UNIVERSITÄT OSNABRÜCK

Bachelorarbeit

Quantum GIS 1.2.0 Plugin zur Analyse und Reprojektion von Rasterdaten unter Verwendung der Geospatial Data Abstraction Library (GDAL)

Betreuer: Prof. Dr.-Ing. Manfred Ehlers

Vorgelegt von:

Florian Hillen

Matrikelnummer: xxxxxx

IGF
Institut für Geoinformatik
und Fernerkundung
Universität Osnabrück

Osnabrück, den 09.10.2009

Studiengang Geoinformatik

Institut für Geoinformatik und Fernerkundung

Fachbereich 6: Mathematik / Informatik

Universität Osnabrück

Inhaltsverzeichnis

1 Einleitung

Open Source Software, also Software deren Quellcode für jeden Anwender frei einsehbar ist, erlangt auch im Bereich der Geographischen Informationssysteme (GIS) eine immer größer werdende Rolle.[1] Viele Antworten auf räumliche Fragestellungen, die in früheren Zeiten nur von teurer Software beantwortet werden konnten, sind nun kostenlos verfügbar und die dafür benötigten Programmierungen werden von stetig wachsenden Benutzergemeinschaften übernommen.[2] Open Source GIS Projekte wie GRASS GIS, Open JUMP und auch Quantum GIS werden stetig weiter entwickelt und erreichen in bestimmten Bereichen bereits die Funktionalität von proprietären GIS-Lösungen.

Aus dem Wunsch diese Funktionalität selbst erweitern zu können entstand die Idee zur Programmierung einer Erweiterung für eines der oben genannten Geographischen Informationssysteme. Ich entschied mich auf Grund der einfachen Erweiterbarkeit und der zugrundeliegenden Programmiersprache für Quantum GIS.

Durch die eingeschränkte Funktionalität von Quantum GIS im Bereich der Rasterdatenverarbeitung, wuchs die Idee zur Entwicklung eines Plugins, welches eben diese Funktionalität erweitern soll. Als Vorbild diente hierfür die Konsolenanwendung FWTools, eine Sammlung von Open Source GIS Anwendungen entwickelt von Frank Warmerdam, welche eine Vielzahl an Funktionen auf Grundlage der Bibliothek Geospatial Data Abstraction Library (GDAL) zur Verfügung stellt.

Das Ziel dieser Arbeit soll somit die eigenständige Programmierung eines Plugins für Quantum GIS auf Grundlage der GDAL sein. Die geplante Funktionalität soll sich auf die Analyse und Reprojektion von Rasterdaten fokussieren und somit die Qualität von Quantum GIS im Bereich der Rasterdatenverarbeitung erhöhen. Im ersten Schritt soll eine Rasterdatei analysiert und die in ihr enthaltenen Informationen für den Benutzer dargestellt werden. Im Weiteren soll es Möglich sein mit Hilfe des Plugins die Rasterdatei in ein anderes Koordinatensystem zu transformieren. Dies ist in Quantum GIS bisher nur mit Vektordaten möglich.

[1] Neteler, Mitasova (2008), S. 1-3
[2] Mitchell (2009), S. 2-4

2 Grundlagen

2.1 Quantum GIS

Quantum GIS (auch: QGIS) ist ein Open Source Geoinformationssystem lizensiert unter der GNU General Public License (GPL) und ein offizielles Projekt der Open Source Geospatial Foundation (OSGeo). Das Projekt wurde im Februar 2002 von Gary Sherman initiiert, woraufhin im Mai 2002 die Programmierung begann. Am 19. Juli 2002 erschien das erste Release, das zunächst ausschließlich PostGIS Layer unterstützte und nur partiell funktionstüchtig war.[3] Die derzeitig aktuelle Version von Quantum GIS ist das Release 1.2.0 mit Namen „*Daphnis*“ vom 01. September 2009.[4]

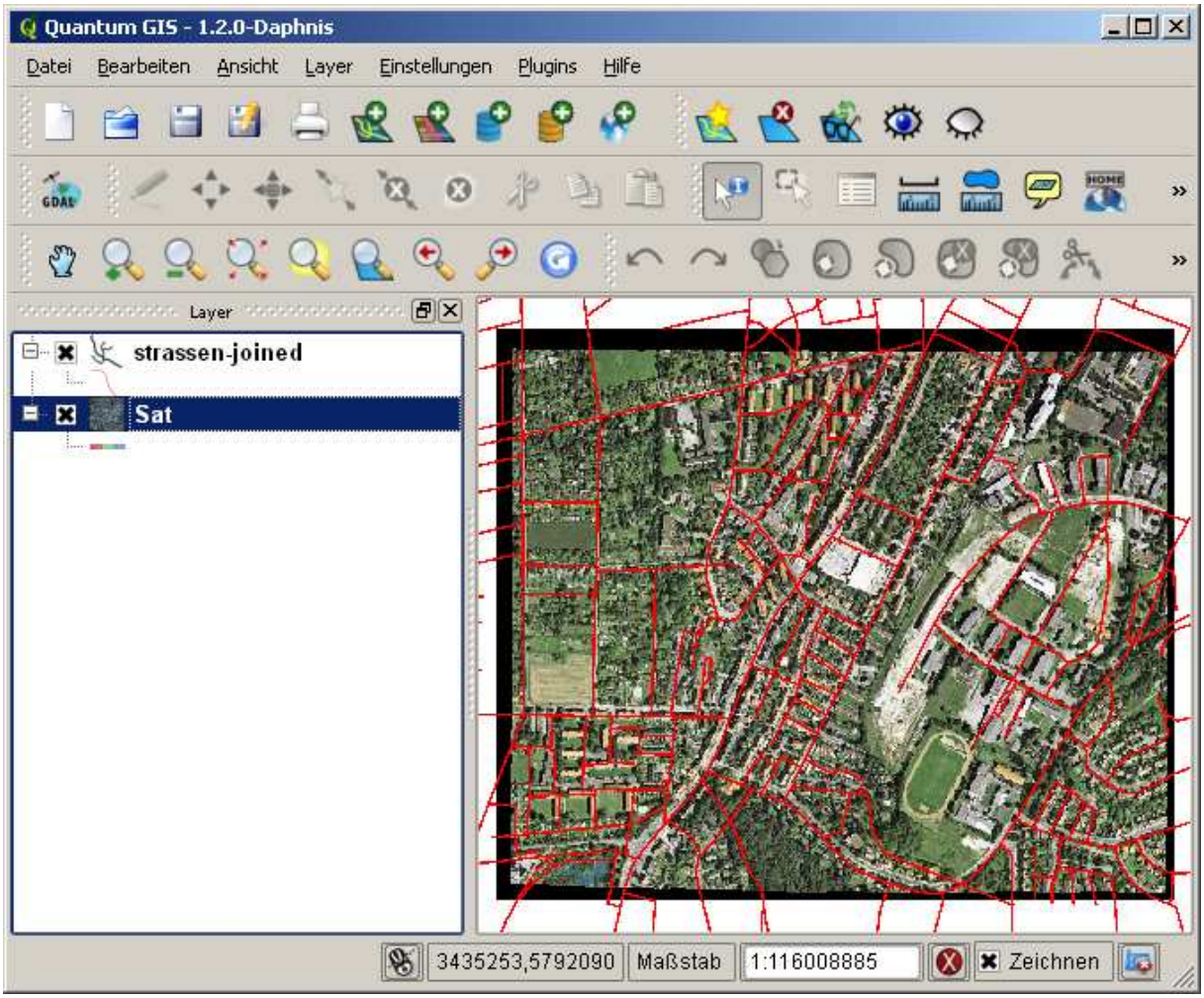

Abb. 1: Graphische Oberfläche von Quantum GIS 1.2.0 „Daphnis“ (Quelle: Eigene Darstellung)

[3] http://www.qgis.org/de/ueber-qgis.html
[4] http://blog.qgis.org/

Quantum GIS erfüllt alle vier Funktionen eines Informationssystems, die durch das Vierkomponentenmodell „*EVAP*“ vorgegeben sind.[5] Es unterstützt alle gängigen Raster- und Vektorformate und bietet die Möglichkeit, Daten von verschiedenen Datenbanksystemen abzufragen. Funktionen zur Kartenerstellung, wie Objektbeschriftung und Signaturen, sind ebenso enthalten wie die Möglichkeit von räumlichen Analysen durch die integrierten GRASS Module. Zur Erweiterung der Funktionalität lassen sich über den „Plugin Manager“ Zusatzprogramme, sogenannte Plugins, variabel zuschalten.[6]

Quantum GIS wird in der Programmiersprache C++, unter Verwendung der Qt Bibliotheken zur Verwaltung und Erstellung von graphischen Oberflächen, entwickelt. Die Architektur von Quantum GIS lässt sich in drei grobe Bestandteile aufteilen: Das Graphical User Interface (GUI), der Kern und die externen Bibliotheken (vgl. Abb. 2).

Das GUI stellt die Benutzerschnittstelle dar und beinhaltet alle graphischen Repräsentationen. Hierzu gehören, wie in Abbildung 1 zu sehen, unter anderem die Werkzeugleiste im oberen Bereich, die Zeichenfläche im rechten Teil darunter, die Layerübersicht im linken Teil und auch die, über die Menüführung erreichbaren, Plugins, die sowohl in C++ als auch in der Programmiersprache Python entwickelt werden können.

Der Quantum GIS Kern beinhaltet die grundlegende Programmierung zur Datenbereitstellung und -verarbeitung. Wichtige Kernbestandteile sind dabei Provider (also „Lieferanten“) für Vektor- und Rasterdaten, sowie allgemeine Karten- und Programmfunktionen (vgl. Abb. 2).

Durch die Einbindung von externen Bibliotheken werden bereits existierende Standards zur Datenverwaltung und Datenanalyse in die Programmierung von Quantum GIS aufgenommen. Beispiele sind hier unter Anderem die GEOS Bibliothek, die als Basis für alle Geometrien dient, PostgreSQL / PostGIS und Sqlite3 als Datenbank-Bibliotheken, sowie die Geospatial Data Abstraction Library (GDAL) für Rasterdaten (vgl. Abb. 2).

