

Projektarbeit

RA₂DIO

PA_01_0506

Studierende: Niklaus Jäggi
Vicki Schmid

Betreuer: Dr. Christoph Stamm

05. Juli 2006

Abstract

Bei RA₂DIO handelt es sich um ein Programm, in welchem Wellenausbreitungen von Mobilfunkantennen berechnet und in parallelperspektivischer Ansicht zweidimensional dargestellt werden können.

Als Bildformat für die gegebenen, geografischen Daten wird PGF für Externspeicher (PGFExt) verwendet. PGFExt erlaubt eine schnelle Kompression und Dekompression von zweidimensionalen Bildern. Dabei können Teilbereiche eines Bildes (Tiles) unabhängig voneinander gelesen und geschrieben werden.

Im Rahmen der Projektarbeit wurde ein Prototyp für RA₂DIO erstellt, in welchem Texturdaten aus PGFExt-Dateien ausgelesen und in einem Viewer dargestellt werden können.

Auftraggeber der Arbeit war die Firma xeraina GmbH in Zürich, welche bereits ein Wellenausbreitungsprogramm in 3D, RA₃DIO, entwickelt hat und vertreibt.

Management Summary RA₂DIO

Aufgabenstellung

RA₂DIO soll in Zukunft –wie schon das Programm RA₃DIO in 3D- Wellenausbreitungen von Mobilfunkantennen berechnen und darstellen können, allerdings zweidimensional und perspektivisch.

Der Benutzer soll fließend über das Gelände navigieren können.

Die Geländedaten werden mittels PGF für Externspeicher (PGFExt) verwaltet.

Im Rahmen der Projektarbeit soll ein Prototyp für die Applikation erstellt werden, auf welchem für die weitere Entwicklung aufgebaut werden kann.

In diesem Prototypen sollen Texturdaten angezeigt werden, wobei ein fließendes Zoomen und Scrollen möglich sein soll.

Durchführung

Nach einer Einarbeitungszeit in die bestehenden Module von RA₃DIO und ins Bildformat PGF für Externspeicher, erstellten wir ein Konzept für die Verwaltung der Geländedaten, sowie für deren Darstellung.

Nach Erstellung der Konzepte wurden diese, soweit es ging umgesetzt. Dabei wurde eine Datenstruktur (das Terrain) erstellt, in die die Geländedaten (für unseren Prototypen waren dies Textur- und Höhendaten) reingeladen und für die Anzeige in der Viewer-GUI aufbereitet werden. Ebenso wurde ein Prototyp für die GUI erstellt.

Grenzen und Ausblick

Im Rahmen der Projektarbeit konnten nur Teile des Konzepts umgesetzt werden.

Momentan können Höhen- und Textur-Daten geladen und angezeigt werden.

Zukünftig soll die Darstellung weiterer Geländedaten (Landnutzungsdaten, Basisstationen), sowie die Berechnung und Darstellung von Wellenausbreitungsdaten der Mobilfunkantennen implementiert werden.

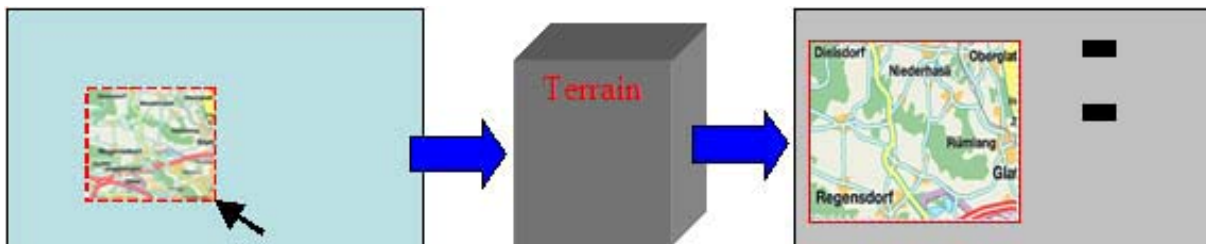
Die geladenen PGFExt- Daten werden im Prototypen noch ganz dekomprimiert. Hier soll in einem weiteren Schritt die Progressivität von PGF bzw. PGFExt ausgenutzt und die Daten nur soweit dekomprimiert werden, wie dies für die Darstellung erforderlich ist.

Resultate

Der aus der Projektarbeit resultierende Prototyp kann Textur- und Höhendaten laden und in einem einfachen Viewer darstellen.

Das Terrain dient hierzu als Zwischenspeicher für die geladenen Geländedaten, vor deren Darstellung im Viewer.

Die Terrain-Struktur, sowie die zugehörigen Klassen wurden modular programmiert und können problemlos für weitere Quelldaten und um zusätzliche Funktionen erweitert werden.



Inhaltsverzeichnis

Abstract	2
1 Abkürzungen und Begriffe	5
2 Aufgabenstellung	6
2.1 Einleitung	6
2.2 Ziel	6
2.3 Einzelaufgaben/ Aufgabenteilung	6
3 Grundlagen	7
3.1 RA ₃ DIO	7
3.2 Geländedaten	7
3.2.1 Quelldaten	7
3.2.2 Basisstationen	8
3.2.3 Koordinatensysteme	8
3.3 PGF	9
3.4 PGF für Externspeicher (PGFExt)	9
4 Gesamtkonzept	11
5 Realisierung	13
5.1 Einlesen und organisieren der Daten	13
5.1.1 Organisation der Daten in Terrain und Tiles	13
5.1.2 Festhalten von weiteren notwendigen Informationen	17
5.1.3 Laden der PGFExtDateien in Terrain/Tile	19
5.1.4 CCoord	19
5.2 Initialisierung und Serialisierung mittels XML	20
5.2.1 Initialisierung mit XML	20
5.3 GUI	20
5.3.1 Aufbau und Struktur	20
5.3.2 Die GUI Klasse	21
5.3.3 Darstellung des Bildes	22
5.4 Serialisierung	22
5.4.1 Allgemeines	23
5.4.2 Die Klasse CRa2dioDataIO	23
6 Anforderungen für weitere Entwicklung	25
6.1 Dekompression der Tiles	25
6.2 Navigation in Terrain	25
6.3 Wellenausbreitungen	25
6.4 LEDA ersetzen mit STL	25
6.5 Realisierung mit geografischen Koordinaten	25
6.6 Weitere Lade-Funktionen im Terrain	25
6.7 .net Framework	25
6.7.1 Andere Syntax	26
6.7.2 Probleme	26
7 Persönliche Bemerkungen	27
7.1 Persönliche Bemerkungen Niklaus Jäggi	27
7.2 Persönliche Bemerkungen Vicki Schmid	28
8 Anhang	30
8.1 Programmier-Richtlinien	30
8.1.1 Formatierung	30
8.1.2 Doxygengerechte Kommentare	30
8.1.3 Benennung von Datentypen	30
8.2 Autoren der Dokumentation	30
8.3 Quellenangaben, Literaturverzeichnis	31

1 Abkürzungen und Begriffe

PGF	Progressive Graphics Format
PGFExt	PGF für Externspeicher
DEM	Digital Elevation Model
GIS	Grafisches Informations-System
STL	Standard Template Library
XML	Extensible Markup Language

2 Aufgabenstellung

2.1 Einleitung

In geographischen Informationssystemen (GIS) werden sehr grosse Mengen an Geländedaten verarbeitet. Zu den Geländedaten zählen Topographiedaten (Höhendaten), Texturdaten (Luftbilder, Pixelkarten usw.), Landnutzungsdaten und weitere thematische Daten, welche mit geografischen Koordinaten verbunden sind. Diese Geländedaten sollten auf Externspeichern (z.B. Festplatte) verwaltet und nur stückweise bei Bedarf in den Hauptspeicher geladen werden. Dadurch können mit normalgrossem Hauptspeicher riesige Geländeabschnitte prozessiert werden.

RA₂DIO soll in ferner Zukunft wie sein grosser Bruder RA₃DIO [1] die Funkwellenausbreitung in riesigen Geländen simulieren. Nun aber ohne echte 3D-Darstellung, sondern mit normaler, parallelperspektivischer 2D-Ansicht. Dafür aber mit einer ausgeklügelten und effizienten Verwaltung der Geländedaten. Diese Verwaltung soll auf der Technik "PGF für Externspeicher" [3] basieren. Dabei werden Bilddaten in Kacheln aufgeteilt und einzeln mit PGF [2] komprimiert abgelegt. Höhen- und Landnutzungsdaten können als weitere "Bildkanäle" der Texturdaten interpretiert und gemeinsam mit den Bilddaten auf dem Externspeicher abgelegt werden.

2.2 Ziel

Ziel der Projektarbeit war es, einen einfachen Geländeviewer zu implementieren, welcher als Prototyp für die weitere RA₂DIO -Entwicklung verwendet werden kann.

Der Prototyp soll Texturdaten, welche im PGFExt-Format vorliegen, zweidimensional darstellen, wobei der Benutzer fliegend über das ganze Gelände navigieren können soll.

2.3 Einzelaufgaben/ Aufgabenteilung

Die Realisierung der Arbeit wurde in zwei Hauptteile unterteilt:

Der erste Teil umfasste die Datenorganisation und das Laden der PGFExt-Tiles in eine Datenstruktur, welche zu definieren war.

Ebenfalls in diesen Aufgabenteil fiel die Initialisierung dieser Datenstruktur beim Aufstarten des Programms.

