_GA` and `Translation_rep_GA`. The strength of CGA is that those objects also can be represented by GA-elements. All three transformation types can be represented by so called versor-multivectors (see also section 2.4.3). A versor V is a GA-element that performs the transformation by 'sandwiching': $VAV^\dagger$. Another advantage of CGA is that the transformation calculations for points, lines, planes, circles, etc. all have this same versor-sandwiching-formula.

[4] For a complete derivation, see [17], p. 15.

6.3.2 Operator symbols

A strength of the programming language C++ is that some operator symbols can be overridden for classes. For example the multiplication-operator `*` can be overridden for GA-elements, so we can write `a*b`.

Using this programming technique, GA-formulas can be easily transformed into C++-language-formulas. For example the GA-formula VAV^{-1} can be translated to C++: `v*a*!v`.
In another programming language without operator overloading, like Java, this would become something like: `product(v, product(a, inverse(v)))`. This is more difficult to read and it is also more difficult to see the original GA-formula in it.

So by making use of the C++ operator overloading functionality, the formulas can be easily read (easy notations) and the correctness of it can easily be proven. The C++-code generator GAIGEN also makes use of this operator overloading. In CGAP we chose to implement the operators as follows:

GA-operation	GA-example	C++ function	C++ example
Geometric product	AB	`operator*`	`a*b`
Outer product	$A \wedge B$	`operator^`	`a^b`
Inner product	$A \cdot B$	`hip`	`hip(a,b)`
Inverse	A^{-1}	`operator!`	`!a`
Dual	A^*	`dual`	`dual(a)`
GA-addition	$A + B$	`operator+`	`a+b`
GA-subtraction	$A - B$	`operator-`	`a-b`

The operator `.` (dot) can not be overridden in C++. So we had to choose another function or symbol for the inner product operation. In this case we took the same notation as GAIGEN: the function name `hip`[5].

Note that the operator precedence is already specified in C++, so parentheses have to be used in some cases. For example, the CGA-formula $A \wedge B + C$ should be made explicit by `(A^B)+C`. When the programmer would just write `A^B+C`, this would equal $A \wedge (B + C)$ since the operator `+` is evaluated before the operator `^`.

6.3.3 Function objects

Function objects in CGAL are used to represent basic geometric operations (see section 3.5.2). Function objects can be used by geometric algorithms. Function objects are divided in predicates and constructions. Examples of function objects are the classes `Side_of_oriented_circle_2` and `Construct_circumcenter_3`.

We made two important guidelines for the code of the function objects:

- It should be as coordinate free as possible.
- It should show the formula expression power of CGA.

For the implementation of the function objects in CGAP, we restricted ourselves to those that were needed by the Delaunay Triangulation and Voronoi Diagram algorithms. The final implementation contained 27 function objects. The kernel can be easily extended with other function objects.

To illustrate some detailed design issues, we give an example.

Example: side of oriented sphere in 3D Euclidean space
Given a sphere through the normalized points p, q, r and s. The question is whether a test point t lies on the oriented side of the sphere or not.

[5] Abbreviation of 'Hestenes Inner Product'

In linear algebra this problem is solved by calculating the sign of the following determinant:

$$\begin{vmatrix} p_x - t_x & p_y - t_y & p_z - t_z & (p_x - t_x)^2 + (p_y - t_y)^2 + (p_z - t_z)^2 \\ q_x - t_x & q_y - t_y & q_z - t_z & (q_x - t_x)^2 + (q_y - t_y)^2 + (q_z - t_z)^2 \\ r_x - t_x & r_y - t_y & r_z - t_z & (r_x - t_x)^2 + (r_y - t_y)^2 + (r_z - t_z)^2 \\ s_x - t_x & s_y - t_y & s_z - t_z & (s_x - t_x)^2 + (s_y - t_y)^2 + (s_z - t_z)^2 \end{vmatrix}$$

In code (not included the C++-code of the determinant calculation, which takes another 25 lines):

```
Oriented_side
side_of_oriented_sphereC3(const FT &px, const FT &py, const FT &pz,
                          const FT &qx, const FT &qy, const FT &qz,
                          const FT &rx, const FT &ry, const FT &rz,
                          const FT &sx, const FT &sy, const FT &sz,
                          const FT &tx, const FT &ty, const FT &tz)
{
  FT ptx = px - tx; FT pty = py - ty; FT ptz = pz - tz;
  FT pt2 = CGAL_NTS square(ptx) + CGAL_NTS square(pty) + CGAL_NTS square(ptz);
  FT qtx = qx - tx; FT qty = qy - ty; FT qtz = qz - tz;
  FT qt2 = CGAL_NTS square(qtx) + CGAL_NTS square(qty) + CGAL_NTS square(qtz);
  FT rtx = rx - tx; FT rty = ry - ty; FT rtz = rz - tz;
  FT rt2 = CGAL_NTS square(rtx) + CGAL_NTS square(rty) + CGAL_NTS square(rtz);
  FT stx = sx - tx; FT sty = sy - ty; FT stz = sz - tz;
  FT st2 = CGAL_NTS square(stx) + CGAL_NTS square(sty) + CGAL_NTS square(stz);
  return Oriented_side(sign_of_determinant4x4(ptx,pty,ptz,pt2,rtx,rty,rtz,rt2,
                                              qtx,qty,qtz,qt2,stx,sty,stz,st2));
}
```

But in CGA only two simple formulas are needed:
Let $S = p \wedge q \wedge r \wedge s$ denote the sphere. This sphere has an orientation.
Then $\sigma = t \cdot S^*$ is a scalar that denotes the positive or negative distance from the boundary of S to t (depending on the orientation of S). For a complete proof of this formula see section 4.3.4. The first precondition is that the CGA-round points have got a positive orientation, otherwise it will influence the sign. Therefore the points have to be normalized. The second precondition is that the points are ordered counter-clockwise.

The CGA-code is:

```
Oriented_side side_of_oriented_sphereGA(const Point_3& p, const Point_3& q,
        const Point_3& r, const Point_3& s, const Point_3& test) const
{
    // Precondition check: p, q, r and t should not be in the same plane
    FT testScalar = (hip((p^q)^(r^s),(p^q)^(r^s) )).scalar();
    CGAL_kernel_assertion(testScalar<-GA_Precision || testScalar> GA_Precision);

    // All points should be normalized:
    GA_Type pp = p.normalize();
    GA_Type qq = q.normalize();
    GA_Type rr = r.normalize();
    GA_Type ss = s.normalize();
    GA_Type testt = test.normalize();

    // Do two GA-calculations
    GA_Type dualSphere = ((pp^qq)^(rr^ss)).dual();
```

```
    FT scalar         = hip(testt,dualSphere).scalar();

    // Determine sign, taking into account the precision
    if (scalar < -GA_Precision) return ON_NEGATIVE_SIDE;
    if (scalar >  GA_Precision) return ON_POSITIVE_SIDE;

    return ON_ORIENTED_BOUNDARY;
}
```

When we compare the cartesian approach with the CGA-approach, we see that CGA is shorter and more intuitively clear. Besides that the risk of programming errors is reduced (for example exchanging a plus and a minus sign).

Point normalization

In most function objects the points had to be normalized, before a calculation could be made. In the above example this is done in the lines like `GA_Type pp = p.normalize();`.
The `PointGA`-class uses round points for its representation. In CGA the round points have got an orientation. Actually, the CGA round points are spheres with radius 0 and spheres usually do have an orientation. In CGAL the point-classes of the kernel do not have this property, because cartesian points do not have an orientation. Therefore, when the GA-element p defines a point, then $-p$ also defines the same point. Actually for every $\alpha \in \mathbf{R} - \{0\}$, αp defines that same point. Many function objects make use of the orientation of a circle or sphere. Therefore we should calculate with positively oriented round points, because otherwise the sign could swap. Also some distance functions between points require that the points have weight 1. (i.e. the factor of $\mathbf{e}_0$ should be 1).

Therefore in many functions the input parameters of type `PointGA` are normalized by the function: $p' = \frac{p}{|p|}$, which sets the factor of $\mathbf{e}_0$ to 1. In the previous section we saw an example of those normalizations in the `side_of_oriented_sphereGA`-function.

Translating points to Linear Algebra

In two function objects it was necessary to fall back on standard linear algebra methods, because no elegant CGA-solution was available: The d-dimensional function objects `Contained_in_simplex_CGAd` and `Contained_in_affine_hull_CGAd`. Those functions test whether a point p lies in a d-dimensional simplex S (or convex hull). Since a simplex or convex hull can not be stored in one CGA-element, we had to solve this in another way. Besides that no CGA-specific methods for solving this were available, so we had to make use of the default method to test whether a point lies in a convex hull/simplex.

