SAGA

Entwurf, Funktionsumfang und Anwendung eines

Systems für Automatisierte Geowissenschaftliche Analysen

Dissertation

zur Erlangung des Doktorgrades

der Mathematisch-Naturwissenschaftlichen Fakultäten

der Georg-August-Universität zu Göttingen

vorgelegt von

Olaf Conrad

aus Dannenberg (Elbe)

Göttingen 2006

D7

Referent: Prof. Dr. Jürgen Böhner

Koreferent: Prof. Dr. Karl-Heinz Pörtge

Tag der mündlichen Prüfung: 6. November 2006

für

Jascha, Mattis & Jannes

„Every decision a person makes stems from the person's values and goals. People can have many different goals and values; fame, profit, love, survival, fun, and freedom, are just some of the goals that a good person might have. When the goal is to help others as well as oneself, we call that idealism. My work on free software is motivated by an idealistic goal: spreading freedom and cooperation. I want to encourage free software to spread, replacing proprietary software that forbids cooperation, and thus make our society better.“

R.Stallmann, 1998,
Copyleft: Pragmatic Idealism
(http://www.gnu.org/philosophy/pragmatic.html)

„Das sind ganz normale Leute, die nach ihrer ganz normalen 60-Stunden-Woche gerne auch noch mal 20, 30 Stunden etwas machen, was Spaß macht. Ich glaube, das entscheidende Kriterium für Leute, die sehr produktiv sind und viel in solchen Projekten arbeiten, ist einfach die Faszination am Projekt. Und diese Faszination bedeutet auch sehr oft, dass man sich abends um sieben hinsetzt, um noch schnell ein kleines Problem zu lösen und auf die Uhr schaut, wenn es vier Uhr früh ist.“

D.Hohndel, Berlin, 1999,
Wizards of OS, Konferenz über Offene Quellen und Freie Software
(nach GRASSMUCK 2002)

VORWORT

Anders als die Mehrzahl wissenschaftlicher Abschlußarbeiten, bezieht sich die vorliegende Dissertation nicht auf ein einzelnes Forschungsvorhaben, sondern vielmehr auf ein Nebenprodukt einer Vielzahl von Projekten aus Forschung, Entwicklung und Lehre, die seit Ende der 1990er Jahre im Umfeld des Geographischen Institutes der Universität Göttingen angesiedelt waren und an deren Durchführung ich beteiligt war. Dieses Produkt, SAGA, hat sich während dieser Zeit von dem Werkzeug einer kleinen Arbeitsgruppe zu einem umfangreichen Softwaresystem für die Verarbeitung von Geodaten entwickelt, das seit seiner Veröffentlichung als Free Open Source Software weltweite Verbreitung gefunden hat.

Nicht zuletzt um auf die enge Verflechtung der SAGA Entwicklung mit verschiedenen Forschungsprojekten aufmerksam zu machen, wurde der Hauptteil dieser Arbeit, der Entwicklung, Aufbau und Funktionsumfang des Systems aus verschiedenen Blickwinkeln beleuchtet, um eigene Publikationen aus dem Veröffentlichungszeitraum 2001 bis 2006 ergänzt. Auf diese Weise vereinigt diese Arbeit sehr viele Aspekte in sich, von der Softwareentwicklung über die Forschungsarbeit bis hin zum Einsatz in der Lehre, und wäre in dieser Form ohne direkte oder indirekte Zuarbeiten und Hilfestellungen einer Vielzahl von KollegInnen und FreundInnen nicht denkbar gewesen. Wohlwissend, dass ich nicht jeden Einzelnen an dieser Stelle angemessen erwähnen kann, möchte ich vorab schon einmal allen meinen herzlichen Dank aussprechen, die ich im Nachfolgenden vergessen habe.

Besonderer Dank gebührt vor allem meinem langjährigen Kollegen Prof. Dr. Jürgen Böhner, der diese Arbeit von ihren ersten Anfängen an mitbegleitet hat und bei dem ich immer neben fachlicher auch persönliche Unterstützung gefunden habe. Ein großer Dank geht an Prof. Dr. Karl-Heinz Pörtge für die Übernahme des Koreferats sowie zahlreichen kritischen Diskussionen. Bedanken möchte ich mich an dieser Stelle auch bei Prof. Dr. Jürgen Hagedorn, der durch die seinerzeit mutige und vorrausschauende Förderung computergestützter Methoden für die Reliefanalyse nicht nur einen wichtigen Grundstein für die spätere Arbeit der Arbeitsgruppe gelegt hat, sondern auch mir persönlich eine entsprechend ausgerichtete Diplomarbeit erst ermöglichte.

Von den Mitgliedern der Göttinger Arbeitsgruppe sei vor allem Rüdiger Köthe gedankt, der die SAGA Entwicklung von Anfang an nicht nur in fachlichen Fragen begleitet hat. In technischen Fragen waren besonders in der kritischen Anfangsphase die Anregungen von Andre Ringler eine unabkömmliche Hilfe. Aber auch alle anderen, oft nur kurzzeitig involvierten Mitglieder der Arbeitsgruppe haben, sei es durch Diskussionen, als Testnutzer oder durch das Schreiben von Dokumentationen, ihren individuellen Beitrag geliefert. Stellvertretend genannt seien Michael Bock, Dr. Jens Gross, Frank Haselein, Gabriele Kehr, Angela Kreikemeyer, Thomas Schmeja, Dr. Thomas Selige und Anke Wehmeyer.

Mit der Quelltextfreigabe haben sich zunehmend Beitragende ausserhalb der Arbeitsgruppe um die Weiterentwicklung von SAGA verdient gemacht. Einen Meilenstein stellen die Beiträge von Victor Olaya dar, dessen Handbuch einen Multiplikatoreffekt für die Popularität

von SAGA mit sich brachte. Keinesfalls unerwähnt bleiben dürfen an dieser Stelle Vern Cimmery, Stefan Liersch, Dr. Thomas Schorr und Dr. Volker Wichmann, die sich seitdem durch viele unverzichtbare Zuarbeiten eingebracht haben.

Abschließend gilt mein Dank der noch kleinen aber wachsenden Zahl der Mitglieder der SAGA User Group e.V. sowie den zahlreichen kritischen wie positiven Rückmeldungen aus der Gemeinde der SAGA Nutzer.

Ich verbinde mit dieser Arbeit die Hoffnung, dass viele Anwender auf die in SAGA implementierten Methoden aufmerksam werden und sie für ihre eigenen Forschungen in Wert setzen können, und würde mich besonders freuen, wenn der eine oder andere SAGA Nutzer, sich in Zukunft dafür entscheidet, SAGA als Plattform für die Entwicklung neuer Methoden zu wählen und sich vielleicht sogar an der Weiterentwicklung des Systems selbst beteiligt.

Göttingen, November 2006 Olaf Conrad

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ABKÜRZUNGEN

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CMD	Kommandozeileninterpreter von SAGA
DBMS	Datenbank-Managementsystem
DGM	Digitales Geländemodell
DLL	Dynamic Link Library
DOS	Disk Operating System
FOSS	Free Open Source Software
GCC	GNU Compiler Collection
GDAL	Geospatial Data Abstraction Library
GIS	Geographisches Informationssystem
GPL	GNU General Public License
GPS	Global Positioning System
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
LGPL	GNU Lesser General Public License
MFC	Microsoft Foundation Class Library
MinGW	Minimalist GCC for Windows
OS	Operating System
OSS	Open Source Software
PC	Personal Computer
PDF	Portable Document Format
SO	Shared Object
SQL	Structured Query Language
SRTM	Shuttle Radar Topography Mission
SVG	Scalable Vector Graphics
TIN	Triangulated Irregular Network
TWI	Topographic Wetness Index
USLE	Universal Soil Loss Equation
VC	Microsoft Visual C++

1 EINFÜHRUNG

Wie nahezu alle Wissenschaften, haben auch die geographischen Wissenschaften von der Computerrevolution der letzten drei Jahrzehnte des 20. Jahrhunderts profitiert. Die technischen Möglichkeiten, immer größer werdende Datenmengen erheben und speichern zu können, eröffnen seitdem der Geographie neue, ungeahnte Möglichkeiten, die aber gleichzeitig auch eine methodische Herausforderung darstellen. Die Nutzung der Computertechnologie für geographische Aufgaben und Fragestellungen, von der Grundlagenforschung bis hin zur Anwendung, führte zur Etablierung einer neuen Teildisziplin, der Geoinformatik, die sich mit Computermethoden für die Verwaltung und Analyse raumbezogener Daten auseinandersetzt. Die Entwicklung geoinformatischer Methoden erhielt ihre Impulse dabei aus verschiedenen Richtungen. Von grundlegender Bedeutung ist die Verfügbarkeit von Computersystemen selbst, die erst allmählich durch das Aufkommen von Personal Computern (PC) seit der Mitte der 1980er Jahre ein breiteres Anwendungsspektrum in den Geowissenschaften ermöglichte. Einher ging die Entwicklung der Betriebssysteme von kommandozeilengesteuerten Textkonsolen zu graphisch orientierten Oberflächen. Mittlerweile werden hoch performante PC-Systeme praktisch flächendeckend eingesetzt und sind ein unverzichtbares Werkzeug in Forschung und Lehre geworden. Um ein Computersystem aber sinnvoll einsetzen zu können, wird auf der anderen Seite eine Software benötigt, mit der sich das System bedienen lässt und benutzerdefinierte Aufgaben durchführen lassen. Die Entwicklung von Software verlangt jedoch ein gewisses Maß an Wissen über die Funktionsweise und Programmierung von Computern, das durch die Kernausbildung des Geographiestudiums nicht vermittelt wird. In der Folge muss für die Durchführung einer geographischen Datenanalyse auf Methoden zurückgegriffen werden, die von mehr oder weniger fachfremden Programmierern entwickelt wurden. Die Herausbildung spezieller Software für die Verarbeitung räumlicher Daten, die u.a. durch die Verbreitung der PC-Systeme begünstigt wurde und allgemein als GIS-Software bezeichnet wird, gleicht dieses Defizit durch die Bereitstellung breit gefächerter, allgemeiner Methoden wie auch leicht erlernbarer Skriptsprachen, mit denen komplexere Verarbeitungsschritte aus allgemeineren Methoden zusammengesetzt werden können, teilweise aus. Gerade aber die Entwicklung gänzlich neuer Methoden, deren Integration in die Infrastruktur bestehender GIS-Software sowie der darauf folgende operationelle Einsatz gestaltet sich nach wie vor schwierig. Und auch die verfügbaren Methodensammlungen haben ihre Grenzen. So lässt sich bei kommerzieller Software die korrekte Arbeitsweise der angebotenen Methoden nur an Hand der Ergebnisse überprüfen, die sie liefern. Eine Modifizierung der zu Grunde liegenden Algorithmen ist hier von vornherein ausgeschlossen. Selbst bei quelltextoffener Software sind die Algorithmen auch mit guten Programmierkenntnissen oft nur sehr schwer nachvollziehbar. Zudem stellt quelltextoffene Software den Anwender meist vor andere Hürden, z.B. dadurch, dass sie sehr spezialisiert ist, nur auf weniger verbreiteten Betriebssystemen läuft oder eine umständliche Benutzerführung hat.

Das System für Automatisierte Geowissenschaftliche Analysen (SAGA), das im Zentrum dieser Arbeit steht, setzt genau an den zuletzt genannten Punkten an. SAGA ist ein quelltext-offenes Softwaresystem für die Bearbeitung, Analyse und Modellierung raumbezogener Daten, verfügt über eine intuitiv bedienbare Benutzeroberfläche und kann unter verschiedenen Betriebssystemen ausgeführt werden. Das Fundament des Systems ist aber eine Schnittstelle, die speziell für die objektorientierte Programmierung von Methoden zur Verarbeitung raumbezogener Daten entwickelt wurde und dadurch zu einer schnell wachsenden Zahl der in SAGA implementierten Methoden beiträgt. Die Motivation für die Entwicklung von SAGA wurzelt in der Arbeit an verschiedenen Projekten der von J.Böhner geleiteten Arbeitsgruppe Geosystemanalyse am Geographischen Institut der Universität Göttingen.

1.1 Motivation

Um die methodische und fachliche Kompetenz für die Durchführung verschiedener Projekte im Bereich Klima, Relief, Wasser und Boden zu bündeln, schlossen sich 1998 Mitarbeiter der Abteilung für Physische Geographie der Universität Göttingen als Vertreter der Forschung sowie der Scilands GmbH als Dienstleistungsunternehmen zur AG Geosystemanalyse zusammen. Ziel dieser Kombination von Forschung und Dienstleistung ist es seitdem, die in Forschungsprojekten entwickelten Methoden auf kürzestem Weg in die Anwendung zu bringen und so auch die Attraktivität für das Einwerben von Drittmitteln durch die Arbeitsgruppe zu steigern. Für die wissenschaftliche Leitung zeichnet sich J.Böhner, Abteilung für Physische Geographie, verantwortlich, während die Scilands GmbH durch R.Köthe vertreten ist, der selbst bis Ende der 1990er Jahre in der Abteilung für Physische Geographie tätig war.

Ein Forschungsschwerpunkt, der eine lange Tradition in der Abteilung aufweist, ist die computergestützte Analyse des Reliefs. Ein erster Meilenstein wurde mit der Entwicklung des Systems zur Automatischen Reliefanalyse (SARA) gesetzt (KÖTHE & LEHMEIER 1993, KÖTHE et al. 1996). SARA arbeitet mit rasterbasierten Digitalen Geländemodellen (DGM) und leitet von diesen morphometrische Reliefparameter ab. Die Reliefparameter werden u.a. dazu benutzt, um weitergehende Reliefgliederungen vorzunehmen. Ein wesentliches Ziel dieser Reliefgliederungen, das insbesondere durch die Kooperation mit dem Niedersächsischen Landesamt für Bodenforschung (NLfB) und der Bundesanstalt für Geowissenschaften und Rohstoffe (BGR) gefördert wurde, war die Unterstützung der Bodenkartierung. Mitte der 1990er Jahre entstand in paralleler Entwicklung das System zur Analyse und Diskretisierung von Oberflächen (SADO), das die Fähigkeiten von SARA zunächst vor allem um statistische Verfahren ergänzte (BÖHNER et al. 1997). Mit dem Programm DiGeM (Abb. 1, CONRAD 2002) ist eine dritte Softwareentwicklung zu nennen, die im gleichen Zeitraum im Umfeld von SARA entstand und mit seinem Schwerpunkt vor allem auf die Ableitung hydrologisch relevanter Reliefparameter ausgerichtet ist (CONRAD 1998).

Tab. 1: Projekte der AG Geosystemanalyse seit 1999 (nach BÖHNER 2006)

Zeitraum	Projekt	Träger	Durchführung
1999–2001	WEELS – Wind Erosion on European Light Soils	EU	Univ. College London (GB), Univ. Wageningen (NL), Univ. Lund (S), NLfB Bremen, Univ. Göttingen
1999–2001	Systemanalytische Erfassung und Rekonstruktion des jungquartären Klima- und Landschaftswandels in Zentralasiatischen Trockengebieten am Beispiel des Ordos-Plateaus (VR China)	DFG	Univ. Göttingen, Univ. Erfurt, Univ. Xi'an (VR-China)
1998–2001	Regionalisierung der rezenten, vorzeitlichen und potentiell-zukünftigen Klimaverhältnisse Zentralasiens – Modellierung auf Basis von geomorphologisch-paläoökologischen Befunden, direkten Klimadaten und GCM-Simulationen	BMBF/DLR	Univ. Göttingen, Univ. Aachen
1999–2000	Reliefanalyse und Klimaregionalisierung	BGR Berlin	Univ. Göttingen
1999–2002	GEOS – Validierung und Kalibrierung von Einflüssen durch bio- und geophysikalische Eigenschaften des Georeliefs auf das SRTM-Höhenmodell und Beurteilung seiner Eignung für geowissenschaftliche Anwendungen	BMBF/DLR	TU München, Univ. Göttingen, scilands GmbH Göttingen
2000–2003	Geoökologische Datenanalyse und Segmentierung von Satellitendaten	TU München	TU München, Univ. Göttingen
2001–2003	Handlungsempfehlungen zur effizienten, umweltverträglichen Planung von Windenergieanlagen für den norddeutschen Raum – Entwicklung von Methoden zur Landschaftsbildanalyse auf Basis segmentierter Satellitendaten (IRS LISS III)	DBU, Döpel Landschafts-planung	Univ. Göttingen
2001–2003	Entwicklung von Verfahren zur Rauhigkeitsparametrisierung auf Basis von Satellitendaten (Landsat TM) und Modellierung der Windenergieressourcen ausgewählter Standorte in Niedersachsen, Thüringen und Sachsen	Meridian GmbH	Univ. Göttingen
2001–2003	Schlagbezogene Modellierung von Bodenabtrag durch Wassererosion	NLfB Bremen	Univ. Göttingen
2002–2003	Reliefanalyse, Klimaregionalisierung und Bodenregionalisierung	BGR Berlin	Univ. Göttingen
2002–2004	FRW – Fachwissenschaftliches Raumplanungskonzept Windenergie	Meridian GmbH	Univ. Göttingen
2001–2004	GeoVis – Geographie und Visualisierung: Interaktive 3D-Visualisierung von Oberflächenformen und Klimaparametern zur Veranschaulichung in der Lehre	BMBF/GMD	HU Berlin, Univ. Göttingen, TU Dresden
2002–2006	SeReK – Semi-empirische Regionalisierung von Klimaparametern	NLfB Hannover, RWTH Aachen	Univ. Göttingen
2005–2008	GEOSTEP – Geoinformationstechnologien für standorteffiziente Pflanzenproduktion	Bayrische Forschungs-stiftung	TU München, geo-Konzept GmbH, Wimex GmbH, Agrolab GmbH, GSF, Universität Göttingen

Die Bündelung des durch diese Softwarelösungen repräsentierten *Know How* durch die Gründung der AG Geosystemanalyse in 1998 ging einher mit der Aufstellung neuer Konzepte für die Regionalisierung von Bodenparametern und begründet die bis heute fortgesetzte Zusammenarbeit mit dem NLfB und der BGR (BÖHNER et al. 1998, BÖHNER et al. 1999). Zu den neuen Konzepten gehört der Einsatz geostatistischer Verfahren ebenso wie die Einbindung weiterer bodenrelevanter Größen, wie Klima und Hydrologie, die schließlich die Grundlage für die Modellierung bodenrelevanter Prozesse bilden und mittlerweile erfolgreich ihren Weg in die Anwendung gefunden haben (BÖHNER et al. 2002, BÖHNER & KÖTHE 2003, BÖHNER & SELIGE 2006). Die zunehmend breitere Fächerung von Forschungsgebieten und Anwendungen spiegelt sich auch in den Projekten wider, an denen die AG Geosystemanalyse seit 1999 beteiligt ist (Tab. 1), und für deren Durchführung der neu entwickelten Softwarelösung SAGA eine zunehmend tragende Rolle zukommt[1].

Die Entwicklung der Softwarelösungen SARA, SADO und DiGeM begründet sich u.a. darin, dass zu ihrer Entstehungszeit die Schwierigkeit bestand, rasterbasierte Methoden unter Verwendung der verfügbaren GIS-Software neu zu entwickeln. Entweder verfügte diese, im Fall von vektororientierten Programmen, gar nicht über entsprechende Fähigkeiten oder die vorhandenen Schnittstellen waren nicht flexibel genug. Ebenso stellten die z.T. nicht unerheblichen Kosten für den Erwerb von Nutzerlizenzen eine Hürde dar, sich für ein potentiell geeignetes System zu entscheiden. Die quelltextoffene und freie GIS-Software GRASS (Geographical Resource Analysis Support System) kam hier den Anforderungen am meisten entgegen (NETELER 2003, GRASS 2006). GRASS hat aber die Nachteile, auf UNIX-Betriebssysteme fixiert zu sein – erst seit 1995 wird immerhin auch das UNIX-ähnliche Linux unterstützt – und eine in den 1980er Jahren entwickelte, umständlich zu benutzende Programmierschnittstelle zu besitzen. Auf der anderen Seite behinderte die Verwendung von drei verschiedenen Softwarelösungen zunehmend die alltägliche Projektarbeit. Die Entwicklung neuer Methoden verteilte sich auf die Softwarelösungen je nach Vorliebe des Entwicklers. Während SADO und DiGeM in der Programmiersprache C++ für Windows-Betriebssysteme entwickelt worden waren, war SARA in Fortran77 programmiert und lief nur unter UNIX. Um die Vorteile jedes einzelnen Programms auszunutzen, erfolgte der Datenaustausch also nicht nur zwischen verschiedenen Programmen, sondern auch zwischen verschiedenen Betriebssystemen. Für spezielle Aufgaben, wie der Koordinatentransformation oder der Erstellung von Karten, wurden außerdem zusätzliche Programme benötigt und z.T. eigens entwickelt. Vor diesem Hintergrund kristallisierte sich die Entscheidung heraus, die verschiedenen Software-

[1] Die von der AG Geosystemanalyse entwickelten Verfahren zur Regionalisierung von Bodenparametern sowie Ergebnisse der Projekte WEELS (Wind Erosion on European Light Soils) und GeoVis (Geographie und Visualisierung) sind im Anhang zu dieser Arbeit durch den Abdruck von Veröffentlichungen dokumentiert (BÖHNER et al. 2002, BÖHNER et al. 2003, CONRAD 2005).

lösungen durch ein einziges System zu ersetzen, das alle Stadien von der Methodenentwicklung bis zur routinemäßigen Datenverarbeitung und Präsentation von Ergebnissen unterstützt.

1.2 Entwicklung

Die Kernanforderungen für die Entwicklung von SAGA wurden von den Mitgliedern der Arbeitsgruppe wie folgt formuliert:

- Das System muss eine komfortable Programmierschnittstelle zur Unterstützung der Implementierung räumlich arbeitender Methoden bereitstellen.
- Methoden sollen modular aufgebaut sein, so dass sie unabhängig vom Gesamtsystem entwickelt und verwaltet werden können.
- Die Methoden sollen sich selbst dokumentieren.
- Die routinemäßige Anwendung neu entwickelter Methoden soll unmittelbar nach ihrer Erstellung durch beliebige Nutzer möglich sein.
- Es sollen Tabellen, Vektor- und Rasterdaten verarbeitet werden können.
- Das System soll über eine einfach bedienbare Benutzerschnittstelle verfügen mit der Möglichkeit der direkten graphischen Darstellung von Dateninhalten einschließlich der kartographischen Ausgabe.

Nachdem SADO wie auch DiGeM in C++ programmiert wurden, sollte auch SAGA in dieser objektorientierten Programmiersprache geschrieben werden. Mit der Möglichkeit Objektmodelle zu erstellen (Kap. 2.1), ist C++ besonders dafür geeignet, eine effiziente und übersichtliche Programmierschnittstelle zu erstellen, um dadurch die Entwicklung von Methoden für die Arbeit mit raumbezogenen Daten zu unterstützen. Das grundlegende Datenobjektmodell, aber auch die Funktionalität der Benutzeroberfläche, wurde dem Programm DiGeM entlehnt, das so zur Basis der SAGA-Entwicklung wurde. Die eigentliche SAGA-Entwicklung begann zwischen 2000 und 2001 und konzentrierte sich zunächst auf die Entwicklung einer Programmierschnittstelle, die ein modulares Methodenkonzept ermöglicht.

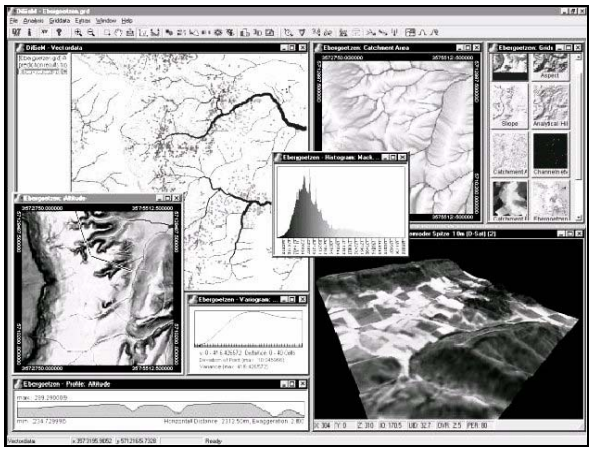


Abb. 1: DiGeM 2.0

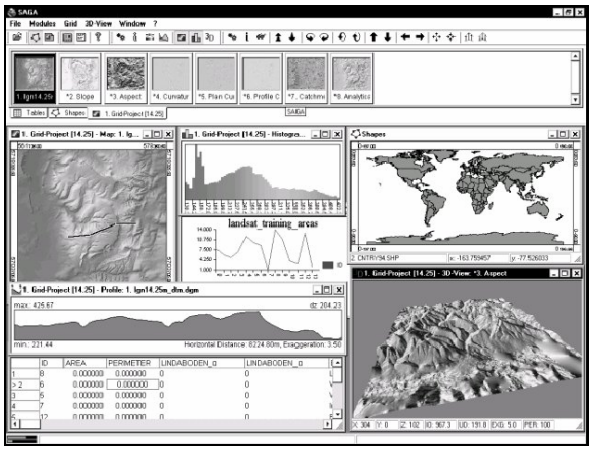
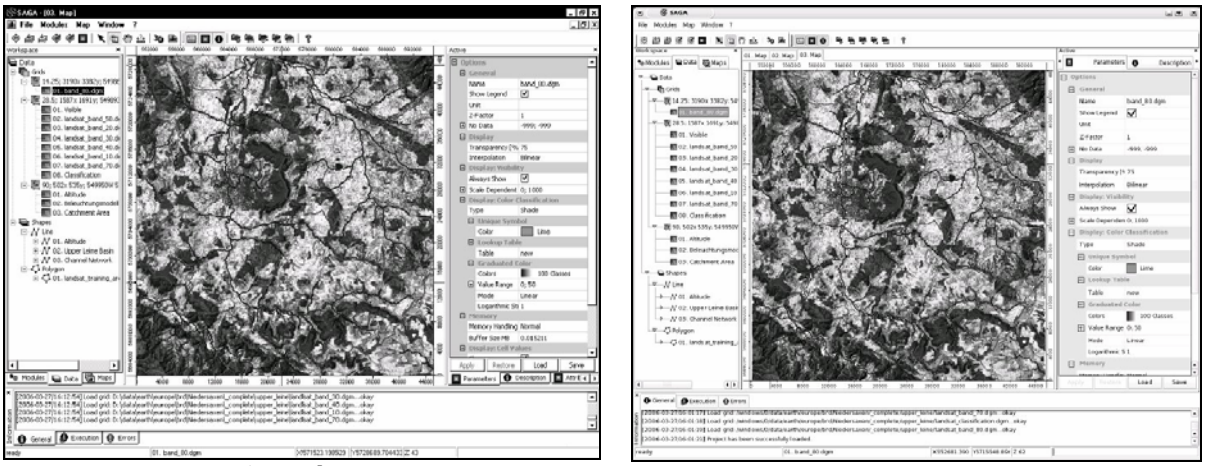


Abb. 2: SAGA 1.2

Die Module wurden zunächst noch fest in DiGeM eingebettet. Erst mit der Entwicklung einer eigenständigen Benutzeroberfläche, die es gestattet, Module dynamisch zu laden und auszuführen, wurde es möglich, Methoden unabhängig vom Gesamtsystem zu erstellen und die damit verbundenen Programmierarbeiten auf verschiedene Entwickler der Arbeitsgruppe zu verteilen. Dadurch konnte SAGA während der folgenden zwei Jahre zunehmend die vorigen Softwarelösungen ersetzen und sich als gemeinsame Basis für die Projektarbeit der Arbeitsgruppe etablieren.

Im Februar 2004 wurde ein Großteil der SAGA-Quelltexte nach ausgiebiger Diskussion innerhalb der Arbeitsgruppe unter eine Open Source Lizenz gestellt und mit der Versionsnummer 1 als Open Source Software (OSS) veröffentlicht (Abb. 2). Der Hauptgrund, SAGA zu veröffentlichen, bestand darin, dass SAGA sich zu einem fähigen und dank einiger Publikationen auch nachgefragten Werkzeug entwickelt hatte. Vor allem aber versprach sich die Arbeitsgruppe durch eine Veröffentlichung positive Rückkopplungseffekte für die Einwerbung weiterer Drittmittel. Eine Freigabe der Quelltexte verhindert zwar die Möglichkeit einer direkten kommerziellen Vermarktung einer Software, steigert aber auf der anderen Seite automatisch ihre Attraktivität. Da sich der kommerzielle Vertrieb von SAGA allein schon wegen der fehlenden Infrastruktur kaum gelohnt hätte, waren die Vorteile einer Quelltextfreigabe letztlich ausschlaggebend, zu denen neben der Attraktivitätssteigerung durch die Kostenfreiheit auch der Aufbau einer Nutzergemeinde gehört, die sich aktiv an der Weiterentwicklung der Software beteiligt. Insbesondere bei Software, die aus der Forschung entstanden ist, hat sich das Konzept der Quelltextfreigabe bewährt, da hierdurch die enthaltenen Algorithmen dokumentiert sind und die Wissenschaftsgemeinde diese anwenden, überprüfen und verbessern kann. Die Quelltextfreigabe von SAGA erfolgte jedoch nicht uneingeschränkt. Ermöglicht durch den modularen Aufbau von SAGA, hat sich die Arbeitsgruppe vorbehalten, eine Reihe von Methoden, die auf speziellem Fachwissen beruhen, u.a. für eine kommerzielle Vermarktung in Form von Dienstleistungen, zurückzuhalten.

Um in erster Linie eine Reihe von Einschränkungen in der Benutzeroberfläche zu



a) Windows b) Linux

Abb. 3: SAGA 2.0

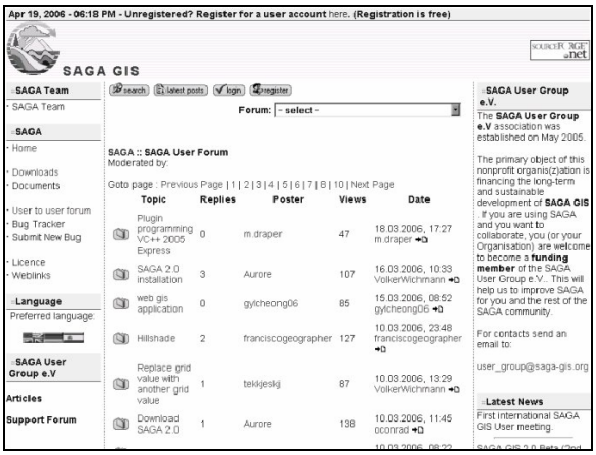


Abb. 4: SAGA-Homepage

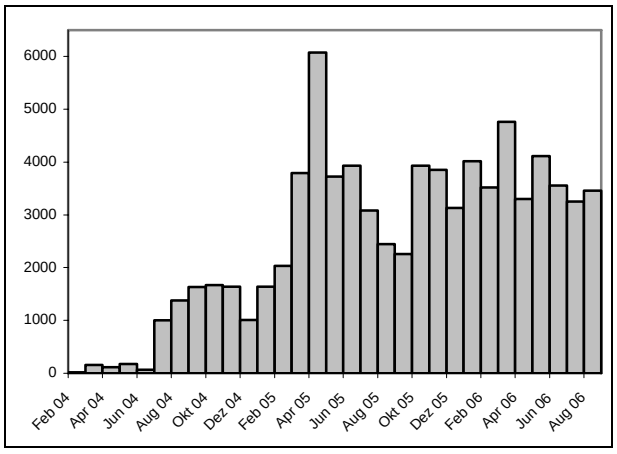


Abb. 5: Monatliche Downloads

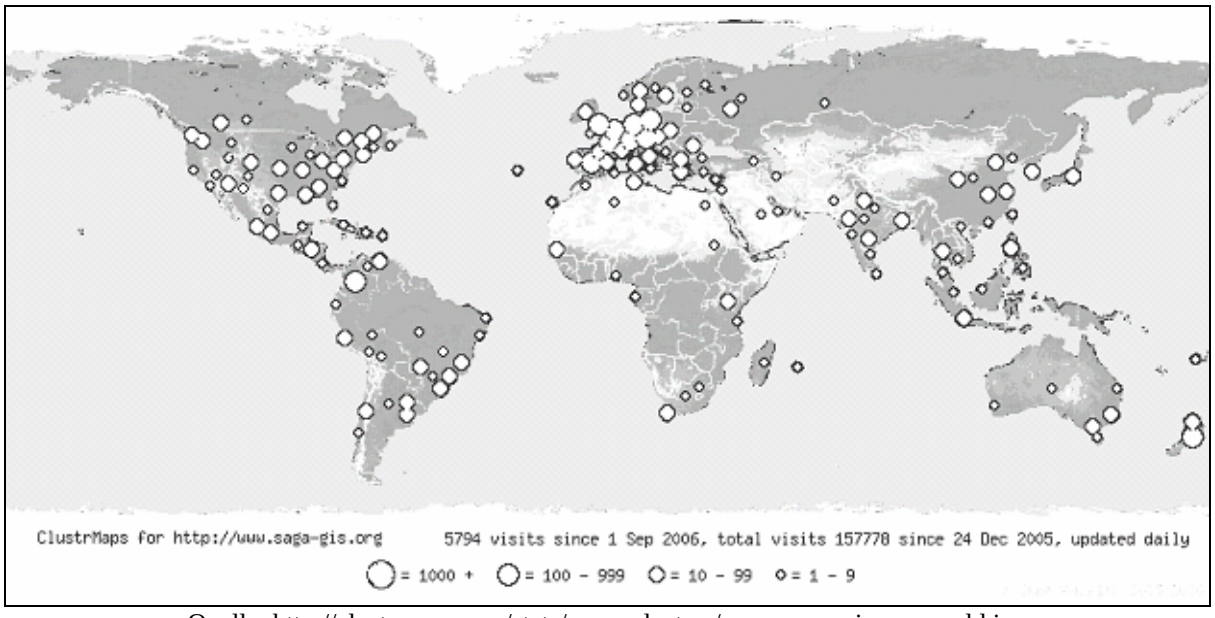
umgehen, die noch auf die Softwarearchitektur von DiGeM zurückgehen, wurde im Sommer 2004 die Entwicklung von SAGA in der Version 2 begonnen. Die Benutzeroberfläche wurde dabei völlig neu programmiert und auf die Verwendung der Betriebssystem übergreifenden Bibliothek wxWidgets umgestellt, die nun die vorher benutzte Microsoft Foundation Class Library (MFC) ersetzt. In der Folge lässt sich die im Frühjahr 2005 erstmals veröffentlichte Version 2 von SAGA sowohl unter Windows als auch unter Linux Betriebssystemen ausführen (Abb. 3). Auch der Funktionsumfang der Programmierschnittstelle wurde erheblich erweitert, jedoch ohne dass sich tiefgreifende Änderungen für die Quelltexte bereits vorhandener Module ergeben haben. Eine Bestätigung für das erfolgreiche Konzept von SAGA und seiner Programmierschnittstelle ist der seitdem ständig wachsende Umfang unter SAGA verfügbarer Methoden. Zur Zeit konzentriert sich die Weiterentwicklung des Systems neben der obligatorischen Entfernung von Programmierfehlern und der Ergänzung weiterer Funktionen auf die Erstellung einer Schnittstelle für Skriptsprachen, die u.a. für die Durchführung des GeoSteP-Projekts benötigt wird (Tab. 1, GEOSTEP 2006).

1.3 Verbreitung

Die Veröffentlichung von SAGA mit gleichzeitiger Freigabe der Quelltexte verfolgt vor allem das Ziel, durch eine größere Popularität und Verbreitung des Systems positive Rückwirkungen für die Arbeitsgruppe zu erreichen. Um die Quelltexte wie auch die ausführbaren Programmdateien verfügbar zu machen, werden sie über eine eigene Homepage für den Download angeboten, die von dem Internetdienstleister SourceForge bereitgestellt wird (http://sourceforge.net/saga-gis). SourceForge bietet OSS-Projekten eine kostenlose, überwiegend werbungsfinanzierte Infrastruktur an und gehört mit über 100.000 Einträgen zu den größten Dienstleistern auf diesem Gebiet (OPEN SOURCE TECHNOLOGY GROUP 2006). Zum Funktionsumfang gehört neben der Bereitstellung der Quelltexte und Programmdateien eine einfach gehaltene Homepage mit Foren, in denen SAGA-Benutzer Programmfehler berichten und Wünsche über die weitere Programmentwicklung äußern können, ein Benachrichtigungssystem für angemeldete Benutzer, die Bereitstellung von Statistiken über die Projektaktivität sowie ein Quelltextverwaltungssystem. Um zusätzliche und freundlicher aufbe-

reitete Inhalte für SAGA bereitzustellen, wurde eine weitere Homepage in der Domain der Universität Göttingen eingerichtet (http://www.saga-gis.uni-goettingen.de), die u.a. auch ein Forum enthält, in dem Nutzer ihre Erfahrungen mit SAGA untereinander austauschen können (Abb. 4). Die Homepage liegt außer in englischer Sprache auch in deutscher und spanischer Übersetzung vor. Durch die Anmeldung der Homepage bei Internetsuchdiensten und der Herausgabe von Pressemitteilungen an Internetfachzeitschriften wurde allmählich das Interesse der Öffentlichkeit an SAGA geweckt, was sich auch an der von SourceForge zur Verfügung gestellten Download-Statistik ablesen lässt (Abb. 5). Die Spitze im April 2005 ist in Zusammenhang mit der kurz zuvor stattgefundenen SAGA-Präsentation auf der CeBIT Hannover zu sehen (FUHRMANN-KOCH 2005). Verantwortlich für diese Präsentation war der u.a. von den Mitgliedern der AG Geosystemanalyse im Januar 2005 gegründete und als allgemeinnützig anerkannte Verein SAGA User Group e.V., der sich der Förderung von SAGA verschrieben hat. Zu den in der Vereinssatzung (SAGA 2006) festgeschriebenen Teilaufgaben gehört die Koordination der Systementwicklung, die Einwerbung finanzieller Mittel, die Präsentation auf Tagungen und Fachmessen sowie die Organisation regelmäßiger Nutzertreffen. Das erste international gehaltene Nutzertreffen fand in 2006 im Rahmen der Fachmesse für Angewandte Geoinformatik (AGIT) in Salzburg statt und wurde von einem Sonderband der Göttinger Geographischen Abhandlungen über Anwendungen von SAGA begleitet (BÖHNER et. al. 2006).

Einen ersten Überblick über die internationale Verbreitung von SAGA geben bereits die Einträge im Nutzerforum auf der SAGA-Homepage. Eine geographische Aufschlüsselung der IP-Adressen (Internet Protocol), von denen aus auf die Homepage zugegriffen wird, zeigt, dass SAGA mittlerweile durchaus auf ein weltweites Interesse stößt (Abb. 6). Die verbreitete Anwendung von SAGA schlägt sich zunehmend auch in der steigenden Zahl von Veröffent-



Quelle: http://clustrmaps.com/stats/maps-clusters/www.saga-gis.org-world.jpg

Abb. 6: Zugriffe auf die SAGA-Homepage

lichungen nieder, die Forschungs- und Projektergebnisse dokumentieren, welche unabhängig von der AG Geosystemanalyse zu sehen sind. Die meisten Arbeiten heben dabei bestimmte methodische Fähigkeiten hervor, die in dieser Form nur in SAGA verfügbar sind. Die Nähe zur Arbeitsgruppe bedingt allerdings einen methodischen Vorsprung für Arbeiten, die an der Universität Göttingen durchgeführt werden, und auch sehr neue in SAGA verfügbare Ansätze benutzen (z.B. STOCK 2005, KRÜGER 2006). Im allgemeinen werden am häufigsten die reliefanalytischen und rasterbasierten Fähigkeiten von SAGA eingesetzt, deren Umfang und Kompetenz auf den Ursprung von SAGA zurückzuführen sind (z.B. BEHRENS & SCHOLTEN 2006). Auch bei Vergleichen mit anderer GIS-Software wird SAGA vor allem auf diesen Gebieten eine sehr gute Konkurrenzfähigkeit bescheinigt (Tab. 2, KLINGSEISEN 2004), wobei aber auch andere Aspekte, insbesondere mit Rücksicht auf die kostenlose Verfügbarkeit des Programms, positiv bewertet werden (TRAUN 2005).

Am interessantesten für das Projekt sind jedoch Nutzer, die selber wieder Beiträge für die Weiterentwicklung von SAGA beisteuern. Hier bewährt sich das modulare Konzept und die komfortable Programmierschnittstelle, die eine Reihe von Nutzern dazu veranlasst hat, eigene Methoden als SAGA-Module zu implementieren, auch wenn diese nicht immer unmittelbar für andere Nutzer zur Verfügung gestellt werden (z.B. HECKMANN & BECHT 2006, WICHMANN 2006). Der größte einzelne Einfluss ging bis jetzt von dem Projekt Sistema Extremeno de Analisis Territorial (SEXTANTE) aus, in dessen Rahmen eine spanischsprachige Version von SAGA als GIS-Lösung für öffentliche Einrichtungen der spanischen Verwaltungseinheit Extremadura erstellt wurde (SEXTANTE 2006). Neben einer Reihe neuer SAGA-Module entstand hieraus auch ein SAGA-Handbuch in englischer und spanischer Sprache (OLAYA 2004), das mit seiner Veröffentlichung auf der SAGA-Homepage zu einer erheblichen Popularitätssteigerung von SAGA führte. Solche Projekteinbindungen von SAGA, die über das einfache Anwenden von Methoden hinausgehen oder Nutzer aus nicht vorwiegend wissenschaftlich orientierten Kreisen mit einbeziehen, sind eine wesentliche Triebfeder für die Weiterentwicklung von SAGA. Neben dem bereits erwähnten GeoSteP Projekt, an dem die AG Geosystemanalyse direkt beteiligt ist, soll hier noch das von der EU geförderte NeWater Projekt angeführt werden, bei dem der Einsatz von SAGA im integrierten Einzugsgebietsmanagement geplant ist und an dem u.a. das Umweltforschungszentrum (UFZ) Leipzig beteiligt ist (GIORDANO et al. 2006, NEWATER 2006). Durch dieses Projekt werden neue Anforderungen an SAGA gestellt, zu denen eine Datenbankanbindung gehört, aber auch eine vereinfachte Benutzeroberfläche, die sich leicht durch ortsansässige Nutzer ohne fundierte GIS-Kenntnisse bedient lässt.

Ein weiteres wichtiges Feld ist der Einsatz von SAGA in der Lehre, wie er im Geographischen Institut der Universität Göttingen bereits seit 2003 praktiziert wird (CONRAD 2005). Hier bietet SAGA mit seinem umfassenden Methodenschatz eine kostenlose Alternative zu den sonst dominierenden kommerziellen Programmen und unterstützt dadurch auch das eigenverantwortliche Selbststudium. Es kann neben der universitären Lehre aber auch an den Einsatz in Schulen gedacht werden, wie z.B. die Einführung von KOPPERS (2006) zeigt. In

Tab. 2: Reliefanalytische Fähigkeiten verschiedener GIS Software (nach KLINGSEISEN 2004).

Programm	Hangneigung	Exposition	Horizontalwölbung	Vertikalwölbung	Andere Reliefparameter	Reliefklassifikation	Räumliche Interpolation	Zonale u. lokale Statistik	Depressionen entfernen	Individuell anpassbar	Unterstütze Betriebssysteme	Unterstützte Dateiformate
GIS Software												
ESRI ArcInfo 7.0.2	O	O					O	O	O	O	Windows, Unix	Alle gängigen Bild- u. Rasterformate
ESRI Spatial Analyst 8.2	O	O					O	O		O	Windows	Alle gängigen Bild- u. Rasterformate
GeoMedia Grid 5.1b	O	O					O	O	O	O	Windows	Alle gängigen Bild- u. Rasterformate
IDRISI Kilimanjaro	O	O	O	O		O	O	O	O	O	Windows	Alle gängigen Bild- u. Rasterformate
TNT Mips 6.8	O	O					O	O	O	O	Windows, Unix, MacOS	Alle gängigen Bild- u. Rasterformate
SAGA 2.0	O	O	O	O	O	O	O	O	O		Windows, Unix	Alle gängigen Bild- u. Rasterformate
Freie Software, spezialisiert auf Reliefanalyse												
TAPES-G	O	O	O	O					O		Unix	ESRI ASCII Grid, XYZ
DiGeM 2.0	O	O	O	O	O	O			O		Windows	ESRI ASCII Grid, Surfer
Landserf	O	O	O	O		O				O	Windows, Unix	Alle gängigen Bild- u. Rasterformate

einer von mir selbst durchgeführten Übung konnte auch erfolgreich gezeigt werden, dass die SAGA-Programmierschnittstelle durchaus dazu geeignet ist, Geographiestudenten mit nur geringen Vorkenntnissen an die Programmierung räumlicher Analyseverfahren heranzuführen. Eine weitere Verbreitung von SAGA ist schließlich durch das derzeitige Angebot von SAGA-Schulungen am Zentrum für Geoinformatik Salzburg (Z_GIS) zu erwarten, die zukünftig durch ein bereits im Aufbau befindliches Online-Seminar ergänzt werden.

1.4 Aufbau der Arbeit

Mit SAGA, dem Thema dieser Arbeit, wird kein einzelnes Forschungsprojekt vorgestellt. Vielmehr handelt es sich dabei um ein Werkzeug, das gewissermaßen als Nebenprodukt verschiedener Forschungsprojekte entstanden ist und nicht nur zahlreiche Forschungsergebnisse in sich vereint, sondern diese auch der unmittelbaren Anwendung, z.B. in anderen

Forschungsarbeiten oder in der universitären Lehre, zugänglich macht. Als Softwaresystem für die Bearbeitung, Analyse und Modellierung raumbezogener Daten bietet SAGA eine Vielzahl von Standardfunktionalitäten einer GIS-Software wie auch eine Reihe sehr spezieller und einzigartiger Methoden an. Die Bedienung des Systems erfolgt über eine innovative Benutzeroberfläche. Komplexere Arbeitsabläufe können aber auch mit Hilfe von Skriptdateien automatisiert werden. Eine besondere Eigenschaft ist die leicht zugängliche Programmierschnittstelle, mit deren Hilfe sich Algorithmen für raumbezogene Daten effektiv in vom eigentlichen System unabhängigen Modulen umsetzen lassen. Der Entwurf der Systemarchitektur und die anschließende Implementierung des Systems wurde in jeder Phase von zahlreichen fruchtbaren Diskussionen innerhalb der Arbeitsgruppe begleitet, die Durchführung wurde aber überwiegend vom Autor dieser Arbeit selbst vorgenommen, worauf sich auch der Anspruch begründet im nachfolgenden einen umfassenden Einblick in die Funktionsweise von SAGA zu geben. Dabei wird versucht alle Aspekte, die für das Arbeiten mit SAGA von Interesse sind, verständlich und übersichtlich darzustellen, wobei ein besonderes Anliegen in der Vermittlung der Funktionsweise der Programmierschnittstelle besteht, da gerade in der Entwicklung neuer geowissenschaftlicher Methoden durch eine weit gestreute Entwicklergemeinde das größte Potential von SAGA für die raumbezogen arbeitenden Wissenschaften gesehen wird. Um abschließend einige der bisherigen Einsatzgebiete von SAGA sowie meine eigenen in diesem Zusammenhang gemachten Beiträge in Forschung und Lehre zu dokumentieren, wurden fünf ausgewählte Publikationen in den Anhang dieser Arbeit aufgenommen.

Im Anschluss an diese Einführung werden in Teil 2 zunächst Grundlagen der Informatik vorgestellt, die einerseits von allgemeiner Bedeutung für den Aufbau und die Funktionsweise des Systems sind und zum anderen Grundkenntnisse der Programmierung vermitteln, wie sie für die Verwendung von SAGA's Programmierschnittstelle benötigt werden. Teil 3 behandelt das System selber und geht abgesehen vom Aufbau und der Verwendung des Systems, die von allgemeinerem Interesse sind, auch auf die Erstellung von Modulen ein. Zunächst werden in Kapitel 1 verschiedene Installationsmöglichkeiten beschrieben. Kapitel 2 gibt einen Überblick über die Systemarchitektur, stellt aber auch das besondere, hierauf basierende Softwarelizenzmodell vor. Kapitel 3 bis 6 haben die einzelnen Systemkomponenten zum Inhalt. Kapitel 3 beginnt mit der SAGA-Programmierschnittstelle, die die Grundlage des Gesamtsystems bildet. Auch wenn ihre Kenntnis für ein besseres Verständnis nützlich sein kann, ist sie für den einfachen Nutzer von SAGA nicht erforderlich. Die hier aufgeführten Funktionalitäten und Objektmodelle sind vor allem für Leser von Interesse, die eigene Module unter der SAGA-Umgebung entwickeln wollen. Kapitel 4 und 5 beschreiben die graphische bzw. die kommandozeilengesteuerte Benutzerschnittstelle. Kapitel 6 ist ein thematisch gegliederter Überblick über alle Module, die zur Zeit in der OSS-Distribution von SAGA enthalten sind. Das letzte Kapitel von Teil 4 ist eine Einführung in die Modulprogrammierung, die auf einer vom Autor durchgeführten Übung basiert und gleichzeitig grundlegende Konzepte der Geoinformatik vertieft. Alle Quelltexte zu dieser Einführung sind in Anhang B vollständig

abgedruckt. Teil 4 gibt schließlich einen Ausblick auf die Weiterentwicklung und das zukünftige Anwendungspotential von SAGA. In Ergänzung zum Hauptteil dieser Arbeit dokumentieren die abgedruckten Publikationen in Anhang A den Einsatz von SAGA und den unter SAGA implementierten Methoden in Forschung und Lehre. Die begleitende CD-Beilage enthält die aktuellen Quelltext- und Binärdistributionen von SAGA, eine unter Windows direkt ausführbare SAGA-Installation sowie einige Testdatensätze, so dass sich der Inhalt dieser Arbeit leicht durch praktisches Arbeiten mit dem Programm nachvollziehen lässt.

2 GRUNDLAGEN

Die Grundlagen, die für die Entwicklung von SAGA, aber auch für das Verständnis des Systems und seine Anwendung als Entwicklungsplattform geographischer Methoden von Bedeutung sind, lassen sich im wesentlichen auf zwei Teilgebiete der Informatik zurückführen. Zum einen handelt es sich um den Teil der praktischen Informatik, der die Programmierung bzw. die Entwicklung von Software zum Inhalt hat, wodurch vor allem programmiertechnische Aspekte abgedeckt werden, aber auch solche, die sich mit dem Schutz und der Verbreitung von Software beschäftigen. Die zweite Teildisziplin, die im allgemeinen der angewandten Informatik zugeordnet wird und eine direkte Verbindung zu den raumbezogen arbeitenden Wissenschaften herstellt, ist die Geoinformatik. Sie liefert die theoretischen Grundlagen für die computergestützte Verarbeitung geographischer Daten.

2.1 Programmierung

Die Wahl der Programmiersprache ist nicht unerheblich für die Implementierung eines Programms. Entscheidungskriterien sind u.a. der Funktionsumfang der Sprache, die Verfügbarkeit von Compilern, die Portabilität der Quelltexte, die Möglichkeit der Integration fremder Komponenten und die Unterstützung graphischer Benutzeroberflächen. SAGA wurde in der objektorientierten Programmiersprache C++ entwickelt, die durch ihr hohes Abstraktionsvermögen einer strukturierten Programmierung sehr entgegenkommt und so die nachhaltige Pflege und Wiederverwendbarkeit der Quelltexte unterstützt. C++ leitet sich ab von der prozeduralen Programmiersprache C, die gewissermaßen eine Untermenge von C++ darstellt und auch dafür bekannt ist, eine effektive, maschinennahe Programmierung zu ermöglichen. Beide Sprachen sind sehr weit verbreitet und es gibt sowohl eine Vielzahl von Compilern für verschiedenste Betriebssysteme als auch eine immense Anzahl integrierbarer Quelltexte und Bibliotheken, die oft einen wissenschaftlichen Hintergrund haben. Daneben gibt es aber auch Bibliotheken, die z.B. den Zugriff auf spezielle Dateiformate erlauben oder für die Gestaltung von Benutzeroberflächen gedacht sind, wie die in SAGA 2.0 verwendete Bibliothek wxWidgets. Bei der Verwendung solcher externen Ressourcen kommt ein anderer Aspekt der Programmentwicklung ins Spiel, bei dem es um die Urheber- und Nutzungsrechte geht. So fordern viele der verbreiteten freien Softwarelizenzen, dass unter sie gestellte Programme und Bibliotheken nur von ebenfalls freier und quelltextoffener Software benutzt werden dürfen. Eine derartige Lizenz, die im Sinne eines frei zugänglichen Wissens durchaus zu begrüßen ist, wurde schließlich auch zur rechtlichen Absicherung von SAGA gewählt. In diesem Kapitel werden die soeben angerissenen Themenblöcke etwas näher beleuchtet. Die kurze Einführung in die Programmiersprachen C und C++ soll dabei auch beim Nachvollziehen der etwas spezielleren Kapitel über den Aufbau der SAGA-API (Kap. 3.3) und die Einführung in die Modulprogrammierung (Kap. 3.7) helfen. Nicht zuletzt besteht die Hoffnung, dass hierdurch für viele Leser die Hemmschwelle gesenkt wird, Programmierung als Werkzeug für die Lösung eigener Fragestellungen zu benutzen.

2.1.1 Programmiersprachen

Ein Computer ist im Kern eine Sammlung von Schaltkreisen, die, gesteuert durch Prozessorbefehle, an- oder ausgeschaltet werden. Die Zustände der Schaltkreise werden zur Darstellung von Daten bzw. Datenwerten benutzt, die durch eine geeignete Reihenfolge nacheinander auszuführender Prozessorbefehle zum Berechnen neuer Datenwerte benutzt werden können. Über eine entsprechende Computerperipherie können die Datenwerte z.B. an einen Bildschirm ausgegeben oder für die Steuerung einer Maschine benutzt werden. Eine Reihenfolge von Prozessorbefehlen nennt man ein Programm. Je nach Zusammenstellung der Prozessorbefehle ist ein Computer in der Lage eine Vielzahl von Problemstellungen zu lösen. Programmieren ist anders formuliert, die Aufstellung einer Liste von Arbeitsanweisungen für einen Computerprozessor (HOPFER 1987).

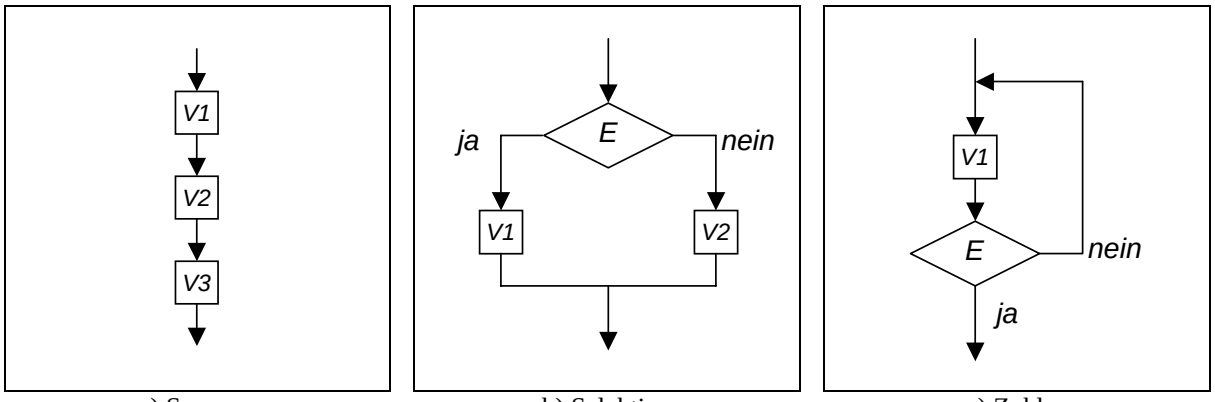
Ein Schaltkreis repräsentiert die kleinste von einem Computer verwaltbare Informationseinheit, die Bit genannt wird, und kann, je nachdem ob der Schaltkreis aus- oder angeschaltet ist, die Werte 0 oder 1 repräsentieren. Um andere Werte darzustellen, werden in den meisten Computersystemen acht nebeneinander liegende Bits zu einem sogenannten Byte zusammengefasst. Jedes Bit eines Bytes hat eine andere Wertigkeit, die sich für jede Position vom ersten zum letzten Bit hin verdoppelt. Wenn es gesetzt ist, entspricht das zweite Bit also nicht dem Wert 1 sondern dem Wert 2, das dritte dem Wert 4, das vierte dem Wert 8. Das achte Bit entspricht schließlich dem Wert 128. Den Wert eines Bytes erhält man durch Addition der Werte aller gesetzten Bits. Auf diese Weise lassen sich je nach Schaltmuster eines Bytes beliebige Werte zwischen 0 und 255 darstellen. Diese Zahlendarstellung entspricht dem Binärsystem. Durch Hinzunehmen weiterer Bytes lässt sich der darstellbare Wertebereich schnell erweitern. Negative Werte oder Fließkommazahlen lassen sich in gleicher Weise durch veränderte Interpretation der Wertigkeit einzelner Bits darstellen. Die Prozessorbefehle eines Programms werden im Speicher des Computers ebenfalls durch das Bitmuster eines Bytes bzw. dem durch ihm dargestellten Wert angesprochen, so dass sich ein Computerprogramm eigentlich nur aus einer Folge computerlesbarer Zahlen zusammensetzt. Diese Form, Programme als Folgen von Zahlencodes zu schreiben, wird Maschinensprache genannt. Es ist die einzige Sprache, die ein Computer direkt versteht.

Maschinensprachen setzen genaue Kenntnisse des jeweiligen Prozessors, seiner Speicherverwaltung, wie auch seiner peripheren Schnittstellen voraus. Sie sind, nicht zuletzt wegen der ausschließlichen Verwendung von Zahlencodes, sehr fehleranfällig und werden heute kaum noch benutzt. In der verbreiteten Einteilung von Programmiersprachen in Generationen sind Maschinensprachen die Sprachen der ersten Generation (HOPFER 1987, LOUDEN 1994). Die maschinenorientierten Assemblersprachen, die in dieser Einteilung die Position der zweiten Generation von Programmiersprachen einnehmen, ersetzen die Zahlencodes der Maschinensprache durch anschaulichere symbolische Buchstabenfolgen, die allerdings vor ihrer Ausführung erst durch einen Compiler in maschinenlesbaren Code übersetzt werden müssen. Im übrigen haben Assemblersprachen die gleichen Eigenschaften, wie die Maschi-

nensprache selbst. Seit den fünfziger Jahren bieten leistungsfähigere Computer die Voraussetzung für die Entwicklung sogenannter höherer Programmiersprachen, die einen höheren Abstraktionsgrad aufweisen und eine mehr oder weniger maschinenunabhängige Programmierung ermöglichen (HOPFER 1987). Zu den Eigenschaften der höheren Programmiersprachen gehört die Verwendung von Variablen sowie spezielle Befehlssätze für die bedingte Verzweigung und die Bildung von Schleifen. Die Quelltexte, die die Anweisungen für ein Programm enthalten, werden von einem Compiler vollständig in die jeweilige Maschinensprache des Computers übersetzt, bevor sie ausgeführt werden können. Eine Alternative sind Interpretersysteme, die eine Anweisung unmittelbar nach ihrer Übersetzung ausführen. Interpretierte Programme haben aber den Nachteil, dass ihre Rechenzeit oft sehr hoch ist, und werden daher vornehmlich in Verbindung mit Skriptsprachen eingesetzt, durch die dann die eigentlichen, kompilierten Programmfunktionen aufgerufen werden.

Es gibt heute sehr viele verschiedene höhere Programmiersprachen, die zum einen für spezielle Aufgaben konzipiert wurden, zum anderen aber auch eine Verschiebung der zu Grunde liegenden Programmierparadigmen widerspiegeln. Die zunehmende Abstraktion der Programmiersprachen führte zunächst zu den prozeduralen Sprachen, die die Strukturierung eines Programms in einzelne Prozeduren oder Funktionen für die Ausführung bestimmter Aufgaben ermöglichen. Zu den Vorteilen zählen die Modularisierung der Quelltexte und die Wiederverwendbarkeit einmal definierter Funktionen durch andere Programme. Typische Beispiele für prozedurale Sprachen, die auch heute noch eine weite Verbreitung haben, sind Fortran, Pascal und C. Objektorientierte Sprachen verbinden Dateninhalte mit datenspezifischen Methoden zu Objekten. Die Verwendung von Objektklassen und Objektklassenhierarchien zeigt, wie weit sich diese Sprachen von der maschinenorientierten Programmierung weg in Richtung problemorientierter Programmierung entwickelt haben. Die Möglichkeit, spezielle Objektklassen von allgemeineren abzuleiten, so dass sie deren Eigenschaften übernehmen, diese aber gleichzeitig um neue erweitern, wird Vererbung genannt. Vererbung begünstigt eine weitergehende Modularisierung und damit auch effektivere Lösungen für viele Problemstellungen und erleichtert nicht zuletzt auch die nachhaltige Pflege und Wiederverwendbarkeit der Quelltexte. Beispiele für objektorientierte Sprachen sind Java und C++.

Trotz des im Detail oft sehr unterschiedlichen Sprachaufbaus, verfügen alle Programmiersprachen über einige grundlegende Konzepte, die sich bereits in der Maschinensprache finden lassen. Hierzu gehören neben einfachen Rechenoperationen und Wertzuweisungen insbesondere die Möglichkeit den Programmablauf unter definierten Bedingungen verzweigen zu lassen und auf diese Weise Schleifen bilden zu können. Bei der Entwicklung von Algorithmen kann es sehr hilfreich sein, sie durch Flussbilder darzustellen, die auf diesen Konzepten beruhen. Theoretische Untersuchungen haben gezeigt, dass alle überhaupt denkbaren Algorithmen aus nur wenigen Grundstrukturen, sogenannten Algorithmenbausteinen, realisierbar sind (HOPFER 1987). Besondere Bedeutung haben Sequenzen (Abb. 7a), die sich aus einer Reihenfolge von Verarbeitungsschritten zusammensetzen, Selektionen (Abb. 7b), durch die eine Entscheidung über den nächsten auszuführenden Verarbeitungs-



a) Sequenz b) Selektion c) Zyklus

Abb. 7: Algorithmische Strukturen in Flussbilddarstellung.

schritt getroffen wird (bedingte Verzweigung), und Zyklen (Abb. 7c), durch die eine Wiederholung von Verarbeitungsschritten stattfindet (Schleifen).

Unabhängig von den Möglichkeiten der jeweiligen Programmiersprache haben sich in der Programmierpraxis einige Programmierregeln bewährt, die u.a. eine systematische und strukturierte Programmierung ausmachen (HERRLICH & LINDNER 1981, GRAMS 1992).

- Die Programmentwicklung erfolgt in definierten Arbeitsetappen mit vorgegebenem Inhalt und Leistungsumfang.
- Der Programmentwurf wird durch wiederholte funktionelle Zergliederung und strukturelle Verfeinerung in abgeschlossene Bausteine eingeteilt (Stepwise Refinement).
- Es gilt die Redundanz zu reduzieren, durch die Verwendung weniger, übersichtlicher (standardisierter) Algorithmen- und Funktionsbausteine.
- Die Funktionsbausteine verbergen die Details ihrer Funktionsweise (Information Hiding), um zu verhindern, dass schwer überschaubare Nebenwirkungen z.B. bei Änderungen der Funktionsweise entstehen.
- Es ist stets auf die Lesbarkeit der Quelltexte zu achten, um logische Fehler möglichst auszuschließen. Hierzu gehört die sparsame Verwendung von Schnittstellen und Variablendeklarationen, wie auch die Umsetzung von Algorithmen in enger Anlehnung an die zugrundeliegende Theorie.
- Es sind möglichst selbsterklärende Variablen- und Funktionsnamen zu wählen. Quelltextkommentare sollten nur eingefügt werden, wenn sie wirklich hilfreich sind. Überflüssige Quelltextkommentare stören die Lesbarkeit.
- Die Optimierung rechenzeitintensiver Algorithmen erfolgt erst nach dem erfolgreichen Test einer Funktion.

2.1.2 Die prozedurale Sprache C

Die Programmiersprache C wurde Anfang der siebziger Jahre von D.M.Ritchie im Zusammenhang mit der Implementation des Betriebssystems UNIX entwickelt (KERNIGHAN & RITCHIE 1978), aber mittlerweile stehen zahlreiche Compiler für die meisten modernen Betriebssysteme zur Verfügung. C ist eine prozedurale Programmiersprache, die der strukturierten Programmierung sehr entgegenkommt, und ist mit nur 32 Schlüsselwörtern (Tab. 3) von ihrem Umfang her eine kompakte, leicht erlernbare Sprache. Die auf C aufbauende objektorientierte Programmiersprache C++, die für die Entwicklung von SAGA verwendet wurde, erweitert den Sprachumfang von C um Konzepte, auf die im nächsten Kapitel eingegangen wird. Doch zunächst soll in die Grundlagen von C selbst eingeführt werden, die auch detailliert in den zahlreichen Einführungen zu dieser Sprache nachgearbeitet werden können (z.B. KERNIGHAN & RITCHIE 1978, LOWES & PAULIK 1992).

Tab. 3: Die wichtigsten C Schlüsselwörter

Schlüsselwort	**Bedeutung/Kontext**
void	Undefinierter Datentyp
char	Datentyp, Ganzzahl (1 Byte)
int	Datentyp, Ganzzahl
float	Datentyp, Fließkommazahl (4 Byte)
double	Datentyp, Fließkommazahl (8 Byte)
short	Datentyp Spezifikation, für die Präzision von Ganzzahlen (2 Byte)
long	Datentyp Spezifikation, für die Präzision von Zahlen (int: 4 Byte, double 10 Byte)
signed	Datentyp Spezifikation, Wertebereich umfasst positive und negative Werte
unsigned	Datentyp Spezifikation, Wertebereich umfasst nur positive Werte
const	Datentyp Spezifikation, Wert kann nicht verändert werden
static	Datentyp Spezifikation, Wert bleibt zwischen Funktionsaufrufen erhalten
struct	Definition komplexer Datentypen
typedef	Datentypdefinition. Für die einfachere Verwendung benutzerdefinierter Datentypen
sizeof	Liefert die Bytezahl, die von einem Datentyp o. einer Datenstruktur belegt wird
if	Auswahlanweisung
else	Auswahlanweisung (in Verbindung mit if)
switch	Auswahlanweisung
case	Auswahlanweisung (in Verbindung mit switch)
default	Auswahlanweisung (in Verbindung mit switch)
while	Schleifenanweisung
do	Schleifenanweisung (in Verbindung mit while)
for	Schleifenanweisung
continue	Fortsetzen mit der nächsten Iteration einer Schleifenanweisung
break	Abbruch eines Anweisungsblocks
return	Verlassen einer Funktion, ggf. mit Rückgabe eines Funktionswertes.

In C werden alle Aufgaben eines Programms in Funktionen geschrieben, die eine variable Anzahl von Funktionsparametern haben und einen Wert zurückliefern können. Eine besondere Funktion ist die *main* Funktion, die in C Programmen die Hauptroutine darstellt, die bei Programmausführung automatisch aufgerufen wird. Ein einfaches C-Programm kann z.B. so aussehen.

```
int Verdoppeln(int i)
{
   return( 2 * i );
}

void main(void)
{
   int   i;

   i  = Verdoppeln(5);
}
```

Das Programm besteht aus der Hauptroutine und einer weiteren Funktion, die von der Hauptroutine aus aufgerufen wird. Diese Unterfunktion verdoppelt den Wert, der an sie übergeben wird, und gibt ihn an die aufrufende Funktion zurück. Jeder Funktion folgt ein Anweisungsblock, der durch ein Paar von Klammern (`{}`) eingeschlossen ist. Anweisungsblöcke können auch ineinander geschachtelt sein, um zusammenhängende Einzelanweisungen zusammenzufassen. Die Rückgabe des Funktionswerts erfolgt meistens am Ende einer Funktion mit der *return* Anweisung. Variablen müssen in C immer explizit mit ihrem Datentyp deklariert werden. Hierzu wird der Variablenname dem Schlüsselwort für den Datentyp nachgestellt. Mit *int* wird in dem Beispiel eine Ganzzahl deklariert. Funktionen können eine beliebige Anzahl von Funktionsparametern besitzen, die ebenfalls mit ihrem Typ deklariert werden müssen. Das gleiche gilt für den Rückgabewert. Das Schlüsselwort *void* (von engl. *leer*) in der *main* Funktion gibt an, dass diese Funktion keine Parameter erwartet und auch keinen Wert zurückgibt. Die Standarddatentypen von C sind in Tab. 4 aufgelistet. Die Angaben zur vom Datentyp benutzten Anzahl an Bytes und der davon abhängigen darstellbaren Wertegrenzen bezieht sich auf gängige C-Compiler für aktuelle 32-bit Betriebssysteme und kann für andere Compiler unterschiedlich ausfallen.

Tab. 4: Einfache Datentypen in C

Typ	**Bytes**	**Bedeutung**
char	1	Character, dient auch zur Codierung von ASCII Zeichen (-127 bis 127)
unsigned char	1	wie char, aber nur für positive Werte (0 bis 255)
short int	2	Ganzzahlwert (-32767 bis 32767)
unsigned short int	2	Ganzzahlwert, nur für positive Werte (0 bis 65535)
int	4	Ganzzahlwert (-2147483647 bis 2147483647)
unsigned int	4	Ganzzahlwert (0 bis 4294967295)
float	4	Fließkommazahl Präzision 10^{-5}
double	8	Fließkommazahl Präzision 10^{-9}

Zur Bildung von berechenbaren Ausdrücken stehen Operatoren für die Grundrechenarten Addition (+), Subtraktion (-), Multiplikation (*) und Division zur Verfügung. Die Zuweisung eines ausgewerteten Ausdrucks erfolgt mit dem = Operator. Der Zuweisungsoperator kann auch mehrfach in einem Ausdruck benutzt werden, wobei die Auswertung von rechts nach links erfolgt. Als Besonderheit von C sind Operatoren zu nennen, die Auswertung und Zuweisung in einem Schritt vornehmen und sich aus dem arithmetischen Operator und dem Zuweisungsoperator zusammensetzen (+=, -=, *=, /=) sowie die Inkrementierungs- (++) und Dekrementierungsoperatoren (--), die den Wert einer Variablen um 1 erhöhen bzw. erniedrigen. Hier einige Beispiele.

```
int    i, j;

i   = j  = 2;
i   = 1 * 2 * j;
j  *= i++;
i  += j;
```

C unterstützt die Definition komplexer Datenstrukturen, die sich aus einfachen Datentypen zusammensetzen. Die Deklaration erfolgt mit dem Schlüsselwort *struct*. Der Zugriff auf die Elemente der Datenstruktur erfolgt mit dem Punktoperator (.). In den nachfolgenden Beispielen wird das Schlüsselwort *typedef* benutzt, um die hierdurch bezeichnete Datenstruktur im Quelltext wie einen der Standarddatentypen verwenden zu können. Um zu verdeutlichen, dass es sich nicht um Variablennamen handelt, wird der Name der Struktur mit einem „S“ und der Name der Typdefinition mit „T“ begonnen.

```
struct SPunkt
{
   char     Symbol;

   double   x, y;
};

typedef struct SPunkt TPunkt;

void main(void)
{
   TPunkt   Punkt1, Punkt2;

   Punkt1.x       = 10;
   Punkt1.y       = -5;
   Punkt1.Symbol  = 'A';

   Punkt2         = Punkt 1;
}
```

Eine Besonderheit von C ist, dass die Sprache es nicht nur ermöglicht, sondern für viele Problemlösungen sogar notwendig macht, den Arbeitspeicher eigenverantwortlich zu manipulieren bzw. zu verwalten. Der Zugriff auf die Adressbereiche des Arbeitspeichers erfolgt dabei über sogenannte Zeiger (engl. *pointer*) bzw. Zeigerwerte, die im Quelltext in Zeigervariablen zwischengespeichert werden können. Das Arbeiten mit Zeigern erlaubt ein höchst effektives und maschinennahes Programmieren, führt aber gerade bei Programmieranfängern häufig zu fatalen Fehlern, da ein Zugriff auf einen falschen Speicherbereich ein Programm oft sofort abstürzen lässt. Zur Deklaration einer Zeigervariablen wird dieser ein Stern vorangestellt. Die Zeigervariable *zWert* im nachfolgenden Beispiel kann auf einen Speicherbereich

verweisen, der einen Wert vom Typ *int* repräsentiert. Um für einen beliebigen Datentyp die Speicheradresse zu erhalten, benutzt man den Adressoperator (*&*). Um in umgekehrter Weise auf den Wert zuzugreifen, der in der Adresse einer Zeigervariablen steht, benutzt man den wieder durch einen Stern dargestellten Dereferenzierungsoperator. Zeiger lassen sich auch für Datenstrukturen erstellen. Der Zugriff auf die Elemente einer Datenstruktur über einen Zeiger auf diese erfolgt aber nicht mit dem Punktoperator, sondern mit dem Pfeiloperator (`->`).

```
int      Wert, *zWert;
TPunkt   Punkt1, Punkt2, *zPunkt1;

zWert             = &Wert;
*zWert            = 1;

zPunkt1           = &Punkt1;
zPunkt1->x        = 10;
zPunkt1->x        = -5;
zPunkt1->Symbol   = 'A';

Punkt2            = *zPunkt1;
zPunkt1           = NULL;
```

Ein besonderer Zeiger ist der *NULL*-Zeiger, durch den z.B. deutlich gemacht werden kann, dass eine Zeigervariable noch nicht initialisiert wurde. Eingesetzt werden Zeiger z.B. für die dynamische Speicherverwaltung oder bei der Übergabe komplexer Datenstrukturen an Funktionen, ohne dass deren Werte kopiert werden sollen. Auch Zeichenketten werden in C durch Zeiger dargestellt, wobei diese auf ein Feld (engl. a*rray*) vom Typ *char* verweisen, das die ASCII codierten Zeichenwerte enthält.

Neben den arithmetischen Operatoren, gibt es auch Operatoren für die Auswertung von logischen Ausdrücken. Ist ein logischer Ausdruck wahr, dann wird als Ergebnis seiner Auswertung der Wert 1 zurückgeliefert, sonst 0. Die Vergleichsoperatoren < (kleiner), <= (kleiner oder gleich), > (größer), >= (größer oder gleich), == (gleich) und != (ungleich) vergleichen Werte bzw. Werteausdrücke miteinander. Der Operator ! negiert seinen Operanden, d.h. aus wahr wird falsch und umgekehrt. Das logische Und (&&) liefert wahr zurück, wenn beide Ausdrücke, die er verbindet, wahr sind, und das logische Oder (||) liefert wahr zurück, wenn einer der beiden Ausdrücke, die er verbindet, wahr ist.

Das wichtigste Einsatzgebiet der logischen Operatoren ist die bedingte Verzweigung mit Hilfe einer Auswahlanweisung. Einfache Auswahlanweisungen werden mit der *if* Abfrage programmiert. Eine nachfolgende Anweisung oder ein nachfolgender Anweisungsblock wird nur dann ausgeführt, wenn die Bedingung der *if* Abfrage wahr ist. Ein alternativer Anweisungsblock kann durch eine darauf folgende *else* Anweisung angegeben werden. Solche *if*/*else* Anweisungen lassen sich beliebig schachteln und nacheinander schalten.

```
if( Wert == 1 )
{
   Ergebnis = 1;
}
else if( Wert > 1 )
{
   if( Wert > 2 && Wert <= 3 )
      Ergebnis = 3;
   else
      Ergebnis = 2;
}
else
{
   Ergebnis = 4;
}
```

Eine Alternative, die verschiedene Auswertungsergebnisse mit innerhalb einer Anweisung berücksichtigt, ist die *switch* Anweisung. Mit der *switch* Anweisung können bedingt Bereiche in dem nachfolgenden Anweisungsblock angesprungen werden. Die Sprungziele werden mit dem Schlüsselwort *case* und einem entsprechenden Wert für das Auswertungsergebnis gekennzeichnet. Damit nicht alle folgenden Anweisungen des Blocks auch noch ausgeführt werden, kann mit *break* der *switch* Anweisungsblock wieder verlassen werden. Das Schlüsselwort *default* wird angesprungen, wenn keine *case* Anweisung dem Auswertungsergebnis der *switch* Anweisung entspricht.

```
Switch( Wert )
{
case 1:
   Ergebnis = 1;
   break;

case 2:
   Ergebnis = 2;
   break;

case 3:
   Ergebnis = 3;
   break;

default:
   Ergebnis = 4;
   break;
]
```

Mit der *break* Anweisung kann prinzipiell jeder Anweisungsblock verlassen werden. Sinnvoll ist das jedoch immer nur in Verbindung mit einer bedingten Abfrage. Bedingte Abfragen werden auch bei der Definition von Schleifenanweisungen benutzt. Die einfachste Möglichkeit, eine Schleife zu formulieren, bietet das Schlüsselwort *while*, das, wie die *if* Anweisung, einen logischen Ausdruck auswertet. Im Unterschied zur *if* Anweisung wird der nachfolgende Anweisungsblock so oft ausgeführt, bis der logische Ausdruck falsch ist. Es ist daher sinnvoll, für den logischen Ausdruck eine Variable zu benutzen, die im Verlauf der Schleifenausführung einen Wert annimmt, der zum Abbruch der Schleife führt.

```
int   i  = 0;

while( i < 10 )
{
   i++;
}
```

In diesem Beispiel wurde die Variable *i* in der Schleife so oft inkrementiert, bis sie den Wert 10 erreicht hat. Die Schleife lässt sich mit dem Schlüsselwort *do* auch so formulieren, dass die Abbruchbedingung erst nach Ausführung des Anweisungsblocks überprüft wird.

```
int   i  = 0;

do
{
   i++;
}
while( i < 10 );
```

Mit dem Schlüsselwort *for* lässt sich diese *do* Schleife noch etwas kompakter formulieren. Die *for* Anweisung hat dafür drei durch Semikola getrennte Argumente. Das erste Argument wird i.d.R. für die Initialisierung einer Schleifenvariablen benutzt, die im zweiten Argument zur Überprüfung der Abbruchbedingung nach Ausführung des Anweisungsblocks dient. Mit dem dritten Argument wird der neue Wert der Schleifenvariable nach einem Schleifendurchlauf berechnet. In dem Beispiel werden alle Zahlen von 0 bis 9 summiert.

```
int   i, j  = 0;

for(i=0; i<10; i++)
{
   j  += i;
}
```

Damit eine bestimmte Funktion von einer beliebigen Stelle eines Programms aufgerufen werden kann, muss bekannt sein, wie sie heißt und welche Funktionsparameter sie erwartet. In dem Beispiel zu Beginn dieses Kapitels kann die *main* Funktion die Funktion zum Verdoppeln eines Wertes aufrufen, weil diese im Quelltext vorher definiert wurde. Um auch komplexere Programme übersichtlich entwickeln zu können, kann man die Definition einer Funktion von ihrer Deklaration trennen. Die Funktionsdeklaration besteht einfach aus dem Funktionsrumpf. Das Eingangsbeispiel lässt sich so auch folgendermaßen schreiben.

```
int Verdoppeln(int i);

void main(void)
{
   int   i;

   i  = Verdoppeln(5);
}

int Verdoppeln(int i)
{
   return( 2 * i );
}
```

Da für die Verwendung einer Funktion nur ihre Funktionsdeklaration bekannt sein muss, kann die Implementierung der Funktion auch in einer separaten Datei erfolgen. Separate

Quelltextdateien werden auch als Moduln bezeichnet. Der Compiler sorgt beim Erstellen des Programms mit dem sogenannten *Linking* dafür, dass die in Maschinencode übersetzten Funktionen zu einem Programm zusammengeführt werden. Damit eine Funktion von verschiedenen Quelltextmoduln aus aufgerufen werden kann, ist es in C üblich, die Funktionsdeklarationen in eigene Dateien, die sogenannten *Header*, zu schreiben. Die Header bieten sich auch dafür an, die Definitionen von Datentypen sowie Präprozessoranweisungen aufzunehmen, die in dem Programm benutzt werden. Der C-Compiler wertet, wie bereits der Name andeutet, Präprozessoranweisungen vor der eigentlichen Kompilierung aus. Eine wichtige Präprozessoranweisung betrifft die Einbindung von Header-Dateien mit der *#include* Direktive.

```
#include “kopf.h“

void main(void)
{
   int   i;

   i  = Verdoppeln(5);
}
```

Im diesem Beispiel wird die Header-Datei „*kopf.h*“ eingebunden, die z.B. die Deklaration der Funktion „*Verdoppeln*“ enthält. Eine häufig benutzte Anweisung für den Präprozessor ist die Definition von Makros mit der *#define* Direktive. Makros dienen u.a. als Platzhalter für andere, frei wählbare Zeichenketten, und werden während der Präprozessierung durch diese ersetzt. Ein typisches Beispiel ist die Definition der Zahl π.

```
#define PI  3.14159265358979323846264338327795

void main(void)
{
   double   x;

   x  = 2 * PI;
}
```

Sehr nützlich ist die Möglichkeit, Kommentare in den Quelltext einfügen zu können, besonders wenn mehrere Programmierer an der Programmentwicklung beteiligt sind. Einzelne Kommentarzeilen werden durch zwei Schrägstriche als Kommentar angeführt (*//*). Ein Kommentar kann sich über mehrere Zeilen erstrecken, wenn er durch die Zeichenfolge */** eingeleitet und mit ***/* wieder geschlossen wird.

```
// Ein einzeiliger Kommentar

void main(void)
{
   double   x;    /* Definition der Variablen x */

   x  = 2 * PI;   // x entspricht jetzt dem Kreisumfang
}

/* Ein Kommentar kann sich
   auch über mehrere Zeilen
   erstrecken
*/
```

Bis hierher wurden bereits alle wesentlichen Elemente der Sprache C selbst vorgestellt. Ihre besondere Stärke erlangt die Sprache jedoch erst durch die Standardbibliothek, die zum Umfang der meisten C Compiler dazu gehört. Die Standardbibliothek bietet zahlreiche Funk-

tionen an, z.B. für das Arbeiten mit Zeichenketten, die dynamische Speicherverwaltung, die Abfrage der Tastatur, die Bildschirmausgabe von Texten oder das Lesen und Schreiben von Dateien. Zum Verwenden dieser umfangreichen Bibliothek müssen im wesentlichen nur die entsprechenden Headerdateien mit den benötigten Funktionsdeklarationen eingebunden werden. Im Linker-Prozess werden die dahinter stehenden Funktionen vom Compiler bei der Programmerstellung in das Programm eingefügt. Das kleine Beispiel verwendet die Funktion *printf*, um einen einfachen Text auf einer Textkonsole auszugeben.

```
#include <stdio.h>   // Funktionen für die Ein-/Ausgabe

void main(void)
{
   printf(“Hallo Welt“);   // Ausgabe von Text auf den Bildschirm
}
```

2.1.3 Die objektorientierte Sprache C++

Die Programmiersprache C++ (sprich: C plus plus) baut auf C auf und erweitert deren Sprachumfang u.a. um Konzepte, die eine objektorientierte Programmierung ermöglichen. Da C Quelltexte vollständig kompatibel zu C++ sind, spricht man oft auch von C/C++. Viele der neuen Konzepte in C++ zielen darauf ab, die Programmierung effektiver und sicherer zu machen. Die Wiederverwendbarkeit einmal erstellter Programmmoduln bildet dabei einen wesentlichen Schwerpunkt. Entwickelt wurde C++ von B.Stroustrup während der achtziger Jahre. Sein grundlegendes Buch über die C++ Programmiersprache ist nach wie vor eine der umfangreichsten Quellen zu dem Thema (STROUSTRUP 1995), wurde aber sehr rasch durch zahlreiche Einführungen und Fachbücher für verschiedene Zielgruppen mit unterschiedlichen Niveaus ergänzt (z.B. DEWHURST & STARK 1990, FISCHER & AHRENS 1996).

