Interaktive Mathematik im Browser

Werkzeuge, Bibliotheken und Programmierumgebungen

Von der Universität Bayreuth
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

vorgelegt von

Michael Gerhäuser

aus

Neuendettelsau

Gutachter: Prof. Dr. Peter Baptist und Prof. Dr. Volker Ulm

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Einsatz von Informations- und Kommunikationstechnik (ICT) im schulischen Umfeld begann bereits in den 70er Jahren mit der Ablösung von Rechenschiebern und Logarithmentafeln durch Taschenrechner. In den 80er Jahren hielten in Schulen die ersten Personalcomputer (PC) in speziellen Computerräumen Einzug. Aufgrund des beschränkten und geteilten Zugangs zu diesen Räumen sind die PCs im Unterricht nicht immer verfügbar. Notebooks sollten Abhilfe schaffen. Allerdings sind diese in der Anschaffung relativ teuer und benötigen viel Platz.

Daher sind heute nach wie vor (graphikfähige) Taschenrechner neben Interaktiven Whiteboards (IWB) und dazugehörigen PC die einzigen omnipräsenten ICT Geräte im Klassenzimmer. Das könnte sich jetzt mit der Verbreitung von Tablets und anderen mobilen Geräten ändern.

Vom Nutzen von Tablets, insbesondere für den Unterricht, war Computerpionier und Informatiker Alan Kay bereits 1972 überzeugt[19]. Durch ein geringes Gewicht und handlichen Größen, einer langen Akkulaufzeit und den mittlerweile sehr günstigen Anschaffungspreisen sind Tablets sowohl Notebooks als auch PCs überlegen und machen diese zu idealen Begleitern im Unterricht. Sie bieten Schülern die Möglichkeit, ICT aktiv im Unterricht zu nutzen, anstatt passiv über Präsentationen am IWB [24].

Parallel zur breiten Akzeptanz von Tablets beginnen E-Books klassische, gedruckte Bücher abzulösen. Neben den physikalischen und organisatorischen Vorteilen Gewicht, Aktualisierung und Distribution bieten E-Books

eine neue, interaktive Ebene, direkt in das Medium integriert. Digitale Zusatzmaterialien und klassische Buchinhalte selbst müssen nicht mehr getrennt in unterschiedlichen Medien betrachtet werden. Sie sind im E-Book integriert und sofort und ohne Medienwechsel verfügbar. Die Grenzen zwischen Anwendungen, Büchern und Webseiten verschwimmen.

In einigen Ländern gibt es Projekte mit dem Ziel, Schulbücher zu digitalisieren und teilweise auch kostenfrei zur Verfügung zu stellen. CK-12[1] in Palo Alto, CA in den Vereinigten Staaten ist eine gemeinnützige Organisation, die Lehrern die Möglichkeit gibt, eigene Schulbücher nach dem Baukastenprinzip zusammenzustellen. In Slowenien finanziert das Ministerium für Bildung, Wissenschaft und Unterricht[2] das Projekt e-um[3]. Bereits 2011 kündigte die Regierung in Südkorea die Entwicklung von digitalen Schulbüchern bis 2015 an.

Diese neuen Technologien sind ohne entsprechende Software nicht viel wert. Bislang im Mathematikunterricht eingesetzte Software ist auf den neuen Geräten nicht direkt einsetzbar. Die Gründe dafür sind zum einen bei den technischen Einschränkungen zu suchen. In den Browsern der meisten mobilen Touchgeräte steht weder Java noch Flash zur Verfügung. Zum anderen sind klassische Oberflächen mit Menüs und Toolbars auf die Bedienung mit einer Maus ausgelegt, einem sehr präzisen Eingabegerät. Hier bietet es sich insbesondere an, mit neuen Konzepten das User Interface so anzupassen, dass die kognitive Belastung der Nutzer reduziert wird[25].

Die seit Jahren abnehmende Unterstützung von Java im Web-Browser macht die technische Überarbeitung existierender Software nötig. Besonders im Bereich der Dynamischen Geometrie Systeme (DGS). Viele DGS setzen komplett auf Java (GEONExT, GeoGebra, C.a.R., Cinderella.2), einige nutzen Java zur Darstellung interaktiver Inhalte im Browser (Geometer's Sketch-Pad).

[1] http://ck12.org, März 2014
[2] http://www.mizs.gov.si/, März 2014
[3] http://www.e-um.si, http://www.itextbooks.eu/, März 2014

Aufbau

In dieser Arbeit werden Konzepte und Softwareprojekte beschrieben, wie Software realisiert werden kann, die den jüngsten Veränderungen im Bereich ICT im Bezug auf den Mathematikunterricht Rechnung trägt.

Alle in dieser Arbeit beschriebenen Projekte (JessieCode, practice, sketchometry und SketchBin) basieren auf JSXGraph oder nutzen dieses seit Herbst 2007 am Lehrstuhl für Mathematik und ihre Didaktik entwickelte dynamische Geometriesystem. Abbildung 1.1 verdeutlicht die Abhängigkeiten unter diesen Projekten. JSXGraph unterscheidet sich in zweierlei Hinsicht von bekannten Geometriesystemen wie GeoGebra oder C.a.R. Zum einen ist es komplett in JavaScript entwickelt, zum anderen ist es ein reines *Application Programming Interface* (API). JSXGraph ist damit ein hervorragendes Werkzeug für E-Learning im Browser. Als API macht es das Erstellen von beinahe beliebig komplexen Konstruktionen möglich, erfordert aber Erfahrung im Umgang mit der Programmierung in JavaScript. **Kapitel 2** beschreibt die grundlegende Architektur und die Herausforderungen bei der Implementierung und Optimierung von JSXGraph.

Die weiteren Projekte sollen das Erstellen von Konstruktionen mit JSXGraph weiteren Nutzerkreisen möglich machen: Schülern und Lehrern (sketchometry), Nutzern von Online-Plattformen zum gemeinsamen Wissensaustausch (JessieCode) und Autoren von dynamischen Arbeitsblättern (SketchBin) (siehe [23], Teil II für eine ausführliche Einführung in das Konzept dynamischer Arbeitsblätter).

Kapitel 3 geht auf Arbeiten zur geometrischen Ortsberechnung ein, die auf den Ergebnissen meiner Diplomarbeit [12] basieren und in [13] bereits teilweise veröffentlicht wurden. Nach einer kurzen Einführung in die Ortsberechnung mit Gröbnerbasen stelle ich zwei Möglichkeiten vor, mit deren Hilfe diese Berechnungen beschleunigt werden können. Mit einer Translation der Konstruktion können die zur Ermittlung der Ortskurven benötigten Gleichungssysteme verkleinert werden. Unter bestimmten Voraussetzungen kann ein Ort auch über die Verkettung von mindestens zwei kleineren Orten berechnet werden.

Kapitel 4 konzentriert sich auf den Austausch interaktiver mathematischer Inhalte in Online-Austauschplattformen mit anderen Nutzern. Das Problem ist, dass der Austausch von beliebigem JavaScript unter den Nutzern einer

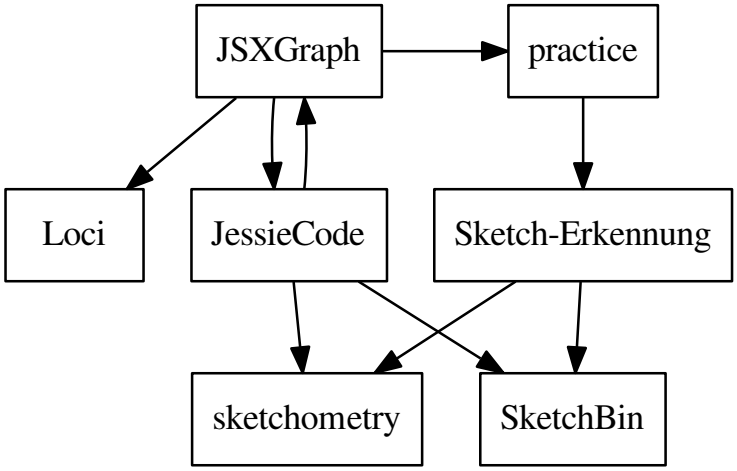


Abbildung 1.1: *JSXGraph und die in dieser Arbeit beschriebenen Projekte, die darauf basieren.*

Plattform ein enormes Sicherheitsrisiko birgt. Es wird diskutiert, wie dieses Risiko minimiert oder gar eliminiert werden kann und wie die Skriptsprache JessieCode dazu beiträgt. Weiterhin wird die JessieCode-Architektur vorgestellt und wie das JSXGraph-API darin integriert ist. Abgeschlossen wird dieses Kapitel mit einer Referenz der Sprache: Welche Datentypen zur Verfügung stehen, welche Kontrollstrukturen bereitgestellt werden und welche Funktionen des JSXGraph-API Entwickler direkt nutzen können.

Bei E-Learning-Angeboten gibt es neben der reinen Präsentation von (interaktiven) Inhalten auch das Bedürfnis, Wissen und Können automatisiert zu prüfen. Sehr weit verbreitet sind hier Fragen mit mehreren Antwortmöglichkeiten ("multiple choice") sowie die Eingabe von Textantworten, Zahlen oder Gleichungen. Die Eingabe von graphischen Lösungen geometrischer Probleme in Form von Konstruktionen ist genauso wie deren Auswertung möglich. Allerdings ist letztere aufwändig in der Umsetzung.

Das in **Kapitel 5** vorgestellte practice reduziert diesen Aufwand durch die Übernahme der kombinatorischen Komponente der Auswertung sowie dem Bereitstellen robuster Algorithmen zur Auswertung der geometrischen Eigen-

schaften einer Konstruktion. Neben dem E-Learning findet die automatisierte Auswertung einer Konstruktion auch in der Sketch-Erkennung in sketchometry Anwendung.

Kapitel 6 beschreibt diese Sketch-Erkennung. Zunächst jedoch werden die durch Tablets und Smartphones weit verbreiteten Bedienkonzepte beschrieben und wie diese in dynamischer Geometrie-Software eingesetzt werden können. Hierbei verlieren wir allerdings nicht den Blick auf die traditionellen Plattformen wie PCs und Notebooks und berücksichtigen diese ebenfalls bei der Entwicklung. Es folgt eine Beschreibung der in der Sketch-Erkennung verwendeten Algorithmen und wie die im vorherigen Kapitel vorgestellte JSXGraph-Erweiterung practice mit wenig Erweiterungen für die Sketch-Erkennung genutzt werden kann. Abgerundet wird das Kapitel mit einer Analyse der Stärken und Schwächen der Sketch-Erkennung über eine Evaluation, an der 116 Schüler in fünf Schulklassen insgesamt 2808 Aufgaben bearbeitet haben.

sketchometry zielt auf einen Einsatz im Unterricht zur schnellen Erstellung von Skizzen ab. Es ist daher eher als eine Art Rapid Prototyping System zur Erstellung einfacher Konstruktionen gedacht. Es fehlt die Möglichkeit, komplexe Konstruktionen mit nicht-geometrischen Abhängigkeiten und eigenen Bedienelementen zu erstellen. JSXGraph und JessieCode bieten diese Möglichkeit, sind für den Einsteiger aber weniger geeignet. Die Erstellung von Grundkonstruktionen ist im Vergleich zu klassischen Systemen und sketchometry langwierig. In **Kapitel 7** wird der Prototyp eines Autorentools beschrieben, das JSXGraph, JessieCode und sketchometry vereint. Die einfach gehaltene Bedienoberfläche von sketchometry kombiniert mit den Möglichkeiten, die eine Skriptsprache und ein API bieten. Beide Eingabeformen sind dabei gleichberechtigt. Änderungen am Programmcode ändern die graphische Repräsentation sofort und umgekehrt wird der Programmcode durch die in sketchometry üblichen Bearbeitungen der graphischen Repräsentation verändert, zum Beispiel durch das Ziehen eines freien Punktes.

Kapitel 2

JSXGraph

Die JSXGraph-Webseite[1] schreibt über das Projekt:

> JSXGraph is a cross-browser library for interactive geometry, function plotting, charting, and data visualization in a web browser. It is implemented completely in JavaScript, does not rely on any other library, and uses SVG, VML, or canvas. JSXGraph is easy to embed and has a small footprint: less than 100 KByte if embedded in a web page. No plug-ins are required! Special care has been taken to optimize the performance.
>
> JSXGraph supports multi-touch devices running iOS, Android, firefoxOS, Windows 8 (at least).

Für fast alle in dieser Arbeit vorgestellten Projekte bildet JSXGraph das Fundament. Es dient zur Ausgabe von Konstruktionen für sketchometry und SketchBin. Für JessieCode dient es als Laufzeitumgebung und practice macht ebenfalls Gebrauch von den bereitgestellten mathematischen Funktionen und Algorithmen.

Die Entwicklung begann 2007 im Rahmen eines Seminars "**Graph**ik mit **J**ava**S**cript" – daher auch der Name JSXGraph. Das **X** entstammt dem eigentlichen Vorhaben, GEONE**x**T Dateien im Browser darzustellen, ohne auf Browserplugins wie Java oder Flash zurückgreifen zu müssen.

[1] http://jsxgraph.org, März 2014

2.1 Architektur

JSXGraph teilt sich in verschiedene Module auf, die eng miteinander zusammenarbeiten. Abbildung 2.1 bietet eine Übersicht über die wichtigsten Module und wie sie voneinander abhängen.

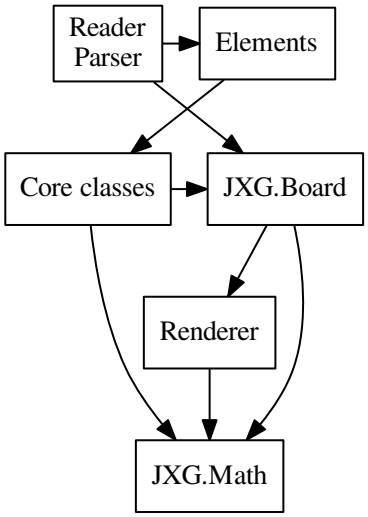


Abbildung 2.1: *Übersicht über die Hauptmodule und deren Abhängigkeiten untereinander in JSXGraph.*

Das Modul `JXG.Math` stellt den restlichen Modulen Datenstrukturen und Algorithmen aus verschiedenen Bereichen der Mathematik zur Verfügung, wie beispielsweise Lineare Algebra, Geometrie, Statistik und Analysis. Diese sind in der JSXGraph-Referenz[2] von `JXG.Math`, `JXG.Math.Geometry`, `JXG.Math.Numerics` und `JXG.Math.Statistics` ausführlich dokumentiert.

Bei der graphischen Ausgabe ist JSXGraph auf die jeweilige Umgebung angewiesen. Moderne Browser bieten zur graphischen Darstellung das Canvas-API [15] und eine Implementierung der Scalable Vector Graphics (SVG) Spezifikation [16] an. Canvas ist ein Teil der HTML5 Spezifikation und stellt Funktionen zur Manipulation von Bitmap-Graphiken zur Verfügung. SVG ist eine auf

[2] http://jsxgraph.uni-bayreuth.de/docs, März 2014

XML basierende und vom W3C[3] empfohlene Spezifikation zur Beschreibung von Vektorgraphiken. SVG wird von den Browsern Firefox, Chrome, (Mobile) Safari und Opera unterstützt. Der Android-Browser unterstützt SVG seit Android 3.0. Im Microsoft Internet Explorer wird SVG seit Version 9 nativ unterstützt, das heißt es sind keine Plugins von Drittanbietern nötig, um SVG im Internet Explorer darzustellen. Alle Browser, die SVG unterstützen, stellen auch das Canvas API bereit. Im Android-Browser steht Canvas bereits seit Android 2.1 zur Verfügung. Zur Darstellung im Microsoft Internet Explorer bis einschließlich Version 8 greift JSXGraph auf die Vector Markup Language (VML) zurück. VML ist einer der Vorgänger von SVG und ebenfalls eine auf XML basierende Auszeichnungssprache (*markup language*) zur Beschreibung von Vektorgraphiken. Das Rendering-Modul dient als Abstraktionsebene zwischen den unterschiedlichen Rendering APIs SVG, VML und Canvas.

In den Core-Klassen `Point, Line, Curve, Image, Polygon, Turtle` und `Text` sind die grundlegenden Datenstrukturen für alle Elemente definiert, die in `JXG.GeometryElement` eine gemeinsame Basis finden. Jedes Element ist zusammengesetzt aus mindestens einem anderen Element oder einer Basisklasse. `point, midpoint, parallel point` und `glider` sind alles Elemente, die auf der `Point`-Klasse basieren. Das `slider`-Element besteht aus zwei `point`-Elementen, zwei `line`-Elementen, einem `text`-Element sowie einem `glider`-Element, das letztendlich auch den gesamten Slider repräsentiert. Siehe hierzu Abbildung 2.2. Mit Version 0.99 wird JSXGraph insgesamt über 70 Elemente unterstützen. An dieser Stelle sei auf die JSXGraph-Referenz[4] verwiesen. Zusätzlich ist im Projekt-Wiki[5] zu fast jedem Element ein Beispiel zu finden.

Kernstück einer Konstruktion ist eine Instanz der Klasse `JXG.Board`, in der alle Fäden zusammenlaufen: Es verwaltet die Elemente und nimmt alle Eingaben des Benutzers entgegen. Im weiteren Verlauf der Arbeit wird der Begriff des *Boards* wiederholt auftreten. Damit ist eine Instanz dieser Klasse gemeint. Ein Board empfängt alle Eingaben des Benutzers über Event-Handler, interpretiert diese je nach Konfiguration des Boards und aktualisiert entsprechend betroffene Elemente, andere Boards und die Darstellung über seinen Renderer. Die Verwaltung der Elemente beginnt bereits bei deren Initialisierung: Dies geschieht grundsätzlich nur über die Board-Methode `create()`

[3] http://www.w3.org/, März 2014
[4] http://jsxgraph.uni-bayreuth.de/docs, März 2014
[5] http://jsxgraph.uni-bayreuth.de/wiki, März 2014

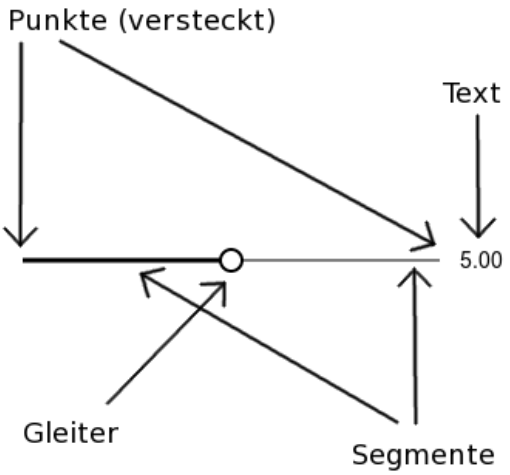


Abbildung 2.2: *Ein Slider inklusive aller enthaltenen Basisklassen.*

(siehe Abschnitt 2.2).

Reader machen sich alle bereits besprochenen Teilmodule zu nutze, um Konstruktionen, die mit anderen Geometriesystemen wie Tracenpoche, GEONExT und GeoGebra erstellt wurden, in JSXGraph zu importieren und darzustellen.

2.2 create()-Funktion

Elemente werden mit der Methode `create()` der Board-Klasse initialisiert. Das Ziel hierbei war ein einheitliches Interface für alle Elemente zu bieten. Zur Erzeugung eines Elements benötigt `create()` immer die gleichen Parameter: den Namen des Elements und ein Array mit allen Elternelementen (*parents*). Parents sind Konstanten, Funktionen oder andere, bereits existierende Elemente, welche die Abhängigkeiten des neuen Elements definieren. Optional kann als dritten Parameter ein JavaScript-Objekt angegeben werden, das die visuellen Eigenschaften (*attributes*) festlegt, die von den Standardwerten abweichen sollen.

Listing 2.1: ***Aufruf von board.create() zur Erzeugung eines grünen Punktes an der Position (4, 5)***

```
var board = JXG.JSXGraph.initBoard('box');
board.create('point', [4, 5], {
    strokeColor: 'green',
    fillColor: 'green',
    face: 'o',
    size: 7
});
```

`create()` schlägt bei einem Aufruf den Elementnamen in einem in `JXG.elements` abgelegten Verzeichnis nach. `JXG.elements` ist ein JavaScript-Objekt, in dem jedem Elementnamen eine *Creator-Funktion* zugewiesen ist. Gefüllt wird dieses Verzeichnis durch Aufrufe von `JXG.registerElement(name, creator)`, das ein neues Element mit Namen `name` die Funktion `creator` zuweist. Creator-Funktionen sind immer gleich aufgebaut und akzeptieren drei Parameter: Eine Referenz auf das Board, zu dem sie gehören werden, ein parent Array und ein Attributes-Objekt.

Typischer Ablauf einer Creator-Funktion ist,

1. zu überprüfen, ob das Element mit den übergebenen parents erzeugt werden kann. Ist dies nicht der Fall, wird eine Exception geworfen.
2. die Attribute aller zu erzeugenden Elemente aus dem übergebenen attributes Objekt zu extrahieren und mit den in `JXG.Options` hinterlegten Standardwerten zusammenzuführen. JSXGraph bietet hierzu die Hilfsfunktion `JXG.copyAttributes()` an, die genau das erledigt.
3. alle erforderlichen Elemente zu initialisieren.
4. gegebenenfalls die erzeugten Elemente nachzubereiten, um Methoden ergänzen oder Eigenschaften korrigieren zu können.

Dank dieses Systems ist es sehr leicht, JSXGraph um neue Elemente zu erweitern, die sich nahtlos in das bestehende System integrieren.

Listing 2.2: ***Definition eines Elements 'triangle'.***

```
// Standard-Eigenschaften definieren
// Diese basieren auf den Eigenschaften des
// verwendeten Elements 'polygon'
JXG.Options.triangle = {
    hasInnerPoints: true,
    fillColor: 'none',
    withLines: true,
    borders: {
        withLabel: false
    },
    label: [0, 0]
};

createTriangle = function (board, parents, attributes) {
    if (parents && parents.length === 3 &&
        JXG.isPoint(parents[0]) &&
        JXG.isPoint(parents[1]) &&
        JXG.isPoint(parents[2])) {

        var attr = JXG.copyAttributes(attributes,
            board.options,
            'triangle'
        );
        return board.create('polygon', parents, attr);
    } else {
        throw new Error('Can\'t create triangle.');
    }
};

JXG.registerElement('triangle', createTriangle);
```

Auch *Themes* lassen sich damit sehr einfach umsetzen. Ein Theme definiert die visuellen Eigenschaften eines Elements, die nicht im `create()`-Aufruf übergebenen Attributes-Objekt definiert werden. Um ein eigenes Theme zu definieren, genügt es, die geänderten Eigenschaften analog zur Struktur des `JXG.Options`-Objekts in einem eigenen Objekt abzulegen, nach JSXGraph in das Dokument zu laden und mittels `JXG.merge()` mit `JXG.Options` zu verschmelzen.

2.3 Optimierung

Für ein interaktives System wie JSXGraph ist es enorm wichtig, dass Berechnungen effizient durchgeführt werden. Im Zugmodus oder bei Animationen muss eine Konstruktion fortlaufend aktualisiert werden können. Neben schnellen Algorithmen zur Berechnung von Schnitten, Projektionen, Integralen etc. sind vor allem spezifisch auf die Laufzeitumgebung ausgelegte Optimierungen wichtig.

Im Vergleich zur Berechnung der Koordinaten eines Schnittpunkts zweier Geraden ist das Aktualisieren des DOM sehr aufwändig. Daher werden alle für die visuelle Darstellung relevanten Eigenschaften eines Elements zwischengespeichert und während einer Neuberechnung nur dann im DOM aktualisiert, wenn sie sich ändern.

Das Trennen des Updates in einen Berechnungs-Schritt (`Board.update()`) und einen Renderschritt (`Board.updateRenderer()`) erlaubt das vorübergehende Aushängen des SVG-Baums aus dem DOM-Baum für die Dauer der Aktualisierung der Zeichnung. Dies kann den Aktualisierungsvorgang erheblich beschleunigen, da die Anzahl der Reflow-Berechnungen stark reduziert wird.

Um das Darstellen von Kurven zu verbessern, kann die Auflösung von Kurven reduziert werden, beispielsweise im Zugmodus. Mit Hilfe des Ramer-Douglas-Peucker (RDP) [9, 26] Algorithmus kann die Zahl der an das Rendering übergebenen Punkte abhängig von der dargestellten Kurve weiter verringert werden, ohne die Darstellungsqualität negativ zu beeinflussen.

Derzeit werden in JSXGraph bei einer Aktualisierung des Boards alle Elemente aktualisiert, deren Eigenschaft `needsRegularUpdate` den Wert `true` hat. Diese ist bei allen außer dem axis-Element standardmäßig auf `true` gesetzt. Dies verursacht bei einer Aktualisierung Berechnungen, die nicht zwingend notwendig sind. Der Grund, warum hier bislang keine Optimierungen implementiert sind, ist, dass die Abhängigkeiten nicht zuverlässig aufgelöst werden können. JSXGraph verfolgt Abhängigkeiten unter den Elementen, falls dies möglich ist. Bei der Konstruktion einer parallelen Gerade über das Element `parallel` wird die erzeugte Gerade als Kind bei den Elternelementen eingtragen und umgekehrt. Bei der Erzeugung eines berechneten Punktes wie in Listing 2.3 ist die Abhängigkeit von B von A nicht automatisch ableitbar. In den meisten Fällen ist dies vernachlässigbar. Ist diese Optimie-

rung gewünscht oder sogar notwendig, muss der Autor einer Konstruktion bislang mit Hilfe der `needsRegularUpdate`-Eigenschaft den Update-Prozess von Hand optimieren. Da dies aufwändig und fehleranfällig ist, ist eine Erweiterung des Update-Prozesses geplant, die anhand eines Abhängigkeitsbaumes den Aktualisierungsvorgang optimiert. Die manuelle Optimierung über `needsRegularUpdate` wird damit ersetzt durch die Angabe der Abhängigkeiten unter den Elementen durch den Nutzer.

Listing 2.3: ***Konstruktion eines berechneten Punktes, dessen Abhängigkeiten nicht automatisch bestimmbar sind.***

```
// Initialisiere das Board
var board = JXG.JSXGraph.initBoard('...');

// Erzeuge einen freien Punkt
var A = board.create('point', [0, 0]);

// Erzeuge einen berechneten Punkt, der von A aus
// um (3, 3) verschoben ist.
var B = board.create('point', [function () {
    return A.X() + 3;
}, function () {
    return A.Y() + 3;
}]);
```

Kapitel 3

Geometrische Orte

Im Rahmen meiner Diplomarbeit implementierte ich die computergestützte Berechnung geometrischer Orte mittels Gröbnerbasen in JSXGraph [12]. Dieses Kapitel beschäftigt sich zunächst mit geometrischen Orten, Gröbnerbasen und wie beide zusammenhängen. Da die Berechnung von Gröbnerbasen sehr aufwändig ist, werden im Anschluss Möglichkeiten diskutiert, die Berechnung geometrischer Orte in JSXGraph zu beschleunigen [13] und somit die Performance der Software zu erhöhen.

3.1 Grundlagen

Unter einem geometrischen Ort versteht man eine Menge von Punkten, die eine bestimmte Eigenschaft erfüllen. Im Allgemeinen umfasst dies auch Flächen. Wir beschränken uns im weiteren Verlauf auf die Untermenge der Ortskurven im $\mathbb{R}^2$. In Dynamischen Geometriesystemen werden Ortskurven meist durch zwei Punkte definiert, einem Gleiter und einem davon abhängigen Punkt. Ein Gleiter ist ein an ein eindimensionales Trägerobjekt, wie eine Kreislinie, eine Gerade oder eine algebraische Kurve, gebundener Punkt und besitzt daher nur einen Freiheitsgrad. Der abhängige Punkt ist (in-)direkt vom Gleiter abhängig.

Um eine Ortskurve zu berechnen gibt es verschiedene Strategien. Die einfachste Methode berechnet für unterschiedliche Positionen des Gleiters auf

dem Trägerobjekt die entsprechenden Koordinaten des abhängigen Punktes und interpoliert das Ergebnis. Dieser Ansatz hat jedoch einige Nachteile: Man erhält nicht die definierende Gleichung der Ortskurve und es gibt Fälle, in denen Teile der Ortskurve fehlen. Diese Methode ist in JSXGraph als Element *tracecurve* implementiert. Der Unterschied zum Element *locus* ist in den Abbildungen 3.1 und 3.2. zu sehen. Beide zeigen den Versuch, eine Wattkurve zu berechnen. A und B sind freie Mittelpunkte zweier Kreise c_1 und c_2. G ist ein Gleiter auf der Kreislinie c_1. c_3 ist ein Kreis um G, der sich in D mit c_2 schneidet. E ist der Mittelpunkt von G und D. Die Abbildung des Elements locus zeigt den vollständigen Ort (Abbildung 3.2). Im Vergleich dazu fehlt in der Darstellung des Elements tracecurve die Hälfte der Kurve (Abbildung 3.1). Der Grund dafür ist die Identifizierung der Schnittpunkte von Kreisen: In JSXGraph werden Schnittpunkte als Lösung einer quadratischen Gleichung aufgefasst und über ihr Vorzeichen unterschieden. Dadurch kann ein bestimmter Schnittpunkt nur auf einer bestimmten Hälfte des Kreises gefunden und nur die Hälfte der Ortskurve numerisch ermittelt werden

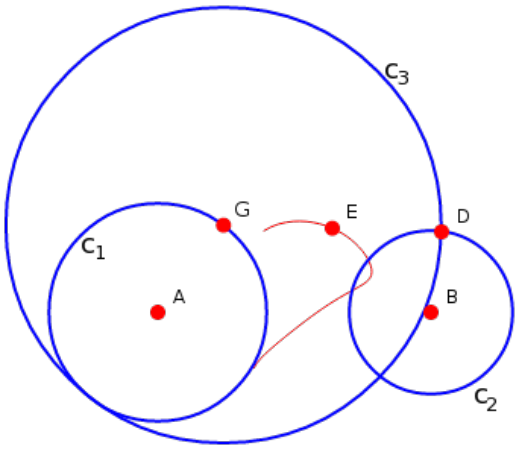


Abbildung 3.1: *Die unvollständige geometrische Ortskurve von E, dargestellt über das Element* tracecurve.

Zur Vermeidung fehlender Zweige im Ergebnis nutzt beispielsweise Cinderella.2 ein Trace Modul [20]. Zur Bestimmung der Kurvengleichung gibt es Versuche über einen numerischen Ansatz [21].

Im Bereich der algebraischen Möglichkeiten gibt es die Methode von Wu [34] und die Berechnung über Gröbnerbasen [6, 11, 4, 27, 5]. Erstere ist bislang in noch keinem DGS implementiert, Gröbnerbasen finden sowohl in GeoGebra als auch JSXGraph Anwendung – in GeoGebra allerdings bislang nur über eine Berechnung auf entfernten, zentralen Servern und nicht lokal.

Die Berechnung in JSXGraph ist technisch gesehen zwar ebenfalls über einen Webserver realisiert, dieser kann jedoch durchaus auch lokal eingerichtet und genutzt werden.

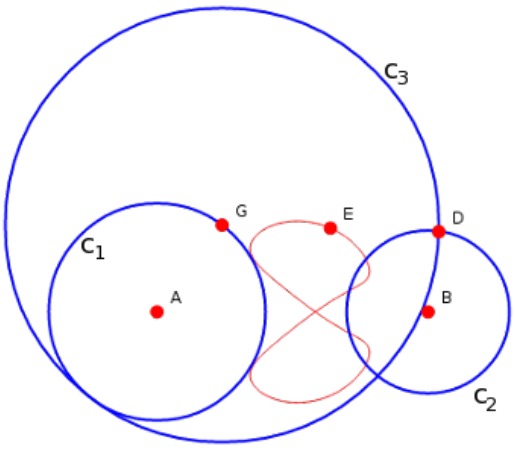


Abbildung 3.2: *Die geometrische Ortskurve von E dargestellt über das Element* locus.

3.2 Gröbnerbasen

Im weiteren Verlauf dieses Abschnitts bezeichnet $\mathbb{K}$ einen *Körper* und $R = \mathbb{K}[X_1, \ldots, X_n]$ einen *Polynomring* über $\mathbb{K}$ in n Veränderlichen. $\prec$ bezeichnet eine *Monomordnung*. Für eine genaue Definition dieser Begriffe, siehe [8]. Ist $f \in R$ ein Polynom, dann bezeichnet $\mathrm{lt}(f)$ den Leitterm und $\mathrm{lm}(f)$ das Leitmonom von f. Der Leitterm ist das Produkt des Leitkoeffizienten mit dem Leitmonom. Ist $I \subseteq R$ ein Ideal des Polynomrings R, dann ist $\mathrm{lt}(I) := \{\mathrm{lt}(f) \mid f \in I\}$.

Ist $R = \mathbb{K}[X_1, \ldots, X_n]$ ein Polynomring in n Veränderlichen über dem Körper $\mathbb{K}$ und $\prec$ eine Monomordnung auf R, dann bezeichnen wir eine endliche Teilmenge $G := \{g_1, \ldots, g_s\}$ des Ideals $I \subseteq R$ als *Gröbnerbasis*, wenn

$$\mathrm{Ideal}\left(\mathrm{lt}(g_1), \ldots, \mathrm{lt}(g_s)\right) = \mathrm{Ideal}\left(\mathrm{lt}(I)\right).$$

Jedes Ideal $I \subseteq R$ eines Polynomrings besitzt eine Gröbnerbasis

$$G = \{g_1, \ldots, g_s\},$$

die gleichzeitig auch eine Basis von I ist, das heißt

$$I = \text{Ideal}(g_1, \ldots, g_s).$$

Für einen Beweis dieser beiden Behauptungen, siehe Satz 2.5 in [12].

