Technische Universität München

Lehrstuhl für Wasserbau und Wasserwirtschaft

Lehrstuhl für Computergestützte Modellierung und Simulation

Master´s Thesis

Realistische Echtzeit-Visualisierung von CFD-Ergebnissen

Johanna Frank

München, 14.03.2014

Master´s Thesis

zur Erlangung des Grades eines Master of Science (M.Sc.)

der Fachrichtung Bauingenieurwesen

an der

Technischen Universität München

Lehrstuhl für Wasserbau und Wasserwirtschaft

Lehrstuhl für Computergestützte Modellierung und Simulation

Realistische Echtzeit-Visualisierung von CFD-Ergebnissen

Prüfer:	Univ. –Prof. Dr.-Ing. Peter Rutschmann Univ. -Prof. Dr.-Ing. André Borrmann
Betreuer:	M.Sc. Tobias Liepert Dipl.-Ing. (FH) Simon Daum
eingereicht am:	14.03.2014
von:	Johanna Frank

Zusammenfassung

Im Rahmen der vorliegenden Masterarbeit wurde mit dem Softwaresystem VTK eine Anwendung implementiert. Dieses erstellte Programm CFDVis/VisualisierungCFD ist eine Echtzeit-Visualisierung von Wasseroberflächen-Geschwindigkeitswerten aus einer CFD-Simulation, die zur Laufzeit für verschiedene Zeitschritte dargestellt werden. Der Standort der simulierten Wasserdaten ist ein Flussabschnitt mit Fokus auf ein Wasserkraftwerk. Dabei ist eine Interaktion des Benutzers, mit implementierten und in VTK schon vorhandenen Methoden, zur Laufzeit gegeben. Ein wichtiger Aspekt ist auch die realistische Gestaltung der Umgebung zur Einbindung der Wasserdaten.

Abstract

This Master's Thesis aims at the implementation of the application CFDVis/VisualisierungCFD using the software system VTK. This programme is a real-time visualisation of flow velocity data at the river surface. These data are output results of a CFD simulation and are rendered at run-time for given timesteps. The site including the simulation domain for the CFD calculation is a part of a river with focus on a hydropower plant. The application provides user interaction capabilities that have been implemented or taken from existing VTK methods. An important aspect of this work is also the realistic modelling of the environment in which the above described flow data is embedded.

Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit eigenständig ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet. Das gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet und unveröffentlichte Quellen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und war bisher nicht Bestandteil einer Studien- oder Prüfungsleistung.

Ich weiß, dass die Arbeit in digitalisierter Form daraufhin überprüft werden kann, ob unerlaubte Hilfsmittel verwendet wurden und ob es sich – insgesamt oder in Teilen – um ein Plagiat handelt. Zum Vergleich meiner Arbeit mit existierenden Quellen darf sie in eine Datenbank eingestellt werden und nach der Überprüfung zum Vergleich mit künftig eingehenden Arbeiten dort verbleiben. Weitere Vervielfältigungs- und Verwertungsrechte werden dadurch nicht eingeräumt.

München, den 14.03.2014 _____

Ich möchte mich besonders bei Herrn Liepert und Herrn Daum für die aktive Betreuung dieser Masterarbeit bedanken.

Inhaltsverzeichnis

1 Einleitung und Motivation 10

2 CFD-Datensätze 12

3 Visualization Toolkit (VTK) und die Struktur einer mit VTK implementierten Anwendung 15

3.1 Das Softwaresystem Visualization Toolkit (VTK) 15

3.2 Pipeline der Visualisierung mit VTK 16

3.3 Implementierung einzelner Objekte des Visualisierungs-Netzwerkes und Rendering zur Darstellung dieser Daten 18

4 Einstellungs-Hinweise zu dem Programm VTK 20

4.1 Einstellungen zu dem Einbeziehen der VTK-Header-Dateien 20

4.2 Einstellungen zu den LIB-Dateien 20

4.3 Zusammenstellung weiterer hilfreicher Einstellungen 21

5 Hilfreiche Funktionalitäten des Programmes ParaView 22

6 Funktionalitäten aus dem Programm Blender: Fertigstellung der Daten, die in CFDVis/VisualisierungCFD eingelesen werden 26

7 Programmbeschreibung des implementierten Modells CFDVis/VisualisierungCFD. 29

8 Einlesen der VTS-Daten und Erzeugung der Wasseroberflächen mit VTK 37

8.1 VTS-Daten als vtkStructuredGrid 37

8.2 VTS-Daten-Ausgabe 38

8.3 CellData und PointData 39

8.3.1 Allgemeines Prinzip 39

8.3.2 Genaue Implementierung in VTK 40

8.3.3 Zu beachtende Eigenschaften des Filters _ Charakteristiken der Berechnungsmethode . 44

8.4 vtkContourFilter zur Erzeugung der Wasseroberfläche 46

9 Prinzip und Möglichkeiten der Texturierung mit VTK 47

9.1 Möglichkeiten der Texturierung mit VTK 47

9.2 Skalarwerte-Visualisierung durch eine vtkLookupTable 48

9.2.1 Definition eines Farbübergangs zwischen zwei vorgegebenen Werten 48

9.2.2 Bestimmung der Farbe für einzelne Skalarwerte 51

10 Texturierung zur Wiedergabe von CFD Datensätzen_ Texturierung der Wasseroberflächen mit VTK 52

10.1 Erzeugung eines Objektes der Klasse vtkLookupTable 54

10.1.1 Konstruktor und Speicherfreigabe 54

10.1.2 Eigenschaften und Aufbau des Objektes der Klasse vtkLookupTable / Untersuchung der Methode void vtkLookupTable::ForceBuild() 54

10.1.3 Einfluss der Anzahl der Farbwerte der Tabelle und der gewählten Grenz-Farbwerte 60

10.2 Zuordnung der Farben für definierte Skalarwerte eines Objektes 61

11 Texturierung des Geländemodells mit einem Luftbild 64

11.1 Überblick zu Datenformat und Erzeugung der Texturkoordinaten 64

11.2 vtkImplicitTextureCoords zur Erzeugung von Textur-Koordinaten 64

11.3 Berechnung der Texturkoordinaten aus den Punktkoordinaten 65

11.3.1 Skalierungs- und Verschiebungsimplementierung zur Texturkoordinatenberechnung in der Klasse vtkImplicitTextureCoords 65

11.3.2 Skalierungs- und Verschiebungsimplementierung zur Texturkoordinatenberechnung für die Projektion eines gesamten Bildes auf eine Oberfläche 67

12 Echtzeit-Rendering 69

12.1 Implementierung selbstdefinierter Events und Prinzip der Interaktion 69

12.2 In VTK vorhandene Interaktionsmöglichkeiten 71

12.2.1 Implementierung 71

12.2.2 Implementierte Interaktionen und Interaktionsstile 73

13 Fazit und Ausblick 76

Anhang DVD CFDVisVersion1.0

Anhang DVD CFDVisSource

Anhang 1 (Code) Einlesen der VTS-Daten mit VTK 83

Anhang 2 (Code) Visualisierung eines definierten VTS-Daten Ausschnittes mit vtkThreshold 85

Anhang 3 (Code) Attributdaten-Konvertierung mit vtkCellDataToPointData 88

Anhang 4 (Code) Erzeugung einer Oberfläche aus einem vtkStructuredGrid mit der Klasse vtkContourFilter 91

Anhang 5 (Code) Texturierung einer Oberfläche mit einem Bild / Berechnung der Texturkoordinaten nach vtkImplicitTextureCoords 92

Anhang 6 (Code) Texturierung einer Oberfläche mit einem Bild / Berechnung der Texturkoordinaten zur Überlagerung der Ränder des Bildes und der Oberfläche 95

Anhang 7 URL-Sammlung verwendeter Codebeispiele (nach Themen geordnet) 98

Anhang 8 Allgemeine Bemerkungen 100

1 Einleitung und Motivation

Die Visualisierung von Standorten, Prozessen und Bauwerken ist eine wertvolle und effektive Art, Daten zu analysieren, zu interpretieren, sowie eine geeignete Methode, diese zu präsentieren und zu veranschaulichen. Besonders in Phasen der Entwicklung und der Genehmigung, u.a. von Bauvorhaben wie von Wasserbaulichen Anlagen, können Standort-Visualisierungen hilfreich sein.

Wesentliche Aspekte der Darstellung sind die quantitative Richtigkeit der dargestellten Daten. Dem steht der Aspekt der realistischen Einbindung dieser Daten gegenüber.

Die Nutzung von Programmen wie z.B. Blender zur Erstellung dreidimensionaler Modelle und Animationen ermöglicht eine realistische Gestaltung der Umgebung. Simulationsberechnungen wie z. B. *Physics Fluid* aus Blender können verwendet werden, wenn es darum geht Fluide intuitiv realistisch und ansprechend zu gestalten. Z.B. durch die Beschränkung der Größe der Berechnungs-*Domain* müssen tatsächlich, zur Darstellung von Wasserkraftanlagen, physikalische Eigenschaften so angepasst werden, dass der Bereich der Simulation größer erscheint, als dieser eingestellt wird.

Die Visualisierung von Fluidsimulationen (CFD-Ergebnissen) ist hier das Ziel, welches den quantitativen Ansprüchen für die numerischen Berechnungen entspricht.

Durch eine kombinierte Nutzung von Blender und dem Sofwaresystem VTK (welches zur Visualisierung von CFD-Daten in der Anwendung ParaView verwendet wird) ist eines der Ziele dieser Arbeit, die numerisch hochwertigen Datensätze in eine realistisch gestaltete Umgebung einzubinden.

Ein weiterer Aspekt, der bei der Erstellung von Modellen zu beachten ist, ist die Automatisierung der Standortmodellierung. Durch Implementierung einer Anwendung kann erreicht werden, dass der Modellierungsaufwand für einen gegebenen, neuen Standort reduziert wird.

VTK ist zudem für Echtzeit-Visualisierungen geeignet, bei denen Daten durch implementierte Benutzerinteraktionen gezielt und flexibel untersucht und präsentiert werden können.

Im Rahmen der vorliegenden Masterarbeit wurde mit dem Softwaresystem VTK eine Anwendung implementiert. Dieses erstellte Programm CFDVis/VisualisierungCFD ist eine Echtzeit-Visualisierung von

Wasseroberflächen-Geschwindigkeitswerten aus einer CFD-Simulation, die zur Laufzeit für verschiedene Zeitschritte dargestellt werden. Der Standort der simulierten Wasserdaten ist ein Flussabschnitt mit Fokus auf ein Wasserkraftwerk. Dabei ist eine Interaktion des Benutzers, mit implementierten und in VTK schon vorhandenen Methoden, zur Laufzeit gegeben. Ein wichtiger Aspekt ist auch die realistische Gestaltung der Umgebung zur Einbindung der Wasserdaten.

Vor der Beschreibung des erstellten Modells und seiner Funktionalitäten (Abschnitt 7) werden die Eingangsdaten vorgestellt, und einige Aspekte, zu den für die Implementierung verwendeten Programmen, aufgeführt (Abschnitte 2 bis 6). In den Abschnitten 8 bis 12 wird dann im Detail auf einige in CFDVis/VisualisierungCFD implementierte und aus VTK verwendete Methoden eingegangen.

Eine wichtige Eigenschaft des implementierten Programmes soll auch sein, dass der Aufwand, zur Anpassung des Programmes zur Nutzung neuer Eingangsdaten für die Erstellung eines ähnlichen Modells für einen anderen Standort, möglichst gering gehalten wird. In der Programmbeschreibung wird daher auch auf Möglichkeiten der Modellanpassung und Veränderung eingegangen.

An dieser Stelle wird darauf hingewiesen, dass durch die Nutzung des Softwaresystems VTK, technische Hintergründe, Definitionen von VTK-Objekten und –Methoden und in VTK implementierte Vorgehensweisen und Pipelinevorgaben zur eigenen Implementierung für diese Masterarbeit aus folgenden Quellen entnommen oder mithilfe dieser Quellen nachvollzogen wurden. Dies gilt für die gesamte Arbeit und Implementierung, wobei nur an bestimmten Stellen ausdrücklich darauf verwiesen werden kann. Die wichtigsten für die eigene Anwendungsimplementierung herangezogenen Programmierungsbeispiele aus dem Internet sind in Anhang 7 aufgeführt.

Allgemeine Quellenangaben:

- Der VTK Quellcode (Header- und CXX-Dateien) ist auf der beiliegenden DVD „CFDVisSource“ in CFDVisSource\VTK6.0.0 enthalten und kann über diesen Pfad geöffnet und untersucht werden.

- Über www.vtk.org wird auf die Seite www.vtk.org/Wiki/VTK/Examples/Cxx verwiesen (vtk.org 2014 e). Darin sind Code-Beispiele zu verschiedenen Themen und Funktionalitäten aufgeführt. Auf Beispiele die für die Implementierung von CFDVis verwendet wurden, wird in Anhang 7 nochmals, nach in dieser Arbeit verwendeten Themenbereichen geordnet, verwiesen.

- www.vtk.org/doc/nightly/html/classvtkLookupTable.html gibt beispielhaft den Link für die Klasse vtkLookupTable wieder (vtk.org 2014 f). Diese Seiten geben für die betreffende Klasse u.a. Vererbungsdiagramme an. Methoden und Attribute werden dargestellt und eine Beschreibung der Klasse ist angegeben.

- Über www.vtk.org kann neben den zuvor genannten Seiten (die hier am häufigsten gebraucht wurden) noch weitere Dokumentation gefunden werden (z.B. auch ein Klassenindex etc.).

- Kitware, Inc. The VTK User's Guide. (Updated for VTK Version 4.4) Kitware, Inc., 2004.

- Schroeder W., Martin K., und Lorensen B. The Visualization Toolkit. An Object-Oriented Approach To 3D Graphics. Kitware, Inc., 2006.

2 CFD-Datensätze

Als erstes werden nun die Eingangsdaten für das implementierte Modell beschrieben. Die im Rahmen dieser Arbeit erstellte Anwendung dient der Visualisierung eines Standortes am Lech (Wie schon erwähnt ist dabei die Anpassungsfähigkeit für andere ähnliche Standorte zu beachten). Für ein Schachtkraftwerk (ein an der TU München entwickeltes Wasserkraftkonzept) werden in dieser Umgebung Strömungsberechnungs-Ergebnisse dargestellt. Die damit erzeugten Wasseroberflächen geben Geschwindigkeitswerte visuell wieder.

Dieser Abschnitt beschäftigt sich mit den CFD-Daten die zur Erstellung der Wasseroberflächen der Anwendung visualisiert wurden.

CFD steht für *computational fluid dynamics*. CDF behandelt vor allem bewegte Fluide und wie dessen Verhalten sich auf Prozesse wie z.B. den Wärmeübergang auswirkt. Die physikalischen Eigenschaften der Fluidbewegung werden i.d.R. durch mathematische Gleichungen (i.d.R partielle Differentialgleichungen) beschrieben die den betrachteten Prozess steuern. Die Untersuchung der Fluidströmung erfolgt durch Nutzung von numerischen Simulationen. Zur Untersuchung verschiedener Aspekte der Fluidströmung und zur Entwicklung von Anlagen und industriellen Prozessen, welche mit Fluidströmung und Wärmeübergang zusammenhängen, werden weiterhin experimentelle und analytische Methoden herangezogen. Die computerbasierte Herangehensweise wird für industrielle Anwendungen immer häufiger verwendet (Tu, Yeoh und Liu 2013, 1-3).

Die Visualisierung numerischer Ergebnisse durch Nutzung von Vektoren, Kontouren oder Animationen von instationären Strömungen ist eindeutig die effektivste Interpretationsform der Menge an daraus resultierenden numerisch berechneten Daten. (Dabei ist jedoch darauf zu achten, dass die numerischen interpretierten Daten quantitativ richtig sind. Berechnete Ergebnisse sollten kritisch analysiert und bewertet werden) (Tu, Yeoh und Liu 2013, 5).

Oftmals können CFD-Simulationen eine entscheidende Rolle bei der Lösung von Umweltproblemen spielen. CFD wird z.B. zur Vorhersage der Dispersion von Schadstoffen verwendet. CFD kann auch dabei helfen, die Einhaltung strikter rechtlicher Vorgaben in den ersten Planungsphasen von Bauvorhaben zu gewährleisten (Tu, Yeoh und Liu 2013, 16f). Auch im Bereich des Wasserbaus und der Planung von Wasserkraftanlagen und –Konzepten ist dies zutreffend.

Im Folgenden werden bestimmte Aspekte der CFD-Berechnung kurz beschrieben.

Zunächst wird in der CFD-Analyse eine Strömungsregion definiert, und dessen Geometrie generiert. Diese Region wird in ein Zellen-Gitter unterteilt. Für jede dieser Zellen werden die Fluidströmungsparameter (z.B. Geschwindigkeit, Druck, Temperatur) numerisch berechnet. Die Anzahl der Zellen in der Berechnungsregion beeinflusst die Genauigkeit der CFD-Ergebnisse maßgebend. Die Ausgangsdaten werden auch durch Parameter wie die Art des Gitters, Eigenschaften der numerischen Methoden, oder die Entsprechung der gewählten Techniken gegenüber den physikalischen Eigenschaften der Aufgabe beeinflusst. Die Genauigkeit der Ergebnisse ist jedoch stark von Hardware und Berechnungszeiten abhängig (Tu, Yeoh und Liu 2013, 34-37).

Im Folgenden werden Möglichkeiten beschrieben, wie CFD-Ergebnisse üblicherweise graphisch wiedergegeben werden (Tu, Yeoh und Liu 2013, 50-58).

Die erste Darstellungsart sind „X-Y plots“. Solche 2D-Grafiken geben die numerischen Daten am genausten wieder, indem die Veränderung einer abhängigen Ergebnis-Variable mit einer anderen unabhängigen Variable in Verbindung gesetzt wird. Solche Grafiken eignen sich zum Vergleich der numerischen Daten mit experimentellen Ergebnissen aus Labor-Versuchen.

„Vector Plots“ beinhalten (für diskrete Punkte) die Richtung und die Größe für i.d.R. Geschwindigkeitsdaten in Form eines Vektorfeldes.

„Contour Plots“ gehören zu den gängigsten grafischen Darstellungsformen von CFD-Daten. Dabei werden Linien bzw. Flächen gleichen Wertes (die Eigenschaft ist räumlich für eine Linie bzw. Fläche konstant) für gegebene Daten erzeugt. Diese

Art der Darstellung ist in einer VTK-Klasse (vtkContourFilter -> Abschnitt 8.4) implementiert und wurde zur Erstellung des Modells in dieser Arbeit verwendet.

Stromlinien werden auch als Darstellungsform zur Untersuchung von Strömungseigenschaften herangezogen.

In bestimmten Fällen können die instationären Eigenschaften einer Strömung wichtig sein. Daten können für eine bestimmte Anzahl an maßgebenden Ergebnisvariablen verfolgt werden, und in Dateien ausgegeben werden, um eine zutreffende Beschreibung der zeitabhängigen Antwort des Strömungsprozesses zu erzielen. In der Anwendung CFDVis/VisualisierungCFD wurde die Darstellung solcher instationärer Simulationsergebnisse implementiert.

Schließlich werden Animationen aus CFD-Simulationen zur physikalischen Darstellung von Fluidströmungsprozessen produziert.