Die wichtigste Neuerung gegenüber C ist die Einführung eines Klassenkonzeptes, auf dem die Fähigkeiten zur Objektorientierung der Sprache letztlich beruhen. Klassen können als objektorientierte Erweiterung der in C mit dem Schlüsselwort *struct* definierbaren Datenstrukturen aufgefasst werden und werden mit dem Schlüsselwort *class* deklariert. Um zu verdeutlichen, dass es sich bei dem Datentyp um eine Klasse handelt, wird in den nachfolgenden Beispielen Klassennamen immer ein „C“ vorangestellt. Das erste Beispiel erstellt eine sehr einfache Klasse.

```
class CPunkt
{
public:
   CPunkt(void)   {  x  = 0.0;   y  = 0.0;   }

   double   x, y;
};
```

Die Klasse *CPunkt* enthält zwei Variablen vom Datentyp *double*. Zusätzlich besitzt sie einen sogenannten Konstruktor. Konstruktoren sind besondere Klassenfunktionen, die immer denselben Namen wie die Klasse haben und keinen Rückgabewert liefern. Der Konstruktor wird automatisch aufgerufen, wenn ein Objekt vom Typ der Klasse instanziiert wird. Er eignet sich also für die Initialisierung der Datenmitglieder der Klasse. Das Schlüsselwort *public* erklärt, dass Quelltexte, die die Klasse benutzen, auf die nachfolgenden Funktionen

und Variablen, wie in dem nächsten Beispiel gezeigt, direkt zugreifen können. Der Klassenname wird dabei wie ein einfacher Datentyp verwendet, um die Objektinstanz *Punkt* der Klasse *CPunkt* zu erstellen.

```
   CPunkt Punkt;

   Punkt.x  =  1.0;
   Punkt.y  = -5.0
```

Mit dem Schlüsselwort *private* lassen sich direkte Zugriffe auf die Daten unterbinden. Um trotzdem auf die Datenwerte zugreifen zu können, muss die Klasse spezielle Zugriffsfunktionen bereitstellen. Eine solche Kapselung der Daten erlaubt z.B. zu überprüfen, ob Datenwerte, die zugewiesen werden sollen, sinnvoll sind. In dem folgenden Beispiel wird der lesende Zugriff auf die Datenmitglieder durch die Funktionen *Gib_x*/*Gib_y* ermöglicht. Mit dem Überladen von Funktionen verwendet das folgende Beispiel gleich noch eine weitere Erneuerung von C++. Durch das Überladen ist es möglich, Funktionen oder Operatoren kontextabhängig mit unterschiedlichen Argumenten aufzurufen. So kann der Konstruktor jetzt einmal ohne Argumente aufgerufen werden und einmal mit der Übergabe von zwei Werten, die für die Initialisierung der Klasse benutzt werden.

```
class CPunkt
{
public:
   CPunkt(void)                         {  x  = 0.0;   y  = 0.0;    }
   CPunkt(double xInit, double yInit) {  x  = xInit; y  = yInit;  }

   double   Gib_x(void)    {  return( x );   }  // Abfrage des Datenwerts x
   double   Gib_y(void)    {  return( y );   }  // Abfrage des Datenwerts y

   void     Versetze(double xVersatz, double yVersatz)
   {
      x  += xVersatz;
      y  += yVersatz;
   }

   void     Versetze(const CPoint &Versatz);
   {
      x  += Versatz.Gib_x();
      y  += Versatz.Gib_y();
   }

private:
   double   x, y; // auf die Datenmitglieder x, y kann nicht direkt zugegriffen werden!
};
```

Auch die Funktion *Versetze* wurde überladen. Während die erste Version zwei Werte vom Typ *double* verlangt, erwartet die zweite Version als Argument ein Objekt, das ebenfalls vom Typ der Klasse *CPunkt* ist. Der hier verwendete *&* Operator bewirkt bei Funktionsargumenten, dass eine Referenz auf die Variable und nicht eine Kopie von ihr übergeben wird, wie sonst bei C üblich. Das ist besonders bei Objekten wichtig, da dies sonst schnell zu Problemen führen kann, z.B. wenn das Objekt einen großen Datensatz repräsentiert. Auch lassen sich über Referenz übergebene Objekte von der Funktion direkt manipulieren. Die Funktion *Versetze* macht in diesem Fall aber gleich durch das Schlüsselwort *const* deutlich, dass es das übergebene Objekt nicht verändert, sondern seine Werte oder Methoden nur abfragt. Die Verwendung der Klasse kann so aussehen.

```
   CPunkt   A, B(2.0, 3.2);

   A.Versetze(4.5, -1.7);
   B.Versetze(A);
```

Das Überschreiben von Operatorfunktionen kann die Verwendung einer Klasse sehr viel anschaulicher machen, wie die folgende, etwas andere Definition der Klasse *CPunkt* demonstriert. Die hier nebenbei aufgezeigte Verwendung von Standardwerten für die Funktionsargumente des Konstruktors hat einen ähnlichen Effekt wie die Überladung. Eine Funktion kann dadurch mit oder ohne Angabe der Argumente, für die Standardwerte definiert sind, aufgerufen werden.

```
class CPunkt
{
public:
   CPunkt(double xInit = 0, double yInit = 0)
   {
      x  = xInit;
      y  = yInit;
   }

   void     operator += (CPunkt Punkt)
   {
      x  += Punkt.x;
      y  += Punkt.y;
   }

   CPunkt   operator - (CPunkt Punkt)
   {
      return( CPunkt(x - Punkt.x, y - Punkt.y) );
   }

   double   x, y;
};
```

Die Verwendung der Klasse gestaltet sich dadurch sehr übersichtlich.

```
   CPunkt   A(3, -2), B(2, 2), C;

   C  = A - B;
   A  += C;
```

Wirklich mächtig wird das Klassenkonzept aber erst durch die Möglichkeit der Vererbung. Durch die Vererbung kann eine Klasse die Eigenschaften einer anderen übernehmen und diese um neue, spezialisierte Methoden und Datenmitglieder erweitern oder vorhandene Methoden modifizieren. Die Klasse, deren Eigenschaften geerbt werden sollen, werden dazu durch einen Doppelpunkt getrennt der einleitenden Klassendeklaration nachgestellt. Es kann hierbei spezifiziert werden, ob die geerbten Eigenschaften öffentlich zugänglich sein sollen, oder nicht. Um Eigenschaften nach außen hin zu verbergen, es abgeleiteten Klassen jedoch zu erlauben, auf diese zuzugreifen, werden diese nicht als *private*, sondern als *protected* deklariert. Damit eine abgeleitete Klasse eine Methode der Basisklasse mit einer modifizierten Version überschreiben kann, muss die Methode mit dem Schlüsselwort *virtual* als überschreibbar gekennzeichnet sein. Das sehr einfach gehaltene Beispiel veranschaulicht die soeben erwähnten Mechanismen der Vererbung.

```
class CRechteck
{
public:
   CRechteck(double xA = 0, double yA = 0, double xB = 1, double yB = 1)
   {
      A.x   = xA; A.y   = yA;    B.x   = xB; B.y   = yB;
   }

   virtual double Flaeche  (void)
   {
      return( (A.x - B.x) * (A.y - B.y) );
   }

   CPunkt         Zentrum  (void)
   {
      return( CPunkt(A.x + (B.x - A.x), A.y + (B.y - A.y)) );
   }

protected:
   CPunkt   A, B;
};

class CDreieck : public CRechteck
{
public:
   CDreieck(double xA=0, double yA=0, double xB=1, double yB=1) : CRechteck(xA, yA, xB, yB)
   {
      // keine Anweisung
   }

   virtual double Flaeche  (void)
   {
      return( CRechteck::Flaeche() / 2.0 );
   }
};
```

Der Konstruktor von *CDreieck* selbst führt hier keine Anweisung aus, ruft aber den mit einem Doppelpunkt nachgestellten Konstruktor von *CRechteck* auf, um diesen die eigentliche Initialisierung vornehmen zu lassen. Danach wird die Funktion *Flaeche* von *CDreieck* neu definiert. Sie benutzt dabei die gleichnamige Funktion ihrer Basisklasse und verändert deren Rückgabewert, bevor sie ihn selbst zurückgibt. Um sicher zu stellen, dass die Funktion der Basisklasse aufgerufen wird, wird ihr Name getrennt durch zwei Doppelpunkte dem Funktionsnamen vorangestellt. Da die Funktion *Zentrum* nicht als *virtual* deklariert wurde, kann sie von *CDreieck* auch nicht überschrieben werden.

In den bisherigen Beispielen wurden die Objektinstanzen wie einfache Datentypen erklärt und verwendet. Objekte, die auf diese Weise in einem Anweisungsblock definiert werden, sind lokal zu diesem und werden automatisch zur Laufzeit beim Betreten des Anweisungsblocks (z.B. beim Aufruf der entsprechenden Funktion) erzeugt und beim Verlassen des Blocks wieder aus dem Speicher entfernt. Alternativ können Objekte zur Laufzeit dynamisch auf explizite Nutzeranforderung erzeugt werden. Um eine Objektinstanz dynamisch zu erstellen, wird der *new* Operator benutzt, der einen Zeiger auf das neu erstellte Objekt liefert und auch automatisch seinen Klassenkonstruktor aufruft. Danach liegt es allerdings in der Verantwortung des Programmierers, Objekte, die er mit *new* angefordert hat, mit dem *delete* Operator wieder aus dem Speicher zu löschen. Wie bei Datenstrukturen erfolgt der Zugriff auf die Eigenschaften von Klassen bei Zeigern mit dem Pfeiloperator. Die Verwendung von Zeigern auf Objekte führt in Zusammenhang mit Objektklassenhierarchien zu einem weiteren Aspekt der objektorientierten Programmierung, der Polymorphie. Die Polymorphie erlaubt es Instanzen verschiedener Objektklassen über ihre gemeinsame Basisklasse, so sie sich denn eine teilen, anzusprechen. Hierdurch können gleichbedeutende Funktionen für verschieden-

artige Objekte benutzt werden. Im Beispiel besitzen die Klassen *CRechteck* und *CDreieck* die gleiche Funktion *Flaeche*, die jedoch unterschiedlich implementiert wurde.

```
double      Flaeche;
CPunkt      Zentrum;
CRechteck   Rechteck, *Zeiger;
CDreieck    Dreieck;

Zeiger   = new CRechteck(-1, -1, 1, 1);
Zentrum  = Zeiger->Zentrum();
delete Zeiger;

Zeiger   = &Dreieck;
Flaeche  = Zeiger->Flaeche();

Zeiger   = &Rechteck;
Flaeche  = Zeiger->Flaeche();
```

In diesem Kapitel wurden einige grundlegende Konzepte der Programmiersprache C++ vorgestellt, die aber nur einen Einblick in alle Möglichkeiten der Sprache geben können. Die Entwicklung komplexer Objektmodelle verlangt ein gewisses Maß an Erfahrung. Darüber hinaus ist es zur Zeit des Modellentwurfs selten möglich, alle Anforderungen abzuschätzen, die an ein Objektklassenmodell zu einem zukünftigen Zeitpunkt gestellt werden. Auch kann aus einer Problembeschreibung nicht in jedem Fall der geeignetste Objekttyp für ein Programm bestimmt werden. Die Festlegung einer Abstraktion, die einer Reihe von Objekten gemein ist, ist nicht unbedingt ein geradliniger Prozess, so dass es gerade für den ungeübten Programmierer viele Versuche kosten wird, ein nachhaltig einsetzbares Objektmodell zu implementieren. (DEWHURST & STARK 1992). Auf der anderen Seite kann die Verwendung eines bestehenden, etablierten Objektmodells die Entwicklung neuer Programme durch die Bereitstellung intuitiv zugänglicher Objektklassen erheblich vereinfachen. Gerade hier können auch ungeübte Programmierer schnell zu effektiven Ergebnissen kommen, wie in Kap. 3.7 am Beispiel der SAGA-API gezeigt wird.

2.1.4 Betriebssysteme, Benutzeroberflächen, Compiler

Ein Betriebssystem (OS, von engl. *operating system*) ist eine spezielle Software, die einerseits die Hardware eines Computers verwaltet und andererseits die Ausführung von Anwendungsprogrammen steuert. Die verbreitetsten Betriebssysteme für Personal Computer (PC), die sich durch ihre geringere Leistung von den vorwiegend für Spezialaufgaben eingesetzten Groß- und Superrechnern unterscheiden, sind die verschiedenen Versionen von Microsoft Windows, Linux und MacOS. Zum Aufgabenbereich eines Betriebssystems gehört die Steuerung des Datenflusses zwischen Hardware und Software, die Bereitstellung von Betriebssystemfunktionen für Anwendungsprogramme und die Kommunikation mit dem Anwender durch eine Benutzerschnittstelle. Nahezu alle modernen Betriebssysteme für PCs bieten graphische Benutzeroberflächen (GUI, von engl. *graphical user interface*) an, die eine intuitive und komfortable Bedienung ermöglichen. Typische Bedienelemente sind Befehlsmenüs und Werkzeugleisten mit graphischen Symbolen sowie die Unterstützung der Mauszeigerfunktion.

Die Übersetzung von C/C++ Quelltexten in ausführbare Programme erfolgt durch einen entsprechenden, für die Sprache entwickelten Compiler. Dank ihrer Popularität und weiten Verbreitung stehen zahlreiche C/C++ Compiler für nahezu jedes moderne Betriebssystem zur Verfügung. Ein Grund hierfür ist sicher die Tatsache, dass viele Betriebssysteme, wie Windows, UNIX und Linux, selbst zum größten Teil in C/C++ implementiert wurden. Dank der betriebssystemspezifischen Umsetzung der Standardbibliothek (Kap. 2.1.2), die ein integrierter Bestandteil der meisten C/C++ Compiler ist, sind C/C++ Quelltexte prinzipiell portierbar. D.h., dass Programme, die nur die Standardbibliothek verwenden, auf den verschiedenen Betriebssystemen kompiliert und ausgeführt werden können. Schwierigkeiten der Portabilität ergeben sich dann, wenn betriebssystemabhängige Funktionen, die nicht von der Standardbibliothek abgedeckt werden, in den Quelltexten benutzt werden. Dies ist besonders dann der Fall, wenn die graphischen Fähigkeiten des Betriebssystems für die Erstellung einer Programm-GUI benutzt werden, da die verschiedenen Betriebssysteme diesbezüglich sehr große Unterschiede aufweisen. Für den Zugriff auf graphische Funktionalitäten bieten die Betriebssysteme Bibliotheken mit einer speziellen Anwendungsprogrammierschnittstelle (API, von engl. *application programming interface*) an. Durch die Verwendung einer solchen Bibliothek verliert ein Quelltext jedoch seine Portierbarkeit auf andere Betriebssysteme. So ist SAGA in Version 1, bedingt durch die Verwendung der Windows API, nur unter Windows Betriebssystemen lauffähig. Damit ein Programm mit einer graphischen Oberfläche trotzdem portierbar bleibt, gibt es jedoch die Möglichkeit, nicht direkt auf die API-Funktionen des Betriebssystems zu zugreifen. Als Alternative gibt es eine Reihe betriebssystemübergreifender Graphikbibliotheken, die den direkten Zugriff auf die API-Funktionen für mehrere Betriebssysteme kapseln. Durch die Verwendung der plattformübergreifenden GUI-Bibliothek *wxWidgets* (SMART et al. 2005, WXWIDGETS 2006) steht SAGA ab Version 2 derzeit sowohl für Windows als auch für Linux Betriebssysteme zur Verfügung.

Der Compiler eignet sich nicht nur dazu, ausführbare Programme zu erstellen und dabei gegebenenfalls mit anderen Programmbibliotheken zu verbinden, die zusätzliche Funktionalitäten bieten, wie z.B. die C Standardbibliothek oder wxWidgets. Er kann auch zur Erstellung neuer Bibliotheken benutzt werden, die dann selbst wieder von anderen Programmen benutzt werden können. Die Erstellung einer neuen Bibliothek bietet sich z.B. an, um nützliche und häufig gebrauchte Funktionen für verschiedene Programme zur Verfügung zu stellen, ohne dass diese die Quelltexte übernehmen müssen, oder um ein komplexes Programmsystem durch eine Aufteilung seiner Funktionen übersichtlicher zu strukturieren. Es gibt zwei Typen von Programmbibliotheken, die sich darin unterscheiden, wie ein Programm auf die benutzten Funktionalitäten zugreift.

- Bei statischen Bibliotheken werden die benutzten Funktionen komplett in den Maschinencode des erstellten Programms eingefügt.
- Dynamische Bibliotheken werden von dem erstellten Programm bei seiner Ausführung geladen, um Zugriff auf die benutzten Funktionen zu erhalten.

Durch die Verwendung statischer Bibliotheken wird die erstellte Programmdatei größer, aber das Programm kommt dafür ohne die Installation zusätzlicher Bibliotheksdateien aus, wie es bei Verwendung dynamischer Bibliotheken erforderlich ist. Der Einsatz dynamischer Bibliotheken ist besonders dann sinnvoll, wenn viele andere Programme oder Programmbibliotheken auf ihre Funktionalitäten zugreifen, wie es z.B. oft für API-Bibliotheken der Fall ist. Ein weiterer Vorteil ist, dass eine dynamische Bibliothek bei Einhaltung ihrer Schnittstellen, also der Art und Aufrufkonvention ihrer Funktionen, austauschbar ist und, dass sie auch bedingt geladen bzw. entladen werden kann. Auf diesen Vorteil basiert auch das Modulkonzept von SAGA (Kap. 3.2). Dynamische Bibliotheken werden mit leichten Unterschieden von den meisten Betriebssystemen unterstützt. Unter Windows Betriebssystemen werden sie als Dynamic Link Libraries (DLL), unter UNIX und Linux als Shared Objects (SO) bezeichnet.

Um einem Compiler mitzuteilen, welche Quelltextdateien er in ein maschinenlesbares Programm zu übersetzen hat, werden üblicherweise sogenannte *Makefiles* benutzt, in denen alle Quelltexte aufgelistet sind, aber auch zusätzliche Anweisungen, durch die sich das Verhalten des Compilers steuern lässt. Hierzu gehören z.B. Optimierungen des Maschinencodes und ob ein ausführbares Programm oder eine Programmbibliothek erstellt werden soll. Auch externe Bibliotheken, die von dem Programm benutzt werden, müssen hier angegeben werden. Die genaue Syntax solcher Makefiles ist vom jeweils benutzten Compiler abhängig und die Pflege komplexer Projekte gestaltet sich mitunter sehr aufwändig. Eine erhebliche Vereinfachung bieten die heutzutage verbreiteten Integrierten Entwicklungsumgebungen (IDE, von engl. *integrated development environment*), die über eine graphische Benutzerschnittstelle für die Verwaltung von Quelltextprojekten, das Editieren der Quelltexte, die Kompilierung und auch die Fehlersuche anbieten.

2.1.5 Open Source Software

Der Begriff der Free Open Source Software (FOSS) ist seit den 1990er Jahren immer stärker in das Zentrum öffentlichen Interesses gerückt, nicht zuletzt, weil FOSS eine transparente und kostengünstige Alternative zu den sich immer stärker monopolisierenden Anbietern kommerzieller Software bieten. Als prominente Beispiele für monopolartige Software können hier die Windows Betriebssysteme der Firma Microsoft oder die GIS-Software der Firma ESRI („*The World's GIS Leader*“, ESRI 2006) angeführt werden, die u.a. durch gezielte Firmenpolitik eine hohe Abhängigkeit der Nutzergemeinde von ihren quelltextgeschlossenen Produkten erreichen konnten. Eine Vielzahl von Studien hat sich mittlerweile mit ökonomischen, juristischen und soziologischen Auswirkungen von FOSS beschäftigt (z.B. LUTTERBERG et al. 2005). FOSS wird von der Free Software Foundation (FSF), einer Dachorganisation, die sich der Förderung freier Software verschrieben hat, als Software definiert, die für ihre Verwendung vier Freiheiten garantiert (FREE SOFTWARE FOUNDATION 2004):

- Die Freiheit, das Programm für jeden Zweck auszuführen.

- Die Freiheit, die Funktionsweise eines Programms zu untersuchen, und es an seine Bedürfnisse anzupassen
- Die Freiheit, Kopien weiterzugeben und damit seinem Nachbarn zu helfen
- Die Freiheit, ein Programm zu verbessern, und die Verbesserungen an die Öffentlichkeit weiterzugeben, sodass die gesamte Gesellschaft profitiert.

Quelltextoffene Software, die diese Freiheiten nicht durch die Verwendung einer entsprechenden Lizenz garantiert, ist zwar eine Open Source Software (OSS), wird von der FSF aber nicht als frei angesehen, da sie auch in quelltextgeschlossene, proprietäre Software legal einfließen kann. Es ist aber anzumerken, dass eine kommerzielle Verwertung von FOSS hierdurch in keiner Weise eingeschränkt ist. Die zweite Freiheit impliziert, dass FOSS quelltextoffen verfügbar ist. Für den beachtlichen Erfolg von FOSS entscheidend ist auch der letzte Punkt, durch den eine qualitative Verbesserung einer freien Software erst ermöglicht wird. Für den Nutzer von Software gibt es also eine Reihe von Gründen sich für eine (F)OSS Lösung zu entscheiden. Das stärkste Gegenargument, dass oft angeführt wird, ist, dass die Weiterentwicklung einer OSS in erster Linie auf unbezahlte und freiwillige Programmierer angewiesen ist und kein kommerzieller Anbieter für Support-Dienstleistungen in Anspruch genommen werden kann. Erfolgreiche OSS Projekte haben aber gezeigt, dass das nicht der Fall sein muss. So haben sich eine Reihe von Dienstleistungsanbietern rund um das freie Betriebssystem Linux etabliert, durch die wiederum die Entwicklung von Linux und unter Linux lauffähiger Software gefördert wird. Im Bereich von Netzwerkservern bietet Linux bereits seit Jahren eine konkurrenzfähige Alternative zu kommerziellen Lösungen (GRASSMUCK 2002). Mittlerweile führt die Abwägung zwischen den Vorteilen freier und unfreier Software zunehmend auch bei öffentlichen Institutionen und Behörden zu einer Entscheidung für OSS Lösungen (z.B. PORTAL MÜNCHEN BETRIEBS-GMBH 2006).

Nachdem die Vorteile für den Anwender auf der Hand zu liegen scheinen, stellt sich die Frage, was sich der Softwareentwickler von einer Freigabe seiner Quelltexte verspricht. In der Regel startet ein OSS Projekt ohne auf eine ausgebaute Infrastruktur für seine Vermarktung zurückgreifen zu können, obwohl der Aufbau eines Dienstleistungsangebots, wie im Fall von SAGA, eine lohnenswertere Alternative zur direkten kommerziellen Vermarktung einer Software sein kann. Es gibt jedoch auch verschiedene individuelle und soziale Beweggründe, die einen Programmierer dazu bringen eine Software frei zu geben oder sich an einem OSS-Projekt unentgeltlich zu beteiligen. Hierzu zählen der soziale Kontakt mit Leuten, die dieselben Ideen und Interessen haben, die intellektuelle Herausforderung, die Kreativität, aber auch der dadurch entstehende Ruhm (HELMERS & SEIDLER, 1995). Hinzu kommt in den meisten Fällen ein gewisser Idealismus, der auch schon in der Formulierung der vierten Freiheit der FOSS-Definition anklingt. Ohne die eben angeführten Motivationen in Frage zu stellen, ist der Ursprung freier Software oft doch pragmatischerer Natur. So hat es freie Software im Prinzip seit Beginn der Computerentwicklung gegeben. Vor allem im Bereich wissenschaftlicher Software ist es immer verbreitet gewesen, neu entwickelte Algorithmen als

Quelltext offen zu legen. Dies entspricht allein schon dem wissenschaftlichen Grundgedanken, neue Denkansätze, Modelle oder Thesen der Wissenschaftsgemeinde zugänglich zu machen, so dass sie überprüft, angewendet und verbessert (oder verworfen) werden können. Von einigen Autoren wird die Open-Source-Bewegung sogar dem wissenschaftlichen System als selbständiges System an die Seite gestellt (GÖRLICH & HUMBERT 2005). Ein weiterer wichtiger Grund, der bis jetzt ausgeklammert wurde, betrifft die rechtliche Absicherung von Software durch eine Softwarelizenz.

Die Aufgabe einer Softwarelizenz ist in erster Linie der Schutz der Rechte des Programmierers, insbesondere wenn es sich dabei um eine quelltextoffene Software handelt. Auch wenn nach deutschem Recht, im Gegensatz zum anglikanischen, das Urheberrecht eines Programmierers an seinen Quelltexten immer erhalten bleibt, empfiehlt es sich, darauf in den Quelltexten hinzuweisen. Ebenfalls sollte betont werden, dass keine Haftung für irgendwelche Programmierfehler übernommen wird und keine sonstigen Ansprüche gegenüber dem Programmierer geltend gemacht werden. Es gibt eine Vielzahl rechtlich überprüfter Lizenzmodelle für OSS (GRASSMUCK 2002, FREE SOFTWARE FOUNDATION 2006), die entsprechende Formulierungen enthalten, und diese gegebenenfalls um zusätzliche Punkte ergänzen. Die populärste OSS Lizenz ist die von R.Stallmann ins Leben gerufene GNU General Public License (GPL), auf die im wesentlichen auch die oben erwähnte FOSS Definition der FSF beruht (GNU 2006). Die von der GPL garantierten Freiheiten sind jedoch etwas strenger formuliert. Die wichtigste Bedingung besteht darin, dass die von einer unter der GPL stehenden Software abgeleiteten Programme ebenfalls unter die GPL gestellt werden müssen. Dieser Mechanismus der automatischen Lizenzvererbung wird *Copyleft* genannt (im Gegensatz zum Copyright). Ziel dieser von ihren Gegnern häufig als infektiös bezeichneten Klausel ist es, eine Privatisierung von kollektiv erzeugtem Wissen zu verhindern und den Gesamtbestand an freier Software beständig zu erweitern (GRASSMUCK 2002). Das Copyleft betrifft auch Programme, die eine unter der GPL stehende Bibliothek benutzen. Um nun die Hemmschwelle für Hersteller proprietärer Software zu senken, FOSS Bibliotheken zu verwenden, was ihrer weiteren Verbreitung entgegenstünde, wurde eine abgeschwächte Version der GPL, die GNU Lesser General Public License (LGPL), entworfen, die bis auf das Copyleft der GPL entspricht. Zur bekanntesten GPL-Software gehört das bereits mehrfach erwähnte Betriebssystem Linux. SAGA verwendet neben der GPL auch die LGPL, um die Grundfunktionalitäten der Programmierschnittstelle für proprietäre Softwareentwicklungen zu ermöglichen (Kap. 3.2).

2.2 Geoinformatik

Die Geoinformatik beschäftigt sich mit den besonderen Eigenschaften raumbezogener Information, die in Form von Geodaten gespeichert und verarbeitet wird (BARTELME 1995). Die Datenverwaltung erfolgt dabei im allgemeinen mit Hilfe computerbasierter Geographischer Informationssysteme (GIS), für deren Aufbau die Geoinformatik die theoretischen Grundlagen liefert. Die Entwicklung von Geoinformatik und GIS wurde stark von der

Entwicklung der Computer- und Informationstechnologie (IT) beeinflusst. Mit der zunehmenden Verfügbarkeit von PC-Systemen Ende der 1980er konnte sich auch die GIS-Technologie als neues Werkzeug in den raumbezogen arbeitenden Wissenschaften, insbesondere der Geographie, etablieren. Nicht zuletzt durch die rasante Entwicklung des Internets bedingt gewinnen in den letzten Jahren vor allem für Verwaltungsaufgaben und Projekte mit hohem Datenaufkommen netzwerkverteilte Informationssysteme gegenüber Installationen auf Einzelcomputern eine immer größere Bedeutung. Schwerpunkte in der aktuellen Diskussion sind daher auch die Integration von GIS-Funktionalitäten in Datenbank-Managementsysteme (DBMS), z.B. durch eine funktionale Erweiterung der Datenbankabfragesprache SQL (Structured Query Language), und die Festlegung offener Standards für den Software-unabhängigen Zugriff auf netzwerkverteilte Geodaten (OGC 2006). Ohne auf diese neueren Aspekte der Geoinformatik oder auf spezifische Softwarelösungen einzugehen, konzentriert sich dieses Kapitel auf die für die SAGA-Entwicklung nötigen theoretischen Grundlagen für die Verwaltung von Geodaten, die möglichen Datenstrukturen für ihre Speicherung sowie deren Eignung für raumbezogene Analysen, wie sie z.B. in dem Lehrbuch von BARTELME (1995) ausführlich dargestellt werden (s.a. BURROUGH & MCDONNEL 1998, KAPPAS.2001, MCCLOY 2006).

2.2.1 Geographische Informationssysteme

Wie bereits erwähnt erfolgt die Verwaltung von Geodaten durch sogenannte Geographische Informationssysteme oder Geoinformationssysteme, wobei verschieden umfangreiche Definitionen mit unterschiedlichen Schwerpunkten für diese im Umlauf sind. Häufig wird die Software, die für die Datenverwaltung benutzt wird, für sich allein genommen als GIS bezeichnet. Insbesondere die Softwarehersteller scheinen darauf zu setzen, mit diesem Schlagwort mehr Kundenaufmerksamkeit zu erzielen. Auch für die Internetpräsenz von SAGA wird der Namenszusatz GIS verwendet, um z.B. bei entsprechenden Anfragen in der Ergebnisliste von Internetsuchmaschinen aufzutauchen. Trotzdem schließt sich diese Arbeit einer abweichenden, umfassenderen Definition an, bei der die Daten als zentrales Element eines Informationssystems aufgefasst werden. Die Argumentation beruht u.a. darauf, dass sich ohne Daten keine Information gewinnen oder darstellen lassen. Die Softwarelösung für sich wird zur Verdeutlichung des Unterschieds als GIS-Software bezeichnet.

In einer allgemeineren Definition dient ein Informationssystem der Erfassung, Verwaltung, Analyse und Präsentation von Daten (Abb. 8). Generell erfüllen die meisten Datenbanken die Anforderungen an ein Informationssystem gemäß der eben gegebenen Definition. Im Falle eines GIS wird impliziert, dass die Daten einen direkten Raumbezug besitzen oder dass ein Raumbezug durch Verknüpfung mit anderen Daten indirekt hergestellt werden kann. Daher muss ein GIS auch spezielle, auf die räumliche Natur der Daten abgestimmte Methoden für die gestellten Aufgaben anbieten. In diesem Sinne lassen sich GIS auch als um Raumbezug und räumliche Funktionen erweiterte Datenbanken verstehen. Diese Sichtweise bestätigt sich auch durch die immer besser werdenden Anbindungsmöglichkeiten von GIS-Software an

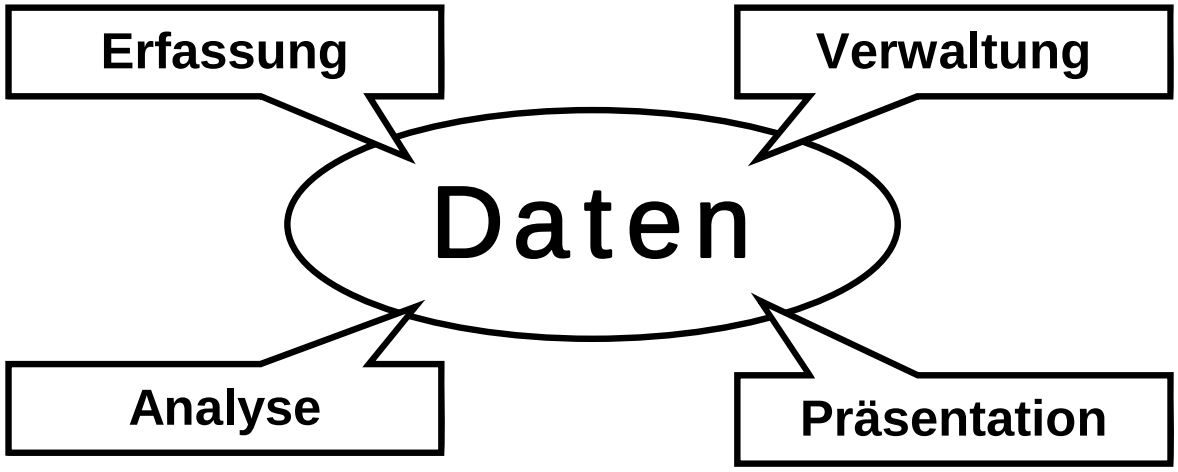


Abb. 8: Aufgaben eines Informationssystems.

bestehende DBMS bzw. SQL-fähige Datenbanken. Trotzdem erfolgt die Datenverwaltung durch die meiste GIS-Software immer noch in erster Linie dateibasiert. Unabhängig davon, wie die Datenspeicherung verwaltet wird, werden für die Abbildung des Raumbezugs besondere Datenstrukturen benötigt. Bevor jedoch auf Datenstrukturen für Geodaten eingegangen wird, sollen zunächst wichtige Konzepte und Funktionalitäten von GIS angesprochen werden.

Grundelement der Speicherung von Geodaten ist die geographische Position in Form von Punktkoordinaten. Für sich allein genommen sind Punktkoordinaten aber nicht aussagekräftig. Sie müssen zusätzlich inhaltlich mit Attributdaten verknüpft sein, die der Position eine Eigenschaft zuordnen. Man unterscheidet in diesem Zusammenhang zwischen Geometrie- und Sachdaten. Die Digitalisierung von Punktkoordinaten kann in unterschiedlicher Weise erfolgen. Sie ist abhängig von dem zu erfassenden Attributmerkmal und bedingt auch die zu verwendende Datenstruktur. So können die Daten z.B. punktweise mit einem Digitalisierbrett oder einem GPS-Empfänger aufgenommen werden. Die Attributdaten werden danach in einem zweiten Schritt zugewiesen werden. Sogenannte Scanner, wie sie z.B. viele Satelliten benutzen, erfassen ein Attributmerkmal flächendeckend und liefern als Ergebnis eine Bilddatei, in der die Attributdaten regelmäßig angeordneten Koordinatenpunkten zugeordnet sind.

Ein verbreitetes GIS-Konzept ist die Sortierung der Geodaten in thematisch gegliederte Schichten (engl. *Layer*), wobei jede Schicht Elemente nur eines thematischen Typs enthält, z.B. die Farbwerte einer Bilddatei oder eine inhaltlich zusammenhängende Serie von GPS-Daten. In Analogie zu relationalen Datenbanken entspricht eine solche Schicht einer Datenbanktabelle. Eine wichtige GIS-Funktion ist der Import von Datensätzen, die, wie z.B. Bilddateien oder GPS-Daten, von anderen Systemen oder Programmen erstellt wurden und die in dem GIS als neue thematische Schicht zur Verfügung stehen sollen. Hier kommt eine weitere wichtige GIS-Funktionalität ins Spiel. Um Geodaten aus unterschiedlichen Schichten mitein-

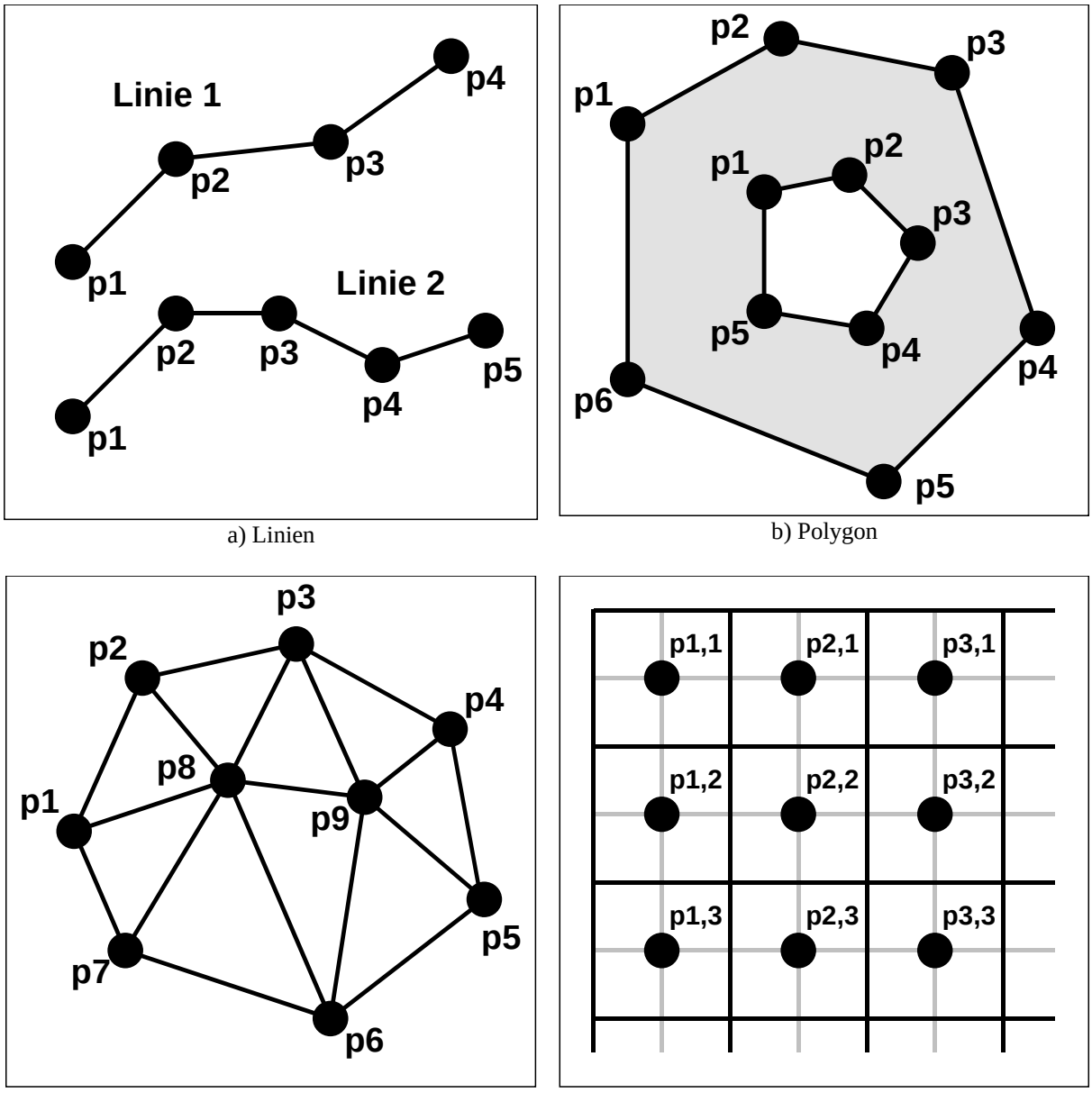
ander verknüpfen zu können, müssen die Bezugssysteme ihrer Punktkoordinaten ineinander überführt werden können. Das Bezugssystem entspricht meist einer kartographischen Projektion. Die Verwendung desselben Bezugssystems ermöglicht es erst, verschiedenste thematische Schichten über ihre räumlichen Eigenschaften miteinander verknüpfen zu können. Hierin besteht eine besondere Stärke von GIS gegenüber anderen Informationssystemen. Ein typisches Beispiel ist die Erstellung einer neuen thematischen Schicht durch die Verschneidung vorhandener Schichten, bei der Geodaten gewissermaßen räumlich vertikal miteinander verknüpft werden. Noch sehr viel mehr Möglichkeiten bietet die Analyse horizontal ausgerichteter Raumbeziehungen, z.B. bei der Erfassung einer Distanzfunktion. Viele der räumlichen Funktionen eines GIS basieren auf den Besonderheiten der von ihnen für die Speicherung von Geodaten unterstützten Datenmodelle bzw. Datenstrukturen.

2.2.2 Datenstrukturen

Das Grundelement, auf dem alle Datenstrukturen für Geodaten aufbauen, ist der Koordinatenpunkt, der eine Position auf der Erdoberfläche abbildet. Im Normalfall genügt ein zweidimensionales Koordinatensystem zur eindeutigen Festlegung einer Position. Neben polaren Bezugssystemen, in denen eine Position in geographischen Koordinaten (Länge, Breite) angegeben wird, werden verbreitet kartesische Bezugssysteme verwendet (z.B. Gauß-Krüger). Kartesische Bezugssysteme haben den Vorteil, dass sich z.B. Distanzfunktionen in ihnen direkt ausführen lassen. Die Wahl des Bezugssystems hat aber im allgemeinen keine Auswirkung auf die für die Datenhaltung benutzte Datenstruktur. Für die Abbildung der geometrischen Eigenschaften von Geodaten auf Basis von Punktkoordinaten kann zwischen zwei prinzipiellen Datenstrukturen gewählt werden.

Vektordaten

Vektordaten verknüpfen mehrere Punktkoordinaten, um die geometrischen Eigenschaften eines räumlichen Objekts abzubilden. Man spricht hier auch von den Stützpunkten eines Objekts. Die Sachdaten für das Objekts werden separat z.B. in einer Tabelle gespeichert und können relativ beliebig zusammengesetzt sein. Das einfachste geometrische Objekt ist ein einzelner Punkt, der sich vorzugsweise zur Abbildung punktbezogener Objekte eignet. Jeder Stützpunkt repräsentiert hier ein eigenständiges Objekt und besitzt auch eigene Attributdaten. Lineare Objekte werden durch Sequenzen von Stützpunkten abgebildet. Hierzu werden die aufeinanderfolgenden Punkte einer einzelnen Sequenz zu einer Linie verbunden (Abb. 9a). Bei der Flächendarstellung durch Polygone werden in gleicher Weise die Flächengrenzlinien durch Punktsequenzen definiert, wobei der letzte Punkt einer Punktsequenz wieder mit dem ersten verbunden wird, so dass sich ein geschlossener Polygonzug ergibt. Ein Flächenobjekt kann aus mehreren Teilpolygonen bestehen, so dass eine Fläche neben einem äußeren Rand auch eine Aussparung (z.B. See) enthalten kann, die mit Hilfe eines weiteren Polygonzugs einen inneren Flächenrand bildet (Abb. 9b). Es lässt sich so aber auch eine komplexe Fläche definieren, die sich aus nebeneinanderliegenden oder ineinander verschachtelten Teilpolygonen zusammensetzt (z.B. Inseln, Inseln in Seen). Auch Linienobjekte aus einer Kombination



a) Linien b) Polygon

c) TIN d) Raster

Abb. 9: Datenstrukturen

mehrerer Teillinien bestehen. Zu berücksichtigen ist hierbei immer, dass sich die inhaltlichen Attributmerkmale auf das gesamte Objekt beziehen. Ein in der Praxis seltener auftretender Fall sind Multipunkte, bei denen sich ein Sachdatensatz auf eine Punktwolke bezieht (z.B. bei der Mischprobenbildung). Während die Speicherung einfacher Punktobjekte trivial ist, wird für die Speicherung komplexerer Geometrien i.d.R. auf eines der beiden folgenden Speichermodelle zurückgegriffen:

- *Objektmodelle* speichern alle Stützpunkte separat für jedes Objekt und werden deshalb auch als Spaghetti-Struktur bezeichnet. Die Objekte werden *Shapes* (deutsch *Formen*) genannt. Der Vorteil ist, dass ein direkter Zugriff auf alle Stützpunkte eines Objekts erfolgen kann, ohne dass sie aus einer anderen Datenstruktur zusammengesucht werden müssen. Der Nachteil ist, dass u.U. eine sehr hohe Datenredundanz entstehen kann, da

insbesondere die Stützpunkte aneinander grenzender Flächen bei diesem Modell mehrfach gespeichert werden.

- *Topologische Modelle* setzen die komplexeren Geometrien von Linien und Polygonen aus Referenzen auf die eigentlichen Stützpunkte zusammen. Anfangs- und Endpunkte von Linienzügen (Kanten) werden als Knoten bezeichnet. Kanten schneiden sich niemals, sondern treffen sich immer in Knoten. Für Knoten wird i.d.R. zusätzlich gespeichert, welchen Kanten von ihnen abzweigen. Linien und Polygone werden durch Verweise auf die Kanten definiert, aus denen sie sich zusammensetzen. Auf diese Weise wird vermieden, dass die Stützpunkte für aneinandergrenzende Polygone redundant gespeichert werden. Ein weiterer Vorteil ist, dass beim Editieren einzelner Stützpunkte die Topologie aller abhängigen Geometrien erhalten bleibt, ohne dass diese separat bearbeitet werden müssen – vorausgesetzt, es werden keine neuen Knoten/Kanten eingefügt. Auch lässt sich eine Vielzahl räumlicher Analysen vergleichsweise leicht implementieren, da die Objekte topologischer Modelle in dem Sinne intelligent sind, dass sie Information über ihre Nachbarschaftsbeziehungen besitzen. Der größte Nachteil besteht in der aufwändigen Erstellung topologischer Datenstrukturen.

Eine besondere topologische Vektordatenstruktur sind Triangulated Irregular Networks (TIN), die durch eine sogenannte Dreiecksvermaschung gebildet werden. Bei TIN sind alle Stützpunkte Knoten, welche so durch Kanten verbunden werden, dass ein Netz aus Dreiecken gebildet wird (Abb. 9c). Die bisher vorgestellten Geometrien bezogen sich auf die als Ebene gedachte Erdoberfläche und lassen sich eindeutig in einem zweidimensionalen Koordinatensystem darstellen. Durch eine Erweiterung um eine dritte Dimension lassen sich aber auch andere Geometrien darstellen, wie z.B. Gebäude oder geologische Schichten.

Rasterdaten

Rasterdaten sind im Vergleich zu Vektordaten relativ einfach strukturiert und speichern meistens nur ein Attributmerkmal. Sie tun dies für eine Anzahl von Punktkoordinaten, die in einem regelmäßigen Raster (engl. *raster*) oder Gitter (engl. *grid*) angeordnet sind. Um die Koordinaten für jeden Punkt zu bestimmen genügt es daher zu wissen, wie viele Punkte sich in den Zeilen und Reihen des Gitters befinden, durch welche Distanz die Gitterpunkte voneinander getrennt sind und wo das Gitter auf der Erdoberfläche positioniert ist. I.d.R. sind Raster an den Achsen des benutzten Koordinatensystems ausgerichtet, so dass keine weiteren Informationen für die Positionsbestimmung einzelner Gitterpunkte nötig sind. Die Attributdaten werden nach Spalten und Reihen sortiert gespeichert, so dass eine Zuordnung zu den entsprechenden Gitterpunkten sehr einfach vorgenommen werden kann. Je nach Sichtweise kann man die Attributdaten statt auf die Gitterpunkte auch auf die Rasterzellen beziehen, die jeden Gitterpunkt umgeben und in der Bildverarbeitung auch als *Pixel* (von engl. *picture element*) bezeichnet werden (Abb. 9d). Am verbreitetsten sind quadratische Raster, bei denen die Distanzen zwischen den Zeilen und Reihen gleich sind. Anders als bei Vektordaten, wo Textattribute oder andere nicht numerische Attribute verwendet werden können, sind die

Attributmerkmale von Rasterdaten immer auf Zahlenwerte beschränkt. Auch Raster lassen sich um weitere Dimensionen erweitern, wodurch sich z.B. Volumen darstellen lassen.

Vektor- und Rasterdaten im Vergleich

Welche der beiden Datenstrukturen im Einzelfall für die Verwaltung von Geodaten am besten geeignet ist, hängt davon ab, welcher Natur die Daten sind und welche Fragestellungen auf sie angewendet werden sollen. Beide Datenstrukturen haben Vor- und Nachteile, die eine klare Entscheidung oft nicht einfach machen. Vektordaten eignen sich besonders für die ortsgenaue Darstellung linearer Sachverhalte einschließlich der Begrenzungslinien diskreter Flächen. Auch unregelmäßig verteilte Punktdaten lassen sich präziser und effektiver in einer Vektordatenstruktur speichern. Rasterdaten entfalten dagegen ihre Stärke bei der flächenhaften Abbildung von Attributmerkmalen, die sich kontinuierlich im Raum verändern und keine scharfen Wertesprünge aufweisen. Meistens wird die Datenstruktur schon durch die Methode vorgegeben, mit der die Daten erfasst werden. So werden GPS-Daten vorzugsweise in einer Vektorstruktur gespeichert, wogegen Bilddaten in einer Rasterstruktur vorliegen. Es gibt aber auch zahlreiche Möglichkeiten Vektordaten in einer Rasterstruktur abzubilden (Aufrastern), z.B. durch Interpolation zwischen Punktdaten, und Rasterdaten in eine Vektorstruktur zu bringen (Vektorisieren), z.B. durch Erstellung von Isolinien. Ein solches Vorgehen kann nötig sein, um auf die analytischen Vorteile der einen oder anderen Struktur zurückgreifen zu können. So ist die Verschneidung und die Nachbarschaftsanalyse mit Rasterstrukturen sehr einfach durchzuführen. Viele Netzwerkanalysen lassen sich dagegen besser mit einer Vektortopologie umsetzen. Schließlich spielen auch die unterschiedlichen kartographischen Darstellungsmöglichkeiten eine Rolle für die Wahl einer Datenstruktur.

Noch bis Mitte der 1990er Jahre war die meiste GIS-Software entweder auf Vektordaten oder auf Rasterdaten fixiert. Heute sind überwiegend hybride Softwaresysteme im Einsatz, die sowohl Vektordaten als auch Rasterdaten verwalten können, wodurch sich die offensichtlichen Vorteile beider Datenstrukturen parallel nutzen lassen. Auch SAGA ist eine hybride GIS-Software und bietet eine Reihe von Methoden für die Analyse und Verknüpfung von Raster- und Vektordaten, aber auch von TIN an.

3 DAS SYSTEM

SAGA wendet sich an reine Anwender wie auch an Entwickler geowissenschaftlicher Methoden. Um die Möglichkeiten von SAGA voll auszuschöpfen, sowohl in der Anwendung als auch in der Methodenentwicklung, ist es unablässig die zu Grunde liegenden Konzepte zu kennen und einen Überblick über den Funktionsumfang des Systems zu haben. Der Erfahrungsaustausch mit Anwendern aus Forschung und Lehre, die Popularitätssteigerung durch ein SAGA-Handbuch (OLAYA 2004) sowie das Spektrum an Fragen im Nutzerforum der SAGA-Homepage (SAGA 2006) zeigen, wie wichtig eine umfassende Software-Dokumentation ist. Die folgenden Kapiteln sollen jedoch keine einfache Anleitung für die Benutzung von SAGA sein. Vielmehr soll ein Einblick in den Aufbau des Systems, seine Besonderheiten und sein Anwendungspotential gegeben werden. Es ist allgemein bekannt, dass sich eine Software am besten durch das Ausprobieren erschließen lässt. Daher sei der Leser an dieser Stelle noch einmal dazu ermutigt, die hier vorgestellten Funktionalitäten mit SAGA selbst durchzuführen. Hilfreich sind hier die Kapitel zur Installation, zur graphischen Benutzeroberfläche, zum Kommandozeileninterpreter und zu den frei verfügbaren Modulen. An Leser, die sich auch dafür interessieren, wie SAGA die Programmierung räumlich arbeitender Methoden unterstützt, richtet sich das Kapitel über die Anwendungsprogrammierschnittstelle sowie das abschließende Kapitel, das in die Modulprogrammierung einführt. Für tiefergehende Einblicke in die programmiertechnische Umsetzung des Systems und seiner Module sei auf die frei zugänglichen, strukturiert programmierten SAGA-Quelltexte selbst verwiesen.

3.1 Installation

Die Installation der SAGA Software unterscheidet sich je nach verwendetem Betriebssystem. Zur Zeit werden die aktuellen Versionen der Betriebssysteme Microsoft Windows und Linux unterstützt. Weitere Unterschiede ergeben sich danach, ob SAGA einfach nur angewendet oder ob auch Programme unter Verwendung der SAGA Umgebung entwickelt werden sollen. Da es kein eigentliches Installationsprogramm für SAGA gibt, werden die hierfür nötigen Schritte nachfolgend vorgestellt. Die Anweisungen beziehen sich auf die aktuelle Version 2.0. Alternative Kurzanleitungen befinden sich in den Archiven der Programmpakete dieser wie auch älterer Versionen.

3.1.1 Installationspakete und zusätzliche Bibliotheken

Bevor die Installation vorgenommen werden kann, muss zunächst das passende Installationspaket, auch Distribution genannt, besorgt werden. Alle Pakete können in Form komprimierter Archivdateien über die SAGA-Projektseite bei SourceForge bezogen werden:

http://sourceforge.net/projects/saga-gis.

Vier Pakete stehen zur Wahl, von denen zwei als ZIP-Archiv und zwei als Tarball vorliegen. Beides sind populäre Archivierungsformate, wobei Tarballs vor allem unter Linux verwendet werden. Entsprechenderweise richten sich die Pakete an Windows bzw. Linux Nutzer,

mit jeweils einem Paket für die Quelltexte (engl. s*ource code*) und einem für die direkt ausführbaren Programme (engl. *binaries*).

- *saga_2.0_bin_mswvc.zip*: Binärdistribution für Windows Betriebssysteme, kompiliert mit dem VC 6.
- *saga_2.0_src.zip*: Quelltextdistribution, direkt kompilierbar mit den Compilern MinGW, VC 6/7/8.
- *saga_2.0_bin_linux.tar.gz*: Binärdistribution für Linux Betriebssysteme, statisch gelinkt mit der wxGTK Bibliothek.
- *saga_2.0_src_linux.tar.gz*: Quelltextdistribution, mit Konfigurationsdateien für die automatisierte Kompilierung mit der GCC unter Linux.

Soll SAGA nicht für die Entwicklung benutzt, sondern nur angewendet werden, reicht es die jeweilige Binärdistribution zu wählen. Soll die SAGA-Umgebung für die Programmentwicklung benutzt werden, werden außerdem zumindest die Header-Dateien der SAGA-API benötigt, die in den Quelltextdistributionen enthalten sind. Die Linux Binärdistribution wurde mit dem standardmäßig bei jeder Linux Distribution enthaltenen GCC Compiler erstellt. Für das Windows Paket wurde der Microsoft Visual C++ 6.0 Compiler (VC) benutzt. Der benutzte Compiler ist insofern von Bedeutung, als dass von verschiedenen Compilern erstellte Klassenbibliotheken leider nicht miteinander kompatibel sind. D.h., dass SAGA Module, die mit dem MinGW Compiler erstellt wurden, nicht von einer SAGA-GUI geladen werden können, die mit dem VC Compiler kompiliert wurde. Wenn SAGA aus den Quelltexten kompiliert werden soll, müssen zusätzlich weitere Bibliotheken installiert werden. Für die Binärdistribution wurden diese Bibliotheken statisch gelinkt, so dass sie nicht zusätzlich installiert werden müssen.

Die wichtigste für die Kompilierung benötigte Bibliothek ist die plattformübergreifende GUI Bibliothek *wxWidgets* in der aktuellen Version 2.8 (Kap. 2.1). wxWidgets ist fundamental für die SAGA-GUI (Kap. 3.4), wird aber auch von der SAGA-API (Kap. 3.3) und einigen Modulen benutzt. wxWidgets ist Bestandteil der meisten Linux Distributionen. Die Bibliothek und ausführliche Informationen zu ihrer Installation finden sich auf der Homepage:

http://wxwidgets.org.

Die Haru – Free PDF Library ist eine Bibliothek für das Schreiben von Dokumenten im PDF-Format. Sie wird nur von einer Klasse in der SAGA-API verwendet. Durch Setzen des Compiler-Flags *_SAGA_DONOTUSE_HARU* kann bei der Kompilierung optional auf ihre Verwendung verzichtet werden, wodurch allerdings auch die Möglichkeit der Dokumenterstellung im PDF-Format entfällt. Quelltexte und Installationsanleitungen können von der Haru Homepage bezogen werden:

http://sourceforge.net/projects/libharu.

Weitere Bibliotheken sind nur für die Kompilierung einzelner Modulbibliotheken erforderlich. Die Geospatial Data Abstraction Library (GDAL), stellt Filter für das Laden von Rasterdaten zur Verfügung (WARMERDAM et al. 2006) und wird von der Modulbibliothek *io_grid_gdal* benutzt (Kap.3.6.1). Da sie auch von vielen anderen Programmen benutzt wird, ist sie in vielen Linux-Distributionen enthalten. Die Homepage

http://www.gdal.org.

bietet Quelltexte, Installationsanleitungen und weitere Informationen. Die Modulbibliothek *pj_proj4* benutzt die Proj.4 Bibliothek (EVENDEN 2003) für die Durchführung kartographischer Projektionen (Kap.3.6.2). Auch diese Bibliothek ist Bestandteil vieler Linux-Distributionen. Die Homepage ist unter

http://www.remotesensing.org/proj/

zu erreichen. Basierend auf dem Geographic Translator (GeoTrans, NATIONAL GEOSPATIAL AGENCY 2006), wird von der Modulbibliothek *pj_geotrans* eine Alternative für die kartographische Projektion angeboten (Kap.3.6.2). Die Quelltexte dieser Bibliothek liegen den SAGA-Quelltextdistributionen bei, da die Bibliotheksschnittstellen gegenüber der Originalversion leicht verändert werden mussten. Mit den beiliegenden Makefiles bzw. Projektdateien für verschiedene Compiler lässt sich die Bibliothek leicht erstellen. Die Originalversion sowie weitere Informationen befinden sich auf der Internetseite

http://earth-info.nga.mil/GandG/geotrans/.

3.1.2 Microsoft Windows

Unterstützt werden die 32bit-fähigen Windows Versionen Windows95/98/ME und WindowsNT/2000/XP. Soll nicht unter der SAGA-Umgebung entwickelt werden, genügt es, das aktuelle Archiv mit den Binärdateien für Windows von der SAGA-Homepage zu laden und mit einem gängigen Archivierungsprogramm in ein beliebiges Verzeichnis zu entpacken. Es gibt eine Reihe frei verfügbarer Archivierungsprogramme, auf deren Verwendung hier nicht eingegangen wird. Unter den extrahierten Dateien befinden sich neben der SAGA-API Bibliothek *saga_api.dll* und den SAGA-Modulbibliotheken, diese stehen standardmäßig im Unterverzeichnis „*modules*“, zwei ausführbare Programmdateien. Der SAGA-Kommandozeileninterpreter *saga_cmd.exe* kann von einem Kommandozeilenfenster oder durch eine Stapeldatei aufgerufen werden (Kap. 3.5). Die graphische Benutzeroberfläche wird durch Ausführen von *saga_gui.exe* gestartet, z.B. durch einfaches Doppelklicken mit der linken Maustaste im Windows-Explorer, und steht danach sofort für das normale Arbeiten zur Verfügung (Kap. 3.4). Programme, die man wie die SAGA Benutzeroberflächen direkt ohne eine weitere Installation ausführen kann, werden *portabel* genannt. Zum Deinstallieren muss das Verzeichnis mit den SAGA-Dateien einfach nur gelöscht werden. Zum temporären Speichern von Benutzereinstellungen zwischen zwei Aufrufen der SAGA-GUI, z.B. der Position der Kontrollfenster oder der zuletzt geladenen Datensätze, werden zwei Konfigurationsdateien, *saga_gui.cfg* bzw. *saga_gui.ini*, im Programmverzeichnis angelegt.

Löscht man diese Dateien, wird die SAGA-GUI beim nächsten Aufruf wieder mit den Standardeinstellungen gestartet.

Sollen unter der SAGA Umgebung Programme entwickelt werden, müssen die Quelltexte installiert und kompiliert werden. Die Quelltexte müssen zunächst aus der Quelltextdistribution, unter Beibehaltung der vorgegebenen Verzeichnisstruktur, in einen beliebigen Dateiordner entpackt werden. Dadurch wird das Installationsverzeichnis „*saga_2*“ erstellt, das wiederum ein Unterverzeichnis „*src*“ mit den eigentlichen Quelltexten enthält. Des weiteren wird vorausgesetzt, dass alle zusätzlich benötigten Bibliotheken (s.o.) bereits installiert sind. Damit diese Bibliotheken und ihre Schnittstellendateien (Header) bei der SAGA-Kompilierung gefunden werden, muss für jede Bibliothek eine Umgebungsvariable mit ihrem Installationspfad gesetzt werden. Auch für SAGA wird eine entsprechende Umgebungsvariable für die Angabe des Installationsverzeichnisses benötigt, damit die abhängigen Modulbibliotheken kompiliert werden können. Bei Verwendung des MinGW Compilers wird ein zusätzlicher Eintrag erwartet, der angibt wo sich die MinGW Installation mit grundlegenden Schnittstellendateien und Bibliotheken befinden. Alle eventuell erforderlichen Umgebungsvariablen sind in Tab. 5 aufgelistet. Weitere Auskunft über das Setzen, Ändern und Löschen von Umgebungsvariablen gibt die Betriebssystemhilfe von Windows.

Für jeden der zur Zeit unter Windows unterstützen Compiler gibt es passende Makefiles bzw. Projektdateien (Tab. 6). Die Projektdateien für den VC Compiler, *saga.dsw* für Version 6 bzw. *saga.sln* für spätere Versionen, können direkt von der IDE des VC geöffnet und für die Kompilierung benutzt werden. Die Makefiles für den MinGW Compiler lassen sich mit der Stapeldatei *saga.bat* ausführen. Das Verzeichnis, das den MinGW Compiler enthält, sollte vorher der Windows Umgebungsvariablen *PATH* hinzugefügt werden, damit der Compiler aus jedem Unterverzeichnis heraus aufgerufen werden kann. Die kompilierten Programme werden im Ordner „*bin*“ des SAGA-Installationsverzeichnisses erstellt und, diesem untergeordnet, in einem je nach verwendetem Compiler speziell benannten Unterverzeichnis (Tab. 6). Dadurch wird es möglich, parallel verschiedene Compiler für denselben Quelltext zu verwenden. Die Deinstallation erfolgt wieder durch einfaches Löschen des Installationsverzeichnisses, wobei zusätzlich noch die Umgebungsvariablen gelöscht werden müssen, um

Tab. 5: Umgebungsvariablen für die Windows Kompilierung

Bibliothek	**Umgebungsvariable**	**Beispielwert**
wxWidgets	WXWIN	C:\Program Files\wxWidgets-2.8.0
Haru	HARU	D:\Libraries\haru
GDAL	GDAL	D:\Libraries\gdal
Proj.4	PROJ4	D:\Libraries\proj4
GeoTrans	GEOTRANS	D:\Libraries\geotrans
SAGA	SAGA	D:\saga_2
MinGW	MINGW	C:\Program Files\Dev-Cpp

Tab. 6: Projektdateien für Windows Compiler

Compiler/DIE	Projektdatei	Ausgabeverzeichnis
Microsoft Visual C++ 6	D:\saga_2\src\saga.dsw	D:\saga_2\bin\saga_vc
Microsoft Visual C++ 8	D:\saga_2\src\saga.sln	D:\saga_2\bin\saga_vc8
MinGW	D:\saga_2\src\saga.bat	D:\saga_2\bin\saga_mingw

die Installation vollständig rückgängig zu machen.

3.1.3 Linux

Üblicherweise werden Programme unter Linux über sogenannte Pakete (engl. *packages*) installiert. Solche Pakete dienen der Automatisierung des Installationsprozesses durch entsprechende Werkzeuge der jeweiligen Linux Distribution, wobei auch benötigte Bibliotheken automatisch in den Installationsprozess einbezogen werden. Die verschiedenen Linux Distributionen verwenden unterschiedliche Paketformate, die nur eingeschränkt miteinander kompatibel sind. Die Erstellung solcher Pakete für die SAGA-Installation befindet sich derzeit noch in der Entwicklung. Ein erster Schritt in diese Richtung ist die Verwendung des Automake-Systems, einem Entwicklungswerkzeug, mit dessen Hilfe sich die Kompilierung automatisiert für verschiedene Linux Installationen konfigurieren lässt. Die Quelltextdistribution enthält bereits eine funktionierende Testversion mit Automake-Dateien, die Regeln für die Konfiguration enthalten, doch soll zunächst die Installation der Binärdistribution vorgestellt werden.

Die Binärdistribution enthält die ausführbaren SAGA Programmdateien, wobei die wxWidgets Bibliothek statisch in diese eingebunden ist. wxWidgets muss daher nicht auf dem System installiert sein, jedoch die Bibliotheken des *GIMP Toolkit* GTK-2, die ebenfalls Bestandteil der meisten aktuellen Linux Distributionen sind. Der erste Schritt ist das Entpakken des Archivs in ein beliebiges Verzeichnis, z.B. mit dem Archivierungswerkzeug „*tar*“:

```
$ tar xfz /home/oconrad/saga_2.0_bin_linux.tar.gz /usr/bin
```

Bevor die ausführbaren SAGA-Programme gestartet werden können, muss zuerst die SAGA-API-Bibliothek *saga_api.so* dem Linux System bekannt gemacht werden, so dass sie von den Programmen gefunden und geladen werden kann. Anders als unter Windows, wird das aktuelle Verzeichnis nämlich nicht automatisch auf dynamisch ladbare Bibliotheken durchsucht. Eine Möglichkeit ist das Kopieren der Bibliothek in eines der Verzeichnisse, das in der Umgebungsvariablen *LD_LIBRARY_PATH* auftaucht, wozu i.d.R. allerdings Administratorenrechte benötigt werden. Diese Rechte erhält man durch Aufruf des Befehls *su* (von engl. *super user*) und anschließender Eingabe des Administratorpassworts:

```
$ su
# copy /usr/bin/saga_gtk/saga_api.so /usr/lib
# exit
```

Tab. 7: Installationsverzeichnisse unter Linux

Datei(en)	Verzeichnis
Ausführbare Programme (saga_gui, saga_cmd)	/usr/local/bin
API Bibliothek (saga_api.so)	/usr/local/lib
API Header	/usr/local/include
Modulbibliotheken	/usr/local/lib/saga

Alternativ kann das SAGA-Installationsverzeichnis auch der Linux Umgebungsvariablen *LD_LIBRARY_PATH* hinzugefügt werden:

```
$ export LD_LIBRARY_PATH=/usr/bin/saga_gtk:$LD_LIBRARY_PATH
```

Danach können SAGA Kommandozeileninterpreter und GUI ausgeführt werden

```
$ /usr/bin/saga_gtk/saga_gui &
```

Zum Deinstallieren genügt es, das Installationsverzeichnis und gegebenenfalls auch die kopierte API-Bibliothek, wieder zu löschen.

Für die Installation der Quelltextdistribution wird nachfolgend auf die mit dem Automake-System erstellten Konfigurationsdateien zurückgegriffen[1]. Damit die Kompilierung erfolgreich durchgeführt werden kann, müssen vorher alle benötigten Bibliotheken installiert sein. Da die Bibliotheken Haru und GeoTrans im Gegensatz zu GDAL und Proj.4 standardmäßig nicht eingebunden werden, kann auf ihre Installation auch verzichtet werden. wxWidgets muss mindestens in der Version 2.8 für Entwickler vorliegen. Die SAGA-Quelltexte werden in ein beliebiges Verzeichnis entpackt. Dort muss das von Automake erstellte Konfigurationsskript ausgeführt werden. Die Kompilierung selbst erfolgt durch den Aufruf von *make*.

```
$ tar xfz /home/oconrad/saga_2.0_src_linux.tar.gz /home/oconrad
$ cd /home/oconrad/saga_2
$ ./configure
$ make
$ su
# make install
# exit
$ saga_gui &
```

Um die Installation vollständig abzuschließen, muss mit Administratorrechten *make install* ausgeführt werden. Dadurch werden die erstellten Programme sowie die Header-Dateien und Bibliothek der SAGA-API in Standardverzeichnisse kopiert (Tab. 7), so dass sich danach die SAGA-Programme ausführen lassen und die SAGA-API auch für eigene Programmentwicklungen benutzt werden kann. In dem Beispiel wird nach der erfolgreichen Installation direkt die graphische Benutzeroberfläche gestartet.

[1] Die Automake-Dateien wurden maßgeblich von T.Schorr, GeoConsult GmbH, im Rahmen des GeoSteP Projekts (Kap. 1.2) erstellt.

3.2 Systemarchitektur

SAGA ist nicht ein einzelnes Programm, sondern ein modular aufgebautes Softwaresystem. Seine Systemarchitektur gliedert sich in drei Ebenen (Abb. 10). Das Fundament bildet eine Programmierschnittstelle, die SAGA-API. Die API ist eine dynamisch ladbare Programmbibliothek, die grundlegende Funktionen und Objekttypen für das gesamte System zur Verfügung stellt (Kap. 3.3). Beim Start des Systems durch Ausführen einer der SAGA Benutzeroberflächen (engl. *user interface*) wird die API automatisch geladen. Die Benutzerschnittstellen bilden als sogenannte *Front Ends* die oberste Ebene des Systems und ermöglichen erst seine Benutzung. SAGA 2.0 bietet zwei alternative Benutzerschnittstellen an. Die graphische Benutzerschnittstelle (GUI), verwendet Fenster um Inhalte darzustellen bzw. zugänglich zu machen. Graphische Kontrollelemente, wie Befehlsmenüs, Dialogfenster, Baumansichten und Werkzeugleisten, dienen einer intuitiven Bedienbarkeit, z.B. bei der Verwaltung, Analyse und Visualisierung von Datensätzen (Kap. 3.4). Die zweite Benutzerschnittstelle ist der SAGA-Kommandozeileninterpreter, der von einer Kommandozeile aus ausgeführt wird und über keine graphischen Fähigkeiten verfügt (Kap. 3.5). Zwischen API und Front End befindet sich die Ebene der Modulbibliotheken, die den Anwender mit Methoden in Form sogenannter Module versorgt. Modulbibliotheken sind, wie die API, dynamisch ladbare Programmbibliotheken, zeichnen sich aber durch eine spezielle Bibliotheksschnittstelle aus. Sie basieren auf den von der API zur Verfügung gestellten Funktionalitäten und müssen für die Ausführung von einem der Front Ends geladen werden. Die

Abb. 10: Systemarchitektur

Module sind nach thematischen Aspekten in den Bibliotheken zusammengefasst. Einen Überblick über die freien Modulbibliotheken gibt Kap. 3.6.

Zu den Vorteilen dieser modularen Systemarchitektur zählt die Möglichkeit verschiedene Front Ends benutzen zu können und es ist durchaus wahrscheinlich, dass das System in Zukunft um spezialisierte Benutzerschnittstellen erweitert wird, z.B. um auf eine GIS-Datenbank aufzusetzen oder um eine vereinfachte Oberfläche für Verwaltungsaufgaben zur Verfügung zu stellen. Am vordergründigsten ist jedoch der Vorteil, dass die Methoden unabhängig vom restlichen System in Modulbibliotheken implementiert werden. An dieser Stelle muss auch auf das für SAGA verwendete Software-Lizenzmodell eingegangen werden (s.a. Kap. 2.1.5). Während die Benutzerschnittstellen unter die GPL gestellt wurden, ihre Quelltexte also nicht für proprietäre Softwareentwicklungen benutzt werden dürfen, steht die API unter der weniger restriktiven LGPL. Auf diese Weise müssen die von der API abhängigen Modulbibliotheken nicht zwangsläufig unter der GPL oder einer zu dieser kompatiblen Lizenz mit ihren Quelltexten veröffentlicht werden. Es ist also völlig legal, quelltextgeschlossene SAGA Module zu entwickeln, um diese z.B. kommerziell vermarkten zu können.

3.3 Anwendungsprogrammierschnittstelle

Die API bildet den Kern von SAGA und ist die Basis für alle anderen Systemelemente. Es handelt sich hierbei um eine dynamisch ladbare C++ Klassenbibliothek, die neben einfachen Funktionen eine Reihe von Objektklassen bereitstellt, die vor allem, aber nicht ausschließlich, für die Verarbeitung raumbezogener Daten verantwortlich sind. Direkt mit der API konfrontiert ist nur der Programmierer, der die API-Funktionalitäten für die eigene Softwareentwicklung benutzt, z.B. bei der Erstellung eines SAGA-Moduls. Voraussetzung für ihre effektive Verwendung ist ein umfassender Überblick über die von der API zur Verfügung gestellten Funktionen und Klassen. Die hierfür benötigten Grundkenntnisse der Programmiersprache C++ wurden bereits in Kapitel 2.1 vermittelt.

Die Benennung von Funktionen, Strukturen und Klassen der API folgt einer gewissen Konvention. Klassennamen beginnen immer mit dem Buchstaben „*C*“, sonstige Typdefinitionen beginnen mit „*T*“. Um die Zugehörigkeit zur SAGA API zu verdeutlichen folgen danach immer die Zeichen „*SG_*“. Globale Funktionen, also solche, die nicht Mitglied einer Klasse sind, beginnen direkt mit „*SG_*“. Durch diese Namenskonvention soll vermieden werden, dass Funktions- und Klassenbezeichner bei Einbindung weiterer Bibliotheken mehrfach in unterschiedlichen Kontexten benutzt werden, was einen Quelltext in der Regel unkompilierbar macht. So besitzt die MFC z.B. die Klassen *CString* und *CPoint*, die anderenfalls mit den Klassen *CSG_String* bzw. *CSG_Point* verwechselt werden könnten[1]. Um für SAGA eine breite, internationale Entwicklergemeinde gewinnen zu können, sind des

[1] Aus demselben Grund fangen alle Funktionen und Klassen der wxWidgets Bibliothek mit „*wx*“ an, z.B. *wxString* und *wxPoint*.

weiteren alle Bezeichner und Quelltextkommentare in englischer Sprache gehalten. Die Bezeichner sind so gewählt, dass sie möglichst selbsterklärend sind und dadurch ein intuitives Programmieren unterstützt wird.

Auch die API verwendet einige Funktionalitäten der wxWidgets Bibliothek, jedoch nicht in dem Ausmaß wie die SAGA-GUI. Die Verwendung von wxWidgets Funktionalitäten ist allerdings so gekapselt, dass sie für den Modulprogrammierer verborgen bleibt und von diesem auch nicht berücksichtigt werden muss. Verwendete wxWidgets Funktionalitäten betreffen u.a. die Verarbeitung von Zeichenketten (engl. *strings*) und den plattformunabhängigen Dateizugriff. Für die Erstellung von PDF-Dokumenten wird die HARU Bibliothek benutzt, die ebenfalls vor dem Nutzer der API verborgen bleibt. Für die Programmierung von SAGA-Modulen bedeutet das, dass außer der API Bibliothek keine zusätzlichen Bibliotheken eingebunden werden müssen.

Um die API in ein anderes Programm einzubinden, muss zum einen ihre Schnittstelle bekannt sein und zum anderen muss die API-Bibliothek während des Linking-Prozesses vom Compiler gefunden werden können. Die Schnittstellendeklarationen befinden sich in den Header-Dateien der API (Tab. 8). Einsprungspunkt ist die Header-Datei „*saga_api.h*“, von der aus die anderen Header automatisch einbezogen werden. Ein alternativer Einsprungspunkt ist der Header „*compatibility.h*“, der ein gewisses Maß an Kompatibilität für Quelltexte bietet, die auf älteren API-Versionen beruhen. Der größte Teil der jüngeren Versionsunterschiede geht auf einfache Umbenennungen von Funktions- und Klassenbezeichnern zurück, die vor allem für eine bessere Lesbarkeit vorgenommen wurden, und werden durch „*compatibility.h*“ vollständig abgefangen. Einige wenige Unterschiede zu der API-Version 1 betreffen jedoch Änderungen von Funktionsaufrufen und müssen, sofern diese Funktionen in einem Quelltext verwendet wurden, manuell angepasst werden. Hinweise hierzu befinden sich ebenfalls in

Tab. 8: API Header

Header Datei	**Beschreibung**
saga_api.h	Standard-Header zum Einbinden der SAGA API
compatibility.h	Kompatibilität für ältere SAGA Versionen
api_core.h	Kernfunktionen und allgemeine Werkzeuge
geo_tools.h	Geometrische Funktionen und Werkzeuge
mat_tools.h	Mathematisch-numerische Funktionen und Werkzeuge
doc_html, doc_pdf.h, doc_svg.h	Werkzeuge für die automatisierte Dokumenterstellung
dataobject.h	Basisfunktionen für Datenobjekte
table.h	Tabellen-Funktionen
shapes.h	Vektor-Funktionen
tin.h	TIN-Funktionen
grid.h	Raster-Funktionen
parameters.h	Funktionalitäten für Parameterlisten
module.h	Basisfunktionen für Module und Modulbibliotheken

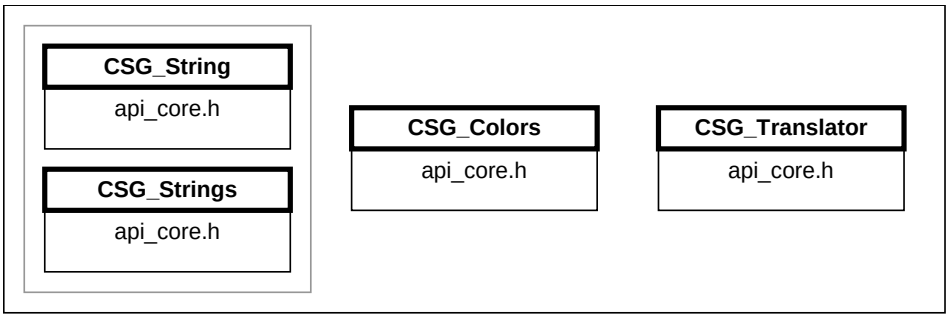


Abb. 11: Klassenübersicht – Allgemeine Werkzeuge

„*compatibility.h*“. Die weiteren Header-Dateien enthalten, thematisch gegliedert, die eigentlichen Funktionalitäten der API. An dieser thematischen Gliederung orientiert sich auch die nachfolgende Beschreibung der Schnittstellen. Klassen und Klassenhierarchien werden für jedes Kapitel in einer einführenden Übersicht abgebildet[1]. Die insgesamt sehr umfangreichen Funktionalitäten der API können im Rahmen dieser Arbeit nicht im Detail erläutert werden und auch die tabellarischen Übersichten von Funktionen beschränken sich nur auf eine Auswahl.. Die Bedeutung nicht explizit erwähnter Funktionen sollte sich jedoch leicht durch ihre Bezeichner erschließen lassen. In einigen Fällen wird zur Verdeutlichung der Funktionsweise einer Klasse ein kurzes Quelltextbeispiel gegeben oder auf beispielhafte Quelltexte unter den SAGA-Modulen verwiesen. Eine allgemeine Einführung in die Verwendung der API für die Modulprogrammierung wird in Kapitel 3.7 gegeben.

3.3.1 Allgemeine Werkzeuge

Allgemeine Werkzeuge gibt es sowohl für SAGA-typische Aufgaben als auch für die Unterstützung der Programmierung häufig benötigter, allgemeinerer Aufgaben. Sie ermöglichen u.a. die Kommunikation mit der Benutzeroberfläche und der Verarbeitung wie auch der Übersetzung von Texten.

Damit Module mit der Benutzeroberfläche kommunizieren können, wird von dieser eine Rückruffunktion (engl. *callback function*) bereitgestellt, die auf vordefinierte Anfragen in geeigneter Weise reagieren oder abgefragte Werte zurückliefern kann. Der Zugriff auf diese Rückruffunktion erfolgt nicht direkt, sondern über eine Reihe speziell zugeschnittener API-Funktionen. Der Modulprogrammierer wird in den meisten Fällen auch nicht auf diese Funktionen direkt zugreifen müssen, da die von ihm verwendete Modulklasse bereits entsprechende Methoden hierfür bereitstellt (s. Kap. 3.3.5). Erwähnung finden sie hier, da sie in einigen Fällen mehr Möglichkeiten bieten, aber auch, weil ein Verständnis ihrer Funktionsweise für Programmierer, die nicht nur an Modulen arbeiten, sondern auch die API oder die Benutzeroberflächen weiterentwickeln möchten, die Vorraussetzung ist. Die zur Zeit angebotenen Interaktionen decken die in Tab. 9 aufgelisteten Bereiche ab.

[1] Abstrakte Klassen sind mit $^{(A)}$ gekennzeichnet. Abstrakte Klassen besitzen Funktionen, die von abgeleiteten Klassen immer überschrieben werden müssen, damit Objektinstanzen von ihnen erstellt werden können.

Tab. 9: Funktionen für die Kommunikation mit der Benutzeroberfläche

Aufgaben	Funktion	Modulklassenfunktion	Beschreibung
Status der Modulausführung	SG_UI_Process_Set_Progress	Set_Progress	Fortschrittsanzeige
	SG_UI_Process_Get_Okay	Process_Get_Okay	Abbruch durch Benutzer
	SG_UI_Process_Set_Text	Process_Set_Text	Setzt Statuszeilentext
Dialoge	SG_UI_Dlg_Message	Message_Dlg	Benachrichtigungs-Box
	SG_UI_Dlg_Error	-	Fehlermeldungs-Box
Benachrichtigungen	SG_UI_Msg_Add	Message_Add	Allgemeine Benachrichtigung
	SG_UI_Msg_Add_Error	Error_Set	Fehlerbenachrichtigung
Datensätze	SG_UI_DataObject_Add	DataObject_Add	Übergabe eines neuen Datensatzes an das Datenverwaltungssystem
	SG_UI_DataObject_Update	DataObject_Update	Datensatz muss neu dargestellt werden
	SG_UI_DataObject_Colors_Set	DataObject_Set_Colors	Farbdarstellung eines Datensatzes ändern

Die Klasse *CSG_String* kapselt die *wxString*-Klasse der wxWidgets-Bibliothek und stellt deren Funktionalität zur Verfügung (Tab. 10). *Strings* repräsentieren Zeichenketten und werden üblicherweise für das Speichern und Verarbeiten von beliebigen Texten benutzt. Ein Vorteil dieser Klasse gegenüber der in C üblichen Verwendung von Byte-Feldern ist die automatische Speicherverwaltung, wodurch einerseits Speicherfehler vermieden werden und anderseits Quelltexte übersichtlicher gestaltet werden. Darüber hinaus wird eine Reihe von Standardfunktionen und Operatoren für die Manipulation und Auswertung von Zeichenketten zur Verfügung gestellt. *CSG_Strings* ist eine einfache Klasse, mit der Listen von *CSG_String*-Objekten erstellt und verwaltet werden können. Die Klasse bietet Funktionen zum Hinzufügen, Löschen und Abfragen einzelner *CSG_String*-Objekte.