Let $s_1, s_2, \ldots s_d$ be the Euclidean Cartesian points in $\mathbb{R}^d$ that define the simplex or convex hull. When there is a combination that solves the linear equation $\alpha_1 s_1 + \alpha_2 s_2 + \ldots + \alpha_d s_d = p$ with all α_i between 0 and 1, then the point p is in the simplex. This problem can be solved by a linear programming method that uses linear algebra techniques. This LP-method is already implemented in CGAL in the `LinearAlgebra`-class. Therefore we implemented the algorithm in the same way as the Cartesian variant.

But in this case the points had to be translated from CGA-representation to their Cartesian versions. To perform those translations a special class `Translators` in the file `Translators.h` was added to the kernel to perform all kinds of translations from and to the CGA kernel primitives.

Sorting
For the sorting and comparison algorithms on CGAL points it was necessary to define a direction in which the points could be sorted/compared. Those algorithms require that the functions `PointGA.x()`, `PointGA.y()` and `PointGA.z()` are implemented. To implement those functions, we had to define an orthogonal basis with unit vectors. This orthogonal basis can be accessed by the kernel function `default_base_element`. The issue here was that we wanted to keep CGA as coordinate-free as possible. Therefore we used the most generic solution to solve the problem. This orthogonal basis is freely adaptable by overriding the kernel. By default the basis $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, is chosen, because that gives the fastest calculations. By overriding or changing the PointGA class another orthogonal basis could be implemented.

6.3.4 Details of other design issues

Orientation

In chapter 3 (section 3.5.1) we mentioned that CGAL uses `Orientation` to indicate the side of a half-space or a sphere. A nice property of CGA is that almost all objects do have an orientation enclosed in its representation. (see section 2.4.4) For lines, (hyper)planes, spheres and circles yields that the CGA-object with the opposite orientation can be calculated by multiplying with -1.

In section 6.3.3 we gave a code-sample where orientation is used.

Origin en null_vector

A CGAL-kernel also has two special classes: `Origin` and `Null_vector`. They model respectively the point at the origin and the free vector with length 0. Those classes should be implemented in the CGAP-kernels as well. A CGA actually is coordinate free, so in the basic the use of coordinates is avoided as much as possible. However, in those cases a solution has to be found. For calculations with the `Origin` class the origin can be easily represented by the point $\mathbf{e}_0$. The null vector can be represented by the GA-element 0 (as free vector).

Constants

In the file `GA_Kernel.h` the CGA constants $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_\infty, \mathbf{e}_0$ are defined. Those constants are also defined in the GAIGEN-library.

Those constants were introduced to make the formulas more readable, for example $p \wedge q \wedge \mathbf{e}_\infty$ becomes:
`REP = p^q^e_ni;` instead of
`REP = p^q^((const GA_Type&) GA_Type::ni);`

C++ supports several ways of defining a constant. In this case, we choose to use the `#define`-directive to define it. In most GA-libraries the terms `no` and `ni` are used for denoting $\mathbf{e}_\infty$ and $\mathbf{e}_0$. In CGAL the variable name `ni` is already in use, by a triangulation traits class that is needed by the Delaunay Triangulation algorithm. Therefore we choose to use the names `e_ni` and `e_no`. For consistency and for avoiding the same kind of name-conflicts, we defined the constants `e_e1`, `e_e2` and `e_e3` instead of `e1`, `e2`, `e3`.

6.4 Debugging/validation

The CGAL library has a lot of internal validation steps. This necessary, because CGAL is a large library with a modular structure that is constantly under development. When one module at a low

level contains a bug, it also affects the higher levels.

For example, when you are debugging a self-written algorithm based on CGAL, it can be difficult to trace the error. You made have made the error yourself, but it is also possible that the error propagates from a predicate (function object) that has been written by someone else.

To minimize those kind of errors, CGAL has an internal validation system: pre-conditions, post-conditions and assertions in many variants. Those conditions can be flagged on or off by a compiler directive. At the beginning of a method the input can be checked by calling its pre-condition. When another function object is called, the output can be checked by performing an assertion. And the final output can be validated with a post-condition.

To guarantee correctness, this validation system was also used in the CGAP-package. This was done in two ways: 1)checking the outcomes of calculations of function objects. 2) checking the input and filtering out degenerate cases.

1) An example of the output checking of function objects is given in the code fragment below. The validation can be toggled on by setting the compiler directive `GA_Validation`. In this validation part, the points/objects are translated back to their Cartesian variants and calculated in the Cartesian way. After that a check is done whether the results are the same. When errors occur, the program stops execution and gives a detailed error message.

```
Oriented_side result = calculateGA(p,q,r,t);
#ifdef GA_VALIDATION
  Oriented_side result2 = side_of_oriented_circleC2<FT>(p.x(), p.y(),
 q.x(), q.y(),
 r.x(), r.y(),
 t.x(), t.y());
    CGAL_assertion(result=result2);
#endif // GA_VALIDATION
```

We implemented this on all possible function objects. In this way we can guarantee correctness of our system. (under the condition that the Cartesian kernel is functioning correct)

2) The second method is the input checking by preconditions. Degenerate cases should be excluded from execution, to prevent strange outcomes and bugs.

```
class Side_of_oriented_circle_2{
    Oriented_side calculateGA( const Point_2& p, const Point_2& q,
         const Point_2& r, const Point_2& t) const
    {
      FT testScalar = (hip(p^(q^r),p^(q^r))).scalar();
      CGAL_kernel_assertion(
        testScalar<-GA_Precision || testScalar> GA_Precision);
      ...
}
}
```

Those measures were helpful in the development and implementation phase of this project. They also might be helpful in future development. Besides that we build an extra debugging feature in the PointGA class: a method to print all point details.

Precision errors with floats

When working with floating point numbers, it is difficult to determine whether two floating points are identical. Due to calculations on the numbers some rounding errors can occur.

An example of this is the square root of 2. When working with 8 decimals $\sqrt{2}$ equals 1.41421356. However the square of that number equals 1.99999999. Those floating point numbers are not equal, while they should be.

CGAL-users mostly use exact arithmetic in computational geometry calculations. However, in CGA-elements it is difficult to integrate this kind of arithmetic. In the GAIGEN-library we calculate with floating point numbers. Therefore we had to use floating point numbers as well. But therefore, a solution had to be found to avoid rounding errors. In the PointGA-module the implementation of the function `operator==` makes use of a so-called 'distance measure'. Two points are equal when they are at most this distance from each other. This maximum distance is given by the constant `GA_Precision` (in the CGAP-file GA_kernel), which indicates the precision this measure should be in to catch rounding errors. Default this value is set to $1 \cdot 10^{-6}$. At other points this distance measure is used as well.

As a future extension, exact arithmetic could be added to CGAP. One way would be to add an exact arithmetic floating point type to the GAIGEN library. An even better way of solving this, would be to implement a whole new `GA_Type`-class that has been designed for this purpose. In this new class the internal algebraic CGA-expressions should be stored and only evaluated when it is really needed (for example when a comparison is made or a scalar value should be returned). This class could also be equipped with algebraic formula optimizers.

Degenerate cases

Sometimes degenerate cases occur: points appear on the same line, or multiple delaunay triangulations are possible (because of a regular structure of the points). For a correct implementation of the Voronoi diagram and Delaunay triangulation algorithms, those degenerate cases should also be dealt with.

For completeness, it would be important that degenerate cases are dealt in the right way. The main goal of this project was to research, model and implement a CGA package in CGAL. It would take too much time to implement all degenerate case situations in the function objects. Therefore we did not deal with all details with respect to degenerate cases..

Therefore, when working with the demo-algorithms, the axiom should be that the input is completely random, so that degenerate cases can hardly occur.

6.5 Other libraries

For this CGAP-project some other libraries were used for handling user input and displaying objects on the screen.

6.5.1 FLTK

For implementing the GUI of the demos and examples FLTK was used. The Fast, Light ToolKit is a free GUI-library. It has an interface to OpenGL for 3D graphics programming. It is a multi platform library: its visualization and event systems are separated from the underlying system-dependent code. It is usable in windows, linux and OS2. It includes fluid (FLTK User Interface Designer), a graphical GUI designer.

Opposite to 3D visualization libraries like wxWidgets and Qt, FLTK only implements GUI functions, and lets OpenGL perform the 3D graphics. Therefore it is small and it compiles quickly. The GA-viewer from the University of Amsterdam is also based on FLTK.[6]

More information on FLTK can be found on `www.fltk.org`.