Dieser Teil wurde durch Vicki Schmid realisiert.

Der zweite Teil umfasste die Darstellung der Bilddaten, welche sich in der oben genannten Datenstruktur befinden, in einer GUI. Vorgenommene Benutzereinstellungen müssen serialisiert werden und beim erneuten Aufstarten des Programms zur Verfügung stehen. Für diesen Teil war Niklaus Jäggi zuständig.

3 Grundlagen

3.1 RA₃DIO

RA₃DIO simuliert Funkwellenausbreitungen in 3D. Dabei kann der Benutzer selber Antennen durch Reinklicken erstellen oder die Antennen-Daten via Datenbank verwalten.

In RA₃DIO werden die Geländedaten dreidimensional dargestellt. Der Benutzer kann somit aus verschiedenen Perspektiven im Gelände navigieren.

Weiter kann er auswählen, welche Geländedaten (Landnutzung, Höhendaten, Texturdaten, etc) er angezeigt haben möchte.

Er kann mittels Maus oder Eingabefelder im Gelände navigieren.

Die Geländedaten sind in RA₃DIO in einzelnen PGF-Dateien abgespeichert.

3.2 Geländedaten

3.2.1 Quelldaten

Es liegen folgende Geländedaten vor:

- Texturdaten
- Höhendaten
- Landnutzungsdaten
- Wellenausbreitungsdaten
- Basisstationen

Höhendaten werden immer gebraucht, alle anderen Daten sind optional. Optionale Daten können vom Benutzer ein- oder ausgeschaltet werden. Es können mehrere Datensätze gleichzeitig eingeschaltet sein und werden dementsprechend gleichzeitig angezeigt im Terrain-Viewer.

3.2.1.1 Texturdaten

Texturdaten sind Geländedaten, wie man sie von Landeskarten kennt. Normalerweise sind dies Pixelkarten, Luft- oder Satellitenaufnahmen. Sie liegen im PGFExt-Format vor.

Je nach Auflösung sind verschiedene Datensätze vorhanden. Dies muss beim Laden der Daten beachtet werden: wird eine höhere bzw. geringere Auflösung für die Darstellung im Terrain-Explorer des RA₂DIO -Viewers gefordert, muss ab einer gewissen Stufe ein neuer Datensatz gelesen werden.

Innerhalb eines Datensatzes soll die Progressivität von PGFExt genutzt werden, d.h. die Daten sollen nur soweit decodiert werden, wie es für die Anzeige bzw. ein fließendes Zoomen nötig ist.

3.2.1.2 Höhendaten

Höhendaten liegen als Graustufenbild im PGFExt-Format vor.

Ein bestimmter Höhenwert hat dabei einen bestimmten Grauwert. Es ist eine Auflösung von bis zu 31Bit möglich.

Es sind Werte zwischen -10'000 und +10'000 Meter vorhanden, was 20'000 dm entspricht bzw. 18Bit.

Meistens sind Höhenwerte auf den Dezimeter genau erhältlich.

Höhendaten werden meist als schattiertes Relief und/oder mit hypsometrischen Farbtönen dargestellt.

Ist eine Höhenangabe nicht bekannt bzw. wird sie nicht benötigt, so ist sie auf die Konstante „NoHeight“ zu setzen (vergl. Constants.h) und nicht auf 0 (dies würde bedeuten 0m ü.M.).

3.2.1.3 Landnutzungsdaten

Landnutzungsdaten liegen als indiziertes PGFExt-Bild vor.

Jedem Landnutzungstyp (Wasser, Stadt, Land etc.) ist ein Index zugeordnet.

Über dem Bild kann man sich ein Gitter vorstellen, in deren Zellen der vorherrschende Landnutzungstyp festgehalten wird. Möglich ist z.B. eine Auflösung von 100m x 100m. In RA₂DIO sollen 8 Landnutzungstypen unterschieden werden können.

Landnutzungsdaten werden mit nutzungsbezogenen Farben und/oder mit Strukturrastern dargestellt.

Höhendaten und Landnutzungsdaten liegen mit derselben Gitter-Auflösung vor.

3.2.2 Basisstationen

Basisstationen sind als Punktdaten in einer Datenbank gespeichert.

Beim Darstellen von Wellenausbreitungsdaten ist zu beachten, dass auch dann Wellenausbreitungsdaten angezeigt werden müssen, wenn die Basisstation selber nicht im anzuzeigenden Ausschnitt steht.

3.2.2.1 Wellenausbreitungsdaten und Basisstationen

Wellenausbreitungsdaten werden in einem separaten Modul von RA₂DIO berechnet. Sie zeigen die Signalstärke einer Mobilfunkantenne, berechnet an den Gitterpunkten der Höhendaten.

Die Ausbreitungsdaten werden mit Falschfarben dargestellt.

Bei einem Neustart von RA₂DIO müssen die berechneten Wellenausbreitungsdaten wieder geladen werden und verfügbar sein.

3.2.3 Koordinatensysteme

Die Geländedaten können in zwei verschiedenen Koordinatensystemen vorliegen:

- Geografisches Koordinatensystem (lamda= Längengrad, phi= Breitengrad, z-Achse in metrischer Auflösung)
- Metrisches, kartesisches (Landes-)Koordinatensystem.

Beim metrischen Koordinatensystem befindet sich der Ursprung bei Bordeaux in Frankreich. Dies bedeutet, dass die Ausbreitung in x-Richtung von 480 km - 840 km die Ausbreitung in y-Richtung von 72 km - 300 km geht.

Referenzpunkt in der Schweiz ist die Sternwarte in Bern bei 600/200 km.

Man geht für RA₂DIO davon aus, dass alle Daten im selben Koordinatensystem vorliegen.

Die Projektarbeit wurde nur für metrische Koordinaten realisiert. Zum Teil wurde Platz einkalkuliert für die Implementation mit geografischen Koordinaten und im Code ein entsprechender Kommentar gesetzt.

Weitere Informationen zu den Koordinatensystemen als auch zu den Geländedatentypen können der Web-Site des Bundesamtes für Landestopographie entnommen werden. [4]

3.3 PGF

Das Bildformat PGF (Progressive Graphics Format) wurde ursprünglich für RA₃DIO entwickelt und basiert auf einer schnellen diskreten Wavelettransformation.

Es kann als Ersatz für JPEG gesehen werden, bietet aber meist eine schnellere und höhere Kompression als JPEG.

Aufgrund der schnellen Kompression und Dekompression sowie seiner Progressivität, eignet sich PGF besonders gut für grosse Datenmengen, wie sie in geografischen Anwendungen wie RA₃DIO verwendet werden.

Der Nachteil von PGF, gerade im Zusammenhang mit RA₃DIO ist jedoch, dass nicht auf einzelne Bildausschnitte zugegriffen werden kann. Wird ein Ausschnitt aus einer Karte angefordert, muss die ganze Datei geladen, im Hauptspeicher dekomprimiert und dann der entsprechende Bildausschnitt herauskopiert werden, was Zeit und Ressourcen braucht.

Dieses Problem wurde mit der Entwicklung von PGFExt (PGF für Externspeicher) gelöst.

3.4 PGF für Externspeicher (PGFExt)

PGFExt erlaubt den Zugriff auf einzelne Bildausschnitte.

Dazu wird das Bild durch ein fixes Gitter in gleich grosse, rechteckige Teile (Tiles) unterteilt. Diese können unabhängig voneinander mit individueller Tiefe und Qualität komprimiert bzw. dekomprimiert werden. PGFExt-Daten werden in Pixmaps dekomprimiert.

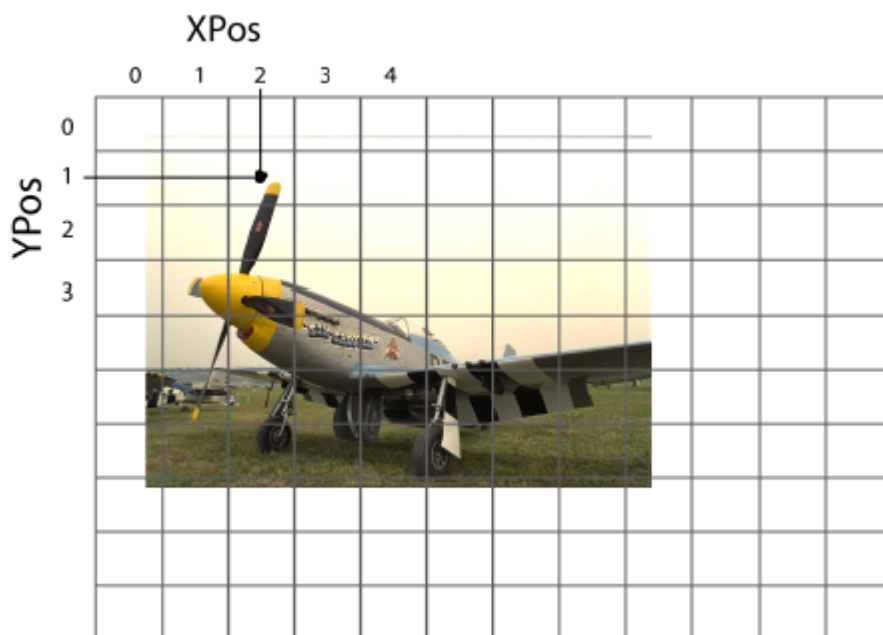


Abb. 1: Aufteilung der PGFExt Datei in Tiles [3]

Ein Tile hat per Standard eine Grösse von 200 x 200 Pixel.