[5] de Lange (2005), S. 320
[6] http://www.qgis.org/de/ueber-qgis/features.html

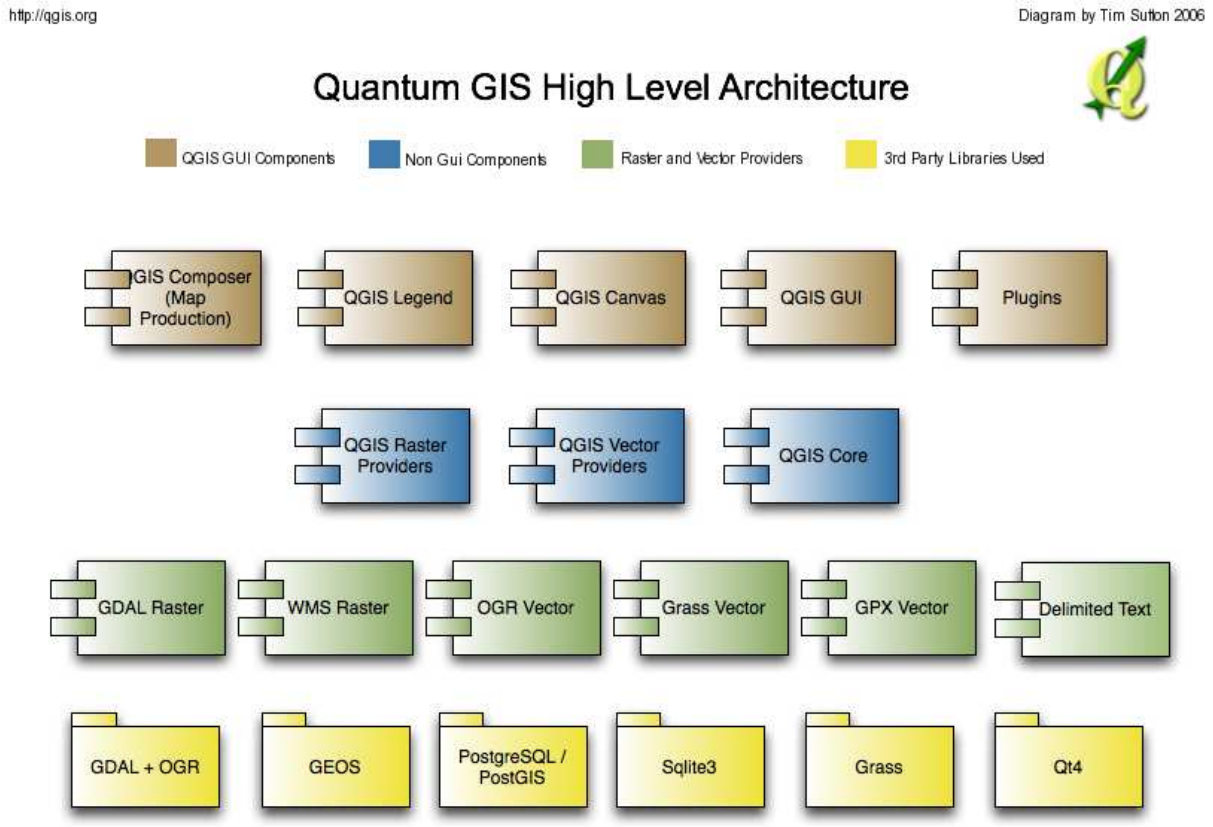

Abb. 2: Quantum GIS Architektur (Quelle: http://www.qgis.org/wiki/File:QGISHighLevelArchitecture.png)

2.2 Geospatial Data Abstraction Library (GDAL)

Die Geospatial Data Abstraction Library (nachfolgend: GDAL) ist eine Open Source Bibliothek lizensiert unter der MIT-Lizenz zur Übersetzung räumlicher Rasterdaten. Sie liefert ein einheitliches Datenmodell sowohl für gängige Formate (JPEG, GeoTIFF, PNG, usw.) als auch für viele weitere.[7]

Die Klasse `GDALDataset` liefert die grundlegende Struktur zur Speicherung von Rasterdaten. Neben den Rasterbändern werden in dieser Klasse auch die Größe des Rasters, das Koordinatenreferenzsystem, Transformationsparameter sowie Metadaten gespeichert (vgl. Abb. 3).[8]

Das Koordinatenreferenzsystem liegt mit Name, geodätischem Datum, Bezugsellipsoid, Projektionstyp, Darstellungseinheit und vielen weiteren Informationen vor. Repräsentiert

[7] http://www.gdal.org/
[8] http://www.gdal.org/gdal_datamodel.html

werden diese Daten in der, vom Open Geospatial Consortium (OGC) beschlossenen, Well-known Text (kurz: WKT) Darstellung.[8]

Die Parameter zur Transformation zwischen Rasterpunkten und georeferenzierten Koordinaten können in einem `GDALDataset` auf zwei Arten beschrieben werden. Bei der affinen Transformation werden 6 Koeffizienten gespeichert, durch welche die Rasterpunkt in georeferenzierte Koordinaten umgerechnet werden können (vgl. Kap. 2.3). Bei der Transformation mittels Ground Control Points (GCPs) werden von GDAL lediglich die Punkte selbst gespeichert, der Transformationsvorgang muss von der Anwendung, in der GDAL ausgeführt wird, selbst implementiert werden.[8]

Die Bänder einer Rasterdatei werden in GDAL durch eine eigene Klasse namens `GDALRasterBand` repräsentiert. Neben grundlegenden Eigenschaften wie Breite, Höhe und Datentyp können in ihr auch optionale Parameter, wie der NoData-Wert oder Kategorie-namen für thematische Bilder, gespeichert werden. Ebenso kann einem Rasterband eine Farbtabelle (hier: Color Table) zugeordnet sein, in der Farbwerte in einem beliebigen Farbsystem (RGB, CMYK, HLS, Graustufen) gespeichert sind und mit Bildpunkten der Rasterdatei verknüpft werden können.[9]

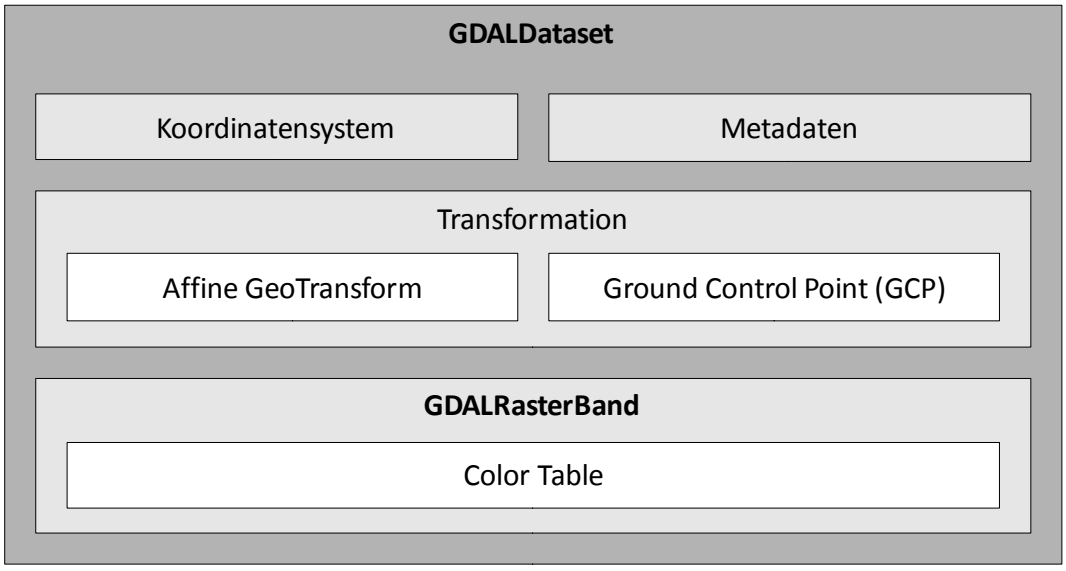

Abb. 3: Das GDAL Datenmodell (Quelle: Eigene Darstellung)

[9] http://www.gdal.org/gdal_datamodel.html

2.3 Koordinatensysteme und Koordinatentransformation

Koordinatensysteme bilden die Grundlage zur Darstellung von Geoobjekten im Vektor- und Rastermodell, da durch sie ein Raumbezug hergestellt werden kann.[10] Ein Problem stellt dabei die Abbildung der dreidimensionalen Erdoberfläche auf ein zweidimensionales Bezugsystem dar.[11]

Mit dem Ziel ein System zu erhalten, dass im Hinblick auf die Genauigkeit den Ansprüchen von Behörden und Planungseinrichtungen genügte, wurde im 19. Jahrhundert regional damit begonnen sog. Festpunktnetze zu entwickeln. Dazu wurde ein Netz von vermarkten Punkten aufgebaut, das durch Winkel- und Streckenmessungen exakt bestimmt und um die Erdkrümmung korrigiert wurde.[12] Die unregelmäßige Struktur der Erde wurde in diesen Systemen auf Grundlage von Ellipsoiden, welche die Form der Erde annähern sollten, entwickelt. Um eine bestmögliche Annäherung an die Realität in den unterschiedlichen Regionen der Erde zu erreichen, ergeben sich hieraus viele unterschiedliche Bezugsellipsoide (vgl. Abb. 4). Durch das sog. geodätische Datum wird die Beziehung eines solchen Ellipsoids zu einem globalen Bezugssystem hergestellt. Diese Parameter liefern die Grundlage für eine Transformation zwischen verschiedenen Koordinatenreferenzsystemen[13].

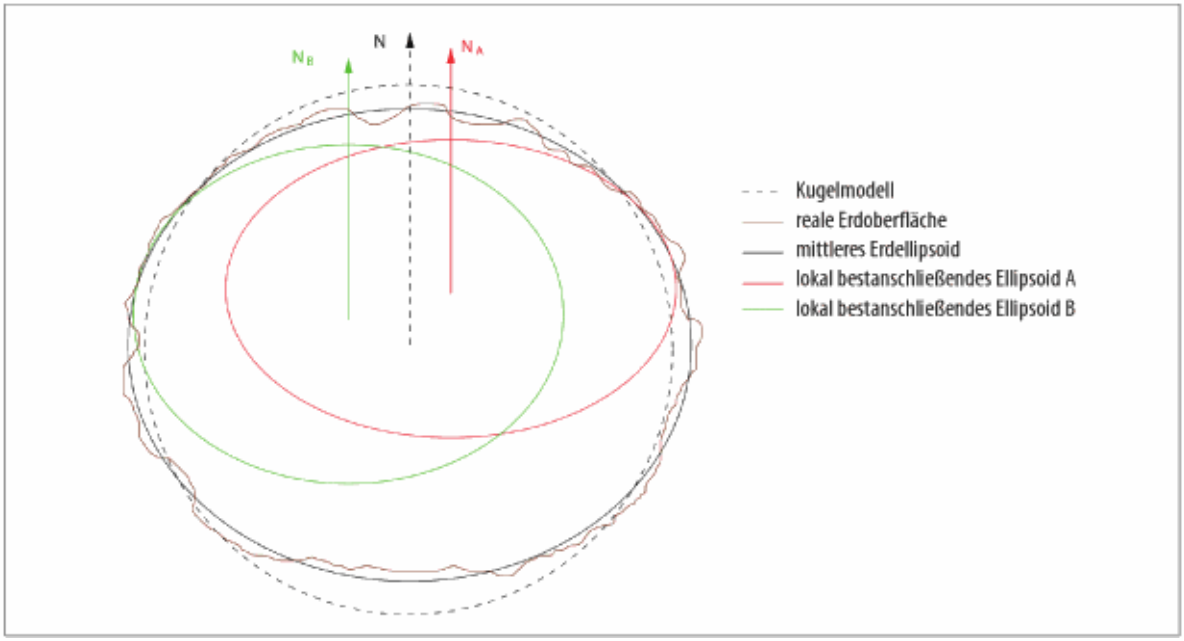

Abb. 4: Position unterschiedlicher Bezugsellipsoide (Quelle: Mitchell et al. (2008), S. 386)

[10] de Lange (2005), S. 167
[11] de Lange (2005), S. 177
[12] Mitchell et al. (2008), S. 385
[13] de Lange (2005), S. 181-183

Koordinatenreferenzsysteme (engl. Coordinate Reference System, kurz: CRS) werden definiert durch die verwendeten Koordinaten und ein bestimmtes geodätisches Datum. Der Begriff Bezugssystem umfasst hingegen neben den geometrischen Zusammenhängen, die durch das geodätische Datum definiert werden, auch noch physikalische Parameter, wie die Gravitationskonstante oder die Rotationsgeschwindigkeit der Erde, um die Position von Objekten eindeutig zu beschreiben.[14]

Um Koordinaten in ein anderes System zu überführen werden bestimmte Transformationsparameter zwischen den jeweiligen Daten benötigt. Stellt man sich die Koordinaten in einem 3-dimensionalen kartesischen Koordinatensystem vor, so genügen jeweils eine Translation und eine Rotation um jede der drei Achsen, um den Ursprung und die Achsen der beiden Systeme auf einander anzupassen. Durch die eventuelle Anpassung des Maßstabs, kommt zu diesen sechs Parametern noch ein siebter hinzu. Die Transformation zwischen zwei Koordinatenreferenzsystemen kann also über die sog. 7-Parameter-Transformation (auch: Helmert-Transformation) gelöst werden.[15]

Die Verwaltung dieser Transformationsinformationen im sog. EPSG Geodetic Parameter Dataset wurde von 1986 von der European Petroleum Survey Group (EPSG) geleistet, dessen Aufgabe ab dem Jahr 2005 das Surveying and Positioning Committee der International Association of Oil & Gas Producers (OGP) übernahm.[16] In dieser Datenbank werden viele Koordinatenreferenzsysteme der Erde samt ihren spezifischen Parametern vorgehalten und können über einen Code, den sog. EPSG Code, eindeutig identifiziert werden. Tabelle 1 zeigt anhand des EPSG Codes 31467 (entspricht: Gauß-Krüger Zone 3), welche Parameter für die Transformation zur Verfügung stehen.