Es gibt eine Reihe von Funktionen mit denen unabhängig von den Besonderheiten des Betriebssystems auf Verzeichnisse und Dateien zugegriffen werden kann (Tab. 11). So kann

Tab. 10: Funktionen für Zeichenketten

Funktion	Beschreibung
CSG_String & operator = (const char *String)	Kopiert eine Zeichenkette
char & operator [] (int i)	Zeichen an Position i
Int Length(void)	Anzahl der Zeichen
void Clear(void)	Löscht die Zeichenkette
Int Printf(const char *Format, ...)	Gegenstück zu printf der C Standardbibliothek
CSG_String & Append(const char *String)	Hinzufügen einer Zeichenkette
Int Cmp(const char *String)	Vergleich mit einer Zeichenkette, Gegenstück zu strcmp der C Standardbibliothek
Int Find(const char *String)	Sucht nach einer Zeichenkette
Int asInt(void), double asDouble(void)	Konvertiert die Zeichenkette in eine Zahl

Tab. 11: Funktionen für den Dateizugriff

Funktion	Beschreibung
bool SG_Directory_isValid(const char *Directory)	Überprüft, ob ein Verzeichnis existiert
bool SG_Directory_Make(const char *Directory)	Erstellt ein Verzeichnis
bool SG_File_Delete(const char *FileName)	Löscht eine Datei
CSG_String SG_Get_Temp_File_Name(const char *Prefix, const char *Directory)	Erstellt einen einmalig vorkommenden Namen für eine temporäre Datei
CSG_String SG_File_Get_Name(const char *full_Path)	Extrahiert einen Dateinamen aus einem Dateipfad
CSG_String SG_File_Make_Path(const char *Directory, const char *Name, const char *Extension)	Erstellt einen Dateipfad
bool SG_File_Cmp_Extension(const char *File_Name, const char *Extension)	Vergleicht die Erweiterung eines Dateinamens auf Übereinstimmung
bool SG_Read_Line(FILE *Stream, CSG_String &Line)	Liest eine Zeile aus einer geöffneten Textdatei

abgefragt werden, ob ein Verzeichnis oder eine Datei existiert. Verzeichnisse und Dateien lassen sich erstellen oder löschen. Auch Dateinamen lassen sich auswerten oder neu zusammensetzen. Weitere Funktionen erleichtern das Auslesen und Schreiben von Dateien, wobei neben Funktionen für Textdateien auch Lese- und Schreibfunktionen für binäre Daten genutzt werden können.

Farbpaletten werden für die graphische Darstellung von Datensätzen durch die Benutzeroberfläche eingesetzt. Es gibt auch Module, die Farbpaletten verwenden, z.B. um den von ihnen erstellten Datensätzen eine farbpsychologisch sinnvolle Farbdarstellung zu geben. Die Verwaltung von Farbpaletten übernimmt die Klasse *CSG_Colors* (Tab. 12). Die Farbcodierung der einzelnen Farben entspricht einem bei Computersystemen üblichen Schema, das von den graphischen Schnittstellen von Betriebssystemen, aber auch von vielen Bildformaten für die Darstellung von Echtfarben benutzt wird. Hierbei wird eine Farbe durch die Intensitäten ihrer roten (R), grünen (G) und blauen (B) Anteile definiert. Jeder Farbanteil wird in einem Byte gespeichert und kann einen Wert zwischen 0 (kein Anteil) und 255 (volle Intensität) annehmen. Der Programmierer kann die Anzahl der Farben und ihre Farbwerte frei definieren. Es steht aber auch aus einer Reihe von vordefinierten Farbpaletten gewählt werden.

Die Klasse *CSG_Translator* ist das Herzstück des Internationalisierungsmechanismus in SAGA. Die Klasse selbst dient vor allem der internen Verwaltung von Übersetzungstabellen und wird daher nicht weiter vorgestellt. Es soll aber an einem kurzen Beispiel gezeigt werden, wie der Mechanismus benutzt wird, um Texte einer Modulbibliothek übersetzbar zu machen.

Tab. 12: Funktionen für die Farbdarstellung

Funktion	Beschreibung
bool Set_Count(int nColors)	Ändert die Zahl der Paletteneinträge
int Get_Count(void)	Zahl der Paletteneinträge
bool Set_Color(int i, int R, int G, int B)	Setzt die Farbe eines Paletteneintrags
bool Set_Palette(int Palette, bool bRevert)	Erstellt eine vordefinierte Palette
bool Random(void)	Erstellt eine Palette aus zufällig ausgewählten Farben

Dazu muss als erstes jede Zeichenkette, die übersetzt werden soll, von dem Makro *_TL* eingeschlossen sein, wie z.B. in diesem Quelltextfragment:

```
printf( _TL(“Translate me!“) );
```

Als zweites muss eine Textdatei mit den Übersetzungen angelegt werden. Damit diese Datei beim Laden der Modulbibliothek automatisch gefunden und ausgewertet werden kann, muss sie im selben Verzeichnis stehen und denselben Dateinamen wie die Bibliothek haben. Die Dateinamenserweiterung der Textdatei mit der Übersetzungstabelle muss jedoch „*lng*“ lauten. Eine Übersetzungstabelle für die Modulbibliothek „*ta_lighting.dll*“ muss folglich „*ta_lighting.lng*“ heißen. Ein zu übersetzender Text wird in der Übersetzungstabelle dem Schlüsselwort *ENTRY* in Klammern nachgestellt. Darauf folgt der von Anführungszeichen eingeschlossene Text der Übersetzung. Ein Übersetzungseintrag sieht damit so aus:

```
ENTRY(Translate me!)
   "Übersetze mich!"
```

Ein Vorteil von diesem Übersetzungsmechanismus ist, dass die Übersetzungen nicht fest in den Programmcode einkompiliert werden und jederzeit mit einem Texteditor bearbeitet oder vom Benutzer ausgetauscht werden können. Bisher wurden Übersetzungstabellen nur im Einzelfall für eine der frei verfügbaren Modulbibliotheken erstellt. Eines der wenigen Beispiele gibt die Modulbibliothek **ta_lighting**.

3.3.2 Mathematisch-numerische Funktionen und Klassen

Eine Reihe häufig benötigter mathematisch-numerischer Werkzeuge wurde in die API aufgenommen. Geometrische Funktionalitäten aber auch das Erstellen von Indizes sind die Grundlage für viele GIS-Funktionen und werden auch von anderen Klassen der API intensiv genutzt. Viele dieser Funktionalitäten vereinfachen die Umsetzung von Algorithmen bei der Modulprogrammierung. So eignet sich z.B. die Matrix-Klasse zum Lösen linearer Gleichungssysteme, ein Formelparser kann als Zeichenkette gespeicherte, benutzerdefinierte Formeln auswerten und Korrelations- und Regressionsanalysen lassen sich einfach durchführen.

Punkte, Rechtecke und verwandte Funktionen

Punktkoordinaten sind von zentraler Bedeutung für die räumliche Datenverarbeitung. Die Datenstruktur *TSG_Point*:

```
typedef struct
{
   double   x, y;
}
TSG_Point;
```

repräsentiert die in SAGA benutzten Koordinatentupel. Die Klasse *CSG_Point* erweitert diesen Datentyp um punktspezifische Funktionen zum Addieren, zur Differenzbildung und zum Vergleich von Koordinaten. Eng verwandt sind *TSG_Point_Int*, eine Struktur für

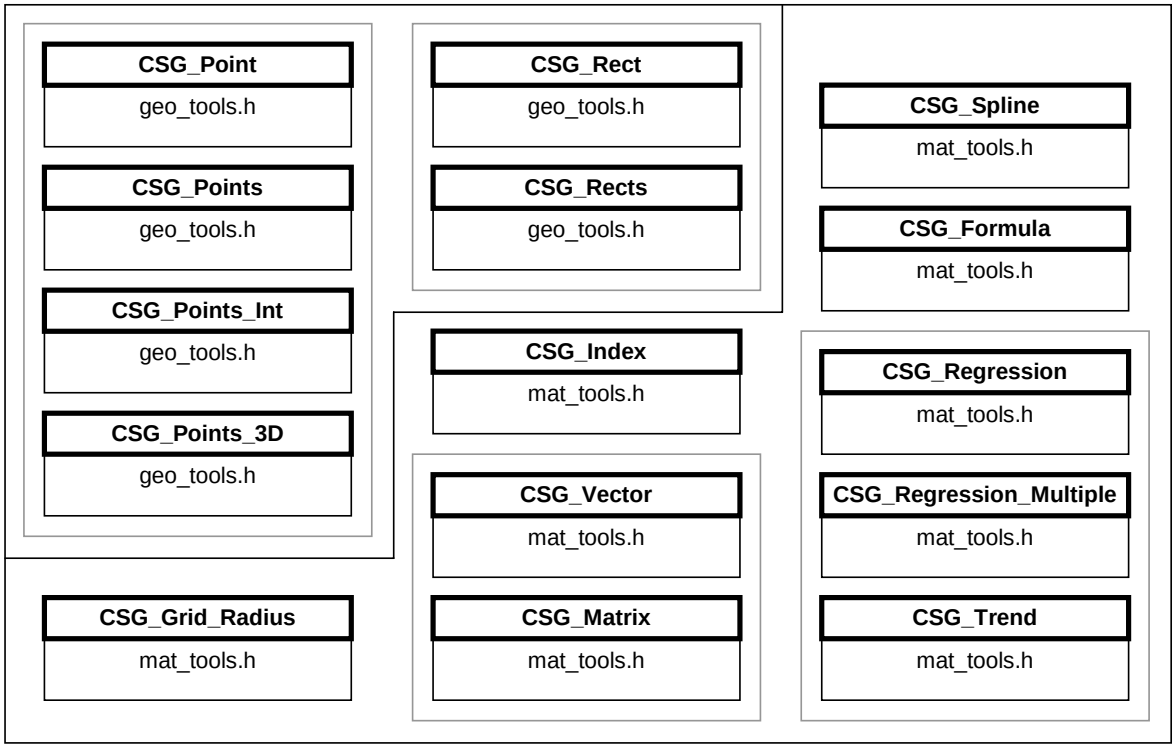


Abb. 12: Klassenübersicht – mathematisch-numerische Werkzeuge

ganzzahlige Koordinatentupel, und *TSG_Point_3D*, die ein Koordinatentripel enthält. Zum Verwalten von Punktlisten gibt es für jede dieser drei Punkttypen eine gesonderte Klasse, *CSG_Points*, *CSG_Points_Int* bzw. *CSG_Points_3D*, mit Funktionen zum Zufügen, Löschen, Kopieren und Abfragen von Punkten. Auch die Koordinaten von Rechtecken werden oft benötigt, und können in der Struktur *TSG_Rect*

```
typedef struct
{
   double   xMin, yMin, xMax, yMax;
}
TSG_Rect;
```

bzw. in der korrespondierenden Klasse *CSG_Rect* gespeichert werden. *CSG_Rect* bietet verschiedene Funktionen zum Kombinieren und Vergleichen von Rechtecken und wird ergänzt von einer Klasse *CSG_Rects* zum Verwalten von Rechteck-Listen. Für die Ableitung komplexerer Eigenschaften von Punktbeziehungen gibt es weitere Funktionen Die euklidische Distanz liefert

```
double SG_Get_Distance  (const TSG_Point &A, const TSG_Point &B);
```

Die Richtung in der Punkt *B* von Punkt *A* aus gesehen liegt, wird von

```
double SG_Get_Angle_Of_Direction (const TSG_Point &A, const TSG_Point &B);
```

berechnet. Die Funktion

```
bool   SG_Get_Crossing(
                TSG_Point &Crossing,
                const TSG_Point &A1, const TSG_Point &A2,
                const TSG_Point &B1, const TSG_Point &B2,
                bool bExactMatch = true
         );
```

bestimmt den Schnittpunkt zweier Geraden, die durch die Punkte *A1*, *A2* bzw. *B1*, *B2* definiert sind. Mit *bExactMatch* kann die Berechnung auf die zwischen den Punktpaaren

gelegenen Strecken eingeschränkt werden. Sind die Geraden zueinander parallel oder liegt der Schnittpunkt bei gewünschter Einschränkung außerhalb der eingeschlossenen Strecken, ist der Rückgabewert der Funktion „*false*“, sonst „*true*“. Den Abstand des Punktes *Point* von der durch die Punkte *Ln_A*, *Ln_B* definierten Geraden liefert die Funktion

```
double  SG_Get_Nearest_Point_On_Line(
                const TSG_Point &Point, const TSG_Point &Ln_A, const TSG_Point &Ln_B,
                TSG_Point &Ln_Point,
                bool bExactMatch = true
        );
```

Die Berechnung kann wiederum auf die zwischen *Ln_A* und *Ln_B* liegende Strecke eingeschränkt werden. Nach der Berechnung enthält *Ln_Point* die Koordinaten des Punktes auf der Geraden, der *Point* am nächsten gelegen ist. Die Fläche eines Polygons wird von der Funktion

```
double  SG_Get_Polygon_Area   (const CSG_Points &Points);
```

berechnet, wobei *Points* eine Liste mit den Polygon-Punkten ist. Bei negativem Vorzeichen des Rückgabewerts sind die Punkte im Uhrzeigersinn, bei positivem gegen den Uhrzeigersinn angeordnet.

Indexerstellung

Mit der Klasse *CSG_Index* lässt sich sehr einfach ein Index für eine Liste von beliebigen Werten erstellen, so dass die Werte anschließend in der gewünschten Sortierreihenfolge angesprochen werden können. Die Indexerstellung erfolgt entweder direkt bei der Klassenkonstruktion oder zu einem beliebigen späteren Zeitpunkt mit einer der drei *Create* Funktionen

```
bool    Create   (int nValues, int    *Values, bool bAscending = true);
bool    Create   (int nValues, double *Values, bool bAscending = true);
bool    Create   (int nValues, TSG_PFNC_Compare fCompare, bool bAscending = true);
```

Mit *nValues* wird die Anzahl der zu sortierenden Werte und mit *bAscending* die Sortierrichtung übergeben. Die ersten beiden Versionen erwarten die Werte in Form eines Feldes. Die dritte Version erwartet die Übergabe einer Vergleichsfunktion der Form

```
typedef int (* TSG_PFNC_Compare) (const int iElement_1, const int iElement_2);
```

und ist damit zugleich die flexibelste Möglichkeit der Indexerstellung. *IElement_1* und *iElement_2* sind die Feldpositionen der zu vergleichenden Werte. Der Rückgabewert muss negativ sein, wenn der das erste Element kleiner als das zweite ist, positiv, wenn er größer ist, und gleich 0 sein, wenn beide Elemente identische Werte enthalten. Der erstellte Index wird mit der Funktion

```
int     Get_Index    (int i, bool bInv = false);
```

oder dem Operator

```
int     operator []  (int i);
```

abgefragt.

Vektoren und Matrizen

Die Klassen *CSG_Vektor* und *CSG_Matrix* vereinfachen die Anwendung von Methoden der Vektor- und Linearen Algebra. Beide Klassen sind bewusst sehr einfach gehalten und überschreiben in erster Linie die verschiedenen mathematischen Operatoren, so dass Addition, Subtraktion und Multiplikation für die Kombinationen von Vektoren mit Vektoren, Vektoren mit Matrizen und Matrizen mit Matrizen zur Verfügung stehen. Die Vektormultiplikation unterscheidet zwischen innerem (Skalarprodukt) und äußerem Produkt (Kreuzprodukt). Darüber hinaus bietet die Matrixklasse Funktionen zum Transponieren und zum Invertieren an, so dass sie z.B. für das Lösen linearer Gleichungssysteme eingesetzt werden kann. Das lineare Gleichungssystem

$$\begin{aligned} b_1 &= a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \\ b_2 &= a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \\ &... \\ b_m &= a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \end{aligned} \quad \text{bzw. } \vec{b} = A\vec{x} \qquad [1]$$

mit den bekannten Elementen $a_{\mu,\nu}$ der Matrix A, den bekannten Vektorkomponenten b_μ und den gesuchten x_ν (für $\mu = 1, 2, ..., m$ und $\nu = 1, 2, ..., n$) lässt sich durch Invertieren der Matrix A zu A^{-1} lösen (BRONSTEIN et al. 1997)

$$\vec{x} = A^{-1}\vec{b} \; . \qquad [2]$$

Ein entsprechender Quelltext sieht z.B. so aus

```
...
   CSG_Vector  x(n), b(m);
   CSG_Matrix  A(n, m);
...
   // set the elements of A and b...
...
   x  = A.Get_Inverse() * b;
```

Da sich Gleichungssysteme nicht immer lösen lassen, kann die Invertierung auch separat ausgeführt und vor weiteren Operationen auf Erfolg überprüft werden. Der Zugriff auf die Vektor- und Matrix-Elemente erfolgt mit dem [ ] Operator. Die Multiplikation einer Matrix mit einem Vektor lässt sich also auch folgendermaßen formulieren:

```
   CSG_Vector  x(n), y(m);
   CSG_Matrix  A(m, n);

   for(i=0; i<m; i++)
   {
      y[i]  = 0;

      for(j=0; j<n; j++)
         y[i]  += A[i][j] * x[j];
   }
```

Formelparser

Mit der Klasse *CSG_Formula* lassen sich durch eine Zeichenkette definierte Formeln auswerten. Neben Standardoperatoren für Addition, Subtraktion, Multiplikation und Division sowie zur Verschachtelung von Berechnungsebenen durch Klammern, können trigonometrische, Wurzel-, Exponential- und Logarithmus-Funktionen benutzt sowie bedingte Abfragen formuliert werden. Variablen werden durch einzelne Buchstaben bezeichnet. Das Setzen einer Formel mit der Zeichenkette *Formula* erfolgt mit

```
bool   Set_Formula(const char *Formula);
```

Der Rückgabewert gibt an, ob die Formel gültig ist. Mit der Funktion

```
void   Set_Variable(char Variable, double Value);
```

können danach Werte für die Variablen a, b gesetzt werden. Ausgewertet wird die Formel mit der Funktion

```
double Val(void);
```

Sollte bei der Auswertung ein Fehler aufgetreten sein, z.B. im Falle einer Division durch Null, liefert die Funktion

```
bool   Get_Error(int *Pos = NULL, const char **Msg = NULL);
```

den Wahrheitswert „*false*“ zurück. Erläuternde Fehlermeldungen können optional mit *Pos* und *Msg* abgerufen werden. Ein einfaches Beispiel für die Verwendung des Formelparsers sieht so aus:

```
CSG_Formula Formula;

Formula.Set_Formula("(a - b) / (a + b)");

Formula.Set_Variable('a', 1.5);
Formula.Set_Variable('b', 2.0);

double   c  = Formula.Val();
```

Komplexere Beispiele für die Verwendung des Formelparsers sind das Modul *CSG_Grid_Calculator* der Modulbibliothek **grid_calculus** und die nachfolgend vorgestellte API Klasse *CSG_Trend* zur Ermittlung einer Trendfunktion.

Korrelations- und Regressionsanalyse

Mit den statistischen Verfahren der Korrelations- und Regressionsanalyse wird der mathematisch-funktionale Zusammenhang von zwei oder mehreren Variablen bestimmt. Ziel der Korrelationsanalyse ist es, die Stärke des Zusammenhangs der Variablen zu ermitteln, während die Regressionsanalyse nach einer Abhängigkeitsfunktion zur Beschreibung des Zusammenhangs sucht. Die Abhängigkeitsfunktion kann anschließend z.B. für die Vorhersage von Werten benutzt werden. Beide Verfahren sind eng miteinander verknüpft und werden von den vorgestellten Klassen in einem Schritt ausgeführt. Bei der Einfachregression mit der Klasse *CSG_Regression* wird untersucht, wie eine Variable *y* von der Variablen *x* abhängt. Neben dem linearen Regressionsmodell

$$y = f(x) = b + ax \quad [3]$$

stellt die Klasse alternative nicht-lineare Funktionen für die Regression zur Verfügung:

$$f(x) = b + \frac{a}{x} \quad [4]$$

$$f(x) = \frac{b}{a + x} \quad [5]$$

$$f(x) = bx^a \quad [6]$$

$$f(x) = be^{ax} \quad [7]$$

$$f(x) = b + a\log(x) \quad [8]$$

Für die Durchführung der Regressionsanalyse müssen der Klasse als erstes die zu analysierenden Wertepaare übergeben werden, entweder einzeln mit der Funktion

```
void    Add_Values    (double x, double y)
```

oder als Wertefeld mit

```
void    Set_Values    (int nValues, double *xValues, double *yValues);
```

Die Analyse selbst wird mit

```
bool    Calculate     (TSG_Regression_Type Type = REGRESSION_Linear);
```

ausgeführt, wobei gleichzeitig das gewünschte Regressionsmodell gewählt werden kann. Danach können die Korrelations- und Regressionskoeffizienten abgefragt werden. Werte der abhängigen Variablen *y* lassen sich auch direkt mit

```
double  Get_Y         (double x);
```

abrufen. Etwas flexibler, aber ansonsten ähnlich aufgebaut, ist die Klasse *CSG_Trend*, bei der nahezu beliebige Regressions-Funktionen durch eine Zeichenkette definiert werden können. Zum Auswerten der Zeichenkette benutzt *CSG_Trend* die Klasse *CSG_Formula*. Das Modul *CSG_Table_Trend* der Bibliothek **table_calculus** gibt ein gutes Beispiel für die Verwendung dieser Klasse.

Wurde bei der Einfachregression die Beziehung zweier Variablen untersucht, so bietet die Klasse *CSG_Regression_Multiple* die Möglichkeit, die Abhängigkeit einer Variablen von einer beliebigen Anzahl unabhängiger Variablen zu bestimmen, wobei die Regression auf das lineare Modell

$$y = a_1x_1 + a_2x_2 + ... + a_nx_n + b\,. \quad (9)$$

beschränkt ist. Die Übergabe der zu analysierenden Werte erfolgt hier ausschließlich in Form einer Tabelle, wie sie als Datenobjekt von der API bereitgestellt wird (s. Kap.3.3.4). Die

Klasse verwendet ein vorwärtsgerichtetes Verfahren zur Ermittlung der einzelnen Regressionskoeffizienten $a_1, a_2, ..., a_n$ (s.a. BAHRENBERG et al. 1992, BRONSTEIN et al. 1997). Benutzt wird sie z.B. von dem Modul *CGSGrid_Regression_Multiple* der Bibliothek **geostatistics_grid**, um Punktdaten mit Rasterdaten zu korrelieren und die gefundene Beziehung für die flächenhafte Wertevorhersage zu verwenden.

Weitere Klassen

Die Klasse *CSG_Grid_Radius* ist einfaches Werkzeug, um schnell alle Gitterpunkte bzw. Rasterzellen zu ermitteln, die sich innerhalb einer kreisförmigen Umgebung mit gegebenen Radius befinden (Abb. 13). Für einen beliebigen Radius, angegeben in Rasterzellen, kann mit

```
int    Get_nPoints  (int iRadius);
```

die Anzahl im Umkreis enthaltener Rasterzellen abgefragt werden, deren relative Positionen wiederum von

```
double Get_Point    (int iRadius, int iPoint, int &x, int &y);
```

an die Funktionsparameter *x*, *y* übergeben werden. Rückgabewert ist die tatsächliche Distanz zum Gitterpunkt. Kreisförmige Suchfunktionen werden u.a. von den meisten Filtern der Bibliothek **grid_filter** benutzt.

Mit der Klasse *CSG_Spline* steht eine einfache Klasse für die Spline-Interpolation der Form *y=f(x)* zur Verfügung (z.B. FAIRES & BURDEN 1994, BRONSTEIN et al. 1997). Die bekannten Werte (x, y), zwischen denen interpoliert werden soll, werden einzeln mit der Funktion

```
void   Add     (double x, double y);
```

oder in einem Schritt als Feld übergeben mit

```
bool   Create      (double *xValues, double *yValues, int nValues);
```

Danach können sofort interpolierte Werte mit

```
double Get_Value    (double x);
```

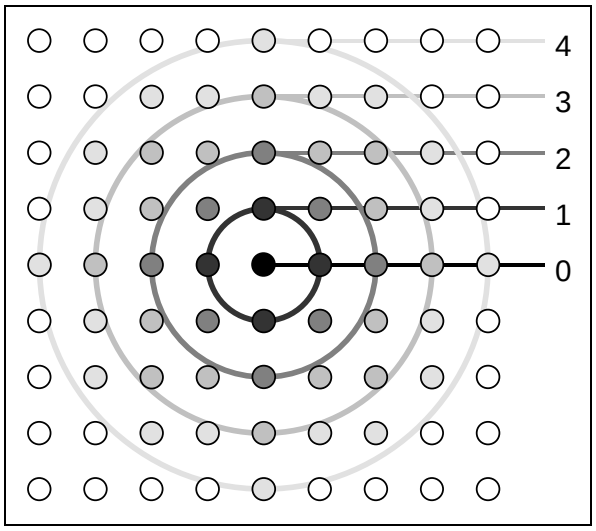


Abb. 13: Gitterpunkte im Umkreis

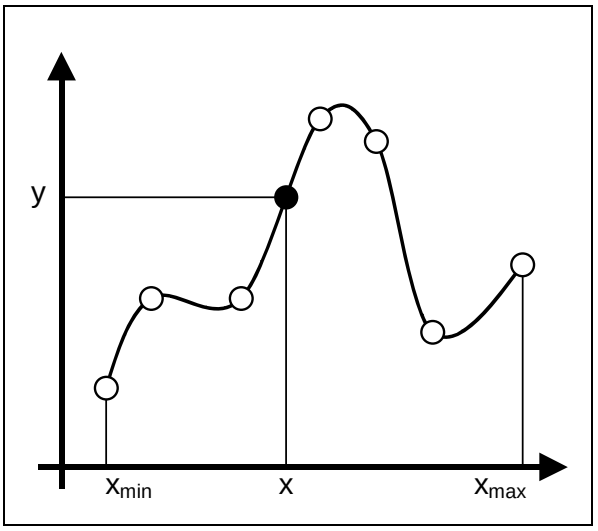


Abb. 14: Spline Interpolation

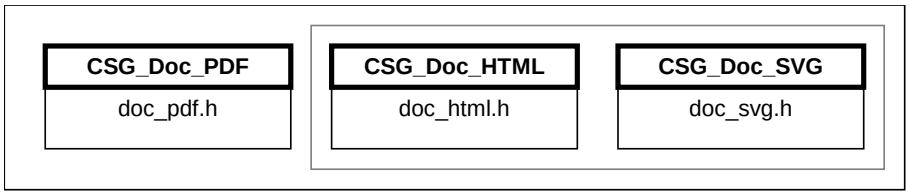


Abb. 15: Klassenübersicht – Dokumenterstellung

für beliebige *x* berechnet werden, wobei der zulässige Wertebereich mit

```
double Get_xMin (void);
double Get_xMax (void);
```

abgefragt werden kann (Abb. 14).

3.3.3 Dokumenterstellung

Die direkte Erstellung von Dokumenten einschließlich der automatisierten Darstellung von Tabellen und Karten wird zur Zeit von den Klassen *CSG_Doc_PDF* für das Portable Document Format (PDF) sowie *CSG_Doc_HTML* und *CSG_Doc_SVG* für webbasierte Inhalte in den Formaten Hypertext Markup Language (HTML) bzw. Scalable Vector Graphics (SVG) unterstützt. Die Verwendung dieser zur Zeit noch experimentellen Klassen funktioniert nach dem Schema:

- Erstellen eines leeren Dokuments mit der Funktion *Open*
- Zufügen von Inhalten, wie Seitenumbrüche, Text, Tabellen, Graphiken, durch spezielle Klassenmethoden
- Speichern des Dokuments mit der Funktion *Save*

Ein einfaches Beispiel für ein PDF-Dokument sieht z.B. so aus:

```
CSG_Doc_PDF pdf;

pdf.Open(PDF_PAGE_SIZE_A4, PDF_PAGE_ORIENTATION_PORTRAIT, "Test");

pdf.Draw_Text(10, 10, “This is a test.“, 12);
pdf.Draw_Line(10, 20, 50, 20);

pdf.Save(“C:\test.pdf“);
```

Eine besondere Stärke dieser Klassen sind Funktionen für die graphische Darstellung von Vektor- und Rasterdaten. Ein ausführliches Beispiel für die graphische Darstellung von Vektordaten in einem PDF-Dokument gibt das Modul *CSG_Shapes_Report* der Bibliothek **shapes_tools**.

3.3.4 Datenobjekte

Für die Arbeit mit Datensätzen, einfachen Tabellen wie solchen mit Raumbezug, stellt die API Datenobjekt-Klassen zur Verfügung. Zur Zeit können mit den Klassen *CSG_Table*, *CSG_Shapes*, *CSG_TIN* und *CSG_Grid* Tabellen, Vektordaten, TIN und Rasterdaten verwaltet werden. Für jeden Datenobjekttyp unterstützt die API mindestens ein Dateiformat, so dass sich Datensätze auch direkt speichern und laden lassen. Die Einführung in die

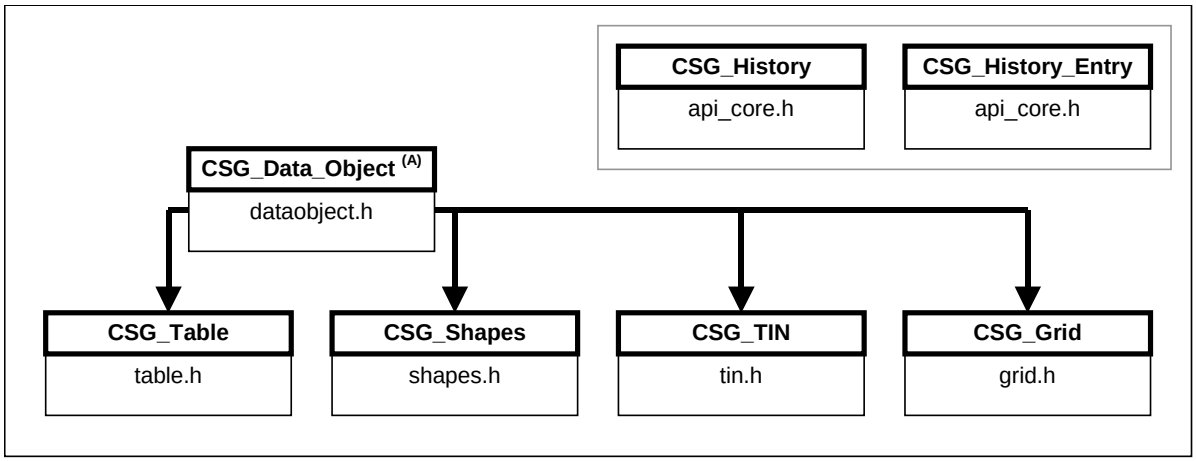


Abb. 16: Klassenübersicht – Datenobjekte

Modulprogrammierung (Kap. 3.7) gibt, mit Ausnahme der in SAGA bisher eher selten eingesetzten TIN-Klasse, ausführliche Anweisungen und Quelltextbeispiele für ihre Verwendung.

Basisklasse

Alle Datenobjekt-Klassen leiten sich von derselben Basisklasse ab. Diese Basisklasse *CSG_DataObject* ist eine abstrakte Klasse und hat die Aufgabe, Grundfunktionen zu definieren, die alle Datenobjekte gemeinsam haben (Tab. 13), und eine polymorphe Verwendung der abgeleiteten Klassen zu ermöglichen (s. Kap. 2.1.3). Einige dieser Funktionen müssen von den abgeleiteten Klassen überschrieben werden, um spezifische Eigenschaften des Datentyps berücksichtigen zu können. So ist zwar der Aufruf der Speicherfunktion *Save* für alle Datenobjekte gleich, unterscheidet sich aber durch ihre jeweilige Implementierung in den abgeleiteten Klassen.

Eine spezielle Eigenschaft von Datenobjekten ist ein *CSG_History*-Objekt, das Informationen über die Erstellung des Datensatzes bereithält. Die Klasse *CSG_History* ist ein Container für eine Liste von *CSG_History_Entry*-Objekten mit Angaben zu Verfahren und Eingangsdaten, die für die Datensatzerstellung benutzt wurden. Im einfachsten Fall, dass der Datensatz direkt aus einer Datei ausgelesen wurde, liegt nur ein einziger Eintrag mit dem Dateinamen vor. Wurde der Datensatz mit einem Modul erstellt, so wird die Geschichte von

Tab. 13: Grundfunktionen von Datenobjekten

Funktion	**Beschreibung**
bool Destroy(void)	Löscht alle Inhalte des Datenobjekts.
bool Save(const char *File_Name)	Speichert die Inhalte des Datenobjekts in der Datei *File_Name*.
const char * Get_File_Name(void)	Liefert den Dateinamen, der dem Datenobjekt zugeordnet ist.
void Set_Name(const char *Name)	Setzt einen deskriptiven Namen für das Datenobjekt.
const char * Get_Name(void)	Abfrage des deskriptiven Namens.
bool is_Modified(void)	Gibt an, ob die Datenobjekt-Inhalte geändert wurden.
CSG_History & Get_History(void)	Zugriff auf die Datenobjekt-Geschichte.

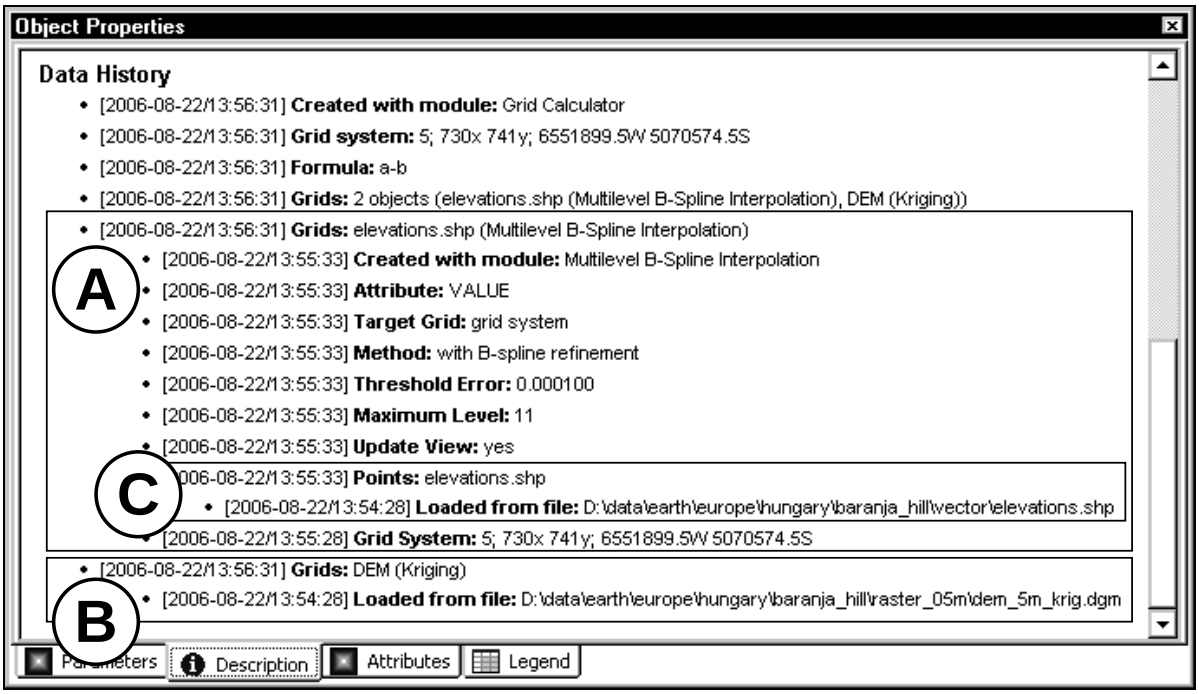


Abb. 17: Erstellungsgeschichte eines Datensatzes (Eingabedaten A, B, C)

der Modulbasisklasse (Kap. 3.3.5) automatisch durch Einträge ersetzt, die das Verfahren und die verwendeten Einstellungen beschreiben. Benutzte Eingabedatensätze werden mit ihrer eigenen Geschichte eingetragen, so dass sich für jeden Datensatz der komplette Weg, der zu seiner Erstellung führte, nachvollziehen lässt (Abb. 17). Beim Speichern eines Datenobjektes wird seine Geschichte in einer namensgleichen Datei mit einer der Dateierweiterungen

- „*htab*“ für Tabellen
- „*hshp*“ für Vektordaten
- „*htin*“ für TIN
- „*hgrd*“ für Rasterdaten

abgespeichert. Wenn eine solche Datei mit einer Datensatzgeschichte vorhanden ist, wird diese beim Laden des Datensatzes durch die Datenobjekt-Klasse automatisch wieder ausgelesen. Dank der automatisierten Geschichtsschreibung durch die Modulbasisklasse, muss der Modulprogrammierer nicht selbst auf diese Funktionalitäten zugreifen.

Tabellen

Die Klasse *CSG_Table* ist in SAGA verantwortlich für die Verwaltung von Tabellen. Tabellen speichern numerische Werte, Zeichen und Zeichenketten in Reihen, die oft als Datensätze der Tabelle bezeichnet werden. Da in dieser Arbeit auch die Gesamtdaten der Datenobjekte als Datensatz bezeichnet werden, wird für Tabellenreihen nachfolgend der englische Begriff *Record* benutzt. Records folgen einer für die gesamte Tabelle vorgegebenen Spalten- bzw. Felddefinition. Jedes Feld (engl. *field*) hat als Eigenschaft einen Namen und

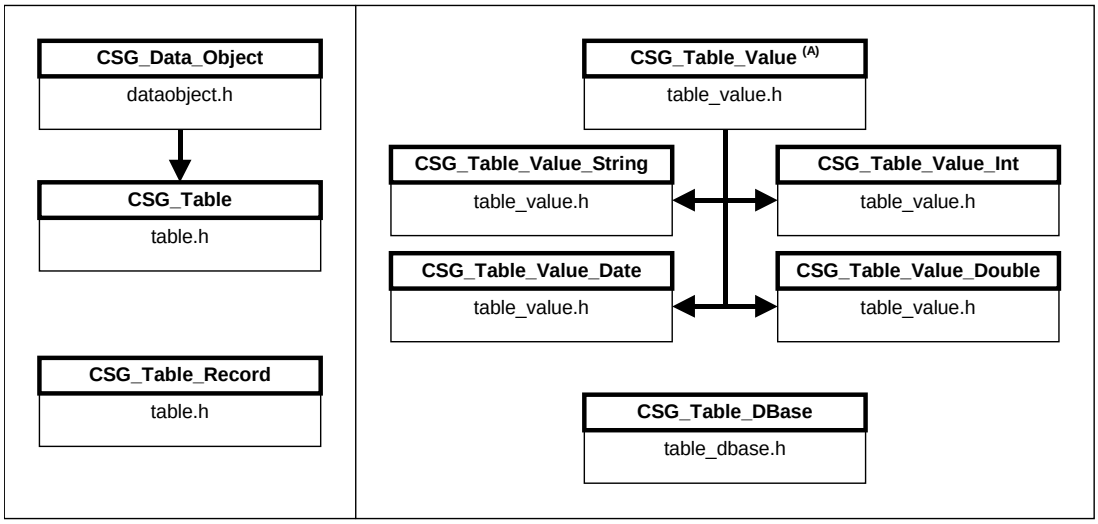


Abb. 18: Klassenübersicht – Tabellen

einen Datentyp, dem alle Spaltenwerte entsprechen müssen. Die Struktur einer Tabelle ist festgelegt durch die Anzahl ihrer Spalten und dem jeweiligen Datentyp der Spalte. Die Tabellenstruktur von *CSG_Table* kann jederzeit durch Einfügen oder Löschen von Spalten geändert werden. Der Zugriff auf die eigentlichen Tabellendaten erfolgt über ihre Records, die durch die Klasse *CSG_Table_Record* repräsentiert werden. Nachdem die Struktur einer Tabelle festgelegt ist, können Records in sie eingefügt oder gelöscht werden. *CSG_Table_Record* bietet Funktionen für das Auslesen und Schreiben von Werten mit automatischer Konvertierung von alphanumerischen in numerische Werte und umgekehrt. Der [ ] Operator wurde für *CSG_Table* und *CSG_Table_Record* so überschrieben, dass auf Tabelleneinträge auch in übersichtlicherer Weise zugegriffen werden kann:

```
   CSG_Table   Table;
...
   for(iRow=0; iRow<Table.Get_Record_Count(); iRow++)
      for(iCol=0; iCol<Table.Get_Field_Count(); iCol++)
         Sum   += Table[iRow][iCol];
```

Tabellenobjekte können entweder als Texttabellen oder im verbreiteten DBase-Format gespeichert und geladen werden. Bei Texttabellen werden die Records zeilenweise gespeichert. Als Trennzeichen für die Spalten eines Records wird ein Tabulatorzeichen benutzt. *CSG_Table* wird von den Datenobjektklassen *CSG_Shapes* und *CSG_TIN* benutzt, um Attributdaten zu speichern. Die weiteren in der Klassenübersicht aufgeführten Klassen *CSG_Table_DBase* sowie *CSG_Table_Value* und ihre Ableitungen werden nur intern von der API benutzt. Sie stehen dem Modulprogrammierer nicht direkt zur Verfügung und werden daher auch nicht weiter diskutiert.

Tab. 14: Funktionen für die Verarbeitung von Tabellen

Funktion	Beschreibung
CSG_Table	
int Get_Field_Count (void)	Anzahl der Spalten
void Add_Field (const char *Name, TTable_FieldType Type, int iField = -1)	Einfügen einer Spalte mit dem Namen *Name* und dem Datentyp *Type* an Position *iField*. Entspricht *iField* keiner gültigen Spalte, wird die Spalte der letzten angefügt.
bool Del_Field (int iField)	Löscht die Spalte an Position *iField*
int Get_Record_Count (void)	Anzahl der Records
CSG_Table_Record * Add_Record (CSG_Table_Record *pValues = NULL)	Hinzufügen eines Records. Wenn ungleich NULL, werden die Werte von *pValues* in den neuen Record kopiert.
CSG_Table_Record * Get_Record (int iRecord)	Zugriff auf den Record an Position *iRecord* der Tabelle
bool Del_Record (int iRecord)	Löscht den Record an Position *iRecord* der Tabelle
CSG_Table_Record	
const char * asString (int iField)	Gibt den Wert in Spalte *iField* als Zeichenkette zurück
const char * asDouble (int iField)	Gibt den Wert in Spalte *iField* als Fließkommazahl zurück
bool Set_Value (int iField, const char *Value)	Weist Spalte *iField* den Wert der Zeichenkette *Value* zu
bool Set_Value (int iField, double Value)	Weist Spalte *iField* den Wert von „*Value*“ zu

Vektordaten

Die Datenobjektklasse *CSG_Shapes* verwaltet Vektordaten in der Form einzelner *CSG_Shape*-Objekte. *CSG_Shape* ist eine abstrakte Basisklasse und bietet mit ihren Ableitungen spezialisierte Klassen an für Vektordaten vom Typ Punkt (*CSG_Shape_Point*), Punktliste (*CSG_Shape_Points*), Linie (*CSG_Shape_Line*) und Polygon (*CSG_Shape_Polygon*). Attributdaten werden von *CSG_Shapes* in einem Tabellenobjekt der Klasse *CSG_Table* gespeichert, wobei für jedes von *CSG_Shapes* verwaltete *CSG_Shape*-Objekt ein korrespondierender Record verfügbar ist. Der Zugriff auf die Attributdaten erfolgt entweder

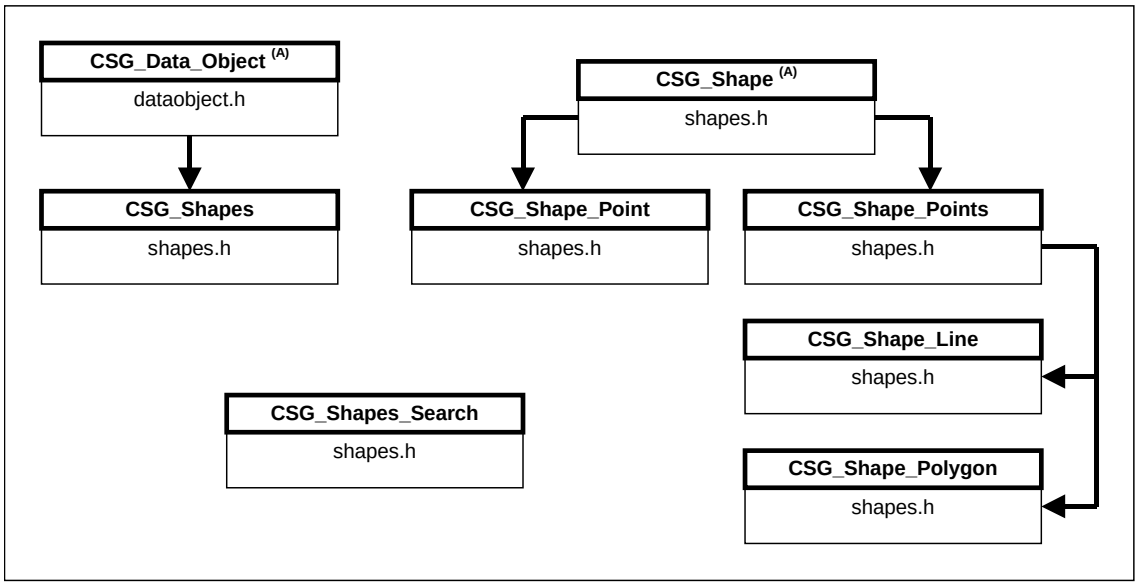


Abb. 19: Klassenübersicht – Vektordaten

über das Tabellenobjekt von *CSG_Shapes* oder über die Record-Eigenschaft des jeweiligen *CSG_Shape*-Objekts. Bevor von der *CSG_Shapes*-Klasse Vektordaten verwaltet werden können, muss der Vektordatentyp festgelegt und die Struktur der Attributdatentabelle definiert werden. Die Funktionen zum Einfügen und Löschen von *CSG_Shape*-Objekten sind angelehnt an die von *CSG_Table* zur Verwaltung von Records benutzten. Neu sind hier die Methoden zum Abfragen und Setzen der geometrischen Eigenschaften, also der Anordnung der Koordinatenpunkte von *CSG_Shape*-Objekten. Mit Ausnahme der Klasse *CSG_Shape_Point*, die immer genau einen Punkt verwaltet, können Punkte bzw. ihre Koordinaten beliebig zu einem *CSG_Shape*-Objekt hinzugefügt, geändert und gelöscht werden. Die Punkte können dabei verschiedenen Sequenzen oder Teilen (engl. *parts*) zugeordnet werden, z.B. um bei einem Polygon eine Teilfläche auszugrenzen (See). Eine Eigenschaft, die für einzelne *CSG_Shape*-Objekte wie auch für den gesamten Datensatz abgefragt werden kann, ist die rechteckige Ausdehnung (engl. *extent*). Neben einer allgemein verfügbaren Distanzfunktion, bieten die abgeleiteten Klassen weitere, für den jeweiligen Vektordatentyp spezifische Methoden an, wie z.B. die Flächenfunktion für Polygone. Für den lesenden und schreibenden Dateizugriff auf Vektordaten wird das verbreitete Shapefile-Format der Firma ESRI benutzt (ESRI 1998).

Mit Hilfe der ergänzenden Klasse *CSG_Shapes_Search* können schnell die nächsten

Tab. 15: Funktionen für die Verarbeitung von Vektordaten

Funktion	**Beschreibung**
CSG_Shapes	
int Get_Count(void)	Anzahl der *CSG_Shape*-Objekte
CSG_Shape * Get_Shape(int iShape)	Zugriff auf das *CSG_Shape*-Objekt an Position *iShape*
CSG_Shape * Add_Shape(CSG_Table_Record *pValues = NULL)	Hinzufügen eines *CSG_Shape*-Objektes. Attributdaten können mit *pValues* übergeben werden.
bool Del_Shape(int iShape)	Löscht ein *CSG_Shape*-Objekt
CSG_Rect Get_Extent(void)	Liefert die rechteckigen Erstreckung
CSG_Shape	
int Get_Part_Count(void)	Anzahl der vorhandenen Punktsequenzen
int Get_Point_Count(int iPart)	Anzahl der Punkte in einer Punktsequenz
TSG_Point Get_Point (int iPoint, int iPart)	Liefert die Koordinaten des Punktes *iPoint* der Sequenz *iPart*
int Add_Point (TSG_Point Point, int iPart)	Hinzufügen eines Punktes zur Sequenz *iPart*
int Del_Point (int iPoint, int iPart)	Löschen des Punktes *iPoint* der Sequenz *iPart*
CSG_Rect Get_Extent(void)	Liefert die rechteckigen Erstreckung
CSG_Shape_Line	
double Get_Length(void)	Länge der Linie
CSG_Shape_Polygon	
double Get_Area(void)	Fläche des Polygons
bool is_Containing(const TSG_Point &Point)	Abfrage, ob der Punkt *Point* des Polygons liegt.

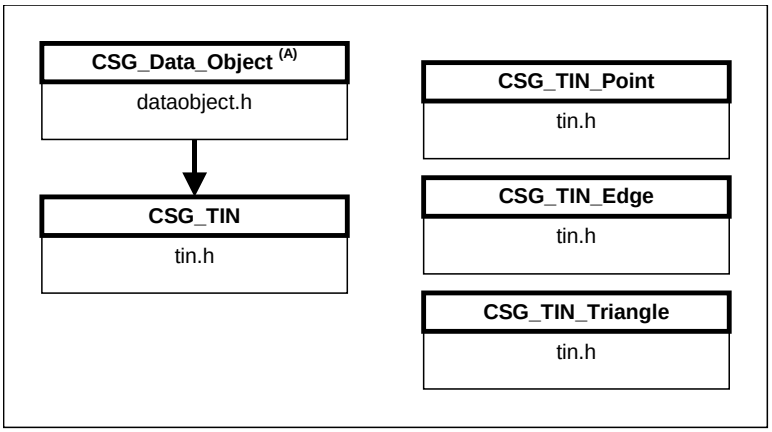


Abb. 20: Klassenübersicht – TIN

Punkte eines Vektordatensatzes für beliebige Koordinaten gefunden werden. Typische Beispiele für die Verwendung dieser Klasse geben einige Module zur Interpolation von Punktdaten der Bibliothek **grid_gridding**.

TIN

Die Klasse *CSG_TIN* baut auf einer Liste von Punkten der Klasse *CSG_TIN_Point* auf und verknüpft diese auf Basis einer Delaunay-Triangulation (z.B. LEE & SCHACHTER 1980). Ähnlich wie bei *CSG_Shapes*, werden Attributdaten für jeden Punkt in einem Tabellenobjekt gespeichert. Die Verknüpfung der Punkte untereinander kann mit entsprechenden Funktionen abgefragt werden. Alternativ kann auch auf die durch die Triangulation definierten Dreiecke (*CSG_TIN_Triangle*) und Kanten (*CSG_TIN_Edge*) zugegriffen werden. Zusatzfunktionen erlauben die Ableitung von Gradienten sowie Thiessen-Polygonen. Für TIN wurde kein

Tab. 16: Funktionen für die Verarbeitung von TIN

Funktion	**Beschreibung**
CSG_TIN	
CSG_TIN_Point * Add_Point(TSG_Point Point, CSG_Table_Record *pRecord)	Hinzufügen eines Punktes
bool Del_Point(int iPoint	Löschen eines Punktes
int Get_Point_Count(void)	Anzahl vorhandener Punkte
CSG_TIN_Point * Get_Point (int Index)	Abfrage eines Punktes
int Get_Edge_Count (void)	Anzahl vorhandener Kanten
CSG_TIN_Edge * Get_Edge (int Index)	Abfrage einer Kante
int Get_Triangle_Count(void)	Anzahl vorhandener Dreiecke
CSG_TIN_Triangle * Get_Triangle (int Index)	Abfrage eines Dreiecks
CSG_Rect Get_Extent(void)	Liefert die rechteckigen Erstreckung
CSG_TIN_Point	
int Get_Neighbor_Count(void)	Anzahl der Nachbarpunkte
CSG_TIN_Point * Get_Neighbor(int iNeighbor)	Liefert den mit *iNeighbor* abgefragten Nachbarpunkt
bool Get_Polygon(CSG_Points &Points)	Liefert das zugehörige Thiessen-Polygon als Punktliste

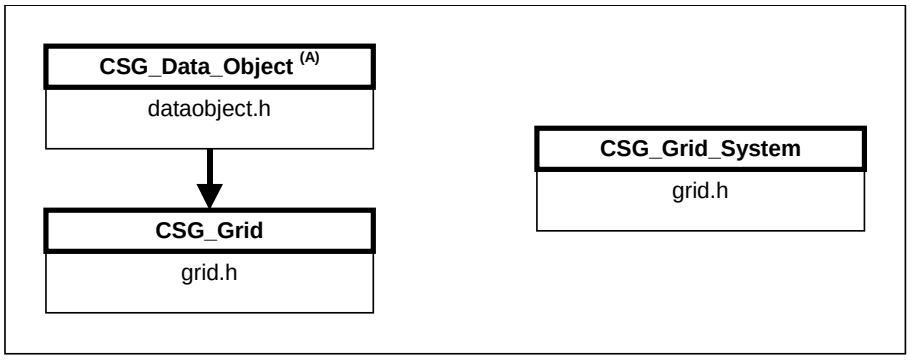


Abb. 21: Klassenübersicht – Rasterdaten

eigenes Dateiformat eingeführt. Da die Delaunay-Triangulation immer eindeutige Ergebnisse für eine bestimmte Punktmenge liefert, werden nur die Punkte mit ihren Attributen im ESRI-Shapefile Format gespeichert bzw. geladen (ESRI 1998). TIN werden u.a. bei der Interpolation von Punktdaten von dem Modul *CInterpolation_Triangulation* der Bibliothek **grid_gridding** benutzt.

Rasterdaten

Die Klasse *CSG_Grid* für die Verwaltung von Rasterdaten ist vermutlich die ausgereifteste Komponente der API und wurzelt mit ihren Anfängen noch in dem Programm DiGeM (s. Kap. 1.2). Ein wesentliches Element von *CSG_Grid* ist ein Objekt der Klasse *CSG_Grid_System*. Mittels *CSG_Grid_System* werden die Grundeigenschaften eines Rasters definiert, die sich aus der Spalten- und Reihenanzahl, der Rasterweite und den Koordinaten eines Bezugspunkts, des linken unteren Gitterpunkts des Rasters, zusammensetzen. Die Klasse bietet eine Reihe nützlicher Hilfsfunktion, u.a. zum Umrechnen der Position einer Rasterzelle in Weltkoordinaten, zum Berechnen der Fläche einer Rasterzelle oder zum Vergleich auf Übereinstimmung mit anderen Rastersystemen. Für eine einfachere Handhabung können die am häufigsten gebrauchten dieser Funktionen durch gleichlautende der *CSG_Grid*-Klasse aufgerufen werden. Um mit einem Rasterdatensatz arbeiten zu können, muss das Rastersystem zunächst definiert werden. Daneben muss auch angegeben werden, welchem Datentyp die Rasterwerte entsprechen. Dies können zur Zeit entweder Ganzzahlen mit einer Datentiefe von 1 Bit sowie 1, 2, 4 oder 8 Byte sein oder Fließkommazahlen mit einer Datentiefe von 4 oder 8 Byte. Für das Setzen und Abfragen von Rasterzellen stehen entsprechende Funktionen zur Verfügung. Es kann auch ein Wert bzw. ein Wertebereich gesetzt werden, der als fehlender Datenwert (engl. *no data*) interpretiert werden soll, und ebenfalls zellenweise gesetzt bzw. getestet werden kann. Die *CSG_Grid*-Klasse erstellt bei Bedarf einen Index, so dass die Rasterzellen in der Sortierreihenfolge vom höchsten zum niedrigsten Wert hin angesprochen und auch Perzentile abgelesen werden können. Für die Verwaltung großer Datensätze bietet die *CSG_Grid*-Klasse die Verwendung eines Dateipuffers (engl. *file cache*) an, bei dessen Verwendung immer nur eine festgelegte Datenmenge im Speicher gehalten wird und abgefragte Rasterzellenwerte nur bei Bedarf nachgeladen werden. Für Datensätze mit seltenen Wertewechseln zwischen benachbarten

Tab. 17: Funktionen für die Verarbeitung von Rasterdaten

Funktion	Beschreibung
CSG_Grid	
int Get_NX(void), int Get_NY(void)	Anzahl der Spalten und Zeilen
long Get_NCells(void)	Anzahl der Rasterzellen
double Get_Cellsize(void)	Rasterweite
double Get_Cellarea(void)	Fläche einer Rasterzelle
const CSG_Rect & Get_Extent(void)	Erstreckung des Rasters
double Get_ZMin(), double Get_ZMax(), double Get_ArithMean(), double Get_Variance(), double Get_Percentile(double Percent)	Minimum, Maximum, Mittelwert, Varianz und Perzentil der Rasterzellenwerte
double Get_NoData_Value(void), double Set_NoData_Value(double Value)	Abfrage und Setzen eines *No-Data* Wertes
void Set_Value(int x, int y, double Value)	Setzen des Wertes in Spalte *x* und Zeile *y* auf *Value*
double asDouble(int x, int y)	Abfrage des Wertes in Spalte *x* und Zeile *y*
double Get_Value(TSG_Point Point)	Abfrage des Wertes an den Weltkoordinaten *Point* mit optionaler Interpolation
bool Get_Sorted(long Position, int &x, int &y)	Abfrage der Position vom höchsten zum niedrigsten Wert
CSG_Grid_System	
TSG_Point Get_Grid_to_World(int x, int y)	Weltkoordinaten der Spalte *x* und Zeile *y*
bool Get_World_to_Grid(int &x, int &y, TSG_Point ptWorld)	Berechnung von Spalte *x* und Zeile *y* für die Weltkoordinaten in *ptWorld*
double Get_Length(int Direction)	Distanz zu einer der 8 mit *Direction* angegebenen Nachbarzellen
bool operator == (const CSG_Grid_System &System)	Vergleich auf Übereinstimmung mit einem anderen Rastersystem

Rasterzellen, wie er z.B. bei klassifizierten Daten zu finden ist, kann erhebliche Speicherplatzersparnis auch durch eine Laufzeitlängenkomprimierung (RTL, von engl. *run time length compression*) erzielt werden. Für den Dateizugriff auf Rasterdaten wird ein einfaches und flexibles, aber leider auch SAGA-spezifisches Dateiformat verwendet, das noch auf das Programm DiGeM zurückgeht. Das Dateiformat trennt die eigentlichen Daten von den Metainformationen. Die Daten werden in einer Datei mit der Endung „*sdat*“ sequentiell gespeichert, entweder als maschinenlesbarer Binärcode oder als Textdatei mit Zahlenwerten. Die Information zur Interpretation der Daten befindet sich in einer Textdatei mit der Endung „*sgrd*“. Hierin sind die Werte für die Anzahl der Zeilen und Spalten, der Rasterweite, der Georeferenz und der Zahlencodierung durch entsprechende Schlüsselworte gekennzeichnet.

3.3.5 Module

Auch wenn die API selbst keine Module enthält, so liefert sie doch alle für die Modulerstellung benötigten Grundlagen. Neben vier Modulklassen gehört hierzu eine Klasse zum Verwalten von Parameterlisten und eine Klasse für die Definition der Modulbibliotheks-

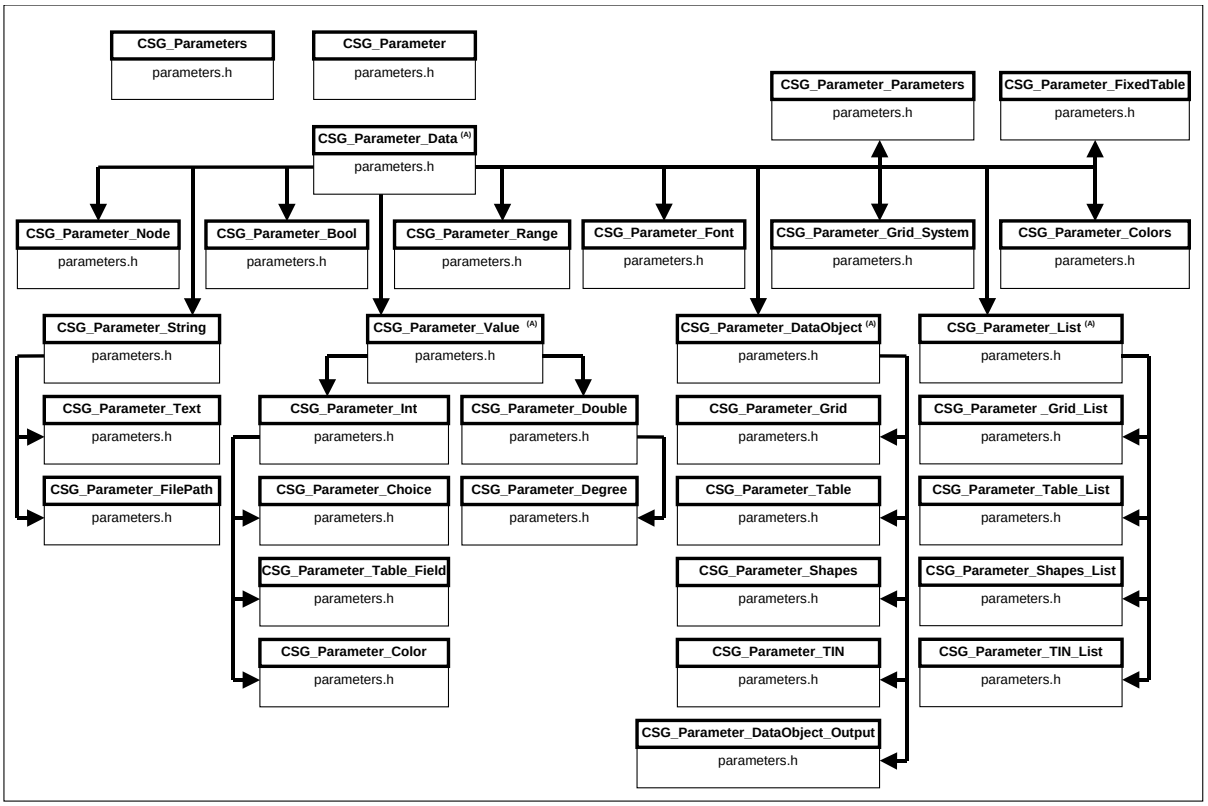


Abb. 22: Klassenübersicht – Parameterlisten

schnittstelle. Die Modulerstellung steht im Mittelpunkt von Kapitel 3.7, so dass hier auf vertiefende Programmierbeispiele verzichtet wird.

Parameterlisten

Für die Verwendung von Modulen ist es unumgänglich dem Benutzer die Möglichkeit zu geben, die Modulausführung zu steuern. Neben den obligatorischen Eingabedaten wird ein Modul auch über verschiedene andere Eingabeparameter verfügen, mit der sich die Ergebnisse der Modulausführung beeinflussen lassen. Schließlich muss auch spezifiziert werden, welche Datenobjekte die Ergebnisse aufnehmen sollen. Mit der Klasse *CSG_Parameters* bietet die API ein flexibles Werkzeug an, um diesen Anforderungen gerecht zu werden. *CSG_Parameters* verwaltet eine Liste von *CSG_Parameter*-Objekten, die wiederum verschiedenste Datentypen repräsentieren können. Die *CSG_Parameter*-Klasse kapselt ein Objekt vom Typ der Klasse *CSG_Parameter_Data*, von der es spezielle Ableitungen für alle benötigten Arten von Parametern gibt (Abb. 22, Tab. 19). Der Modulprogrammierer wird aber nur in seltenen Fällen direkt auf das *CSG_Parameter_Data*-Objekt zugreifen müssen, da seine am häufigsten gebrauchten Methoden durch entsprechende Funktionen von *CSG_Parameter* aufrufbar sind.

Die verfügbaren Parametertypen teilen sich in zwei Kategorien. Die erste Kategorie besteht aus Parametern, die für numerische Werte, alphanumerische Werte oder einfache Datenstrukturen stehen. Es handelt sich ausschließlich um Eingabeparameter, mit der die Modulausführung gesteuert wird. Einfache Eingabeparameter entsprechen Zahlenwerten oder Zeichenket-

Tab. 18: Funktionen für Parameterlisten

Funktion	Beschreibung
CSG_Parameters	
int Get_Count(void)	Anzahl vorhandener Parameter
CSG_Parameter * Get_Parameter(const char *Identifier), CSG_Parameter * operator() (const char *Identifier)	Zugriff auf den mit *Identifier* spezifizierten Parameter. Der ()-Operator kann in gleicher Weise benutzt werden.
CSG_Parameter * Get_Parameter(int iParameter), CSG_Parameter * operator() (int iParameter)	Zugriff auf den Parameter an Position *iParameter* der Liste. Der ()-Operator kann in gleicher Weise benutzt werden.
CSG_Parameter * Add_Value(...), CSG_Parameter * Add_String(...), CSG_Parameter * Add_Table(...), CSG_Parameter * Add_Shapes(...), CSG_Parameter * Add_Grid(...), ...	Funktionen zum Hinzufügen von Parametern verschiedenen Typs zur Parameterliste
bool Serialize (const char *File_Name, bool bSave)	Speichern und Laden der Parameterlistendatei *File_Name*. *bSave* gibt an, ob geladen oder gespeichert werden soll.
CSG_Parameter	
TParameter_Type Get_Type(void)	Abfrage des Parametertyps
const char * Get_Name(void)	Abfrage des Parameternamens
const char * Get_Description(void)	Abfrage der Parameterbeschreibung
bool Set_Value(double Value), bool Set_Value(const char *Value), bool Set_Value(void *Value) ...	Verschiedene Versionen der Funktion zum Setzen des Parameterwertes
double asDouble(void), const char * asString(void), CSG_Table * asTable(void), CSG_Shapes * asShapes(void), CSG_Grid * asGrid(void), ...	Verschiedene Funktionen zum Abfragen des Parameterwertes. Die jeweils passende Funktion hängt vom Parametertyp ab und muss vom Modulprogrammierer entsprechend gewählt werden.

ten. Parametertypen wie *Double* und *Degree* unterscheiden sich nur darin, wie sie durch die Benutzeroberfläche dargestellt werden. Komplexer aufgebaute Parameter sind Farbpaletten, Rastersysteme sowie einfache Tabellen, deren Struktur bei der Parameterdefinition festgelegt wird. Die zweite Kategorie von Parametern bezieht sich auf die Datenobjekte, die als Eingabe dienen oder die Ausgabeergebnisse aufnehmen. Für alle in Kap. 3.3.4 aufgeführten Datenobjekttypen sind entsprechende Parametertypen vorhanden. Darüber hinaus stehen auch variable Listen von Datenobjekten zur Verfügung. Der Parametertyp *Node* fällt aus diesen beiden Kategorien heraus, denn er steht nicht für einen Wert und wird nur benutzt, um andere Parameter zu gliedern, so dass sie durch die Benutzeroberfläche übersichtlicher dargestellt werden.

Das Erstellen von Parametern erfolgt immer durch die entsprechenden Methoden der Parameterliste. Bei Modulen werden die Parameterwerte durch die Benutzeroberfläche gesetzt. Für das Abfragen der gesetzten Werte stehen verschiedenen Funktionen zur Verfügung, wobei es in der Verantwortung des Modulprogrammierers liegt, die für den jeweiligen Parametertyp passende Funktion zu wählen (Tab. 18). Parameterlisten werden nicht nur von den nachfolgend vorgestellten Modulklassen benutzt. Auch die SAGA-GUI verdankt einen großen Teil ihrer Flexibilität der vielseitigen Verwendbarkeit der *CSG_Parameters* Klasse.

Tab. 19: Parametertypen

Parametertyp	**Bedeutung**	**Parametertyp**	**Bedeutung**
Node	Gliederung von Parametern	Colors	Farbpalette (Farbwertliste)
Bool	Wahrheitswert	Fixed Table	Tabelle mit festgelegter Struktur
Integer	Ganzzahl	Grid System	Rastersystem
Double	Fließkommazahl	Table Field	Auswahl einer Tabellenspalte
Degree	Fließkommazahl, Grad-Darstellung	Grid	Raster
Range	Wertebereich (Fließkommazahl)	Table	Tabelle
Choice	Auswahlliste (Ganzzahl)	Shapes	Vektordaten
String	Einfache Zeichenkette (Text)	TIN	TIN
Text	Längere Zeichenketten (Text)	Grid List	Liste von Rastern
Filepath	Dateiname (Text)	Table List	Liste von Tabellen
Font	Schrifttyp	Shapes List	Liste von Vektordaten
Color	Farbwert (Ganzzahl)	TIN List	Liste von TIN

Module

Im Gegensatz zu allen bisher vorgestellten Klassen werden die Modulklassen vom Modulprogrammierer nicht einfach nur benutzt. Sie dienen vielmehr als Schablonen für die Erstellung eigener Modulklassen. Die Funktionalitäten, die nachfolgend vorgestellt werden, gehen daher nicht darauf ein, wie die Instanz einer Modulklasse quasi von außen verwendet wird, denn das ist u.a. die Aufgabe der Modulverwaltung durch die Benutzeroberfläche. Für den Modulprogrammierer ist es vor allem von Interesse, welche internen Funktionen die Modulklassen für die Unterstützung der Programmierung anbietet. Tatsächlich sind alle Modulklassen abstrakt, so dass Instanzen von ihnen selbst nicht erstellt werden können. Der erste Schritt bei der Modulerstellung ist es nun eine neue Klasse von einer der hier bereitgestellten Modulklassen abzuleiten. Die abgeleitete Klasse muss als zweites mindestens die abstrakte Methode *On_Execute* überschreiben, damit sie instanziierbar und ausführbar wird. Diese Methode ist elementar, denn wenn ein Modul ausgeführt werden soll, führt die Basisklasse *CSG_Module* eine Reihe von Initialisierungen durch, u.a. die Überprüfung auf gültige Datenobjekte, und ruft danach diese Methode auf. Damit ist die Funktion *On_Execute* genau die Stelle, an der die Implementierung der modulspezifischen Funktion vorgenommen werden muss.

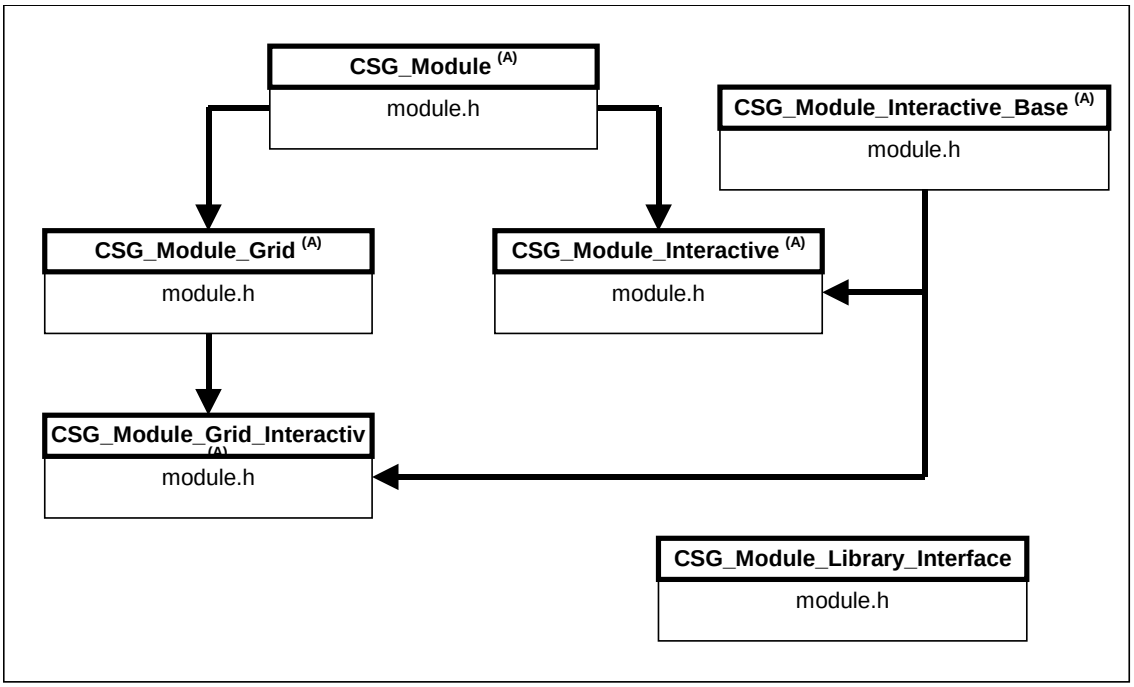


Abb. 23: Klassenübersicht – Module

Die Basisklasse aller Modulklassen ist *CSG_Module*, die sämtliche Grundfunktionen von Modulen bereitstellt. Die Klasse *CSG_Module_Grid* vereinfacht zusätzlich das Programmieren von Modulen, die mit einem festen Rastersystem arbeiten. Grundfunktionen für Module, die interaktiv ausgeführt werden, definiert die Klasse *CSG_Module_Interactive_Base*. Diese Klasse kann nicht direkt verwendet werden, sondern dient als Basisklasse für *CSG_Module_Interactive* und *CSG_Module_Grid_Interactive*. Die interaktive Ausführung wird durch Aufruf der *On_Execute*-Funktion gestartet. Damit danach auf Maus- bzw. Tastaturereignisse interaktiv reagiert werden kann, stehen dem Modulprogrammierer zwei gesonderte, überschreibbare Funktionen zur Verfügung, die bei solchen Ereignissen mit zusätzlichen Informationen über das Ereignis von der SAGA-Umgebung aufgerufen werden. Eine weitere überschreibbare Funktion ermöglicht beim Beenden der interaktiven Modulausführung abschließende Aufgaben durchzuführen (Tab. 20).

Für alle Module können allgemeine Informationen, wie Name, Beschreibung und Autorenrechte, gesetzt werden. Die meisten anderen Funktionen dienen der Kommunikation mit der Benutzeroberfläche. Von besonderer Bedeutung ist das Parameterlistenobjekt *Parameters*, mit dem der Programmierer die modulspezifischen Parameter festlegt. Typischerweise werden die Parameter im Konstruktor der abgeleiteten Modulklasse definiert, so dass diese sofort nach Erstellung einer Modulinstanz von der Benutzeroberfläche abgefragt werden können. Bei der Modulausführung wird dem Benutzer vor Aufruf der Funktion *On_Execute* die Gelegenheit gegeben, die Parameterwerte in seinem Sinne zu setzen.

Tab. 20: Funktionen der Modulklassen

Funktion	Beschreibung
CSG_Module	
virtual bool On_Execute(void)	Diese Funktion muss mit eigener Funktionalität überschrieben werden. Wird bei der Ausführung aufgerufen.
CSG_Parameters Parameters	Die Parameterliste des Moduls
void Set_Name(const char *Name), void Set_Description(const char *Text), void Set_Author(const char *Author)	Setzen von Name, Beschreibung und Autor des Moduls
bool Process_Get_Okay(void)	Überprüft, ob der Benutzer die Modulausführung abbrechen möchte
bool Set_Progress(double Position, double Range)	Setzen der Fortschrittsanzeige der Benuzteroberfläche. Der Rückgabewert gibt an, ob der Benutzer die Modulausführung abbrechen möchte.
void Message_Add(const char *Text)	Übermittelt der Benutzeroberfläche eine Nachricht zum Hinzufügen in die Nachrichtenliste
bool Error_Set(const char *Text)	Ausgabe einer Fehlermeldung und Abfrage, ob die Modulausführung gestoppt werden soll.
bool DataObject_Update (CSG_DataObject *pDataObject, bool bShow)	Mitteilung an die Benutzeroberfläche, dass sich ein Datensatz geändert hat. *bShow* wird veranlasst, dass der Datensatz, wenn noch nicht geschehen, angezeigt wird
CSG_Module_Grid	
CSG_Grid_System * Get_System(void)	Liefert während der Modulausführung das von der Benutzeroberfläche gesetzte Rastersystem
int Get_NX(), int Get_NY(), int Get_NCells()	Anzahl der Rasterzellen in Spalten, Reihen und gesamt
CSG_Module_Interactive_Base	
virtual bool On_Execute_Position(CSG_Point ptWorld, TModule_Interactive_Mode Mode)	Diese Funktion wird aufgerufen, wenn in der GUI eine Mausaktion in einem Kartenfenster ausgeführt wird. *ptWorld* enthält die Weltkoordinaten und *Mode* bezeichnet die Art der Aktion (z.B. Doppelklick der rechten Maustaste)
virtual bool On_Execute_Keyboard(int Character)	Diese Funktion wird aufgerufen, wenn in der GUI die Tastatur betätigt wird. *Character* enthält das Tastaturzeichen.
virtual bool On_Execute_Finish(void)	Diese Funktion wird aufgerufen, wenn die interaktive Modulausführung beendet wird.
void Set_Drag_Mode(int Drag_Mode)	Graphische Darstellung beim Ziehen der Maus mit gedrückter Taste (z.B. Linie, Rechteck oder Kreis)
bool is_Shift(void)	Abfrage, ob die Hochstelltaste gedrückt ist.

Modulbibliotheken

Damit ein Modul von einem der SAGA Front Ends ausgeführt werden kann, muss es in einer SAGA-Modulbibliothek eingebunden sein. Modulbibliotheken sind dynamisch ladbare Programmbibliotheken, die über eine spezielle Schnittstelle verfügen. Aufgabe dieser Schnittstelle ist es, ausführbare Instanzen von den in der Bibliothek enthaltenen Modulen zu erstellen. Daneben übernimmt die Schnittstelle weitere Aufgaben, wie das automatische Laden von Übersetzungstabellen oder die Ausgabe allgemeiner Informationen über die Bibliothek. Ein Großteil der genannten Aufgaben wird von der Klasse *CSG_Module_Library_Interface*

abgedeckt. Der Modulprogrammierer muss mit dieser Klasse jedoch nicht direkt arbeiten. Ihr Anteil an der Schnittstellefunktionalität wird in sehr einfacher Weise durch die Verwendung des Makros *MLB_INTERFACE* implementiert. Um die Schnittstelle zu vervollständigen, muss der Modulprogrammierer allerdings zwei Funktionen definieren, die im Quelltext vor dem Makro auftauchen müssen. Beide Funktionen müssen eine Anweisung zur bedingten Verzweigung enthalten, um in Abhängigkeit von dem an sie übergebenen Funktionsparameter *i* einen Rückgabewert zu liefern. Die Rückgabewerte der ersten Funktion

```
const char *    Get_Info (int i)
```

sind Texte (Zeichenketten) mit allgemeinen Informationen zur Bibliothek. Die zweite Funktion

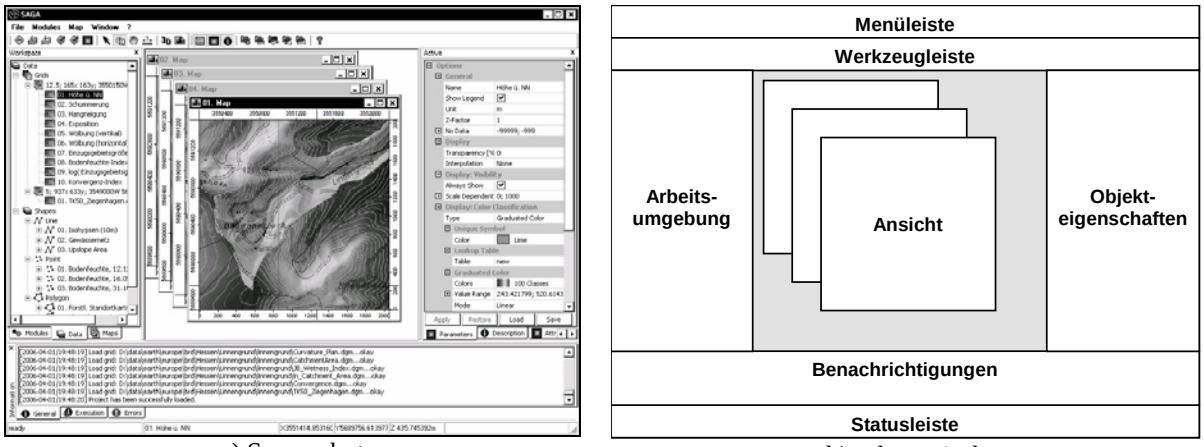
```
CSG_Module *    Create_Module(int i)
```

liefert die Instanzen der verfügbaren Module. Das Makro *MLB_INTERFACE* kann direkt danach folgen. Eine vollständige Schnittstelle zeigt das abschließende Quelltextbeispiel.

```
#include <saga_api/saga_api.h>  // Einbindung der SAGA-API Header

#include "exercise_01.h"        // Einbindung der Header mit den
#include "exercise_02.h"        // Klassendeklarationen fuer die
#include "exercise_03.h"        // Module der Bibliothek

const char * Get_Info(int i)    // ...liefert Informationen ueber die Bibliothek
{
   switch( i )
   {
   case MLB_INFO_Name:          return( "Modul-Programmierung" );
   case MLB_INFO_Author:        return( "Olaf Conrad" );
   case MLB_INFO_Description:   return( "Eine Einführung in die Modul-Programmierung." );
   case MLB_INFO_Version:       return( "1.0" );
   case MLB_INFO_Menu_Path:     return( "Modul-Programmierung" );
   }

   return( NULL );
}

CSG_Module * Create_Module(int i)  // ...erstellt die Modulinstanzen
{
   switch( i )
   {
   case 0:  return( new CExercise_01 );
   case 1:  return( new CExercise_02 );
   case 2:  return( new CExercise_03 );
   }

   return( NULL );
}

MLB_INTERFACE                   // Vollstaendige Implementierung der Schnittstelle
```

3.4 Graphische Benutzerschnittstelle

Die graphische Benutzeroberfläche (GUI) ist als standardmäßiges Front End für SAGA vorgesehen. Durch die Verwendung graphischer Steuerelemente ermöglicht die GUI eine weitgehend intuitive und komfortable Bedienung des Systems. Daneben gehört die Datenvisualisierung zu ihren wichtigsten Aufgabenfeldern. Eine wesentliche Vereinfachung graphischer Benutzeroberflächen resultiert aus der Verwendung des Mauszeigers, durch die die sonst übliche Tastatureingabe in vereinfachender Weise ergänzt wird. Die GUI macht exzessiven Gebrauch von den Funktionalitäten der SAGA-API, z.B. um Module und Datensätze



a) Screenshot b) schematisch

Abb. 24: GUI Aufbau

anzusprechen und zu verwalten. Für die Programmierung der GUI spielt die betriebssystemübergreifende GUI-Bibliothek wxWidgets (SMART et al. 2005) eine entscheidende Rolle. U.a. ist ihre Verwendung auch dafür verantwortlich, dass sich das System sowohl unter Windows als auch unter Linux Betriebssystemen in gleicher Weise verwenden lässt. Die GUI benutzt die in Kapitel 3.3.1 vorgestellte API-Funktionalität für die automatische Übersetzung von Zeichenketten, um so eine Internationalisierung ihrer Oberfläche zu erlauben. Standardmäßig sind alle Texte in der GUI in englischer Sprache abgefasst, weshalb sich auch im nachfolgenden auf die englischsprachigen Begriffe bezogen wird. Eine vorläufige Übersetzungstabelle für eine deutsche Sprachversion liegt mit der Datei „*saga.ger*“ den aktuellen SAGA-Distributionen bei. Durch Umbenennung der Datei in „*saga.lng*“ und anschließendem Neustart der GUI wird die Übersetzungstabelle aktiviert. Auch wenn die GUI einer intuitiven Bedienung des Systems sehr entgegenkommt, bedürfen die vielfältigen Möglichkeiten, wie auch der prinzipielle, SAGA-typische Aufbau der Oberfläche im nachfolgenden einer näheren Erläuterung.

3.4.1 Aufbau

Die GUI verfügt über eine Reihe von Kontrollelementen, die für die meisten graphischen Benutzeroberflächen typisch sind, wie Befehlsmenüs, Werkzeug- und Statusleisten. Darüber hinaus gibt es Kontrollelemente, die für die Verwendung mit SAGA entworfen wurden und in dieser spezialisierten Form nicht bei anderen Benutzeroberflächen zu finden sind. Ergänzt werden diese Kontrollelemente durch verschiedene Arten von Datenansichten, die in gesonderten Fenstern (engl. *windows*) dargestellt werden und z.T. zusätzliche Möglichkeiten der interaktiven Steuerung anbieten. Abb. 24 gibt eine Übersicht über den prinzipiellen Aufbau der GUI. Das Befehlsmenü enthält vier immer verfügbare Haupteinträge für Dateibefehle („*File*“), modulbezogene Befehle („*Modules*“), Befehle zur Anordnung von Kontrollelementen und Fenstern mit Datenansichten („*Window*“) sowie einem Eintrag zum Aufruf von Programminformation und Hilfe („*?*“). Dateibefehle dienen zum Öffnen von Datensätzen und zum Beenden des Programms. Über die Modulbefehle lassen sich Modulbibliotheken laden und deren Module ausführen.