Um die Tiles laden zu können, muss bekannt sein, wo sich im Hauptspeicher die Tiles befinden bzw. wie weit sie komprimiert sind.

Deshalb wird im Hauptspeicher ein Index gehalten, in welchem diese Informationen festgehalten sind.

PGFExt bietet bereits die Schnittstelle um Tiles mit gewünschter Dekomprimierung zu laden.

Weiter stellt PGFExt zwei Caches zur Verfügung:

- Ein Tile-Cache für unkomprimierte Tiles; die Tiles werden verwaltet, der Speicher freigegeben, etc.
- Ein Bildcache für die dekomprimierten Pixmaps. Dieser Bildcache wurde für den RA₂DIO-Prototypen nicht verwendet, sondern durch eine eigene Datenstruktur ersetzt.

4 Gesamtkonzept

Basierend auf den oben genannten Grundlagen, erstellen wir unser Konzept für den RA₂DIO Prototypen.

Der Aufbau unseres RA₂DIO -Prototypen folgt einem objektorientierten Ansatz.

Im Terrain-Explorer wählt der Benutzer entweder per Maus oder per Tastatureingabe einen Ausschnitt an, welcher dargestellt werden soll. Dieser Ausschnitt wird durch eine Auswahl von Tiles im PGFExt abgedeckt. Dabei ist es möglich, dass Tiles vom Benutzer nur teilweise angewählt wurden, jedoch ganz geladen werden müssen.

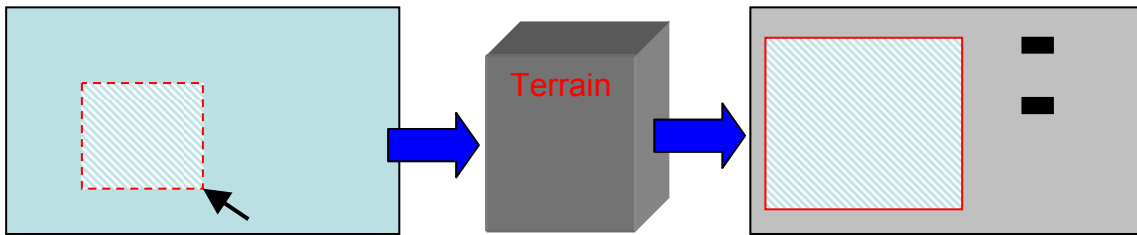


Abb.2: Grobkonzept laden und anzeigen der Geländedaten

Es werden die von der Anfrage betroffenen Tiles so weit für die Anzeige nötig dekomprimiert und geladen. Dabei ist zu beachten, dass die Tiles aus verschiedenen Datenquellen kommen, je nachdem welche Geländedaten der Benutzer sehen will.

Um ein fließendes Scrollen zu ermöglichen, sollen in den Randgebieten vorausschauend mehr Tiles geladen werden, als für die Anzeige nötig sind.

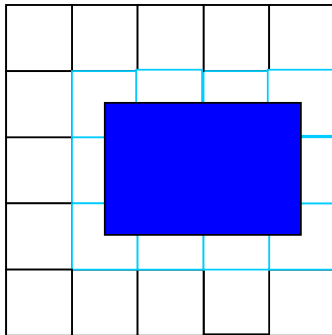


Abb.3: für die Anzeige benötigte Tiles

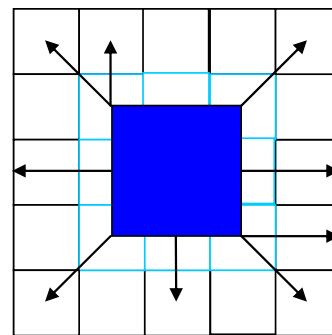


Abb.4: Laden zusätzlicher Tiles für fließendes Scrollen

Ebenso sollen für ein fließendes Zoomen die Tiles etwas weiter als für die Anzeige nötig dekomprimiert werden.

Die vom Externspeicher geladenen, dekomprimierten Tiles werden ins Terrain geladen. Das Terrain ist eine doppelt verkettete, zweidimensionale Liste aus einzelnen Tiles, über welche ein Gitter, die sogenannten Vertices, gelegt werden. Diese Vertices werden mit den Informationen aus den verschiedenen PGFExt-Dateien gefüllt.

Die gefüllten Terrain-Tiles werden schliesslich zu einer Pixmap zusammengefügt und im Terrain-Explorer des Viewers dargestellt.

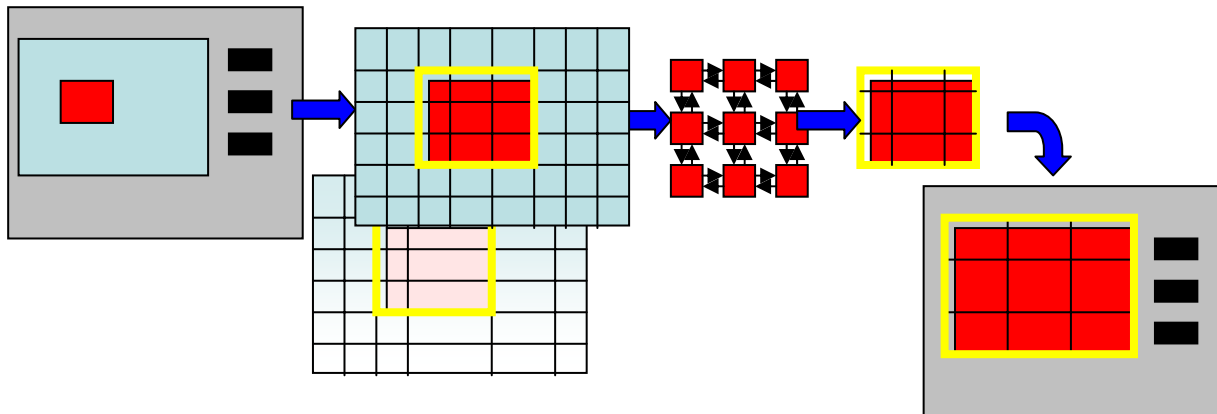


Abb.5: Laden der Tiles in Terrain und Darstellung im Terrain-Viewer

Man hätte anstatt in einer Liste, die Tiles auch in einem ein- oder zweidimensionalen Array speichern können. Die verwendete Listenstruktur erlaubt jedoch eine einfache Verwaltung der Tiles, z.B. können am Rand einfach Tiles angehängt und weggelöscht werden.

5 Realisierung

5.1 Einlesen und organisieren der Daten

5.1.1 Organisation der Daten in Terrain und Tiles

5.1.1.1 Übersicht

Geladene PGFExt-Daten müssen in einen Buffer geladen werden, bevor sie angezeigt werden können. Diese Funktion übernimmt die Klasse Terrain, eine zweidimensionale, doppelt verkettete Liste.

Diese ist wiederum aufgebaut aus einzelnen Tiles, wobei ein PGFExt-Tile einem Terrain-Tile entspricht.

Ein Tile besteht aus einem Array von Vertices (siehe 5.1.4.4 VertexInfo) in welchen die Informationen aus den PGFExt-Dateien gespeichert sind.

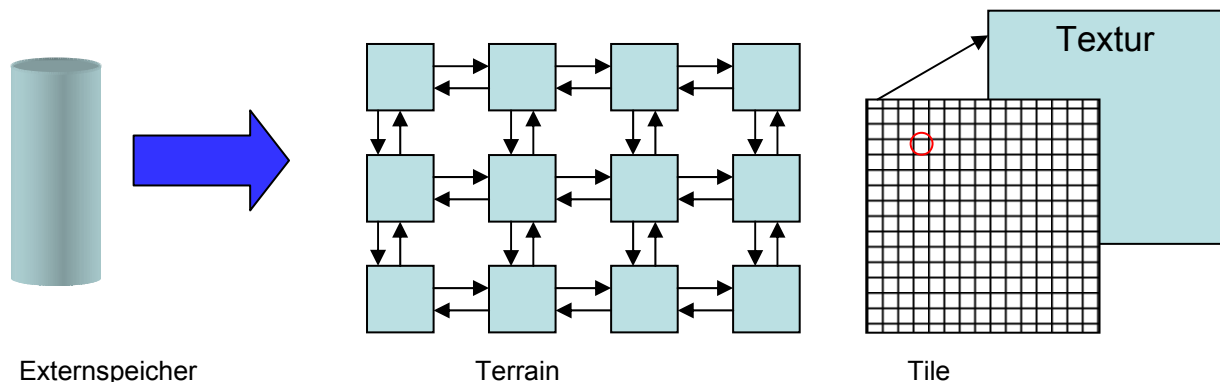


Abb. 6: Gliederung des Zwischenspeichers in Terrain und Tiles

5.1.1.2 Terrain

Terrain bietet die Schnittstelle zwischen dem Bildspeicher und dem Explorer des Viewers. Anfragen des Viewers an den Bildspeicher laufen über Terrain und umgekehrt werden dekomprimierte Tiles ins Terrain geladen, wo der Viewer sie holen und anzeigen kann.