Im Folgenden sollen numerische Berechnungsmethoden der Ergebnisdaten der CFD-Simulation kurz erwähnt werden. Die Geschwindigkeits- und Druckwerte ergeben sich aus den Navier-Stokes-Gleichungen (Tu, Yeoh und Liu 2013, 72ff).

Verschiedene numerische Methoden können angewandt werden. Dazu zählen z.B. *Reynolds-Averaged Navier Stokes* (RANS), dessen instationäre Version (uRANS), *Large Eddy Simulation* (LES) und *Direct Numerical Simulation* (DNS) (www2.le.ac.uk 2014).

Weitere Ergebnisdaten, die für die Implementierung der Anwendung CFDVis von Bedeutung sind ist der Datensatz f. Eine Zelle kann bei der Berechnung von Wasserströmungen in einen festen Anteile, eine Wasserfraktion und eine Luftfraktion eingeteilt werden. Die Zelldaten f aus [0;1] bezeichnen dabei die Wasserfraktion der Zelle. Dabei gilt für eine mit Wasser gefüllte Zelle f = 1. Eine Zelle, die kein Wasser enthält hat einen f-Wert von 0. Bei einem Wert f = 0,5 wird im Wasserbau die Wasseroberfläche angenommen. Abb. 1 zeigt einen der CFD-Ergebnisdatensätze, die in dem erstellten Modell visualisiert wurden. Die Simulation wurde für ein Kraftwerksmodell berechnet. Die Zellen sind in Abb. 1 durch die f-Werte gefärbt. In dem Bauwerksbereich ist tatsächlich f = 0 (blau). Darüber sind die Zellen bis zu der Wasseroberfläche mit Wasser gefüllt. Darüber gilt wieder f = 0. Auch Strukturen wie der Überfallstrahl werden mit f ungleich null sichtbar.

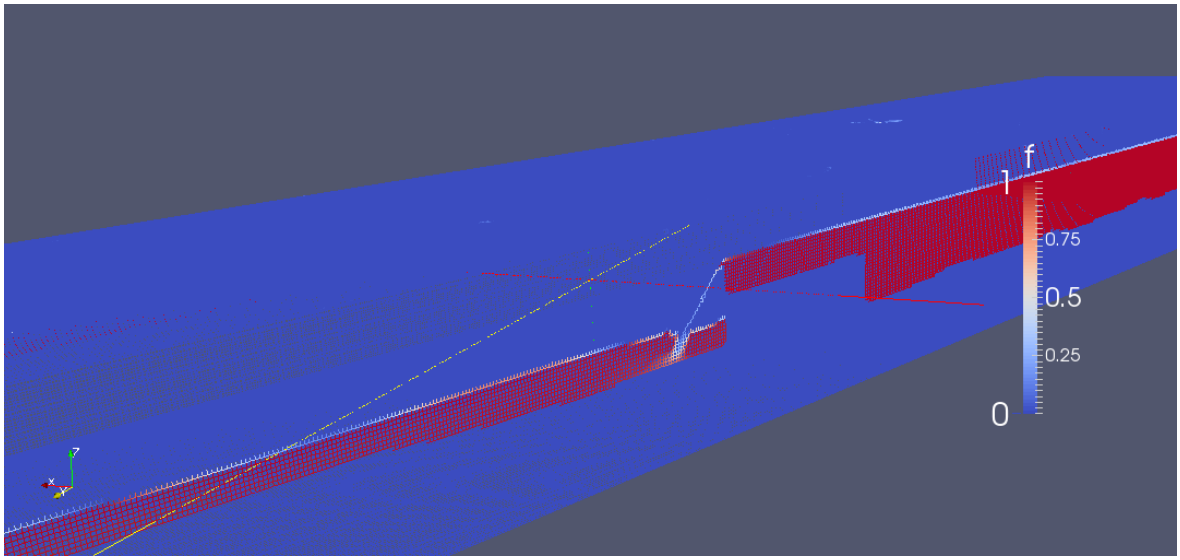


Abb. 1 Darstellung eines CFD-Ergebnis-Datensatzes mit ParaView. Färbung der Zellen durch f [0;1]

3 Visualization Toolkit (VTK) und die Struktur einer mit VTK implementierten Anwendung

Nun sollen das verwendete Softwaresystem VTK, und der Aufbau einer damit implementierten Anwendung beschrieben werden.

3.1 Das Softwaresystem Visualization Toolkit (VTK)

VTK wurde durch das Kitware-Team entwickelt, welches für eine kontinuierliche Erweiterung sorgt. VTK ist ein open-source Softwaresystem für 3D-Computergrafik, Bildverarbeitung und Visualisierung (vtk.org 2014 a).

VTK beinhaltet eine Sammlung an Klassen die zur Implementierung von, in 3.2 und 3.3 beschriebenen, Visualisierungen zusammenwirken können.

Ein Softwaresystem kann tatsächlich folgender Weise definiert werden: „Programmiertechnisch […] gesehen besteht ein Softwaresystem aus Programmen […], aus deren Zusammenwirken sich die Lösung eines Problems ergibt.“ (wirtschftslexikon.gabler.de 2014)

Die Klassen sind in C++ implementiert. Anwendungen können auch mit Tcl/Tk, Java und Python programmiert werden (vtk.org 2014 a).

Die Lizenz von VTK ist eine *BSD license*. Verbreitung und Nutzung (*source* oder *binary*) mit oder ohne Veränderung sind erlaubt, wenn bestimmte Konditionen

erfüllt sind (vtk.org 2014 d). Diese können dem im Code vorhandenen Lizenztext entnommen werden (DVD „CFDVisSource“: CDFVisSource \ VTK6.0.0 \ Copyright.txt).

Für die in dieser Arbeit implementierte Anwendung wurde die Version VTK6.0.0 verwendet. Als Programmierumgebung wurde Visual Studio 2012 benutzt.

3.2 Pipeline der Visualisierung mit VTK

Zur Visualisierung von Daten mit VTK kann zunächst zwischen Objekten unterschieden werden, die die Daten beinhalten, und Objekten, die Daten verarbeiten. Letztere sind die so genannten „process objects“ und können wiederum in drei Gruppen eingeteilt werden. Die „source objects“ können externe Daten verarbeiten, wie z.B. `vtkXMLStructuredGridReader` zum Einlesen von VTS-Daten (Abschnitt 8 und Anhang 1). Dabei wird eine Umwandlung der externen eingelesenen Dateien in eine interne Form durchgeführt. Filter verwenden Eingangsdaten und geben Output-Daten aus. Mapper-Objekte haben auch Input-Daten, beenden aber den Datenfluss der Visualisierungs-Pipeline. Wie im Laufe dieses Abschnittes näher beschrieben wird, werden Mapper-Objekte oft zur Konvertierung der Daten zu graphischen Einheiten verwendet. Diese können z.B. auch Daten in Dateien ausgeben (z.B. `vtkXMLStructuredGridWriter` siehe Anhang 1). (Schroeder, Martin und Lorensen 2006, 85f)

Die Output-Werte eines VTK-Filter-Objektes werden in einem oder mehreren so genannten „output ports“ gespeichert. Dabei enthält ein „output port“ eine logische Einheit des Outputs wie z.B. ein Farbbild (Schroeder, Martin und Lorensen 2006, 103). Mit `SetInputConnection()` und `GetOutputPort()` werden Ausgabewerte eines „process object“ als Input-Werte eines folgenden Objektes übergeben. Es entsteht eine Pipeline zur Visualisierung der Daten (Abb. 2).

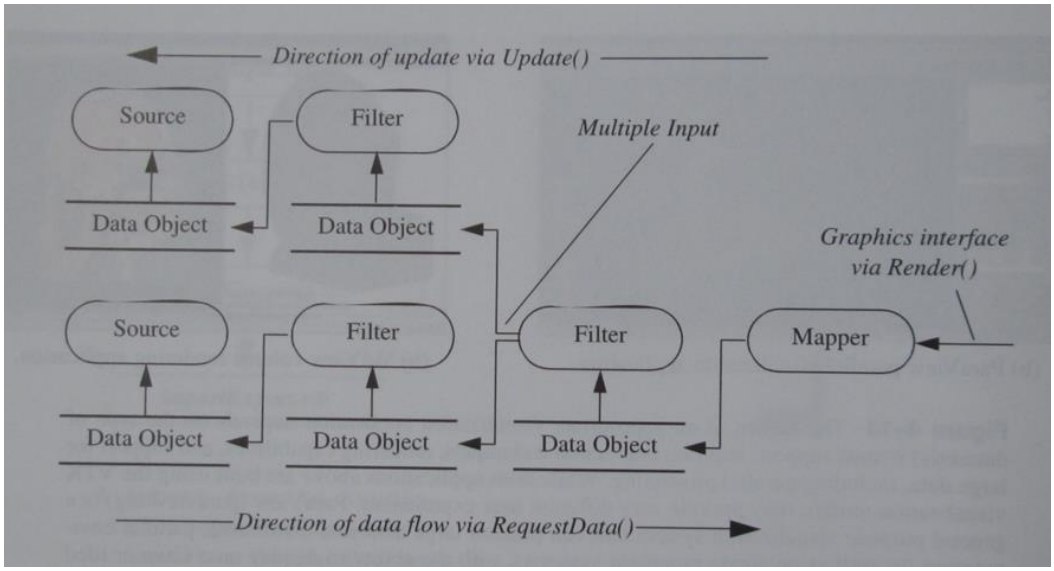


Abb. 2 Visualisierungs-Pipeline / Ausführung des Netzwerkes als „implicit execution process“ wie es in VTK implementiert ist (Schroeder, Martin und Lorensen 2006, 104)

Die graphische Darstellung der, aus der Pipeline resultierenden Output-Daten des Mapper-Objektes, wird im Laufe des Abschnittes erläutert. Hier wird zunächst der Aufruf Render() für den Mapper angenommen.

Durch diese Methode (d.h. wenn für ein bestimmtes Objekt das Output benötigt wird) wird Update() für die Elemente (d.h. VTK-Klassen) aufgerufen, bis ein „Source“-Objekt erreicht wird (Abb. 2). Die Pfeile zwischen den Filtern und den Daten stellen die Update-Prozess-Richtung dar. Mit RequestData() werden in entgegengesetzter Richtung die Filter ausgeführt und Output-Daten generiert, falls eine Veränderung vorliegt. Dieses Modell zur Ausführung der Visualisierung wird als „Implicit Execution“ bezeichnet (Schroeder, Martin und Lorensen 2006, 91 und 104).

Die Eigenschaft, dass das Visualisierungsnetzwerk nur dann ausgeführt wird, wenn Ausgabewerte abgefragt werden, wird als „demand-driven control“ bezeichnet (Schroeder, Martin und Lorensen 2006, 89ff).

Die Update() Methode ist in VTK als public deklariert und kann damit in der Implementierung aufgerufen werden, um Teile der Pipeline auszuführen (Schroeder, Martin und Lorensen 2006, 112).

Die Methoden RequestData() der VTK-Klassen (die als Pipeline-Objekte verwendet werden) können analysiert werden, um Ausgaben zu untersuchen. Dies wird für bestimmte Klassen in den folgenden Abschnitten im Detail dargestellt.

3.3 Implementierung einzelner Objekte des Visualisierungs-Netzwerkes und Rendering zur Darstellung dieser Daten

Alle verwendeten VTK-Klassen-Objekte wurden zur Implementierung dieser Arbeit wie folgt deklariert: `vtkSmartPointer<vtkPolyDataMapper> mapper = vtkSmartPointer<vtkPolyDataMapper>::New();` (Beispiel für ein Mapper-Objekt).

vtkSmartPointer ist ein Klassentemplate (vtk.org 2014 b). Dessen Nutzung erleichtert die Speicherverwaltung, da Objekte nicht explizit mit Delete gelöscht werden müssen (vtk.org 2014 c).

Im Folgenden werden die VTK-Klassen beschrieben, die zur Visualisierung der Output-Daten des Mapper-Objektes nacheinander verwendet werden.

Wie schon beschrieben gibt der Mapper die durch die Visualisierungs-Pipeline veränderten Daten aus. Ein Objekt der Klasse `vtkPolyDataMapper` kann genutzt werden um Textureinstellungen anhand von Skalarwerten vorzunehmen. In dem erstellten Modell wurde der Wasseroberfläche in dem Mapper, der Farbverlauf in Abhängigkeit der Geschwindigkeit, mit einer `vtkLookupTable` zugewiesen (Abschnitt 9 und 10). Für die Projektion des Luftbildes auf das Gelände wurde in dem Mapper festgelegt, welche Koordinaten zu verwenden sind (Abschnitt 11).

Das Mapper-Objekt wird einem Actor-Objekt `vtkActor` übergeben. Damit können für das Objekt Oberflächeneigenschaften eingestellt werden, wie z.B. eine Färbung (Angabe einer Farbe für die gesamte Objektoberfläche) oder Lichtspiegelungseigenschaften („specular color“). Die Zuweisung einer Textur (z.B. zur Luftbildprojektion verwendet) und Darstellungseigenschaften, wie z.B. „surface“ oder „wireframe“, sind weitere Objekteigenschaften, die an dieser Stelle festgelegt werden können (Kitware 2004, 48f). Zu beachten ist, dass die Actorfarbe nur einen Einfluss hat, wenn in dem Mapper keine Skalarwerte vorhanden sind. Mit vtkMapper::ScalarVisibilityOff() können die skalaren Werte ignoriert werden, und die Actorfarbe dargestellt werden (Kitware 2004, 50). Das Actor-Objekt kann mit vtkProp3D::SetPosition() in *world coordinates* platziert werden (`vtkActor` ist eine Unterklasse von `vtkProp3D`). Mit vtkProp3D::AddPosition() kann das Objekt verschoben werden (Kitware 2004, 47f). Das Objekt Actor entspricht den Daten, wie diese darzustellen sind. Mit der Erstellung der Actors ist das Modell zur Übergabe für das Rendering bereit.

Durch einen `vtkRenderer` werden die Objekte der Klasse `vtkActor` in einem Fenster `vtkRenderWindow` dargestellt. (Ein Renderer kann auch in nur einem Abschnitt des RenderWindows darstellen, sodass mehrere Renderer in einem Fenster rendern können.) Mit dem `vtkRenderer` werden die Kamera- (Abb. 33 und CFDVis-Modellimplementierung) und Lichteinstellungen übernommen. Es ist wichtig, die Lichter der Szene zu erzeugen, bevor Texturen erstellt und bearbeitet werden, da die Belichtung das Aussehen sehr stark verändert.

Werden keine Lichter oder Kamera erzeugt, werden in VTK automatisch für das Rendern welche eingestellt. In VTK ist das Licht ein Objekt der Klasse `vtkLight`. Durch die Einstellungen können zwei Arten von Lichtern verwendet werden. Das eine wird mit einer Position im Unendlichen angenommen, d.h. mit parallelen Lichtstrahlen. Es wird als *directional light* bezeichnet. Durch Hinzufügen von `PositionalOn()` und `SetConeAngle()` wird das Licht zu einem Spotlicht *positional light* (Kitware 2004, 45ff). (Der Implementierung der CFD-Visualisierung können weitere Einstellungen entnommen werden.)

Die Darstellung von Schatten kann mit VTK nicht durch die Lichter und Objekteigenschaften erreicht werden, wie es in Programmen wie Blender einzustellen ist. Es gibt Möglichkeiten, Schatten durch die Vorgabe von Rendervorgängen zu erzeugen. Dabei sind die Klassen vtkShadowMapBakerPass und vtkShadowMapPass von besonderer Bedeutung. Die Darstellung der Schatten ist jedoch aufwändig. Da bei der Implementierung des erstellten Modells ein wichtiger Aspekt die Echtzeit-Visualisierung ist, wurde an dieser Stelle mehr Wert auf die Begrenzung der Rechenzeiten gelegt, als auf Feinheiten wie die Darstellung der Schatten. Damit wird hier nicht weiter auf die Erzeugung von Schatten eingegangen.

Dem Objekt `vtkRenderer` werden damit die Actors, die aktive Kamera und die Lichter übergeben. Auch die Einstellung einer Hintergrundfarbe kann mit dem Renderer vorgenommen werden.

Dieser wird dem `vtkRenderWindow` zugewiesen. Die Größe des Fensters kann bestimmt werden. Mit Render() wird die zuvor geschilderte Update()-Schleife ausgelöst, sodass die Visualisierungs-Pipeline ausgeführt wird, und die Renderansicht in dem Fenster erscheint.

Schließlich ermöglicht ein `vtkRenderWindowInteractor` die Interaktion des Benutzers und die Echtzeit-Visualisierung der CFD-Ergebnisse (Abschnitt 12).

4 Einstellungs-Hinweise zu dem Programm VTK

Zu VTK sollen abschließend noch Hinweise zu einzelnen Einstellungen aufgeführt werden, bevor dann die zusätzlich zu VTK verwendete Software vorgestellt wird.

Nach der Installation sind einzelne Einstellungen von besonderer Bedeutung, um das Laufen des erstellten Codes zu ermöglichen. In diesem Abschnitt werden solche kurz aufgeführt, mit dem Ziel, das Starten und Erlernen des Softwaresystems zu erleichtern.

4.1 Einstellungen zu dem Einbeziehen der VTK-Header-Dateien

Ein regelmäßig zu behebender Fehler ist die fehlende Angabe von VTK-Header-Dateien in dem entsprechenden Verzeichnis. Die zugehörige Fehlermeldung ist in Abb. 3 dargestellt.

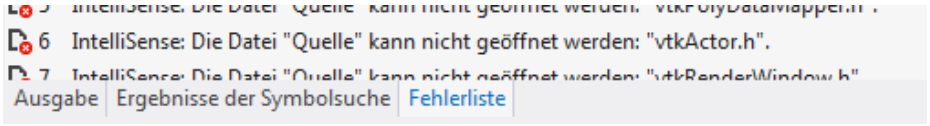


Abb. 3 Fehlermeldung bei fehlender Einbeziehung von Header-Dateien

In dem Projektmappen-Explorer wird mit Rechtsklick auf den Projektnamen, das Eigenschaften-Fenster geöffnet. Bei „Konfigurationseigenschaften-> VC++-Verzeichnisse->Includeverzeichnisse->Bearbeiten“ wird in der *Listbox* der Pfad zu dem Ordner, der die benötigte VTK-Header-Datei beinhaltet, hinzugefügt und mit „OK“ bestätigt.

4.2 Einstellungen zu den LIB-Dateien

Wenn die LIB-Dateien nicht geöffnet werden können, kann mit „Rechtsklick auf den Projektnamen->Eigenschaften->Konfigurationseigenschaften->Linker->Eingabe->Zusätzliche Abhängigkeiten->Bearbeiten“ in der *Listbox* eine Liste der Dateinamen (mit .lib-Endung) eingegeben werden.

Der Pfad zu diesen Dateien wird wie folgt angegeben. Mit „Rechtsklick auf den Projektnamen->Eigenschaften->Konfigurationseigenschaften->VC++-Verzeichnisse->Bibliotheksverzeichnisse->Bearbeiten“ wird in der *Listbox* der Pfad des Orders, der die LIB-Dateien beinhaltet, eingefügt.