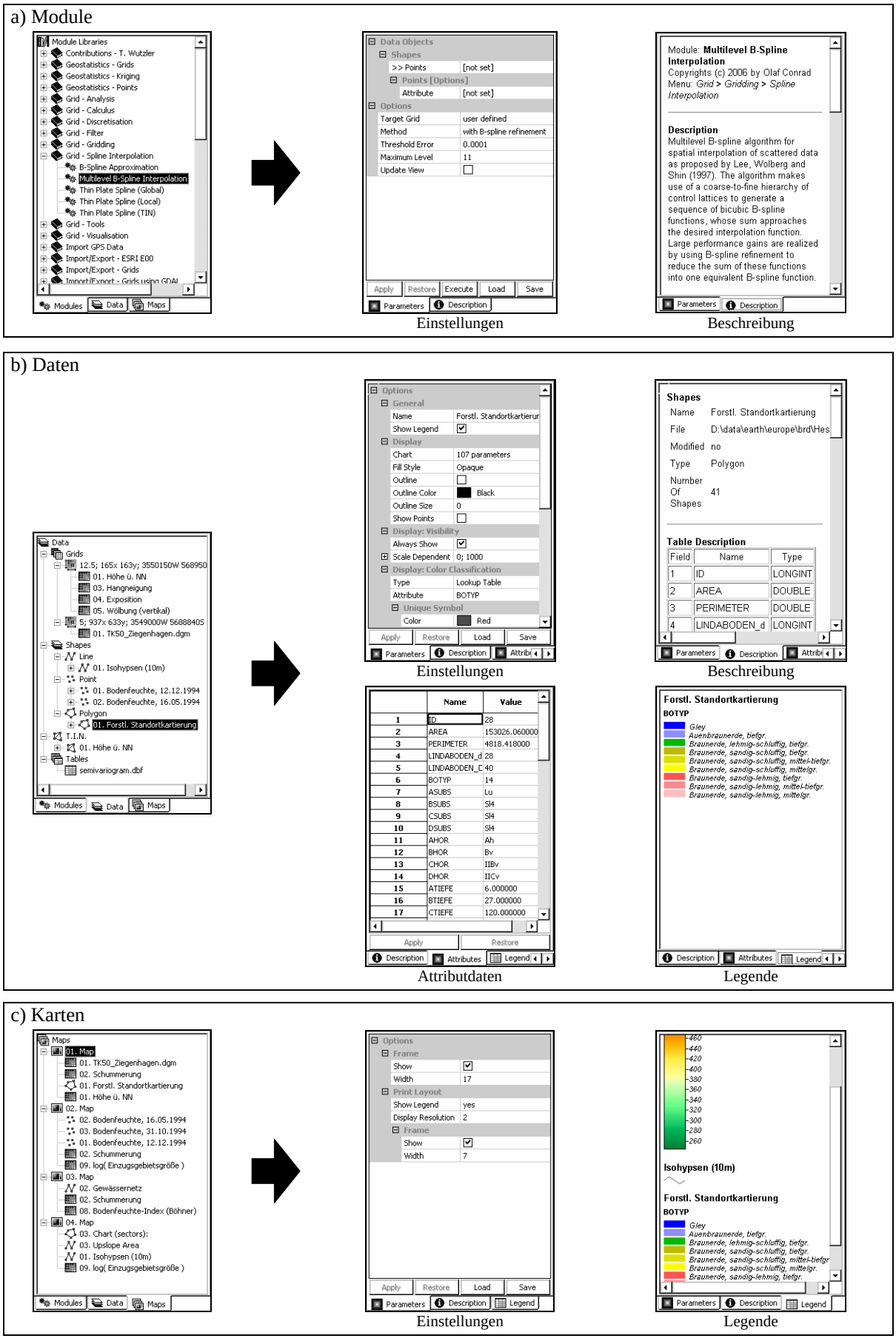


Abb. 25: Arbeitsumgebung und Objekteigenschaften

Ist ein Fenster mit einer Datenansicht geöffnet, so enthält die Menüleiste einen weiteren Eintrag für Befehle, über die sich die Datenansicht kontrollieren lässt. Häufig gebrauchte Menübefehle können auch über Schaltflächen in der Werkzeugleiste ausgeführt werden, die durch besondere Symbole gekennzeichnet sind. In der Statusleiste werden Informationen über den aktuellen Status des Systems angezeigt, z.B. der Fortschritt einer Berechnung oder die Werte eines Datensatzes an der Position des Mauszeigers. Ausführlichere Informationen werden in die drei Benachrichtigungsfenster („*Messages*“) ausgegeben und stehen dort permanent zur Verfügung bis das Programm beendet wird. Die Unterteilung erfolgt in allgemeine Benachrichtigungen („*General*“), z.B. über das Laden und Schließen von Datensätzen und Modulbibliotheken oder das Ausführen eines Moduls, spezielle Information über die Ausführung eines Moduls („*Execution*“), z.B. die verwendeten Modulparameter, und Fehlermeldungen („*Errors*“), z.B. bei der Modulausführung.

Auch wenn sich ein großer Teil der Funktionalitäten bereits über die Menüleiste erreichen lässt, erfolgt die Verwaltung der Module, Datensätze und Datenansichten am einfachsten über die sogenannte Arbeitsumgebung („*Workspace*“) und die mit ihr assoziierten Objekteigenschaften („*Object Properties*“). Für eine bessere Übersichtlichkeit besitzt die Arbeitsumgebung je ein Unterfenster für Module, Daten und Karten, der Standardansicht für räumliche Datensätze. Zwischen den Unterfenstern kann z.B. durch Mausklick auf den entsprechend benannten Reiter gewechselt werden. (Reiter sind kleine Schaltflächen). Jedes Unterfenster gliedert die in ihr enthaltenen Objekte in einer hierarchisch strukturierten Baumansicht. Die Objekte lassen sich mit dem Mauszeiger oder der Tastatur auswählen. Für jedes Objekt kann mit der rechten Maustaste ein Kontextmenü mit objektspezifischen Befehlen aufgerufen werden. Die Eigenschaften des ausgewählten Objekts sind in dem Fenster mit den Objekteigenschaften verfügbar. Auch dieses Fenster besteht aus mehreren Unterfenstern, die über Reiter zugänglich sind. Welche Unterfenster verfügbar sind und welche Eigenschaften angezeigt oder editiert werden können, hängt vom Typ des aktuell ausgewählten Objekts ab. Typische Objektfenster werden in Abb. 25 gezeigt. Ein Objektfenster, das bei allen Objekttypen vorhanden ist, zeigt die objekttypischen Einstellungen („*Settings*“) an und ermöglicht es Änderungen an diesen vorzunehmen. Im Fall von Modulen kann auf diese Weise die Modulausführung gesteuert werden. Einstellungen für Datensätze und Kartenansichten beeinflussen vor allem deren graphische Darstellung, aber z.B. im Fall von Rasterdaten auch die Speicherverwaltung. Ein weiteres Objektfenster, das immer angezeigt wird, enthält eine Beschreibung („*Description*“) des Objekts. Zu den Objektfenstern, die nur für bestimmte Objekte verfügbar sind, gehören Legenden und Attributtabellen.

3.4.2 Module

Die graphische Verwaltung von SAGA-Modulen erfolgt über das Modulfenster der Arbeitsumgebung. Die Befehle zum Laden und Schließen von Modulbibliotheken sowie die Befehle zum Ausführen der einzelnen Module können aber auch direkt über die Menüleiste erreicht werden. Alle geladenen Modulbibliotheken werden dem Wurzeleintrag „*Module*

Libraries“ der Baumstruktur des Modulfensters untergeordnet aufgelistet. Den Modulbibliotheken untergeordnet sind wiederum ihre jeweiligen Module. Das Laden einer Modulbibliothek erfolgt über einen Dateidialog, der entweder über die Menüleiste oder über das Kontextmenü des Wurzeleintrags aufgerufen wird. Wird der Befehl „*Close*“ für den Wurzeleintrag gewählt, dann werden alle geladenen Modulbibliotheken geschlossen. Der gleiche Befehl kann aber auch für einzelne Modulbliotheken aufgerufen werden. Die GUI merkt sich beim Beenden die zuletzt geladenen Bibliotheken und lädt diese automatisch beim nächsten Start wieder. Die einzige Einstellung, die zur Zeit für den Wurzeleintrag angeboten wird, legt fest, ob ein Modul nach seiner Ausführung ein akustisches Signal ausgibt. Für Modulbibliotheken gibt es keine Einstellungen. Ihre Beschreibung besteht u.a. aus ihrem Namen, den Autoren, dem Dateipfad der Bibliothek sowie einer Liste aller enthaltenen Module.

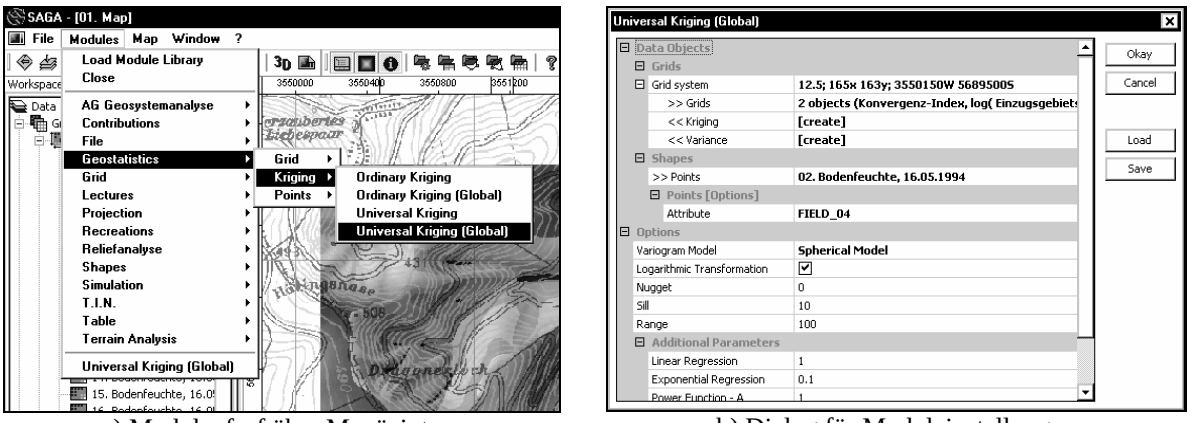
Der einzige Befehl, der auf ein Modul angewendet werden kann, betrifft seine Ausführung. Den Befehl dazu findet man im Kontextmenü oder über die Menüleiste (Abb. 26a). Der Ausführung eines Moduls kann aber auch über eine eigene Schaltfläche („*Execute*“) im Einstellungsfenster erfolgen. Über die Einstellungen werden Eingabe- und Ausgabedatensätze wie auch alle anderen modulspezifischen Parameter für die Ausführung festgelegt. Wird das Modul über einen Menüeintrag aufgerufen, so wird der Benutzer noch einmal von einem gesonderten Dialogfenster aufgefordert, die Einstellungen für die Modulparameter zu überprüfen und zu bestätigen (Abb. 26b). Wenn ein oder mehrere Modulparameter ungültig sind, z.B. wenn keine Auswahl für einen obligatorischen Eingabedatensatz getroffen wurde, wird eine Fehlermeldung ausgegeben und die Modulausführung wird abgebrochen. Ansonsten wird die eigentliche Ausführung des Moduls gestartet. Eine entsprechende Ausgabe erfolgt in den Benachrichtigungsfenstern. Während der Modulausgabe wird der Fortschritt der Ausführung durch einen Fortschrittsbalken dargestellt. Nach Ausführung wird eine Nachricht über den Erfolgsstatus ausgegeben. Während der Ausführung eines Moduls kann kein zweites Modul ausgeführt werden. Die Ausführung eines Moduls lässt sich aber mit der Abbruchtaste („*Escape*“) oder durch erneutem Aufruf des Modulbefehls abbrechen. Interaktive Module warten nach ihrem Aufruf auf Mauseingaben durch das Betätigen der Maustasten, während der Mauszeiger in einem Kartenfenster positioniert ist. Da Mauseingaben für unterschiedliche Aktionen benutzt werden können, z.B. für die Navigation im Kartenfenster oder für die Messung von Distanzen, muss der Mausmodus entsprechend gesetzt sein, damit Mauseingaben an das Modul weitergeleitet werden (s. Kap. 3.4.4). Um die interaktive Modulausführung zu beenden, den Modulbefehl erneut aufzurufen.

Die Aufgabe von Modulen ist es Eingabedatensätze zu analysieren und Ausgabedatensätze zu erstellen. Die hierfür nötige Auswahl von Datensätzen für die Eingabe und Ausgabe in den Moduleinstellungen erfolgt über einfach strukturierte Listen, in denen alle verfügbaren und kompatiblen Datensätze angezeigt werden. Damit ein Datensatz hier erscheinen kann, muss er vorher von einer Datei geladen oder von einem Modul erstellt worden sein, so dass er über die Datenverwaltung von SAGA zur Verfügung gestellt werden kann.

3.4.3 Daten

Im Vergleich zu Modulen und Karten bietet die Datenverwaltung, allein schon wegen der unterschiedlichen Einstellungsmöglichkeiten für die verschiedenen Datensatztypen, ein wesentlich größeres Spektrum an Eigenschaften an. Die graphische Verwaltung von Datensätzen erfolgt über das Datenfenster der Arbeitsumgebung. Wie bereits erwähnt wurde, können Module für ihre Ausführung nur auf Datensätze zugreifen, die hier auftauchen. Hier werden alle Datensätze verwaltet, die entweder von einer Datei geladen von einem Modul neu erstellt wurden. Um eine Datei zu laden kann ein Dateidialog über das Dateimenü der Menüleiste aufgerufen werden. Alternativ lassen sich Dateien auch mit der Maus in das Programm ziehen (*drag'n'drop*). Erfolgreich geladene Datensätze werden sortiert nach ihrem Datentyp in die Baumstruktur des Datenfensters eingegliedert. Zum Schließen oder Speichern stehen entsprechende Einträge im Kontextmenü des Datensatzes zu Verfügung. Soll ein neu erstellter oder ein modifizierter Datensatz geschlossen werden, wird nachgefragt ob und in welche Datei er gespeichert werden soll. Wenn mehrere Datensätze gleichzeitig geschlossen werden sollen, z.B. beim Beenden des Programms, erscheint ein einzelner Dialog, mit dem entsprechende Angaben in übersichtlicher Weise in einem Schritt für alle ungesicherten Dateien vorgenommen werden kann. Wenn keine Datensätze geladen sind, ist nur der Wurzeleintrag des Datenfensters sichtbar. Der Wurzeleintrag trägt die Bezeichnung „*Data*" und bietet allgemeine Einstellungen an, die zunächst vorgestellt werden sollen, bevor auf die Eigenschaften der verschiedenen Datensatztypen selbst eingegangen wird.

Die Beschreibung enthält eine kurze Zusammenfassung über die geladenen Datensätze. Mit der Einstellung „*Start Project*" lässt sich für den nächsten Aufruf von SAGA festlegen, ob das zuletzt geöffnete Projekt (s.u.), die zuletzt geöffneten Datensätze mitsamt ihren Einstellungen oder gar keine Datensätze geladen werden. Mit den „*Grid File Caching*" Optionen wird die automatische Verwendung des Dateipuffer-Mechanismus für Rasterdaten gesteuert. Dieser Mechanismus kann für bereits geladene Rasterdatensätze auch separat eingestellt werden. Um aber das Arbeiten mit großen Datensätzen, die nicht komplett in den Arbeitsspeicher geladen werden können, überhaupt zu ermöglichen, muss er bereits vor dem



a) Modulaufruf über Menüeintrag b) Dialog für Moduleinstellungen

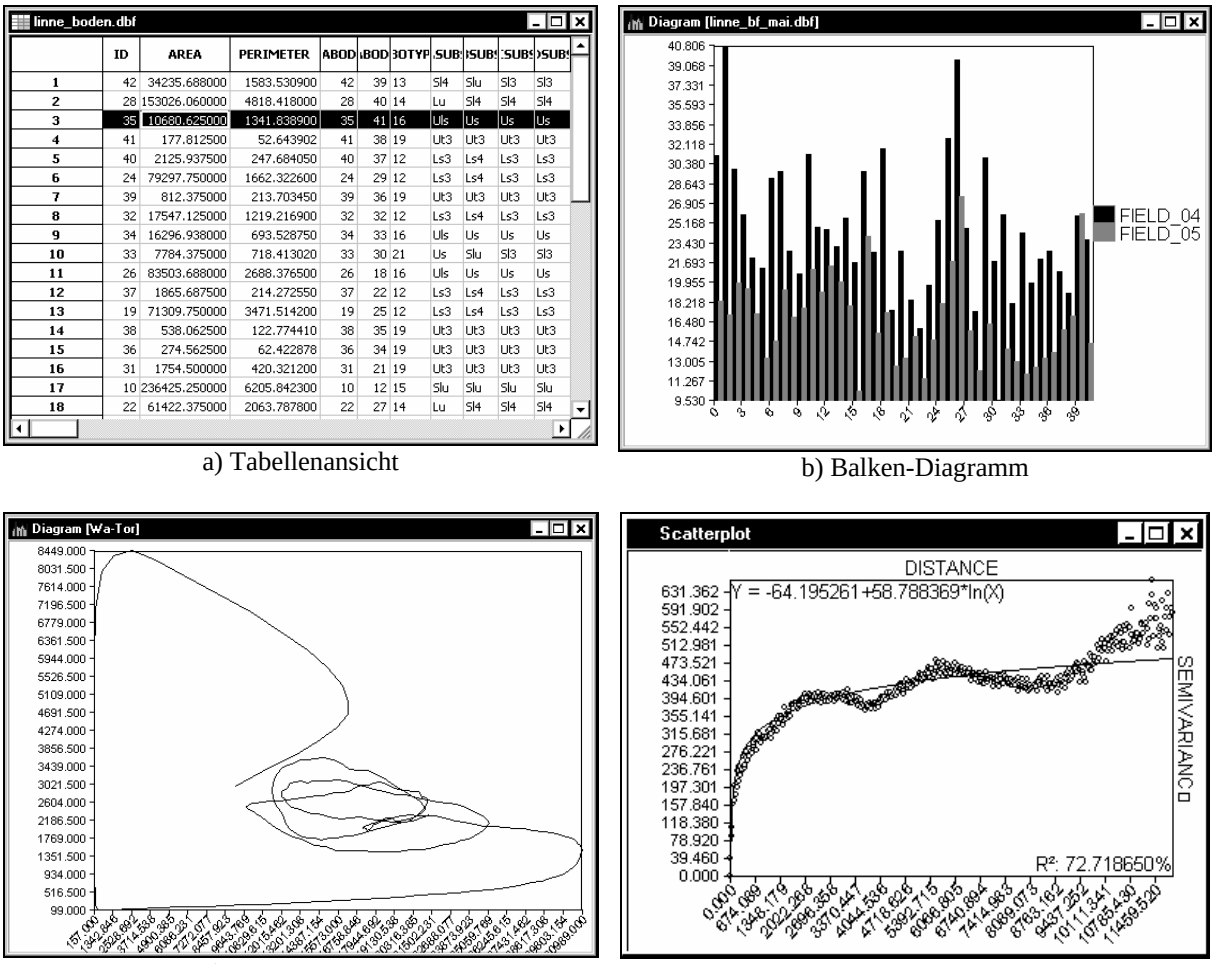
Abb. 26: Modulausführung

Öffnen gewählt werden können, was sich mit diesen Einstellungen automatisieren lässt. Bei der Verwendung des Dateipuffers werden nur die Zeilen eines Rasterdatensatzes in den Speicher geladen, auf die zuletzt zugegriffen wurde. Wie viele Zeilen gleichzeitig im Speicher gehalten werden, wird mit dem Parameter „*Threshold for automatic Mode*“ bestimmt, der, angegeben in Megabyte, einen Schwellenwert für den Arbeitsspeicher festlegt, der für einen Datensatz maximal bereit gestellt wird. Wenn „*Automatic Mode*“ angeschaltet ist, wird beim Laden von Rasterdaten überprüft, ob deren Dateigröße den Schwellenwert überschreitet, um gegebenenfalls einen Dateipuffer zu verwenden. Da Datensätze, die von SAGA neu erstellt werden, keiner Datei zugeordnet sind, wird für diese ein temporärer Dateipuffer verwendet, der in dem mit „*Temporary Files*“ ausgewählten Verzeichnis erstellt wird. Ist hier kein Verzeichnis angegeben, dann wird das Temporärverzeichnis des Betriebssystems benutzt.

Abgesehen von Tabellen, haben alle Datensatztypen, die von SAGA verwaltet werden, einen Raumbezug und teilen sich dadurch bedingt eine Reihe gemeinsamer Eigenschaften, die u.a. die Möglichkeit der direkten kartographischen Darstellbarkeit betreffen. Um einen räumlichen Datensatz in einem Kartenfenster darzustellen, genügt es, einen doppelten Mausklick auf seinen Datenfenstereintrag auszuführen oder seinen Kontextmenübefehl „*Show*“ aufzurufen. Wenn bereits ein oder mehrere Kartenfenster geöffnet sind, kann über eine Dialogbox angegeben werden, ob ein neues Kartenfenster erstellt oder ob der Datensatz einem bestehenden Kartenfenster hinzugefügt werden soll. Die Ausschnitt des Kartenfensters wird danach automatisch der räumlichen Ausdehnung des hinzugefügten Datensatzes angepasst. Für jeden räumlichen Datensatz gibt es ein Objektfenster mit einer Legendendarstellung, die die Farbklassifikation, gegebenenfalls auch die Größenklassifikation und die symbolische Darstellung, wiedergibt. Ein weiteres Fenster ermöglicht das Editieren von Attributdaten, das sich allerdings im Fall von Rasterdaten auf die Werte markierter Rasterzellen und im Fall von Vektordaten bzw. TIN auf die Attributdaten eines ausgewählten Elements bezieht. Eine Übersicht über Einstellungen, die für alle räumlichen Datensätze verfügbar sind, gibt Tab. 21. Auf die besonderen Einstellungen und Visualisierungsmöglichkeiten der verschiedenen Datensatztypen, auch die von Tabellen, gehen die nachfolgenden Unterkapitel ein. Zu den Einstellungen, die für alle räumlichen Datensätze verfügbar sind, gehören neben einem frei wählbaren Datensatznamen auch Angaben zur skalenabhängigen Darstellung und zum Typ der Farbklassifikation. Drei grundsätzliche Farbdarstellungen werden angeboten, wobei die einfachste die Verwendung eines einheitlichen Farbwertes für den gesamten Datensatz ist. Eine Nachschlagetabelle kann benutzt werden, um manuell Werteklassen zu definieren und ihnen einen Farbwert, einen Namen und eine Beschreibung zuzuweisen. Die dritte Möglichkeit ist die automatische Einteilung eines frei wählbaren Wertebereiches in kontinuierliche Werteklassen. Die Abstände zwischen den Klassengrenzen müssen dabei nicht gleich groß sein, sondern können auch logarithmisch skaliert werden.

Tab. 21: Einstellungen für räumliche Datensätze

Name	Typ	Beschreibung
General		Allgemeine Einstellungen
Name	Zeichenkette	Frei wählbarer Datensatzname, z.B. zur Legendenbeschriftung
Show Legend	Wahrheitswert	Schaltet die Anzeige der Legende in Kartenansichten an oder aus
Display Visibility		Festlegung der skalenabhängigen Anzeige bzw. Sichtbarkeit des Datensatzes
Show Always	Wahrheitswert	Wenn wahr, wird der Datensatz immer angezeigt. Ansonsten wird er ausgeblendet, wenn der Kartenmaßstab außerhalb des Wertebereichs von „*Scale Dependent*“ liegt.
Scale Dependent	Wertebereich	Gibt den Maßstabsbereich an, in dem der Datensatz angezeigt wird
Display: Color Classification		Einstellungen für die Farbklassifikation
Type	Auswahl	Art der datenwertabhängigen Farbklassifizierung. Für alle Datensatztypen verfügbar sind: - Unique Symbol einheitliches Symbol - Lookup Table Nachschlagetabelle - Graduated Color kontinuierliche Werteklassen (Rasterdatensätze bieten zwei weitere Möglichkeiten)
Unique Symbol		Einheitliche Farbe für alle Datenwerte
Color	Farbwert	Farbe für die Datensatzdarstellung
Lookup Table		Nachschlagetabelle bestimmt Farbe für Datenwerte
Table	Tabelle	Jede Farbklasse wird durch einen Record in der Nachschlagetabelle definiert. Diese hat 5 Spalten: - COLOR: der Farbwert - NAME: Klassenname - DESCRIPTION: Klassenbeschreibung - MINIMUM: Untergrenze der Klasse - MAXIMUM: Obergrenze der Klasse
Graduated Color		Kontinuierliche Werteklassen
Colors	Farbpalette	Definiert Anzahl und Farben der Werteklassen
Value Range	Wertebereich	Wertebereich, über den die Werteklassen verteilt werden
Mode	Auswahl	Steuert die Größen der einzelnen Werteklassen - Linear Gleich große Werteklassen - Logarithmic (up) Logarithmische Zunahme der Werteklassengröße - Logarithmic (down) Logarithmische Abnahme der Werteklassengröße
Logarithmic Stretch Factor	Fließkommazahl	Bestimmt die Stärke der Größenzunahme bzw. -Abnahme für logarithmische Klasseneinteilungen



a) Tabellenansicht

b) Balken-Diagramm

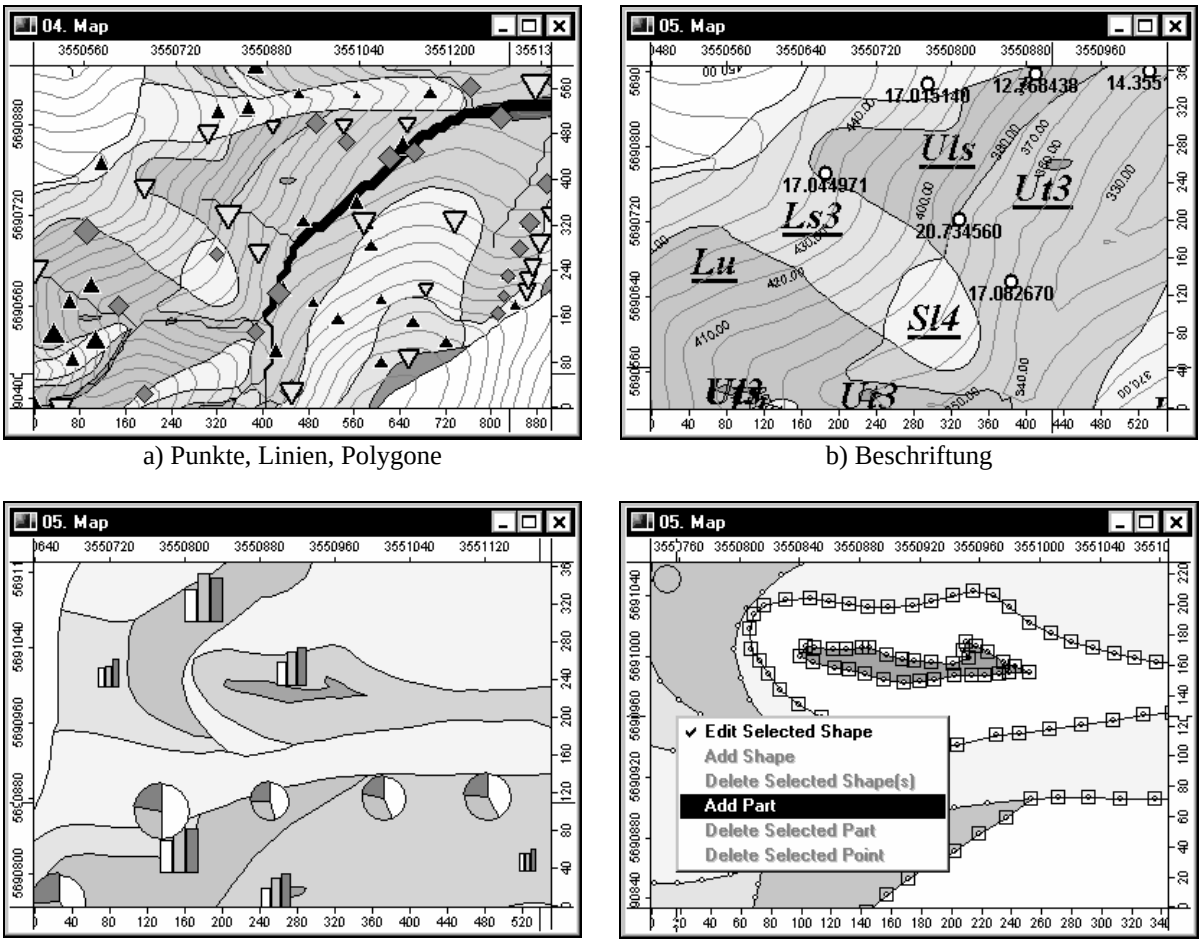
c) X/Y-Diagramm

d) Scatterplot mit Regression

Abb. 27: Ansichten von Tabellendaten

Tabellen

Tabellen können als einziger Datensatztyp nicht direkt kartographisch dargestellt werden, da sie keinen unmittelbaren Raumbezug haben und diesen nur durch Verknüpfung mit räumlichen Datensätzen erlangen können. Dieser Raumbezug ist z.B. für die Attributtabellen von Vektordatensätzen gegeben, bei denen jede Tabellenzeile (*Record*) einem Raumelement des Vektordatensatzes zugeordnet ist. Die Funktionalitäten für solche Attributtabellen sind im wesentlichen die gleichen wie die für unabhängige Tabellen. Tabellen besitzen in der aktuellen SAGA-Version keine eigenen Einstellungen. Es werden jedoch mehrere Möglichkeiten angeboten, die Tabellendaten zu visualisieren. Die Standardansicht für Tabellendaten ist das Tabellenfenster (Abb. 27a), in der die Tabellenwerte in Reihen und Spalten angeordnet sind und editiert werden können. Wenn ein Tabellenfenster angezeigt wird und aktiviert ist, bietet der Eintrag „*Table*“ in der Menüleiste eine Befehlsauswahl für das Arbeiten mit der Tabellenansicht an. Alternativ können diese auch über das Kontextmenü des Tabellenobjekts erreicht werden. Über diese Befehle können z.B. Datenspalten (oder Felder) eingefügt oder gelöscht werden. Für Datenreihen ist das in gleicher Weise möglich, sofern es sich nicht um die Attributtabelle eines Vektordatensatzes oder TIN handelt. In diesem Fall erfolgt das



a) Punkte, Linien, Polygone

b) Beschriftung

c) Balken- u. Kreis-Diagramme

d) Digitalisieren u. Editieren

Abb. 28: Ansichten von Vektordaten

Einfügen bzw. Löschen von Datenreihen über die entsprechenden Funktionen des die Tabelle besitzenden räumlichen Datensatzes. Die Tabellenreihen lassen sich nach ihren Datenwerten sortieren. Die gewählte Sortierreihenfolge beeinflusst auch die Darstellung in der Diagrammansicht, die ebenso wie die Darstellung von Scatterplots ebenfalls für abhängige Attributtabellen aufgerufen werden kann. Für die Diagrammdarstellung stehen eine Reihe von Optionen zur Verfügung. Es kann z.B. ausgewählt werden, welche Datenspalten angezeigt werden und durch welchen Farbwert sie repräsentiert werden. Die Ausgabe kann in Form von Balken (Abb. 27b), Linien oder Punkten erfolgen. Für die Beschriftung, aber auch für die Skalierung der X-Achse kann eine Datenspalte gewählt werden (Abb. 27c). In den Ansichten mit Scatterplots oder Scattergrammen wird der Merkmalsraum, der von zwei Datenspalten aufgespannt wird, in einem Diagramm dargestellt, so dass sich ein Bild der Abhängigkeitsbeziehung ihrer Werte ergibt (Abb. 27d). Zusätzlich wird eine Regressionsanalyse, mit wahlweiser Anpassung an verschiedene Funktionstypen, durchgeführt. Mit Scatterplots können auch Rasterdatensätze untereinander als auch mit Vektordaten verglichen werden.

Vektordaten

Ein Vektordatensatz setzt sich aus geometrischen Objekten („*Shapes*“) vom selben Typ, entweder Punkt, Multipunkt, Linie oder Polygon, zusammen. Im Datenverwaltungsfenster

werden die geladenen Vektordatensätze nach ihrem Typ gegliedert angezeigt. Jedes Objekt eines Vektordatensatzes verfügt über die gleichen Attributmerkmale. Die Attributdaten für diese Merkmale werden in einer Tabelle gespeichert, die im Datenverwaltungsfenster dem Vektordatensatz hierarchisch untergeordnet ist. Die Attributtabelle lässt sich mit den im vorigen Kapitel vorgestellten Funktionalitäten darstellen und bearbeiten.

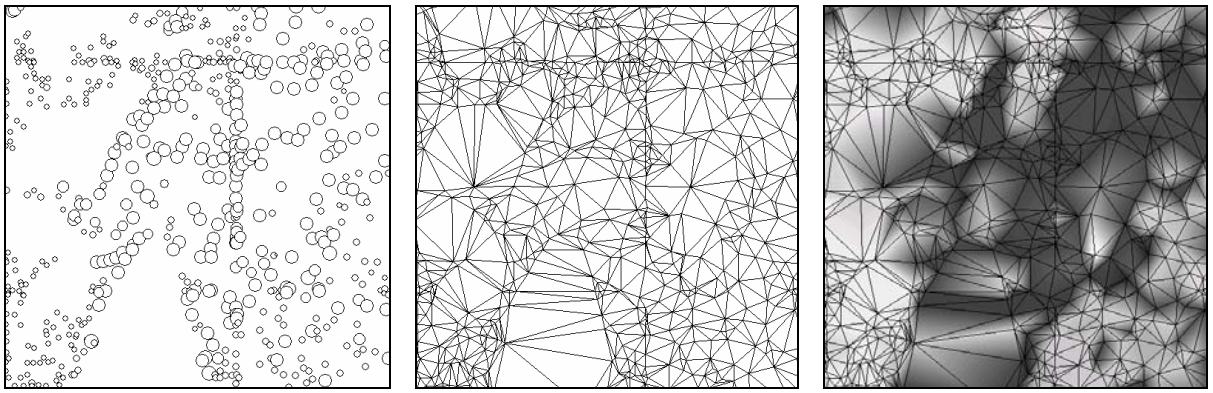
Neben den bereits vorgestellten allgemeinen Einstellungen für räumliche Datensätze, gibt es für Vektordaten eine Reihe weiterer Einstellungen, die zum Teil auch wieder nur für spezielle Vektordatentypen vorhanden sind. Die meisten Einstellungen beziehen sich auf die kartographische Darstellung. So muss, anders als bei Rasterdaten, für die graphische Klassifikation von Vektordaten immer ein Attribut ausgewählt werden, dessen Werte für die Klassifikation herangezogen werden sollen. Neben der für alle Vektortypen verfügbaren Farbklassifikation, können Punkte und Linien auch durch verschiedene Größenklassen dargestellt werden. Punkte werden durch verschiedene Punktsymbole dargestellt (Abb. 28a). Neben Standardsymbolen, wie Kreise oder Dreiecke, können auch beliebige Graphikdateien für die Symboldarstellung verwendet werden. Alle Vektortypen lassen sich mit einer Beschriftung durch eines ihrer Attribute versehen (Abb. 28b). Kreis- und Balkendiagramme werden über einen speziellen Dialog definiert, in dem Angaben zu den darzustellenden Attributen, aber auch zur Größenskalierung gemacht werden können (Abb. 28c). Zwei weitere Einstellungen dienen zur Festlegung von Regeln für das Editieren von Vektordatensätzen. Im Detail handelt es sich hierbei um die Auswahl von Vektordatensätzen, deren Stützpunkte und Liniensegmente zum Einfangen (engl. *snapping*) der editierten Punkte benutzt werden sollen, sowie der maximalen Distanz, bis zu der ein editierter Punkte eingefangen werden soll. Um die Elemente eines Vektordatensatzes zu editieren oder ihm neue Elemente hinzuzufügen, müssen drei Bedingungen erfüllt sein:

- der Datensatz muss in der Arbeitsumgebung als aktuelles Objekt ausgewählt sein
- der Datensatz muss in einem Kartenfenster dargestellt werden
- der Mausmodus des Kartenfensters muss auf „*Action*“ gesetzt sein (s.a Kap.3.4.4)

Sind die Bedingungen erfüllt, so kann mit einem rechten Mausklick in das Kartenfenster ein Kontextmenü mit Befehlen für das Editieren aufgerufen werden. Wenn kein Element des Datensatzes markiert wurde, dann steht nur ein Befehl zum Hinzufügen eines neuen Elements zur Auswahl („*Add Shape*“). Nach dem Aufruf des Befehls können solange Punkte für das neue Elemente mit der linken Maustaste gesetzt werden, bis die rechte Maustaste betätigt wird. Danach besteht die Möglichkeit die gesetzten Punkte normal zu editieren. Um ein vorhandenes Element zu editieren, muss dieses zuerst mit der linken Maustaste ausgewählt werden. Danach stehen im Kontextmenü zwei weitere Befehle zur Auswahl, einer für das Löschen („*Delete Selected Shape(s)*“) und einer für das Editieren („*Edit Selected Shape*“) des markierten Elements.

Tab. 22: Einstellungen für Vektordaten

Name	**Typ**	**Beschreibung**
Display		Allgemeine Darstellung
Chart	Parameterliste	Zeigt einen Dialog, mit dem die Balken- bzw. Kreisdiagrammdarstellung der Attributdaten eingeschaltet und gesteuert wird
Display: Color Classification		Einstellungen für die Farbklassifikation
Attribute	Auswahl	Attributdaten, deren Werte für die Farbklassifikation benutzt werden sollen
Edit		Regeln für das Editieren der Vektordaten
Snap Distance	Fließkommazahl	Maximale Distanz, bis zu der Punkte eingefangen werden
Snap to...	Liste	Hier werden Vektordatensätze ausgewählt, deren Stützpunkte beim Editieren zum Einfangen neuer Punkte benutzt werden
Nur für Punkte		
Display		Allgemeine Darstellung
Symbol Type	Auswahl	Symbol, das für die Darstellung der Punkte verwendet wird. Neben einfachen Symbolen, wie Kreisen, Dreiecken, Quadraten und Rhomben, kann auch eine Bilddatei gewählt werden.
Symbol Image	Datei	Bilddatei für die Darstellung als Punktsymbol
Nur für Punkte und Linien		
Display Size		Größe von Punktsymbolen bzs. Breite von Linien
Attribute	Auswahl	Attribut, dessen Werte für die Größenklassifikation benutzt werden. Wenn nicht gesetzt, wird „*Default Size*“ für alle Vektorelemente benutzt
Size relates to...	Auswahl	Bezugssystem für die Größendarstellung - Screen Bildschirmpixel - Map Units Kartenmaßstab
Default Size	Fließkommazahl	Standardgröße, wenn kein Attribut für die Größenklassifikation gewählt wurde
Size Range	Wertebereich	Minimale und maximale Größe der Größenklassen
Nur für Punkte und Polygone		
Display		Allgemeine Darstellung
Fill Style	Auswahl	Legt fest, wie Polygone bzw. Punktsymbole ausgefüllt werden. Neben der undurchsichtigen (opaken) und durchsichtigen (transparenten) Darstellung sind verschiedene Schraffuren möglich
Outline	Wahrheitswert	Bestimmt, ob Polygone bzw. Punktsymbole mit einer verschieden farbigen Umrandung versehen werden
Outline Color	Farbwert	Farbe der Umrandung
Outline Size	Ganzzahl	Linienbreite der Umrandung
Nur für Linien und Polygone		
Display		Allgemeine Darstellung
Show Points	Wahrheitswert	Anzeige der Stützpunkte



a) Punktdaten b) Dreiecksvermaschung c) Farbinterpolation

Abb. 29: Ansichten von TIN

Wenn ein Element zum Editieren ausgewählt wurde, werden alle seine Stützstellen angezeigt (Abb. 28d). Bei Punktdatensätzen ist immer nur ein Stützpunkt vorhanden. Die Stützpunkte können mit der Maus an neue Positionen gezogen werden, wobei sie gegebenenfalls von anderen Stützpunkten eingefangen werden können. Bei Linien und Polygonen können mit der linken Maustaste neue Stützstellen in ihren Liniensegmenten eingefügt werden. Stützpunkte können mit der Maus markiert werden, um sie mit dem Kontextmenübefehl „*Delete Point*“ oder der „*Entfernen*“-Taste zu löschen. Polygone wie auch Linien können aus mehreren Teilelementen bestehen, die mit dem Menübefehl „*Add Part*“ hinzugefügt oder, wenn ein Stützpunkt des Teilelements markiert ist, mit „*Delete Part*“ gelöscht werden. Das Editieren eines Elements wird durch erneutem Aufruf des Menübefehls „*Edit Selected Shape*“ beendet, worauf der Benutzer gefragt wird, ob er die Änderungen übernehmen oder verwerfen möchte. Die Attributdaten des selektierten Elements lassen sich direkt in dem Attributfenster der Objekteigenschaften bearbeiten.

TIN

Die Eigenschaften von TIN decken sich zum Teil mit denen der zuvor diskutierten Vektordatensätze. So besitzen TIN eine Attributtabelle, die sich allerdings derzeit nur auf die Punkte, nicht jedoch auf die Kanten und Dreiecksflächen, bezieht. Auch ist eine räumliche Editierfunktion für TIN zur Zeit nicht implementiert. Unterschiedlich sind vor allem die Möglichkeiten der Visualisierung. Zunächst lassen sich die Punkte, Kanten und Dreiecksflächen wahlweise anzeigen oder ausblenden (Abb. 29). Die weiteren Einstellungen betreffen die Darstellung der Flächen, für die z.B. ein Wert für ihre Transparenz gesetzt werden kann. Abgesehen von einer einheitlichen Farbgebung, kann auch eine Farbklassifizierung auf Basis einer Nachschlagetabelle oder kontinuierlicher Werteklassen vorgenommen werden. Für die Klassifizierung muss ein Attribut gewählt werden. Da sich die Attributwerte auf die Punkte des TIN beziehen, erfolgt die flächenhafte Darstellung der Attributwerte durch lineare Interpolation zwischen den Punkten jeder Dreiecksfläche (Abb. 29c).

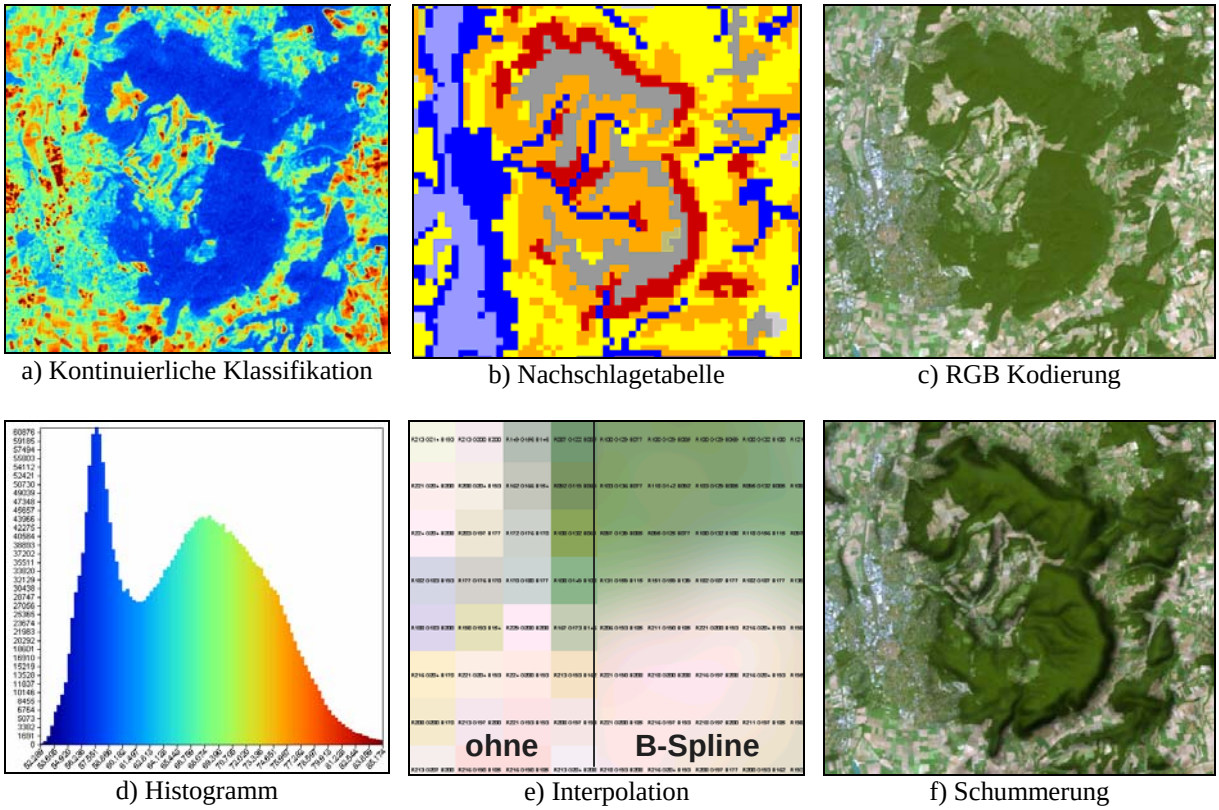
Tab. 23: Einstellungen für TIN

Name	Typ	Beschreibung
Display		Allgemeine Darstellung
Show Points	Wahrheitswert	Zeigt die Punkte des TIN an oder blendet sie aus
Show Edges	Wahrheitswert	Zeigt die Kanten des TIN an oder blendet sie aus
Show Filled	Wahrheitswert	Zeigt die Flächen des TIN an oder blendet sie aus
Transparency [%]	Fließkommazahl	Transparenzwert zwischen 0 und 100%. Ermöglicht die transparente Darstellung der Dreiecksflächen
Display: Color Classification		Einstellungen für die Farbklassifikation
Graduated Color		Kontinuierliche Werteklassen
Attribute	Auswahl	Auswahl der Attributdaten für die Farbklassifikation

Rasterdaten

Rasterdatensätze („*Grids*“) werden im Datenverwaltungsfenster nach ihren räumlichen Eigenschaften in Rastersysteme („*Grid System*“) untergliedert, die durch die Anzahl ihrer Spalten und Reihen, der Distanz zwischen den Rasterzellen sowie der geographischen Position eindeutig festgelegt sind. Bei Rasterdatensätzen, die dem gleichen Rastersystem angehören, lassen sich zellenweise Verknüpfungen sehr einfach durchführen, weshalb viele Module ihre Berechnungen jeweils auf die Daten eines Rastersystems beschränken.

Auch für Rasterdaten gibt es eine Reihe spezieller Eigenschaften. Neben ihrer kartographischen Darstellung, können ihre Datenwerte in Histogrammen, die die Häufigkeit der auftre-



a) Kontinuierliche Klassifikation b) Nachschlagetabelle c) RGB Kodierung

d) Histogramm e) Interpolation f) Schummerung

Abb. 30: Ansichten von Rasterdaten

tenden Datenwerte für jede Farbklasse anzeigen (Abb. 30d), und Scatterplots, die die Datenwerte mit anderen Raster- oder Vektordatensätzen in Beziehung setzen (s.o.), visualisiert werden. Der Aufruf dieser alternativen Datenansichten erfolgt in der Arbeitsumgebung über das Kontextmenü des jeweiligen Rasterdatensatzes. Nicht alle Einstellungen dienen ausschließlich der kartographischen Darstellung. Von ihrer Struktur her decken Rasterdaten immer ein rechteckiges Gebiet ab. Da aber nicht immer für alle Rasterzellen Werte vorliegen müssen, kann zur Markierung solcher Rasterzellen ein No-Data Wert bzw. Wertebereich bestimmt werden. Standardmäßig liegt dieser Wertebereich zwischen *–99999* und *–999*. Rasterzellen, die einem No-Data Wert entsprechen, werden in den Kartenansichten nicht dargestellt. Aber auch die Mehrzahl der Module berücksichtigt die gesetzten No-Data Werte und ignoriert diese Rasterzellen in ihren Berechnungen. U.a. für die Legendendarstellung kann eine Einheit für die Datenwerte angegeben werden. Ein Multiplikatorwert erlaubt es darüber hinaus, die Datenwerte für die Darstellung auf einen anderen Wertebereich zu skalieren, z.B. um sich Winkel, die im Bogenmaß gespeichert sind, in Grad ausgeben zu lassen. Einstellungen zur Speicherverwaltung kommen der Arbeit mit großen Rasterdatensätzen entgegen (s.a. Kap. 3.4.3). Statt den gesamten Rasterdatensatz im Arbeitsspeicher zu halten, kann dieser in einer Datei gepuffert werden. Für Rasterdatensätze, deren Datenwerte nur selten von Rasterzelle zur Rasterzelle wechseln, ist die Laufzeitlängenkomprimierung eine gute Alternative zur Verwendung eines Dateipuffers. Zu den speziellen Einstellungen für die kartographische Darstellung gehört das Setzen eines Wertes für die Transparenz, so dass unterlegte Datenschichten durchscheinen. Es kann zwischen verschiedenen Methoden für die graphische Interpolation der Datenoberfläche gewählt werden, so dass die Übergänge zwischen den einzelnen Rasterzellen fließend erscheinen (Abb. 30e). Werden die Rasterzellen in einer Karte ausreichend groß angezeigt, dann werden die einzelnen Rasterzellen mit ihren Datenwerten beschriftet. Die Beschriftung lässt sich wahlweise ausschalten oder in Größe und Erscheinungsbild anpassen. Neben der Darstellung durch einen einheitlichen Farbwert und der Farbklassifikation auf Basis einer Nachschlagetabelle (Abb. 30b) bzw. kontinuierlicher Werteklassen (Abb. 30a), stehen für Rasterdaten zwei weitere Methoden für die Farbdarstellung zur Wahl. So lassen sich die Datenwerte, insbesondere von Bilddateien, als RGB-codierte Echtfarbwerte interpretieren (Abb. 30c) oder die Datenwerte werden benutzt, um unterlegte Datenschichten zu schummern (Abb. 30f).

Die Datenwerte eines Rasterdatensatzes lassen sich auch direkt im Attributfenster seiner Objekteigenschaften editieren. Wie beim Editieren von Vektordaten, muss zunächst dafür gesorgt werden, dass

- der Datensatz in der Arbeitsumgebung als aktuelles Objekt ausgewählt ist
- der Datensatz in einem Kartenfenster dargestellt wird
- der Mausmodus des Kartenfensters auf „*Action*“ gesetzt ist (Kap.3.4.4)

Tab. 24: Einstellungen für Rasterdaten

Name	Typ	Beschreibung
General		Allgemeine Einstellungen
Unit	Zeichenkette	Einheit der Datenwerte
Z-Factor	Fließkommazahl	Multiplikator für die Darstellung der Kartenwerte
No Data	Wertebereich	Wert oder Wertebereich, der fehlende Datenwerte kennzeichnet
Display		Allgemeine Darstellung
Transparency [%]	Fließkommazahl	Transparenzwert zwischen 0 und 100%. Ermöglicht das Durchscheinen lassen anderer Datensätze
Interpolation	Auswahl	Interpolationsmethode für die Darstellung der Datenoberfläche - None keine Interpolation - Bilinear bilineare Interpolation - Inverse Distance distanzgewichtete Interpolation - Bicubic Spline alternative Spline Interpolation - B-Spline Basis Spline Interpolation
Display: Color Classification		Einstellungen für die Farbklassifikation
Type	Auswahl	Zusätzliche Farbklassifikationen für Rasterdaten - RGB interpretiert Datenwerte als RGB-codierte Echtfarben - Shade nutzt die Datenwerte zum Schummern unterlegter Datenschichten
Display: Cell Values		Darstellung der Rasterzellenwerte
Show	Wahrheitswert	Gibt an, ob die Rasterzellenwerte angezeigt werden
Font	Schrift	Schriftart für die Darstellung der Werte
Size	Fließkommazahl	Schrifthöhe in Prozent, bezogen auf die Größe einer Rasterzelle
Decimals	Ganzzahl	Zahl der Nachkommastellen, die ausgegeben wird
Memory		Einstellungen zur Speicherverwaltung
Memory Handling	Auswahl	Methode der Speicherverwaltung. Unterstützt das Arbeiten mit großen Datensätzen - Normal Alle Datenwerte befinden sich unkomprimiert im Arbeitspeicher - RTL Compression Laufzeitlängekomprimierung - File Cache Dateipuffer
Buffer Size [MB]	Fließkommazahl	Legt bei Verwendung von Dateipuffer/Komprimierung fest, welche Datenmenge, ausgedrückt in Megabyte, tatsächlich im Arbeitsspeicher vorgehalten wird

Danach können mit der Maus einzelne oder Blöcke von Rasterzellen im Kartenfenster ausgewählt werden, deren Werte als Tabelle im Attributfenster dargestellt werden und sich editieren lassen. Änderungen werden erst nach der Bestätigung durch den Benutzer übernommen.

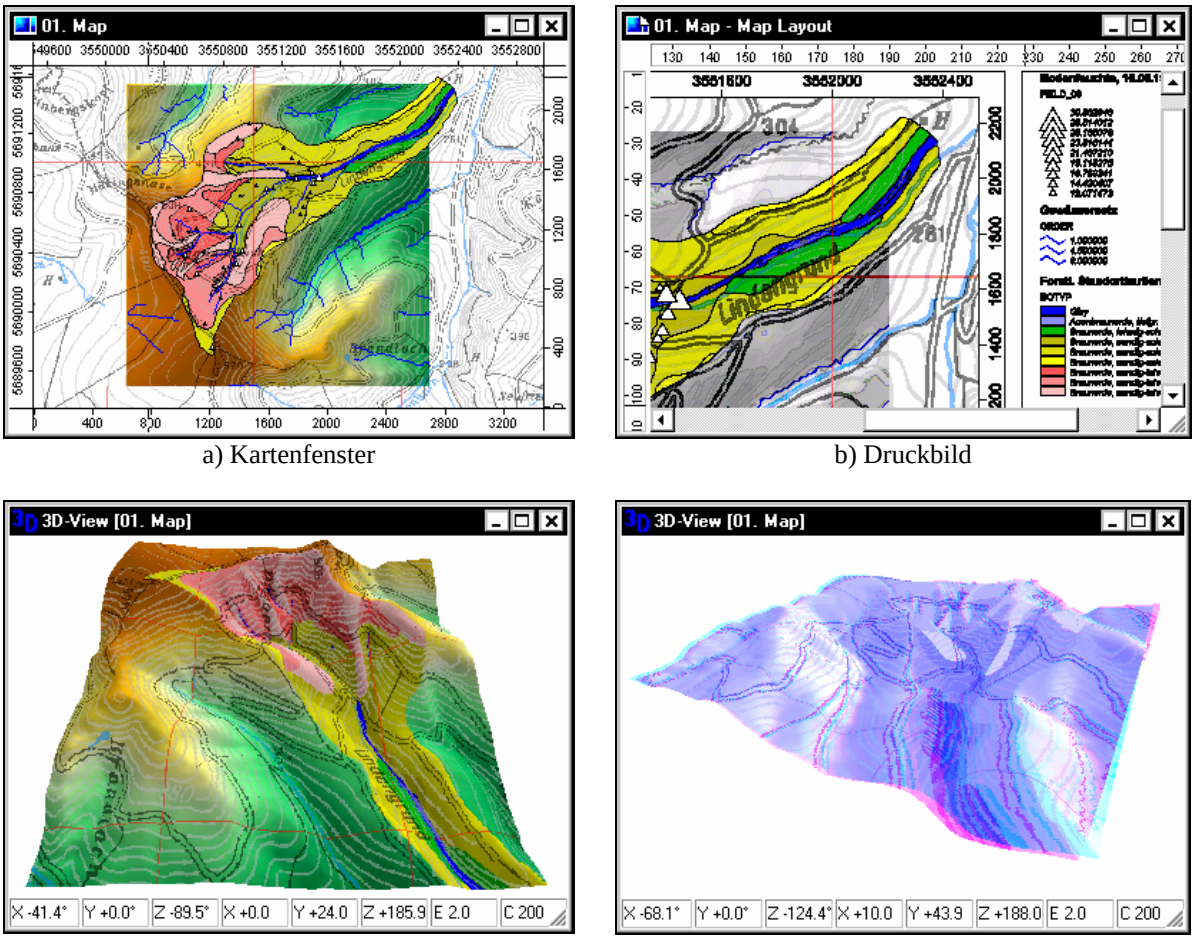
Projekte

Bei der alltäglichen Arbeit wird man in der Regel mit immer wiederkehrenden Zusammenstellungen von Datensätzen arbeiten. Auch wird man Zusammenstellungen, die man vor längerer Zeit bearbeitet hat, oft zu späteren Zeitpunkten wieder aufrufen müssen. Zwar lassen sich mit der SAGA-GUI mehrere unterschiedliche Datensätze gleichzeitig öffnen. Sind diese jedoch auf mehrere Verzeichnisse verteilt, oder benötigt man nur einen Teil der in einem Verzeichnis versammelten Datensätze, so wird man diese mühsam heraussuchen müssen. Schließlich werden beim Laden oder Erstellen neuer Datensätze deren Einstellungen immer auf die gleichen Standardwerte gesetzt, die jedoch selten den gewünschten Darstellungseigenschaften entsprechen. Das gilt insbesondere auch für einmal erstellte aussagekräftige Kartenansichten, die sich vielleicht sogar aus in einer bestimmten Reihenfolge überlagerten Datenschichten zusammensetzen. Um dieser Problematik entgegen zu kommen, bietet die SAGA-GUI die Möglichkeit an, Projektdateien zu erstellen, die jederzeit wieder geöffnet werden können.

In den Projektdateien werden alle geladenen Datensätze mit ihrem Dateipfad gespeichert. Für jeden Datensatz werden außerdem seine kompletten Einstellungen festgehalten. Alle Karten, die erstellt wurden, werden mit Angabe der in ihnen überlagerten Datensätze ebenfalls in der Projektdatei festgehalten. Um eine neue Projektdatei mit den aktuellen Eigenschaften – geladene Datensätze, Datensatzeinstellungen, Karten – zu speichern, genügt es den Befehl „*Save Project As...*“ im Dateimenü der Menüleiste aufzurufen und in dem darauf erscheinenden Dateidialog einen Dateinamen einzugeben. Danach erfolgte Änderungen können auch direkt mit dem Befehl „*Save Project*“ in die soeben erstellte Projektdatei geschrieben werden. Geöffnet werden erstellte Projektdateien mit dem Befehl „*Load Project*“, der sich ebenfalls im Dateimenü befindet. Wenn bereits Datensätze geladen sind, fragt die GUI nach, ob diese vor dem Laden des Projekts geschlossen werden sollen. Wird das verneint, dann werden die Datensätze des Projekts zusätzlich geladen. Auf diese Weise lassen sich auch mehrere Projekte miteinander kombinieren. Datensätze, die während des Ladens nicht gefunden werden, z.B. weil sie zwischenzeitlich gelöscht wurden, werden unter Ausgabe einer Fehlermeldung ignoriert. Beim Speichern eines Projektes wird der Benutzer gefragt, ob Datensätze, deren Datenwerte modifiziert wurden oder denen noch kein Dateiname zugewiesen wurde, weil sie neu erstellt wurden, gespeichert werden sollen. Die Entscheidung kann für jeden Datensatz einzeln getroffen werden. Datensätze, die keiner Datei zugeordnet sind, werden logischerweise nicht in die Projektdatei aufgenommen.

3.4.4 Karten

Die kartographische Darstellung ist die Standardmethode für die Visualisierung von Geodaten. Wenn ein räumlicher Datensatz in einer Karte dargestellt werden soll, kann der Benutzer zunächst angeben, ob er einer bestehenden Kartenansicht hinzugefügt oder ob für ihn eine neue Kartenansicht angelegt werden soll. Ist noch keine Kartenansicht verfügbar, wird automatisch eine neue Karte erstellt und ein Fenster für diese geöffnet. Das Schließen



a) Kartenfenster
b) Druckbild
c) 3D Ansicht
d) 3D Stereo Anaglyph

Abb. 31: Ansichten für Karten

eines Kartenfensters löscht eine Karte jedoch nicht aus der Arbeitsumgebung. Hierzu muss auf die Kartenverwaltung zurückgegriffen werden, über die auch eine Reihe weiterer Karteneigenschaften angesprochen werden können. Die Kartenverwaltung erfolgt über das entsprechend benannte Unterfenster „*Maps*“ der Arbeitsumgebung (Abb. 25c). Hier kann das Fenster einer Kartenansicht sowohl wieder ein- als auch ausgeblendet werden, und nur hier kann eine erstellte Karte wieder gelöscht werden, z.B. durch Aufruf des Kontextmenübefehls „*Close*“.

Der Aufbau einer Kartenansicht selbst ist sehr einfach gehalten. Er besteht im wesentlichen aus einer Liste von Datensätzen, die in der Reihenfolge in der Kartenansicht dargestellt werden, in der sie unter dem Kartenobjekt im Verwaltungsfenster aufgelistet sind. Die Änderung der Reihenfolge aber auch das Entfernen eines Datensatzes aus der Liste erfolgt am einfachsten über die Befehle des Kontextmenüs, das in der Kartenverwaltung den Datensätzen zugeordnet ist. Das Kontextmenü, das man hier für einen Datensatz aufrufen kann, unterscheidet sich von dem, das in der Dateiverwaltung der Arbeitsumgebung erscheint, und enthält u.a. Befehle zum Ändern der Darstellungsreihenfolge in der Kartenansicht. Die Objekteigenschaften der Datensätze sind die gleichen, die auch in der Datenverwaltung zur Verfügung stehen, so dass sich ihre Darstellungseigenschaften auch bequem über die Karten-

verwaltung erreichen lassen. Im Übrigen können Datensätze auch gleichzeitig in mehreren Karten angezeigt werden.

Zu den Objekteigenschaften von Karten gehört eine Legende, in der alle Datensätze aufgeführt sind, die in der Kartenansicht angezeigt werden und deren Legendendarstellung nicht deaktiviert wurde. Die Einstellungen für Kartenansichten beschränken sich zur Zeit auf die Darstellung des Kartenrahmens.

Jedes Kartenfenster besitzt einen Kartenrahmen, in dem sowohl Koordinaten als auch ein Maßstab dargestellt sind. Die aktuelle Mausposition wird im Rahmen durch Striche markiert. Wie alle Fenster für Datenansichten (Tabellen, Diagramme, Scatterplots und Histogramme), haben Karten ein eigenes Menü in der Menüleiste, über das die spezifischen Befehle für Karten gefunden werden können. Die Menübefehle werden durch entsprechende Schaltflächen in der Werkzeugleiste ergänzt. Eine Gruppe dieser Befehle kontrolliert die Verwendung der Maus und schaltet zu diesem Zweck zwischen verschienen Modi der Maussteuerung um. Im Standardmodus wird die Maus benutzt, um den Kartenausschnitt zu vergrößern oder zu verkleinern („*Zoom*“). Zwei weitere Modi erlauben das Verschieben des aktuellen Kartenausschnitts („*Pan*“) und das Messen von Distanzen („*Measure Distance*“). Der vierte, mit „*Action*“ bezeichnete Modus, muss aktiviert werden, um kartenbezogene Aktionen vorzunehmen. Mögliche Aktionen sind entweder das Selektieren und Editieren von Datensatzelementen oder die Ausführung eines interaktiven Moduls. Der aktuelle Kartenausschnitt kann auch durch spezielle Befehle gesetzt werden:

- Ausschnitt auf den vorherigen Ausschnitt zurücksetzen
- Ausschnitt an die Gesamtausdehnung aller Datensätze der Karte anpassen
- Ausschnitt an die Ausdehnung des aktiven Datensatzes anpassen
- Ausschnitt an das(die) selektierte(n) Datensatzelement(e) anpassen
- Benutzerdefinierten Ausschnitt über Dialogeingabe setzen
- Ausschnitt aller Kartenansichten mit dem des aktuellen Fensters synchronisieren

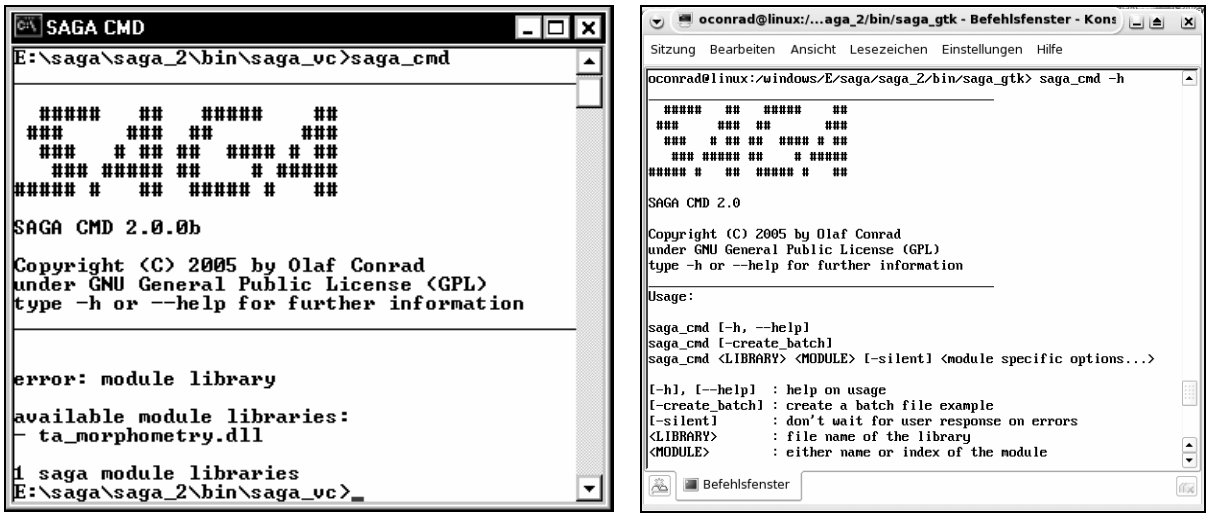
Drei Dateiformate werden unterstützt um eine Kartenansicht direkt zu speichern. Die einfachste Möglichkeit ist das Speichern in einem gängigen Graphikdateiformat, wobei die Bildauflösung frei gewählt werden kann. Die Legende wird wahlweise in einer separaten Datei abgelegt. Für die Erstellung eines PDF-Dokuments kann zusätzlich ein Vektordatensatz angegeben werden, für dessen Elemente jeweils eine eigene Karte erstellt wird. Die Ausgabe im SVG-Format erstellt eine interaktive Karte, die sich von vielen aktuellen Webbrowsern öffnen lässt (z.B. *Mozilla Firefox* ab Version 1.5, http://www.mozilla.org).

Zwei weitere Befehle öffnen Fenster mit speziellen Darstellungen für die Kartenansicht. Das erste dieser Fenster dient der Darstellung und Gestaltung des Druckbildes (Abb. 31b). Es bezieht sich auf das über die Druckeigenschaften eingestellte Papierformat und bietet zur Zeit nur einfache Funktionen zum Ändern des Layouts, wie die Positionierung der Legende und

der Berechnung bzw. Anpassung des Kartenmaßstabs. Das zweite Fenster ermöglicht die dreidimensionale Darstellung der Karte (Abb. 31c). Vorraussetzung für die 3D-Darstellung ist ein Rasterdatensatz mit Höhenwerten, der das Gebiet der Karte abdeckt. Für 3D-Ansichten gibt es zahlreiche Visualisierungsoptionen, wie die Art der Projektion oder die Stärke der Überhöhung. Der Blickwinkel und die Distanz können sehr einfach durch Ziehen mit der Maus bei gedrückter Maustaste geändert werden. Die linke Maustaste dient dabei zum Drehen, die rechte zum Verschieben. Animierte Bewegungsabläufe, sogenannte *Fly Throughs*, können sehr schnell durch Hinzufügen der aktuellen Blickposition zu einer Liste von Positionen erfolgen, die anschließend unter Verwendung einer frei wählbaren Anzahl von Interpolationsschritten wiedergegeben werden kann. Die Darstellung kann auch als Stereo-Anaglyph erfolgen, so dass sich beim Betrachten durch eine Brille mit rotem bzw. grünem Farbfilter für das linke bzw. rechte Auge ein dreidimensionaler Eindruck ergibt (Abb. 31d).

3.5 Kommandozeileninterpreter

Mit dem Kommandozeileninterpreter, nachfolgend *CMD* genannt, bietet SAGA eine alternative Benutzeroberfläche zur GUI an. Anders als die GUI, verfügt der CMD über keine Möglichkeiten für die Datenverwaltung und -visualisierung. Der CMD ist ausschließlich dafür gedacht, Module auszuführen. Wie die Name bereits andeutet, wird der CMD durch Tastatureingabe von Befehlszeilen z.B. in einem Kommandozeilenfenster (Abb. 32) ausgeführt. Ein solcher Befehl setzt sich aus dem Programmnamen „*saga_cmd*“, dem Namen und eventuell dem Dateipfad einer Modulbibliothek, dem Namen oder der Identifikationsnummer des auszuführenden Moduls und einer Reihe von weiteren modulspezifischen Parametern zusammen. Es ist offensichtlich, dass die Benutzung der GUI zur Ausführung eines Moduls wesentlich einfacher ist und im Normalfall von jedem Anwender bevorzugt werden wird. Welche Vorteile hat aber der CMD zu bieten, um seine Existenz zu rechtfertigen? Zum einen ist dies die Möglichkeit, Skriptdateien zu schreiben, so dass komplexe oder häufig durchgeführte Arbeitsabläufe automatisiert und ohne großen Aufwand auf verschiedene Datenprojekte angewendet werden können. Zum anderen ermöglicht der CMD die Ausführung von Modulen aus anderen Anwendungen heraus, z.B. einem Webserver. Im folgenden wird die allgemeine Verwendung des CMD vorgestellt und ein Beispiel für eine Stapeldatei (engl. *batch file*) gegeben. Bezogen wird sich dabei auf die Kommandozeilenfunktion von Microsoft Betriebssystemen, deren Befehlssatz auch unter Windows weitgehend dem von *Microsoft DOS* (von engl. *disc operating system*) angelehnt ist. Die Verwendung einer sogenannten *Shell* (deutsch *Schale*) unter Linux Betriebssystemen ist aber wesentlichen gleich. Hilfe zum Aufrufen und zur Benutzung eines Kommandozeilenfensters bietet die jeweilige Betriebssystemdokumentation.



a) Windows Kommandozeilenfenster b) Bash Shell unter Linux

Abb. 32: SAGA Kommandozeileninterpreter

Nach dem Öffnen eines Kommandozeilenfensters können in diesem Befehle eingegeben und durch Betätigen der Eingabetaste (engl. *enter* oder *return*) ausgeführt werden. Programme werden durch Eingabe des Programmnamens aufgerufen, wobei die Dateinamenserweiterung, die in der Regel „*exe*“ (von engl. *executable*) lautet, nicht eingegeben werden muss. Der Befehlsinterpreter sucht daraufhin das angegebene Programm zunächst im aktuellen Verzeichnis. Wird es dort nicht gefunden, dann werden alle Verzeichnisse durchsucht, die in der Umgebungsvariablen *PATH* gesetzt sind. Damit das CMD-Programm unabhängig davon gefunden werden kann, in welchem Verzeichnis ein Kommando ausgeführt wird, bietet es sich also an, die Umgebungsvariable *PATH* um das Verzeichnis der SAGA-Installation zu erweitern. In den nachfolgenden Beispielen wird sich auf das Installationsverzeichnis „*C:\SAGA*“ bezogen. Das Setzen der *PATH* Variablen erfolgt durch die Befehlszeile:

```
C:\>PATH=%PATH%;C:\SAGA
C:\>_
```

Das aktuelle Verzeichnis, in dem sich das Kommandozeilenfenster befindet, wird meist automatisch vor dem Befehlseingabefeld angezeigt, das hier dem Zeichen „>“ folgt. Mit dem Befehl *cd* (von engl. *change directory*) wird das aktuelle Verzeichnis gewechselt. Nach Eingabe von

```
C:\>cd SAGA
C:\SAGA>_
```

ist das aktuelle Verzeichnis das der SAGA Installation. Von hier aus kann der CMD auch ohne vorheriges Setzen der *PATH* Variablen direkt aufgerufen werden. Dem CMD können bei seinem Aufruf eine Reihe von sogenannten Programm- oder Kommandozeilenoptionen übergeben werden, durch die sich seine Ausführung steuern lässt. Die Optionen werden dem Programmnamen durch Leerzeichen getrennt nachgestellt. Die Option *-h* veranlasst den CMD dazu, einen Hilfetext in das Kommandofenster auszugeben:

```
C:\SAGA>saga_cmd -h
_____________________________________________
  #####   ##   #####    ##
 ###     ###  ##       ###
  ###   # ## ##  #### # ##
   ### ##### ##    # #####
##### #   ##  ##### #   ##

SAGA CMD 2.0

Copyright (C) 2005 by Olaf Conrad
under GNU General Public License (GPL)
type -h or --help for further information
_____________________________________________
Usage:

saga_cmd [-h, --help]
saga_cmd [-create_batch]
saga_cmd <LIBRARY> <MODULE> [-silent] <module specific options...>

-h, --help      : help on usage
[-create_batch] : create a batch file example
<LIBRARY>       : file name of the library
<MODULE>        : either name or index of the module
[-silent]       : don't wait for user response on errors

example:
  saga_cmd Terrain_Analysis_Morphometry
           "Local Morphometry"
           -ELEVATION c:\dem.sgrd
           -SLOPE     d:\slope.sgrd
           -ASPECT    d:\aspect.sgrd
           -METHOD    1

_____________________________________________
Module libraries are expected to be located in
the directory, that is specified by the environment
variable 'SAGA_MLB'. If this is not found the
current working directory will be searched for instead.

C:\SAGA>_
```

Der Hilfetext beschreibt die wichtigsten Kommandozeilenoptionen und gibt ein Beispiel für die Ausführung eines Moduls. Ein wichtiger Hinweis betrifft den Suchpfad für die Modulbibliotheken. Damit Modulbibliotheken von beliebigen Verzeichnissen heraus aufgerufen werden können, kann man die Umgebungsvariable *SAGA_MLB* so definieren, dass sie, ähnlich wie die zuvor vorgestellte *PATH*-Variable, auf das Verzeichnis mit den Modulbibliotheken verweist. *SAGA_MLB* wird, anders als die *PATH*-Variable, nur von dem SAGA-CMD verwendet und muss, sofern nicht bereits geschehen, neu angelegt werden. Die Neudefinition einer Umgebungsvariablen erfolgt mit dem DOS-Befehl *SET*:

```
C:\SAGA>SET SAGA_MLB = C:\SAGA\Modules
C:\SAGA>_
```

Das Setzen von Umgebungsvariablen kann unter Windows-Betriebssystemen alternativ auch über die Systemsteuerung vorgenommen werden (s. Windows-Hilfefunktion). Ein Vorteil hiervon ist, dass die Umgebungsvariablen bereits beim Öffnen eines Kommandozeilenfensters gesetzt sind und nicht jedes Mal neu angepasst werden müssen. Nachdem die Umgebungsvariable *SAGA_MLB* gesetzt ist, listet der CMD, wenn er ohne zusätzliche Optionen aufgerufen wird, alle Modulbibliotheken auf, die in dem Verzeichnis „*C:\SAGA\Modules*“ liegen. Ist *SAGA_MLB* nicht definiert, wird stattdessen das aktuelle Verzeichnis nach gültigen Modulbibliotheken durchsucht.

```
C:\SAGA>saga_cmd

...

error: module library

available module libraries:
- shapes_grid.dll
- ta_morphometry.dll
- geostatistics_kriging.dll

...

C:\SAGA>_
```

Für die Ausführung eines Moduls muss als erste Programmoption der Name der Bibliothek, die das Modul zur Verfügung stellt, angegeben werden. Werden keine weiteren Optionen übergeben, dann listet der CMD alle in der Bibliothek enthaltenen Module auf, in diesem Beispiel für die Modulbilbliothek **ta_morphometry**:

```
C:\SAGA>saga_cmd ta_morphometry

...

error: module

available modules:
 0      - Convergence Index
 1      - Convergence Index (Search Radius)
 2      - Curvature Classification
 3      - Hypsometry
 4      - Local Morphometry
 5      - Real Area Calculation
 6      - Morphometric Protection Index

C:\SAGA>_
```

Dabei werden alle Module mit ihrem Modulnamen und einer bei 0 beginnend, fortlaufend durchnummerierten Identifikationsnummer ausgegeben. Um ein Modul für die Ausführung auszuwählen, wird nach dem Bibliotheksnamen entweder der Modulname oder seine Identifikationsnummer angegeben. Wenn der Modulname Leerzeichen enthält, muss dieser von Anführungszeichen eingeschlossen sein. Der Aufruf des Moduls *Convergence Index* erfolgt also mit

```
C:\SAGA>saga_cmd ta_morphometry “Convergence Index“
```

oder mit

```
C:\SAGA>saga_cmd ta_morphometry 0
```

und liefert, wenn keine weiteren Optionen angegeben wurden, die folgende Ausgabe:

```
...

_________________________________________________
library path:   C:\SAGA\Modules
library name:   ta_morphometry.dll
module name:    Convergence Index
_________________________________________________
go...
Usage: 0 -ELEVATION <str> -RESULT <str> [-METHOD <num>]
  -ELEVATION:<str>      Elevation
        [Grid]
  -RESULT:<str>         Convergence Index
        [Grid]
  -METHOD:<num>         Method
        [Choice]
        Choices:
        [0] Aspect
        [1] Gradient

The value for the option 'ELEVATION' must be specified.

error: executing module [Convergence Index]

C:\SAGA>_
```

Da das Modul obligatorische Eingabe- und Ausgabeparameter besitzt, kann es ohne Angaben für diese nicht ausgeführt werden. Der CMD gibt eine entsprechende Fehlermeldung, eine Zeile mit der Aufrufkonvention („*Usage:*“) sowie eine Liste aller Modulparameter aus. Jeder Modulparameter wird mit seiner Kommandozeilenoption, seinem Namen und Datentyp angegeben. Für einige Parametertypen folgen Zusatzinformationen, wie im Beispiel für den Parameter vom Typ *Choice*. Optionale Parameter sind in der Zeile mit der Aufrufkonvention von eckigen Klammern ([ ]) eingeschlossen. Das Setzen der Modulparameter erfolgt allgemein durch Übergabe der Kommandozeilenoption mit dem darauf folgenden, durch ein Leerzeichen getrennten Parameterwert. Der CMD besitzt keine besonderen Fähigkeiten zur Verwaltung von Datensätzen und arbeitet ausschließlich dateibasiert. Parameter vom Typ *Table*, *Shapes*, *TIN* und *Grid* müssen daher durch einen Dateipfad und Dateinamen spezifiziert werden. Handelt es sich um Eingabedatensätze, so müssen die so bezeichneten Dateien in einem von der SAGA-API unterstützten Dateiformat vorliegen (s. Kap. 3.3.4). Ausgabedatensätze werden ebenfalls nur in diesen Formaten abgelegt. Alle anderen Parameterwerte werden je nach Typ entweder als Zahlen, Zeichen oder Zeichenketten ihrer Kommandozeilenoption nachgestellt. Im folgenden Beispiel wird das Modul *Convergence Index* mit dem Eingabedatensatz „*C:\hoehe.sgrd*“ ausgeführt. Für die Berechnung des Ergebnisses, das in die Datei „*C:\konvergenz.sgrd*“ geschrieben werden soll, wurde die Methode „*Gradient*“ gewählt.

```
C:\SAGA>saga_cmd ta_morphometry 0 -ELEVATION C:\hohe.sgrd -RESULT C:\konvergenz.sgrd -METHOD 1

...

go...

Load grid: C:\hohe.sgrd...
100%
okay

Parameters:
[Grid system] Grid system: 14.25; 1167x 1027y; 561735W 5703705S
[Grid] Elevation: hohe.sgrd
[Grid] Convergence Index: new
[Choice] Method: Gradient
100%

Save grid: C:\konvergenz.sgrd...
100%
okay

C:\SAGA>_
```

Während der Ausführung gibt das Modul Information über den Ausführungsfortschritt und die verwendeten Modulparameter an. Nach der Ausführung kann der nächste Befehl in der Kommandozeile eingegeben werden.

Nachdem nun gezeigt wurde, wie der CMD prinzipiell zum Ausführen von Modulen benutzt wird, soll ein Blick darauf geworfen werden, wie eine einfache Stapeldatei zur Automatisierung eines Arbeitsablaufs aussehen kann, der aus mehreren nacheinander geschalteten Modulen besteht. Der Befehl *@ECHO OFF* in der ersten Zeile des Skripts unterdrückt die Bildschirmausgabe der nachfolgenden Befehlszeilen, so dass nur die Modulausgaben auf dem Bildschirm erscheinen. Danach werden für den Fall, dass dies noch nicht geschehen ist, die Umgebungsvariablen *SAGA_MLB* und *PATH* gesetzt. Dadurch können die folgenden CMD Aufrufe von beliebigen Verzeichnissen ausgeführt werden. Über die Funktionsweise und die Bedeutung der einzelnen Parameter der Module informiert man sich am einfachsten durch die in der GUI verfügbare Modulbeschreibung.Es werden zunächst einige Module zur Berechnung von Reliefparametern aufgerufen, deren Ein- und Ausgaben Rasterdatensätze sind. Eingabedatensatz für die ersten drei Module ist ein mit *-ELEVATION* angegebener Rasterdatensatz mit Höhenwerten. Das vierte Modul benutzt zwei der zuvor erstellten Datensätze als Eingabe. Mit dem Modul *Contour Lines* der Bibliothek **shapes_grid** wird ein Vektordatensatz mit Isolinien einer Äquidistanz von 2.5, gemessen in der Einheit der Rasterdaten, erstellt. Eingabedatensatz ist das zuvor berechnete Raster „*slope.sgrd*“ mit Hangneigungswerten. Das zuletzt aufgerufene Modul führt eine Clusteranalyse für eine beliebige Anzahl von Rasterdatensätzen durch. Diese werden dem Eingabeparameter *-INPUT* als Liste von Dateinamen übergeben, die durch Semikola getrennt sind. Neben dem Rasterdatensatz „*cluster.sgrd*“ mit Angaben zur Clusterzugehörigkeit der Rasterzellen, wird auch eine Texttabelle „*cluster.txt*“ mit den statistischen Eigenschaften der Clusterklassen erstellt. Insgesamt leitet das Skript also von einem einzigen Eingabedatensatz, einem Raster mit Höhendaten, zwölf neue Datensätze ab, zehn Rasterdatensätze, einen Vektordatensatz sowie eine Tabelle.

```
@ECHO OFF

SET SAGA_MLB = C:\SAGA\Modules
SET PATH = %PATH%;C:\SAGA

saga_cmd ta_lighting         0   -ELEVATION .\dem.sgrd -SHADE shade.sgrd -METHOD 0 -AZIMUTH -
45
                     -DECLINATION 45

saga_cmd ta_morphometry      4   -ELEVATION .\dem.sgrd -SLOPE .\slope.sgrd -ASPECT
.\aspect.sgrd
                     -HCURV .\hcurv.sgrd -VCURV .\vcurv.sgrd

saga_cmd ta_hydrology        0   -ELEVATION .\dem.sgrd -CAREA .\carea.sgrd

saga_cmd ta_hydrology       14   -SLOPE .\slope.sgrd -AREA .\carea.sgrd -WETNESS
.\wetness.sgrd
                     -STREAMPOW .\streampow.sgrd -LSFACTOR .\lsfactor.sgrd

saga_cmd shapes_grid         5   -INPUT .\slope.sgrd -CONTOUR .\slope.shp -ZSTEP 2.5

saga_cmd grid_discretisation 1   -INPUT .\dem.sgrd;.\slope.sgrd;.\hcurv.sgrd;.\vcurv.sgrd
                     -RESULT .\cluster.sgrd -STATISTICS .\cluster.txt -NCLUSTER 10

PAUSE
```

Eine Stapeldatei, wie das hier abgebildete Skript, muss unter Windows die Dateiendung „*bat*“ besitzen und kann auch direkt aus einem Windows-Explorer ausgeführt werden. Ein größeres Maß an Flexibilität erreicht man durch die Verwendung von Variablen für die Dateinamen, die dem CMD bei seinem Aufruf durch das Skript übergeben werden. Es soll abschließend noch erwähnt werden, dass die Verwendung des CMD nicht auf Stapeldateien beschränkt ist. Prinzipiell kann jede Skript- bzw. Programmiersprache dazu benutzt werden den CMD auszuführen, von der aus es möglich ist, externe Programme mit der Übergabe von Programmoptionen aufzurufen.