[6] See also `http://en.wikipedia.org/wiki/FLTK`

6.5.2 OpenGL

For the 2D and 3D graphics of the demos and examples of CGAP we used OpenGL. Open Graphics Library is a standard specification, that defines an API for writing applications that produce 2D and 3D computer graphics. The interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL is available as a multi-platform, open source and free library.[7]

More information on OpenGL can be found on `www.opengl.org`

6.6 Summary and conclusion

The strength of CGAP lies in the combination of the CGAL-library and the CGA-technology. For CGAL -programmers as well as CGA-programmers this system offers the ability to get familiar with the combination.

For the CGA-programmer, CGAP makes a wide variety of geometric datastructures and algorithms available that are in CGAL. With this framework new function objects and primitives can be easily implemented, because of the expression power of CGA-formulas. Besides that, scientists that have an own CGA-library/-application -written in C++- can easily adapt it, and integrate it into CGAP.

For the CGAL-programmer, CGAP is an easy way to get accustomed with Conformal Geometric Algebra. Existing applications and algorithms based on CGAL can easily be converted to their CGA-version. This is done by replacing the Cartesian or Homogeneous kernel by a CGAP-kernel.

When compared to other CGAL-kernels, CGAP consumes more memory space and processor time to perform its calculations. This is up till a constant factor. This issue is becoming less important, since due to ongoing new technology, memory space and processor time are still increasing.

At the beginning of this chapter, we defined the two main requirements that had to be taken into account with the development of CGAP: showing the power of CGA and the CGAL design principles. In the next two subsections we will evaluate how those occur in CGAP.

6.6.1 CGA strengths

CGAP shows the strengths of CGA, especially when compared to the Cartesian coordinate system. It uses coordinate free calculations, which results in shorter formulas and methods. Those formulas are easier to read, program and debug. Most geometric primitives have a direct geometric representation in CGA. As many geometric operators as possible have been overloaded for the use of CGAL.

Another important aspect of CGAP is the dimension independent design: 2D, 3D and dD use the same underlying methods, formulas and representations. As shown in chapter 2, this is one of the useful properties of CGA.

6.6.2 CGAL design principles

To evaluate how CGAP integrates into CGAL, we compare it with the CGAL design principles. We recall from section 3.2 that CGAL has been designed according to the following principles:

- flexibility (modularity, adaptability, extensibility, openness.)

[7] See also `http://en.wikipedia.org/wiki/OpenGL`

- correctness
- robustness
- ease-of-use
- efficiency

Flexibility

Modularity
The CGAP-kernels have a similar modular structure as the CGAL kernel. Only the distinction between two and three dimensions vanishes in CGA, because we wanted to show the strengths of CGA. Another modularity issue is that CGAP is not dependent of Cartesian.

Adaptability
The number type the kernels use for CGA-elements are adaptable, since they are a template parameter of the kernels. Default GAIGEN output can be used, but also other number types are supported. User defined and existing number types can be easily ported.

Extensibility
Our library is not complete. The kernels are designed in such a way that other programmers can easily extend its geometric primitives, and operations. (see section 6.2.1)

Openness
This means that CGAP should be open to work with other libraries. The GA-number type itself needs an internal number type. Due to the use of GAIGEN, this number type can be given by a template parameter. In this way also LEDA and other normal or exact arithmetic calculation packages can be used. In the examples the graphical output library OpenGL was used.

Correctness

For quality assurance assertions were added to the code, like in CGAL. Besides that a validation mechanism was added to guarantee that the results of function objects in the CGA kernel are the same as the cartesian kernel. (see section about debugging)

Robustness

Exact computation is possible by using exact arithmetic in the GAIGEN number type. Besides that a mechanism to handle floating-point rounding errors was implemented. (see section 6.4)

Ease of use

For further application and development of CGAP it is important that people can and want to use and extend it. To stimulate this, one of the first objectives is to make it easy to use and understand.

To make CGAP easy to learn we gave it documentation in three levels: a user manual, a reference manual and the design description (in this master thesis). Besides that, the package has been provided with examples and demos from easy to difficult.

The design and implementation of CGAP is uniform with CGAL. Wherever possible the CGAL-standard was kept (notations, comments, validations, rules, etc.).

To express CGA in it, we tried to use as much CGA-formulas as possible, to let CGA-people get accustomed with it.

Efficiency

The last design principle is efficiency. CGAL is meant as an algorithmic library that performs well in scientific and industrial applications.

The disadvantage of CGA is that it is less efficient than Cartesian calculations. It consumes more memory and processor time. In the code of CGAP we tried to be as efficient as possible, but when other design principles could be reached, we sacrificed efficiency.

6.7 Future work

Building a whole CGA-kernel costs a lot of implementation and debugging time. Therefore we choose to implement only those parts that were needed by the Voronoi Diagram, Delaunay Triangulation and Convex Hull implementations. Those three algorithms need the most important geometric operations, so the kernel developed is a good foundation for further development.

CGAP has been built in such a way that programmers can easily extend it, when programmers need a certain functionality.

CGAP contains the most important geometric primitives and an implementation of some function objects. But compared to the function objects of the standard Cartesian CGAL kernel, there are a lot of constructions and predicates that still have to be implemented in the CGAP-package. In section 6.3.1 we already mentioned several ways of future extension possibilities.

To make CGAP more complete, it will be necessary to:

- implement more function objects
- implement some additional geometric primitives
- intercept all degenerate cases in code

Besides that it would be an idea to improve the graphical output possibilities. Now an interface GAGL was implemented that converts a limited amount of CGA-primitives to OpenGL. It would be useful to add more primitives to this interface. A connection to other open source graphical output libraries (like QT-win) would be useful too.

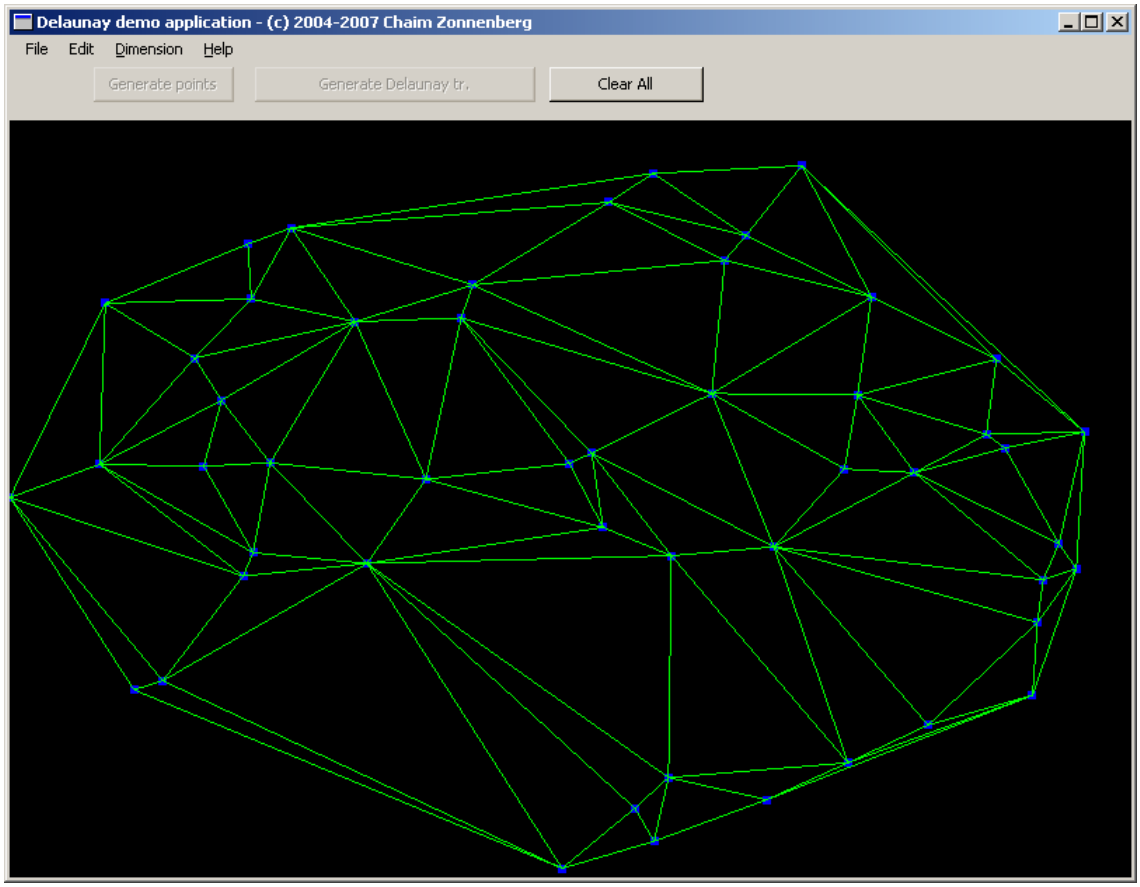


Figure 6.2: Screenshot of 2D Delaunay triangulation application, made with CGAP-kernel.