Im Buchberger-Algorithmus werden Gröbnerbasen mittels S-Polynomen berechnet. Sind $f, g \in \mathbb{K}[X_1, \ldots, X_n] \backslash \{0\}$, dann bezeichnen wir

$$S(f,g) := \frac{\text{kgV}\,(\text{lm}(f), \text{lm}(g))}{\text{lt}(f)} \cdot f - \frac{\text{kgV}\,(\text{lm}(f), \text{lm}(g))}{\text{lt}(g)} \cdot g$$

als das S-Polynom von f und g. $\text{kgV}(a,b)$ ist das kleinste gemeinsame Vielfache von a und b.

Listing 3.1: ***Buchberger-Algorithmus***

```
Eingabe: Ein Ideal F = (f_1, ..., f_t).
Ausgabe: Die Gröbnerbasis G = (g_1, ..., g_s) mit F ⊂ G.

Setze G := F
Wiederhole
  Setze G' := G
  Für jedes Paar (p,q) mit p ≠ q ∈ G'
    Setze S := S(p,q) rem G'
    Falls S ≠ 0, dann
      Setze G := G ∪ {S}
bis G = G'
```

Der Zusammenhang von Gröbnerbasen mit der Berechnung geometrischer Orte wird deutlich, wenn wir einen Blick auf die Eliminationseigenschaft von Gröbnerbasen werfen. Ist $I \subseteq R$ ein Ideal des Polynomrings R und $0 \leq l < n$, dann heißt

$$I_l := I \cap \mathbb{K}[X_{l+1}, \ldots, X_n]$$

das l-te Eliminationsideal von I. Wird bei der Berechnung der Gröbnerbasis $G = \{g_1, \ldots, g_s\}$ von I als Monomordnung eine Eliminationsordnung verwendet, kann dessen l-tes Eliminationsideal sehr einfach berechnet werden:

$$I_l = G \cap \mathbb{K}[X_{l+1}, \ldots, X_n].$$

Geometrisch entspricht dies einer Projektion π_l der durch das Ideal I bestimmten Varietät V auf einen Unterraum. Allerdings gilt

$$\pi_l(V) \subseteq V(I_l).$$

Das heißt, als geometrischer Ort kann auch eine Obermenge des eigentlichen Ortes berechnet werden.

Beispiel Wattkurve

Im Beispiel der Wattkurve im vorherigen Abschnitt kann $G(u_1, u_2)$ durch die Kreisgleichung

$$u_1^2 + u_2^2 - 16 = 0$$

beschrieben werden, wenn A im Ursprung liegt und der Radius von c_1 gleich 4 ist. Liegt B in $(8, 0)$ und hat c_2 den Radius 3, erhalten wir für den Schnittpunkt $D(u_3, u_4)$ der Kreise c_2 und c_3 die beiden Kreisgleichungen

$$(u_3 - 8)^2 + (u_4 - 0)^2 - 9 = 0, (u_3 - u_1)^2 + (u_4 - u_2)^2 - 36 = 0,$$

vorausgesetzt der Radius des Kreises c_3 beträgt 6. $E(x, y)$ ist der Mittelpunkt von G und D. Dadurch liegt er auf einer Geraden, die durch G und D definiert ist:

$$(u_2 - u_4)x + (u_3 - u_1)y + u_1u_4 - u_2u_3 = 0.$$

Außerdem besitzt E zu D und G jeweils den gleichen Abstand:

$$(x - u_1)^2 + (y - u_2)^2 = (x - u_3)^2 + (y - u_4)^2.$$

Zusammen ergeben diese fünf Gleichungen ein polynomiales Gleichungssystem. Betrachten wir nur die Polynome, erhalten wir ein Ideal

$$I \subseteq \mathbb{Q}[u_1, \ldots, u_4, x, y].$$

Durch die Berechnung des Eliminationsideals erhalten wir die Ortsgleichung

$$\begin{aligned} 4y^6 + 12x^2y^4 - 96xy^4 + 292y^4 + 12x^4y^2 - 192x^3y^2 + 1096x^2y^2 \\ -2512xy^2 + 49y^2 + 4x^6 - 96x^5 + 804x^4 - 2512x^3 + 561x^2 \\ +6384x + 3136 = 0. \end{aligned} \tag{3.1}$$

3.3 Transformation der Konstruktion

Eine einfache Möglichkeit, die Berechnung zu beschleunigen, ist eine Transformation der Konstruktion. Es wird ein freier Punkt A gewählt und in den Ursprung verschoben. Anschließend wird ein zweiter freier Punkt B gewählt und die gesamte Konstruktion so um Punkt A rotiert und gestreckt, dass B in $(1, 0)$ liegt. Dadurch werden bis zu drei Koordinaten auf 0 gesetzt und manche Monome aus den Polynomgleichungen eliminiert – das Gleichungssystem wird kleiner. Nach der Berechnung der Kurve werden die invertierten Transformationen auf die Datenpunkte der Kurve angewandt.

3.3.1 Beispiel Limaçon

Listing 3.2: ***Pascalsches Limaçon, Grundkonstruktion***

```
$board.setView([-2, 20, 20, -2], true);

A = point(8, 3);
B = point(8, 8);
c1 = circle(B, 4);

D = glider(0, 0, c1);
g = line(A, D);

c2 = circle(D, 3);

T = intersection(c2, g, 0);
```

Als Gleichungssystem der untransformierten Konstruktion erhalten wir

$$\begin{aligned} (u_1 - 8)^2 + (u_2 - 8)^2 - 16 &= 0, \\ (x - u_1)^2 + (y - u_2)^2 - 9 &= 0, \\ 3x - 3u_1 + yu_1 - 8y + 8u_2 - xu_2 &= 0, \end{aligned} \tag{3.2}$$

wobei (u_1, u_2) die Koordinaten des Punktes D und (x, y) die Koordinaten des Punktes T sind. Verschieben wir A nach $(0, 0)$ so liegt B jetzt in $(0, 5)$ und wir erhalten als Gleichungssystem:

$$\begin{aligned} u_1^2 + (u_2 - 5)^2 - 16 &= 0, \\ (x - u_1)^2 + (y - u_2)^2 - 9 &= 0, \\ yu_1 - xu_2 &= 0. \end{aligned} \tag{3.3}$$

In diesem konkreten Beispiel kann die reine Berechnungszeit[1] der Gröbnerbasis um bis zu 29% verkürzt werden: Zur Ermittlung der Basis waren circa 45 Sekunden für 10000 Testläufe des nicht optimierten Gleichungssystems

[1] Intel Core2Quad CPU Q9550@2.83GHz mit CoCoA 5

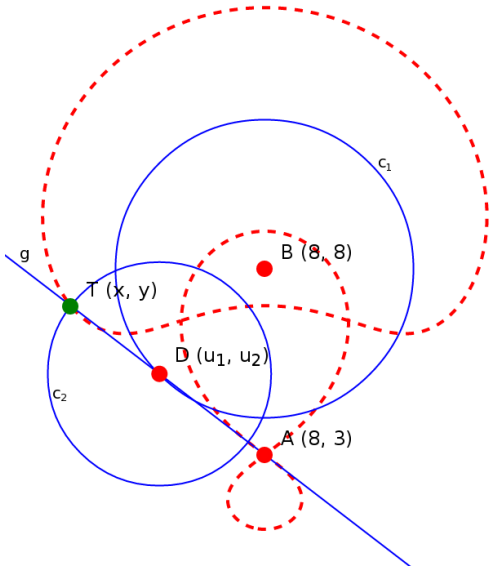


Abbildung 3.3: *Das Pascalsche Limaçon.*

nötig und etwa 32 Sekunden dauerte die Berechnung von 10000 Durchläufen der optimierten Variante des Gleichungssystems.

3.4 Verketten der Berechnung

In diesem Abschnitt wird ein Verfahren vorgestellt, das einen geometrischen Ort nicht direkt berechnet, sondern diesen sukzessive über die Berechnung kleinerer Orte ermittelt. Manchmal ergibt sich daraus ein Geschwindigkeitsvorteil. Allerdings ist zu beachten, dass dieses Verfahren auch eine Obermenge der Lösung der direkten Berechnung liefern kann.

3.4.1 Beispiel Limaçon

Erweitern wir das Pascalsche Limaçon um einen Kreis c_3 um T, den Schnittpunkt C des Kreises um B und c_3 und den Mittelpunkt E von C und T (siehe Abbildung 3.5) erhalten wir erneut einen neuen geometrischen Ort.

Hier ergeben sich zwei Möglichkeiten diesen zu berechnen.

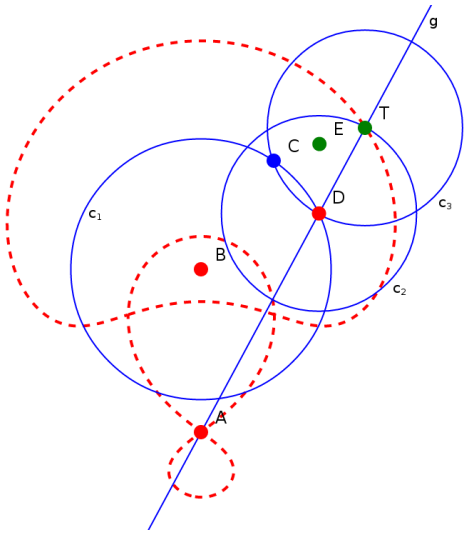


Abbildung 3.4: *Eine direkte Erweiterung der Grundkonstruktion des Pascalschen Limaçons.*

Direkte Berechnung

Zur direkten Berechnung erweitern wir die Konstruktion, wie beschrieben, direkt an $T(u_1, u_2)$ und ermitteln die Gröbnerbasis des Ideals erzeugt aus den Polynomen der Gleichungen 3.2 ergänzt um insgesamt vier Polynome für $E(x, y)$ und $C(u_5, y_6)$. D wird hier von den Koordinaten (u_3, u_4) repräsentiert. Wir erhalten Gleichung 3.4.

$$\begin{aligned}
(u_1 - u_3)^2 + (u_2 - u_4)^2 - 9 &= 0,\\
3u_1 - 3u_3 + u_2u_3 - 8u_2 + 8u_4 - u_1u_4 &= 0,\\
(u_3 - 8)^2 + (u_4 - 8)^2 - 16 &= 0,\\
(u_5 - u_1)^2 + (u_6 - u_2)^2 - 9 &= 0,\\
(u_5 - 8)^2 + (u_6 - 8)^2 - 16 &= 0,\\
u_2x - u_2u_5 + yu_5 - u_1y + u_1u_6 - xu_6 &= 0,\\
u_1^2 - 2u_1x + u_2^2 - 2u_2y - u_5^2 + 2u_5x - u_6^2 + 2u_6y &= 0.
\end{aligned} \tag{3.4}$$

Verkettete Berechnung

Alternativ kann zunächst das Limaçon berechnet werden. Anstatt die Erweiterung der Konstruktion direkt an T zu beginnen, wird ein Gleiter $T'(u_1, u_2)$ auf der Ortskurve definiert. T' dient als Mittelpunkt für c_3 und $C(u_3, u_4)$ bleibt ein Schnittpunkt der Kreise c_3 und c_1. $E(x, y)$ ist hier der Mittelpunkt von C und T'. Bestimmen wir das Gleichungssystem der Konstruktion ausgehend von E, so erhalten wir die fünf Gleichungen aus 3.5.

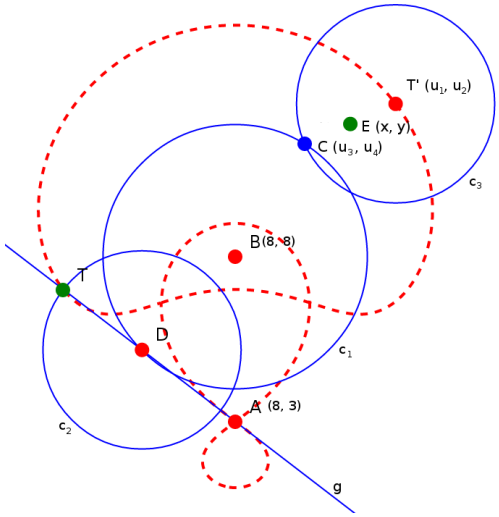


Abbildung 3.5: *Eine Erweiterung des Pascalschen Limaçons über einen Gleiter auf dem Limaçon selbst.*

$$
\begin{aligned}
u_2^6 - 38u_2^5 + 3u_1^2u_2^4 - 48u_1u_2^4 + 727u_2^4 - 76u_1^2u_2^3 \\
+1216u_1u_2^3 - 8404u_2^3 + 3u_1^4u_2^2 - 96u_1^3u_2^2 + 1774u_1^2u_2^2 \\
-16096u_1u_2^2 + 63535u_2^2 - 38u_1^4u_2 + 1216u_1^3u_2 - 16596u_1^2u_2 \\
+109888u_1u_2 - 300806u_2 + u_1^6 - 48u_1^5 + 1047u_1^4 - 13024u_1^3 \\
+97395u_1^2 - 415536u_1 + 790009 = 0 \qquad (3.5) \\
(u_3 - u_1)^2 + (u_4 - u_2)^2 - 9 = 0 \\
(u_3 - 8)^2 + (u_4 - 8)^2 - 16 = 0 \\
u_2x - u_2u_3 + yu_3 - u_1y + u_1u_4 - xu_4 = 0 \\
u_1^2 - 2u_1x + u_2^2 - 2u_2y - u_3^2 + 2u_3x - u_4^2 + 2u_4y = 0
\end{aligned}
$$

Vergleich

Über den zweiten Ansatz kann man bei diesem konkreten Beispiel bis zu 75% an Rechenzeit einsparen (siehe [13]). Beim Grundverfahren gibt es das Problem, dass eine Obermenge der tatsächlich gesuchten Lösungsmenge gefunden wird, wenn degenerierte Fälle nicht behandelt werden. Ein ähnliches Problem stellt sich hier. Es kann passieren, dass wir eine Obermenge der tatsächlich gesuchten Lösung erhalten:

Gegeben sei die Konstruktion in Abbildung 3.6. g ist die Gerade durch die beiden freien Punkte A und B. c_1 ist ein Kreis um A. c_2 ist ein Kreis um den Gleiter $G(u_3, u_4)$, der auf c_1 liegt. Der Kreis c_2 und die Gerade g schneiden sich in $I_1(u_1, u_2)$. $M(x, y)$ ist der Mittelpunkt der Punkte G und I_1.

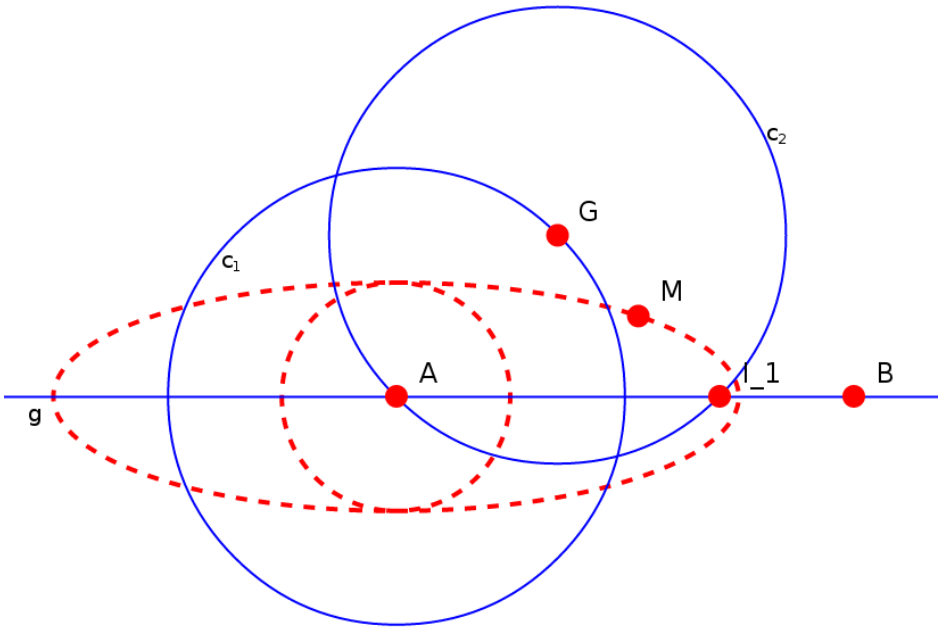


Abbildung 3.6: *Die Grundkonstruktion für eine "Eikurve".*

Das Gleichungssystem zur Bestimmung der "Eikurve" ist:

$$\begin{aligned}
(u_1 - u_3)^2 + (u_2 - u_4)^2 - 16 &= 0 \\
7u_1 + 14u_2 - 8u_2 - 7u_1 - 42 &= 0 \\
(u_3 - 8)^2 + (u_4 - 7)^2 - 16 &= 0 \\
u_2x - u_2u_3 + yu_3 - u_1y + u_1u_4 - xu_4 &= 0 \\
u_1^2 - 2u_1x + u_2^2 - 2u_2y - u_3^2 + 2u_3x - u_4^2 + 2u_4y &= 0
\end{aligned} \tag{3.6}$$

Als Lösung erhalten wir die Gleichung

$$\begin{aligned}
&-x^4 - 10x^2y^2 - 9y^4 + 32x^3 + 140x^2y + 160xy^2 + 252y^3 \\
&-834x^2 - 2240xy - 3214y^2 + 9248x + 20300y - 51121 = 0
\end{aligned} \tag{3.7}$$

Erweitern wir diese Konstruktion um einen Kreis um M und schneiden diesen mit g, erhalten wir einen Schnittpunkt I_2. Da wir nun den Ort von I_2 bestimmen wollen, erhält dieser Punkt die Koordinaten (x, y) und M wird im Gleichungssystem 3.8 von den Koordinaten (u_5, u_6) repräsentiert. Die restlichen Punkte behalten ihre Koordinaten. Als Ortskurve erhalten wir die Gleichung $y = 7$: Eine Gerade, die identisch zu l ist. In Abbildung 3.7 ist die Lösung der Teil der schwarz gestrichelten Kurve, die auf der Geraden g liegt.

$$\begin{aligned}
u_2u_5 - u_2u_3 + u_6u_3 - u_1u_6 + u_1u_4 - u_5u_4 &= 0 \\
u_1^2 - 2u_1u_5 + u_2^2 - 2u_2u_6 - u_3^2 + 2u_3u_5 - u_4^2 + 2u_4u_6 &= 0 \\
(u_1 - u_3)^2 + (u_2 - u_4)^2 - 16 &= 0 \\
7u_1 + 14u_2 - 8u_2 - 7u_1 - 42 &= 0 \\
(u_5 - 8)^2 + (u_4 - 7)^2 - 16 &= 0 \\
7x + 14y - 8y - 7x - 42 &= 0 \\
(x - u_5)^2 + (y - u_6)^2 - 9 &= 0
\end{aligned} \tag{3.8}$$

Wird die Konstruktion anstatt um M einen Kreis zu ziehen ein Gleiter $C(u_1, u_2)$ auf der in Gleichung 3.7 und Abbildung 3.6 dargestellten Kurve gelegt und um diesen ein Kreis gezogen, so erhalten wir bei Betrachtung des Schnittpunkts $I_3(x, y)$ das Gleichungssystem

$$\begin{aligned}(u_2^2 - 14u_2 + u_1^2 - 16u_1 + 109)(9u_2^2 - 126u_2 + u_1^2 - 16u_1 + 469) &= 0\\ 7x + 14y - 8y - 7x - 42 &= 0\\ (x - u_1)^2 + (y - u_2)^2 - 9 &= 0.\end{aligned} \tag{3.9}$$

Als Ortskurve für I_3 errechnen wir daraus

$$(y - 7)(x^2 - 16x + 59) = 0. \tag{3.10}$$

Bei der Verkettung erhalten wir zusätzlich zwei Geraden $x = \pm\sqrt{5} + 8$ (siehe Abbildung 3.7).

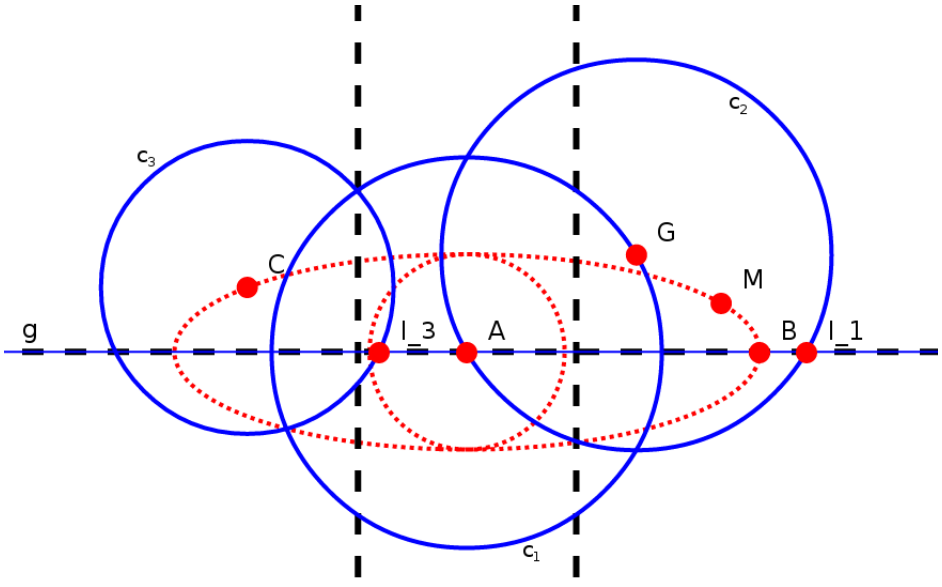


Abbildung 3.7: *Eine Erweiterung der "Eikurve".*

Kapitel 4

JessieCode

4.1 Motivation & Ziele

Das Hauptziel hinter der Entwicklung von JessieCode war ursprünglich das Verhindern von Cross-Site-Scripting (XSS) Attacken, um JSXGraph auch auf kollaborativen Webplattformen wie Wikis, Foren oder allgemein Plattformen mit von Benutzern generierten Inhalten einsetzen zu können.

Das Problem bei von Benutzern generierten Inhalten ist, dass die Eingabe von beliebigem JavaScript Code normalerweise unterbunden werden muss, da dies ein Sicherheitsrisiko darstellt. Durch JavaScript erhalten böswillige Benutzer Zugriff auf den gesamten DOM-Baum der Seite. Unter Umständen sind auch sicherheitsrelevante Daten wie Cookies oder der localStorage von anderen Benutzern betroffen, sobald diese die Inhalte des Angreifers in ihrem Browser laden. Ein einfacher Angriff wäre das Auslesen und Versenden der SessionID eines anderen Nutzers, mit der dessen Account zumindest temporär von einem Angreifer kontrolliert werden kann. Dank des uneingeschränkten Zugriffs auf den DOM-Baum ist es einem Angreifer möglich, ein Ende der Sitzung nachzuahmen, mit dem ein Nutzer zur Eingabe seiner Zugangsdaten in ein gefälschtes Anmeldeformular animiert wird. Die Zugangsdaten lassen sich anschließend unbemerkt verschicken.

Um XSS-Angriffe zu verhindern gibt es verschiedene Strategien. In den meisten Fällen bietet es sich an, HTML und damit JavaScript komplett aus der

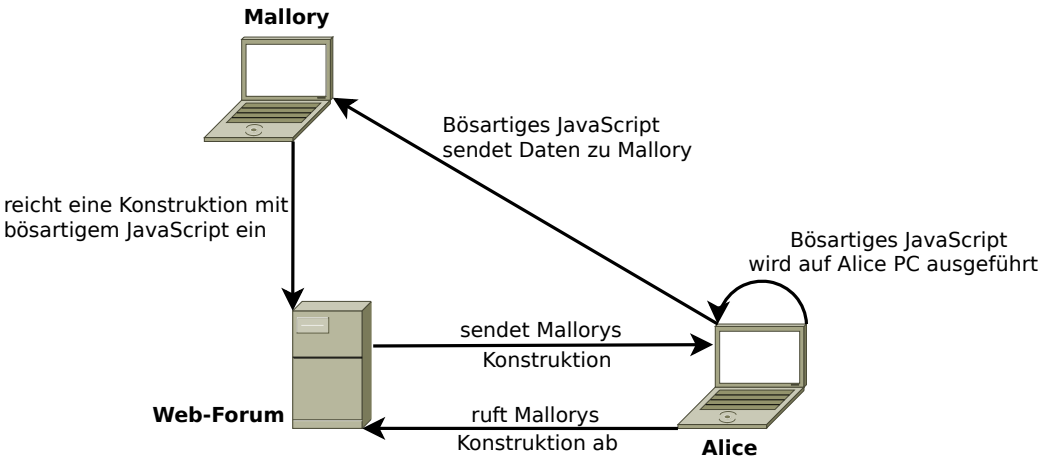


Abbildung 4.1: *Graphische Darstellung eines XSS Angriffs.*

Eingabe heraus zu filtern. Werden HTML und/oder JavaScript Eingaben benötigt, gibt es die Möglichkeit solche Eingaben bei der Darstellung über Subdomains in einen `iframe` zu laden und so von der eigentlichen Seite weitgehend abzuschotten. Google stellt mit Caja [14] eine Softwarelösung bereit, mit der diese Inhalte in "virtuellen iframes" ausgeführt werden und dadurch von der restlichen JavaScript-Laufzeitumgebung abgeschottet werden.

JavaScript-Eingaben filtern ist für JSXGraph nicht praktikabel, weil es ohne JavaScript keine JSXGraph-Konstruktion gibt. Google Caja benötigt eine Installation und Konfiguration auf Seiten der Entwickler die JSXGraph basierte Nutzerbeiträge in ihren Projekten zulassen wollen.

Mit JessieCode ist es möglich, den Zugriff auf die JSXGraph Konstruktion selbst zu beschränken und gleichzeitig die Flexibilität einer Skript-Sprache und der JSXGraph API nicht zu verlieren.

Um mit JessieCode XSS Angriffe zu verhindern, wird der Zugriff auf `document` und `window` unterbunden. Textelemente filtern HTML Tags aus übergebenen Strings. Per Whitelists wird der Zugriff auf Eigenschaften und Methoden von JSXGraph-Elementen und JavaScript-Objekte eingeschränkt.

Neben den genannten technischen Aspekten wird in Kapitel 7 mit SketchBin ein weiterer Grund für die Entwicklung von JessieCode vorgestellt.

4.2 Architektur

JessieCode besteht aus

- einem Tokenizer und Parser,
- einem Interpreter,
- einem Compiler,
- Speicher.

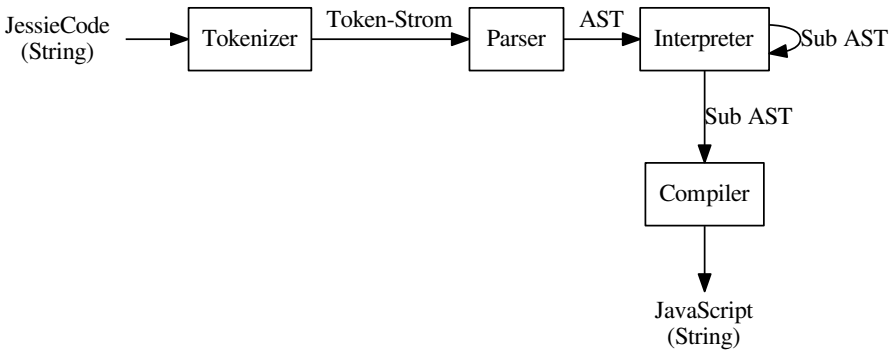


Abbildung 4.2: *JessieCode Architektur.*

4.2.1 Tokenizer und Parser

Der Tokenizer wird zusammen mit dem LALR(1) Parser von *Jison* [7] generiert. Dabei wird anhand einer Liste von Token-Definitionen und einer Syntaxbeschreibung in einer der Backus-Naur-Form ähnelndem Format ein JavaScript-Programm generiert.

Das resultierende JavaScript akzeptiert ein JessieCode Programm als Eingabe. Zunächst zerlegt der Tokenizer das Programm in eine Sequenz von Tokens. Ein Token repräsentiert einen Bezeichner, ein Schlüsselwort, einen

Operator oder eine Konstante. Kommentare und Whitespaces werden verworfen. Diese Tokensequenz wird anschließend vom Parser entsprechend der in der Grammatik definierten Produktionsregeln analysiert und ein Syntaxbaum (AST) erzeugt. Dieser Baum kann vom Interpreter und Compiler genutzt werden, um den Programmcode entweder direkt auszuführen oder nach JavaScript zu übersetzen.

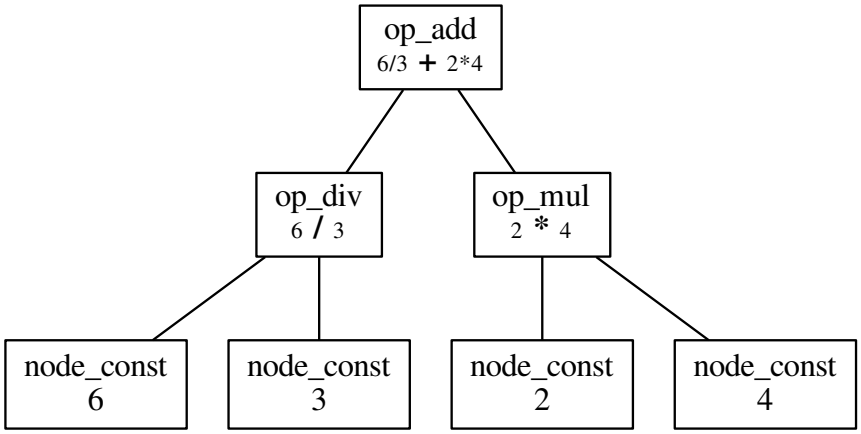


Abbildung 4.3: *Beispiel eines vereinfachten* Abstract Syntax Tree (AST). *Geparst wurde der Ausdruck* `6/3+2*4`.

4.2.2 Interpreter und Compiler

Aho, Sethi und Ullman beschreiben Interpreter und Compiler in ihrem "Drachenbuch" [1] wie folgt:

> Simply stated, a compiler is a program that reads a program written in one language – the *source* language – and translates it into an equivalent program in another language – the *target* language.
>
> Instead of producing a target program as a translation, an interpreter performs the operations implied by the source program.

Die Arbeitsweise des Interpreters und des Compilers sind weitgehend identisch, sie unterscheiden sich lediglich in der Ausgabe. Beide durchlaufen den AST ausgehend von den Blättern, wobei der Interpreter die eingelesenen Knoten sofort ausführt und der Compiler aus den Daten des Knotens entweder JavaScript- oder JessieCode generiert.

Die Möglichkeit des Compilierens nach JessieCode mag auf den ersten Blick unsinnig erscheinen, findet jedoch Anwendung in graphischen Oberflächen wie sketchometry. Dort ist es dem Benutzer meist möglich, berechnete Elemente zu definieren. Das sind beispielsweise Texte, die Berechnungen ausgeben oder Punkte, deren Koordinaten von anderen Elementen abhängen. Diese Abhängigkeiten werden vom Benutzer angegebenen, indem andere Elemente anhand ihres Namens referenziert werden. Im weiteren Verlauf der Konstruktion können die Namen dieser Elemente vom Benutzer geändert werden. Um die Konsistenz der Konstruktion zu gewährleisten, filtert der Parser den Zugriff auf Elemente per Name und ersetzt sie durch eine interne Referenz die der Benutzer nicht beeinflussen kann. Will der Benutzer die Berechnung bearbeiten wird nicht die Originaleingabe des Benutzers geladen und angezeigt, sondern der gespeicherte AST der Berechnung verwendet, darin die Referenzen durch die aktuellen Namen ersetzt und der AST nach JessieCode compiliert und ausgegeben.

4.2.3 Laufzeitumgebung

Die Klasse JessieCode bildet, zusammen mit JSXGraph, eine Laufzeitumgebung für JessieCode-Programme. Neben dem API regelt JessieCode (siehe Abschnitt 4.3) vor allem die Speicherverwaltung.

Jede Instanz der Klasse JessieCode verwaltet ihren eigenen Speicher. Dabei wird grundsätzlich auf die Speicherverwaltung der jeweiligen JavaScript VM zurückgegriffen. Es werden lediglich Symboltabellen geführt, in denen den Variablennamen ihre jeweiligen Werte und Referenzen zugeordnet werden. Diese Tabellen werden in sogenannten Scope-Objekten gespeichert.

Unter einem *Scope* versteht man den Sichtbarkeitsbereich von Variablen. In JessieCode werden, genau wie in JavaScript, neue Scopes nur durch Funktionen definiert. Bei jedem Funktionsaufruf wird daher für den jeweiligen Aufruf ein neues Scope-Objekt initialisiert, welches das jeweils aktuelle ersetzt. Initialisiert wird JessieCode mit einem Scope-Objekt für globale Variablen. In

jedem neuen Scope-Objekt wird eine Referenz auf das vorherige gesetzt, um das aus JavaScript bekannte *Scope-Chaining* zu imitieren. Damit sind auch in JessieCode *Closures* möglich, das heißt eine innerhalb einer Funktion definierten Funktion kann auf die lokalen Variablen und Parameter der äußeren Funktion zugreifen.

In Listing 4.1 werden diese Sachverhalte exemplarisch dargestellt. Auch wenn JessieCode noch nicht ausführlich vorgestellt wurde, sollte das Beispiel mit JavaScript-Kenntnissen nachvollziehbar sein. Ansonsten sei auf den folgenden Abschnitt verwiesen, in dem die verwendeten Datentypen, Operatoren und Funktionen erläutert werden.