Aufbau

Terrain ist –wie schon erwähnt- eine zweidimensionale, doppelt verkettete Liste, in welcher einzelne (Terrain-) Tiles zusammengehängt sind. Die Liste hat dabei keine Ringstruktur. Ein Tile in RA₂DIO entspricht einem PGFExt-Tile.

Auch wenn Terrain absichtlich so implementiert wurde, dass mehrere Instanzen von Terrain möglich wären, wird in unserem Prototypen nur ein einziges Terrain instanziiert. In die einzelnen Terrain-Tiles, werden die Daten aus den jeweiligen PGFExt-Tiles gefüllt, genauer in deren VertexArray.

Das Terrain ist rechteckig und gerade ausgerichtet. Es besteht immer aus mindestens einem Tile.

Wird ein Terrain erstellt, wird es zuerst aus leeren Tiles aufgebaut, die erst im Nachhinein, mittels Lade-Funktionen, separat gefüllt werden. Theoretisch ist es somit möglich, dass es im

Terrain leere Tiles hat. (Dies muss bei der Anzeige im Terrain-Viewer entsprechend berücksichtigt werden.)

Die Funktionen um das Terrain zu skalieren und verschieben laden nur leere Tiles. Sie müssen separat via die bereitgestellten Methoden `LoadTexture()` bzw. `LoadDEM()` gefüllt werden.

Für unseren Prototypen hat das Terrain die Grösse der im Explorer angezeigten Tiles. Zukünftig sollen, gemäss unserem Konzept, zusätzliche Tiles in Terrain geladen werden, um ein fließendes Scrollen zu ermöglichen.

Navigation

Die Navigation innerhalb des Terrains ist mittels der Methoden `MoveTerrain()` und `ResizeTerrain()` in der Terrain-Klasse implementiert:

`MoveTerrain(TileNum tiles, Direction dir)`

Dieser Funktion werden als Parameter die Anzahl Tiles, um welche das Terrain verschoben wird, sowie die Richtung (Nord, Süd, Ost oder West) angegeben.

Um nicht mehr angezeigte Tiles aus dem Terrain zu entfernen bzw. um für die Anzeige neue Tiles hinzuzuladen, werden die Funktionen `DeleteRows()` / `DeleteColumns()` bzw. `AddRows()` / `AddColumns()` aufgerufen.

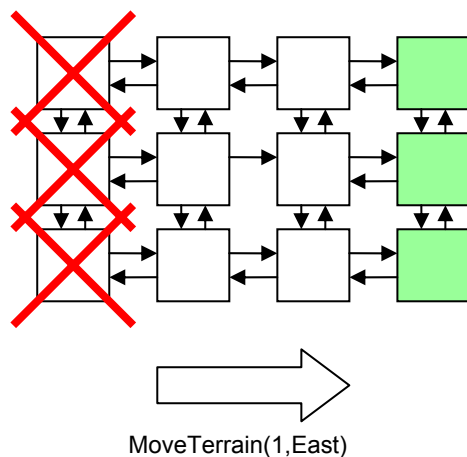


Abb. 7: Beispiel für `MoveTerrain()`

`ResizeTerrain(TileNum width, TileNum height)`

Der `ResizeTerrain()`-Funktion werden als Parameter die Anzahl Tiles in Höhe und Breite, um die das Terrain vergrössert bzw. verkleinert werden soll angegeben.

Auch diese Funktion ruft `DeleteRows()` / `DeleteColumns()` bzw. `AddRows()` / `AddColumns()` auf.

An beiden Seiten werden gleichviele Tiles gelöscht oder hinzugefügt.

Wenn z.B. eine gerade Anzahl Tiles angegeben wird, welche in der Breite angehängt werden soll, so werden rechts und links des Terrains je $\text{AnzTiles}/2$ Tiles angehängt.

Wird für die Höhe und/oder die Breite eine ungerade Anzahl anzuhängender oder zu löschender Tiles angegeben, werden „überflüssige“ Tiles unten bzw. rechts vom Terrain angehängt oder gelöscht. (vergl. Abb.8)

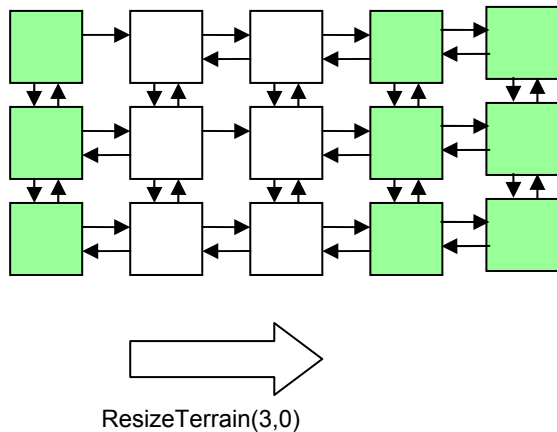


Abb.8: Beispiel für ResizeTerrain()

Laden der Textur- und Höhendaten im Terrain

Textur- und Höhendaten werden mittels der in Terrain implementierten Funktionen `LoadTexture()` bzw. `LoadDEM()` geladen. Diese Funktionen iterieren über alle Tiles im Terrain und rufen für jedes Tile die gleichnamigen Funktionen der Klasse `CTile` auf.

Bilderstellung im Terrain

Terrain/Tiles zu Bitmap

Die Texturdaten des aktuellen Ausschnittes befinden sich in Tiles unterteilt im Terrain-Objekt. Um diese Anzuzeigen müssen sie zusammengelegt und zu einem Bitmap konvertiert werden. Die Daten sind allerdings im `CPixmap` Format, welches bereits eine Möglichkeit enthält, Bilder ineinander zu kopieren.

Um den Pitch und das Bildformat beizubehalten, wird zuerst ein Bitmap erstellt, von welchem alle Einstellungen übernommen werden um ein `CPixmap` zu erstellen. Das `CPixmap` erhält bei der Erstellung einen Pointer auf die Bilddaten vom Bitmap.

```
Bitmap^ CTerrain::getTileMap()
```

Um einen Pointer auf die Bilddaten des Bitmaps zu erhalten, müssen die Daten zuerst mit der Methode `Bitmap->LockBits()` im Speicher angelegt werden. Diese Methode gibt ein `BitmapData` Objekt zurück, von welchem mit `data->Scan0.ToPointer()` einen Pointer auf das erste Bit des Bildes erhältlich ist. Um diesen für ein `CPixmap` brauchbar zu machen, muss er noch mit `(UINT16 *)` konvertiert werden.

Danach werden die einzelnen Texturen aus den Tiles mit `CTile->GetTexture()->CopyTo(Zielbild, RECT)` in das Gesamtbild kopiert werden.

Zum Schluss werden noch die Daten aus dem Speicher mit `Bitmap->UnlockBits(BitmapData);` freigegeben.

5.1.1.3 Tile

Ein Terrain-Tile entspricht einem Tile aus PGFExt.

Es ist aus einem VertexArray (Klasse VertexInfo) aufgebaut, in dessen Vertices die Daten aus der PGFExt-Datei gefüllt werden.

Man kann sich den VertexArray auch als Gitter vorstellen, welches über jedes Tile gelegt wird.

Die Auflösung der Vertices ist dabei variabel: Der Benutzer kann die gewünschte Auflösung spezifizieren. Je nachdem wird nur die Information von z.B. jedem zweiten oder dritten Vertex in diesem Gitter gespeichert und angezeigt.

Da ein Tile ein Bestandteil einer doppelt verketteten Liste ist, muss es seine Nachbarn in alle vier Richtungen kennen.

Terrain hat keine Ringstruktur, Tiles am Rand zeigen auf NULL. So können Tiles am Rand gut erkannt werden.

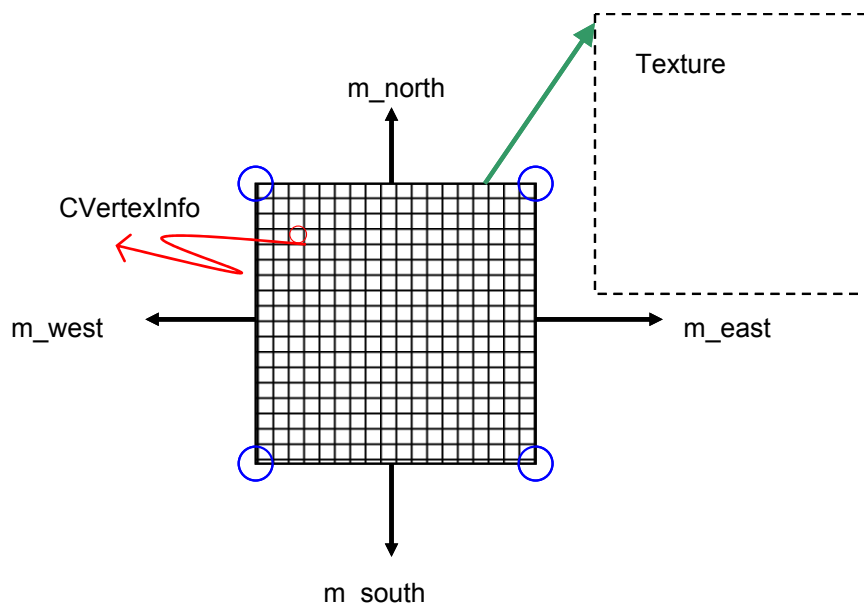


Abb.9: Terrain-Tile

Laden der Textur- und Höhendaten im Tile

Die Funktionen LoadTexture() und LoadDEM() der Klasse CTile laden die Textur- bzw. Höhendante für ein einzelnes Tile.

