ein automatisches geometrisches
Beweissystem

Georg Timo von Oertzen

6. September 1999

Diplomarbeit

---

nach einem Thema von Prof. Dr. Günter Hotz in der Technischen Fakultät, Fachbereich 14 – Informatik, der Universität des Saarlandes, Saarbrücken.

Hiermit versichere ich an Eides Statt, dass ich diese Arbeit nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Saarbrücken, 31.08.1999

Anmerkung

Das in dieser Arbeit beschriebene Programm *Cedric* steht über das WWW unter der Adresse http://www-hotz.cs.uni-sb.de/~oertzen zur Verfügung. Diese Arbeit wurde nach den Regeln der neuen Rechtschreibung erstellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

In den Ursprüngen der Mathematik wurden die Menschen, wie man sich vorstellen kann, weniger von den formalen Sätzen der Analysis oder der Algebra motiviert, sondern vielmehr von der Geometrie der Dinge, die sie umgaben. Die Mathematik entsprang dem Wunsch, Häuser und Tempel stabiler zu bauen, die Räder an den Wagen zu verbessern und - es läßt sich nun mal nicht abstreiten - die Flugbahn der Katapultsteine besser abschätzen zu können.

Die Mathematik, von je her eng verwandt mit der Philosophie und der Musik, entwickelte sich aber auch zu etwas Ästhetischem, das ihr heute nach wie vor innewohnt. Eine abgerundete Definition, ein eleganter Beweis oder ein nicht erwarteter Satz kann einen Mathematiker mit ähnlicher Freude erfüllen wie die Betrachtung eines kunstvollen Gemäldes.

Die Geometrie nimmt dabei in der Mathematik die Rolle ein, die in der Musik das Instrumentenspiel ist. Obwohl heute die Mathematik weit über die Geometrie hinausgewachsen ist, fühlen doch viele Menschen (und besonders Schüler), die sonst keine große Freude an der Mathematik haben, sich mit der Geometrie noch am ehesten verbunden. Es ist wohl anzunehmen, dass diese Tatsache ein Rest des Ursprunges der Mathematik ist, ein intuitiver Wunsch, die Geometrie der Welt um uns herum besser zu verstehen.

Das in dieser Arbeit vorgestellte Programm versucht, dieses Interesse zu wecken und darauf aufzubauen. Durch die hohe Rechengeschwindigkeit der heutigen Computer ist es möglich, die Wahrheit auch komplizierterer Sätze unmittelbar begreifbar zu machen. So kann man mit Hilfe von *Cedric* geometrische Sätze auf dem Bildschirm darstellen und ohne viel Zeitaufwand modifizieren oder mit ihnen spielen. Dadurch wird es zum Beispiel möglich, Schülern den Einblick in die Geometrie rein anschaulich näher zu bringen. Es erübrigt sich, den selben geometrischen Satz mehrmals zu zeichnen, um seine Richtigkeit zu erkennen; erstens kann man eine Konstruktion in Cedric beliebig verschieben und bewegen (so dass didaktisch kein Bedarf mehr besteht, eine Aussage glaubwürdig zu machen), und zweitens kann man mit Cedric automatisch Beweise oder Verifikationen einer Aussage anfertigen lassen. Dieser letzte Punkt soll natürlich auch dazu motivieren, mit Punkten, Geraden und Kreisen herumzuexperimentieren und neue Sätze in der Geometrie zu finden.

Die Komplexität eines geometrischen Beweises steigt exponentiell in der Anzahl der am Beweis beteiligten Objekte. Dennoch ist Cedric als einer der schnellsten geometrischen Beweiser in der Lage, sehr viele geometrische Sätze innerhalb einer Sekunde vollständig zu beweisen. Es hat den Anschein, als würde sich der menschliche Umgang mit der Geometrie die Ursprünglichkeit bewahren, aus der einst die Mathematik entsprungen ist.

1.2 Geschichte des geometrischen Beweisens

Die ersten Grundlagen für ein automatisches Beweisen von geometrischen Sätzen ist die Erfassung der Geometrie in exakten formalen Darstellungen. Diese entwickelten sich natürlich seit Euklid (ca. 300 v.Chr.) in langsam ansteigendem Maße. Eine vollständige axiomatische Erfassung der Geometrie kam aber erst von D. Hilbert in seinem Buch 'Grundlagen der Geometrie' [6].

Die Idee, geometrische Beweise algorithmisch zu finden, ist dann fast eine notwendige Folgerung dieser Entwicklung. W. T. Wu berichtet in [7], dass auch hier Hilbert den Anstoß gab. Er stellte eine Methode vor, mit der man eine Teilklasse von geometrischen Sätzen, die sogenannten reinen Schnittpunktsätze, beweisen kann.

Einen weiteren Schritt in diese Richtung wurde von H.Gelernter im Jahre 1959 unternommen, der das erste mal eine praktisch anwendbare Maschine zum Beweisen geometrischer Sätze lieferte [1]. Dieser Beweiser, die sogenannte *Geometry Machine*, funktionierte auf der Basis der Logik. Es wurden formale, axiomatische Grundlagen der Geometrie vorausgesetzt und dann Beweise als logische Folgerung der Axiome bewiesen, wie es heute bei einigen Beweisern auf anderen Gebieten immer noch gemacht wird.

Die mit der Logik verbundene Allgemeinheit stellt dabei natürlich ein mächtiges Werkzeug dar (dasselbe Prinzip kann man für Beweiser aller Art benutzen). Allerdings versagten diese Beweiser, wenn kompliziertere geometrische Sätze gezeigt werden sollten. Man gewinnt den Eindruck, dass die Allgemeinheit der Logik in abgegrenzten Gebieten nicht immer von Vorteil ist.

Unter diesem Eindruck entstand die von Wu [7] zuerst erdachte Methode, die Unbekannten einer Konstruktion in Form von Polynomen weiterzureichen und die Gültigkeit von Behauptungen dann in den Quotientenkörper über dem Polynomring zu überprüfen. Gilt dabei eine Aussage in dem Quotientenkörper, so kann auf diese Weise auch eine Aussage für alle Einsetzungen gemacht werden (am stärksten dann, wenn am Ende einer Rechnung ein konstantes Polynom, etwa das Nullpolynom, auftritt; man kann dann aussagen, dass diese Konstante das Ergebnis der Rechnung auch im Grundkörper des Polynomringes sein wird, und zwar unter allen Einsetzungen).

Ein weiterer geometrischer Beweiser, das System *GEO-Prover* von Chou [8] verdient in diesem Zusammenhang ebenfalls Erwähnung. Es war eine weitere, konsequente Fortsetzung des Beweisprinzips mit Hilfe von Polynomen und konnte eine große Zahl von geometrischen Sätzen beweisen.

Aufgrund der heute möglichen Rechenzeiten, aber auch aufgrund immer neuer Überarbeitung und Verbesserung der Methoden, sind heutige geometrische Beweiser weitaus effektiver als die Programme aus der Pionierzeit des automatischen Beweisens.

Mit Hilfe von modernen Algebrasystemen wie beispielsweise Maple lassen sich geometrische Beweise sozusagen aus externen Quellen beziehen, und es genügt, eine Schnittstelle zum Benutzer einzurichten. Eine solche Schnittstelle stellt zum Beispiel die Arbeit von Eva Rohnert [9] dar, die auch mit Blick auf die Verwendung in Schulen konstruiert war.

Auch der *Geometry Expert* [10] ist ein leistungsfähiger Beweiser für geometrische Sätze, in dem allerdings die Eingabe einzelner Sätze (etwa des Vierecksatzes, siehe Anhang) nicht möglich ist.

Darüber hinaus arbeiten diese Beweiser ausschließlich lokal und können nicht über das WWW genutzt werden. Hier stellt das an der ETH Zürich entwickelte System Cinderella [11] eine echte Neuheit dar; allerdings ist Cinderella nicht im eigentlichen Sinne ein geometrischer Beweiser, da keine exakten Beweise durchgeführt werden, sondern nur automatisch einzelne Einsetzungen in die Konstruktion gemacht werden und dann Aussagen gesucht werden. In Cinderella ist es nicht möglich, eine spezielle Anfrage direkt zu stellen; das System ist eher als Präsentationsmedium entwickelt und hat eine entsprechend sehenswerte Oberfläche.

Cedric ist als Java-Applet ebenfalls auf dem WWW lauffähig. Trotz der mit dieser Sprache verbundenen Verluste an Laufzeit führt Cedric die Beweise unabhängig von externen Programmen und exakt durch (wobei auf Wunsch auch eine nicht beweiskräftige Überprüfung des Satzes durch Einsetzen einzelner Werte in die Konstruktion gemacht werden kann). Die Laufzeiten einiger Sätze von Cedric finden sich im Anhang dieser Arbeit.

1.3 Gliederung

In der vorliegenden Arbeit soll das Beweissystem Cedric nahezu vollständig erläutert werden. Dabei soll nur auf einige, triviale Implementierungen verzichtet werden.

Im Mittelpunkt des wissenschaftlichen Interesses steht natürlich der Beweiser. Hierfür werden in Kapitel 2 zunächst die für Cedric relevanten Definitionen für geometrisches Beweisen vorgestellt und insbesondere die Begriffe Konstruktionen, Anfragen und geometrische Sätze geklärt. Ein weiterer, wichtiger Abschnitt des Kapitels 2, der Abschnitt 2.5, beschäftigt sich mit den aus der projektiven Geometrie stammenden homogenen Koordinaten, mit denen viele Konstruktionen in Cedric beschleunigt werden konnten. Dabei wird auf eine vollständige Erläuterung der projektiven Geometrie verzichtet; interessierte Leser seien auf das Buch *Analytische Geometrie* von Gerd Fischer [2] verwiesen. Im Rahmen dieser Arbeit werden uns nur die homogenen Koordinaten und das Dualitätsprinzip weiter interessieren.

In Kapitel 3 werden dann die implementierten Klassen beschrieben, mit denen die für geometrisches Beweisen wesentlichen Ausdrücke mit mehreren Unbekannten manipuliert und aufgelöst werden können. Dabei werden vornehmlich die inhaltlichen Konzepte dieser Klassen zum Ausdruck kommen, während auf die Angabe von unmittelbaren Quellcode naheliegenderweise verzichtet wird. Die Namen der Klassen und einiger ausgewählter Methoden werden genannt, so dass ein Nachschlagen im Programm ohne viel Zeitaufwand möglich sein sollte.

In Kapitel 4 wird dann näher darauf eingegangen, wie die in Kapitel 2 beschriebenen Elemente im Beweiser umgesetzt sind. Hier werden zunächst einige vorbereitende Tricks erläutert, die das geometrische Beweisen erleichtern. Es muss dabei immer im Gedächnis behalten werden, was für eine Bedeutung die Laufzeit für automatisches Beweisen innehat: Je schneller das Beweisen funktioniert, desto mehr Sätze können bearbeitet werden.

Im Anschluss werden alle wesentlichen Methoden des Beweisers, die sich im Quellcode in der Klasse *Vektor* wiederfinden, näher erläutert. Es ist Ziel dieser Arbeit, dass anhand dieser Kapitel die wesentlichen Inhalte soweit vermittelt sind, dass jedem Leser eine Neuprogrammierung des Beweisers gelingen würde. In den Abschnittsnamen ist jeweils der Name der wesentlichen Kapitel mit aufgenommen, so dass auch hier ein Seitenblick in das Programm möglich bleibt.

In Kapitel 5 wird die Ebene des reinen Beweisers verlassen und ein näherer Blick auf die Schnittstelle zum Benutzer geworfen. Es besteht dabei eine scharfe Trennung zwischen dem Beweiser und den anderen Teilen des Programmes, die hier noch einmal deutlich wird. Hier tauchen die in Kapitel 2 bereits eingeführten Elemente erneut auf, wobei der Inhalt der Implementierung an dieser Stelle bereits durch die Komplexität der entsprechenden parallelen Methoden des Beweiser subsummiert wird, weshalb auf eine zweite Ausführung an dieser Stelle verzichtet wird.

Während der Beweiser nur auf der Basis von Punkten arbeitet, kommen im Darstellungsteil auch andere Objekte explizit vor. Hierbei sind besonders die in Abschnitt 5.1.4 beschriebenen Kegelschnitte von Interesse, die aufgrund ihrer Komplexität teilweise nicht vom Beweiser verarbeitet werden können.

In Kapitel 6 wird dann eine Reihe von interessanten, zusätzlichen Dingen näher erläutert, die inhaltlich weder dem Beweiser noch der Konstruktionsdarstellung auf dem Monitor direkt zuzuschreiben sind. In diesem Kapitel findet sich zunächst der Abschnitt 6.1 über den Gleichungseditor, in dem Syntax und Umsetzung der Gleichungen näher erläutert wird, mit dem Abfragen auf Streckenlängen gemacht werden können.

Im Abschnitt 6.2 geht es dann um das Format der linearen Mitschrift der Konstruktion, die während der Konstruktion beobachtet werden kann und auch in externen Dateien selbst geschrieben werden kann; in diesem Abschnitt ist beschrieben, wie diese Konstruktion dann von Cedric auf dem Bildschirm sichtbar gemacht wird.

Daran schließt sich unmittelbar der Abschnitt 6.3 über das Dualisieren an, indem die Funktionsweise, aber auch die Einschränkungen dieser Methode näher beschrieben wird.

Ein wesentlicher Teil des Programmes ist der letzte Abschnitt dieses Kapitels, der Abschnitt 6.4 über den Validerer, mit dessen Hilfe geometrische Sätze in wesentlich kürzerer Zeit als mit dem Beweiser überprüft werden können. Dabei wird hier auf einen exakten Beweis verzichtet und statt dessen einfach nur einige Punktkonstellationen ausprobiert, um zu überprüfen, ob Gegenbeispiele gefunden werden können; ist dies der Fall, so kann man sicher von der Falschheit eines Satzes ausgehen, bei genügend häufigen Versuchen wiederum ist ein Ausbleiben von Gegenbeispielen ein starker Indikator für die Wahrheit eines Satzes.

Um ein Verständnis für Cedric zu bekommen und Spaß an der Sache zu haben, ist es natürlich unumgänglich, sich mit dem Umgang des Programmes vertraut zu machen. Aus diesem Grunde beginnt der Anhang dieser Arbeit mit einer kurzen, handbuchartigen Erklärung der Funktionsweise von Cedric. Außerdem finden sich im Anhang eine größere Auswahl von Beispielsätzen, die zum Ausprobieren von Cedric einladen. Eine tabellarische Auflistung der Laufzeiten dieser Sätze soll ein ungefähres Gefühl vermitteln, wie komplex bestimmte Konstruktionen für Cedric sind.

Einer großen Anzahl von Menschen muss ich für ihre Hilfe an dieser Arbeit danken. Da es sich dabei zumeist um fruchtbringende Diskussionen, Anregungen oder fachlichen Austausch handelt, gehört die Danksagung eng zum Literaturverzeichnis und befindet sich im Anhang dieser Arbeit.

Kapitel 2

Formale Grundlagen

2.1 Definition von Konstruktionselementen

Diese Arbeit beschäftigt sich mit den geometrischen Sätzen, die mit Cedric implementiert und bewiesen werden können. Dies motiviert einige spezielle Definitionen, was in diesem Zusammenhang unter Konstruktionen, Anfragen und Sätzen zu verstehen ist; diese Definitionen sind dabei an die Implementierung von Cedric angelehnt. Insbesondere bemerke man, dass diese Definitionen streng punktorientiert sind, d.h. alle Geraden, Kreise oder andere geometrische Objekte verstehen sich nur als Menge von Punkten.

Für eine formale Betrachtung soll zunächst definiert werden, was unter Punkten und geometrischen Objekten zu verstehen ist. Dabei ist die Definition der Punkte noch verhältnismäßig einfach:

Definition

Sei K ein Körper und V der Vektorraum K^2. Dann heißt jedes Element aus V *Punkt über K*.

Als nächstes wollen wir definieren, wie die Elemente aufgebaut sind, aus denen geometrische Konstruktionen zusammengesetzt werden. Dabei handelt es sich um Abbildungen, die von einer Menge von Punkten und u.U. zusätzlichen Elementen des Körpers jeweils einen neuen Punkt erzeugen. Ein Beispiel wäre das Erzeugen der Mitte einer Strecke: Hierfür benötigen wir eine Abbildung, die zwei Punkte (die beiden Randpunkte der Strecke) auf den Mittelpunkt abbildet.

Allgemein sollen die Konstruktionselemente folgende Form haben:

Definition

Sei K ein Körper und V der Vektorraum K^2. Ein *Konstruktionselement ϕ über K* ist eine Abbildung $\phi : K^k \times V^v \to V, \qquad k \in \{0, 1, 2\}$ und $v \in \mathbb{N}$.

Ist $k = 2$, so heißt das Konstruktionselement *frei*, ist $k = 1$, so heißt es *semifrei*, und ist $k = 0$, so heißt das Konstruktionselement *unfrei*.
Die k Argumente aus K heißen die *freien Variablen des Konstruktionselements*.

Beispiele

Diese Definition erscheint im ersten Moment nicht allzu kompliziert, erweist sich aber als vielseitiger, als man zuerst denkt. Betrachten wir zum besseren Verständnis folgende drei Beispiele:

- Sei $k = 1$, $v = 2$ und $\phi\Big((\lambda), (X, Y)\Big) = X + \lambda * (Y - X)$. Dieses Konstruktionselement ist ein semifreier Punkt auf der Verbindungsgeraden $\overline{XY}$.

- Sei $k = 2$, $v = 0$ und $\phi\Big((\lambda, \mu), ()\Big) = \lambda * (1, 0) + \mu * (0, 1)$. Dieses Konstruktionselement ist ein beliebiger freier Punkt im Vektorraum.

- Feste Punkte lassen sich ebenfalls durch Konstruktionselemente darstellen: Sei $k = 0$, $v = 0$, $\phi\Big(()()\Big) = (0, 0) \in V$. Dieses Konstruktionselement stellt den Ursprungspunkt des Koordinatensystems dar.

Jetzt wollen wir definieren, was wir unter einer Konstruktion verstehen. Zuerst einmal besteht eine Konstruktion natürlich aus einer Folge von Konstruktionselementen; hinzu kommt dann noch für jedes Konstruktionselement ϕ eine zusätzliche Information, die den Zusammenhang von ϕ zu vorangegangenen Konstruktionselementen beschreibt.

Für jedes Argument von ϕ aus V ist hierfür eine natürliche Zahl angegeben, die eine Position in der Konstruktion beschreibt. Das Ergebnis des Konstruktionselementes an dieser Position ist dann das entsprechende Argument für ϕ.

Formal notiert erhalten wir folgende Definition einer Konstruktion:

Definition

Sei $n \in \mathbb{N}$, $\phi_1, \ldots, \phi_n$ Konstruktionselemente über einem Körper K, v_i die Stelligkeit von ϕ_i in Argumenten aus V. Sei weiterhin $arg_i \in (\mathbb{N}\backslash\{0\})^{v_i}$ für $i = 1, \ldots, n$, wobei für jede Zahl aus der Folge arg_i gelten soll: $arg_{ij} < i \quad \forall i, j$. Dann heißen die beiden Folgen $(\phi_1, \ldots, \phi_n)$ und $(arg_1, \ldots, arg_n)$ zusammen eine *Konstruktion über K*.
Seien k_i die Anzahl der freien Variablen von ϕ_i. Dann ist $k := \sum_{i=1}^{n} k_i$ die Anzahl der freien Variablen der Konstruktion.

Satz

Für jede Konstruktion ist $v_1 = 0$, d.h jede Konstruktion beginnt mit einem Punkt.

Beweis

Da jedes Element $arg_{ij} < i$ sein soll, muss insbesondere jedes Element von $arg_1 < 1$ sein. Da die Elemente von arg_i aber aus $\mathbb{N}\backslash\{0\}$ stammen ist dies nur möglich, wenn die Folge arg_1 leer ist, also $v_1 = 0$.

□

2.2 Definition von Anfragen

Mit den Definitionen des letzten Kapitels können wir nun jede geometrische Konstruktion über dem Körper K erfassen. Jetzt interessieren uns Anfragen auf die Konstruktion, zum Beispiel ob etwa drei bestimmte Punkte auf einer Geraden liegen. Dafür definieren wir hier formal eine geometrische Anfrage, die uns für vorgegebene Punkte sagt, ob diese eine gestellte Bedingung erfüllen oder nicht:

Definition

Sei $\mathbb{B} = \{wahr, falsch\}$, K ein Körper und V der Vektorraum K^2. Eine *Anfrage* ψ ist eine Abbildung $\psi : V^v \to \mathbb{B}$ mit $v \in \mathbb{N}$.

Beispiele

Betrachten wir zur Veranschaulichung wieder zwei Beispiele, ein Trivialbeispiel und eine von den Anfragen, die in Cedric implementiert ist.

- Wir wollen wissen, ob zwei vorgegebene Punkte identisch sind. Dann ist $v = 2$ und

$$\psi(X, Y) = \begin{cases} wahr & (X = Y) \\ falsch & (X \neq Y) \end{cases}$$

- Eine gebräuchliche Abfrage ist, ob drei Punkte auf einer Geraden liegen. Man sagt dann, die drei Punkte seien *kollinear*. Hierfür brauchen wir also $v = 3$ Argumente und eine Anfrage der Form

$$\psi(X, Y, Z) = \begin{cases} wahr & \exists \lambda \in K : \quad Z = X + \lambda * (Y - X) \\ falsch & \text{sonst} \end{cases}$$

2.3 Definition von geometrischen Sätzen

Mit Konstruktionen und Anfragen können wir jetzt auch geometrische Sätze definieren. Sätze werden wir als 3-Tupel definieren, und zwar sinngemäß die betrachtete Konstruktion, eine Anfrage und Verweise, auf welche aus den Konstruktionselementen hervorgegangenen Punkte sich die Anfrage bezieht.

Zunächst müssen wir die Konstruktion mit Bedeutung erfüllen, d.h. wir müssen alle freien Variablen der Konstruktion mit Werten aus dem Körper K belegen. Dafür definieren wir für jede Konstruktion C eine zugehörige Belegungsfunktion, die für jede freie Variable ein Körperelement einsetzt. Wenn v_C die Anzahl der in C frei vorkommenden Variablen bezeichnet, dann soll diese Belegungsfunktion für jedes Argument aus K^{v_C} ein n-Tupel von Punkten aus V liefern, die sich durch Ausführen der Konstruktion mit den Variablen des Argumentes ergibt.

Haben wir zum Beispiel eine Konstruktion, die zwei freie Punkte und deren Mittelpunkt enthält, so ist die zugehörige Funktion auf K^4 definiert und liefert uns für jede Belegung die zwei Punkte mit den jeweiligen Koordinaten und den Mittelpunkt zurück. Es folgt jetzt zunächst die formale Definition, die im ersten Augenblick verhältnismäßig kompliziert wirkt; danach werde ich eben genanntes Beispiel noch einmal ausführen.

Definition

Sei $C = \Big((\phi_1, \ldots, \phi_n)\,,\,(arg_1, \ldots, arg_n)\Big)$ eine Konstruktion über einem Körper K und v_C die Anzahl der freien Variablen in C. Dann ist die *zu C gehörende Belegungsfunktion* Ψ eine Funktion $\Psi : K^{v_C} \to V^n$, die den Werten $(t_1, \ldots, t_n) \in K^{v_C}$ ein n-Tupel $(p_1, \ldots, p_n)$ von Punkten aus V zuordnet, die induktiv definiert sind durch

$$p_i := \phi_i\Big((t_{i1}, ..., t_{ik_i})(p_{arg_{i1}}, \ldots, p_{arg_{iv_i}})\Big)$$

Beispiel

Betrachten wir zum Beispiel den Körper der rationalen Zahlen und die eben genannte Konstruktion aus drei Konstruktionselementen, von denen die ersten zwei freie Punkte darstellen, und das dritte den Mittelpunkt der dadurch definierten Strecke. Die ersten beiden Konstruktionselemente haben jeweils zwei freie Variablen (zwei Argumente aus dem Körper), während das dritte Konstruktionselement keine freien Variablen hat. Es ist also $v_C = 4$. Die Funktion Ψ ist also auf $\mathbb{Q}^4$ definiert und liefert ein Element aus V^3 zurück. Es ist zum Beispiel:

$$\Psi(2,2,4,4) = \Big((2,2)\,,\,(4,4)\,,\,(3,3)\Big)$$

Für die Benennung eines einzelnen Punktes des Ergebnistupels ist es eine übliche Notation, die Position des Tupels als Index anzufügen. Ist also t irgendein Argument der Funktion (also in unserem Fall $t \in K^4$), so bezeichnet $(\Psi(t))_i$ den i-ten Ergebnispunkt. In diesem Beispiel ist also $(\Psi(2,2,4,4))_1 = (2,2)$, $(\Psi(2,2,4,4))_2 = (4,4)$ und $(\Psi(2,2,4,4))_3 = (3,3)$.

Bemerkung

Der induktiven Definition der Ergebniselemente der hier dargestellten Belegungsfunktion scheint im ersten Moment ein Induktionsanfang zu fehlen. Man rufe sich aber ins Gedächnis, dass in der Konstruktion nur Verweise auf vorangegangene Konstruktionselemente gemacht werden, und dass insbesondere das erste Konstruktionselement, also der Induktionsanfang, ein Punkt mit $v_1 = 0$ ist.

Damit gelangen wir also zur Definition eines geometrischen Satzes. Hierfür benötigen wir zunächst eine Konstruktion und eine daran angehängte Anfrage. Die Belegungsfunktion läßt sich auf die eben definierte Weise eindeutig aus der Konstruktion bestimmen. Um dann zu überprüfen, ob die Anfrage auf die Konstruktion wahr ist, benötigen wir natürlich noch die Information, auf welche

Punkte der Konstruktion sich die Anfrage bezieht. Hierfür benutzen wir für jedes Argument der Anfrage eine natürliche Zahl, die jeweils auf die Position des zugehörigen Argumentes in der Konstruktion verweist. Dieses Tupel von natürlichen Zahlen wird in der nun folgenden Definition als arg_{n+1} bezeichnet.

Definition

Sei $C = \Big((\phi_1, \ldots, \phi_n)\,, (arg_1, \ldots, arg_n) \Big)$ eine Konstruktion über einem Körper K. Sei ψ eine Anfrage mit Stelligkeit v_{n+1} und arg_{n+1} das zugehörige Tupel von Verweisen auf die Konstruktion, wobei $arg_{n+1} \in (\mathbb{N}\backslash\{0\})^{v_{n+1}}$ und $arg_{n+1,i} \leq n \quad \forall i$. Dann heißt das Tupel $\Big((\phi_1, \ldots, \phi_n, \psi)\,, (arg_1, \ldots, arg_n, arg_{n+1}) \Big)$ ein *geometrischer Satz über K*.

Definition

Sei $\Big((\phi_1, \ldots, \phi_n, \psi)\,, (arg_1, \ldots, arg_n, arg_{n+1}) \Big)$ ein Satz über K und Ψ die zu $C = \Big((\phi_1, \ldots, \phi_n)\,, (arg_1, \ldots, arg_n) \Big)$ gehörende Belegungsfunktion mit der Stelligkeit k. Sei weiterhin v_{n+1} die Stelligkeit der Funktion ψ. Der geometrische Satz heißt *wahr* $\Longleftrightarrow^{Def.}$

$$\forall t \in K^k : \psi(\, (\Psi(t))_{arg_{n+1,1}}, \ldots, (\Psi(t))_{arg_{n+1,v_{n+1}}}) = wahr$$

Beispiel

Setzten wir unser Beispiel für eine Konstruktion aus 2.2 zu einem geometrischen Satz fort, indem wir zum Beispiel eine dreistellige Anfrage hinzufügen, ob die drei Punkte auf einer Geraden liegen. Diese Anfrage ist also durch eine dreistellige Funktion ψ gegeben, die wahr zurückliefert, falls die drei Argumentpunkte auf einer Geraden liegen. Für die Argumente müssen wir noch die nötigen Verweise angeben, die in unserem Falle auf die drei Konstruktionselemente verweisen würde. Der Satz ist jetzt wahr, falls für alle Einsetzungen für die Funktion Ψ das Ergebnis drei Punkte sind, die auf einer Geraden liegen.

2.4 Spezielle Konstruktionselemente und Anfragen

In diesem Abschnitt werden einige Beispiele von Konstruktionselementen und Abfragen definiert, die in Cedric Verwendung finden. Die Darstellung dieser Elemente ist bewusst intuitiv gehalten, so dass nachvollzogen werden kann, was die einzelnen Funktionen bewirken; für die Auswertung mit einem Computer sind die hier angegebenen Definitionen nicht immer geeignet. Die Implementation im automatischen Beweiser wird in Kapitel 4 dann ausführlich behandelt.