[14] Mitchell et al. (2008), S. 386
[15] Mitchell et al. (2008), S. 387
[16] http://www.epsg.org

COORD_REF_SYS_CODE	31467
COORD_REF_SYS_NAME	"DHDN / Gauss-Kruger zone 3"
UOM_CODE	9001 (entspricht: metre)
SOURCE_GEOGCRS_CODE	4314 (entspricht: DHDN)
COORD_OP_CODE	16263 (entspricht: 3-degree Gauss-Kruger zone 3)
COORD_OP_METHOD_CODE	9807 (entspricht: Transverse Mercator)
SHOW_CRS	1
DEPRECATED	0
COORD_SYS_CODE	4530 (entspricht: Cartesian 2D CS. Axes: northing, easting (X,Y). Orientations: north, east. UoM: m.)
PARAMETER_CODE_1	8801 (entspricht: Latitude of natural origin)
PARAMETER_VALUE_1	0
PARAMETER_UOM_1	9102 (entspricht: degree)
PARAMETER_CODE_2	8802 (entspricht: Longitude of natural origin)
PARAMETER_VALUE_2	9
PARAMETER_UOM_2	9102 (entspricht: degree)
PARAMETER_CODE_3	8805 (entspricht: Scale factor at natural origin)
PARAMETER_VALUE_3	1
PARAMETER_UOM_3	9201 (entspricht: unity)
PARAMETER_CODE_4	8806 (entspricht: False easting)
PARAMETER_VALUE_4	3500000
PARAMETER_UOM_4	9001 (entspricht: metre)
PARAMETER_CODE_5	8807 (entspricht: False northing)
PARAMETER_VALUE_5	0
PARAMETER_UOM_5	9001 (entspricht: metre)
PARAMETER_CODE_6	-
PARAMETER_VALUE_6	-
PARAMETER_UOM_6	-
PARAMETER_CODE_7	-
PARAMETER_VALUE_7	-
PARAMETER_UOM_7	-

Tab. 1: EPSG Parameter für das Koordinatenreferenzsystem Gauß-Krüger in der Zone 3
(Quelle: Eigene Zusammenstellung aus Dateien des OSGeo4W Paket)

3 Die Programmierung des Plugins

3.1 Entwicklungsumgebung unter Microsoft Windows XP

Um individuelle Anpassungen in Quantum GIS unter Microsoft Windows XP vornehmen zu können, muss zunächst die Entwicklungsumgebung hierfür eingerichtet werden. Als Editorwerkzeug empfiehlt sich Microsoft Visual C++ 2008 Express, ein komfortabler und zudem kostenloser Editor mit integriertem C/C++ Compiler und Linker. Um Quantum GIS erfolgreich zu kompilieren, werden jedoch noch die Zusatzprogramme Flex, Bison und CMake benötigt, welche ebenfalls kostenlos heruntergeladen werden können.

Alle benötigten externen Bibliotheken für Quantum GIS sind in dem Paket OSGeo4W der Open Source Geospatial Foundation enthalten. Über die Umgebungsvariable „PATH“ müssen diese, unter Angabe der absoluten Pfade, dem System bekannt gemacht werden.

Zur Erstellung der graphischen Benutzeroberfläche eines Quantum GIS Plugins wird zusätzlich der Qt-Designer benötigt. Dieses Programm bietet, auf Grundlage der Qt Bibliotheken (vgl. Kap. 2.1), die Möglichkeit per Drag & Drop interaktive Bedienelemente in eine vorgefertigte graphische Oberfläche einzufügen und automatisch den entsprechenden Programmcode zu generieren. Nach der Implementierung in den eigenen Quelltext, kann auf die graphischen Elemente zugegriffen und so eine Interaktionsmöglichkeit für den Nutzer geschaffen werden.

Quantum GIS unterliegt der Open Source Versionsverwaltungssoftware Subversion (kurz: SVN), die den aktuellen Quellcode über das Internet zur Verfügung stellt. So können Änderungen jederzeit und von jedermann, sofern die Berechtigung vorhanden ist, global gespeichert werden. Subversion kann kostenlos im Internet heruntergeladen werden und wird zwingend benötigt um den Quellcode von Quantum GIS in der aktuellen Version bearbeiten zu können.

3.2 Grundstruktur des Quellcodes

Nach dem Download des aktuellen Quantum GIS Quellcodes mit Hilfe von SVN befindet sich dieser zur Bearbeitung auf der lokalen Festplatte. Im Ordner *src* befindet sich, in

Unterordnern aufgeteilt, die gesamte Programmierung von Quantum GIS, sodass bei Bedarf auch Änderungen am Kern oder der graphischen Benutzeroberfläche von Quantum GIS selbst vorgenommen werden können.

Zur Entwicklung eines Plugins wurden von den Entwicklern bereits vorgefertigte Strukturen im zugehörigen Ordner *plugins* abgelegt. Diese sollen als Vorlage dienen und garantieren somit weitgehend, dass sich alle Plugin-Entwickler an die gleichen Vorgaben und Konventionen halten. Die Vorlage besteht aus vier wesentlichen Bestandteilen: Die Plugin-Grundlagendateien (*plugin.cpp* und *plugin.h*), die Qt-Designer Datei als Vorlage für die graphische Benutzeroberfläche (*pluginguibase.ui*), die Plugin-Hauptdateien (*plugingui.cpp* und *plugingui.h*) sowie ein Icon des Plugins als .png-Datei. Das Hinzufügen weiterer Dateien für ein spezielles Plugin, sowie das Umbenennen der obigen Dateien steht jedem Entwickler jedoch frei.

Abb. 5: Schaltfläche zum Starten des Plugins in der QGIS Werkzeugleiste (Quelle: Eigene Darstellung)

Die Grundlagendateien *plugin.cpp* und *plugin.h* liefern grundlegende Funktionen, die jedes Plugin besitzen muss um innerhalb von Quantum GIS gestartet werden zu können. Hierzu gehören unter Anderem die Einbindung des Icons in die Quantum GIS Werkzeugleiste (`initGui()`) (vgl. Abb. 5), das Ausführen und Anzeigen des Plugins (`run()`) sowie das Entfernen aus der Werkzeugleiste bei Deaktivierung des Plugins durch den Benutzer (`unload()`).

Die Datei *pluginguibase.ui* ist eine XML-Datei zur Speicherung der wichtigsten Informationen der graphischen Benutzeroberfläche, welche mit dem Qt-Designer geöffnet und angepasst werden können. Während der Kompilierung wird diese Datei, mit Hilfe der Qt Zusatzbibliotheken, zur Header-Datei *ui_pluginguibase.h* konvertiert, sodass bei Import dieser Datei in einem C++-Dokument auf alle graphischen Elemente zugegriffen werden kann.

Die Hauptdateien *plugingui.cpp* und *plugingui.h* beinhalten die Hauptprogrammierung, sofern diese nicht in andere Klassen ausgelagert ist, und bilden die Schnittstelle zur graphischen Benutzeroberfläche. Durch den Import der zuvor erstellten Header-Datei *ui_pluginguibase.h* lässt sich ein Zugriff auf jedes graphische Element individuell durch sogenannte Slots realisieren. Dabei steht für jeden Zustand den ein Element einnehmen kann, ein Slot zur Verfügung. Bei Schaltflächen lassen sich somit neben dem eigentlichen

Klick (`clicked()`) auch das Drücken (`pressed()`) und das Loslassen (`released()`) der Maustaste mit individuellen Programmierungen belegen.

3.3 Das Graphical User Interface (GUI)

Das Graphical User Interface (kurz: GUI, zu Deutsch: graphische Benutzeroberfläche) bildet einen wichtigen Teil des Plugins. Es stellt die Schnittstelle zwischen dem Programmcode und der Benutzerinteraktion dar und sollte daher sowohl selbst erklärend und klar strukturiert als auch intuitiv benutzbar sein.

Das GUI ist in drei Bereiche unterteilt. Im oberen Bereich befindet sich die Auswahl der Ausgangsdatei, welche mit der Tastatur eingegeben oder mit Hilfe eines Dateibrowsers gewählt werden kann. Im darunter liegenden Bereich befinden sich zwei Karteireiter: „Info“ und „Reproject“. Die beiden Reiter sind im Startzustand des Plugins deaktiviert und somit nicht wählbar, da eine Informationsdarstellung oder eine Auswahl zur Reprojektion ohne Ausgangsdatei nicht sinnvoll ist (vgl. Abb. 6). Erst bei Auswahl einer, von GDAL unterstützten, Datei werden die Reiter aktiviert und die volle Funktionalität des Plugins kann genutzt werden. Ebenso wird das Informationsfenster mit Daten, die der Ausgangsdatei entnommen werden können, gefüllt (vgl. Abb. 7). Die Details hierzu werden in Kapitel 3.4 genauer erläutert.