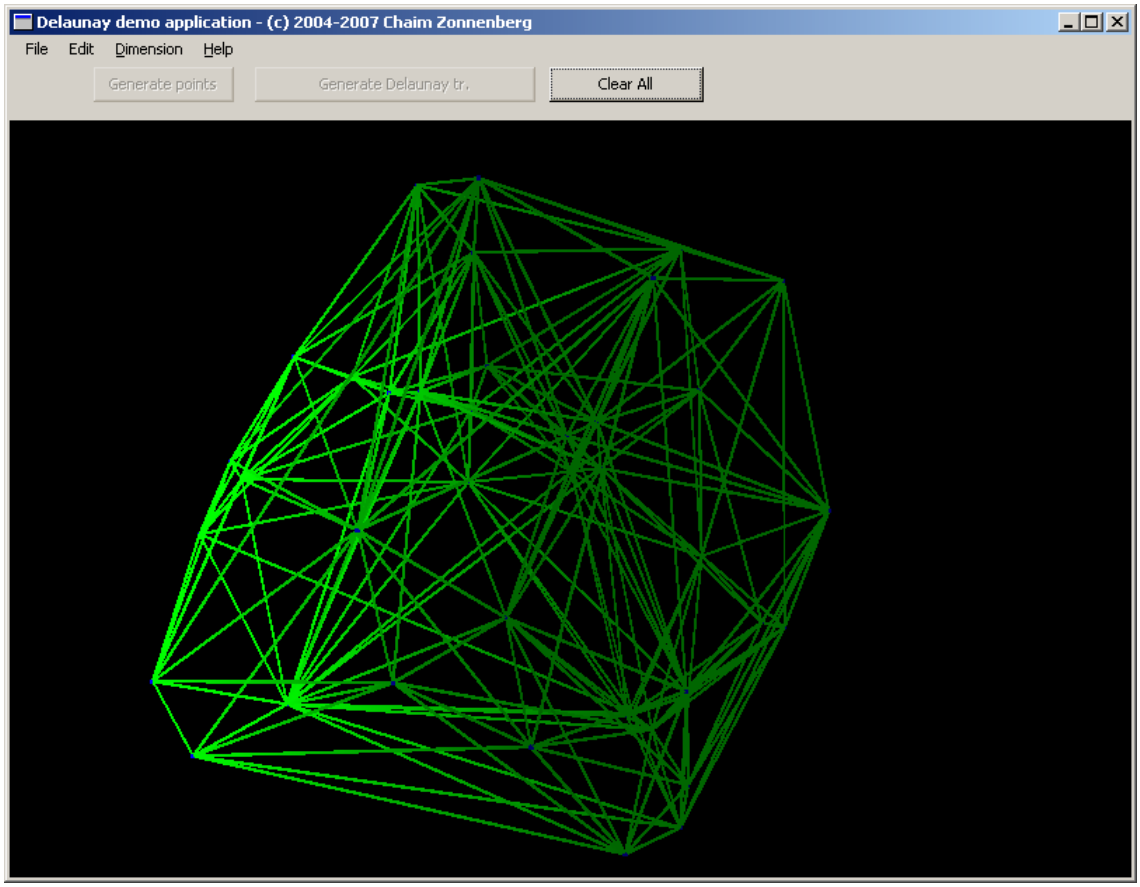


Figure 6.3: Screenshot of 3D Delaunay triangulation application, made with CGAP-kernel.

CHAPTER 7

Application: constructing dD Voronoi Diagrams using CGA-flags

7.1 Introduction

In this final chapter we will show how CGA-flags can be used in concrete geometric algorithms. We will give an algorithm that generates the finite and infinite edges of a Voronoi Diagram in dD. In this chapter important concepts and implementations of the previous chapters will come together. Parts of the Voronoi diagram and halfplane lifting algorithms, discussed in chapter 4, are used; the CGAP-kernel of chapter 6 (that was built on the knowledge of chapter 2 and 3) will be used for the implementation. The CGA-flags of chapter 5 will be used as an internal representation.

7.2 Definition and analysis of the problem

The problem is to generate the edges of a d-dimensional Voronoi diagram, based on a set of given generator points $p_1, \ldots, p_n$. The goal is to show the possibilities of use of CGA-flags and to show the strength of CGA (expression power).

Non-degeneracy assumption. To keep it simple and easy to understand, the point set should be in general position: no $d+1$ points are on the same hyperplane. No $d+2$ points are on the same hypersphere. Degenerate cases are beyond the scope of this thesis.

A new algorithm in CGAL. The default way to implement a CGAL algorithm that works with CGA would be defining a traits-class that uses the CGAP kernel. In the traits-class the CGAP-objects and operations the `Voronoi_d<R>`-algorithm needs, have to be defined. However, in the current CGAL version (CGAL 3.2) a `Voronoi_d`-algorithm does not exist. So we have to define our own `Voronoi_flags_d`-algorithm.

7.3 The algorithm design and implementation

In section 4.3.2 we described the Edelsbrunner algorithm that builds a dD Voronoi Diagram. This is done by lifting the generator points to the paraboloid in $(d+1)$D. For every generator point the tangent hyperplane is defined. After that, the upper envelope of the intersecting hyperplanes is calculated. When this upper envelope is projected back onto dD, it results in the Voronoi diagram.

For the design and implementation of this algorithm we need several building bricks, that have been described in the previous chapters. Those building bricks are:

- The Voronoi diagram algorithm. The Edelsbrunner algorithm that makes use of the upper envelope. (see section 4.3.2).
- Flags, needed to track the internal state of the iterator. (see also chapter 5)
- For the implementation in C++: CGAL with our own CGAP-package in combination with the GAIGEN-code of the CGA-number type. (see chapter 6)
- CGA-operations that are needed by the Edelsbrunner algorithm:
  - the intersection of a tangent hyperplane and the flag-datastructure (see subsections in 5.2.2)
  - the lifting of the point (see section 4.3.2).
  - the calculation of the tangent plane to the paraboloid in the lifted point (see also section 4.3.2).

In chapter 5 we already mentioned that an iterator can be used in the Edelsbrunner algorithm for walking through all the edges of the upper envelope. Now we have to show how this 'walking' exactly works. Those are the last building bricks of the d-dimensional Voronoi diagram algorithm.

7.3.1 Using an iterator

A d-dimensional Voronoi diagram can be described by its vertices and its finite and infinite edges. It has the following properties:

- Every vertex in it is the center of a hypersphere through $d+1$ generator points.
- Every finite edge in it is determined by $d+2$ generator points.
- Every infinite edge in it is determined by $d+1$ generator points.

Lifted to $\mathbb{R}^{d+1}$ the properties above can be rewritten as follows. The $(d+1)$-dimensional upper envelope has the following properties:

- Every vertex of the upper envelope is determined by $d+1$ tangent hyperplanes T_i.
- Every finite edge of the upper envelope is determined by $d+2$ tangent hyperplanes T_i.
- Every infinite edge of the upper envelope is determined by $d+1$ tangent hyperplanes T_i.

To construct the $d+1$-dimensional upper envelope of a set of tangent hyperplanes T_i, we can check for every sequence of $d+2$ hyperplanes whether its finite edge belongs to the upper envelope and for every sequence of $d+1$ hyperplanes whether its corresponding infinite edge belongs to the upper envelope.

However, this can be done in a more efficient way: every set (random order) of d tangent hyperplanes forms a line. A part of this line can be an edge in the upper envelope. There are three possibilities:

- The line forms an infinite edge that is a part of the upper envelope, because it is intersected by just one other hyperplane.
- The line forms a finite edge, since it is intersected by two hyperplanes.
- The line is not part of the upper envelope, since there are other hyperplanes that are above the line.

A way to determine the upper envelope would be to check for every combination S of d tangent hyperplanes, whether its corresponding line forms a finite or infinite edge in the upper envelope (or not). We have to test for all other tangent planes where they intersect the line.