2.4.1 In Cedric vorkommende Konstruktionselemente

Sei K wieder ein Körper und V der Vektorraum K^2. Folgende Konstruktionselemente über K mit jeweils v Argumenten aus V und k Elementen aus K sind für Cedric von Bedeutung:

- Freier Punkt: $(k = 2, v = 0)$
  $\mathit{freierPunkt}(\lambda, \mu)()$ = Punkt mit den Koordinaten (λ, μ).
- Punkt auf Gerade: $(k = 1, v = 2)$
  $\mathit{aufGerade}(\lambda)(a, b)$ = beliebiger Punkt auf der Geraden $\overline{ab}$.
- Punkt auf Kreis: $(k = 1, v = 2)$
  $\mathit{auf\ Kreis}(\lambda)(M, R)$ = beliebiger Punkt auf dem Kreis um M, wobei R ein Punkt auf dem Umfang des Kreises ist.
- Schnitt zweier Geraden: $(k = 0, v = 4)$
  $\mathit{schnittGeradeGerade}()(A, B, C, D)$ = Der Schnitt von $\overline{AB}$ und $\overline{CD}$
- Mittelpunkt des Umkreises: $(k = 0, v = 3)$
  $\mathit{kreismittelpunkt}()(A, B, C)$ = Mittelpunkt des Umkreises um A,B und C.
- Streckenmittelpunkt: $(k = 0, v = 2)$
  $\mathit{streckenmittelpunkt}()(A, B)$ = Mittelpunkt der beiden Punkte A und B.
- Quadratergänzung: $(k = 0, v = 2)$
  $\mathit{rechtwinklig}()(A, B)$ = Punkt auf der Senkrechten zu $\overline{AB}$ durch B, wobei der Abstand des neuen Punktes zu B gleich dem Abstand von A und B ist. Der entstehende Winkel beträgt dabei 90° gegen den Uhrzeigersinn.
- Punkt auf Mittelsenkrechter und Thaleskreis: $(k = 0, v = 2)$
  $\mathit{schnittThalesMittelsenkrechte}()(A, B)$ = ein Punk auf der Mittelsenkrechten von $\overline{AB}$ im Abstand $\frac{1}{2}\overline{AB}$ zu $\overline{AB}$. Der Winkel am Fusspunkt beträgt dabei wieder 90° gegen den Uhrzeigersinn.
- Parallelogrammergänzungspunkt: $(k = 0, v = 2)$
  $\mathit{parallelogrammpunkt}()(A, B, C)$ = Punkt auf der Parallelen zu $\overline{AB}$ durch C, so dass der Abstand des neuen Punktes zu C gleich $\overline{AB}$ ist.
- Erster Schnitt zweier Kreise: $(k = 0, v = 4)$
  $\mathit{schnittKreisKreisRechts}()(M_1, R_1, M_2, R_2)$ = Einer der beiden Schnittpunkte der Kreise um M_1 bzw. um M_2, jeweils mit R_1 bzw. R_2 auf dem Umfang.
- Zweiter Schnitt zweier Kreise: $(k = 0, v = 4)$
  $\mathit{schnittKreisKreisLinks}()(M_1, R_1, M_2, R_2)$ = Der andere Schnittpunkt der beiden Kreise.
- Erster Schnitt eines Kreises mit einer Geraden: $(k = 0, v = 4)$
  $\mathit{schnittGeradeKreisRechts}()(M, R, A, B)$ = Einer der beiden Schnittpunkte des Kreises um M mit R auf dem Umfang und der Geraden $\overline{AB}$.
- Zweiter Schnitt eines Kreises mit einer Geraden: $(k = 0, v = 4)$
  $\mathit{schnittGeradeKreisLinks}()(M, R, A, B)$ = Der andere Schnittpunkt des Kreises um M mit R auf dem Umfang und der Geraden $\overline{AB}$.

2.4.2 In Cedric vorkommende Anfragen

Folgende Anfragen über K mit v Argumenten aus V müssen weiterhin noch definiert werden:

- Drei Punkte liegen auf einer Geraden: $(v = 3)$
  $$kollinear(A, B, C) = \begin{cases} wahr & A, B \text{ und } C \text{ liegen auf einer Geraden} \\ falsch & \text{sonst} \end{cases}$$

- Drei Geraden schneiden sich in einem Punkt: $(v = 6)$
  $$GeradenSchnitt(A, B, C, D, E, F) = \begin{cases} wahr & \text{Die Geraden } \overline{AB},\ \overline{CD} \text{ und } \overline{EF} \text{ schneiden sich in einem Punkt} \\ falsch & \text{sonst} \end{cases}$$

- Punkte liegen auf einem Kreis: $(v = 4)$
  $$aufEinemKreis(A, B, C, D) = \begin{cases} wahr & \text{Die vier Punkte } A, B, C, D \text{ liegen auf einem Kreis} \\ falsch & \text{sonst} \end{cases}$$

- Geraden liegen senkrecht: $(v = 4)$
  $$geradenSenkrecht(A, B, C, D) = \begin{cases} wahr & \text{Die Geraden } \overline{AB} \text{ und } \overline{CD} \text{ liegen senkrecht zueinander} \\ falsch & \text{sonst} \end{cases}$$

- Geraden liegen parallel: $(v = 4)$
  $$geradenParallel(A, B, C, D) = \begin{cases} wahr & \text{Die Geraden } \overline{AB} \text{ und } \overline{CD} \text{ liegen parallel zueinander} \\ falsch & \text{sonst} \end{cases}$$

- Winkel sind Gleich: $(v = 6)$
  $$winkelGleich(A, B, C, D, E, F) = \begin{cases} wahr & \text{Die Winkel } ABC \text{ und } DEF \text{ haben den gleichen Cosinus} \\ falsch & \text{sonst} \end{cases}$$

- Kreise treffen sich: $(v = 6)$
  $$KreisSchnitt(M_1, R_1, M_2, R_2, M_3, R_3) = \begin{cases} wahr & \text{Die drei Kreise treffen sich in einem Punkt} \\ falsch & \text{sonst} \end{cases}$$

2.5 Berechnungen in homogenen Koordinaten

Der zweidimensionale Vektorraum, mit dem sich Cedric beschäftigt, läßt sich auf zwei wesentlich unterschiedliche Arten darstellen. Zum einen können wir die klassischen, kartesischen Koordinaten verwenden, in der jeweils zwei Zahlen des Grundkörpers die Position eines Punktes bestimmen. An einigen Stellen erweist es sich für die Laufzeit aber als günstiger, die in der projektiven Geometrie verwendeten homogenen Koordinaten zu nutzen.

Die zweidimensionale Ebene, die durch die homogenen Koordinaten dargestellt werden soll, ist bei dieser Darstellung in einen dreidimensionalen Raum eingebettet, und zwar als die $(z = 1)$-Ebene des dreidimensionalen Raumes. Diese Ebene liegt im Abstand von einer Längeneinheit parallel zur XY-Ebene.

Ein Punkt P wird in homogenen Koordinaten durch drei Größen (p_1, p_2, p_3) dargestellt, die eine Position im dreidimensionalen Raum markieren. Durch diese Position und den Nullpunkt des Koordinatensystems ist eine Gerade im dreidimensionalen Raum definiert, die durch diese beiden Punkte geht. Der durch die Koordinaten (p_1, p_2, p_3) dargestellte Punkt P der Ebene ist dann der Schnittpunkt von g mit der $(z = 1)$-Ebene.

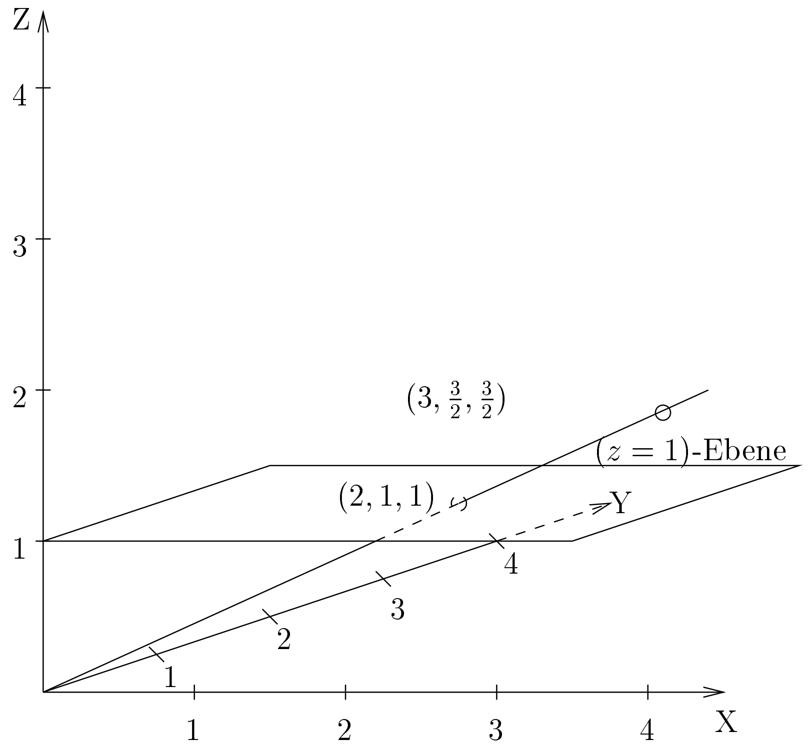

Abbildung 2.1: *Darstellung eines zweidimensionalen Punktes in homogenen Koordinaten*

Betrachten wir hierfür das Beispiel aus Abbildung 2.1: Wir haben einen Punkt in den homogenen Koordinaten $P = (3, \frac{3}{2}, \frac{3}{2})$ gegeben. Die eingezeichnete Gerade ist die dadurch definierte Gerade p, die die $(z = 1)$-Ebene im Punkt $(2, 1, 1)$ schneidet. Durch die homogenen Koordinaten $(3, \frac{3}{2}, \frac{3}{2})$ wird also der

Punkt $(2,1)$ (in kartesischen Koordinaten) dargestellt.

Im Allgemeinen erhält man aus einem gegebenen Punkt $(a,b,c), c \neq 0$ in homogenen Koordinaten den dargestellten Punkt in kartesischen Koordinaten durch Normierung nach c, d.h.

$$(a,b,c)_{homogen} = \left(\frac{a}{c}, \frac{b}{c}\right)_{kartesisch}$$

Die homogenen Koordinaten haben einen weiteren Vorteil: Ist der Grundkörper der Quotientenkörper eines Ringes, dessen Elemente einfacher darzustellen sind als die Elemente des Körpers (also zum Beispiel $\mathbb{Q}$ oder der Körper der rationalen Funktionen), so genügen für die drei homogenen Koordinaten Elemente des Ringes.

Seien z_1, z_2, n_1 und n_2 vier Elemente des Ringes, die den Punkt $(\frac{z_1}{n_1}, \frac{z_2}{n_2})$ in kartesischen Koordinaten darstellen, so findet man den zugehörigen Punkt in homogenen Koordinaten durch:

$$\left(\frac{z_1}{n_1}, \frac{z_2}{n_2}\right)_{kartesisch} = (z_1, z_2, n_1 \cdot n_2)_{homogen}$$

Wir stellen also einen beliebigen Punkt nur durch Ringelemente dar.

Man beachte, dass die oben beschriebene Gerade g nicht unbedingt die $(z = 1)$-Ebene schneiden muss, falls sie nämlich parallel zu dieser (also in der XY-Ebene) verläuft. Die hierdurch beschriebenen Punkte werden in der projektiven Geometrie als unendlich ferne Punkte bezeichnet, und die Menge aller dieser Punkte als die unendlich ferne Gerade. Die Sätze der projektiven Geometrie funktionieren unabhängig davon, ob ein in der Konstruktion enthaltener Punkt unendlich fern ist oder nicht [2].

Geraden lassen sich in der projektiven Ebene durch die selben Koordinaten beschreiben wie Punkte auch. Auch hier gibt man durch drei Koordinaten eine Position P im dreidimensionalen Raum an, die gemeinsam mit dem Nullpunkt eine Gerade p im dreidimensionalen Raum beschreibt. Diese Gerade wird nun als Normalenvektor auf einer Ebene aufgefasst, die durch den Nullpunkt gehen soll; der Schnitt dieser Ebene mit der $(z = 1)$-Ebene stellt jetzt die Gerade in der Ebene dar.

Beispielsweise betrachte man Abbildung 2.2. Hier ist die Gerade g durch die homogenen Koordinaten $(-1,-1,1)$ beschrieben. Dieser Vektor steht senkrecht auf der Ebene E, die die $(z = 1)$-Ebene in der Gearden $g = (0,1) + \lambda(-1,0)$ schneidet.

Diese Darstellung hat einige Vorteile. Der erste ergibt sich in dem folgendem Satz (Man betrachte zur Anschauung noch einmal Abbildung 2.2):

Satz

Sind $a = (a_1, a_2, a_3)$ und $b = (b_1, b_2, b_3)$ zwei Punkte in homogenen Koordinaten. Dann wird die Verbindungsgerade g von a und b in homogenen Koordinaten durch den Vektor c beschrieben, wobei

$$c = a \times b$$

gilt.

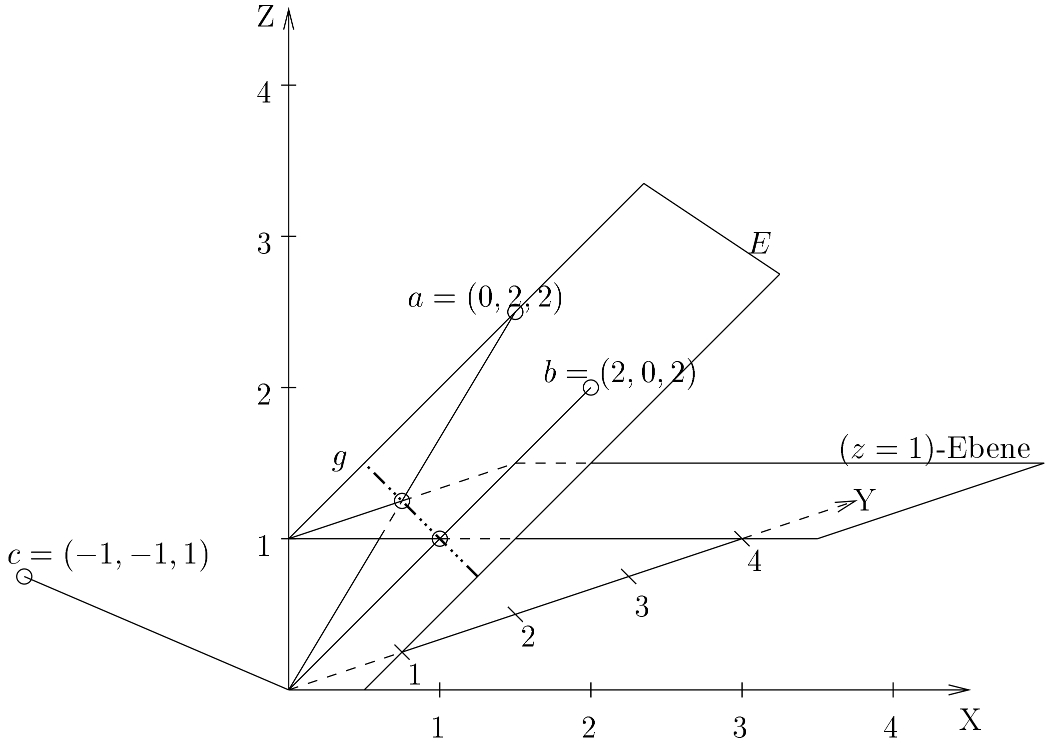

Abbildung 2.2: *Darstellung einer Geraden in homogenen Koordinaten*

Beweis

Das Kreuzprodukt c von a und b steht senkrecht sowohl auf a als auch auf b. Die Ebene E, die durch den Normalenvektor c beschrieben wird, ist die Menge aller Vektoren, die auf c senkrecht stehen; also liegen insbesondere a und b in E und schneiden die $(z = 1)$-Ebene in Punkten, in denen auch E diese Ebene schneidet, d.h. a und b liegen beide auf der durch c in homogenen Koordinaten beschriebenen Schnittgeraden g.

$\square$

Umgekehrt gilt genauso einfach (siehe Abbildung 2.3):

Satz

Sind g und h zwei Geraden mit den homogenen Koordinaten $a = (a_1, a_2, a_3)$ und $b = (b_1, b_2, b_3)$. Dann wird der Schnittpunkt c von g und h in homogenen Koordinaten durch den Vektor c beschrieben, wobei

$$c = a \times b$$

gilt.

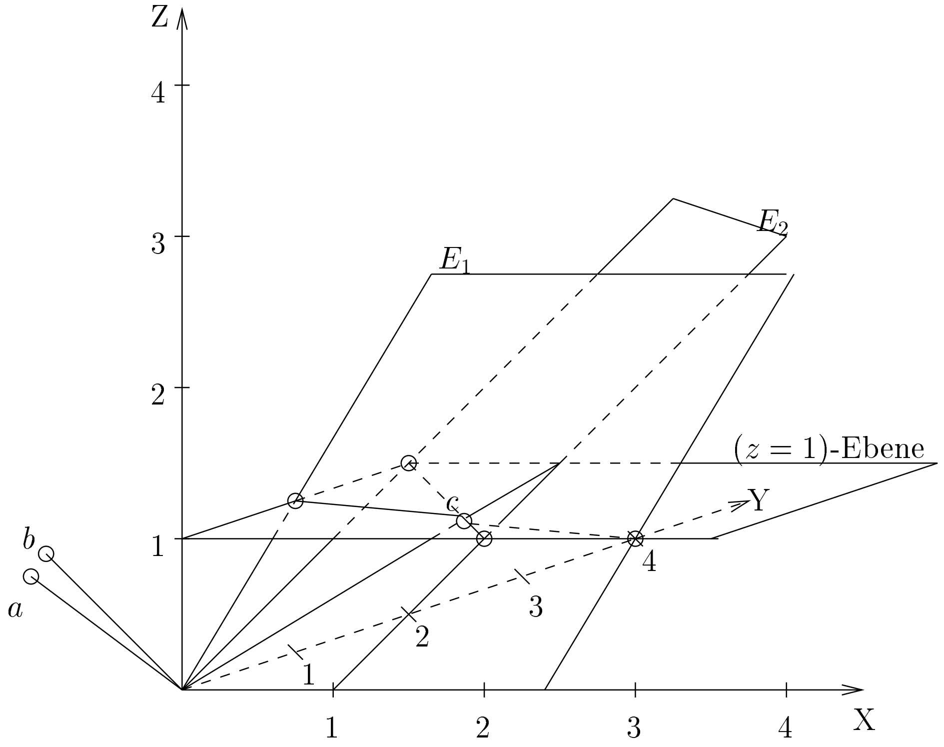

Abbildung 2.3: *Berechnung des Schnittpunktes zweier durch homogene Koordinaten dargestellten Geraden.*
Der Schnittpunkt von a und b hat seinerseits wieder die homogenen Koordinaten $a \times b$.

Beweis

Sind E_1 und E_2 die beiden Ebenen in dreidimensionalen, deren Schnitt mit der $(z = 1)$-Ebene g und h bilden. Die durch den Vektor c definierte Gerade muss die $(z = 1)$-Ebene in einem Punkt schneiden, in dem sowohl E_1 wie auch E_2 die $(z = 1)$-Ebene schneiden, also in dem Schnitt dieser drei Ebenen. Entsprechend muss c sowohl in E_1 als auch in E_2 liegen, also senkrecht zu a und b stehen, was duch die Definition von $a \times b$ gegeben ist.

$\square$

Diese einfachen Zusammenhänge ermöglichen es uns, auf einfache Weise festzustellen, ob drei vorgegebene Punkte a,b und c auf einer Geraden liegen. Dies geschieht durch folgenden Satz:

Satz

Sind $a = (a_1, a_2, a_3)$, $b = (b_1, b_2, b_3)$, $c = (c_1, c_2, c_3)$ drei Punkte in homogenen Koordinaten. Dann liegen a, b und c auf einer Geraden $\Longleftrightarrow det(abc) = 0$.

Beweis

a, b und c liegen auf einer Geraden dann und genau dann, wenn c auf der durch a und b definierten Geraden liegt, die in homogenen Koordinaten durch $a \times b$ beschrieben wird. c liegt auf dieser Geraden, falls der Vektor c in der durch $a \times b$ definierten Ebene liegt, d.h. falls c senkrecht auf $a \times b$ steht. Dies ist wiederum genau dann der Fall, wenn das Skalarprodukt von $a \times b$ mit c null ergibt, also falls $(a \times b)c = 0$. Dabei ist $(a \times b)c$ die Definition der Determinante $det(abc)$.

$\square$

Durch exakt den gleichen Beweis können wir das Duale dieser Aussage ebenfalls zeigen:

Satz

Sind $a = (a_1, a_2, a_3)$, $b = (b_1, b_2, b_3)$, $c = (c_1, c_2, c_3)$ die homogenen Koordinaten dreier Geraden. Dann schneiden sich die durch a, b und c dargestellten Geraden in einem Punkt $\Longleftrightarrow det(abc) = 0$.

Beweis

Die drei Geraden schneiden sich dann und genau dann in einem Punkt, falls der Schnittpunkt der durch a und b dargestellten Geraden auf der durch c dargestellten Geraden liegt, d.h. der entsprechende Vektor der homogenen Koordinate $a \times b$ senkrecht auf c steht, was wir wiederum durch das Skalarprodukt bestimmen können; somit erhalten wir schließlich wieder die Abfrage $(a \times b)c = 0 \Leftrightarrow det(abc) = 0$.

$\square$

Der hier an mehreren Stellen auffallende Zusammenhang zwischen Geraden und Punkten ist durchaus kein Zufall; in der projektiven Geometrie kann man alle Geraden durch Punkte und alle Punkte durch Geraden derselben Koordinaten ersetzen und erhält eine zu der ursprünglichen Konstruktion duale Konstruktion. Auch zu Anfragen der projektiven Geometrie existiert jeweils eine duale Anfrage, so dass eine Anfrage auf einer Konstruktion genau dann wahr wird, wenn die duale Anfrage auf der dualen Konstruktion wahr ist (siehe [2]). Das Dualisieren ist auch in Cedric möglich, wie in Kapitel 6.3 beschrieben wird.

Kapitel 3

Realisierung

Die in den letzten Kapiteln vorgestellten Konstruktionselemente bilden nun die Grundlage, um auf die Implementierung von Cedric eingehen zu können. Dazu werden in diesem Kapitel zunächst einmal die dem automatischen Beweiser zugrundeliegenden Arbeitsmaterialien beschrieben. Die verwendeten Termini, zum Beispiel der Begriff 'Klasse', entstammen dem üblichen Begriffen in der Programmiersprache Java. Trotzdem sind die hier beschriebenen Datenstrukturen natürlich auch in anderen Sprachen ohne Probleme umsetzbar.

Zunächst geht es dabei um die Klassen, mit denen die algebraischen Elemente implementiert werden, die zur Repräsentation der euklidischen Ebene dienen. Mit diesen Ergebnissen wird dann im folgenden Kapitel die Funktionsweise des Beweisers näher betrachtet.

3.1 Die Klasse QElement

Im geometrischen Beweiser ist es ungünstig, mit Fließkommazahlen zu arbeiten. Rundungsfehler treten zu leicht auf und können im Normalfall nur schwer erkannt werden, so dass von keiner Beweissicherheit gesprochen werden kann. Die Sicherheit des Ergebnisses wäre dann nicht größer als die Sicherheit, die auch durch vielfaches Einsetzen von Zahlen erhältlich wäre, wobei damit wesentlich weniger Aufwand verbunden wäre!

In Cedric wird daher eine ganzzahlige Repräsentation von Zahlen genommen, die durch die Klasse *QElement* repräsentiert werden. Natürlich ist es mit dem endlichen Speicher des Rechners nicht möglich, die Mengen $\mathbb{N}$ oder $\mathbb{Z}$ wirklich darzustellen. Statt dessen wird auf eine Bitrepräsentation mit 32 Bits zurückgegriffen. Falls diese Grenze während eines Beweises gesprengt werden sollte, wird dies dem Benutzer in einer Fehlermeldung mitgeteilt; dies ist allerdings in keinem der getesteten Sätze geschehen.

Des besseren Verständnisses halber will ich trotzdem die Mengen der durch 32 Bit darstellbaren ganzen bzw. natürlichen Zahlen weiterhin mit $\mathbb{Z}$ bzw. $\mathbb{N}$ bezeichnen; die damit verbundene Umdefinition ist zwar für den Inhalt dieser Arbeit ohne Bedeutung, sollte aber dennoch nicht aus den Augen verloren werden.

Die Menge der durch die Datenstruktur *QElement* darstellbaren Objekte können wir formal definieren zu

Definition

$$QElement = \mathbb{Z} \times \mathbb{N}$$

Semantisch stellt diese Klasse eine Annäherung an die Elemente von $\mathbb{Q}$ dar. Dabei wird die erste Zahl als Zähler, die zweite Zahl als Nenner eines Bruches interpretiert.

Für das geometrische Beweisen ist es ungünstig, Fälle von Division durch 0 ganz zu verbieten (wobei es allerdings nötig ist, diese Fälle im Auge zu behalten; vergleiche dazu Abschnitt 3.5). Zu diesem Zweck nehmen wir in unsere Ergebnismenge auch noch ein Element ∞ hinzu, das als Interpretation eines *QElement* dienen soll, falls der Nenner 0 ist. Im Folgenden wird diese Menge des besseren Verständnisses halber weiterhin mit $\mathbb{Q}$ bezeichnet; die damit verbundene Umdefinition sei auch hier im Gedächnis zu behalten.

Mit folgender Funktion läßt sich dann ein *QElement* in diese Menge hinein interpretieren:

$$I_{QEl}(z, n) = \frac{z}{n}$$

Wenn ein Objekt der Klasse *QElement*[1] verändert oder erstellt wird (etwa als Ergebnis einer Rechnung), so wird der größte gemeinsame Teiler von Zähler und Nenner mit Hilfe des euklidischen Algorithmus gefunden und gekürzt. Weiterhin darf natürlich nur der Zähler ein negatives Element enthalten. Ist der Zähler 0, so wird der Nenner auf eins gesetzt.

3.2 Die Klasse Term

Die Klasse *Term* dient dazu, Polynome in mehreren Variablen über $\mathbb{Q}$ darzustellen. Die jeweilige Anzahl an Variablen muss vorgegeben sein. Hat man also eine geordnete Variablenmenge V mit $\mid V \mid = n$, so wird jedes einzelne Monom im Polynom dargestellt über einen Faktor (mit Hilfe der Klasse *QElement*) und über die Potenzen, in denen die Variablen auftreten.

Die einzelnen Monome werden durch eine einfach verkettete Liste zusammengefügt. Formal notiert stellt die Klasse *Term* für eine Variablenmenge der Größe n also folgende Menge dar:

Definition

$$\begin{aligned} Term &= QElement \times \mathbb{N}^n \\ &\cup QElement \times \mathbb{N}^n \times Term \end{aligned}$$

[1] im Folgenden wird anstelle von Objekten einer Klasse direkt der Klassenname benutzt werden, d.h. ich spreche von '*QElementen*' anstelle von 'Objekten der Klasse *QElement*'.

Die natürlichen Zahlen in jedem Produkt sind die jeweiligen Potenzen der Variablen. Folgende Interpretationsfunktion übersetzt formal die Elemente der Klasse *Term* in den Polynomring $\mathbb{Q}[X_1 ... X_n]$:

Definition

$$\begin{aligned} I_{Term} &: \quad Term \to \mathbb{Q}[X_1 ... X_n] \\ I_{Term}(q, (p_1, ..., p_n)) &= I_{QElement}(q) \cdot \prod_{i=1}^{n} X_i^{p_i} \\ I_{Term}(q, (p_1, ..., p_n), t) &= I_{QElement}(q) \cdot \prod_{i=1}^{n} X_i^{p_i} + I_{Term}(t) \end{aligned}$$

3.3 Die Klasse FTerm

Die Darstellung der Polynome über die Klasse *Term* hat einen großen Nachteil in der praktischen Anwendung: Es ist sehr aufwendig, von zwei gegebenen Polynomen einen gemeinsamen Teiler zu finden. Dies ist aber besonders interessant, wenn später auch Brüche von Polynomen dargestellt werden sollen. Gerade in geometrischen Konstruktionen treten viele Faktoren häufiger auf, etwa das Skalarprodukt zwischen zwei Punktvektoren, ohne dass dieses trivial aus der Konstruktion erkennbar wäre.

Die Rechengeschwindigkeit ist entscheidend dafür, welche Sätze vom Programm noch bearbeitet werden können; daher ist es lohnend, an dieser Stelle einigen Aufwand zu treiben, um Rechenzeit zu sparen. Dies geschieht in der Klasse *FTerm*, die mathematisch betrachtet ebenfalls, genau wie die Klasse *Term*, Polynome in n Variablen über $\mathbb{Q}$ repräsentiert.

Die Klasse *FTerm* besteht aus zwei Teilen: Der erste, statische Teil der Klasse besteht aus einer Datenbank, in der Objekte der Klasse *Term* gespeichert werden. Die dynamischen Objekte der Klasse *FTerm* enthalten nun eine Liste von Verweisen auf diese Datenbank, wobei das dadurch repräsentierte Polynom das Produkt aller Einträge der Datenbank ist, auf die die Verweise zeigen.

Werden zwei Objekte der Klasse *FTerm* multipliziert, so werden deren Verweislisten einfach aneinander gehängt. Findet eine Addition von Termen statt, so werden beide *FTerme* expandiert (d.h. die Datenbankeinträge werden in der Klasse *Term* ausmultipliziert), die beiden entstehenden *Terme* werden addiert, und das Ergebniss wird in die Datenbank als neues Element aufgenommen. Das dynamische Objekt, in dem das Produkt zurückgegeben wird, enthält dann eine einelementige Liste mit genau diesem neuen Element.

In der Datenbank sind alle Elemente normiert (d.h. der Koeffizient des ersten Summanden im Polynom ist immer eins). Das neue Element kann somit ohne weiters daraufhin überprüft werden, ob es bereits in der Datenbank vorhanden ist; in diesem Falle muss es nicht neu aufgenommen werden.

Im ersten Moment gewinnt man den Eindruck, dieser Vorgang verschwende eine ganze Menge Zeit, da das neue Objekt der Klasse *Term* mit jedem bisherigen Eintrag der Datenbank verglichen werden muss. Für geometrische Beweise ist es aber meist so, dass die Länge der Polynome in der Anzahl der Schritte exponentiell ansteigt, so dass vorhergegangene Polynome meist kurz gegen das neu aufgenommen und erst recht kurz gegen das letzte Polynom ist, das in der Konstruktion auftauchen wird, und das somit für die Laufzeit des Beweises ausschlaggebend ist.

Will man jetzt einen möglichst großen gemeinsamen Teiler zweier *FTerme* finden, so kann man die Listen beider *FTerme* durchgehen und gemeinsame Elemente heraussammeln. Dies ist zwar im Allgemeinen nicht der größte gemeinsame Teiler, nichtsdestoweniger ist er groß genug, um eine immense Zeitersparnis zu erhalten.

Die Datenbank wird durch ein dynamisches Array dargestellt. Für die formale Darstellung definieren wir eine Funktion $\omega_{FTerm} : \mathbb{N} \to Term$, die die Feldposition auf das entsprechende Arrayelement abbildet. Die Definition der durch *FTerm* dargestellten Menge läßt sich dann formal notieren:

Definition

$$FTerm \quad = \quad QElement \times \mathbb{N} \times \mathbb{N}^k$$

Das erste Argument aus der Menge der natürlichen Zahlen enthält dabei den Wert k, der angibt, wie viele weitere Argumente aus den natürliche Zahlen noch folgen werden. Dies wird durch folgende Interpretationsfunktion klar, die einen *FTerm* in ein Element aus $\mathbb{Q}$ abbildet. Diese Funktion hat folgende Form:

Definition

$$\begin{aligned} I_{FTerm} \quad &: \quad FTerm \to \mathbb{Q}[X_1 ... X_n] \\ I_{FTerm}(q, k, (v_1, ..., v_k)) \quad &= \quad I_{QElement}(q) \cdot \prod_{i=1}^{k} I_{Term}\Big(\omega_{FTerm}(v_i)\Big) \end{aligned}$$

3.4 Die Klasse CTerm

Für einen großen Teil der im letzten Kapitel eingeführten Konstruktionselemente und Abfragen reicht *FTerm* völlig aus, um geometrische Sätze zu beweisen. In einigen Fällen ist es jedoch notwendig, die Quadratwurzeln aus Polynomen ziehen zu können (etwa um den Schnittpunkt eines Kreises mit einer Geraden zu bestimmen). Dabei muss die zusätzliche Operation die Quadrierung umkehren, d.h. wir suchen jeweils die positive und die negative Wurzel. Für diese Erweiterung ist die Klasse *CTerm* zuständig.