3.6 Module

Das modulare Konzept der SAGA-Architektur sieht vor, dass Methoden in Form von Modulen bereitgestellt werden, die wiederum in Modulbibliotheken zusammengefasst sind. Die SAGA-API ist besonders darauf ausgerichtet, die Modulprogrammierung zu unterstützen, so dass Methoden für die Prozessierung von Geodaten sehr einfach in ein SAGA-Modul übersetzt werden können (s. Kap. 3.3 u. 3.7). Als Ergebnis verfügt SAGA über eine umfangreiche und schnell wachsende Methodensammlung. Die quelltextoffenen Module der aktuellen SAGA-Distribution, die in diesem Kapitel vorgestellt werden, umfassen derzeit 236 Module, die sich auf 44 Bibliotheken verteilen. Die Aufteilung der Module auf die Bibliotheken erfolgt dabei weitgehend nach thematischen Aspekten. Abgedeckt werden neben Standardfunktionen, wie Datenimport und Koordinatentransformation, vor allem die Forschungsschwerpunkte der Arbeitsgruppe Geosystemanalyse (s. Kap. 1.2). Ein Großteil der Module wurde vom Autor dieser Arbeit programmiert. Doch die Zahl der von anderen Entwicklern beigesteuerten Module nimmt seit der Freigabe der SAGA-Quelltexte stetig zu. In den tabellarischen Übersichten, die nachfolgend jeder thematischen Kategorie vorangestellt sind, werden für diese Module ihre Urheber namentlich erwähnt. Da das Lizenzmodell von SAGA nicht verlangt, dass Module mit Quelltexten veröffentlicht werden müssen (s. Kap. 3.2), ist die Gesamtzahl entwickelter SAGA-Module vermutlich um einiges höher. So hat die AG Geosystemanalyse Module im Rahmen von Forschungsprojekten entwickelt, die zumin-

dest kurzfristig nicht unter eine OSS-Lizenz gestellt werden, u.a. um ihre kommerzielle Verwertung zu ermöglichen. Die eng mit der AG Geosystemanalyse zusammen arbeitende SciLands GmbH (s. Kap. 1.2) hat für die firmeninterne Verwendung bereits weit über 100 Module entwickelt (nach mündlicher Auskunft). Es gibt auch andere Gründe, die Freigabe von Modulen zumindest zu verzögern. So kann es z.B. sinnvoll sein, Module, die neuesten Forschungsansätzen entsprungen sind, zurückzuhalten, damit die korrekte Anerkennung ihrer wissenschaftlichen Bedeutung nicht durch ihre falsche Verwendung gestört wird (z.B. HECKMANN & BECHT 2006 und WICHMANN 2006). In diesem Sinne ist der folgende Überblick nur als Momentaufnahme der beständig wachsenden methodischen Fähigkeiten von SAGA zu sehen.

3.6.1 Import und Export

Die Verwaltung und der Austausch von Daten erfolgt im allgemeinen dateibasiert, wobei die verwendeten Dateiformate vor allem durch die Software bestimmt werden, die für die Datenerstellung benutzt wurde. SAGA selbst unterstützt direkt nur jeweils ein Dateiformat für Tabellen, Vektor- und Rasterdaten (s. Kap.3.3.4). Das Einlesen stellt somit eine erste Hürde bei der Datenverarbeitung dar. Auf der anderen Seite ist es oft notwendig, neu erstellte Datensätze mit verschiedenen weiteren Programmen öffnen zu müssen, so dass auch hierfür entsprechende Dateischnittstellen benötigt werden. Die erste thematische Kategorie, die hier vorgestellt wird, befasst sich daher mit Modulbibliotheken, die Schnittstellen für verbreitete Dateiformate bereitstellen. Der wichtigste Modulparameter ist daher im allgemeinen der Dateiname der zu lesenden bzw. zu schreibenden Datei. Module für den Dateiimport legen neue Datensätze in der SAGA-Umgebung an, Module für den Datenexport erzeugen aus geladenen Datensätzen Dateien im gewünschten Format.

Rasterdaten (allgemein)

Die Bibliothek **io_grid** enthält acht Module für das Lesen und drei für das Schreiben von Rasterdaten. Auf die Formate *Arc/Info Grid* (ESRI 2006) und *Surfer Grid* (GOLDEN SOFT 2006) kann sowohl lesend als auch schreibend zugegriffen werden. Beide Formate unterstützen binären wie auch die textbasierten Datenzugriff. Das Modul „*Import Binary Raw Data*“ ist nicht auf ein spezielles Dateiformat zugeschnitten, sondern ermöglicht das flexible Einlesen beliebiger binärer Rasterdateien, sofern die Daten unkomprimiert sind. Damit die Daten eingelesen werden können, müssen Angaben gemacht werden zur Speichereinheit der Datenwerte sowie der Anzahl der Reihen und Spalten. Weitere Modulparameter betreffen die Georeferenzierung und die Startposition des Datenbereichs innerhalb der Datei. Rasterdaten, die in einer Texttabelle gespeichert sind, können direkt mit „*Import Grid from Table*“ geöffnet werden. Das Modul „*Export Grid to XYZ*“ speichert jeden Rasterzellenwert zusammen mit seinen Koordinaten als Zeile in einer Tabelle. Das Rasterformat *Erdas LAN/GIS*, das vor allem in der Satellitenbildverarbeitung gebräuchlich ist und auch mehrere Rasterdatensätze in einer Datei enthalten kann, wird nur für den Lesezugriff unterstützt (LEICA GEOSYSTEMS 2006).

Tab. 25: Module für den Import u. Export von Datensätzen

Name/Bibliothek	Beschreibung	Module
Grids **io_grid**	Verschiedene Import und Export Filter für Rasterdatenformate	• Export ESRI Arc/Info Grid • Import ESRI Arc/Info Grid • Export Surfer Grid • Import Surfer Grid • Import Binary Raw Data • Import Grid from Table • Import Erdas LAN/GIS • Import USGS SRTM Grid • Import SRTM30 DEM • Import MOLA Grid (MEGDR) • Export Grid to XYZ • Export True Color Bitmap
Images **io_grid_image**	Import und Export Filter für Bilddateien	• Export Image (bmp, jpg, png) • Import Image (bmp, jpg, png, tif, gif, pnm, xpm)
Grids using GDAL **io_grid_gdal**	Import von Rasterdaten unter Verwendung der Geospatial Data Abstraction Library (GDAL)	• Import using GDAL (various raster formats)[AR]
Shapes **io_shapes**	Verschiedene Import und Export Filter für Vektordatenformate	• Export GStat Shapes • Import GStat Shapes • Export Shapes to XYZ • Import Shapes from XYZ • Export Shapes to Generate • Export Surfer Blanking File • Import Surfer Blanking File • Export Atlas Boundary File • Import Atlas Boundary File
GPS Data **io_gps**	Module für den Import gängiger GPS Dateiformate	• GPX to shapefile[VO] • GPSBabel[VO]
Tables using ODBC **io_table_odbc**	Datenbankzugriff via ODBC	• Import Table via ODBC
ESRI E00 **io_esri_e00**	Import Filter für E00 Dateien der Firma ESRI	• Import ESRI E00 File

[VO]V.Olaya, [AR]A.Ringeler

Ebenfalls nur lesend kann auf Dateien in den Formaten der frei verfügbaren Datensätze der Shuttle Radar Topography Mission (SRTM) mit Höhendaten der Erdoberfläche (FARR & KOBRICK 2000) und des Mars Orbiter Laser Altimeter (MOLA) der Mars Global Surveyor (MGS) Mission, mit dem ein Höhenmodell des Mars erstellt wurde (NASA 2006), zugegriffen werden. „*Export True Color Bitmap*“ schreibt farbcodierte Rasterdaten als Graphikdatei im

Windows Bitmap Format. Dieses Modul ist für große Datensätze effizienter in der Ausführung als das entsprechende Modul der folgenden Bibliothek.

Bilddateien

Die beiden Module der Bibliothek **io_grid_image** benutzen Funktionen der wxWidgets-Bibliothek (s. Kap. 3.1.1) für den Zugriff auf verschiedene Graphikdateiformate. Unterstützt werden u.a. die Formate *Windows Bitmap* (BMP), *Joint Photographic Expert Group* (JPEG), *Tagged Image File Format* (TIFF) und *Portable Network Graphics* (PNG). Beim Schreiben wird eine zusätzliche Textdatei mit Angaben zur Georeferenzierung des Datensatzes angelegt, die beim Laden, sofern vorhanden, automatisch geöffnet und zur Herstellung eines Raumbezugs für den Datensatz benutzt wird. Da Graphikdateien Farbwerte speichern, müssen für den Export die Werte der Rasterzellen in Farben übersetzt werden. Das geschieht, ähnlich wie bei der Farbklassifikation durch die GUI (Kap. 3.4.3), durch Auswahl einer Farbpalette. Beim Import wird die Graphikdatei je nach Benutzerauswahl und gegebener Farbtiefe entweder als ein einziges, RGB-codiertes Raster geladen oder getrennt in drei Raster mit den Intensitäten des roten, grünen und blauen Kanals.

GDAL

Die Geospatial Data Abstraction Library (GDAL, WARMERDAM et al. 2006) ist eine C++ Klassenbibliothek, die den Zugriff auf etwa 40 verschiedene Rasterdateiformate ermöglicht (s.a. Kap. 3.1.1). Die GDAL ist ein Bestandteil vieler Linux Distributionen und wird u.a. von GRASS GIS verwendet (GRASS 2006). Mit dem einzigen Modul der Bibliothek **io_grid_gdal**, steht auch unter SAGA eine GDAL-basierte Schnittstelle für den Import von Rasterdaten zur Verfügung.

Vektordaten

Die Bibliothek **io_shapes** unterstützt zwei programmspezifische Schnittstellen für Vektordaten. Eine einfache Schnittstelle erlaubt den Datenaustausch mit dem geostatistischen Analyseprogramm *Gstat* (PEBESMA & WESSELING 1998). Der Austausch von Vektordaten mit dem Programm Surfer wird durch das *Blanking* Dateiformat ermöglicht (GOLDEN SOFTWARE 2006). Etwas weiter verbreitet ist das *Atlas Boundary File* Format (RETAIL PROFIT MANAGEMENT 2006). Nur für Punktdaten geeignet sind die Module „*Import/Export XYZ*“, die auf der Basis von Texttabellen arbeiten und jeden Punkt mit seinen Koordinaten und Attributen speichern bzw. laden.

GPS

Die beiden Module der Bibliothek **io_gps** dienen zum Import von Punktkoordinaten, die von einem Gerät zum Auswerten von GPS-Daten (Global Positioning System) erzeugt werden. Da es eine Vielzahl von GPS-Geräten mit unterschiedlicher Software gibt, gibt es auch eine entsprechend hohe Zahl von Dateiformaten für GPS-Daten. Das GPS Exchange Format (GPX) ist ein verbreitetes Dateiformat für den Austausch von GPS-Daten, das von

dem Modul „*GPX to shapefile*“ unter Verwendung des quelltextoffenen Programms *gps2shp* (HIRAOKA 2006) gelesen und in einen Vektordatensatz konvertiert werden kann. Eine große Spannweite von GPS-Formaten kann von dem ebenfalls unter einer OSS-Lizenz veröffentlichten Programm *GPS Babel* gelesen werden. Die Funktionen beider Programme sind nicht direkt in die gleichnamigen Module eingebunden. Stattdessen rufen die Module das jeweilige Programm unter Angabe des Installationsverzeichnisses auf, um den Import von GPS-Daten durchzuführen.

Tabellen (ODBC)

Die Bibliothek **io_table_odbc** besitzt zur Zeit einen experimentellen Status und soll vor allem aufzeigen, wie eine Datenbank von einem SAGA-Modul heraus unter Verwendung der Open Database Connection (ODBC) Schnittstelle angesprochen werden kann. Das einzige Modul der Bibliothek etabliert eine Verbindung zu einer über ODBC frei gegebenen Datenbank. Die Verbindung erfolgt unter den Sicherheitseinstellungen der gewählten Datenbankverbindung, so dass gegebenenfalls ein Benutzername und ein Passwort angegeben werden muss. Konnte eine Verbindung erfolgreich hergestellt werden, so wird eine Liste mit allen Tabellen der Datenbank erstellt, aus der der Benutzer eine Tabelle für den Import auswählen kann. Nachdem die Tabelle importiert wurde, wird die Verbindung wieder geschlossen. ODBC Schnittstellen gibt es für nahezu alle modernen Datenbank Management Systeme (DBMS) und lassen sich mittlerweile auch unter Linux problemlos installieren. ODBC ermöglicht den Datenbankzugriff unter Verwendung der Structured Query Language (SQL) und somit den Zugriff auf beliebige SQL-Datenbanken.

E00

Das *ARC/INFO Interchange File* Format, gekennzeichnet durch die Dateiendung „*E00*“, speichert ganze Datenprojekte, die aus mehreren Vektor- und Rasterdatensätzen sowie Tabellen bestehen können (ESRI 2006). Das einzige Modul der Bibliothek **io_e00** importiert Dateien in diesem Format. Für den Zugriff auf komprimierte E00-Dateien verwendet das Modul die C Bibliothek *E00Compr* (MORISSETTE 2006).

Tab. 26: Module für die Georeferenzierung und Koordinatentransformation

Name/Bibliothek	**Beschreibung**	**Module**
Georeferencing **pj_georeference**	Georeferenzierung von Datensätzen	• Create Reference Points [interactive] • Georeferencing – Grids[AR] • Georeferencing – Shapes
GeoTrans **pj_geotrans**	Koordinatentransformation basierend auf GeoTrans	• GeoTrans (Shapes) • GeoTrans (Grid)
Proj4 **pj_proj4**	Koordinatentransformation basierend auf Proj.4	• Proj.4 (Shapes) • Proj.4 (Grid)

[AR]A.Ringeler

3.6.2 Georeferenzierung und Projektionen

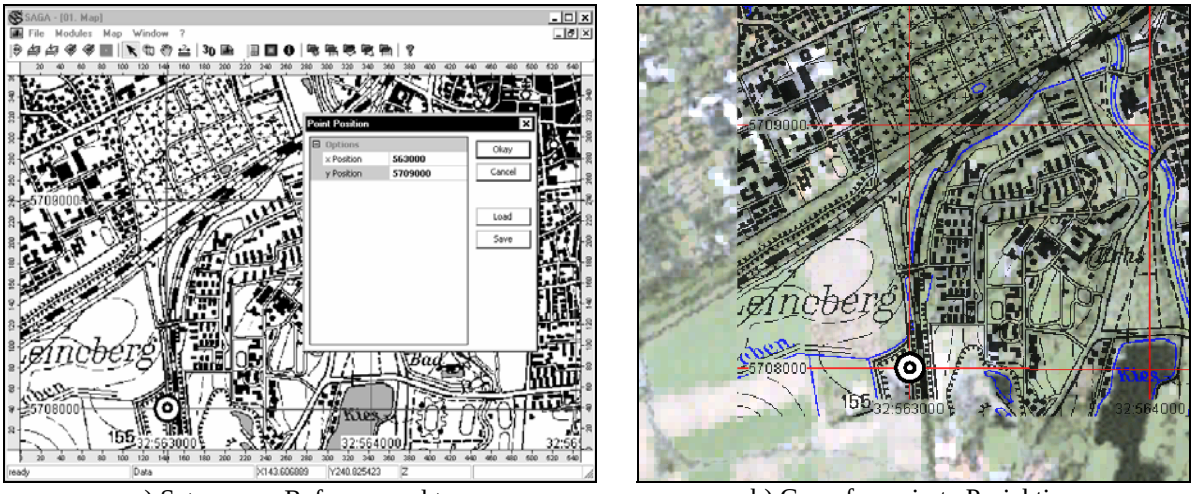
Nach dem Import von Datensätzen, besteht die nächste Hürde beim Arbeiten mit räumlichen Daten oft darin, dass alle Datensätze vor ihrer Verknüpfung in demselben Koordinatensystem vorliegen müssen. Während von einem Scanner digitalisierte Daten erst noch georeferenziert werden müssen, reicht bei bereits referenzierten Datensätzen die Projektion in das gewünschte Koordinatensystem. Für kartographische Projektionen stehen zwei alternative Bibliotheken zur Verfügung.

Georeferenzierung

Die Bibliothek **proj_georeference** enthält je ein Modul zum Georeferenzieren von Vektor- und Rasterdaten. Ein weiteres Modul Namens „*Create Reference Points*“ wird interaktiv ausgeführt und vereinfacht das Digitalisieren von Referenzpunkten, die für die Georeferenzierung verwendet werden sollen. Das interaktive Setzen von Referenzpunkten erfolgt durch Betätigen der linken Maustaste in einem Kartenfenster mit der Ausgangsprojektion und wird mit einer Dialogbox beantwortet, in der die Koordinaten der Zielprojektion eingegeben werden (Abb. 33a). Die Referenzpunkte werden in einem Punktdatensatz gespeichert, der dann an die Module zum Georeferenzieren übergeben werden kann. Auf diese Weise kann dieselbe Referenzierung bequem für mehrere Datensätze vorgenommen, aber auch nachbearbeitet werden. Voraussetzung ist, dass mindestens drei Referenzpunkte gegeben sind, damit ein Gleichungssystem für die Projektion erstellt werden kann. Für die Anpassung des Gleichungssystems werden die Funktionen der Minpack-Bibliothek benutzt (MORE et al. 1999).

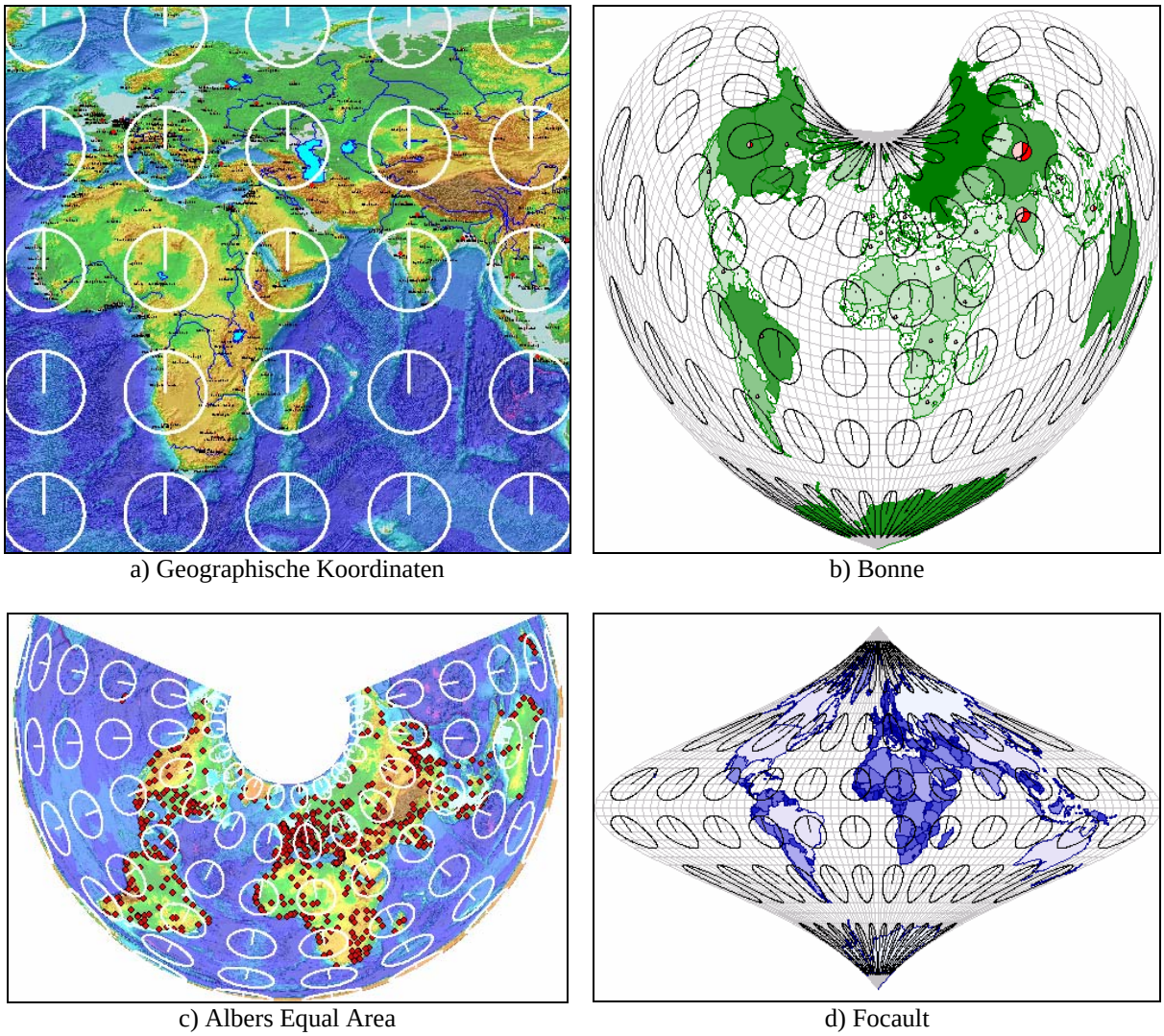
GeoTrans

Die Bibliothek **proj_geotrans** benutzt die Projektionsfunktionen der quelltextoffenen Bibliothek Geospatial Translator (GeoTrans), die von der National Imagery and Mapping Agency (NIMA) entwickelt wurde (NATIONAL GEOSPATIAL AGENCY 2006). Sie bietet jeweils ein Modul zum projizieren von Vektor- und Rasterdaten. Koordinatentransformationen



a) Setzen von Referenzpunkten b) Georeferenzierte Projektion

Abb. 33: Georeferenzierung einer gescannten Karte.



a) Geographische Koordinaten b) Bonne

c) Albers Equal Area d) Focault

Abb. 34: Kartographische Projektionen

können direkt von einer Projektion in eine andere vorgenommen werden. Datum und Ellipsoid können aus frei definierbaren Texttabellen gewählt werden. Es werden derzeit 34 Projektionen unterstützt (Abb. 34), zu denen neben verschiedenen Azimuthal-, Kegel- und Zylinderprojektionen auch die Universal Transverse Mercator (UTM) und die Transverse Mercator Projektion, auf der das in Deutschland verbreitete „Gauß-Krüger“ System basiert.

PROJ.4

Die quelltextoffene Projektionsbibliothek PROJ.4, die von den Modulen der Bibliothek **proj_proj4** für die Projektion von Vektor- bzw. Rasterdaten benutzt wird, ist vergleichbar mit der GeoTrans Bibliothek. PROJ.4 wurde von der United States Geological Survey (USGS) ins Leben gerufen (EVENDEN 2003), ist Bestandteil vieler Linux-Distributionen und wird u.a. auch von GRASS GIS verwendet (GRASS 2006). Datum und Ellipsoid lassen sich frei definieren. Projektionen erfolgen immer von geographischen Koordinaten in Projektionskoordinaten und umgekehrt. Anders als bei GeoTrans können Datensätze daher nicht direkt von einer

beliebigen Projektion in eine andere transformiert werden. Mit zur Zeit etwa 120 verschiedenen Projektionen ist PROJ.4 jedoch deutlich umfangreicher.

3.6.3 Tabellen

Das Arbeiten mit reinen Tabellendaten ist bis jetzt kein ausgesprochener Schwerpunkt von SAGA. Trotzdem werden auch bei der Arbeit mit raumbezogenen Daten immer wieder Funktionen benötigt, um Tabellen zu manipulieren oder auszuwerten, von denen einige als Modul implementiert wurden. Da Attributtabellen eines Vektordatensatzes von der GUI anders angesprochen werden als einfache Tabellen, werden für einige Funktionalitäten alternative Modulversionen angeboten, die auf Vektordatenattribute zugeschnitten sind.

Allgemeine Werkzeuge

Die SAGA-GUI selbst bietet keine Möglichkeit neue Tabellen anzulegen. Stattdessen wird diese wichtige Funktionalität von dem Modul „*Create empty table*“ angeboten, durch das eine leere Tabelle mit benutzerdefinierten Feldeigenschaften, wie Anzahl der Spalten und deren Namen und Datentypen, angelegt wird. Eine weitere Grundfunktion, die die Bibliothek **table_tools** für Tabellen bereitstellt, liefert das Modul „*Rotate table*“, das eine Tabelle kopiert, wobei eine Umstrukturierung von Reihen zu Spalten bzw. Spalten zu Reihen vorgenommen wird. „*Enumerate table data*“ weist allen gleichen Werten eines gewählten Tabellenfelds eine eindeutige Indexnummer zu und schreibt diese in eine zusätzlich angefügte Tabellenspalte. Mit „*Enumerate shapes attributes*“ lässt sich dieselbe Funktion auch auf die Attributtabellen von Vektordaten anwenden.

Mathematische Werkzeuge

Die ersten beiden Module der Bibliothek **table_calculus**, „*Calculator for table data*“ bzw. „*Calculator for shapes attributes*“, verknüpfen die Spaltenwerte eines Records und Schreiben das Ergebnis in eine neu angefügte Spalte. Die Art der Verknüpfung wird mit einer benutzerdefinierten Formel festgelegt, wobei die Tabellenspalten mit fortlaufenden Buchstaben (a, b,

Tab. 27: Module für das Arbeiten mit Tabellen

Name/Bibliothek	Beschreibung	Module
Tools **table_tools**	Allgemeine Werkzeuge	• Create empty table • Rotate table • Enumerate table data • Enumerate shapes attributes
Calculus **table_calculus**	Mathematische Werkzeuge	• Calculator for table data[VO] • Calculator for shapes attributes [VO] • Trend for table data • Trend for shapes attributes • Function Fit[AR]

[VO]V.Olaya, [AR]A.Ringeler

..., z) angesprochen werden. Auch für die Trendberechnung für die Spaltenwerte einer Tabelle stehen Modulversionen für einfache Tabellen wie für Vektordatenattribute zur Verfügung. Die Trendfunktion selbst ist über eine Formel frei definierbar. Ihre angepassten Funktionsparameter werden bei Erfolg in dem Benachrichtigungsfenster der GUI ausgegeben. Eine Spalte mit den Werten der Trendfunktion wird der Tabelle angefügt.

3.6.4 Vektordaten

Die etwas umfangreicheren Werkzeuge für die Analyse und Manipulation von Vektordaten sind unterteilt in allgemeine Werkzeuge, spezielle Werkzeuge für Punkte, Linien und Polygone sowie Werkzeuge zur Verknüpfung mit Rasterdaten bzw. zur Ableitung von Vektordaten auf der Basis von Rasterdaten.

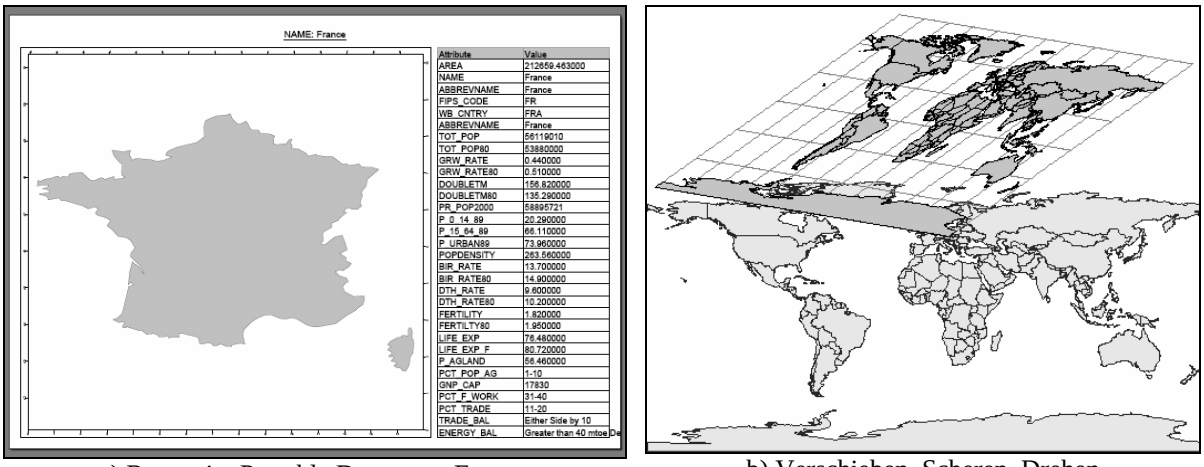
Allgemeine Werkzeuge für Vektordaten

Die Bibliothek **shapes_tools** versammelt Werkzeuge, die sich auf Punkte genauso wie auf Linien und Polygone anwenden lassen. Wie für Tabellen (Kap. 3.6.3) gibt es auch für die direkte Erstellung neuer Vektordatensätze keine Funktion in der SAGA-GUI, so dass hierfür das Modul „*Create Empty Shapes Layer*“ benutzt werden muss. Das Modul erwartet die Eingabe des gewünschten Vektordatentyps sowie die Definition der Attributtabellenstruktur. Mit „*Assign a Table to Shapes*“ können die Werte einer Tabelle den Attributen eines Vektordatensatzes zugewiesen werden, wobei die Verknüpfung über die Identifikationswerte der vom Benutzer auszuwählenden Spalten erfolgt. Zwei Vektordatensätze vom gleichen Vektordatentyp können mit „*Merge Shapes Layers*“ zu einem Vektordatensatz zusammengefügt werden. Das Modul „*Transform Shapes*“ verschiebt, dreht und streckt die Punktkoordinaten eines Vektordatensatzes (Abb. 35b). Mit „*Separate Shapes*“ werden alle Elemente, die in einem vom Benutzer ausgewählten Attributfeld denselben Wert haben, in einen neuen Vektordatensatz kopiert. Wenn in der GUI Elemente eines Vektordatensatzes markiert wurden, können diese mit „*New Layer from selected Shapes*“ in einen neuen Vektordatensatz kopiert werden. Zum Markieren von Elementen eines Vektordatensatzes kann z.B. das Modul „*Query builder for shapes*“ benutzt werden, wobei die Auswahl die Syntax des API Formelparsers verwendet und Attributfelder fortlaufend mit den Buchstaben a, b, ..., z bezeichnet werden. In ähnlicher Weise funktioniert das Modul „*Search in attributes table*“, wobei bei diesem Modul nicht ein Formelausdruck ausgewertet wird, sondern der als Parameter angegebene Wert selbst als Auswahlkriterium dient. Eine dritte Möglichkeit Elemente zu markieren bietet das Modul „*Select by theme*“, welches auf räumliche Übereinstimmung mit einem weiteren Vektordatensatz prüft, ob ein Element markiert werden soll oder nicht. Mit „*Create Graticule*“ kann ein Gitternetz als Linien- oder Polygondatensatz für weitere Darstellungs- oder Analysefunktionen erstellt werden. Ein weiterer sehr spezieller Vektordatensatz mit Polygonen, die Kreis- oder Balkendiagramme darstellen, wird von dem Modul „*Create Chart Layer*“ erstellt. Dieses Modul ist obsolet, da die Darstellung von Vektorattributen in Kreis- u. Balkendiagrammen von der GUI der aktuellen SAGA-Version direkt unterstützt werden (Kap. 3.4).

Tab. 28: Module für das Arbeiten mit Vektordaten

Name/Bibliothek	**Beschreibung**	**Module**
Shapes – Tools **shapes_tools**	Allgemeine Werkzeuge für Vektordaten	• Create Empty Shapes Layer • Assign a Table to Shapes • Merge Shapes Layers[VO] • New layer from selected shapes[VO] • Query builder for shapes[VO] • Search in attributes table[VO] • Select by theme[VO] • Separate Shapes[VO] • Transform Shapes[VO] • Create Chart Layer (bars/sectors)[VO] • Create graticule[VO] • Create PDF Report for Shapes Layer[VO] • Summary[VO] • Create Web Content [interactive][VO]
Shapes – Points **shapes_points**	Werkzeuge für Punkte	• Convert a Table to Points[VO] • Count Points in Polygons[VO] • Create Point Grid[VO] • Distance Matrix[VO] • Points From Lines[VO] • Add Coordinates to points[VO]
Shapes – Lines **shapes_lines**	Werkzeuge für Linien	• Convert Polygons to Lines
Shapes – Lines (extended) **shapes_lines_ex**	Werkzeuge für Linien (erweitert)	• Simplify Lines[VO]
Shapes – Polygons **shapes_polygons**	Werkzeuge für Polygone	• Polygon Intersection • Polygon Centroids • Geometrical Properties of Polygons[VO] • Convert Lines to Polygons • Polygon statistics from points[VO]
Shapes – Grid **shapes_grid**	Werkzeuge zur Verknüpfung von Raster- und Vektordaten	• Add Grid Values to Points • Clip Grid with Polygon[SL] • Gradient from Grid • Grid Statistics for Polygons • Grid Values to Points • Grid Values to Points (randomly) • Contour Lines from Grid • Vectorising Grid Classes

[SL]Stefan Liersch, [VO]V.Olaya



a) Report im Portable Document Format

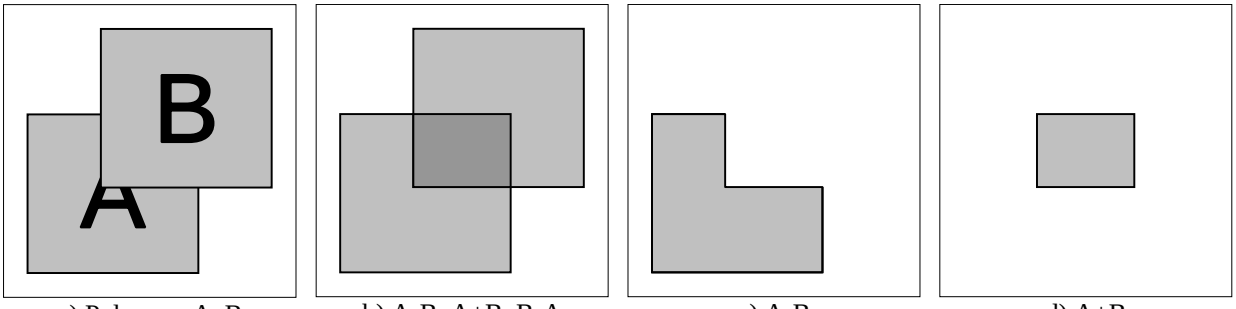
b) Verschieben, Scheren, Drehen

Abb. 35: Allgemeine Werkzeuge für Vektordaten

Die Module „*Create PDF Report*“ und „*Summary*“ erstellen PDF-Dokumente mit kartographischer und tabellarischer Ausgabe für jedes Element des gewählten Vektordatensatzes (Abb. 35a). In ähnlicher Weise erzeugt „*Create Web Content*“ ein HTML-Dokument, wobei die Vektorobjekte, die dem Dokument hinzugefügt werden sollen, interaktiv auswählbar sind.

Werkzeuge für Punkte

Werkzeuge speziell für die Erstellung und Analyse von Punktdaten stellt **shapes_points** zur Verfügung. Daten werden oft zusammen mit zugehörigen Punktkoordinaten in Tabellen gespeichert. Mit „*Convert a Table to Points*“ können solche Tabellen direkt in einen Punktdatensatz konvertiert werden. In umgekehrter Weise können mit „*Add Coordinates to Points*“ Koordinaten den Attributdaten eines Punktdatensatzes in zwei zusätzlichen Feldern angefügt werden. Punktdatensätze können mit „*Points from Lines*“ auch aus den Punkten eines Liniendatensatzes erstellt werden, wobei jeder Punkt die Attributdaten seiner ursprünglichen Linie erhält. „*Create Point Grid*“ erstellt einen Datensatz mit in einem regelmäßigen Raster angeordneten Punkten. Mit „*Distance Matrix*“ wird eine Tabelle erstellt, in der die Distanz zwischen jedem Punkt eines Punktdatensatzes eingetragen ist. Das Modul „*Count Points in Polygons*“ zählt, wie viele Punkte eines Punktedatensatzes in jedem Polygon eines Polygondatensatzes liegen, und fügt dem Polygondatensatz eine Spalte mit dem Ergebnis an.



a) Polygone A, B

b) A-B, A+B, B-A

c) A-B

d) A+B

Abb. 36: Verschneidung von Polygonen.

Werkzeuge für Linien

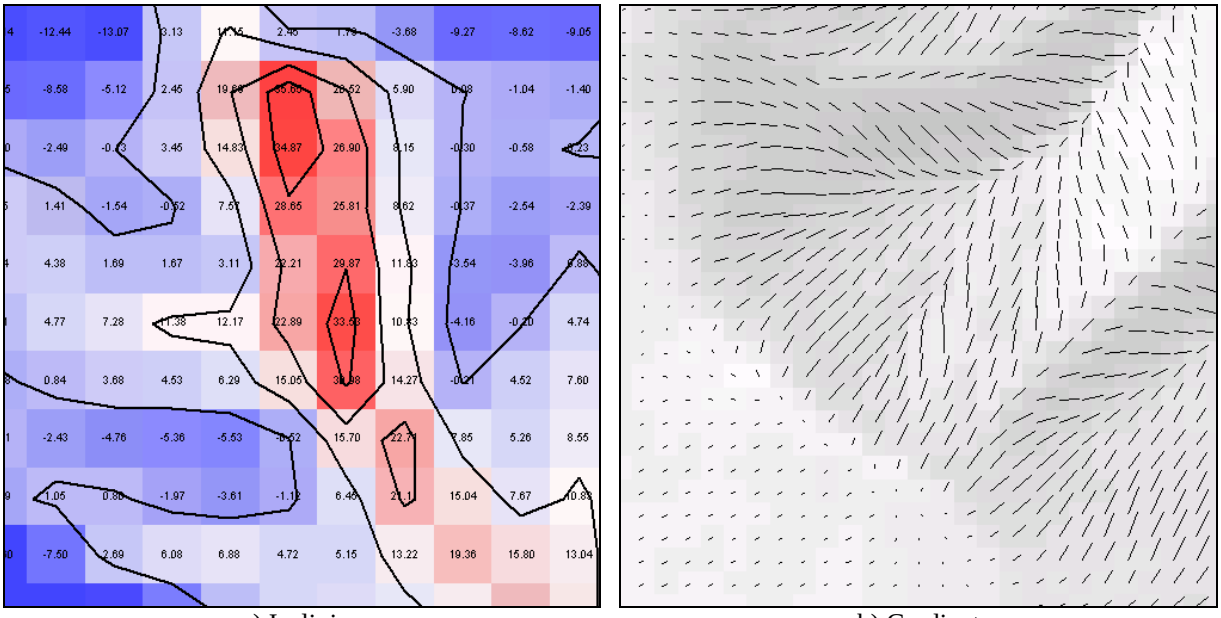
Die Kanten eines Polygondatensatzes werden von „*Convert Polygons to Lines*“ in einen Liniendatensatz konvertiert. Bei Linien, die sich aus übermäßig vielen Einzelpunkten zusammensetzen, lässt sich mit „*Simplify Lines*“ die Anzahl ihrer Stützpunkte so reduzieren, dass ihre ursprüngliche Geometrie möglichst wenig verändert. Da dieses Modul intern ein komplexeres Datenmodell verwendet, das nicht mit allen Compilern kompatibel ist, wurde sie nicht in die Bibliothek **shapes_lines**, sondern in eine eigene namens **shapes_lines_ex** ausgelagert.

Werkzeuge für Polygone

Die Bibliothek **shapes_polygons** versammelt Funktionen für das Arbeiten mit Polygonen. Mit „*Convert Lines to Polygons*“ kann ein Liniendatensatz in einen Polygondatensatz konvertiert werden. Ob das Ergebnis sinnvoll ist, hängt allerdings von der Geometrie der ursprünglichen Linien ab. „*Polygon Centroids*“ erstellt einen Punktdatensatz mit den Koordinaten der Zentroide sowie den Attributen der Elemente eines Polygondatensatzes. Die Verschneidung von zwei Polygondatensätzen wird von dem Modul „*Polygon Intersection*“ ausgeführt, wobei zwischen vollständiger Verschneidung und Differenzbildung unterschieden werden kann (Abb. 36). Das Modul „*Geometrical Properties of Polygons*“ berechnet die Fläche und den Umfang für jedes Polygon und fügt diese der Attributtabelle hinzu.

Werkzeuge zur Verknüpfung von Raster- und Vektordaten

In der Bibliothek **shapes_grid** befinden sich Module, die auf der Verknüpfung von Vektor- und Rasterdaten basieren. Mit „*Add Grid Values to Points*“ können für jeden Punkt eines Punktdatensatzes die räumlich übereinstimmenden Werte einer beliebigen Anzahl von Rasterdatensätzen abgefragt und als neue Attribute hinzugefügt werden. Das Modul „*Clip*



a) Isolinien b) Gradient

Abb. 37: Ableitung von Vektordaten aus Rasterdaten.

Grid with Polygon“ setzt alle Rasterzellen auf einen „No-Data“ Wert, die nicht von dem gewählten Polygondatensatz abgedeckt werden, so dass z.B. bei einer Einzugsgebietsanalyse einzugsgebietsfremde Bereiche von folgenden Analysen ausgeschlossen werden. „*Grid Values to Points*“ konvertiert alle Rasterzellen eines Rasterdatensatzes zu Punkten eines Punktdatensatzes. Ein zweites, gleichlautendes Modul extrahiert in gleicher Weise, aber in einer benutzerdefinierten Frequenz, eine zufällige Untermenge aller Rasterzellen. Diese Funktion ist z.B. hilfreich bei der Erstellung von Testdatensätzen. Die Erstellung von Isolinien mit einer gegebenen Äquidistanz für die Datenoberfläche eines Rasters wird mit dem Modul „*Contour Lines from Grid*“ vorgenommen (Abb. 37a). Die Stützstellen einer Isolinie werden dabei mittels linearer Interpolation bestimmt. Anders als bei der Isolinienerstellung, interpoliert das Modul „*Vectorising Grid Classes*“ nicht zwischen den Rasterzellen, sondern ermittelt Grenzlinien für jeden Wertewechsel, der zwischen Rasterzellen auftritt. Das Modul eignet sich dafür, zusammenhängende Flächen mit gleichen Datenwerten zu vektorisieren. Ein weiteres Modul zur Vektorisierung von Rasterdaten unterstützt vor allem die graphische Darstellung. „*Gradient from Grid*“ berechnet aus dem Wertefeld eines Rasters den Gradienten und legt für jede Rasterzelle eine Linie mit seiner Richtung und einer Länge, die der Stärke des Gradienten entspricht, an (Abb. 37b).

3.6.5 TIN

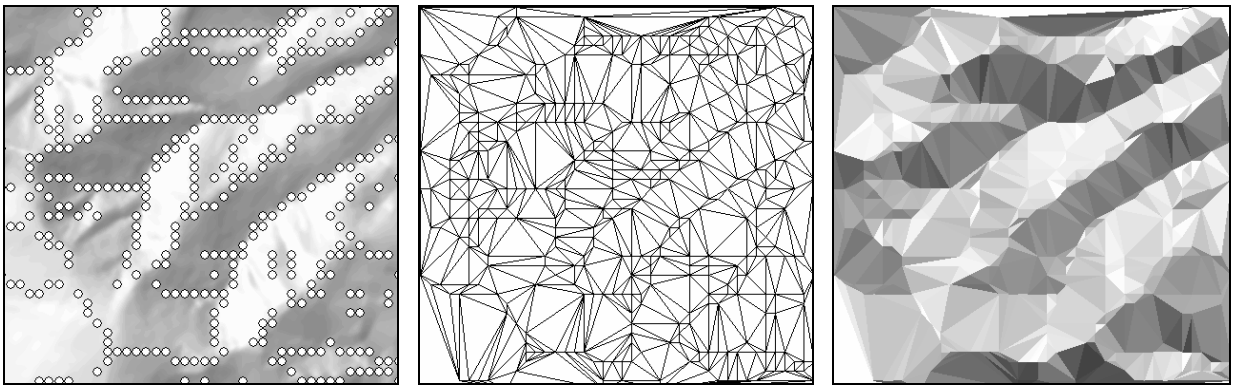
TIN sind eine spezielle Vektordatenstruktur und gleichzeitig auch die jüngste Datenstruktur, die der SAGA-API hinzugefügt wurde. Daher wurden bisher auch nur wenige Module speziell für die Verarbeitung von TIN erstellt. Die meisten Module beschränken sich auf die Konvertierung von und zu anderen Datensatztypen. Auch die drei reliefanalytischen Methoden dienen in erster Linie der Demonstration der Funktionsweise von TIN und sollten in dieser Form nicht für wissenschaftliche Analysen verwendet werden. Die TIN-Module werden von den Bibliotheken **tin_tools** und **tin_terrain_analysis** bereitgestellt.

Konvertierung

Mit „*Shapes to TIN*“ wird ein TIN aus den Stützpunkten eines Vektordatensatzes erzeugt, wobei jeder Stützpunkt die Attributdaten des ihm zu Grunde liegenden Vektorobjekts erhält.

Tab. 29: Module für das Arbeiten mit TIN

Name/Bibliothek	**Beschreibung**	**Module**
TIN – Tools *tin_tools*	Werkzeuge für TIN	• Grid to TIN • Grid to TIN (Surface Specific Points) • Shapes to TIN • TIN to Shapes
Terrain Analysis for TIN *tin_terrain_analysis*	Reliefanalyse für TIN	• Gradient • Flow Accumulation (Trace) • Flow Accumulation (Parallel)



a) Hervorgehobene Reliefpositionen b) Dreiecksvermaschung c) Exposition

Abb. 38: Reliefanalyse mit TIN.

Diese Konvertierung ist identisch mit dem direkten Laden eines Vektordatensatzes mit der TIN-Funktion der SAGA-GUI (Kap.3.4). Das Gegenstück hierzu bildet „*TIN to Shapes*“ für die Konvertierung eines TIN in Vektordatensätze, welche für die durch das TIN definierten Punkte, Kanten, Dreiecke und Voronoi- bzw. Thiessen-Polygone erstellt werden. Das Modul „*Grid to TIN*“ erstellt ein TIN aus allen Gitterpunkten eines Rasterdatensatzes. Die Attributtabelle erhält die entsprechenden Rasterzellenwerte zugewiesen. Die durch dieses Modul generierten TIN sind sehr dicht und bringen gegenüber der Speicherung als Raster keinen Vorteil, abgesehen von der Möglichkeit, TIN spezifische Methoden auf die Daten anwenden zu können. Das zweite Modul zur Raster-TIN Konvertierung erstellt ein weniger dichtes Datennetz, da es nur hervorstechende Merkmale bzw. Rasterzellen, die eine geringe Repräsentativität für ihre Umgebung besitzen, in das TIN aufnimmt (Abb. 38a, b). Die Methoden zur Identifizierung solcher Rasterzellen wurden ursprünglich für die Ermittlung hervorgehobener Reliefpositionen entwickelt (s. Kap. 0, „*Surface Specific Points*“).

Reliefanalyse

Das Modul „*Gradient*“ eignet sich z.B. zur Ableitung von Hangneigungs- und Expositionswerten aus Höhendaten (Abb. 38c). Die Berechnung erfolgt für jede Dreiecksfazette des TIN. Da die Dreiecksfazetten unter SAGA derzeit keine eigene Attributtabelle besitzen, wird ein aus Dreiecken bestehender Polygondatensatz erstellt, um das Ergebnis zu speichern. Anders ist es mit der Berechnung der Einzugsgebietsgrößen durch die prinzipiell zu gleichen Ergebnissen führenden Module „*Flow Accumulation (Trace)*“ und „*Flow Accumulation (Parallel)*“. Hier beziehen sich Berechnung und Ergebnis auf die Stützpunkte, so dass das Ergebnis wieder als TIN ausgegeben wird. Im Vergleich mit den rasterbasierten Methoden ist die Reliefanalyse auf Basis von TIN mit den Modulen dieser Bibliothek deutlich unterlegen. Als Vorteil von TIN gilt, dass sie Datenredundanz verringern können. Ihre Stärke entfalten TIN vor allem bei rechenzeitintensiven Problemlösungen, z.B. bei dynamischen Simulationen mit Methoden der Finiten Elemente (FEM) arbeiten (z.B. FAIRES & BURDEN 1994).

3.6.6 Rasterdaten

Die Zahl der Werkzeuge und Methoden, die speziell auf die Rasterverarbeitung zugeschnitten sind, verrät, dass SAGA ursprünglich vor allem auf Rasterdaten ausgerichtet war. Die Gruppe der in dieser Kategorie zusammengefassten Module enthält einfache und allgemeine Werkzeuge für die Erstellung, Aufbereitung, Verschneidung und Analyse von Rasterdaten.

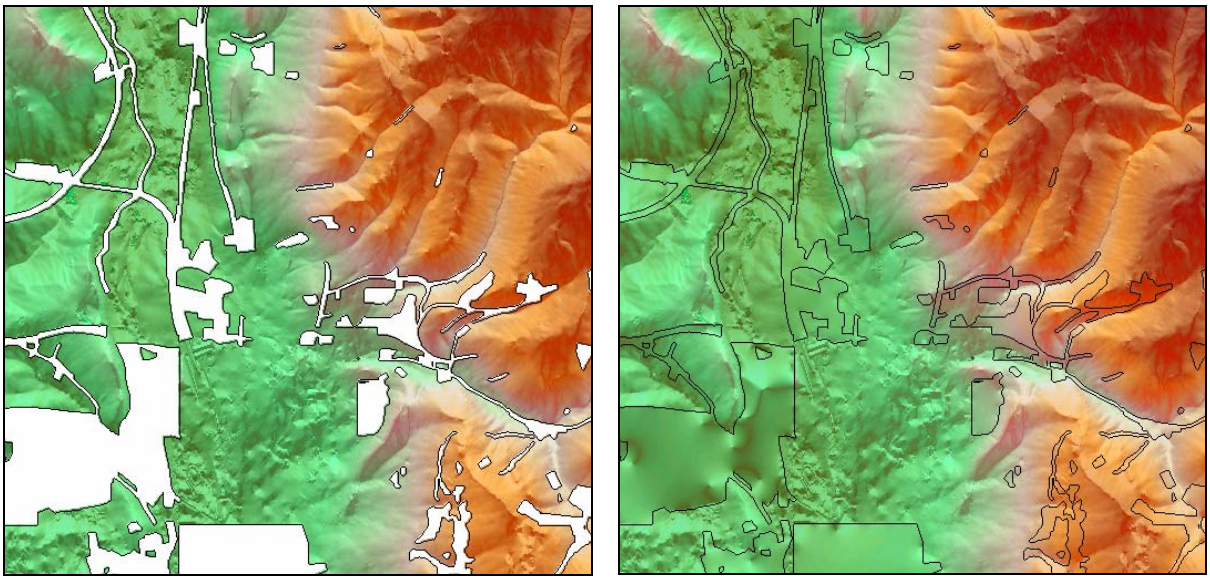
Tab. 30: Module für das Arbeiten mit Rasterdaten

Name/Bibliothek	**Beschreibung**	**Module**
Grid – Tools **grid_tools**	Allgemeine Werkzeuge für Rasterdaten	• Resampling • Aggregate[VO] • Cutting [interactive] • Merging • Create Constant Grid[VO] • Completion • Close One Cell Gaps • Close Gaps • Grid Buffer[VO] • Threshold Buffer[VO] • Convert Data Storage Type • Change Grid Values • Change Cell Values [interactive][VO] • Grid Value Request [interactive] • Reclassify Grid Values[VW] • Change Grid Values - Flood Fill • [interactive][AR] • Crop to Data[VO] • Invert Data/No-Data[VO] • Grid Orientation • Combine Grids[VO] • Sort Grid[VO] • Grids from classified grid and table[VO]
Grid – Gridding **grid_gridding**	Rasterung von Vektordaten	• Inverse Distance • Nearest Neighbour • Modifed Quadratic Shepard[AR] • Shapes to Grid • Triangulation
Grid – Spline Interpolation **grid_spline**	Rasterung von Vektordaten mittels Spline-Interpolation	• Thin Plate Spline (Global) • Thin Plate Spline (Local) • Thin Plate Spline (TIN) • B-Spline Approximation • Multilevel B-Spline Interpolation

Tab. 30: Module für das Arbeiten mit Rasterdaten (Fortsetzung)

Name/Bibliothek	Beschreibung	Module
Grid – Filter **grid_filter**	Filter für Rasterdaten	• Simple Filter • Gaussian Filter[AR] • Laplacian Filter[AR] • Multi Direction Lee Filter[AR] • User Defined Filter (3x3)
Grid – Calculus **grid_calculus**	Rasterbasierte Kalkulation	• Grid Normalisation • Grid Calculator[AR] • Grid Volume • Function[AR] • Geometric Figures • Random Terrain Generation[VO] • Random Field
Grid – Discretisation **grid_discretisation**	Werkzeuge für die Diskretisierung und Klassifikation von Rasterdaten	• Supervised Classification • Cluster Analysis for Grids • Grid Segmentation • Grid Skeletonization
Grid – Analysis **grid_analysis**	Funktionen für die Analyse von Rasterdaten	• Accumulated Cost (Isotropic)[VO] • Accumulated Cost (Anisotropic)[VO] • Polar To Rect[VO] • Rect To Polar[VO] • Least Cost Path [interactive][VO] • Vegetation Index[distance based][VO] • Vegetation Index[slope based][VO] • Fuzzy intersection grid[BG] • Fuzzy union grid[BG] • Fuzzify, Change Vector Analysis[VO] • Covered Distance[VO] • Pattern Analysis[VO] • Layer of extreme value[VO] • Analytical Hierarchy Process[VO] • Aggregation Index[VO] • Cross-Classification and Tabulation[VO]
Grid – Visualisation **grid_visualisation**	Funktionen für die Visualisierung von Rasterdaten	• Color Palette Rotation • Color Blending • Fit Color Palette to Grid Values • RGB Composite • Create 3D Image

[BM]A.Boggia u. G.Massei, [VO]V.Olaya, [AR]A.Ringeler, [VW] V.Wichmann



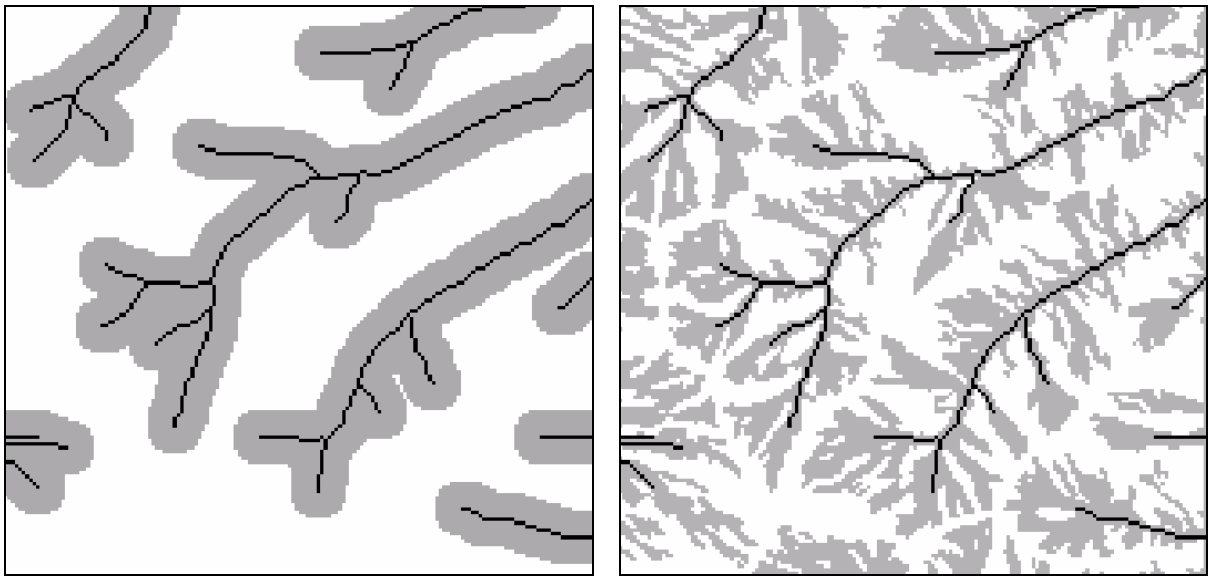
a) Lückenhaftes Original b) Lückenloses Ergebnis

Abb. 39: Ergänzung von Werten.

Allgemeine Werkzeuge für Rasterdaten

Die Module der Bibliothek **grid_tools** sind bereits so zahlreich geworden, dass eine weitere Aufteilung auf speziellere Bibliotheken, z.B. für die Erstellung von Pufferzonen oder der Manipulation von Rasterzellenwerten, durchaus sinnvoll erscheint. Die Menüstruktur, in der die Module in der SAGA-GUI eingeordnet sind, zeigt bereits eine solche Untergliederung.

Eine Gruppe von Modulen dient der Erstellung neuer Rasterdatensätze. Das Modul „*Create Constant Grid*“ erstellt ein Raster, in dem alle Zellen auf denselben, frei definierbaren Wert gesetzt sind. Die Rasterwerte eines mit „*Cutting*“ interaktiv wählbaren Gebiets werden in ein neu erstelltes Raster kopiert. „*Crop to Data*“ ermittelt das von mehreren Rasterdatensätzen tatsächlich mit gültigen Werten (nicht No-Data) abgedeckte Gebiet und kopiert ihre Werte in neue Raster mit passend geschrumpfter Ausdehnung. Randgebiete, die in allen Rastern ausschließlich No-Data Werte enthalten, werden so abgeschnitten. Mit „*Merging*“ können mehrere nebeneinander liegende Rasterdatensätze zu einem einzigen Rasterdatensatz zusammengesetzt werden. Für überlappende Bereiche werden Mittelwerte gebildet. Bei unterschiedlichen Rasterzellengrößen der Eingabedatensätze wird die kleinste für die Ausgabe gewählt. Mit „*Resampling*“ werden einzelne Rasterdatensätze auf größere oder kleinere Rasterzellengrößen skaliert, gegebenenfalls auch mit einem Versatz des Rasters, wobei zwischen verschiedene Interpolationsmethoden gewählt werden kann. „*Aggregate*“ fasst mehrere Rasterzellen zu einer Zelle zusammen. Der Wert der aggregierten Rasterzellen ist wahlweise die Summe, Mittelwert, Minimum oder Maximum der Ausgangsrasterzellen. Aus einem Raster mit Klassenwerten und einer Tabelle mit den Klassen zugehörigen Attributwerten können mit „*Grids from classified grid and table*“ neue Raster mit den Attributwerten erstellt werden.



a) Einfacher Puffer b) Schwellenwert-Puffer

Abb. 40: Pufferzonen

Raster, die aus No-Data Werten bestehende Lücken aufweisen, lassen sich unkompliziert mit drei Modulen vervollständigen. „*Close One Cell Gaps*“ setzt Lücken, die nur aus einer Rasterzelle bestehen, auf den Mittelwert der sie umgebenden Rasterzellen. Das Schließen größere Lücken ermöglicht „*Close Gaps*“ unter Verwendung eines etwas aufwändigeren Interpolationsverfahrens (Abb. 39). Mit „*Patching*“ können Lücken auch durch die Werte eines anderen Rasterdatensatzes ergänzt werden.

Zwei Module dienen der Ausgabe von Pufferzonen. Beide erwarten die Eingabe eines Rasterdatensatzes, in dem die Pufferzonenkerne durch gültige Werte, alle sonstigen Rasterzellen durch No-Data Werte gekennzeichnet sind. „*Grid Buffer*“ erstellt distanzabhängige Pufferzonen. Standardmäßig wird eine einheitliche Pufferdistanz verwendet (Abb. 40a). Optional kann aber auch der Wert des Pufferkerns zur lokalen Festlegung der Pufferzonendistanz benutzt werden. „*Threshold Buffer*“ arbeitet nicht mit einer vorgegebenen Pufferzonendistanz, sondern erweitert den Puffer um alle Rasterzellen, die ausgehend vom Pufferzonenkern einen gegebenen Schwellenwert nicht überschreiten (Abb. 40b).

Eine weitere Gruppe von Modulen dient der gezielten Änderung von Datenwerten. Eine sinnvolle Hilfsfunktion bietet in diesem Zusammenhang das Modul „*Convert Data Storage Type*“ zum Ändern des für die Datenspeicherung verwendeten Datentyps. Mit dem Datentyp wird der durch die Daten darstellbare Wertebereich festgelegt. Der Datentyp Byte kann z.B. nur ganzzahlige Werte zwischen 0 und 255 speichern (s.a. Kap. 2.1.1), wodurch zwar Speicherplatz gespart werden kann, viele Fragestellungen aber nicht ausreichend gelöst werden können. Die Module „*Change Grid Values*“ und „*Reclassify Grid Values*“ funktionieren nach dem gleichen Prinzip, wobei das letztere aber über wesentlich mehr Optionen verfügt. Beide Module benutzen eine Nachschlagetabelle, um vorgegebene Werte oder Wertebereiche durch neue Werte zu ersetzen. Eine Nachschlagetabelle wird auch von

„*Combine Grids*“, für die Festlegung von Zielwerten für unterschiedliche Wertekombinationen zweier Eingaberaster benutzt. Mit „*Change Cell Values*“ werden einzelne Rasterzellen interaktiv ausgewählt und auf einen neuen Wert gesetzt. „*Change Grid Values - Flood Fill*“ arbeitet ähnlich, ersetzt aber nicht nur den Wert der ausgewählten Rasterzelle, sondern auch die aller Rasterzellen, die gleiche Werte aufweisen oder durch Rasterzellen mit gleichen Werten in direkter Verbindung zu ihr stehen. Statt nur gleiche Werten zu nehmen, kann auch eine zu tolerierende Wertespanne angegeben werden.

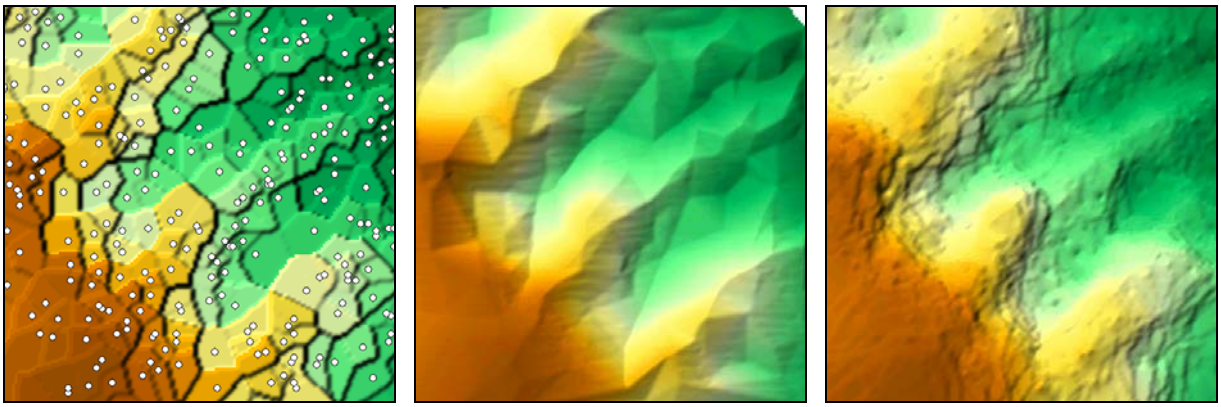
Das Modul „*Invert Data/No-Data*“ erstellt ein Raster, in dem alle Zellen, die im Ausgangsraster gültige Werte haben, auf einen No-Data Wert gesetzt sind. No-Data Zellen des Ausgangsrasters werden auf den Wert 1 gesetzt. Dadurch eignet sich das Modul z.B. zur Erstellung von Masken. „*Grid Orientation*“ hat zwei Optionen zum Spiegeln der Datenwerte an der horizontalen bzw. vertikalen Mittellinie des Rasters, und eine zum Invertieren, so dass der höchste Wert zum niedrigsten wird und umgekehrt. „*Sort Grid*“ nummeriert die Zellen eines Rasters vom niedrigsten zum höchsten Wert und schreibt die Nummerierung in ein neues Raster.

Rasterung von Vektordaten

Für die Erstellung von Rasterdaten aus Vektordaten, dem sogenannten Aufrastern, können verschiedenste Methoden benutzt werden, deren jeweilige Eignung von der Art und Beschaffenheit der Eingangsdaten als auch dem Verwendungszweck abhängt. Für das Aufrastern muss immer ein Attributmerkmal des Vektordatensatzes gewählt werden, dessen Werte in ein Raster übertragen werden sollen. Kriging-Verfahren verfolgen einen geostatistischen Ansatz für die Interpolation von Punktdaten und werden mit ihrer besonderen Methodik erst in Kapitel 3.6.7 diskutiert. Module für die Spline-Interpolation befinden sich ebenfalls in einer eigenen Bibliothek und werden im nächsten Kapitel vorgestellt. Die Bibliothek **grid_gridding** vereint vergleichsweise einfache Methoden.

Das Modul „*Shapes to Grid*“ schreibt die Attributwerte in jede Rasterzelle, die von einem Vektorobjekt abgedeckt wird. Stimmt eine Rasterzelle mit mehreren Vektorelementen überein, so werden die Attributwerte gemittelt. Allen Rasterzellen, für die keine räumliche Übereinstimmung mit einem Vektorobjekt gefunden wird, erhalten einen No-Data Wert zugewiesen. Dieses Modul ist das einzige, das Linien und Polygone direkt in ein Raster überträgt. Alle anderen Module zur Aufrasterung verwenden Interpolations- oder Approximationsverfahren, die sich auf Punktdaten beziehen. Im Falle von Linien und Polygonen werden deren Stützstellen als Punktdaten mit dem Attributwert des jeweiligen Vektorobjekts interpretiert.

Durch das Modul „*Nearest Neighbour*“ erhält jede Rasterzelle den Wert des am nächsten gelegenen Datenpunktes (Abb. 41a). Diese Methode eignet sich speziell zum Aufrastern von diskreten oder klassifizierten Werten, zwischen denen nicht interpoliert werden soll. „*Triangulation*“ erstellt aus den Datenpunkten ein TIN (s.a. Kap. 3.6.5). Die Ebenen, die durch die Dreiecksfazetten des TIN definiert werden, können darauf unmittelbar für die



a) Nearest Neighbour b) Triangulation c) Inverse Distance

d) Modified Quadratic Shepard e) Thin Plate Spline (mit Glättung) f) Multilevel B-Spline

Abb. 41: Rasterung von Punktdaten mit Höhenwerten

lineare Interpolation benutzt werden (Abb. 41b). Das Modul „*Inverse Distance*“ bildet für jede Rasterzelle einen distanzgewichteten Mittelwert von allen Datenpunkten, die innerhalb eines benutzerdefinierten Suchradius liegen (Abb. 41c). Die Zahl der Datenpunkte lässt sich auf eine Maximalzahl nächstgelegener Punkte einschränken. Das Verfahren von „*Modified Quadratic Shepard*“ ist eine Variation des Inverse Distance Verfahrens, das die Überbewertung sehr dicht gelegener Datenpunkte ausgleicht und dadurch eine ruhigere Datenoberfläche erzeugt (Abb. 41d).

Rasterung von Vektordaten mittels Spline-Interpolation

Zwei grundlegende Spline-Verfahren werden von der Bibliothek **grid_spline** für die Rasterung von Vektordaten angeboten. *Thin Plate Splines*“ (TPS), auch bekannt als *Tension Splines* oder *Minimum Curvature Splines* (z.B. MITASOVA & MITAS 1993), werden in ihrem Verhalten mit einer dünnen elastischen Platte verglichen, die sich an die Datenpunkte derart anpasst, dass sie minimal gewölbt ist. Das hier verwendete Verfahren zur Lösung der TPS-Funktion basiert auf dem Artikel von DONATO & BELONGIE (2002). Wenn der Relaxationsparameter der TPS-Funktion größer als 0 ist, wird die Spannung der Platte verringert. In der Folge wird die Datenoberfläche geglättet und die Datenpunkte werden unter Umständen nur noch angenähert von der Spline-Funktion getroffen (Abb. 41e). Die Bibliothek bietet drei Module für die TPS-Interpolation bzw. Approximation an. Das erste Modul (*global*) löst die

TPS-Funktion für alle gegebenen Datenpunkte und eignet sich nur für kleine Datenmengen. Da der Einfluss der Datenpunkte mit zunehmender Entfernung abnimmt, beschränken sich die beiden anderen Module auf die jeweils nächstgelegenen Punkte. Während das eine Modul (*local*) mit einem Suchradius arbeitet, benutzt das andere ein TIN, um schnellen, flächendeckenden Zugriff auf die nächstgelegenen Datenpunkte zu erhalten.

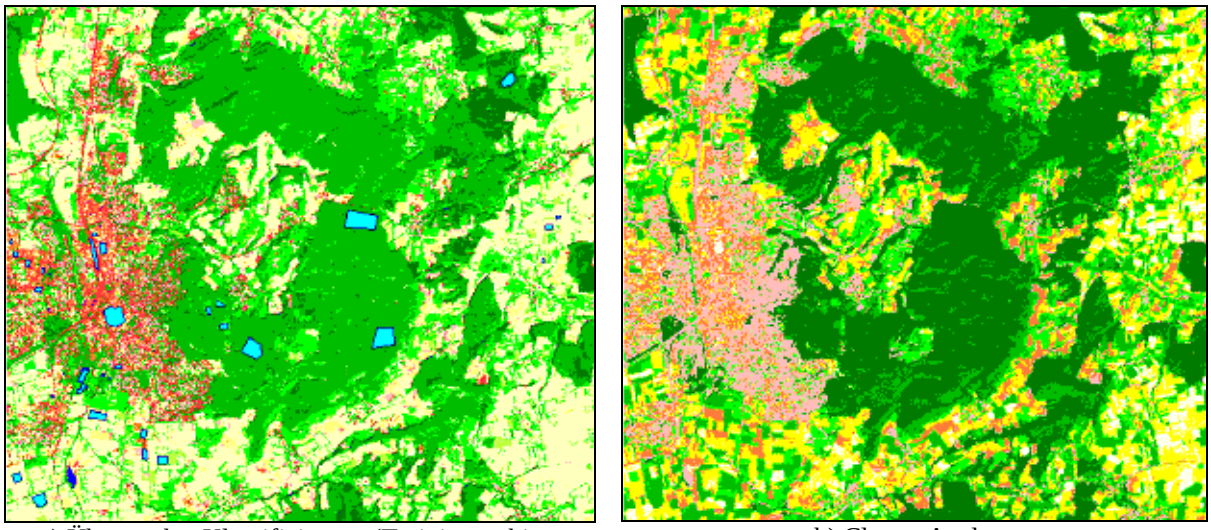
Das zweite Spline-Verfahren verwendet *bikubische B-Splines* (Basis-Splines) für die Interpolation. Das Modul „*B-Spline Approximation*“ berechnet allerdings nur den Einfluss der Datenpunkte auf ihre Umgebung mittels einfacher B-Spline Funktionen für gegebene Auflösungen und ist für sich genommen nicht für die eigentliche Dateninterpolation geeignet. Aber es demonstriert die Grundfunktion, die von dem Modul „*Multilevel B-Spline Interpolation*“ verwendet wird (Abb. 41f), welches das von LEE et al. (1997) vorgeschlagene Verfahren implementiert.

Filter

Filter, wie die von **grid_filter** bereitgestellten, werden für die Aufbereitung von Rasterdaten benutzt, aber auch, im Sinne der Bildverarbeitung, für die Herausarbeitung z.B. linienförmiger Elemente. Filtermethoden analysieren die Nachbarschaft einer Rasterzelle, um ihren Wert gemäß vorgegebener Regeln neu zu bestimmen. Tiefpassfilter berechnen einen Mittelwert, um die Datenoberfläche zu glätten, und eignen sich u.a. zur Reduzierung des Datenrauschens. Hochpassfilter verstärken dagegen die vorhandenen Unterschiede in benachbarten Datenwerten, so dass z.B. Kanten hervorgehoben werden. Das Modul „*Simple Filter*“ führt wahlweise eine Glättung, Schärfung oder Kantenhervorhebung durch und analysiert dafür alle Rasterzellen in einem benutzerdefinierten Suchradius. Mit „*User Defined Filter*“ kann frei angegeben werden, wie die Werte innerhalb eines 3x3 Rasterzellen großen Suchfensters für den Filterungsprozess gewichtet werden. „*Gaussian Filter*“ nimmt für die Mittelwertbildung eine Distanzgewichtung der Datenwerte vor. Das Modul „*Laplacian Filter*“ ist ein Hochpassfilter für das Herausarbeiten von Kanten. Der „*Multi Directon Lee Filter*“ ermöglicht die Glättung der Datenoberfläche bei gleichzeitigem Erhalt linearer Bildelemente (s.a. SELIGE et al. 2006).

Mathematische Funktionen

Die Bibliothek **grid_calculus** versammelt Module, die einfache Berechungen wie auch komplexere mathematische Operationen durchführen. Ein Teil der Module wurde dafür entworfen, spezielle Datensätze für Testzwecke zu erstellen. Das flexibelste Modul ist der „*Grid Calculator*“, mit dem sich eine beliebige Zahl von Rasterdatensätzen über eine frei definierbare Formel zellenweise miteinander Verknüpfen lassen. Die Auswertung der Formel erfolgt durch den in der SAGA-API implementierten Formelparser (Kap. 3.3.2). Die Eingabedatensätze werden in der Formel mit Buchstaben (a, b, ..., z) bezeichnet. Auch das Modul „*Function*“ greift auf den Formelparser zurück, diesmal aber, um ein Raster anhand einer frei definierbaren Funktion *f(x, y)* zu erstellen. Einfache Datenoberflächen, wie Kegel



a) Überwachte Klassifizierung (Trainingsgebiete hellblau)

b) Cluster Analyse

Abb. 42: Klassifikation von Satellitenbilddaten.

und beliebig geneigte Ebenen, lassen sich direkt mit „*Geometric Figures*“ erstellen. Die Module „*Gaussian Landscape*“ und „*Random Terrain Generation*“ generieren Datenoberflächen mit Algorithmen der Fraktalen Geometrie (MANDELBROT 1983). Das Modul „*Random Field*“ erzeugt ein Raster, dessen Datenoberfläche ein normalverteiltes Rauschen aufweist. Mit „*Grid Normalisation*“ werden die Datenwerte eines Rasters normalisiert. D.h., die Werte werden so skaliert, dass ihr Mittelwert bei 0 und ihre Standardabweichung bei 1 liegt. Das Modul „*Grid Volume*“ berechnet das Volumen, das unter der Datenoberfläche liegt. Die Bezugsebene für die Volumenberechnung ist frei wählbar.

Diskretisierung und Klassifikation

Die Module der Bibliothek **grid_discretisation** dienen der Klassifizierung von Rasterdaten bzw. der Zuordnung von Rasterzellen zu bestimmten diskreten Einheiten. Das Modul „*Supervised Classification*“ ist eine Standardmethode der Bildverarbeitung für die überwachte Klassifikation einer beliebigen Anzahl von Rasterdatensätzen. Trainingsgebiete werden als Polygondatensätze an das Modul übergeben (Abb. 42a). Als Diskriminator-Funktion kann zwischen der *Minimalen Distanz* und der *Maximalen Wahrscheinlichkeit* gewählt werden (s.a. MCCLOY 2006). In ähnlicher Weise kann eine nicht überwachte Klassifikation, die keine Trainingsgebiete benötigt, mittels Cluster-Analyse von dem Modul „*Cluster Analysis for Grids*“ durchgeführt werden. Die Cluster-Analyse basiert auf den von FORGY (1965) und RUBIN (1967) vorgeschlagenen Algorithmen. Neben einem Raster mit der Klassenzugehörigkeit jeder Rasterzelle wird auch eine Tabelle ausgegeben, in der Klasseneigenschaften, wie die Lage der Zentroide, die Anzahl der Elemente (Rasterzellen) und die Varianz aufgelistet sind.

Während die bisher vorgestellten Klassifikationsverfahren nur die Datenwerte Zelle für Zelle analysieren, berücksichtigen die beiden folgenden Module räumliche Beziehungen

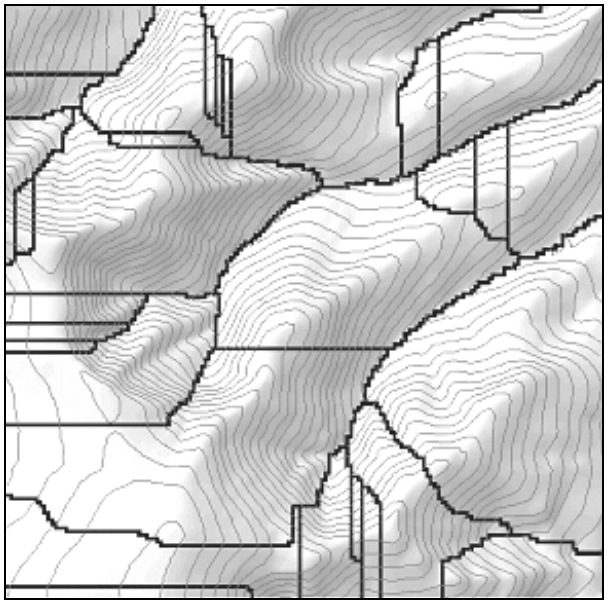


Abb. 43: Segmentierung

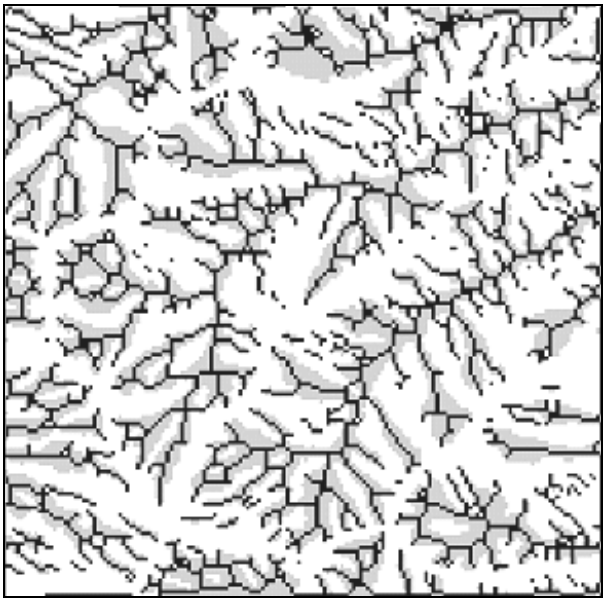


Abb. 44: Skelettierung

zwischen den Rasterzellen als Kriterium für eine Diskretisierung. Das Modul „*Grid Segmentation*“ identifiziert lokale Maxima der Datenoberfläche und ordnet danach alle anderen Rasterzellen einem dieser Maxima zu. Die Zuordnung erfolgt anhand des Wertegefälles der Datenoberfläche, indem die Zugehörigkeit zu einem Maximum jeweils an tiefergelegene Nachbarzellen weitergereicht wird. Rasterzellen, die eine Zugehörigkeit zu zwei oder mehreren Maxima aufweisen, markieren den Grenzbereich zwischen den Einflussgebieten der Maxima. Für Abb. 43 wurde das Verfahren auf Höhenwerte angewendet, so dass die Maxima Gipfelpunkten entsprechen und die Segmentgrenzen großenteils in den Gerinnen verlaufen. Das Verfahren lässt sich wahlweise auch umkehren, so dass die Minima analysiert werden. Das Modul „*Grid Skeletonisation*“ bietet drei Methoden an Flächen, die von zusammenhängenden Rasterzellen gebildet werden, auf ihre Kernbereiche zu reduzieren, so dass ein zusammenhängendes Skelett zurückbleibt (Abb. 44). Neben zwei Methoden für die *Skelettierung* nach Hilditch (NACCACHE & SHINGHAI 1984) kann auch eine Methode gewählt werden, die speziell für die Ableitung von Gerinnenetzen aus DGM entwickelt wurde (MEISEL et al. 1995). Die Methode hat jedoch ihre Grenzen und es empfiehlt sich, auf die in Kapitel 0 vorgestellten Methoden zur Bestimmung von Gerinnenetzen zurückzugreifen.

Funktionen für die Analyse

Die Bibliothek **grid_analysis** versammelt Analysefunktionen für Rasterdaten, die breit gefächerte Anwendungsgebiete haben oder aus anderen Gründen bisher keiner spezielleren Modulbibliothek zugeordnet wurden. Mit den Modulen „*Accumulated Cost (Isotropic)*“ und „*Accumulated Cost (Anisotropic)*“ lassen sich alternative Distanzmaße für die Distanzen zwischen Rasterzellen in Form sogenannter Kostenoberflächen (engl. *cost surface*) erstellen. Das interaktive Modul „*Least Cost Path*“ folgt von der gewählten Position aus dem in diesem Sinne kostengünstigsten Pfad, der durch die Kostenoberfläche bestimmt wird und mehr oder

weniger von der euklidischen Distanz abweicht (MCCLOY 2006). Zwei Module dienen zur Berechnung verschiedener Vegetationsindizes. Alle von den Modulen „*Vegetation Index*“ berechneten Indizes basieren auf Messungen des roten sowie des nahen infraroten Spektralbereichs, wie sie z.B. von satellitengestützten Multispektralsensoren als Bilddateien geliefert werden (MCCLOY 2006). Zu den berechneten Indizes gehören der Ratio Vegetation Index (RVI), der Normalised Difference Vegetation Index (NDVI) und der Perpendicular Vegetation Index (PVI). Das Modul „*Change Vector Analysis*“ analysiert die zeitliche Veränderung in zwei Spektralbereichen. Die Module „*Fuzzy intersection grid*“, „*Fuzzy union grid*“ und „*Fuzzify*“ erlauben das Arbeiten mit unscharfen (engl. *fuzzy*) Mengen (s.a. BANDEMER & GOTTWALD 1993). Das Modul „*Pattern Analysis*“ ist auf klassifizierte sowie auf Bilddaten ausgerichtet und benutzt für jede Rasterzelle ein auf sie zentriertes quadratisches Suchfenster, um die Diversität der auftretenden Werteklassen zu beschreiben. Das Modul „*Layer of Extreme Value*“ findet für jede Rasterzelle aus einer beliebigen Anzahl von Rasterdatensätzen denjenigen, der den maximalen bzw. minimalen Wert enthält. Auf die weiteren Module der Bibliothek wird hier nicht eingegangen, da sie sich noch in der Entwicklung befinden bzw. ihre Aufgabe und Funktionsweise von ihren Autoren noch nicht dokumentiert wurde.