Definition 28 *(Own definition) Let T be a set of n halfplanes. Then we define an (x)-combination of T as a combination of x different halfplanes out of T. Those halfplanes may be randomly permutated. From combinatorics we know that $\binom{n}{x}$ different (x)-combinations can be extracted from T.*

As seen above, every (d)-combination of T forms a line L in $\mathbb{R}^{d+1}$. Given those facts, we can perform backtracking on the set of all possible combinations of hyperplanes.

When we combine the line L with one other halfplane, another intersection point can be found. This intersection point can be the starting point of a finite or infinite edge.

Definition 29 *(Own definition) Let T be a set of n halfplanes. Then we define an (x^+)-combination of T as an (x)-combination combined with another extra halfplane H. So it consists of a set S of halfplanes and an extra halfplane H. We denote this tuple as (S, H). From combinatorics we know that $\binom{n}{x}(n-x)$ different (x)-combinations can be extracted from T.*

Given a (d^+)-combination (S, H) of T, we can calculate the infinite edge F that corresponds to the intersection of $T_{S_1}, T_{S_2}, \ldots, T_{S_d}, H$. Then we have to check for every other hyperplane V ($V \notin S \cup \{H\}$) whether it intersects or is above F.

```
// Calculates the next (d)-combination
int nextCombinationIntern()
{
  int topLevel = -1;
  // Start with the element d and walk back
  for (int i=dimension-1; i>=0; i--) {
    topLevel = i;
    comb[i]++;
    if (comb[i]<=numOfPoints-(dimension-1-i)) break;
    comb[i] = 0;
  }
  if (isEndPosition()) return -1;

  for (int i=topLevel+1; i<=dimension-1; i++) {
    comb[i] = comb[i-1] + 1;
  }
  return topLevel;
}

// Calculates the next combination+
int nextCombination()
```

```
{
  if (isStartPosition()) { initializeCombination(); return 0;}
  if (isEndPosition()) return -1;

  int result = dimension;
  while (!isEndPosition()) {
    comb[dimension]++;

    if (comb[dimension]>numOfPoints) {
      result = nextCombinationIntern();
      comb[dimension]=1;
    }
    //Check if the combination is valid,
    //  i.e. does not contain duplicates
    if (isValidCombination()) return result;
  }
  return result;
}
```

Example 25 *Suppose we have 4 points in $\mathbb{R}^2$. Then the function returns as (d^+)-combinations one-by-one:* $(\{1,2\},3)$*;* $(\{1,2\},4)$*;* $(\{1,3\},2)$*;* $(\{1,3\},4)$*;* $(\{1,4\},2)$*;* $(\{1,4\},3)$*;* $(\{2,3\},1)$*;* $(\{2,3\},4)$*;* $(\{2,4\},1)$*;* $(\{2,4\},3)$*;* $(\{3,4\},1)$*;* $(\{3,4\},2)$*.*

7.3.2 The use of the flagGA

When walking through the (d^+)-combinations, one way to output the right segments would be to calculate the complete intersection of all hyperplanes in the (d^+)-combinations and then checking whether it is an edge of the upper envelope.

But this way would be inefficient. A better solution is to use a flag to track the internal state of the iterator. In the flag all intersection hyperplanes have been stored. So, when just the last hyperplane of the (d^+)-combinations changes, we just have to remove the last intersected hyperplane from the flag and calculate the new one.

For doing this we need functions to add and remove hyperplanes from the flag. In section 5.2.2 we have defined those functions for `FlagGA`. The following code calculates the new flag based on a (d^+)-combination. Therefore the `nextCombination`-function results the last position that has changed. So this function knows from which position on it should refresh the flag.

```
void nextFlag()
{
  // The last position that was changed in the combination
  lastPos = combination.nextcombination();

  if (combination.isEndPosition()) return;

  int toReset = dimension+2 - lastPos;

  // Clear the changed combinations from the flag
  currentFlag.clearGradeFromTo(1, toReset);

  // Add the added combinations into the flag
  for (int i=lastPos; i<dimension+1; i++) {
    int ptIndex = combination.comb[i]-1;
```

```
    Lifted_Hyperplane_d plane = planes[ptIndex];

    currentFlag.intersectHyperplane(plane);
  }
}
```

7.3.3 Checking the edge

In the last subsection, we gave a method to walk through the possible (in)finite edges of the Voronoi diagram. Every (d)-combination results in a line that could be partly part of the Voronoi diagram. Now we need a function to determine whether a part of the line is contained in the upper envelope. This is simplified by investigating every infinite edge of every (d^+)-combination.

From the end of section 7.3.1 we can simply deduce that there are three possibilities: 1) the infinite edge is completely a part of the upper envelope, 2) the infinite edge is partly part of the upper envelope, since it intersects with another hyperplane: it forms a finite edge, 3) the infinite edge is not a part of the upper envelope, since there is a tangent hyperplane 'above' the starting point of the ray.

To determine whether this is the case we should check 2 things for every tangent hyperplane not in the (d^+)-combination[1].
1) whether the hyperplane is above the starting point. In that case the flag has `validity` set to `INVALID_FIRST_POINT_NOT_IN_LAST_PLANE` and then the infinite edge can not be in the upper envelope. (see possibility 3)
2) whether the hyperplane intersects the ray. When this is not the case the flag `validity` is set to `INVALID_ORIENTATION_EQUAL`. When the hyperplane intersects the ray, the flag is VALID. It could be possible that there are more hyperplanes that intersect the flag. In that case the intersection point with the lowest distance to the starting point of the ray forms the segment.

In C++ code we wrote a function to test whether the current ray can be formed to a correct edge:

```
bool canConstructValidEdge()
{
  int minDistPt = -1;
  FT minSqrDistance = 0;
  // For every ray should be tested whether a hyperplane intersects it
  for (int i=0; i<numOfPoints; i++) {
    // Check whether it is in the combination
    if (!combination.containsPoint(i))
    {
      // Clear the 1 dimensional part of the flag
      currentFlag.clearGradeFromTo(1,1);

      currentFlag.intersectHyperplane(planes[i]);

      // In this case, the starting point is below the hyperplane of i
      if (currentFlag.validity==
          currentFlag.INVALID_FIRST_POINT_NOT_IN_LAST_PLANE)
        return false;

      if (currentFlag.validity==currentFlag.VALID)
      {
        FT sqrDist = currentFlag.segmentSqrDistance();
```

[1] The `intersectRay` in `FlagGA` sets a `validity` status to indicate those cases.

```
        if (minDistPt==-1) {
          minSqrDistance = sqrDist; minDistPt = i;
        }
        else {
          if (sqrDist<minSqrDistance)
          { minSqrDistance = sqrDist; minDistPt = i;}
        }
      }
    }
  }
  currentFlag.clearGradeFromTo(1,1);

  // When minDistPt is specified,
  //   the flag consists of two points: a finite edge
  // Otherwise it is an infinite edge
  if (minDistPt>-1) {
    // Prevent the segment from occurring twice in the output
    if (combination.comb[dimension]<minDistPt+1)
    {
      Lifted_Hyperplane_d plane = planes[i];
      currentFlag.intersectHyperplane(plane);
    }
    else return false;
  }

  return true;
}
```

7.3.4 The algorithm in a class Voronoi_Flags_d

All described functions of this section were integrated in one algorithm class that is designed according to CGAL-design principles: `Voronoi_Flags_d`. Within this class an iterator was defined to iterate over the finite and infinite edges of the Voronoi diagram.

The `++-operator` uses the `nextFlag`-function and the `canConstructValidEdge`-function of the previous subsections:

```
self& operator++() {
  while (!combination.isEndPosition()) {
    nextFlag();
    if (combination.isEndPosition()) return;

    if (canConstructValidEdge()) {
      if (currentFlag.isInfiniteEdge()) {
        Lifted_Ray_d r = currentFlag.extractRay();
        currentEdge.setRay(unlift(r));
      }
      else {
        Lifted_Segment_d s = currentFlag.extractSegment();
        currentEdge.setSegment(unlift(s));
      }
      return;
    }
```

```
  }
  return *this;
}
```

The final application

To let the algorithm work and test it with some input, we built a small file application. This application reads a set of d-dimensional points $(2 \leq d \leq 5)$, calculates the corresponding Voronoi diagram using the `Voronoi_flags_d` module and writes the output segments to a file. Besides that, a viewer for 2D and 3D cases was built. See figures 7.2 and 7.3.