Sie verwenden beide die Funktion

```
LoadImage(CPixmap*& pixMap, UINT32 width, UNIT32 height, ...)
```

aus PGFExt.

LoadTexture()

Das Laden der Texturdaten ist einfach.

Zuerst wird eine Pixmap der Klasse CPixmap erstellt und dem Texturzeiger des Tiles angehängt.

Anschliessend werden mittels CPGFExt::LoadImage() die Texturdaten in diese Pixmap geladen.

LoadDEM()

Wie schon in 3.2.1.2 erwähnt, liegen Höhendaten in Form eines Grauwertbildes vor, wobei jeder Grauwert für eine Höheninformation steht.

Zuerst wird eine (temporäre) PixMap der Klasse CPixMap erstellt, in welche mittels `CPGExt::LoadImage()` die Höhendaten aus den entsprechenden Tiles in die angelegte PixMap geladen werden.

Nun wird mittels VertexIterator parallel durch die Vertices im Tile und durch die Pixel der PixMap iteriert, wobei die Grauwerte aus der PixMap in die Vertices des Tiles gefüllt werden.

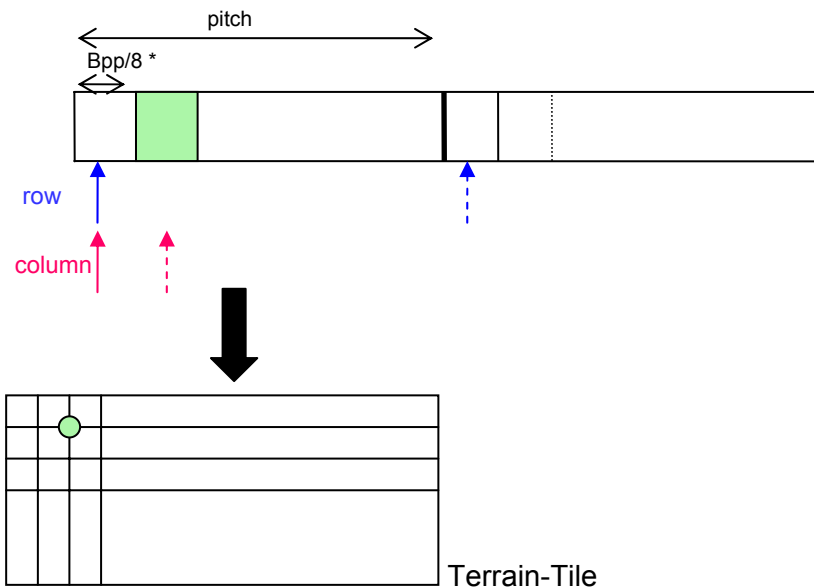


Abb. 10: Auslesen der Grauwerte aus der CPixMap und einfüllen in Vertices der Tiles

5.1.1.4 VertexInfo

In der Klasse VertexInfo sind diverse Informationen zu einem Vertex innerhalb eines Tiles gespeichert:

- Höhe in Meter über Meer
- Landnutzungstyp
- Wellenausbreitungsdaten

Ein Vertex hält somit also Informationen aus verschiedenen PGFExt-Dateien fest.

5.1.1.5 VertexIterator

Die Klasse VertexIterator dient dazu, über die Vertices innerhalb eines Tiles zu iterieren, z.B. um diese mit den Infos aus der PGFExt-Datei zu füllen, Wellenausbreitungen der Antennen zu berechnen, oder um Geländedaten zu visualisieren.

Die Klassen VertexInfo und VertexIterator wurden für unseren Prototypen von RA₃DIO übernommen und wo nötig angepasst.

5.1.2 Festhalten von weiteren notwendigen Informationen

Nebst den Daten, welche aus der PGFExtDatei direkt im Terrain bzw. in den Vertices der einzelnen Tiles gespeichert werden, gibt es eine Reihe von Informationen, welche ebenfalls

für die Anzeige benötigt werden und in der Datenstruktur festgehalten werden müssen, z.B. Bildgrösse, Koordinaten der Geländedaten, Tilegrösse etc.

Dabei wird zwischen zwei Sorten von Informationen unterschieden:

- Daten, welche für das ganze Terrain gültig sind, und welche über alle PGFExt-Dateien angewendet werden können. Diese werden im PGEDataSet festgehalten werden.
- Daten, welche nur für eine einzelne PGFExt-Datei gültig sind. Diese werden in den PGEPackages gespeichert.

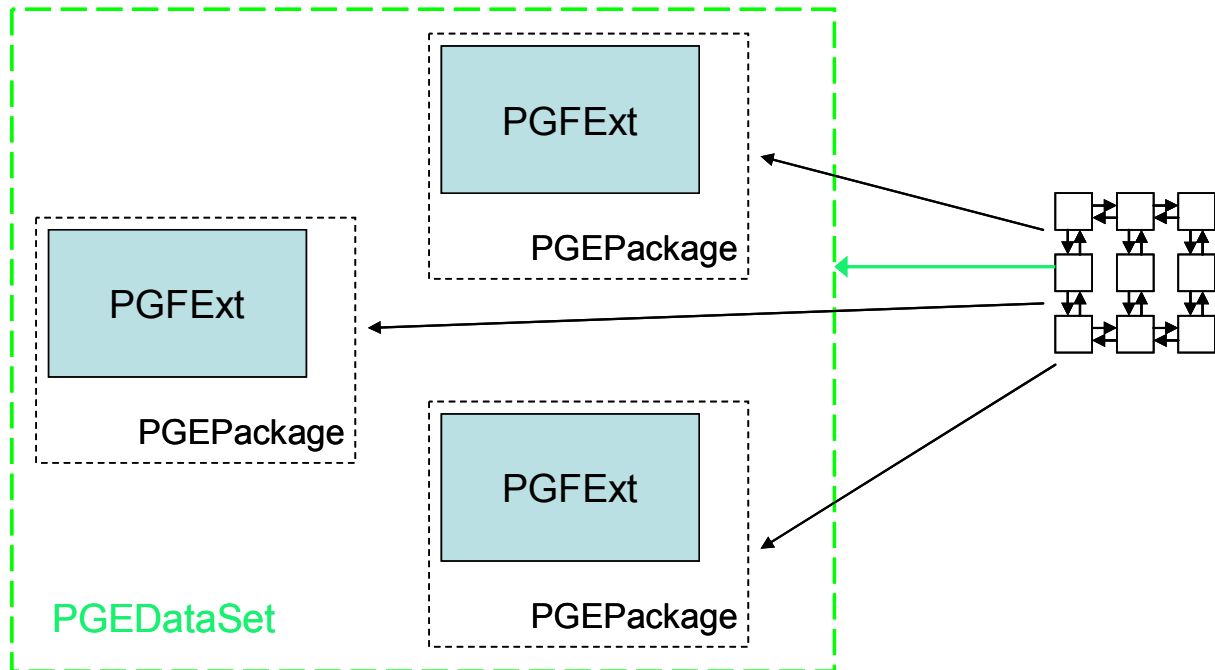


Abb.11: Organisation PGEDataSet und PGEPackage

5.1.2.1 PGEDataSet

In der Klasse PGEDataSet werden Informationen gehalten, welche für alle PGFExt-Dateien gültig sind.

- Höhe und Breite der PGFExt-Dateien in Pixel
- Höhe und Breite eines einzelnen PGFExt-Tiles in Pixel
- Koordinatensystem (ob geografische oder metrische Koordinaten) (es wird davon ausgegangen, dass alle PGFExt-Dateien im selben Koordinatensystem vorliegen, wobei wir in unserem Prototypen in metrischen Koordination arbeiteten)

Sowie:

- Ein Array mit Zeigern auf die jeweiligen PGEPackages (siehe unten)
- Index, welches PGEPackage aus dem jeweiligen Array gerade aktiv ist .)
- Anzahl PGFExts, die pro PGEPackage vorhanden sind

Da nur ein einziges PGEDataSet instanziiert werden soll, wurde die Klasse statisch programmiert.

*) Es wurden pro Datenquelle 4 PGE-Dateien bzw. PGEPackages einkalkuliert. Jedoch ist pro Datenquelle nur eines dieser PGEPackages aktiv. Dieses wird verwendet für die Anzeige, Umrechnungsfunktionen usw. Im PGEDataset, ist deshalb zusätzlich der Index des aktiven PGEPackages pro Datenquelle festgehalten

5.1.2.2 PGEPackage

In einem PGEPackage werden Spezifische Informationen festgehalten, welche nur für eine einzelne PGExt-Datei gültig sind. Diese Daten sind vorwiegend in Landeskoordinaten.

- Pointer auf die PGFExt-Datei
- Breite der PGFExt-Datei in Landeskoordinaten.
- Maschenweite in Landeskoordinaten
- Ursprung in Landeskoordinaten, wobei der Ursprung oben links in der PGFExt-Datei angenommen wird.

Sowie:

- Der Pfad der PGFExt-Datei.

Weiter bietet PGEPackage Methoden zur Umrechnung von Pixel in metrische bzw. geografische Koordinaten und umgekehrt an.