Visualisierung

Die Module, die der Bibliothek **grid_visualisation** zugeordnet sind, bieten besondere Visualisierungsmöglichkeiten für Rasterdaten an. Einige der Module greifen dabei auf spezielle Darstellungsmöglichkeiten der SAGA-GUI zurück. Das Modul „*RGB Composite*“ erstellt einen farbcodierten Rasterdatensatz, dessen Datenwerte sich von der SAGA-GUI als Echtfarben interpretieren lassen (s. Kap. 3.4.3). Als Eingabe wird für den roten, grünen und blauen Kanal jeweils ein Rasterdatensatz benötigt. Optional kann mit einem vierten Datensatz die Transparenz für jede Rasterzelle bestimmt werden. Das Modul „*Fit Color Palette to Grid Values*“ manipuliert die von der Benutzeroberfläche für den Rasterdatensatz gesetzte Farbpalette derart, dass die Farben gleichmäßig über die Datenwerte verteilt werden. Mit „*Color Palette Rotation*“ lassen sich die Farben der Farbpalette zyklisch vertauschen, wobei der Farbwechsel von der SAGA-GUI unmittelbar dargestellt wird. Die Farben werden solange getauscht, bis der Benutzer die Ausführung des Moduls abbricht. In ähnlicher Weise funktioniert das Modul „*Color Blending*“, das eine visuelle Überblendung zwischen mehreren Rasterdatensätzen ermöglicht und so z.B. zur Veranschaulichung zeitlicher Abläufe geeignet ist. Das Modul „*Create 3D Image*“ erstellt aus Höhendatensätzen 3D-Panoramen und bietet einige Zusatzfunktionen gegenüber der in der SAGA-GUI verankerten 3D-Visualisierung. Die Ausgabe erfolgt in einen farbcodierten Rasterdatensatz.

3.6.7 Geostatistik

Statistische Verfahren sind von grundlegender Bedeutung für die Datenanalyse. Alle hier vorgestellten Verfahren ermitteln räumlich determinierte, statistische Zusammenhänge. Im engeren Sinne werden mit geostatistischen Verfahren aber vor allem solche bezeichnet, die die räumliche Autokorrelation von Daten analysieren und für die flächenhafte Prognose von Werten benutzen.

Statistische Analysen für Rasterdaten

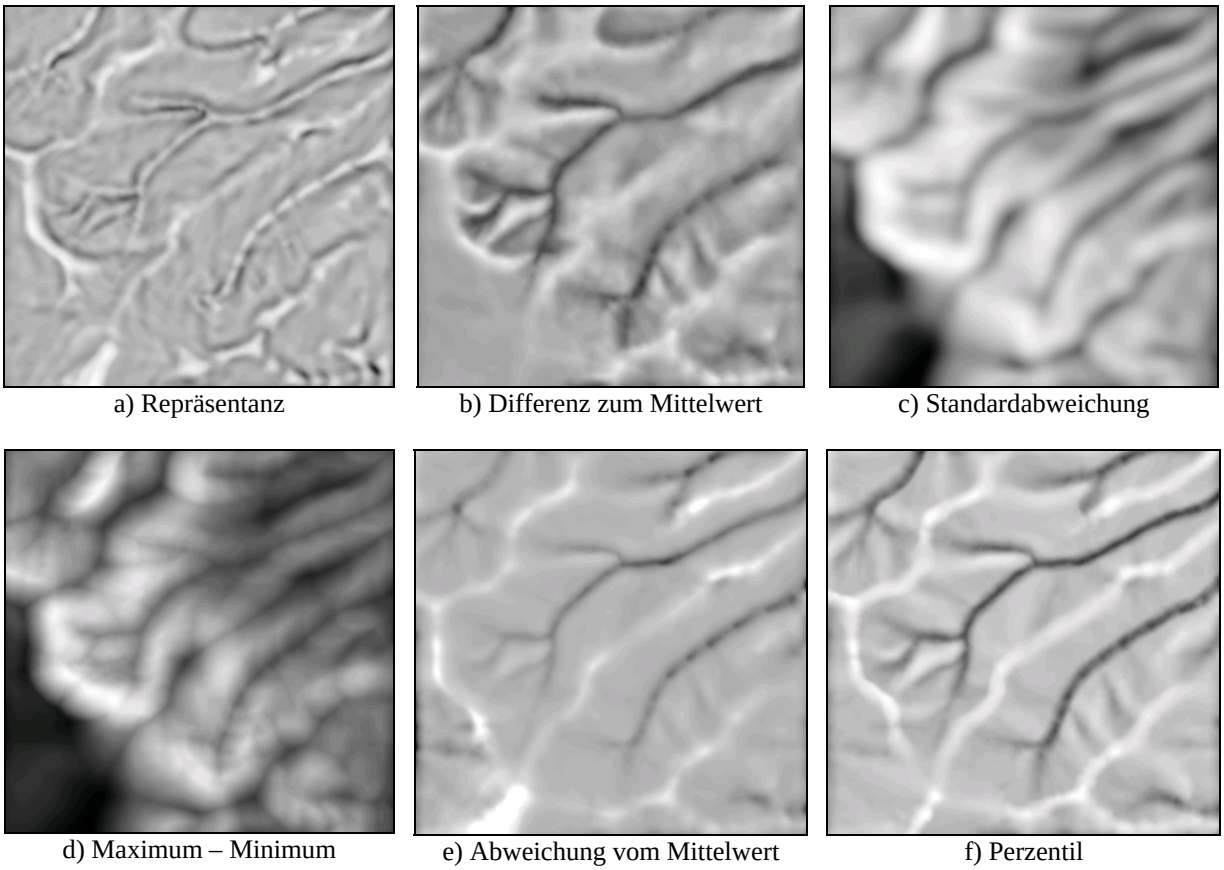
Die Bibliothek **geostatistics_grid** vereint Module für die statistische Analyse von Rasterdaten. Das Modul „*Statistics for Grids*“ bestimmt zellenweise statistische Kennwerte wie Mittelwert, Minimum und Maximum, Varianz und Standardabweichung für die Daten einer beliebigen Anzahl von Rasterdatensätzen. Für jeden Kennwert wird ein eigener Rasterdatensatz ausgegeben. In ähnlicher Weise ermittelt das Modul „*Zonal Grid Statistics*“ statistische Kennwerte für Zonen bzw. Gebiete, die durch die Datenwerte eines zusätzliches Eingaberasters definiert werden. Ergebnis ist eine Tabelle mit den Kennwerten für jede Zone.

Die nächsten drei Module untersuchen die statistischen Zusammenhänge zwischen den Rasterzellen und ihren jeweiligen Nachbarschaften. Die Nachbarschaft wird dabei durch einen vom Benutzer zu wählenden Suchradius definiert. Mit „*Residual Analysis*“ wird der Wert der zentralen Rasterzelle direkt in Beziehung zu seiner Nachbarschaft gesetzt (WILSON & GALLANT 2000). Neben dem Mittelwert gehören zu den berechneten Kennwerten die Differenz zum Mittelwert, die Standardabweichung, der Wertebereich (Differenz zwischen Maximum und Minimum), die Abweichung vom Mittelwert und das Perzentil der zentralen

Tab. 31: Module für geostatistische Verfahren

Name/Bibliothek	**Beschreibung**	**Module**
Geostatistics – Grids **geostatistics_grid**	Statistische Analysen für Rasterdaten	• Residual Analysis (Grid) • Representativeness (Grid) • Radius of Variance (Grid) • Regression Analysis (Grid/Points) • Multiple Regression Analysis (Grids/Points) • Statistics for Grids • Zonal Grid Statistics[VW]
Geostatistics – Points **geostatistics_points**	Geostatistische Analysen von Punktdaten	• Semivariogram
Geostatistics – Kriging **geostatistics_kriging**	Kriging – geostatistische Prognoseverfahren	• Ordinary Kriging • Ordinary Kriging (Global) • Universal Kriging • Universal Kriging (Global)

[VW] V.Wichmann



a) Repräsentanz b) Differenz zum Mittelwert c) Standardabweichung

d) Maximum – Minimum e) Abweichung vom Mittelwert f) Perzentil

Abb. 45: Statistische Kennwerte (Höhe, Suchradius: 7 Rasterzellen)

Rasterzelle (Abb. 45b-f). Das Modul „*Representativeness*“ benutzt die Varianzanalyse, um ein Maß dafür zu erhalten, wie repräsentativ eine Rasterzelle für ihre Umgebung ist (Abb. 45a, BÖHNER et al. 1997). Dieses Verfahren wird von „*Radius of Variance*“ gewissermaßen umgekehrt, indem hier bestimmt wird, für welche kreisförmige Nachbarschaft eine benutzerdefinierte Repräsentanz nicht überschritten wird.

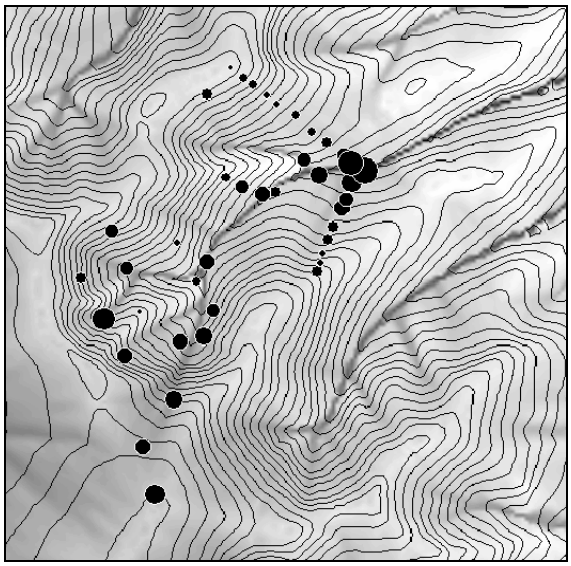
Aufgabe der Regressionsanalyse ist es den statistischen Zusammenhang zwischen Datensätzen mathematisch zu beschreiben. Das Modul „*Regression Analysis (Grid/Points)*“ für die Einfachregression mit verschiedenen Regressionsfunktionen analysiert die Korrelation von Punktdaten mit den räumlich zusammenfallenden Werten eines Rasterdatensatzes. Regressions- und Korrelationskoeffizienten werden ebenso ausgegeben wie die Anwendung der Regressionsfunktion auf das gesamte Raster. Die Residuen der Punktdaten werden optional in einem neuen Punktdatensatz gespeichert. In ähnlicher Weise führt das Modul „*Multiple Regression Analysis*“ eine lineare Regressionsanalyse durch, die jedoch den Zusammenhang mit einer beliebigen Anzahl von Rasterdatensätzen analysiert (Abb. 46a). Beide Module verwenden intern die von der SAGA-API bereitgestellten Funktionalitäten zur Regressionsanalyse (s. Kap. 3.3.2).

Semivariogramme

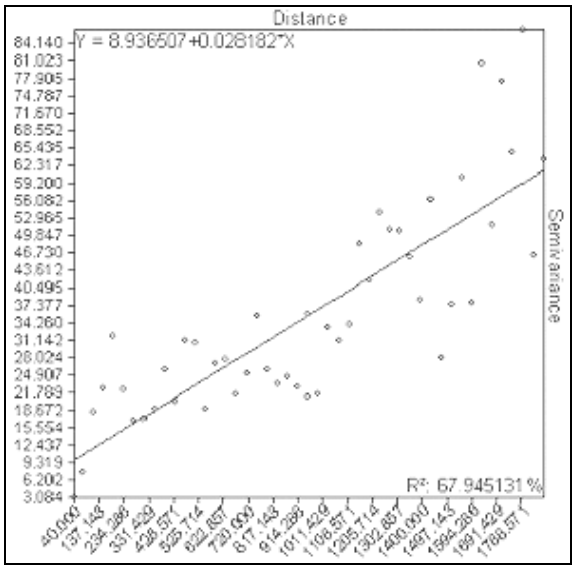
Das einzige Modul der Bibliothek **geostatistics_points** analysiert die räumliche Autokorrelation von Punktdaten und schreibt das Ergebnis in eine Tabelle, deren Daten sich dann als Semivariogramm darstellen lassen (Abb. 46b). Nach dem Anpassen einer Trendfunktion an das Semivariogramm kann diese Funktion zusammen mit den Punktdaten für die flächenhafte Datenprognose mit den geostatistischen Kriging-Verfahren benutzt werden.

Kriging

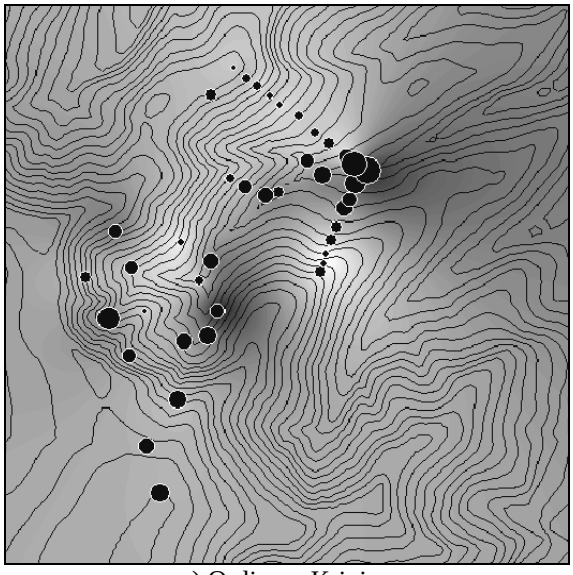
Kriging Verfahren dienen der räumlichen Dateninterpolation und basieren auf dem statistischen, distanzabhängigen Verhältnis der Eingangsdaten zueinander, wie es durch ein Semivariogramm dargestellt wird. Das Verfahren beruht auf der Annahme, dass, statistisch gesehen, dicht beieinanderliegende Punkte ähnlichere Werte haben als weiter voneinander



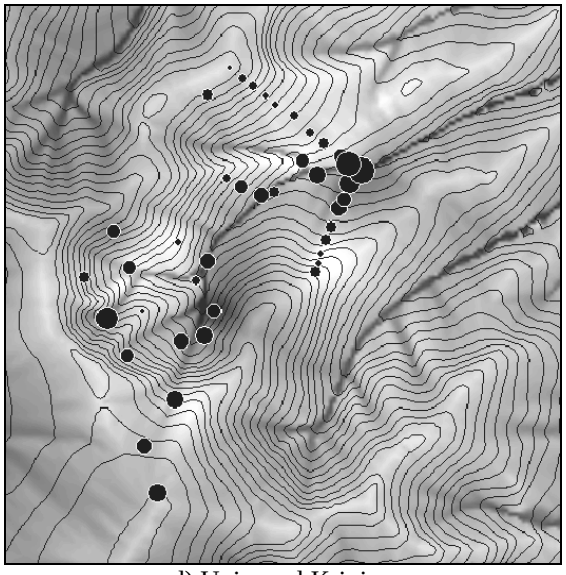
a) Multiple Regressionsanalyse



b) Semivariogramm



c) Ordinary Kriging



d) Universal Kriging

Abb. 46: (Geo-)statistische Verfahren für die flächenhafte Schätzung von Werten.

entfernte. Das Semivariogramm dient zur Analyse dieses Autokorrelation genannten Zusammenhangs, der in mathematisch-funktionaler Form direkt in das Kriging-Verfahren einfließt. Die Bibliothek **geostatistics_kriging** bietet zwei Kriging-Varianten an, die als Eingabe einen Punktdatensatz erwarten, für den eines seiner Attribute flächenhaft geschätzt werden soll. Ausgabe sind zwei Rasterdatensätze, einer mit den geschätzten Werten und einer, der die Qualität der Schätzung beschreibt. Das Ausgaberaster ist vom Benutzer frei wählbar. Beide Verfahren haben jeweils eine Variante, die mit einem eingeschränkten Suchradius arbeitet, und eine, die für jede Schätzung den kompletten Punktdatensatz verwendet (*global*). Die globalen Verfahren sind gut geeignet für kleine Eingabedatensätze. Für große Punktdatenmengen empfiehlt es sich jedoch den Suchradius bzw. die Zahl maximal zu analysierender Datenpunkte sinnvoll einzuschränken. Das Ordinary Kriging ist das einfachste der hier versammelten Kriging-Verfahren, und analysiert nur die Punktdaten selbst (Abb. 46c). Mit Universal Kriging wird zusätzlich eine beliebige Anzahl potentiell erklärender Rasterdatensätze in die Punktdatenanalyse einbezogen und bei der Schätzung berücksichtigt (Abb. 46d). Für alle Module wird optional auch Block-Kriging unterstützt, bei dem jede Schätzung nicht punktuell, sondern auf eine Fläche bezogen vorgenommen wird.

3.6.8 Reliefanalyse

Die Reliefanalyse auf Basis Digitaler Geländemodelle (DGM) ist gewissermaßen der Ausgangspunkt für die Entwicklung von SAGA gewesen (s. Kap.1.1), was sich auch in der Zahl vorhandener Module zu dieser Thematik widerspiegelt. Ein großer Teil der Verfahren wurde bereits in dem SAGA-Vorläufer DiGeM implementiert und von CONRAD (1998) ausführlich beschrieben. Einen Überblick über die Reliefanalyse mit SAGA geben OLAYA & CONRAD (2006). Standardverfahren für die Berechnung von Reliefparametern, wie Hangneigung, Exposition und Einzugsgebietsgrößen, werden durch allgemeine Module ergänzt, die besonders auf eine Verwendung mit DGM zugeschnitten sind, wie z.B. der Erstellung von Geländeprofilen. Daneben repräsentieren viele der vorgestellten Module die aktuelle Forschung der AG Geosystemanalyse (s. Kap. 1.1) und sind in dieser Form zur Zeit nur in SAGA ausführbar.

Tab. 32: Module für die Reliefanalyse

Name/Bibliothek	Beschreibung	Module
Morphometry *ta_morphometry*	Morphometrie	• Convergence Index • Convergence Index (Search Radius) • Curvature Classification • Hypsometry • Local Morphometry • Real Area Calculation[VO] • Morphometric Protection Index[VO] • Surface Specific Points

Tab. 32: Module für die Reliefanalyse (Fortsetzung)

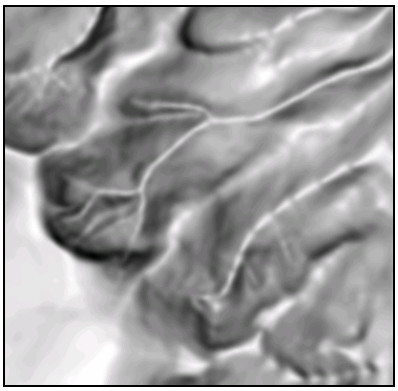
Name/Bibliothek	Beschreibung	Module
Lighting, Visibility *ta_lighting*	Beleuchtung und Sichtbarkeit	• Analytical Hillshading • Solar Radiation • Solar Radiation (B) • Visibility (single point) [interactive]
Preprocessing *ta_preprocessor*	Präprozessierung	• Sink Drainage Route Detection • Sink Removal • Fill Sinks (Planchon/Darboux, 2001)[VW]
Hydrology *ta_hydrology*	Hydrologische Analysen und Modelle	• Parallel Processing • Recursive Upward Processing • Flow Tracing • Upslope Area • Upslope Area [interactive] • Downslope Area [interactive] • Flow Path Length • Slope Length[VO] • Topographic Indices • SAGA Wetness Index • Isochrones Constant Speed [interactive][VO] • Isochrones Variable Speed [interactive][VO] • Cell Balance[VO] • Flow Sinuosity [interactive][VO] • Flow Depth [interactive][VO] • Edge Contamination[VO]
Channels *ta_channels*	Gewässernetze und Einzugsgebiete	• Channel Network • Watershed Basins • Watershed Basins (extended)[VO] • Vertical Distance to Channel Network • Overland Flow Distance to Channel Network • D8 Flow Analysis • Strahler Order[VO]
Profiles *ta_profiles*	Profile	• Profile [interactive] • Flow Path Profile [interactive] • Swath Profile [interactive] • Profile from points[VO] • Cross Sections[VO]

[VO]V.Olaya, [VW] V.Wichmann

Morphometrie

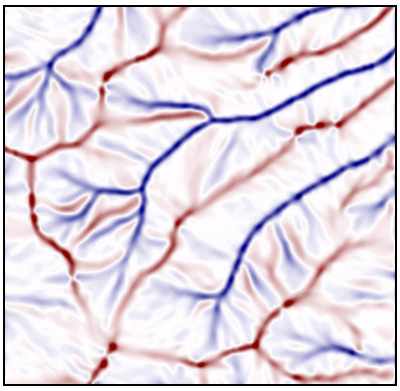
Die Morphometrie des Reliefs oder Geomorphometrie hat die Quantifizierung der Form der Geländeoberfläche zur Aufgabe und stützt sich heutzutage nahezu ausschließlich auf die Analyse rasterbasierter DGM. Die Ableitung von Reliefparametern ist wiederum die Grundlage für eine Parametrisierung reliefgesteuerter Prozesse und somit nicht zuletzt auch für eine Vielzahl geoökologischer Fragestellungen relevant. Die Bibliothek **ta_morphometry** vereinigt Module, deren Methoden auf der direkten, meist auf die unmittelbare Nachbarschaft einer Rasterzelle beschränkten Auswertung der Höhenwerte beruhen.

Das Modul „*Local Morphometry*“ berechnet die Reliefparameter Hangneigung (Abb. 47a) und Exposition (Abb. 47b) sowie die Tangential-, Horizontal- und Vertikalwölbung (Abb. 47d, Abb. 47e), wobei verschiedene Methoden zu ihrer Berechnung zur Auswahl stehen (z.B. HEERDEGEN & BERAN 1982, ZEVENBERGEN & THORNE 1987, TARBOTON 1997). Eine ähnliche Aussage, wie die Horizontalwölbung, liefert der mit „*Convergence Index*“ berechenbare Konvergenz-Index (Abb. 47c, KÖTHE & LEHMEIER 1996). Eine Variante des originalen Algorithmus analysiert nicht nur die unmittelbare Nachbarschaft einer Rasterzelle, sondern alle Rasterzellen innerhalb eines frei wählbaren Suchradius. Datensätze mit Horizontal- und Vertikalwölbungen können für die wölbungsbasierte Reliefklassifikation an das Modul „*Curvature Classification*“ nach DIKAU (1988) übergeben werden (Abb. 47f). Das Modul „*Surface Specific Points*“ vereinigt einige Methoden aus der Pionierzeit der Digitalen Relief-

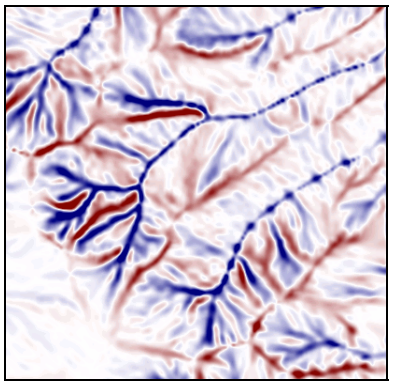


a) Hangneigung

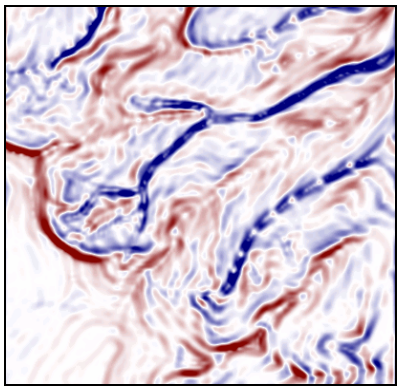
b) Exposition



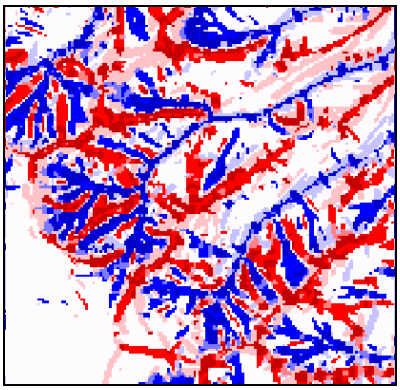
c) Konvergenz Index



d) Horizontalwölbung

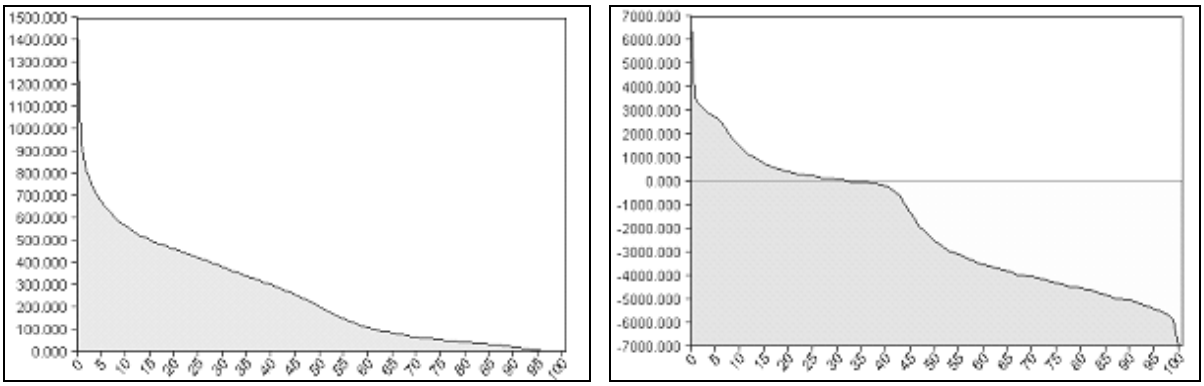


e) Vertikalwölbung



f) Wöbungsklassfikation

Abb. 47: Lokale Morphometrie.



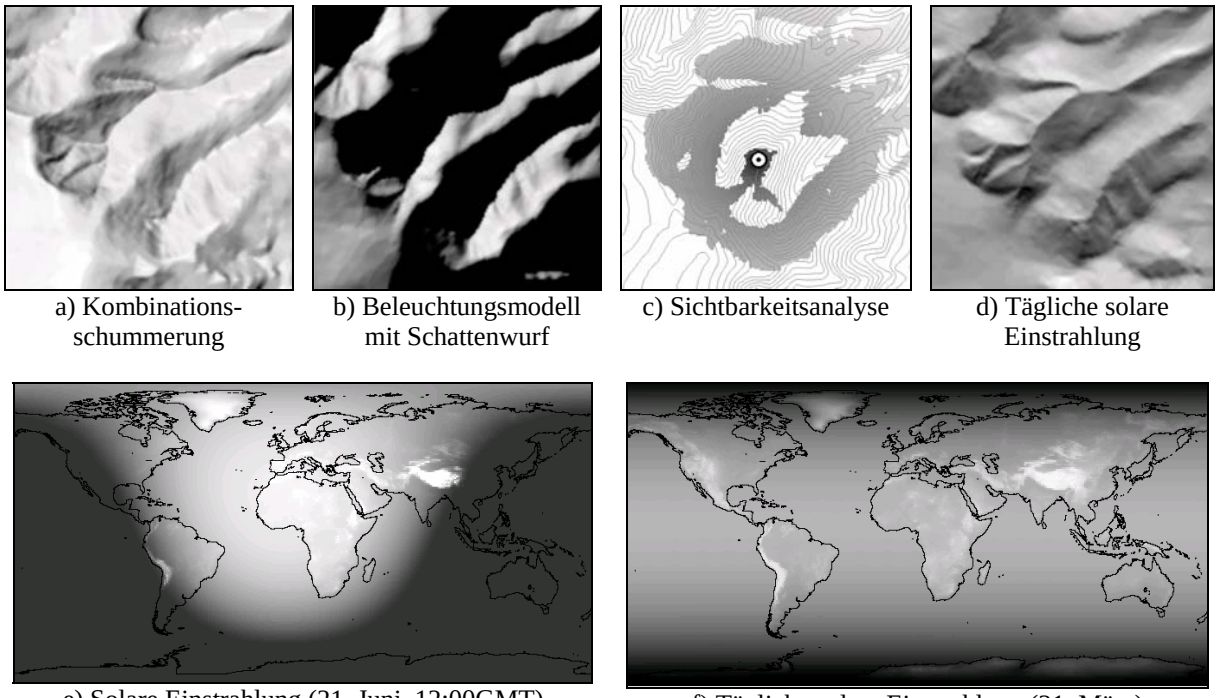
a) BRD b) Erde

Abb. 48: Hypsometrische Kurven.

analyse und dient zur Identifikation hervorstechender Reliefpositionen, wie Gipfel, Grate und Gerinne (s.a Kap. 3.6.5). Hypsometrische Kurven zeigen die Häufigkeitsverteilung der Höhenwerte eines DGM (Abb. 48). Der Kurvenverlauf kann u.a. Hinweise auf die Formungsprozesse kleinerer Einzugsgebiete geben (z.B. LUO 2000). Das Modul „*Real Area Calculation*“ berechnet aus der Grundfläche und den Höhenwerten eines DGM die tatsächliche Oberfläche des Geländes. Mit „*Morphometric Protection Index*“ wird ein Index erstellt, der ein Maß für die Reliefenergie innerhalb einer benutzerdefinierten Nachbarschaft für jede Rasterzelle liefert.

Beleuchtung und Sichtbarkeit

Die Module der Bibliothek **ta_lighting** analysieren den Einfluss der Geländeoberfläche auf die Ausbreitung von Licht. Das Modul „*Analytical Hillshading*“ erstellt Beleuchtungsmodelle, die in erster Linie für die Reliefschummerung benutzt werden. Die Standardmethode ermittelt einfach den Winkel, unter dem Licht, das aus einer gegebenen Richtung einfällt, von dem Gelände reflektiert wird. Um den Kontrast zwischen bewegtem und flachem Gelände zu betonen, kann dieser Wert mit der Hangneigung gewichtet werden (Abb. 49a). Eine weitere Methode verfolgt auch den Schattenwurf, der von jeder Rasterzelle ausgeht (Abb. 49b). Diese Methode wird von den Modulen von „*Incoming Solar Radiation*“ bzw. „*Insolation*“ für die Berechnung der potentiell auf die Geländeoberfläche einfallenden Solarstrahlung benutzt. Die Berechnung von Strahlungssummen für benutzerdefinierte Zeiträume erfolgt in diskreten Zeitschritten, für die ein Beleuchtungsmodell des jeweils aktuellen Sonnenstands berechnet wird. Aus dem Beleuchtungsmodell lässt sich dann direkt die einfallende Solarstrahlung bilanzieren (Abb. 49d). Der Einfluss der Atmosphäre kann mit verschiedenen Optionen berücksichtigt werden. Das zweite Modul „*Insolation*“, das auf dem in SADO (s. Kap. 1.1) implementierten Verfahren zurückgeht (BÖHNER & PÖRTGE 1997), bietet hier weitergehende Möglichkeiten. Neben der Abschätzung des diffusen Strahlungsanteils, ist vor allem die Möglichkeit hervorzuheben, die Krümmung der Erdoberfläche für großräumige Untersuchungsgebiete berücksichtigen zu können (Abb. 49e, f).

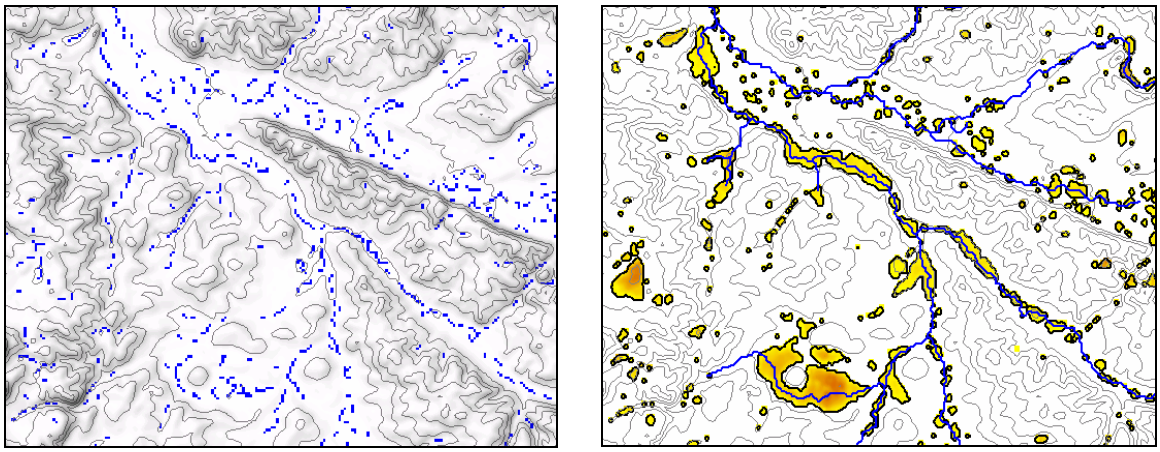


a) Kombinations-schummerung
b) Beleuchtungsmodell mit Schattenwurf
c) Sichtbarkeitsanalyse
d) Tägliche solare Einstrahlung
e) Solare Einstrahlung (21. Juni, 12:00GMT)
f) Tägliche solare Einstrahlung (21. März)

Abb. 49: Beleuchtung u. Sichtbarkeit.

Präprozessierung für hydrologische Analysen

U.a. durch Generalisierungseffekte bedingt, treten bei rasterbasierten DGM besonders in Senkenbereichen häufig Rasterzellen auf, die nur von höheren Nachbarzellen umgeben sind. Für die in den nachfolgenden Kapiteln vorgestellten Methoden zur Ableitung von Reliefparametern, die durch den Oberflächenabfluss definiert sind, stellen solche Rasterzellen ein Hindernis bzw. eine Senke dar, da keine niedrigere Reliefposition für die Abflussdurchleitung ermittelt werden kann. Damit diese Methoden Abfluss durch solche artifiziellen Senken leiten können, müssen die Senken vorher entsprechend prozessiert bzw. entfernt werden. Das Modul „*Sink Drainage Route*“ analysiert jede Senke und sucht dabei den niedrigsten Überlauf für den zugehörigen Senkenbereich, wobei auch ineinander geschachtelte Senkenbereiche berücksichtigt werden. Die Senken werden in Abhängigkeit von der Geländeoberfläche auf dem kürzesten Weg mit ihren Überläufen durch Entwässerungswege verbunden (Abb. 50a). Die Entwässerungswege werden in einem Raster gespeichert, wobei die Rasterzellenwerte angeben, in welche Nachbarzelle der Abfluss weitergeben werden soll. Einige abflussbasierte Module akzeptieren diese Entwässerungswege als Eingabe. Für allgemeine Anwendungen ist es jedoch einfacher, das DGM so zu verändern, dass es keine Senken mehr enthält. Das Modul „*Sink Removal*“ benutzt das zuvor beschriebene Verfahren, um Senken zu entfernen, entweder durch Auffüllen des Senkenbereiches, so dass ein minimales Gefälle in Richtung des Überlaufes besteht (Abb. 50b), oder durch Eintiefen der Entwässerungswege, ebenfalls unter Einhaltung eines minimalen Gefälles. Alternativ steht mit „*Fill Sinks*“ eine weitere, sehr effektive Methode zum Auffüllen von Senkenbereichen zur Verfügung, bei dem die Geländeoberfläche zunächst mit einer dicken Wasserschicht überzogen wird, die dann



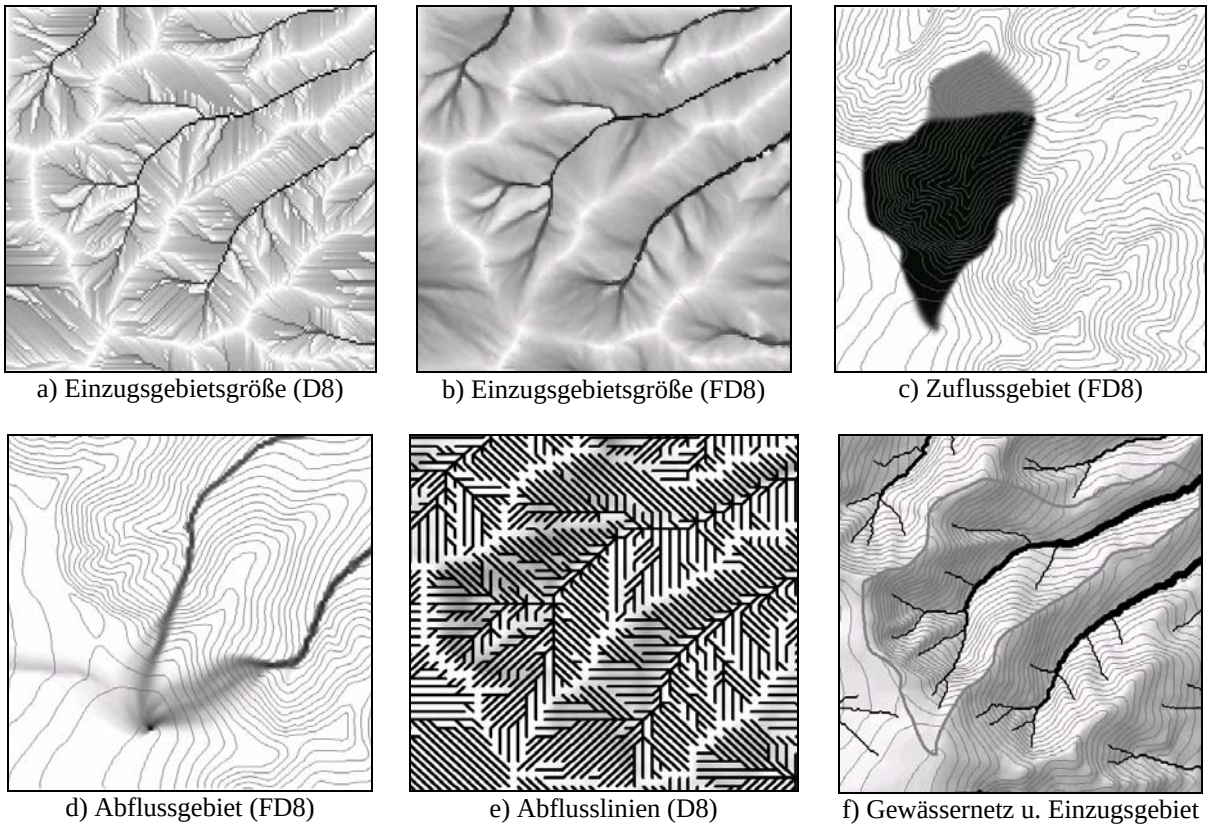
a) Routen zur Entwässerung von Senken b) Aufgefüllte Senken und Gewässernetz

Abb. 50: Analyse u. Entfernung von abflusslosen Senken.

sukzessive unter der Prämisse entfernt wird, dass für ihre Oberfläche immer eine Abflussrichtung bestimmt werden kann (PLANCHON & DARBOUX 2001).

Hydrologische Analysen und Modelle

Die Module der Bibliothek **ta_hydrology** leiten Reliefparameter ab, die durch den Oberflächenabfluss bestimmt sind. Der prominenteste Reliefparameter dieser Kategorie ist die Einzugsgebietsgröße, die von den Modulen „*Parallel Processing*“, „*Recursive Upward Processing*“ und „*Flow Tracing*“ mit sehr verschiedenen, in der Literatur beschriebenen Methoden berechnet werden können, u.a. mit Deterministic 8 oder D8 (O'CALLAGHAN & MARK 1984), Rho 8 (FAIRFIELD & LEYMARIE 1991), dem Braunschweiger Reliefmodell (BAUER et al. 1985), Deterministic Infinity (TARBOTON 1997), Multiple Flow Direction oder FD8 (QUINN et al. 1991, FREEMAN 1991), Kinematic Routing Algorithm (LEA 1992) und DEMON (COSTA-CABRAL & BURGES 1994). Wegen ihrer einfachen Programmierbarkeit einerseits und ihrer guten Aussagekraft andererseits werden D8 und FD8 auch von verschiedenen anderen Modulen für die Abbildung von Abflussprozessen benutzt, wobei D8 (Abb. 51a) besser für die Modellierung linearen und FD8 (Abb. 51b) für die flächenhaften Abflusses geeignet ist. Alle drei Module bieten ähnliche Zusatzfunktionen an. So können die Mittelwerte u.a. der Höhe und Hangneigung für das Einzugsgebiet jeder Rasterzelle ermittelt werden. Es kann eine Einzugsgebietgröße als Schwellenwert festgelegt werden, oberhalb der Abfluss immer mit der linear mit D8 weitergeleitet wird. Auch lassen sich die Abflussmengen mit den Werten eines optionalen Eingaberasters gewichten. Durch eine entsprechende Gewichtung kann man die Abfluss liefernden Gebiete gezielt einschränken und so den Einflussbereich ausgewählter Rasterzellen bestimmen. Das Modul „*Upslope Area*“ bestimmt in umgekehrter Weise das Einzugsgebiet für ein frei wählbares Zielgebiet. Die interaktiven Module „*Upslope Area*“ und „*Downslope Area*“ bestimmen die Einzugs- bzw. Abflussgebiete für einzelne Rasterzellen ebenfalls mit verschiedenen Abflussverteilungsmodellen (Abb. 51c, d). Die Module „*Flow Path Length*“ und „*Slope Length*“ berechnen die maximale Fließweglänge, die Oberflächenabfluss bis zu jeder Rasterzelle zurücklegt. Das erste Modul erlaubt die



a) Einzugsgebietsgröße (D8) b) Einzugsgebietsgröße (FD8) c) Zuflussgebiet (FD8)

d) Abflussgebiet (FD8) e) Abflusslinien (D8) f) Gewässernetz u. Einzugsgebiet

Abb. 51: Hydrologische Reliefparameter u. -gliederungen.

Angabe zusätzlicher Ausgangspunkte und kann auch FD8 als Alternative zu D8 verwenden. Das Modul „*Cell Balance*“ benutzt die Methoden D8 bzw. FD8, um die Zufluss- und Abflussmengen einer Rasterzelle bezogen auf ihre unmittelbare Nachbarschaft zu bilanzieren.

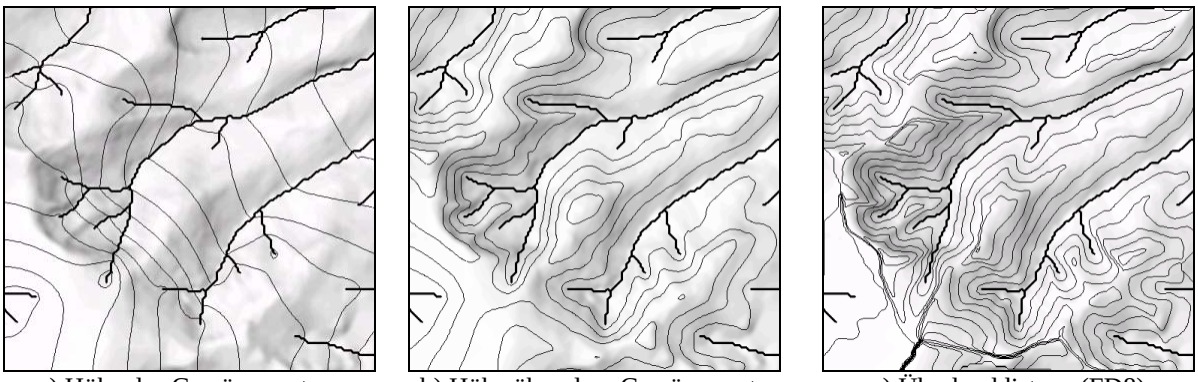
Die topographischen Indizes, die von dem gleichnamigen Modul („*Topographic Indices*“) berechnet werden, setzen die Hangneigung einer Rasterzelle ins Verhältnis zu ihrer Einzugsgebietsgröße. Hierbei wird für die Berechnung des für den Reliefeinfluss verantwortlichen LS-Faktors der Universal Soil Loss Equation (USLE) die Hanglänge durch die Einzugsgebietsgröße substituiert (MOORE et al. 1992). Der Topographische Bodenfeuchte-Index (TWI) gilt als Maß für die reliefbedingte Bodenfeuchte und ist auch die Basis für das hydrologische Modellkonzept TOPMODEL (s.a. Kap. 3.6.9, BEVEN et al. 1984). Der SAGA-Bodenfeuchte-Index, erstellbar mit dem Modul „*SAGA Wetness Index*“, greift auf das gleiche Konzept zurück, verwendet aber ein modifiziertes Verfahren zur Bestimmung der Einzugsgebietsgrößen, bei dem der Abfluss nicht als sehr dünner Wasserfilm behandelt wird. Als Ergebnis erhalten Rasterzellen mit geringer vertikaler Distanz zu einem Gerinne, auch wenn sie eine größere horizontale Distanz zu diesem aufweisen, einen vergleichsweise höheren und realistischeren Bodenfeuchte-Index zugewiesen (BÖHNER 2006).

Die Begrenzungen eines Rasters sind immer rechteckig und orientieren sich nicht an Einzugsgebietsgrenzen. Das Modul „*Edge Contamination*“ ist hier hilfreich, um alle Rasterzellen zu identifizieren, die von den Rastergrenzen her beeinflusst werden. Die Zeit, die Abfluss braucht, um zu einer Rasterzelle zu gelangen, kann mit den interaktiven Modulen

„*Isochrones Constant Speed*“ und „*Isochrones Variable Speed*“ geschätzt werden, wobei das zweite Modul dem Höhenmodell eine Reihe weiterer Angaben ermöglicht, u.a. der Oberflächenrauhigkeit.

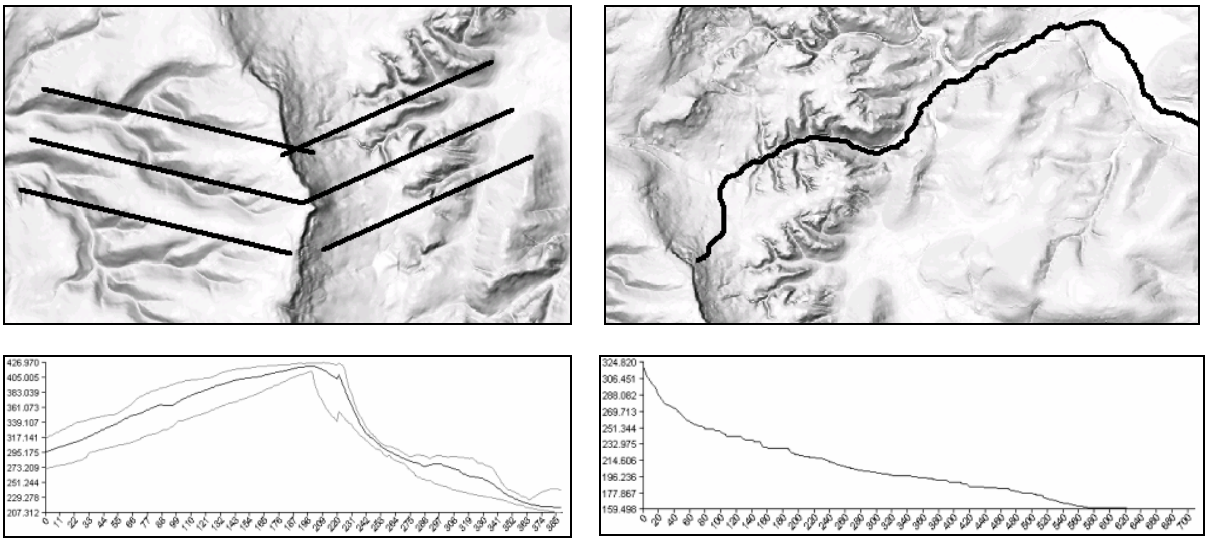
Gewässernetze und Einzugsgebiete

Eine weitere Kategorie abflussbasierter, reliefanalytischer Methoden beschäftigt sich mit der Identifikation von Gewässer- bzw. Gerinnenetzen sowie der Gliederung in Einzugsgebiete. Die Module verwenden hierfür ausschließlich das Abflussmodell Deterministic 8, da dieses im Gegensatz zu Modellen, die den Abfluss auf mehrere tiefergelegene Zellen verteilen, immer eindeutig determinierte Abflusszuweisungen vornimmt. Das einfachste Verfahren, „*D8 Flow Analysis*“, erstellt ein Raster sowie einen Liniendatensatz mit den durch D8 gegebenen Abflussrichtungen (Abb. 51e). Ein weiterer Rasterdatensatz gibt mit der Verbindungszahl an, wie viele der Nachbarzellen auf jede Rasterzelle gerichtet sind. Das Modul „*Strahler Order*“ geht etwas weiter und bestimmt, ausgehend von der Anzahl solcher Verbindungen, für jede Rasterzelle die Flussordnungszahl nach STRAHLER (1952). Am flexibelsten kann das Modul „*Channel Network*“ für die Ableitung von Gewässernetzen eingesetzt werden. Durch entsprechende Parameter lassen sich mit dem Modul u.a. die Dichte des Gewässernetzes und die minimale Länge seiner Segmente steuern. Die Ausgabe erfolgt als Liniendatensatz wie auch als Raster, in dem Verbindungen von Gewässernetzsegmenten (Mündungen) durch spezielle Werte markiert sind. Diese Werte können direkt an das Modul „*Watershed Basins*“ übergeben werden, um automatisch für jede Mündung das zugehörige Einzugsgebiet zu ermitteln (Abb. 51f). Das Modul „*Watershed Basins (extended)*“ erweitert diese einfache Einzugsgebietsermittlung um weitere Optionen und erstellt zusätzlich Polygone für jedes Teileinzugsgebiet. Zwei Module berechnen Distanzen zu einem Gewässernetz. Die Abflussdistanzen des Moduls „*Overland Flow Distance*“ basieren auf mit D8 oder FD8 berechneten Fließweglängen (Abb. 52c). Das Modul „*Vertical Distance*“ interpoliert zunächst die Höhe des Gewässernetzes (Abb. 52a) und subtrahiert diese dann von der Geländehöhe, um die Höhe über dem Gewässernetz (Abb. 52b) zu berechnen. Diese kann als Maß für die Grundwassernähe angenommen werden (s.a. ETZRODT et al. 2002, BÖHNER & KÖTHE 2003).



a) Höhe des Gewässernetzes b) Höhe über dem Gewässernetz c) Überlanddistanz (FD8)

Abb. 52: Distanz zum Gewässernetz (Isoliniendarstellung).



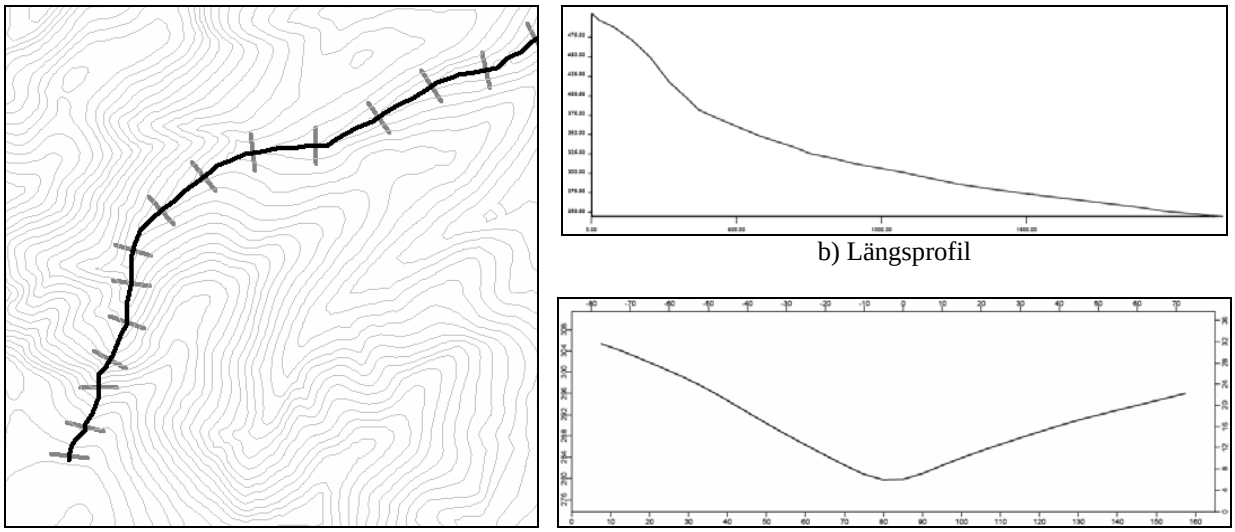
a) Streifenprofil b) Fließwegprofil

Abb. 53: Geländeprofile.

Geländeprofile

Die Bibliothek **ta_profile** versammelt Module zum Erstellen von Geländeprofilen. Ein Profil wird durch zwei oder mehr aufeinanderfolgende Koordinatenpunkte definiert, die je nach Modul entweder interaktiv gesetzt oder aber als Punkdatensatz übergeben werden. Die Ausgabe erfolgt entweder in tabellarischer Form oder als Punktdatensatz. Einfache Profile werden mit dem interaktiv arbeitenden Modul „*Profile*“ erstellt. Die Punkte des resultierenden Profils werden im Abstand der Rasterzellen gesetzt. Das gleiche Ergebnis erzielt man mit „*Profile from Points*“ für einen bereits vorliegenden Punktdatensatz. Mit „*Swath Profile*“ werden nicht nur die Werte, die auf der Profillinie liegen, berücksichtigt, sondern alle Werte die sich innerhalb eines benutzerdefinierten Streifens um die Profillinie befinden. Die Ausgabe enthält, neben der Höhe auf der Profillinie, Angaben über Minimum, Maximum, Mittelwert und Standardabweichung der im Streifen liegenden Rasterzellen (Abb. 53a). Fließwegprofile, die den auf D8 basierenden Ablussrichtungen folgen, werden von „*Flow Path Profile*“ ausgegeben (Abb. 53b).

Zwei Module ermöglichen die automatische Erstellung von Querprofilen in einer frei wählbaren Frequenz für die Linien eines Liniendatensatzes. Die Ausgabe erfolgt ebenfalls als Liniendatensatz, wobei die Stützstellen der Querprofile in der zugehörigen Attributtabelle und optional auch in einem PDF-Dokument gespeichert werden. Einfache Querprofile werden mit „*Cross Profiles*“ erstellt. Abb. 54 zeigt ein Beispiel für ein Fliesswegprofil, dessen Querprofile z.B. für die Simulation der Abflussdynamik benutzt werden können. Das Modul „*Cross Sections*“ wurde speziell für die Straßenbauplanung entwickelt und ermöglicht eine Abschätzung der bei einer Straßenbaumaßnahme anfallenden Erdarbeiten.



a) Querprofile (Aufsicht)

b) Längsprofil

c) Querprofil

Abb. 54: Querprofile.

3.6.9 Simulation

Computergestützte Simulationen ermöglichen die Untersuchung und Anwendung von Modellen dynamischer Prozesse. Die Aufgabe von Simulationsmodellen ist dabei, die zeitliche Änderung der Zustandsgrößen eines Systems gemäß den verantwortlichen Prozessen zu beschreiben. In vielen Fällen sind Simulationen die einzige Möglichkeit die zu Grunde liegenden Modelle zu überprüfen, z.B. für Prozesse, die sich nicht in der nötigen Auflösung messen lassen. Validierte Simulationsmodelle erlauben es dann, fundierte Aussagen über Abläufe zu treffen, die sich in der Realität so nicht beobachten oder durchführen lassen. Dies

Tab. 33: Module für die Simulation von Prozessen

Name/Bibliothek	**Beschreibung**	**Module**
Cellular Automata sim_cellular_automata	Zelluläre Automaten	• Conway's Life • Wa-Tor
Modelling the Human Impact on Nature *sim_ecosystems_hugget*	Numerische Modelle für ökologische Prozesse	• A Simple Litter System • Carbon Cycle Simulation for errestrial Biomes • Spatially Distributed Simulation of Soil Nitrogen Dynamics
Fire Spreading Analysis *sim_fire_spreading*	Analyse des Brandrisikos	• Fire Risk Analysis[VO] • Simulation[VO]
Hydrology *sim_hydrology*	Modellierung hydrologischer Prozesse	• Soil Moisture Content • Overland Flow - Kinematic Wave D8 • TOPMODEL • Water Retention Capacity[VO]

[VO]V.Olaya

betrifft insbesondere auch zukünftige oder in der Vergangenheit liegende Abläufe, wie z.B. das Klimageschehen. Ein weiterer Vorteil von Simulationen ist ihre Verwendung mit sogenannten Szenarios, durch die das Simulationsprogramm gewissermaßen zum Experimentallabor wird. Zum Teil sind die nachfolgend vorgestellten Module auf die Lehre ausgerichtet und sollen vor allem die Prinzipien dynamischer Simulationen veranschaulichen. Daneben finden sich aber auch Simulationsmodelle, die sich bereits in der Praxis bewährt haben oder einen Ausgangspunkt für zukünftige Anwendungen bilden.

Zelluläre Automaten

Zelluläre Automaten sind einfache dynamische Systeme, die in den meisten Fällen durch zweidimensionale, seltener auch durch dreidimensionale Gitter repräsentiert werden. Ihre Dynamik basiert auf einfachen lokalen Regeln, die sich jeweils auf die Abfrage in unmittelbarer Nachbarschaft liegender Gitterpunkte beschränken. Trotzdem zeigen zelluläre Automaten ein oft sehr komplexes Verhalten. Zelluläre Automaten haben sich erfolgreich bei der Simulation oszillierender chemischer Reaktionsprozesse bewährt (DEWDNEY 1989). Aber auch für ökologische Prozesse bieten zelluläre Automaten geeignete Konzepte, z.B. für die Simulation der Verbreitungsmuster der Vegetation oder des räumlichen Genflusses (SORK et al. 1998). Selbst für die Beschreibung sozial-ökonomisch determinierter Prozesse, wie der Stadtentwicklung, können zelluläre Automaten sinnvoll eingesetzt werden (CLARKE & GAYDOS 1998). Die beiden Module in „*sim_cellular_automata*“ sind sehr simpel und nicht für die praktische Anwendung gedacht, mögen aber als Ausgangspunkt für konkrete Problemlösungen dienen. Life, ausführbar mit dem gleichnamigen Modul, ist einer der populärsten zellulären Automaten und wird in Kap. 3.7.5 für die Einführung in die Programmierung dynamischer Simulationsmodelle ausführlich vorgestellt. Das Modul „*Wator*“ simuliert ein einfaches Räuber-Beute System, dessen Populationswerte für jeden Zeitschritt in eine Tabelle ausgegeben werden.

Numerische Modelle für ökologische Prozesse

Die drei Module der Bibliothek **sim_ecosystems_hugget** basieren auf der Einführung in die Systemanalyse umweltrelevanter Probleme von HUGGET (1993) und sollen in erster Linie den Einsatz numerischer Modelle für die Simulation demonstrieren. Die ersten beiden Module simulieren den standortbezogenen Kohlenstoffhaushalt ohne Berücksichtigung räumlicher

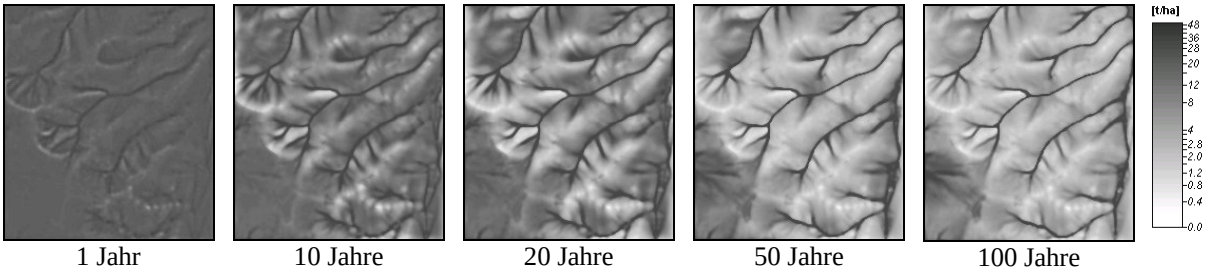


Abb. 55: Simulation der N-Verlagerung.

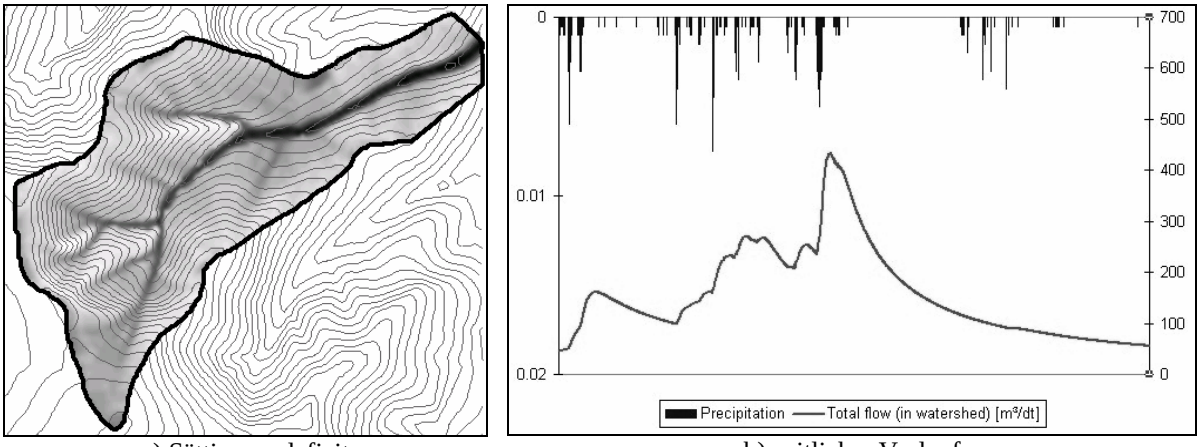
Wechselwirkungen und speichern ihre Ergebnisse in tabellarischer Form. Mit Raumbezug arbeitet dagegen das Modul zur Simulation der reliefgesteuerten Stickstoffverlagerung (Abb. 55).

Simulation der Entstehung und Ausbreitung von Bränden

Die beiden Module der Bibliothek **sim_fire_spreading** für die Analyse des Brandrisikos und der Feuerausbreitung basieren auf der BEHAVE Bibliothek für die stochastische Simulation von Waldbränden (ANDREWS 1986, FIRE.ORG 2006). Das erste Modul analysiert das Brandrisiko und generiert Brandherde mittels Monte-Carlo Simulation. Das zweite Modul simuliert die Ausbreitung von Bränden. Angaben in Form von Rasterdaten müssen u.a. für das brennbare Material, seine Feuchtigkeit sowie Windrichtung und Windgeschwindigkeit gemacht werden.

Modellierung hydrologischer Prozesse

Einfache hydrologische Simulationsmodelle werden von der Bibliothek **sim_hydrology** zur Verfügung gestellt. Die für landwirtschaftliche Anwendungen konzipierte Simulation des Bodenwassergehalts mit dem Modul „*Soil Moisture Content*" erwartet als Eingabe Rasterdatensätze mit Angaben zu Feldkapazität, permanentem Welkepunkt und Landnutzung sowie eine Tabelle mit dem zeitlichen Verlauf der Klimawerte Niederschlag, Temperatur und Luftfeuchtigkeit. Das Modul basiert auf einem einfachen, standortbezogenen Modell (DVWK 1996) und berücksichtigt zur Zeit keine horizontalen Stoffflüsse. Das Modul wurde erfolgreich für die Bestimmung von Zeiträumen eingesetzt, in denen die Bodenoberfläche trocken und damit potentiell durch Wind erodierbar ist (BÖHNER et al. 2003). Das Modul „*Overland Flow – Kinematic Wave*" ist ein Ansatz, den Oberflächenabfluss unter Verwendung der Approximation einer kinematischen Welle zu simulieren (JOHNSON & MILLER 1997), wobei die Abflussverteilung nach der D8 Methode vorgenommen wird (O'CALLAGHAN & MARK 1984). Das Modul demonstriert die prinzipielle Funktionsweise kinematischer Wellen, muss aber für die praktische Verwendung zukünftig erweitert werden. Die Implementierung des



a) Sättigungsdefizit b) zeitlicher Verlauf

Abb. 56: Hydrologische Simulation mit TOPMODEL

ausführlich in der Literatur diskutierten topographisch basierten hydrologischen Modells TOPMODEL (z.B. BEVEN et al. 1984, BEVEN 1997), basiert auf den originalen Fortran Quellen (TOPMODEL 2006) sowie ihrer Übersetzung in dem gleichnamigen GRASS-Modul (GRASS 2006). TOPMODEL ist ein bewusst sehr einfach gehaltenes Niederschlags-Abfluss-Modell, dessen räumlich differenzierte Vorhersagen auf der Analyse der Topographie basieren. Das Modell simuliert den Interflow wie auch den bei Sättigungs- und Infiltrationsüberschuss auftretenden Oberflächenabfluss. Der topographische Bodenfeuchte-Index (TWI), der zunächst aus einem DGM abgeleitet werden muss (s. Kap. 3.6.8), wird zur Klassifizierung des Reliefs in hydrologisch gleich reagierende Flächen benutzt. Die eigentliche Simulation arbeitet mit diesen Klassen und berücksichtigt die räumlichen Beziehungen also nur indirekt über den TWI. Als Eingabe erwartet das Modul ein Raster mit dem TWI und eine Tabelle mit dem zeitlichen Verlauf von Niederschlag und potentieller Evapotranspiration. Die Ausgabe von Überlandabfluss, Infiltrationsrate, Bodenwassersättigungsdefizit u.a. erfolgt in tabellarischer Form. Ausgaben als Raster sind optional möglich (Abb. 56).

3.7 Einführung in die Modulprogrammierung

Die in diesem Kapitel vorgestellte Einführung in die Programmierung von SAGA-Modulen basiert auf einer im WS 2003/04 am Geographischen Institut der Universität Göttingen durchgeführten Übung mit dem Titel *Einführung in die Programmierung geowissenschaftlicher Anwendungen*. Grundlegende Programmierkenntnisse wurden vorausgesetzt, spezielle Kenntnisse von C/C++ jedoch nicht verlangt, da sich die wenigen benötigten C/C++ spezifischen Kenntnisse über die Verwendung von Zeigern und Klassenobjekten relativ schnell vermitteln lassen. Die ursprünglich 14 Übungen wurden für diese Einführung vereinfacht und auf 10 Übungen reduziert. Jeder Übung steht eine Aufgabenstellung voran, die es zu lösen gilt, sowie eine Aufstellung der neu eingeführten Funktionalitäten. Der hier vermittelte Lehrinhalt gibt einen umfassenden Einblick in die Verarbeitung von Raster- und Vektordaten. Am Ende dieser Einführung sollte der Leser genug Werkzeug in der Hand haben, um eigene Fragestellungen selbständig umsetzen zu können.

Grundkenntnisse der Programmierung in C/C++, wie sie für ein besseres Verständnis dieser Einführung nötig sind, können Kapitel 2.1 entnommen werden. Einen Überblick über den Aufbau und Umfang der SAGA-API gibt Kapitel 3.3. Auch die nachfolgenden Quelltexte halten sich im Sinne einer besseren Lesbarkeit an gewisse Namenskonventionen. So werden Zeigervariablen immer mit dem führenden Buchtstaben „*p*“ (für engl. *pointer*) und Klassenvariablen mit „*m*“ (für engl. *member*) gekennzeichnet. Die Quelltexte folgen der in C/C++ üblichen Trennung in Deklaration und Definition. Über Deklarationen werden Funktionen und ihre Eigenschaften bekannt gegeben, so dass andere Funktionen sie ansprechen können. Die Definition, also die Implementierung der Funktionalität, muss für die Verwendung einer Funktion nicht bekannt sein und erfolgt meist in einer separaten Datei. Deklarationen werden in Header-Dateien mit der Dateierweiterung „*h*“ geschrieben. Definitionen erfolgen in Quelltextdateien mit der für C++ typischen Endung „*cpp*“.

Es wird vorausgesetzt, dass SAGA gemäß Kapitel 3.1 für die Modulprogrammierung konfiguriert wurde, so dass Header und Bibliothek der SAGA-API während der Kompilierung vom Compiler gefunden werden können. Die vollständigen Quelltexte sind in Anhang B abgedruckt. Auf der CD-Beilage befinden sich zusätzlich Projektdateien bzw. Makefiles für die Compiler von GCC, MinGW und VC 6/8, so dass die Quelltexte direkt kompiliert, getestet und erweitert werden können.

3.7.1 Das erste Modul

Aufgabe	Kopieren eines Rasterdatensatzes	
Lernziel	Erstellen eines SAGA Moduls	
Quelltextdateien	exercise_01.cpp, exercise_01.h, mlb_interface.cpp	
Neue Funktionen	CSG_Module	Konstruktor
		On_Execute
		Set_Name, Set_Author, Set_Description
	CSG_Parameters	Add_Grid
		() Operator
	CSG_Parameter	asGrid
	CSG_Grid	Assign
	Modulbibiliothek	Get_Info
		Create_Module
		MLB_INTERFACE

Ziel der ersten Übung ist es einen Rasterdatensatz zu kopieren und dabei die Prinzipien der Modulerstellung kennen zu lernen. Anders als bei den folgenden Übungen, wird daher detailliert auf die verschiedenen Deklarations- und Implementierungsschritte, den Aufbau der Modulbibliotheksschnittstelle, wie auch auf die exemplarische Erstellung von Makefiles für die Kompilierung einer Modulbibliothek eingegangen.

Die Header-Datei

Der erste Schritt ist die Deklaration der neuen Modulklasse *CExercise_01* in der Header-Datei „*example_01.h*“. Die Funktionen der Klasse werden hier nur bekannt gegeben. Ihre Implementierung erfolgt in der Datei „*example_01.cpp*“. Die Schnittstelle der Modulbibliothek greift später auf die Header-Datei zu, damit sie eine Instanz von dieser Klasse erstellen kann. Die recht kompakte Datei soll insgesamt folgendermaßen aussehen:

```
#ifndef HEADER_INCLUDED__Exercise_01_H
#define HEADER_INCLUDED__Exercise_01_H

#include <saga_api/saga_api.h>

class CExercise_01 : public CSG_Module_Grid
{
public:
   CExercise_01(void);       // constructor

protected:
   virtual bool   On_Execute(void); // always override this function
};

#endif // #ifndef HEADER_INCLUDED__Exercise_01_H
```

Da die Header-Datei von mehreren Quelltextdateien eingebunden werden wird, soll mit den Präprozessor-Anweisungen am Anfang und am Ende des Headers verhindert werden, dass sie vom Compiler mehrfach ausgewertet wird, was zu Fehlermeldungen führen und das Quelltextprojekt unkompilierbar machen kann. In dieser Übung sind sie nicht zwingend erforderlich, werden aber im Sinne eines nachhaltig erweiterbaren Programmierkonzeptes auch in den weiteren Übungen konsequent mitgeführt.

```
#ifndef HEADER_INCLUDED__Exercise_01_H
#define HEADER_INCLUDED__Exercise_01_H

...

#endif // #ifndef HEADER_INCLUDED__Exercise_01_H
```

Um Zugriff auf die Funktionalitäten der SAGA-API zu erhalten, müssen zunächst die entsprechenden Header-Dateien mit der folgenden Zeile eingebunden werden.

```
...
#include <saga_api/saga_api.h>
...
```

Damit kann im nächsten Schritt die Klasse *CExercise_01* deklariert werden. Als Basisklasse wird *CSG_Module_Grid* gewählt, die wiederum von *CSG_Module* abgeleitet ist und diese um spezielle Methoden für das Arbeiten mit Rasterdaten erweitert (s. Kap. 3.3.5). Die Klasse *CExercise_01* verfügt dadurch über alle Eigenschaften und Methoden von *CSG_Module* und *CSG_Module_Grid*.

```
class CExercise_01 : public CSG_Module_Grid
{
   ...
};
```

Zwei Funktionen werden dieser Klasse hinzugefügt. Zunächst ein Klassen-Konstruktor, der für die Initialisierung der Klasse verantwortlich ist und immer denselben Namen wie die Klasse haben muss.

```
...
public:
   CExercise_01(void);       // constructor
...
```

Da der Konstruktor für die Erstellung einer Instanz der Klasse öffentlich aufrufbar sein muss, wird er in die *public* Sektion der Klasse eingetragen. Anders ist es mit der zweiten Funktion *On_Execute*, die vor öffentlichen Zugriffen geschützt werden soll und daher in die *protected* Sektion geschrieben wird..

```
...
protected:
   virtual bool   On_Execute(void); // always override this function
...
```

Die *On_Execute*-Funktion wurde von der Basisklasse *CSG_Module* geerbt und wird von der SAGA-Umgebung automatisch bei der Ausführung des Moduls aufgerufen. Somit ist diese Funktion genau die Stelle, an der die Kopierfunktion zu implementieren ist. Tatsächlich muss diese Funktion sogar überschrieben werden, um die Modulklasse instanziierbar zu machen, da sie von *CSG_Module* zwar deklariert, aber nicht mit einer Standardfunktionalität versehen wurde. Um dem Compiler mitzuteilen, dass die Funktion jetzt mit einer neuen Funktionalität überschrieben werden soll, wird der Funktionsdeklaration das C++ Schlüsselwort *virtual* voran gestellt.

Die Implementierung

Nachdem die Klasse deklariert wurde, erfolgt die Implementierung ihrer Funktionen in der Datei „*exercise_01.cpp*“. Als erstes wird die Header-Datei „*exercise_01.h*“ mit der zuvor erstellten Deklaration eingebunden.

```
#include "exercise_01.h"
...
```

Der Konstruktor ist eine besondere Klassenfunktion, die immer dann ausgeführt wird, wenn eine Instanz der Klasse erstellt wird. Somit ist der Konstruktor die Funktion, in der alle Initialisierungen einer Klasse vorgenommen werden. In Modulklassen beschränkt sich die Initialisierung meistens auf die Beschreibung und die Parameterliste des Moduls. Die folgenden Funktionen, die *CExercise_01* von der Basisklasse *CSG_Module* geerbt hat, dienen dazu dem Benutzer Informationen über das Modul zu geben:

```
void    CSG_Module::Set_Name          (const char *String)
void    CSG_Module::Set_Author        (const char *String)
void    CSG_Module::Set_Description   (const char *String)
```

Mit dem Funktionsparameter *String* können hier jeweils geeignete Texte gesetzt werden. Ebenfalls von der Basisklasse *CSG_Module* geerbt wurde die Eigenschaft *Parameters* vom Typ *CSG_Parameters*. Hierbei handelt es sich um eine Liste, die die Parameter des Moduls enthält. Beim Aufruf des Konstruktors ist diese Liste immer leer. Für das neue Modul werden nun zwei Parameter hinzu gefügt. Beide stehen für Rasterdatensätze, wobei der erste derjenige sein soll, dessen Werte kopiert werden sollen, der zweite derjenige, der die Werte aufnehmen soll. Zum Hinzufügen dieser Parameter zu der Liste dient die Funktion

```
CSG_Parameter * CSG_Parameters::Add_Grid(
   CSG_Parameter *pParent,
   const char *Identifier,
   const char *Name,
   const char *Description,
   int Constraint
)
```

der Klasse *CSG_Parameters*. Wie später zu sehen sein wird, stehen ähnliche Funktionen auch für verschiedene andere Parametertypen zur Verfügung. Bei jeder dieser Funktionen sind die ersten vier Funktionsargumente (*pParent*, *Identifier*, *Name*, *Description*) immer identisch. Der Funktionsparameter *pParent* dient nur einer übersichtlicheren Strukturierung der

Parameterliste durch die Benutzeroberfläche. Wird hier ein zuvor erstellter Parameter als Zeiger übergeben, so wird der neu erstellte Parameter diesem hierarchisch untergeordnet. Soll das nicht geschehen, wird standardmäßig der Wert *NULL* übergeben. Über die als *Identifier* übergebene Zeichenkette kann ein erstellter Parameter im späteren Programmverlauf wieder abgefragt werden. Er fungiert gewissermaßen als Variablenname. Es ist daher wichtig, dass dieser Wert in einer Liste nur einmal vorkommt. *Name* ist die für den Benutzer sichtbare Bezeichnung des Parameters. *Description* dient der Erläuterung des Parameters. Der Funktionsparameter *Constraint* ist spezifisch für die *Add_Grid* Funktion und legt fest, ob der Datensatz als Eingabe oder als Ausgabe dient, und, ob der Datensatz optional oder obligatorisch ist. Die entsprechenden Werte sind in der API vordefiniert:

```
#define PARAMETER_INPUT           0x01
#define PARAMETER_OUTPUT          0x02
#define PARAMETER_OPTIONAL        0x04
#define PARAMETER_INPUT_OPTIONAL  (PARAMETER_INPUT  | PARAMETER_OPTIONAL)
#define PARAMETER_OUTPUT_OPTIONAL (PARAMETER_OUTPUT | PARAMETER_OPTIONAL)
```

Der vollständige Konstruktor sollte in etwa so aussehen:

```
CExercise_01::CExercise_01(void)
{
   Set_Name          ("01 - Mein erstes Modul");
   Set_Author        ("Olaf Conrad");
   Set_Description   ("Übung 1: Mein erstes Modul kopiert einen Rasterdatensatz.");

   Parameters.Add_Grid(
      NULL, "INPUT"  , "Das Original",
      "Dieser Datensatz soll kopiert werden.",
      PARAMETER_INPUT
   );

   Parameters.Add_Grid(
      NULL, "OUTPUT" , "Die Kopie",
      "In diesen Datensatz soll kopiert werden.",
      PARAMETER_OUTPUT
   );
}
```

Das Kopieren des Rasterdatensatzes wird in der Funktion *On_Execute* implementiert, deren Aufruf automatisch erfolgt, wenn das Modul ausgeführt wird. Zum Zeitpunkt des Aufrufs kann davon ausgegangen werden, dass die Parameterliste gültige, vom Benutzer gesetzte Werte enthält. Verantwortlich für die Gültigkeitsüberprüfung ist die Basisklasse *CSG_Module*. Die für diese Übung gewählte Umsetzung der Kopierfunktion ist bewusst sehr einfach gehalten, indem sie auf eine Funktion der Klasse *CSG_Grid* zurückgreift.

```
bool CExercise_01::On_Execute(void)
{
   CSG_Grid *pInput, *pOutput;

   pInput   = Parameters("INPUT")   ->asGrid();
   pOutput  = Parameters("OUTPUT")  ->asGrid();

   pOutput->Assign(pInput);

   return( true );
}
```

Aber zuerst erfolgt die Abfrage der über die Parameterliste angeforderten Eingabe- und Ausgabeparameter mittels der für diese gesetzten *Identifier* (s.o.). Dazu genügt es, die Parameterliste wie eine Funktion mit dem Namen des gesuchten *Identifiers* aufzurufen, wodurch ein Zeiger auf den entsprechenden Parameter zurückgegeben wird. Möglich ist das

dadurch, dass der () Operator von der *CSG_Parameters*-Klasse entsprechend überschrieben wurde:

```
CSG_Parameter * CSG_Parameters::operator () (const char *Identifier)
```

Da es sich bei beiden angeforderten Parametern um Rasterdatensätze handelt, wird die *asGrid*-Funktion für das zurückgelieferte *CSG_Parameter*-Objekt aufgerufen, um Zugriff auf die eigentlichen Rasterdatenobjekte vom Typ *CSG_Grid* zu erhalten

```
CSG_Grid *          CSG_Parameter::asGrid (void)
```

asGrid liefert einen Zeiger auf das vom Benutzer ausgewählte Rasterdatenobjekt zurück. Der Übersichtlichkeit halber werden die Zeiger für die weitere Verarbeitung in den Variablen *pInput* bzw. *pOutput* zwischengespeichert. Für das Kopieren der Datenwerte selbst wird die Funktion *Assign* der *CSG_Grid* Klasse benutzt.

```
bool            CSG_Grid::Assign          (CSG_Grid *pGrid)
```

Um der SAGA-Umgebung abschließend mitzuteilen, dass die Modulausführung ohne Fehler verlief, wird die Funktion *On_Execute* mit der Rückgabe des Wahrheitswerts *true* beendet.