Im Beispiel werden drei Scopes angelegt. Der globale Scope wird zusammen mit der JessieCode-Instanz initialisiert. Dort werden drei Symbole abgelegt: $x = 2, g$ und f. Beim Aufruf der Funktion $g()$ wird ein neues Scope-Objekt angelegt, in dem der Variable x der Wert 5 zugewiesen wird. Diesem Scope wird eine Referenz auf das globale Scope-Objekt zugewiesen. Beim Aufruf der Funktion f mit dem Parameter 7 wird erneut ein neues Scope-Objekt angelegt. Diesmal wird jedoch das Scope-Objekt referenziert, das beim Aufruf der Funktion g angelegt wurde. Bei der Berechnung des Rückgabewerts muss der JessieCode-Interpreter die beiden Variablen x und y auflösen. Die Variable y ist im aktuellen Scope hinterlegt, für die Variable x allerdings muss der Interpreter in der Scope-Chain zurückgehen, bis er in einem Scope-Objekt eine Symboltabelle findet, in der eine Variable x definiert ist. In diesem Fall ist dies bereits beim nächsten Scope-Objekt der Fall. Dort hat x den Wert 5. Das Ergebnis, das die *log*-Funktion ausgibt, ist daher 12.

Listing 4.1: ***Scope und Closures***

```
x = 2;

g = function () {
    x = 5;

    return function (y) {
        return x + y;
    };
};

f = g();
$log(f(7));
```

4.3 JessieCode Referenz

4.3.1 Datentypen

Da der JessieCode Parser in JavaScript implementiert ist, bietet es sich an auf die dort implementierten Datentypen zurückzugreifen. Daher wird im weiteren Verlauf dieses Abschnitts vorwiegend auf die Unterschiede unter den Datentypen in JessieCode und JavaScript eingegangen.

Zahlenwerte werden vom Datentyp *Number* in Form eines *double precision floating point* nach IEEE 754 [18] dargestellt. Die besonderen Werte *unendlich*, *-unendlich* und *not-a-number* in IEEE 754 werden in JessieCode von `+Infinity`, `-Infinity` sowie `NaN` repräsentiert. Vordefinierte Konstanten sind `PI` für π und `EULER` für die Eulersche Zahl e. Im Gegensatz zu JavaScript ist Number in JessieCode jedoch kein *wrapper object*, hat demnach auch keine Methoden zum Runden oder konvertieren in andere Datentypen wie Strings.

Für Zeichenketten existiert der Datentyp *String*. Genau wie *Numbers* sind Strings vom gleichnamigen Datentyp in JavaScript abgeleitet. Sie sind jedoch keine Objekte, haben demnach keine eigenen Methoden und entsprechen daher simplen Zeichensequenzen. Strings werden durch zwischen einfachen oder doppelten Anführungszeichen begrenzten UTF-16 Zeichenketten definiert. Escape-Sequenzen werden unterstützt und durch ein Backslash \ eingeleitet.

Boolesche Werte werden im Datentyp *Boolean* abgelegt. Gültige Werte sind `true` und `false`.

Der Datentyp *Array* ist für das Speichern von Listen vorgesehen. Er wird, genau wie sein JavaScript-Äquivalent, intern als Objekt behandelt mit fast den gleichen Eigenschaften und Methoden. Nicht zur Verfügung stehen der `Array()` Konstruktor sowie dessen Methode `isArray()`. Die einzige Möglichkeit ein Array zu erhalten sind durch Rückgabewerte vordefinierter Methoden und Funktionen und durch Literalnotation (siehe Listing 4.2). Ein Array besteht aus keinem oder mindestens einem Element, dessen Wert einen beliebigen Datentypen haben kann, insbesondere auch ein Array. Es lassen sich beliebige Datentypen im selben Array kombinieren. Der Zugriff auf einzelne Elemente eines Arrays erfolgt über den `[]` Operator mittels eines Index. Das

erste Element besitzt den Index `0`.

Listing 4.2: ***Definition eines Arrays in JessieCode***

```
a = [1, 2, 3];
b = ['Eine Zeichenkette', 42, <<an: 'object'>>];

// Erzeugt einen Text an Position (2, 3) mit dem
// Inhalt 'Eine Zeichenkette'
text(a[1], a[2], b[0]);
```

Funktionen werden in JessieCode als Werte vom Typ *function* angesehen, auf die der `()` Operator angewendet werden kann. Definiert werden Funktionen durch ein Konstrukt analog den *anonymous function expressions* in JavaScript: Das Keyword `function`, gefolgt von einer Liste von Parametern umschlossen von `()`, gefolgt vom Funktionskörper umschlossen von `{}` (siehe Listing 4.3). Im Gegensatz zu ihrem JavaScript-Pendant ist eine Funktion in JessieCode kein Objekt und besitzt deshalb keinerlei Eigenschaften oder Methoden.

Listing 4.3: ***Definition einer Funktion in JessieCode***

```
// Eine Funktion, die keine Parameter erwartet
// und immer 0 zurueckliefert
f = function () {
        return 0;
};

// Eine Funktion, die zum uebergegebenen Wert 1
// hinzuaddiert und das Ergebnis zurueckgibt
g = function (x) {
        return x + 1;
};
```

Abbildungen sind eine Spezialisierung des Funktionsdatentyps. Anstatt eines Anweisungsblocks ist lediglich ein einzelner Ausdruck erlaubt, in dem ausschließlich mathematische Ausdrücke und Funktionsaufrufe erlaubt sind. Momentan ist der Einsatzzweck einer Abbildung identisch mit dem einer Funktion. Geplant ist die Verwendung als Element, auf das (symbolische) Integrations- und Differentiationsoperatoren angewandt werden können. Die Definition einer Abbildung wird eingeleitet durch das Schlüsselwort `map` gefolgt von einer Parameterliste wie bei einer Funktionsdefinition. Anstelle eines

Anweisungsblocks folgt ein Pfeil und ein einzelner Ausdruck (siehe Listing 4.4).

Listing 4.4: ***Definition einer Abbildung in JessieCode***

```
// Eine Abbildung, die keine Parameter erwartet
// und immer 0 zurueckliefert
f = map () -> 0;

// Eine lineare, Abbildung die zum uebergegebenen
// Wert 1 hinzuaddiert
g = map (x) -> x + 1;
```

Der *object*-Datentyp dient in JessieCode in erster Linie zur Ablage von Elementen und zum Erstellen von Attributlisten. Dementsprechend sind die Möglichkeiten zum Erzeugen von Objekten eingeschränkt – analog den Arrays sind sie nur durch vordefinierte Funktionen wie zum Beispiel Creator und Literalnotation erstellbar. Konstruktorfunktionen stehen nicht zur Verfügung.

Listing 4.5: ***Definition eines Objekts in JessieCode***

```
// A ist ein Objekt, eine Instanz der JavaScript
// Klasse JXG.Point.
A = point(1, 1);

// Ein Objektliteral, das spaeter als Attributobjekt
// Verwendung findet.
red = <<
    strokeColor: 'red',
    fillColor: '#ff1122',
    opacity: 0.5
>>;
B = point(3, 4) red;

// Erzeugt eine Gerade
line(A, B);
```

4.3.2 Creator-Funktionen

Ein Creator ist eine vordefinierte Funktion, die zur Erstellung von Elementen dient. Der Name ist angelehnt an die Methode `create(type, parents, attributes)` der Klasse `JXG.Board` in JSXGraph. Tatsächlich sind JessieCode Creator stark mit `JXG.Board.create()` verknüpft. Jedes in JSXGraph verfügbare Element hinterlegt über `JXG.registerElement()` in dem internen Verzeichnis (`JXG.elements`) einen Verweis auf eine Funktion mit dem fest definierten Parameterschema `(board, parents, attributes)`. `create()` nutzt dieses Verzeichnis, um die dort hinterlegten Funktionen aufzurufen.

Trifft der JessieCode Interpreter bei einem Funktionsaufruf auf eine Funktion, die nicht innerhalb der JessieCode Instanz aufgelöst werden kann, so wird unter anderem in diesem Elementverzeichnis nachgeschlagen, ob ein solches Element registriert wurde. Ist dies der Fall, wird angenommen dass ein Element erstellt werden soll und `JXG.Board.create(type, parents, attributes)` aufgerufen. Der Funktionsname wird als `type` Parameter übergeben. Alle Parameter des Funktionsaufruf werden in einem Array zusammengefasst und als `parents` übergeben. Elementattribute werden zwischen dem Funktionsaufruf und dem den Funktionsaufruf abschließenden Strichpunkt `;` als Liste von Objekten angegeben. Diese Liste wird zu einem Objekt zusammengeführt. Die Priorität der Objekteigenschaften steigt hierbei von links nach rechts: Werden mehrere Werte für das Attribut `strokeColor` angegeben, wird nur der am weitesten rechts angegebene Wert übernommen.

Diese Herangehensweise erlaubt eine flexible Integration von JSXGraph-Elementen in JessieCode. Sobald ein neues Element hinzugefügt wird oder sich Änderungen an existierenden Elementen ergeben, stehen diese Änderungen sofort und uneingeschränkt in JessieCode zur Verfügung.

Beispiele für Creator und das Übergeben eines Attributobjekts wurden bereits in Listing 4.5 gegeben. Ein Beispiel für Attributlisten ist in Listing 4.6 zu sehen.

Listing 4.6: ***Beispiel für Attributlisten***

```
small = <<
    size: 1
>>;
```

```
big = <<
        size: 7
>>;

red = <<
    strokeColor: 'red',
    fillColor: '#ff1122'
>>;

// Erzeugt einen grossen roten Punkt
A = point(3, 4) big, red;

// Erzeugt einen kleinen blauen Punkt
B = point(4, 2) big, small, <<color: 'blue'>>;
```

4.3.3 Kommentare

In JessieCode stehen zwei Möglichkeiten zur Verfügung, den Quelltext durch Kommentare zu ergänzen. Zwei Schrägstriche `//` machen sämtliche Eingaben bis zum nächsten Zeilenende zum Kommentar. Mehrzeilige Kommentare werden eingeleitet durch einen Schrägstrich gefolgt von einem Asterisk `/*` und durch ein Asterisk gefolgt von einem Schrägstrich `*/` beendet. Kommentare werden vom Parser ignoriert.

4.3.4 Operatoren

Logische Operatoren

JessieCode stellt die drei logischen Operationen *OR*, *AND* und *NOT* mit Operatoren `||`, `&&` und `!` zur Verfügung. Alle drei verhalten sich wie ihr JavaScript-Äquivalent.

Vergleichsoperatoren

Von JavaScript übernommen wurden auch ==, <=, >=, <, > und != zum Vergleichen zweier Werte ohne Typprüfung. Sie verhalten sich genauso wie ihre Pendants. In der Tat greift JessieCode hier direkt auf die JavaScript Vergleichsoperatoren zurück, wie in Listing 4.7 beispielhaft anhand der Implementierung des == Operators zu sehen ist.

Listing 4.7: ***JessieCode Implementierung des == Operators***

```
case 'op_equ':
    // == is intentional
    ret = this.execute(node.children[0]) ==
        this.execute(node.children[1]);
    break;
```

Hinzu kommt ein neuer Operator *annähernd gleich* ~=, der zum Vergleich zweier Floatwerte dient und den klassischen Vergleich über den Absolutwert der Differenz ersetzt. Die Genauigkeit wird über `JXG.eps` gesteuert.

Zuweisungsoperator

Von den zahlreichen Zuweisungsoperatoren in JavaScript wurde nur das grundlegende = übernommen.

Arithmetische Operatoren

Auch die arithmetischen Operatoren wurden weitgehend übernommen. Allerdings wurden diese in ihrer Funktionalität stark erweitert. So ist es mit den Operatoren +, -, *, / sowie % möglich nicht nur Zahlen zu addieren und subtrahieren, sondern in bestimmten Fällen auch Vektoren in Form von Arrays. Der Additions- und der Subtraktions-Operator erwartet entweder zwei Skalarwerte oder zwei Vektoren. Im Falle von zwei Vektoren wird die Operation komponentenweise durchgeführt. Die Multiplikation operiert auf zwei Skalaren, einem Skalarwert und einem Vektor (Reihenfolge irrelevant) oder zwei Vektoren. Ein Skalarwert wird komponentenweise mit dem Vektor multipliziert, die Multiplikation zweier Vektoren entspricht dem Skalarprodukt der beiden Vektoren. Der Divisions- beziehungsweise der Modulo-Operator

erwartet zwei Skalare oder einen Vektor und einen Skalar (Reihenfolge wichtig). Im letzteren Fall wird die Division beziehungsweise der Modulus komponentenweise berechnet, das Ergebnis ist wieder ein Vektor.

Eine andere Bedeutung als in JavaScript hat der `^` Operator dessen Ergebnis von `a^b` die Potenz a^b ist. In JavaScript handelt es sich hierbei um den `XOR` Operator, der in JessieCode, genau wie die gesamte bitweise Arithmetik, wegfällt.

Zeichenkettenoperatoren

Zur Konkatenation von Zeichenketten dient der `+` Operator. Ist einer der beiden Operanden keine Zeichenkette, so wird vor der Konkatenation eine implizite Umwandlung durchgeführt.

Spezielle Operatoren

Für den Zugriff auf Elemente eines Arrays dient der `[]`-Operator. Dieser dient gleichzeitig auch zum Zugriff auf Eigenschaften und Methoden eines Objekts, genau wie der `.`-Operator.

Der konditionale Operator `?:` liefert, wie in JavaScript auch, einen von zwei Werten zurück. Die Auswahl des zurückgelieferten Wertes beruht auf einer Bedingung: Ist das Ergebnis der Bedingung gleich `true`, wird der Wert vor dem `:` zurückgeben. Ist das Ergebnis gleich `false` der Wert nach dem `:`.

Der Aufruf einer Funktion oder einer Methode wird durch den `()`-Operator initiiert.

4.3.5 if-Anweisung

Listing 4.8: ***Syntax der if-Anweisung***

```
"IF" "(" Expression ")" Statement1
"IF" "(" Expression ")" Statement1 "ELSE" Statement2
```

Zunächst wird der Ausdruck *Expression* in den Klammern ausgewertet. Ist das Ergebnis des Ausdrucks *false*, also gleich dem Boolean-Wert `false`, einem leeren String oder `0`, wird, falls vorhanden, *Statement2* ausgeführt. In jedem anderen Fall wird *Statement1* ausgeführt und *Statement2*, falls vorhanden, übersprungen.

Im weiteren Verlauf wird false als Ergebnis eines Ausdrucks und der Boolean-Wert `false` synonym verwendet.

4.3.6 Iterationsanweisungen

while

Listing 4.9: ***Syntax der while-Anweisung***

```
"WHILE" "(" Expression ")" Statement
```

Die Anweisungen in *Statement* werden solange ausgeführt, bis die Auswertung von *Expression* false zurückliefert. Es ist möglich, dass die Anweisungen nie ausgeführt werden.

do-while

Listing 4.10: ***Syntax der do-while-Anweisung***

```
"DO" Statement "WHILE" "(" Expression ")"
```

Die Anweisungen in *Statement* werden solange ausgeführt, bis die Auswertung von *Expression* false liefert. *Statement* wird mindestens einmal ausgeführt.

for

Listing 4.11: ***Syntax der for-Anweisung***

```
"FOR" "(" Expression1 ";"
  Expression2 ";" Expression3 ")" Statement
```

Statement wird so lange wiederholt, bis *Expression2* false ergibt. *Expression1* und *Expression3* können auch leere Ausdrücke sein. Üblicherweise wird

Expression1 zur Initialisierung einer Laufvariable genutzt und nur zu Beginn ausgewertet. Das Ergebnis wird verworfen. *Expression3* kann zur Aktualisierung der Laufvariable verwendet werden und wird nach jedem Durchlauf einmal ausgewertet. Auch dieses Ergebnis wird nicht verwendet.

Zu beachten ist, dass es sich hierbei ausschließlich um die aus der Programmiersprache `C` bekannte for-Anweisung handelt. Die aus JavaScript bekannte `for ( ... in ... )` Anweisung steht nicht zur Verfügung.

4.3.7 Vordefinierte Konstanten

Alle vordefinierten Zahlenkonstanten sind aus JavaScript übernommen. Dazu gehören die weiter oben bereits angesprochenen speziellen Werte `-Infinity`, `+Infinity` und `NaN` sowie alle Konstanten des statischen JavaScript Objekts `Math`: die Eulerschen Zahl `E`, `LN2`, `LN10`, `LOG2E`, `LOG10E`, `PI`, `SQRT1_2` und `SQRT2`. Die Eulersche Zahl e wird in JessieCode `EULER` genannt, da für Variablen von Punktelementen oftmals Großbuchstaben herangezogen werden und bei mehr als vier Punkten die Konstante `E` schnell versehentlich überschrieben würde.

4.3.8 Vordefinierte Funktionen

Vordefinierte Funktionen sind Funktionen, die im globalen Scope dem Benutzer automatisch zur Verfügung stehen.

log(<*>)

Gibt die übergebenen Werte in der JavaScript Entwicklerkonsole aus, falls die Umgebung `console.log()` bereitstellt.

import(<string>)

Lädt einen JSXGraph Namespace. Zur Verfügung stehen `'math'` für `JXG.Math`, `'math/geometry'`, `'math/statistics'` und `'math/numerics'` zum Zugriff auf `JXG.Math.Geometry`, `JXG.Math.Statistics` beziehungsweise

`JXG.Math.Numerics`. Der Rückgabewert ist eine Referenz auf den entsprechenden Namespace. Die Methoden, die von den jeweiligen Namespaces bereitgestellt werden, können in der JSXGraph Dokumentation[1] nachgeschlagen werden.

$(<string, object, function>)

Durchsucht das Board nach Elementen. Im Fall eines Strings wird dieser zuerst als ID interpretiert. Existiert kein Element mit dieser ID, so wird versucht ein Element mit diesem Namen zu finden. Wird ein Objekt übergeben, dann werden dessen Eigenschaften durchlaufen und die angegeben Werte mit den entsprechenden Werten der Eigenschaften und Attribute aller Elemente verglichen. Funktionen werden vor dem Vergleich evaluiert. Jedes Element, das zu allen gefragten Eigenschaften ein Attribut oder eine Eigenschaft mit dem gleichen Wert besitzt, wird in eine Liste übernommen. Aus dieser wird die zurückgegebene Composition erzeugt. Kann kein Element gefunden werden, wird die Eingabe zurückgegeben.

Mathematische Funktionen

Der Großteil der vordefinierten mathematischen Funktionen ist aus dem globalen JavaScript Math-Objekt importiert. Diese Integration ist dynamisch implementiert, das heißt trifft der Interpreter auf ein undefiniertes Symbol, schlägt er unter anderem im Math-Objekt nach, ob es dort eine Methode mit dem selben Namen gibt. Ist dies der Fall, wird diese Methode aufgerufen und deren Ergebnis zurückgegeben. Daher soll an dieser Stelle ein Verweis auf die ausgezeichnete Dokumentation dieser Methoden im Mozilla Developer Network genügen[2].

Diese Funktionen werden ergänzt durch `trunc(<number> n, <number> p)`. `trunc` schneidet die Zahl `n` nach der `p`-ten Dezimalstelle ab. `p` ist optional, wird es nicht angegeben werden alle Dezimalstellen verworfen und eine ganze Zahl zurückgegeben.

[1] http://jsxgraph.uni-bayreuth.de/docs/, März 2014
[2] https://developer.mozilla.org/, März 2014

Um JessieCode auch zum Interpretieren von Ausdrücken aus GEONExT-Dateien benutzen zu können, wurden weitere globale Funktionen hinzugefügt:

- `V(<slider,angle>)` zum Abfragen des Wertes eines Sliders oder zum Messen eines Winkels,
- `L(<line>)` bestimmt die Länge eines Segments,
- `X(<point>)` und `Y(<point)` zum Ermitteln der X- beziehungsweise Y-Koordinate eines Punktes,
- `dist(<point>, <point>)` berechnet die Entfernung zwischen zwei Punkten,
- Zur Berechnung eines Winkels in Grad steht `deg(<point>, <point>, <point>)` zur Verfügung. Den Wert in Radiant berechnet `rad(<point>, <point>, <point>)`.

Bei `dist`, `deg` und `rad` ist es auch möglich mit einem Großbuchstaben zu beginnen.

4.3.9 $board-Objekt

- `update()` aktualisiert die Daten aller Elemente eines Boards, deren Attribut `needsRegularUpdate` auf `true` gesetzt ist, initialisiert eine Neuzeichnung (Rendering in HTML5 2D Canvas) oder Aktualisierung des DOM-Unterbaums (Rendering über SVG und VML) und ruft die `update()` Methoden aller abhängigen Boards auf.
- `fullUpdate()` funktioniert wie `update()` mit der Ausnahme, dass die Daten aller Elemente aktualisiert werden, unabhängig vom Wert des Attributs `needsRegularUpdate`.
- `suspendUpdate()`, `unsuspendUpdate()` dienen zur temporären Unterbindung von Updates, sowohl vom Benutzer als auch automatisch angestoßene.
- `on(<string>, <function>)` registriert einen neuen Event-Handler. Erwartet werden ein Eventtyp in Form eines Strings und eine Funktion die

aufgerufen wird, wenn der Event eintritt. Eine Liste aller zur Verfügung stehender Events ist in der JSXGraph API Referenz zu finden[3].

- `off(<string>, <handler>)` entfernt einen Event-Handler. Benötigt wird ein String, der angibt, welcher Eventtyp entfernt werden soll. Wird keine Handlerfunktion angegeben entfernt `off()` sämtliche für diesen Eventtyp registrierten Handler.

- `trigger(<string>, <array>)` löst einen bestimmten Event aus. Der Eventtyp wird als String angegeben. Eventuell an die Handler-Funktionen zu übergebende Parameter können in Form eines Array angegeben werden.

- `setView(<array>, <boolean>)` ändert den sichtbaren Bildausschnitt des Boards auf den im Array angegebenen Bereich. Das Array muss vier Zahlenwerte enthalten. Der erste Wert bestimmt die Begrenzung auf der x-Achse zum linken Rand, der zweite auf der y-Achse zum oberen Rand. Die beiden nächsten Werte begrenzen das Sichtfeld am rechten beziehungsweise unteren Rand. Der zweite Parameter ist optional (default: `false`). Ist dieser gleich `true`, werden die angegebenen Begrenzungen so angepasst, dass das Seitenverhältnis des umgebenden HTML Elements übernommen wird. Das angegebene Sichtfeld bleibt hierbei auf jeden Fall sichtbar und wird gegebenenfalls zentriert dargestellt. Alternativ kann anstelle von `setView` auch der von JSXGraph bekannte Alias `setBoundingBox` genutzt werden.

- `migratePoint(<point>, <point>)` verschmilzt zwei Punkte zu einem Punkt. Alle vom ersten Punkt ausgehenden Abhängigkeiten werden übertragen auf den zweiten Punkt. Der erste Punkt wird vom Board entfernt.

- `colorblind(<string>)` ändert alle Farben aller Elemente um Farbenfehlsichtigkeit zu emulieren. Zur Auswahl stehen Protanopia, Deuteranopia, Tritanopia. Die Funktion erwartet einen dieser Begriffe in einem String um die Emulation einzuschalten. Ein Aufruf mit einem beliebigen anderen Wert beendet eine eventuell laufende Emulation.

- `clearTraces()` entfernt alle eventuell vorhandenen Spurkurven, die durch das Elementattribut `trace` entstanden sind.

- `left(), right(), up(), down()` verschiebt den Bildausschnitt nach links, rechts, oben beziehungsweise unten.

[3] http://jsxgraph.uni-bayreuth.de/docs/symbols/JXG.Board.html, März 2014

- `zoomIn(<number>, <number>), zoomOut(<number>, <number>)` vergrößert beziehungsweise verkleinert den Maßstab der Konstruktion. Die beiden optionalen Parameter ergeben eine Position auf dem Board, die im Fokus des Zooms steht. Werden keine Koordinaten angegeben liegt das Zentrum im Fokus.

- `zoom100()` setzt den Maßstab zurück auf 1.

- `zoomElements(<array>)` modifiziert den Maßstab derart, dass alle im Array angegebenen Elemente sichtbar sind.

- `remove(<element>), removeObject(<element>)` entfernt ein Element vom Board.

Kapitel 5

Practice

Bei E-Learning-Angeboten gibt es neben der reinen Präsentation von (interaktiven) Inhalten auch das Bedürfnis, Wissen und Können automatisiert zu prüfen. Es existieren bereits Projekte, die solch ein sogenanntes *E-Assessment*-System implementieren. Ein solches Projekt ist Numbas[1], das an der Newcastle University entwickelt wird. Numbas akzeptiert als Eingaben verschiedene Arten von multiple choice, durch Reguläre Ausdrücke analysierte Texteingaben sowie Numerische und Algebraische Ausdrücke.

Das Übungsmodul (dort *Assessments* genannt) von CK12[2] besitzt einen ähnlichen Funktionsumfang.

Mit der JSXGraph-Erweiterung *practice*[3] wird in diesem Kapitel eine Möglichkeit beschrieben, wie solche Assessment-Systeme um graphische Eingaben erweitert und wie diese Eingaben numerisch ausgewertet werden können.

Zur numerischen Auswertung kann eine Konstruktion auf verschiedene Eigenschaften untersucht werden, indem die Existenz der erwarteten Elemente überprüft wird und deren Eigenschaften mit erwarteten Werten abgeglichen werden. Wenn zum Beispiel der Nutzer einen Punkt an der Stelle $(2, 4)$ konstruieren soll, wird überprüft, ob ein Punkt existiert und ob seine X-Koordinate beziehungsweise Y-Koordinate den Wert 2 beziehungsweise 4 hat. Mit der Komplexität der Konstruktion steigt allerdings auch die Komplexität dieser Abfragen.

[1] http://www.ncl.ac.uk/maths/numbas/, März 2014
[2] http://ck12.org, März 2014
[3] http://jsxgraph.github.io/practice/, März 2014

Um ein rechtwinkliges Dreieck erkennen zu können, muss zuerst die Existenz dreier Punkte und eines Polygons festgestellt werden. Anschließend müssen drei Winkel untersucht werden. Zur Erkennung eines gleichseitigen Dreiecks müssen, zusätzlich zu den Existenzprüfungen, alle Seiten des Polygons paarweise verglichen werden.

Diese kombinatorischen Aspekte können von practice übernommen werden.

5.1 Nomenklatur

Der erste Schritt besteht in der Definition einer Konstruktion. Dies geschieht bei practice mit mindestens einem *Verifier*. Ein Verifier ist eine Klasse, mit der eine Eigenschaft der Konstruktion beschrieben wird.

Der *Line-Verifier* beispielsweise überprüft die Existenz und die Elternelemente einer Geraden. Er erwartet bei seiner Initialisierung drei Zeichenketten. Die erste legt den Namen der Geraden fest. Die anderen beiden legen die Namen der beiden Punkte fest, durch die diese Gerade definiert ist. In Listing 5.1 sehen wir, wie der Line-Verifier benutzt wird. Außerdem ist dort der *Collinear-Verifier* zu sehen. Dieser erwartet ebenfalls drei Zeichenketten, die drei Punkte repräsentieren. In diesem konkreten Fall werden sowohl die beiden Punkte der Geraden mit A und B bezeichnet, als auch zwei der Punkte im Kollinearitätstest. Dies ist kein Zufall, diese Zeichenketten werden zur Identifikation von Elementen über mehrere Verifier hinweg genutzt. Das konkrete Beispiel überprüft daher, ob es eine Gerade durch A und B gibt und ein dritter Punkt C auf dieser Geraden liegt.

Listing 5.1: ***Überprüft, ob es eine Gerade g durch die beiden Punkte A und B gibt und ob Punkt C auf g liegt.***

```
new Assessment(
        new Verifier.Line('g', 'A', 'B'),
        new Verifier.Collinear('A', 'B', 'C')
)
```

Eine weitere neue Klasse in Listing 5.1 ist *Assessment*. Diese Klasse ist das Kernstück von practice. Sie verwaltet eine Übung und verifiziert eine konkrete Konstruktion anhand der bei ihrer Initialisierung übergebenen Verifier. Diese Liste von Verifiern nennen wir im Folgenden auch *rules* oder *Regelsatz*.

Fixture	X	Y	l
#1	A	B	g
#2	B	A	g
#3	C	D	h
#4	D	C	h

Tabelle 5.1: *Alle Fixtures, die die* `collect()` *Methode des Assessments in Listing 5.2 nach Analyse der Konstruktion in Abbildung 5.1 liefert.*

Die Klasse Assessment stellt zur Auswertung die beiden Methoden `collect()` und `verify()` zur Verfügung. Beide Methoden erwarten ein JSXGraph-Board und überprüfen, ob alle im Regelsatz des Assessments enthaltenen Verifier von der im Board enthaltenen Konstruktion erfüllt werden. Gleichzeitig werden die in den Verifiern definierten Elemente auf die Elemente des JSXGraph-Boards abgebildet. Eine solche Abbildung wird auch *Fixture* genannt. Während `verify()` nach dem ersten gefundenen Fixture die Suche beendet und das gefundene Fixture als Ergebnis zurückgibt, sucht `collect()` weiter und gibt alle Fixtures zurück.

Zur Verdeutlichung betrachten wir das Beispiel in Listing 5.2. Der Regelsatz des Assessments besteht aus nur einem Line-Verifier.

Listing 5.2: ***Überprüft, ob es eine Gerade*** l ***durch die beiden Punkte*** X ***und*** Y ***gibt.***

```
new Assessment(
        new Verifier.Line('l', 'X', 'Y')
)
```

Übergeben wir diesem Assessment die in Abbildung 5.1 gezeigte JSXGraph-Konstruktion, dann liefert `verify()` als Ergebnis ein Fixture, das X auf A, Y auf B und l auf g abbildet. `collect()` liefert vier Fixtures, die in Tabelle 5.1 gelistet werden.

Um aus einer solchen Liste an Fixtures ein geeignetes auswählen zu können, besitzt jedes Fixture einen Bewertungsvektor *score*. In diese Bewertung können Verifier eine Bewertung eintragen. *PointOnLine('A', 'l')* beispielsweise prüft, ob Punkt A annähernd auf der Geraden l liegt und trägt im Erfolgsfall als Bewertung die Distanz des Punktes zur Geraden in den Bewertungsvektor ein.

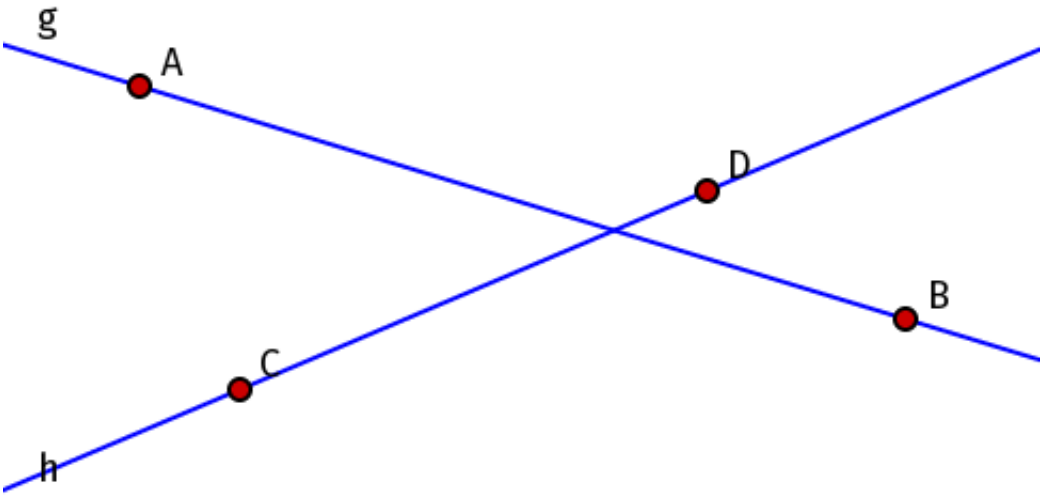


Abbildung 5.1: *Eine Beispielkonstruktion, die mit dem Assessment in Listing 5.2 ausgewertet wird.*

Neben *Elemental Verifiern* wie dem Line-Verifier und *Geometric Relation Verifiern* wie dem Collinear-Verifier gibt es auch *Compare-Verifier* wie der *Equal-Verifier* (Details siehe Abschnitt 5.2.3). Diese finden Verwendung bei der Auswertung von Zahlenwerten, wie zum Beispiel dem Abstand zweier Punkte, dem Wert der X-Koordinate eines Punktes oder der Steigung einer Geraden. Diese Werte werden den Verifiern als Instanzen der *Value-Klassen* übergeben (siehe Abschnitt 5.3).

5.2 Verifier

5.2.1 Elemental Verifier

Zum Definieren von Elementeigenschaften, wie der Typ eines Elements oder ob es sich um ein eingesammeltes oder gezeichnetes Element handelt, werden *elemental verifier* herangezogen.

- *Point(e)* prüft den Typ eines Elements und gibt `true` zurück, wenn es sich dabei um einen Punkt handelt.

- *Line(e, A, B)* ist nur `true`, wenn es sich beim übergebenen Element um eine Gerade handelt. A und B sind optional. Werden sie angegeben, muss die Gerade durch diese beiden Punkte definiert sein.
- *Angle(e, A, B, C)* prüft, ob es sich bei e um einen Winkel, der durch die drei Punkte A, B und C definiert wird, handelt.
- *Polygon(e, ...)* akzeptiert neben e eine beliebige Anzahl Punkte und bestätigt, ob e ein Polygon ist und durch die übergebenen Punkte definiert ist.
- *Tangent(e)* bestätigt, ob es sich beim Element e um eine Tangente handelt.

Bei diesen Verifiern sind immer sichere Aussagen möglich, daher wird der Score jedes dieser Tests mit dem Standardwert 0 bewertet.