Es soll dabei möglich sein, alle Elemente aus *FTerm* darzustellen, und weiterhin noch die Wurzel aus allen Elementen, die über *CTerm* schon darstellbar sind; genauso natürlich wie in den anderen Klassen das Produkt und die Summe zweier *CTerme*. So soll also zum Beispiel auch die Wurzel einer Wurzel, also die vierte Wurzel, darstellbar sein, genauso wie die Wurzel der Summe zweier Wurzeln. Dagegen ist es für die Geometrie der im letzten Kapitel genannten Konstruktionselemente nicht nötig, etwa die dritte Wurzel ziehen zu können.

Der Ring $\mathbb{Q}[X_1, \ldots, X_n]$ muss also noch um die Wurzeln aller Elemente und um die Summe bzw. das Produkt zweier Elemente erweitert werden. Dabei sollen für die Addition und für die Multiplikation die üblichen Gesetze gelten, die auch schon in $\mathbb{Q}[X_1, \ldots, X_n]$ gelten, das heißt es soll zum Beispiel wegen des Kommutativgesetzes $a+b$ und $b+a$ das selbe Element darstellen. Den Abschluss von $\mathbb{Q}[X_1, \ldots, X_n]$ unter der Wurzeloperation und Addition und Multiplikation definieren wir formal wie folgt:

$$\begin{array}{rcl} A & = & \mathbb{Q}[X_1, \ldots, X_n] \\ & \cup & \{\pm\sqrt{x} | x \in A\} \\ & \cup & \{x + y, x \cdot y | x, y \in A\} \end{array}$$

Hierbei ist das $\pm$ vor der Wurzel von Bedeutung, denn wir suchen immer beide Lösungen der Umkehrung der Quadrierung. Kommt eine Wurzel in einer Gleichung vor, so muss die Gleichung als wahr erkannt werden, falls entweder die positive oder die negative Wurzel die Gleichung erfüllt; zum Beispiel muss sowohl die Gleichung $\pm\sqrt{x^2} = x$ als auch $\pm\sqrt{x^2} = -x$ als wahr erkannt werden. Im Allgemeinen ist es sonst auch nicht möglich, das Vorzeichen eines Ausdrucks zu ermitteln, da dieser bei verschiedenen Einsetzungen für die Unbekannten verschiedene Vorzeichen haben kann.

Beim Auflösen der Wurzeln wird im Beweiser die Gleichung nach der Wurzel aufgelöst und dann quadriert, wodurch das $\pm$ dann jeweils wieder wegfällt.

Man beachte insbesondere, dass ein partielles Wurzelziehen hier in jedem Fall möglich ist, d.h. es gilt für zwei Terme a und b die Gleichheit $\pm\sqrt{a^2 \cdot b} = a \cdot (\pm\sqrt{b})$, und zwar unabhängig vom Vorzeichen von a. Ist a positiv, so ist dies unmittelbar einleuchtend; ist a negativ, so nimmt man statt a^2 einfach $(-a)^2$ und zieht $(-a)$ vor die Wurzel; da beim Überprüfen einer Gleichung mit diesem Term das $\pm$ vor der Wurzel sowieso beide Vorzeichen zuläßt, ist das zusätzliche Minuszeichen dann ohne Bedeutung.

Natürlich tritt auch hier innerhalb einer Konstruktion die gleiche Wurzel unter Umständen mehrfach auf, so dass sich wieder eine Datenbankkonstruktion wie für die *FTerme* lohnt. *CTerm* stellt also genau wie *FTerm* einen statischen Teil zur Verfügung, in dem diese Datenbank verwaltet wird; die Elemente der Datenbank repräsentieren jeweils den Inhalt einer Wurzel. Wegen der eben erwähnten Verschachtelung müssen die Elemente dieser Datenbank natürlich ihrerseits wieder *CTerme* sein. Die dynamischen Objekte der Klasse enthalten einen *FTerm* als Faktor und eine Liste von Verweisen auf die Datenbank; zur Interpretation muss der *FTerm* mit den Wurzeln der Datenbankeinträge multipliziert werden.

Um auch Summen von Wurzeln darstellen zu können, ist auch *CTerm* in einer Listenstruktur ähnlich wie *Term* implementiert, das heißt jeder *CTerm* enthält einen zusätzlichen Verweis auf einen anderen *CTerm*, der entweder die Fortsetzung der Summe oder eine Endmarkierung enthält.

Will man aus einem vorgegebenen *CTerm* c die Wurzel ziehen, so laufen folgende Prozesse ab: Zunächst werden alle gemeinsamen Teiler der *FTerme* der Liste c gesucht. Diese werden dann überprüft, ob einzelne Datenbankverweise (der Datenbank von *FTerm*) doppelt auftauchen, so dass man aus diesen die

Wurzel echt ziehen kann. Diese werden dann aus c gestrichen und für das Ergebnis in den Vorfaktor aufgenommen. Der übrigbleibende *CTerm* wird jetzt in die Datenbank von *CTerm* aufgenommen, wobei auch hier zunächst getestet wird, ob schon eine identische Version gespeichert ist. An dieser Stelle ist ein Test auf Gleichheit zweier *CTerme* sehr aufwendig, weshalb hier nur überprüft wird, ob zwei CTerm durch die selben Listen von Datenbankelemente dargestellt sind. Ist diese der Fall, so stellen sie natürlich auch insbesondere die gleiche Funktion dar.

Zur formalen Notation sei auch hier eine Funktion $\omega_{CTerm} : \mathbb{N} \to CTerm$ eingeführt, die für eine Feldposition den entsprechenden Eintrag der Datenbank zurückliefert. Dies ermöglicht dann folgende Definition der durch die *CTerme* beschriebenen Menge:

Definition

$$\begin{aligned} CTerm &= FTerm \times \mathbb{N} \times \mathbb{N}^k \\ &\cup \; FTerm \times \mathbb{N} \times \mathbb{N}^k \times CTerm \end{aligned}$$

wobei auch hier das k, genau wie bei der Klasse *FTerm*, im ersten Argument aus den natürlichen Zahlen steht.

Die Interpretationsfunktion bildet die *CTerme* wie folgt auf die Elemente der oben definierten Menge A ab:

Definition

$$\begin{aligned} I_{CTerm} &: CTerm \to A \\ I_{CTerm}(f,k,(p_1,...,p_n)) &= I_{FTerm}(f) \cdot \pm\sqrt{\prod_{i=1}^{k} I_{CTerm}\Big(\omega_{CTerm}(p_i)\Big)} \\ I_{CTerm}(f,k,(p_1,...,p_n),c) &= I_{FTerm}(f) \cdot \pm\sqrt{\prod_{i=1}^{k} I_{CTerm}\Big(\omega_{CTerm}(p_i)\Big)} \\ &\quad + I_{CTerm}(c) \end{aligned}$$

Interessant in dieser Klasse ist noch die Abfrage, ob ein *CTerm* identisch 0 ist, d.h. ob er unter allen Einsetzungen 0 ergibt. Hierfür wird die Ordnung der Wurzeln genutzt, die sich aus der Reihenfolge der Erzeugung der Datenbankeinträge ergibt. Konstruktionsbedingt kann ein in der Datenbank an Position i gespeicherter Eintrag nur auf Datenbankeinträge verweisen, deren Position $< i$ ist (da bei der Erzeugung von Datenbankeintrag i nur diese zur Verfügung standen).

Bei dem Test auf 0 wird also ein *CTerm* zunächst nach dem Datenbankeintrag an höchster Position i, der in einem der additiven Terme als Verweis auftaucht, aufgelöst. Danach wird die Gleichung quadriert, wodurch eine Gleichung entsteht, in der nur noch Datenbankeinträge mit Positionen $< i$ vorkommen. Durch Subtraktion der einen Seite erhält man wieder einen *CTerm*, der auf 0 getestet werden soll, und die Methode kann rekursiv aufgerufen werden. Kommt keine Wurzel in dem *CTerm* vor (d.h. die Liste der Verweise auf die Datenbank ist in jedem additiven Teilterm leer), so werden die *FTerme* der Liste aufsummiert und der Nulltest der Klasse *FTerm* aufgerufen. Die Terminierung dieser Methode ist dann sichergestellt, da bei jedem neuerlichen Aufruf des Tests auf 0 in *CTerm* das übergebene Argument mindestens ein Datenbankeintrag weniger enthält als beim vorherigen Aufruf.

3.5 Die Klasse Ratfunc

Für automatisches Beweisen ist es notwendig, mit Polynomen auch Divisionen durchführen zu können. Für diese Operation benötigt man den Quotientenkörper über der eben beschriebenen Erweiterung A des Polynomringes. Für diesen Zweck gibt es die Klasse *Ratfunc*, die einfach nur einen Zähler- und einen Nenner-*CTerm* verwaltet. Der Nenner-*CTerm* wird dabei stets positiv gehalten (das heißt der Faktor des *FTerms* des ersten Monoms ist positiv), und gemeinsame Teiler werden soweit wie möglich gekürzt.

Es wird im Allgemeinen Einsetzungen für das Nenner-Polynom geben, so dass dieses 0 wird. An den Stellen in Cedric, in denen die Wahrheit eines geometrischen Satzes überprüft wird, handelt es sich um die Abfrage, ob eine aus der Konstruktion erhaltene *Ratfunc* für alle Einsetzungen den Wert 0 darstellt. Falls das Zählerpolynom für alle Einsetzungen 0 ist, das Nennerpolynom jedoch nur an endlich vielen Stellen 0 ist, so lassen sich die nicht definierten Stellen entsprechend ergänzen, analog einer stetigen Ergänzung in den reelen Zahlen; es genügt für diesen Fall also, einen Test durchzuführen, ob das Zählerpolynom konstant 0 ist. Ist das Nennerpolynom an allen Stellen 0, während es für das Zählerpolynom eine Einsetzung gibt, die ungleich 0 ist, so ist die *Ratfunc* unter dieser Einsetzung sicherlich nicht 0, und der Test, ob die *Ratfunc* unter allen Einsetzungen identisch 0 ist, liefert konsequenterweise *falsch* zurück.

Interessant ist noch der Fall, dass sowohl das Zähler- als auch das Nennerpolynom für alle Einsetzungen 0 ist. Dieser Sonderfall wird explizit abgefangen (denn hier ist keine valide Aussage möglich), und dem Benutzer wird das Fehlschlagen des Beweises durch die Fehlermeldung 'Division Null durch Null aufgetreten' mitgeteilt.

Die durch die Klasse *Ratfunc* beschriebene Menge läßt sich formal durch folgende Notation beschreiben:

Definition

$$Ratfunc \quad = \quad CTerm \times CTerm$$

Mit der Interpretationsfunktion:

Definition

$$\begin{aligned} I_{Ratfunc} &: Ratfunc \to Quot(A) \\ I_{Ratfunc}(z,n) &= \frac{I_{CTerm}(z)}{I_{CTerm}(n)} \end{aligned}$$

3.6 Die Klasse Vektor

Es wäre eine Möglichkeit, die Vektoren als Paare von Objekten der Klasse *Ratfunc* darzustellen, die - bei entsprechender Interpretation - die kartesischen Koordinaten im Raum angeben. Es hat sich jedoch herausgestellt, dass die Nutzung der in der projektiven Geometrie verwendeten homogenen Koordinaten in den meisten Fällen eine bessere Laufzeit ermöglichen. In dieser Darstellung wird ein Punkt durch drei ganzzahlige Koordinaten (bzw. in unserem Fall durch drei *CTerme*) dargestellt. Näheres dazu findet sich in Kapitel 2.5.

Falls innerhalb einer Rechnung für den dreidimensionalen Vektor der homogenen Koordinate der Nullvektor herauskommen sollte, bricht das Programm mit der oben beschriebenen Ausnahme 'Division Null durch Null' ab.

Die Klasse *Vektor*[2] benutzt also homogene Koordinaten zur Darstellung der Punkte der Ebene, wobei die drei einzelnen Koordinaten durch *CTerm* implementiert sind. Soweit möglich werden auch hier Faktoren, die in allen drei Koordinaten vorkommen, gekürzt (dies geschieht nach jeder Erstellung oder Veränderung eines Vektors). Formal stellt *Vektor* folgende Menge dar:

Definition

$$Vektor = CTerm \times CTerm \times CTerm$$

mit der Interpretation in homogenen Koordinaten:

Definition

$$\begin{aligned} I_{Vektor} &: Vektor \to A \times A \times A \\ I_{Vektor}(x,y,z) &= (I_{CTerm}(x), I_{CTerm}(y), I_{CTerm}(z)) \end{aligned}$$

[2]nicht zu verwechseln übrigens mit der in der Java-Standardbibliothek zur Verfügung gestellten Klasse *Vector*

Kapitel 4

Der Beweiser

Mit Hilfe dieser Klassen kann jetzt der formale Beweiser beschrieben werden. Dieser soll zeigen können, dass ein geometrischer Satz nach Definition aus Kapitel drei wahr oder falsch ist.

Cedric arbeitet nicht mit einem für Menschen anschaulichen Beweis, sondern rechnet die gesamte Konstruktion symbolisch durch und überprüft am Ende, ob die Anfrage für alle Einsetzungen wahr ergibt (was im üblichen Falle dem Test entspricht, ob eine *Ratfunc* identisch Null ist)[1].

Jeder Punkt, der in der Konstruktion auftritt, wird während des Beweises symbolisch erstellt und durch ein Objekt der Klasse *Vektor* dargestellt. Auf diese Weise erscheint jeder einzelne Schritt des Beweises genauso einfach, wie die entsprechende Konstruktion - mit Zahlen statt Symbolen - auf dem Papier von Hand auszurechnen.

Der Beweiser ist von der Benutzerschnittstelle verhältnismäßig streng abgegrenzt; das heißt insbesondere, dass während der Eingabe keine Beweisschritte vorgenommen werden. Erst wenn der Benutzer eine Anfrage stellt, wird dem eigentlichen Beweiser die gesamte Konstruktion samt der Anfrage von der Konstruktionsebene mitgeteilt. Eine schematische Darstellung der Verknüpfungen zwischen den zum Beweiser gehörenden Klassen findet sich auf Abbildung 4.1; eine Gesamtübersicht des Aufbaus wird dann in Abbildung 5.1 dargestellt.

4.1 Beweisvorbereitungen

Die Benutzerschnittstelle kreiert für jeden Beweis einen eigenen Thread, der von diesem Zeitpunkt an unabhängig von der Oberfläche läuft (d.h. während eines Beweise kann man in aller Ruhe an der Konstruktion weiterarbeiten; nur ein weiterer Beweisaufruf bricht den alten Beweis ab). Ein weiterer Thread beobachtet den Beweis, gibt Rückmeldung an den Benutzer und läßt eine Uhr mitlaufen.

[1]Betrachtet man die von Cedric schon bei verhältnismäßig einfachen Sätzen, wie zum Beispiel des Satzes von Pappus (siehe Anhang), bearbeiteten Polynome, die bereits ganze Absätze füllen können, so erscheint es höchst erstaunlich, dass der Mensch durch geschicktes Finden von Hilfssätzen und Anschauungen in der Lage ist, derartige Beweise in endlicher Zeit fertig zu stellen.

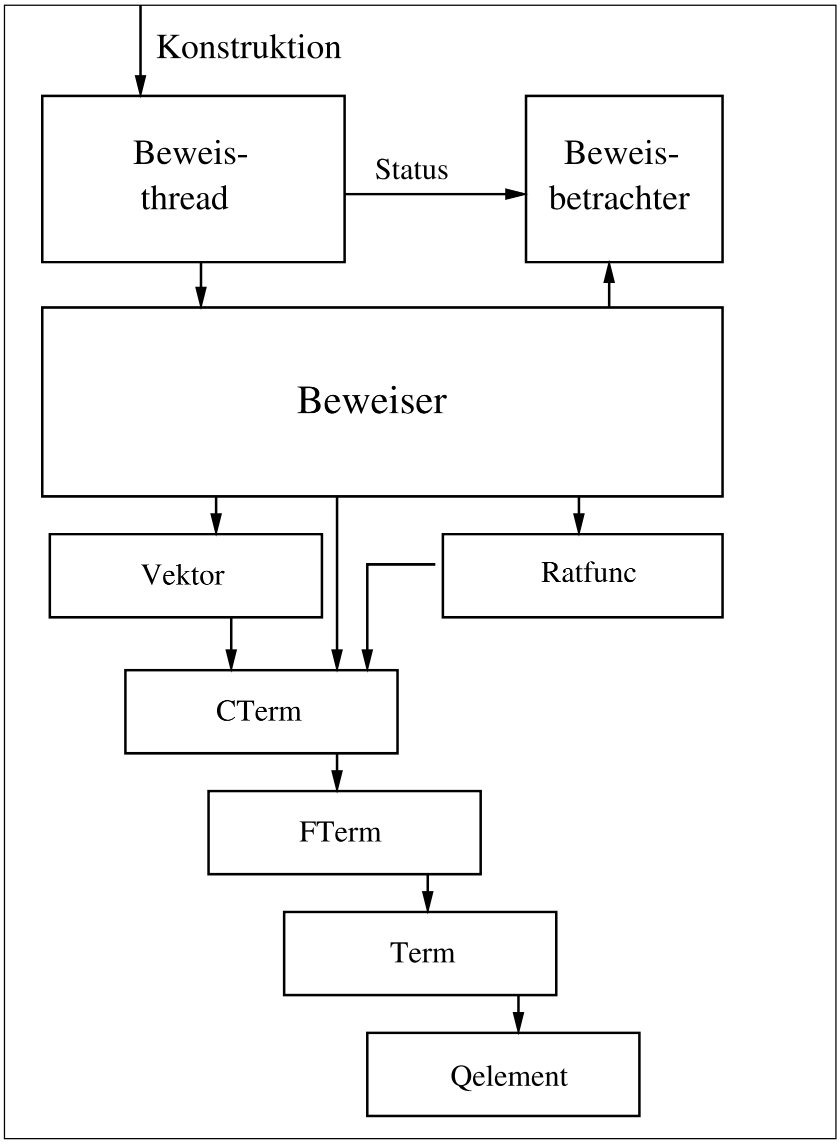

Abbildung 4.1: *Der prinzipielle Aufbau des Beweiser von Cedric.*

Der Beweiser betrachtet zunächst die Konstruktionselemente, auf die die Anfrage zielt. Alle Konstruktionselemente, die nicht für die Konstruktion der abgefragten Elemente benötigt werden, können dann im Folgenden vom Beweiser rigoros ignoriert werden (was bei ungünstiger Dateneingabe oder bei Konstruktionen mit mehreren Teilaussagen zu bedeutenden Zeiteinsparungen führen kann).

Von den verbleibenden Konstruktionselementen werden die freien Variablen gezählt und der Klasse *Term* mitgeteilt, die diesen Wert zur effizienten Darstellung der Polynome benötigt (es werden damit keine unnötigen Daten mitgeschleppt, obwohl - ebenfalls aus Gründen der Zugriffszeit - an dieser Stelle keine dynamischen Arrays oder Listen benutzt werden).

Die Anzahl der auftretenden Variablen ist für die Laufzeit von elementarer Bedeutung[2]. Cedric versucht daher mit Hilfe von affinen Transformationen, diese Zahl zu minimieren.

Dafür wird zunächst die gesamte Konstruktion so verschoben, dass der erste freie Punkt der Konstruktion (also nach Kapitel 2 das erste Konstruktionselement) auf den Ursprung des Koordinatensystems verlagert wird. Direkte Abfragen der Punktkoordinaten durch den Endbenutzer sind nicht möglich (allein schon die Eingabe verbietet ein gezieltes Setzen von Punkten in bestimmte Koordinatenpositionen), somit bleibt die Richtigkeit aller Sätze von dieser Transformation unberührt, und zwei Variablen sind eingespart.

Genauso wenig beeinträchtigend für die Korrektheit eines Satzes ist die Rotation der Konstruktion dergestalt, dass der zweite frei auftretende Punkt auf die Y-Achse gedreht wird, wodurch eine weitere Variable gespart wird.

Da keine Koordinateneichung vorgegeben ist, können wir nun auch noch die Einheiten des Koordinatensystems an dem Abstand dieser beiden freien Punkte definieren, so dass der zweite freie Punkt schließlich auf dem Punkt (1,0) zu liegen kommt.

Mit dieser Transformation wird nicht nur eine weitere Variable gespart, sondern auch noch ein weiterer, angenehmer Nebeneffekt erzeugt: Alle Strecken der Konstruktion sind jetzt an der Strecke der ersten beiden Punkte geeicht. Mit Hilfe des Gleichungseditors (siehe nächstes Kapitel) ist es damit möglich, auch explizit die Längen von Strecken abzufragen.

Eine bestimmte Auswahl von Konstruktionselementen und Abfragen sind darüber hinaus auch in der projektiven Geometrie zulässig. Cedric überprüft, ob die Anfrage und die dafür benötigten Konstruktionselemente zu dieser Auswahl gehören; falls dies der Fall ist, schaltet der Beweiser automatisch auf 'beweisen in der projektiven Ebene' um. In diesem Falle ist es zulässig, die Winkel der Konstruktion zu verändern und die Eichung der Strecken aufzugeben. Halten wir also die ersten beiden freien Punkte fest und zerren die Konstruktion jetzt am dritten freien Punkt, so können wir diesen auf den Koordinaten (0,1)

[2]Im Allgemeinen ist die Dauer eines Beweises exponentiell in der Zahl der verwendeten Variablen und der verwendeten Konstruktionselemente; erstaunlicherweise zeigt sich jedoch, dass die meisten von menschlichen Benutzern gemachten Anfragen trotz verhältnismäßig hoher Zahl an Variablen und Konstruktionselementen schnell bearbeitet werden können. Es drängt sich der Verdacht auf, dass Menschen intuitiv nur geometrische Sätze einfallen, die weitgehend unkompliziert sind!

fixieren, was zwei weitere Variablen einspart. Man beachte, dass insbesondere die im letzten Abschnitt erwähnte Abfrage über den Gleichungseditor, mit dessen Hilfe man auf die Längen von Strecken zugreifen kann, nicht zu besagter Auswahl von Abfragen gehört, die derartige Zerrungen erlauben.

Damit können in einigen Fällen also bis zu sechs Variablen, in allen Fällen aber wenigstens vier Variablen eingespart werden.

4.2 Konstruktionselemente

Der folgende Abschnitt enthält die einzelnen Implementierungen der Konstruktionselemente, die in 2 aufgezählt wurden. Diese sind teilweise sehr interessant, teilweise aber auch nur trivial. Ich werde mich daher bei den jeweiligen Herleitungen auf die interessanteren Fälle konzentrieren und mich bei den anderen entsprechend kurz fassen. Für genauer Interessierte sind die exakten Implementierungen in der Klasse *Vektor* zu finden, wobei der Methodenname von Konstruktionselementen üblicherweise mit on_ (zum Beispiel on_cut) eingeleitet wird.

4.2.1 Freier Punkt (Konstruktor)

Punkte können durch die verschiedenen Konstruktoren der Klasse *Vektor* erzeugt werden. Es existiert ein Konstruktor, der drei Variablennummern von bisher ungenutzten Variablen als Argument bekommt und automatisch einen neuen, freien Punkt erzeugt. Durch die Angabe von 3 *CTermen* kann man auch einen festen Punkt (etwa den Ursprungspunkt) erzeugen.

4.2.2 Punkt auf Gerade (on_line)

Die Methode übernimmt neben dem aktiven Objekt X einen weiteren Vektor Y als Argument, um die Gerade zu definieren, und eine *Ratfunc* λ als Faktor (mit Hilfe des Konstruktors der Klasse *Ratfunc* läßt sich hier leicht eine neue Variable einsetzen). Zurückgeliefert wird der Punkt $X+\lambda(Y-X)$. Aus Effizienzgründen wird dieser Term in $(1-\lambda)X+\lambda y$ umgeformt, da $1-\lambda$ genauso einfach wie λ zu berechnen ist, und danach nur noch zwei statt drei Additionen ausgeführt werden müssen.

4.2.3 Punkt auf Kreis (on_circle)

Da in Cedric nur mit Polynomen und nicht mit trigonometrischen Funktionen gearbeitet wird, muss der Kreis parametrisiert werden. Die geeignetste Parametrisierung findet sich durch die beiden folgenden Funktionen:

$$f_{sin} = \frac{1-t^2}{1+t^2} \tag{4.1}$$

$$f_{cos} = \frac{2t}{1+t^2} \tag{4.2}$$

Diese Funktionen findet man, wenn man die Tangensfunktion durch $\frac{t}{2}$ darstellt. Man erkennt schnell, dass die Funktionen die drei folgenden Eigenschaften aufweisen:

- $\forall t : -1 \leq f_{sin}(t) \leq 1 \,\text{und}\, -1 \leq f_{cos}(t) \leq 1$

- $\forall t : f_{sin}(t)^2 + f_{cos}(t)^2 = (\frac{1-t^2}{1+t^2})^2 + (\frac{2t}{1+t^2})^2 = \frac{1-2t^2+t^4+4t^2}{1+2t^2+4t^4} = 1$

- $\forall i, -1 \leq i \leq 1 : \ \exists t_1, t_2 : f_{sin}(t_1) = i \,\text{und}\, f_{cos}(t_2) = i$

Nimmt man also die Menge $M := \{(f_{sin}(t), f_{cos}(t)) | t \in \mathbb{Q}\}$, so enthält diese Menge Punkte des Einheitskreis in $\mathbb{Q}^2$, wobei für jeden Punkt auf dem Einheitskreis ein beliebig dicht daran liegender Punkt in M existiert. Bei Einsetzung eines symbolischen Parameters t erhält man also einen freien Punkt auf dem Einheitskreis.

Die Prozedur on_circle übernimmt neben dem aktiven Objekt M (als Mittelpunkt des Kreises) noch ein Objekt U (für einen Punkt auf dem Kreisrand) und einen *CTerm* t für die parametrisierte Darstellung des Winkels. Dieser *CTerm* kann natürlich wieder eine bisher ungenutzte Variable sein, so dass der sich ergebene Punkt semifrei auf dem Kreis liegt. Aus U und M wird ein Vektor $R := U - M$ bestimmt, der einen Radius des Kreises darstellt. Der neue Punkt auf dem Kreis berechnet sich dann zu:

$$M + f_{sin}(t) \cdot R + f_{cos}(t) \cdot R^{\perp}$$

Ist hierin $R = (r_1, r_2)$, dann bezeichnet $R^{\perp}$ den Vektor $(r_2, -r_1)$. Der Ergebnispunkt schließlich wird von der Methode on_circle zurückgegeben.

4.2.4 Schnitt zweier Geraden (on_cut)

Für die schnelle Schnittberechnung zweier Geraden eignen sich die homogenen Koordinaten besonders gut. Wie schon in Kapitel 2.5 beschrieben, ist der durch die Koordinaten (x, y, z) dargestellte Punkt der Schnittpunkt der Verbindungsgeraden des Ursprungspunktes und des Punktes (x, y, z) mit der $(z = 1)$-Ebene. Mit den selben Koordinaten lassen sich auch Geraden darstellen. Durch die Koordinaten (x, y, z) wird die Schnittgerade der $(z = 1)$-Ebene mit der Ebene E dargestellt, wobei E durch den Ursprungspunkt geht und durch (x, y, z) als Normalenvektor definiert ist (siehe dazu noch einmal Abbildung 2.3 in Kapitel 2.5).

In homogenen Koordinaten ergibt sich die Darstellung der Verbindungsgeraden zweier Punkte A und B zu dem Kreuzprodukt $A \times B$, während der Schnittpunkt zweier durch g und h beschriebenen Geraden ebenfalls durch $g \times h$ bestimmt wird.

In der Funktion on_cut sind die beiden Geraden durch das aktive Objekt (a_1, a_2, a_3) und drei weitere Punkte (b_1, b_2, b_3), (c_1, c_2, c_3) und (d_1, d_2, d_3) gegeben. Zunächst werden jetzt die homogenen Koordinaten der beiden Geraden g und h berechnet, deren Schnitt bestimmt werden soll:

$$g = A \times B = (a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1)$$

$$h = C \times D = (c_2d_3 - c_3d_2, c_3d_1 - c_1d_3, c_1d_2 - c_2d_1)$$

Damit läßt sich der Schnittpunkt x bereichnen zu:

$$\begin{aligned} X &= g \times h \\ &= (g_2h_3 - g_3g_2, g_3h_1 - g_1h_3, g_1h_2 - g_2h_1) \end{aligned} \tag{4.3}$$

Einige Terme tauchen hier doppelt auf. Diese werden natürlich hier nur einmal berechnet. Der Punkt X wird dann letztendlich als Ergebnis zurückgeliefert.

4.2.5 Mittelpunkt des Umkreises (on_center_of_circle)

Diese Methode bekommt neben dem aktiven Objekt a noch zwei weitere Vektoren b und c. Hier ist es günstiger, mit klassischen kartesischen Koordinaten zu rechnen; durch die Abbildung $(x_1, x_2, x_3) \mapsto (a_1 = \frac{x_1}{x_3}, a_2 = \frac{x_2}{x_3})$ lassen sich die homogenen Koordinaten schnell in kartesische umwandeln; im Folgenden wird also jeweils mit a_1 die kartesische x-Koordinate und mit a_2 die y-Koordinate des Punktes A bezeichnet (entsprechend für die anderen Punkte).