To show the ease of use, we give a code sample from the `viewer`-function of the application:

```
for (Voronoi_Edge_Iterator vei = voronoi.all_edges(); !vei->isEndPostition();)
{
  Voronoi_Edge edge = *vei;
  outputEdge(edge);
  ++vei;
}
```

Time complexity

In section 4.3.2 we mentioned two algorithms that returned the upper envelope. The first algorithm of Edelsbrunner, O'Rourke and Seidel, returns the whole combinatorial structure of the Voronoi diagram in time $O(n^{d+1})$. The second algorithm worked only on finite and infinite edges. The worst case bound of summing up all finite and infinite edges of an upper envelope equals $O(n^{\lceil \frac{1}{2}(d+1) \rceil})$ for $d \geq 3$. Notice that we work in a lifted dimension $(d+1)$, so the edges of a d-dimensional Voronoi could be found in $O(n^{\lceil \frac{1}{2}(d+2) \rceil})$ for $d \geq 2$.

The algorithm described in this chapter is comparable to the first algorithm (of Edelsbrunner, O'Rourke and Seidel). With some adaptations our iterator could iterate over all 2-faces or arbitrary dimensional faces (see future work, section 7.6). So it could be made capable of iterating over the whole combinatorial structure.

Our algorithm walks through all (d^+)-combinations. In section 7.3.1 we saw that there are $\binom{n}{d}$ different (d)-combinations. Every (d)-combination has $(n-d)$ (d^+)-combinations. For every (d^+)-combination, we iterate over all $(n-d-1)$ left points in the funtion `canConstructValidEdge`. Multiplying those factors results in the time complexity of our algorithm: $O(\binom{n}{d}(n-d)^2)$.

With a smart optimization technique we could exceed the time complexity of Edelsbrunner, O'Rourke and Seidel: $O(n^{d+1})$. In the current algorithm every (d)-combination, results in a line L on which we test wheter a part of it belongs to the Voronoi structure. This testing process now takes $O((n-d)^2)$, as we have seen above. This can be optimized to $O(n)$ by first calculating all intersection points of the halfplanes and L. Note that for the (in)finite edge yields that 1) it lies on the line L and 2) it lies in every halfplane. With every intersection point we associate its direction (of the halfplane) with respect to the line. This first walkthrough would take time $O(n)$. When we now walk through the intersection points (including directions) we could determine which part of the line belongs to the upper evelope (or that no part of the line does belong to the envelope). This second walkthrough also is $O(n)$. So the testing process for every (d)-combination would have complexity $O(n)$. So our algorithm would be $O(\binom{n}{d}n)$ in its optimized form (this is even stronger than $O(n^{d+1})$ of Edelsbrunner, O'Rourke and Seidel).

7.4 Why an iterator?

At the University of Amsterdam research was done for the use of CGA's in Voronoi Diagram algorithms. In paper [28], M. Zaharia describes two ways to generate a d-dimensional Voronoi diagram in the programming language C:

1. In the first algorithm every d-dimensional Voronoi cell is represented by the node of a directed acyclic graph (DAG). The datastructure for that DAG can be described as follows (translated to C++):

   ```
   class face{
       int g; // grade
       List<face> facelist;  // List of children-faces of grade (g-1).
                             //    If g=1 then facelist is NULL
       Point p[2]; // if (g==1) then the cell is a finite segment
   }
   ```

   The points `p[0]` and `p[1]` are the leaves of the DAG. A Voronoi cell belonging to a generator point p_i is calculated by intersecting a 'bounding box' with all halfspaces that are given by the mid(hyper)planes of p_i with the other p_j's. In this algorithm CGA is only used to store the points (leaves of the DAG) and CGA is not used for the internal representation.

   In figure 7.1 an example of the memory structure of a filled polytope-DAG is given. It represents the polytope-DAG of a simple 3D-cube with vertices A-H.

2. The second algorithm produces the edges of a dD voronoi diagram by backtracking over all possible combinations of halfspace intersections that produce an edge. This algorithm uses a variant of the flag-datastructure in chapter 5 and also uses a C-variant of the `intersectHyperplane` implementation.

We investigated the possibility whether flags could be used in the internal representation of this DAG-datastructure. But this does not seem to give any advantage, since a flag represents just one path from top to bottom through the DAG. In the simple example of figure 7.1 $6 \times 3 \times 2 = 36$ flags can be made. A possible flag is the sequence

$$cube(ABCDEFGH), square(BCGF), square(CG), vertex(G).$$

When the dimension d and the number of points n are increased, the memory space also increases. In higher dimensions and when using more points, storing all flags will cost a lot of memory space. The number of flags can be calculated by the formula $n(d!)$ (for $n > d$).

A flag is not suited for representing a whole DAG of CGA-objects, but it can be used to represent the current state of the algorithm. For example, in the second algorithm of Zaharia, a flag is used to track the current state of the backtrack algorithm for constructing a dD Voronoi diagram. This algorithm was written in C and did not use any object oriented modeling.

For this project we extended the given flag-datastructure and intersection function to build the dD Voronoi diagram. We did not use a backtracking programming paradigm directly, but instead of that we used the CGAL-style iterator to keep track of the current state of the algorithm.

In our solution we provide an iterator that returns the finite and infinite edges the Voronoi diagram consists of. The iterator just needs to store its current state, so we do not have to store the whole polytope in memory. Besides that the iterator structure can be used to show the strength of CGA-flags.

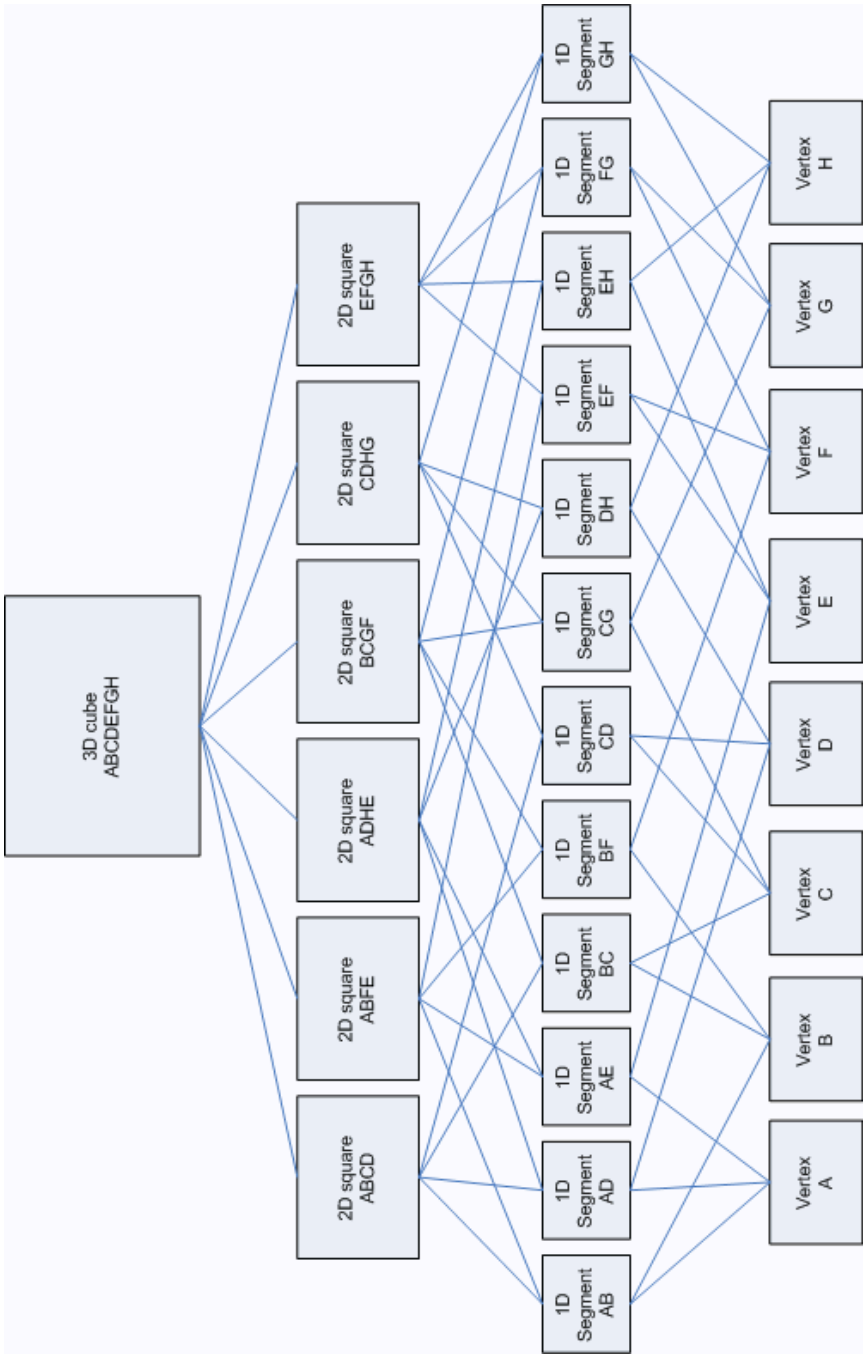


Figure 7.1: DAG-structure of a simple 3D-cube with vertices A-H

7.5 Conclusion

In this chapter we have shown how CGA-flags can be used in a d-dimensional Voronoi diagram algorithm. In this application, all important parts of this thesis come together. The Edelsbrunner algorithm was used for the implementation. In the implementation of the algorithm we used an iterator to sum up all finite and infinite line segments. The advantage of an iterator when compared to an internal polytope-datastructure, is that it is more memory efficient.