5.2.2 Geometric Relation Verifier

Die Relation zweier Elemente zueinander bestimmen die *Geometric Relation Verifier*. Hier wird sehr oft auf die homogenen Koordinaten von Punkten und Geraden zurückgegriffen. Diese sind bei Geraden in der Eigenschaft `stdform` gespeichert und bei Punkten direkt in den Koordinaten.

- *Collinear(A, B, C)* verifiziert, ob die drei angegebenen Punkte kollinear sind.
- *Parallel(g, h)* ist `true`, wenn beide Geraden in etwa parallel sind. Der verwendete Test und der Score entspricht dem Steigungsvergleich analog zum MatchLines Verifier.
- Auch der *Normal(g, h)* Verifier zieht die Steigungen der zu vergleichenden Geraden heran, indem aus den jeweiligen `stdform` die erste Komponente entfernt und aus den resultierenden Vektoren das Skalarprodukt gebildet wird. Ist der Absolutwert nahe genug an der 0, werden die Linien als senkrecht zueinander angesehen. Das Ergebnis des Skalarprodukts wird als Score verwendet.

- *PointOnLine(l, P)* nutzt das Skalarprodukt der `stdform` der Linie mit den homogenen Koordinaten des Punktes, um zu ermitteln, ob der Punkt nahe genug an der Linie liegt. Das Ergebnis findet Verwendung im Score.
- *OnCircleCurve(P, c)* prüft, ob ein gegebener Punkt nahe genug an einem Kreis oder einer Kurve liegt, um näherungsweise "auf" dem Kreis oder der Kurve zu sein. Dieser Test nutzt die JSXGraph interne Methode `hasPoint`. Da `hasPoint` nur `true` oder `false` zurückgibt stehen zur Berechnung des Score nur diese beiden Werte zur Verfügung.
- *NotIdentical(a, b)* vergleicht die JSXGraph IDs der beiden Elemente, um auszuschließen, dass es sich dabei um dasselbe Element handelt. Da hier eine absolute und sichere Aussage möglich ist, wird dieser Test immer mit dem Standardscore 0 bewertet.
- *Score(v)* schlägt niemals fehl und speichert die in v angegebene Value als seinen Score.

5.2.3 Compare Verifier

Compare Verifier finden Verwendung beim Vergleich zweier Values. Die meisten sind selbsterklärend:

- *Between(a, b, c)* prüft, ob $b \leq a \leq c$.
- *Equals(a, b)* ist `true`, wenn die beide Werte a und b annähernd gleich sind (annähernd deshalb, weil ein direkter Vergleich von Gleitkommawerten in der Regel nicht funktioniert).
- *Less(a, b)* ergibt `true`, wenn $a < b$.
- *LEQ(a, b)* ist wahr, wenn der Wert von a kleiner oder annähernd gleich dem Wert von b ist.
- *Greater(a, b)* ergibt `true`, wenn $a > b$.
- *GEQ(a, b)* liefert `true`, wenn $a \geq b$ oder a annähernd gleich b.

Score(v) nimmt einen einzelnen Value v entgegen und speichert diesen Wert als seinen Score. Dies findet Verwendung bei der Linienskizze B.3. Alle anderen Compare Verifier geben den Standardscore 0 zurück.

5.3 Values

Values berechnen und speichern Zahlenwerte und werden als Parameter für Compare Verifier verwendet. Diese akzeptieren auch Variablen des JavaScript Typs `number`, wandeln die jedoch intern in eine Instanz der *Number-Value* um. Weiterhin gibt es folgende Values:

- *Angle(a)* ermittelt den Wert eines Winkels.
- *Angle3P(A, B, C)* ermittelt den Wert des Winkels $\angle(ABC)$.
- *Distance(a, b)* erlaubt die Berechnung der Distanz zweier Punkte.
- *NumberElements(class)* zählt die Anzahl der Elemente einer bestimmten Klasse, zum Beispiel kann *class* gleich 'points', 'lines', 'circles' oder 'polygons' sein.
- *SlopeY(l, type)* bestimmt die Steigung oder den Y-Achsenabschnitt einer Geraden. *type* ist entweder 'slope' oder 'Y' und bestimmt, was berechnet wird.
- *XY(p, type)* gibt entweder die X- oder die Y-Koordinate des Punktes p zurück. Auch hier wird mit *type* der Wert gewählt: *type* ist 'X' oder 'Y'.
- *Sum(v)* zur Summation der im Array v angegeben Values.
- *Div(a, b)* zum Bestimmen des Quotienten $\frac{a}{b}$.

5.4 Algorithmus

Zur Auswertung einer Konstruktion werden alle im Assessment hinterlegten Verifier durchlaufen und ausgewertet. Dies geschieht grundsätzlich in zwei Schritten:

1. `choose(f)` nimmt ein Fixture entgegen und generiert passend dazu neue Fixtures, die den Test des Verifiers bestehen.

2. `verify(f)` testet, ob das gegebenene Fixture die im Verifier definierte Bedingung erfüllt. Ist dies der Fall, wird das Fixture an die `choose()`-Methode des nächsten Verifiers übergeben. Gibt es keinen weiteren Verifier und wird die Konstruktion über die `verify()` Methode ausgewertet, dann meldet das Assessment einen Erfolg und gibt das gefundene Fixture zurück. Wird die Konstruktion mit der `collect()`-Methode ausgewertet oder erfüllt das Fixture den Verifier nicht, dann wird die Liste der Verifier so weit zurückgegangen, bis ein alternatives Fixture gefunden wird. Von dort aus arbeitet das Assessment wie beschrieben weiter. Gibt es kein weiteres Fixture, mit dem weitergearbeitet werden kann, ist die Auswertung der Konstruktion fehlgeschlagen.

Das folgende Beispiel soll diese beiden Schritte veranschaulichen.

Listing 5.3: ***Überprüft, ob es eine Gerade*** l ***durch die beiden Punkte*** X ***und*** Y ***gibt.***

```
new Assessment(
        new Verifier.Point('X'),
        new Verifier.Point('Y'),
        new Verifier.Line('l', 'X', 'Y')
)
```

Wir überergeben dem Assessment in Listing 5.3 die in Abbildung 5.2 gezeigte JSXGraph-Konstruktion.

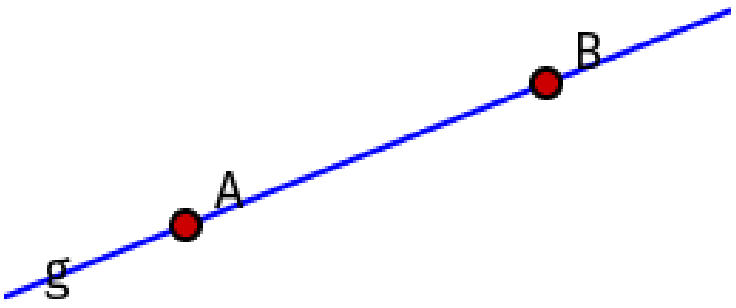


Abbildung 5.2: *Eine Beispielkonstruktion, die mit dem Assessment in Listing 5.3 ausgewertet wird.*

Zuerst generiert das Assessment ein leeres Fixture, das an die Methode `choose()` des ersten Point-Verifiers übergeben wird. Das Ergebnis dieses Aufrufs ist in Tabelle 5.2 zu sehen.

Fixture	X	Y	l
#1	A	-	-
#2	B	-	-

Tabelle 5.2: *Alle Fixtures, die die* `choose()` *Methode des Point('X')-Verifiers in Listing 5.3 nach Analyse der Konstruktion in Abbildung 5.2 liefert.*

Fixture	X	Y	l
#1	A	A	-
#2	A	B	-

Tabelle 5.3: *Alle Fixtures, die die* `choose()`*-Methode des Point('Y')-Verifiers in Listing 5.3 nach Analyse der Konstruktion in Abbildung 5.2 unter Berücksichtigung der Fixtures in Tabelle 5.2 liefert.*

Von den beiden Fixtures, die wir dadurch erhalten, wird das erste an die `choose()`-Methode des zweiten Point-Verifiers übergeben. Wir erhalten die Fixtures in Tabelle 5.3.

Von diesen beiden Fixtures wird wiederum das erste an die `choose()`-Methode des Line-Verifiers übergeben. Allerdings scheitert dies, da dieser keine Gerade finden kann, die allein durch den Punkt A definiert ist. Das Assessment geht nun in der Liste der Verifier so weit zurück, bis ein alternatives Fixture gefunden werden kann. In diesem Beispiel ist das Fixture #2 in Tabelle 5.3. Übergeben wir dieses Fixture an den Line-Verifier, erhalten wir das Fixture in Tabelle 5.4.

Wird die Konstruktion mit der Methode `verify()` der Assessment-Klasse ausgewertet, sind wir hier fertig und das Verfahren wird mit Erfolg beendet. Das Ergebnis ist das Fixture in Tabelle 5.4.

Wenn die Konstruktion mit der `collect()`-Methode analysiert wird, geht das Assessment in der Liste der Verifier wieder zurück, bis es das erste alternative Fixture findet. In diesem Fall ist es Fixture #2 in Tabelle 5.2. Analog zu

Fixture	X	Y	l
#1	A	B	g

Tabelle 5.4: *Alle Fixtures, die die* `choose()` *Methode des Line-Verifiers in Listing 5.3 nach Analyse der Konstruktion in Abbildung 5.2 unter Berücksichtigung der Fixtures in Tabelle 5.3 liefert.*

Fixture #1 liefert der Point('Y')-Verifier ein Fixture, das Y auf A abbildet. Der Line-Verifier akzeptiert dies und erweitert das Fixture um die Abbildung von l auf g. `collect()` geht auch jetzt wieder zurück, um alternative Fixtures zu untersuchen. Der zweite Point-Verifier liefert eines, das jedoch den Test des Line-Verifiers nicht besteht.

Die beiden erfolgreichen Fixtures bilden das Ergebnis dieses Aufrufs.

Kapitel 6

Sketch-Erkennung

6.1 Einführung

6.1.1 Tablets

Mit *Personal Digital Assistants* (PDA) erschienen die ersten Tablets bereits in den 1990er Jahren. Mit der Vorstellung des Apple iPad 2010 wurde diese bis dato weniger bekannte Geräteklasse einem großen Kreis von Nutzern zugänglich gemacht. Ihre physikalischen Eigenschaften machen Tablets zu geeigneten Werkzeugen für den Unterricht. Dies wurde bereits früh erkannt, in [10] finden sich beispielsweise folgende Punkte:

- Standortunabhängigkeit und schnelle Verfügbarkeit am Schülerarbeitsplatz.
- Arbeiten in gewohnter Umgebung im Klassenzimmer anstelle eines Computerraums ist möglich, klassische Unterrichtsmaterialien wie Bücher und Hefte werden nicht verdrängt.
- Die Kommunikation der Schülern untereinander und mit der Lehrkraft ist durch Tablets weniger eingeschränkt als in vielen Computerräumen durch große Monitore und manchmal auch PCs auf den Tischen.

- Im Vergleich zu (graphischen) Taschenrechnern können Tablets in allen Fächern Verwendung finden, nicht nur im naturwissenschaftlichen Bereich.

Allerdings gibt es auch neue Herausforderungen an die Hersteller von Software für den Unterricht. Der Einsatz von Programmen, die bisher an Desktop Computern und Notebooks genutzt wurden, scheitert auf diesen neuen Geräten meist an technischen Hindernissen, wie einer fehlenden Java Runtime Environment. Des Weiteren stellen Tablets durch die Bedienung mit dem Finger andere Anforderungen an das User Interface: Klassische Menüs sind mit dem Finger nur schwer bedienbar. Buttons müssen groß genug sein, um sie eindeutig mit dem Finger treffen zu können.

6.1.2 Dynamische Geometriesysteme und Tablets

Zu den ersten Versuchen, Geometrie auf Tablets zu betreiben, gehörte bereits Anfang der 2000er Jahre GEONExT auf einem Siemens SIMpad SL4 [10]. GEONExT wurde hier nicht direkt auf den Tablets ausgeführt, sondern auf einem Terminalserver. Auf den SIMpads wurde GEONExT lediglich in einer Terminal-Client-Sitzung angezeigt.

Eine der charakterisierenden Eigenschaften von dynamischen Geometriesystemen ist die Interaktivität. Konstruktionen sind nicht einfach statische Zeichnungen, sie lassen sich verändern und sämtliche geometrischen Nebenbedingungen bleiben erhalten. Diese Interaktivität war ursprünglich beschränkt auf freie Punkte. Diese waren die einzigen Elemente, die der Benutzer direkt auf graphischer Ebene beeinflussen konnte. Alle übrigen Elemente, Geraden, Kreise und abhängige Punkte, konnten nur indirekt über das Ziehen von freien Punkten beeinflusst werden.

Zumindest in GeoGebra, C.a.R. und Geometer's SketchPad wurde dies ein Stück weit aufgeweicht. Nutzer haben die Möglichkeit zur Translation von Geraden und Kreisen durch Ziehen. Geometer's SketchPad erlaubt auch die Rotation von Geraden und das Strecken von Kreisen durch spezielle Zug-Werkzeuge, allerdings sind das Rotations- und Streckzentrum offenbar vordefiniert. JSXGraph erweitert dieses Verhalten auf Multitouch fähigen Geräten durch das Ziehen von Geraden und Kreisen mit zwei Eingabepunkten. Hierdurch kann eine (freie) Gerade rotiert und ein (freier) Kreis gestreckt werden

und so genau wie freie Punkte in einem Zug in die gewünschte Position und Größe gebracht werden.

Viel mehr Möglichkeiten bietet die Toucheingabe jedoch für die Erstellung von Konstruktionen. Anstatt auf klassische Konstruktionsmenüs und Werkzeugleisten zu klicken, kann der Benutzer die Konstruktion zeichnen. Dies erlaubt eine natürlichere und damit einfacher erlernbare Eingabe einer Konstruktion. Mit ein wenig Übung können Konstruktionen zudem wesentlich schneller erzeugt werden.

Die Eingabe von freien Punkten, Geraden und Kreisen ist relativ einfach zu realisieren. Der Nutzer zeichnet diese Elemente auf der Zeichenfläche. Ein Programm erkennt die Eingabe und ermittelt mittels Regression eine exakte Repräsentation des Elements, die der Eingabe am nächsten kommt.

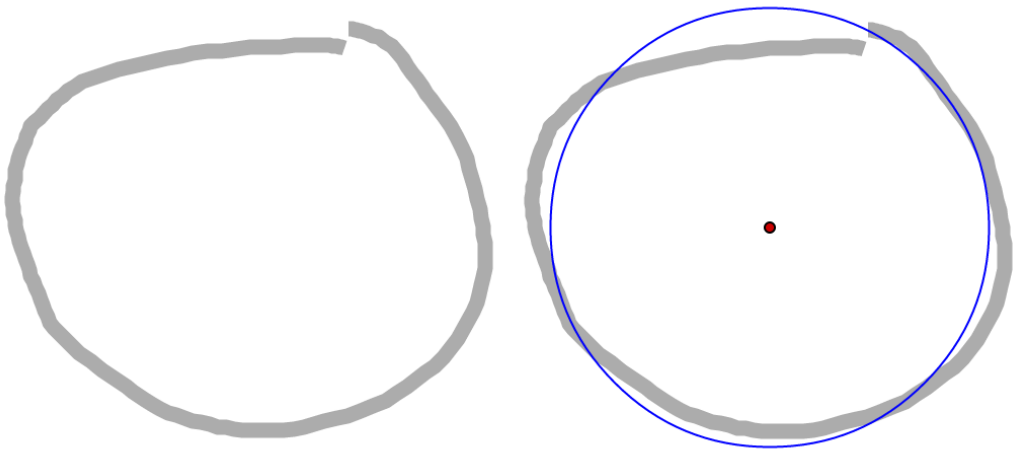


Abbildung 6.1: *Die Zeichnung eines Kreises links und die Interpretation dieser Zeichnung durch die Sketch-Erkennung rechts.*

Schnittpunkte und Gleiter sind ebenfalls unproblematisch: Befindet sich an der Stelle, an die der Benutzer getippt hat, nur ein einzelnes Element, das kein Punkt ist, wird ein Gleiter erzeugt. Befinden sich dort zwei Elemente, wird ein Schnittpunkt angelegt.

Bei allen weiteren abhängigen Elementen besteht die Schwierigkeit darin, deren Abhängigkeiten anzugeben. Beispielsweise genügt es nicht, bei einer Senkrechten oder einer parallelen Geraden, diese lediglich einzuzeichnen. Allein durch die Zeichnung kann nicht entschieden werden, ob es sich dabei um

eine abhängige oder eine freie Gerade handelt. Dies würde für eine statische Zeichnung oftmals ausreichen, aber nicht für eine interaktive, in der diese geometrischen Beziehungen bei einer Änderung der Konstruktion erhalten bleiben müssen.

6.1.3 Nomenklatur

Wir unterscheiden zwischen *Gesten* und *Sketches*. Die Eingabe ist identisch – in beiden Fällen zeichnet der Nutzer mit dem Eingabegerät eine oder mehrere 2D Kurven. Der Unterschied liegt in der Interpretation der Zeichnung.

Gesten finden häufig bei der Steuerung von Programmen Verwendung, beispielsweise zur Navigation in einem Browser oder auf einem Tablet. Zeichnet man einen nach links zeigenden Pfeil, so geht der Browser eine Seite zurück. Die Programmverwaltung eines iPads wird aufgerufen, indem der Benutzer mit gleichzeitig mindestens vier Fingern von unten nach oben über den Bildschirm fährt.

Sketches hingegen werden oft in Zeichenprogrammen verwendet. Sie dienen weniger der Programmsteuerung, sondern der direkten Eingabe einer aufbereiteten Repräsentation ihrer selbst. Im Fall des in Abbildung 6.1 gezeichneten Kreises beschreiben die Eingabe (Kreiszeichnung) und das Ergebnis (Kreis-Element) die gleiche mathematische Struktur.

Die Eingabe von Sketches und Gesten erfolgt über das Zeichnen sogenannter *Strokes*. Ein Stroke wird auf Touchgeräten durch das Berühren des Touchbildschirms und der anschließenden Bewegung des Fingers über den Bildschirm gezeichnet. Auf Geräten mit Maussteuerung wird ein Stroke durch das Drücken der linken Maustaste und der anschließenden Bewegung der Maus über die Tischfläche gezeichnet. Während der Bewegung des Fingers beziehungsweise der Maus wird in regelmäßigen Abständen die aktuelle Position des Eingabegeräts in einem *Datenpunkt* abgelegt. Durch das Loslassen des Fingers oder der Maustaste wird die Eingabe eines Strokes abgeschlossen. Das Ergebnis ist die Menge der eingesammelten, geordneten Datenpunkte, die zusammen einen *Streckenzug* definieren.

Ein Sketch kann auch aus mehreren Strokes bestehen, die gleichzeitig oder nacheinander gezeichnet werden. Man spricht bei einem Sketch, der aus mehreren Strokes besteht, auch von einem *Multi Stroke*. Analog besteht ein *Single Stroke* aus nur einem Stroke.

6.1.4 Ziele

In diesem Abschnitt werden wichtige Aspekte und Ziele eines idealen Sketch-basierten Geometriesystems beschrieben. Derzeit sind in sketchometry nicht alle diese Ziele realisiert. An den entsprechenden Stellen wird auf die Gründe hierfür hingewiesen.

Unterstützung unterschiedlicher Plattformen

Es ist wichtig, möglichst viele Plattformen zu unterstützen. Während Tablets die anvisierte Hauptplattform sind, sollte die Sketch-Erkennung auf möglichst vielen Plattformen lauffähig sein. Diese umfasst auf Seiten der Hardware unter anderem Desktop-PCs, Tablets, interaktive Tische und Whiteboards, Notebooks, tragbare Multimediageräte, Smartphones und Spielekonsolen. Damit verbunden sind deren unterschiedliche Eingabegeräte wie Maus, Digitizer, kapazitive (Multi-)Touch-Displays, induktive Displays mit Eingabestift, Touchpad, Trackpad, Bewegungssensoren und Webcams. Im Bezug auf Software gibt es neben den fünf bekanntesten Betriebssystemen Microsoft Windows, Apple OS X, Linux basierte Systeme sowie Android und iOS noch unzählige andere, unter anderem Firefox OS, WebOS und diverse BSD Derivate.

Für den Unterricht relevant sind Desktop-PCs, Tablets oder interaktive Whiteboards die sowohl mit Touch- beziehungsweise Stifteingabe als auch Eingabe per Maus bedienbar sind. An Betriebssystemen sind in Klassenzimmern meist Windows, OS X, Linux, Android und iOS anzutreffen. Auf fast allen genannten Systemen sind moderne Webbrowser wie Mozilla Firefox und Chromium verfügbar. Damit ist eine effiziente JavaScript-Umgebung zur Programmierung und HTML, CSS und SVG zur graphischen Darstellung gewährleistet. Mit JSXGraph existiert außerdem ein dynamisches Geometriesystem in JavaScript, auf dem problemlos aufgebaut werden kann, sowohl technologisch als auch lizenzrechtlich[1].

Die unterschiedlichen Eingabemethoden, Systeme und Browser schlagen sich unter anderem in der Anzahl von Datenpunkten nieder, die pro Sketch zur Verfügung stehen. Sind es auf einem iPad 2 unter iOS 7 nur etwa 40 Da-

[1] JSXGraph ist verfügbar unter der LGPL und MIT Lizenz (siehe https://github.com/jsxgraph/jsxgraph/blob/master/COPYRIGHT, März 2014)

tenpunkte pro Sekunde (points per second, p/s) liefert eine Maus auf einem Desktop-PC mit Vierkernprozessor, Linux und Google Chrome 29 etwa $70 - 100p/s$. Auf dem gleichen Rechner liefert der Digitizer *Wacom Bamboo* über $120p/s$. Eine Sketch-Erkennung muss mit dieser Bandbreite an Auflösungen zurechtkommen.

Bei der Eingabe ist auch die Einschränkung der insgesamt möglichen Eingabeformen auf einigen Geräten zu beachten. Es gibt berührungsempfindliche Geräte, die nicht mehrere Berührungen simultan verarbeiten können. Diese sind beschränkt auf

- kurze Berührung,
- lange Berührung mit einer vordefinierten Wartezeit und
- Bewegung.

Diese müssen nicht auf jedem Gerät in genau dieser Form angewandt werden. So ist es durchaus üblich, eine Aktion, die auf Tablets durch eine lange Berührung ausgelöst wird, auf Desktop-PCs mit Mauseingabe über einen Rechtsklick anzusteuern.

Im weiteren Verlauf wird auf verschiedene Eingabegeräte mit unterschiedlichen Eingabeformen Bezug genommen. Daher sollen künftig, sofern nichts anderes vermerkt, die Begriffe *klicken*, *drücken* und *berühren* als synonym betrachtet werden.

Anforderungen an das User Interface

Die Eingabe sollte so natürlich wie möglich gestaltet werden. Hierfür eignen sich für ein dynamisches Geometriesystem Sketches besonders gut, da hier das Ergebnis weitestgehend der Eingabe entspricht (siehe Abbildung 6.1). Im Idealfall könnte der Nutzer die Konstruktion wie mit Bleistift auf Papier zeichnen. Jedoch kann durch einen Sketch ausschließlich die Form und die Lage eines Elements angegeben werden. Geometrische Abhängigkeiten, die für dynamische Geometriesysteme von elementarer Bedeutung sind, fehlen vollständig. Ein interaktives Verändern der Konstruktionen wäre damit nicht

möglich, ohne jeglichen geometrischen Zusammenhang unter den Elementen der Konstruktion zu verlieren.

Es ist wichtig, dem Benutzer nach Beendigung des Zeichenvorgangs darzustellen, was konstruiert wurde. Diese Darstellung muss mindestens die folgenden Informationen besitzen: Die genaue Bezeichnung des konstruierten Elements (zum Beispiel Gerade, Normale, Dreieck, etc.), sowie sämtliche zur Konstruktion herangezogenen Elternelemente (zum Beispiel die beiden definierenden Punkte einer Geraden oder die Gerade und den Punkt von denen die konstruierte Parallele abhängt etc). Der Benutzer erhält diese Informationen zur Bestätigung der aktuellen Konstruktion. Fehlt diese Bestätigung, kann es vorkommen, dass visuell ähnliche Elemente konstruiert werden, beispielsweise eine senkrecht liegende, freie Gerade anstelle einer Normalen durch einen Punkt. Der Benutzer merkt dies unter Umständen erst wesentlich später, wodurch ein Teil der Konstruktion auf falschen Ausgangselementen basiert.

Eine bereits während des Konstruierens ständig aktualisierte Anzeige erleichtert die Bedienung, da der Nutzer sofort sieht, wann er den Zeichenvorgang beenden kann. Insbesondere wenn Sketches sich sehr ähnlich sind, kann dies für den Nutzer hilfreich sein (siehe Abbildung 6.2). Dies erfordert jedoch eine hinreichend schnelle Implementierung oder hinreichend gute Heuristiken für die Aktualisierung während des Konstruierens.

Diese ständig aktualisierte Anzeige ist derzeit in sketchometry implementiert, allerdings beruht die dargestellte Information auf einem Teilergebnis der verwendeten Sketch-Erkennung. Es kommt daher in seltenen Fällen vor, dass ein anderes Element erzeugt wird, als angezeigt wird.

Um die Geschwindigkeit einer schnellen Sketch-Erkennung stabil zu halten, sollte die Eingabe nur aus dem Sketch selbst und den unmittelbar beteiligten, bereits konstruierten Elementen bestehen. Dies hält die Laufzeit der Erkennung hinreichend klein. Das Analysieren der gesamten Konstruktion zur Ermittlung von Abhängigkeiten verlängert die Laufzeiten mit steigender Komplexität der Konstruktion.

Da während der Zeichnung nur diskrete Datenpunkte gesammelt werden, kann der Abstand zwischen zwei Datenpunkten durchaus mehrere Pixel betragen. Deshalb genügt es nicht, die Elemente zu ermitteln, die in der Nähe konkreter Datenpunkte liegen, um festzustellen, welche Elemente an der Konstruktion beteiligt sein könnten. Stattdessen muss geprüft werden, welche Elemente in der Nähe des Segments zwischen je zwei aufeinanderfolgenden

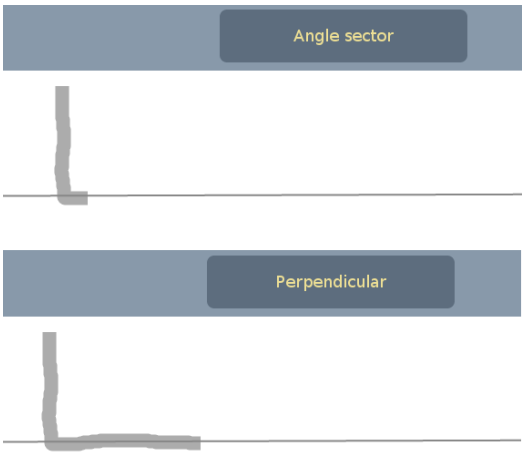


Abbildung 6.2: *Ein Sketch zur Konstruktion einer Normalen wird erst korrekt erkannt, wenn die existierende Gerade lange genug "überfahren" wurde.*

Datenpunkten liegen. Dies kann für Elemente wie Kurven sehr aufwändig werden. Daher wird diese Prüfung in sketchometry auch nur für Punktelemente durchgeführt.

Für eine natürliche Eingabe sollten weiterhin typisch technische Eingabeformen und Einschränkungen auf das Nötigste reduziert werden, zum Beispiel durch eine Reduktion der Anzahl der Klicks auf Buttons und Elemente. Dazu zählt auch das Umschalten zwischen verschiedenen Modi wie Konstruktions- und Zugmodus. Ein kombinierter Modus, in dem es möglich ist neue Elemente zu konstruieren und vorhandene Elemente zu ziehen, eliminiert die Notwendigkeit eines Umschaltmechanismus.

Ein kombinierter Zug- und Konstruktionsmodus hat weiterhin den Vorteil, dass mehrere Nutzer gleichzeitig an einer Konstruktion arbeiten können. Während ein Nutzer einen Punkt zieht, kann ein anderer auf der gleichen Zeichenfläche einen Kreis konstruieren. Da dies auf den kleinen Bildschirmen derzeit verfügbarer Tablets und Smartphones keinen Sinn macht und ein PC mit Maus keine simultane Eingabe erlaubt, ist ein solcher Kollaborationsmodus derzeit nicht in sketchometry implementiert.

6.1.5 Vorarbeiten zur Sketch-Erkennung

Erste Schritte im Bereich der Sketch-Eingabe und -Erkennung in dynamischen Geometriesystemen unternahmen Ulrich Kortenkamp und Dirk Materlik [32] in Form eines *Cinderella*-Moduls namens *Scribbling*. Sie beschreiben drei Lösungsansätze:

Nachträgliches Markieren

Abhängigkeiten werden durch nachträgliches Markieren definiert. Beispielsweise wird eine Normale durch das Skizzieren des Symbols eines rechten Winkels am Schnittpunkt zweier Geraden definiert. Die Parallelität einer Geraden lässt sich durch zwei Markierungen auf dieser und ihrer Parallelen verdeutlichen. Dies ist jedoch nicht unproblematisch. Das Einfügen von Nebenbedingungen in eine Konstruktion ist zu einem späteren Zeitpunkt oft nicht möglich, weil sie im Widerspruch zu bereits existierenden Nebenbedingungen stehen. Ist das Einfügen der Nebenbedingung widerspruchsfrei möglich, so bliebe noch zu klären, welches Element gebunden werden soll und welches frei bleibt. Des Weiteren gibt es beispielsweise ein Unterscheidungsproblem: wie können Notationsskizzen von Konstruktionsskizzen unterschieden werden, wenn der Nutzer nach der Konstruktion zweier sich schneidender Geraden deren Schnittwinkel einzeichnet?

Die in [32] beschriebene Lösung, diese Markierung auf das zuletzt gezeichnete Element zu beschränken, versucht damit einige der genannten Probleme zu beheben. Zur Lösung des Unterscheidungsproblems ist ein Signal an das Programm nötig, das die Notationsphase beendet. Dies kann eine Geste sein, ein Button-Klick oder ein simpler Timer. Jedoch führen alle diese Lösungen zu unnatürlichen Einschränkungen des Nutzers, deren Notwendigkeit zudem schwer zu vermitteln ist.

Automatisches Erkennen

Nachdem der Nutzer das Skizzieren des Punktes, der Geraden oder des Kreises abgeschlossen hat, ließe sich automatisch erkennen, ob das gezeichnete Element bestimmte geometrische Eigenschaften annähernd erfüllt, beispielsweise ob eine Gerade nahezu senkrecht zu einer bereits existierenden Geraden

gezeichnet wurde. Ist dies der Fall, wird eine Normale anstelle einer freien Geraden erzeugt. Dies entspricht der Arbeitsweise der Konstruktion mit Papier, Bleistift, Zirkel und Lineal und wäre damit der natürlichste und intuitivste Weg der Konstruktion. Jedoch steigt die Anzahl der Möglichkeiten ein neues Element zu interpretieren mit der Anzahl der konstruierbaren Elemente sowie der Zahl bereits konstruierter Elemente. Damit erhöht sich auch die Zahl der Fehlinterpretationen.

In einem im Rahmen des sketchometry-Projekts erstellten Prototypen wurde dieses Problem der Fehlinterpretationen relativiert, indem der Benutzer nach dem Zeichnen des Elements aus einer Liste der erkannten geometrischen Nebenbedingungen die Gewünschte auswählt. Allerdings erreichte diese Liste in Tests schnell eine unpraktikable Länge. Des Weiteren stellte sich heraus, dass es sehr schwer ist, bestimmte Elemente auf Anhieb korrekt zu platzieren. Bei Mittelpunkten und Senkrechten ist es leicht, sie korrekt einzuzeichnen. Bei Parallelen und gespiegelten Punkten ist es ungleich schwerer, die Elemente richtig zu platzieren.

Deshalb kann bei dieser Methode auch nur mit einem gewissen Akzeptanzbereich gearbeitet werden. Bei einem Mittelpunkt einer Strecke beispielsweise kann man sich einen Kreis vorstellen, mit geeignetem Radius und dem Zentrum im Mittelpunkt der Strecke. Solange der Nutzer einen Punkt innerhalb dieser Kreisfläche platziert, wird er als Mittelpunkt erkannt. Den Radius dieses Kreises zu bestimmen, ist schwer: Ist er zu klein, müssen Nutzer den Mittelpunkt äußerst exakt treffen, was nicht immer gelingt. Ist der Radius zu groß gewählt, werden die Auswahllisten zu lang.

Aus diesen Gründen wurde der Prototyp wieder verworfen.

Vorauswahl

Im Gegensatz zu einer Auswahl der Abhängigkeiten nach dem Zeichnen werden bei der Vorauswahl die Abhängigkeiten durch Antippen vor dem Zeichnen des eigentlichen Sketches ausgewählt. Zum Erzeugen einer Normalen wird erst die Gerade, zu der die Normale senkrecht sein wird, ausgewählt und anschließend eine Gerade senkrecht dazu gezeichnet. Vor dem Zeichnen eines Mittelpunkts werden die beiden Elternpunkte ausgewählt.

Aus technischer Sicht löst dieser Ansatz einige Probleme. Allerdings ist er nicht geeignet, wenn mehrere Personen gleichzeitig an einer Konstruktion arbeiten wollen, beispielsweise über ein multitouch-fähiges interaktives Whiteboard oder Tisch. Des Weiteren ist das Antippen der Elternelemente eine technische Lösung, die dem Ziel der möglichst natürlichen Eingabe entgegensteht.

6.2 Erkennungs-Algorithmus

In sketchometry wurde daher ein neuer Ansatz implementiert, bei dem die Abhängigkeiten nicht vor oder nach dem Zeichnen definiert werden sondern in die Zeichnung integriert sind. Diese Vorgehensweise reduziert die Konstruktion eines neuen Elements auf eine einzelne Aktion.