Die Methode soll einen Punkt zurückgeben, der den Umkreismittelpunkt um die Punkte A, B und C darstellt. Sei $X = (x_1, x_2)$ dieser gesuchte Mittelpunkt und r der Radius des entsprechenden Kreises um X durch A, B und C. Das Skalarprodukt eines Vektors X mit sich selber sei im Folgenden mit $|x|^2$ bezeichnet; dann gilt:

$$\begin{aligned} & |X - A| = |X - B| = |X - C| = r \\ \Rightarrow \ & |X - A|^2 = |X - B|^2 = |X - C|^2 = r^2 \end{aligned}$$

In wenigen, einfachen Schritten erhalten wir drei Gleichungen der Form:

$$\begin{aligned} r^2 - |X| + 2a_1x_1 + 2a_2x_2 &= |A| \\ r^2 - |X| + 2b_1x_1 + 2b_2x_2 &= |B| \\ r^2 - |X| + 2c_1x_1 + 2c_2x_2 &= |C| \end{aligned}$$

Dieses Gleichungssystem können wir auflösen und erhalten:

$$\begin{aligned} x_1 &= \frac{|A|(b_2 - c_2) - |B|(a_2 - c_2) + |C|(a_2 - b_2)}{2 \cdot (a_2(c_1 - b_1) + a_1(b_2 - c_2) + b_1c_2 - c_1b_2)} \\ x_2 &= \frac{|A|(b_1 - c_1) - |B|(a_1 - c_1) + |C|(a_1 - b_1)}{2 \cdot (a_1(c_2 - b_2) + a_2(b_1 - c_1) + b_2c_1 - c_2b_1)} \end{aligned}$$

Die einzelnen Additionen der Koordinaten wäre verhältnismäßig rechenzeitaufwendig (da für die Addition der *Ratfuncs* die Nenner identisch gemacht werden müssten). Um dies zu vereinfachen, werden zunächst die Koordinaten aller drei Vektoren auf einen gemeinsamen Hauptnenner gebracht (wobei wegen der homogenen Darstellung die Koordinaten (a_1, a_2) eines Vektors bereits den gleichen Nenner haben), danach wird obige Rechnung nur noch mit den Zählern vorgenommen. Man beachte dabei allerdings, dass die Koordinaten im Zähler des oben genannten Bruches kubisch, im Nenner jedoch nur quadratisch vorkommen; es muss also einmal noch der Nenner des Ergebnisterms mit dem Hauptnenner multipliziert werden.

Auch in dieser Rechnung lohnt es sich, die einzelnen, teilweise mehrfach auftretenden Differenzen einmal zu berechnen und an verschiedenen Stellen einzusetzen; ich führe dies hier nicht näher aus.

4.2.6 Streckenmittelpunkt (on_line)

Zur Darstellung des Mittelpunktes zweier Punkte wird wieder die in Abschnitt 4.2.2 beschriebene Funktion on_line zur Berechnung eines Punktes auf einer Geraden aufgerufen, diesmal nur mit einem festen Faktor $\frac{1}{2}$ statt mit einer neuen Variablen.

4.2.7 Quadratergänzung (on_rect)

Diese Methode nimmt neben dem aktiven Objekt A noch einen weiteren Vektor B als Argument. Der gesuchte Punkt soll den dritten Punkt C eines Quadrates $ABCD$ liefern, d.h. die Strecken $\overline{AB}$ und $\overline{BC}$ sollen gleich lang sein und senkrecht aufeinander stehen. Berechnet wird dieser Punkt einfach durch $C = B + (A - B)^{\perp}$.

4.2.8 Ergänzung zum gleichschenkligen rechtwinkligen Dreieck (on_thales_and_rect)

Diese Methode stellt eine weitere, für einige Konstruktionen wichtige Funktion dar. Die Argumente sind auch hier das aktive Objekt A und ein Argument B, die eine Strecke definieren; der neue Punkt C soll der Schnittpunkt der Mittelsenkrechten von $\overline{AB}$ und dem Thaleskreis über $\overline{AB}$ sein. ABC ist dann ein Dreieck mit rechtem Winkel bei C. Zur Berechnung dieses Punktes ist es natürlich nicht nötig, umständlich den Schnitt von Thaleskreis und Mittelsenkrechte zu bestimmen; man erhält ihn einfacher durch die Formel

$$C = B + \frac{A - B}{2} + \frac{(A - B)^{\perp}}{2} = \frac{(A + B) + (A - B)^{\perp}}{2}$$

4.2.9 Parallelogrammergänzung (on_parallelogram)

Für die Bestimmung eines Parallelogramms müssen drei Punkte A, B und C gegeben sein; berechnet werden soll dann der vierte Punkt D dergestalt, dass $ABCD$ ein Parallelogramm ergibt. Dieser Punkt läßt sich leicht zu $D = C + B - A$ bestimmen.

4.2.10 Schnitt zweier Kreise (on_cut_cc)

Diese Methode interpretiert ihr aktives Objekt A als Mittelpunkt des ersten Kreises und übernimmt drei weitere Vektoren Q, B und R, wobei Q auf dem Umfang des ersten Kreises liegt, B der Mittelpunkt des zweiten Kreises ist und R auf dem Umfang des zweiten Kreises liegt. Mit Hilfe der Punkte auf den Kreisen und der Mittelpunkte können wir problemlos die Radien r_1 und r_2 der beiden Kreise bestimmen.

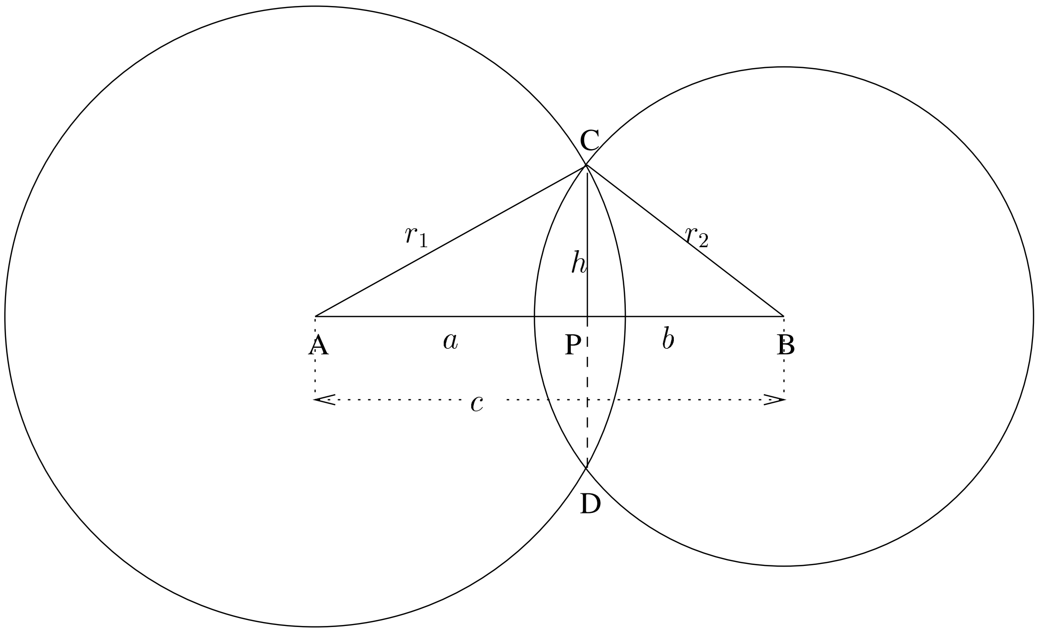

Abbildung 4.2: *Konstruktion für die Berechnung des Schnittes zweier Kreise.*

Im Folgenden wollen wir mit C und D die beiden Schnittpunkte der Kreise bezeichnen. Die Gerade $\overline{CD}$ steht senkrecht auf der Verbindungsgeraden $\overline{AB}$ der beiden Kreismittelpunkte. Der Schnittpunkt dieser beiden Geraden heiße im Folgenden P. Da die Schnittpunkte der Kreise C und D auf den Kreisen liegen, ist deren Abstand zu den Kreismittelpunkten jeweils durch die Radien bekannt. Bezeichnen wir der Einfachheit halber den Abstand der Punkte A und P als a, den Abstand der Punkte B und P als b, den Abstand der Punkte A und B als c und den Abstand der Punkte P und C als h.

Durch Anwendung des Satzes von Pythagoras auf das Dreieck APC und BCP erhalten wir die beiden folgenden Gleichungen:

$$\begin{aligned} {r_1}^2 &= a^2 + h^2 \\ {r_2}^2 &= b^2 + h^2 \end{aligned}$$

Subtrahieren wir diese Gleichungen voneinander, so erhalten wir:

$${r_1}^2 - {r_2}^2 = a^2 - b^2$$

Ersetzen wir jetzt b durch $c - a$, so ergibt dies:

$$r_1{}^2 - r_2{}^2 = a^2 - (c-a)^2 = 2ac - c^2$$

Zur Bestimmung von P interessieren wir uns für den Bruch $\lambda = \frac{a}{c}$; addieren wir zu der Gleichung c^2 und dividieren dann durch $2c^2$, so erhalten wir:

$$\frac{r_1{}^2 - r_2{}^2 + c^2}{2c^2} = \frac{a}{c} = \lambda$$

Erfreulicherweise finden wir auf der linken Seite dieser Gleichung keine Strecken mehr ohne Quadrat, d.h. wir können den Faktor λ ohne Wurzeln darstellen. Auf diesen Vorteil wird in Kapitel 4.3.7 noch einmal verwiesen werden.

Für P ergibt sich:

$$P = A + \lambda \cdot (B - A)$$

Somit können wir die Gerade beschreiben, auf der die gesuchten Punkte C und D liegen. Für deren genaue Positionsbestimmung müssen wir jetzt noch den Faktor $\mu = \frac{h}{c}$ bestimmen, um zu wissen, wie 'hoch' C bzw. D über $\overline{AB}$ liegt. Diesmal kommen wir um Wurzeln mit allen Tricks[3] nicht umhin.

Wieder durch den Pyhtagoras in Dreieck APC erhalten wir:

$$\begin{aligned} h^2 &= r_1{}^2 - a^2 \\ \frac{h^2}{c^2} &= \frac{r_1{}^2}{c^2} - \frac{a^2}{c^2} \\ \frac{h}{c} &= \sqrt{\frac{r_1{}^2}{c^2} - \lambda} \\ \mu &= \sqrt{\frac{r_1{}^2}{c^2} - \lambda} \end{aligned}$$

Die Punkte C und D lassen sich dann bestimmen zu

$$C, D = P \pm \mu (B - A)^{\perp}$$

Mit der Funktion on_cut_cc soll sowohl C als auch D bestimmbar sein; hierfür wird ein Wahrheitswert als weiteres Argument übernommen, das besagt, ob an dieser Stelle addiert oder subtrahiert werden soll.

In der implementierten Methode finden sich diese Rechnung im Wesentlichen wieder.

4.2.11 Schnitt eines Kreises mit einer Geraden (on_cut_lc)

Diese Methode betrachtet das aktive Objekt A und das erste Argument B als die Punkte, die die Gerade definieren, und die beiden weiteren Argumente M und R als den Mittelpunkt und einen Umkreispunkt des Kreises. Für alle

[3] Das man hier ohne Wurzeln nicht auskommt, läßt sich einfach beweisen: Man nehme zwei Kreise mit Radius zwei LE und einem Abstand von zwei LE; in dem Fall wäre $h = \sqrt{2^2 - 1^2} = \sqrt{3}$ und damit $\mu = \sqrt{\frac{3}{4}}$, was nicht ohne Wurzeln darstellbar ist.

Punkte P des Kreises gilt $|P - M| = r$, und alle Punkte der Geraden lassen sich beschreiben als $P = A + \lambda(B - A)$. Setzt man die Geradengleichung in die Kreisgleichung ein, so erhält man:

$$\begin{aligned} |A + \lambda(B - A) - M| &= r \\ (A + \lambda(B - A) - M)^2 &= r^2 \\ A^2 + \lambda^2(B - A)^2 + M^2 + 2\lambda(B - A)(A - M) - 2AM &= r^2 \\ \lambda^2 + \frac{2(B - A)(A - M)}{(B - A)^2} \cdot \lambda + \frac{(A^2 - 2AM + M^2) - r^2}{(B - A)^2} &= 0 \end{aligned}$$

$$\begin{aligned} \lambda = & \frac{-(B - A)(A - M)}{(B - A)^2} \\ & \pm\sqrt{\frac{\Big((B - A)(A - M)\Big)^2 - (B - A)^2(A - M)^2 + r^2(B - A)^2}{(B - A)^4}} \end{aligned}$$

Die Berechnung des λ erfordert also auch hier den Einsatz von Wurzeln[4]. Dieses λ setzt man jetzt in die Geradengleichung ein und bestimmt somit den Ergebnisschnittpunkt P. Ähnlich wie die Methode zur Berechnung des Schnittes zweier Kreise zeigt auch hier ein zusätzlicher Parameter an, ob der Wurzelausdruck addiert oder subtrahiert werden soll.

4.3 Anfragen

Der folgende Abschnitt enthält die einzelnen Implementierungen der Anfragen, die in 2 aufgezählt wurden. Ich beschränke mich auch hier, genau wie bei den Konstruktionselementen, auf die interessanteren Inhalte und überspringe jeweils das Triviale. Die exakten Implementierungen finden sich in der Klasse *Vektor*, wobei der Methodenname von Konstruktionselementen üblicherweise mit is_ (zum Beispiel is_collinear) eingeleitet wird.

Die Testmethoden geben jeweils einen Wahrheitswert zurück, der dann und genau dann *wahr* ist, falls die Argumente die gestellte Anfrage erfüllen.

4.3.1 Test auf Kollinearität (is_collinear)

Diese Methode übernimmt neben dem aktiven Objekt A noch zwei weitere Punkte B und C. Sie soll wahr zurückliefern, falls diese drei Punkte auf einer Geraden liegen. Für den Test auf Kollinearität eignen sich wieder wunderbar homogene Koordinaten. Wie in Kapitel 2.5 gezeigt, läßt sich die Gerade, die durch die beiden Punkte A und B dargestellt wird, durch das Kreuzprodukt von A und B berechnen. Das Ergebnis ist dann der Normalenvektor der Ebene

[4] Ähnlich wie bei dem Schnitt zweier Kreise kann man auch hier einfach zeigen, dass eine Wurzel vorkommen muss: Man schlage einen Kreis mit Radius zwei LE um einen Punkt M und ziehe eine Gerade g im Abstand von einer LE von M. Von M aus fällt man das Lot auf g und erhält so einen Punkt X; sei weiterhin einer der Schnittpunkte des Kreise mit g der Punkt Y. Dann ist $\overline{MY} = 2$ und $\overline{MX} = 1$, also nach Pythagoras $\overline{XY} = \sqrt{3}$.

E, dessen Schnitt mit der $(z = 1)$-Ebene die Verbindungsgerade von A zu B darstellt.

Wenn wir überprüfen wollen, ob C auf dieser Geraden liegt, so müssen wir testen, ob die durch (c_1, c_2, c_3) dargestellte Gerade die $(z = 1)$-Ebene in einem Punkt schneidet, der seinerseits auf dem Schnitt der Ebene E und der $(z = 1)$-Ebene liegen muss. Dies ist natürlich genau dann der Fall, wenn die durch (c_1, c_2, c_3) und den Ursprungspunkt definierte Gerade in E liegt, was wiederum heißt, dass (c_1, c_2, c_3) senkrecht auf dem Normalenvektor $a \times b$ der Ebene E steht. Wir kommen durch diese Überlegungen also auf folgenden einfachen Zusammenhang:

$$a, b \quad \text{und} \quad c \text{ liegen auf einer Geraden } \Leftrightarrow (a \times b)c = 0 \Leftrightarrow det(abc) = 0$$

Diese Determinante läßt sich effizient berechnen.

4.3.2 Test auf gemeinsamen Schnitt von Geraden (is_concur)

Diese Methode übernimmt neben dem aktiven Objekt fünf weitere Objekte. Die Implementierung ist simpel: Die ersten beiden Geraden werden geschnitten, und mit dem Schnittpunkt und den zwei verbleibenden Punkten der letzten Gerade wird der Test auf Kollinearität aufgerufen. Diese Anfrage ist in der projektiven Geometrie die duale Anfrage zum Test auf Kollinearität.

4.3.3 Test auf gemeinsamen Umkreis (is_one_circle)

Diese Routine übernimmt außer dem aktiven Objekt drei weitere Punkte und überprüft, ob die vier Punkte auf einem gemeinsamen Kreis liegen.

Der Umkreismittelpunkt eines Dreieckes läßt sich berechnen, indem man zwei Mittelsenkrechten der Seiten schneidet. Nimmt man noch die Mittelsenkrechte eines der Dreieckspunkte und des vierten zu testenden Punktes hinzu, so muss diese Gerade ebenfalls durch den selben Punkt gehen, damit die vier Punkte auf einem gemeinsamen Umkreis liegen.

Mit Hilfe der Konstruktionselemente für den Streckenmittelpunkt und des Schnittpunktes von Thaleskreis und Mittelsenkrechten lassen sich die Mittelsenkrechten effizient bestimmen. Mit den drei sich ergebenen Geraden muss dann nur noch der Test auf gemeinsamen Schnitt durchgeführt werden.

4.3.4 Test auf rechten Winkel (is_perpendicular)

Diese Methode testet, ob zwei gegebene Geraden senkrecht aufeinander stehen. Naheliegenderweise werden hierfür die Richtungsvektoren der Geraden bestimmt und das Skalarprodukt (mit den kartesischen Koordinaten) auf 0 getestet.

4.3.5 Test auf Parallelität zweier Geraden (is_parallel)

Diese Methode funktioniert völlig analog zum Test auf rechten Winkel, nur wird hier das Kreuzprodukt der beiden Richtungsvektoren auf 0 getestet, wobei unter dem Kreuzprodukt von A und B der Term $a_1b_2 - a_2b_1$ zu verstehen ist, also die 3. Koordinate des dreidimensionalen Kreuzproduktes der Vektoren $(a_1, a_2, 0)$ und $(b_1, b_2, 0)$.

4.3.6 Test auf gleichen Winkel (is_equal_angle)

Diese Methode testet nicht wirklich, ob zwei vorgegebene Winkel identisch sind, sondern ob sie den gleichen Kosinus haben. Ein Test auf gleiche Winkel wäre für den Benutzer in einigen Situationen etwas verwirrend, da leicht Verwechslungen zwischen einem Winkel und seinem Komplementärwinkel zu 180° oder 360° auftreten können. Allein beim Test auf Gleichheit der gegenüberliegenden Winkel in einem Parallelogramm kann es schon zu Irrtümern kommen, wenn man die Winkel in unterschiedlichen Richtungen markiert.

Diese Methode hat formal einen wesentlichen Unterschied zu allen vorhergegangenen Methoden: Während vorher immer gefordert war, dass die Argumente unterschiedliche Punkte waren, können die Argumente dieser Funktion in beschränktem Maße identisch sein: gefordert ist hier nur, dass das aktive Objekt a und die ersten beiden Argumente b und c paarweise verschieden sind, und dass die hinteren drei Argumente d, e und f paarweise verschieden sind.

Der Kosinus der Winkel wird mit Hilfe des Skalarproduktes bestimmt. Die beiden Kosinus sind gleich, falls gilt:

$$\cos(abc) = \cos(def) \Leftrightarrow \frac{(a-b)(c-b)}{|a-b||c-b|} - \frac{(d-e)(f-e)}{|d-e||f-e|} = 0$$

Der Betrag der Strecken kann mit Hilfe von Wurzeln ohne weiteres berechnet werden. Das Ergebnis des entsprechenden Vergleiches wird von der Methode zurückgeliefert.

4.3.7 Test auf gemeinsamen Schnitt von Kreisen (is_circle_meet)

Diese Methode verarbeitet neben dem aktiven Objekt M_1 noch fünf weitere Argumente R_1, M_2, R_2, M_3, R_3. Dabei stellen die M_i jeweils die Mittelpunkte dreier Kreise K_i dar, während die R_i jeweils ein Punkt auf dem Umfang von K_i sind. Die Methode soll wahr zurückliefern, falls die drei Kreise einen gemeinsamen Schnittpunkt haben.

An dieser Stelle ergibt sich ein überaus interessanter Aspekt: Obwohl (wie oben gezeigt) der Schnittpunkt der drei Kreise (so er existiert) im Allgemeinen nicht ohne Verwendung von Wurzeln bestimmt werden kann[5], kann dennoch die Frage nach der Existenz dieses Schnittpunktes ohne Zuhilfenahme des Wurzeloperators gelöst werden! Diese Methode ist in der Implementierung (aufgrund des nicht geringen Aufwandes für die Wurzeln) schneller als die Alternative, die Schnittpunkte zweier Kreise mit on_cut_circle zu bestimmen und von beiden zu prüfen, ob sie ein Element des dritten Kreises sind.

Bei der Berechnung des Schnittpunktes zweier Kreise (s. Abschnitt 4.2.10) wurde gezeigt, wie man ohne Wurzeloperator die Gerade bestimmen kann, die

[5] da man sonst zwei identische Kreise mit einem dritten Kreis schneiden könnte und so den in Fußnote 3 erwähnten Fall erhalten würde.

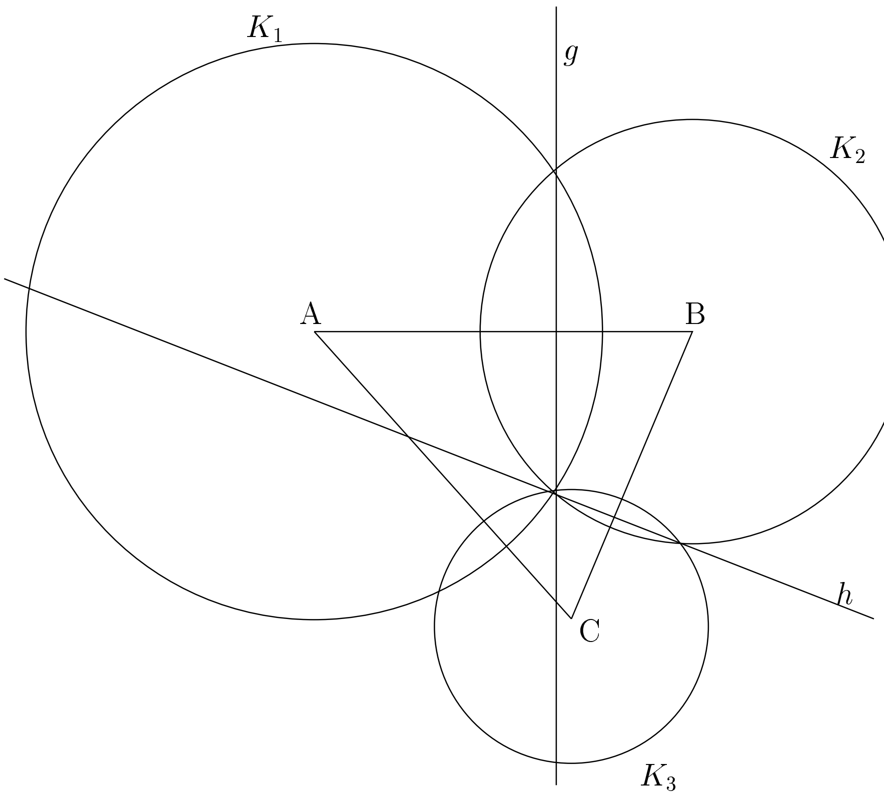

Abbildung 4.3: *Test, ob drei Punkte sich in einem Punkt schneiden. Dieser Test ist ohne Zuhilfenahme von Wurzeln möglich. K_1, K_2 und K_3 schneiden sich dann und genau dann in einem Punkt, wenn der Schnittpunkt von g und h auf K_2 liegt.*

durch beide Kreisschnittpunkte läuft. Wenden wir diese Methode auf K_1 mit K_2 und K_2 mit K_3 an, so erhalten wir zwei Geraden g als Gerade durch die Schnittpunkte von K_1 und K_2 und h als Gerade durch die Schnittpunkte von K_2 und K_3 (siehe Abbildung 4.3). Jetzt benötigen wir noch folgendes Lemma:

Lemma

Seien K_1, K_2 und K_3 drei Kreise und g und h wie eben. Dann schneiden sich K_1, K_2 und K_3 in einem Punkt $\iff$ Der Schnittpunkt von g, h liegt auf K_2.

Beweis

'$\Rightarrow$' Die drei Kreise schneiden sich in einem Punkt x. Nach Konstruktion geht g durch die gemeinsamen Schnittpunkte von K_1 und K_2, also insbesondere auch durch x. Genauso verläuft auch h durch die Schnittpunkte von K_2 und K_3, also insbesondere durch x, d.h. g und h schneiden sich in x. x liegt als Schnittpunkt aller drei Kreise auch auf K_2.

'$\Leftarrow$' Die Geraden g und h schneiden sich in einem Punkt x, der auf K_2 liegt. Da g den Kreis K_2 an den Stellen schneidet, an denen auch K_1 den Kreis K_2

schneidet, ist $x \in K_1$. Da h den Kreis K_2 an den Schnittstellen von K_2 und K_3 schneidet, ist $x \in K_3$. Also ist x auf allen drei Kreisen, die dann hier einen gemeinsamen Schnittpunkt haben.

□

Besonders betrachten sollte man den Fall, dass K_1 oder K_3 (sagen wir o.B.d.A. K_1) keinen Schnittpunkt mit K_2 hat. In diesem Fall wird die Gerade g (siehe Abbildung 4.3) ebenfalls senkrecht auf der Verbindungsline der Kreise stehen, und zwar zwischen den beiden Kreisen. In diesem Fall liegt kein Punkt von g, also insbesondere auch nicht der Schnittpunkt mit h, auf K_2, wodurch die rechte Seite der Äquivalenz sicher falsch wird; dass die drei Kreise sich nicht in einem Punkt schneiden können, wenn zwei der Kreise gar keinen Schnittpunkt haben, liegt wohl nahe. Dieser Sonderfall ist damit also gelöst.

Mit diesem Lemma können wir den Test, ob sich die drei Kreise in einem Punkt schneiden, problemlos durchführen: Wir berechnen wie oben beschrieben g und h, schneiden die beiden Geraden in einem Punkt X und prüfen, ob X auf K_2 liegt, d.h. ob der Abstand zwischen X und M_2 gleich dem Abstand zwischen M_2 und R_2 ist (was genau dann der Fall ist, wenn $(X - M_2)^2 = (R_2 - M_2)^2$ gilt).

Ein Sonderfall verdient noch Erwähnung: Es kann sein, dass g und h parallel liegen. Das würde bedeuten, dass M_1, M_2 und M_3 kollinear sind. In diesem Fall reduziert sich die Abfrage darauf, ob g und h identisch sind; sind sie es, so scheiden sich die drei Kreise in einem Punkt (genaugenommen sogar zwei mal; diese Punkte sind allerdings nicht mehr unter allen Umständen ohne Wurzeln bestimmbar), andernfalls schneiden sie sich nicht in einem Punkt.

4.3.8 Gleichungsanfrage (is_equal)

Diese Anfrage hat einen besonderen Status. Im Gegensatz zu den vorher genannten sind hier nicht eine feste Anzahl von Punkten involviert; statt dessen wird die Zahl der für diese Anfrage wesentlichen Punkte durch die Gleichung bestimmt, die textuell eingegeben werden kann. Beide Seiten des Gleichheitszeichens müssen mathematische Ausdrücke nach einer speziellen Grammatik enthalten, die dann von einem eigenen Programmteil[6] geparst werden (siehe Kapitel 6.1). Für die Abfrage auf Gleichheit werden dann beide Seiten zu *Ratfuncs* interpretiert; dann werden die Seiten voneinander subtrahiert und die resultierende *Ratfunc* auf 0 geprüft.

[6]Dabei handelt es sich um die Klasse 'formulatree' im Quellcode

Kapitel 5

Aufbau der Benutzerschnittstelle

Der Beweiser und die Benutzerschnittstelle sind in Cedric streng getrennt, was ermöglichen soll, dass der Beweiser später noch mit einem anderen Interface verwendet werden kann. In diesem Kapitel wird beschrieben, wie die Schnittstelle zum Benutzer aufgebaut ist. Interessant ist dies auch für das Verständnis des Validierers, der im folgendem Kapitel vorgestellt wird.

Im Gegensatz zum eigentlichen Beweiser, in dem alle Konstruktionselemente durch Punkte repräsentiert werden, gibt es auf der Darstellungsebene vier voneinander getrennte Darstellungsobjekte. Punkte spielen dabei nur insofern eine Sonderrolle, da der Benutzer in Anpassung an den Beweiser gezwungen sein soll, seine Konstruktion so zu halten, dass keine Punkte später eingefügt werden müssen (zum Beispiel kann der Benutzer nicht direkt eine freie oder semifreie Gerade legen; er muss dafür Fixationspunkte festlegen und kann die Gerade dann durch diese Punkte ziehen).

5.1 Darstellungsobjekte

Alle Darstellungsobjekte erfüllen die Spezifikationen der abstrakten Klasse Darstellunsobjekt, in der auch für alle Darstellungsobjekte relevante Mitglieder und Methoden zur Verfügung gestellt werden. Zunächst einmal ist ein Array von fünf Verweisen von Bedeutung, die auf andere Darstellungsobjekte zeigen, von denen dieses Objekt in der Konstruktion abhängig ist. Für eine Gerade sind dieses zum Beispiel die beiden Punkte, durch die die Gerade definiert ist. Ein weiteres wichtiges Mitglied ist eine Statusfeld, in dem die genaue Art des Konstruktionselementes (und damit auch die Vorschrift, nach der die Argumente verrechnet werden müssen) gespeichert ist.

Ein weiterer Verweis mit der Bezeichnung 'nächster' ist im Darstellungsteil ungenutzt, wird aber vom Validierer benutzt, um die Darstellungsobjekte linear zu verketten. Weiterhin wird für jedes Darstellungsobjekt eine Farbe gespeichert und eine Reihe von Zustandsvariablen (ob das Objekt markiert wurde, ob die Maus sich ihm nähert oder ob es vom Benutzer gegriffen wurde und gerade über den Bildschirm bewegt wird).

Zwei Methoden dieser Klasse sorgen für eine anschauliche Übersetzung eines Darstellungsobjektes in einen String (eine Methode liest einen String ein und baut daraus ein Darstellungsobjekt, die andere übersetzt das aktive Darstellungsobjekt in einen String). Dieser String erscheint auf dem Bildschirm in dem Feld für die Konstruktionsmitschrift auf der rechten Seite und wird auch für die Speicherung von Konstruktionen genutzt. Näheres dazu findet sich im Kapitel 6.2.

Die Klasse Darstellungsobjekt fordert von den von ihr abgeleiteten Klassen folgende wesentliche Methoden:

- paint: Die Darstellung des Objektes auf dem Monitor.
- recalculate: Berechnet aus den Vorgängern des Objektes und seiner alten Position eine neue Position.
- nearest_on_object: Berechnet zu einer vorgegebenen Position die nächste Position auf dem Objekt.
- close: Gibt einen Wahrheitswert zurück, ob eine vorgegebene Position innerhalb eines gewissen Toleranzbereiches in der Nähe des Objektes liegt.
- get_next_name: Gibt den nächsten kanonischen Namen des Objektes zurück.