5.1.3 Laden der PGFExtDateien in Terrain/Tile

Die Klasse Tile bietet folgende Funktionen an:

- `LoadTexture()`
Lädt die Texturdaten für ein einzelnes Tile und hängt diese via Zeiger an die Tile-Instanz an.
- `LoadDEM()`
Lädt die Höhendaten für ein einzelnes Tile, iteriert über den `VertexArray` des Tiles und füllt die Daten ein.

Beider Funktionen verwenden die Funktion `LoadImage()` von PGFExt.

In der Klasse Terrain sind dieselben Funktionen nochmals implementiert. Dort wird über alle Tiles innerhalb des Terrains iteriert und pro Tile die oben genannten Funktionen aufgerufen.

5.1.4 CCoord

Wie schon erwähnt, können Geländedaten im geografischen oder metrischen Koordinatensystem vorliegen.

Dies wird mittels der Klasse `CCoord` umgesetzt.

`CCoord` ist die Basisklasse für das Koordinatensystem, wobei die Klassen `CSwissCoord` (metrische Koordinaten) und `CGeoDecCoord` (geografische Koordinaten) abgeleitet sind.

Die Klasse `CCoord` wurde von RA₃DIO übernommen und wo nötig angepasst.

U.a. wurde `Coordinate` um die Klasse `CDist` erweitert:

`CDist` ist die Basisklasse für `CSwissDist` (Distanzen in metrischen Koordinaten) und für `CGeoDist` (Distanzen in geografischen Koordinaten).

5.2 Initialisierung und Serialisierung mittels XML

In RA₃DIO wurden die Daten mittels Registry-Eintrag bzw. .ini-File initialisiert und serialisiert. (Für die Serialisierung, vergl. Kap.5.4.)

Da RA₂DIO mit VisualStudio2005 entwickelt wurde, wurden hier Anpassungen an die .net-Umgebung gemacht, und statt der .ini-Files XML-Files verwendet.

5.2.1 Initialisierung mit XML

Im XML-File sind Daten gespeichert, welche für die Initialisierung von RA₂DIO benötigt werden, aber nicht aus den PGFExt-Dateien bzw. deren Header gelesen werden können. Dies sind:

- Bildpfad der PGFExt-Datei
- Landeskoordinaten des NordWest- und des SüdOst-Punktes der PGFExt-Datei
- Koordinatensystem (geografisch oder metrisch)

5.2.1.1 TinyXml

Zum parsen des XML-Files wurde TinyXml [5] verwendet.

TinyXml ist ein kostenloser XML-Parser, welcher einfach mittels .cpp und .h-Files ins Projekt eingebunden werden kann.

5.3 GUI

5.3.1 Aufbau und Struktur

Mit dem .net Framework und Visual Studio.net 2005 geht die GUI Entwicklung etwas weg vom Dokument-Viewer Prinzip von MFC. Der ins VS integrierte GUI-Designer trägt seinerseits dazu bei, die Gestaltung eines GUIs relativ einfach zu gestalten.

Der Editor bietet:

- Wysiwyg-Gestaltung der Oberfläche
- Einfaches definieren und ändern von Attributen und Events

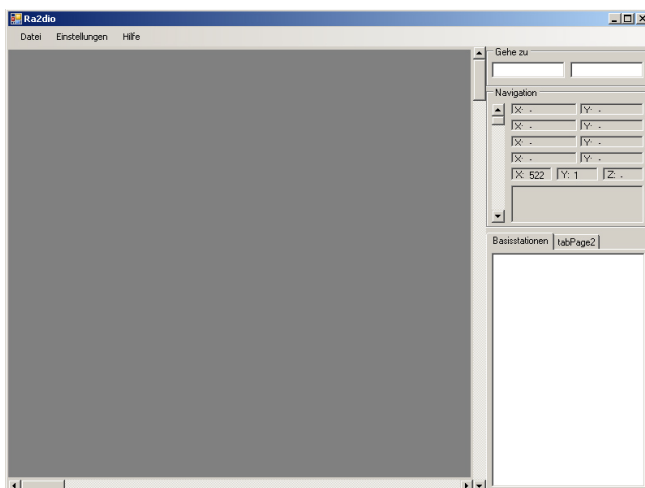


Abb.12: Viewer-GUI von RA₂DIO

Der Viewer von RA₂DIO ist nun in 3 Teile aufgeteilt. Der Erste Teil ist das Menu das sich am oberen Rand befindet.

Der zweite Teil und gleichzeitig der grösste Teil des Viewers ist die Bildfläche zur Ansicht der Karte.

Der Dritte Teil befindet sich am rechten Rand des Viewers. In ihm befinden sich die grundlegenden Informations-Felder.

Zudem enthält dieser Teil im unteren Bereich Sliders bereit, welche für versch. Einstellungen genutzt werden können.

5.3.2 Die GUI Klasse

Diese Klasse steht leider noch unter dem Name Form1 im Projekt. Leider ist es nur umständlich möglich, diesen Namen zu ändern, solange das Projekt in Subversion befindet. Die Klasse wurde im GUI Designer von Visual Studio erstellt, und besitzt einen Codeteil, welcher automatisch generiert wurde.

5.3.2.1 Aufbau

Die Klasse besteht einzig aus dem .h File, in welchem alles deklariert und definiert ist. Zu Beginn steht der Konstruktor, welcher die RA₂DIO-Applikations-Klasse startet. Danach werden die GUI Komponenten initialisiert und das Double-Buffering aktiviert. Zum Schluss wird noch ein erstes Bild geladen.

```
public ref class Form1 : public System::Windows::Forms::Form
{
    public:
        Form1(void)
        {
            app = gcnew CRa2dioApp();
            InitializeComponent();
            this->DoubleBuffered = true;
            LoadNewTiles();
        }
}
```

5.3.2.2 Initialisierung

Die Methode `void InitializeComponent(void)` instanziert alle GUI Komponenten. Diese Methode wurde vom GUI-Editor erstellt, und sollte nicht verändert werden, da der nicht generierte Code beim ändern des GUIs in dieser Methode zu unerwarteten Ergebnissen führen oder auch einfach verschwinden kann. Weitere Initialisierungen die vom Programmierer erwünscht sind, sollten somit im Konstruktor gesetzt werden.

5.3.2.3 Event-Methoden

Im .net Framework werden nicht wie in den meisten anderen GUI Frameworks einfach die Event-Handler überschrieben, sondern es werden Methoden geschrieben welche dem entsprechenden Objekt als Handler übergeben werden.

```
System::Void beendenToolStripMenuItem_Click(System::Object^ sender,
                                             System::EventArgs^ e) {
    this->Close();
}
```

Die Methode besitzt 2 Parameter, welche das Sender Objekt und ein Argument-Objekt repräsentieren.

Im Gegensatz zu überladenen Event-Methoden ist hier der Name der Methode unwichtig da er im Initialisierungscode mit

```
this->beendenToolStripMenuItem->Click +=  
gcnew System::EventHandler (this, &Form1::beendenToolStripMenuItem_Click);
```

dem Objekt zugewiesen wird. Es wird ein Event-Handler kreiert, welcher den Sender des Events enthält (in diesem Falle `this`), und eine Referenz auf die entsprechende Methode.

Der GUI Designer von Visual Studio erspart dem Programmierer allerdings diese Zuweisung und erstellt automatisch die Event-Methode, welche danach implementiert werden kann.

5.3.3 Darstellung des Bildes

Die GUI Klasse erhält durch die Applikationsklasse einen Pointer auf das Terrain, von welchem sie das Bitmap erhält, welches Terrain aus den einzelnen Tiles zusammenstellt. Um nicht bei jedem kleinen Scrollen neu laden zu müssen, ist das Bitmap um einiges grösser als der Teil der angezeigt wird.

5.3.3.1 Bitmap zur Ansicht

Für die Anzeige wird entsprechend nur ein Ausschnitt benötigt, welcher mit der Methode `Bitmap->Clone(Rect, Format)` aus dem Gesamtbild in ein neues Bitmap herauskopiert wird. Dieses neue Bitmap wird anschliessend dem PictureBox Element übergeben und angezeigt.

5.3.3.2 Bildgrenzen und Scrollen

Beim Scrollen wird, solange sich der angezeigte Ausschnitt nicht über festgelegte Grenzen bewegt, einfach nur ein neuer Ausschnitt aus dem bestehenden Bild kopiert. Sollten der angezeigte Ausschnitt diese Grenzen aber überschreiten, wird ein Befehl an das Terrain geleitet, welcher entsprechend neue Tiles lädt. Die Bildgrenzen werden anschliessend neu gesetzt.

5.3.3.3 Buffering

Eine möglichst kurze Ladezeit des Bildes wird in 3 Stufen erreicht. Die erste Stufe erfolgt bereits mit der Verwendung von PGE, bei welchem nur der benötigte Teil gerade in die Tiles übernommen wird. Die zweite Stufe erfolgt mit dem Umkopieren in ein Gesamtbild, welches nur neu erstellt werden muss, falls auch neue Tiles geladen werden.

Stufe drei ist das normale Double-Buffering welches von Windows Forms bereitgestellt wird. Das Double Buffering wird einfach im Konstruktor der Form mit

```
this->DoubleBuffered = true;
```

eingeschaltet. Ein weiteres manuelles Eingreifen ist dadurch überflüssig.

5.4 Serialisierung

Die Serialisierung wird für das Speichern der aktuellen Einstellungen für ein bestimmtes Set von PGE Daten benutzt. Serialisiert wird zudem mit managed Klassen im .net Framework.