Die nächsten Abschnitte beschreiben die einzelnen Schritte der Sketch-Erkennung:

- Klassifikation der gezeichneten Kurve
- Zerlegen des Sketches in geometrische Elemente
- Finden der Abhängigkeiten
- Erzeugen des Elements

6.2.1 Klassifikation der gezeichneten Kurve

Der erste Schritt dient der groben Einordnung der Zeichnung in eine der Kategorien *angle, bisector, circle, circle2points, line, midpoint, normal, parallel, quadrilateral, reflection, tangent* oder *triangle*. Diese Kategorien entsprechen den verschiedenen Sketches (siehe Anhang B). Zur Ermittlung der Kategorie wird auf $N-Protractor [3] zurückgegriffen, eine bereits existierende und erprobte Lösung.

$N-Protractor

Zu jeder Kategorie gibt es mehrere vordefinierte Zeichnungen, sogenannte *Vorlagen*, die festlegen, wie Sketches dieser Kategorie aussehen. Die in sketchometry verwendeten Vorlagen für die Kategorie bisector sind in Abbildung 6.3 zu sehen.

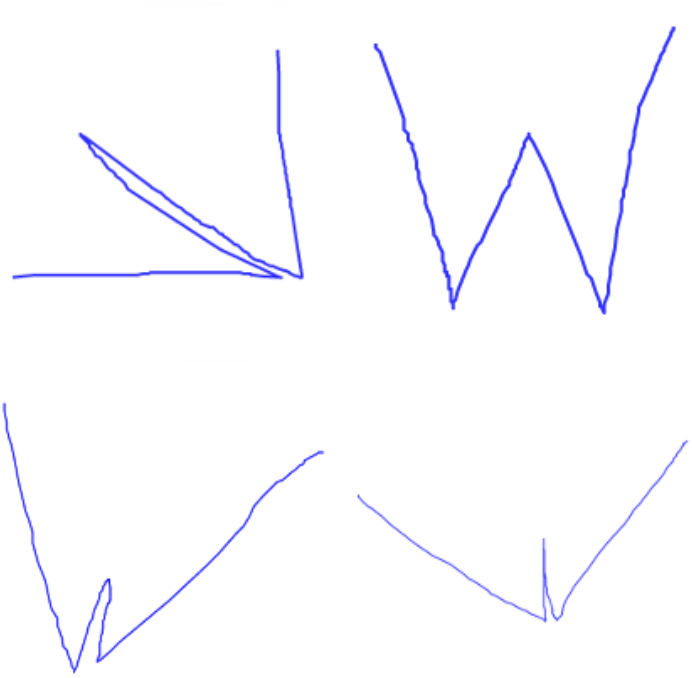


Abbildung 6.3: *Die vier Vorlagen der Kategorie bisector*

Zur Erkennung der Zeichnung g eines Nutzers vergleicht $N-Protractor diese mit den *Vorlagen* aller Kategorien. Bei jedem Vergleich wird ein Wert berechnet. Als Ergebnis wird eine Liste der Kategorien mit dem besten Wert aus jeder Kategorie zurückgegeben.

Um zwei Zeichnungen vergleichen zu können, müssen diese zuerst normalisiert werden. Dies geschieht in drei Schritten:

- Berechnen aller Single Strokes, die sich aus einem Multi Stroke ergeben,
- Umwandeln des Sketches in einen Streckenzug P von N äquidistanten Punkten (*Resampling*) und

- Normalisieren des Streckenzugs P durch Translation und Rotation.

Der erste Schritt generiert aus einem Multi Stroke alle Single Strokes, die durch Permutation der Reihenfolge aller enthaltenen Strokes entstehen können. Da in sketchometry ausschließlich Single Strokes zum Einsatz kommen, ist die Ausgabe in diesem Fall identisch zur Eingabe. Dieser Schritt kann daher auch weggelassen werden.

Wir beginnen in sketchometry daher direkt mit dem Resampling der Zeichnung und erhalten den Streckenzug P. In sketchometry hat sich $N = 96$ bewährt.

Anschließend wird der Streckenzug so verschoben, dass der geometrische Schwerpunkt im Ursprung liegt. Als *indicative angle* bezeichnet Li in [22] den Winkel zwischen der positiven x-Achse und der Verbindungsgeraden des geometrischen Schwerpunkts mit dem Anfangspunkt des verschobenen Streckenzugs. Zur Normalisierung der Orientierung wird der Streckenzug so rotiert, dass der indicative angle gleich 0 ist.

Betrachten wir die Koordinaten der Datenpunkte des transformierten Streckenzugs $(x_1, y_1, x_2, y_2, \ldots, x_N, y_N)$, erhalten wir einen Vektor $v_g \in \mathbb{R}^{2N}$. Bezeichnen wir die Menge der Vorlagen mit T, so erhalten wir für jede Vorlage $t \in T$ über die geschilderte Normalisierung einen Vektor $v_t \in \mathbb{R}^{2N}$. Daraus lässt sich über die Bewertungsfunktion

$$S(t, g) = \arccos \frac{\langle v_t, v_g \rangle}{\|v_t\| \|v_g\|}.$$

die Zeichnung g des Nutzers mit den Vorlagen $t \in T$ über den Winkel zwischen den beiden Vektoren v_g und v_t vergleichen.

Da der indicative angle nur den ersten Datenpunkt berücksichtigt, stellt dieser Winkel nur einen groben Anhaltspunkt für die Orientierung der Zeichnung dar. Deshalb ist es notwendig, vor der Abstandsberechnung eine Korrektur vorzunehmen und die Vorlage t um einen Winkel

$$\begin{aligned}\theta_{opt} &= \underset{-\frac{\pi}{2}<\theta<\frac{\pi}{2}}{\arg\min}\left(arccos\frac{\langle v_t(\theta), v_g\rangle}{\|v_t(\theta)\|\|v_g\|}\right) \\ &= \underset{-\pi<\theta<\pi}{\arg\max}\left(\frac{\langle v_t(\theta), v_g\rangle}{\|v_t(\theta)\|\|v_g\|}\right)\end{aligned} \tag{6.1}$$

zu drehen, der die Ähnlichkeit maximiert. Die zweite Identität in Gleichung 6.1 folgt aus der Monotonie des Arkuskosinus. Vor der Berechnung der Ableitung sehen wir uns $v_t(\theta)$ genauer an. Da dieser Vektor auch als Menge von Punkten im $\mathbb{R}^2$ aufgefasst werden kann, ist

$$v_t(\theta) = Rv_t,$$

mit der $2N \times 2N$ Matrix

$$R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 & \cdots & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos\theta & -\sin\theta & \cdots & 0 & 0 \\ 0 & 0 & \sin\theta & \cos\theta & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos\theta & -\sin\theta \\ 0 & 0 & 0 & 0 & \cdots & \sin\theta & \cos\theta \end{pmatrix}.$$

Daraus ergibt sich für das Skalarprodukt aus Gleichung 6.1 durch ausmultiplizieren

$$\langle Rv_t, v_g\rangle = \sum_{i=1}^{N}\left[x_{gi}(-\sin(\theta)x_{ti} - \cos(\theta)y_{ti}) + y_{gi}(\cos(\theta)x_{ti} - \sin(\theta)y_{ti})\right].$$

Zur Berechnung von θ_{opt} setzen wir zur Berechnung der Extrema

$$\frac{\mathrm{d}}{\mathrm{d}\,\theta}\frac{\langle v_t(\theta), v_g\rangle}{\|v_t(\theta)\|\|v_g\|} = 0.$$

Weil die Norm rotationsinvariant ist, können wir den Nenner in der weiteren Betrachtung ignorieren und erhalten durch Ausmultiplizieren und Ordnen der Terme

$$\cos(\theta) \cdot \sum_{i=1}^{N}(x_{ti}y_{gi} - x_{gi}y_{ti}) - \sin(\theta) \cdot \sum_{i=1}^{N}(x_{gi}x_{ti} + y_{gi}y_{ti}) = 0.$$

Die zweite Summe ist gleich dem Skalarprodukt $\langle v_t, v_g \rangle$. Ist dieses gleich 0, dann liegt das Extremum bei $\frac{\pi}{2}$. Ist das Skalarprodukt ungleich 0, dann ist auch $\cos\theta$ ungleich 0 und wir erhalten

$$\tan\theta = \frac{\sin\theta}{\cos\theta} = \frac{\sum_{i=1}^{N} x_{ti}y_{gi} - x_{gi}y_{ti}}{\langle v_t, v_g \rangle}.$$

Da die Extrema nur zwischen $-\frac{\pi}{2}$ und $\frac{\pi}{2}$ gesucht werden (ECMAScript Implementierungen von arctan liefern keine anderen Werte) und unsere Vektoren normiert sind, erhalten wir genau ein Extremum, das entweder ein Maximum oder ein Minimum ist. Im Fall eines Maximums landet die Vorlage auf dem letzten Platz der Bewertungsliste der Vorlagenklasse. Wir berechnen daher

$$\theta_{opt} = \arctan\left(\frac{\sum_{i=1}^{N} x_{ti}y_{gi} - x_{gi}y_{ti}}{\langle v_t, v_g \rangle}\right).$$

Nach einer Rotation der Vorlage t um θ_{opt} erhalten wir als Bewertung

$$S(t, g) = \arccos\left(\frac{\langle v_t(\theta_{opt}), v_g \rangle}{\|v_t(\theta_{opt})\|\|v_g\|}\right).$$

Wahl von $N-Protractor

Neben $N-Protractor gibt es von den gleichen Autoren weitere, ähnliche Algorithmen: $1 [35], $N [2] sowie $P [33]. $1 ist die single stroke Variante von $N. Letzterer erweitert $1 um die gleiche kombinatorische Komponente, die auch in $N-Protractor für die Umwandlung von Multi Strokes in mehrere

Single Strokes sorgt (siehe vorherigen Abschnitt). $N nutzt $1 zur Analyse der errechneten Single Strokes.

Im Gegensatz zu $N-Protractor findet bei $1 ein iterativer Ansatz Verwendung, der in $N-Protractor von einer geschlossenen Form abgelöst wird. Diese geschlossene Form bringt wesentliche Geschwindigkeitsverbesserungen bei gleichbleibender Erkennungsrate [22]. $P umgeht die Single/Multi Stroke Problematik, indem der gesamte Sketch zu einer Punktewolke zusammengefasst wird.

Die Wahl fiel auf $N-Protractor, da es die beste Laufzeit in diesem Bereich bietet [33]. Außerdem halten wir uns die Möglichkeit offen, Multi Stroke Eingaben in sketchometry zu integrieren. $P ist nicht rotationsinvariant und wird deshalb nicht in sketchometry eingesetzt.

6.2.2 Geometrische Analyse

Zur geometrischen Analyse wird der Sketch zunächst in Strecken und Kreisbögen zerlegt. Der gezeichnete Streckenzug wird hierfür in Teilstreckenzüge zerlegt. Jeder Teilstreckenzug repräsentiert entweder eine Strecke oder einen Kreisbogen. Zwei Teilstreckenzüge werden durch eine Ecke getrennt. Abbildung 6.4 verdeutlicht diese Unterteilung anhand eines Beispiels.

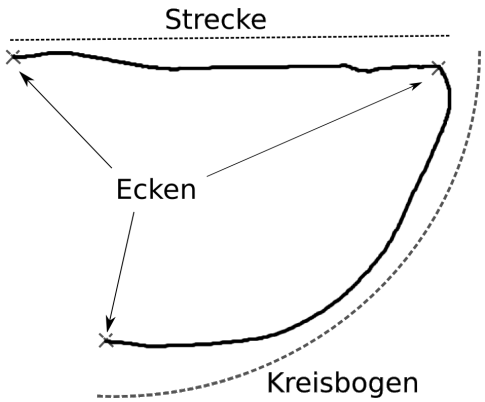


Abbildung 6.4: *Unterteilung des Sketches in einen Strecken-Teilstreckenzug und einen Kreisbogen-Teilstreckenzug.*

Wir bestimmen zunächst die Ecken aus der Menge der Datenpunkte des Sketches. Anschließend wird für jeden Teilstreckenzug zwischen zwei Ecken

entschieden, ob es sich hierbei um eine Strecke oder einen Kreisbogen handelt. Dieser Schritt erfolgt vollkommen unabhängig von der ersten Klassifizierung durch $N-Protractor.

Daraus lässt sich aus dem Sketch eine Konstruktion bestehend aus Punkten, Strecken und Kreisbögen ableiten. Zusammen mit den während der Zeichnung "überfahrenen" Elementen bildet diese Sketch-Konstruktion die Eingabe zur Analyse der Zeichnung. Dies geschieht anhand des Algorithmus zur numerischen Verifikation von geometrischen Eigenschaften einer Konstruktion, den wir in Kapitel 5 kennengelernt haben. Das Ergebnis von $N-Protractor fließt hier in der Auswahl der Regelsätze ein, die von practice zur Analyse der Konstruktion herangezogen wird.

Ecken

Zur Bestimmung der Ecken verwenden wir *ShortStraw* [36]. ShortStraw ist zur Analyse von Streckenzügen entwickelt worden. Wenn der Sketch ganz oder teilweise aus Kreisbögen besteht, werden in diesen Bögen zu viele falsche Ecken (*false positives*) erkannt. Um dieses und einige weitere Probleme zu vermeiden verwenden wir daher zusätzlich Teile der ShortStraw-Erweiterung *IStraw* [37], die Kreisbögen erkennen kann.

In einem ersten Schritt wird, wie auch bei $N-Protractor, die Menge der gesammelten Datenpunkte aufbereitet. Hierfür bestimmen wir zunächst eine Konstante D. l bezeichnet Länge der Diagonale des kleinsten axial ausgerichteten Rechtecks, das die Zeichnung enthält. Dann ist

$$D = \frac{l}{40}.$$

Die Konstante wurde in [36] experimentell bestimmt.

Durchlaufen wir den Streckenzug der Zeichnung, können wir die euklidischen Distanzen zwischen je zwei Punkten bestimmen und aufsummieren. Sobald die Summe den Wert von D erreicht oder übersteigt, platzieren wir auf dem Teilsegment in geeignetem Abstand einen neuen Punkt und fügen diesen Punkt der Ergebnismenge R hinzu und beginnen von dieser Position erneut mit dem Aufsummieren der Längen der Teilstrecken, bis wir ans Ende des

Streckenzugs gelangen. In Abbildung 6.5 sind die ursprünglichen Datenpunkte (o) sowie die Punktemenge nach dem Resampling (x) veranschaulicht.

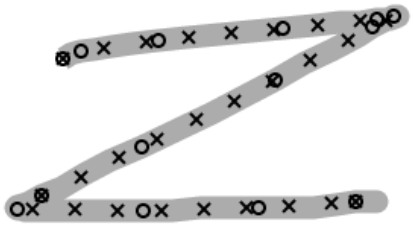


Abbildung 6.5: *Ein Vergleich der originalen Datenpunkte (o), die während der Konstruktion aufgezeichnet wurden und der resampleten Datenpunkte (x) nach IStraw.*

Jedem der resampleten Punkte $p_i \in R, 1 < i \leq N$ wird ein *Straw*-Wert $straw_i$ zugewiesen:

$$
\begin{aligned}
straw_1 &= 0, \\
straw_2 &= \frac{3}{2} \cdot \|p_5 - p_1\|, \\
straw_3 &= \frac{6}{5} \cdot \|p_6 - p_1\|, \\
&\vdots \\
straw_i &= \|p_{i+3} - p_{i-3}\|,\ 3 < i < N-2, \\
&\vdots \\
straw_{N-2} &= \frac{6}{5} \cdot \|p_N - p_{N-5}\|, \\
straw_{N-1} &= \frac{3}{2} \cdot \|p_N - p_{N-4}\|, \\
straw_N &= 0.
\end{aligned}
$$

Die ersten und die letzten drei Punkte erhalten spezielle Werte, da die allgemeine Formel für diese Punkte nicht definiert ist.

Anhand der Straw-Werte werden im nächsten Schritt aus der Menge R die Eckenkandidaten $c_i \in C \subset R$ bestimmt, indem die errechneten Straw-Werte

durchlaufen werden und sämtliche lokalen Minima in den Bereichen berechnet werden, die unter eine bestimmte Schwelle

$$t = 0,95 \cdot \text{mean}(straw_R)$$

fallen. $\text{mean}(straw_R)$ ist der Mittelwert über alle Straw-Werte. [36] verwendet den Median, Tests in [37] haben jedoch ergeben, dass der Mittelwert bessere Ergebnisse liefert. Auch der Wert $0,95$ wurde aus [36] übernommen.

Da Straw-Werte keine negativen Werte annehmen, stellt der Wert 0 für den ersten und den letzten Punkt sicher, dass diese beiden Punkte immer in die Menge der Eckenkandidaten aufgenommen werden.

Die folgenden drei Tests erweitern die Liste der Ecken oder filtern fälschlicherweise hinzugefügte Ecken aus der Kandidatenliste.

Streckentest

Der *Streckentest* berechnet den Quotienten aus dem euklidischen Abstand und der Pfadlänge

$$r = \frac{\|c_i - c_j\|}{\text{len}(c_i, c_j)}$$

zwischen zwei Ecken c_i und c_j. Die Pfadlänge $\text{len}(c_i, c_j)$ ist definiert als die Summe der Längen aller Strecken im Streckenzug der resampleten Punkte zwischen c_i und c_j. Liegt der Quotient höher als ein festgelegeter Schwellenwert, wird der gesamte Streckenzug als eine Strecke betrachtet. Liegt der Quotient darunter, ist die Pfadlänge wesentlich grösser als der euklidische Abstand. Das bedeutet, zwischen den beiden Ecken muss eine weitere Ecke liegen. Der Punkt mit dem niedrigsten Straw-Wert in einer Umgebung des Mittelpunkts zwischen beiden Ecken wird als weiterer Eckenkandidat der Menge R hingefügt.

Kollinearitätstest

Der *Kollinearitätstest* betrachtet jeweils drei Ecken c_i, c_j und c_k und führt den Streckentest an c_i und c_k durch. Ist der Test erfolgreich, so sind die drei

Ecken kollinear und c_j kann aus C entfernt werden. Um zu verhindern, dass anstelle von "richtigen" Ecken "falsche" behalten werden, wird der Kollinearitätstest zweimal durchlaufen. Beim ersten Durchlauf wird ein relativ hoher Schwellenwert von $1,0$ für den Streckentest gewählt. Im zweiten Durchlauf wird dieser auf $0,97$ gesenkt. Die beiden Werte wurden experimentell bestimmt.

Winkeltest

Der *Winkeltest* dient der Entfernung von "falschen" Ecken in Kreisbögen. Hierfür werden fünf Punkte betrachtet: Ein Eckenkandidat c und jeweils die beiden Punkte, die $shift = 15$ Stellen beziehungsweise $shift/3 = 5$ Stellen vor und nach der Ecke zu finden sind.

Dadurch erhalten wir zwei Winkel α und β. Diese sind definiert durch die Ecke und die beiden mit $shift$ beziehungsweise $shift/3$ gefundenen Punkte. Abbildung 6.6 verdeutlicht die Wahl von α und β.

Ist die Differenz $\beta - \alpha$ kleiner als der Schwellenwert

$$t_\alpha = 0.175 + 0.243/(\alpha + 0.611),$$

so handelt es sich bei der Ecke um eine korrekte Ecke. Ist die Differenz größer, so handelt es sich um einen Kreisbogen und die Ecke wird wieder aus C entfernt und das entsprechende Segment als Kreisbogen markiert. Die Konstanten wurden erneut aus [37] übernommen.

Konstruktionsabgleich

Mit der Berechnung der Ecken lässt sich die Zeichnung in ihre geometrischen Teile zerlegen: Eckpunkte, Segmente und Kreisbögen. Durch Zuordnung dieser Sketch-Elemente zu den bereits existierenden Elementen finden wir die Abhängigkeiten des neu konstruierten Elements.

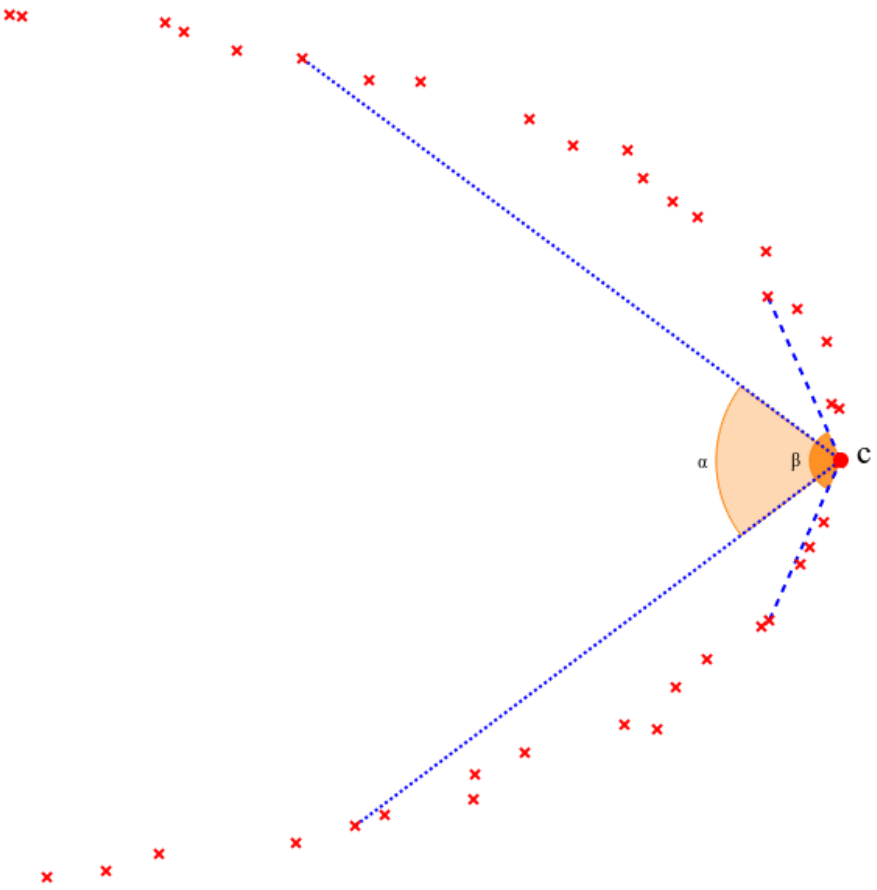


Abbildung 6.6: *Durch den Vergleich zweier Winkel kann ein Kreisbogen erkannt werden.*

Listing 6.1: ***Abhängigkeitsregeln für Geraden***

```
new Assessor.Assessment(
    new ver.Equals(
        new val.NumberSketchElements('arcs'),
        0
    ),
    new ver.LEQ(
        new val.NumberSketchElements('lines'),
        2
        ),
    new ver.Sketch('a'),
    new ver.Line('a')
), new Assessor.Assessment(
```

```
    new ver.Collected('P'),
    new ver.Point('P')
), new Assessor.Assessment(
    new ver.Collected('Q'),
    new ver.Point('Q'),
    new ver.NotIdentical('P', 'Q'),
    new ver.Score(
        new val.Div(
                1,
                new val.Distance('P', 'Q')
        )
    )
)
```

Für jedes konstruierbare Element gibt es eine Instanz der Klasse Assessment (siehe Kapitel 5), die diese Zuordnung definiert. In Listing 6.1 sind die Regeln zur Konstruktion einer Geraden zu sehen. Wie dort bereits zu sehen ist, wurde practice für sketchometry um einige neue Verifier und eine neue Value erweitert:

Elemental Verifier

- *Sketch(e), NotSketch(e), Collected(e)* testen den Status des Elements. Wurde es gezeichnet oder existierte dieses Element schon vorher und wurde beim Zeichnen überstrichen?
- *SketchArc(e)* führt zwei Tests durch: Handelt es sich bei dem Element um einen Sketch und hat IStraw diesen Teilstreckenzug als Kreisbogen erkannt?

Geometric Relation Verifier

- *MatchLines(g, h)* vergleicht die Steigung und den Y-Achsenabschnitt zweier Linien miteinander. Für eine robustere Berechnung wird, anstatt die Steigungen direkt zu berechnen, die zweite und dritte Komponente y und x der `stdform` der Linien herangezogen. $\frac{y_1}{x_1}$ beziehungsweise $\frac{y_2}{x_2}$ entspricht der Steigung der ersten beziehungsweise zweiten Linie. Wir berechnen jedoch $y_1 x_2$ und vergleichen den Wert mit $x_1 y_2$. Der Y-Achsenabschnitt entspricht der ersten Komponente der `stdform`.

Als Score werden die Differenz des Steigungsvergleichs und des Y-Achsenabschnittvergleichs zurückgegeben.

- *PointOnSketchSegment(l, P)* erweitert den PointOnLine Test um eine Messung des Abstands des Punkts zum Mittelpunkt der definierenden Punkte der Strecke, um festzustellen, ob der Punkt noch auf dem Segment liegt. Das Ergebnis des Skalarprodukts sowie der Bool-Wert, ob der Punkt auf der Strecke liegt, bilden zusammen den Score.

Value

- *NumberSketchElements(t)* dient zur Ermittlung der Anzahl der Sketchelemente vom Typ $t \in$ {'lines', 'arcs'}.

Integration von practice in die Sketch-Erkennung

Die ersten ein bis maximal drei Ergebnisse des $N-Protractor-Durchlaufs bestimmen, welche der insgesamt 13 Regelsätze im weiteren Verlauf verwendet werden. In Tabelle 6.1 ist exemplarisch das Ergebnis nach Analyse der Kreiszeichnung in Bild 6.7 zu finden. Demnach würde zuerst versucht, den Regelsatz für das Element *circle* anzuwenden. Sollte dies fehlschlagen, wird nach *quadrilateral* versucht den Regelsatz zu *triangle* anzuwenden, bevor die Sketch-Erkennung abbricht.

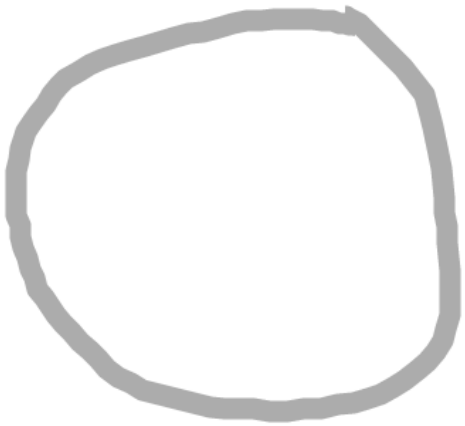


Abbildung 6.7: *Die Skizze zur Auswertung in Tabelle 6.1.*

Klasse	Bewertung
circle	-0.998683941294157
quadrilateral	-0.9936367454665259
triangle	-0.9737129543534823
circle2points	-0.904884193867908
normal	-0.7847591472702016
...	...

Tabelle 6.1: *Ausschnitt aus der Ergebnisliste einer $N-Protractor Analyse der Kreiszeichnung in Abbildung 6.7.*

Blicken wir wieder auf die Regeln zur Geraden in Listing 6.1 sehen wir drei Instanzen der Klasse `Assessment`. Die erste Instanz stellt sicher, dass in der Zeichnung keinerlei Kreisbögen zu finden sind und höchstens zwei Segmente. Die Parameter `Sketch` und `Line` weisen einem vorhandenen Segment den Namen a zu, um ihn in späteren Instanzen gegebenenfalls referenzieren zu können. Instanz zwei durchsucht die Menge der während der Zeichnung eingesammelten Elemente nach einem Punkt und weist ihm die interne Referenz P zu. Das dritte `Assessment`-Objekt überprüft, ob noch einer oder mehrere Punkte eingesammelt wurden, gibt ihm die Referenz Q und stellt sicher, dass die beiden Punkte P und Q nicht identisch sind. Für den Fall, dass mehr als zwei Punkte überfahren wurden, ermittelt `Score` den Kehrwert der Distanz von P und Q.

In sketchometry werden Assessments im Sammelmodus verwendet. Wir erhalten daher eine Liste von Fixtures, die alle im Regelsatz definierten Bedingungen erfüllen. Aus dieser Liste wird das Fixture gewählt, dessen Bewertungsvektor *score* die kleinste euklidische Norm besitzt. Gibt es mehrere mit der gleichen Bewertung, wird das erste aus der Liste übernommen. Aus der Fixierung erhalten wir alle zur Konstruktion des Elements benötigten Elternelemente.

Werden, wie bei den Regeln zur Geraden, mehrere Assessments konsekutiv ausgeführt, so wird das zuletzt bestimmte Fixture zur Initialisierung des nächsten Assessments benutzt.

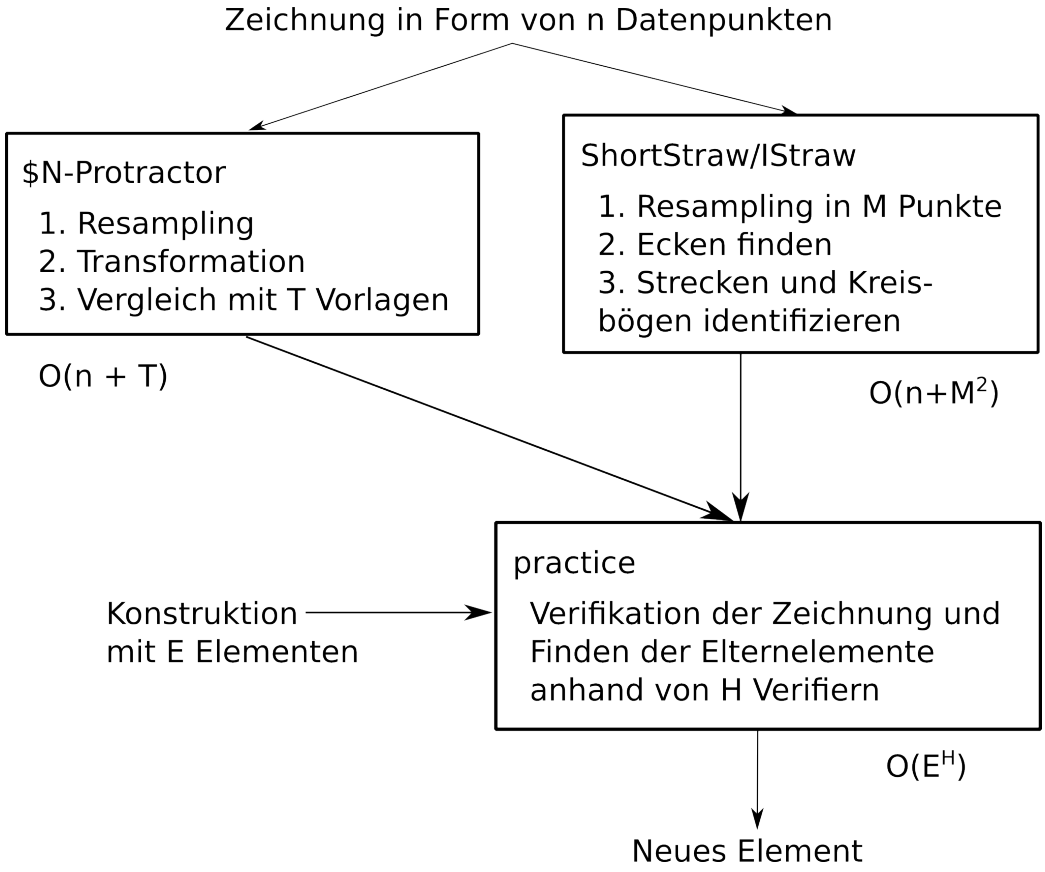


Abbildung 6.8: *Die Sketch-Erkennung im Überblick. Die Laufzeiten werden in Abschnitt 6.2.3 hergeleitet.*

6.2.3 Laufzeitanalyse

Die Normalisierung der Eingabe der Länge n für $N-Protractor besitzt eine asymptotische Laufzeit von $O(n)$. Zur Ermittlung der Klassifizierungsliste wird die normalisierte Eingabe mit T Vorlagen verglichen. Jeder Vergleich berechnet Skalarprodukte des Eingabevektors mit der Vorlage und benötigt daher $O(N)$ Schritte. Der gesamte Vergleich hat daher eine Laufzeit von $O(N \cdot T)$. Insgesamt erhalten wir für $N-Protractor $O(n + N \cdot T)$. Da N in unserem Fall konstant $N = 96$ ist, erhalten wir $O(n + T)$.

Da beim Resampling-Schritt in IStraw die neu berechneten, auf der Kurve äquidistanten Punkte in der Berechnung der Normalisierung mit einfließen, beträgt die Laufzeit hier $O(n+M)$, wobei M gleich der Anzahl der resampleten Punkte ist. Zur Ermittlung der Eckenkandidaten werden lediglich die M

resampleten durchlaufen. Wir erhalten $O(M)$. In der Nachbearbeitung der Kandidatenliste betrachten wir alle C Eckenkandidaten. Für je zwei Ecken kommen im schlimmsten Fall zur Ermittlung einer neuen Ecke im Linientest die dazwischenliegenden resampleten Punkte hinzu. Sowohl der Kollinearitätstest als auch der Winkeltest findet keine neuen Ecken und läuft demnach in $O(2C)$ beziehungsweise $O(C)$. Im schlimmsten Fall ist die Laufzeit der Nachbearbeitungsphase daher $O(M^2 + 3 \cdot C) = O(M^2 + C)$. Diese ist gleich $O(M^2)$, weil $C < M$. Insgesamt erhalten wir für IStraw $O(n + M^2)$.