5.1.1 Punkte

Die Klasse der Darstellungspunkte definiert eine Anzahl von möglichen Zuständen, die ein Punkt annehmen kann und die als Status des Punktes bezeichnet werden. Dabei kann ein Punkt zum Beispiel frei sein, auf einem anderen Darstellungsobjekt semifrei liegen, oder durch eine Konstruktion entstanden sein und eine feste Position innehaben.

Mit diesem Status, den angegebenen Vorgängerobjekten des Punktes und unter Umständen - falls der Punkt frei oder semifrei ist - seiner alten Position berechnet recalculate dann die neue Position des Punktes. Die Position wird im Darstellungsteil immer durch kartesische Koordinaten angegeben (und zwar direkt die Bildschirmkoordinaten, an denen der Punkt am Ende erscheint). Diese beiden Koordinaten sind durch Fließkommazahlen dargestellt, um eine möglichst große Genauigkeit zu ermöglichen; für die Darstellung auf dem Monitor müssen sie natürlich jeweils auf den nächsten Integerwert gerundet werden.

Die Verwendung der Fließkommazahlen (anstatt direkt mit Integers zu arbeiten) hat noch einen zweiten Grund: Der Validierer, der durch Einsetzen zufälliger Koordinaten einen Satz zwar nicht beweisen, aber doch stark untermauern kann, arbeitet exakt mit den selben Klassen wie der Darstellungsteil (siehe näheres zum Validierer in Kapitel 6.4).

Für jedes in Kapitel 2 aufgeführtes Konstruktionselement stellt die Klasse Darstellungspunkt eine Methode zur Verfügung, die von recalculate aufgerufen wird und die Position des Punktes in Abhängigkeit von den anderen Konstruktionselementen bestimmt. Diese Methoden sind im Allgemeinen wie die entsprechenden Methoden des Beweisers konstruiert, nur operieren sie auf

Fließkommazahlen statt auf *CTermen* und sind darüber hinaus weit weniger zeitlich optimiert geschrieben; größere Tricks sind hier einfach unnötig, da die Berechnung mit diesen beispielhaften Zahlen wesentlich schneller geht als die Berechnung mit symbolischen Werten im Beweiser.

Bei der Darstellung des Punktes werden zwei Farben verwendet: Zum einen die definierte Farbe, mit der der Punkt ausgefüllt wird, und dann noch eine weitere Farbe, mit der der Punkt eingerahmt wird, und die eine Aussage über den Zustand des Punktes machen soll. Die Farbe Gelb steht für eine Markierung des Punktes, Blau heißt, dass der Punkt vom Benutzer gegriffen wurde und im Moment bewegt wird, und Weiß, dass der Benutzer einen anderen Punkt in die unmittelbare Nähe des Punktes gezogen hat. Wenn die Konstruktion an einem Punkt gegriffen und bewegt wird, bewegen sich folgerichtig alle Konstruktionselemente, die von diesem ersten abhängen, ebenfalls; diese Objekte sind daher auch gelb gekennzeichnet.

Die Methode close ist im Falle des Punktes denkbar einfach gemacht: Es wird dabei nur überprüft, ob die übergebenen Koordinaten innerhalb eines zehn Pixel großes Quadrates um den Punkt herum liegen.

Um die Namensgebung der Objekte zu ermöglichen, haben alle Darstellungsobjekte einen Zähler, der jeweils mitschreibt, wie viele Objekte dieser Art in der Konstruktion bereits vorkommen. Mit Hilfe dieses Zählers funktioniert dann die automatische Benennung der Objekte; die Punkte werden hierbei mit Großbuchstaben benannt, angefangen mit dem Punkt 'A', danach in alphabetischer Reihenfolge fortgesetzt. Sind alle Buchstaben des Alphabetes verbraucht, wird mit Kombinationen aus zwei Buchstaben fortgefahren, d.h. auf 'Z' folgt 'AA', dann 'AB' und so fort.

5.1.2 Linien

Geraden und Strecken sind im Darstellungsbereich in einer Klasse zusammengefasst. Wird eine Konstruktion in die für den Beweiser lesbare Form übersetzt, so werden alle Geraden, Kreise und Kegelschnitte ohnehin ignoriert, das heißt der Unterschied zwischen Strecken und Geraden geht in den Beweis auf keine Weise ein. Man beachte, dass also definitionsgemäß in Cedric insbesondere in jedem Fall ein Schnittpunkt zweier Strecken existiert, der allerdings nicht notwendigerweise auf einer dieser Strecken liegen muss.

Für die Unterscheidung dieser beiden Objekte wird das Statusfeld des Darstellungsobjektes genutzt. Der einzige Unterschied, der zwischen Geraden und Strecken besteht, befindet sich in der Methode paint: Während Geraden immer vom Zeichenblattrand aus gezeichnet werden, bleiben Strecken auf den Bereich zwischen ihren definierenden Endpunkten beschränkt.

Die beiden Endpunkte der Linie werden durch zwei mal zwei Fließkommazahlen repräsentieren, die direkt die Bildschirmkoordinaten enthalten, an denen eine Linie beginnt bzw. endet. Im Falle einer Geraden sind dieses immer Punkte auf dem Rand des Zeichenblattes, im Falle der Strecken die jeweiligen definierenden Punkte (solange diese sich auf dem Zeichenblatt befinden; ansonsten handelt es sich ebenfalls um Randpunkte). Diese Punkte werden von der Methode recalculate berechnet. Während dies für Strecken verständlicherweise wenig Arbeit macht, müssen für Geraden alle vier Schnittpunkte mit den Randbegrenzungslinien berechnet werden und dann entsprechend der Lage dieser Punkte zwei als Endpunkte gewählt werden.

Auch hier ist es erwünscht, dass neben der Farbe der eigentlichen Geraden auch noch eine Darstellung des Zustandes der Geraden erfolgen kann. Dafür berechnet die paint-Methode zwei direkt benachbarte Geraden bzw. Strecken, deren Farbe nach dem jeweiligen Zustand ausgesucht wird: Gelb wieder für eine markierte Gerade, Weiß für eine Annäherung an die Gerade. Es ist unmöglich, eine Gerade ohne Greifen ihrer Endpunkte zu bewegen, daher fällt der dritte Fall hier weg.

Um einen Punkt auf einer Geraden bewegen zu können, wird die Methode nearest_on_objekt benötigt, die für eine vorgegebenen Position (etwa des Mauszeigers) die dichteste Position auf der Geraden berechnet. Bei Geraden wird hierfür die senkrechte Projektion der Koordinaten auf die Gerade verwendet.

Auf diese Weise kann auch ohne viel Aufwand die Methode close implementiert werden. Diese Methode ruft einfach die eben beschriebene Methode nearest_on_object auf und läßt sich den Projektionspunkt zurückgeben, um dann mit dessen Position als Argument die close-Mehode der Klasse Darstellungspunkt aufzurufen.

5.1.3 Kreise

Für die Kreise werden in der Klasse Darstellungskreis drei Fließkommazahlen gespeichert: Zwei für die Position des Mittelpunktes, eine weitere für den Radius des Kreises. Mit diesen Angaben ist das Zeichnen des Kreises kein großes Problem, eine entsprechende Prozedur ist in den Java-Standardbibliotheken vorhanden.

Ein gewisses Problem stellen hier natürlich die Zeichenblattbegrenzungen dar. Nur Kreissektoren zu zeichnen und die Sektoren, die übers Blatt hinausragen, nicht mitzuzeichnen, ist ein unvertretbarer Zeitaufwand. Statt dessen wird der Kreis ohne Berücksichtigung des Randes gezeichnet, und direkt danach die Bildschirmabschnitte außerhalb des Zeichenblattes gelöscht. Es kommt zwar vor, dass man dadurch bei schnellen Bewegungen ein leichtes Flackern außerhalb des Zeichenblattes bemerkt, dies rechtfertigt aber nicht den gewaltigen Zeitaufwand der Alternativlösung.

Für die Methode nearest_on_object wird die Verbindungsgerade vom Kreismittelpunkt zu den übergebenen Koordinaten mit dem Kreisumfang geschnitten (indem der Verbindungsvektors auf die Länge des Radius normiert wird). Die Funktion close überprüft, ob der Abstand der übergebenen Koordinaten zum Mittelpunkt maximal fünf Pixel vom Radius abweicht.

5.1.4 Kegelschnitte

Die Kegelschnitt bieten noch einmal ein interessanteres Problemfeld. Es sollen gleichermaßen Ellipsen, Parabeln und Hyperbeln dargestellt werden können, und zwar dergestalt, dass man sie einfach ineinander überführen kann, das heißt man kann das Problem nicht umgehen, indem man drei einzelne Darstellungsobjekte für die drei Arten von Kegelschnitten zur Verfügung stellt. Ein Kegelschnitt soll durch fünf vorgegebene Punkte beschreibbar sein, und auf dem Kegelschnitt sollen semifreie Punkte gesetzt werden können.

Weiterhin wünschenswert wäre es, Punkte auch auf den Schnitt von Kegelschnitten und anderen Objekten legen zu können oder eine Tangente an den Kegelschnitt anlegen zu können. Bedauerlicherweise sprengen diese letzten Punkte, jedenfalls im Moment, noch die Möglichkeiten von Cedric. Genausowenig ist es möglich, Konstruktionen mit Kegelschnitten im Beweiser bearbeiten zu können (versucht man es, meldet das Programm einen Fehler zurück). Mit dem Validierer dagegen, der die Darstellungsobjekte benutzt, ist eine Verarbeitung von Kegelschnitten kein Problem. Damit geht es zum Beispiel auch, den Satz von Pascal (siehe Anhang) zu bestätigen.

Man bedenke, dass ein einzelner Kegelschnitt in der projektiven Geometrie immer als Kreis dargestellt werden kann; hat man also einen Satz der projektiven Geometrie, in dem nur ein einzelner Kegelschnitt vorkommt, so kann man statt des Kegelschnittes einen Kreis wählen und den Satz auch mit dem Beweiser exakt belegen[1].

Die Kegelschnitte sind durch fünf Fließkommazahlen $a_{11}, a_{12}, a_{22}, b_1, b_2$ und c repräsentiert, die nach [3] folgendermaßen die Menge der Punkte auf dem Kegelschnitt darstellen:

$$\mathit{Kegelschnitt} \quad = \quad \{(x, y) \mid a_{11}x^2 + 2a_{12}xy + a_{22}y^2 + 2b_1x + 2b_2y + c = 0\}$$

Und falls $a_{22} \neq 0$:

$$= \quad \{(x, y) \mid y^2 + \frac{2a_{12}x + 2b_2}{a_{22}} y + \frac{a_{11}x^2 + 2b_1x + c}{a_{22}} = 0\}$$

Dabei können die Koeffizienten dieser Gleichung durch fünf Punkte bestimmt werden; dies geschieht in der Methode recalculate, die diese Punkte in eine Matrix einsetzt, für die eine eigene Klasse existiert. In dieser Klasse ist ebenfalls eine Lösung des Gaußalgorithmus implementiert, der in verhältnismäßig kurzer Zeit die fünf Unbekannten bestimmen kann[2].

Die letztere der beiden oben genannten Darstellungen der Kegelgleichung ermöglicht es der Methode paint, bei gegebener x-Koordinate durch Aufruf einer Funktion zum finden der Lösungen der quadratischen Gleichung die möglichen zugehörigen y-Koordinaten zu berechnen.

Entsprechend durchläuft die paint-Methode das Zeichenblatt pixelweise von links nach rechts und zeichnet an allen Stellen die beiden Äste des Kegelschnit-

[1] Falls der Kegelschnitt im Laufe der Konstruktion durch fünf Punkte definiert wird, muss man Freiheitsgrade bewegen: Man suche mindestens zwei der fünf definierenden Punkte, die noch mindestens einen Freiheitsgrad haben (d.h. die mindestens semifrei sind; gibt es keine solchen Punkte, so muss noch weiter zurückgegangen werden und zunächst diesen Punkten Freiheitsgrade gegeben werden). Diese beiden Punkte schiebt man in der Konstruktion hinter den Kegelschnitt und fixiert sie auf dem Kegelschnitt (das heißt einen freien Punkt legt man als Punkt auf den Kreis, einen Punkt auf einer Geraden macht man zum Schnittpunkt der Geraden mit dem Kreis).

[2] Die Laufzeit des Gaußalgorithmus ist in $O(n^3)$; siehe dazu [4].

tes. Zu Beginn wird dazu für den linken Blattrand die y-Positionen des Kegelschnittes gemerkt und in zwei Feldern gespeichert. Beim Durchlaufen wird jeweils die nächste Position berechnet und eine Linie von der gemerkten Position zu dieser neuen Position hin gezogen. Laufen die beiden Äste zusammen, das heißt wird beim Durchlaufen die erste x-Position betreten, bei der die p-q-Formel keine Ergebnisse zurückliefert, so werden die beiden letzten Posititonen durch eine Linie verbunden; entsprechend werden die beiden Äste geöffnet, falls das erste mal valide Positionen auftauchen. Die Zeichnung beschränkt sich dabei natürlich auf das Zeichenblatt. Dies liefert auf dem Bildschirm die dichtestmögliche Annäherung an die Form des Kegelschnittes.

Die Berechnung des nächsten Punktes auf dem Kegelschnitt, die Methode nearest_on_object, wird mit dem Lagrange-Ansatz gelöst. Hierfür stellen wir den Kegelschnitt zunächst einfacher dar. Dafür wollen wir zuerst eine Translation dergestalt finden, dass das Zentrum des Kegelschnittes auf dem Koordinatenursprung liegt. Bezeichnet (m_1, m_2) das Zentrum des Kegelschnittes, so würden wir gerne die Menge aller Punkte des Kegelschnittes beschreiben durch:

$$\{(x,y)\,|\,a_{11}(x-m_1)^2 + 2a_{12}(x-m_1)(y-m_2) + a_{22}(y-m_2)^2 + k = 0\}$$

Expandiert man diese Formel, so erhält man:

$$a_{11}x^2 + 2a_{12}xy + a_{22}y^2 + 2(-a_{11}m_1 - a_{12}m_2)x + 2(-a_{22}m_2 - a_{12}m_1)y + c = 0$$

Das konstante Glied ist dabei für die Berechnung des Zentrums ohne Bedeutung.

Um m_1 und m_2 zu erhalten, haben wir jetzt bei gegebenen b_1 und b_2 aus der ursprünglichen Formel für den Kegelschnitt zwei Bestimmungsgleichungen, nämlich

$$\begin{aligned} a_{11}m_1 + a_{12}m_2 + b_1 &= 0 \\ a_{12}m_1 + a_{22}m_2 + b_2 &= 0 \end{aligned}$$

Dieses Gleichungssystem läßt sich ebenfalls mit dem Gaußverfahren lösen. Sonderfälle, in denen es nicht genau eine Lösung gibt, können durch minimale Verschiebungen, die auf dem Bildschirm nicht sichtbar sind, vermieden werden.

Durch Normierung der Gleichung nach der Konstanten erhalten wir also für den Kegelschnitt um den Ursprungspunkt eine angenehm einfache Darstellung. Fassen wir die normierten Koeffizienten als eine symmetrische Matrix A auf und schreiben die Punkte auf dem Kegelschnitt jeweils als einen Vektor x, so erhalten wir folgende Darstellung:

$$\mathit{Kegelschnitt} = \{x\,|\,x^TAx = 1\}$$

Durch diese Formel beschreiben wir jetzt einen Kegelschnitt um den Ursprungspunkt. Für die Methode nearest_on_object werden die zu testende Position p ebenfalls um den Vektor $-(m_1, m_2)$ verschoben; das Ergebnis wird dann am Ende der Methode wieder um (m_1, m_2) zurückgeschoben.

Zum Finden des Punktes, der p am nächsten ist, verwenden wir den Ansatz von Lagrange [3], indem wir die Extrema folgender Funktion suchen:

$$f(x) = (x-p)^2 + \lambda(x^TAx - 1)$$

Abgeleitet nach x und λ ergibt sich:

$$\begin{aligned}\frac{\partial f}{\partial x} &= 2(x-p)+2\lambda Ax\\ \frac{\partial f}{\partial \lambda} &= x^T Ax - 1\end{aligned}$$

Setzen wir $\frac{\partial f}{\partial x}$ gleich null und lösen nach x auf, so erhalten wir:

$$x = (\lambda A + I)^{-1} \cdot p$$

Wobei I die Einheitsmatrix bezeichnet. Dieses x setzen wir in $\frac{\partial f}{\partial \lambda}$ ein, setzen diesen Term ebenfalls gleich null und lösen die Gleichung nach den Koeffizienten von λ. Dann erhalten wir folgende Gleichung:

$$\begin{aligned} 0 &= \left(-(a_{11}a_{22}-a_{12}^2)^2\right)\lambda^4\\ &+ \left(-2(a_{22}+a_{11})(a_{11}a_{22}-a_{12}^2)\right)\lambda^3\\ &+ \Big(2a_{12}^2-a_{11}^2-a_{22}^2+a_{22}p_2^2a_{11}^2-a_{12}^2p_2^2a_{11}\\ &\quad +2a_{12}^3p_1p_2-2a_{11}p_1a_{22}a_{12}p_2-a_{12}^2p_1^2a_{22}+a_{11}p_1^2a_{22}^2-4a_{11}a_{22}\Big)\lambda^2\\ &+ \left(-2p_1^2a_{12}^2-2p_2^2a_{12}^2+2a_{22}p_2^2a_{11}+2a_{11}p_1^2a_{22}-2a_{11}-2a_{22}\right)\lambda\\ &+ \left(-1+a_{11}p_1^2+2a_{12}p_2p_1+a_{22}p_2^2\right)\end{aligned}$$

Zum Lösen von Gleichungen vierten Grades existiert im Programm eine eigene Klasse *Gleichungslsg*[3]. Die resultierenden Werte für λ werden dann genutzt, um die entsprechenden Vektoren x zu finden; von diesen Punkten wird der p am nächsten liegende Punkt als Ergebnis gewählt.

Beim Lösen der Gleichung vierten Grades treten jedoch unvermeidbarerweise leichte Rundungsfehler auf. Für die Zeichnung auf dem Bildschirm ist dies kein Problem, da die Rundungsfehler sowieso unterhalb des darstellbaren Bereiches liegen. Da jedoch die selben Methoden auch für den Validierer genutzt werden, ist es notwendig, dass der Ergebnispunkt exakt auf dem Kegelschnitt liegt. Hierfür wird die x- und die y-Koordinate des gefundenen Punktes genutzt, um ein dem Achsenkreuz paralleles Kreuz zu erstellen, das dann mit der Methode zum Lösen quadratischer Gleichungen mit dem Kegelschnitt geschnitten werden kann, wodurch die Rundungsfehler ausgeglichen werden. Von diesen Schnittpunkten wird noch einmal der der Position p am naheliegenste ausgewählt und schließlich nach der Rücktranslation um (m_1, m_2) zurückgegeben.

Die Methode close, die überprüfen soll, ob eine Position dicht am Kegelschnitt liegt, ruft jetzt einfach die Methode nearest_on_object auf und überprüft, ob der dadurch erhaltene nächste Punkt auf dem Kegelschnitt genügend dicht an der übergebenen Position liegt.

[3]Programmiert nach einem Programm in C++ von Jochen Schwarze, 1990.

5.2 Der Objektminister

Der Objektminister ist eine Klasse, die die Verwaltung der Darstellungsobjekte übernimmt. Die Objekte werden dabei in der von den Standard-Javabibliotheken zur Verfügung gestellten Datenstruktur *Vector* gespeichert[4]. Diese Klasse funktioniert im Grunde wie eine Liste, an deren Ende Elemente angefügt oder entfernt werden können, erlaubt aber auch Zugriffe auf innere Elemente.

Eine Reihe von Methoden des Objektministers erlauben eine Aufzählung der in ihm gespeicherten Elemente nach außen, so dass diese einzeln bearbeitet werden können. Der Zähler für die Aufzählung kann dabei auch zwischengespeichert werden, um zunächst auf weitere Elemente der Liste zugreifen zu können.

Der Objektminister stellt weiterhin zwei Methoden zur Verfügung, mit denen die Namen der Konstruktionselemente bearbeitet werden können: Die eine Methode zählt die vorhanden Objekte jeder der vier Klassen durch und setzt den Namenszähler dieser Klassen, so dass die automatisch vergebenen Namen dahinter ansetzen. Die zweite Methode geht noch einen Schritt weiter und ersetzt alle Namen durch die kanonisch auftretenden Namen. Besonders nach einer Dualisierung, einem Ladevorgang oder der Erstellung eines vorgefertigten Hilfsobjektes sind diese Routinen sehr nützlich.

Um Schritte rückgängig machen zu können oder rückgängig gemachte Schritte wieder vorwärts gehen zu können, verwaltet der Objektminister zwei Listen, in denen jeweils die vollständige Datenstruktur der Darstellungsobjekte gespeichert ist. Eine Methode save_for_undo wird nach jeder wichtigeren Operation aufgerufen und fügt den momentanen *Vector* von Darstellungsobjekten an die undo-Liste an. Wird ein undo aufgerufen, so wird die momentane Position in der redo-Liste gespeichert, und das oberste Element der undo-Liste wird wieder als aktive Situation genommen. Beim redo läuft der selbe Vorgang andersherum ab. Wird ein save_for_undo aufgerufen, so wird die redo-Liste automatisch gelöscht, während die momentane Situation in der undo-Liste gespeichert wird.

In einer weiteren Liste werden alle Elemente gespeichert, die im Moment markiert sind. Diese Markierungsliste kann nach außen weitergegeben werden, es ist aber auch möglich, sich die markierten Elemente aufzählen zu lassen.

Eine Routine close dient der Feststellung, welche Objekte der übergebenen Position nahe sind. Dabei wird in einer eigenen Klasse Closewrapper folgende Objekte übergeben: Das absolut nächste Darstellungsobjekt, der nächste Punkt, die nächste Linie, die zweitnächste Linie (wesentlich zum Beispiel für Schnitte), der nächste Kreis, der zweitnächste Kreis, der nächste Kegelschnitt und der zweitnächste Kegelschnitt. Diese Objekte in dieser kompakten Form erleichtern an vielen Stellen den Zugriff wesentlich.

Eine Funktion translate übersetzt die im Objektminister gespeicherten Darstellungsobjekte in eine linear verkettete Liste, die der Validierer als Eingabe benötigt.

Das Hinzufügen von Darstellungsobjekten ist im Allgemeinen nicht trivial, da verschiedene Vereinfachungen für den Benutzer erreicht werden sollen. Um

[4]Ich bitte erneut darauf zu achten, dass diese Standardklasse nichts mit der in Cedric implementierten Klasse *Vektor* zu tun hat; beide haben völlig unterschiedliche Aufgaben und Funktionsweisen.

eine möglichst intuitive Bedienung zu erzielen, arbeitet der Objektminister mit dem Konzept eines Schattenpunktes. Falls der Benutzer auf dem Zeichenblatt den Mausknopf betätigt und sich Cedric im Moment nicht im Bewegungsmodus befindet, so wird ein solcher Schattenpunkt erzeugt. Er erscheint dem Benutzer ganz wie ein neuer Punkt, den er noch gegriffen hält (d.h., der Punkt ist blau umrandet). Hält er die Maustaste weiterhin gedrückt, so bewegt er den Schattenpunkt auf dem Zeichenblatt (nähert er sich dabei einem Objekt, so wird dieses weiß markiert). Sobald er den Knopf losläßt, wird der Schattenpunkt materialisiert, was im Objektminister die Methode materialize_shadow übernimmt.

Befindet sich zum Zeitpunkt der Materialisierung ein anderer Punkt p in der Nähe des Schattenpunktes, so verschwindet der Schattenpunkt sofort wieder ins Nichts, und der Punkt p übernimmt lediglich die Farbe des Schattenpunktes. Falls Cedric in einem Status ist, der das Markieren von Punkte erfordert (zum Beispiel falls eine Gerade gezogen werden soll), so wird p markiert beziehungsweise, falls er schon markiert war, wieder demarkiert. Hat der Benutzer also nur einmal kurz auf einen existierenden Punkt geklickt, so wurde ein Schattenpunkt erstellt und unmittelbar darauf wieder gelöscht, während der entsprechende Punkt markiert wurde.

Ist der Schattenpunkt zum Zeitpunkt des Materialisierens isoliert von allen anderen Objekten der Konstruktion, so wird ein neuer, freier Punkt an dieser Position erzeugt. Wird eine Markierung von Punkten erwartet, so erscheint der neue Punkt direkt als markierter Punkt. Dadurch ist es zum Beispiel möglich, auf leerem Blatt eine Gerade einfach zu erzeugen, indem man an zwei Positionen klickt, durch die die Gerade gehen soll, auch wenn an diesen Positionen vorher noch keine Punkte lagen.

Befindet sich der Schattenpunkt bei der Materialisierung im Umfeld von anderen Objekten, die ihrerseits keine Punkte sind, so wird ein neuer Punkt erzeugt, der auf diesen Objekten fixiert wird. Dabei wird immer versucht, möglichst viele Objekte einzubeziehen; klickt man also in den Bereich des Schnittes zweier Geraden, so wird nicht ein semifreier Punkt auf einer der Geraden, sondern der Schnittpunkt erzeugt. Es haben im Allgemeinen Linien eine höhere Affinität als Kreise und diese wiederum als Kegelschnitte; innerhalb einer Klasse hat immer das ältere Objekt den Vorrang. Wird ein Schattenpunkt also zum Beispiel auf dem Schnitt eines Kreises und dreier Geraden materialisiert, so wird er an die ältesten beiden Geraden gebunden.

5.3 Der Darstellungsraum

Eine gewisse Zentralposition im Programm hat die Klasse Darstellungsraum inne, die, obwohl sie die Aufgaben der anderen Programmteile eigentlich nur verwaltet, die umfangreichste Klasse des Programmes ist. Im Ablauf wird diese Klasse nur einmal instanziiert, und zwar direkt vom Applet aus. Alle Benutzerkommandos werden an den Darstellungsraum weitergeleitet.

Eine zentrale Aufgabe des Darstellungsraum ist das Zeichnen des Zeichenblattes. Nacheinander durchläuft die Methode hierfür mit Hilfe des Objektministers zuerst die Punkte, dann die Geraden, dann die Kreise und zuletzt die Kegelschnitte und ruft deren paint-Methoden auf. Falls ein Beweis läuft, wird hier auch der Statusdialog des Beweises gezeichnet.

Die Kommunikationsleiste, die unter dem Zeichenblatt dem Benutzer kurze Erklärungen zum momentanen Stand der Dinge gibt, wird ebenfalls vom Darstellungsraum verwaltet. Es gibt 28 verschiedene Status, in denen sich das Programm befinden kann, und zwischen denen der Benutzer durch die Menuleiste oder die Buttons wechseln kann; entsprechende Hinweistexte werden vom Darstellungsraum erzeugt.

Auch die zusätzlichen Funktionen wie die Beweismitschrift, Speichern und Laden, Rücknahme und Wiederherstellung des letzten Schrittes, Löschen der Konstruktion und Ändern der Farben sowie das Dualisieren (siehe Abschnitt 6.3) sind in dieser Klasse implementiert.

Eine weitere Zentralaufgabe ist die Verwaltung der Mauskontrolle auf dem Zeichenblatt. Die entsprechenden Funktionsweisen des Objektministers wurden im letzten Abschnitt beschrieben; der Darstellungsraum koordiniert dabei nur die Kommunikation zwischen der Maus und dem zuständigen Objektminister. Die in Cedric bereitgestellten Konstruktionselemente sind möglichst wenig redundant gestaltet, das heißt man soll möglichst wenig Konstruktionselemente aus elementaren Kombinationen anderer Elemente erhalten. So existiert zum Beispiel kein Konstruktionselement für die Tangente an einen Kreis, da man dieses Element auch durch einen Punkt auf dem Kreisumfang, einen Quadratergänzungspunkt zum Mittelpunkt und dem neu gesetzten Punkt und einer Verbindungsgerade dieser beiden neuen Punkte erhält.

Um aber trotzdem möglichst komfortabel Konstruktionen eingeben zu können, sind einige wichtige Kombinationen von Elementen direkt im Darstellungsraum gespeichert. Von der Oberfläche aus kann man ein solches Kombinationselement anwählen; hat man dann alle nötigen Argumentobjekte markiert, so fügt Cedric automatisch eine Reihe von Konstruktionselementen ein, die die erwünschte Kombination darstellen. Es sind folgende fünf Kombinationen möglich:

Fällen eines Lotes: Hierfür muss eine Gerade und ein Punkt gewählt werden, der auf dem Lot liegt. Es wird zunächst ein Fußpunkt des Argumentpunktes auf der Geraden erzeugt. Dieser hat die Farbe 'verdeckt', d.h. er kann vom Benutzer nicht gesehen werden (er taucht allerdings in der sequentiellen Auflistung auf). Der Fußpunkt und der ursprüngliche Punkt werden dann durch eine sichtbare Strecke verbunden.

Tangente an Kreis legen: Der Benutzer klickt auf eine Position in der Nähe eines Kreises. An dieser Stelle wird automatisch ein versteckter Punkt gesetzt; mit dem Mittelpunkt und diesem Punkt wird dann ein Quadratergänzungspunkt gesetzt, ebenfalls vor den Augen des Benutzers versteckt. Diese beiden Punkte werden dann durch eine sichtbare Gerade verbunden.

Erstellen einer Senkrechten zu einer Geraden: Für diese Kombination wird eine bestimmte Stelle einer Geraden erwartet, an der ein versteckter

Punkt gesetzt wird. Mit einem der beiden die Gerade definierenden Punkte wird jetzt ein Quadratergänzungspunkt gesetzt, der ebenfalls versteckt ist. Diese beiden Punkte werden durch eine Gerade verbunden.

Legen einer Parallelen zu einer Geraden: Die Stelle, an der der Benutzer die Gerade markiert, zu der eine Parallele gelegt werden soll, ist hier ohne Bedeutung. Im Abstand von fünf Pixel zur Geraden wird ein sichtbarer, freier Punkt erzeugt, ein Parallelogrammergänzungspunkt zu den beiden die Gerade definierenden Punkte und diesem Punkt wird verdeckt hinzugefügt. Die beiden neu gesetzten Punkte werden durch eine Gerade verbunden, die sich jetzt mit dem ersten Punkt bewegen läßt, dabei aber immer parallel zur Argumentgerade bleibt.