5.4.1 Allgemeines

`ref class CRa2dioDataIO` Ist die zentrale Klasse hinter dem Speichern. Sie erstellt ein `ref class CRadioDokument` Objekt, in welches alle zu speichernden Daten übertragen werden. Damit diese Klasse serialisiert werden kann, besitzt sie den Tag `[Serializable]` welcher der Klassendefinition voransteht.

Für die Serialisierung werden zudem noch folgende includes und namespaces benötigt.

```
#using <system.dll>
#using <system.messaging.dll>
#using <System.Runtime.Serialization.Formatters.Soap.dll>

using namespace System;
using namespace System::IO;
using namespace System::Runtime::Serialization::Formatters::Soap;
```

Das .net Framework verwendet für die Serialisierung ein eigenes xml Format. Die Einstellungen dafür befinden in der Soap.dll.

Zu beachten:

- Falls ein Objekt mit .net als Managed Code serialisiert wird, dürfen keine unmanaged Objekte darin enthalten sein.
- Ein File wird wie bis anhin als Stream bearbeitet, aber mit Hilfe eines Soap-Formatters konvertiert

5.4.2 Die Klasse `CRa2dioDataIO`

Sie wird im RA₂DIO Programm beim erstellen eines neuen Dokumentes erstellt. Im Konstruktork werden die Klassen, welche die zu speichernden Informationen haben, übergeben.

Die Methoden `ActualizeObject()` und `ActualizeDataSet()` dienen zur Übertragung der Daten vom Speicher-Objekt zu den entsprechenden Laufzeit-Objekten und umgekehrt.

Die Methoden dienen für den Import und Export der Daten. Bei jedem Benutzen kann ein neuer Name für das File angegeben werden. Ist bei der Erstellung des `CRa2dioDataIO` Objektes schon ein Pfad angegeben worden, kann bei diesen Methoden der Pfad weggelassen werden.

5.4.2.1 Sichern

Die Methode erstellt einen Stream, welcher danach mithilfe eines `SoapFormatters` mit den Daten gefüllt wird. Es wird nur gesichert, wenn der Pfad in Ordnung ist und es überhaupt ein zu speicherndes Objekt existiert. Vor dem Sichern sollte immer die Methode `ActualizeObject()` aufgerufen werden, um sicherzustellen, dass die aktuellsten Daten gespeichert werden.

```
m_stream = File::Open(name, FileMode::Create );
m_formatter = gcnew SoapFormatter;
m_formatter->Serialize(m_stream, m_obj);
m_stream->Close();
```

5.4.2.2 Laden

Beim Laden wird ein FileStream erstellt, aus welchem mit Hilfe des SoapFormatters das Sicherungs-Objekt wiederhergestellt wird. Nach dem Laden muss mit der Methode `ActualizeDataSet ()` die Daten im Laufzeitobjekt wiederhergestellt werden.

```
m_stream = File::Open(name, FileMode::Open );  
if(m_stream){  
    m_formatter = gcnew SoapFormatter;  
    m_obj =  
        dynamic_cast<CRadioDokument^>(m_formatter->Deserialize(m_stream));  
    m_stream->Close();  
}
```

Die Methode `ClearObject()` ist ein reset des zu speichernden Objektes. Sie sollte nach dem Speichern angewendet werden um sicher zu gehen, dass nicht fälschlicherweise alte Daten zurück ins Programm übertragen werden.

Wieso so umständlich?

Wieso nicht einfach die Objekte Serialisieren, welche man Speichern will? Dies hat unter anderem den Grund, dass diese Objekte zusätzliche Informationen enthalten die nicht nötig sind. Zudem wird das Serialisieren mithilfe des .net 05 Frameworks realisiert, welches für Objekte mit Managed Code eine sehr simple methode besitzt.

Daher werden die wichtigen Daten in ein Managed Objekt gespeichert. Die Daten müssen leider zusätzlich alle in den Managed Code übersetzt werden, da die Serialisierung leider keine Mischung zulässt.

6 Anforderungen für weitere Entwicklung

Im Rahmen der Konzept-Erstellung wurden einige Anforderungen festgelegt, die wir nicht mehr umsetzen konnten.

Bei der weiteren Entwicklung gilt es, diese zu beachten.

6.1 Dekompression der Tiles

In unserem Prototypen werden geladene Tiles vollständig dekomprimiert (gemäss der PGFExt- Standardeinstellung). Bei der weiteren Entwicklung soll aber die Progressivität von PGFExt (bzw. PGF) ausgenutzt werden, und die Geländedaten nur soweit dekomprimiert, wie dies für die Darstellung bzw. ein fließendes Zoomen notwendig ist.

6.2 Navigation in Terrain

Damit der Benutzer fließend zoomen und scrollen kann, müssen „vorausschauend“ zusätzliche Tiles ins Terrain geladen werden.

Wenn der Benutzer über den Rand des Bildgebietes herausscrollen will, muss dies abgefangen werden, damit er sich nicht plötzlich im „Nichts“ befindet.

6.3 Wellenausbreitungen

Module zur Berechnung und Darstellung von Wellenausbreitungsdaten müssen noch realisiert werden.

Es ist zu beachten, dass die Wellenausbreitungen auch jener Basisstationen angezeigt werden müssen, welche nicht sichtbar sind.

6.4 LEDA ersetzen mit STL

In RA₃DIO wurde die Bibliothek LEDA verwendet, welche aber für RA₂DIO nicht mehr zur Verfügung steht.

Von RA₃DIO übernommene Module, welche Datentypen und Methoden von LEDA verwenden, müssen entsprechend angepasst werden. Dies kann durch den Einsatz von entsprechenden Algorithmen aus der STL geschehen.

6.5 Realisierung mit geografischen Koordinaten

Im Rahmen der Projektarbeit realisierten wir den Code unter der Annahme, dass die Quelldaten in metrischen Koordinaten vorliegen.

Für geografische Koordinaten muss der Code noch geschrieben werden.

Zum Teil wurde dafür im Code Platz einkalkuliert und ein entsprechender Vermerk gemacht.

6.6 Weitere Lade-Funktionen im Terrain

Es wurden für den Prototypen nur die Lade-Funktionen für die Höhendaten und die Texturdaten implementiert.

Ladefunktionen für weitere Quelldaten (Landnutzungsdaten, Basisstationen) müssen noch realisiert werden.

6.7 .net Framework

Mit der .net 2 Umgebung hat Microsoft den c++ Code um Managed Klassen erweitert. Dies bedeutet zwar, dass uns nun ein Garbage Collector zur Verfügung steht, allerdings der Code auch in gewissen Bereichen anders angegangen werden muss.

6.7.1 Andere Syntax

Damit der Compiler des Visual Studios auch versteht dass es sich um Managed Objekte handelt, werden Pointers mit dem ^ Zeichen referenziert, und mit `gcnew` Alloziert. Um eine Klasse allerdings auch unter dieser Notation verwenden zu können, muss die Klasse durch die Bezeichnung `ref class` als Managed Code bezeichnet werden.

6.7.1.1 „Neue“ Klassen

Einige Klassen wurden für das .net Framework überarbeitet und laufen nun innerhalb des System Namespaces. Unter anderem gilt das für die String Klasse. Auch Arrays wurden neu implementiert als Generische Array Klasse welche als Template zur Verfügung steht.

Diese Änderungen führen leider auch zu Formatschwierigkeiten, wenn man das .net Framework mit „normalem“ c++ Code verwendet.

6.7.1.2 Wichtige Namespaces

Alle Klassen unter dem .net Framework laufen unter dem Namespace

```
using namespace System;
```

Alle wichtigen Klassen und Tools sind in diesem Namespace untergebracht. Der Namespace ist dafür weiter unterteilt in Sub-Namespaces. Die wichtigsten wären:

```
using namespace System::IO;
```

Beinhaltet Klassen für In- und Output und Streams

```
using namespace System::Collections;
```

Ist die Sammlung von Datenstrukturen für Objekte wie Queues, Listen und Bäume.

```
using namespace System::Windows::Forms;
```

Wird für die Verwendung des GUIs benötigt. Der Namespace beinhaltet alle Klassen welche Form Objekte repräsentieren.

```
using namespace System::Drawing;
```

Dieser Namespace beinhaltet zugriff auf die GDI+ funktionalität zum zeichnen. Für weitere Funktionen stehen die Unter-Namespaces `System::Drawing::Drawing2D`, `System::Drawing::Imaging` und `System::Drawing::Text` zur Verfügung.

6.7.2 Probleme

Diese traten vorwiegend da auf, wo die alte c++ Syntax (teilw. auch MFC) mit dem neuen .net Framework zusammentraf.

6.7.2.1 Beispiel

Die Bilddaten in den Tiles sind als CPixmap Format gespeichert, welches schon in Ra3dio benutzt wurde. Dieses Format wurde dementsprechend auf MFC angepasst. Für die Anzeige des Bildes unter dem .net GUI, wird aber ein Bitmap benötigt, das als Managed Objekt instanziiert wurde. (siehe Terrain zu Bild)

7 Persönliche Bemerkungen

7.1 Persönliche Bemerkungen Niklaus Jäggi

Beginn

Zu Beginn waren wir vor allem daran, Konzepte zu erstellen und haben uns lange damit herumgeschlagen, anstatt mal etwas anzugehen. Schliesslich hab ich aber angefangen den GUI Designer des Visual Studios anzuschauen.