Die Modulbibliotheksschnittstelle

Damit das Modul aus der SAGA Umgebung heraus aufgerufen werden kann, muss noch die Modulbibliotheksschnittstelle um dieses Modul erweitert werden. Der Quelltext der Modulbibliotheksschnittstelle wird dazu in der Datei „*mlb_interface.cpp*“ abgelegt. Auch hier werden als erstes die Header der SAGA-API eingebunden.

```
#include <saga_api/saga_api.h>
...
```

Allgemeine Informationen über die Modulbibliothek werden, unterschieden nach dem Wert *Info_ID*, in die *Get_Info* Funktion eingetragen. Die zulässigen Werte für *Info_ID* sind in der API definiert. Neben Namen und Beschreibung gehört hierzu die optionale Angabe eines Stammmenüs, dem die Module der Bibliothek untergeordnet werden sollen.

```
const char * Get_Info(int Info_ID)
{
   switch( Info_ID )
   {
   case MLB_INFO_Name:        return( "Modul-Programmierung" );
   case MLB_INFO_Author:      return( "Olaf Conrad" );
   case MLB_INFO_Description: return( "Eine Einführung in die Modul-Programmierung." );
   case MLB_INFO_Version:     return( "1.0" );
   case MLB_INFO_Menu_Path:   return( "Übungen zur Modul-Programmierung" );
   }

   return( NULL );
}
```

Modulinstanzen werden mit der Funktion *Create_Module* erstellt. Bevor aber von *CExercise_01* eine Instanz erstellt werden kann, muss noch der Header „*excercise_01.h*“ mit der zugehörigen Klassendeklaration eingebunden werden. Die Instanziierung geschieht mit dem *new* Operator, der einen Zeiger auf das neu erzeugte Objekt vom Typ *CExercise_01* liefert. Dieser wird schließlich von der Funktion zurückgegeben. Die SAGA-Umgebung wird diese Funktion später solange bei 0 beginnend mit aufsteigenden Werten für *iModule* aufrufen (0, 1, 2, ...), bis die Funktion den Wert *NULL* zurückliefert. Da bis jetzt nur ein Modul erstellt

wurde, wird auch nur der Fall behandelt, dass *iModule* gleich *0* ist. Für alle anderen Werte wird der *Null*-Zeiger zurückgegeben.

```
#include "Exercise_01.h"

CSG_Module * Create_Module(int iModule)  // Create instances of your modules here...
{
   switch( iModule )
   {
   case 0:  return( new CExercise_01 );
   default: return( NULL );
   }
}
```

Am Ende der Datei muss das Makro *MLB_INTERFACE* stehen, das für die Implementierung der eigentlichen Modulbibliotheksfunktionen verantwortlich ist. Die Details des Makros sind für die Modulprogrammierung unerheblich.

```
...

MLB_INTERFACE
```

Kompilation und Ausführung

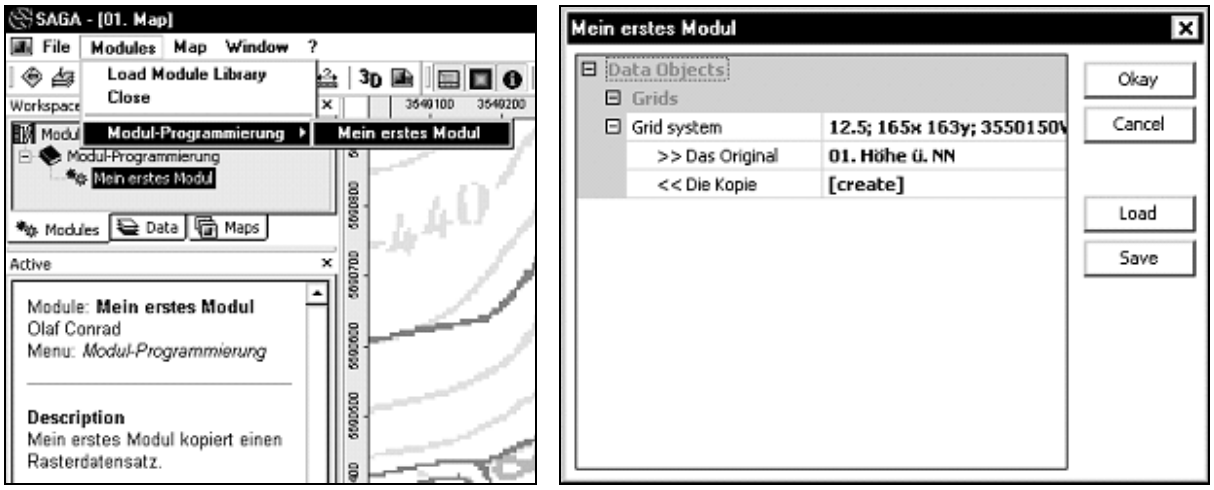
Die Erzeugung einer lauffähigen Modulbibliothek ist abhängig vom verwendeten Compiler und Betriebssystem. Neben der Angabe der zu kompilierenden Quelltextdateien muss dem Compiler mitgeteilt werden, wo sich die Header-Dateien und Bibliothek der SAGA-API befinden, und auch, dass das Ausgabeformat eine dynamisch ladbare Bibliothek sein soll. Als Beispiel ist hier ein Makefile für den MinGW Compiler abgebildet.

```
# MinGW Makefile created by O. Conrad

CPP      = g++.exe
LD       = dllwrap.exe
DLL_NAME = excercises.dll
OBJ      = MLB_Interface.o Exercise_01.o
CXXFLAGS = -I"$(MINGW)/include" -I"$(SAGA)/src/saga_core" -DBUILDING_DLL=1 -D_SAGA_MSW -
D_TYPEDEF_BYTE -D_TYPEDEF_WORD
LDFLAGS  = -L"$(MINGW)/lib" -L"$(SAGA)/bin/saga_mingw" --driver-name c++

.PHONY: all all-before all-after clean clean-custom

all: all-before $(DLL_NAME) all-after

clean: clean-custom
.  rm -f $(OBJ) $(DLL_NAME)

$(DLL_NAME): $(OBJ)
.  $(LD) $(LDFLAGS) $(OBJ) -lsaga_api -o $(DLL_NAME)

MLB_Interface.o: MLB_Interface.cpp
.  $(CPP) -c MLB_Interface.cpp -o MLB_Interface.o $(CXXFLAGS)

Exercise_01.o: Exercise_01.cpp
.  $(CPP) -c Exercise_01.cpp -o Exercise_01.o $(CXXFLAGS)
```

Die Kompilierung mit dem MinGW Compiler wird von der Kommandozeile aus mit dem Befehl

```
C:\exercises>make -f makefile.mingw
```

ausgeführt. Nach erfolgreicher Kompilierung kann die neue Modulbibliothek von SAGA geladen und das neue Modul ausgeführt werden.



a) Modulaufruf über Menü b) Parameterdialog

Abb. 57: Ausführung des ersten Moduls.

3.7.2 Einfache Operationen mit Rasterdaten

Aufgabe	Zellenweise Verknüpfung von Rasterdaten durch einfache arithmetische Operationen	
Lernziel	Lese- und Schreibzugriff auf die Werte der Zellen von Rasterdatensätzen	
Quelltextdateien	exercise_02.cpp, exercise_02.h, mlb_interface.cpp	
Neue Funktionen	CSG_Module	Message_Dlg
	CSG_Module_Grid	Set_Progress
		Get_NX, Get_NY
	CSG_Parameters	Add_Value
		Add_Choice
	CSG_Parameter	asDouble
		asInt
	CSG_Grid	is_NoData, Set_NoData, Get_NoData_Value
		asDouble
		Set_Value

In der zweiten Übung sollen zwei Rasterdatensätze Zelle für Zelle über einfache arithmetische Funktionen miteinander verknüpft werden. Die Deklaration des zweiten Moduls ist bis auf den Klassennamen weitgehend identisch mit der der vorangegangenen Übung. Der komplette Quelltext der Datei „*exercise_02.h*“ sieht folgendermaßen aus:

```
#ifndef HEADER_INCLUDED__Exercise_02_H
#define HEADER_INCLUDED__Exercise_02_H

#include <saga_api/saga_api.h>
```

```
class CExercise_02 : public CSG_Module_Grid
{
public:
   CExercise_02(void);

protected:
   virtual bool   On_Execute(void);
};

#endif // #ifndef HEADER_INCLUDED__Exercise_02_H
```

In angemessener Weise werden im Konstruktor Name und Beschreibung des Moduls gesetzt und die Parameterliste initialisiert. In dieser Übung werden zwei Rasterdatensätze als Eingabe sowie ein einzelner Rasterdatensatz für die Ausgabe angefordert. Zusätzlich soll über weitere Modulparameter die Art der Verknüpfung gesteuert werden können. Die komplette Initialisierung soll folgendermaßen aussehen.

```
CExercise_02::CExercise_02(void)
{
   Set_Name          ("02 - Einfache Rasteroperationen");
   Set_Author        ("Olaf Conrad");
   Set_Description   ("Übung 2: Einfache Operationen mit Rasterdaten.");

   Parameters.Add_Grid(
      NULL, "GRID_A"    , "A",
      "Der erste Rasterdatensatz.",
      PARAMETER_INPUT
   );

   Parameters.Add_Grid(
      NULL, "GRID_B"    , "B",
      "Der zweite Rasterdatensatz.",
      PARAMETER_INPUT
   );

   Parameters.Add_Grid(
      NULL, "GRID_C"    , "A kombiniert mit B",
      "Dieser Datensatz soll das Ergebnis enthalten.",
      PARAMETER_OUTPUT
   );

   Parameters.Add_Value(
      NULL, "FACTOR_A"  , "Faktor A",
      "Skalierungsfaktor für Rasterdatensatz A.",
      PARAMETER_TYPE_Double,
      1.0
   );

   Parameters.Add_Value(
      NULL, "FACTOR_B"  , "Faktor B",
      "Skalierungsfaktor für Rasterdatensatz B.",
      PARAMETER_TYPE_Double,
      1.0
   );

   Parameters.Add_Choice(
      NULL, "METHOD"    , "Operation",
      "Wähle eine arithmetische Operation.",
      "Addition|Subtraktion|Multiplikation|Division|"
   );
}
```

Zusätzlich zu den Rasterdatensätzen werden mit der *CSG_Parameters*-Funktion *Add_Value* zwei weitere Parameter angefordert, die für vom Benutzer frei wählbare Zahlenwerte stehen.

```
void CSG_Parameters::Add_Value(
   CSG_Parameter *pParent,
   const char *Identifier,
   const char *Name,
   const char *Description,
   TParameter_Type Type, double Value,
   double Minimum = 0, bool bMinimum = false,
   double Maximum = 0, bool bMaximum = false
)
```

Mit *Add_Value* werden Zahlenwerte angefordert, die je nach dem gesetzten Wert für *Type* als einer der folgenden Subtypen spezifiziert werden

```
PARAMETER_TYPE_Bool      Wahrheitswert
PARAMETER_TYPE_Int       Ganzzahl
PARAMETER_TYPE_Double    Fließkommazahl
```

Ein Standardwert wird mit *Value* vorgegeben. Der zulässige Wertebereich kann mit *Minimum*/*Maximum* gesetzt und mit *bMinimum*/*bMaximum* aktiviert werden. Zum Schluss wird mit der Funktion *Add_Choice* die Parameterliste noch um eine Auswahlliste erweitert. Das vom Benutzer ausgewählte Listenelement kann später als Ganzzahl abgefragt werden und liefert die Position in der Liste.

```
CSG_Parameter * CSG_Parameters::Add_Choice(
   CSG_Parameter *pParent,
   const char *Identifier,
   const char *Name,
   const char *Description,
   const char *Items,
   int Default = 0
)
```

Die Auswahlliste selbst wird als Zeichenkette mit dem Funktionsargument *Items* übergeben. Die Abgrenzung der einzelnen Listenelemente innerhalb der Zeichenkette erfolgt mit dem Zeichen „|“ (engl. *pipe*). In dieser Übung wird die Auswahlliste zur Festlegung der durchzuführenden arithmetischen Operation benutzt. Die beiden Fließkommazahlen sollen zur Skalierung der Rasterwerte dienen. Die Abfrage dieser Parameter erfolgt mit den Funktionen der *CSG_Parameter*-Klasse

```
double CSG_Parameter::asDouble   (void)
```

für Fließkommazahlen und

```
int    CSG_Parameter::asInt  (void)
```

für Ganzzahlen bzw. für das gewählte Element der Auswahlliste. Bei Auswahllisten gilt zu beachten, dass die Zählung der Elementposition in C-typischer Weise mit 0 beginnt und nicht mit 1. Eine Auswahl des ersten Elements liefert also eine 0 zurück, des zweiten eine 1. Element 3 entspricht daher der Methode „*Division*“ aus der Auswahlliste.

Die *On_Execute*-Funktion gliedert sich in die Deklaration der lokal verwendeten Variablen, dem Auslesen der angeforderten Parameter, einer zusätzlichen Überprüfung auf gültige Parameterwerte und der eigentlichen Routine zum Berechnen des Ergebnisses.

```
bool CExercise_01::On_Execute(void)
{
   int      x, y, Method;
   double   Factor_A, Factor_B, a, b, c;
   CSG_Grid *pA, *pB, *pC;

   pA       = Parameters("GRID_A")     ->asGrid();
   pB       = Parameters("GRID_B")     ->asGrid();
   pC       = Parameters("GRID_C")     ->asGrid();
   Factor_A = Parameters("FACTOR_A")   ->asDouble();
   Factor_B = Parameters("FACTOR_B")   ->asDouble();
   Method   = Parameters("METHOD")     ->asInt();

   if( Method == 3 && Factor_B == 0.0 )  // Check for invalid parameter settings...
   {
      Message_Dlg("Division by zero is not allowed !!!");
      return( false );
   }
```

```
   for(y=0; y<Get_NY() && Set_Progress(y); y++)   // each row...
   {
      for(x=0; x<Get_NX(); x++)             // each column...
      {
         if( pA->is_NoData(x, y) || pB->is_NoData(x, y) )  // don't work with 'no data'...
            pC->Set_NoData(x, y);
         else
         {
            a  = Factor_A * pA->asDouble(x, y);
            b  = Factor_B * pB->asDouble(x, y);

            switch( Method )
            {
            case 0:  // Addition...
               c  = a + b;
               break;

            case 1:  // Subtraction...
               c  = a - b;
               break;

            case 2:  // Multiplication...
               c  = a * b;
               break;

            case 3:  // Division...
               if( b == 0.0 ) // prevent division by zero
                  c  = pC->Get_NoData_Value();
               else
                  c  = a / b;
               break;
            }

            pC->Set_Value(x, y, c);
         }
      }
   }

   return( true );
}
```

Nach dem Auslesen der Parameter findet eine zusätzliche Überprüfung statt. Wurde nämlich der Parameter *Factor_B* für den Fall der Division auf 0 gesetzt, wird eine Fehlermeldung mit der Funktion

```
void   CSG_Module::Message_Dlg   (const char *Text, const char *Caption)
```

ausgegeben und die weitere Modulausführung mit Rückgabewert *false* abgebrochen. Im Normalfall folgt jedoch die zellenweise Durchführung der gewählten arithmetischen Operation. Dazu wird eine geschachtelte Schleife für die Abfrage aller Spalten und Reihen des Rasters formuliert. Die jeweilige Rasterzellenzahl erhält man mit

```
int    CSG_Module_Grid::Get_NX   (void)
```

für die Anzahl der Spalten und

```
int    CSG_Module_Grid::Get_NY   (void)
```

für die Anzahl der Reihen. Da Berechnungen für größere Raster längere Zeit in Anspruch nehmen können, soll dem Benutzer der Fortschritt der Berechnung mit der Funktion

```
bool   CSG_Module_Grid::Set_Progress(int iRow)
```

mitgeteilt werden. Diese Funktion zeigt den Fortschritt in Prozent der Anzahl der Reihen an. Sobald der Benutzer die Modulausführung abbrechen möchte, liefert *Set_Progress* den Wahrheitswert *false* anstelle von *true* zurück. Indem der Rückgabewert von *Set_Progress* in die Abbruchbedingung der *for* Schleife einbezogen wird, kann die Schleife also jederzeit vom Benutzer abgebrochen werden. Rasterdatensätze können so genannte *No-Data* Werte enthalten, für die keine Berechnung vorgenommen und die auch im Ausgaberaster als No-Data

gekennzeichnet werden sollen. Für die Abfrage und das Setzen von No-Data Werten stehen die Funktionen

```
bool   CSG_Grid::is_NoData   (int x, int y)
```

und

```
bool   CSG_Grid::Set_NoData  (int x, int y)
```

der Klasse *CSG_Grid* zur Verfügung. Lese- und Schreibzugriff auf die Werte der Rasterzellen erhält man mit den Funktionen

```
bool   CSG_Grid::Set_Value   (int x, int y, double Value)
```

und

```
double CSG_Grid::asDouble    (int x, int y)
```

Für den Fall der Division wird außerdem der potentiell auftretende Fall der Division durch 0 verhindert. Mit Hilfe von

```
double CSG_Grid::Get_NoData_Value(void)
```

wird für diesen Fall eine alternative Möglichkeit aufgezeigt, um No-Data Werte in das Ausgaberaster zu schreiben.

Abschließend ist zur Verdeutlichung der Funktionsweise der Modulbibliotheksschnittstelle noch einmal die um das Modul der zweiten Übung erweiterte *Create_Module*-Funktion in „*mlb_interface.cpp*“ wiedergegeben:

```
...

#include "Exercise_00.h"
#include "Exercise_01.h"

CSG_Module *      Create_Module(int i)
{
   switch( i )
   {
   case 0:  return( new CExercise_01 );
   case 1:  return( new CExercise_02 );
   default: return( NULL );
   }
}

...
```

Auch das Makefile für den MinGW Compiler muss um die neue Quelltextdatei ergänzt werden, damit die Bibliothek korrekt erstellt werden kann:

```
OBJ      = MLB_Interface.o Exercise_01.o Exercise_02.o

...

Exercise_02.o: Exercise_02.cpp
.  $(CPP) -c Exercise_02.cpp -o Exercise_02.o $(CXXFLAGS)
```

3.7.3 Rasteroperationen mit Nachbarschaftsbeziehungen

Aufgabe	Mittelwertbildung über die Umgebung einer Rasterzelle mit beliebigen Suchfenstern	
Lernziel	Zugriff auf Nachbarschaften im Raster	
Quelltextdateien	exercise_03.cpp, exercise_03.h	
Neue Funktionen	allgemein	Verwendung von Klassen-Variablen
	CSG_Module_Grid	Get_xTo, Get_yTo
	CSG_Parameters	Add_Value mit Wertebereichsüberprüfung
	CSG_Parameter	asBool
	CSG_Grid	is_InGrid

In der dritten Übung wird am Beispiel der Mittelwertbildung die Einbeziehung der Nachbarschaft einer Rasterzelle in Berechnungen mit zwei verschiedene Methoden vorgestellt.

Hierzu wird die Klasse *CExercise_03* um die zwei Funktionen *Get_Mean_01* und *Get_Mean_02* ergänzt. Da diese nur für den internen Gebrauch bestimmt sind, werden sie in der *private* Sektion der Klasse eingefügt. Beide Funktionen sollen Zugriff auf das Eingaberaster haben. Aus diesem Grund wird der Klasse noch die Variable *m_pInput* vom Typ eines Zeigers auf ein *CSG_Grid*-Objekt hinzugefügt, die während der Modulausführung auf das Eingaberaster zeigen soll. Das führende „*m*“ dient der besseren Lesbarkeit und soll verdeutlichen, dass es sich hierbei um ein Mitglied der Klasse und nicht um eine lokale Variable handelt. Im Übrigen entspricht *CExercise_03* den vorher erstellten Modulklassen.

```
class CExercise_03 : public CSG_Module_Grid
{

...

private:
   CSG_Grid *m_pInput;

   double   Get_Mean_01 (int x, int y);
   double   Get_Mean_02 (int x, int y, int Radius);
};
```

Im Konstruktor werden neben den zwei Parametern für die Eingabe- bzw. Ausgaberaster sowie einer Auswahlliste zur Festlegung der Methode, noch zwei weitere Werte angefordert. Die Suchfenstergröße, ausgedrückt als Zahl der Rasterzellen, ist ein Ganzzahlenwert, der auf einen Wertebereich von 1 bis 100 begrenzt wird. Die zweite Option ist ein Wahrheitswert, mit dem festgelegt wird, ob die Mittelwerte selbst oder die Differenz zum Mittelwert berechnet werden soll.

```
   Parameters.Add_Value(
      NULL, "RADIUS" , "Suchfenster",
      "Größe des benutzerdefinierten Suchfensters in Anzahl der Rasterzellen.",
      PARAMETER_TYPE_Int, 5, 1, true, 100, true
   );

   Parameters.Add_Value(
      NULL, "DIFFER" , "Bilde die Differenz",
      "Wenn gesetzt wird die Differenz zum Mittelwert berechnet.",
      PARAMETER_TYPE_Bool, false
   );
```

Die *On_Execute*-Funktion liest zunächst wieder alle angeforderten Parameter aus. Dabei wird das Eingaberaster nicht in einer lokalen, auf die Funktion beschränkten Variablen, sondern in der klassenweit gültigen Variablen *m_pInput* gespeichert, so dass auch von den anderen Funktionen der Klasse auf das Raster zugegriffen werden kann. Anschließend werden mit einer Doppelschleife alle Rasterzellen prozessiert, wobei der Ergebniswert in Abhängigkeit der gewählten Methode von einer der beiden Unterfunktionen *Get_Mean_01* oder *Get_Mean_02* berechnet wird.

```
bool CExercise_03::On_Execute(void)
{
   bool     bDifference;
   int      x, y, Method, nCells;
   double   Result;
   CSG_Grid    *pOutput;

   m_pInput    = Parameters("INPUT")   ->asGrid();
   pOutput     = Parameters("OUTPUT")  ->asGrid();
   bDifference = Parameters("DIFFER")  ->asBool();
   Method      = Parameters("METHOD")  ->asInt();
   nCells      = Parameters("RADIUS")  ->asInt();

   for(y=0; y<Get_NY() && Set_Progress(y); y++)
   {
      for(x=0; x<Get_NX(); x++)
      {
         if( m_pInput->is_NoData(x, y) )
            m_pOutput->Set_NoData(x, y);
         else
         {
            switch( Method )
            {
            case 0:  Result   = Get_Mean_01(x, y);       break;
            case 1:  Result   = Get_Mean_02(x, y, nCells); break;
            }

            if( bDifference )
               Result   -= m_pInput->asDouble(x, y);

            pOutput->Set_Value(x, y, Result);
         }
      }
   }

   return( true );
}
```

Die erste Unterfunktion summiert zunächst die Werte der acht direkt benachbarten Rasterzellen. Um die Position der jeweiligen Rasterzelle zu erhalten, werden die Funktionen

```
int    CSG_Module_Grid::Get_xTo(int Direction, int x = 0)
int    CSG_Module_Grid::Get_yTo(int Direction, int y = 0)
```

benutzt. *Direction* bezeichnet die von 0 bis 7 im Uhrzeigersinn durchnummerierten Nachbarzellen (Abb. 58a), *x* die Spalte und *y* die Reihe der zentralen Rasterzelle. Von dieser Möglichkeit der richtungsabhängigen Nachbarschaftsabfrage wird in der nachfolgenden Übung

noch Gebrauch gemacht werden. Die Überprüfung, ob die berechnete Position auch wirklich in das Raster fällt, was bei Randzellen nicht für jede Richtung gegeben ist, erfolgt mit

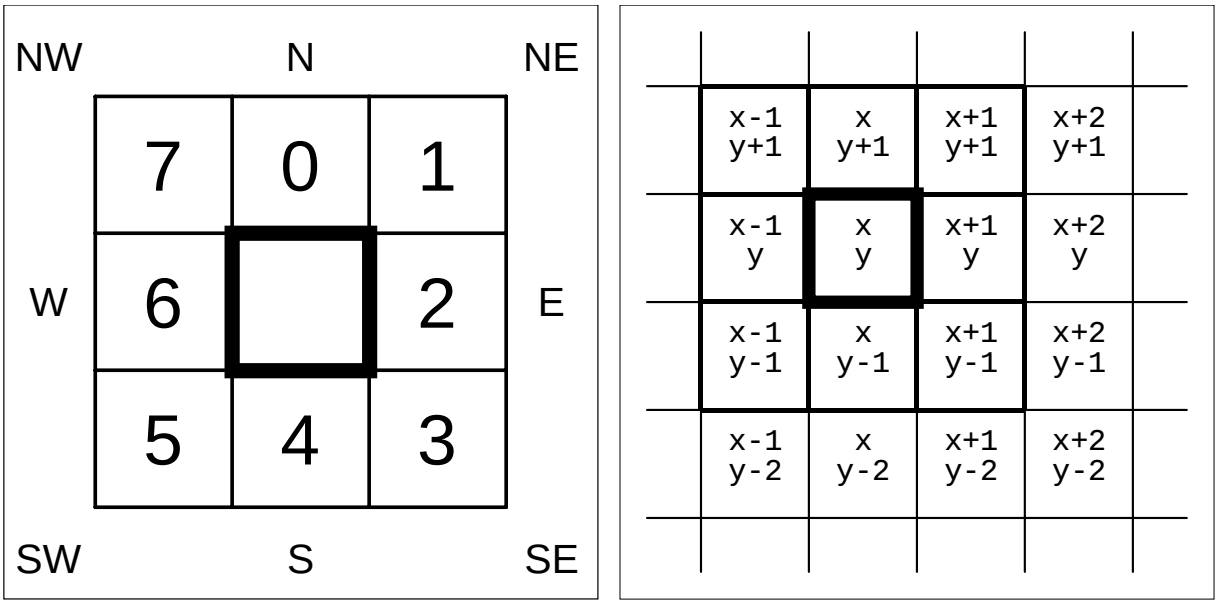
```
bool   CSG_Grid::is_InGrid(int x, int y, bCheckNoData = true)
```

Mit dieser Abfrage wird gleichzeitig sichergestellt, dass nicht mit No-Data Werten gearbeitet wird. Rückgabewert der Funktion ist der Mittelwert für die Rasterzelle an Position *x, y* und ihre unmittelbaren Nachbarn.

```
double CExercise_03::Get_Mean_01(int x, int y)
{
   int      n, i, ix, iy;
   double   s;

   s  = m_pInput->asDouble(x, y);
   n  = 1;

   for(i=0; i<8; i++)
   {
      ix = Get_xTo(i, x);
      iy = Get_yTo(i, y);

      if( m_pInput->is_InGrid(ix, iy, true) )
      {
         s  += m_pInput->asDouble(ix, iy);
         n++;
      }
   }

   return( s / n );
}
```

Die zweite Unterfunktion arbeitet mit einem quadratischen Suchfenster beliebiger Größe. Hier wird eine Doppelschleife verwendet, um alle im Suchfenster enthaltenen Rasterzellen abzufragen (Abb. 58b). Wiederum wird die Funktion *is_InGrid* der *CSG_Grid*-Klasse benutzt, um außerhalb des Rasters gelegene Positionen sowie No-Data Werte von der Berechnung auszuschließen.

```
double CExercise_03::Get_Mean_02(int x, int y, int nCells)
{
   int      n, ix, iy;
   double   s;

   s  = 0.0;
   n  = 0;

   for(iy=y-nCells; iy<=y+nCells; iy++)
   {
      for(ix=x-nCells; ix<=x+nCells; ix++)
      {
         if( m_pInput->is_InGrid(ix, iy, true) )
         {
            s  += m_pInput->asDouble(ix, iy);
            n++;
         }
      }
   }

   return( s / n );
}
```



a) Konvention f. d. Nummerierung der unmittelbaren Nachbarzellen

b) Relative Positionen der Nachbarzellen

Abb. 58: Nachbarschaften im Raster

3.7.4 Wege durchs Raster

Aufgabe	Berechnung der Einzugsgebietsgrößen aus einem Rasterdatensatz mit Höhenwerten	
Lernziel	Wertesortierung, Verwendung rekursiver Funktionen	
Quelltextdateien	exercise_04.cpp, exercise_04.h, exercise_05.cpp, exercise_05.h	
Neue Funktionen	CSG_Module_Grid	Get_NCells
		Set_Progress_NCells
		Get_Length
	CSG_Grid	Assign mit Wert
		Get_Sorted
		Get_Cellarea
		Get_Gradient_NeighborDir
		Get_xFrom(), Get_yFrom
		Add_Value
		*= Operator

Bisher wurden Doppelschleifen benutzt, um auf die Reihen und dann auf die einzelnen Zellen einer Reihe zuzugreifen. Es gibt jedoch eine Vielzahl von Fragestellungen, für die ein solches, sequentielles Abarbeiten der Rasterzellen nicht zu dem gewünschten Ergebnis führt, sondern sich auf anderen Wegen durch das Raster bewegt werden muss. Als Beispiel dient

hier die Berechnung der Einzugsgebietsgrößen aus einem Digitalen Geländemodell. Um diese korrekt zu berechnen, müssen die von Schwerkraft und Geländeform determinierten Abflusswege nachvollzogen werden. Nachfolgend wird das als *Deterministic 8* (kurz D8) bekannt gewordene Verfahren verwendet, bei dem der gesamte Abfluss einer Rasterzelle immer an die Nachbarzelle weitergegeben wird, zu der das größte Gefälle besteht (O'CALLAGHAN & MARK 1986). Zwei grundlegende Möglichkeiten der Implementierung werden in den beiden folgenden Übungen aufgezeigt. Auf Details der Initialisierung und dem Hinzufügen von klasseneigenen Funktionen und Variablen wird ab dieser Übung nicht mehr explizit eingegangen.

Von der höchsten zur tiefsten Rasterzelle

Übung 4 arbeitet das Raster von der höchsten zur tiefsten Rasterzelle ab. Um die Rasterzellen in der nach der Höhe sortierten Reihenfolge anzusprechen, wird die Funktion

```
bool   CSG_Grid::   Get_Sorted(int n, int &x, int &y, bool bDown = true)
```

für das Raster mit den Höhenwerten aufgerufen. Nach dem Funktionsaufruf enthalten die Variablen *x* bzw. *y* die Position der Rasterzelle mit dem Rang *n*. Da diesmal nicht zeilen- und spaltenweise gearbeitet wird, wird die Gesamtzahl aller Rasterzellen, die mit der Funktion

```
int    CSG_Module_Grid::Get_NCells(void)
```

abgefragt wird, für die Definition einer Schleife benutzt, die alle Rasterzellen anspricht. Die entsprechende Funktion für die Fortschrittsanzeige lautet

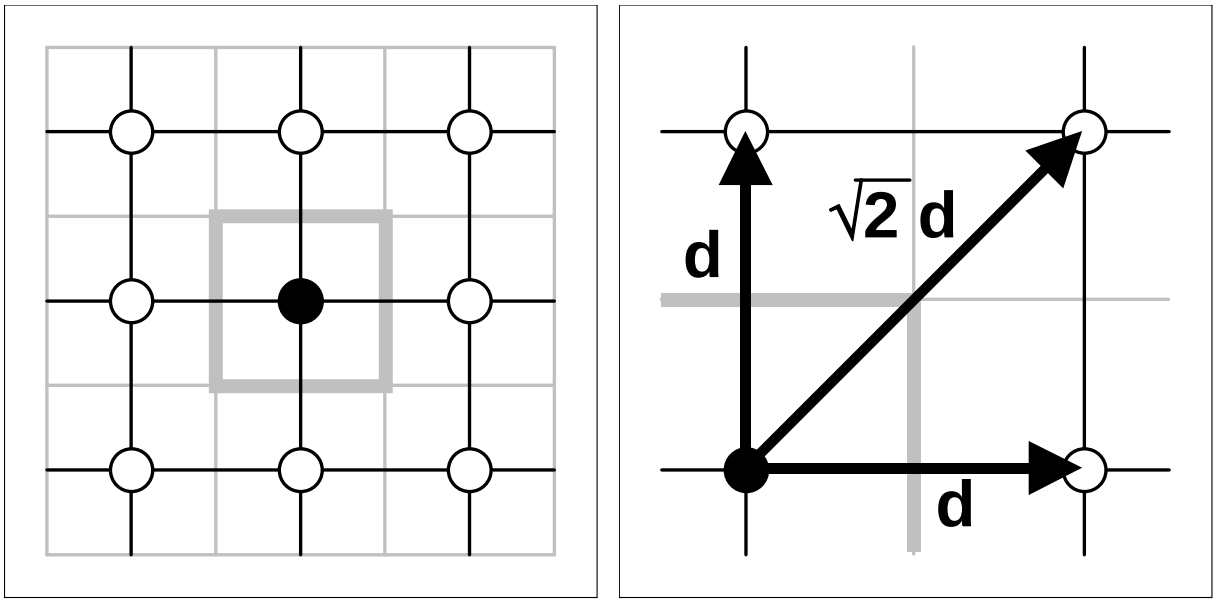
```
bool   CSG_Module_Grid::Set_Progress_NCells(int iCell)
```

Ziel dieser Schleife ist es, für jede Rasterzelle, von der höchsten zur niedrigsten, die Funktion *Set_Area* auszuführen. Doch zunächst sei ein Überblick über die vollständige *On_Execute*-Funktion gegeben:

```
bool CExercise_04::On_Execute(void)
{
   int      n, x, y;

   m_pDEM   = Parameters("INPUT")   ->asGrid();
   m_pArea  = Parameters("OUTPUT")  ->asGrid();

   m_pArea->Assign(0.0);   // Initialise areas

   for(n=0; n<Get_NCells() && Set_Progress_NCells(n); n++)
   {
      m_pDEM->Get_Sorted(n, x, y, true);

      if( m_pDEM->is_NoData(x, y) )
         m_pArea->Set_NoData(x, y);
      else
         Set_Area(x, y);
   }

   if( Parameters("CELLAREA")->asBool() )
      *m_pArea *= m_pArea->Get_Cellarea();

   return( true );
}
```

Damit die Berechnung richtig funktionieren kann, müssen zunächst alle Zellen des Ausgaberasters mit dem Wert 0 initialisiert werden, was hier mit der Funktion

```
bool   CSG_Grid::   Assign(double Value)
```



a) Rasterzellen u. Gitterpunkte b) Distanz zu den Nachbarpunkten

Abb. 59: Eigenschaften von Rasterzellen und Gitterpunkten

erreicht wird. Standardmäßig berechnet das Modul im nachfolgenden die Anzahl der Abfluss liefernden Rasterzellen, nicht jedoch die Gebietsgröße. Um die Umrechnung durchzuführen, wird die Anzahl der Abfluss liefernden Zellen mit der Flächengröße einer Zelle multipliziert. Ein einfacher Weg, dies zu erreichen, ist die Verwendung des Operators

```
void   CSG_Grid::operator *= (double Value)
```

und der Funktion

```
void   CSG_Grid::Get_Cellarea(void)
```

der *CSG_Grid*-Klasse, wodurch jede Zelle des Ausgaberasters mit dem Wert der Zellenfläche multipliziert wird. Da sich der Operator auf das Objekt, nicht jedoch auf einen Zeiger auf das Objekt bezieht, muss *m_pArea* erst mit dem * Operator dereferenziert werden.

Die Funktion *Set_Area* nimmt die eigentliche Abflussverteilung vor. Zuerst erhöht die Funktion die Zahl der Abfluss liefernden Zellen für die aufgerufene Zelle um eins mittels der Funktion

```
void CSG_Grid::Add_Value(int x, int y, double Value)
```

Ein einfaches Setzen des Wertes auf 1 wäre hier falsch, da damit der bereits von höheren Zellen erhaltene Abfluss überschrieben werden würde. Als nächstes muss die Nachbarzelle bestimmt werden, an die der gesammelte Abfluss der Rasterzelle weitergegeben werden soll. Das soll aber nicht einfach die niedrigste Nachbarzelle sein, sondern diejenige, zu der das größte, abwärts gerichtete Gefälle besteht. Die Berechnung erfolgt als Verhältnis der Höhendifferenz zu ihrer Distanz. Die jeweilige Distanz zu den acht Nachbarzellen erhält man mit

```
double CSG_Module_Grid::Get_Length(int Direction)
```

wobei *Direction* die von 0 bis 7 im Uhrzeigersinn durchnummerierten Nachbarzellen bestimmt (Abb. 58a) und gleichbedeutend verwendet wird, wie für die Funktionen *Get_xTo*

bzw. *Get_yTo*. Berechnungsgrundlage sind die den Rasterzellen entsprechenden Gitterpunkte (Abb. 59). Wenn eine Nachbarzelle gefunden wird, dann wird ihr der gesamte Abfluss hinzugefügt. Auf diese Weise kann eine Rasterzelle von mehreren ihrer höher gelegenen Nachbarn Abfluss zugewiesen bekommen, bevor sie selbst prozessiert wird und den Abfluss an eine niedriger gelegene Rasterzelle weiterleitet.

```
void CExercise_04::Set_Area(int x, int y)
{
   int      i, ix, iy, iMax;
   double   z, dz, dzMax;

   m_pArea->Add_Value(x, y, 1);

   z     = m_pDEM->asDouble(x, y);
   dzMax = 0.0;

   for(i=0; i<8; i++)
   {
      ix    = Get_xTo(i, x);
      iy    = Get_yTo(i, y);

      if( m_pDEM->is_InGrid(ix, iy, true) )
      {
         dz    = (z - m_pDEM->asDouble(ix, iy)) / Get_Length(i);

         if( dz > dzMax )
         {
            iMax     = i;
            dzMax = dz;
         }
      }
   }

   if( dzMax > 0.0 )
      m_pArea->Add_Value(Get_xTo(iMax, x), Get_yTo(iMax, y), m_pArea->asInt(x, y));
}
```

Rekursive Funktion

Die Hauptroutine der rekursiven Lösung ähnelt weitgehend der von Übung 4. Anders ist, dass diesmal wieder mit einer Doppelschleife gearbeitet werden kann, um für jede Rasterzelle wenigstens einmal die für die Berechnung zuständige Funktion *Get_Area* aufzurufen.

```
bool CExercise_05::On_Execute(void)
{
   int      x, y;

   m_pDEM   = Parameters("INPUT")   ->asGrid();
   m_pArea  = Parameters("OUTPUT")  ->asGrid();

   m_pArea->Assign(0.0);   // Initialise areas...

   for(y=0; y<Get_NY() && Set_Progress(y); y++)
   {
      for(x=0; x<Get_NX(); x++)
      {
         if( m_pDEM->is_NoData(x, y) )
            m_pArea->Set_NoData(x, y);
         else
            Get_Area(x, y);
      }
   }

   if( Parameters("CELLAREA")->asBool() )
      *m_pArea *= m_pArea->Get_Cellarea();

   return( true );
}
```

Der Rückgabewert der Funktion *Get_Area* ist die Anzahl der Abfluss liefernden Rasterzellen für die aufgerufene Position *x*, *y*. Das Besondere an dieser Funktion ist, dass sie sich mit veränderten Funktionsargumenten gegebenenfalls selbst wieder aufruft, was als *rekursiver* Funktionsaufruf bezeichnet wird. Bei dieser Implementierung kann es vorkommen, dass eine Rasterzelle mehrmals abgefragt wird. Wurde die Funktion für eine abgefragte Position schon einmal aufgerufen, dann enthält die entsprechende Rasterzelle in *m_pArea* bereits die Zahl aller Zufluss liefernden Rasterzellen, einschließlich sich selbst. Der Wert ist also mindestens 1 und kann direkt ohne weitere Prozessierung zurückgegeben werden. Wurde die Position noch nicht prozessiert, dann steht der Wert noch auf 0 und wird als erstes auf 1 gesetzt, was dem Abfluss entspricht, den die Zelle selbst generiert. Danach wird für jede ihrer Nachbarzellen abgefragt, ob sie Abfluss an die Rasterzelle abgibt. Die Funktion

```
int    CSG_Grid::Get_Gradient_NeighborDir(int x, int y)
```

liefert die Richtung der Rasterzelle, zu der sie das größte Gefälle aufweist. In Übung 4 wurde diese Berechnung noch explizit programmiert. Da die Abfragerichtung in dieser Übung entgegengesetzt sein soll, wird statt *Get_xTo*/*Get_yTo*

```
int    CSG_Module_Grid::Get_xFrom(int Direction, int x)
int    CSG_Module_Grid::Get_yFrom(int Direction, int y)
```

benutzt. Ist die so definierte Richtung einer Nachbarzelle gleich der Richtung ihres größten Gefälles, dann ist diese Nachbarzelle Abflusslieferant und es wird für diese Zelle (rekursiv!) ebenfalls die *Get_Area*-Funktion aufgerufen. Der Rückgabewert wird nun zur Gebietsgröße hinzugefügt. Am Ende der Funktion wurde die Anzahl der Abfluss liefernden Rasterzellen für die abgefragte Position vollständig berechnet.

```
int CExercise_05::Get_Area(int x, int y)
{
   int      i, ix, iy;

   if( m_pArea->asInt(x, y) == 0 )  // cell has not been processed yet...
   {
      m_pArea->Set_Value(x, y, 1);  // add this cell's contribution...

      for(i=0; i<8; i++)
      {
         ix = Get_xFrom(i, x);
         iy = Get_yFrom(i, y);

         if( m_pDEM->is_InGrid(ix, iy) && m_pDEM->Get_Gradient_NeighborDir(ix, iy) == i )
            m_pArea->Add_Value(x, y, Get_Area(ix, iy)); // recursive function call...
      }
   }

   return( m_pArea->asInt(x, y) );} // return this cell's area...
```

3.7.5 Dynamische Simulation

Aufgabe	Programmierung des zellulären Automaten „Life“ mit dynamischer Visualisierung	
Lernziel	Prinzipien dynamischer Simulation, Rasterverarbeitung ohne CSG_Module_Grid	
Quelltextdateien	exercise_06.cpp, exercise_06.h	
Neue Funktionen	global	SG_Create_Grid
	CSG_Module	DataObject_Add
		DataObject_Set_Colors
		DataObject_Update
		Process_Get_Okay
		Process_Set_Text
	CColors	Konstruktor
	CSG_Grid	Set_Name
		Get_NX, Get_NY
		Get_System
		asByte
	CSG_Grid_System	Get_xTo, Get_yTo

Dynamische Simulationen sind eine wichtige Methode für die Modellierung von Prozessen. Übung 6 stellt mit dem zellulären Automaten *Life* eine einfache dynamische Simulation vor (s. EIGEN & WINKLER 1987). Zu den Prinzipien der Computer gestützten dynamischen Simulation zählt, dass die Zeit in diskreten Schritten abgebildet wird. Dabei ergibt sich der Systemzustand zum Zeitpunkt $t + 1$ aus dem vorhergehenden Zustand zum Zeitpunkt t.

In dieser Übung soll nicht auf ein bestehendes Rastersystem zurückgegriffen werden. Daher wird im Gegensatz zu den bisherigen Übungen die neue Klasse direkt von *CSG_Module* statt von *CSG_Module_Grid* abgeleitet.

```
class CExercise_06 : public CSG_Module
{
public:
   CExercise_06(void);  // constructor

protected:
   virtual bool   On_Execute  (void);     // always overwrite this function

private:
   CSG_Grid          *m_pLife, *m_pTemp;

   void           Next_Step   (void);
};
```

Zwei Zeigervariablen für *CSG_Grid*-Objekte und die Funktion *Next_Step* werden zusätzlich eingeführt. Im Konstruktor werden nur zwei Ganzzahlwerte für die Anzahl der Spalten und Reihen des Ausgaberasters angefordert.

Die Funktion *On_Execute* initialisiert zunächst das Zielraster sowie ein zu diesem kompatibles Hilfsraster. Danach folgt bereits die Schleife, in der das Setzen der Zeitschritte erfolgt. Vor dem Verlassen der Hauptroutine wird das Hilfsraster wieder gelöscht.

```
bool CExercise_06::On_Execute(void)
{
   int      x, y, n;

   x        = Parameters("NX")->asInt();
   y        = Parameters("NY")->asInt();
   m_pTemp  = SG_Create_Grid(GRID_TYPE_Byte, x, y);  // a temporary grid is needed...
   m_pLife  = SG_Create_Grid(GRID_TYPE_Byte, x, y);
   m_pLife->Set_Name("Life");
   DataObject_Add(m_pLife);
   DataObject_Set_Colors(m_pLife, CColors(2, COLORS_PALETTE_BLACK_WHITE, true));

   for(y=0; y<m_pLife->Get_NY(); y++)    // Initialise life's world...
   {
      for(x=0; x<m_pLife->Get_NX(); x++)
      {
         m_pLife->Set_Value(x, y, rand() > RAND_MAX / 2 ? 0 : 1);
      }
   }

   for(n=1; Process_Get_Okay(true); n++) // Execution...
   {
      Process_Set_Text(CSG_String::Format("%d. Generation", n));
      Next_Step();
   }

   delete(m_pTemp);

   return( true );
}
```

Das Ausgaberaster *m_pLife* wird in Abhängigkeit der vom Benutzer gewünschten Spalten- und Reihenzahl (*NX*, *NY*) mit der Funktion

```
CSG_Grid * SG_Create_Grid(
   TGrid_Type Type,
   int NX, int NY,
   double Cellsize = 0.0,
   double xMin = 0.0, double yMin = 0.0
)
```

neu erstellt. Mit *Type* wird der Datentyp des Rasters festgelegt. Für die Simulation ist eine Datentiefe von einem Byte pro Rasterzelle ausreichend. Mit

```
bool CSG_Grid::Set_Name(const char *String)
```

wird dem Datensatz ein aussagekräftiger Name zugewiesen. Anders als bei den bisherigen Übungen wurde dieser Datensatz nicht von der SAGA-Umgebung bereitgestellt. Damit die SAGA-Umgebung diesen Datensatz im weiteren Verlauf verwalten kann, muss er daher mit

```
bool CSG_Module::DataObject_Add(CSG_DataObject *pObject)
```

bei dieser angemeldet werden. Nach Aufruf dieser Funktion wird liegt die Verwaltung des Datensatzes in der Verantwortung der SAGA-Umgebung. Anders ist es mit dem Hilfsraster *m_pTmp*, das nur für den internen Gebrauch durch das Modul vorgesehen ist. Insbesondere verbleibt es in der Verantwortung des Modulprogrammierers, dieses Objekt vor dem Verlassen der *On_Execute*-Funktion wieder aus dem Arbeitsspeicher zu löschen. Des weiteren soll die für die Darstellung des Ausgaberasters benutzte Farbpalette neu gesetzt werden. Mit

```
CColors::CColors(int nColors, int Palette = COLORS_PALETTE_DEFAULT, bool bRevert =false)
```

wird ein Farbpalettenobjekt erzeugt, das zusammen mit dem Zeiger auf das Ausgaberaster über die Funktion

```
bool CSG_Module::DataObject_Set_Colors(CSG_DataObject *pObject, CColors &Colors)
```

an die SAGA Umgebung übergeben wird.

Bei *Life* haben alle Rasterzellen nur zwei mögliche Zustände, an (=1) oder aus (=0). Die nun folgende Doppelschleife initialisiert alle Rasterzellen durch Verwendung einer Zufallsfunktion mit entsprechenden Werten. Da die speziellen Rasterfunktionen von *CSG_Module_Grid* in diesem Modul nicht verfügbar sind, wird auf die analogen Funktionen der *CSG_Grid*-Klasse zurückgegriffen. Mit

```
int    CSG_Grid::Get_NX  (void)
```

und

```
int    CSG_Grid::Get_NY  (void)
```

erhält man die Anzahl der Spalten und Reihen eines Rasters.

Zur Ermittlung des Zustands zum nächsten Zeitschritt wird der Zustand der Rasterzellen zum aktuellen Zeitpunkt analysiert. Der neue Zustand einer Rasterzelle ergibt sich in Abhängigkeit von der Anzahl seiner an- bzw. ausgeschalteten Nachbarzellen (Abb. 7). Drei Regeln sind hier nur zu befolgen:

- Eine Zelle wird immer angeschaltet, wenn sie genau 3 Nachbarn hat.
- Sie behält ihren Status, wenn sie genau 2 Nachbarn hat.
- In allen anderen Fällen, weniger als 2 oder mehr als 3 Nachbarn, wird sie ausgeschaltet.

Die Dauer der Simulation soll allein vom Benutzer bestimmt werden. Der Abbruch erfolgt, wenn

```
int    CSG_Module::Process_Get_Okay   (bool bBlink = false)
```

den Wahrheitswert *false* zurück gibt. Information über den Fortschritt der Simulation wird mit

```
int    CSG_Module::Process_Set_Text   (const char *Text)
```

angezeigt.

Die Berechnung des Systemzustands zum nächsten Zeitschritt findet in der Funktion *Next_Step* statt. Damit bei der Berechnung der nächsten Generation $t + 1$ die Werte der Generation t nicht vor ihrer Auswertung überschrieben werden, wird hier das Hilfsraster zum Zwischenspeichern des Ergebnisses benutzt. Innerhalb der Doppelschleife zum Abfragen aller Rasterzellen werden zwei Schritte vorgenommen. Im ersten werden die angeschalteten Nachbarzellen ausgezählt, im zweiten wird die abgefragte Zelle im temporären Rasterdatensatz gemäß der Regeln an- bzw. ausgeschaltet.

```
void CExercise_06::Next_Step(void)
{
   int      x, y, i, ix, iy, n;

   for(y=0; y<m_pLife->Get_NY(); y++)
   {
      for(x=0; x<m_pLife->Get_NX(); x++)
      {
         for(i=0, n=0; i<8; i++) // Count neighours...
         {
            ix = m_pLife->Get_System().Get_xTo(i, x);
            iy = m_pLife->Get_System().Get_yTo(i, y);

            if( m_pLife->is_InGrid(ix, iy) && m_pLife->asByte(ix, iy) == 0 )
               n++;
         }

         switch( n )             // Dead or alive...
         {
         case 2:     // Keep status quo...
            m_pTemp->Set_Value(x, y, m_pLife->asByte(x, y));
            break;

         case 3:     // Birth...
            m_pTemp->Set_Value(x, y, 0);
            break;

         default:    // Dead...
            m_pTemp->Set_Value(x, y, 1);
            break;
         }
      }
   }

   m_pLife->Assign(m_pTemp);
   DataObject_Update(m_pLife, true);
}
```

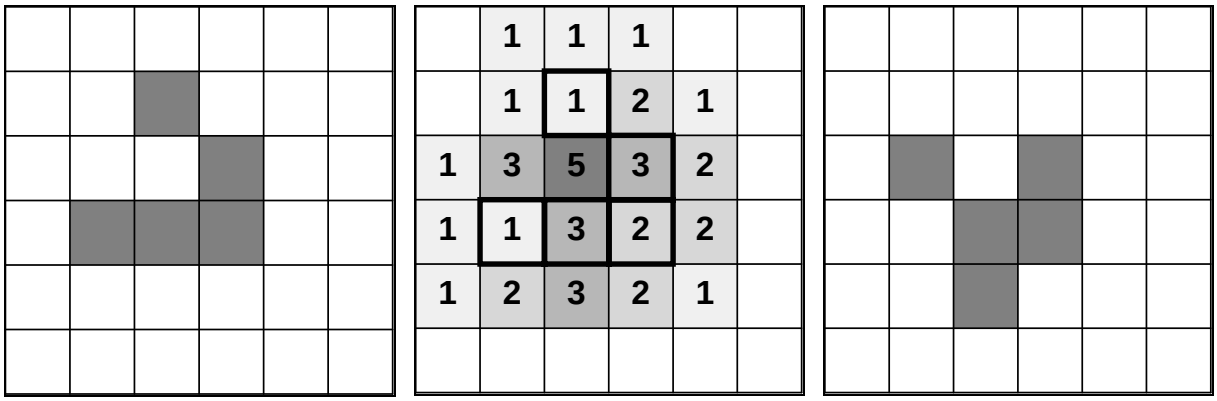
Zugriff auf die bereits bekannten Funktionen *Get_xTo*/*Get_yTo* für die Positionsbestimmung der acht Nachbarzellen bietet das *CSG_Grid_System*-Objekt von *CSG_Grid*, welches man mit der Funktion

```
CSG_Grid_System * CSG_Grid::Get_System(void)
```

erhält. Nachdem alle Zellen des Rasters prozessiert wurden, wird das Ergebnis aus dem Hilfsraster in den Ausgabedatensatz kopiert. Damit die Änderungen am Datensatz von der SAGA-Umgebung aktualisiert dargestellt werden, wird abschließend

```
bool CSG_Module::DataObject_Update(CSG_DataObject *pObject, bool bShow = false)
```

aufgerufen. Auf diese Weise können die sich mit der Zeit verändernden Muster der *Life*-Simulation vom Benutzer verfolgt werden.



a) Generation t b) Anzahl der Nachbarn c) Generation t + 1

Abb. 60: Life - Setzen der nächsten Generation.

3.7.6 Arbeiten mit Vektordaten

Aufgabe	Kopieren eines Vektordatensatzes	
Lernziel	Zugriff auf Vektordaten	
Quelltextdateien	exercise_07.cpp, exercise_07.h	
Neue Funktionen	CSG_Module	Set_Progress
		DataObject_Set_Colors
	CSG_Parameters	Add_Shapes
	CSG_Parameter	asShapes
	CSG_Shapes	Create
		Get_Type
		Get_Table
		Get_Count
		Get_Shape
		Add_Shape
	CSG_Shape	Get_Record
		Get_Part_Count
		Get_Point_Count
		Get_Point
		Add_Point

Das Arbeiten mit Vektordaten unterscheidet sich deutlich von dem mit Rasterdaten. Diese Übung führt in die Vektordatenverarbeitung mit der SAGA-API ein. Da keine Rasterdaten verwendet werden, kann auch für dieses Modul die Basisklasse *CSG_Module* gewählt werden. Im Konstruktor werden über die Parameterliste zwei Vektordatensätze angefordert, von denen einer als Eingabe dient und der andere die Kopie aufnehmen soll.

```
Parameters.Add_Shapes(
   NULL  , "INPUT"   , "Eingabe",
   "Dieser Vektordatensatz soll kopiert und verschoben werden.",
   PARAMETER_INPUT
);

Parameters.Add_Shapes(
   NULL  , "OUTPUT"  , "Ausgabe",
   "In diesen Datensatz soll das Ergebnis geschrieben werden.",
   PARAMETER_OUTPUT
);
```

Zum Anfordern von Vektordatensätzen stellt die CSG_Parameters Klasse mit *Add_Shapes* eine ähnlich wie die bereits eingeführte *Add_Grid*-Funktion aufgebaute Funktion zur Verfügung.

```
CSG_Parameter * CSG_Parameters::Add_Shapes(
   CSG_Parameter *pParent,
   const char *Identifier,
   const char *Name,
   const char *Description,
   int Constraint,
   TShape_Type Shape_Type = SHAPE_TYPE_Undefined
)
```

Mit dem Argument *Shape_Type* kann hier zusätzlich der zulässige Vektordatentyp eingeschränkt werden. Mögliche Werte sind in der API vordefiniert:

```
SHAPE_TYPE_Point
SHAPE_TYPE_Points
SHAPE_TYPE_Line
SHAPE_TYPE_Polygon
SHAPE_TYPE_Undefined
```

In der ausführenden *On_Execute*-Funktion erfolgt der Zugriff auf das angeforderte Vektordatenobjekt mittels

```
CSG_Shapes * CSG_Parameter::asShapes(void)
```

Die Zeiger auf die angeforderten Vektordatensätze werden in den Variablen *pShapes_A* und *pShapes_B* zwischengespeichert. Vor dem Kopieren wird mit

```
bool   CSG_Shapes::Create(TShape_Type Type, const char *Name, CSG_Table *pStructure)
```

der Ausgabedatensatz initialisiert. Mit

```
bool   CSG_Shapes::Get_Type(void)
```

und

```
CSG_Table & CSG_Shapes::Get_Table(void)
```

werden dabei der Vektordatentyp sowie die Struktur der Attributtabelle mit dem Eingabedatensatz synchronisiert. Um die Vektorobjekte von aus dem Eingabe- in den Ausgabedatensatz zu kopieren, muss über alle Objekte des Eingabedatensatzes iteriert werden. Hierfür liefert

```
int    CSG_Shapes::Get_Count(void)
```

die Anzahl der Objekte des Vektordatensatzes. Das einzelne Vektordatenobjekt vom Typ *CSG_Shape* erhält man mit

```
CSG_Shape * CSG_Shapes::Get_Shape(int iShape)
```

Für das Hinzufügen eines neuen Objektes gibt es die Funktion

```
CSG_Shape * CSG_Shapes::Add_Shape(CSG_Table_Record *pValues = NULL)
```

Der Rückgabewert der Funktion ist das neu hinzugefügte Vektorobjekt. Optional kann hier mit dem Funktionsargument *pValues* ein Datensatz mit Attributdaten zum Kopieren übergeben werden. Die Attribute eines Vektorobjekts erhält man mit

```
CSG_Table_Record * CSG_Shape::Get_Record(void)
```

Die geometrischen Daten, also die Koordinaten der einzelnen Stützpunkte, sind zu Punktsequenzen, den so genannten *Parts*, zusammengefasst. Objekte vom Typ *Point* haben immer genau einen *Part*, der wiederum genau einen Punkt enthält. Alle anderen Typen können jedoch aus mehreren Punktsequenzen bestehen, z.B. Polygone, die Aussparungen haben oder sich aus mehreren *Inseln* zusammensetzen. Um alle Stützpunkte eines Vektor-

objekts abzufragen, wird also zuerst eine Schleife gebraucht, die alle Punktsequenzen durchläuft, und darin geschachtelt eine, die alle Stützpunkte der Sequenz anspricht. Die Anzahl der Sequenzen bzw. der Punkte einer Sequenz können mit

```
int     CSG_Shape::Get_Part_Count      (void)
int     CSG_Shape::Get_Point_Count (int iPart)
```

abgefragt werden. Die Koordinaten eines Punkts werden von

```
TSG_Point CSG_Shape::Get_Point(int iPoint, int iPart)
```

in der Datenstruktur *TSG_Point* zurückgegeben, die schließlich die Koordinaten *x*, *y* als Fließkommazahlen enthält. Das Hinzufügen eines Punkts erfolgt mit

```
bool CSG_Shape::Add_Point(TSG_Point Point, int iPart)
```

Während des Kopiervorgangs soll der Benutzer über seinen Fortschritt informiert werden und gegebenenfalls seine Ausführung abbrechen können. Die Funktion

```
bool CSG_Module::Set_Progress(double Position, double Range)
```

ist eine allgemeine Version der bisher benutzten Versionen der *Set_Progress*-Funktion. Die vollständige *On_Execute*-Funktion sollte in etwa so aussehen:

```
bool CExercise_07::On_Execute(void)
{
   int         iShape, nShapes, iPart, iPoint;
   double      dx, dy;
   TSG_Point   Point;
   CSG_Shapes     *pShapes_A, *pShapes_B;
   CSG_Shape      *pShape_A, *pShape_B;

   pShapes_A   = Parameters("INPUT")   ->asShapes();
   pShapes_B   = Parameters("OUTPUT")  ->asShapes();
   dx          = Parameters("DX")      ->asDouble();
   dy          = Parameters("DY")      ->asDouble();

   nShapes     = pShapes_A->Get_Count();
   pShapes_B->Create(pShapes_A->Get_Type(), "Copy and Shift", &pShapes_A->Get_Table());

   for(iShape=0; iShape<nShapes && Set_Progress(iShape, nShapes); iShape++)
   {
      pShape_A = pShapes_A->Get_Shape(iShape);
      pShape_B = pShapes_B->Add_Shape(pShape_A->Get_Record()); // copy attributes...

      for(iPart=0; iPart<pShape_A->Get_Part_Count(); iPart++)
      {
         for(iPoint=0; iPoint<pShape_A->Get_Point_Count(iPart); iPoint++)
         {
            Point    = pShape_A->Get_Point(iPoint, iPart);

            Point.x  += dx;   // perform the translation before
            Point.y  += dy;   // you add the point to the new shape...

            pShape_B->Add_Point(Point, iPart);
         }
      }
   }

   return( true );
}
```

3.7.7 Verknüpfen von Raster- und Vektordaten

Aufgabe	Vektorisierung von Abflusslinien aus einem Rasterdatensatz mit Höhenwerten, Hinzufügen von Rasterwerten zu den Attributen eines Vektordatensatzes	
Lernziel	Verknüpfen von Vektor- u. Rasterdaten, Verwendung von Tabellen	
Quelltextdateien	exercise_08.cpp, exercise_08.h, exercise_09.cpp, exercise_09.h	
Neue Funktionen	CSG_Parameters	Add_Grid_List
	CSG_Parameter	asGridList
	CSG_Parameter	asShapes
	CSG_Parameter_Grid_List	Get_Count
		asGrid
	CSG_Table	Get_Field_Count
		Add_Field
	CSG_Table_Record	Set_Value
	CSG_Shapes	Assign
	CSG_Shape	Get_Record
	CSG_Grid_System	Get_Grid_To_World
	CSG_Grid	Get_Name
		is_InGrid_byPos
		Get_Value

Es folgen zwei Übungen, die aufzeigen, wie Vektordaten mit Rasterdaten über räumliche Zusammenhänge miteinander verknüpft werden können.

Ableitung von Vektordaten aus einem Rasterdatensatz

Die in Übung 4 und 5 eingeführte Methode Deterministic 8 soll in dieser Übung dazu benutzt werden, einen Vektordatensatz mit Abflusslinien von einem Rasterdatensatz mit Höhenwerten abzuleiten. Als Basisklasse wird *CSG_Module_Grid* gewählt, so dass wieder bequem auf spezielle rasterbezogene Methoden zugegriffen werden kann. Der neuen Klasse wird die Variable *m_pFlow_Dir* vom Typ eines Zeigers auf ein *CSG_Grid*-Objekt hinzugefügt, die während der Modulausführung auf einen Rasterdatensatz mit der Abflussrichtung zeigen soll, sowie eine Funktion *Get_Flow_Line*, die weiter unten detailliert vorgestellt wird. Der vollständige Header „*exercise_08.h*“ sieht folgendermaßen aus:

```
class CExercise_08 : public CSG_Module_Grid
{
public:
   CExercise_08(void);  // constructor

protected:
   virtual bool   On_Execute     (void);     // always overwrite this function

private:
   CSG_Grid          *m_pFlow_Dir;

   void           Get_Flow_Line  (int x, int y, CSG_Shape *pLine, int ID);
};
```

Im Konstruktor wird über die Parameterliste ein Rasterdatensatz mit Höhenwerten als Eingabe und ein Vektordatensatz vom Typ Linie für die Ausgabe angefordert.

```
   Parameters.Add_Grid(
      NULL, "INPUT"  , "Höhe",
      "Dieser Datensatz enthält die Höhenwerte.",
      PARAMETER_INPUT
   );

   Parameters.Add_Shapes(
      NULL, "OUTPUT" , "Abflusslinien",
      "In diesen Datensatz werden die Abflusslinien gespeichert.",
      PARAMETER_OUTPUT, SHAPE_TYPE_Line
   );
```

Vorab sei die *On_Execute*-Funktion vollständig abgebildet:

```
bool CExercise_08::On_Execute(void)
{
   int      n, x, y;
   CSG_Grid    *pDEM;
   CSG_Shapes  *pLines;

   pDEM     = Parameters("INPUT")   ->asGrid();
   pLines   = Parameters("OUTPUT")  ->asShapes();

   pLines->Create(SHAPE_TYPE_Line, "Abflusslinien");
   pLines->Get_Table().Add_Field("ID"    , TABLE_FIELDTYPE_Int);
   pLines->Get_Table().Add_Field("LENGTH", TABLE_FIELDTYPE_Double);

   m_pFlow_Dir = SG_Create_Grid(pDEM, GRID_TYPE_Char);

   for(y=0; y<Get_NY() && Set_Progress(y); y++)   // Initialise flow direction matrix...
   {
      for(x=0; x<Get_NX(); x++)
         m_pFlow_Dir->Set_Value(x, y, pDEM->Get_Gradient_NeighborDir(x, y));
   }

   for(n=0; n<Get_NCells() && Set_Progress_NCells(n); n++) // Get the flow lines...
   {
      pDEM->Get_Sorted(n, x, y, true);

      if( m_pFlow_Dir->asInt(x, y) >= 0 )
         Get_Flow_Line(x, y, pLines->Add_Shape(), pLines->Get_Count() + 1);
   }

   delete(m_pFlow_Dir); // Clean up...
   return( true );
}
```

Nach dem Auslesen der angeforderten Parameter wird der Vektordatensatz initialisiert. Der zunächst leeren und undefinierten Attributtabelle werden mit

```
bool CSG_Table::Add_Field(const char *Name, TTable_FieldType Type)
```

zwei Felder vom Datentyp

```
TABLE_FIELDTYPE_Int      Ganzzahl
```

bzw.

```
TABLE_FIELDTYPE_Double       Fließkommazahl
```

hinzugefügt. Ein temporärer Rasterdatensatz mit den Abflussrichtungen jeder Rasterzelle wird erstellt und der Zeiger auf ihn in *m_pFlow_Dir* gespeichert. Dieser Datensatz wird nicht an die SAGA-Umgebung übergeben und muss vor dem Verlassen der Funktion wieder gelöscht werden. Die hierin gespeicherten Abflussrichtungen entsprechen der bereits in Übung 4 und 5 verwendeten Nummerierungskonvention (Abb. 58a). In einer weiteren Schleife werden nun vom höchsten zum niedrigsten Höhenwert die Positionen aller Rasterzellen abgefragt. Unter der Bedingung, dass die abgefragte Rasterzelle einen gültigen Wert für die Abflussrichtung enthält, wird dann die Funktion *Get_Flow_Line* aufgerufen, die die eigentliche Vektorisierung vornimmt und dem Ausgabedatensatz zunächst ein neues Linienobjekt hinzufügt. Dann folgt *Get_Flow_Line* solange den jeweiligen Abflussrichtungen durch das Raster, bis die Abflussrichtung aus dem Raster herausführt oder einen ungültigen Wert besitzt. Dabei fügt sie dem Linienobjekt jede Rasterzellenposition als neuen Stützpunkt hinzu. Da die Position als Spalten- und Zeilenkoordinaten vorliegen, werden diese vorher mit der Funktion

```
TSG_Point CSG_Grid_System::Get_Grid_To_World(int x, int y)
```

in Weltkoordinaten konvertiert. Damit jede Abflusslinie nur einmal vektorisiert wird, wird die Abflussrichtung der aktuellen Position vor dem Wechsel auf die neue Position auf einen ungültigen Wert gesetzt (:=-1). Wenn keine gültige Abflussrichtung mehr gefunden wird, werden abschließend die berechnete Linienlänge und eine Identifikationsnummer mit

```
bool   CSG_Table_Record::Set_Value(int iField, double Value)
```

in die Attributliste des Linienobjekts eingetragen.

```
void CExercise_08::Get_Flow_Line(int x, int y, CSG_Shape *pLine, int ID)
{
   int      Flow_Dir;
   double   Length;

   if( (Flow_Dir = m_pFlow_Dir->asInt(x, y)) >= 0 )
   {
      Length   = 0.0;
      pLine->Add_Point(m_pFlow_Dir->Get_System()->Get_Grid_to_World(x, y));

      do
      {
         m_pFlow_Dir->Set_Value(x, y, -1);  // mark as processed
         x        += Get_xTo(Flow_Dir);
         y        += Get_yTo(Flow_Dir);
         Length   += Get_Length(Flow_Dir);
         pLine->Add_Point(Get_System()->Get_Grid_to_World(x, y));
      }
      while( m_pFlow_Dir->is_InGrid(x, y) && (Flow_Dir = m_pFlow_Dir->asInt(x, y)) >= 0 );

      pLine->Get_Record()->Set_Value(0, ID);
      pLine->Get_Record()->Set_Value(1, Length);
   }
}
```

Hinzufügen von Rasterwerten zu einem Vektordatensatz

In Übung 9 sollen die Attribute eines Punktdatensatzes um die an den jeweiligen Punktkoordinaten befindlichen Werte beliebig vieler Rasterdatensätze ergänzt werden. Die Rasterdatensätze sollen nicht auf ein bestimmtes Rastersystem festgelegt sein. Daher wird als Basisklasse wieder *CSG_Module* benutzt. Im Konstruktor wird eine variable Liste mit Rasterdatensätzen mit der Funktion

```
CSG_Parameter * CSG_Parameters::Add_Grid_List(
   CSG_Parameter *pParent,
   const char *Identifier,
   const char *Name,
   const char *Description,
   int Constraint, bool bSystem_Dependent = false
)
```

angefordert (Abb. 61a). Der Zugriff auf diese Liste erfolgt später mit

```
CSG_Parameter_Grid_List * CSG_Parameter::asGridList(void)
```

Die für die Eingabe und Ausgabe benötigten Vektordatensätze werden auf den Typ *Point* eingeschränkt. Über eine Auswahlliste wird festgelegt, wie die Rasterwerte interpoliert werden sollen.

```
Parameters.Add_Shapes(
   NULL, "SHAPES"    , "Punkte",
   "",
   PARAMETER_INPUT, SHAPE_TYPE_Point
);

Parameters.Add_Grid_List(
   NULL, "GRIDS"     , "Raster",
   "",
   PARAMETER_INPUT, false
);

Parameters.Add_Shapes(
   NULL, "RESULT"    , "Punkte mit Rasterwerten",
   "",
   PARAMETER_OUTPUT, SHAPE_TYPE_Point
);

Parameters.Add_Choice(
   NULL, "INTERPOL"  , "Raster Interpolation",
   "",
   "Nearest Neighbor|"
   "Bilinear Interpolation|"
   "Inverse Distance Interpolation|"
   "Bicubic Spline Interpolation|"
   "B-Spline Interpolation|", 4
);
```

Bei der Ausführung wird nach dem Auslesen der Parameterliste zunächst der als Eingabe dienende Punktdatensatz mittels

```
bool   CSG_Shapes::Assign(CSG_Shapes *pShapes)
```

in den Ausgabedatensatz kopiert. Dessen Attributtabelle wird zusätzlich um die Einträge für die Rasterwerte erweitert. Die Anzahl der Rasterdatensätze liefert

```
int    CSG_Parameter_Grid_List::Get_Count(void)
```

Zugriff auf das jeweilige Raster erhält man mit

```
CSG_Grid *  CSG_Parameter_Grid_List::asGrid(int iGrid)
```

Die ursprüngliche Anzahl der Attributeinträge, die mit der Funktion

```
int    CSG_Table::Get_Field_Count(void)
```

abgefragt wird, wird für die spätere Bestimmung der Attributfelder für der hinzuzufügenden Rasterwerte in *nFields* gespeichert. Für jeden Rasterdatensatz wird nun mit

```
void   CSG_Table::Add_Field(const char *Name, TTable_FieldType Type);
```

ein weiteres Attributfeld ergänzt. Unter Verwendung von

```
const char * CSG_Grid::Get_Name(void)
```

wird dem Feld der Name des Rasterdatensatzes zugewiesen. Nun werden in der folgenden Schleife die Rasterwerte für jeden Punkt ausgelesen. Die Schleife durchläuft alle Punkte des Punktdatensatzes und sucht sich dann in der darin geschachtelten inneren Schleife die Werte für alle Rasterdatensätze. Dazu wird für jeden Rasterdatensatz mit der Funktion

```
bool   CSG_Grid::is_InGrid_byPos(TSG_Point Point)
```

abgefragt, ob die Position des aktuellen Punktes in das Raster fällt. Ist das der Fall, wird der Wert des Rasterdatensatzes für die Position mit

```
double CSG_Grid::Get_Value(TSG_Point Point, int Interpolation)
```

abgerufen und in die Attributliste des Punktes eingetragen. Da die abgefragte Position bei dieser Methode des Datenzugriffs i.d.R. zwischen das Rastergitter fällt, bietet diese Funktion mit *Interpolation* verschiedene Interpolationsmethoden an.

```
bool CExercise_09::On_Execute(void)
{
   int                     iPoint, nPoints, iGrid, nFields, Interpolation;
   double                  Value;
   TSG_Point               Point;
   CSG_Shapes              *pPoints;
   CSG_Shape               *pPoint;
   CSG_Parameter_Grid_List *pGrids;

   pGrids         = Parameters("GRIDS")      ->asGridList();
   pPoints        = Parameters("RESULT")     ->asShapes();
   Interpolation  = Parameters("INTERPOL")   ->asInt();

   if( pPoints != Parameters("SHAPES")->asShapes() ) // Initialise points...
   {
      pPoints->Assign(Parameters("SHAPES")->asShapes());
   }

   nPoints  = pPoints->Get_Count();
   nFields  = pPoints->Get_Table().Get_Field_Count();

   for(iGrid=0; iGrid<pGrids->Get_Count(); iGrid++)
   {
      pPoints->Get_Table().Add_Field(
         pGrids->asGrid(iGrid)->Get_Name(), TABLE_FIELDTYPE_Double
      );
   }

   for(iPoint=0; iPoint<nPoints && Set_Progress(iPoint, nPoints); iPoint++)
   {
      pPoint   = pPoints->Get_Shape(iPoint);
      Point    = pPoint ->Get_Point(0);  // Point shapes always have exactly one point...

      for(iGrid=0; iGrid<pGrids->Get_Count(); iGrid++)
      {
         if( pGrids->asGrid(iGrid)->is_InGrid_byPos(Point) )
            Value = pGrids->asGrid(iGrid)->Get_Value(Point, Interpolation, true);
         else
            Value = -99999;

         pPoint->Get_Record()->Set_Value(nFields + iGrid, Value);
      }
   }

   return( true );
}
```

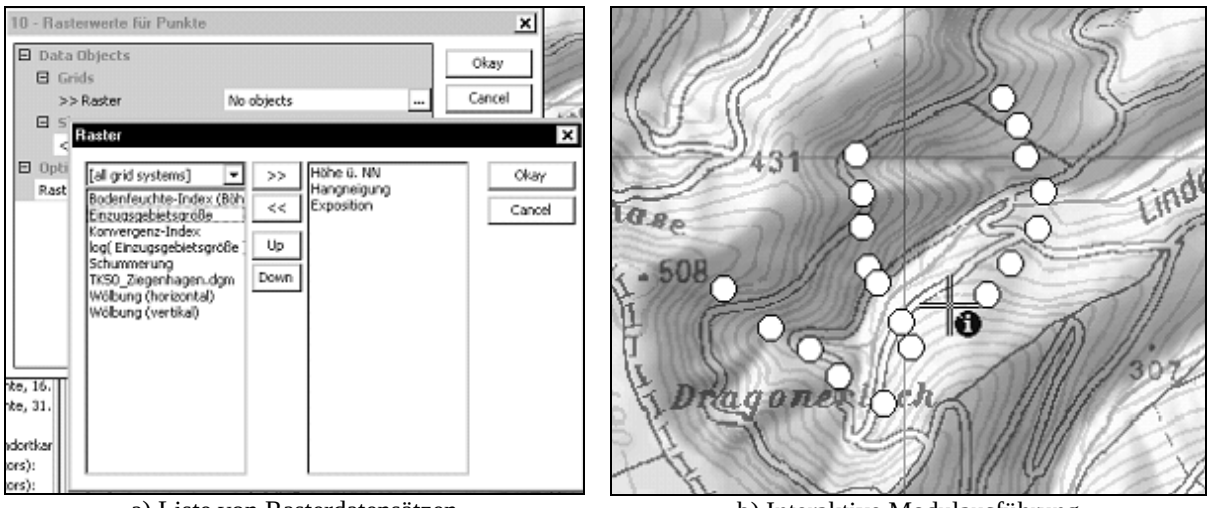
3.7.8 Interaktive Ausführung

Aufgabe	Interaktives Hinzufügen von Punkten zu einem Vektordatensatz	
Lernziel	Erstellung interaktiver Module	
Quelltextdateien	exercise_10.cpp, exercise_10.h	
Neue Funktionen	CSG_Module_Interactive	On_Execute_Position

Übung 10 ist weitgehend identisch mit Übung 9, nur dass diesmal kein Vektordatensatz vorgegeben wird, sondern dass die Punkte interaktiv vom Benutzer mit Hilfe der Mausfunktionen angelegt werden. Der wichtigste Unterschied besteht in der Verwendung der Basisklasse *CSG_Module_Interactive*. Zusätzlich zu *On_Execute* wird die von *CSG_Module_Interactive* geerbte *On_Execute_Position*-Funktion überschrieben. Ähnlich der *On_Execute*-Funktion wird *On_Execute_Position* automatisch von der SAGA-Umgebung aufgerufen, allerdings immer dann, wenn mit der Maus eine Aktion in einer Karte ausgeführt wird. Für das Modul werden noch drei klasseninterne Variablen zum Speichern der Interpolationsmethode, der Rasterdatensätze sowie des Punktdatensatzes deklariert.

```
class CExercise_10 : public CSG_Module_Interactive
{
public:
   CExercise_10(void);

protected:
   virtual bool   On_Execute          (void);
   virtual bool   On_Execute_Position  (CSG_Point ptWorld, TSG_Module_Interactive_Mode Mode);

private:
   int                     m_Interpolation;
   CSG_Shapes              *m_pPoints;
   CSG_Parameter_Grid_List *m_pGrids;
};
```

Im Gegensatz zu Übung 9 kann hier auf einen Punktdatensatz für die Eingabe verzichtet werden. Ansonsten ist der Konstruktor identisch. Bei interaktiv arbeitenden Modulen dient die *On_Execute*-Funktion vor allem der Initialisierung. Die eigentliche Ausführung findet in



a) Liste von Rasterdatensätzen b) Interaktive Modulausführung

Abb. 61: Verknüpfen von Raster- u. Vektordaten

der *On_Excecute_Position*-Funktion statt. Auch die Initialisierung gleicht der von Übung 9, nur dass diesmal ein leerer Punktdatensatz erstellt wird, der lediglich Attributfelder für die Rasterwerte enthält. Nach Ausführung von *On_Execute* reagiert das Modul auf jede Mauseingabe, bis der Benutzer die interaktive Ausführung des Moduls wieder beendet.

```
bool CExercise_10::On_Execute(void)
{
   m_pGrids         = Parameters("GRIDS")      ->asGridList();
   m_pPoints        = Parameters("RESULT")     ->asShapes();
   m_Interpolation  = Parameters("INTERPOL")   ->asInt();

   m_pPoints->Create(SHAPE_TYPE_Point, "Rasterwerte");

   for(int iGrid=0; iGrid<m_pGrids->Get_Count(); iGrid++)
   {
      m_pPoints->Get_Table().Add_Field(
         m_pGrids->asGrid(iGrid)->Get_Name(), TABLE_FIELDTYPE_Double
      );
   }

   return( true );
}
```

Die *On_Execute_Position*-Funktion wird mit zwei Funktionsargumenten aufgerufen. *ptWorld* enthält die Weltkoordinaten des Ereignisses und *Mode* informiert über den Status der Maus. Mögliche Werte für *Mode* sind von der API vordefiniert und umfassen u.a.:

```
MODULE_INTERACTIVE_LDOWN       linke Maustaste wird gedrückt
MODULE_INTERACTIVE_LUP         linke Maustaste wird losgelassen
MODULE_INTERACTIVE_LDCLICK     Doppelklick auf linker Maustaste
MODULE_INTERACTIVE_RDOWN       rechte Maustaste wird gedrückt
MODULE_INTERACTIVE_MOVE_LDOWN  Maus wird bewegt bei gedrückter linker Maustaste
```

In dieser Übung wird nur reagiert, wenn die linke Maustaste gedrückt wird. Ist das der Fall, dann wird dem Punktdatensatz ein Punkt mit den Koordinaten von *ptWorld* hinzugefügt und die Attributliste des Punktes mit den entsprechenden Werten der Rasterdatensätze gefüllt. Damit die Änderung am Punktdatensatz von der Benutzeroberfläche unmittelbar reflektiert wird, wird abschließend die Funktion *DataObject_Update* aufgerufen (Abb. 61b).

```
bool CExercise_10::On_Execute_Position(CSG_Point ptWorld, TSG_Module_Interactive_Mode Mode)
{
   int         iGrid;
   double      Value;
   CSG_Shape   *pPoint;

   if( Mode == MODULE_INTERACTIVE_LDOWN )
   {
      pPoint   = m_pPoints->Add_Shape();
      pPoint->Add_Point(ptWorld);

      for(iGrid=0; iGrid<m_pGrids->Get_Count(); iGrid++)
      {
         if( m_pGrids->asGrid(iGrid)->is_InGrid_byPos(ptWorld) )
            Value = m_pGrids->asGrid(iGrid)->Get_Value(ptWorld, m_Interpolation, true);
         else
            Value = -99999;

         pPoint->Get_Record()->Set_Value(iGrid, Value);
      }

      DataObject_Update(m_pPoints, false);
   }

   return( true );
}
```

4 AUSBLICK

Es ist mehr oder weniger eine Gesetzmäßigkeit, dass eine Software, die sich nicht weiterentwickelt, früher oder später nicht mehr benutzt wird. Das liegt zum einen daran, dass sich verändernde Rahmenbedingungen, wie das Betriebssystem, die verwendeten Programmbibliotheken oder die aufkommenden Datenmengen, ständige Softwareanpassungen erzwingen. Auf der anderen Seite ist eine Software so gut wie nie wirklich völlig fehlerfrei oder ausgereift. Und selbst wenn sie es ist, wird immer eine Nachfrage dafür bestehen, weitere Optimierungen vorzunehmen oder neue benutzerfreundliche Funktionalitäten zu ergänzen. In Teil 4 wurde bereits an einigen Stellen auf fehlende oder wünschenswerte Funktionalitäten hingewiesen. Zur Zeit ist SAGA noch eine junge Software, die gerade an der Schwelle steht, eine Nutzergemeinde zu entwickeln, die groß genug ist, um ein verwertbares Feedback zu erhalten, sei es durch Kritik an seiner praktischen Einsatzfähigkeit, durch Weiterempfehlungen an andere potentielle Nutzer oder durch aktive Beteiligung an seiner Weiterentwicklung. In welche Richtung und in welcher Geschwindigkeit SAGA sich weiterentwickelt, hängt also entscheidend davon ab, wie verbreitet es eingesetzt wird und welche Rückkopplungen sich durch die Nutzer von SAGA ergeben. Das Konzept von SAGA ist so offen angelegt, dass eine Weiterentwicklung in viele Richtungen möglich ist. Aber wie bereits in Kapitel 1 dargestellt wurde, wird SAGA zur Zeit noch von einer relativ kleinen Kerngruppe in erster Linie für die Lösung projektspezifischer, vorwiegend wissenschaftlicher Fragestellungen entwickelt. Zwar führte der wissenschaftliche Leitgedanke, entwickelte Methoden der Allgemeinheit zur Überprüfung und Anwendung zur Verfügung zu stellen, schließlich zur Veröffentlichung von SAGA als Open Source Software. Trotzdem wird seine Weiterentwicklung voraussichtlich auch mittelfristig vor allem von den Interessen der AG Geosystemanalyse geprägt sein. Diese fächern sich jedoch sehr weit und betreffen sowohl die universitäre Forschung und Lehre, als auch die kommerzielle Verwertbarkeit, so dass die nachfolgend vorgestellten Ideen sicher auch auf ein breiteres Interesse stoßen werden.

4.1 Weiterentwicklung von SAGA

Ein obligatorisches Element jeder Softwareentwicklung ist die Identifizierung und Beseitigung von Programmierfehlern, sogenannten Bugs. Ein aktuelles Ziel ist es daher immer eine möglichst fehlerfreie Version zu erstellen. Hier, wie auch bei der Integration einer neuen oder verbesserten Benutzerführung, sind vor allem die Nutzer von SAGA gefragt, entsprechende *Bugs* und *Feature Requests* anzumelden, die auch schon einmal in Form einer Rezension erfolgen kann (z.B. TRAUN 2005). Abgesehen davon ist das Hauptanliegen von SAGA, die Sammlung an wissenschaftlichen Methoden in den SAGA-Modulen weiterzuentwickeln und auszubauen. Hier können neue API-Funktionen dazu dienen, neue Themengebiete durch Module erschließbar zu machen. Damit sich der Kreis von beitragenden Modulprogrammierern wiederum erweitert, müssen auch die API-Funktionen besser dokumentiert werden, sei es durch Programmiereinführungen, wie die in Kapitel 3.7, oder durch eine direkt in den Quelltext integrierte API-Dokumentation. Eine weitere Popularitätssteigerung ist von der

Erstellung von Installationspaketen für Linux sowie deren Platzierung in den verschiedenen, besonders bei wissenschaftlichen Programmierern beliebten Linux-Distributionen zu erwarten. Für die meisten Anwender ist aber selbstredend die Bedienbarkeit der graphischen Benutzeroberfläche das entscheidende Kriterium für die Wahl einer Software. Von den vielen Baustellen in der GUI, wie der benötigte Ausbau von Menüs und Werkzeugleisten einschließlich des Designs graphischer Schaltflächen, das geradezu künstlerische Qualitäten verlangt, ist der Schwerpunkt sicher in den kartographischen Fähigkeiten des Programms zu sehen. Neben erweiterten Möglichkeiten insbesondere bei der Vektordarstellung, wie Schraffuren und Beschriftung, fehlt vor allem eine funktionsreiche Schnittstelle für die Gestaltung der Druckausgabe. Die greifbarsten Pläne für die Weiterentwicklung API betreffen die Bereitstellung neuer Datensatztypen und den Aufbau einer Schnittstelle für Skriptsprachen.

Änderungen und Ergänzungen der in SAGA verfügbaren Datensatztypen betreffen in erster Linie die API, haben aber auch Auswirkungen auf die GUI, die entsprechende Verwaltungs- bzw. Darstellungsmöglichkeiten anbieten muss. Eine kleinere Ergänzung ist die Berücksichtigung von drei- oder gegebenenfalls mehrdimensionalen Stützpunkten für das für das jetzige Vektordatenmodell, wie es bereits vom Shapefile-Format unterstützt wird (ESRI 1998). Darüber hinaus sollen zwei neue Datensatztypen die jetzigen zukünftig ergänzen. Ein Multi-Raster soll die Verwaltung und Analyse von Multispektraldaten, Zeitreihen und Volumen ermöglichen. Hierfür ist sinnvoll, sowohl ein spezielles Dateiformat als auch eine spezielle Visualisierungsfunktion bereit zu stellen. Mit einem topologischen Vektormodell (Kap. 2.2.2) soll eine Alternative zum bestehenden objektorientierten Vektormodell bereit gestellt werden. Ähnlich wie ein TIN, haben Vektortopologien intelligente Elemente, Knoten, Kanten und Flächen, die wissen, in welcher Beziehung sie zu anderen Elementen des Datensatzes stehen. So lässt sich mit einem topologischen Vektormodell eine Netzwerkanalyse z.B. wesentlich einfacher durchführen als mit dem aktuellen Vektormodell. An die Visualisierung würden hierdurch keine wesentlich neuen Anforderungen entstehen.