Ziehen eines Kreises mit eingestelltem Zirkel: Diese Kombination eignet sich, um einen Kreis mit einem festgelegten Radius zu ziehen. Hierbei werden vom Benutzer zunächst zwei Punkte markiert, deren Abstand den Radius darstellt, und danach der Mittelpunkt des Kreises festgelegt. Mit den beiden markierten Punkten und dem Mittelpunkt wird ein versteckter Parallelogrammergänzungspunkt gesetzt, der den gleichen Abstand zum Mittelpunkt hat wie die beiden markierten Punkte voneinander. Um den Mittelpunkt wird durch diesen versteckten Punkt ein sichtbarer Kreis gezogen.

In der Menuleiste befindet sich außerdem noch ein Menupunkt Hilfsobjekte, in dem vollständig abgespeicherte Konstruktionen (wie etwa verschiedene Dreiecke, verschiedene Vierecke, ein allgemeines Fünfeck und ein allgemeines Sechseck) direkt erstellt werden können. Im Darstellungsraum wird dafür direkt die Textrepräsentation der Konstruktion erzeugt und an eine Methode gesendet, die ähnlich wie die in Kapitel 6.2 beschriebene Laderoutine funktioniert.

Für das Gesamtverständnis des Programmaufbaus kann Abbildung 5.1 hilfreich sein, die einen ungefähren Überblick über die Verknüpfung der einzelnen Klassen liefert. Der Inhalt der Box mit der Beschriftung 'Beweiser' ist in Abbildung 4.1 in Kapitel 4 noch näher ausgeführt.

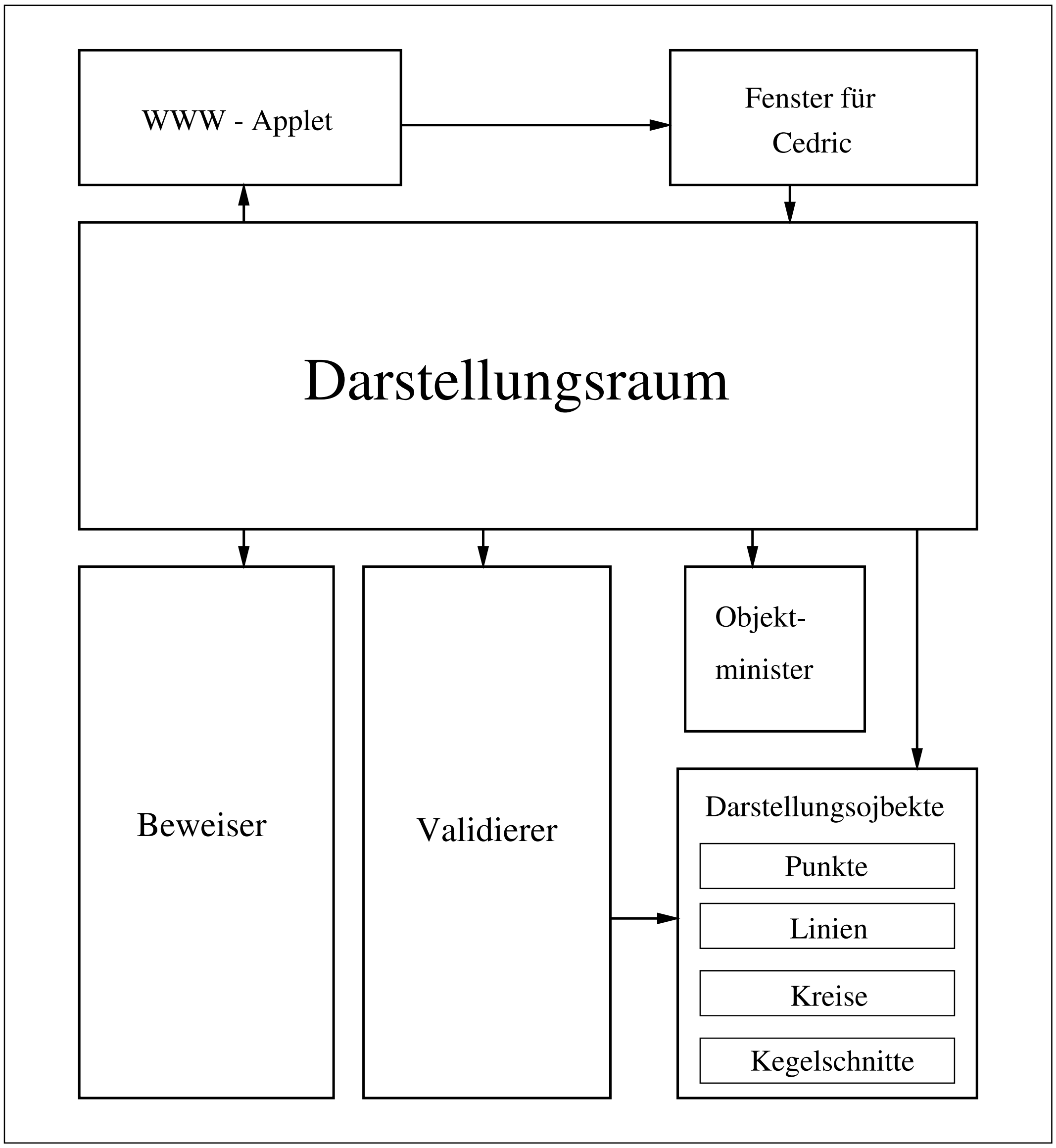

Abbildung 5.1: *Der Gesamtaufbau von Cedric*

Kapitel 6

Weitere Eigenschaften

In den vorangegangenen Kapiteln wurde der Beweiser und die Darstellungsebene von Cedric erläutert. Im Wesentlichen stellen diese beiden Teile das ganze Programm dar.

In diesem Kapitel wird auf den verbleibenden Rest eingegangen: Dinge, die sich schwer in die genannten beiden Themengebiete einordnen. Das ist zum einen die spezielle, in Kapitel 4 erwähnte Anfrage nach der Gültigkeit einer frei einzugebenen Gleichung, dann das Speicherformat der Konstruktionen, die Dualisierungsfunktion und der Validierer.

6.1 Der Gleichungseditor

Für einige geometrische Sätze ist es notwendig, bestimmte Zusammenhänge zwischen Streckenlängen abzufragen (vergleiche zum Beispiel Anhang, Satz von Ceva). Dies ist natürlich nicht mit einer Anfrage mit fester Zahl von Argumenten zu erreichen.

Daraus stellt sich für Cedric der Anspruch, auch allgemeine Anfragen bezüglich Streckenlängen in einer Konstruktion bearbeiten zu können. Für diesen Zweck gibt es den Gleichungseditor. Der Benutzer kann jederzeit per Hand oder mit verschiedenen Buttons eine Gleichung in das Formeltextfeld gleich unter den Buttonleisten schreiben; aus dem Menu oder mit dem entsprechenden Button kann die Gleichung dann überprüft werden.

6.1.1 Aufbau der Gleichungen

In der Gleichung dürfen nur ganze Zahlen vorkommen, wobei sich allerdings alle Brüche mit Hilfe des Divisionsoperators erstellen lassen. Ein Minuszeichen vor einer Zahl als Argument einer der Operationen wird als unäres Minus erkannt und entsprechend verarbeitet.

Neben den Zahlen dürfen auch Streckenlängen aus der Konstruktion in der Gleichung vorkommen. Als Verweise auf diese werden hier die Namen der Punkte verwendet (sollten aus irgendeinem Grund zwei Punkte den selben Namen haben, so referiert der Name immer auf den zuerst erstellten der beiden Punkte). Die Punktnamen werden durch ein Kommazeichen getrennt und von $

umschlossen; so wird die Strecke $\overline{AB}$ etwa durch $\$A,B\$$ repräsentiert.

Diese Elemente können durch die vier Grundrechenarten $\{+,-,\cdot,\div\}$ verknüpft werden. Außer zwischen zwei Zahlen (die sonst als eine interpretiert werden würde) darf ein $\cdot$ weggelassen werden und wird automatisch ergänzt. Der Parser beachtet die Punkt- vor Strichrechnung, wobei die beiden Operationen einer Art jeweils gleiche Prioritäten haben. Alle Operatoren sind linksassoziativ, eigene Auswertungsreihenfolgen können durch Klammern erzwungen werden.

Betrachten wir die Menge V als die endliche Menge aller Punktnamen der Konstruktion, dann läßt sich der Aufbau der Gleichungen formal durch folgende kontextfreie Grammatik darstellen:

$$
\begin{array}{lcll}
S & ::= & T = T & \\
T & ::= & T + T \,|\, T - T \,|\, T \cdot T \,|\, T \div T \,|\, TT \,|\, A & \\
A & ::= & (T) \,|\, N \,|\, -N \,|\, \$\, v_1, v_2\, \$ & v_1, v_2 \in V \\
N & ::= & ZN \,|\, N & \\
Z & ::= & 1\,|\,2\,|\,3\,|\,4\,|\,5\,|\,6\,|\,7\,|\,8\,|\,9\,|\,0 &
\end{array}
$$

Die Benennung der hier vorkommenden Nichtterminale ist nach deren jeweiligen Aufgaben gewählt; so steht T für einen Term, A für ein Atom, N für eine Nummer und Z für eine Ziffer.

6.1.2 Der Parser

Der Gleichungsstring wird in eine Datenstruktur geparst, die durch die Klasse Formulatree repräsentiert wird. Beide Seiten des Gleichheitszeichens werden dabei zunächst getrennt und unabhängig voneinander in jeweils einen Formulatree übersetzt.

Diese Datenstruktur besteht aus einem binären Baum, in den jeder der beiden Seiten des Gleichheitszeichen umgewandelt wird. Die inneren Knoten dieses Baumes sind mit einem Operator beschriftet, die Blätter jeweils entweder mit einer ganzen Zahl oder mit zwei Knotennamen der Konstruktion, deren Abstand dann durch diesen Knoten dargestellt wird.

Der Formulatree ist dann für den Beweiser recht einfach in eine *Ratfunc* umzuwandeln, indem er für jeden inneren Knoten die *Ratfunc* der beiden Kinder ermittelt und diese mit der entsprechenden Operation verknüpft beziehungsweise für Blätter den Zahlenwert oder den Abstand der Punkte einsetzt. Die für beide Seiten der Gleichung ermittelten *Ratfuncs* müssen dann voneinander subtrahiert und auf 0 geprüft werden.

Der Parser basiert im Wesentlichen auf drei Methoden, die jeweils einen größeren Teil des Strings parsen. Diese drei Methoden werden jetzt vorgestellt.

Die Methode stringToAtom

Diese Methode bekommt einen String als Übergabeparameter, der mit einem Argument für eine der zugelassenen mathematischen Operationen, einem Atom, beginnt. Die Methode übersetzt dieses Argument in einen Formulatree und löscht es aus dem String. Der resultierende Formulatree und der verkürzte String werden zurückgegeben. Zusätzlich liefert die Methode noch einen Wahrheitswert zurück, der angibt, ob das Parsing fehlerfrei durchgeführt werden konnte. Gibt die Methode falsch zurück, so war der Übergabestring kein Element der durch oben beschriebene Grammatik definierten Sprache, und der Benutzer bekommt eine entsprechende Rückmeldung.

Falls der Argumentstring mit einem Minuszeichen beginnt, so wird dieses als unäres Minus vor einem Zahlenwert betrachtet. Ein entsprechender Marker wird gesetzt, und es wird gefordert, dass der String mit einer Ziffer fortgesetzt wird (ist dies nicht der Fall, so bricht die Routine mit einem Parsefehler ab).

Falls mit einer Ziffer fortgesetzt wird oder der String von vornherein mit einer Ziffer begann, werden alle folgenden Ziffern eingelesen und zu einer Zahl zusammengesetzt. Als Formulatree für die Rückgabe wird ein einzelner Blattknoten zurückgegeben, der als Nummerknoten markiert ist und die gefundene Zahl enthält.

Begann der String mit einem \$-Zeichen, so liest die Methode zunächst den weiteren String bis zum Komma ein, und danach den String bis zum abschließenden \$-Zeichen (falls vor dem Komma bzw. dem \$ das Stringende erreicht wird, wird mit einem Fehler abgebrochen). Für die beiden Strings wird überprüft, ob sie als Name in der Konstruktion auftreten; ist dies so, so werden die Namen und ein Identifizierungscode der entsprechenden Punkte in einem neuen Blatt gespeichert und dieses als Ergebnis zurückgegeben.

Sollte der String mit einer öffnenden Klammer beginnen, so ist das Argument der Operation ein eigener, getrennt auszuwertender Term. Es wird dann zunächst nach der zugehörigen schließenden Klammer gesucht. Dafür wird ein Zähler auf eins initialisiert und der String weiter durchlaufen, wobei für jede öffnende Klammer der Zähler um eins erhöht, bei jeder schließenden Klammer um eins verringert wird. Dies wird fortgesetzt, bis der Zähler wieder auf 0 zurückgekehrt ist oder das Stringende erreicht ist, wobei in letzterem Fall mit einem Parsefehler abgebrochen wird. Die gefundenen äußersten Klammern werden entfernt, und die Methode stringToFormula mit dem dazwischen liegendem String als Argument verschränkt rekursiv aufgerufen. Da der String wenigstens um die beiden Klammern und damit die Stringlänge wenigstens um zwei abgenommen hat, ist die Terminierung dieses Aufrufes abgesichert. Der zurückerhaltene Formulatree wird dann als Ergebnis nach unten weitergereicht.

Die Methode stringToFormula

stringToFormula bekommt einen String, der einen vollständigen Term repräsentiert, als Argument übergeben, und soll daraus den zugehörigen Formulatree erstellen. Auch diese Methode liefert einen Wahrheitswert zurück, der anzeigt, ob das Parsing fehlerfrei beendet werden konnte.

Zunächst wird der String in einzelne Tokens zerlegt, die jeweils aus einer Operation und dem nachfolgenden Atom bestehen (wobei zu bemerken ist, dass ein eingeklammerter Term seinerseits als Atom zählt). Der Operator des ersten Tokens ist nicht belegt. Die Atome werden einzeln mit der Methode stringToAtom in Formulatrees umgewandelt und sequentiell in einer Tokenliste abgespeichert.

Diese Liste wird jetzt elementweise durchlaufen. Hat ein Token einen Operator höherer Priorität (also $\cdot$ oder $\div$), so wird ein neuer innerer Knoten mit diesem Operator als Beschriftung generiert, der den Formulatree des vorhergehenden Listentokens als linken Unterbaum und den Formulatree des aktuellen Listenelementes als rechten Unterbaum erhält. Ein neuer Token mit diesem neu generierten Formulatree und dem Operator des vorhergehenden Listenelementes wird erstellt; dieser neue Token ersetzt in der Liste jetzt die beiden verarbeiteten Token (also den vorhergehenden und den aktuellen Token). Auf diese Weise greifen sich die Operatoren höchster Priorität sozusagen die ihnen am nahestehendsten Argumente; da die Liste von Links nach Rechts durchlaufen wird, bleibt die Linksassoziativität erhalten.

Ist die Liste vollständig durchlaufen, so befinden sich in den Tokens der Liste nur noch die Operatoren $+$ und $-$. Analog wird die Liste noch einmal von Links nach Rechts durchlaufen und zu einem Formulatree zusammengesetzt. Dieser wird schließlich zurückgegeben.

Die Methode stringToEquation

Der Parsevorgang beginnt, indem zuerst die statische Methode stringToEquation der Klasse Formulatree mit dem String aus dem Gleichungsfeld aufgerufen wird. Um die Formel fehlertoleranter zu machen, werden zunächst alle Leerzeichen aus dem String entfernt. Danach sucht die Methode nach dem Gleichheitszeichen und trennt den String in die beiden Teilstrings auf, die dann einzeln an die Methode stringToFormula weitergegeben werden. Diese Methode ruft dann ihrerseits im Ablauf die Methode stringToAtom auf.

Die Methode myToString

Auch wenn die Gleichung vollständig erfolgreich geparst wurde, kann es - besonders bei Benutzern, die mit üblichen Notationen der Mathematik nicht vertraut sind - noch zu Missverständnissen kommen (der Benutzer kann zum Beispiel versehentlich die Punkt- vor Strichrechnung vergessen haben). Um hier die Fehlergefahr zu vermindern, gibt Cedric eine Rückmeldung, wie der Gleichungsstring interpretiert worden ist. Dafür wird der Formulatree mit der Methode myToString in einen String zurückverwandelt, und das Gleichungstextfeld auf der Oberfläche wird mit diesem String überschrieben.

Die Funktionsweise der Methode ist simpel; für ein Blatt wird der nummerische Wert beziehungsweise die beiden Knotennamen angegeben (wobei negative Zahlen noch einmal separat eingeklammert werden). Für jede Operation wird eine öffnende Klammer, der String des linken Kindes, das Zeichen für den Operator, der String des rechten Kindes und schließlich eine schließende Klammer ausgegeben.

6.2 Das Speicherformat

Nach dem Erstellen einer größeren Konstruktion kann es vorkommen, dass man die Übersicht etwas verloren hat. Besonders die Reihenfolge, in der die Punkte und Geraden gesetzt wurden, ist oft schwer zu behalten. Außerdem würde man, gerade bei einem neuen Satz, natürlich gerne eine einfache und kompakte Möglichkeit haben, diesen Satz an andere weiterzureichen.

Um dies zu ermöglichen, schreibt Cedric die Konstruktion während des Aufbaus am rechten Bildschirmrand mit. Hierfür existiert in der Klasse Darstellungsobjekt eine statische Methode, die ein Darstellungsobjekt in einen String umwandelt, wobei diese Strings folgendes Format haben:

<Name> (<Farbe >): <Konstruktionselement> (<Arg>)(<Pos>)

Hierbei ist

- Name: Die jeweilige Bezeichnung des Punktes beziehungsweise der Geraden, Strecke etc. auf dem Zeichenblatt.
- Farbe: Eine der Farbattribute rot, grün, blau, orange, weiß, schwarz, oder verdeckt.
- Konstruktionselement: Eines der folgenden Schlüsselworte:
  - Freier_Punkt
  - Auf_Geraden
  - Auf_Kreis
  - Auf_Kegelschnitt
  - Schnitt_Gerade_Gerade
  - Kreismittelpunkt
  - Streckenmittelpunkt
  - Rechtwinklig
  - Schnitt_Thales_Mittelsenkrechte
  - Fusspunkt
  - Parallelogrammpunkt
  - Schnitt_Kreis_Kreis_L
  - Schnitt_Kreis_Kreis_R
  - Schnitt_Gerade_Kreis_L
  - Schnitt_Gerade_Kreis_R
  - Verbindungsgerade
  - Verbindungsstrecke
  - Kegelschnitt

  - Kreis

- Arg: Die für dieses Konstruktionselement benötigten Argumente, wobei auch hier die jeweiligen Bezeichnungen der Argumente auf dem Zeichenblatt genommen werden, und die einzelnen Argumente durch Kommata voneinander getrennt sind.

- Pos: Dieser zweite Argumentteil wird nur für freie oder semifreie Punkte angegeben (das heißt wo die Lage des Objektes durch die Angabe von Konstruktionselement und Argumenten noch nicht eindeutig bestimmt ist). Er bezeichnet die Koordinaten des Punktes auf dem Bildschirm.

Die gleichen Strings werden auch verwendet, wenn die Konstruktion abgespeichert wird[1]. Entsprechend muss für das Laden einer Konstruktion eine Methode zur Verfügung stehen, die einen solchen String wieder in eine Konstruktion umwandelt. Diese Methode wird ebenfalls von der Klasse Darstellungsobjekt zur Verfügung gestellt, wobei dieser Textparser natürlich weit weniger kompliziert ist als der im letzten Abschnitt beschriebene Parser für die Gleichungen.

Für die Bildschirmkoordinaten gilt dabei noch Folgendes zu beachten: Erstens müssen bei einem semifreien Punkt die angegebenen Koordinaten *nicht* auf dem Argumentobjekt liegen. Hat man zum Beispiel einen Punkt auf einer Geraden, so können die Koordinaten auch einen Punkt über oder unter der Gerade beschreiben; der Punkt wird dann auf den nächsten Ort auf der Geraden verschoben.

Zweitens ist es auch nicht unbedingt nötig, überhaupt Bildschirmkoordinaten mit anzugeben (will man die Konstruktion zum Beispiel veröffentlichen, machen sich die zusammenhangslosen Zahlen doch etwas schlecht). Falls beim Laden einer Konstruktion das zweite Klammerpaar für die Bildschirmkoordinaten bei einem freien oder semifreien Punkt fehlt, so benutzt Cedric automatisch einen eingebauten Algorithmus, um die Punkte möglichst gut auf dem Bildschirm anzuordnen. Der gleiche Algorithmus wird auch verwendet, um das duale einer Konstruktion darzustellen (siehe den folgenden Abschnitt).

Dieser Verteilungsalgorithmus stellt zunächst fest, wie viele freie Punkte in der Konstruktion vorkommen. Handelt es sich um n Punkte, so werden diese Punkte nun auf einem regelmäßigen n-Eck angeordnet. Verschiedene semifreie Punkte auf einem Objekt werden möglichst gleichmäßig darauf verteilt.

6.3 Dualisieren

Wie eingangs schon erwähnt wurde, existiert in der projektiven Geometrie eine Dualität zwischen den einzelnen Konstruktionselementen. Dies wird allein schon durch die Darstellung in homogenen Koordinaten deutlich, wo zwischen der Darstellung eines Punktes und einer Geraden nicht unterschieden wird!

Auch zwischen den Anfragen der projektiven Geometrie besteht diese Dualität; so ist etwa zu der Anfrage, ob drei Punkte auf einer Geraden liegen, die

[1] Man bedenke hierbei, dass das Abspeichern unmöglich wird, wenn Cedric auf einem Server ohne Schreiberechtigung für den Benutzer läuft. Das Problem stellt sich natürlich nicht, wenn man Cedric etwa lokal auf dem eigenen Computer über die Java Virtuell Maschine laufen läßt.

Anfrage dual, ob sich drei Geraden in einem Punkt schneiden. Auf diese Weise lassen sich ganze geometrische Sätze dualisieren, wobei jeweils gilt, dass ein Satz genau dann wahr ist, wenn der duale Satz wahr ist [2].

Leider ist Cedric zwar in der Lage, allgemeine Kegelschnitte darzustellen und mit dem Validierer zu bearbeiten, aber es sind nur sehr begrenzte Operationen auf den Kegelschnitten möglich; insbesondere fehlt die Tangente an Kegelschnitte, die das Duale zu einem Punkt auf dem Kegelschnitt darstellt. Entsprechend ist auch die Dualisierung nur dann möglich, wenn die Konstruktion sogar nur eine Teilmenge der projektiven Konstruktionselemente enthält, und zwar exakt nur freie Punkte, Punkte auf Geraden, Schnittpunkte und Geraden. Zu diesen Elementen gibt es auch nur zwei relevante Abfragen, die Abfrage auf Kollinearität dreier Punkte beziehungsweise die zugehörige duale Abfrage, die Überprüfung auf einen gemeinsamen Schnitt dreier Geraden. Erfüllt die Konstruktion diese Bedingungen nicht, so wird der Dualisierungsauftrag vom Programm zurückgewiesen.

Aber auch innerhalb dieser Einschränkung gibt es einige interessante Sätze[2], die sich zu dualisieren lohnen.

Die strenge Orientierung an Punkten, die an anderen Stellen des Programmes eine wesentliche Vereinfachung darstellt, wird beim Dualisieren zu einem gewissen Problem. Will man einen freien Punkt dualisieren, so muss daraus eine freie Gerade entstehen; die gibt es jedoch im Aufbau von Cedric nicht! Statt dessen müssen zwei neue, freie Punkte gesetzt werden und die Gerade durch diese Punkte gezogen werden. Genauso muss ein Punkt, der auf einer Geraden liegt, in eine Gerade übersetzt werden, die durch einen festen Punkt geht. Hier muss also der entsprechende zweite Punkt erstellt werden. Beim Dualisieren werden diese Ersatzpunkte schwarz gezeichnet, um sie gut von der restlichen Konstruktion unterscheiden zu können.

Auf diese Weise steigt aber die Zahl der Konstruktionselemente und damit auch die Zahl der in der Konstruktion vorkommenden Variablen an, was sich im Allgemeinen unangenehm sowohl auf die Übersicht als auch auf die Rechenzeit des Beweisers auswirkt. Die zusätzlichen Punkte sind leider nicht ganz zu vermeiden[3]. Allerdings kann man ihre Zahl durch einen kleinen Trick verhältnismäßig klein halten.

Wir erschaffen dafür einen Zwischenzustand, indem auch Geraden verschiedene Freiheitsgerade zugeordnet bekommen können. Dieser Zwischenzustand bleibt nach außen unsichtbar und tritt nur in einem lokalen Objekt der Klasse Objektminister innerhalb der Dualisierungsmethode auf.

Eine Konstruktion wird aus dem normalen Zustand in diesen Zwischenzustand übersetzt, indem möglichst viele Freiheitsgerade von Punkten weg auf Geraden verschoben werden.

Um dies zu veranschaulichen, betrachte man folgendes Beispiel, das in Abbildung 6.1 dargestellt ist: Angenommen, wir haben zwei Punkte im Raum,

[2] siehe für ein Beispiel den Satz von Pappus oder den Satz von Desargues aus dem Anhang.

[3] Man bedenke den einfachen Fall, dass die Konstruktion nur aus einem Punkt besteht; dann muss im Dualen wenigstens eine Gerade auftauchen, für die aber wiederum mindestens zwei Punkte benötigt werden; man sieht, dass die Zahl der Konstruktionselemente notwendigerweise ansteigt.

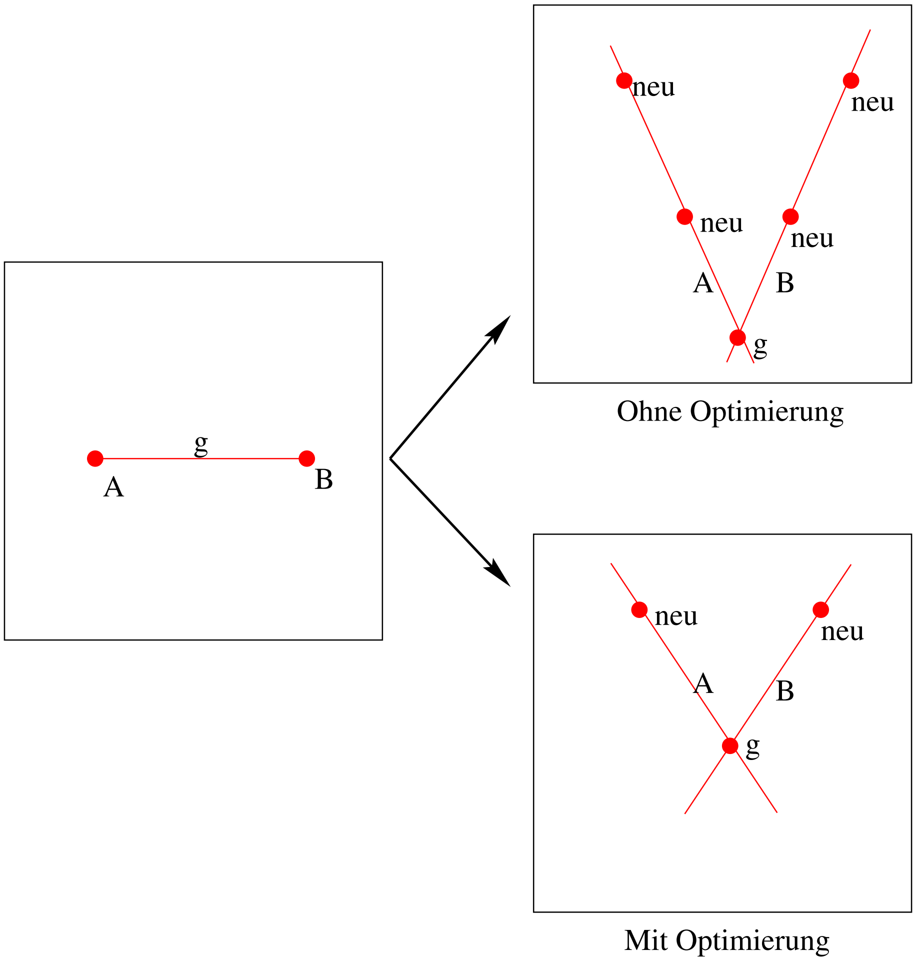

Abbildung 6.1: *Verschiebung der Freiheitsgerade beim Dualisieren*
Dadurch, dass die Gerade g zunächst jeweils einen Freiheitsgrad von den beiden Punkte, die sie definieren, erhält, verringert sich die Zahl der neu einzufügenden Punkte.

die durch eine Gerade verbunden sind. Dann hat die Gerade keine Freiheiten, während die Punkte jeweils zwei Freiheitsgrade haben. Wir erhalten aber das selbe Bild, wenn wir die Gerade als frei annehmen und die beiden Punkte als semifrei auf der Geraden fixiert betrachten. Jetzt hat die Gerade zwei Freiheitsgerade, während die Punkte nur noch jeweils einen Freiheitsgrad haben. Wenn wir die erste Darstellung mit brute force in Cedric dualisieren, so müssen wir vier neue Punkte für die beiden entstehenden Geraden einsetzen, während die ursprüngliche Verbindungsgerade zum Schnittpunkt dieser beiden neuen Geraden wird.

Nach der neuen Darstellung aber können wir die Gerade als einen freien Punkt darstellen und müssen nur noch zwei Punkte neu erschaffen, um die beiden semifreien Geraden darzustellen. Es ist leicht ersichtlich, dass beide Ergebnisse wieder die selbe Konstruktion darstellen (nämlich zwei freie Geraden und ihr Schnittpunkt, also genau das Duale zur ursprünglichen Konstruktion).

Die Methode dualizeDegreesOfFreedom der Klasse Objektminister verfährt genau nach diesem Schema. Sie geht zunächst alle Geraden durch, die in der Konstruktion vorkommen. Für diese wird überprüft, ob die die Gerade definierenden Punkte noch Freiheitsgerade haben (d.h. ob es sich nicht um Schnittpunkte handelt). Ist dies der Fall, so wird der entsprechende Punkt in der Konstruktionsreihenfolge hinter die Gerade verschoben, die Gerade wird als semifrei markiert, und der Punkt wird auf der Geraden fixiert. Es wird also aus einem freien Punkt ein Punkt auf einer Geraden und aus einem Punkt auf einer Geraden ein Schnittpunkt zweier Geraden. Für den anderen die Gerade definierenden Punkt wird genauso verfahren, und wenn möglich die Gerade dann für frei erklärt. Die dann folgende Dualisierung achtet sowohl bei Geraden als auch bei Punkten auf die verschiedenen möglichen Freiheitsstufen; nur bei freien oder teilweise freien Punkten müssen jetzt noch zusätzliche Punkte in die duale Konstruktion eingebaut werden.