Der letzte Schritt der Sketch-Erkennung ist eine reine Tiefensuche. Die Lösungsmenge lässt sich auf einen Baum abbilden. Die Anzahl der möglichen Elemente, die ein Verifier wählen kann, gibt die Verzweigungen wieder. Die Anzahl der Verifier entspricht theoretisch der Höhe des Baums. Wir erhalten demnach im schlechtesten Fall eine Laufzeit von $O(E^H)$ [30] mit der Anzahl der zur Auswahl stehenden Elemente E und der Anzahl der Regeln H. Die Anzahl der Elemente wird minimiert, indem nur die Elemente in Betracht gezogen werden, die während des Zeichnens überfahren wurden. Hinzu kommen hier noch die über IStraw ermittelten Sketch-Elemente. Insgesamt liegt die Anzahl der zur Auswahl stehenden Elemente in der Regel jedoch im unteren zweistelligen Bereich. Unter den Regelsätzen zählen derzeit in sketchometry die Regeln zu *tangent* sowie *parallel* zu den größten mit bis zu neun Regeln in einem Satz. Allerdings führt nicht jede Regel neue Verzweigungen ein, beispielsweise beginnt *parallel* mit den beiden Regeln in Listing 6.2.

Listing 6.2: ***Ausschnitt aus Regelsatz zur Konstruktion einer Parallelen***

```
new ver.Sketch('a')
new ver.Line('a')
```

Dort fügt nur die erste Regel Verzweigungen ein. Die zweite Regel erweitert lediglich die Bedingung *Sketch* um eine weitere Bedingung *Line*. Unter Berücksichtigung dieses Umstands liegt die maximale Höhe aller Regelsätze momentan bei $H \leq 5$.

Insgesamt erhalten wir eine Laufzeit von maximal

$$O(n + T + M^2 + E^H) \qquad \text{mit } H \leq 5.$$

6.3 Evaluation

Um festzustellen, wie sketchometry sich in der Praxis verhält, insbesondere mit Blick auf Einsteigerfreundlichkeit und Nutzbarkeit auf Tablets und PC, wurde eine Evaluation durchgeführt. Diese ist nicht repräsentativ, genügt aber, um Probleme in der Sketch-Erkennung aufzudecken.

Teilgenommen haben insgesamt 116 Schüler. Diese sind unterteilt in eine 7. Klasse, die zum Zeitpunkt der Evaluation bereits zwei Monate mit sketchometry gearbeitet hat, mit 21 Datensätzen und je zwei 8. und 10. Klassen mit insgesamt 47 beziehungsweise 48 Datensätzen. Ein Datensatz enthält die Antworten auf zwei gestellte Fragen und die Lösungsversuche von 21 beziehungsweise 27 Aufgaben. Zur Auswahl standen 27 Aufgaben, wovon die Schüler der 7. Klasse aufgrund ihres Kenntnisstands nur 21 bearbeiten konnten. Den 8. und 10. Klassen wurden alle 27 Aufgaben zur Bearbeitung gestellt. Auf der beiliegenden CD-ROM sind die Fragen und die Aufgaben zusammen mit dem Evaluationsprogramm zu finden, im Folgenden wird eine kurze Übersicht gegeben.

6.3.1 Durchführung

Die beiden Fragen lauteten:

- "Hast Du sketchometry schon einmal ausprobiert?" mit den Antwortmöglichkeiten `Ja` und `Nein`.
- "Welches Eingabegerät nutzt du?" mit den Antwortmöglichkeiten `Finger`, `Maus`, `Stylus/Stift` und `Touchpad (Notebook)`.

Die Richtigkeit dieser Angaben kann nicht überprüft werden. Allerdings wurden bei der Bearbeitung der Aufgaben von den Schülern ausschließlich Tablets ohne Stylus und Stift oder PCs mit Maus verwendet. Dennoch haben sechs Teilnehmer angegeben, sie hätten mit Notebooks gearbeitet. Einer gab an, er hätte die Aufgaben mit Stylus oder Stift bearbeitet. Die Vermutung liegt nahe, dass es hier zu Verwechslungen kam. Zur Vereinfachung der Auswertung werden die sieben Falschangaben zu den Tablets hinzugezählt.

Nach dieser Korrektur wurden die Aufgaben von 57 Teilnehmern mit Tablets bearbeitet. 59 Schüler benutzten einen Desktop PC mit Maus. 71 Teilnehmer gaben an, sketchometry vor dem Test schon ausprobiert zu haben. 45 haben eigenen Angaben zufolge vorher noch nie mit sketchometry gearbeitet.

Die zur Verfügung stehenden 27 Aufgaben wurden anhand ihrer zugrunde liegenden Sketch-Form in elf Aufgabenklassen aufgeteilt:

- *bisector*, vier Aufgaben zum Einzeichnen einer Winkelhalbierenden,
- *circle2p*, eine Aufgabe zur Erzeugung eines Kreises anhand eines Mittel- und eines Kreispunkts,
- *circumcircle*, Zeichnen eines Umkreises,
- *free*, Zeichnen einer freien Gerade sowie jeweils eines freien Kreises, Dreiecks und Vierecks,
- *line*, Konstruktion jeweils einer Strecke, einer Halbgeraden und einer von zwei Punkten abhängigen Geraden,
- *midpoint*, zwei Aufgaben zum Zeichnen eines Mittelpunkts,
- *normal*, fünf Aufgaben zur Konstruktion einer Normalen,
- *parallel*, Erzeugen von drei Parallelen,
- *polygon*, Zeichnen jeweils eines Dreiecks oder Vierecks durch drei beziehungsweise vier vordefinierte Punkte,
- *reflection*, Spiegelung eines Punktes an einer Geraden und
- *tangent*, Einzeichnen einer Tangente.

Aus dem Aufgabenvorrat der Schüler der 7. Klasse wurden die der Klasse circumcircle, parallel und tangent komplett, sowie eine Aufgabe aus dem Bereich normal weggelassen, da die dazugehörigen Themen im Unterricht noch nicht behandelt wurden.

Vor der Bearbeitung der Aufgaben wurden neben einer Bitte, die Angaben aufmerksam durchzulesen, auch die folgenden Hinweise eingeblendet:

- Alle Aufgaben sind mit genau einem Sketch lösbar.
- Bereits existierende Elemente dürfen verschoben werden, auch wenn das zur Lösung keineswegs notwendig ist.
- Mit dem Button *Erneut versuchen* kann eine Aufgabe jederzeit zurückgesetzt werden.

Die Teilnehmer der 8. und 10. Klassen erhielten außerdem eine kurze Einführung in die Konstruktion mit sketchometry. Diese umfasste neben einer gedruckten Übersicht über alle Sketches, die ihnen die gesamte Zeit zur Verfügung stand, eine kurze Erklärung und einmalige Demonstration der relevanten Sketches.

6.3.2 Auswertung

Zu jeder Aufgabe wurden alle Sketches sowie (falls vorhanden) der dazugehörige Konstruktionsschritt, das Ziehen vorhandener Elemente und jedes Zurücksetzen der Aufgabe durch den Nutzer aufgezeichnet. Einer oder mehrere solcher Schritte werden in der Auswertung zusammengefasst und nach den folgenden Regeln bewertet. Zu jeder bearbeiteten Aufgabe entstand so eine Liste von Bewertungen.

- *all-good* umfasst einen Sketch- und einen Konstruktionsschritt, in dem der Nutzer korrekt zeichnet und die Sketch-Erkennung die Zeichnung korrekt erkennt. Es werden auch Zugschritte mit dazugenommen, wenn der Nutzer vor der Konstruktion Elemente umpositioniert oder hinterher die Konstruktion überprüft. Es wird höchstens eine all-good Bewertung pro Nutzer und Aufgabe aufgenommen.
- *accidental-move*, ein Element wird versehentlich gezogen. Das heißt, ein Element wird entlang eines Pfads verschoben, auf dem ein Sketch zu erwarten ist. Ein Sketch beginnt im nächsten Schritt an der ursprünglichen Stelle des Elements oder der Nutzer setzt anschließend die Aufgabe zurück.

- *accidental-point*, ein Punkt wird erzeugt, der in der weiteren Bearbeitung der Aufgabe keine Verwendung findet.
- *missed-deps*, die Zeichnung des Nutzers ist zu weit entfernt von Elementen, die für die Konstruktion wichtig sind.
- *wrong-sketch*, der gezeichnete Sketch weicht zu stark vom erwarteten Sketch ab.
- *wrong-element*, trotz eines korrekt gezeichneten Sketches wird ein anderes Element erkannt und konstruiert.
- *no-recognition*, trotz eines korrekt gezeichneten Sketches wird nichts erkannt und es erfolgt daher kein Konstruktionsschritt.
- *wrong-deps*, die Sketch-Erkennung erkennt Abhängigkeiten nicht, obwohl sie in der Zeichnung erkennbar sind und der Nutzer diesen Fehler durch weitere Sketches versucht zu korrigieren.
- *wrong-props*, korrekt konstruierte Elemente erhalten falsche Eigenschaften. Dies betrifft hauptsächlich Linienelemente, beispielsweise wenn eine Strecke gezeichnet wurde und statt dessen eine Gerade erzeugt wird. Bei circle2points wurden manchmal die Elternelemente vertauscht: Aus dem Kreispunkt wurde der Mittelpunkt und umgekehrt, trotz korrekter Zeichnung.
- *wrong-place*, das korrekt konstruierte Element wurde an einer falschen Stelle platziert oder ist deformiert (nur Polygone). Dieser Fehler trat nur bei Parallelen, Winkelhalbierenden und Polygonen auf.
- *other-sketch-mistake*, sonstige Fehler, die auf die Sketch-Erkennung zurückzuführen sind.
- *user-mistake*, sonstige Fehler, die auf eine fehlerbehaftete Eingabe zurückzuführen sind.

6.3.3 Ergebnisse

Viele der Bewertungen sind eng genug gefasst, um keiner Beispiele oder ausführlichere Erläuterungen zu bedürfen. In den folgenden Abschnitten werden einige Bewertungen genauer betrachtet.

user-mistake & other-sketch-mistake

Die Bewertungen user-mistake und other-sketch-mistake sind Sammelkategorien für Fehler, die nicht häufig genug auftreten, um ihnen eine eigene Kategorie zu widmen. Meist beruhen sie auf Missverständnissen seitens der Teilnehmer. So wurde vereinzelt anstelle eines Umkreises oder Dreiecks drei Strecken eingezeichnet. Andere haben versehentlich die Hand auf dem Tablet abgelegt und dadurch den Sketch unkenntlich gemacht. Mit Anteilen von 0,17% beziehungsweise 1,18% an der Gesamtzahl sind beide Bewertungen vernachlässigbar.

wrong-sketch

Das größte Problem an dieser Stelle ist nicht das Verwechseln der Sketches. Dies trat zwar auf, erweckt aber eher den Eindruck, als wären die Aufgaben nicht gelesen worden. Beispielsweise beim Konstruieren einer Parallelen: obwohl eine Normale gefordert war zeichneten Teilnehmer vereinzelt eine Parallele. wrong-sketch trat häufig bei Aufgaben der Klassen bisector, line und circle2point auf. Anstatt den circle2point Sketch durchzuführen, wurde von manchen Teilnehmern versucht ein Kreis um den Mittelpunkt zu zeichnen und dabei den Kreispunkt zu überstreichen.

Um bei der Konstruktion von Halbgeraden und Strecken das Problem mit dem korrekten Abstand zum ersten Elternpunkt zu umgehen, versuchten einige Probanden mit einer Strecke senkrecht zur eigentlichen Linie zu beginnen, wie in Abbildung 6.9 zu sehen. Die Sketch-Erkennung erkennt dies allerdings als Winkel oder Normale. Selten kam es zu Situationen wie in Abbildung 6.10.

wrong-element

Aufgrund der Aufteilung der Sketch-Erkennung in zwei Phasen und der Ähnlichkeit mancher Zeichnungen, kann es zu Verwechslungen und damit zur Konstruktion falscher Elemente kommen. Um solche Situationen zu verhindern, werden für manche Sketches für die geometrische Analyse bereits die Elternelemente herangezogen. Einige Fehlinterpretationen der Sketch-Erkennung lassen sich damit verhindern, aber nicht alle. Bei den freien Ele-

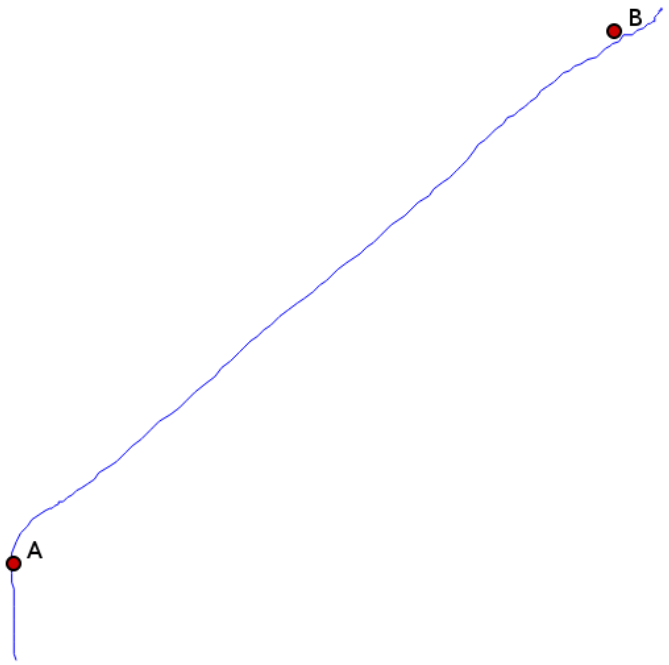


Abbildung 6.9: *Um zu verhindern, dass anstelle eines Sketches das Ziehen eines Elements begonnen wird, versuchen manche Nutzer der eigentlichen Zeichnung eine kleine, meist orthogonale Strecke voranzustellen. Dies klappt bei manchen Sketches, bei Strecken und Halbgeraden allerdings (noch) nicht.*

menten werden bei nicht sauber genug gezeichneten Skizzen statt Kreisen Vierecke oder Dreiecke gezeichnet. Umgekehrt treten ähnliche Probleme auf. Welche Skizzen mit welchen Elementen verwechselt wurden zeigt Tabelle 6.2.

Allgemeine Auswertung

Unproblematisch ist die Erzeugung von freien Elementen (freie Gerade und Kreis sowie Polygone ohne bereits existierende Punkte), Parallelen und Normalen. Diese gelingen in den meisten Fällen auf Anhieb. Die aufgetretenen Probleme bei freien Elementen betreffen meist Polygone. Oft handelte es sich hier um falsche Sketches. Auch eine zu ungenaue Zeichnung führt zu falsch erkannten Elementen. Alle Probleme betrafen jeweils weniger als 10% der Nutzer.

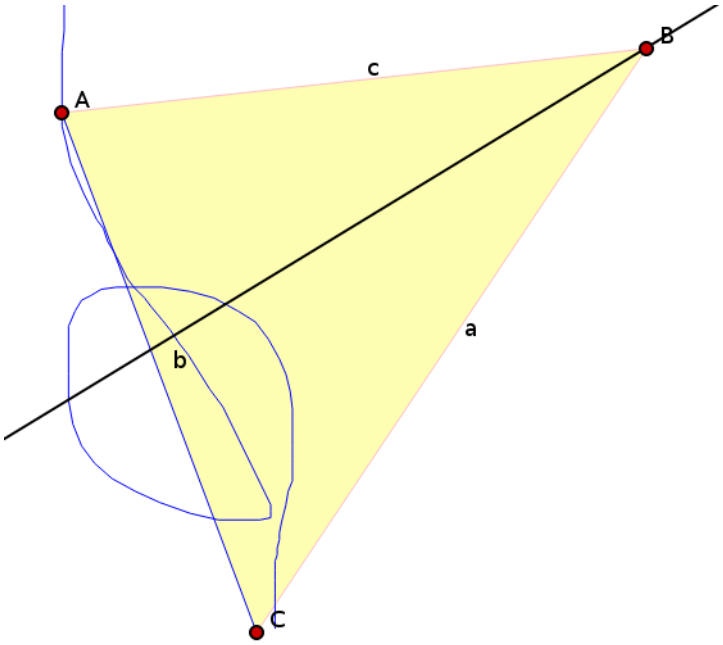


Abbildung 6.10: *Selten kam es zu Kuriositäten wie hier. Die Aufgabe bestand darin, eine Winkelhalbierende zu konstruieren. Der Nutzer zeichnet aber einen Mittelpunkt-Sketch mit zu großer Schleife und verfehlt den zweiten Punkt – die Sketch-Erkennung erkennt eine Winkelhalbierende.*

Bei Normalen und Parallelen fällt die Häufung von *wrong-sketch* auf (33% bei Normalen, 19% bei Parallelen). Einige Nutzer haben anstatt des Z- beziehungsweise L-Konstruktions-Sketches einfach parallel beziehungsweise orthogonal liegende Linien eingezeichnet. Das ansonsten gute Abschneiden dieser beiden Sketches liegt vermutlich an deren Schlichtheit. Auch die Tatsache, dass Nutzer hier nur selten direkt über Elternelementen beginnen kommt hier zum Tragen. So besitzt *accidental-move* einen bemerkenswert niedrigen Anteil am Ergebnis von 9% bei Normalen beziehungsweise 12% bei Parallelen.

Versehentliches Ziehen (*accidental-move*) ist auch der Grund, warum abhängige Polygone schlechter abschneiden. Nutzer beginnen hier oft zu nahe an einem Punkt, was bei 31% der Nutzer zu einer Verschiebung eines Punktes führt, anstatt eine Konstruktion anzustoßen. Wird dieses Verhalten aus der Auswertung entfernt, erhält Polygon ähnliche Werte wie Parallele und Normale. Weitere Probleme sind falsch erkannte Abhängigkeiten (14%) und falsch gezeichnete Sketches (11%). Die falschen Abhängigkeiten rühren von

bisector	angle, circle, line, midpoint, normal, parallel, tangent
circle	quadrilateral, triangle
circle2point	midpoint, quadrilateral, reflection, triangle
circumcircle	line, triangle
line	angle
midpoint	bisector, line, normal, parallel, reflection
normal	angle, line, parallel
parallel	circle, line, normal, quadrilateral
quadrilateral	circle, triangle
reflection	circle, circle2point, line, normal, parallel
tangent	circle, circle2point, midpoint, reflection, triangle
triangle	circle, quadrilateral

Tabelle 6.2: *wrong-element: Welche Elemente in der Evaluation von der Sketch-Erkennung verwechselt wurden.*

unsauber gezeichneten Ecken und eines absichtlich in der Aufgabe ungünstig platzierten Punktes her. Dieser verursacht bei unachtsamen Nutzern fälschlicherweise das Erzeugen eines Fünfecks anstelle eines Vierecks. Die Bewertung eines Versuchs als "falsche Zeichnung" schließt neben völlig falschen Zeichnungen, oftmals basierend auf einem Unverständnis der Aufgabenstellung, auch Konstruktionsversuche mit ein, bei denen der Zeichenvorgang unbeabsichtigt unterbrochen wurde.

Auch bei Umkreisen ist das versehentliche Ziehen das größte Problem (71%). Anstatt eines Kreises wurden in 27% der Fälle auch andere Sketches gezeichnet: Teilweise deuten diese Zeichnungen auch hier auf Schwierigkeiten beim Zeichenvorgang selbst hin, der versehentlich abgebrochen wird (unvollständige Kreise, teilweise reduziert auf kurze Geradenstücke), oder fehlendes Verständnis der gestellten Aufgabe (Zeichnen von Polygonen oder Mittelpunkt-Sketches). Versehentlich erzeugte Punkte betrafen 20% der Nutzer. Nicht überstrichene beziehungsweise falsch erkannte Abhängigkeiten liegen mit 16% beziehungsweise 14% relativ niedrig.

Beim Mittelpunkt fallen die abhängigkeitsbezogenen *missed-deps* und *wrong-deps* verhältnismäßig niedrig aus (<10%). Das liegt auch daran, dass knapp verpasste Abhängigkeiten in diesem Fall zu keiner Konstruktion (*no-recognition*) führen, die hier mit 64% relativ hoch ausfällt. Versehentliches Ziehen und Punkte erzeugen sind auch hier im üblichen Ausmaß vertreten (36% und 12%). Ansonsten sind noch *wrong-sketch* und *wrong-element* zu erwähnen (32% und 20%), beide ebenfalls im üblichen Ausmaß.

Winkelhalbierende sind insofern problematisch, als dass ein Sketch auf einer Linie beginnt und diese Linie entlanggefahren werden muss. Abgesehen vom Deaktivieren des Zugmodus, kann man sich hier nur mit einem Trick behelfen: Außerhalb der Linie mit einem sehr kurzen Lot auf diese Linie beginnen (siehe Abbildung 6.11). Vor diesem Hintergrund ist es wenig erstaunlich, dass *accidental-move* bei jedem Nutzer mindestens einmal auftritt. Ein Lösungsansatz wäre, die Strategie einiger Nutzer, dieses Problem zu umgehen, aufzugreifen und als gültigen Sketch zu akzeptieren: Anstatt auf einem der Schenkel beginnt man in der Mitte mit der Winkelhalbierenden. Die Form des Sketches ist ansonsten gleich. In der aktuellen Auswertung wurde dies jedoch als *wrong-sketch* (44%) gewertet. *accidental-point* tritt bei 23% der Nutzer auf. Die Ursache für ein falsch erkanntes beziehungsweise kein erkanntes Element liegt auch oft in der Vermeidung von versehentlichem Ziehen (28% bzw 17%).

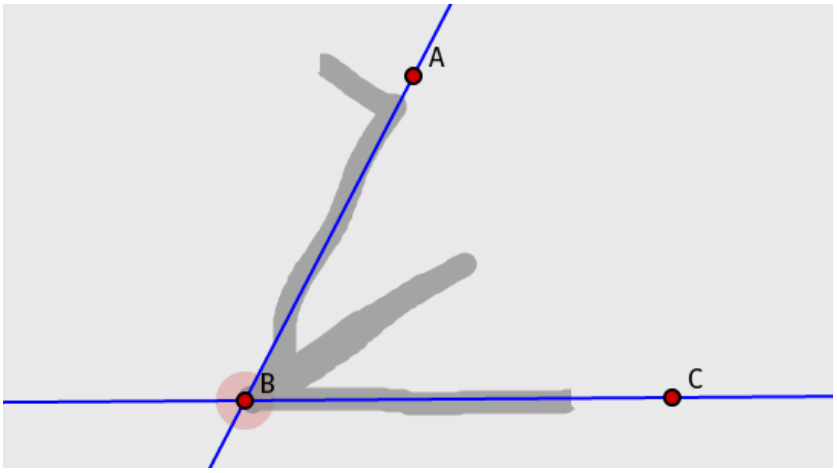


Abbildung 6.11: *Um zu verhindern, dass eine der Geraden verschoben wird, anstatt eine Konstruktionszeichnung zu beginnen, muss man derzeit im kombinierten Modus außerhalb der Geraden anfangen. Die Sketch-Erkennung ignoriert das kurze Lot.*

Auch bei abhängigen Linien wird meist in der Nähe von Punkten begonnen, insbesondere bei Strecken. Bei (Halb-)Geraden ist dies nicht der Fall, beziehungsweise ließe es sich vermeiden. Das ist auch der Grund, warum versehentliches Ziehen hier nicht über 81% steigt. Selbst wenn es möglich ist, Halbgeraden so zu zeichnen, dass man nicht über einem Punkt beginnt, tendieren Nutzer eher dazu, von der linken Seite der Zeichenfläche aus nach rechts oder von oben nach unten zu zeichnen, auch wenn das bedeutet über einem kritischen Punkt zu beginnen. Einige Nutzer haben die bei der Winkelhalbierende beschriebene Strategie zur Vermeidung des versehentlichen Ziehens

verwendet. Sie versuchen den Sketch mit einer kurzen orthogonalen Strecke auch bei Strecken zu beginnen, was in eindeutigen Fällen mit *wrong-sketch* (24%) und in weniger deutlichen Fällen mit *no-recognition* (18%) bewertet wurde. Auch die Bewertung *wrong-props* (39%) hat ihren Ursprung in diesem Problem: Beginnt oder beendet der Nutzer seine Zeichnung zu weit entfernt, wird leicht aus einer Strecke eine (Halb-)Gerade, was intern als Eigenschaft des Linienobjekts geregelt wird – daher auch der Name: prop steht für *property*, die Eigenschaft. *wrong-deps* (40%) und *missed-deps* (20%) stehen für nicht eingesammelte Abhängigkeiten.

Bei Tangenten besteht das Hauptproblem aus falschen Sketch-Zeichnungen (115%). Die Dokumentation ist hier mehrdeutig, siehe auch Abbildung 6.12. Hinzu kommt der häufige Versuch, ähnlich wie bei Parallelen und Normalen, eine einfache Linie zu zeichnen. Wird der Sketch korrekt gezeichnet, aber ist der orthogonale Teil zum Kreismittelpunkt hin zu lang, werden fälschlicherweise Kreise, Mittelpunkte und weitere Elemente erzeugt (88%), oder überhaupt nichts erkannt (45%). Versehentliches Ziehen tritt bei Tangenten relativ selten auf (27%).

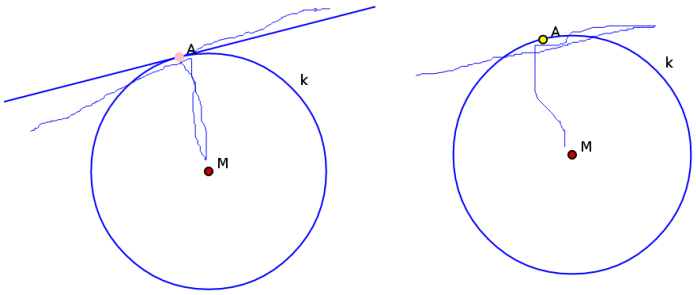


Abbildung 6.12: *Die Vorlage zur Tangente (Abb. B.10) verwirrte einige Nutzer, die statt der korrekten Zeichnung auf der rechten Seite einen Sketch ähnlich dem linken zeichneten.*

Die größten Probleme hatten die Schüler bei der Konstruktion von Kreisen, die anhand eines Mittelpunkts und eines Kreispunkts definiert werden (*circle2point*). Auch die Spiegelung eines Punktes an einer Geraden bereitete Schwierigkeiten.

Da der circle2point-Sketch nahe beim Mittelpunkt begonnen wird, ist das versehentliche Ziehen stark vertreten (101%). Außerdem bereitet der Unterschied zwischen "Kreis durch zwei Punkte" und "Kreis" Verständnisprobleme. Oft wurde versucht, das Element mit einem einfachen Kreis um den Mittelpunkt und durch den Kreispunkt zu zeichnen. Diese und andere falsche Sket-

ches traten bei 93% der Nutzer auf. Falsche beziehungsweise keine Elemente wurden in 69% beziehungsweise 40% der Abgaben konstruiert. Dies ist meist auf das Komplettieren des Sketches zu einem vollen Kreis zurückzuführen, wogegen die Sketch-Erkennung nur einen etwa viertel Kreisbogen erwartet. Versehentlich erzeugte Punkte betrafen 59% der Nutzer, verpasste und falsch erkannte Abhängigkeiten nur 20% und 14%.

Aufgrund der Ähnlichkeit des Reflektions-Sketches zu den Sketch-Klassen angle, normal, circle2point und bisector kam es sehr häufig zu den Bewertungen *no-recognition* (251%) und *wrong-element* (63%). Wegen des Starts der Zeichnung am zu spiegelnden Punkt, kam es oft zur Verschiebung des Punkts anstelle einer Konstruktion (66%). Anstatt des Reflektions-Sketches wurden Linien, selten komplett andere, auch sonst nicht verwendete Zeichnungen versucht (66%). Unabsichtliche Punkte (20%) kamen auch hier vor. Selten (14%) wurde die Pfeilspitze zu weit entfernt von der Spiegelachse gezeichnet.

Einsteiger und Erfahrene

Im Schnitt brauchten Einsteiger in fast allen Aufgabenklassen mehr Versuche als Erfahrene, um ein bestimmtes Element zu konstruieren. Lediglich bei der Zeichnung von Mittelpunkten und Polygonen schnitten Einsteiger besser ab. Die Unterschiede sind jedoch minimal.

Freie Elemente sind beiderseits unproblematisch, lediglich falsch erkannte Elemente fallen hier geringfügig auf. Diese wurden in 75% der Fälle verursacht durch als Viereck erkannte Kreise und als Kreis erkannte Vierecke. Bei Normalen und Parallelen sind Einsteiger ebenfalls entweder gleichauf mit erfahrenen Nutzern, oder die absoluten Fallzahlen der einzelnen Ergebnisse sind zu gering, um Schlüsse ziehen zu können.

Interessant sind die Resultate der abhängigen Polygone. Hier haben Einsteiger größtenteils wesentlich besser abgeschnitten als Erfahrene. Bei Umkreisen hatten Einsteiger wesentlich öfter Probleme mit dem versehentlichen Ziehen und Erzeugen von Punkten, sowie dem Verwenden der verkehrten Sketches.

Bei den restlichen Aufgaben wurden Sketch-Versuche von Einsteigern öfter mit *no-recognition* und *wrong-element* bewertet. *wrong-sketch* und *accidental-point* trat bei den Aufgaben zu Winkelhalbierenden, Kreis durch zwei Punkte

und Reflektion häufiger auf. Erfahrenen Nutzern fiel es außerdem leichter, bei Reflektionen und abhängigen Linien die Elternelemente bei der Zeichnung zu überstreichen. Ebenfalls bei Linien werden bei erfahrenen Nutzern weniger falsche Abhängigkeiten und Eigenschaften erkannt.

Tablets und PCs

Beim Vergleich von Geräten mit Finger-Bedienung zu Maus gesteuerten Desktop-Rechnern fällt in der Gesamtübersicht auf, dass die Anzahl der nötigen Versuche niedriger ausfällt als bei Mauseingabe. Filtert man jedoch die Sketches mit Ergebnis *accidental-move* und *accidental-point* heraus, ergibt sich ein deutlich ausgewogeneres Bild.

Die größten Probleme gab es nach dem Filtern nur noch bei den Zeichnungen zu den Elementen Umkreis, Winkelhalbierende, Reflektion und Tangente. Bei den Ergebnissen fallen hauptsächlich *wrong-sketch* und *missed-deps* auf. Letzteres liegt zum Teil auch an der Tatsache, dass die Eingabe auf dem Ausgabegerät geschieht. Während der Zeichnung wird die Hand über den Bildschirm bewegt und versperrt teilweise die Sicht. Außerdem kann mit einem Finger oder einem Eingabestift für kapazitive Displays ("Stylus") nicht so exakt gearbeitet werden wie mit einer Maus. Die verstärkt auftretenden falschen Zeichnungen sind dadurch bedingt, dass Nutzer auf den Oberflächen eines Tablets abrutschen (zu sehen an unvollständigen Sketches und der Konstruktion eines Punktes am Ende der Zeichnung). Dadurch wird die Konstruktion abgebrochen und es kommt zu einer unvollständigen Zeichnung, die als *wrong-sketch* bewertet wird.

Auffallend ist das bessere Abschneiden der Tablet-Nutzer bei der Konstruktion von Mittelpunkten. Die Werte für *wrong-element* und *no-recognition* sind im Vergleich erstaunlich niedrig. Nimmt man die Ergebnisse von freien Kreisen mit hinzu, sieht man leicht, dass es auf Tablets einfacher ist, Kreise zu zeichnen, als an einem PC mit einer Maus. Vorausgesetzt, es gibt keine Restriktionen an die Kreisposition oder -größe.

6.3.4 Fazit

Das am häufigsten auftretende Problem ist das versehentliche Verschieben von Elementen im kombinierten Konstruktions- und Zugmodus. Mit einer zeitweisen Abschaltung des Zugmodus lässt sich dem effektiv entgegenwirken. Gegen versehentlich erzeugte Punkte oder falsche Zeichnungen hilft dies jedoch nicht.

Falsch und nicht erkannte Zeichnungen haben zwar die gleiche Ursache, allerdings hilft gegen falsch Erkanntes eine Verbesserung der Sketch-Erkennung nur bedingt. Gegen eine gescheiterte Erkennung könnten die weiteren Ergebnisse des $N-Protractor genommen und die geometrische Analyse darauf angewandt werden, sofern die Rechenleistung des verwendeten Gerätes das erlaubt.

Winkelhalbierende, Reflektion und Tangente könnten sowohl von einer Verbesserung der Dokumentation, als auch einer Verbesserung der Sketch-Erkennung profitieren. Damit bei der Winkelhalbierenden die Skizze nicht direkt über bestehenden Elementen begonnen werden muss, würde es sich anbieten, die Winkelhalbierende zuerst zu zeichnen und dann mit der Auswahl der Schenkel zu beenden. Auch wenn die Sketch-Erkennung beide Varianten erkennen sollte, die Dokumentation der neuen Variante würde es Einsteigern erleichtern. Hierfür müsste die Dokumentation der Sketches in der Übersicht insoweit angepasst werden, dass auch die Zeichenreihenfolge der einzelnen Linien und Kreisbögen eines Sketches ersichtlich wird. Dies käme den Nutzern auch bei anderen Sketches zugute, vor allem die statische Zeichnung der Tangente in der Übersicht ist nicht eindeutig in der Zeichenreihenfolge (siehe Abbildung 6.12).

Aufgrund der Probleme mit dem Reflektions-Sketch, muss über eine Neugestaltung des Sketches nachgedacht werden.

Kapitel 7

SketchBin

7.1 Einführung

Die Verknüpfung von dynamischen Geometriesystemen mit programmierbaren Elementen begann mit berechneten Punkten und Texten und wurde in vielen Systemen erweitert mit programmierten Nebenbedingungen. So erlauben unter anderem GEONExT, GeoGebra und Cinderella.2 Nebenbedingunen an Konstruktionen zu stellen, um Elemente unter bestimmten Voraussetzungen ein anderes Erscheinungsbild zu geben oder sie zu verstecken [29, 17, 28]. Zumindest GeoGebra und Cinderella.2 erlauben auch das Konstruieren geometrischer Elemente. Allerdings sind diese Elemente in ihrer Funktion in GeoGebra eingeschränkt, mittels `Point[{x, y}]` erzeugte Punkte lassen sich in der aktuellen Version nicht ohne weiteres verschieben. Im Handbuch zu Cinderella.2 wird bei Verwendung der `create()` Funktion zu Vorsicht geraten:

> Due to the fact that algorithms may create multiple outputs several subtleties arise. This function is meant for expert use only.