Gui-Editor von VS.net

Der Wysiwyg Editor vom Visual Studio für die .net Umgebung ist sehr gut gelungen. Es lässt sich relativ schnell etwas erstellen. Allerdings besitzt auch er seine Tücken, mit denen ich Anfangs zu kämpfen hatte.

Unter anderem gibt es mehrere Layout Funktionen die unterschiedlich angegeben werden, und auch etwas Erfahrung brauchen damit diese richtig funktionieren. Ich musste schnell feststellen, dass Änderungen im automatisch generierten Code schnell zu unerwarteten Ergebnissen führten. Teilweise verschwand der zusätzliche Code, oder auch die Design-Vorschau funktionierte schnell nicht. Lässt man aber den Automatischen Code wie er ist, treten keine Probleme mehr auf, auch wenn dieser nicht allzu schön ist.

.net Framework

Bisher kannte ich eigentlich nur das QT Framework für c++ und AWT/Swing unter Java. Einiges konnte ich von den beiden Frameworks zwar auf .net übertragen, aber .net hat doch einige Eigenheiten.

Neben Windows Forms wird mit .net 2.0 auch noch Managed Classes eingeführt. Diese geben noch Änderungen an der Syntax mit, welche auch sehr gewöhnungsbedürftig sind. Zudem besitzt das .net Framework auch für viele bestehende Klassen eine eigene Variante wie z.B. Strings oder Arrays.

Die Arbeit mit dem .net Framework war zuerst etwas ungewohnt, aber nachdem ich mich etwas damit beschäftigt hatte, fand ich den Code doch recht intuitiv. Gerade die Arbeit mit den Windows Forms machte einigen Spass und ist mit dem GUI Designer des VS.net sehr schnell und einfach zu erstellen.

Ärger mit dem Managed Code

Die Umstellung auf die CLR und Managed Code brachte aber nicht nur Vorteile. An vielen Orten kam es auch zu kleineren Schwierigkeiten. Gerade die Schnittstellen von „altem“ c++ Code und dem managed .net Code boten immer wieder Bastelstunden bis diese wirklich liefen. Auch in den Dokumenteinstellungen musste immer wieder was gemacht werden, um die Kompilation zu gewährleisten.

Wie bereits oben erwähnt wurden mit dem .net 2.0 Framework auch einige Klassen überarbeitet und in den neuen Namespace eingefügt. Dies führte auch wieder zu einigen Schwierigkeiten, da diese gleich Benannt wurden wie existierende Klassen. (Bsp. String)

7.2 Persönliche Bemerkungen Vicki Schmid

Am Anfang hatten wir uns viel vorgenommen. Leider konnte nicht alles wie geplant umgesetzt werden.

Wir hatten keine Erfahrung mit grossen Projekten und brauchten dementsprechend länger als geplant für die Einarbeitung.

Auch programmiertechnisch gab es eine Menge Neues zu lernen. Zwar hatten wir uns im Unterricht theoretisch mit objektorientierter Programmierung befasst, hatten dies aber bisher nie wirklich angewendet.

Nach anfänglichen Startschwierigkeiten, führte ich vermehrt Besprechungen mit unserem Dozenten (der auch Auftraggeber war) durch, was sich als kluge Entscheidung erwies. So konnten Missverständnisse und Unklarheiten aus dem Weg geräumt werden, bevor sie sich ausweiteten und unnötig Zeit in Anspruch nahmen.

Den am Anfang erstellten Zeitplan würde ich wieder gleich anlegen:

Es war für mich sinnvoll, die Sitzungen mit unserem Dozenten durchgehend in Protokollen festzuhalten. An diesen konnte ich mich beim Programmieren orientieren. Ausserdem waren sie nützlich für diese Dokumentation.

Ebenfalls gut angelegt waren die Zeitpunkte für die Vorbereitung der Projektpräsentation sowie die definitive Erstellung dieser Dokumentation.

Was das Projekt selber angeht konnten wir uns leider nicht an den Zeitplan halten:

Bis Ende März war ein 1.Prototyp geplant. Ich denke, dass der geplante Termin (bis spätestens Ostern) von meiner Seite aus realistisch gewesen wäre, denn um diese Zeit stand bereits der grösste Teil des Terrains und der anderen Klassen.

Während der Entwicklungszeit ging ich stets von einem iterativen Prototypen aus bzw. davon, dass meine Lade- und Verwaltungsfunktionen und –klassen fortlaufend vom Viewer getestet wurden, dies war innerhalb des Teams auch so abgesprochen. Leider konnte aber der Terrain-Viewer bis fast zum Schluss keine Geländedaten anzeigen.

Als ich mich mit der Initialisierung beschäftigte, war ich um diese Erkenntnis klüger und testete das Auslesen der XML-Datei in einem separaten Projekt, ehe ich es ins RA₂DIO-Projekt integrierte, um so die Wahrscheinlichkeit von Fehlern zu minimieren.

Schlussendlich litt unser Projekt unter der Tatsache, dass die Zusammenarbeit in unserem Team nicht sehr effizient war.

Ich möchte hier nicht ins Detail gehen, aber darauf hinweisen, dass eine solche Arbeit, wo ein Teil vom anderen abhängt, nur dann funktioniert, wenn sich beide Team-Mitglieder ihrer Verantwortung für den anderen bewusst sind bzw. für Ihren Teil Verantwortung übernehmen können.

Was meinen Teil betrifft, habe ich mich stets bemüht, ihn erweiterbar zu halten. Ich denke, dass mir dies, obwohl ich nicht alles implementieren konnte, grösstenteils gelungen ist, und mit Terrain und den angegliederten Klassen eine solide Grundlage gelegt wurde, auf der zukünftig aufgebaut werden kann.

Auch wenn die Ziele des Projekts nicht alle erreicht wurden, erwiesen sich die anfänglich genannten „Stolpersteine“ als Pluspunkt, und es bleibt bei mir das gute Gefühl, vieles gelernt zu haben, das mir auch zukünftig nützlich sein wird.

Schliesslich möchte ich an dieser Stelle Herrn Stamm danken für die ausgiebige und hilfreiche Betreuung und Unterstützung und nicht zuletzt für die Motivierung, wenn ich im Terrain vor lauter Bäume den Wald nicht mehr sah.

8 Anhang

8.1 Programmier-Richtlinien

Um den Code so übersichtlich wie möglich zu halten, legen wir einige Richtlinien fest, bei denen es sich empfiehlt, sie auch in Zukunft weiter zu verfolgen.

8.1.1 Formatierung

Punkte, welche festgelegt wurden bezüglich Formatierung sind:

- Funktionsnamen werden gross geschrieben
- Funktionen sollen möglichst kurz gehalten werden
- `#pragma once` verwenden anstatt `#define`-Markos
- Instanzvariablen haben die Form:
m_name
- Klassennamen beginnen mit einem C:
CBeispielKlasse
- Klassenvariablen haben die Form:
s_name

8.1.2 Doxygengerechte Kommentare

- Doxygen erfordert Kommentare in der Form:
`///` oder `/**`
Damit das Debuggen via Auskommentieren erleichtert wird, entschieden wir uns, alle Doxygenkommentare nach der ersten Methode (`///`) zu schreiben.
- Kommentare, die im mit Doxygen generierten Dokumentations-File angezeigt werden, wurden wo möglich in Englisch verfasst.

8.1.3 Benennung von Datentypen

Es wurde beschlossen, Datentypen, wo dies nicht schon der Fall war, einen eigenen, möglichst aussagekräftigen Namen zu geben.

So wurde z.B. statt

```
Ctile* FindTile(int x, int y)
```

geschrieben :

```
Ctile* FindTile(TilePos x, TilePos y)
```

Die umbenannten Datentypen sind im File RA₂DIOTypes.h definiert.
Alle Namen fangen mit Grossbuchstaben an.

8.2 Autoren der Dokumentation

Diese Dokumentation wurde wie folgt erstellt:

vs: Kapitel 1, 2, 3, 4, 5.1 (ausser 5.1.3 Abschnitt „Bilderstellung im Terrain“), 6.1, 6.2, 6.3, 7.2, 8, Abstract und Management Summary

nj: Kapitel 5.1.3 Abschnitt „Bilderstellung im Terrain“, 5.3, 5.4, 6.4, 7.1

8.3 Quellenangaben, Literaturverzeichnis

- [1] Dr. Christoph Stamm: „Algorithms and Software for Radio Signal Coverage Prediction in Terrains“, 2001, DISS. ETH No.14283
- [2] Dr. Christoph Stamm: “PGF A new progressive file format for lossy and lossless image compression“, 2002
- [3] Basil Achermann: „PGF für Externspeicher“, 2003
- [4] Bundesamt für Landestopographie
<http://www.swisstopo.ch>
- [5] TinyXml: <http://www.grinninglizard.com/tinyxml>
Download auf: <http://sourceforge.net/projects/tinyxml>

Ehrlichkeitserklärung

Hiermit bestätigen die unterzeichnenden Autoren dieses Berichts, dass alle nicht klar gekennzeichneten Stellen von Ihnen selbst erarbeitet und verfasst wurden.

Muttenz, 05. Juli 2006

Niklaus Jäggi

Vicki Schmid