The advantages of this solution are:

- Using iterators in d-dimensional halfplane algorithms prevents memory problems.
- The CGA-flag is a very useful structure for the internal representation of the Voronoi iterator.
- The CGA-flag datastructure can handle finite and infinite line segments/rays in one.
- The flag contains CGA-objects, so all advantages of CGA -as mentioned in section 2.6- also yield for flags. (dimension independence, less programming error sensitive, more intuitive, etc.)

Comparison with the work of M. Zaharia

The idea of using flags for n-dimensional Voronoi diagrams in combination with the Edelsbrunner algorithm comes from a draft paper ([28]) of M. Zaharia of the University of Amsterdam. In this paper he mentions that the (in)finite edges could be found by a backtracking technique. He gives some formulas for calculating the tangent plane in Homogenous Geometric Algebra. And he introduces a `CUTFLAG` function that uses a flag to construct the upper envelope. However, this paper has not been finished. The most important adaptations, extensions and additions that were made in this project are the following:

- M. Zaharia mentions backtracking as a possibility to walk through the (in)finite edges. However, he does not work out this method at all. He only mentions some building bricks for an algorithm. In this thesis project we found out that using an iterator is an efficient, low-memory and CGAL-like way to walk over all the edges.
- We added some optimization techniques (extra booleans) to the flag structure, to speed up calculations
- Furthermore we mentioned in this thesis possibilites for future work (see section 7.6) and iterating through higher dimensional faces of the Voronoi diagram structure.
- In the Zaharia paper C-code is used to perform the calculations. In this project we used the object oriented C++ language and integrated it with CGAL and our own CGAP library.

7.6 Future work

Several additions could be made to expand the current algorithm:

- The current algorithm uses some pruning techniques to prevent the backtracking part of the algorithm from searching the whole combinations space. Further research could be done on where other the search trees could be cut off. The whole sequence of intersections within a flag can be intersected by one CGA-meet operation. By performing an even more smart backtracking and pruning method, many combinations can be excluded before they are really checked.

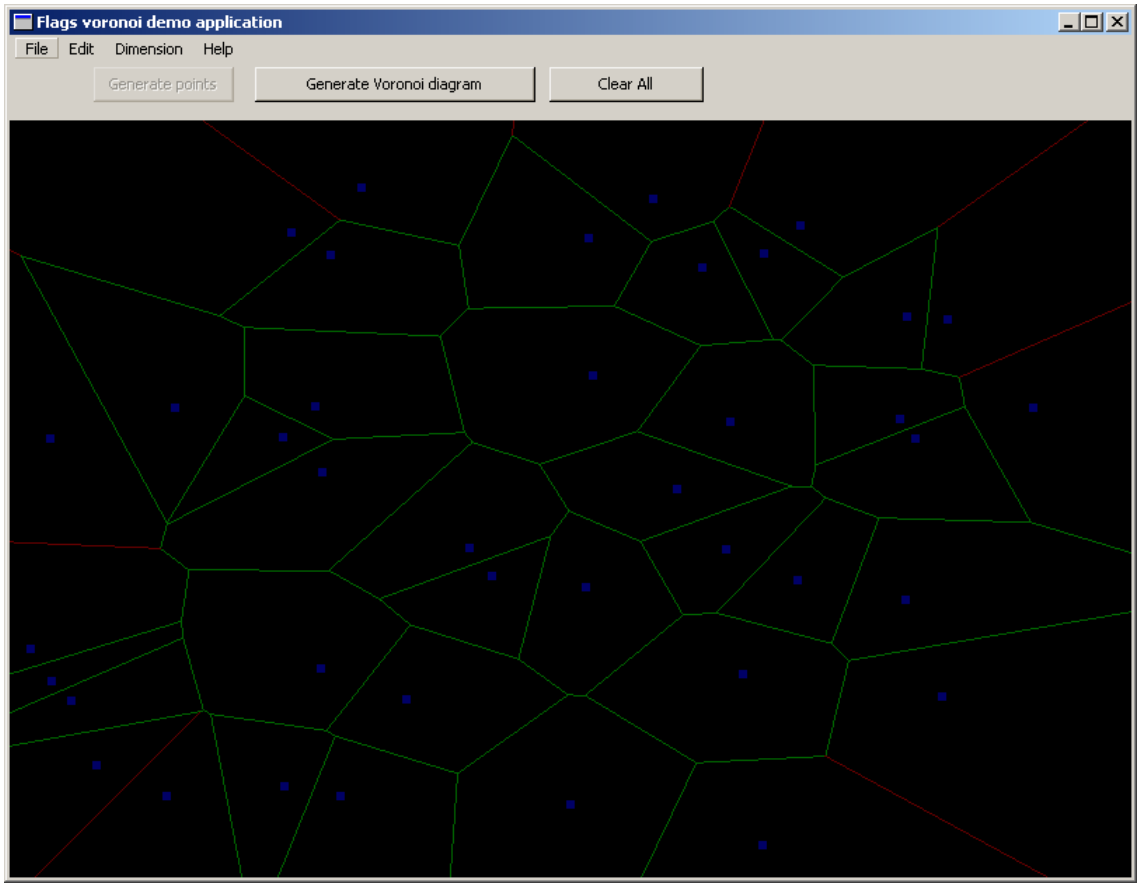


Figure 7.2: Screenshot of 2D Voronoi diagram application, made with CGAP-kernel and CGA-flags.

- This implemented Voronoi diagram algorithm works for dimensions 2D to 5D. This is due to the fact that the GAIGEN-code generator is limited to dimension 8. 2 dimensions are needed for the $\mathbf{e}_0$ and $\mathbf{e}_\infty$ component. Another dimension is needed for the lifting from $\mathbb{R}^d$ to $\mathbb{R}^{d+1}$. So it is limited to dimension 8-2-1=5. Using another CGA element type that supports higher dimensions, would reach this higher dimension.
- The technique to iterate over Voronoi edges, described in this thesis, can also be easily used to sum up the edges of a certain subspace / x-dimensional Voronoi face($x > 1$). For such a case, the initial flag should be set up to the subspace and the first (d)-combination should be set up in the right way: the first (d-x) elements of the (d)combination should be locked (i.e. may not be changed).
- With flags also a way can be found to iterate through higher dimensional faces of the Voronoi diagram. Not only 0-faces (vertices) and 1-faces ((in)finite edges) can be summed up, but also x-faces ($0 \leq x < d$).