Hat man eine Konstruktion vorgegeben und dualisiert diese zwei mal, so erhält man wieder den urspünglichen Satz (siehe [2]). Bedauerlicherweise gibt es trotz der eben genannten Modifikationen einige Sätze, bei der in Cedric mit jeder erneuten Dualisierung die Anzahl der Konstruktionselemente durch die zusätzlichen schwarzen Punkte ansteigt[4]. Beim Satz von Pappus (siehe Anhang) tritt dieser Anstieg jedoch beispielsweise nicht auf; zudem kann durch die schwarze Färbung der zusätzlichen Punkte das Prinzip der wiederholten Dualisierung trotz dieser zusätzlichen Elemente gut beobachten werden.

Für die Anordnung der dualen Konstruktion wird der selbe Algorithmus verwendet, der schon im letzten Abschnitt über die Interpretation textuell verfasster Konstruktionen vorgestellt wurde. In einigen Fällen verschwinden dabei konstruierte Schnittpunkte natürlich vom Zeichenblatt; man sollte unter Umständen ein wenig manuell nachbessern.

6.4 Der Validierer

Der in Kapitel 4 vorgestellte Beweiser erarbeitet mit symbolischen Werten exakte Beweise für die geometrischen Sätze. Der Umstand, die Wahrheit eines geometrischen Satzes auf diese Weise so einfach beweisen zu können, ist natürlich besonders für Mathematiker eine gewisse innere Befriedigung.

Für die praktische Anwendung ist es allerdings nicht immer nötig, einen echten, lückenlosen Beweis zu haben. Gerade der Computer ermöglicht es uns, uns auch ohne diese Beweise von der Richtigkeit eines Satzes schnell zu überzeugen, indem wir die freien und semifreien Punkte auf dem Bildschirm bewegen und das Resultat beobachten. Eine weitergehende Sicherheit können wir gewinnen, wenn wir den Computer automatisch die Punkte bewegen lassen und ihn überprüfen lassen, ob eine Anfrage für alle beobachteten Fälle stimmt. Ist die Anzahl dieser Fälle sehr hoch, so können wir auch ohne formalen Beweis getrost von der Richtigkeit des Satzes ausgehen.

[4]Der geneigte Leser möge zum Beispiel probieren, einen einzelnen Punkt wiederholt zu dualisieren. Das sich ergebene Muster entbhert nicht einer gewissen Schönheit.

Diese Aufgabe übernimmt in Cedric die Klasse Validierer. Auf der Oberfläche läßt sich zwischen dem Beweiser und dem Validierer wechseln; ist dort statt des Beweisers der Validierer aktiviert, so wird bei einer Anfrage automatisch die Konstruktion an diesen statt an den formalen Beweiser weitervermittelt.

Der Validierer arbeitet genau wie der Darstellungsraum mit den Erweiterungen der Klasse Darstellungsobjekt. Genau wie dieser benutzt auch er im Wesentlichen die Methoden der Klasse Darstellungspunkt, um die komplizierteren Konstruktionspunkte zu berechnen. Im Gegensatz zum Darstellungsraum allerdings arbeitet der Validierer nicht mit dem Objektminister, sondern mit einer linearen Verkettung der Darstellungsobjekte, und auch nicht mit bildschirmangepassten Positionen, sondern mit Einsetzungen im Bereich von -100 bis +100. Bei einer Einsetzung kann es passieren, dass die Anfrage nicht mehr möglich ist (etwa falls eine Anfrage über den Schnitt zweier Kreise gemacht wurden, die sich bei einer bestimmten Einsetzung gar nicht schneiden). Solche Fälle werden automatisch verworfen, während bei den restlichen gezählt wird, wie oft die Anfrage wahr und wie oft falsch ergab.

Abgebrochen wird der Vorgang, sobald eine Einsetzung falsch ergab oder eine vorher festgelegte Anzahl von Einsetzungen wahr ergaben. Für die verwendeten Konstruktionen und Hardware stellte sich eine Zahl von 1.000 Versuchen als gutes Mittelmaß zwischen Laufzeit und Sicherheit heraus.

Damit der Validierer in keiner Situation ewig laufen kann, existiert eine Abbruchbedingung für zu viele verworfene Versuche. Mussten mehr Versuche als ein vorgegebener Schwellenwert von Versuchen verworfen werden, weil die Anfrage nicht legal war, so gibt der Validerer auf und meldet eine Fehlermeldung zurück. Im Allgemeinen tritt ein solcher Fall nicht ein.

Die Beschreibungen der Objekte in den Darstellungsobjekten wird durch Fließkommazahlen gestaltet; entsprechend ist auch das Ergebnis der Rechnung im Validierer nur eine Fließkommazahl. Um zu überprüfen, ob zwei Fließkommazahlen gleich sind, muss eine gewisse Toleranzschwelle erlaubt sein. Sind zwei Zahlen weiter als diese Toleranzschwelle voneinander entfernt, so werden sie als ungleich betrachtet, ansonsten als gleich. Der Validierer erhebt, wie schon erwähnt, keinen Anspruch auf mathematisch exakte Beweiskraft, so dass diese Näherung hier kein Problem darstellt.

Anhang A

A.1 Kurzbedienungsanleitung von Cedric

Cedric ist als Applet für das WWW implementiert, das heißt, Cedric benötigt zum Laufen einen Internet-Browser oder eine Java Virtual Machine. Die Datei Cedric.html enthält einen einfachen Aufruf des Applets, über die Virtual Machine muss die Klasse 'Cedric.class' gestartet werden. Automatisch wird ein eigenes Fenster erzeugt, in dem Cedric dann unabhängig läuft. Insbesondere können im Browser in der Zwischenzeit andere Seiten besucht werden.

A.1.1 Die Oberfläche

Die Oberfläche (siehe Abbildung A.1) von Cedric enthält eine Menuleiste, von der aus alle Befehle aufgerufen werden können, darunter zwei Leisten mit Funktionsknöpfen und eine Leiste für ein Eingabefeld, in dem Gleichungen eingegeben werden könne. Mit einem Ausdruck der Form $ <name1>,<name2> $, indem name1 und name2 jeweils für Namen von Punkten aus der Konstruktion stehen, kann man in der Gleichung auf den Abstand der beiden Punkte in der Konstruktion verweisen. Der Ausdruck stellt immer den positiven Abstand der beiden Punkte dar, unabhängig von der Reihenfolge, in der sie angegeben werden.

Der größte Teil des Fensters wird vom Zeichenblatt ausgefüllt, das beim Start des Programmes leer ist. Rechts neben dem Zeichenblatt befinden sich noch einige weitere Elemente: Einmal das Logo von Cedric, darunter den Namen der aktuellen Konstruktion, dann eine Auswahlleiste, in der man Cedric zwischen dem Betrieb im formalen Beweiser und im Validierer umschalten kann, und schließlich eine Liste der verwendeten Konstruktionselemente als Programmcode.

Die einzelnen Konstruktionselemente und die Abfragen sind jeweils sowohl durch einen Funktionsknopf als auch über die Menuleiste erreichbar. Zusätzlich gibt es noch einige Knöpfe bzw. Menueinträge, die etwas genauerer Beschreibung bedürfen:

Datei Neu: Löscht die momentane Konstruktion.

Datei Speichern / Öffnen: Speichert beziehungsweise öffnet die Konstruktion, deren Namen im Namens-Textfeld steht. Die meisten Browser sind so eingestellt, dass diese Operation übers Netz nicht zur Verfügung steht.

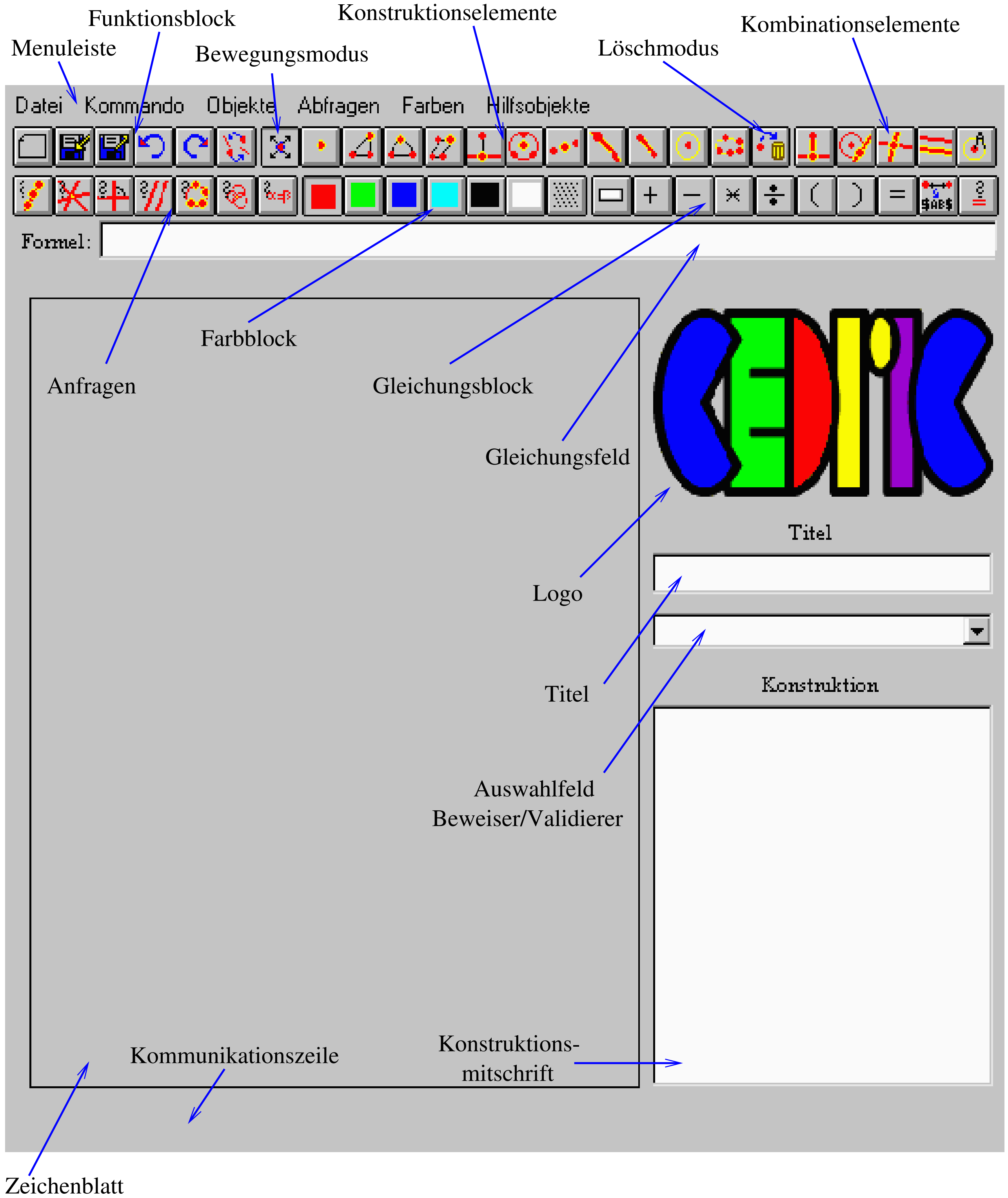

Abbildung A.1: *Die Oberfläche von Cedric*

Letzte Aktion zurücknehmen / Wiederherstellen: Diese beiden Knöpfe nehmen die letzte Aktion zurück bzw. stellen die zurückgenommene Aktion wieder her. Bei Aktionen, die ganze Konstruktionen erstellen (also zum Beispiel Dualisiseren oder Datei öffnen) können die einzelnen Konstruktionselemente zurückgenommen werden; hat man auf diese Weise die ganze Konstruktion zurückgenommen und betätigt den Knopf bei leerem Zeichenblatt noch einmal, so wird die Situation vor dem Dualisieren bzw. Öffnen wieder hergestellt.

Dualisieren: Diese Operation steht nur zur Verfügung, wenn ausschließlich Punkte und Geraden (bzw. Strecken) in der Konstruktion vorkommen. Dann wird die duale Konstruktion berechnet und angezeigt.

Objekt löschen: Mit dieser Operation läßt sich ein Konstruktionselement aus der Konstruktion entfernen. Alle anderen Konstruktionselemente, die von diesem Objekt abhängig waren, werden ebenfalls entfernt.

Punkt bewegen: In diesem Modus ist es möglich, einen Punkt mit der Maus zu greifen und an eine beliebige neue Stelle zu ziehen. Unter Umständen verschwinden bei diesen Bewegungen andere Objekte aus dem Sichtfeld; für den Beweiser sind diese Objekte natürlich nach wie vor vorhanden.

Objekte und Abfragen: Mit diesen Knöpfen kann man neue Darstellungsobjekte einfügen oder auf der bereits gezeichneten Konstruktion eine Anfrage durchführen.

Farbknöpfe: Hier kann man die aktuelle Zeichenfarbe einstellen. Wählt man mit der rechten Maustaste ein Objekt an, so wird dieses in der aktuellen Zeichenfarbe eingefärbt. Mit der am weitesten rechts liegenden Farbe läßt sich ein Objekt in die Hintergrundfarbe umwandeln, so dass es in der Ansicht einer komplexeren Konstruktion, für die es selber keinen Einfluss mehr hat, nicht stört.

Gleichungsblock: Diese Knöpfe haben keine Entsprechung in der Menuleiste. Sie dienen der einfacheren Gestaltung der Eingabe einer Gleichung. Beim Anwählen der Knöpfe mit den Rechenzeichen werden diese im Gleichungsfeld zum aktuellen String hinzugefügt. Der Knopf mit der Abbildung eines leeren Textfeldes löscht das Gleichungsfeld, der Knopf mit dem Fragezeichen über dem Gleichheitszeichen initiiert die Anfrage. Der Knopf mit der aufgezeichneten Strecke ermöglicht es, durch Wahl zweier Punkte A und B auf dem Zeichenbrett einen String A,B in das Gleichungsfeld aufzunehmen, der von Cedric als die Strecke zwischen A und B interpretiert wird.

A.1.2 Erstellen von Konstruktionen

Beim Programmstart ist das Zeichenblatt leer, und Cedric befindet sich im Normalzustand, dem Bewegungsmodus. Durch die Buttons für die Konstruktionselemente oder wahlweise auch durch das Menu kann man Cedric in verschiedene Modi bringen, die die Erstellung aller in Kapitel 2 aufgeführten Objekte ermöglicht. In der Kommunikationszeile unter dem Zeichenblatt wird dem Benutzer jeweils eine kurze Hilfe gegeben, in welchem Modus er sich im Moment befindet und welche Aktion als nächstes von ihm erwartet wird.

In der Farbleiste kann man die momentane Zeichenfarbe wählen; alle neuen Objekte werden dann in dieser Farbe gezeichnet. Drückt man die rechte Maustaste, während sich der Mauszeiger in der Nähe eines Objektes befindet, so erhält dieses Objekt die momentane Zeichenfarbe. Mit Hilfe der letzten Farbe (dem graukarierten Feld) läßt sich ein Objekt in die Hintergrundfarbe transformieren (genaugenommen wird es nicht mehr gezeichnet, damit es nicht andere Objekte überdeckt). Dieses Objekt ist aber immer noch genauso da wie vorher, kann bewegt oder gelöscht werden oder wieder mit der rechten Maustaste in eine sichtbare Farbe zurückgewandelt werden (vorausgesetzt, man weiß noch, wo sich das Objekt befand).

Der zweite Knopf von Links in der Buttonleiste für Konstruktionselemente, der dem Menupunkt 'Punkt setzen' entspricht, fasst allgemein das Setzen von Punkten zusammen. Klickt man in diesem Modus an eine freie Stelle des Zeichenblattes, wird an dieser Stelle ein neuer, freier Punkt gesetzt. Solange man den Knopf gedrückt hält, kann man den neuen Punkt auch noch beliebig verschieben.

Bewegt man den neuen Punkt dabei auf ein bestehendes Objekt (beziehungsweise klickt gleich in die Nähe eines bestehenden Objektes), so wird der Punkt auf diesem Objekt fixiert. Dabei wird der Punkt auf möglichst vielen nahen Objekten fixiert, und zwar zunächst soweit wie möglich auf Geraden oder Strecken, dann auf Kreisen und zuletzt auf Kegelschnitten. Bewegt man den Punkt in die Nähe eines bereits gesetzten Punktes, so wird der neue Punkt nicht gesetzt; der alte Punkt bekommt statt dessen die momentan gesetzte Zeichenfarbe.

Die anderen Modi erwarten jeweils, dass der Benutzer einige Punkte auswählt (etwa die beiden Endpunkte einer Strecke). Diese Punkte müssen nicht notwendigerweise vorher schon vorhanden gewesen sein, denn genau wie im Modus für neue Punkte kann der Benutzer einen neuen Punkt erschaffen, ihn mit festgehaltener Maustaste bewegen und an beliebiger Stelle absetzen. Dieser Punkt wird dann automatisch gesetzt und markiert. Setzt man den Punkt in die unmittelbare Nähe eines schon vorhandenen Punktes, wird dieser markiert. Wenn genug Punkte für das Konstruktionselement markiert sind, wird das neue Objekt eingezeichnet und alle Punkte demarkiert.

Mit den Kombinationselementen kann man sich einige Arbeit sparen; hier werden häufig gebrauchte Teilkonstruktionen, etwa das Legen einer Tangente an einen Kreis oder das Erstellen einer Parallelen zu einer Geraden, automatisch durchgeführt. Diese Kombinationen sind keine eigenen Konstruktionselemente, sondern umfassen nur eine Anreihung von anderen Konstruktionselementen.

Eine ähnliche Vereinfachung bieten die Hilfskonstruktionen, in denen vorgefertigte, geometrische Objekte zur Verfügung stehen. Die alte Konstruktion wird automatisch gelöscht (kann aber durch wiederholtes Betätigen des Rücknahmeknopfes wieder erstellt werden).

Jedes neue Objekt bekommt zusätzlich noch einen Eintrag in die lineare Auflistung der Konstruktionselemente am rechten Bildschirmrand.

In der Buttonleiste für die Konstruktionselemente befindet sich am rechten Rand ein Button, in dem ein Punkt dargestellt ist, der in einen Mülleimer verschoben wird. In diesem Modus kann jedes Konstruktionselement (natürlich nicht nur Punkte, sondern auch Geraden, Kreise und Kegelschnitte) gelöscht werden. Alle Objekte, die von dem gelöschten Objekt abhängig waren, werden ebenfalls gelöscht. Die Elemente lassen sich im Nachhinein mit dem Rücknahmeknopf wieder herstellen.

A.1.3 Durchführen eines Beweises

Ist die Konstruktion fertig, so kann man eine Anfrage starten. Achten Sie darauf, ob sie einen formalen Beweis wünschen (der naturgemäß etwas länger dauert), oder sich mit einer Abschätzung des Rechners über die Richtigkeit der Konstruktion zufrieden geben (siehe Kapitel 6.4). Entsprechend muss das Auswahlfeld über der Konstruktionsmitschrift eingestellt sein.

Falls Sie eine Gleichung eingeben wollen, bedienen Sie sich der in Kapitel 6.1 beschriebenen Syntax. Im Prinzip ist es dabei unnötig, Zeichen mit der Tastatur einzugeben. In der Buttonleiste für die Gleichungen finden sie alle nötigen Symbole, einen Knopf zum Löschen des Gleichungsfeldes und einen Knopf, der Cedric in den Gleichungseingabemodus versetzt. In diesem Modus wird erwartet, dass zwei Punkte auf dem Zeichenblatt markiert werden; für diese Punkte wird die Strecke in die Gleichung eingetragen. Der letzte Knopf startet schließlich die Gleichungsabfrage.

Für alle anderen Abfragen müssen ebenfalls Objekte markiert werden; man erreicht die entsprechenden Eingabemodi durch die Buttonleiste für die Anfragen (zweite Zeile der Buttonleisten, ganz links) oder durch den Menupunkt 'Anfragen'. Handelt es sich bei den zu markierenden Objekten um Punkte, so gelten die selben Zusatzvereinfachungen wie oben für Konstruktionspunkte beschrieben. Sobald die nötige Anzahl an Punkten markiert ist, öffnet Cedric den Beweiser und ein eigenes Dialogfenster, in dem eine Uhr die Zeitdauer des Beweises misst. Durch drücken der Taste 'Abbrechen' können sie den Vorgang jederzeit vorzeitig beenden. Während des Beweises können sie die Konstruktion bearbeiten, ohne dass der Beweis dadurch gestört wird; allerdings wird der Beweis automatisch abgebrochen, falls sie vor seinem Ende einen neuen Beweis starten. Im selben Dialogfenster werden Ihnen auch eventuelle Fehler oder das Ergebnis des Satzes mitgeteilt.

Sollte der Server, auf dem Cedric läuft, es zulassen, oder haben sie Cedric bei sich zu Hause installiert, so empfiehlt es sich, von der Möglichkeit des Abspeicherns Gebrauch zu machen. Sie können dann die Konstruktion im Nachhinein noch einmal nachladen oder den Programmtext problemlos mit jedem Texteditor bearbeiten und ausdrucken (an die Dateien wird automatisch die Endung '.ced' angehängt; es handelt sich um den Programmtext in ASCII-Format).

A.2 Konstruktionsbeispiele

Unter anderem wurde Cedric mit den folgenden Sätzen getestet. Die Laufzeiten für die einzelnen Sätze finden sich am Schluss dieses Abschnittes. Neben diesen

Beispielen wurde Cedric noch an einer Reihe von falschen Sätzen ausprobiert, auf deren Aufzählung hier verzichtet wird.

Die Sätze bilden einen guten Grundstock, um den Umgang mit Cedric zu üben und ein gewisses Gefühl für dessen Leistungsfähigkeiten und Grenzen zu bekommen. Mit der Zeit kann es natürlich etwas langweilig werden, ständig diese Sätze einzugeben; dann ist wahrscheinlich der richtige Zeitpunkt erreicht, selber etwas auszuprobieren.

Die hier angegebenen Sätze sind eine Auswahl von geeigneten Sätzen. Die meisten stammen aus anderen Arbeiten zu geometrischen Beweisern wie dem Geometry Expert [10] oder der Diplomarbeit von Eva Rohnert [9], einige stammen auch aus Vorschlägen von Benutzern des Programmes.

A.2.1 Satz von Pappus

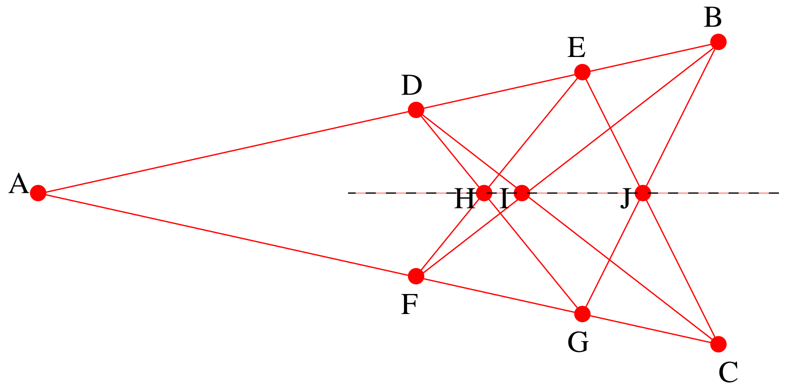

Abbildung A.2: *Satz von Pappus*

Seien $\overline{AB}$ und $\overline{AC}$ zwei Geraden mit jeweils drei Punkten D, E und B auf der einen, F, G und C auf der anderen Geraden. Sei H der Schnitt von $\overline{FE}$ und $\overline{DG}$, I der Schnitt von $\overline{FB}$ und $\overline{DC}$ und J der Schnitt von $\overline{EC}$ und $\overline{BG}$. Dann sind H, I und J kollinear.

A.2.2 Satz von Desargues

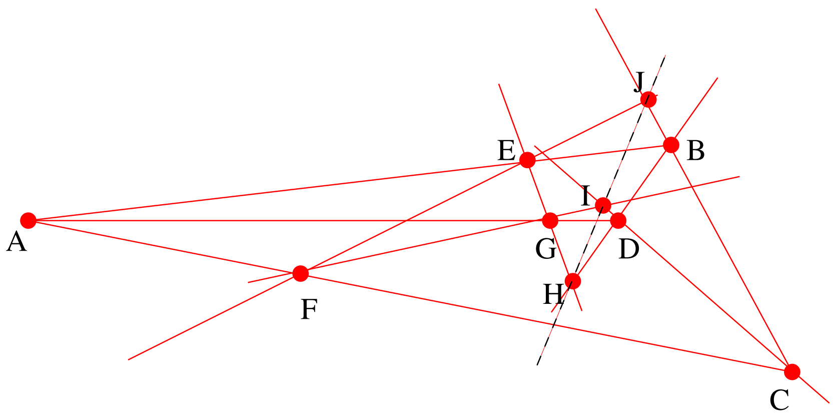

Abbildung A.3: *Satz von Desargues*

Sei A ein freier Punkt und BCD und EFG zwei Dreiecke auf drei von A ausgehenden Strahlen. Seien weiterhin $H = \overline{BD} \cap \overline{EG}$, $I = \overline{CD} \cap \overline{FG}$ und $J = \overline{CB} \cap \overline{FE}$. Dann sind H, I und J kollinear.

A.2.3 Satz von Pascal auf einem Kreis

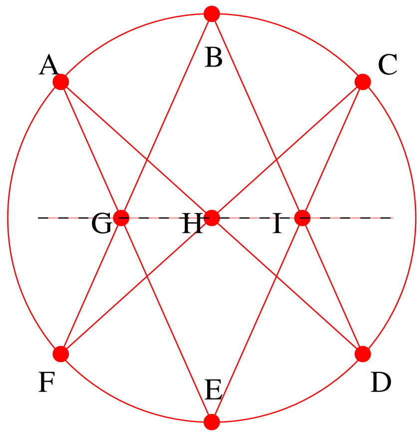

Abbildung A.4: *Satz von Pascal auf einem Kreis*

Seien A, B, C, D, E, und F sechs Punkte auf einem Kreis. Sei $G = \overline{BF} \cap \overline{AE}$, $H = \overline{AD} \cap \overline{CF}$ und $I = \overline{CE} \cap \overline{BD}$. Dann sind G, H und I kollinear.

A.2.4 Der allgemeine Satz von Pascal

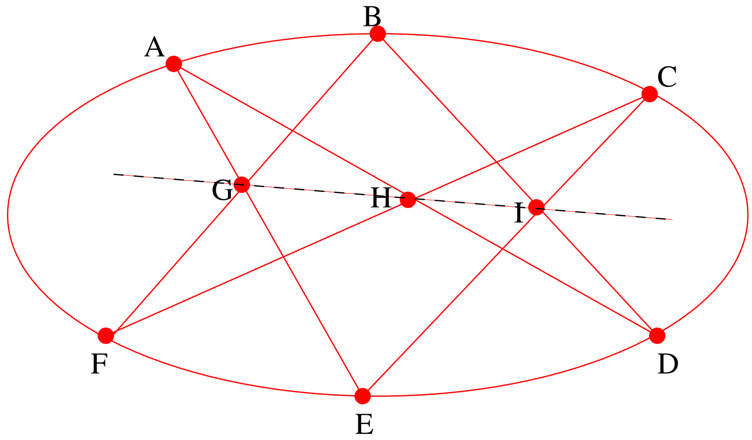

Abbildung A.5: *Der allgemeiner Satz von Pascal*

Seien A, B, C, D, E, und F sechs Punkte auf einem beliebigen Kegelschnitt. Sei $G = \overline{BF} \cap \overline{AE}$, $H = \overline{AD} \cap \overline{CF}$ und $I = \overline{CE} \cap \overline{BD}$. Dann sind G, H und I kollinear.
Diesen Satz kann Cedric nur validieren.

A.2.5 Satz von Cantor

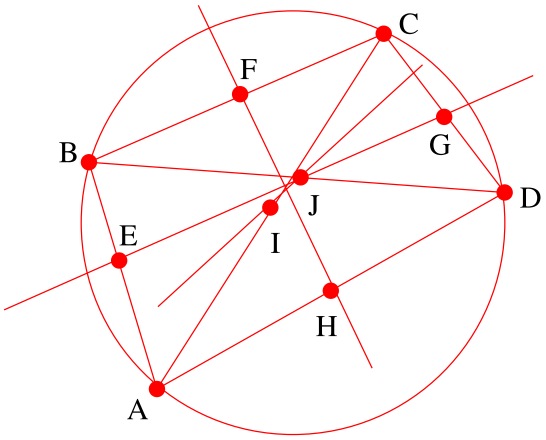

Abbildung A.6: *Satz von Cantor*

Sei $ABCD$ ein Viereck auf einem Kreis. Seien E, F, G, H, I und J jeweils die Mittelpunkte der vier Seiten und der beiden Diagonalen. Dann schneiden sich $\overline{EG}$, $\overline{FH}$ und $\overline{IJ}$ in einem Punkt.

A.2.6 Der Schmetterlingssatz

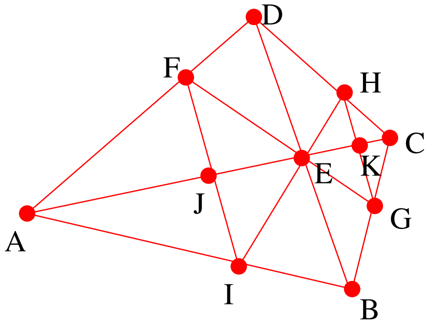

Abbildung A.7: *Schmetterlingssatz*

Sei $ABCD$ ein beliebiges Viereck und E der Schnittpunkt der Diagonalen. Seien F und H jeweils Punkte auf $\overline{AD}$ und $\overline{DC}$. Sei $G = \overline{FE} \cap \overline{BC}$ und $I = \overline{HE} \cap \overline{AB}$. Seien J und K jeweils die Schnitte von $\overline{FI}$ und $\overline{HG}$ mit der Diagonalen $\overline{AC}$. Dann gilt $\overline{EJ} \cdot \overline{KC} \cdot \overline{AE} = \overline{AJ} \cdot \overline{EK} \cdot \overline{CE}$.