4.3 Zusammenstellung weiterer hilfreicher Einstellungen

Zur erfolgreichen Nutzung von VTK muss auch folgendes noch beachtet werden. Mit Rechtsklick auf den Projektnamen wird wie zuvor das Eigenschaftenfenster geöffnet. Bei „Konfigurationseigenschaften->C/C++->Präprozessor->Präprozessordefinitionen->Bearbeiten“ können in der *Listbox* folgende Zeilen eingetragen werden:

WIN32

_WINDOWS

_DEBUG

vtkRenderingCore_INCLUDE="*Pfad zur folgenden Header-Datei* \vtkRenderingCore_AUTOINIT_vtkInteractionStyle_vtkRenderingFreeType_vtkRenderingFreeTypeOpenGL_vtkRenderingOpenGL.h"

vtkRenderingVolume_AUTOINIT=1(vtkRenderingVolumeOpenGL)

CMAKE_INTDIR="Debug"

Weiter sollte das Arbeitsverzeichnis (*Projekteigenschaftenfenster*->Konfigurationseigenschaften->Debugging->Arbeitsverzeichnis) gleich dem Ausgabeverzeichnis (*Projekteigenschaftenfenster*->Konfigurationseigenschaften->Allgemein->Ausgabeverzeichnis) sein.

Dateien, die in dem Projekt verwendet werden, z.B. STL-Daten die eingelesen werden, können in den Ordner eingefügt werden, in dem die EXE-Datei liegt. Auf die Dateien kann somit in dem Code mit Angabe des *Dateinamen.Endung* zugegriffen werden.

Wenn in dem Ausgabe-Fenster bei dem Debuggen gemeldet wird, dass die Symboldateien (.pdb) nicht gefunden wurden, kann in Microsoft Visual Studio bei Tools->Optionen->Debugging->Symbole „Microsoft-Symbolserver“ aktiviert werden. Nach Bestätigung mit „OK“ kann der Code ausgeführt werden. In dem Ausgabefenster sollte dann „Symbole wurden geladen.“ gemeldet werden. Dies ist weiterhin der Fall, auch wenn „Microsoft-Symbolserver“ wieder deaktiviert wird. (Dabei war ein Pfad zu „Symbole in diesem Verzeichnis zwischenspeichern“ angegeben, und bei „Symbole automatisch laden für“ wurde „Alle nicht ausgeschlossenen Module“ gewählt.)

5 Hilfreiche Funktionalitäten des Programmes ParaView

Die Fehlersuche und Nachverfolgung von Programmteilen mit Debugging ist durch die in Abschnitt 3.2 beschriebene Struktur zur Ausführung der Visualisierungs-Pipeline erschwert. Die Verwendung von weiteren Programmen ist dabei, und für die Suche nach passenden VTK-Klassen für bestimmte Funktionalitäten, sehr hilfreich.

Bei der Implementierung der in dieser Arbeit programmierten Anwendung CFDVis/VisualisierungCFD wurde das Programm ParaView verwendet. Die wichtigsten Funktionalitäten, die hier genutzt wurden, werden in diesem Abschnitt beschrieben.

ParaView ist eine open-source Anwendung zur Datenanalyse und Visualisierung (paraview.org 2014). Es wurde durch Kitware entwickelt und verwendet VTK (vtk.org 2014 h). Hier wurde mit der Version ParaView 4.0.1 gearbeitet.

Allgemein kann ParaView zur Visualisierung und Analyse von CFD-Ergebnissen verwendet werden. Die für diese Arbeit gegebenen Simulationsergebnisse im VTS-Format, dessen Darstellung implementiert wurde, können intuitiv in ParaView geladen und bearbeitet werden (Abb. 1). Dazu wird mit „File->Open“ die VTS-Datei gewählt und geöffnet. In der „Properties-View“ (auf der linken Bildschirmseite) werden die Zell- bzw. Punktdaten der VTS-Datei angezeigt (Abschnitt 8). Mit „Apply“ wird der Datensatz dargestellt. In „Properties“ kann mit „Representation“ die Art der Oberflächendarstellung bestimmt werden, z.B. Wireframe. In der Rubrik „Coloring“ kann ein Datensatz ausgewählt werden, dessen Werte zur Färbung der Zellen bzw. Punkte verwendet werden sollen. Mit „Show“ wird der Farbverlauf (Legende) angezeigt, der zur Färbung benutzt wird. Mit „Edit“ kann der Wertebereich gewählt werden, der dem Farbverlauf zugeordnet ist (Abb. 4).

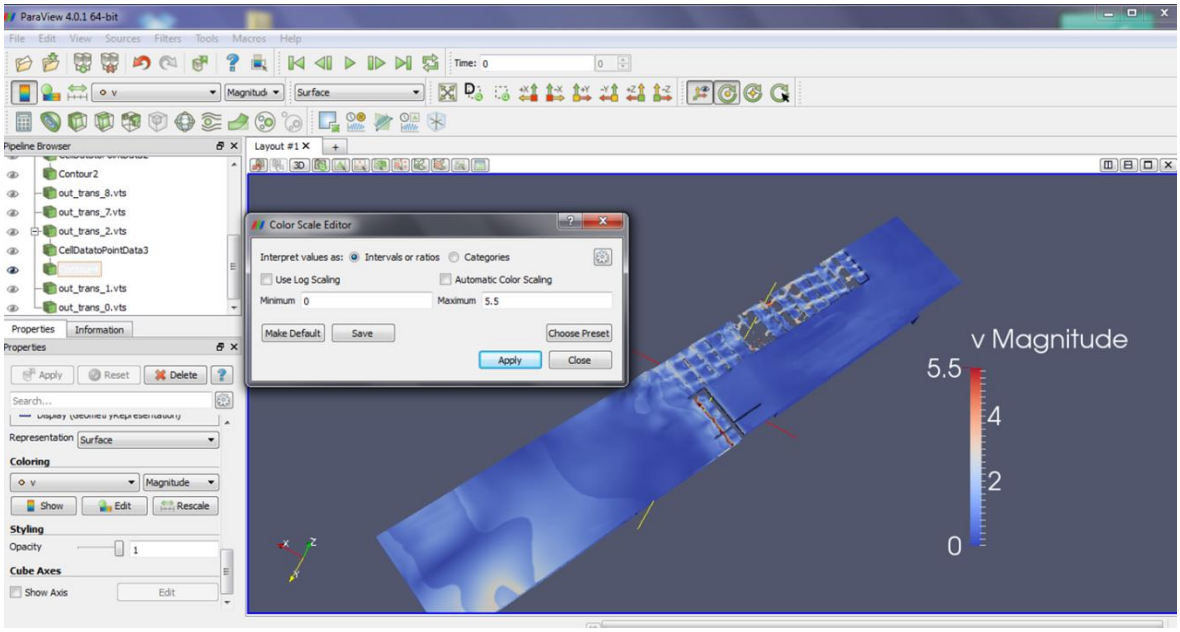


Abb. 4 Färbung eines VTS-Datensatzes mit ParaView. Anpassung des Wertebereiches der Farbskala

Diese Funktionalität kann bei der eigenen Implementierung der Texturierung einer Oberfläche durch Skalarwerte genutzt werden, um den Wertebereich zu ermitteln, der für die Färbung durch einen Farbverlauf definiert werden soll. In dieser Arbeit wurde die Methode bei der Texturierung der Wasseroberflächen in Abhängigkeit von der Geschwindigkeit verwendet (Abschnitte 9 und 10).

Mit „File->Save State“ kann ein aktueller Zustand gespeichert werden (Endung .pvsm). Mit „File->Load State“ kann die Ansicht wieder geöffnet und weiter bearbeitet werden.

ParaView verfügt über die Möglichkeit, den Code (Python), der für die aufgerufenen Funktionalitäten ausgeführt wird, anzuzeigen. Um die Codeanzeige zu starten wird „Tools->Start Trace“ gewählt. Dann werden ParaView Funktionalitäten normal aufgerufen und durchgeführt, für die der Code angezeigt werden soll. Zum Beenden der Codeanzeige wird „Tools->Stop Trace“ gewählt. Der Code öffnet sich automatisch und kann als *Python script* gespeichert werden. Dadurch, dass ParaView VTK benutzt, kann der Python Code dabei helfen, die benötigten VTK-Klassen für die eigene Anwendungsimplementierung zu finden, und bestimmte Klassen-Objekteinstellungen oder den Ablauf zu implementierender Anweisungen zu verstehen. Dies war bei der Implementierung dieser Arbeit besonders für die Erstellung der Wasseroberflächen zum Einlesen der VTS-Daten (Anhang 1 und 2), zur Konvertierung der Zell- zu Punktdaten (Abschnitt 8.3 und

Anhang 3) und zur Oberflächenerzeugung mit vtkContourFilter (Abschnitt 8.4 und Anhang 4) und dessen Texturierung mit vtkLookupTable (Abschnitte 9 und 10) hilfreich.

In ParaView können Filter durch Markieren eines Datensatzes mit *Strg+Leertaste* hinzugefügt werden. Mit „Properties->Apply“ wird das Filter auf die gewählten Daten angewendet. In „Properties“ können davor die Parameter eingestellt werden. Für das Filter „Cell Data to Point Data“ kann z.B. zur Erhaltung von Zell- und Punktdaten im Output „Pass Cell Data“ gewählt werden. Für das Filter „Contour“ kann mit „Properties->Contour By“ der Datensatz gewählt werden, dessen Werte, z.B. zur Bestimmung der Fläche konstanten Wertes, verwendet werden sollen. Der für die Fläche zu verwendende Wert dieses Datensatzes wird in dem Textfeld „properties->Isosurfaces“ eingestellt. Weitere Informationen zu den Filtern werden in den Folgenden Abschnitten zu VTK geschildert.

Abb. 5 zeigt beispielhaft Teile eines *Python script* aus ParaView zum Öffnen einer VTS-Datei (CFD-Ergebnisse), Konvertierung derer Zelldaten zu Punktdaten, und Erstellung einer Oberfläche für die Punktdaten f = 0,5. Die Färbung der Oberfläche wurde mit den Geschwindigkeits-Zellwerten [0;3] vorgegeben.

```
out_trans_1_vts = XMLStructuredGridReader( FileName=[,Pfad zu\\Debug\\VTS_Wasser\\out_trans_1.vts'] )

out_trans_1_vts.PointArrayStatus = []
out_trans_1_vts.CellArrayStatus = ['f', 'p', 'bac', 'vf', 'ijkmob', 'v']
----------------------------------------------------------------------------------
CellDatatoPointData1 = CellDatatoPointData()
----------------------------------------------------------------------------------
CellDatatoPointData1.PassCellData = 1
----------------------------------------------------------------------------------
Contour1.Isosurfaces = [0.5]
Contour1.ContourBy = ['POINTS', 'f']
----------------------------------------------------------------------------------
DataRepresentation3.ColorArrayName = ('CELL_DATA', 'v')
DataRepresentation3.LookupTable = a3_v_PVLookupTable
DataRepresentation3.ColorAttributeType = 'CELL_DATA'
```

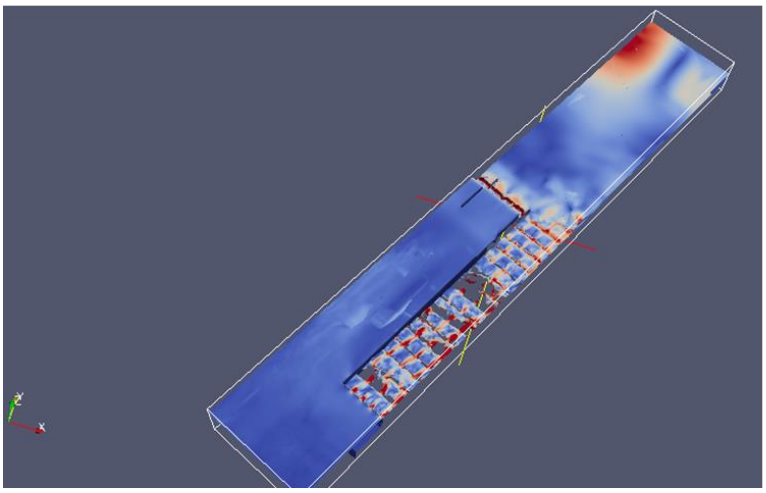


Abb. 5 Ausschnitte eines *Python script* aus ParaView (oben) und dessen Ausgabe (unten)

Schließlich wurde ParaView in dieser Arbeit auch zur Analyse von Datensätzen und dessen Darstellung zur Erleichterung des Verständnisses einzelner VTK-Klassen verwendet. Dabei ist die Markierung und Extraktion einzelner Zellen und Punkte, und die Anzeige von Attributwerten besonders wichtig. Mit dem Filter „Extract Subset“ können bestimmte Zellen aus den Eingangsdaten herausgenommen werden. Mit „Select Cells On (s)“ oder „Select Points On (d)“ können Zellen bzw. Punkte markiert werden. Die Attributwerte können in Tabellen angezeigt werden: Diese werden durch Hinzufügen eines „Layout“ als „Spreadsheet View“ wiedergegeben. Markierte Elemente der 3D Ansicht erscheinen in den Tabellen blau hervorgehoben.

6 Funktionalitäten aus dem Programm Blender: Fertigstellung der Daten, die in CFDVis/VisualisierungCFD eingelesen werden

Bevor die in dieser Arbeit implementierte Anwendung dokumentiert wird (Abschnitt 7), erfolgt in diesem Abschnitt eine Beschreibung einzelner Vorgehensweisen, wie mit dem Programm Blender, die Eingangsdaten im STL- und OBJ-Format fertiggestellt wurden.

Blender ist eine open-source Software zur Erstellung von 3D-Modellen mit Testurierung, Beleuchtung und Visualisierung. Die 3D-Szenen können gerendert und animiert werden. Das Programm unterliegt einer GNU-General-Public-Lizenz (Stiller 2011, 9). Hier wurde die Version Blender 2.5 verwendet.

Die statischen Modellkomponenten der implementierten Anwendung wurden aus Blender exportiert, um dann in der Anwendung eingelesen zu werden.

Dabei handelt es sich um das Kraftwerksmodell und die Umgebungselemente, das digitale Geländemodell und die Bäume.

Navigation in Blender

Zunächst werden kurz einzelne Navigationsmöglichkeiten in Blender aufgelistet.

Mit dem Mausrad wird in dem 3D View-Fenster gezoomt.

Zum Markieren wird die rechte Maustaste verwendet. Mit der Taste „a“ können in dem *3D View-Fenster* alle Objekte markiert oder die Markierung für alle Objekte wieder aufgehoben werden.

Durch Drücken des Mausrades kann die Ansicht verdreht werden.

Mit *Shift* + Drücken des Mausrades kann die Ansicht verschoben werden.

Je nachdem in welchem Ansichtsfenster die Maus platziert ist wirken sich Befehle unterschiedlich aus.

Mit *Shift* + Rechtsklick werden in dem *3D View-Fenster* mehrere Objekte markiert. In dem *Outliner* werden mehrere Objekte durch *Shift* + Linksklick markiert.

Geländefertigstellung

Das Geländemodell liegt im STL-Format vor und kann somit direkt in der Implementierung eingelesen werden. In dieser Arbeit wurde jedoch eine, dem eingebauten Kraftwerk angepasste Version des Geländes eingelesen. Auch die relativen Koordinaten wurden in Blender eingestellt.

Das Gelände kann mit „File->Import->Stl“ in Blender geladen werden. Allerdings liegt das Objekt mit 7-stelligen x- und y-Koordinaten weit vom Szenennullpunkt entfernt. Mit „View->Align View->View Selected“ kann die Ansicht auf das Gelände gerichtet werden. In dem „3D View-Fenster“ wird das Navigationspanel mit der Taste „n“ eingeblendet. Bei „Transform->Location“ werden die Koordinaten des Objektes eingegeben. Im *Object Mode* (vom Object Mode (gesamtes Objekt) zum Edit Mode (Gitter) kann im 3D View-Fenster mit der Tab-Taste gewechselt werden) sind das die Koordinaten des *Object Origin*. Dieser wird mit dem Objekt vom Nullpunkt weg verschoben, damit das Gelände näher am Szenennullpunkt liegt. Wird die Maus in dem 3D View-Fenster platziert, und ist das Gelände im *Object Mode* markiert, kann mit „Strg+A->Apply->Location“ der *Object Origin* zum Szenennullpunkt verschoben werden, ohne die Position des Geländes weiter zu beeinflussen. (Objektdrehungen passieren um den *Object Origin* und im *Edit Mode* werden die lokalen Koordinaten relativ zu diesem Punkt angegeben.)

Hilfreiche Werkzeuge zur Geländeveränderung für das Einbinden des Kraftwerkes werden im Folgenden aufgeführt.

Wenn das Gelände markiert ist, kann in dem *3D View-Fenster* in den *Edit Mode* gewechselt werden (*Tab-Taste*). Einzelne Punkte können markiert werden um dessen Koordinaten in dem Navigationspanel bei „Transform->Vertex“ zu verändern. Händisch werden markierte Punkte mit der Taste „g“ verschoben. Die Verschiebung wird in diesem Fall durch die Mausbewegung vorgegeben und mit Linksklick beendet.

Ein Ausschnitt kann aus dem Gelände zu einem unabhängigen Objekt extrahiert werden. Dazu wird eine Fläche (oder Punktmenge) des Geländes im *Edit Mode* markiert. Die Maus wird im *3D View-Fenster* platziert. Durch „Leertaste->Separate->Selection“ erhält man die markierte Fläche als ein neues Objekt. Das Gelände weist an dieser Stelle eine Aussparung auf.

Das veränderte und platzierte Gelände kann nun exportiert werden, um in der Implementierung eingelesen werden zu können. Vor dem Export ist es wichtig

sicherzustellen, dass in dem Navigationspanel *Location=(0,0,0)*, *Rotation=(0,0,0)* und *Scale=(1,1,1)* sind um zu gewährleisten, dass nach dem Einlesen der Objekte, dessen relative Anordnung richtig ist. Wie für Location wird diese Einstellung mit „Strg+A->Apply->Rotation“ und „Strg+A->Apply->Scale“ durchgeführt.

Ist das Gelände markiert, wird es mit „File->Export->Stl“ als STL-Datei an dem angegebenen Ort gespeichert. Wenn mehrere Objekte markiert sind, werden diese als eine STL-Datei exportiert (wiki.blender.org 2014 a).

Fertigstellung des Kraftwerksmodelles

Wie das Gelände werden die Kraftwerkskomponenten als STL-Dateien in Blender importiert und platziert. Wenn Skalierungen oder Translationen notwendig sind können die Schaltflächen links des *3D View-Fensters* (mit „t“ einblendbar) unter „Object Tools->Transform“ verwendet werden.

Für den Export ist zu bedenken, dass jede erzeugte STL-Datei, nach dem Einlesen in der implementierten Anwendung einem Mapper und einem Actor als Einheit übergeben wird. Dadurch werden alle Objekte, die in einer Datei exportiert wurden auf gleiche Weise texturiert. In der für diese Arbeit implementierten Anwendungsversion wurde das Kraftwerk als ein Objekt gehandhabt.