Skriptsprachen ermöglichen die externe Steuerung von Anwendungsprogrammen und dienen z.B. der Automatisierung von wiederkehrenden Aufgaben. In Kapitel 3.5 wurde bereits gezeigt, wie der SAGA-Kommandozeileninterpreter für die Automatisierung komplexer Arbeitsschritte von einem Skript aus aufgerufen werden kann. Die Verwendung des Kommandozeileninterpreters ist jedoch auf den Aufruf von Modulen beschränkt und der Datensatzzugriff erfolgt ausschließlich dateibasiert, wobei nur die von der SAGA-API direkt unterstützten Dateiformate verwendet werden können. Daher soll die API in Zukunft durch spezielle Schnittstellen für die Verwendung mit Skriptsprachen geöffnet werden. Hierfür bietet sich der Schnittstellen-Compiler SWIG als ein geeignetes Werkzeug an, das eine Anbindung verschiedenster skriptfähiger Sprachen (u.a. Java, Perl, PHP, Python) für in C/C++ geschriebene Programme und Bibliotheken ermöglicht (SWIG 2006).

Ein längerfristiges Ziel ist die Erstellung einer integrierten und flexiblen Datenbankschnittstelle, so dass SAGA Datensätze nicht mehr ausschließlich dateibasiert speichert,

sondern als Client auf zentrale oder netzwerkverteilte Datenbankserver zugreifen kann. Wie der Zugriff auf SQL-fähige Datenbanken durch ein Modul erfolgen kann, wurde bereits in Kapitel 3.6.1 am Beispiel von Tabellen gezeigt. Eine vollständige Unterstützung für die Anbindung an Datenbanken über die Benutzeroberfläche verlangt jedoch sehr viel mehr. So muss z.B. bekannt sein, in welcher Struktur der Raumbezug in einer Datenbank vorliegt. Für die Datenobjekte der API wird es wünschenswert sein, dass sie die Daten nicht mehr vollständig im Arbeitsspeicher des Clientrechners vorhalten, sondern diese nur bei Bedarf von dem Datenbankserver beziehen bzw. Änderungen an diesen übermitteln.

In der Summe benötigen die soeben skizzierten Entwicklungspotentiale für ihre Umsetzung ein erhebliches Maß an Arbeitskraft, die sich nur durch eine gezielte Förderung von Entwicklungsvorhaben einerseits und die Beteiligung weiterer enthusiastischer Entwickler andererseits mittelfristig erlangen lässt. Ein wichtige Schlüsselfunktion hierfür ist sicherlich in der Weiterverbreitung von SAGA zu sehen.

4.2 Weiterverbreitung von SAGA

Da SAGA eine freie, kostenlos beziehbare Software ist, stehen direkt keine Mittel für die Finanzierung seiner Weiterentwicklung zur Verfügung. Für die Weiterentwicklung von SAGA wird auf einen für erfolgreiche FOSS-Projekte typischen, positiven Rückkopplungseffekt gesetzt (Kap. 2.1.5). Je attraktiver und verbreiteter ein Projekt ist, desto mehr wird seine Entwicklung gefördert. Abgesehen von interessierten Hobbyprogrammierern, die bereitwillig ihre Freizeit opfern, kann eine sinnvolle Weiterentwicklung vor allem im Rahmen von Projekten stattfinden, die besondere, noch nicht implementierte Anforderungen an SAGA haben. Letztlich wird die gesamte SAGA-Nutzergemeinde von jedem gemachten Beitrag profitieren. Vor diesem Hintergrund hat sich die SAGA User Group e.V. mit der primären Zielsetzung gegründet, die Verbreitung von SAGA zu fördern (Kap. 2.2). Das erste Mittel dieses Ziel zu erreichen ist die vom Verein geförderte Öffentlichkeitsarbeit, zu der neben dem Ausbau der Internetpräsenz und Präsentationen auf Messen und Tagungen vor allem die einmal im Jahr stattfindenden, international gehaltenen Nutzertreffen sind. In 2006 fand das Nutzertreffen im Rahmen der vom Zentrum für Geoinformatik Salzburg (Z_GIS 2006) veranstalteten Fachmesse für Angewandte Geoinformatik (AGIT) statt und wurde für den Erfahrungsaustausch und Präsentationen über den Einsatz von SAGA in Forschungsprojekten genutzt, die zum überwiegenden Teil in einem Sonderband der Göttinger Geographischen Abhandlungen zusammengefasst sind (BÖHNER et. al 2006).

Ein weiterer Ansatzpunkt ist die Verankerung von SAGA in der universitären wie auch der schulischen Lehre, wo SAGA als einfach zu bedienende, kostenlose und kompetente Software für viele Lehrinhalte eine sinnvolle Alternative zu den kommerziellen Standardprogrammen sein kann (CONRAD 2005). In diesem Zusammenhang wird derzeit ein Lehrmodul für die vom Z_GIS betriebene eLearning Plattform UNIGIS entwickelt, über die Fernstudienlehrgänge mit dem Schwerpunkt Geoinformatik angeboten werden. Ergänzt werden soll das Lehrmodul durch das zusätzliche Angebot von Blockseminaren für die Präsenzlehre (Z_GIS 2006).

Am wichtigsten für die aktuelle Weiterentwicklung sind zur Zeit Beiträge, die im Rahmen von Forschungsprojekten gemacht werden. Neben der Entwicklung neuer Module (z.B. HECKMANN & BECHT 2006, WICHMANN 2006), ergeben sich hier auch Beiträge für die Funktionalitäten der API und GUI, wie z.B. die Ergänzung einer Schnittstelle zum Erstellen von Dokumenten (Kap. 3.3.3) im Rahmen des SEXTANTE Projekts (Kap. 1.3). Besonders hervorzuheben ist momentan das von der Bayrischen Forschungsstiftung geförderte Projekt Geoinformationstechnologien für standorteffiziente Pflanzenproduktion (GeoSteP 2006), bei dem SAGA auf verschiedenen Ebenen für die Geodatenprozessierung eingesetzt wird, die für ein *Precision Farming*, also ein teilflächenspezifisches, landwirtschaftliches Standortmanagement benötigt wird (Kap. 1.1). Das Einsatzgebiet reicht dabei von der einmaligen Erstellung räumlich differenzierter Datensätze von Bodenmerkmalen bis zur zeitnahen Vorhersage produktionsrelevanter Eigenschaften, wie dem Wassergehalt des Bodens oder des Reifezustands der Anbaufrucht. Hierfür müssen zum einen vorhandene Methoden weiterentwickelt und durch neue ergänzt werden. Die Anforderung, zeitnahe Vorhersagen z.B. auf Anfrage einer Webserver-Applikation automatisiert durchführen zu lassen, macht aber auch zusätzliche Entwicklungen des Systems selbst erforderlich. Der täglichen Auswertung aktueller Klimadaten, die mehr oder weniger große Räume mit einer einmaligen Prozessierung abdeckt, steht dabei die teilflächenspezifische Erstellung der eigentlichen Zieldatensätze, z.B. in Form von Düngeempfehlungen, gegenüber. Die Modulausführung kann für solche Aufgaben nicht mehr effizient durch den manuellen Aufruf der Benutzeroberfläche erfolgen und soll stattdessen von einer Serverapplikation ausgeführt werden. Um SAGA-Module flexibel von einer Serverapplikation aus ansteuern zu können, ist im Rahmen des Projekts der Aufbau einer Schnittstelle für Skriptsprachen eines der vorrangigen Ziele für die Weiterentwicklung des Gesamtsystems. Im Gegensatz zu den Modulen, die für dieses Projekt entwickelt werden, muss eine Skriptsprachenanbindung für die unter der LGPL stehenden SAGA-API ebenfalls als Open Source veröffentlicht werden (Kap. 2.1.3), so dass hiervon die gesamte SAGA-Nutzergemeinde profitieren kann.

Es sollte damit deutlich geworden sein, dass die Verbreitung von SAGA eine wichtige Triebfeder für seine Weiterentwicklung ist. Das Hinzufügen einer neuen, nützlichen Funktionalität, wie die der Skriptsprachenanbindung, wird das System wiederum attraktiver machen und neue Nutzer an SAGA heranführen, die sich gegebenenfalls selbst wieder an einer Weiterentwicklung beteiligen werden. In diesem Sinne und der Überzeugung, dass es sich um ein wertvolles Werkzeug handelt, von dem nicht nur die Wissenschaft, sondern auch die Gesellschaft als Ganzes profitieren kann, steht zu hoffen, dass SAGA zukünftig einen zunehmend breiteren Raum in der Anwendung geowissenschaftlicher Methoden einnehmen wird.

5 ZUSAMMENFASSUNG

Das im Zentrum dieser Arbeit stehende System für Automatisierte Geowissenschaftliche Analysen (SAGA) entstand aus der praktischen Notwendigkeit, neue Methoden für die Analyse von Geodaten entwickeln und in die Anwendung bringen zu müssen. Um die Durchführung von Forschungsprojekten effizient zu unterstützen zu können, wurden eine Reihe von Anforderungen an das System formuliert und in der Folge umgesetzt.

Eine Anwendungsprogrammierschnittstelle (API) bildet das Fundament des Systems und versorgt insbesondere den Programmierer von Methoden mit einem komfortablen Objektmodell für die Verarbeitung von Raster- wie auch Vektordaten. Methoden werden als Module in dynamisch ladbaren, vom Kernsystem getrennten Programmbibliotheken abgelegt. Die hierdurch erzielte Modularisierung ermöglicht die parallele, voneinander unabhängige Methodenentwicklung durch verschiedene Programmierer und kommt so einer Aufteilung von Entwicklungsaufgaben in größeren Projekten sehr entgegen. Für die Ausführung von Modulen bietet das System zur Zeit zwei alternative Benutzerschnittstellen. Die graphische Benutzeroberfläche (GUI) erlaubt eine weitgehend intuitive Bedienbarkeit des Systems. Zu ihren Aufgaben gehört neben der Modulausführung die Verwaltung und Visualisierung von Geodaten. Die zweite Benutzerschnittstelle ist ein Kommandozeileninterpreter (CMD), der ausschließlich für die Ausführung von Modulen konzipiert wurde. Der CMD lässt sich aus Skriptdateien heraus aufrufen, so dass komplexe Arbeitsabläufe, die sich aus nacheinander geschalteten Aufrufen verschiedener Module zusammensetzen, weitergehend automatisiert werden können.

Die Implementierung des Systems erfolgte in der weit verbreiteten, objektorientierten Programmiersprache C++, die auf der einen Seite ein maschinennahes Programmieren und dadurch bedingt diverse Optimierungen und Performanzsteigerungen ermöglicht, auf der anderen Seite aber auch eine strukturierte Programmierung durch die Entwicklung einfach zu benutzender Objektmodelle unterstützt. Die SAGA-API bietet neben Objektmodellen für Geodaten zahlreiche Funktionalitäten, die bei der Programmierung von Modulen hilfreich sind, z.B. für das Lösen mathematischer Aufgaben oder die Erstellung von Dokumentdateien. Da für betriebssystemspezifische Aufgaben auf die plattformübergreifende C++ Klassenbibliothek wxWidgets zurückgegriffen wurde, ist SAGA sowohl unter Windows als auch unter Linux Betriebssystemen lauffähig.

Das Kernsystem, bestehend aus API, GUI und CMD sowie eine große Zahl an Modulen wurden unter eine Free Open Source Software (FOSS) Lizenz gestellt und mitsamt ihren Quelltexten veröffentlicht. Quelltexte, ausführbare Programmdateien und weitere Informationen über das System können von der Internetadresse http://sourceforge.net/saga-gis bezogen werden. Allein die freie Verfügbarkeit von SAGA weckte bereits ein breites Interesse unter Anwendern, die mit der Verarbeitung von Geodaten beschäftigt sind. Dank der Quelltextoffenheit und der effizienten Gestaltung der API ergeben sich seitdem positive Rückkopp-

lungseffekte für die Weiterentwicklung des Systems auch durch Beiträge von Entwicklern einer internationalen SAGA-Nutzergemeinde.

Derzeit sind etwa 230 Module in der frei verfügbaren SAGA-Distribution enthalten. Die Bandbreite der Methoden reicht dabei von einfachen Hilfsfunktionen bis zu komplexen Analysen und Simulationsmodellen. Module für den Datenimport und –Export erlauben den Datenaustausch mit anderen Programmen. Von grundlegender Bedeutung ist die Projektion von Geodaten in verschiedene Koordinatensysteme. Zahlreiche Standardfunktionen stehen für die Prozessierung von Tabellen, Raster- und Vektordaten zur Verfügung. Als methodische Schwerpunkte sind neben der allgemeinen Verarbeitung von Rasterdaten aber vor allem geostatistische und reliefanalytische Verfahren zu nennen, die letztendlich wieder zum Ausgangspunkt der SAGA-Entwicklung zurückführen. Zunächst bestand dieser in der Entwicklung neuer reliefanalytischer Methoden für die Unterstützung von Bodenkartierungen. Daraus entwickelte sich jedoch sehr schnell ein breiter angelegtes Konzept, das über die reliefbasierte Abbildung von Prozessen u.a. zur Regionalisierung von Bodeneigenschaften mit Hilfe geostatistischer Verfahren führte. SAGA und die unter ihm implementierten Methoden bilden seitdem eine Basis für die erfolgreiche Durchführung zahlreicher Forschungsprojekte. Aber auch für den Einsatz in der Lehre hat SAGA sich bereits erfolgreich bewährt, insbesondere da es eine kostengünstige Alternative zu vergleichbarer kommerzieller Software darstellt.

Die SAGA-Entwicklung ist keineswegs abgeschlossen und dank seiner offen gehaltenen Systemarchitektur sind viele Weiterentwicklungen denkbar. In welche Richtung sich SAGA entwickeln wird, hängt entscheidend davon ab, wie verbreitet es eingesetzt wird und welche Anforderungen dabei an das System gestellt werden. Dass sich das System weiterentwickeln wird, scheint jedoch ziemlich sicher zu sein.

6 SUMMARY

The System for Automated Geoscientific Analyses (SAGA), which is in the focus of this work, originates from the practical need to develop new methods for the analysis of spatial data and to apply them immediately. To efficiently support the work in research projects a number of demands has been outlined and successively been implemented.

An application programming interface (API) forms the system's foundation and supplies particularly the programmer of methods with a comfortable object model for the processing of raster as well as vector data. Methods are placed as modules in dynamically loadable program libraries, which are separated from the core system. Due to this modularisation a parallel and independent development of methods by different programmers becomes possible, which e.g. enables a splitting of development tasks in larger projects. For the execution of modules the system offers currently two alternative user interfaces. The graphical user interface (GUI) allows to a large extent an intuitive usage of the system. Besides the execution of modules its tasks include the management and visualisation of spatial data. The second user interface is a command line interpreter (CMD), which has been designed exclusively for module execution. The CMD can be executed from a script file, so that a further automation of complex work flows, which are composed of different successive calls to modules, becomes possible.

The implementation of the systems has been done using the wide spread object oriented programming language C++, which on the one hand gives an extensive control over the computer management and therefore allows diverse optimisations and performance gains. On the other hand C++ supports a structured programming by the construction of easily usable object models. Besides object models for spatial data the SAGA-API offers numerous helpful functionalities for the programming of modules, e.g. for solving mathematical problems or for the creation document files. Operating system specific functions are accessed using the platform independent C++ class library wxWidgets. Therefore SAGA works under Windows as well as Linux operating systems.

The system's core, consisting of API, GUI and CMD, as well as numerous modules have been published together with their source codes under a Free Open Source Software (FOSS) license. Source codes, executable programs and further information about the system can be obtained from the internet site at http://sourceforge.net/saga-gis. The free availability itself aroused a broad interest among users, who have to deal with spatial data, and the opening of the sources and the efficient design of the API resulted in a positive feedback of the young SAGA user community.

Currently there are about 230 modules included in the free available SAGA distribution. The spectrum of methods covers simple tools as well as complex analyses and simulation models. Modules for the data import and export enable the data exchange with other software. Fundamental capabilities are modules for the projection of spatial data into different coordinate systems. Many standard functions are provided for the work with tables, raster and vector data. But besides the general processing of raster data, the methodological focus has to

be seen in the modules for terrain analysis and geostatistics, which point back to the origin of the SAGA development, when it was the objective to derive terrain parameters and classifications based on digital elevation models for the support of soil mapping. This was the starting point for a rapid development of a wider methodological framework, which lead from the terrain based modelling of processes to the regionalisation of soil properties with the help of geostatistical procedures. Since then SAGA and its implemented methods form a basis for the successful execution of various research projects. But SAGA proved itself for educational purposes too, not at least because it is a free of charge alternative to comparable commercial software.

The SAGA development is not at all finished yet. Due to its open system architecture many future developments are possible. In which direction SAGA will develop depends very much on the number of its users, their demands and their will to support the further system development. One thing seems to be certain anyway: a further development of the system will take place.

7 LITERATUR UND QUELLEN*

ANDREWS, P.L. (1986): BEHAVE: Fire Behavior Prediction and Fuel Modeling System - Burn Subsystem, Part 1. U.S. Department of Agriculture, Forest Service General, Technical Report INT-194.

BAHRENBERG, G., GIESE, E. & NIPPER, J. (1990): Statistische Methoden in der Geographie 1 – Univariate und bivariate Statistik. Stuttgart, 233S.

BAHRENBERG, G., GIESE, E. & NIPPER, J. (1992): Statistische Methoden in der Geographie 2 – Multivariate Statistik. Stuttgart, 415S.

BAUER, J., ROHDENBURG, H. & BORK, H.-R. (1985): Ein Digitales Reliefmodell als Vorraussetzung für ein deterministisches Modell der Wasser- und Stoff-Flüsse. in: BORK, H.-R. & ROHDENBURG, H. [Hrsg.]: Parameteraufbereitung für deterministische Gebiets-Wassermodelle – Grundlagenarbeiten zur Analyse von Agrar-Ökosystemen. Landschaftsgenese und Landschaftsökologie, 10:.1-15.

BANDEMER, H. & GOTTWALD, S. (1993): Einführung in Fuzzy-Methoden. Berlin, 264S.

BARTELME, N. (1993): Geoinformatik – Modelle, Strukturen, Funktionen. Berlin, 414S.

BEHRENS, T. & SCHOLTEN, T. (2006): Digital Soil Mapping in Germany – a Review, Journal of Plant Nutrition and Soil Sciences, 169: 434-443.

BEVEN, K., KIRKBY, M.J., SCHOFIELD, N. & TAGG, A.F. (1984): Testing a Physically-Based Flood Forecasting Model (TOPMODEL) for Three U.K. Catchments. Journal of Hydrology, 69: 119-143.

BEVEN, K. (1997): TOPMODEL – A Critique. Hydrological Processes, 11: 1069-1085.

BÖHNER, J. (2006): J.Böhner in der Abteilung für Physische Geographie, Univ. Göttingen. HP: http://www.geogr.uni-goettingen.de/pg/.

BÖHNER, J. & KÖTHE, R. (2003): Neue Ansätze zur Bodenregionalisierung. Petermanns Geographische Mitteilungen, 147: 72-82.

BÖHNER, J., KÖTHE, R., CONRAD, O. & RINGELER, A. (1998): Weiterentwicklung der Programmsysteme SARA und SADO zur Regionalisierung bodenkundlich relevanter

* Wegen der großen Bedeutung von Internetressourcen für diese Arbeit wurden neben den üblichen Literaturhinweisen, die sich auf Druckmedien beziehen, auch Verweise auf Webseiten von Projekten und Firmen in die Literaturliste integriert. Diese Verweise sind durch die Abkürzung „HP“ (für Homepage) gekennzeichnet. Für Dokumente, die ausschließlich im Internet veröffentlicht sind, werden die entsprechenden Quellenverweise mit der Abkürzung „DL“ (für Download) angegeben. Die Jahresangaben für Internetquellen beziehen sich, sofern auf der jeweiligen Internetseite kein anderes Datum angegeben ist, auf den letzten Aufruf (in allen Fällen das aktuelle Jahr).

Geofaktoren. - Forschungsbericht zum Vertrag Nr. 2-812045 vom 16.9.1998 zwischen der Bundesanstalt für Geowissenschaften und Rohstoffe (BGR) und der Georg-August-Universität Göttingen. Göttingen, 16 S. [Geogr. Inst. Univ. Göttingen, unveröff.]

BÖHNER, J., KÖTHE, R., CONRAD, O. & RINGELER, A. (1999): Reliefanalyse, Klimaregionalisierung, Prozessparametrisierung und Bodenregionalisierung. Forschungsbericht zum Vertrag Nr. 2-99005843 vom 6.8.1999 zwischen der Bundesanstalt für Geowissenschaften und Rohstoffe (BGR) und der Georg-August-Universität Göttingen. Göttingen, 89 S. [Geogr. Inst. Univ. Göttingen, unveröff.]

BÖHNER, J., KÖTHE, R. CONRAD, O., GROSS, J., RINGELER, A. & SELIGE, T. (2002): Soil Regionalisation by Means of Terrain Analysis and Process Parameterisation. In: MICHELI, E., NACHTERGAELE, F. & MONTANARELLA, L. [Hrsg.]: Soil Classification 2001. European Soil Bureau, Research Report No.7, EUR 20398 EN, Luxembourg. S.213-222.

BÖHNER, J., KÖTHE, R. & TRACHINOW, C. (1997): Weiterentwicklung der automatischen Reliefanalyse auf der Basis von Digitalen Geländemodellen. Göttinger Geographische Abhandlungen, 100: 3-21.

BÖHNER, J., MCCLOY, K.R. & STROBL, J. [Hrsg.] (2006): SAGA – Analysis and Modelling Applications. Göttinger Geographische Abhandlungen, 115:130S.

BÖHNER, J. & PÖRTGE, K.H.. (1997): Strahlungs- und expositionsgesteuerte tagesperiodische Schwankungen des Abflusses in kleinen Einzugsgebieten. Petermanns Geographische Mitteilungen, 141: 35-42.

BÖHNER, J., SCHÄFER, W., CONRAD, O., GROSS, J. & RINGELER, A. (2003): The WEELS Model: Methods, Results and Limitations. Catena, 52: 289-308.

BÖHNER, J. & SELIGE, T. (2006): Spatial Prediction of Soil Attributes Using Terrain Analysis and Climate Regionalisation. in: BÖHNER, J., MCCLOY, K.R. & STROBL, J. [Hrsg.]: SAGA – Analysis and Modelling Applications. Göttinger Geographische Abhandlungen, 115: 13-27.

BRONSTEIN, I.N., SEMENDJAJEW, K.A., MUSIOL, G. & MÜHLIG, H. (1997): Taschenbuch der Mathematik. Frankfurt, 1082S.

BURROUGH, P.A. & MCDONNEL, R.A. (1998): Principles of Geographic Information Systems. Oxford, 346S.

CLARKE, K.C. & GAYDOS, J. (1998): Loose-Coupling a Cellular Automaton Model and GIS: Long-Term Urban Growth Prediction for San Francisco and Washington/Baltimore. International Journal of Geographic Information Science, 12/7: 699-714.

CONRAD, O. (1998): Die Ableitung hydrologisch relevanter Reliefparameter am Beispiel des Einzugsgebietes Linnengrund. Diplomarbeit, Geographisches Institut der Universität Göttingen, DL: http://www.geogr.uni-goettingen.de/saga/digem/download/.

CONRAD, O. (2002): DiGeM – Software for Digital Terrain Analysis. HP: http://www.geogr.uni-goettingen.de/pg/saga/digem/.

CONRAD, O. (2005): Digitale Reliefanalyse in der multimedialen Lehre. Arbeitsberichte des Geographischen Instituts der Humboldt-Universität zu Berlin, 109: 37-47.

CONRAD, O. (2006): SAGA – Program Structure and Current State of Implementation. in: BÖHNER, J., MCCLOY, K.R. & STROBL, J. [Hrsg.]: SAGA – Analysis and Modelling Applications. Göttinger Geographische Abhandlungen, 115: 39-52.

COSTA-CABRAL, M. & BURGES, S.J. (1994): Digital Elevation Model Networks (DEMON): A Model of Flow Over Hillslopes for Computation of Contributing and Dispersal Areas, Water Resources Research, 30/6: 1681-1692.

DVWK - DEUTSCHER VERBAND FÜR WASSERWIRTSCHAFT UND KULTURBAU E.V. (1996): Ermittlung der Verdunstung von Land- und Wasserflächen. DVWK Merkblätter 238, Bonn, 135S.

DEWDNEY, A.K. (1984): Wa-Tor - An Ecological Simulation of Predator-Prey Populations, Scientific American, H.12.

DEWDNEY, A.K. (1989): Wellen aus der Computer-Retorte. Spektrum der Wissenschaft – Computer-Kurzweil III, S.38-41.

DEWHURST, S.C. & STARK, K.S. (1990): Programmieren in C++. München, 265S..

DIKAU, R. (1988): Entwurf einer geomorphographisch-analytischen Systematik von Reliefeinheiten. Heidelberger Geographische Bausteine, 5: 1-45.

DONATO G. & BELONGIE, S. (2002): Approximation Methods for Thin Plate Spline Mappings and Principal Warps, In HEYDEN, A., SPARR, G., NIELSEN, M. & JOHANSEN, P. [Hrsg,]: Computer Vision – ECCV 2002: 7th European Conference on Computer Vision, Copenhagen, Denmark, May 28–31, 2002, Proceedings, Part III, Lecture Notes in Computer Science. Heidelberg, S.21-31.

EIGEN, M. & WINKLER, R. (1985): Das Spiel - Naturgesetze steuern den Zufall. München, 404S.

ETZRODT, N., ZIMMERMANN, R. & CONRAD, O. (2002): Upscaling Water Cycle Parameters Using Geomorphometric Terrain Parameters and Topographic Indices Derived from Interferometric DEM. In: WILSON, A. [Hrsg.]: Proceedings of the 3rd International Symposium on Retrieval of Bio- and Geophysical Parameters from SAR Data for Land Applications, 11-14 September, 2001 in Sheffield, UK. S.251-254.

ESRI (1998): ESRI Shapefile Technical Description, 34S. DL: http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf.

ESRI (2006): ESRI - Environmental Systems Research Institute. HP: http://www.esri.com.

EVENDEN, G.I. (2003): Cartographic Projection Procedures – Release 4. 42S., DL: ftp://ftp.remotesensing.org/proj/new_docs/proj.4.3.pdf.

FARR, T.G. & M. KOBRICK (2000): Shuttle Radar Topography Mission Produces a Wealth of Data. American Geophysical Union Eos, 81: 583-585.

FAIRES, D.J. & BURDEN, R.L. (1994): Numerische Methoden. Heidelberg, 630S.

FAIRFIELD, J. & LEYMARIE, P. (1991): Drainage Networks from Grid Digital Elevation Models, Water Resources Research, 27: 709-717.

FIRE.ORG (2006): Public Domain Software for the Wildland Fire Community, HP: http://fire.org/.

FISCHER, J. & AHRENS, K. (1996):. Objektorientierte Prozeßsimulation in C++. Bonn, 360S.

FORGY, E. (1965): Cluster Analysis of Multivariate Data: Efficiency vs. Interpretability of Classifications. Biometrics, 21: 768-769.

FREEMAN, G.T. (1991): Calculating Catchment Area with Divergent Flow Based on a Regular Grid. Computers and Geosciences, 17: 413-22.

FREE SOFTWARE FOUNDATION (2004): Was ist freie Software? DL: http://www.fsfeurope.org/documents/freesoftware.de.html.

FUHRMANN-KOCH, M. (2005): Göttinger Wissenschaftler präsentieren IT-Neuentwicklungen auf der CeBIT 2005. 2S. DL: http://www.gwdg.de/aktuell/presse/pi-cebit-2005.pdf.

GEOSTEP (2006): GeoSteP - Geoinformationstechnologien für standorteffiziente Pflanzenproduktion. HP: http://www.geostep.de.

GIORDANO, R., HAASE, D., LIERSCH, S., TIMMERMAN, J. & VURRO, M. (2006): Role of Information and New Concepts for Adaptive Monitoring Systems. Ecology and Society, http://www.ecologyandsociety.org [in Vorbereitung].

GNU (2006): GNU is not UNIX. HP: http://www.gnu.org.

GÖRLICH, C.F. & HUMBERT, L. (2005): Open Source – Die Rückkehr der Utopie? In: LUTTERBERG, B., GEHRING, R.A. & BÄRWOLFF, M. [Hrsg.]: Open Source Jahrbuch 2005 – Zwischen Softwareentwicklung und Gesellschaftsmodell. Berlin, S.311-327.

GOLDEN SOFTWARE (2006): Golden Software, Inc.. HP: http://www.goldensoftware.com.

GPS BABEL (2006): GPS BABEL. HP: http://www.gpsbabel.org/.

Grams, T. (1992): Simulation, strukturiert und objektorientiert programmiert. Mannheim, 222S.

GRASS (2006): GRASS – Geographical Resources Analysis Support System. HP: http://grass.itc.it/.

GRASSMUCK, V. (2002): Freie Software – zwischen Privat- u. Gemeineigentum. Bundeszentrale für politische Bildung, DL: http://freie-software.bpb.de/, 440S.

HEERDEGEN, R.G. & BERAN, M.A. (1982): Quantifying Source Areas through Land Surface Curvature. Journal of Hydrology, 57: 359-373.

HELMERS, S. & SEIDLER, K. (1994): Linux: Cooperative Software Development and Internet. Proceedings of the 1st Dutch International Symposium on Linux – December 1994 at Amsterdam, S.56-59.

HERRLICH, O. & LINDNER, U. (1981): Strukturierte Programmierung. Leipzig, 168S.

HIRAOKA, T. (2006): GPX2SHP. HP: http://gpx2shp.sourceforge.jp..

HOPFER, R. (1987): Mikrorechentechnik – allgemeinverständlich: Integrierte Schaltkreise, Mikroprozessoren, Programme. Leipzig, 152S.

HUGGET, R.J. (1993): Modelling the Human Impact on Nature. Oxford, 202S.

JOHNSON, D.L. & MILLER, A.C. (1997): A Spatially Distributed Hydrological Model Utilizing Raster Data Structures. Computers & Geosciences, 23: 267-272.

KAPPAS, M. (2001): Geographische Informationssysteme. Braunschweig, 315S.

KERNIGHAN, B.W. & RITCHIE, D.M. (1978): The C Programming Language. New Jersey, 274S.

KLINGSEISEN, B. (2004): GIS Based Generation of Topographic Attributes for Landform Classification. Diplomarbeit, FHT Kärnten, Villach, DL: http://www.fh-kaernten.at/cms/geo/dateien/Geo_DA_2004_KLingseisen.pdf.

KOPPERS, L. (2006): SAGA GIS 2.0 – Einführung. Ityco – Freie Schulungsunterlagen für Lehrende und Lernende, DL: http://www.ityco.com/sonstige_programme/saga_gis.php.

KÖTHE, R., GEHRT, E. & BÖHNER, J. (1996): Automatische Reliefanalyse für geowissenschaftliche Anwendungen – derzeitiger Stand und Weiterentwicklung des Programms SARA. Arbeitshefte Geologie, 1/1996: 31-37.

KÖTHE, R. & LEHMEIER, F. (1993): SARA – Ein Programmsystem zur Automatischen Relief-Analyse. Zeitschrift für Angewandte Geographie, 4/1993: 11-21.

KRÜGER, J.P. (2006): Waldkonversion und Bodendegradation im tropischen Tiefland von Ostbolivien. Dissertation, Geographisches Institut der Universität Göttingen, 181S.

LEA, N.L. (1992): An Aspect Driven Kinematic Routing Algorithm. in: PARSONS, A.J. & ABRAHAMS, A.D. [Hrsg.]: Overland Flow: Hydraulics and Erosion Mechanics. London, S.147-175.

LEE, D.T. & SCHACHTER, B.J. (1980): Two Algorithms for ConstruCSG_TINg a Delaunay Triangulation. International Journal of Computer and Information Sciences, 9/3: 219-242.

LEE, S., WOLBERG, G. & SHIN, S.Y. (1997): Scattered Data Interpolation with Multilevel B-Splines, IEEE Transactions on Visualisation and Computer Graphics, 3/3: 228-244.

LEICA GEOSYSTEMS (2006): ERDAS IMAGINE. HP: http://gi.leica-geosystems.com/.

LOUDEN, K.C. (1994): Programmiersprachen: Grundlagen, Konzepte, Entwurf. Bonn, 810S.

LOWES, M. & PAULIK, A. (1992): Programmieren mit C – ANSI Standard. Stuttgart, 272S.

LUO, W. (2000): Quantifying Groundwater-Sapping Landforms with a Hypsometric Technique. Journal of Geophysical Research, 105: 1685-1694.

LUTTERBERG, B., GEHRING, R.A. & BÄRWOLFF, M. [Hrsg.] (2005): Open Source Jahrbuch 2005 – Zwischen Softwareentwicklung und Gesellschaftsmodell. Berlin, 508S.

MANDELBROT, B.B. (1983): The Fractal Geometry of Nature. New York, 490S.

MCCLOY, K.R. (2006): Resource Management Information Systems: Remote Sensing, GIS and Modeling. 575S.

MEISELS, A., RAIZMAN, S. & KARNIELI, A. (1995): Skeletonizing a DEM into a Drainage Network. Computers and Geosciences, 21/1: 187-196.

MITASOVA, H. & MITAS, L. (1993): Interpolation by Regularized Spline with Tension : I. Theory and Implementation. Mathematical Geology,. 25: 641-655.

MOORE, I.D., GRAYSON, R.B. & LADSON, A.R. (1992): Digital Terrain Modelling: A Review of Hydrological, Geomorphological and Biological Applictations. Hydrological Processes, 5: 7-35.

MORE, J., GARBOW, B. & HILLSTROM, K. (1999): Minpack – Software for Solving Nonlinear Equations and Nonlinear Least Squares Problems. DL: http://www.netlib.org/minpack/.

MORISSETTE, D. (2006): E00Compr. HP: http://avce00.maptools.org/e00compr/index.html.

NACCACHE, N.J. & SHINGHAL, R. (1984): An Investigation into the Skeletonization Approach of Hilditch. Pattern Recognition, 17: 279-284.

NASA (2006): Mars Global Surveyor (MGS) – Mars Orbiter Laser Altimeter (MOLA). National Aeronautics and Space Administration (NASA), HP: http://pds-geosciences.wustl.edu/missions/mgs/mola.html

NATIONAL GEOSPATIAL AGENCY (2006): GeoTrans – Geographic Translator. HP: http://earth-info.nga.mil/GandG/geotrans/.

NETELER, M. (2003): GRASS-Handbuch, Version 1.1. 266S. DL: http://www.gdf-hannover.de.

NEWATER (2006): New Approaches to Adaptive Water Management Under Uncertainty. http://www.newater.info.

O'CALLAGHAN, J.F. & MARK, D.M. (1984): The Extraction of Drainage Networks from Digital Elevation Data. Computer Vision, Graphics and Image Processing, 28: 323-344.

OGC (2006): OGC – Open Geospatial Consortium. HP: http://www. opengeospatial.org.

OLAYA, V. (2004)::A Gentle Introduction to SAGA GIS. 216S. DL: http://sourceforge.net/saga-gis/.

OLAYA, V. & CONRAD, O. (2006): Geomorphometry in SAGA. In: HENGL, T. & REUTER, H.I. [Hrsg.]: Geomorphometry: Concepts, Software, Applications. In Revision.

OPEN SOURCE TECHNOLOGY GROUP (2006)::SourceForge – Open Source Development Web Site. HP: http://sourceforge.net.

PEBESMA, E.J. & WESSELING, C.G. (1998): Gstat: A Program for Geostatistical Modelling, Prediction and Simulation. Computers & Geosciences, 24/1: 17-31.

PLANCHON, O. & DARBOUX, F. (2001): A fast, Simple and Versatile Algorithm to Fill the Depressions of Digital Elevation Models. Catena, 46: 159-176.

PORTAL MÜNCHEN BETRIEBS-GMBH (2006): LiMux - Die IT-Evolution. HP: http://www.muenchen.de/Rathaus/dir/limux/ueberblick/147191/index.html.

QUINN, P.F., BEVEN, K.J., CHEVALLIER, P. & PLANCHON, O. (1991): The Prediction of Hillslope Flow Paths for Distributed Hydrological Modelling Using Digital Terrain Models. Hydrological Processes, 5: 59-79.

RENKA, R. J. (1988): Multivariate Interpolation of Large Sets of Scattered Data. ACM Transaction on Mathematical Software, 14/2: 139-148.

RETAIL PROFIT MANAGEMENT (2006): Atlas GIS. HP: http://home.earthlink.net/~rpminfonet/atlas.html.

RUBIN, J. (1967): Optimal Classification into Groups: An Approach for Solving the Taxonomy Problem. Journal of Theoretical Biology, 15: 103-144.

SAGA (2006): SAGA – System for Automated Geoscientific Analyses. HP: http://www.saga-gis.uni-goettingen.de.

SELIGE, T., RINGELER, A., BÖHNER, J., CONRAD, O. & KÖTHE, R. (2001): Validation of Grid-Based Surface Reconstruction Techniques Applied to Digital Elevation Models Including the Shuttle Radar Topographic Mission. Proceedings of the International Geoscience and Remote Sensing Symposium, IEEE Publication, New York, S.3135-3137.

SELIGE, T., BÖHNER, J. & RINGELER, A. (2006): Processing of SRTM X-SAR Data to Correct Interferometric Elevation Models for Land Surface Process Applications. in: Böhner, J., McCloy, K.R. & Strobl, J. [Hrsg.]: SAGA – Analysis and Modelling Applications. Göttinger Geographische Abhandlungen, 115: 13-27.

SEXTANTE (2006): SEXTANTE – Sistema Extremeno de Analisis Territorial. HP: http://www.unex.es/plasencia/forestales/investigaciones/grupinvfores/.

SMART, J., HOCK, K. & CSOMOR, S. (2005): Cross-Platform GUI Programming with wxWidgets. Prentice Hall, 656 S.

SORK, V.L., CAMPBELL, D., DYER, R., FERNANDEZ, J., NASON, J., PETIT, R., SMOUSE, P. & STEINBERG, E. (1998): . Proceedings from a Workshop on Gene Flow in Fragmented, Managed, and Continuous Populations. Research Paper No.3. National Center for Ecological Analysis and Synthesis, Santa Barbara, California. DL: http://www.nceas.ucsb.edu/nceas-web/projects/2057/nceas-paper3/.

STOCK, I. (2005): Vergleich von Segmentierungsprogrammen für hochauflösende Satellitendaten am Beispiel einer Quickbird-Aufnahme von Zentral-Sulawesi, Indonesien. Bachelor Arbeit im Studiengang für Informatik, Universität Göttingen, 64S.

STRAHLER, A.N. (1952): Hypsometric (Area-Altitude) Analysis of Erosional Topography. Bulletin of the Geological Society of America, 63: 1117-1142.

STROUSTRUP, B. (1995): The C++ Programming Language. 699S.

SWIG (2006): SWIG. HP: http://www.swig.org.

TARBOTON, D.G. (1997): A New Method for the Determination of Flow Directions and Upslope Areas in Grid Digital Elevation Models. Water Ressources Research, 33/2: 309-319.

TRAUN, C. (2005): SAGA. GeoBIT – Geoinformationstechnologie für die Praxis, 12/2005: 42-45.

TOPMODEL (2006): TOPography based hydrological MODEL. HP: http://www.es.lancs.ac.uk/hfdg/topmodel.html.

WARMERDAM, F., KISELEV, A., MORISSETTE, D. & BUTLER, H. (2006): GDAL – Geospatial Data Abstraction Library. HP: http://www.gdal.org/.

WICHMANN, V. (2006): Modellierung geomorphologischer Prozesse in einem alpinen Einzugsgebiet – Abgrenzung und Klassifizierung der Wirkungsräume von Sturzprozessen und Muren mit einem GIS. Eichstätter Geographische Arbeiten, 15: 231S.

WILSON, J.P. & GALLANT, J.C. [Hrsg.] (2000): Terrain Analysis - Principles and Applications. New York, 512S.

Z_GIS (2006): Z_GIS – Zentrum für Geoinformatik Salzburg. HP: http://www.zgis.at.

ZEVENBERGEN, L.W. & THORNE, C.R. (1987): Quantitative Analysis of Land Surface Topography. Earth Surface Processes and Landforms, 12: 47-56.

WXWIDGETS (2006): wxWidgets – Cross-Platform GUI Library. HP: http://www.wxwidgets.org.

ANHANG A

Die im Folgenden angehängten Beiträge dokumentieren meine persönlichen Arbeiten in Forschung und Lehre und machen dabei auch die jeweilige Bedeutung von SAGA deutlich. Meine Forschungsbeiträge zu den beiden Artikeln, in denen ich nicht zu den Erstautoren gehöre (BÖHNER et al. 2002 und BÖHNER et al. 2003), sollen kurz vorgestellt werden.

BÖHNER et al. (2002, Anhang A-I): Diese Arbeit basiert u.a. auf einer Reihe von mir eigens erstellter SAGA-Module. Es handelt sich im Einzelnen um Umsetzungen des geostatistischen Kriging-Verfahrens sowie der von der Arbeitsgruppe konzipierten komplexen Reliefparameter.

BÖHNER et al. (2003, Anhang A-II): Mein Anteil an der Arbeit ist schwerpunktmäßig in der Implementierung des Bodenwasserhaushaltsmodells sowie des Programms zum gezielten Generieren von Landnutzungsszenarios zu sehen. Daneben war ich an der Integration des Gesamtmodells und der Erstellung kartographischer Endprodukte, einschließlich Animationen zur Veranschaulichung der Erosionsdynamik für begleitende Präsentationen, beteiligt.

I. Soil Regionalisation by Means of Terrain Analysis and Process Parameterisation

Jürgen Böhner[1], Rüdiger Köthe[2],
Olaf Conrad[1], Jens Gross[1], Andre Ringeler[1], Thomas Selige[3]

[1]Geographisches Institut der Universität Göttingen, Goldschmidtstr. 5, D-37077 Göttingen
[2]scilands GmbH Göttingen, Goetheallee 11, D-37073 Göttingen
[3]Department für Pflanzenwissenschaften, TU München, Am Hochanger 2, D-85350 München

Soil Classification 2001
European Soil Bureau – Research Report No.7
S.213-222

BÖHNER, J., KÖTHE, R., CONRAD, O., GROSS, J., RINGELER, A. & SELIGE, T. (2002): Soil Regionalisation by Means of Terrain Analysis and Process Parameterisation. – In: MICHELI, E., NACHTERGAELE, F. & MONTANARELLA, L. [HRGS.]: Soil Classification 2001. – European Soil Bureau – Research Report No. 7, EUR 20398 EN, Luxembourg. S.213-222.

II. The WEELS Model: Methods, Results and Limitations

Jürgen Böhner[1], Walther Schäfer[2],
Olaf Conrad[1], Jens Gross[1], Andre Ringeler[1]

[1]Geographisches Institut der Universität Göttingen, Goldschmidtstr. 5, D-37077 Göttingen
[2]Niedersächsisches Landesamt für Bodenforschung (NLfB) – Bodentechnologisches Institut (BTI) Bremen, Friedrich-Missler Str. 46-50, D-28211 Bremen

Catena
52/2003: 289-308

BÖHNER, J., SCHÄFER, W., CONRAD, O., GROSS, J. & RINGELER, A. (2003): The WEELS Model: Methods, Results and Limitations. – Catena, 52/2003: 289-308.

III. Digitale Reliefanalyse in der multimedialen Lehre

Olaf Conrad

Geographisches Institut der Universität Göttingen, Goldschmidtstr. 5, D-37077 Göttingen

Arbeitsberichte
Geographisches Institut der
Humboldt-Universität zu Berlin
109: 37-47

CONRAD, O. (2005): Digitale Reliefanalyse in der multimedialen Lehre. Arbeitsberichte des Geographischen Instituts der Humboldt-Universität zu Berlin, 109: 37-47.

IV. SAGA – Program Structure and Current State of Implementation

Olaf Conrad

Geographisches Institut der Universität Göttingen, Goldschmidtstr. 5, D-37077 Göttingen

Göttinger Geographische Abhandlungen
115: 39-52

CONRAD, O. (2006): SAGA – Program Structure and Current State of Implementation. in: BÖHNER, J., MCCLOY, K.R. & STROBL, J. [Hrsg.]: SAGA – Analysis and Modelling Applications. Göttinger Geographische Abhandlungen, 115: 39-52.

V. Soil Degradation Risk Assessment Integrating Terrain Analysis and Soil Spatial Prediction Methods

Olaf Conrad[1], Jens Peter Krüger[1], Michael Bock[2], Gerhard Gerold[1]

[1]Geographisches Institut der Universität Göttingen, Goldschmidtstr. 5, D-37077 Göttingen
[2]scilands GmbH Göttingen, Goetheallee 11, D-37073 Göttingen

Proceedings of the International Conference

Soil and Desertification
Integrated Research for the Sustainable Management of Soils in Drylands

5-6 May 2006, Hamburg, Germany

Internet-Publication edited by the Coordination of Desert*Net Germany

www.desertnet.de/proceedings/start.htm

CONRAD, O., KRÜGER, J.P., BOCK, M. & GEROLD, G. (2006): Soil Degradation Risk Assessment Integrating Terrain Analysis and Soil Spatial Prediction Methods. Proceedings of the International Conference "Soil and Desertification - Integrated Research for the Sustainable Management of Soils in Drylands", 5-6 May 2006, Hamburg, Germany. Internet-Publication edited by the Coordination of Desert*Net Germany. [www.desertnet.de/proceedings/start.htm]

ANHANG B

I. Quellen für Daten, Quelltexte und Programme

Internetadresse	**Inhalt**
SAGA	
http://sourceforge.net/projects/saga-gis/	SAGA 2.0 Quellen u. ausführbare Programme, Hauptseite
http://downloads.sourceforge.net/saga-gis/saga_2.0.0_src.zip	SAGA 2.0 Quellen
http://downloads.sourceforge.net/saga-gis/saga_2.0.0_src_linux.tar.gz	SAGA 2.0 Quellen mit Installationsdateien für Linux
http://downloads.sourceforge.net/saga-gis/saga_2.0.0_bin_mswvc.zip	SAGA 2.0 Programmdateien für Windows
http://downloads.sourceforge.net/saga-gis/saga_2.0.0_bin_linux.tar.gz	SAGA 2.0 Programmdateien für Linux
Programmbibliotheken	
http://wxwidgets.org/	wxWidgets Cross-Platform GUI Library
http://libharu.sourceforge.net/	Haru Free PDF Library
http://www.gdal.org/	GDAL – Geospatial Data Abstraction Library
http://www.remotesensing.org/proj/	PROJ.4 – Cartographic Projections Library
Datensätze	
http://downloads.sourceforge.net/saga-gis/forest_of_goettingen.zip	Freie Datensätze für das Einzugsgebiet der oberen Leine in der UTM-Projektion, Zone 32 (SRTM, LandSat TM, Ortschaften)
http://downloads.sourceforge.net/saga-gis/upper_leine.zip	Freie Datensätze für den Göttinger Wald in der UTM-Projektion, Zone 32 (SRTM, LandSat TM, Ortschaften)
Einführung in die Programmierung von SAGA Modulen	
http://downloads.sourceforge.net/saga-gis/modul_programmierung_intro.zip	Die kompletten Quelltexte der Beispiele aus Kap. 3.7

II. Quelltexte für die Modulprogrammierung

Übung 1

Datei: excercise_01.h

```
#ifndef HEADER_INCLUDED__Exercise_01_H
#define HEADER_INCLUDED__Exercise_01_H

#include <saga_api/saga_api.h>

class CExercise_01 : public CSG_Module_Grid
{
public:
   CExercise_01(void);             // constructor

protected:
   virtual bool      On_Execute  (void);  // always overwrite this function
};

#endif // #ifndef HEADER_INCLUDED__Exercise_01_H
```

Datei: excercise_01.cpp

```
#include "exercise_01.h"

CExercise_01::CExercise_01(void)
{
   //-----------------------------------------------------
   // Give some information about your module...

   Set_Name          ("01 - Mein erstes Modul");
   Set_Author        ("Olaf Conrad");
   Set_Description   ("Übung 1: Mein erstes Modul kopiert einen Rasterdatensatz.");

   //-----------------------------------------------------
   // Define your parameters list...

   Parameters.Add_Grid(
      NULL, "INPUT"  , "Das Original",
      "Dieser Datensatz soll kopiert werden.",
      PARAMETER_INPUT
   );

   Parameters.Add_Grid(
      NULL, "OUTPUT" , "Die Kopie",
      "In diesen Datensatz soll kopiert werden.",
      PARAMETER_OUTPUT
   );
}

bool CExercise_01::On_Execute(void)
{
   CSG_Grid *pInput, *pOutput;

   //-----------------------------------------------------
   // Get the parameter settings...

   pInput   = Parameters("INPUT")   ->asGrid();
   pOutput  = Parameters("OUTPUT")  ->asGrid();

   //-----------------------------------------------------
   // Do something...

   pOutput->Assign(pInput);

   //-----------------------------------------------------
   // Return 'true' if everything is okay...

   return( true );
}
```

Übung 2

Datei: excercise_02.h

```
#ifndef HEADER_INCLUDED__Exercise_02_H
#define HEADER_INCLUDED__Exercise_02_H

#include <saga_api/saga_api.h>

class CExercise_02 : public CSG_Module_Grid
{
public:
   CExercise_02(void);              // constructor

protected:
   virtual bool      On_Execute  (void);  // always overwrite this function
};

#endif // #ifndef HEADER_INCLUDED__Exercise_02_H
```

Datei: excercise_02.cpp

```
#include "exercise_02.h"

CExercise_02::CExercise_02(void)
{
   Set_Name          ("02 - Einfache Rasteroperationen");
   Set_Author        ("Olaf Conrad");
   Set_Description   ("Übung 2: Einfache Operationen mit Rasterdaten.");

   Parameters.Add_Grid(
      NULL, "GRID_A" , "A",
      "Der erste Rasterdatensatz.",
      PARAMETER_INPUT
   );

   Parameters.Add_Grid(
      NULL, "GRID_B" , "B",
      "Der zweite Rasterdatensatz.",
      PARAMETER_INPUT
   );

   Parameters.Add_Grid(
      NULL, "GRID_C" , "A kombiniert mit B",
      "Dieser Datensatz soll das Ergebnis enthalten.",
      PARAMETER_OUTPUT
   );

   Parameters.Add_Value(
      NULL, "FACTOR_A", "Faktor A",
      "Skalierungsfaktor für Rasterdatensatz A.",
      PARAMETER_TYPE_Double,
      1.0
   );

   Parameters.Add_Value(
      NULL, "FACTOR_B", "Faktor B",
      "Skalierungsfaktor für Rasterdatensatz B.",
      PARAMETER_TYPE_Double,
      1.0
   );

   Parameters.Add_Choice(
      NULL, "METHOD" , "Operation",
      "Wähle eine arithmetische Operation.",
      "Addition|Subtraktion|Multiplikation|Division|"
   );
}
```

```
bool CExercise_02::On_Execute(void)
{
   int      x, y, Method;
   double   Factor_A, Factor_B, a, b, c;
   CSG_Grid *pA, *pB, *pC;

   pA       = Parameters("GRID_A")     ->asGrid();
   pB       = Parameters("GRID_B")     ->asGrid();
   pC       = Parameters("GRID_C")     ->asGrid();
   Factor_A = Parameters("FACTOR_A")   ->asDouble();
   Factor_B = Parameters("FACTOR_B")   ->asDouble();
   Method   = Parameters("METHOD")     ->asInt();

   //-----------------------------------------------------
   // Check for invalid parameter settings...

   if( Method == 3 && Factor_B == 0.0 )
   {
      Message_Dlg("Division by zero is not allowed !!!");
      return( false );
   }

   //-----------------------------------------------------
   // Do the calculations for each grid cell...

   for(y=0; y<Get_NY() && Set_Progress(y); y++)
   {
      for(x=0; x<Get_NX(); x++)
      {
         if( pA->is_NoData(x, y) || pB->is_NoData(x, y) )
         {
            // don't work with 'no data'...
            pC->Set_NoData(x, y);
         }
         else
         {
            a  = Factor_A * pA->asDouble(x, y);
            b  = Factor_B * pB->asDouble(x, y);

            switch( Method )
            {
            case 0:  // Addition...
               c  = a + b;
               break;

            case 1:  // Subtraction...
               c  = a - b;
               break;

            case 2:  // Multiplication...
               c  = a * b;
               break;

            case 3:  // Division...
               if( b == 0.0 ) // prevent division by zero
                  c  = pC->Get_NoData_Value();
               else
                  c  = a / b;
               break;
            }

            pC->Set_Value(x, y, c);
         }
      }
   }

   return( true );
}
```

Übung 3

Datei: excercise_03.h

```
#ifndef HEADER_INCLUDED__Exercise_03_H
#define HEADER_INCLUDED__Exercise_03_H

#include <saga_api/saga_api.h>

class CExercise_03 : public CSG_Module_Grid
{
public:
   CExercise_03(void);                          // constructor

protected:
   virtual bool      On_Execute     (void);     // always overwrite this function

private:
   CSG_Grid          *m_pInput, *m_pOutput;

   double            Get_Mean_01    (int x, int y);
   double            Get_Mean_02    (int x, int y, int Radius);
};

#endif // #ifndef HEADER_INCLUDED__Exercise_03_H
```

Datei: excercise_03.cpp

```
#include "exercise_03.h"

CExercise_03::CExercise_03(void)
{
   Set_Name          ("03 - Nachbarschaften im Raster");
   Set_Author        ("Olaf Conrad");
   Set_Description   ("Übung 3: Rasteroperationen mit Nachbarschaftsbeziehungen.");

   Parameters.Add_Grid(
      NULL, "INPUT"     , "Eingabe",
      "Dieser Datensatz dient als Eingabe.",
      PARAMETER_INPUT
   );

   Parameters.Add_Grid(
      NULL, "OUTPUT" , "Ausgabe",
      "Dieser Datensatz soll das Ergebnis enthalten.",
      PARAMETER_OUTPUT
   );

   Parameters.Add_Choice(
      NULL, "METHOD" , "Methode",
      "Wähle das Berechnungsverfahren.",
      "Direkte Nachbarschaft|Benutzerdefiniertes Suchfenster|"
   );

   Parameters.Add_Value(
      NULL, "RADIUS" , "Suchfenster",
      "Größe des benutzerdefinierten Suchfensters.",
      PARAMETER_TYPE_Int, 5, 1, true
   );

   Parameters.Add_Value(
      NULL, "DIFFER" , "Bilde die Differenz",
      "Wenn gesetzt wird die Differenz zum Mittelwert berechnet.",
      PARAMETER_TYPE_Bool, false
   );
}
```

```
bool CExercise_03::On_Execute(void)
{
   bool     bDifference;
   int      x, y, Method, Radius;
   double   Result;

   m_pInput    = Parameters("INPUT")   ->asGrid();
   m_pOutput   = Parameters("OUTPUT")  ->asGrid();
   bDifference = Parameters("DIFFER")  ->asBool();
   Method      = Parameters("METHOD")  ->asInt();
   Radius      = Parameters("RADIUS")  ->asInt();

   //-----------------------------------------------------
   // Do the calculations for each grid cell...

   for(y=0; y<Get_NY() && Set_Progress(y); y++)
   {
      for(x=0; x<Get_NX(); x++)
      {
         if( m_pInput->is_NoData(x, y) )
         {
            m_pOutput->Set_NoData(x, y);
         }
         else
         {
            switch( Method )
            {
            case 0:
               Result   = Get_Mean_01(x, y);
               break;

            case 1:
               Result   = Get_Mean_02(x, y, Radius);
               break;
            }

            if( bDifference )
               Result   -= m_pInput->asDouble(x, y);

            m_pOutput->Set_Value(x, y, Result);
         }
      }
   }

   return( true );
}

double CExercise_03::Get_Mean_01(int x, int y)
{
   int      n, i, ix, iy;
   double   s;

   s  = m_pInput->asDouble(x, y);
   n  = 1;

   for(i=0; i<8; i++)
   {
      ix = Get_xTo(i, x);
      iy = Get_yTo(i, y);

      if( m_pInput->is_InGrid(ix, iy, true) )
      {
         s  += m_pInput->asDouble(ix, iy);
         n++;
      }
   }

   return( s / n );
}
```

```
double CExercise_03::Get_Mean_02(int x, int y, int Radius)
{
   int      n, ix, iy;
   double   s;

   s  = 0.0;
   n  = 0;

   for(iy=y-Radius; iy<=y+Radius; iy++)
   {
      for(ix=x-Radius; ix<=x+Radius; ix++)
      {
         if( m_pInput->is_InGrid(ix, iy, true) )
         {
            s  += m_pInput->asDouble(ix, iy);
            n++;
         }
      }
   }

   return( s / n );
}
```

Übung 4

Datei: excercise_04.h

```
#ifndef HEADER_INCLUDED__Exercise_04_H
#define HEADER_INCLUDED__Exercise_04_H

#include <saga_api/saga_api.h>

class CExercise_04 : public CSG_Module_Grid
{
public:
   CExercise_04(void);                     // constructor

protected:
   virtual bool      On_Execute  (void);     // always overwrite this function

private:
   CSG_Grid          *m_pDEM, *m_pArea;

   void              Set_Area (int x, int y);
};

#endif // #ifndef HEADER_INCLUDED__Exercise_04_H
```

Datei: excercise_04.cpp

```
#include "exercise_04.h"

CExercise_04::CExercise_04(void)
{
   Set_Name          ("04 - Wege durchs Raster (A)");
   Set_Author        ("Olaf Conrad");
   Set_Description   (
      "Übung 4: Einen Rasterdatensatz vom höchsten zum niedrigsten Wert hin abarbeiten."
   );

   Parameters.Add_Grid(
      NULL, "INPUT"     , "Höhe",
      "Dieser Datensatz dient als Eingabe.",
      PARAMETER_INPUT
   );

   Parameters.Add_Grid(
      NULL, "OUTPUT" , "Größe des Einzugsgebiets",
      "Dieser Datensatz soll das Ergebnis enthalten.",
      PARAMETER_OUTPUT
   );
```

```
   Parameters.Add_Value(
      NULL, "CELLAREA"  , "Ausgabe in Quadratmeter",
      "Ausgabe in Flächeneinheiten, sonst als Anzahl der Rasterzellen",
      PARAMETER_TYPE_Bool, true
   );
}

bool CExercise_04::On_Execute(void)
{
   int      n, x, y;

   m_pDEM   = Parameters("INPUT")   ->asGrid();
   m_pArea  = Parameters("OUTPUT")  ->asGrid();

   m_pArea->Assign(0.0);

   for(n=0; n<Get_NCells() && Set_Progress_NCells(n); n++)
   {
      m_pDEM->Get_Sorted(n, x, y, true);

      if( m_pDEM->is_NoData(x, y) )
      {
         m_pArea->Set_NoData(x, y);
      }
      else
      {
         Set_Area(x, y);
      }
   }

   //-----------------------------------------------------
   // Post processing...

   if( Parameters("CELLAREA")->asBool() )
   {
      *m_pArea *= m_pArea->Get_Cellarea();
   }

   return( true );
}

void CExercise_04::Set_Area(int x, int y)
{
   int      i, ix, iy, iMax;
   double   z, dz, dzMax;

   m_pArea->Add_Value(x, y, 1);

   z     = m_pDEM->asDouble(x, y);
   dzMax = 0.0;

   for(i=0; i<8; i++)
   {
      ix    = Get_xTo(i, x);
      iy    = Get_yTo(i, y);

      if( m_pDEM->is_InGrid(ix, iy)
      &&  dzMax < (dz = (z - m_pDEM->asDouble(ix, iy)) / Get_Length(i)) )
      {
         iMax  = i;
         dzMax = dz;
      }
   }

   if( dzMax > 0.0 )
   {
      m_pArea->Add_Value(Get_xTo(iMax, x), Get_yTo(iMax, y), m_pArea->asInt(x, y));
   }
}
```

Übung 5

Datei: excercise_05.h

```
#ifndef HEADER_INCLUDED__Exercise_05_H
#define HEADER_INCLUDED__Exercise_05_H

#include <saga_api/saga_api.h>

class CExercise_05 : public CSG_Module_Grid
{
public:
   CExercise_05(void);                // constructor

protected:
   virtual bool   On_Execute  (void);  // always overwrite this function

private:
   CSG_Grid       *m_pDEM, *m_pArea;

   int            Get_Area    (int x, int y);
};

#endif // #ifndef HEADER_INCLUDED__Exercise_05_H
```

Datei: excercise_05.cpp

```
#include "exercise_05.h"

CExercise_05::CExercise_05(void)
{
   Set_Name       ("05 - Wege durchs Raster (B)");
   Set_Author     ("Olaf Conrad");
   Set_Description   ("Übung 5: Einen Rasterdatensatz rekursiv abarbeiten.");

   Parameters.Add_Grid(
      NULL, "INPUT"     , "Höhe",
      "Dieser Datensatz dient als Eingabe.",
      PARAMETER_INPUT
   );

   Parameters.Add_Grid(
      NULL, "OUTPUT" , "Größe des Einzugsgebiets",
      "Dieser Datensatz soll das Ergebnis enthalten.",
      PARAMETER_OUTPUT
   );

   Parameters.Add_Value(
      NULL, "CELLAREA"  , "Ausgabe in Quadratmeter",
      "Ausgabe in Flächeneinheiten, sonst als Anzahl der Rasterzellen",
      PARAMETER_TYPE_Bool, true
   );
}
```

```
bool CExercise_05::On_Execute(void)
{
   int      x, y;

   m_pDEM   = Parameters("INPUT")   ->asGrid();
   m_pArea  = Parameters("OUTPUT")  ->asGrid();

   //-----------------------------------------------------
   // Initialise resulting grid...

   m_pArea->Assign_NoData();

   //-----------------------------------------------------
   // Do the calculations for each grid cell...

   for(y=0; y<Get_NY() && Set_Progress(y); y++)
   {
      for(x=0; x<Get_NX(); x++)
      {
         if( m_pDEM->is_NoData(x, y) )
         {
            m_pArea->Set_NoData(x, y);
         }
         else
         {
            Get_Area(x, y);
         }
      }
   }

   //-----------------------------------------------------
   // Post processing...

   if( Parameters("CELLAREA")->asBool() )
   {
      *m_pArea *= m_pArea->Get_Cellarea();
   }

   return( true );
}

int CExercise_05::Get_Area(int x, int y)
{
   int      i, ix, iy;

   if( m_pArea->is_NoData(x, y) )     // cell has not been processed yet...
   {
      m_pArea->Set_Value(x, y, 1);  // add this cell's contribution...

      for(i=0; i<8; i++)
      {
         ix = Get_xFrom(i, x);
         iy = Get_yFrom(i, y);

         if( m_pDEM->is_InGrid(ix, iy)      // drains ith neigbour into this cell???
         && m_pDEM->Get_Gradient_NeighborDir(ix, iy) == i )
         {                          // then recursive call of this function...
            m_pArea->Add_Value(x, y, Get_Area(ix, iy));
         }
      }
   }

   return( m_pArea->asInt(x, y) );  // return this cell's area...
}
```

Übung 6

Datei: excercise_06.h

```
#ifndef HEADER_INCLUDED__Exercise_06_H
#define HEADER_INCLUDED__Exercise_06_H

#include <saga_api/saga_api.h>

class CExercise_06 : public CSG_Module
{
public:
   CExercise_06(void);                    // constructor

protected:
   virtual bool      On_Execute  (void);  // always overwrite this function

private:
   CSG_Grid             *m_pLife, *m_pTemp;

   void              Next_Step   (void);
};

#endif // #ifndef HEADER_INCLUDED__Exercise_06_H
```

Datei: excercise_06.cpp

```
#include "exercise_06.h"

CExercise_06::CExercise_06(void)
{
   Set_Name          ("06 - Dynamische Simulation");
   Set_Author        ("Olaf Conrad");
   Set_Description   ("Übung 6: Conway's Life - eine lebendige Simulation.");

   Parameters.Add_Value(
      NULL, "NX"        , "Breite der Welt",
      "",
      PARAMETER_TYPE_Int, 100, 10, true
   );

   Parameters.Add_Value(
      NULL, "NY"        , "Höhe der Welt",
      "",
      PARAMETER_TYPE_Int, 100, 10, true
   );
}

bool CExercise_06::On_Execute(void)
{
   int      x, y, n;

   //-----------------------------------------------------
   // General initialisations...

   x        = Parameters("NX")->asInt();
   y        = Parameters("NY")->asInt();

   m_pLife  = SG_Create_Grid(GRID_TYPE_Byte, x, y);
   m_pLife->Set_Name("Life");

   DataObject_Add(m_pLife);
   DataObject_Set_Colors(m_pLife, CSG_Colors(2, SG_COLORS_BLACK_WHITE, true));

   //-----------------------------------------------------
   // Initialise life's world...

   for(y=0; y<m_pLife->Get_NY(); y++)
   {
      for(x=0; x<m_pLife->Get_NX(); x++)
      {
         m_pLife->Set_Value(x, y, rand() > RAND_MAX / 2 ? 0 : 1);
      }
   }
```

```
   //-----------------------------------------------------
   // Execution...

   m_pTemp  = SG_Create_Grid(m_pLife, GRID_TYPE_Byte);

   for(n=1; Process_Get_Okay(true); n++)
   {
      Process_Set_Text(CSG_String::Format("%d. Generation", n));

      Next_Step();
   }

   delete(m_pTemp);

   return( true );
}

void CExercise_06::Next_Step(void)
{
   int      x, y, i, ix, iy, n;

   //-----------------------------------------------------
   for(y=0; y<m_pLife->Get_NY(); y++)
   {
      for(x=0; x<m_pLife->Get_NX(); x++)
      {
         //--------------------------------------------
         // Count neighours...

         for(i=0, n=0; i<8; i++)
         {
            ix = m_pLife->Get_System().Get_xTo(i, x);
            iy = m_pLife->Get_System().Get_yTo(i, y);

            if( m_pLife->is_InGrid(ix, iy) && m_pLife->asByte(ix, iy) == 0 )
            {
               n++;
            }
         }


         //--------------------------------------------
         // Dead or alive...

         switch( n )
         {
         case 2:  // Keep status quo...
            m_pTemp->Set_Value(x, y, m_pLife->asByte(x, y));
            break;

         case 3:  // Birth...
            m_pTemp->Set_Value(x, y, 0);
            break;

         default: // Dead...
            m_pTemp->Set_Value(x, y, 1);
            break;
         }
      }
   }

   //-----------------------------------------------------
   m_pLife->Assign(m_pTemp);

   DataObject_Update(m_pLife, true);
}
```

Übung 7

Datei: excercise_07.h

```
#ifndef HEADER_INCLUDED__Exercise_07_H
#define HEADER_INCLUDED__Exercise_07_H

#include <saga_api/saga_api.h>

class CExercise_07 : public CSG_Module
{
public:
   CExercise_07(void);                    // constructor

protected:
   virtual bool      On_Execute(void);    // always overwrite this function
};

#endif // #ifndef HEADER_INCLUDED__Exercise_07_H
```

Datei: excercise_07.cpp

```
#include "exercise_07.h"

CExercise_07::CExercise_07(void)
{
   Set_Name          ("07 - Arbeiten mit Vektordaten");
   Set_Author        ("Olaf Conrad");
   Set_Description   ("Übung 7: Erste Schritte mit Vektordaten.");

   Parameters.Add_Shapes(
      NULL  , "INPUT"   , "Eingabe",
      "Dieser Vektordatensatz soll kopiert und verschoben werden.",
      PARAMETER_INPUT
   );

   Parameters.Add_Shapes(
      NULL  , "OUTPUT"  , "Ausgabe",
      "In diesen Datensatz soll das Ergebnis geschrieben werden.",
      PARAMETER_OUTPUT
   );

   Parameters.Add_Value(
      NULL  , "DX"      , "X Versatz",
      "",
      PARAMETER_TYPE_Double,
      10.0
   );

   Parameters.Add_Value(
      NULL  , "DY"      , "Y Versatz",
      "",
      PARAMETER_TYPE_Double,
      10.0
   );
}
```

```
bool CExercise_07::On_Execute(void)
{
   int         iShape, iPart, iPoint;
   double      dx, dy;
   TSG_Point   Point;
   CSG_Shapes  *pShapes_A, *pShapes_B;
   CSG_Shape   *pShape_A, *pShape_B;

   pShapes_A   = Parameters("INPUT")   ->asShapes();
   pShapes_B   = Parameters("OUTPUT")  ->asShapes();
   dx          = Parameters("DX")      ->asDouble();
   dy          = Parameters("DY")      ->asDouble();

   pShapes_B->Create(pShapes_A->Get_Type(), "Copy and Shift", &pShapes_A->Get_Table());

   //-----------------------------------------------------
   // Copy shapes from layer A and shift each point's position...

   for(iShape=0; iShape<pShapes_A->Get_Count() && Set_Progress(iShape, pShapes_A->
                                                              Get_Count()); iShape++)
   {
      pShape_A = pShapes_A->Get_Shape(iShape);
      pShape_B = pShapes_B->Add_Shape(pShape_A->Get_Record());

      for(iPart=0; iPart<pShape_A->Get_Part_Count(); iPart++)
      {
         for(iPoint=0; iPoint<pShape_A->Get_Point_Count(iPart); iPoint++)
         {
            Point = pShape_A->Get_Point(iPoint, iPart);

            Point.x  += dx;   // perform the translation before
            Point.y  += dy;   // you add the point to the new shape...

            pShape_B->Add_Point(Point, iPart);
         }
      }
   }

   return( true );
}
```

Übung 8

Datei: excercise_08.h

```
#ifndef HEADER_INCLUDED__Exercise_08_H
#define HEADER_INCLUDED__Exercise_08_H

#include <saga_api/saga_api.h>

class CExercise_08 : public CSG_Module_Grid
{
public:
   CExercise_08(void);                    // constructor

protected:
   virtual bool   On_Execute     (void);  // always overwrite this function

private:
   CSG_Grid       *m_pFlow_Dir;

   void           Get_Flow_Line  (int x, int y, CSG_Shape *pLine, int ID);
};

#endif // #ifndef HEADER_INCLUDED__Exercise_08_H
```

Datei: excercise_08.cpp

```
#include "exercise_08.h"

CExercise_08::CExercise_08(void)
{
   Set_Name         ("08 - Abflusslinien");
   Set_Author       ("Olaf Conrad");
   Set_Description  ("Übung 8: Ableitung von Abflusslinien aus einem Höhenwertraster.");

   Parameters.Add_Grid(
      NULL, "INPUT"     , "Höhe",
      "Dieser Datensatz enthält die Höhenwerte.",
      PARAMETER_INPUT
   );

   Parameters.Add_Shapes(
      NULL, "OUTPUT" , "Abflusslinien",
      "In diesen Datensatz werden die Abflusslinien gespeichert.",
      PARAMETER_OUTPUT, SHAPE_TYPE_Line
   );
}

bool CExercise_08::On_Execute(void)
{
   int         n, x, y;
   CSG_Grid    *pDEM;
   CSG_Shapes  *pLines;

   pDEM     = Parameters("INPUT")   ->asGrid();
   pLines   = Parameters("OUTPUT")  ->asShapes();

   //-----------------------------------------------------
   // Initialise shapes layer...

   pLines->Create(SHAPE_TYPE_Line, "Abflusslinien");
   pLines->Get_Table().Add_Field("ID"    , TABLE_FIELDTYPE_Int);
   pLines->Get_Table().Add_Field("LENGTH", TABLE_FIELDTYPE_Double);

   //-----------------------------------------------------
   // Initialise flow direction matrix...

   m_pFlow_Dir = SG_Create_Grid(pDEM, GRID_TYPE_Char);

   for(y=0; y<Get_NY() && Set_Progress(y); y++)
   {
      for(x=0; x<Get_NX(); x++)
      {
         m_pFlow_Dir->Set_Value(x, y, pDEM->Get_Gradient_NeighborDir(x, y));
      }
   }

   //-----------------------------------------------------
   // Get the flow lines...

   for(n=0; n<Get_NCells() && Set_Progress_NCells(n); n++)
   {
      pDEM->Get_Sorted(n, x, y, true);

      if( m_pFlow_Dir->asInt(x, y) >= 0 )
      {
         Get_Flow_Line(x, y, pLines->Add_Shape(), pLines->Get_Count() + 1);
      }
   }

   //-----------------------------------------------------
   // Clean up...

   delete(m_pFlow_Dir);

   return( true );
}
```

```
void CExercise_08::Get_Flow_Line(int x, int y, CSG_Shape *pLine, int ID)
{
   int      Flow_Dir;
   double   Length;

   if( (Flow_Dir = m_pFlow_Dir->asInt(x, y)) >= 0 )
   {
      Length   = 0.0;

      pLine->Add_Point(Get_System()->Get_Grid_to_World(x, y));

      do
      {
         m_pFlow_Dir->Set_Value(x, y, -1);  // mark as processed

         x     += Get_xTo(Flow_Dir);
         y     += Get_yTo(Flow_Dir);
         Length   += Get_Length(Flow_Dir);

         pLine->Add_Point(Get_System()->Get_Grid_to_World(x, y));
      }
      while( is_InGrid(x, y) && (Flow_Dir = m_pFlow_Dir->asInt(x, y)) >= 0 );

      pLine->Get_Record()->Set_Value(0, ID);
      pLine->Get_Record()->Set_Value(1, Length);
   }
}
```

Übung 9

Datei: excercise_09.h

```
#ifndef HEADER_INCLUDED__Exercise_09_H
#define HEADER_INCLUDED__Exercise_09_H

#include <saga_api/saga_api.h>

class CExercise_09 : public CSG_Module
{
public:
   CExercise_09(void);                     // constructor

protected:
   virtual bool      On_Execute  (void);     // always overwrite this function
};

#endif // #ifndef HEADER_INCLUDED__Exercise_09_H
```

Datei: excercise_09.cpp

```
#include "exercise_09.h"

CExercise_09::CExercise_09(void)
{
   Set_Name          ("09 - Rasterwerte für Punkte");
   Set_Author        ("Olaf Conrad");
   Set_Description   ("Übung 9: Fügt Punktdaten die Werte beliebiger Rasterdatensätze
hinzu.");

   Parameters.Add_Shapes(
      NULL, "SHAPES" , "Punkte",
      "",
      PARAMETER_INPUT, SHAPE_TYPE_Point
   );

   Parameters.Add_Grid_List(
      NULL, "GRIDS"  , "Raster",
      "",
      PARAMETER_INPUT, false
   );
```

```
   Parameters.Add_Shapes(
      NULL, "RESULT" , "Punkte mit Rasterwerten",
      "",
      PARAMETER_OUTPUT, SHAPE_TYPE_Point
   );

   Parameters.Add_Choice(
      NULL, "INTERPOL", "Raster Interpolation",
      "",
        "

      "Nearest Neighbor|"
      "Bilinear Interpolation|"
      "Inverse Distance Interpolation|"
      "Bicubic Spline Interpolation|"
      "B-Spline Interpolation|", 4
   );
}

bool CExercise_09::On_Execute(void)
{
   int                     iPoint, iGrid, nFields, Interpolation;
   double                  Value;
   TSG_Point               Point;
   CSG_Shapes              *pPoints;
   CSG_Shape               *pPoint;
   CSG_Parameter_Grid_List *pGrids;

   pGrids         = Parameters("GRIDS")      ->asGridList();
   pPoints        = Parameters("RESULT")     ->asShapes();
   Interpolation  = Parameters("INTERPOL")   ->asInt();

   //-----------------------------------------------------
   // Do intialisations...

   if( pPoints != Parameters("SHAPES")->asShapes() )
   {
      pPoints->Assign(Parameters("SHAPES")->asShapes());
   }

   nFields  = pPoints->Get_Table().Get_Field_Count();

   for(iGrid=0; iGrid<pGrids->Get_Count(); iGrid++)
   {
      pPoints->Get_Table().Add_Field(
         pGrids->asGrid(iGrid)->Get_Name(), TABLE_FIELDTYPE_Double
      );
   }

   //-----------------------------------------------------
   // Get the values for all points...

   for(iPoint=0; iPoint<pPoints->Get_Count() && Set_Progress(iPoint, pPoints->Get_Count());
                                                                              iPoint++)
   {
      pPoint   = pPoints->Get_Shape(iPoint);
      Point    = pPoint ->Get_Point(0);  // Point shapes always have exactly one point...

      for(iGrid=0; iGrid<pGrids->Get_Count(); iGrid++)
      {
         if( pGrids->asGrid(iGrid)->is_InGrid_byPos(Point) )
            Value = pGrids->asGrid(iGrid)->Get_Value(Point, Interpolation, true);
         else
            Value = -99999;

         pPoint->Get_Record()->Set_Value(nFields + iGrid, Value);
      }
   }

   return( true );
}
```

Übung 10

Datei: excercise_10.h

```
#ifndef HEADER_INCLUDED__Exercise_10_H
#define HEADER_INCLUDED__Exercise_10_H

#include <saga_api/saga_api.h>

class CExercise_10 : public CSG_Module_Interactive
{
public:
   CExercise_10(void);

protected:
   virtual bool   On_Execute          (void);
   virtual bool   On_Execute_Position  (CSG_Point ptWorld, TSG_Module_Interactive_Mode Mode);

private:
   int                     m_Interpolation;

   CSG_Shapes              *m_pPoints;

   CSG_Parameter_Grid_List *m_pGrids;
};

#endif // #ifndef HEADER_INCLUDED__Exercise_10_H
```

Datei: excercise_10.cpp

```
#include "exercise_10.h"

CExercise_10::CExercise_10(void)
{
   Set_Name          ("10 - Rasterwerte für Punkte");
   Set_Author        ("Olaf Conrad");
   Set_Description   (
      "Übung 10: Erzeugt interaktiv Punktdaten und fügt “
      “ihnen die Werte beliebiger Rasterdatensätze hinzu."
   );

   Parameters.Add_Grid_List(
      NULL, "GRIDS"     , "Raster",
      "",
      PARAMETER_INPUT, false
   );

   Parameters.Add_Shapes(
      NULL, "RESULT" , "Punkte mit Rasterwerten",
      "",
      PARAMETER_OUTPUT, SHAPE_TYPE_Point
   );

   Parameters.Add_Choice(
      NULL, "INTERPOL"  , "Raster Interpolation",
      "",

      "Nearest Neighbor|"
      "Bilinear Interpolation|"
      "Inverse Distance Interpolation|"
      "Bicubic Spline Interpolation|"
      "B-Spline Interpolation|", 4
   );
}
```

```
bool CExercise_10::On_Execute(void)
{
   m_pGrids          = Parameters("GRIDS")      ->asGridList();
   m_pPoints         = Parameters("RESULT")     ->asShapes();
   m_Interpolation   = Parameters("INTERPOL")   ->asInt();

   m_pPoints->Create(SHAPE_TYPE_Point, "Rasterwerte");

   for(int iGrid=0; iGrid<m_pGrids->Get_Count(); iGrid++)
   {
      m_pPoints->Get_Table().Add_Field(
         m_pGrids->asGrid(iGrid)->Get_Name(), TABLE_FIELDTYPE_Double
      );
   }

   return( true );
}

bool CExercise_10::On_Execute_Position(CSG_Point ptWorld, TSG_Module_Interactive_Mode Mode)
{
   int         iGrid;
   double      Value;
   CSG_Shape   *pPoint;

   if( Mode == MODULE_INTERACTIVE_LDOWN )
   {
      pPoint   = m_pPoints->Add_Shape();
      pPoint->Add_Point(ptWorld);

      for(iGrid=0; iGrid<m_pGrids->Get_Count(); iGrid++)
      {
         if( m_pGrids->asGrid(iGrid)->is_InGrid_byPos(ptWorld) )
            Value = m_pGrids->asGrid(iGrid)->Get_Value(ptWorld, m_Interpolation, true);
         else
            Value = -99999;

         pPoint->Get_Record()->Set_Value(iGrid, Value);
      }

      DataObject_Update(m_pPoints, false);
   }

   return( true );
}
```

Die vollständige Bibliotheksschnittstelle

Datei: MLB_Interface.cpp

```
// 1. Include the SAGA-API header:

#include <saga_api/saga_api.h>


// 2. Place general module library informations here:

const char *   Get_Info(int i)
{
   switch( i )
   {
   case MLB_INFO_Name:        return( "Modul-Programmierung" );
   case MLB_INFO_Author:      return( "Olaf Conrad" );
   case MLB_INFO_Description: return( "Eine Einführung in die Modul-Programmierung." );
   case MLB_INFO_Version:     return( "1.0" );
   case MLB_INFO_Menu_Path:   return( "Modul-Programmierung" );
   }

   return( NULL );
}


// 3. Include the headers of your modules:

#include "exercise_01.h"
#include "exercise_02.h"
#include "exercise_03.h"
#include "exercise_04.h"
#include "exercise_05.h"
#include "exercise_06.h"
#include "exercise_07.h"
#include "exercise_08.h"
#include "exercise_09.h"
#include "exercise_10.h"


// 4. Allow your modules to be created here:

CSG_Module *   Create_Module(int i)
{
   switch( i )
   {
   case 0:  return( new CExercise_01 );
   case 1:  return( new CExercise_02 );
   case 2:  return( new CExercise_03 );
   case 3:  return( new CExercise_04 );
   case 4:  return( new CExercise_05 );
   case 5:  return( new CExercise_06 );
   case 6:  return( new CExercise_07 );
   case 7:  return( new CExercise_08 );
   case 8:  return( new CExercise_09 );
   case 9:  return( new CExercise_10 );
   }

   return( NULL );
}


// 5. Use the MLB_INTERFACE Makro to implement the module library interface:

MLB_INTERFACE
```

Lebenslauf

Persönliche Daten

Name	Olaf Conrad
Geburtsdatum	11. Mai 1967
Geburtsort	Dannenberg (Elbe)
Staatsangehörigkeit	Deutsch

Schulausbildung

1973-1977	Grundschule Lüchow
1977-1979	Realschule Lüchow
1979-1986	Gymnasium Lüchow, Abschluss: Abitur

Wehrpflicht

1986-1987	Grundwehrdienst

Studium

1987-1990	Studium der Physik an der Technischen Universität Hannover
1990-1998	Studium der Geographie an der Georg-August Universität Göttingen mit den Nebenfächern Bodenkunde, Botanik, Geologie, Abschluss: Diplom-Geograph
1994-1995	Auslandsaufenthalt: Studium der Geographie am Trinity College Dublin

Berufspraxis

1998-2000	Geoinformationsdienst GmbH, Rosdorf: Entwicklung eines produktionsorientierten landwirtschaftlichen Rauminformationssystems (POLARIS)
2001-2004	Georg-August Universität Göttingen, Abteilung Physische Geographie: wissenschaftlicher Mitarbeiter im vom BMBF geförderten Projekt GeoVis – Geographie und Visualisierung
2005-2006	SAGA User Group e.V., Göttingen: Angestellter mit der Aufgabe von Pflege und Ausbau von SAGA – System für Automatisierte Geowissenschaftliche Analysen
Seit 2006	Technische Universität München, Abteilung für Pflanzenwissenschaften: wissenschaftlicher Angestellter im von der Bayrischen Forschungsstiftung geförderten Projekt GeoSteP – Geoinformationstechnologien für standorteffiziente Pflanzenproduktion

Promotion

1999-2006	Georg-August Universtität Göttingen Thema: SAGA – Entwurf, Funktionsumfang und Anwendung eines Systems für Automatisierte Geowissenschaftliche Analysen