JSXGraph geht hier den umgekehrten Weg. Anstatt eine graphische Oberfläche anzubieten, die mit einer Programmierschnittstelle für integrierte Script-Sprachen angereichert wird, stellen wir die gesamte Funktionalität eines Dynamischen Geometriesystems über ein JavaScript API bereit.

Aufbauend auf dem JSXGraph API können graphische Benutzeroberflächen entwickelt werden, die in der Funktionalität zwar eingeschränkt werden, gegenüber einem reinen API aber für Einsteiger leichter zu bedienen sind und in der Regel bei einfachen Konstruktionen schneller zum Ergebnis führen. Beispiele für graphische Oberflächen, die auf JSXGraph aufbauen, sind sketchometry, "JSXGraph-GUI" von Darko Drakulic[1] und in einem gewissen Sinne auch der Funktionsplotter von Murray Bourne[2]. Eine Oberfläche, die genau wie sketchometry auf Sketches setzte, war "Chalkboard"[3]. Leider ist die Projektseite nicht mehr erreichbar.

All diese Oberflächen sind hervorragend dazu geeignet, um schnell und vor allem ohne Programmiererfahrung mathematische Visualisierungen erzeugen zu können. Allerdings stoßen Autoren mit allen drei Werkzeugen schnell an Grenzen, wenn es um komplexere Aufgaben geht. Über eine Programmiersprache und einer API wie JSXGraph lassen sich auch komplexe Aufgaben lösen.

Bei der Programmierung von Konstruktionen wie mit JSXGraph verlieren wir jedoch teilweise einen wichtigen Bestandteil dynamischer Geometriesysteme: Jede Änderung des Autors spiegelt sich sofort im Ergebnis wieder. Nicht speziell auf dynamische Geometriesysteme bezogen hat Bret Victor in seinem Vortrag "Inventing on Principle"[4] sich diesem Problem ganz allgemein angenommen. Sein erstes Beispiel im Vortrag ist ein JavaScript Editor, in dem mit Hilfe der HTML5 2D Canvas API eine Landschaftsszene gezeichnet wird. Anstatt die Programmierung und das Ergebnis in verschiedenen Programmfenstern vorzunehmen, werden diese in einem Fenster nebeneinander gelegt. Das gerenderte Bild aktualisiert sich automatisch mit jeder Änderung am Quelltext. Quellcode und Bild sind miteinander verknüpft. Anhand eines Bildpunkts kann die dazugehörige Code-Stelle herausgefunden und angezeigt werden und umgekehrt. Über graphische Bedienelemente wie Schieberegler, Bedienknöpfe und Farbauswahldialoge, die über dem Quelltext eingeblendet werden lassen, sich die Werte einzelner Variablen verändern. Die Zeichnung wird auch hierbei laufend aktualisiert.

[1] http://eudzbenici.rs.ba/svn.eudzbenici.rs.ba/GUI/, März 2014

[2] http://www.intmath.com/functions-and-graphs/graphs-using-jsxgraph.php, März 2014

[3] http://remember2015.info/labs/chalkboard/page/Geometry.html, Dezember 2010

[4] http://vimeo.com/36579366, März 2014

7.2 Architektur/Technische Details

Um die Möglichkeiten zur Entwicklung dynamischer Lernumgebungen für Autoren auf eine Ebene mit dem JSXGraph API zu stellen und gleichzeitig die Dynamik graphischer Eingaben zu behalten, bieten sich einige Ideen aus Bret Victors Vortrag an. Im Folgenden werden einige technische Details diskutiert, wie so ein Editor implementiert werden kann und im prototypischen SketchBin auch implementiert wurde.

Gleichzeitig bietet es sich an, die Idee weiter auszubauen und die Möglichkeit zu nutzen, die sich mit der dynamischen Geometrie bietet: Das direkte Erzeugen der graphischen Ausgabe wie in klassischen DGS oder in sketchometry mit der Maus.

Für SketchBin finden JessieCode und die bereits im vorherigen Kapitel besprochene Sketch-Erkennung Anwendung. Beides ist nicht zwingend notwendig, JessieCode wurde lediglich deshalb gewählt, weil dafür ein Parser mit Zugriff auf den AST vorhanden ist. Die Verwendung der Sketch-Erkennung zur graphischen Eingabe kann durch eine beliebige andere, auch klassische Toolbar und Menü Eingabe ersetzt werden.

Das Layout wurde weitgehend aus "Inventing on Principle" übernommen. Die Ansicht ist zweigeteilt in einen Codebereich und einen graphischen Bereich.

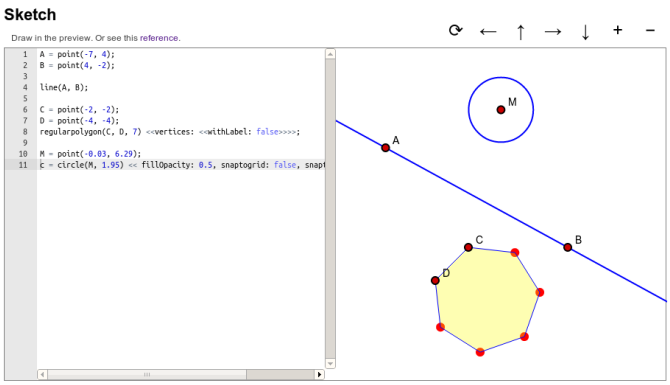


Abbildung 7.1: *SketchBin ist schlicht gehalten: Links das Code-Fenster, rechts das Graphik-Fenster.*

Bidirektionale Verknüpfung

Am ehesten vergleichbar mit der Idee der bidirektionalen Verknüpfung von Code und Graphik ist die zweigeteilte Ansicht von Algebra- und Geometrie-Fenster in GeoGebra. Diese ist allerdings relativ leicht zu realisieren. Ändert der Benutzer etwas in der Graphik, wird der entsprechende Eintrag im Algebra-Fenster neu generiert, indem die relevanten Daten aus internen Strukturen ausgelesen und formatiert werden. Wird im Algebra-Fenster ein Funktionsterm editiert, wird die Änderung geparst und auf die graphische Darstellung angewandt.

Etwas schwieriger ist die Situation, wenn die Code-Ansicht nicht mehr nur aus einer einfachen Liste mit ausgewählten Elementen und deren Eigenschaften besteht, sondern aus einer einzelnen Zeichenkette, welche die gesamte Konstruktion beinhaltet. Änderungen in der Zeichenkette lassen sich leicht auf die Graphik übertragen, indem die Graphik gelöscht und neu angelegt wird. Es mag effizientere Wege geben, aber prinzipiell ist dies ein zulässiger Ansatz, da über den Programm-Code die Konstruktion eindeutig definiert ist.

In der anderen Richtung, von der Graphik zum Code, ist dies nicht mehr der Fall. Der Code kann nicht aus der Graphik abgeleitet werden. Einer der Gründe ist, dass verschiedene Programme die gleiche Ausgabe erzeugen können. Listing 7.1 und Listing 7.2 sind unterschiedlich, erzeugen aber die gleiche Ausgabe in der Graphik-Ansicht.

Listing 7.1: ***Konstruktion von 3 Punkten, Variante 1***

```
point(0, 0);
point(1, 1);
point(2, 2);
```

Listing 7.2: ***Konstruktion von 3 Punkten, Variante 2***

```
for (i = 0; i < 3; i = i + 1) {
    point(i, i);
}
```

Viele Elemente des Programm-Codes gehen außerdem beim Interpretieren verloren, hauptsächlich Whitespaces und Kommentare. Ein Übersetzen der Graphik direkt in Programm-Code, ähnlich wie das in sketchometry im Rahmen der Sketch-Erkennung geschieht, kommt demnach nicht in Frage.

Vielmehr muss der Quelltext gezielt ausgebessert werden, wofür Informationen über die Struktur des Programm-Codes benötigt werden. Diese Informationen sind leicht aus den Daten auszulesen, die beim Parsen im JessieCode Interpreter anfallen. Konkret geht es hier meist um die genauen Positionen bestimmter Token. Diese werden vom Tokenizer über den Parser, der die Start- und Endzeilen und -spalten der Token in den AST Knoten ablegt, zum Interpreter weitergereicht. Trifft der Interpreter auf einen Aufruf einer Creator-Funktion wird diese ausgeführt, um den Rückgabewert zu bestimmen. Dieser Rückgabewert ist eine Referenz auf das erzeugte Element in der Graphik-Ansicht, in dem die Positionsdaten des Funktionsaufrufs gespeichert werden. Bewegt der Nutzer die Maus über das Element in der Graphik-Ansicht, kann die dazugehörige Zeile aus diesen Daten abgelesen und hervorgehoben werden. Die Rückrichtung funktioniert ähnlich: Hierfür wird der Rückgabewert der Creator-Funktion in einem Array hinterlegt, dessen einzelne Komponenten jeweils einer bestimmten Code-Zeile entsprechen. Wird die Cursor-Position im Code geändert, kann hieraus abgelesen werden, welche Elemente in der Graphik hervorgehoben werden müssen.

Zusätzlich zu den Start- und Endpositionen der Creator-Funktion wird die genaue Lage aller Parameter bestimmt und gespeichert. Beim Verschieben von freien Elementen können somit die Elternelemente gezielt und mit minimalen Änderungen am Code ausgebessert werden. Dies ist in SketchBin derzeit nur bei freien Punkten prototypisch implementiert.

Etwas einfacher als das Ändern bestehender Elemente gestaltet sich die Konstruktion neuer Elemente. Zeichnet der Benutzer in der Graphik-Ansicht ein neues Element ein, wird eine leicht abgewandelte Form des in sketchometry genutzten Codegenerators[5] genutzt. Die Änderungen beschränken sich auf die Generierung von besser lesbarem Code. Insbesondere werden Zahlenkonstanten auf zwei Dezimalstellen beschränkt. Die erzeugten Elemente werden automatisch erzeugten Variablen zugewiesen.

Dies sind die derzeit einzigen Möglichkeiten, aus der Graphik-Ansicht heraus die Code-Ansicht zu beeinflussen. Denkbar wäre auch die Translation von Funktionsgraphen. Beim Konvertieren eines Gleiters in einen freien Punkt und zurück könnte anstatt des entsprechend generierten Codes auch die Definition des Punkts modifiziert werden. Hier muss allerdings darauf geachtet werden, dass das Trägerelement vor dem Gleiter definiert ist.

[5] reader/sketch.js, Teil des JSXGraph Projekts

Die andere Richtung, die Aktualisierung der Graphik-Ansicht, nachdem Änderungen am Code vorgenommen wurden, ist zwar intuitiv leichter, verlangt aber auch die Lösung einiger kleinerer Probleme. Besonders die permanente Aktualisierung der Graphik-Ansicht verlangt hier Aufmerksamkeit. Der Unentscheidbarkeit des Halteproblems [31] nach ist es nicht möglich zu ermitteln, ob das eingegebene Programm terminiert oder nicht. Außerdem sollte es möglich sein, eine bereits laufende und noch nicht beendete Aktualisierung abzubrechen, wenn in der Zwischenzeit eine neue Aktualisierung angestoßen werden soll.

Eine Möglichkeit, diese Problematik anzugehen, ist Multithreading. Die im Browser verfügbare Multithreadinglösung heißt WebWorker[6]. Allerdings erlauben WebWorker keinen Zugriff auf den DOM. Das heißt Modifikationen der Graphik-Ansicht sind aus WebWorkern heraus nicht direkt möglich. Die Kommunikation von außen mit einem Worker-Thread geschieht auch niemals direkt, sondern nur über das Senden von *Messages* und Empfangen von *Events*. Hier gibt es zwei Möglichkeiten:

1. Das Rendering in JSXGraph umstellen von einer API auf Messages und Events die auch von WebWorkern aus verwendet werden können, oder
2. Implementieren einer Renderer-Attrappe und Abfangen aller Zugriffe auf den DOM im JSXGraph Kern.

Ersteres würde bedeuten, dass Berechnungen in JSXGraph allgemein mehrere Threads nutzen könnten. Allerdings würde dies massive Änderungen am JSXGraph Kern nach sich ziehen und erzeugt zudem einen Overhead der durchaus Auswirkungen auf die Rechenzeit haben kann. Außerdem müsste nicht nur das Rendering umgestellt werden, sondern insbesondere auch die Handhabung der Events.

Die zweite Lösung ist überschaubar und relativ leicht zu realisieren. Die API ändert sich nicht und ist daher abwärtskompatibel. Der Nachteil hier ist, dass der Code zweimal interpretiert werden muss: Einmal im WebWorker und, nachdem er dort erfolgreich ausgeführt wurde, anschließend im Hauptthread erneut, um die Graphik-Ansicht zu aktualisieren. Angemerkt sei hier, dass der erste Durchlauf im WebWorker dadurch schneller läuft, dass alle DOM Zugriffe wegfallen und einige Optimierungen in der JSXGraph Konfiguration

[6] http://www.w3.org/TR/workers/, März 2014

vorgenommen werden können, zum Beispiel die Anzahl der Funktionsauswertungen für Plots können drastisch reduziert werden.

Deshalb, und auch im Hinblick darauf, dass es sich bei SketchBin um einen Prototypen handelt, wurde vorerst die zweite Lösung gewählt.

Kapitel 8

Ausblick

In den vorangegangenen Kapiteln wurde stellenweise bereits angedeutet, wie die vorgestellten Projekte erweitert und verbessert werden können. Dieses Kapitel gibt darüber einen abschließenden Überblick.

SketchBin

Auch wenn die aktuelle Implementierung von SketchBin Prototyp-Charakter besitzt, ist es bereits einsetzbar. Verbesserungen allerdings sind in vielerlei Hinsicht vorstellbar. Für den Ausbau zu einem Werkzeug für Autoren wichtig ist der Ausbau der Eingabeformen: Neben Konstruktionen und beschreibenden Texten sollte das hinzufügen von Audio-, Video- und Bild-Dateien möglich sein. Der Export in verschiedene Formate ist notwendig, um die erstellten Konstruktionen auch außerhalb der SketchBin-Plattform zugänglich machen zu können. HTML und ePub sind zwei Formate, die auf jeden Fall unterstützt werden müssen. ePub ist ein offener Standard für elektronische Bücher, der auf den HTML5 Standard aufbaut.

Weiterhin sind auch Weiterentwicklungen am User Interface denkbar. Da SketchBin für Desktop-PCs konzipiert ist, ist es sinnvoll über das Hinzufügen eines klassischen User Interfaces mit Menüs und Toolbars nachzudenken – natürlich parallel zur Sketch-Erkennung. Dies würde Nutzern klassischer DGS den Einstieg in die Verwendung von SketchBin sehr erleichtern. Mo-

mentan kann bei Bedarf ein Farbauswahldialog eingeblendet werden, der die Bearbeitung von Farbwerten im Code erleichtert. Dies kann erweitert werden auf Slider für Zahlenwerte und Schaltflächen für Wahrheitswerte. Während Schaltflächen für Wahrheitswerte reinen Komfortcharakter haben, lassen sich mit Slidern für Zahlenwerte kontinuierliche Veränderungen realisieren. Im Gegensatz zu diskreten Änderungen über Tastatureingaben können mit Slidern Spezialfälle entdeckt werden. Dies kann sogar noch weiterentwickelt werden zur automatischen Generation von Slidern in der graphischen Ansicht der Konstruktion für besonders interessante Parameter, die im Endprodukt veränderbar sein sollen.

Betrachten wir den Programm-Code in 8.1, könnte über einen Rechtsklick auf dem konstanten Zahlenwert 4 ein Slider über der Code-Ansicht eingeblendet werden, der bei einer Veränderung sowohl den Wert der Konstanten in der Code-Ansicht als auch die Graphik-Ansicht aktualisiert.

Listing 8.1: ***Definition eines Funktionsplots in JessieCode***

```
plot(function (x) {
        return 4 * sin(x);
});
```

Auch bei der Modifikation des Programm-Codes durch die Graphik-Ansicht sind Verbesserungen denkbar. Das Ziehen und Drehen von Geraden und das Verschieben und Strecken von Kreisen kann sich genauso auf den Code auswirken wie das Ziehen von Punkten. Das Umwandeln von freien Punkten in Gleitern und umgekehrt kann die Definition des Punktes modifizieren, anstatt neuen Code hinzuzufügen.

Im SketchBin-Prototypen wird eine modifizerte Variante des sketchometry-Code-Generators genutzt, der aus einer Zeichnung über die Sketch-Erkennung JessieCode produziert. Allerdings liefert dieser zuviel irrelevanten Code. Eine Verbesserung käme auch dem JessieCode-Export in sketchometry zugute.

Sketch-Erkennung

Die Weiterentwicklung der Sketch-Erkennung wurde am Ende der Evaluation bereits angesprochen. Die Erweiterung auf andere Bereiche der Schulmathematik wie Analysis ist bereits begonnen. Hier kann die Sketch-Erkennung

nur im Rahmen der Handschrifterkennung genutzt werden. Allerdings gibt es einige andere Herausforderungen: Gibt es eine Möglichkeit, die Code- und Text-Eingabe bei berechneten Elementen zu reduzieren oder gar zu vermeiden?

practice

sketchometry lässt sich auch als Eingabe für practice nutzen. Zusammen mit dem Ansatz des automatischen Erkennens (siehe Abschnitt 6.1.5) können aus einer Beispielkonstruktion geometrische Eigenschaften erkannt werden, aus denen der Nutzer nur noch die auswählen muss, die für seine Aufgabe relevant sind. Mit der Integration eines automatischen Theorembeweisers kann man von numerischen Tests losgelöst geometrische Eigenschaften direkt ableiten.

JSXGraph und JessieCode

JSXGraph hat sich in den letzten sechs Jahren in vielen Projekten weltweit bewährt. Neben E-Learning und Nachhilfeseiten für Schüler gibt es auch Dozenten, die ihre Mathematik- und Physik-Vorlesungen online zur Verfügung stellen und mit JSXGraph-Konstruktionen erweitern. Genaue Nutzerzahlen sind zwar nicht ermittelbar, allerdings verzeichnet die Google Group des Projekts derzeit über 150 Mitglieder. Diese stellen nicht nur Fragen, sie helfen auch bei der Weiterentwicklung von JSXGraph aktiv mit. Sie berichten von Fehlern und beheben diese teilweise auch eigenständig und leiten ihre Code-Änderungen (*patches*) weiter. Einige wenige implementieren neue Funktionen und stellen diese zur Verfügung. JSXGraph kann als Open Source Projekt zwar (noch) nicht von der Nutzergemeinschaft selbstständig getragen werden, die Veröffentlichung des Quelltextes hat sich jedoch als vorteilhaft für das Projekt erwiesen.

Auch wenn in JSXGraph bereits eine enorme Anzahl von Funktionen implementiert wurde, sind weitere wünschenswert und denkbar. Inhaltlich sind hier die Darstellung impliziter definierter Funktionen an oberster Stelle zu nennen. Die dreidimensionale Darstellung geometrischer Konstruktionen und Funktionsgraphen rückt mit der Verbreitung von WebGL in greifbare Nähe.

Da JSXGraph-Konstruktionen programmiert werden, ist das Problem des User Interfaces in diesem Bereich auf die Navigation reduziert. Aus technischer Sicht wäre eine bessere Anbindung an bereits existierende Autorensysteme wie LaTeX, Adobe InDesign und Apples iBooks Author von Vorteil. Die Integration von JSXGraph in Web-Plattformen wie Moodle, MediaWiki und Drupal wird bereits durch verschiedene Plugins unterstützt. Hier ist nicht nur eine Erweiterung der Unterstützung auf weitere Plattformen denkbar, sondern auch die Erweiterung auf JessieCode.

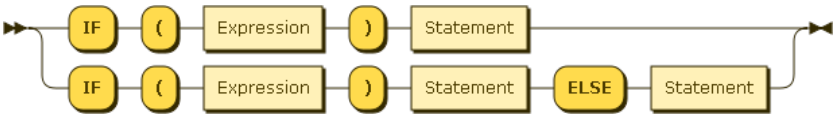
Anhang A

JessieCode Grammatik

Program

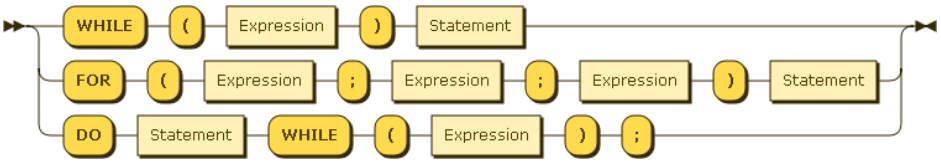

```
Program  ::= StatementList EOF
```

IfStatement



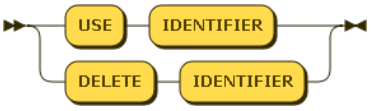

```
IfStatement
 ::= 'IF' '(' Expression ')' Statement
   | 'IF' '(' Expression ')' Statement 'ELSE' Statement
```

LoopStatement



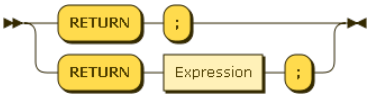

```
LoopStatement
 ::= 'WHILE' '(' Expression ')' Statement
   | 'FOR' '(' Expression ';' Expression ';' Expression ')'
      Statement
   | 'DO' Statement 'WHILE' '(' Expression ')' ';'
```

UnaryStatement



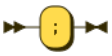

```
UnaryStatement
 ::= 'USE' 'IDENTIFIER'
   | 'DELETE' 'IDENTIFIER'
```

ReturnStatement




```
ReturnStatement
 ::= 'RETURN' ';'
   | 'RETURN' Expression ';'
```

EmptyStatement




```
EmptyStatement
 ::= ';'
```

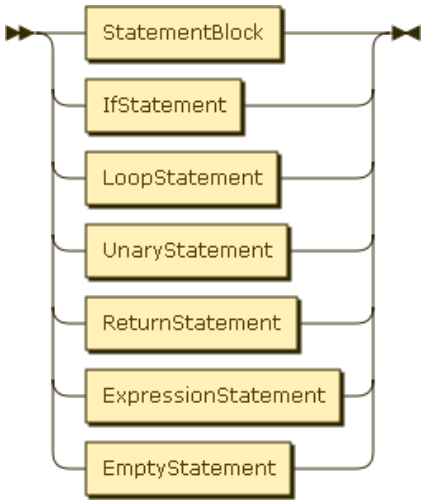
StatementBlock


```
StatementBlock
 ::= '{' StatementList '}'
```

StatementList


```
StatementList
 ::= Statement*
```

Statement




```
Statement
 ::= StatementBlock
   | IfStatement
   | LoopStatement
   | UnaryStatement
   | ReturnStatement
   | ExpressionStatement
   | EmptyStatement
```

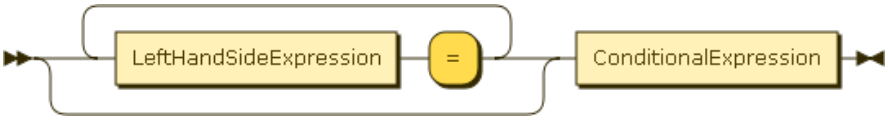
ExpressionStatement


```
ExpressionStatement
 ::= Expression ';'
```

Expression


```
Expression
 ::= AssignmentExpression
```

AssignmentExpression

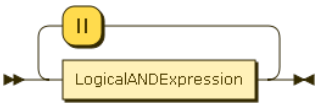



```
AssignmentExpression
 ::= ( LeftHandSideExpression '=' )* ConditionalExpression
```

ConditionalExpression


```
ConditionalExpression
 ::= LogicalORExpression
   | LogicalORExpression '?' AssignmentExpression ':' AssignmentExpression
```

LogicalORExpression

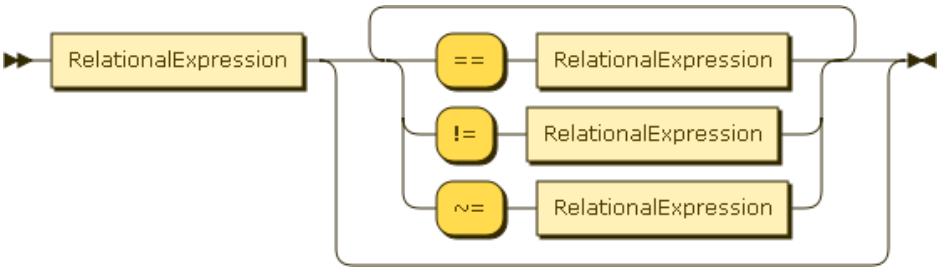



```
LogicalORExpression
 ::= LogicalANDExpression ( '||' LogicalANDExpression )*
```

LogicalANDExpression

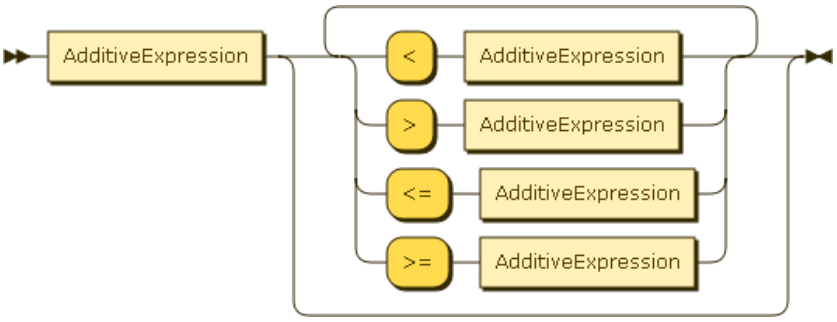

```
LogicalANDExpression
 ::= EqualityExpression ( '&&' EqualityExpression )*
```

EqualityExpression



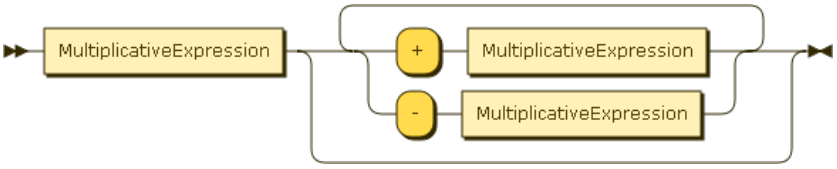

```
EqualityExpression
 ::= RelationalExpression
  ( '==' RelationalExpression |
    '!=' RelationalExpression |
    '~=' RelationalExpression )*
```

RelationalExpression



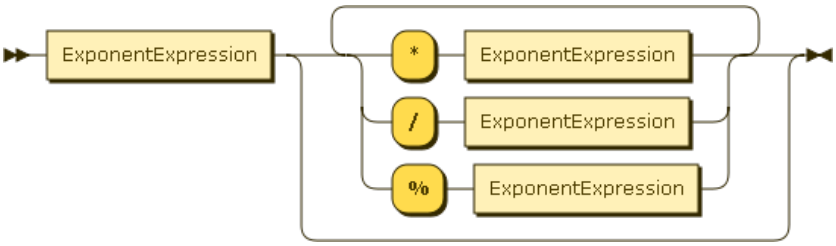

```
RelationalExpression
 ::= AdditiveExpression
  ( '<' AdditiveExpression |
    '>' AdditiveExpression |
    '<=' AdditiveExpression |
    '>=' AdditiveExpression )*
```

AdditiveExpression



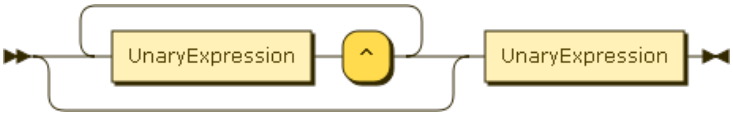

```
AdditiveExpression
 ::= MultiplicativeExpression
  ( '+' MultiplicativeExpression |
    '-' MultiplicativeExpression )*
```

MultiplicativeExpression



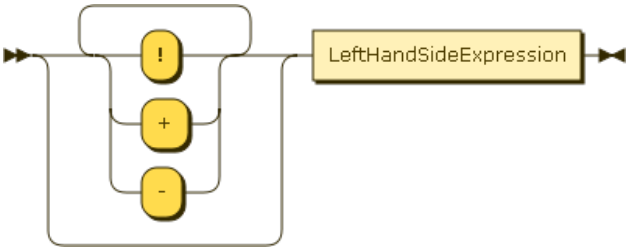

```
MultiplicativeExpression
 ::= ExponentExpression
  ( '*' ExponentExpression |
    '/' ExponentExpression |
    '%' ExponentExpression )*
```

ExponentExpression



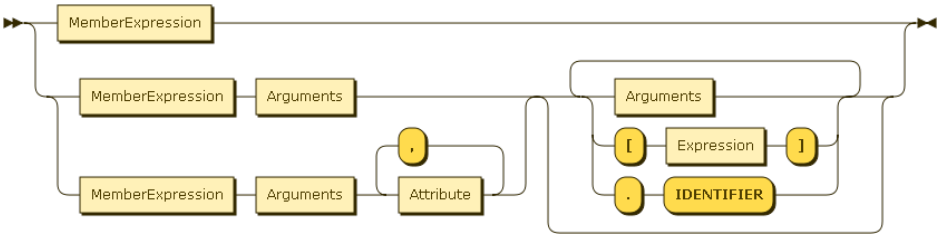

```
ExponentExpression
 ::= ( UnaryExpression '^' )* UnaryExpression
```

UnaryExpression



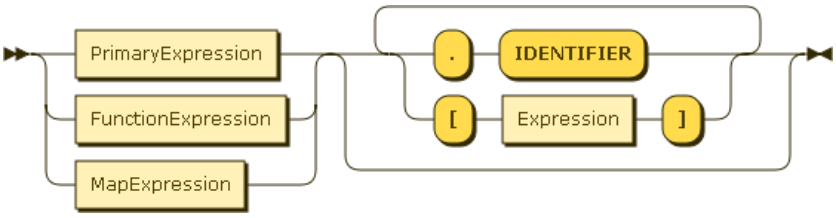
```
UnaryExpression
 ::= ( '!' | '+' | '-' )* LeftHandSideExpression
```

LeftHandSideExpression




```
LeftHandSideExpression
 ::= MemberExpression
   |
    ( MemberExpression Arguments |
      MemberExpression Arguments Attribute
       ( ',' Attribute )*
    )
    ( Arguments |
      '[' Expression ']' |
      '.' 'IDENTIFIER'
    )*
```

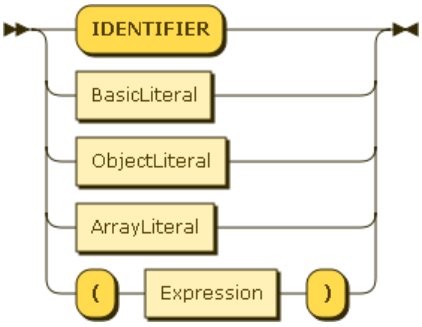
MemberExpression




```
MemberExpression
```

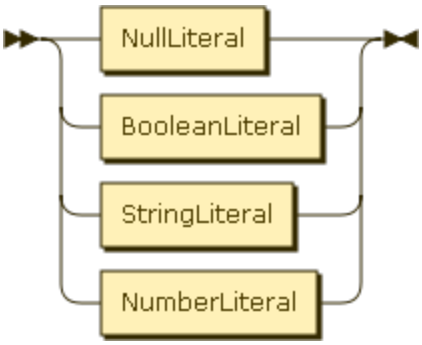
```
 ::= ( PrimaryExpression |
    FunctionExpression |
    MapExpression
  ) (
    '.' 'IDENTIFIER' |
    '[' Expression ']'
  )*
```

PrimaryExpression




```
PrimaryExpression
 ::= 'IDENTIFIER'
   | BasicLiteral
   | ObjectLiteral
   | ArrayLiteral
   | '(' Expression ')'
```

BasicLiteral

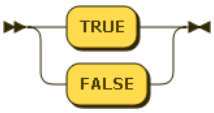



```
BasicLiteral
 ::= NullLiteral
   | BooleanLiteral
   | StringLiteral
   | NumberLiteral
```

NullLiteral


```
NullLiteral
 ::= 'NULL'
```

BooleanLiteral

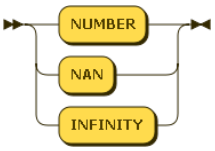



```
BooleanLiteral
 ::= 'TRUE'
   | 'FALSE'
```

StringLiteral

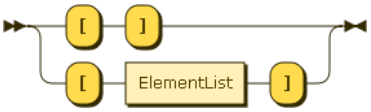

```
StringLiteral
 ::= 'STRING'
```

NumberLiteral



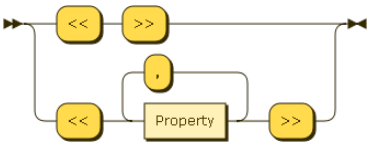

```
NumberLiteral
 ::= 'NUMBER'
   | 'NAN'
   | 'INFINITY'
```

ArrayLiteral



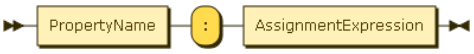

```
ArrayLiteral
 ::= '[' ']'
   | '[' ElementList ']'
```

ObjectLiteral



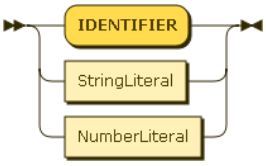

```
ObjectLiteral
 ::= '<<' '>>'
   | '<<' Property ( ',' Property )* '>>'
```

Property



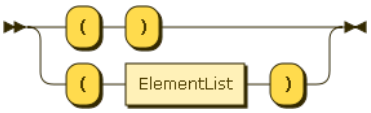

```
Property ::= PropertyName ':' AssignmentExpression
```

PropertyName




```
PropertyName
 ::= 'IDENTIFIER'
   | StringLiteral
   | NumberLiteral
```

Arguments




```
Arguments
 ::= '(' ')'
   | '(' ElementList ')'
```

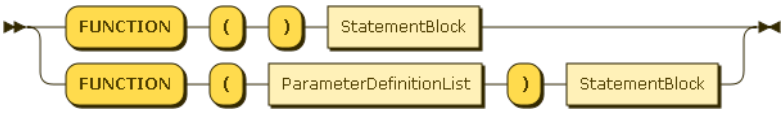
Attribute


```
Attribute
 ::= 'IDENTIFIER'
   | ObjectLiteral
```

ElementList


```
ElementList
 ::= AssignmentExpression ( ',' AssignmentExpression )*
```

FunctionExpression

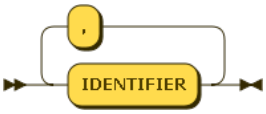



```
FunctionExpression
 ::= 'FUNCTION' '(' ')' StatementBlock
   | 'FUNCTION' '(' ParameterDefinitionList ')' StatementBlock
```

MapExpression


```
MapExpression
 ::= 'MAP' '(' ParameterDefinitionList ')' '->' Expression
```

ParameterDefinitionList




```
ParameterDefinitionList
 ::= 'IDENTIFIER' ( ',' 'IDENTIFIER' )*
```

Anhang B

Erkennungsregeln

B.1 triangle & slopetriangle

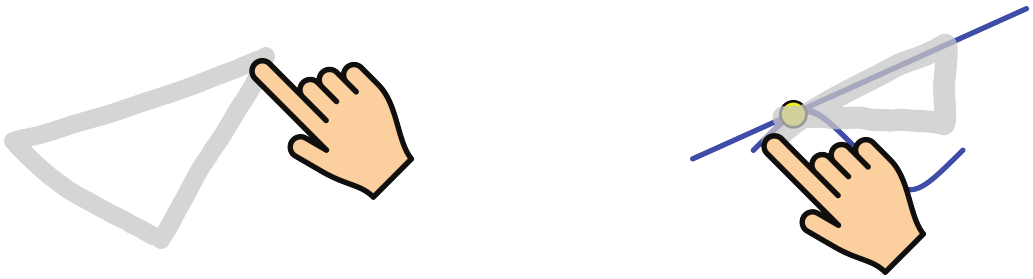


Abbildung B.1: *Zeichnen eines Dreiecks und eines Steigungsdreiecks. Beide Teilen sich die gleichen Grundregeln.*

Das Dreieck gehört zu den einfachsten Sketches. Es ist wichtig, dass alle Ecken als klarer Knick gezeichnet werden und nicht als Kreisbogen. Bei der Zeichnung können bis zu drei verschiedene Punkte überstrichen werden, die als Eckpunkte herangezogen werden. Unabhängig davon, wie viele Punkte eingesammelt wurden, wird, sobald die Dreieckskizze verifiziert wurde, überprüft, ob der Sketch zusätzlich die Regeln B.2 zum Steigungsdreieck erfüllt. Hierfür muss eine Seite eines Dreieck-Sketches mit einer Tangente einer Kurve oder eines Kreises übereinstimmen.