Erzeugung der Bäume

Die Bäume wurden mit dem *Add-On* „Add Curve: Sapling“ erzeugt (wiki.blender.org 2014 b). Um eine sinnvolle Texturierung zu ermöglichen wurden alle Stamm- und Aststrukturen als eine Datei exportiert, alle Blätter als eine zweite. Um die Ladezeit der Blattstrukturen zu optimieren (Abschnitt 7) wurden die Blätter als OBJ-Datei exportiert. Dazu werden folgende Einstellungen verwendet.

Alle Objekte, die als eine Datei zu exportieren sind, werden markiert. Mit „File->Export->Wavefront (.obj)“ öffnet sich der *File Browser*. Auf der linken Seite werden bei „Export OBJ“ Einstellungsoptionen zum Export angezeigt. „Selection Only“ wird gewählt (nur die markierten Objekte werden exportiert), „Materials“ nicht. Für „Forward“ und „Up“ müssen die Koordinatenrichtungen gewählt werden, sodass diese in der Anwendung richtig angezeigt werden. Mit „Export OBJ“ wird die Datei erzeugt (wiki.blender.org 2014 c).

7 Programmbeschreibung des implementierten Modells CFDVis/VisualisierungCFD

Das implementierte Modell besteht zunächst aus einem statischen Teil zur Modellierung des umgebenden Flussabschnittes. Dazu zählen die Darstellung des Geländemodells mit Einbau von Umgebungselementen zur realistischen Wiedergabe (z.B. Bäumen) und die Visualisierung des Wasserkraftwerkes (Abb. 6).

Die Wasseroberfläche wird anhand simulierter Parameter erstellt und in Abhängigkeit der berechneten Geschwindigkeit texturiert. Durch Nutzung zeitlich unterschiedlicher CFD-Datensätze entstehen unterschiedliche Wasseroberflächen, die in der Echtzeit-Visualisierung ausgetauscht werden.

Abb. 6 Renderansicht / Ausgabe von CFDVis/VisualisierungCFD. Implementierte Modellelemente

Die Ausgabe des Modells bzw. die Renderansicht erfolgt, wie in Abb. 6 sichtbar, in einem vtkRenderWindow (Abschnitt 3.3). Zudem können Programminformationen zur Darstellung und zum Programmverlauf der Textausgabe entnommen werden (Abb. 7).

Abb. 7 Beispielansicht der Textausgabe des Programmes CFDVis/VisualisierungCFD

Im Folgenden wird das Programm im Detail beschrieben. Tabelle 1 ist eine Zusammenstellung der Interaktionen, die in CFDVis/VisualisierungCFD implementiert wurden und soll eine Übersicht zur Nutzung des Programms darstellen. Zusätzlich zu diesen Möglichkeiten der Interaktion wurden in VTK implementierte Funktionalitäten verwendet (Abschnitt 12.2), die Tabelle 2 entnommen werden können.

```
//.............................
	//BEREICH ZUR EINGABE DER FÜR DIE VISUALISIERUNG VERWENDETEN DATEN UND
//EINSTELLUNGEN:


	//Eingabe der Namen der Dateien die einzulesen sind.
	//Diese Dateien sind alle in dem "Debug"-Ordner zu speichern.
	std::string inputBaumStaemmeFilename = "Baumstaemme.stl";

	std::string inputBaumBlaetterFilename = "Baumblaetter.obj";

	std::string inputKraftwerkFilename = "Kraftwerk.stl";

		//Gelände-STL-Datei mit zugehörigem Luftbild im TIFF-Format:
	std::string inputGelaendeFilename = "GelaendeMitAusschnitt.stl";
	std::string inputLuftbildFilename = "Luftbild.tif";

		//STL-Geländeausschnitt mit zugehöriger PNG-Textur:
	std::string inputGelaendeAusschnittFilename = "GelaendeOW.stl";
	std::string inputBildOWFilename = "BildGelaendeOW.png";

		//Ordner, der die CFD-Daten (VTS-Dateien) beinhaltet:
	std::string CFDOrdnerName = "VTS_Wasser";
```

```
	//Eingabe des Zeitschrittes zur Veränderung der Wasseroberfläche durch den
//Timer.
	//Hier wird die Zeit in Millisekunden erwartet, in der eine Wasseroberfläche
//sichtbar ist (bis die nächste Wasseroberfläche angezeigt wird).
	unsigned long timerZeitschritt = 2000;

//.............................
```

Abb. 8 VisualisierungCFD.cpp: Bereich zur Eingabe der für das Modell zu verwendenden Daten

Statische Bestandteile

In dem implementierten Programm CFDVis/VisualisierungCFD.cpp wird das Geländemodell im STL-Format eingelesen. Die Texturierung erfolgt durch ein Luftbild welches zu dem Geländeausschnitt passt. Das heißt, dass die Ränder des Bildes mit den Rändern des Geländes bei der Texturierung übereinstimmen. Diese relative Anordnung von Gelände und Luftbild wird für die Berechnung der Texturkoordinaten vorausgesetzt (Abschnitt 11). Ohne weitere Veränderung des Codes können damit das Gelände und das zugehörige Texturbild ausgetauscht werden solange die Ränder zusammen passen, das Gelände im STL-Format und das Bild im TIF-Format vorliegen.

Sind diese Eigenschaften weiterhin vorhanden, erfolgt der Austausch des texturierten Geländes ausschließlich durch Eingabe der neuen Dateinamen für Gelände und Bild in dem Bereich „`BEREICH ZUR EINGABE DER FÜR DIE VISUALISIERUNG VERWENDETEN DATEN UND EINSTELLUNGEN:`“ am Anfang der main-Funktion in VisualisierungCFD.cpp (Abb. 8). Für das Gelände wird der Name der STL-Datei der Variable `inputGelaendeFilename` übergeben. Die Variable `inputLuftbildFilename` dient der Eingabe des TIF-Datei-Namens der Geländetextur. Alle eingelesenen Dateien werden in dem „Debug“-Ordner gespeichert. Dieser Ordner ist auf der beiliegenden DVD „CFDVisSource“ unter CFDVisSource\CFDVis\Debug zu finden. Auf der beiliegenden DVD „CFDVisVersion1.0“ entspricht der „Debug“-Ordner dem Ordner „CFDVisVersion1.0“.

Da der Standort anstatt des Kraftwerkes zur Zeit des Luftbildes noch eine andere Staustufe aufweist, die gegenüber der geplanten Kraftwerksposition verschoben ist, musste die Texturierung an der Stelle der vorhandenen Staustufe verändert werden. Das wurde gemacht, indem der Geländeausschnitt mit der Staustufe von dem Rest des Geländes mit dem Programm Blender (Abschnitt 6) getrennt wurde. In dem zuvor eingelesenen Gelände fehlt dieser Ausschnitt. Das Geländestück wird einzeln mit der Variable `inputGelaendeAusschnittFilename` (Abb. 8) eingelesen. Die

Texturierung wurde mit einem passenden Ausschnitt des Luftbildes als PNG-Datei durchgeführt. Die Texturkoordinaten werden wie zuvor für das Gelände berechnet. Allerdings ist die Wassertextur des Ausschnittes im Gegensatz zu dem Gesamtluftbild relativ einheitlich, bzw. muss nicht an eine definierte Stelle des Objektes projiziert werden. Daher ist die Übereinstimmung der Ränder nicht gegeben und nicht notwendig. Die Berechnung gewährleistet eine vollständige Texturierung der Geländeausschnittoberfläche. Die Bildbereiche die über den Ausschnitt hinausgehen werden nicht dargestellt. Zum Ersetzen des PNG-Bildes kann, ohne weitere Veränderung, der Name der neuen PNG-Datei `inputBildOWFilename` übergeben werden (Abb. 8).

Für das Einlesen der Bäume muss darauf geachtet werden, dass die Koordinaten relativ zum verwendeten Gelände stimmen. Dies kann durch Verschieben bzw. Platzieren der Bäume mit VTK gemacht werden (Abschnitt 3.3). Bei der Implementierung dieser Anwendung wurde, zur relativen Platzierung der Modellelemente, das Programm Blender verwendet (Abschnitt 6). Damit ist ein Einlesen der Objekte ohne weitere Platzierung der Elemente einfach möglich. Da die Bäume aus Blender generiert sind (Abschnitt 6) erscheint diese Arbeitsweise hier sinnvoll. Bei Verwendung eines neuen Geländemodells können die Bäume mit Blender neu positioniert werden. Alle Stämme sind als eine STL-Datei aus Blender exportiert und in VisualisierungCFD.cpp mit Übergabe des Dateinamens an `inputBaumStaemmeFilename` eingelesen. Alle Blattstrukturen werden als eine OBJ-Datei eingelesen, wobei der Dateiname in der Variable `inputBaumBlaetterFilename` gespeichert ist (Abb. 8).

Das Kraftwerk wird als STL-Datei eingelesen. Die Koordinaten wurden hier wieder in Blender relativ zu dem Gelände richtig eingestellt. Der Dateiname für das Kraftwerk wird als `inputKraftwerkFilename` eingegeben (Abb. 8).

Wenn die, in diesem Abschnitt beschriebenen, zu berücksichtigenden Eigenschaften der Implementierung für neue Daten zustimmen (relative Koordinaten und Datenformat), können Eingangsdaten ausgetauscht werden, indem die Dateinamen am Anfang der main-Funktion geändert werden (Abb. 8), und die neuen Dateien wie beschrieben in dem „Debug“-Ordner gespeichert werden.

CFD-Ergebnisse

Die CFD-Ergebnisse liegen als VTS-Dateien vor. Jede dieser Dateien wird eingelesen. Dann wird für f=0,5 die Wasseroberfläche aus den CFD-Daten generiert (Abschnitt 8). Die Texturierung der Wasseroberfläche wird durch die Geschwindigkeitswerte vorgegeben (Abschnitt 10). Für jede VTS-Datei wird auf die Weise eine texturierte Wasseroberfläche erzeugt. Zur Laufzeit des Programmes werden die Oberflächen ausgetauscht, sodass eine Echtzeitveränderung der CFD-Daten zustande kommt. Dazu wird mit der Beschreibung der Interaktion näheres erläutert.

Zur Darstellung neuer CFD-Daten oder zur Veränderung der Anzahl der eingelesenen Datensätze müssen die zu berücksichtigenden VTS-Dateien in dem Ordner DVD „CFDVisSource“->CFDVisSource\CFDVis\Debug\VTS_Wasser gespeichert sein. (Wie schon erwähnt entspricht der Ordner DVD “CFDVisVersion1.0“->CFDVisVersion1.0 dem „Debug“-Ordner.) Das Programm stellt alle VTS-Dateien dieses Ordners dar. Die Anzahl der Dateien ist nicht festgelegt und wird anhand der Anzahl an VTS-Dateien in dem Ordner von dem Programm festgestellt. Aus dem Ordner werden zudem nur die VTS-Dateien berücksichtigt. Daraus folgt, dass in dem Ordner „VTS_Wasser“ alle VTS-Daten die darzustellen sind gespeichert werden müssen, und nur diese VTS-Dateien darin vorhanden sein dürfen. Dateien ohne VTS-Endung werden nicht berücksichtigt und können damit in dem Ordner vorhanden sein.

Alternativ kann auch ein neuer Ordner mit den VTS-Daten angelegt werden. In diesem Fall ist dieser auch in dem „Debug“-Ordner abzulegen. Der Name des Ordners muss am Anfang der main-Funktion in der Variable `CFDOrdnerName` geändert werden (Abb. 8).

Echtzeit-Visualisierung und Benutzer-Interaktion

Durch einen „Timer“ wird die Wasseroberfläche in regelmäßigen Zeitschritten ausgetauscht. Die Wasseroberflächen sind in der Reihenfolge der VTS-Dateien in dem Ordner indexiert. Der „Timer“ wechselt immer zu der Oberfläche mit dem aktuellen Index + 1. Wenn die letzte Wasseroberfläche dargestellt wurde wechselt der „Timer“ wieder zu der ersten. Vor dem Austausch jeder Wasseroberfläche wird in der Textausgabe der Index aus [1;Anzahl der Datensätze] angezeigt. Der Zeitschritt ist auf 2 Sekunden eingestellt. Dieser kann am Anfang der main-Funktion in der Variable `timerZeitschritt` beliebig verändert werden (Abb. 8). Es

muss jedoch darauf geachtet werden, dass der Rendervorgang in der gewählten Zeit abgeschlossen werden kann.

Mit der Taste „n“ wird der „Timer“ angehalten. Die Oberfläche wird nicht mehr ausgetauscht bis der „Timer“ mit der Taste „t“ wieder aktiviert wird. Während der Pause des „Timer“ können die anderen Interaktionen genutzt und ausgeführt werden.

„z“ wechselt zu der vorherigen Wasseroberfläche mit dem aktuellen Index - 1. Wenn die aktuelle Oberfläche die erste ist, wechselt „z“ zu dem letzten Datensatz. Vor der Darstellung wird der Index, wie bei dem „Timer“, in Textform ausgegeben.

„v“ wechselt zu der Wasseroberfläche mit dem aktuellen Index + 1. Wenn die aktuelle Oberfläche die letzte ist, wird der erste Datensatz dargestellt. Die Textausgabe gibt den Index der im Anschluss dargestellten Oberfläche wieder.

In der ersten Renderansicht, bevor die Eventschleife gestartet wird, sind die Bäume nur durch Stamm und Äste dargestellt. Die Blätter sind an dieser Stelle noch nicht eingelesen. Tatsächlich beansprucht der Ladevorgang der Blätter relativ viel Zeit. Deswegen ist die Darstellung der Blätter optional durch drücken der Taste „b“ möglich. Während der Ausführung dieses Events können keine anderen Interaktionen genutzt werden. Nach dem einmaligen Laden der Blattstruktur sind im weiteren Programmverlauf keine zeitlichen Verzögerungen bemerkbar. Durch die Wahl des geladenen Datenformates der Blätter kann die Zeit, die zum Einlesen benötigt wird, signifikant verringert werden. Das Rendern der Blätter konnte von 2 Minuten und 45 Sekunden im STL-Format auf ungefähr 47 Sekunden als OBJ-Datei reduziert werden. (Wie in vtkSTLReader.h dokumentiert, sind STL-Dateien relativ ineffizient, da Knotendefinitionen dupliziert werden.)

Durch drücken der Taste „c“ wird die Ansicht verändert. Die Kamera dreht um 30 Grad, auf einer horizontalen Ebene, um den Normalenvektor dieser Ebene in dem *focal point* der Kamera. Die in der Implementierung verwendete Methode der Klasse vtkCamera `Azimuth()` bewirkt tatsächlich eine Drehung der Kamera um dessen „view up vector“ mit dem „focal point“ als Anfangspunkt (Abb. 33) (Schroeder, Martin und Lorensen 2006, 44).

Mit „a“ wird die Kamera so eingestellt, dass die Ansicht auf den Mittelpunkt der *bounding box* der ersten eingelesenen Wasseroberfläche gerichtet wird. Damit ist ein Ausgangszustand der Kamera gegeben, der zu dem dargestellten Modell zurückführt. Auch bei Veränderung der Eingangsdaten kann mit „a“ die Kamera auf die neuen Objekte gerichtet werden. Es wird dabei angenommen, dass die Oberseite

des Geländes in positive z-Richtung zeigt. Die Anfangseinstellungen der Kamera können, mithilfe der Textausgabe der Koordinaten des Mittelpunktes der *bounding box* der ersten Oberfläche, den neuen Eingangsdaten angepasst werden.

Mit „o“ führt die Kamera eine Parallelprojektion durch.

„8“ ergibt eine perspektivische Ansicht (siehe vtkCamera.h).

Bei der Darstellung mit implementierten Kameraeinstellungen (Anfangsansicht und mit „a“) wird die Ausgabe der Ansicht nicht ganz korrekt angezeigt. Z.B. durch minimale Verdrehung mit linker Maustaste wird die Ansicht richtig ausgegeben.

Schließlich wird noch kurz auf die Lichteinstellungen bei Veränderung der Inputdaten eingegangen. Eine Möglichkeit für eine erste Darstellung, bevor die Lichter explizit neu platziert werden, ist die Auskommentierung der Zeilen `renderer->AutomaticLightCreationOff()` und `renderer->AddLight()`, die das Hinzufügen der Lichter in der Szene implementieren. Dadurch werden Lichter automatisch erzeugt. Alternativ kann z.B. die Ausgabe aus „a“ dazu dienen, die Verschiebung der neuen gegenüber den ursprünglichen Wasseroberflächen zu bestimmen. Die Lichter können dann um diesen Betrag verschoben werden. Für jedes Modell ist es jedoch sinnvoll die Lichter zu optimieren, da diese einen relativ großen Einfluss auf das Aussehen der Texturierung der Objekte haben.

Die in diesem Abschnitt beschriebenen Interaktionen sind in Tabelle 1 in verkürzter Form zusammengefasst.

Tabelle 1 In CFDVis/VisualisierungCFD implementierte Benutzer-Interaktionen (zusätzlich zu den aus VTK verwendeten Interaktionen -> **Tabelle 2**)

Interaktion	**Auswirkung in dem *render window***	**Textausgabe**
Timer	Wechsel der Wasseroberflächen / der CFD-Datensätze in definierten Zeitschritten (index=index+1)	Ausgabe des Index der im Anschluss dargestellten Oberfläche [1; Anzahl der Datensätze]
„n“	Anhalten des „Timer“. Die Wasseroberfläche wird nicht ausgetauscht, bis der „Timer“ mit „t“ wieder aktiviert wird	Bestätigung der Pause und Hinweis auf die Taste „t“ zum aktivieren des „Timer“
„t“	Wenn der „Timer“ mit „n“ deaktiviert wurde, wird mit „t“ der Oberflächenwechsel fortgeführt. Sonst keine Wirkung	keine
„z“	Darstellung der vorherigen Wasseroberfläche/des vorherigen CFD-Datensatzes (index=index-1)	Ausgabe des Index der im Anschluss dargestellten Oberfläche [1; Anzahl der Datensätze]
„v“	Darstellung der folgenden Wasseroberfläche/des folgenden CFD-Datensatzes (index=index+1)	Ausgabe des Index der im Anschluss dargestellten Oberfläche [1; Anzahl der Datensätze]
„b“	Rendern der Blätter für alle vorhandenen Bäume. (Das Laden der Blätter nimmt Zeit in Anspruch. Während dessen können andere Interaktionen nicht ausgeführt werden. Nach	Hinweis auf Wartezeit durch den Ladevorgang

	dem einmaligen Laden sind keine zeitlichen Verzögerungen bemerkbar.)	
„c“	Drehung der Kamera um 30 Grad, auf einer horizontalen Ebene, um den Normalenvektor dieser Ebene in dem *focal point* der Kamera	keine
„a“	Einstellung der Kamera, sodass die Ansicht auf den Mittelpunkt der *bounding box* der ersten eingelesenen Wasseroberfläche gerichtet wird	Ausgabe der Koordinaten des Mittelpunktes der *bounding box* der ersten eingelesenen Wasseroberfläche
„o“	Parallelprojektion	keine
„8“	Perspektivische Projektion	keine

In den Abschnitten 8 bis 12 werden die verwendeten und implementierten Funktionalitäten im Detail erläutert. Dabei wird besonders auf Themen eingegangen, bei denen eine Ergänzung der Dokumentation zur Implementierung der Anwendung notwendig war.