A.2.7 Viereckssatz

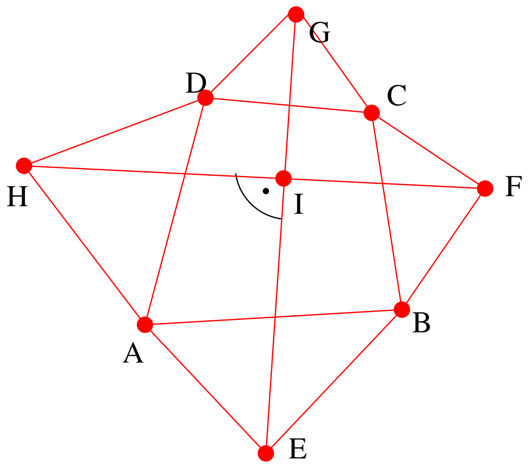

Abbildung A.8: *Satz von rechtwinkligen Dreiecken über Viereckseiten*

Sei $ABCD$ ein beliebiges Viereck und E, F, G und H jeweils die Eckpunkte eines gleichschenkligen, rechtwinkligen Dreieckes auf den Seiten. Dann stehen $\overline{GE}$ und $\overline{HF}$ aufeinander senkrecht und haben die gleiche Länge.

A.2.8 Satz über die Diagonalen im Parallelogramm

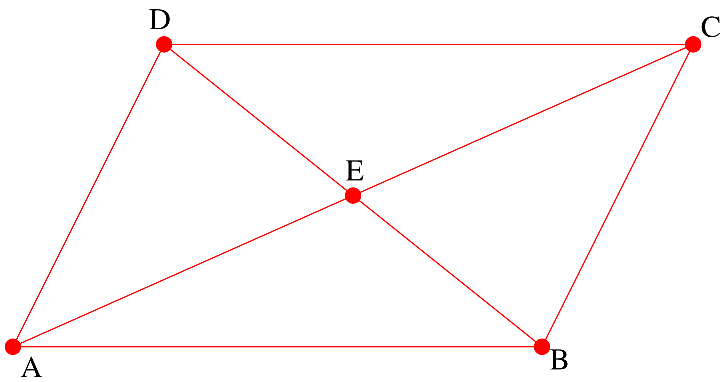

Abbildung A.9: *Satz über die Diagonalen im Parallelogramm*

Der Schnittpunkt der Diagonalen im Parallelogramm teilt die Diagonalen im Verhältniss 1:1.

A.2.9 Satz von den Höhen im Dreieck

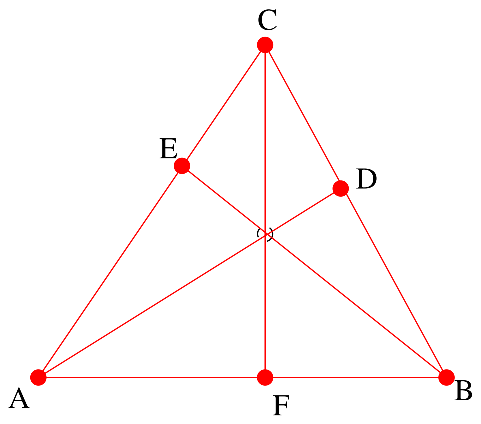

Abbildung A.10: *Satz von den Höhen im Dreieck*

Die drei Höhen eines Dreieckes schneiden sich in einem Punkt.

A.2.10 Satz von den Mittelsenkrechten im Dreieck

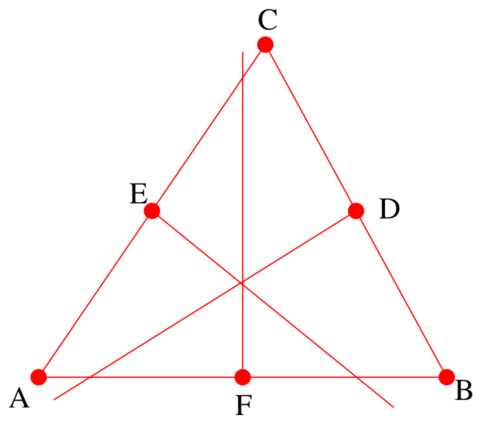

Abbildung A.11: *Der Satz von den Mittelsenkrechten im Dreieck*

Die drei Mittelsenkrechten eines Dreieckes schneiden sich in einem Punkt.

A.2.11 Satz von den Seitenhalbierenden im Dreieck

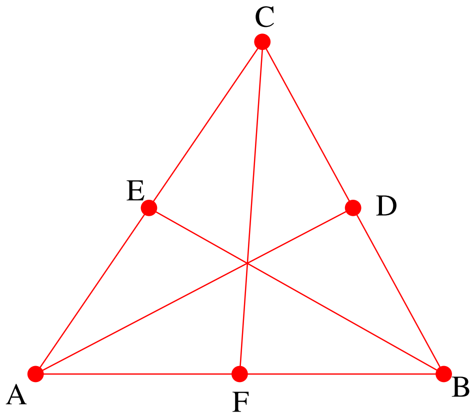

Abbildung A.12: *Satz von den Seitenhalbierenden im Dreieck*

Die drei Seitenhalbierenden eines Dreieckes schneiden sich in einem Punkt.

A.2.12 Satz der Trilinearität

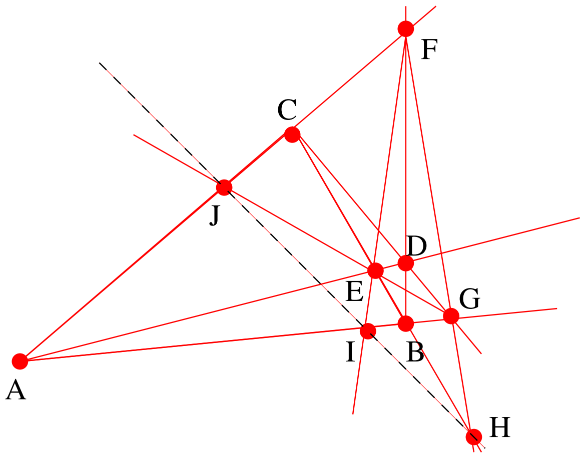

Abbildung A.13: *Satz der Trilinearität*

Sei ABC ein beliebieges Dreieck und D ein weiterer freier Punkt. Sei $E = \overline{AD} \cap \overline{BC}$, $F = \overline{BD} \cap \overline{AC}$ und $G = \overline{CD} \cap \overline{AB}$. Schneidet man jetzt die Seiten des neuen Dreickes EFG mit jeweils einer Seite des alten Dreieckes ABC, so erhält man $H = \overline{BC} \cap \overline{FG}$, $I = \overline{AB} \cap \overline{FE}$ und $J = \overline{AC} \cap \overline{GE}$. Diese drei Punkte I, J und H sind kollinear.

A.2.13 Satz von Ceva

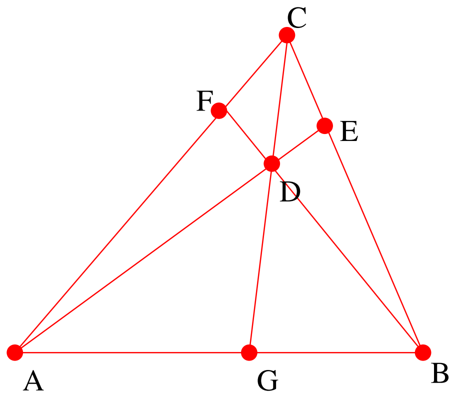

Abbildung A.14: *Der Satz von Ceva*

Sei ABC ein beliebiges Dreieck und D ein weiterer, freier Punkt. Sei $E = \overline{AD} \cap \overline{BC}$, $F = \overline{BD} \cap \overline{AC}$ und $G = \overline{CD} \cap \overline{AB}$. Dann ist $\overline{AG} \cdot \overline{BE} \cdot \overline{CF} = \overline{GB} \cdot \overline{EC} \cdot \overline{FA}$.

A.2.14 Satz vom sechsten Hexagonpunkt

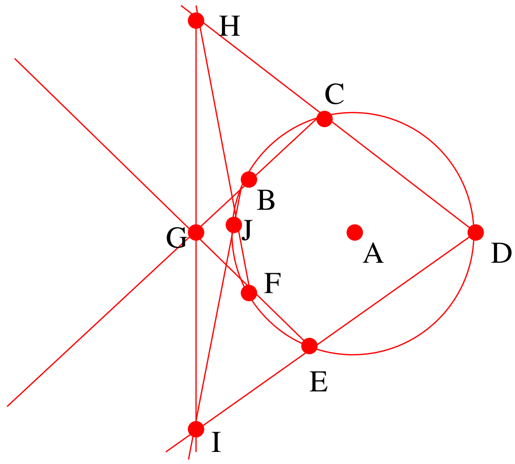

Abbildung A.15: *Der Satz des 6. Hexagonpunkt*

Liegen fünf Punkte B, C, D, E und F eines Hexagons auf einem Kreis und liegen die drei Schnittpunkte gegenüberliegender Seiten auf einer Geraden, so liegt der sechste Punkt J des Hexagons auch auf dem Kreis.
Man bemerke die enge Verwandtschaft zum Satz von Pascal.

A.2.15 Satz vom Fußwinkel

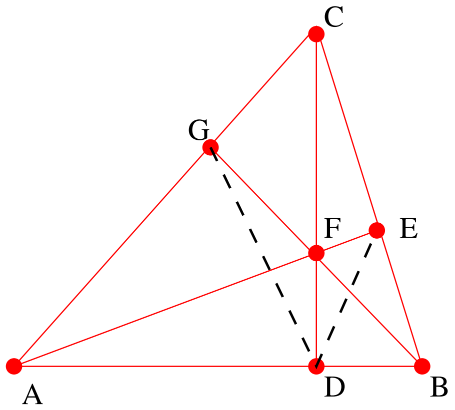

Abbildung A.16: *Der Satz vom Fußwinkel*

Sei ABC ein beliebiges Dreieck und D der Fußpunkt von C auf $\overline{AB}$. Sei E ein beliebiger Punkt auf $\overline{BC}$ und F der Schnittpunkt von $\overline{AE}$ und der Höhe $\overline{CD}$. Sei zuletzt noch $G = \overline{BF} \cap \overline{AC}$. Dann sind die Winkel GDC und CDE gleich.

A.2.16 Die Gaußlinie

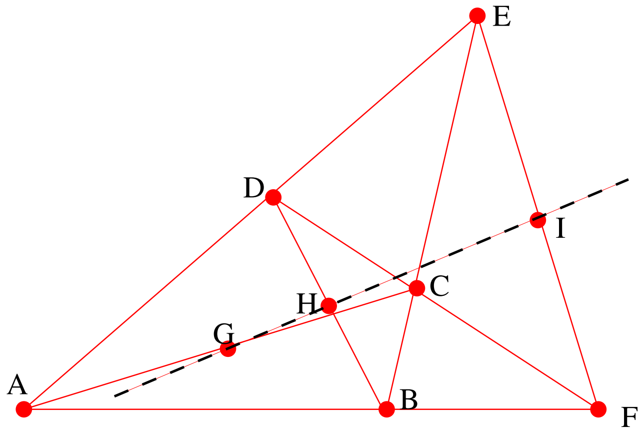

Abbildung A.17: *Satz von der Gaußlinie*

Sei $ABCD$ ein beliebiges Viereck. Seien E und F jeweils die Schnitte gegenüberliegender Seiten, also $E = \overline{AD} \cap \overline{BC}$ und $F = \overline{AB} \cap \overline{CD}$. Seien G, H und I jeweils die Mittelpunkte der Diagonalen $\overline{AC}$ und $\overline{DB}$ beziehungsweise der neuen Strecke $\overline{EF}$. Dann sind G, H und I kollinear.

A.2.17 Affinitätssatz

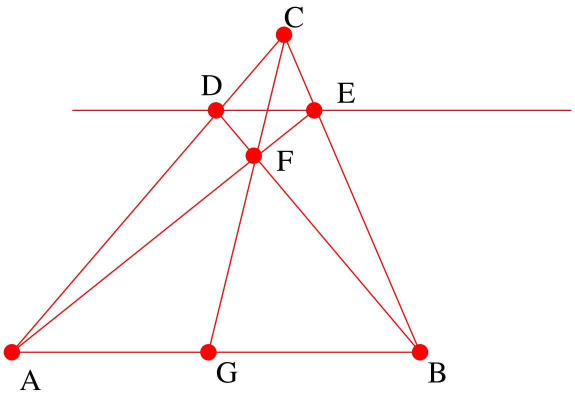

Abbildung A.18: *Der Affinitätssatz*

Sei ABC ein beliebiges Dreieck, D ein Punkt auf $\overline{AC}$ und E der Schnitt von $\overline{BC}$ mit einer Parallelen zu $\overline{AB}$ durch D. Sei weiterhin $F = \overline{AE} \cap \overline{BD}$ und $G = \overline{AB} \cap \overline{CF}$. Dann teilt G die Strecke $\overline{AB}$ im Verhältnis 1:1.

A.2.18 Satz von der Euler-Geraden

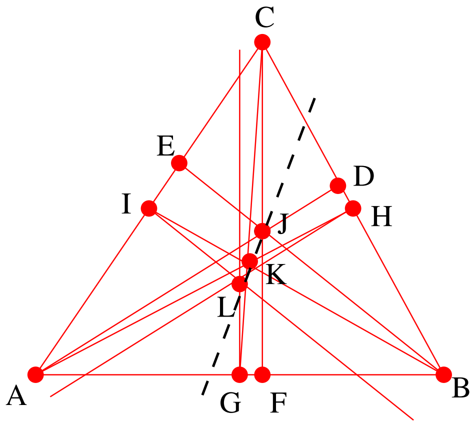

Abbildung A.19: *Die Euler-Gerade*

Sei ABC ein beliebiges Dreieck. Sei J der Schnittpunkt der Höhen, K der Schnittpunkt der Seitenhalbierenden und L der Schnittpunkt der Mittelsenkrechten des Dreieckes. Dann liegen J, K und L auf einer Geraden, und K teilt die Strecke $\overline{JL}$ im Verhältniss 2:1.

A.2.19 Satz von Menelaus

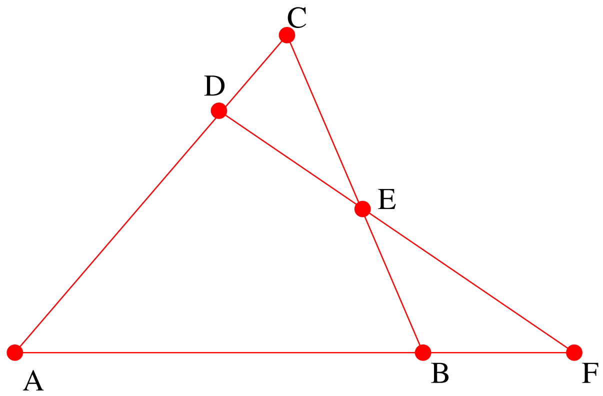

Abbildung A.20: *Der Satz von Menelaus*

Sei ABC ein beliebiges Dreieck, D ein Punkt auf $\overline{AC}$, E ein Punkt auf $\overline{BC}$ und F der Schnitt von $\overline{AB}$ mit $\overline{DE}$. Dann ist $\overline{AF} \cdot \overline{BE} \cdot \overline{CD} = \overline{BF} \cdot \overline{CE} \cdot \overline{AD}$.

A.2.20 Das einbeschriebene Parallelogramm

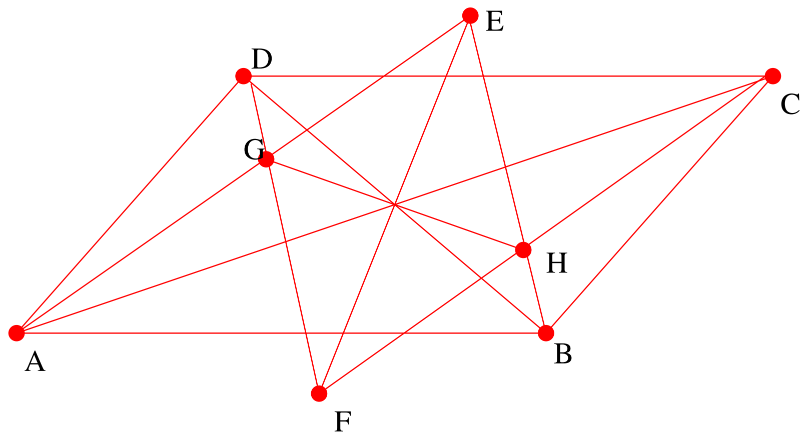

Abbildung A.21: *Satz vom einbeschriebenen Parallelogramm*

Sei $ABCD$ ein Parallelogramm und E ein freier Punkt. Sei F der vierte Punkt des Parallelogrammes aus A, E und C. Sei $G = \overline{AE} \cap \overline{DF}$ und $H = \overline{BE} \cap \overline{CF}$. Dann bilden $FHEG$ ein Parallelogramm, dessen Diagonalen sich mit den Diagonalen von $ABCD$ in einem Punkt schneiden.

A.2.21 Satz von Simson

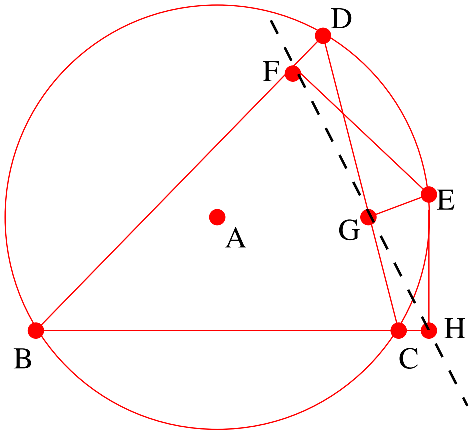

Abbildung A.22: *Satz von Simson*

Sei BCD ein Dreieck auf einem Kreis und E ein weiterer Punkt des Kreises. Seien F, G und H jeweils die senkrechten Projektionen von E auf die Dreieckseiten. Dann sind F, G und H kollinear.

A.2.22 Satz vom Schnitt der Diagonalen im Trapez

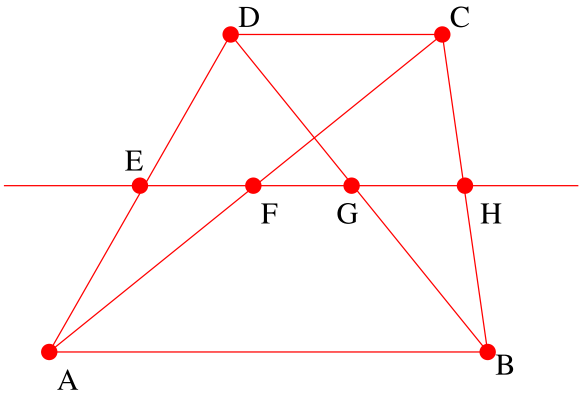

Abbildung A.23: *Satz vom gleichmäßigen Schnitt der Trapez-Diagonalen*

Sei $ABCD$ ein Trapez. Sei E ein beliebiger Punkt auf $\overline{AD}$ und F, G und H die jeweiligen Schnitte einer Parallelen von $\overline{AB}$ durch E mit den beiden Diagonalen und der Seite $\overline{CB}$. Dann ist $\overline{EF} = \overline{GH}$.

A.2.23 Streckenverhältnisse im Dreieck auf einem Kreis

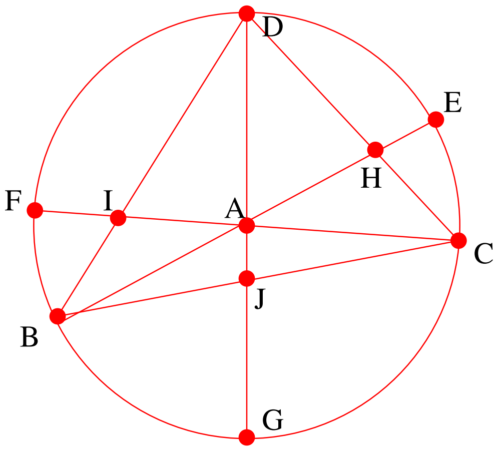

Abbildung A.24: *Satz von den Streckenverhältnissen im Dreieck auf einem Kreis*

Sei A der Mittelpunkt eines Kreises, auf dem ein Dreieck BCD liegt. Seien E, F und G jeweils die den Punkten B, C und D auf dem Kreis gegenüberliegende Punkte. Sei $H = \overline{AB} \cap \overline{CD}$, $I = \overline{AC} \cap \overline{BD}$, und $J = \overline{AD} \cap \overline{BC}$. Dann ist $\frac{\overline{HE}}{\overline{BH}} + \frac{\overline{IF}}{\overline{CI}} + \frac{\overline{JG}}{\overline{DJ}} = 1$.

A.2.24 Satz von Quadraten auf einem Dreieck

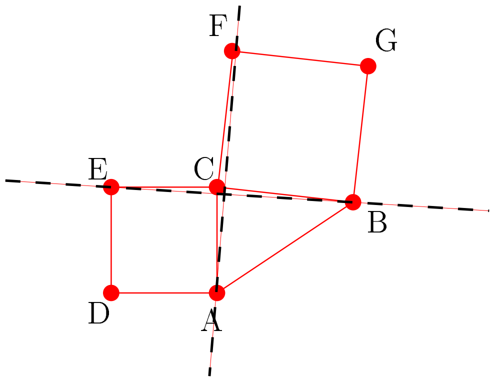

Abbildung A.25: *Satz von Quadraten auf Dreiecksseiten*

Sei ABC ein Dreieck und $ACED$ und $CBGF$ zwei auf den Seiten AC und CB errichtete Quadrate. Dann stehen $\overline{AF}$ und $\overline{BE}$ senkrecht aufeinander.

A.2.25 Satz von Miguel

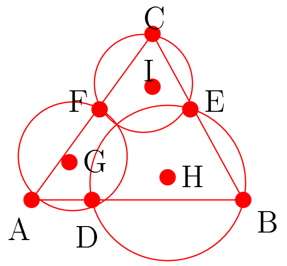

Abbildung A.26: *Satz von Miguel*

Sei ABC ein beliebiges Dreieck und D, E und F drei Punkte auf jeweils einer Dreiecksseite. Seien weiterhin G, H und I jeweils die Umkreismittelpunkte um ADF, BDE und CEF. Dann schneiden sich diese drei Umkreise in einem gemeinsamen Punkt.

A.2.26 Satz vom Projektionsparallelogramm

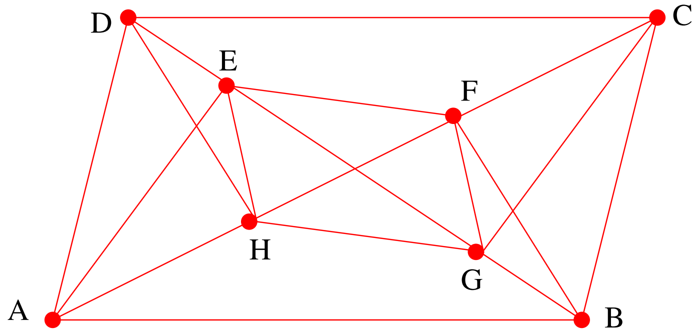

Abbildung A.27: *Satz vom Projektionsparallelogramm*

Sei $ABCD$ ein Parallelogramm und D, E, F und G jeweils die Projektionen der Eckpunkte auf die Diagonalen. Dann bildet $DEFG$ wieder ein Parallelogramm.

A.2.27 Satz von berührenden Quadraten

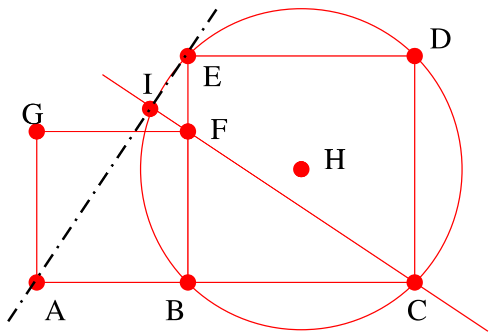

Abbildung A.28: *Satz von berührenden Quadraten*

Seien $ABFG$ und $BCDE$ zwei sich berührende Quadrate und H der Mittelpunkt des Quadrates $BCDE$. Sei I der Schnittpunkt eines Kreises um H durch B und der Geraden $\overline{CF}$. Dann liegen A, I und E auf einer Geraden.

A.2.28 Satz vom erzeugten Quadrat

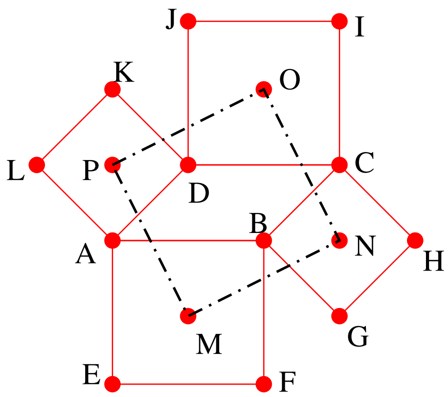

Abbildung A.29: *Satz vom erzeugten Quadrat*

Sei $ABCD$ ein beliebiges Parallelogramm und M, N, O und P die Mittelpunkte der auf den Parallelogrammseiten aufgespannten Quadrate. Dann bilden $MNOP$ wieder ein Quadrat.

A.3 Laufzeiten der Konstruktionsbeispiele

Die Laufzeiten dieser Konstruktionen finden sich in folgender Tabelle. Dabei sei darauf zu achten, dass die Laufzeiten in Abhängigkeit vom Rechner und von der Eingabe schwanken können. Die hier angegebenen Laufzeiten sind auf einem Intel Celeron mit 350 MHz gemessen wurden.

Satz	Laufzeit	
Satz von Pappus	8	ms
Satz von Desargues	8	ms
Satz von Pascal (Kreis)	50	ms
Satz von Cantor	70	ms
Satz der Trilinearität	4	ms
Viereckssatz (rechter Winkel)	7	ms
Viereckssatz (Längen)	14	ms
Sechster Hexagonpunkt	48	s
Höhen im Dreieck	11	ms
Mittelsenkrechten im Dreieck	11	ms
Seitenhalbierenden im Dreieck	8	ms
Satz von der Euler-Geraden	3	ms
Satz von Ceva	40	ms
Diagonalenschnitt im Parallologramm	8	ms
Satz von Menelaus	40	ms
Affinitätssatz	6	ms
Satz von der Gauss-Geraden	3	ms
Satz von Simson	60	ms
Satz vom Fusswinkel	40	ms
Streckenverhältnisse im Kreisdreieck	305	s
Diagonalen im Trapez	9	ms
Schmetterlingssatz	3	s
Einbeschriebenes Parallelogramm	20	ms
Quadrate auf einem Dreieck	5	ms
Satz von Miguel	258	s
Projektionsparallelogramm	14	ms
berührende Quadrate	3	ms
erzeugtes Quadrat	6	ms

In dieser Tabelle fallen natürlich einige Ausreißer besonders auf, und zwar der Satz von sechsten Hexagonpunkt, der Satz von Streckenverhältnissen im Kreisdreieck und der Satz von Miguel, die im Minutenbereich statt, wie die anderen, im Millisekundenbereich liegen. Bei diesen Sätzen sind es besonders die auftretenden Kreise (bzw. Strecken), die die Komplexität in die Höhe treiben.

A.4 Danksagung

Im Kommunikationszeitalter sind viele Dinge in Veränderung begriffen, wovon besonders die Methoden zur Beschaffung von Informationen betroffen sind. So basiert auch diese Arbeit nicht hauptsächlich auf gedruckten Materialien, sondern auf Informationen aus dem weltweiten Netz und den Gesprächen mit Kommilitonen und Freunden. Diese befähigten mich dazu, Cedric zu programmieren und diese Arbeit zu verfassen. Auf diese Weise begründet sich die Nähe der Danksagung zu dem Literaturverzeichnis.

Besonders danke ich Herrn Professor Doktor Hotz, nicht nur für das interessante Thema, sondern insbesondere für seine Zeit, die er mir in vielen Gesprächen geopfert hat, und das Vertrauen, das er in mich setzte.

In Bezug auf zeitaufwendige Gespräche danke ich auch Herrn Professor Doktor Berger, der mir im Aufbau von Cedric in einigen, freundlichen Gesprächen viele mathematische Hilfestellungen geben konnte. Besonders danke ich ihm für seine zusätzlich in den Semesterferien gehaltene Vorlesung über projektive Geometrie, die mir einen wesentlich weiteren Einblick in dieses Themengebiet ermöglichte.

Ebenso danke ich Doktor Elmar Schömer für die Motivation, die er mir gegeben hat, die inhaltlichen Tips und seine Geduld mit mir. Dieser Dank gilt natürlich ebenso für alle anderen Mitglieder des Lehrstuhls von Professor Hotz, die mir durch die freundliche Arbeitsatmosphäre diese Arbeit ermöglichten.

Weiterhin gilt mein Dank Björn Stahmer für seine unermüdliche Suche nach Fehlern (auch wenn mir sein ständiger Erfolg in dieser Angelegenheit wenig gefiel), Tobias Gärtner und Holger Cröni für ihre Tips besonders in den mathematischen Anteilen dieser Arbeit. Außerdem danke ich Manuel Bodirsky, Jan Schwinghammer, Stefan Strobel, Martin Struwe und Markus Zacharski dafür, dass sie sich die Zeit genommen haben, mich auf meine Denkfehler hinzuweisen und diese Arbeit noch einmal für mich durchzusehen.

Nicht zuletzt danke ich allen, die mir bei der ästhetischen Gestaltung der Oberfläche von Cedric geholfen haben, und hierbei besonders Marco Kuhlmann, der das Logo von Cedric entworfen und gezeichnet hat.

Literaturverzeichnis

[1] E.W. Elcock: Representation of Knowledge in a Geometry Machine. Machine Intelligence 8, 1977.

[2] Gerd Fischer: Analytische Geometrie. Vieweg-Verlag, 1979.

[3] E. Zeidler, W. Hackbusch, R. Schwarz: Teubner-Taschenbuch der Mathematik. Teubner-Verlag, 1996.

[4] Hans Rudolf Schwarz: Numerische Mathematik. Teubner-Verlag, 1997.

[5] Larry Smith: Linear Algebra, 3rd Edition. Springer-Verlag, 1998.

[6] D. Hilbert: Grundlagen der Geometrie. Teubner-Verlag.

[7] W. T. Wu: Geometric Theorems Proving in Geometry.

[8] S. C. Chou: GEO-Prover - A Geometry Theorem Prover Developed at UT.

[9] Eva Rohnert: Eine Benutzerschnittstelle zum automatischen Beweisen von Sätzen aus der Geometrie, Universität des Saarlandes, 1997.

[10] S.C. Chou, X.S. Gao, J.Z. Zhang: Machine Proofs in Geometry. World Scientific, Singapore, 1994.

[11] Jürgen Richter-Gebert, Ulrich H. Kortenkamp: The Interactive Geometry Software Cinderella. Springer-Verlag, 1999.

[12] David Cox, John Little, Donal O'Shea: Ideals, Varieties, and Algorithms. Springer-Verlag, 1997.