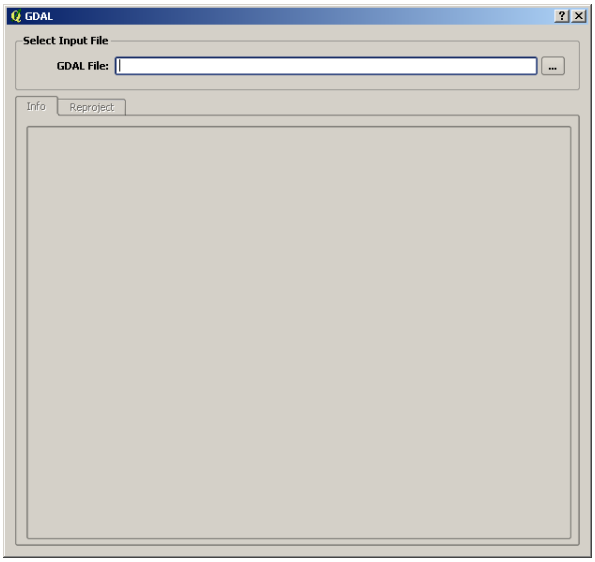

Abb. 6: Zustand GUI vor Auswahl einer Ausgangsdatei (Quelle: Eigene Darstellung)

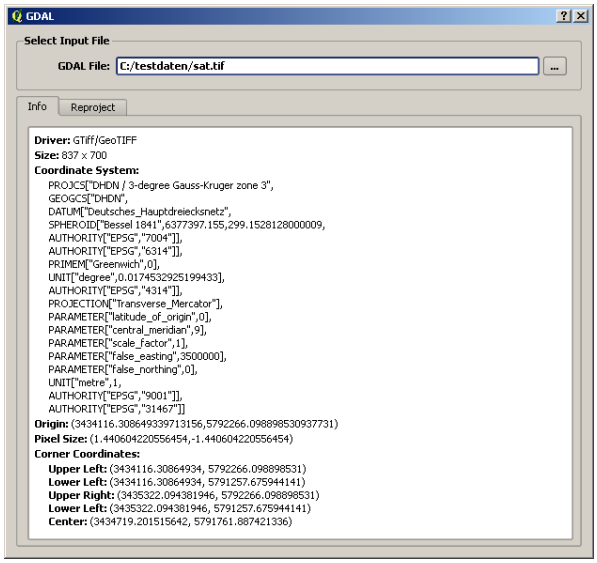

Abb. 7: Zustand GUI nach Auswahl der Ausgangsdatei (Quelle: Eigene Darstellung)

Unter dem Reiter „Reproject“ hat der Benutzer die Möglichkeit das Eingabebild in ein anderes Koordinatenreferenzsystem zu transformieren. Hierzu ist die Auswahl einer Ausgabedatei, sowie eines Koordinatenreferenzsystems, in das transformiert werden soll, erforderlich (vgl. Abb. 8).

Das Koordinatenreferenzsystem kann über ein interaktives Bedienelement, das von Quantum GIS zur Verfügung gestellt wird, ausgewählt werden. Dieses bietet sowohl die Möglichkeit nach Name oder ESPG-Code zu suchen als auch über eine Baumstruktur das entsprechende System direkt auszuwählen. Häufig verwendete Systeme werden durch Schaltflächen am Ende des Elements gut sichtbar aufgelistet (vgl. Abb. 8).

Erst wenn Ausgabedatei und Koordinatenreferenzsystem korrekt gewählt wurden, werden die beiden Schaltflächen, die den Transformationsvorgang starten, aktiviert. Der Benutzer hat die Wahl, ob er über die Schaltfläche „Reproject“ lediglich die Reprojektion durchführen lassen will oder ob mittels der Schaltfläche „Reproject & Add“, nach der erfolgreichen Reprojektion, die neu erstellte Datei direkt zur Layerauswahl von Quantum GIS hinzugefügt werden soll. Über eine Statusanzeige am unteren Rand des Plugin-Fensters wird der Benutzer über den momentanen Zustandes des Plugins während der Transformation informiert (vgl. Abb. 8).

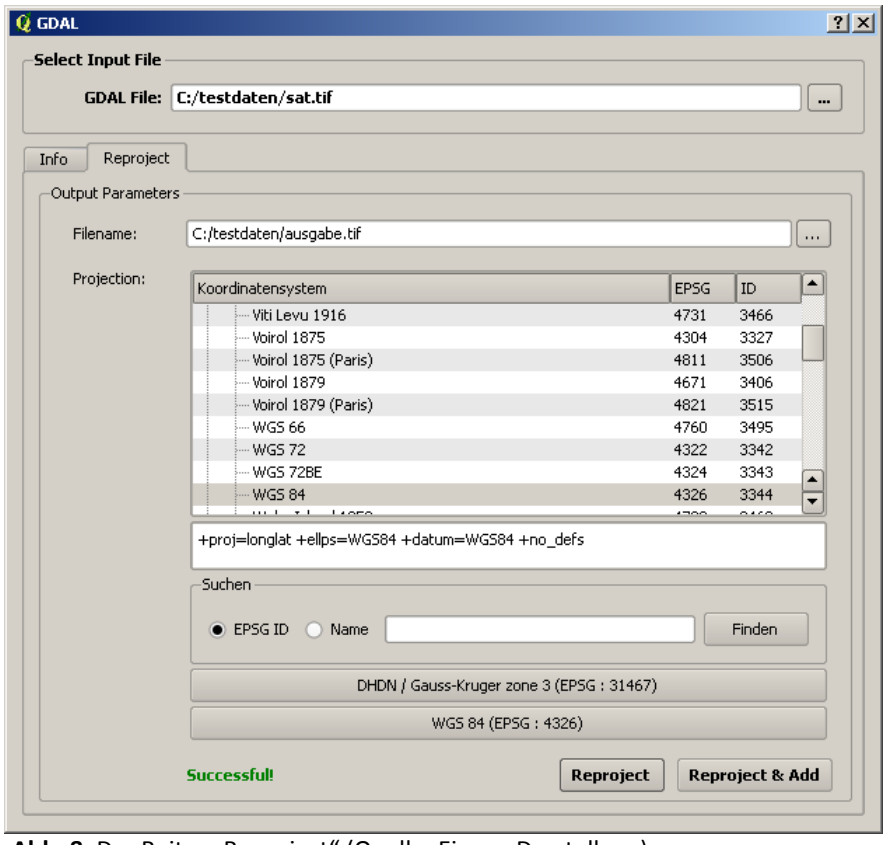

Abb. 8: Der Reiter „Reproject“ (Quelle: Eigene Darstellung)

3.4 Programmierung der Analysefunktion

Zur korrekten Darstellung einer Rasterdatei werden viele verschiedene Informationen benötigt. Hierzu zählen triviale Eigenschaften wie die Anzahl der Pixel in der Breite und der Höhe, aber zum Beispiel auch detailierte Informationen über die einzelnen Rasterbänder. Ein vorbereitender Schritt war es also, den Umfang der darzustellenden Informationen einzugrenzen. Aus eigenen Erfahrungen werden folgende Informationen im Alltag am häufigsten genutzt und sollen daher nachfolgend aus der Rasterdatei ausgelesen werden:

- Treiber bzw. Dateiformat
- Breite und Höhe
- Koordinatenreferenzsystem in WKT-Darstellung
- Ursprung der Rasterdatei
- Pixelgröße in der Szene
- Koordinaten der Ecken sowie der Mitte des Bildes

Bei Auswahl einer, von GDAL unterstützen, Ausgangsdatei über die graphische Oberfläche des Plugins (vgl. Kap. 3.3), wird zum Füllen des Informationsbereiches die Methode `fillInfoField()` aufgerufen. Zu Beginn dieser Methode werden die Variablen, welche für den weiteren Verlauf entscheidend sind, initiiert:

```
GDALDatasetH inputDS;
GDALDriverH inputDriver;
```

Die Variable `inputDS` vom Typ `GDALDatasetH` ist Container für die Ausgangsdatei und `inputDriver` vom Typ `GDALDriverH` speichert den Treiber, also das Dateiformat. Im Folgenden werden diese entsprechend mit Inhalt gefüllt:

```
GDALAllRegister();
inputDS = GDALOpen( leInputFile->text().toUtf8(), GA_ReadOnly );
inputDriver = GDALGetDatasetDriver( inputDS );
```

`GDALAllRegister` registriert alle bekannten GDAL Treiber, was für die folgenden Schritte zwingend von Nöten ist. `leInputFile` ist die globale Variable für das Element zur Erfassung der Ausgangsdatei auf dessen Inhalt mit Hilfe der Methode `text()` zugegriffen werden kann. `toUtf8()` konvertiert diese Zeichenkette zum Datentyp `QByteArray`, der vom Compiler zum Datentyp `const char *` gecastet werden kann und als Parameter von der Methode `GDALOpen()` erwartet wird. `GDALOpen()` öffnet den Datensatz mit einer entsprechenden

Zugriffsmethode (`GA_ReadOnly` für lesenden oder `GA_Update` für schreibenden Zugriff), sodass auf den Inhalt der Ausgangsdatei zugegriffen werden kann. Mit Hilfe der Methode `GDALGetDatasetDriver()` wird der Treiber aus der nun geöffneten Datei ausgelesen.

Die Daten, die in das Informationsfenster eingefügt werden sollen, werden in einer Variable vom Typ QString zwischengespeichert und mittels HTML formatiert:

```
QString information = "<html><body>";
information += "<table width=\"100%\">";
information += "<tr><td>";
information += tr( "<b>Driver:</b> %1/%2" )
                    .arg( GDALGetDriverShortName( inputDriver ) )
                    .arg( GDALGetDriverLongName( inputDriver ) );
information += "</td></tr>";
information += "<tr><td>";
information += tr( "<b>Size:</b> %1 x %2" )
                    .arg( GDALGetRasterXSize( inputDS ) )
                    .arg( GDALGetRasterYSize( inputDS ) );
information += "</td></tr>";
```

Die Variable `information` kann nun mit entsprechenden Daten gefüllt werden. Zur Bestimmung der Treibernamen werden die Methoden `GDALGetDriverShortName()` und `GDALGetDriverLongName()` verwendet, welche den kurzen bzw. den langen Namen des Treibers ausgeben. Die Größe der Rasterdatei wird mit den Methoden `GDALGetRasterXSize()` und `GDALGetRasterYSize()` ermittelt. Die Methode `tr()` der Klasse `QObject` wird in Quantum GIS verwendet um verschiedene Sprachversionen anbieten zu können. `tr()` liefert die übersetzte Version der enthaltenen Zeichenkette, sofern diese entsprechend hinterlegt ist. Mit der Methode `arg()` lassen sich zusätzliche Textbausteine an ein Objekt der Klasse `QString` anfügen. Die Platzhalter, wie z.B. `%1`, werden durch den entsprechenden Textbaustein ersetzt und verschmelzen so zu einer Zeichenkette.

Zur Ermittlung des Koordinatenreferenzsystems der Ausgangsdatei sind einige Zwischenschritte von Nöten. Zunächst muss mit Hilfe der Methode `GDALGetProjectionRef()` überprüft werden, ob die Ausgangsdatei überhaupt eine Projektion besitzt. Diese Methode liefert entweder die Projektion der Rasterdatei als Zeichenkette oder den Nullwert `NULL` zurück. Somit soll eine weitere Überprüfung nur durchgeführt werden, wenn dieser Nullwert nicht zurückgeliefert wird:

```
if( GDALGetProjectionRef( inputDS ) != NULL ){
      …
}
```

Zur besseren Lesbarkeit der Informationen über das Koordinatenreferenzsystem, sollen diese in eingerückter WKT Darstellung angezeigt werden. Diese Darstellung wird von der Unterbibliothek von GDAL für Vektordaten, OGR, unterstützt, sodass wir die Projektion zunächst in die entsprechende Datenstruktur überführen müssen:

```
OGRSpatialReferenceH inputSRS;
char *projection;
projection = (char *) GDALGetProjectionRef( inputDS );
inputSRS = OSRNewSpatialReference(NULL);
if( OSRImportFromWkt( inputSRS, &projection ) == CE_None ){
      …
}
```

In der Variable `projection` wird zunächst das Koordinatenreferenzsystem der Rasterdatei als Zeichenkette gespeichert. Die Variable `inputSRS` vom Typ `OGRSpatialReferenceH` ist der Datenspeicher gemäß OGR für eben dieses und wird durch die Methode `OSRNewSpatialReference()` mit einem leeren Koordinatenreferenzsystem initialisiert. Durch die, in die If-Schleife eingebettete, Methode `OSRImportFromWkt()` wird überprüft, ob aus der Projektion der Rasterdatei ein gültiges Koordinatenreferenzsystem erstellt werden kann. Nur wenn dies der Fall ist, kann die eingerückte WKT-Darstellung erstellt werden:

```
char *prettyWkt = NULL;
OSRExportToPrettyWkt( inputSRS, &prettyWkt, FALSE );
QString wktFormat = prettyWkt;
wktFormat.replace(QString("\n"), QString("<br>"));
```

Die Methode `OSRExportToPrettyWkt()` erledigt diese Aufgabe und speichert das Ergebnis in die Variable `prettyWkt`. Diese Zeichenkette wird nun in einer Instanz der Klasse `QString` namens `wktFormat` gespeichert und mittels der Methode `replace()` an die HTML-Formatierung angepasst. Diese Informationen können nun nach dem oben beschriebenen Verfahren an die Variable `information` angefügt werden.