Abschnitt 8 geht zunächst auf das Einlesen der CFD-Daten und die Erzeugung der Wasseroberflächen ein.

8 Einlesen der VTS-Daten und Erzeugung der Wasseroberflächen mit VTK

8.1 VTS-Daten als vtkStructuredGrid

VTS-Daten (mit der Endung .vts) können mit VTK durch die Klasse vtkXMLStructuredGridReader als vtkStructuredGrid eingelesen werden. Ein Objekt der Klasse vtkStructuredGrid besteht aus Punkten (In Abb. 9 sind diese weiß dargestellt.), die topologisch regelmäßig in Zellen eingeteilt sind (in Abb. 9 als blaues Gitter sichtbar) (Schroeder, Martin und Lorensen 2006, 144). Den Zellen und/oder Punkten können Attributdaten zugewiesen sein. In Abb. 9 sind die Zellwerte f als Färbung der Zellen sichtbar gemacht.

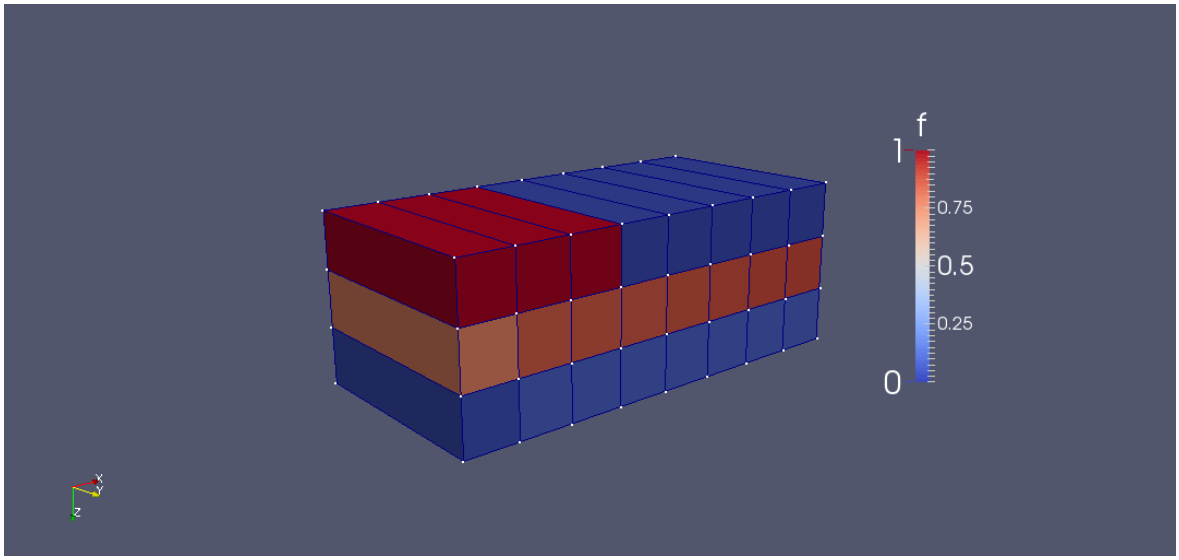


Abb. 9 vtkStructuredGrid Darstellung mit ParaView

8.2 VTS-Daten-Ausgabe

Mit einem Textverarbeitungsprogramm kann die VTS-Datei geöffnet werden, um die zugewiesenen Attributdaten zu kennen. Abb. 10 zeigt den Anfang der Ausgabe einer StructuredGrid ohne Attributdaten als VTS-Datei.

```
<?xml version="1.0"?>
<VTKFile type="StructuredGrid" version="0.1" byte_order="LittleEndian"
compressor="vtkZLibDataCompressor">
  <StructuredGrid WholeExtent="0 110 0 160 0 25">
    <Piece Extent="0 110 0 160 0 25">
      <PointData>
      </PointData>
      <CellData>
      </CellData>
      <Points>
        <DataArray type="Float32" Name="Points" NumberOfComponents="3"
format="appended" RangeMin="225384.6521"          RangeMax="226008.96227"
offset="0"                    />
      </Points>
    </Piece>
  </StructuredGrid>
```

Abb. 10 Textverarbeitungsprogramm-Anzeige einer VTS-Datei ohne Attributdaten

Zwischen <PointData> und </PointData> werden die Attributdaten der Punkte angegeben, zwischen <CellData> und </CellData> die der Zellen. Eine Datei mit den Datensätzen f und v als Punktattribute und als Zellattribute ist in Abb. 11 dargestellt.

```
<?xml version="1.0"?>
<VTKFile type="StructuredGrid" version="0.1"
byte_order="LittleEndian" compressor="vtkZLibDataCompressor">
  <StructuredGrid WholeExtent="0 110 0 160 0 25">
    <Piece Extent="0 110 0 160 0 25">
      <PointData>
        <DataArray type="Float32" Name="f" format="appended"
RangeMin="0"                    RangeMax="1"
offset="0"                   />
        <DataArray type="Float32" Name="v" NumberOfComponents="3"
format="appended" RangeMin="0"
RangeMax="12.555208646"         offset="77792"              />
      </PointData>
      <CellData>
        <DataArray type="Float32" Name="f" format="appended"
RangeMin="0"                    RangeMax="1"
offset="684360"              />
        <DataArray type="Float32" Name="v" NumberOfComponents="3"
format="appended" RangeMin="0"
RangeMax="13.060861912"         offset="717868"             />
      </CellData>
```

Abb. 11 Textverarbeitungsprogramm-Anzeige einer VTS-Datei mit Attributdaten

Die Angabe „NumberofComponents“ lässt darauf schließen, ob es sich wie bei der Geschwindigkeit v um drei Werte für jeden Punkt/jede Zelle handelt für die drei Raumrichtungen (Vektordaten), oder ob es je Punkt/Zelle einen Wert wie für f gibt (Skalarwert). „RangeMin“ und „RangeMax“ geben das Minimum und das Maximum der Werte des jeweiligen Datensatzes an.

8.3 CellData und PointData

8.3.1 Allgemeines Prinzip

Mit der VTK-Klasse vtkCellDataToPointData werden die *CellData* der VTS-Datei zu *PointData* konvertiert. Darunter ist zu verstehen, dass statt der vorhandenen Zellwerte, nach der Konvertierung Werte für die Eckpunkte der Zellen gegeben sind. Bei den eingelesenen Zelldaten war jeder Zelle genau ein Wert zu jedem Datensatz zugeordnet. Durch die Konvertierung wird für jeden Eckpunkt das arithmetische Mittel der Werte der angrenzenden Zellen berechnet. Damit entstehen, wie in Abb. 12 zu erkennen, für eine Zelle im dreidimensionalen Raum mit einem Wert, acht Punkte mit je einem Wert für jeden vorhandenen Datensatz.

CELLS f-Werte (Die Koordinaten sind beliebig, dienen an diese Stelle nur der Verständlichkeit)

z \ x	[0,1]	[1,2]	[2,3]	[3,4]	[4,5]	[5,6]	[6,7]	[7,8]
[2,3]	0,0213061	0,0648968	0,0648968	0,0648968	0,0648968	0,0648968	0,0648968	0,0648968
[1,2]	0,785265	0,866524	0,875034	0,881943	0,888852	0,895761	0,90267	0,909579
[0,1]	1	1	1	0	0	0	0	0

POINTS f-Werte (Die doppelte Eingabe je Wert steht für die zwei Punkte, die in y-Richtung, durch die Ausdehnung der Zelle in die Ebene hinein, entstehen)

z \ x	0	1	2	3	4	5	6	7	8
3	0,0213061	0,0431015	0,0648968	0,0648968	0,0648968	0,0648968	0,0648968	0,0648968	0,0648968
	0,0213061	0,0431015	0,0648968	0,0648968	0,0648968	0,0648968	0,0648968	0,0648968	0,0648968
2	0,403286	0,434498	0,467838	0,471693	0,475147	0,478602	0,482056	0,485511	0,487238
	0,403286	0,434498	0,467838	0,471693	0,475147	0,478602	0,482056	0,485511	0,487238
1	0,892632	0,912947	0,935389	0,689244	0,442699	0,446153	0,449608	0,453062	0,45479
	0,892632	0,912947	0,935389	0,689244	0,442699	0,446153	0,449608	0,453062	0,45479
0	1	1	1	0,5	0	0	0	0	0
	1	1	1	0,5	0	0	0	0	0

Abb. 12 Konvertierung von Zell- zu Punktdaten mit vtkCellDataToPointData

8.3.2 Genaue Implementierung in VTK

Um die Berechnung der Konvertierung der Zellattribute zu Punktattributen in VTK nachzuvollziehen, kann die Funktion RequestData() der Klasse vtkCellDataToPointData (vtkCellDataToPointData.cxx) untersucht werden. Die bei der Konvertierung einbezogenen Klassen und Methoden sind in Abb. 13 aufgeführt.

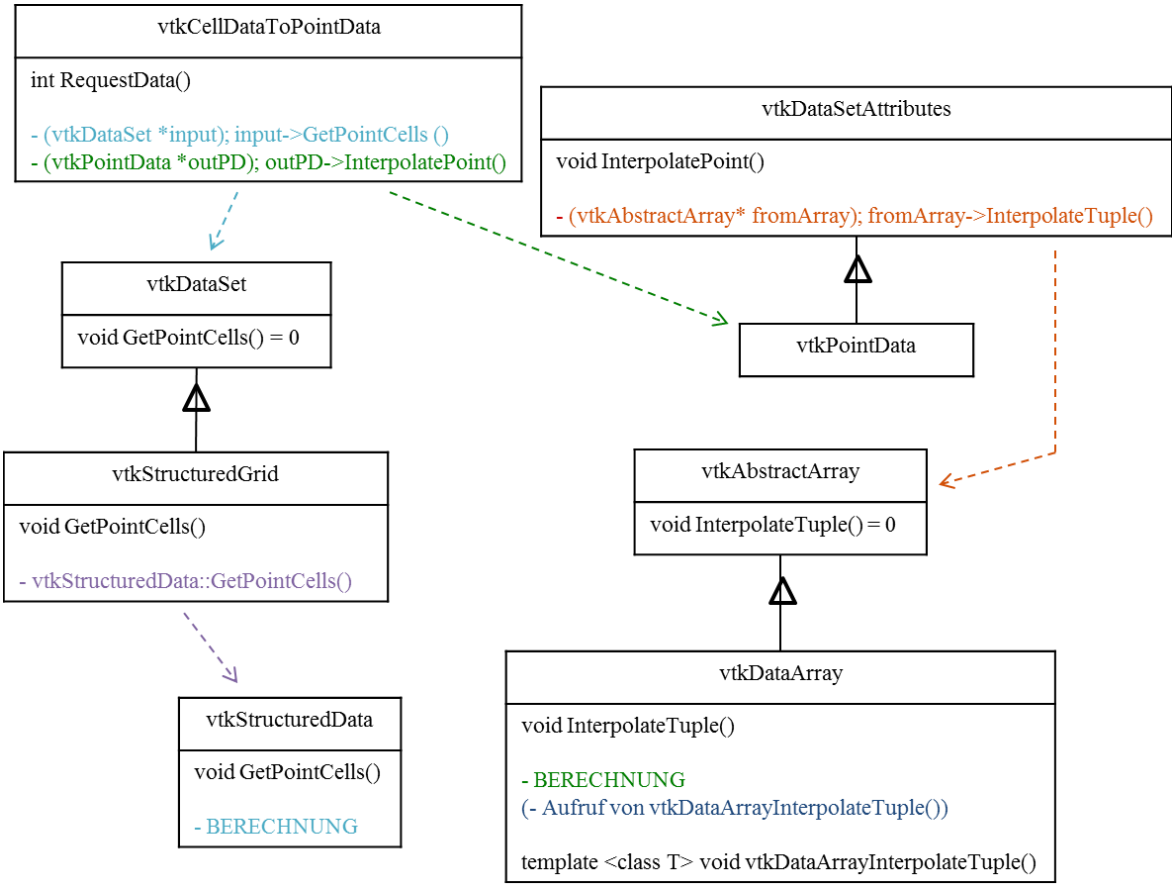


Abb. 13 Klassen, die bei der Berechnung der Punktattributdaten aus Zellattributdaten mit VTK einbezogen werden (Inhalte aus VTK Quellcode ermittelt). Mit farbiger Schrift sind einzelne Inhalte der darüber angegebenen Methoden wiedergegeben. Die gestrichelten Pfeile stellen die Einbeziehung verschiedener Klassen dar.

Durch die Aufrufe der Funktion RequestData() (in Abb. 13 hellblau und grün aufgeführt und mit gestrichelten Pfeilen nachverfolgt) und die daraus resultierenden Aufrufe weiterer Klassen (in Abb. 13 farbig und durch gestrichelte Pfeile dargestellt) werden die Methoden `vtkStructuredData::GetPointCells()` und `vtkDataArray::InterpolateTuple()` ausgeführt. Diese definieren die Berechnung der Punktattributdaten.

In `vtkStructuredData::GetPointCells()` werden für jeden Punkt die Zellindizes der maximal acht Zellen, die den Punkt berühren, in einer `vtkIdList *cellIds` gespeichert. Dabei muss beachtet werden, dass die `vtkIdType cellId` Einträge der `cellIds` fortlaufend nummeriert sind und von 0 bis zur (Anzahl der Zellen – 1) reichen.

In `vtkCellDataToPointData::RequestData()` werden daraus, für den gerade behandelten Punkt, für die Zellen dessen Index in `cellIds` gespeichert wurde, die Werte `double weight` zur Gewichtung der Zellen berechnet (Gleichung (1)).

$$weight = 1{,}0/numCells \qquad (1)$$
$$weights[cellId] = weight$$

Dabei bezeichnet numCells die Anzahl der Einträge in `cellIds`, d.h. die Anzahl der Zellen, die den bestimmten Punkt als Eckpunkt haben (maximal acht). In dem Array weights wird für alle angrenzenden Zellen (cellId von 0 bis numCells) der gleiche Wert weight gespeichert. Dieser Aspekt wird im Laufe dieses Abschnittes mit einem Beispiel näher betrachtet.

Schließlich wird mit der Methode `vtkDataArray::InterpolateTuple()`, für den bestimmten Punkt (mit Index ptId aus [0;(Gesamtanzahl der Punkte)[), der Attributwert c nach Gleichung (2) berechnet.

$$c[ptId \times numComp + k] = \sum_{j=0}^{numCells-1} weights[j] * inCellData[cellIds[j] * numComp + k] \qquad (2)$$

Das Array inCellData enthält die Zellattributdaten der Input-Zelldaten. Je nach Anzahl numComp der Werte pro Attributwert (Dimension der Attributdaten oder „NumberOfComponents“) hat inCellData (numComp*Anzahl der Zellen) Elemente. Gleichung (2) wird für die Variable k aus [0;numComp[durchgeführt.

Mit vtkCellDataToPointData werden auf die Weise für jeden Punkt eines `vtkDataSet *input` wie z.B. eines vtkStructuredGrid Objektes, die Punktattributdaten (Anzahl der berechneten Werte je nachdem, ob die Attributwerte Skalare oder Vektoren sind) als arithmetisches Mittel der vorhandenen Zellattributwerte berechnet.

Abb. 14 zeigt ein Berechnungsbeispiel für ein Objekt (VTS-Datensatz) das aus acht Zellen besteht.

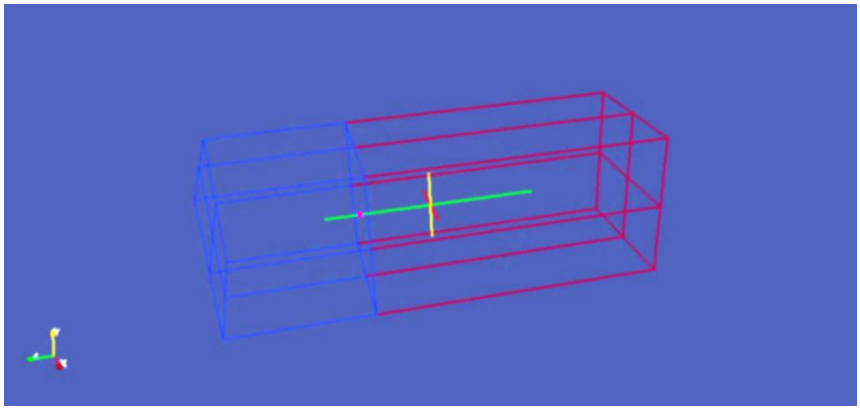


Abb. 14 Wireframe-Ansicht der VTS-Daten für die Beispielberechnung aus ParaView

Die Berechnung der Punktattributwerte wird für den in Abb. 14 innerhalb des Volumens markierten Punkt durchgeführt. Gegeben sind die Zellattributwerte f und p (siehe Abb. 15).

owing CellDatatoPointData1 Attribute: Cell Data

Cell ID	f	p
0	1	5034.08
1	1	5034.08
2	1	5034.08
3	1	5034.08
4	0.5	-9680.92
5	0.5	-9680.92
6	0.5	-9680.92
7	0.5	-9680.92

wing CellDatatoPointData1 Attribute: Point Data

Point ID	f	p
8	1	5034.08
9	0.75	-2323.42
10	0.75	-2323.42
11	0.75	-2323.42
12	0.75	-2323.42
13	0.75	-2323.42
14	0.75	-2323.42

Abb. 15 Ausgabe der Zell- (links) und der Punktattributwerte (rechts) in ParaView nach Berechnung der Punktattributwerte aus den Zellattributwerten

Der in Abb. 14 markierte Punkt (ID 13 in Abb. 15) dient den acht gegebenen Zellen als Eckpunkt. Damit gilt numCells=8 und weight=1,0/8.

Die Attribute f und p sind Skalare, d.h. numComp=1 und k=0.

Für f ergibt das arithmetische Mittel: c[13]=0,75.

Analog ist für den Zellwert p: c[13]=-2323,42 (Abb. 15), obwohl die Zellen 0 bis 3 größer sind als die Zellen 4 bis 7. Die Größe der Zellen wird nicht berücksichtigt, da

die Gewichtung aller an einem Punkt angrenzender Zellen „weight“ gleich gesetzt wird (Gleichung (1)).

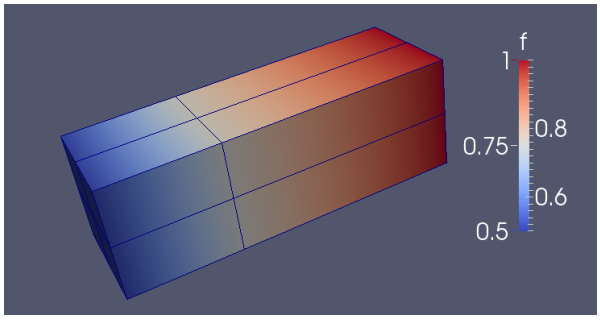


Abb. 16 Färbung der VTS-Daten in ParaView durch f-Punktattributwerte, die aus den Zellwerten berechnet wurden. (Darstellung der Zellgrenzen durch blaue Striche)

In Abb. 16 wird sichtbar, dass der Wert 0,75 in der Ebene der Zellgrenzen liegt. Für bestimmte Anwendungen könnte eine Verschiebung mittig zwischen die Zellmittelpunkte (durch unterschiedliche Gewichtung der Zellen) sinnvoll sein.