Listing B.1: ***Regelsatz zur Konstruktion eines Dreiecks***

```
new Assessor.Assessment(
    new ver.Equals(
        new val.NumberSketchElements('lines'),
        3
    )
),
new Assessor.Assessment(
    new ver.Collected('A')
),
new Assessor.Assessment(
    new ver.Collected('B'),
    new ver.NotIdentical('B', 'A')
),
new Assessor.Assessment(
    new ver.Collected('C'),
    new ver.NotIdentical('B', 'C'),
    new ver.NotIdentical('A', 'C')
)
```

Listing B.2: ***Regelsatz zur Konstruktion eines Steigungsdreiecks***

```
new Assessor.Assessment(
    new ver.Line('a'),
    new ver.Sketch('a'),
    new ver.Tangent('b'),
    new ver.MatchLines('a', 'b')
)
```

B.2 quadrilateral

Ähnlich dem Dreieck-Sketch wird ein Viereck nur gezeichnet. Auch hier müssen die Ecken hinreichend gut erkennbar gezeichnet werden.

Listing B.3: ***Regelsatz zur Konstruktion eines Vierecks***

```
new Assessor.Assessment(
    new ver.Equals(
        new val.NumberSketchElements('lines'),
        4
    )
```

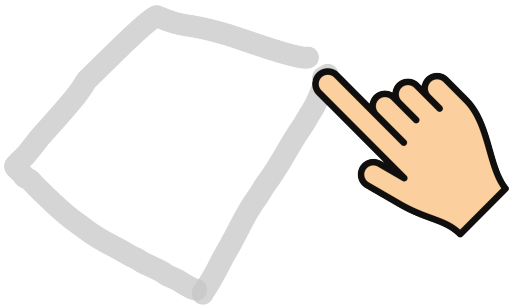


Abbildung B.2: *Sketch zum Erzeugen eines Vierecks.*

```
),
new Assessor.Assessment(
    new ver.Collected('A')
),
new Assessor.Assessment(
    new ver.Collected('B'),
    new ver.NotIdentical('B', 'A')
),
new Assessor.Assessment(
    new ver.Collected('C'),
    new ver.NotIdentical('B', 'C'),
    new ver.NotIdentical('A', 'C')
),
new Assessor.Assessment(
    new ver.Collected('D'),
    new ver.NotIdentical('B', 'D'),
    new ver.NotIdentical('A', 'D'),
    new ver.NotIdentical('C', 'D')
)
```

B.3 line

Die unterschiedlichen Formen einer Linie (Gerade, Halbgerade, Strecke) werden grundsätzlich identisch skizziert. Die Anzahl der überstrichenen Punkte und der Abstand dieser Punkte zum Beginn beziehungsweise Ende der Zeichnung sind ausschlaggebend welches Element schlussendlich erzeugt wird. Für den Fall, dass mehr als zwei Punkte während des Zeichnens eingesammelt wurden, sorgt der Score-Verifier dafür, dass das Punktepaar mit dem größ-

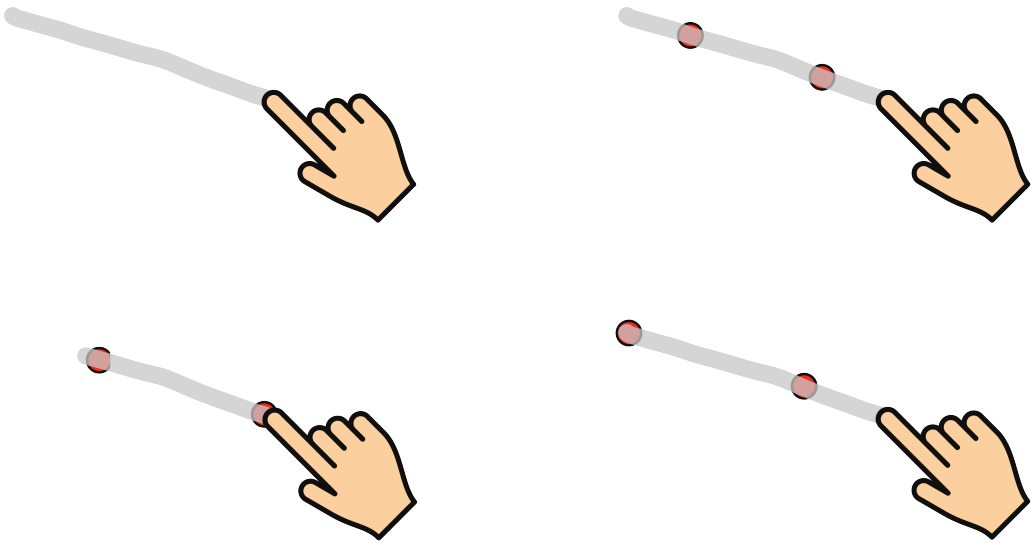


Abbildung B.3: *Die grundlegende Skizze ist bei allen Formen der Gerade gleich.*

ten Abstand zur Definition der Linie herangezogen werden. Siehe hierzu auch Abschnitt 5.2.3 über den Score-Verifier.

Listing B.4: ***Regelsatz zur Konstruktion einer Geraden***

```
new Assessor.Assessment(
    new ver.Equals(
        new val.NumberSketchElements('arcs'),
        0
    ),
    new ver.LEQ(
        new val.NumberSketchElements('lines'),
        2
    ),
    new ver.Sketch('a'),
    new ver.Line('a')
), new Assessor.Assessment(
    new ver.Collected('P'),
    new ver.Point('P')
), new Assessor.Assessment(
    new ver.Collected('Q'),
    new ver.Point('Q'),
    new ver.NotIdentical('P', 'Q'),
    new ver.Score(
        new val.Div(
            1,
```

```
            new val.Distance('P', 'Q')
        )
    )
)
```

B.4 midpoint

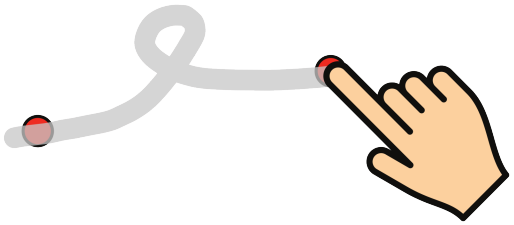


Abbildung B.4: *Skizzieren eines Mittelpunkts.*

Ein Mittelpunkt wird ähnlich wie eine Linie erzeugt. Es wird lediglich in der Mitte eine kleine Schleife ausgeführt. Diese Schleife wird als Kreisbogen erkannt, die sie umgebende gezeichnete Linie wird in der Erkennung daher aufgespalten in zwei Linien. Sind daher zwei gezeichnete Linien und ein gezeichneter Kreisbogen vorhanden und wurden bei der Zeichnung zwei verschiedene Punkte eingesammelt, ist der Mittelpunkt erkannt und kann zur Konstruktion hinzugefügt werden.

Listing B.5: ***Regelsatz zur Konstruktion eines Mittelpunkts***

```
new Assessor.Assessment(
    new ver.GEQ(new val.Sum([
        new val.NumberSketchElements('lines'),
        new val.NumberSketchElements('arcs')
    ]), 3),
    new ver.Collected('P'),
    new ver.Point('P'),
    new ver.Collected('Q'),
    new ver.Point('Q'),
    new ver.NotIdentical('P', 'Q')
)
```

B.5 circle

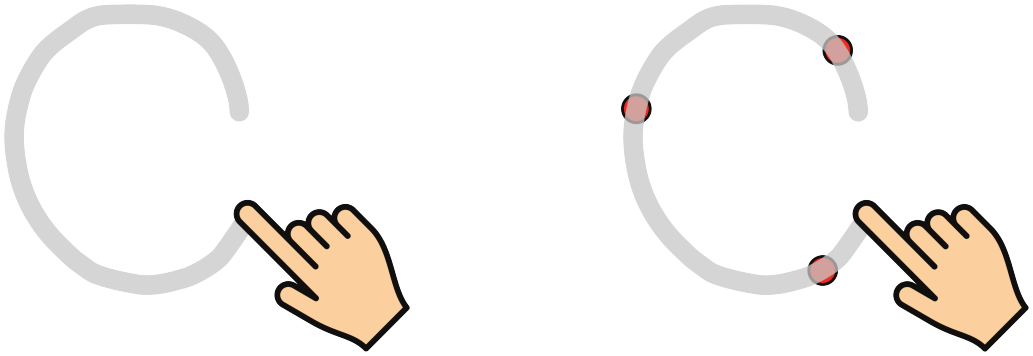


Abbildung B.5: *Kreisskizze.*

Da es nur wenige Sketches gibt, die Kreisbögen enthalten, genügt es hier zu prüfen, ob mehr Kreisbögen als Linien gezeichnet wurden. Nach jedem Teil-Assessment der Regeln in B.6 gilt der Kreis als erfolgreich erkannt und kann konstruiert werden. Wird nur das erste Teil-Assessment erfolgreich verifiziert, konstruiert die Sketch-Erkennung einen freien Kreis um einen neuen Mittelpunkt. Wenn auch das zweite Teil-Assessment verifiziert werden kann, wird ein Kreis um den zuerst und durch den zuletzt eingesammelten Punkt erzeugt. Kann das gesamte Assessment erfolgreich durchlaufen werden, dann wird ein Kreis durch drei Punkte erzeugt.

Listing B.6: ***Regelsatz zur Konstruktion eines Kreises***

```
new Assessor.Assessment(
    new ver.Greater(
        new val.NumberSketchElements('arcs'),
        new val.NumberSketchElements('lines')
    )
),
new Assessor.Assessment(
    new ver.Collected('A'),
    new ver.Point('A')
),
new Assessor.Assessment(
    new ver.Collected('B'),
    new ver.Point('B'),
    new ver.NotIdentical('A', 'B')
),
new Assessor.Assessment(
```

```
    new ver.Collected('C'),
    new ver.Point('C'),
    new ver.NotIdentical('C', 'A'),
    new ver.NotIdentical('C', 'B')
)
```

B.6 angle, sector & circle2points

Abbildung B.6: *Winkel, Sektoren und ein Kreis mit Mittel- und Radiuspunkt werden ähnlich skizziert.*

Die Elemente Winkel, Kreissektor und ein Kreis definiert durch Mittel- und Randpunkt werden sehr ähnlich skizziert und greifen daher auf die gleichen Regelsätze zurück. Für einen Kreis, definiert durch Mittel- und Randpunkt, genügt es, den Radius als Strecke einzuzeichnen und vom Randpunkt aus einen Kreisbogen anzudeuten. In der Regel genügt ein Viertel des Kreisbogens bereits. Dies verifizieren die ersten beiden Regeln. Kann auch die dritte noch verifiziert werden, wurde ein dritter Punkt beim Zeichnen eingesammelt und der Benutzer kann aus einem Auswahldialog zwischen Winkel und Kreissektor wählen.

Listing B.7: ***Regelsatz zur Konstruktion eines Kreissektors, Kreises durch zwei Punkte oder Winkels***

```
new Assessor.Assessment(
    new ver.Sketch('a'),
    new ver.Line('a'),
    new ver.Collected('A'),
    new ver.PointOnLine('a', 'A', 1.0),
    new ver.SketchArc('c')
),
new Assessor.Assessment(
```

```
    new ver.Collected('B'),
    new ver.PointOnLine('a', 'B', 1.0),
    new ver.NotIdentical('B', 'A')
),
// sector or angle
new Assessor.Assessment(
    new ver.Collected('C'),
    new ver.Point('C'),
    new ver.NotIdentical('C', 'A'),
    new ver.NotIdentical('C', 'B')
)
```

B.7 parallel

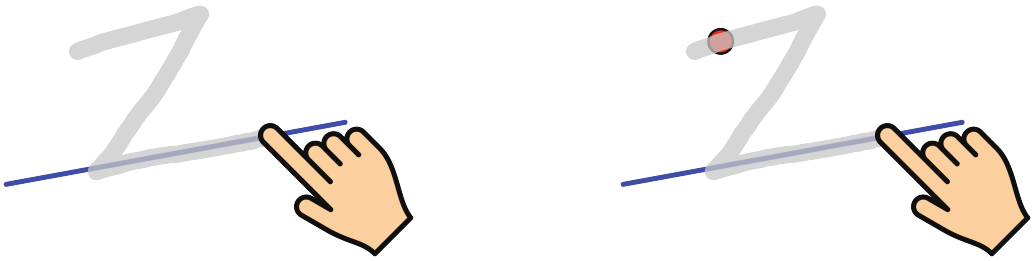


Abbildung B.7: *Sketch einer Parallele.*

Zum Erzeugen einer Parallelen muss diese eingezeichnet werden. Ohne Unterbrechung der Eingabe muss die Linie, zu der diese parallel sein soll, überstrichen werden. Zwangsläufig wird dadurch auch die Verbindungsstrecke mit eingezeichnet, die zur Sketch-Erkennung allerdings irrelevant ist. Der Sketch sieht dadurch aus wie ein Z. Es gilt daher zwei gezeichnete Strecken zu finden: Eine, die annähernd mit einer bereits existierende Strecke oder (Halb-)Gerade übereinstimmt und eine, die parallel verschoben ist.

Die Parallele kann auch optional an einem existierenden Punkt aufgehängt werden. Hierfür muss lediglich dieser Punkt überstrichen werden. Da dies optional ist, wurde dieser Test in ein Teil-Assessment ausgelagert, so dass eine Parallele auch dann erzeugt werden kann, wenn kein Punkt überstrichen wurde.

Listing B.8: ***Regelsatz zur Konstruktion einer Parallelen***

```
new Assessor.Assessment(
    new ver.Sketch('a'),
    new ver.Line('a'),
    new ver.MatchLines('a', 'c'),
    new ver.NotSketch('c'),
    new ver.Sketch('b'),
    new ver.Line('b'),
    new ver.NotIdentical('a', 'b'),
    new ver.Parallel('a', 'b', 0.2),
    new ver.PointOnSketchSegment('b', 'P')
),
new Assessor.Assessment(
    new ver.Collected('P')
)
```

B.8 normal

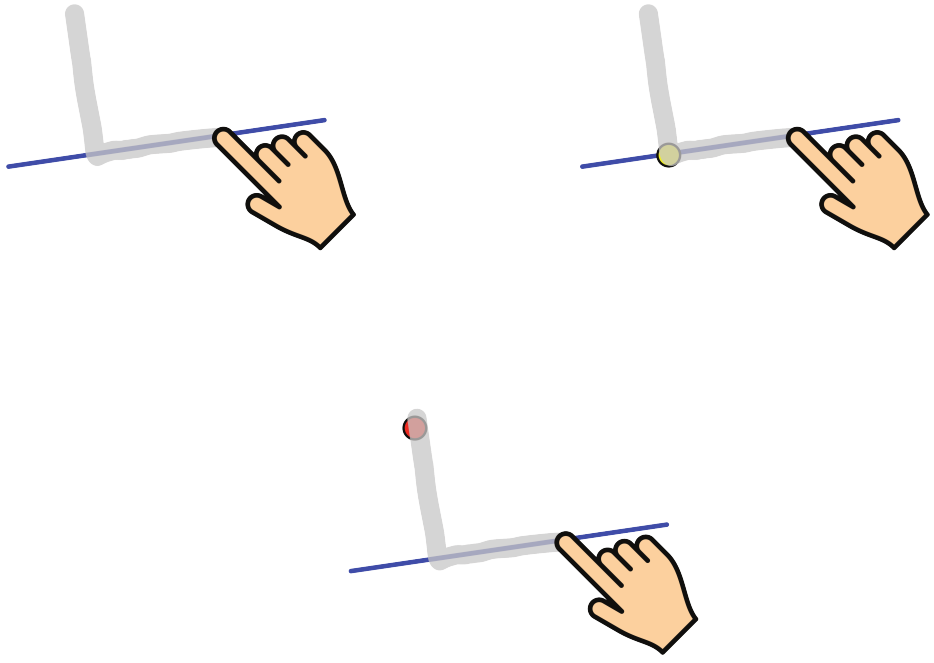


Abbildung B.8: *Sketches zur Erzeugung einer Normalen.*

Um eine Normale zu erzeugen, genügt es, diese einzuzeichnen und anschließend eine kurze Strecke auf einer bereits existierenden Gerade zu überstreichen. Optional kann ein Punkt dabei überstrichen werden, an dem die Normale "angehängt" werden soll. Werden mehrere Punkte überstrichen wählt die Sketch-Erkennung den ersten aus (zweites Teil-Assessment), es sei denn es gibt einen der nahe genug am Schnittpunkt der beiden gezeichneten Streckenabschnitte liegt (drittes Teil-Assessment).

Listing B.9: ***Regelsatz zur Konstruktion einer Normalen***

```
new Assessor.Assessment(
    new ver.Sketch('a'),
    new ver.Line('a'),
    new ver.MatchLines('a', 'c'),
    new ver.NotSketch('c'),
    new ver.Sketch('b'),
    new ver.Line('b'),
    new ver.Normal('a', 'b', 0.5),
    new ver.PointOnSketchSegment('b', 'P')
),
new Assessor.Assessment(
    new ver.Collected('P')
),
new Assessor.Assessment(
    new ver.PointOnLine('c', 'P')
)
```

B.9 bisector

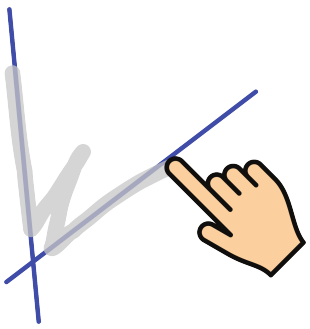


Abbildung B.9: *Skizzieren einer Winkelhalbierenden.*

Die Konstruktion einer Winkelhalbierenden geschieht durch Überstreichen zweier bereits konstruierter Geraden mit dazwischenliegendem Einzeichnen der Winkelhalbierenden. Der Sketch ähnelt daher einem W. Im Regelsatz B.10 wird lediglich überprüft, dass genau vier Strecken gezeichnet wurden und zwei der Strecken annähernd mit eingesammelten Geraden übereinstimmen. Mehr ist zur Verifikation und zum Bestimmen der Elternelemente auch nicht nötig. Die Steigung der Winkelhalbierenden wird erst im Konstruktionsschritt anhand der Steigungen der beiden übereinander liegenden Sketchstrecken bestimmt.

Listing B.10: ***Regelsatz zur Konstruktion einer Winkelhalbierenden***

```
new Assessor.Assessment(
    new ver.Equals(
        new val.NumberSketchElements('lines'),
        4
    ),
    new ver.Sketch('a'),
    new ver.MatchLines('a', 'x'),
    new ver.NotSketch('x'),
    new ver.Sketch('d'),
    new ver.MatchLines('d', 'y'),
    new ver.NotSketch('y'),
    new ver.NotIdentical('x', 'y')
)
```

B.10 tangent

Abbildung B.10: *Skizzieren einer Tangente.*

Um unserer Grundprämisse gerecht zu werden, müsste anstelle eines Lotes zur angezeichneten Tangente, der Kreis oder die Kurve, an die die Tangente anliegen soll, überstrichen werden. Diese Berechnung kann bei Kurven

unter Umständen sehr aufwändig sein. Deshalb wurde hier eine alternative Eigenschaft der Kreistangente gewählt: die Orthogonalität zum Radius. Das zweite Teil-Assessment dient zum Auswählen eines eventuell überstrichenen Gleiters. Ist dieser nicht vorhanden wird ein Gleiter erzeugt.

Listing B.11: ***Regelsatz zur Konstruktion einer Tangenten***

```
new Assessor.Assessment(
    new ver.Sketch('a'),
    new ver.Sketch('b'),
    new ver.MatchLines('a', 'b'),
    new ver.Sketch('c'),
    new ver.Normal('a', 'c'),
    new ver.Normal('b', 'c'),
    new ver.PointOnLine('c', 'A'),
    new ver.PointOnLine('b', 'A'),
    new ver.OnCircleCurve('A', 'k', 2)
),
new Assessor.Assessment(
    new ver.Collected('A')
)
```

B.11 reflection

Die Spiegelung ist in der derzeitigen Konfiguration problematisch. Zum einen stimmt die Form nicht mit dem Grundkonzept überein und zum anderen ist der gewählte Sketch in Pfeilform der Winkelhalbierenden zu ähnlich. Bis eine Alternative gefunden wird muss daher nahe des zu spiegelnden Punktes mit der Zeichnung begonnen werden eine Gerade zu zeichnen. An der als Spiegelachse dienenden Linie wird eine Pfeilspitze eingezeichnet, so dass die Spitze möglichst genau auf der Linie liegt.

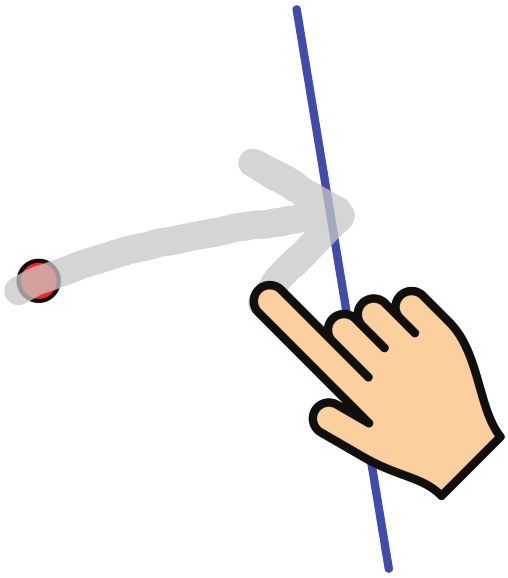


Abbildung B.11: *Spiegeln eines Punktes an einer Geraden.*

Listing B.12: ***Regelsatz zur Konstruktion einer Achsenspiegelung***

```
new Assessor.Assessment(
    new ver.Sketch('a'),
    new ver.PointOnLine('a', 'B'),
    new ver.Collected('B'),
    new ver.Sketch('A'),
    new ver.PointOnLine('a', 'A'),
    new ver.PointOnLine('x', 'A'),
    new ver.NotSketch('x')
)
```

Anhang C

CD-ROM

Auf dem beigelegten Datenträger sind Momentaufnahmen der Projektarchive von JSXGraph, practice und JessieCode in den jeweiligen Unterverzeichnissen zu finden. Bei den Momentaufnahmen handelt es sich um den Stand des master-Zweiges der jeweiligen git-Repositories. Um den aktuellen Stand zu betrachten, sei an dieser Stelle auf die entsprechenden Repositories auf GitHub verwiesen:

- JSXGraph: https://github.com/jsxgraph/jsxgraph
- JessieCode: https://github.com/jsxgraph/JessieCode
- practice: https://github.com/jsxgraph/practice

Die angegebenen Webseiten wurden im März 2014 abgerufen. Gleiches gilt für alle weiteren Web-Adressen in diesem Abschnitt.

Der Quelltext der Implementierung des Sketch-Erkennungsalgorithmus ist im Unterverzeichnis *Sketch-Erkennung* zu finden.

Aufgrund der Lizenz ist sketchometry nur in minimierter Fassung im Unterordner *sketchometry* verfügbar. Diese darf weitergereicht werden, allerdings ausschließlich in unveränderter Form. Auf der Webseite des Projekts ist eine aktuelle Version erhältlich: http://www.sketchometry.org.

Die Quellen des SketchBin-Projekts und der Umgebung zur Evaluation der Sketch-Erkennung sind in den Unterordnern *sketchbin* beziehungsweise *Evaluation/Testumgebung* abgelegt. Beide Projekte nutzen zahlreiche externe Projekte wie JSXGraph oder sketchometry. Diese liegen in den entsprechenden Projekten meist nur in minimierter Form vor. Verwendet wurden:

- ACE, BSD Lizenz, http://ace.c9.io/
- colorpicker, MIT oder GPL Lizenz, http://www.eyecon.ro/colorpicker/
- jquery, MIT Lizenz, http://jquery.com/
- markdown, MIT Lizenz, https://github.com/evilstreak/markdown-js
- express, MIT Lizenz, http://expressjs.com/
- jade, MIT Lizenz, http://jade-lang.com/
- node-orm2, MIT Lizenz, https://github.com/dresende/node-orm2
- node-mysql, MIT Lizenz, https://github.com/felixge/node-mysql
- jszip, MIT oder GPLv3 Lizenz, http://stuk.github.io/jszip/
- requirejs, "New" BSD oder MIT Lizenz, http://www.requirejs.org/
- mustache, MIT Lizenz, https://github.com/janl/mustache.js

Im Unterverzeichnis *Evaluation* liegen die Rohdaten im gleichnamigen Ordner. Diese liegen in drei Text-Dateien für die drei teilnehmenden Jahrgangsstufen. Jede nicht-leere Zeile entspricht einer Abgabe, die erst gzip-komprimiert und anschließend base64-codiert wurde. Die Datei *Ergebnisse.sqlite* ist eine SQLite Datenbank, in der die Ergebnisse der Auswertung gespeichert sind. Sie beinhaltet eine einzige Tabelle `skmeval` deren Struktur in Listing C.1 beschrieben ist.

Listing C.1: *Struktur der Tabelle* skmeval

```
CREATE TABLE skmeval (
    id INTEGER PRIMARY KEY,
    userid TEXT,
    experience INTEGER,
    device INTEGER,
    task TEXT,
```

```
    result TEXT,
    class TEXT
);
```

Der Unterordner *Report* beinhaltet ein Python-Skript zur Auswertung der Daten in *Ergebnisse.sqlite*. Die Auswertung wird im HTML-Format ausgegeben.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: principles and techniques and tools*. Addison-Wesley Longman, Amsterdam, 1985.

[2] Lisa Anthony and Jacob O Wobbrock. A lightweight multistroke recognizer for user interface prototypes. In *Proceedings of Graphics Interface 2010*, pages 245–252. Canadian Information Processing Society, 2010.

[3] Lisa Anthony and Jacob O Wobbrock. $n-protractor: a fast and accurate multistroke recognizer. In *Proceedings of the 2012 Graphics Interace Conference*, pages 117–120. Canadian Information Processing Society, 2012.

[4] T. Becker, V. Weispfenning, and H. Kredel. *Gröbner bases: a computational approach to commutative algebra*. Graduate texts in mathematics. Springer-Verlag, 1993.

[5] Francisco Botana. A web-based intelligent system for geometric discovery. In *Proceedings of the 1st International Conference on Computational Science: PartI*, ICCS'03, pages 801–810, Berlin, Heidelberg, 2003. Springer-Verlag.

[6] Francisco Botana, Miguel A. Abánades, and Jesús Escribano. Computing locus equations for standard dynamic geometry environments. In *Proceedings of the 7th International Conference on Computational Science, Part II*, ICCS '07, pages 227–234, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Zach Carter. Jison. https://github.com/zaach/jison, July 2013.

[8] D.A. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Number v. 10. Springer, 2007.

[9] David H. Douglas and Thomas K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10:112–122, 1973.

[10] Matthias Ehmann. *GEONExT: Integration algebraischer Objekte und Internationalisierung*. PhD thesis, Universität Bayreuth, 2004.

[11] R. Fröberg. *An Introduction to Gröbner Bases*. Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts. Wiley, 1997.

[12] Michael Gerhäuser. Computergestützte Berechnung geometrischer Orte mittels Gröbnerbasen. Diplomarbeit, Universität Bayreuth, 2009.

[13] Michael Gerhäuser and Alfred Wassermann. Automatic calculation of plane loci using gröbner bases and integration into a dynamic geometry system. In *Automated Deduction in Geometry*, pages 68–77, 2010.

[14] Google. Caja. https://developers.google.com/caja/, July 2013.

[15] HTML Working Group. Html. http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html, March 2014.

[16] SVG Working Group. Scalable vector graphics (svg) 1.1. http://www.w3.org/TR/SVG/, March 2014.

[17] Internationales Geogebra Institut. Geogebra manual. http://wiki.geogebra.org/en/Manual:Main_Page, October 2013.

[18] ECMA International. *Standard ECMA-262 ECMAScript Language Specification*, 5.1 edition, June 2011.

[19] Alan C. Kay. A personal computer for children of all ages. In *Proceedings of the ACM Annual Conference - Volume 1*, ACM '72, New York, NY, USA, 1972. ACM.

[20] Ulrich Kortenkamp. *Foundations of Dynamic Geometry*. PhD thesis, 1999.

[21] Peter Lebmeir and Jürgen Richter-Gebert. *Recognition of Computationally Constructed Loci*. 2006.

[22] Yang Li. Protractor: a fast and accurate gesture recognizer. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2169–2172, New York, NY, USA, 2010. ACM.

[23] Carsten Miller. *Entwicklung dynamischer Arbeitsblätter und ihre Realisierung für den Mathematikunterricht.* PhD thesis, Universität Bayreuth, 2004.

[24] Peter Osmon. Tablets are coming to a school near you. In *Proceedings of the British Society for Research into Learning Mathematics*, pages 115–120, 2011.

[25] Sharon Oviatt, Alex Arthur, and Julia Cohen. Quiet interfaces that help students think. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, UIST '06, pages 191–200, New York, NY, USA, 2006. ACM.

[26] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244 – 256, 1972.

[27] Tomás Recio and M. P. Vélez. Automatic discovery of theorems in elementary geometry. *J. Autom. Reasoning*, 23(1):63–82, 1999.

[28] Jürgen Richter-Gebert and Ulrich H. Kortenkamp. The power of scripting: DGS meets programming. *Acta Didactica Napocensia*, 3(2):67–78, 2010.

[29] Jürgen Richter-Gebert and Ulrich H. Kortenkamp. *The Cinderella.2 Manual – Working with the Interactive Geometry Software.* Springer-Verlag, 2012.

[30] Robert Sedgewick. *Algorithms.* Addison-Wesley, 2nd edition, April 1988.

[31] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[32] Dirk Materlik Ulrich Kortenkamp. Pen-based input of geometric constructions. 2004.

[33] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O Wobbrock. Gestures as point clouds: a $p recognizer for user interface prototypes. In *Proceedings of the 14th ACM international conference on Multimodal interaction*, pages 273–280. ACM, 2012.

[34] D. Wang. *Elimination Methods (Texts and Monographs in Symbolic Computation).* Springer, 2001.

[35] Jacob O Wobbrock, Andrew D Wilson, and Yang Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 159–168. ACM, 2007.

[36] A. Wolin, B. Eoff, and T. Hammond. Shortstraw: a simple and effective corner finder for polylines. In *Proceedings of the Fifth Eurographics conference on Sketch-Based Interfaces and Modeling*, SBM'08, pages 33–40, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[37] Yiyan Xiong and Joseph J. LaViola, Jr. Revisiting shortstraw: improving corner finding in sketch-based interfaces. In *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling*, SBIM '09, pages 101–108, New York, NY, USA, 2009. ACM.