Als letzte Informationen sollen Ursprung, Pixelgröße und die Ecken der Rasterdatei ermittelt werden. Hierzu werden die Transformationsparameter aus der Klasse `GDALDataSetH` verwendet:

```
double geotransform[6];
if( GDALGetGeoTransform( inputDS, geotransform ) == CE_None ){
      if( geotransform[2] == 0.0 && geotransform[4] == 0.0 ){
            …
      }else{
            …
      }
}
```

Mit Hilfe der Methode `GDALGetGeoTransform()` werden die Parameter der Rasterdatei ausgelesen und in dem Array `geotransform` gespeichert. Die folgende Analyse soll nur durchgeführt werden, wenn das Ausgangsbild diese Transformationsparameter besitzt.

Der Ursprung und die Pixelgröße sollen nur bei genordeten Rasterbildern ausgegeben werden, welche durch einen Wert von null im dritten und fünften Element des Arrays gekennzeichnet sind.[16] In diesem Fall entspricht der erste und vierte Wert des Arrays dem Ursprung und der zweite und sechste Wert der Pixelgröße[17], die nach dem obigen Verfahren in die Variable `information` eingefügt werden können. Ist der dritte oder der fünfte Werte nicht gleich null, so werden alle sechs Werte des Arrays in die Ausgabe geschrieben.

Zur Bestimmungen der Ecken, also der Ausdehnung der Rasterdatei, lassen sich folgende Formeln definieren:

```
Oben links:   ( X-Wert Ursprung, Y-Wert Ursprung )
Unten links:  ( X-Wert Ursprung, (Höhe * Pixelgröße für Y ) + Y-Wert
                  Ursprung)
Oben rechts:  ( (Breite * Pixelgröße für X) + X-Wert Ursprung, Y-Wert
                  Ursprung)
Unten rechts: ( (Breite * Pixelgröße für X) + X-Wert Ursprung, (Höhe
                  * Pixelgröße für Y ) + Y-Wert Ursprung)
```

Diese lassen sich mit den bekannten Werten des `geotransform` Arrays und den Methoden `GDALGetRasterXSize()` und `GDALGetRasterYSize()` programmtechnisch umsetzen.
Abschließend muss das Informationsfenster mit den gesammelten Daten gefüllt werden:

```
teInformation->clear();
teInformation->setHtml( information );
```

Durch die Methode `setHtml()` wird dem Textelement der darzustellende Inhalt übergeben und die Formatierungsmethode bekannt gemacht. Hinsichtlich des Speicheraufkommens sollte der Datensatz mit der Methode `GDALClose()` geschlossen und die registrierten GDAL-Treiber mit der Methode `GDALDestroyDriverManager()` wieder freigegeben werden:

```
GDALClose( inputDS );
GDALDestroyDriverManager();
```

[17] http://www.gdal.org/gdal_tutorial.html

3.5 Programmierung der Reprojektion

Beide Schaltflächen der graphischen Benutzeroberfläche zur Aktivierung des Transformationsvorganges (vgl. Kap. 3.3) rufen die Methode `reprojectImage()` auf. Diese überführt die Ausgangsdatei in das gewählte Koordinatenreferenzsystem und speichert das Ergebnis unter dem gewählten Dateinamen ab.

Zu Beginn der Methode werden, analog zur Analysefunktion (vgl. Kap. 3.4), die Ausgangsdatei geöffnet und der GDAL Treiber festgelegt:

```
GDALDatasetH sourceData;
GDALDriverH driver;
const char *format = "GTiff";

sourceData = GDALOpen( leInputFile->text().toUtf8(), GA_ReadOnly );
driver = GDALGetDriverByName( format );
```

Anders als bei der Analysefunktion wird hier jedoch als Treiber für die Ausgabedatei das Format GeoTiff festgelegt. Die Integration einer Auswahl des Ausgabeformats hätte auf Grund der differenzierten Schreibrechte für die verschiedenen GDAL-Formate[18] den Zeitrahmen, welche zur Erstellung dieser Arbeit zur Verfügung stand, weit überschritten.

Um im späteren Verlauf einen Zusammenhang zwischen dem Ausgangskoordinatenreferenzsystem und dem neuem Koordinatenreferenzsystem herstellen zu können, müssen diese zunächst zwischengespeichert werden:

```
char *newSRS = "";
char *sourceSRS = "";
sourceSRS = strdup(GDALGetProjectionRef( sourceData ));

QString wkt = QgsCoordinateReferenceSystem(
              projectSelector->selectedCrsId(),
              QgsCoordinateReferenceSystem::InternalCrsId ).toWkt();
newSRS = wkt.toUtf8().data();
```

`GDALGetProjectionRef()` liefert das Koordinatenreferenzsystem der Ausgangsdatei, das mit Hilfe der Methode `strdup()` in die Variable `sourceSRS` kopiert wird. `projectSelector` ist eine Instanz der Quantum GIS Klasse `QgsProjectionSelector`, die zur Auswahl des Koordinatenreferenzsystems in die graphische Oberfläche eingefügt wurde (vgl. Kap. 3.3). Über die Methode `selectedCrsId()` lässt sich aus diesem Objekt der interne Bezeichner des ausgewählten Systems auslesen. Dieser Bezeichner wird zur Instanziierung einer

[18] http://www.gdal.org/formats_list.html

weiteren Quantum GIS Klasse namens `QgsCoordinateReferenceSystem` verwendet. Diese Klasse wird zur Speicherung eines Koordinatenreferenzsystems verwendet und kann mittels der Methode `toWkt()` selbstständig die entsprechende WKT-Darstellung liefern. Diese wird unter Verwendung der Methode `data()`, die ein `QByteArray` in den Datentyp `char *` überführt, in der Variable `newSRS` gespeichert.

Mit Hilfe der Methode `GDALCreateGenImgProjTransformer()` kann nun ein Objekt erstellt werden, das den Zusammenhang zwischen den beiden Koordinatenreferenzsystemen beschreibt. Die 3 letzten Parameter dieser Methode beziehen sich auf die Verwendung von Ground Control Points und können daher vernachlässigt werden:

```
void *transformer;
transformer = GDALCreateGenImgProjTransformer( sourceData, sourceSRS,
                                                NULL, newSRS,
                                                FALSE, 0, 0 );
```

Mit Hilfe der ermittelten Beziehung zwischen den beiden Koordinatenreferenzsystemen lassen sich nun die, für den Transformationsprozess, nötigen Parameter errechnen:

```
double geotransform[6];
int width=0, height=0;
if( GDALSuggestedWarpOutput( sourceData, GDALGenImgProjTransform,
                             transformer, geotransform, &width,
                             &height ) != CE_None )
      return false;
```

Die Methode `GDALSuggestedWarpOutput()` übernimmt diese Aufgabe und errechnet mit Hilfe der vorher erstellten Variable `transformer` die Transformationsparameter (`geotransform`), sowie die Breite (`width`) und die Höhe (`height`) der späteren Ausgabedatei. Liefert diese Methode einen Fehler oder keine Werte, so wird der Reprojektionsvorgang abgebrochen, da eine Fortführung ohne die oben genannten Werte nicht möglich ist.

Auf Grundlage der ermittelten Werte für Breite und Höhe der Ausgabedatei kann diese nun mit Hilfe der Methode `GDALCreate()` erstellt werden:

```
GDALDatasetH newData;
newData = GDALCreate( driver, leOutputFile->text().toUtf8(), width,
      height, GDALGetRasterCount(sourceData),
      GDALGetRasterDataType(GDALGetRasterBand(sourceData,1)), NULL );
```

Dabei werden die Anzahl der Rasterbänder (`GDALGetRasterCount()`) sowie deren Typ (`GDALGetRasterDataType()`) von der Ausgangsdatei übernommen. Über die Methode

text() der Variable leOutputFile lässt sich der gewünschte Speicherort der zu erstellenden Datei auslesen.

Nachdem die Ausgabedatei erstellt wurde, können die Projektion, die Transformationsparameter und eine eventuelle Farbtabelle für die Datei hinterlegt werden:

```
GDALSetProjection( newData, newSRS );
GDALSetGeoTransform( newData, geotransform );

colorTable = GDALGetRasterColorTable( GDALGetRasterBand(sourceData,1)
);
if( colorTable != NULL )
      GDALSetRasterColorTable( GDALGetRasterBand(newData,1),
      colorTable );
```

Projektion und Transformationsparameter werden mit Hilfe der Methoden GDALSetProjection() und GDALSetGeoTransform() gespeichert. Die Methode GDALGetRasterColorTable() ermittelt vom ersten Rasterband der Ausgangsdatei die enthaltene Farbtabelle und übernimmt diese, falls vorhanden, mittels GDALSetRasterColorTable() auch für die Ausgabedatei.

Alle Vorbereitungen für die Transformation sind nun getroffen, sodass diese durch die Methode GDALSimpleImageWarp() durchgeführt werden kann:

```
GDALSimpleImageWarp( sourceData, newData, 0, NULL,
                     GDALGenImgProjTransform, transformer,
                     GDALTermProgress, NULL, NULL );
```

Die Parameter dieser Methode sind neben der Ausgangsdatei (sourceData) und der Ausgabedatei (newData), die Anzahl der zu transformierenden Bänder (0 = Alle Bänder), sowie ein Array eben dieser (hier: NULL), die interne Transformationsfunktion (GDALGenImgProjTransform) mit den zugehörigen Parametern (transformer), eine Reportfunktion (GDALTermProgress) mit zugehörigem Ausgabezeiger (hier: NULL), sowie ein 2-dimensionales Array zur Übergabe von optionalen Transformationsparametern (ebenfalls NULL).

Nach erfolgreichem Abschluss des Reprojektionsvorgangs sind die Datensätze mit der Methode GDALClose() zu schließen und die registrierten GDAL-Treiber mit der Methode GDALDestroyDriverManager() wieder freizugeben:

```
GDALClose( newData );
GDALClose( sourceData );
GDALDestroyDriverManager();
```

4 Zusammenfassung

Das in dieser Arbeit beschriebene Plugin bietet dem Benutzer von Quantum GIS eine schnelle und komfortable Möglichkeit vorhandene Rasterdateien zu analysieren und zu transformieren. Diese Möglichkeit war in Quantum GIS zuvor nicht gegeben und stellt daher für den Anwender einen deutlichen Mehrwert dar.

Abb. 9: Luftbild im Koordinatensystem WGS84 (Quelle: Google Maps)

Mit geringem Zeitaufwand lässt sich nun zum Beispiel ein Luftbild, welches im Koordinatensystem WGS84 vorliegt (vgl. Abb. 9), in das Koordinatensystem Gauß-Krüger (vgl. Abb. 10) überführen. Alle, zu Beginn dieser Arbeit, gesteckten Ziele sind somit durch das Plugin realisiert worden.