Für das in dieser Arbeit erstellte Modell waren Zellen gleicher Größe vorhanden, sodass ein arithmetisches Mittel die richtige Gewichtung darstellt.

In dem nächsten Abschnitt sollen Eigenschaften und Unterschiede der Nutzung von Punkt- und Zelldaten ohne Berücksichtigung der Zellgröße betrachtet werden.

8.3.3 Zu beachtende Eigenschaften des Filters _ Charakteristiken der Berechnungsmethode

Dieser Schritt der Datenkonvertierung ist sehr wichtig um VTS-Daten bearbeiten zu können, da für manche Filter, bei der Angabe des zu bearbeitenden Input-Datensatzes, die Information Cell- oder PointData angegeben werden muss. Bestimmte Klassen können nur mit der Übergabe von einem dieser Datensätze verwendet werden.

Aus dem oben beschriebenen Prinzip ist auch ersichtlich, dass die Darstellung je nach Konvertierung unterschiedlich ist: Eine Färbung einer Oberfläche mit den Punktdaten erscheint gegenüber einer Darstellung der Zelldaten geglättet (Abb. 17).

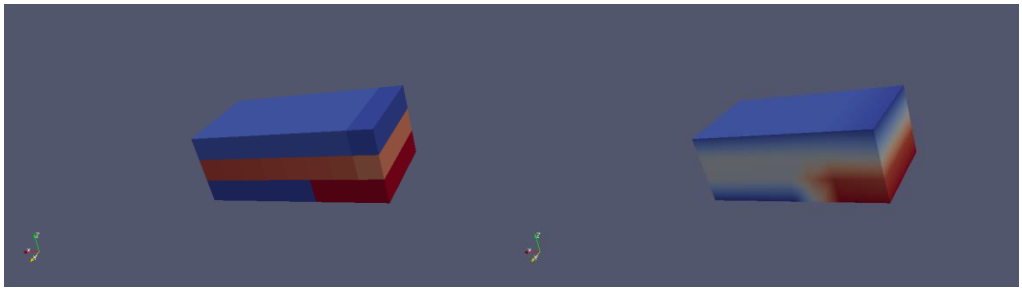


Abb. 17 Darstellung der VTS-CellData f (links) und der VTS-PointData f (rechts) als Färbung in ParaView / Auswirkung der Konvertierung mit vtkCellDataToPointData

Auf die Erstellung des Farbverlaufes wird in Abschnitt 10 im Detail eingegangen. Gleichzeitig mit der Glättung können durch die Nutzung von Punktattributdaten auch Farbmuster entstehen (z.B. gekurvte Farbgrenzen), die nicht immer erwünscht sind. Bei großer Auflösung der Zellen kann, zur Farbdarstellung, der Zellattributdatensatz für die Farbgebung geeignet sein (Abb. 18).

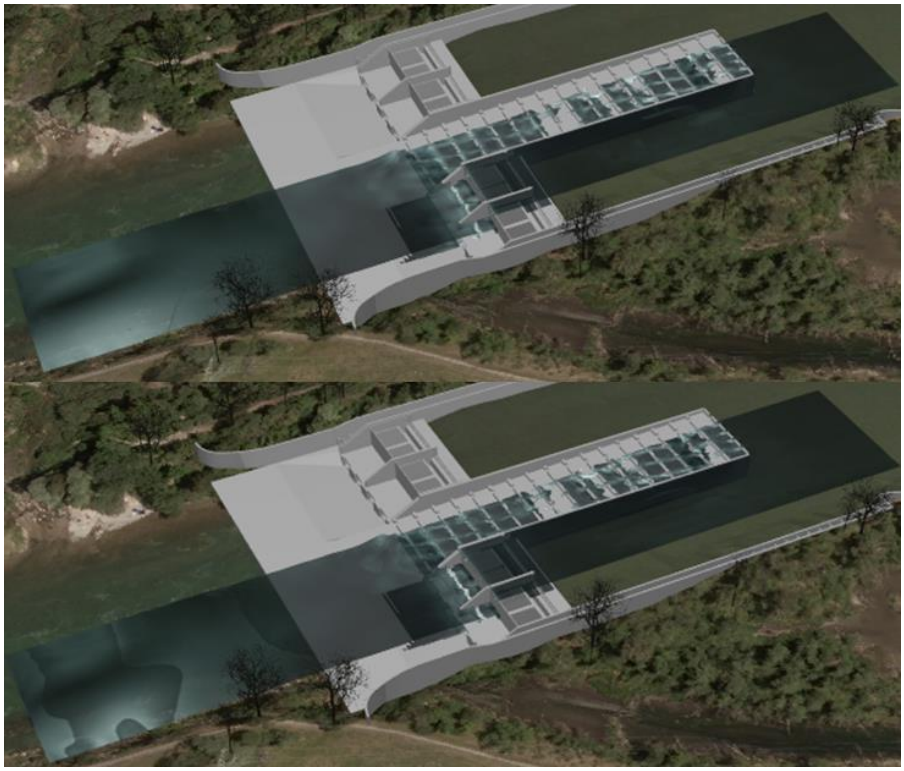


Abb. 18 Färbung einer Oberfläche durch Geschwindigkeitsattributdaten: oben mit Zelldaten, unten mit Punktdaten / Renderansicht zu einer Implementierung mit VTK

Je nachdem für welche Anwendung die Attributdaten verwendet werden ist zu entscheiden, ob auf die beschriebene Weise erzeugte Punktwerte oder vorliegende Input-Zellattributwerte besser geeignet sind.

8.4 vtkContourFilter zur Erzeugung der Wasseroberfläche

Ein Objekt der Klasse vtkContourFilter erzeugt, aus einem Input mit dreidimensionalen Zellen, als Output Isosurfaces. Dazu werden die zuvor beschriebenen Attributdaten interpoliert, um bei einem definierten Wert eine Fläche „durchzulegen“. Der für die Erzeugung der Oberfläche maßgebende Datensatz wird, wie auch der gewünschte Wert dieser Daten, übergeben. In Abb. 12 wurden beispielhaft VTS-CellData f aufgeführt. Durch Konvertierung dieser Zellattribute mit vtkCellDataToPointData wurden die, in der unteren Tabelle aufgeführten PointData f-Werte berechnet. Die roten Markierungen zeigen schematisch, wo die vtkContourFilter-Output-Isosurface für f=0,5 erzeugt würde. Dabei stellen die roten Kanten eine Interpolation zwischen den Punktwerten dar. Der rot markierte Wert liegt auf der Isosurface, da f=0,5 gewählt wurde. Abb. 19 zeigt eine Darstellung mit ParaView der in Abb. 12 geschilderten Daten.

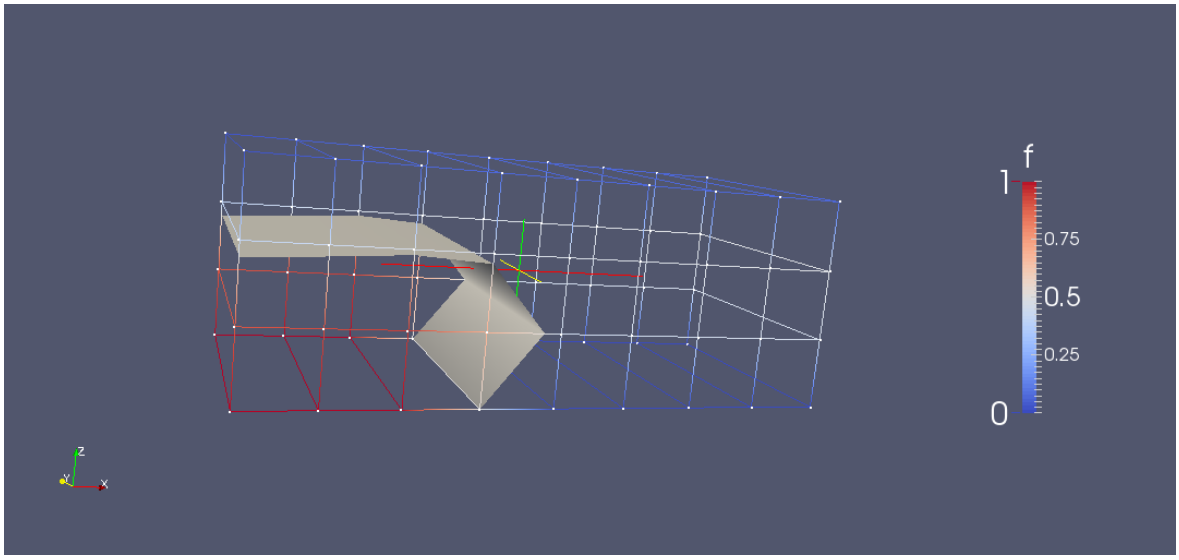


Abb. 19 Darstellung von VTS-Daten mit Färbung durch Punktattributdaten f und Erstellung einer Isosurface (f=0,5) durch den Filter Contour mit ParaView

Die folgenden Abschnitte (9, 10 und 11) beschäftigen sich mit verschiedenen Möglichkeiten der Texturierung von Oberflächen bzw. Objekten mit VTK.

9 Prinzip und Möglichkeiten der Texturierung mit VTK

9.1 Möglichkeiten der Texturierung mit VTK

Es gibt verschiedene Möglichkeiten der Oberflächentexturierung. Eine davon ist das Mappen (die Projektion) eines Bildes auf eine Fläche mit Bestimmung der relativen Koordinaten des Bildes gegenüber des zu texturierenden Objektes. Dieses Vorgehen wird in Abschnitt 11 erläutert und wurde für die Luftbildprojektion auf das Geländemodell herangezogen.

Texturen können mit VTK auch in Abhängigkeit von Skalarwerten erzeugt werden. Dazu wird eine so genannte LookupTable (Klasse vtkLookupTable) definiert. Es handelt sich dabei um einen Wertebereich von Skalarwerten, denen in der Tabelle bestimmte Farbwerte zugewiesen werden. Damit ist für jeden Skalarwert eine Farbe vorgegeben. Das zu texturierende Objekt besitzt je nach Anwendung Punkt- oder Zellwerte. Je nach Wert wird das Objekt mit der, den Skalarwerten der Punkte bzw. Zellen, zugehörigen Farbe dargestellt. Damit wird eine Visualisierung von CFD-Daten in dem mit VTK erzeugten Modell möglich. Ein Ausschnitt einer Oberfläche im Wireframe-Modus ist in Abb. 20 und Abb. 21 dargestellt und zeigt die Punkt- (Abb. 21) bzw. Zellwertfarbe (Abb. 20), die durch die LookupTable definiert ist. Die vtkLookupTable kann in Form einer Farbskala anschaulich wiedergegeben werden.

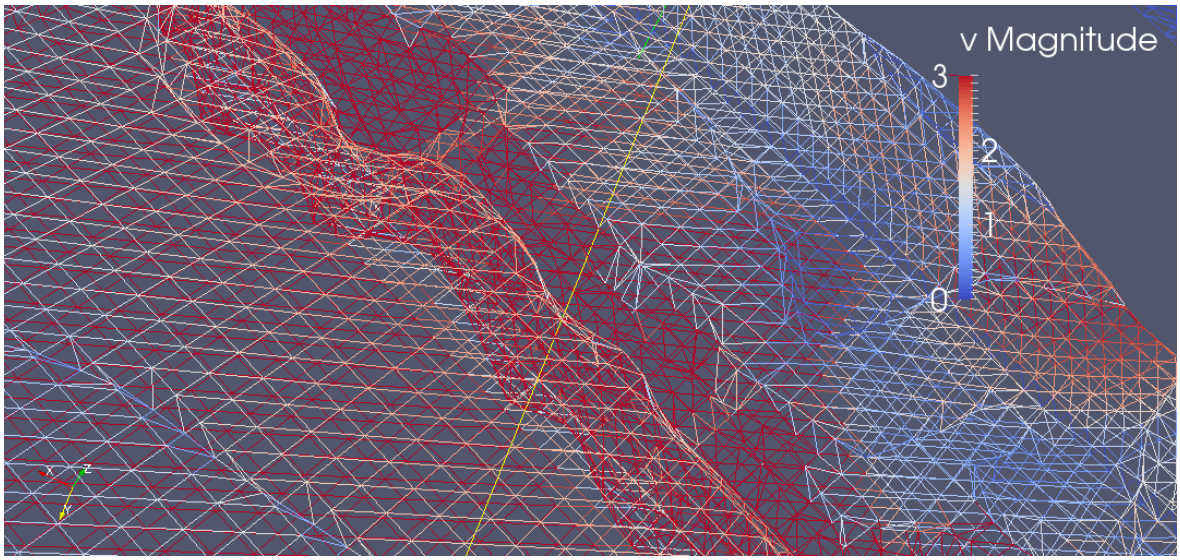


Abb. 20 Färbung des Objektes in Abhängigkeit von den Zellskalarwerten (hier Geschwindigkeitswerte v) in ParaView. Die Skala zeigt die Zuweisung der Farbwerte für definierte Geschwindigkeiten.

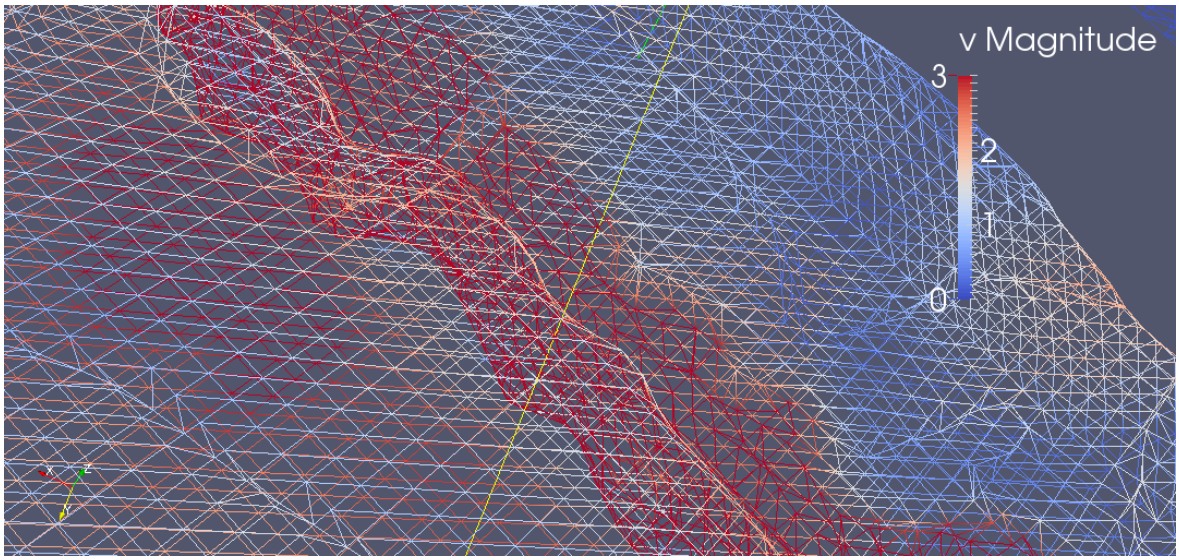


Abb. 21 Ansicht aus ParaView wie in Abb. 20 für die Färbung durch Punktskalarwerte.

Auf dieses Vorgehen mit einer vtkLookupTable und zu der genauen Implementierung wird in Abschnitt 10 im Detail eingegangen.

Im Folgenden sollen, zur Verschaffung eines ersten Überblickes, kurz bestimmte Aspekte dazu geschildert werden.

9.2 Skalarwerte-Visualisierung durch eine vtkLookupTable

Prinzipiell stehen zwei Methoden zur Definition der Farbwerte in Abhängigkeit der Skalarwerte zur Verfügung.

9.2.1 Definition eines Farbübergangs zwischen zwei vorgegebenen Werten

Die Farbwerte können für ein Objekt der Klasse vtkLookupTable als Farbübergang definiert werden, wobei die Farbwerte für den kleinsten und den größten der dargestellten Werte angegeben werden. Zur Umsetzung der Farbangaben als solche Skala (d.h. Interpolation zwischen den zwei angegebenen Werten), müssen die Farben in der Implementierung im HSV-Farbmodell angegeben werden. H bezeichnet den Farbton (hue), S die Sättigung (saturation) und V steht für die Helligkeit (value) (Maschke 2004, 49).

Zur anschaulichen Bestimmung der gewünschten Farben oder um den Farbwert für gegebene Bildbereiche zu kennen kann Blender verwendet werden. Auch die „Konvertierung“ der Farbwerte zwischen den Farbmodellen RGB und HSV kann mit Blender durchgeführt werden. An dieser Stelle werden die wichtigsten Funktionen dazu in Blender kurz aufgeführt.

9.2.1.1 Exkurs zur Farbbestimmung mit Blender

In Blender kann ein beliebiges Objekt mit rechter Maustaste markiert werden (z. B. einfach der Würfel, der beim Öffnen in der Szene schon vorhanden ist). Im Editor Properties->Material (Falls noch kein Material angelegt ist, kann durch Klicken auf „New“ ein Material erstellt werden.) kann bei Diffuse in die Farbfläche geklickt werden (Abb. 22). Dabei erscheint ein Farbauswahlfenster in dem Farben einfach eingestellt werden können. Die Farbwerte können als RGB oder HSV angezeigt werden.

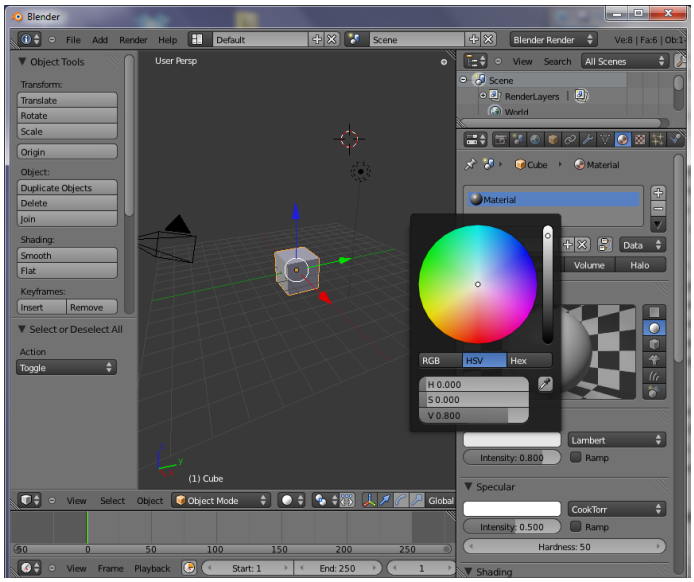


Abb. 22 Farbbestimmung mit Blender

9.2.1.2 Farbwertbestimmung aus einem Bild mit Blender

Wie in 9.2.1.1 kann bei editor->Properties->Texture mit Klick auf „New“ eine Textur erstellt werden. Bei „Type“ wird „Image or Movie“ eingestellt. In dem Abschnitt „Image“ wird mit „Open“ im Dateiexplorer ein Bild ausgewählt und mit „Open Image“ als Textur in Blender verwendet. Mit „F12“ wird aus der eingestellten Kameraansicht gerendert. Durch Klicken mit der linken Maustaste in

der Renderansicht (*UV/Image Editor*) wird der Farbwert an der Mausposition im RBG und im HSV Modell angegeben (Abb. 23).