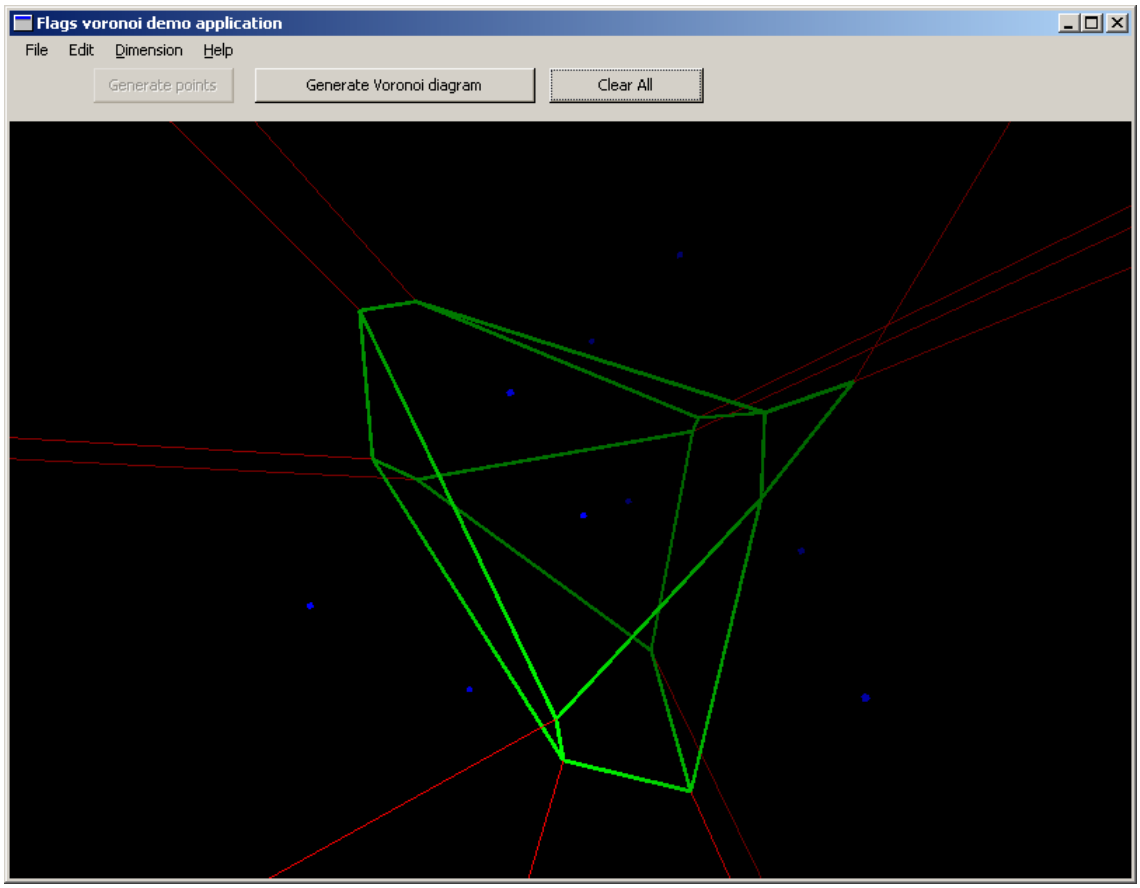


Figure 7.3: Screenshot of 3D Voronoi diagram application, made with CGAP-kernel and CGA-flags.

CHAPTER 8

Summary and Conclusion

In this thesis we discussed the modeling and implementation details of the CGAP-library, in which a CGA-kernel was built for CGAL. We worked out how CGA-flags can be used as iterators in d-dimensional Voronoi diagram algorithms. In the final sections of the previous chapters many conclusions and summaries have already been given. In this chapter we will repeat the most important issues.

The final result of the project, the CGAP-library, can become an important toolkit for both the CGAL-community and the CGA-community. CGAL-users can use CGAP as an extra kernel to implement new operations and algorithms. For CGA-users all algorithms of CGAL become available for CGA-use. Few CGA-algorithm libraries were available until now, but with CGAP a wide spectrum of CGAL-algorithms and output possibilities becomes available for the CGA-user. So the power of CGAP is that the advantages of both CGAL and CGA are combined.

The most important advantages of the CGAP-library are:

- It has been designed according to the design principles of CGAL. When a user is accustomed to STL and has a basic CGAL-knowledge, it is relatively simple to use the CGAP-kernel in CGAL-algorithms.
- Due to the strong expression power, the CGAP-code is often short and easy-to-understand.
- The dimension independence of CGA and the use of consistent formulas in CGA, make it possible that the same geometric objects are used for different dimensions (2D, 3D and dD).

In the final chapter of this thesis we have been showing that CGA-flags can be used for d-dimensional Voronoi diagram constructions. CGA-flags offer the advantages of CGA, and also offer the ability to store a whole nested sequence of geometric objects in one CGA-element. The CGA-flag is used within the iterator-structure of the Voronoi diagram. Both infinite and finite edges can be stored in it. Flags can be used for other polytope/infinite polyhedra algorithms as well.

Future work

Building a complete CGAL-kernel would take a few years. This would not fit within the time reserved for this masters thesis project. For this reason, the current CGAP-project has not been finished yet. Kernels, objects, predicates and constructions have been implemented for the algorithms Voronoi diagram, Delaunay triangulation and convex hull under certain conditions. It would be an easy task to extend the kernel for a certain algorithm. For working with CGAP a firm knowledge of both CGAL and CGA is required. The kernel framework has been developed and documented, so it would be relatively simple to extend this kernel.

Bibliography

[1] Bradford C. Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[2] CGAL Editorial Board. *CGAL 2D and 3D Kernel manual*, February 2004. Release 3.0.1, www.cgal.org.

[3] CGAL Editorial Board. *CGAL Basic Library User and Reference manual*, February 2004. Release 3.0.1, www.cgal.org.

[4] D. Brown. Voronoi diagrams from convex hulls. *Information processing Letters*, 9:223–228, 1979.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[6] L. Dorst. Classification and parametrization of blades in the conformal model of euclidean geometry. in preparation, will appear on `www.science.uva.nl/ga/`, 2003.

[7] L. Dorst and D. Fontijne. *3D Euclidean Geometry through Conformal Geometric Algebra (a GAViewer tutorial)*. University of Amsterdam, December 2003. www.science.uva.nl/ga/.

[8] L. Dorst and S. Mann. Geometric algebra: a computational framework for geometrical applications (part I: algebra). *IEEE Computer Graphics and Applications*, pages 24–31, May/June 2002.

[9] Herbert Edelsbrunner and Raimund Seidel. Voronoi diagrams and arrangements. In *SCG '85: Proceedings of the first annual symposium on Computational geometry*, pages 251–262, New York, NY, USA, 1985. ACM Press.

[10] Günter Ewald. *Combinatorial Convexity and Algebraic Geometry (Graduate Texts in Mathematics)*. Springer, October 1996.

[11] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL a computational geometry algorithms library. *Software Practice and Experience*, 30(11):1167–1202, 2000.

[12] S. Fortune. Computational geometry in directions in computational geometry, 1993.

[13] John B. Fraleigh and Raymond A. Beauregard. *Linear algebra*. Addison Wesley, third edition, 1995.

[14] Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Michael Seel. An adaptable and extensible geometry kernel. *Lecture Notes in Computer Science*, 2141:79–91, 2001.

[15] D. Hestenes. Old wine in new bottles: A new algebraic framework for computational geometry. In G.Sobczyk E. Bayro-Corrochano, editor, *Geometric Algebra with Applications in Science and Engineering*, pages 3–17, 2001.

[16] David Hestenes and Garret Sobczyk. *Clifford algebra to geometric calculus*. Kluwer Academic Publishers, Dordrecht, 1992. 1st. ed 1984, Reprinted with corrections.

[17] D. Hildenbrand, D. Fontijne, C. Perwass, and L. Dorst. Geometric algebra and its application to computer graphics. In *Tutorial 3*. 25th Annual Conference of the European Association for Computer Graphics "Interacting with Virtual Worlds" Grenoble, August 30th September 3rd, 2004.

[18] Dietmar Hildenbrand. Geometric computing in computer graphics using conformal geometric algebra. *Computers and Graphics*, 29(5):795–803, 2005.

[19] V. Klee. On the complexity of d-dimensional voronoi diagrams. *Archiv. Math.*, 34:75–80, 1980.

[20] Serge A. Lang. *Linear Algebra*. Springer-Verlag Telos, third edition, 1987.

[21] Hongbo Li, Peter J. Olver, and Gerald Sommer. *Computer Algebra and Geometric Algebra with Applications: 6th International Workshop, IWMM 2004, Shanghai, China, May 19-21, 2004 and International Workshop, ... Papers (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[22] P. Lounesto. *Clifford Algebras and Spinors*. Cambridge University Press, 2000.

[23] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial tessellations: concepts and applications of Voronoi diagrams*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[24] C. Perwass and D. Hildenbrand. Aspects of geometric algebra in euclidean, projective and conformal space. Technical Report Number 0310, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, September 2003.

[25] Gerald Sommer, editor. *Geometric computing with Clifford algebras: theoretical foundations and applications in computer vision and robotics*. Springer-Verlag, London, UK, 2001.

[26] R.J. Valkenburg and N.S. Alwesh. Calibration of target positions using the conformal model and geometric algebra. *Industrial Research Ltd.*, 2006.

[27] Rich Wareham, Joan Lasenby, and Anthony Lasenby. Computer graphics using conformal geometric algebra. 2004.

[28] M.D. Zaharia. Finding the nd voronoi diagram by geometric algebra means. unpublished. June 2003.

[29] Chaïm Zonnenberg. Het gebruik van geometric algebra's in computer vision. August 2004.

Index