Abb. 10: Luftbild im Koordinatensystem Gauß-Krüger (ursprüngliche Quelle: Google Maps)

Plugins, die in der Programmiersprache C++ geschrieben wurden, müssen in ein offizielles Release von Quantum GIS aufgenommen werden und können nicht ohne Weiteres jedem Benutzer zugänglich gemacht werden. Dies ist lediglich für Plugins, die in der Programmiersprache Python geschrieben wurden, möglich. Diese können in eine öffentliche Datenbank geladen und von jedem Nutzer verwendet werden. Um in ein zukünftiges Quantum GIS Release aufgenommen zu werden, muss das Plugin einer längeren Testphase unterzogen und mögliche Fehler behoben werden. Die Geospatial Data Abstraction Library bietet außerdem noch eine Vielzahl von Funktionen um die das Plugin erweitert werden könnte. So besteht zum Beispiel die Möglichkeit Mosaikbilder zu erstellen oder Vektor- in Rasterdateien zu konvertieren.

Auch am bisherigen Umfang des Plugins lassen sich einige zusätzliche Funktionen ergänzen, dessen Implementierung auf Grund des Zeitrahmens dieser Arbeit nicht möglich war. Hier sind speziell die Auswahl eines Ausgabedateiformates für die Reprojektion (vgl. Kap. 3.5) und

die Transformation anhand von Ground Control Points (vgl. Kap. 2.2) zu nennen. Der Transformationsprozess von GDAL bietet zudem einige spezielle Zusatzoptionen, wie zum Beispiel das Setzen von Alpha- oder NoData-Werten, an, dessen Implementierung für die Zukunft ebenfalls sinnvoll wäre.

Zusammenfassend lässt sich festhalten, dass sich Quantum GIS für Benutzer mit einem speziellen, häufig wiederkehrendem, Anwendungsfall durch ein Plugin gut den eigenen Bedürfnissen anpassen lässt. Nicht von der Hand zu weisen ist jedoch der hohe Aufwand und die nötige Programmierkenntnis zur Realisierung eines solchen Vorhabens.

Generell denke ich, dass Quantum GIS, durch die große Zahl an Mitwirkenden und dem dadurch resultierenden schnellen Entwicklungsprozess in den kommenden Jahren einen großen Funktionsumfang erlangen wird. Aktuelle Erkenntnisse und neue Standards werden über Mailinglisten und andere Medien schnell verbreitet und oft innerhalb weniger Tage programmtechnisch umgesetzt und zur Verfügung gestellt. Die Zeit wird zeigen, inwieweit Quantum GIS die momentane Entwicklungsgeschwindigkeit beibehalten kann.

5 Literatur

Publikationen

de Lange, N. (2005): *Geoinformatik in Theorie und Praxis.* Berlin: Springer, 2. Auflage.

Mitchell, T. (2009): *OSGeo Annual Report 2008.* OSGeo.

Mitchell, T., Emde, A. und Christl, A. (2008): *Web-Mapping mit Open Source-GIS-Tools.* O'Reilly.

Neteler, M., und Mitasova, H. (2008). *Open Source GIS: A Grass GIS Approach.* Berlin: Springer, Third Edition

Internetquellen

GDAL: GDAL – Geospatial Data Abstraction Library

http://www.gdal.org/ vom 27.09.2009

http://www.gdal.org/gdal_datamodel.html vom 30.08.2009

http://www.gdal.org/gdal_tutorial.html vom 29.09.2009

http://www.gdal.org/warptut.html vom 12.09.2009

http://www.gdal.org/annotated.html vom 16.09.2009

http://www.gdal.org/ogr/hierarchy.html vom 14.09.2009

http://www.gdal.org/gdal_8h.html vom 20.09.2009

http://www.gdal.org/formats_list.html vom 29.09.2009

Google Maps

http://maps.google.de/ vom 30.08.2009

OGP Surveying & Positioning Committee

http://www.epsg.org/ vom 22.09.2009

OSGeo.org

http://www.osgeo.org/ vom 24.09.2009

http://www.osgeo.org/journal vom 28.09.2009

Qt 4.5

http://doc.trolltech.com/4.5/classes.html vom 12.09.2009

Quantum GIS

http://www.qgis.org/ vom 27.09.2009

http://blog.qgis.org/ vom 24.08.2009

http://doc.qgis.org/head/ vom 12.09.2009

http://doc.qgis.org/head/classes.html vom 12.09.2009

http://www.qgis.org/de/ueber-qgis.html vom 24.08.2009

http://www.qgis.org/de/ueber-qgis/features.html vom 02.09.2009

http://www.qgis.org/wiki/Developers_Corner vom 24.09.2009

http://www.qgis.org/wiki/File:QGISHighLevelArchitecture.png vom 21.09.2009

http://wiki.qgis.org/qgiswiki/BuildingFromSource vom 01.08.2008

Stefan Jahn – Stefans Webseite – Wiki: Qt installieren

http://stefanjahn.de/cpp:windows:qt_installieren vom 01.08.2009

6 Abbildungs- und Tabellenverzeichnis

Anhang

A.1 Quellcode

Quellcode der Datei *gdalgui.h*:

```
#ifndef GDALGUI_H
#define GDALGUI_H

#include <QDialog>
#include <ui_gdalguibase.h>

class QgisInterface;

class gdalGui : public QDialog, private Ui::gdalGuiBase
{
    Q_OBJECT
  public:
    gdalGui( QgisInterface* theQgisInterface, QWidget* parent = 0,
Qt::WFlags fl = 0 );
    ~gdalGui();

  public slots:
    void on_btnInputFileLoad_clicked();
    void on_btnOutputFileLoad_clicked();
    void on_leOutputFile_textChanged(QString);
    void on_projectSelector_sridSelected(QString);
      void on_btnReproject_pressed();
    void on_btnReproject_clicked();
      void on_btnReprojectAdd_pressed();
    void on_btnReprojectAdd_clicked();

  private:
    bool reprojectImage();
    void fillInfoField();
    static const int context_id = 0;

    QAction* mInputFileSelected;
    QgisInterface* mIface;
};

#endif
```