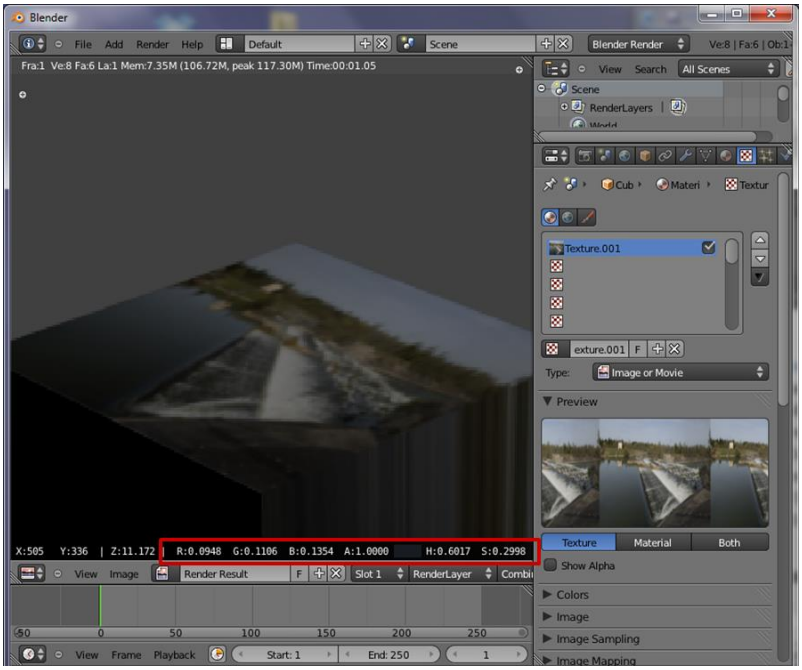


Abb. 23 Bestimmung eines Bildfarbwertes mithilfe von Blender

Der Rechenweg zur Konvertierung von HSV- zu RGB-Farbwerten kann dem VTK-Quellcode entnommen werden. Die Berechnung ist in der Klasse vtkMath in der Methode HSVToRGB() implementiert (Abb. 24). Beispielhaft wurden hier die berechneten Werte für einen Blauton (mit H=0,643; S=0,920; V=0,500) angegeben (hellblaue Schrift).

```
void vtkMath::HSVToRGB(double h, double s, double v,
                       double *r, double *g, double *b)
{
  const double onethird = 1.0 / 3.0;
  const double onesixth = 1.0 / 6.0;
  const double twothird = 2.0 / 3.0;
  const double fivesixth = 5.0 / 6.0;

  // compute RGB from HSV
  if (h > onesixth && h <= onethird) // green/red
    {
    *g = 1.0;
    *r = (onethird - h) / onesixth;
    *b = 0.0;
    }
  else if (h > onethird && h <= 0.5) // green/blue
    {
    *g = 1.0;
    *b = (h - onethird) / onesixth;
    *r = 0.0;
    }
  else if (h > 0.5 && h <= twothird) // blue/green
```

```
    {
    *b = 1.0;                                          //*b=1.0
    *g = (twothird - h) / onesixth;                    //*g=(twothird-0.643)/onesixth=0.142
    *r = 0.0;                                          //*r=0.0
    }
  else if (h > twothird && h <= fivesixth) // blue/red
    {
    *b = 1.0;
    *r = (h - twothird) / onesixth;
    *g = 0.0;
    }
  else if (h > fivesixth && h <= 1.0) // red/blue
    {
    *r = 1.0;
    *b = (1.0 - h) / onesixth;
    *g = 0.0;
    }
  else // red/green
    {
    *r = 1.0;
    *g = h / onesixth;
    *b = 0.0;
    }

  // add Saturation to the equation.
  *r = (s * *r + (1.0 - s));                     //*r=(0.920*0.0+(1.0-0.920))=0.080
  *g = (s * *g + (1.0 - s));                     //*g=(0.920*0.142+(1.0-0.920))=0.2106
  *b = (s * *b + (1.0 - s));                     //*b=(0.920*1.0+(1.0-0.920))=1.0

  *r *= v;                                       //*r=0.080*0.500=0.040
  *g *= v;                                       //*g=0.2106*0.500=0.105
  *b *= v;                                       //*b=1.0*0.500=0.500
}
```

Abb. 24 Konvertierung von HSV- zu RGB-Farbwerten (nach vtkMath.cxx)

9.2.2 Bestimmung der Farbe für einzelne Skalarwerte

In der vtkLookupTable können auch einzelne Farbwerte im RGB-Farbmodell eingesetzt werden. Diese Werte können Farben des Farbverlaufes ersetzen.

Dies ermöglicht die Erstellung von Texturen ähnlich wie es von Programmen wie Blender bekannt ist, wo Muster visuell erzeugt und bearbeitet werden, um realistische Oberflächen zu erreichen. Zu bedenken ist, dass die Farben auch in diesem Fall bestimmten Skalarwerten zugewiesen sind und damit die realen Werte wiedergeben. Es ist jedoch darauf zu achten, dass trotz der Bearbeitung des Farbverlaufes, die physikalischen Eigenschaften der Oberfläche der Farbgebung entsprechen.

Wie in Abschnitt 10 ausführlich behandelt, kann durch die Bestimmung der Anzahl der Farben NumberOfColors in der Tabelle (Abb. 25) die Größe eines Attributwerte-Bereiches festgelegt werden, der pro Farbe zugeordnet wird. Je mehr Farben gespeichert sind, desto kleiner ist der Attributwertebereich pro Farbe. Durch Erhöhung der Anzahl NumberOfColors kann damit bewirkt werden, dass einzelne ersetzte Farben kleinere Punkte in der Texturierung bewirken.

In einer Schleife über die Anzahl der Farben oder einen Ausschnitt daraus (oder für einzelne Farben) kann mit SetTableValue() (Abb. 25) eine beliebige Farbe in der Tabelle eingefügt/ersetzt werden.

10 Texturierung zur Wiedergabe von CFD Datensätzen_ Texturierung der Wasseroberflächen mit VTK

In diesem Abschnitt wird genau darauf eingegangen, wie die Texturierung mit einer vtkLookupTable funktioniert. Das Vorgehen kann in zwei Schritte unterteilt werden. Zunächst wird die Erstellung des Objektes der Klasse vtkLookupTable erläutert. Dann wird beschrieben, wie die Skalarwerte den Farbwerten der LookupTable zugeordnet werden.

Abb. 25 zeigt einige Attribute und Methoden der Klasse vtkLookupTable, die für das Verständnis von Bedeutung sind.

vtkLookupTable
vtkIdType NumberOfColors //Anzahl der Farben / Tuple vtkUnsignedCharArray *Table //Speicherung der Farbwerte double TableRange[2] //Grenzwerte der Skalarwerte (des zu texturierenden Objektes) die durch den Farbverlauf dargestellt werden sollen double HueRange[2] //Grenzfarbwerte für Farbverlauf: Tabellen-Farb-Grenzwerte H double SaturationRange[2] //Tabellen-Farb-Grenzwerte S double ValueRange[2] //Tabellen-Farb-Grenzwerte V double AlphaRange[2] //Tabellen-Grenzwerte alpha

vtkLookupTable (int sze=256, int ext=256) //u.a. Speicherplatz für 4 Werte pro Farbe mit SetNumberOfComponents(4); Speicherplatz für 4*256 Farbwerte mit Allocate()
virtual void SetTableValue (vtkIdType indx, double rgba[4]) //Einfügen einer Farbe an einer bestimmten Stelle der Tabelle in vtkUnsignedCharArray *Table. Angabe als RGBA-Wert aus [0;1]. Build () oder SetNumberOfTableValues() davor notwendig.
void SetNumberOfTableValues (vtkIdType number) //Einstellung der Anzahl an Farben und Speicherplatzfreigabe durch SetNumbeOfTuples
virtual vtkIdType GetIndex (double v) //Berechnung des Index der Farbe der Tabelle (in vtkUnsignedCharArray *Table) für einen gegebenen Skalarwert. Index aus [0;(Anzahl der Tuple – 1)]
void SetRampToLinear () //Art der Interpolation für die Berechnung des Farbverlaufes
void SetRampToSCurve () //Art der Interpolation für die Berechnung des Farbverlaufes
void SetRampToSQRT () //Art der Interpolation für die Berechnung des Farbverlaufes
virtual void Build () //Aufruf von ForceBuild()
virtual void ForceBuild () //Berechnung der Farbwerte des Objektes der Klasse, und Speicherung dieser Werte in vtkUnsignedCharArray *Table -> Füllen der Tabelle mit Farbwerten (als Farbverlauf)

Abb. 25 Zusammenstellung wichtiger Attribute und Methoden der Klasse vtkLookupTable (->vtkLookupTable.h und vtkLookupTable.cxx). (Die Darstellung ist keine vollständige Aufführung der Klasse und weist keine korrekte Anwendung der UML-Notation auf. Die Abbildung dient hier dem Verständnis und soll einen Überblick über den Abschnitt erleichtern.)

10.1 Erzeugung eines Objektes der Klasse vtkLookupTable

Zunächst wird der Konstruktor von vtkLookupTable in vtkLookupTable.cxx untersucht.

10.1.1 Konstruktor und Speicherfreigabe

Das Attribut `vtkUnsignedCharArray *Table` dient der Speicherung der Farbwerte der LookupTable und hat folgende Struktur. Die Anzahl der Arrayeinträge beträgt 4*`NumberOfColors` wobei `NumberOfColors` die Anzahl der Farben der LookupTable ist. In dem Konstruktor wird die Anzahl gleich 256 gesetzt. Dazu wird im Verlauf dieses Abschnittes Weiteres beschrieben. Mit `SetNumberOfTableValues` kann in der eigenen Implementierung für ein Objekt der Klasse vtkLookupTable eine beliebige Anzahl an Farben eingestellt werden. In dieser Prozedur wird dann mit `Table->SetNumberOfTuples(number)` der Speicherplatz für alle Farben in dem Array freigegeben. „Tuples“ bezeichnen die Anzahl der gespeicherten Farben. Pro „Tuple“ werden vier Farbwerte für RGBA bzw. HSVA gespeichert. Diese vier Einträge werden als „Components“ bezeichnet.

Der Speicherplatz pro „Tuple“ wird mit `Table->SetNumberOfComponents(4)` im Konstruktor festgelegt. Der vierte Eintrag pro Farbe ist der Alphawert (daher RGBA bzw. HSVA). Dieser bestimmt die Transparenz (0=vollkommen durchsichtig, 1=vollkommen opak).

Im Konstruktor wird der Speicherplatz für die 256 Farben durch `Table->Allocate(4*sze,4*ext)` freigegeben (mit sze=256 als Standardeinstellung ->vtkLookupTable.h).

Nach dem Konstruktor wird jetzt näher auf das Bilden der LookupTable eingegangen. Die Methoden, die im Folgenden genannt werden, sind von der Klasse vtkLookupTable (vtkLookupTable.cxx).

10.1.2 Eigenschaften und Aufbau des Objektes der Klasse vtkLookupTable / Untersuchung der Methode void vtkLookupTable::ForceBuild()

Für ein Objekt der Klasse vtkLookupTable (im Folgenden als LuT bezeichnet) müssen (wie in Abschnitt 9 schon geschildert) im HSV-Farbmodell zwei Farben angegeben werden, die die Grenzfarben der LuT bilden, und zwischen denen Farbwerte interpoliert werden um einen Farbverlauf zu ergeben. Wenn keine

Angaben dazu implementiert werden, wird die Standardeinstellung des Konstruktors übernommen, d.h. ein Farbübergang von rot zu blau. Zur eigenen Farbbestimmung werden mit `Lut->SetHueRange(); Lut->SetSaturationRange(); Lut->SetValueRange(); Lut->SetAlphaRange();` (Übergabe-Grenzfarbwerte aus [0;1]) die Grenzfarben eingestellt.

Wenn eine LuT neu erstellt wurde, muss die Methode `Build()` aufgerufen werden. Diese ruft wiederum `ForceBuild()` (vtkLookupTable.cxx) auf.

10.1.2.1 ForceBuild() _ Schritt 1 _ Berechnung von RGB-Farbwerten aus linear verteilten HSV-Werten

In der Methode ForceBuild() wird zunächst für jeden der vier Farbwerte „hue“, „saturation“, „value“ und Alpha jeweils die Wertveränderung pro Farbwechsel Δhsva nach Gleichung (3) berechnet. hsvaRange[0] und hsvaRange[1] bezeichnen die jeweiligen Farb-Grenzwerte.

$$\Delta hsva = (hsvaRange[1] - hsvaRange[0])/maxIndex \qquad (3)$$
$$maxIndex = NumberOfColors - 1$$

Im Anschluss werden, für jede Farbe der LookupTable (für die angegebene Anzahl der Farben NumberOfColors), die Farbwerte mit einer linearen Veränderung berechnet (Gleichung (4)). hsva steht für eine Verallgemeinerung des jeweiligen Farbwertes H, S, V oder Alpha. i ist die Laufvariable einer Schleife über alle Farben.

$$hsva = hsvaRange[0] + i \times \Delta hsva \qquad (4)$$

Es folgt eine Konvertierung der HSV-Werte zu RGB-Werten mit der in Abschnitt 9.2.1 (Abb. 24) aufgeführten Prozedur `vtkMath::HSVToRGB()`.

Dann können für jede Farbe vier Werte (RGBA) in die LuT mit `Table->WritePointer(4*i,4)` eingefügt werden. Der Index zum Einfügen in das *Array* Table ist durch [(4 „components“) * (i. „tuple“)] an erster Stelle übergeben. Dann folgt die Anzahl der Einträge (4 „components“ bzw. Farbwerte der gerade berechneten Farbe). An dieser Stelle gibt es verschiedene Möglichkeiten der

Farbverlaufsberechnung. Für das erstellte Objekt LuT kann in der eigenen Implementierung `Lut->SetRampToSQRT();` `Lut->SetRampToLinear();` `Lut->SetRampToSCurve();` aufgerufen werden um die Form der Interpolation für den Farbverlauf anzugeben. Die Berechnung ist in der Methode ForceBuild() implementiert.

10.1.2.2 ForceBuild() _ Schritt 2 _ Funktionen zur Beeinflussung des Farbverlaufes

Zunächst werden wichtige Aspekte der digitalen Farbspeicherung behandelt, die bei dem Einfügen der Farben in der LuT von Bedeutung sind.

10.1.2.2.1 Ausgewählte Aspekte der digitalen Farbspeicherung (Feller 2010)

Jedes Pixel stellt einen definierten Farbwert dar, der über Zahlenwerte berechenbar wiedergegeben wird. Dabei spielen Farbtiefe (über alle Farbkanäle zur Bittiefe zusammen addiert) und das Farbmodell eine maßgebende Rolle (Feller 2010, 44).

Durch die zwei möglichen Zustände eines Bits 0 und 1, lassen sich durch einen Bit zwei Farben darstellen. Um einem Pixel mehr unterschiedliche Farbwerte zuweisen zu können (Zustände) müssen für jeden Bildpunkt mehrere Bits („zur Speicherung des Pixelwertes“) zur Verfügung gestellt werden. „Bei 8 Bit lassen sich für einen Bildpunkt 256 verschiedene Abstufungen erzeugen (2^8 Möglichkeiten).“ (Feller 2010, 44)

„Da sich aber ein gewöhnliches Farbbild (RGB-Bild) aus drei Farbkanälen (Rot, Grün, Blau) zusammensetzt, durch deren additive Mischung die angezeigte Farbe entsteht, ergeben sich bei einem solchen Bild mit einer Farbtiefe von 8 Bit je Kanal 256 mögliche Werte und damit $(2^8)^3$=[…]=16.777.216 maximal mögliche darstellbare Farben pro Bildpunkt.“ (Feller 2010, 46)

In der Methode ForceBuild() der Klasse vtkLookupTable wird mit einer Farbtiefe von 8 Bit je Kanal dargestellt. Hier sind für jedes Pixel Farbinformationen für vier Kanäle gespeichert: die Farbkanäle rot, grün, blau und den Alphakanal.

10.1.2.2.2 Berechnung der Farbwerte der vtkLookupTable in der Methode ForceBuild() mit linearer Interpolation

Sei c_rgba ein *unsigned char Array* mit vier Einträgen zur Speicherung der berechneten Farbwerte der vier Kanäle k (je Farbe der LuT). Die Werte mit linearer Interpolation werden nach Gleichung (5) berechnet.

$$c\_rgba[k] = static\_cast < unsigned\ char > (rgba[k] * 255.0 + 0.5) \quad (5)$$
$$k \in [0; 3]$$

In rgba[k] sind die konvertierten RGBA-Farbwerte aus Schritt 1 (10.1.2.1) gespeichert. Diese sind aus dem Bereich [0;1] und werden zunächst (Gleichung(5)) mit 255 multipliziert und somit auf den Bereich [0;255] skaliert. Durch *static_cast<unsigned char>* wird der Farbwert zu dem Datentyp *unsigned char* umgewandelt.

Ein Zeichen (Datentyp *char*) „wird intern als 1-Byte-Ganzzahl interpretiert.“ (Breymann 2007, 53) Unsigned char hat somit die Größe 8 Bit und liegt in dem Bereich [0;255]. Wenn der zu konvertierende Wert außerhalb des Bereiches [0;255] liegt und deswegen nicht mit einem Byte dargestellt werden kann, werden die, über die 8 Bit hinaus gehenden Daten nicht berücksichtigt (Breymann 2007, 55). Wenn z. B. der *double* Wert (64 Bits) 244,623 zu *unsigned char* umgewandelt wird und dann zu *double* zurückkonvertiert wird, erhält man den Wert 244. Zu dem Datenverlust kommt es für die rgba[k]-*double*-Werte auch. Um eine korrekte Rundung der Werte zu erreichen wird vor der Konvertierung 0,5 addiert (somit würde in dem genannten Beispiel nach der Umwandlung 245 statt 244 ausgegeben).

Schließlich sind die Farbwerte c_rgba[k] Ganzzahlen aus dem Bereich [0;255], die der Farbdarstellung mit einer Farbtiefe von 8 Bit (je Kanal) wie in 10.1.2.2.1 beschrieben entsprechen.

10.1.2.2.3 Berechnung der Farbwerte der vtkLookupTable in der Methode ForceBuild() mit *SQRT*-Interpolation

Die Berechnung wird nach Gleichung (6) durchgeführt.

$$c\_rgba[k] = static\_cast < unsigned\ char > (\sqrt{(rgba[k])} * 255.0 + 0.5) \quad (6)$$
$$k \in [0; 3]$$

Die Formeln ist analog zu der linearen Interpolation zu interpretieren.

10.1.2.2.4 Berechnung der Farbwerte der vtkLookupTable in der Methode ForceBuild() mit *S-Curve*-Interpolation

Die *S-Curve*-Funktion ist Gleichung (7) zu entnehmen.

$$y = (1 + \cos((1 - x) * \pi))/2 \tag{7}$$

$$c\_rgba[k] = static\_cast < unsigned\ char > (y(rgba[k]) * 255.0 + 0.5)$$

10.1.2.2.5 Fazit zu den Interpolationsfunktionen

Die Ergebnis-Farbwerte die schließlich in der LuT gespeichert wurden, sind wie aus den beschriebenen Berechnungen ersichtlich, außer von der Interpolationsfunktion selber, auch von den gewählten Grenzfarbwerten und von der Anzahl der Farbwerte in der Tabelle abhängig.