Quellcode der Datei *gdalgui.cpp*:

```
//standard includes
#include <gdalgui.h>
#include <gdal_priv.h>
#include <gdal.h>
#include <gdalwarper.h>
#include <gdal_alg.h>
#include <ogr_spatialref.h>
#include <ogr_srs_api.h>
#include <qgsrasterlayer.h>
#include <qgscontexthelp.h>
#include <qgsapplication.h>
#include <qgisinterface.h>
#include <qgscoordinatereferencesystem.h>
```

```
#include <cpl_string.h>

//qt includes
#include <QFile>
#include <QFileDialog>
#include <QString>
#include <QSettings>
#include <QMessageBox>
#include <QFileInfo>
#include <QErrorMessage>
#include <QTextStream>

//Constructor
gdalGui::gdalGui( QgisInterface* theQgisInterface, QWidget* parent,
Qt::WFlags fl )
    : QDialog( parent, fl ), mIface( theQgisInterface )
{
  setupUi( this );

  tabWidget->setCurrentIndex(0);
  tabWidget->setEnabled(false);
  btnReproject->setEnabled(false);
  btnReprojectAdd->setEnabled(false);
}

//Destructor
gdalGui::~gdalGui()
{
}

//Click-Event on the InputFileLoad-Button
void gdalGui::on_btnInputFileLoad_clicked()
{
    QSettings settings;
    QString dir = settings.value( "/Plugin-GDAL/inputRasterDir"
).toString();
    if ( dir.isEmpty() )
        dir = ".";
      //OpenFile Dialog
    QString fileName = QFileDialog::getOpenFileName( this,
                                    tr( "Choose a GDAL file" ),
                                    dir,
                                    tr( "GDAL files (*.*)" ) );

    if ( fileName.isNull() )
        return;

    leInputFile->setText( fileName );    //fill textfield

      //check if the file is a valid rasterfile
    if ( !QgsRasterLayer::isValidRasterFileName( fileName ) ){
        QMessageBox::critical( this, tr( "Error" ),
                           tr( "The selected file is not a valid GDAL file."
) );
        return;
    }

    QFileInfo fileInfo( fileName );
      //Remember the choosen dir in plugin settings
    settings.setValue( "/Plugin-GDAL/inputRasterDir", fileInfo.path() );

    fillInfoField();   //fill info field
```

```
    tabWidget->setEnabled(true);    //enable the tab structure
}

//Click-Event on the OutputFileLoad-Button
void gdalGui::on_btnOutputFileLoad_clicked()
{
    QSettings settings;
    QString dir = settings.value( "/Plugin-GDAL/outputRasterDir"
).toString();
    if ( dir.isEmpty() )
        dir = ".";
      //open SaveFile-Dialog
    QString fileName = QFileDialog::getSaveFileName( this,
                                tr( "Choose output" ),
                                dir,
                                tr( "GDAL files (*.*)" ) );

    if ( fileName.isNull() )
        return;

    leOutputFile->setText( fileName );    //fill textfield

    QFileInfo fileInfo( fileName );
      //remember output folder in plugin settings
    settings.setValue( "/Plugin-GDAL/outputRasterDir", fileInfo.path() );
}

//TextChanged-Event on the OutputFile-Textfield to register manipulations
on the output filename
void gdalGui::on_leOutputFile_textChanged(QString content)
{
    QFileInfo fileInfo(leOutputFile->text());  //get fileinfo of the new
filename
    if(!fileInfo.isWritable())                              //check if the
file isn't writable
    {
        btnReproject->setEnabled(false);       //disable the reproject
buttons
        btnReprojectAdd->setEnabled(false);
    }
    else
    {
        if(projectSelector->selectedEpsg() != 0 && leOutputFile->text() !=
"")
        {
            btnReproject->setEnabled(true);          //enable reproject
buttons if a SRS and a valid file are choosen
            btnReprojectAdd->setEnabled(true);
        }
        else
        {
            btnReproject->setEnabled(false);
            btnReprojectAdd->setEnabled(false);
        }
    }
}

//Selection-Event on the projectionSelector-Widget
void gdalGui::on_projectSelector_sridSelected(QString)
{
    if(projectSelector->selectedEpsg() != 0)
        if(leOutputFile->text() != "")
        {
```

```
            btnReproject->setEnabled(true);          //enable buttons if a
srs and a valid file are choosen
            btnReprojectAdd->setEnabled(true);
        }
        else
        {
            btnReproject->setEnabled(false);   //disable them otherwise
            btnReprojectAdd->setEnabled(false);
        }
    else
    {
        btnReproject->setEnabled(false);       //disable them otherwise
        btnReprojectAdd->setEnabled(false);
    }
}

//Pressed-Event on the "Reproject & Add"-Button
void gdalGui::on_btnReprojectAdd_pressed()
{
      laOutput->setText("Please wait...");     //set the status textfield
}

//Click-Event on the "Reproject & Add"-Button
void gdalGui::on_btnReprojectAdd_clicked()
{
      if(reprojectImage()){
                        //start reproject process
            laOutput->setText("<font color='green'>Successful!</font>");
      //change status textfield on success
        mIface->addRasterLayer( leOutputFile->text() );
            //and add the raster as a new layer
    }else
        laOutput->setText("<font color='red'>Error!</font>");
      //change status textfield on error
}

//Pressed-Event on the Reproject-Button
void gdalGui::on_btnReproject_pressed()
{
      laOutput->setText("Please wait...");     //set the status textfield
}

//Click-Event on the Reproject-Button
void gdalGui::on_btnReproject_clicked()
{
      if(reprojectImage())
                        //start reproject process
        laOutput->setText("<font color='green'>Successful!</font>");
      //change status textfield on success
    else
        laOutput->setText("<font color='red'>Error!</font>");
      //change status textfield on error
}

//funtion to reproject the choosen image
bool gdalGui::reprojectImage()
{
      GDALDatasetH           sourceData, newData;               //source
and new Dataset
    const char         *format = "GTiff";                  //default format
is GeoTIFF
      GDALDriverH            driver;
      //gdal driver for the output format
```

```
    char               *newSRS = "";                              //new
spatial reference system
    char               *sourceSRS = "";                           //source
spatial reference system
    int                 order = 0;                                //maximum
order to use for transformation. 0 = autoselect
    void               *transformer;                              //image to
image transformer
    double                    geotransform[6];
      //suggested geotransform for the new image
    int                       width=0, height=0;
      //suggested width and height of the new image
    GDALColorTableH    colorTable;
      //colorTable of the source file

    GDALAllRegister();
      //register all gdal formats

    sourceData = GDALOpen( leInputFile->text().toUtf8(), GA_ReadOnly );
      //open source data with read-only access
      driver = GDALGetDriverByName( format );

    if( sourceData == NULL )
           return false;
      //false if source is null

    if ( GDALGetRasterCount(sourceData) == 0 )
           return false;
      //return false if source has no raster bands

      //save the source srs
      sourceSRS = strdup(GDALGetProjectionRef( sourceData ));

      //create wkt representation of the selected srs
      QString wkt = QgsCoordinateReferenceSystem(
                projectSelector->selectedCrsId(),
                QgsCoordinateReferenceSystem::InternalCrsId ).toWkt();

      newSRS = wkt.toUtf8().data();

      //create image transformer from the source to the new srs
    transformer = GDALCreateGenImgProjTransformer( sourceData, sourceSRS,

      NULL, newSRS,

      FALSE, 0, 0 );
    if( transformer == NULL )
           return false;

      //suggest output parameter: geotransform, width and height
    if( GDALSuggestedWarpOutput( sourceData,
                                 GDALGenImgProjTransform, transformer,
                                 geotransform, &width, &height ) != CE_None
)
           return false;

    GDALDestroyGenImgProjTransformer( transformer ); //clean up

      //create the new image
    newData = GDALCreate( driver, leOutputFile->text().toUtf8(), width,
height,
                          GDALGetRasterCount(sourceData),
```

```
GDALGetRasterDataType(GDALGetRasterBand(sourceData,1)),
                        NULL );
    if( newData == NULL )
            return false;

    GDALSetProjection( newData, newSRS );            //set projection to
the new image
    GDALSetGeoTransform( newData, geotransform );    //set geotransform to
the new image

      //copy colortable if necessary
    colorTable = GDALGetRasterColorTable( GDALGetRasterBand(sourceData,1)
);
    if( colorTable != NULL )
        GDALSetRasterColorTable( GDALGetRasterBand(newData,1), colorTable
);

      //create a new transformer with the new image file
      transformer = GDALCreateGenImgProjTransformer( sourceData, sourceSRS,
                                        newData, newSRS,
                                        FALSE, 0, 0 );
    if( transformer == NULL )
            return false;

      //start the warp process
    GDALSimpleImageWarp( sourceData, newData, 0, NULL,
                         GDALGenImgProjTransform, transformer,
                         GDALTermProgress, NULL, NULL );

    GDALClose( newData );          //close datasets
    GDALClose( sourceData );

    GDALDestroyDriverManager();

      return true;
}

//function to fill the info field
void gdalGui::fillInfoField()
{
    GDALDatasetH inputDS;               //input dataset
    GDALDriverH inputDriver;       //driver of the input file
    double geotransform[6];             //geotransform of the input file


    GDALAllRegister();                  //register all gdal formats

      //open rasterfile with read-only access
      inputDS = GDALOpen( leInputFile->text().toUtf8(), GA_ReadOnly );

    inputDriver = GDALGetDatasetDriver( inputDS );

      //information string formatted by html
    QString information = "<html><body>";
    information += "<table width=\"100%\">";

    //select drivername and append to string
    information += "<tr><td>";
    information += tr( "<b>Driver:</b> %1/%2" )
                    .arg( GDALGetDriverShortName( inputDriver ) )
                    .arg( GDALGetDriverLongName( inputDriver ) );
    information += "</td></tr>";
```

```
    //select rastersize and append to string
    information += "<tr><td>";
    information += tr( "<b>Size:</b> %1 x %2" )
                    .arg( GDALGetRasterXSize( inputDS ) )
                    .arg( GDALGetRasterYSize( inputDS ) );
    information += "</td></tr>";

    //Coordinate System
    if( GDALGetProjectionRef( inputDS ) != NULL ){
        OGRSpatialReferenceH inputSRS;
            //spatial reference of the input file
        char *projection;
                  //projection string in proj4 representation

        projection = (char *) GDALGetProjectionRef( inputDS );

        inputSRS = OSRNewSpatialReference(NULL);
      //create new SRS
        if( OSRImportFromWkt( inputSRS, &projection ) == CE_None ){
      //import projection settings to SRS
            char *prettyWkt = NULL;

            OSRExportToPrettyWkt( inputSRS, &prettyWkt, FALSE ); //export
SRS to pretty WKT

            QString wktFormat = prettyWkt;
            wktFormat.replace(QString("\n"), QString("<br>"));
      //replace newlines

                  //add wkt to information string
            information += "<tr><td>";
            information += tr( "<b>Coordinate System:</b>" );
                  information += "</td></tr>";
                  information += "<tr><td style='padding-left: 15px;'>";
            information += wktFormat;
            information += "</td></tr>";
        }else{
            information += "<tr><td>";
            information += tr( "<b>Coordinate System:</b><br> %1" )
                            .arg( GDALGetProjectionRef( inputDS ) );
            information += "</td></tr>";
        }

        OSRDestroySpatialReference( inputSRS );
    }

    //Geotransform
    if( GDALGetGeoTransform( inputDS, geotransform ) == CE_None ){
        if( geotransform[2] == 0.0 && geotransform[4] == 0.0 ){

            information += "<tr><td>";
            information += tr( "<b>Origin:</b> (%1,%2)" )
                            .arg( geotransform[0], 0, 'f', 15 )
                            .arg( geotransform[3], 0, 'f', 15 );
            information += "</td></tr>";

                  //add pixel size to information string
            information += "<tr><td>";
            information += tr( "<b>Pixel Size:</b> (%1,%2)" )
                            .arg( geotransform[1], 0, 'f', 15 )
                            .arg( geotransform[5], 0, 'f', 15 );
            information += "</td></tr>";
```

```
        }else{
                  //add geotransform to information string
            information += "<tr><td>";
            information += tr( "<b>GeoTransform:</b><br> %1, %2, %3<br> %4,
%5, %6" )
                     .arg( geotransform[0], 0, 'g', 16 )
                     .arg( geotransform[1], 0, 'g', 16 )
                     .arg( geotransform[2], 0, 'g', 16 )
                     .arg( geotransform[3], 0, 'g', 16 )
                     .arg( geotransform[4], 0, 'g', 16 )
                     .arg( geotransform[5], 0, 'g', 16 );
            information += "</td></tr>";
        }
    }

      //Calculate corners and add them to information string
      information += "<tr><td><b>Corner Coordinates:</b></td></tr>";
      information += tr( "<tr><td style='padding-left: 15px;'><b>Upper
Left:</b> (%1, %2)<br>")
                  .arg( geotransform[0], 0, 'g', 16 )
                  .arg( geotransform[3], 0, 'g', 16 );
      information += tr( "<b>Lower Left:</b> (%1, %2)<br>")
                  .arg( geotransform[0], 0, 'g', 16 )
                  .arg( (GDALGetRasterYSize( inputDS ) *
geotransform[5])+geotransform[3], 0, 'g', 16 );
      information += tr( "<b>Upper Right:</b> (%1, %2)<br>")
                  .arg( (GDALGetRasterXSize( inputDS ) *
geotransform[1])+geotransform[0], 0, 'g', 16 )
                  .arg( geotransform[3], 0, 'g', 16 );
      information += tr( "<b>Lower Left:</b> (%1, %2)<br>")
                  .arg( (GDALGetRasterXSize( inputDS ) *
geotransform[1])+geotransform[0], 0, 'g', 16 )
                  .arg( (GDALGetRasterYSize( inputDS ) *
geotransform[5])+geotransform[3], 0, 'g', 16 );
      information += tr( "<b>Center:</b> (%1, %2)")
                  .arg( ((GDALGetRasterXSize( inputDS ) *
geotransform[1])/2)+geotransform[0], 0, 'g', 16 )
                  .arg( ((GDALGetRasterYSize( inputDS ) *
geotransform[5])/2)+geotransform[3], 0, 'g', 16 );
      information += "</td></tr>";

    information += "</table></body></html>";

    teInformation->clear();
    teInformation->setHtml( information );     //fill informatione field
with html content

      GDALClose( inputDS );                          //close dataset
      GDALDestroyDriverManager();
}
```

Quellcode der Datei *plugin.h*:

```
#ifndef GDALPLUGIN_H
#define GDALPLUGIN_H

//QT4 includes
#include <QObject>

//QGIS includes
#include "../qgisplugin.h"
```

```
//forward declarations
class QAction;
class QToolBar;

class QgisInterface;

/**
* \class Plugin
* \brief [name] plugin for QGIS
* [description]
*/
class gdalPlugin: public QObject, public QgisPlugin
{
    Q_OBJECT
  public:

    //////////////////////////////////////////////////////////////////////
    //
    //                MANDATORY PLUGIN METHODS FOLLOW
    //
    //////////////////////////////////////////////////////////////////////

    /**
    * Constructor for a plugin. The QgisInterface pointer is passed by
    * QGIS when it attempts to instantiate the plugin.
    * @param theInterface Pointer to the QgisInterface object.
     */
    gdalPlugin( QgisInterface * theInterface );
    //! Destructor
    virtual ~gdalPlugin();

  public slots:
    //! init the gui
    virtual void initGui();
    //! Show the dialog box
    void run();
    //! unload the plugin
    void unload();
    //! show the help document
    void help();

  private:

    //////////////////////////////////////////////////////////////////////
    //
    // MANDATORY PLUGIN PROPERTY DECLARATIONS  .....
    //
    //////////////////////////////////////////////////////////////////////

    int mPluginType;
    //! Pointer to the QGIS interface object
    QgisInterface *mQGisIface;
    //!pointer to the qaction for this plugin
    QAction * mQActionPointer;
    //////////////////////////////////////////////////////////////////////
    //
    // ADD YOUR OWN PROPERTY DECLARATIONS AFTER THIS POINT.....
    //
    //////////////////////////////////////////////////////////////////////
};

#endif //test_H
```

Quellcode der Datei *plugin.cpp*:

```
/*  $Id: plugin.cpp 9327 2008-09-14 11:18:44Z jef $ */

//
// QGIS Specific includes
//

#include <qgisinterface.h>
#include <qgisgui.h>

#include <plugin.h>
#include <gdalgui.h>

//
// Qt4 Related Includes
//

#include <QAction>
#include <QToolBar>


static const char * const sIdent = "$Id: plugin.cpp 9327 2008-09-14
11:18:44Z jef $";
static const QString sName = QObject::tr( "GDAL Plugin" );
static const QString sDescription = QObject::tr( "GDAL Plugin" );
static const QString sPluginVersion = QObject::tr( "Version 0.01" );
static const QgisPlugin::PLUGINTYPE sPluginType = QgisPlugin::UI;

//////////////////////////////////////////////////////////////////////
//
// THE FOLLOWING METHODS ARE MANDATORY FOR ALL PLUGINS
//
//////////////////////////////////////////////////////////////////////

/**
 * Constructor for the plugin. The plugin is passed a pointer
 * an interface object that provides access to exposed functions in QGIS.
 * @param theQGisInterface - Pointer to the QGIS interface object
 */
gdalPlugin::gdalPlugin( QgisInterface * theQgisInterface ):
    QgisPlugin( sName, sDescription, sPluginVersion, sPluginType ),
    mQGisIface( theQgisInterface )
{
}

gdalPlugin::~gdalPlugin()
{

}

/*
 * Initialize the GUI interface for the plugin - this is only called once
when the plugin is
 * added to the plugin registry in the QGIS application.
 */
void gdalPlugin::initGui()
{

  // Create the action for tool
  mQActionPointer = new QAction( QIcon( ":/gdal.png" ), tr( "GDAL" ), this
);
  // Set the what's this text
```

```
  mQActionPointer->setWhatsThis( tr( "GDAL Functions" ) );
  // Connect the action to the run
  connect( mQActionPointer, SIGNAL( triggered() ), this, SLOT( run() ) );
  // Add the icon to the toolbar
  mQGisIface->addToolBarIcon( mQActionPointer );
  mQGisIface->addPluginToMenu( tr( "&GDAL" ), mQActionPointer );

}
//method defined in interface
void gdalPlugin::help()
{
  //implement me!
}

// Slot called when the menu item is triggered
// If you created more menu items / toolbar buttons in initiGui, you should
// create a separate handler for each action - this single run() method
will
// not be enough
void gdalPlugin::run()
{
  gdalGui *myPluginGui = new gdalGui( mQGisIface, mQGisIface->mainWindow(),
QgisGui::ModalDialogFlags );
  myPluginGui->setAttribute( Qt::WA_DeleteOnClose );

  myPluginGui->show();
}

// Unload the plugin by cleaning up the GUI
void gdalPlugin::unload()
{
  // remove the GUI
  mQGisIface->removePluginMenu( "&GDAL", mQActionPointer );
  mQGisIface->removeToolBarIcon( mQActionPointer );
  delete mQActionPointer;
}



////////////////////////////////////////////////////////////////////////
//
//
//  THE FOLLOWING CODE IS AUTOGENERATED BY THE PLUGIN BUILDER SCRIPT
//    YOU WOULD NORMALLY NOT NEED TO MODIFY THIS, AND YOUR PLUGIN
//      MAY NOT WORK PROPERLY IF YOU MODIFY THIS INCORRECTLY
//
//
////////////////////////////////////////////////////////////////////////


/**
 * Required extern functions needed  for every plugin
 * These functions can be called prior to creating an instance
 * of the plugin class
 */
// Class factory to return a new instance of the plugin class
QGISEXTERN QgisPlugin * classFactory( QgisInterface *
theQgisInterfacePointer )
{
  return new gdalPlugin( theQgisInterfacePointer );
}
// Return the name of the plugin - note that we do not user class members
as
```

```
// the class may not yet be insantiated when this method is called.
QGISEXTERN QString name()
{
  return sName;
}

// Return the description
QGISEXTERN QString description()
{
  return sDescription;
}

// Return the type (either UI or MapLayer plugin)
QGISEXTERN int type()
{
  return sPluginType;
}

// Return the version number for the plugin
QGISEXTERN QString version()
{
  return sPluginVersion;
}

// Delete ourself
QGISEXTERN void unload( QgisPlugin * thePluginPointer )
{
  delete thePluginPointer;
}
```

Quellcode der XML-Datei *gdalguibase.ui*:

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>gdalGuiBase</class>
 <widget class="QDialog" name="gdalGuiBase">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>620</width>
    <height>564</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>GDAL</string>
  </property>
  <property name="windowIcon">
   <iconset>
    <normaloff/>
   </iconset>
  </property>
  <layout class="QGridLayout">
   <property name="margin">
    <number>9</number>
   </property>
   <property name="spacing">
    <number>6</number>
   </property>
   <item row="0" column="0">
    <widget class="QGroupBox" name="groupBox">
     <property name="minimumSize">
```

```
 <size>
  <width>0</width>
  <height>60</height>
 </size>
</property>
<property name="maximumSize">
 <size>
  <width>16777215</width>
  <height>60</height>
 </size>
</property>
<property name="font">
 <font>
  <weight>75</weight>
  <bold>true</bold>
 </font>
</property>
<property name="autoFillBackground">
 <bool>false</bool>
</property>
<property name="title">
 <string>Select Input File</string>
</property>
<property name="flat">
 <bool>false</bool>
</property>
<property name="checkable">
 <bool>false</bool>
</property>
<widget class="QLineEdit" name="leInputFile">
 <property name="geometry">
  <rect>
   <x>106</x>
   <y>24</y>
   <width>451</width>
   <height>20</height>
  </rect>
 </property>
</widget>
<widget class="QLabel" name="label">
 <property name="geometry">
  <rect>
   <x>45</x>
   <y>26</y>
   <width>91</width>
   <height>16</height>
  </rect>
 </property>
 <property name="text">
  <string>GDAL File:</string>
 </property>
</widget>
<widget class="QToolButton" name="btnInputFileLoad">
 <property name="geometry">
  <rect>
   <x>561</x>
   <y>24</y>
   <width>25</width>
   <height>20</height>
  </rect>
 </property>
 <property name="text">
  <string>...</string>
```

```
    </property>
   </widget>
  </widget>
 </item>
 <item row="1" column="0">
  <widget class="QTabWidget" name="tabWidget">
   <property name="enabled">
    <bool>true</bool>
   </property>
   <property name="currentIndex">
    <number>1</number>
   </property>
   <widget class="QWidget" name="tabInfo">
    <property name="layoutDirection">
     <enum>Qt::LeftToRight</enum>
    </property>
    <attribute name="title">
     <string>Info</string>
    </attribute>
    <widget class="QTextEdit" name="teInformation">
     <property name="geometry">
      <rect>
       <x>10</x>
       <y>10</y>
       <width>581</width>
       <height>441</height>
      </rect>
     </property>
    </widget>
   </widget>
   <widget class="QWidget" name="tabReproject">
    <attribute name="title">
     <string>Reproject</string>
    </attribute>
    <widget class="QGroupBox" name="groupBox_2">
     <property name="geometry">
      <rect>
       <x>10</x>
       <y>7</y>
       <width>581</width>
       <height>441</height>
      </rect>
     </property>
     <property name="title">
      <string>Output Parameters</string>
     </property>
     <widget class="QLabel" name="label_2">
      <property name="geometry">
       <rect>
        <x>25</x>
        <y>28</y>
        <width>71</width>
        <height>16</height>
       </rect>
      </property>
      <property name="text">
       <string>Filename:</string>
      </property>
     </widget>
     <widget class="QLineEdit" name="leOutputFile">
      <property name="geometry">
       <rect>
        <x>105</x>
```

```
          <y>26</y>
          <width>433</width>
          <height>20</height>
         </rect>
        </property>
       </widget>
       <widget class="QToolButton" name="btnOutputFileLoad">
        <property name="geometry">
         <rect>
          <x>541</x>
          <y>26</y>
          <width>25</width>
          <height>20</height>
         </rect>
        </property>
        <property name="text">
         <string>...</string>
        </property>
       </widget>
       <widget class="QLabel" name="label_3">
        <property name="geometry">
         <rect>
          <x>25</x>
          <y>60</y>
          <width>61</width>
          <height>16</height>
         </rect>
        </property>
        <property name="text">
         <string>Projection:</string>
        </property>
       </widget>
       <widget class="QgsProjectionSelector" name="projectSelector"
native="true">
        <property name="geometry">
         <rect>
          <x>105</x>
          <y>59</y>
          <width>461</width>
          <height>341</height>
         </rect>
        </property>
       </widget>
       <widget class="QPushButton" name="btnReproject">
        <property name="geometry">
         <rect>
          <x>372</x>
          <y>410</y>
          <width>75</width>
          <height>24</height>
         </rect>
        </property>
        <property name="font">
         <font>
          <weight>75</weight>
          <bold>true</bold>
         </font>
        </property>
        <property name="text">
         <string>Reproject</string>
        </property>
       </widget>
       <widget class="QPushButton" name="btnReprojectAdd">
```

```
        <property name="geometry">
         <rect>
          <x>456</x>
          <y>410</y>
          <width>111</width>
          <height>24</height>
         </rect>
        </property>
        <property name="font">
         <font>
          <weight>75</weight>
          <bold>true</bold>
         </font>
        </property>
        <property name="text">
         <string>Reproject &amp;amp;amp; Add</string>
        </property>
       </widget>
       <widget class="QLabel" name="laOutput">
        <property name="geometry">
         <rect>
          <x>106</x>
          <y>415</y>
          <width>130</width>
          <height>14</height>
         </rect>
        </property>
        <property name="font">
         <font>
          <weight>75</weight>
          <bold>true</bold>
         </font>
        </property>
        <property name="text">
         <string/>
        </property>
       </widget>
      </widget>
     </widget>
    </widget>
   </item>
  </layout>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <customwidgets>
  <customwidget>
   <class>QgsProjectionSelector</class>
   <extends>QWidget</extends>
   <header>qgsgenericprojectionselector.h</header>
   <container>1</container>
  </customwidget>
 </customwidgets>
 <resources/>
 <connections/>
</ui>
```

A.2 CD-Rom mit gesamtem Quellcode

Zusammenfassung

Diese Bachelorarbeit beschreibt die Konzeption und Entwicklung eines Plugins zur Rasterdatenanalyse und –reprojektion für Quantum GIS 1.2.0 unter Verwendung der Geospatial Data Abstraction Library (GDAL). Dazu werden zunächst die Grundlagen über Quantum GIS, GDAL, Koordinatensysteme und Koordinatentransformationen erläutert. Zu Beginn des praktischen Teils der Arbeit wird die Einrichtung der Entwicklungsumgebung für Windows XP Professional und die allgemeine Struktur des Quellcodes von Quantum GIS Plugins beschrieben. Im Folgenden werden der Aufbau der graphischen Benutzeroberfläche und anschließend die zwei Hauptfunktionen des Plugins behandelt. Diese Funktionen werden anhand von Quellcode-Auszügen Schritt für Schritt beschrieben. In der abschließenden Zusammenfassung wird der Mehrwert des Plugins, durch ein aufgeführtes Beispiel, verdeutlicht und mögliche Ansätze zur Verbesserung und Erweiterung des Plugins genannt.

Abstract

This bachelorthesis describes the conception and the development of a plugin to analyse and reproject rasterdata for Quantum GIS 1.2.0 using the Geospatial Data Abstraction Library (GDAL). In the first instance the fundamentals of Quantum GIS, GDAL, coordinate systems and coordinate transformation are explained. At the beginning of the practical part of the thesis the setup of the development environment for Windows XP Professional and the general structure of the source code of Quantum GIS plugins are described. The following part deals with the layout of the graphical user interface and the two main functions of the plugin. These functions are described with source code extracts step by step. In the concluding summary the additional benefit of the plugin are pointed up by an example and proposals to improve and enlarge the plugin are mentioned.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Osnabrück, den 09.10.2009 _____

Florian Hillen