Für die Darstellung der Ergebnisse der unterschiedlichen Interpolationsfunktionen wurden die Berechnungen der Methode ForceBuild() mit Microsoft Excel nachgerechnet und als Diagramm veranschaulicht (Abb. 26). Dazu wurde die Anzahl der Farben der Tabelle gleich 256 gesetzt, und der Farbbereich für die Interpolation gleich schwarz bis weiß, zur Darstellung des gesamten möglichen Farbbereiches, gewählt. Die Berechnung wird hier beispielhaft für den Farbkanal Rot durchgeführt.

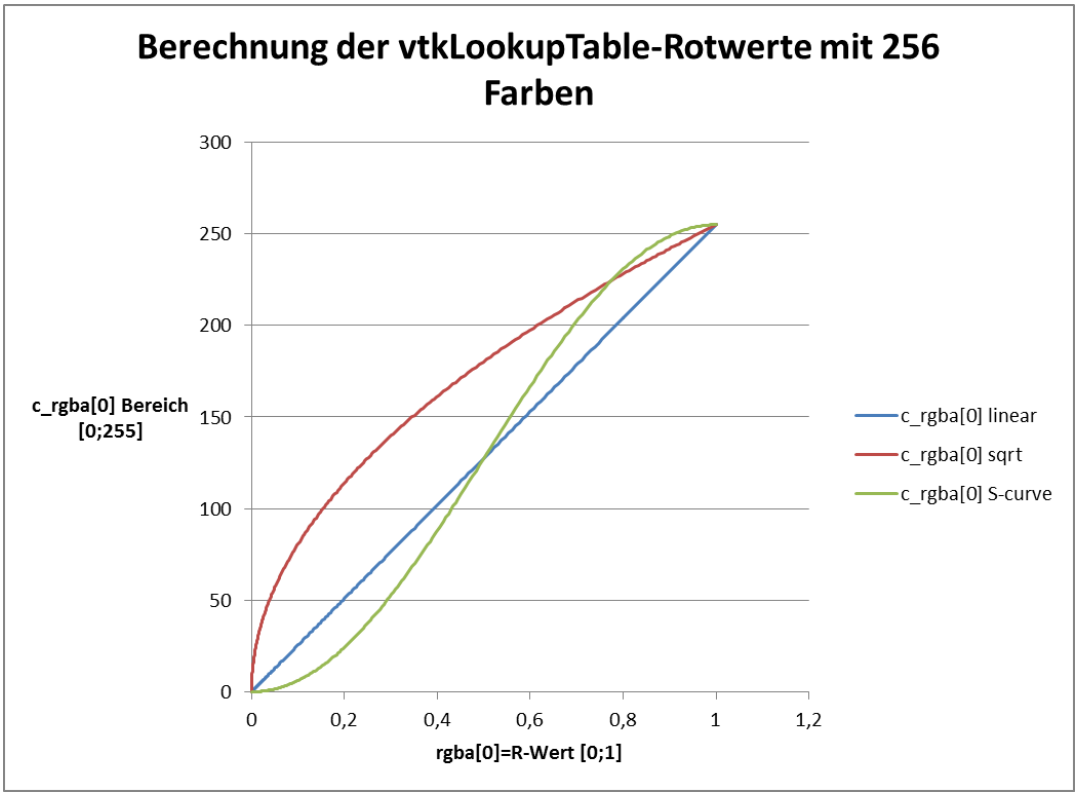


Abb. 26 Darstellung der in ForceBuild() berechneten Farben der vtkLookupTable für die drei Interpolationsfunktionen am Beispiel des Rotkanals. Der Farbverlauf geht von schwarz bis weiß, mit 256 Farben in der Tabelle.

Bei der Texturierung eines Objektes durch Zuweisung der Farben der LuT je nach Skalarwert ist der Farbunterschied zwischen den drei Interpolationsberechnungen eindeutig sichtbar. Dies wird anhand einer Oberflächendarstellung je nach Geschwindigkeitswert veranschaulicht (Abb. 27). Der Farbverlauf geht von dunkelblau und etwas transparent zu weiß und opak. Gegenüber einer linearen Interpolation wird mit *SQRT* die gesamte Oberfläche heller/weißer. Der dunkle Bereich ist sehr reduziert. Durch *S-Curve* werden die dunkelblauen und komplett weißen Bereiche größer und häufiger. Die Oberfläche erscheint mit mehr Kontrast.

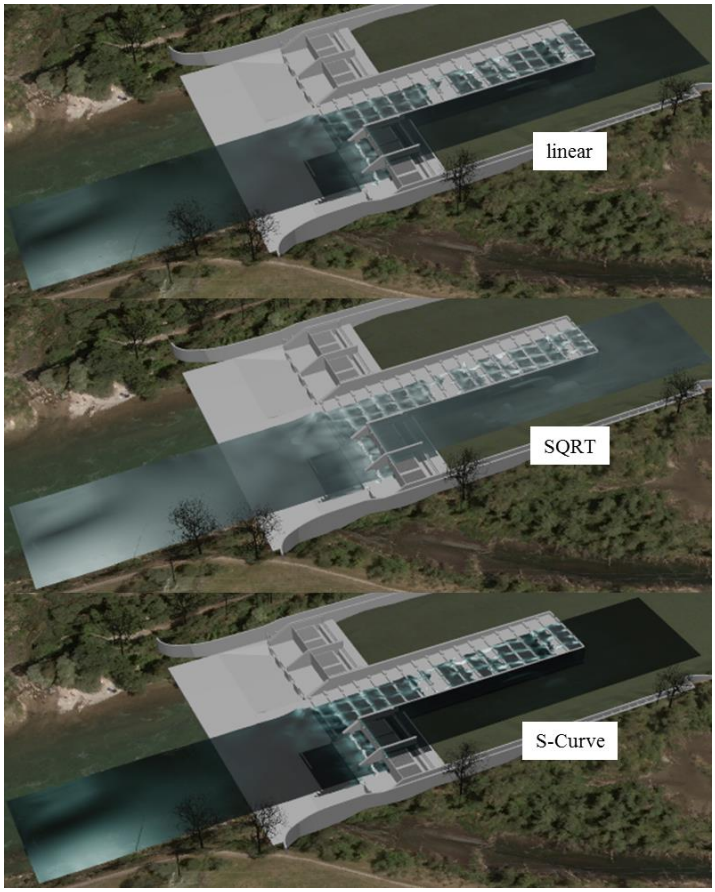


Abb. 27 Unterschiede der Texturierung einer Oberfläche mit einer vtkLookupTable zwischen linearer, *SQRT* und *S-Curve* Interpolation zur Berechnung der Farbwerte der Tabelle / Renderansicht zu einer Implementierung mit VTK

10.1.3 Einfluss der Anzahl der Farbwerte der Tabelle und der gewählten Grenz-Farbwerte

Abb. 26 zeigt, dass schon mit 256 Farben für einen Übergang von schwarz zu weiß die Interpolationsfunktionen relativ genau abgebildet werden. Bei Betrachtung der dazu berechneten Werte kann für einen Kanal beobachtet werden, dass erstens durch die Konvertierung der HSV- zu RGB-Werten, der Farbunterschied zwischen den Farben rgba[k] nicht konstant ist. Damit werden auch bei linearer Interpolation nicht alle 256 gleichzeitig darstellbaren Farbwerte des Kanals verwendet. Zweitens steigt durch die Berechnung der anderen Interpolationsfunktionen die Anzahl der 256 Farben des Kanals, die nicht repräsentiert sind.

Durch Erhöhung der Farbwertanzahl in der Tabelle kann die Anzahl der Farbwerte pro Kanal besser ausgenutzt werden. Es ist jedoch zu beachten, dass nicht mehr als 256 unterschiedliche Werte pro Kanal möglich sind, und sich darüber hinaus die Farbwerte wiederholen.

Weiter muss beachtet werden, dass sich eine Verringerung der Ausdehnung der Farbskala ((Farbgrenzwert[1]- Farbgrenzwert[0])<1) bei gleicher Anzahl der Farben der Tabelle (NumberOfColors), bei der Berechnung der Farbwerte wie eine Erhöhung der Anzahl der Farben in dem Farbbereich auswirkt.

Bei der Berechnung der interpolierten Farbwerte mit *static_cast<unsigned char>* entsteht ein Rundungsfehler der auf die Begrenzung auf 256 Farbwerte pro Kanal zurückzuführen ist. Dadurch entsteht allerdings eine maximale Farbverschiebung von 0,5 in dem Farbbereich [0;255] bzw. 0,5/255=0,0020 in [0;1]. Eine solche Veränderung der Farbwerte je Kanal ist so gering, dass sie bei der Darstellung der Farben nicht sichtbar wird.

10.2 Zuordnung der Farben für definierte Skalarwerte eines Objektes

Die, wie zuvor beschrieben, erzeugte LuT kann durch ein Objekt der Klasse vtkPolyDataMapper verwendet werden, um die Textur des Objektes zu bestimmen (Abschnitt 3.3). Mit `mapper->SetScalarRange()` können die Skalarwerte festgelegt werden (LuT->TableRange[0] und LuT->TableRange[1]), die den Grenz-Farbwerten zuzuordnen sind. Skalare Werte, die größer sind als der Maximalwert werden dem oberen Farb-Grenzwert zugeordnet. Skalare Werte, die kleiner sind als der Minimalwert werden dem unteren Farb-Grenzwert zugewiesen.

Das Vorgehen der Auswahl der Farbe aus der LuT wird hier anhand der Prozedur `void vtkLookupTable::GetColor(double v, double rgb[3])` untersucht (v bezeichnet den skalaren Wert wie z.B. einen Geschwindigkeitswert).

In dieser Methode wird die Farbe durch den Aufruf `this->MapValue(v)` bestimmt. In dieser Funktion wird der Index des „Tuples“ (der Farbe) gebraucht, der aus der LuT zu dem skalaren Wert v passt. Dazu wird `this->GetIndex(v)` aufgerufen.

Diese Funktion `vtkIdType vtkLookupTable::GetIndex(double v)` soll genau beschrieben werden um das Vorgehen der Darstellung von Werten durch die Texturierung mit einer vtkLookupTable vollständig zu erläutern.

Zunächst werden zwei *double* Werte *shift* und *scale* definiert, um die skalaren Werte auf den Bereich [0;(NumberOfColors-1)] zu transformieren (Gleichung (8)).

$$shift = -(LuT \rightarrow TableRange[0]) \tag{8}$$
$$scale = (NumberOfColors)/((LuT \rightarrow TableRange[1]) - (LuT \rightarrow TableRange[0]))$$

Damit wird nach Gleichung (9) der Index „findx“ der Tabellenfarbe („tuple“ / Zur Indexierung in dem *Array* „Table“ der LuT, der alle Farbwerte c_rgba[k] enthält, muss als Index 4*findx verwendet werden.) berechnet, die für einen skalaren Wert v dargestellt werden soll (in Abb. 28 grün dargestellt).

$$findx = (v + shift) \times scale \tag{9}$$

Aus Gleichung (9) lassen sich *double* Werte findx berechnen, die je nach Objekt-Attributwert v auch Nachkommastellen aufweisen können. Der Zeiger auf eine bestimmte Tabellenfarbe muss auf einen Index des *Arrays* LuT->Table verweisen. Da in LuT->Table die Anzahl „NumberOfColors“ an Farben gespeichert ist, muss der Index ein Integer Wert aus dem Bereich [0;(NumberOfColors-1)] sein.

Mit `static_cast<int>(findx)` werden die findx-Werte zu *integer* Werten konvertiert und können somit durch einen *Pointer* verwendet werden, um auf eine Farbe der LuT zu verweisen (in Abb. 28 rot dargestellt).

Abb. 28 stellt die Implementierung der Methode `vtkIdType vtkLookupTable::GetIndex(double v)` graphisch dar.

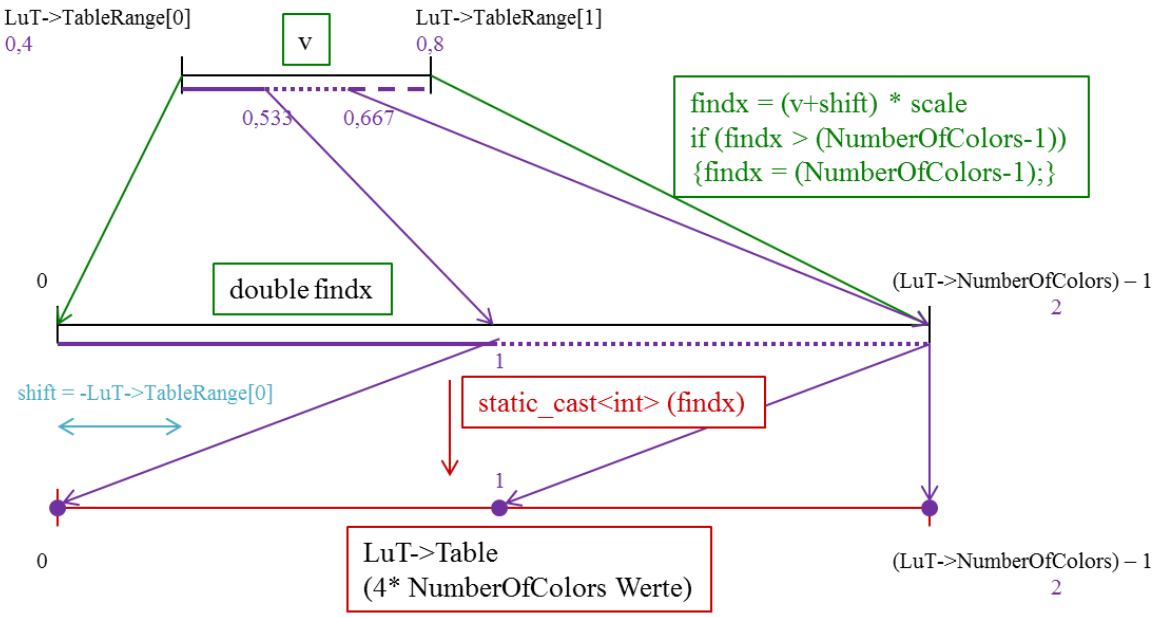


Abb. 28 Graphische Darstellung der Methode vtkLookupTable::GetIndex(double v) (vtkLookupTable.cxx) zur Berechnung des Index der Tabellenfarbe für einen gegebenen skalaren Wert v

In der Abbildung ist lilafarben ein Beispiel aufgeführt. Folgende Werte werden gewählt: TableRange[0]=0,4; TableRange[1]=0,8; NumberOfColors=3.

Der, durch einen lila durchgängigen Strich, markierte Bereich [0,4;0,533] der Werte v ergibt findx-Werte in dem entsprechend markierten Bereich [0;1]. v aus [0,533;0,667] führt zu findx aus [1;2] (mit gepunkteter Linie dargestellt). v aus [0,667;0,8] ergeben findx-Werte aus [2;3], welche alle gleich 2 gesetzt werden (gestrichelte Linie).

Durch die Umwandlung zu *integer* Werten ergeben damit alle v-Werte [0,4;0,533[den Index 0. Alle skalaren Werte aus [0,533;0,667[erhalten die Tabellenfarbe mit dem Index 1. Für v aus [0,667;0,8] wird die dritte Farbe der Tabelle (Index 2) wiedergegeben.

Daraus ist klar zu erkennen, wie die Anzahl der Tabellenfarben die Genauigkeit der Wiedergabe der skalaren Werte beeinflusst. Je größer die Anzahl der gespeicherten Werte, desto kleiner werden die Intervalle der v Werte, die zu einem gleichen Farbindex führen.

Es sollte jedoch auch darauf geachtet werden, dass sich ab einer bestimmten Anzahl, wie schon zuvor in diesem Abschnitt beschrieben, die Farbwerte wiederholen, und somit die Farbgenauigkeit trotz Erhöhung der Indexgenauigkeit gleich bleibt.

Aus Abb. 28 wird zudem der Einfluss des Wertebereiches der skalaren Werte LuT->TableRange[2] sichtbar. Selbstverständlich muss, für die gleiche Genauigkeit, die Anzahl der Tabellenfarben erhöht werden, wenn der v-Werte-Bereich größer gewählt wird.

Zur Bestimmung der für eine bestimmte Aufgabe angebrachten Anzahl der Farben -LuT->NumberOfColors- müssen die verschiedenen in diesem Abschnitt aufgeführten Parameter und deren Zusammenspiel berücksichtigt werden.

Wie schon erwähnt ist eine weitere Art der Texturierung mit VTK durch das *Mapping* eines Bildes auf eine Oberfläche gegeben. Dies wird in dem folgenden Abschnitt erläutert.

11 Texturierung des Geländemodells mit einem Luftbild

Das Gelände wurde als STL-Datei eingelesen, und das Luftbild, welches als Textur dienen soll, ist im TIF-Format vorhanden. Der entscheidende Schritt für das Mappen des Bildes ist die Erzeugung der Texturkoordinaten. Darauf soll wie folgt näher eingegangen werden.

11.1 Überblick zu Datenformat und Erzeugung der Texturkoordinaten

Durch das Einlesen der STL-Daten mit einem Objekt der Klasse vtkSTLReader wird das Gelände als vtkPolyData ausgegeben. Dessen Punkte sind als vtkPoints vorhanden. Wie auch bei den Punkten der vtkStructuredGrid, können den Punkten der vtkPolyData Attributdaten zugewiesen werden. Die Attributdaten TCoords sind die Texturkoordinaten der entsprechenden Punkte. Damit besteht die Aufgabe darin, den Geländepunkten die Texturkoordinaten als Punktattribute zuzuweisen. Zu beachten ist, dass die Texturkoordinaten auf den Bereich [0;1] transformiert werden müssen, und damit eine Skalierung der Werte vor der Übergabe als Texturkoordinaten vorgenommen werden muss. Mit der Programmierung der Skalierung und Verteilung der Werte auf den Bereich [0;1] für die Vorgabe der Texturkoordinaten wird es möglich, die Art der Projektion des Bildes auf das Gelände für die vorliegende Aufgabe zu bestimmen. Dies wird im Laufe des Abschnittes genau erläutert.

In VTK ist die Klasse vtkImplicitTextureCoords zur Erstellung bestimmter Texturkoordinaten implementiert. Für die vorliegende Aufgabe konnte diese nicht angewendet werden. Code-Teile werden jedoch für die eigene Programmierung in veränderter und angepasster Form verwendet. Zunächst werden einige Aspekte der Klasse vtkImplicitTextureCoords erläutert, die zum Verständnis sehr hilfreich sind.

11.2 vtkImplicitTextureCoords zur Erzeugung von Textur-Koordinaten

Mit der Klasse vtkImplicitTextureCoords werden für bestimmte Input-Punkte die Attributdaten TCoords berechnet und den Output-Punkten übergeben. (Wie in der Funktion `int vtkImplicitTextureCoords::RequestData(vtkInformation *vtkNotUsed(request), vtkInformationVector **inputVector, vtkInformationVector *outputVector)` implementiert.) Dazu werden einem Objekt der Klasse vtkImplicitTextureCoords mit SetRFunction, SetSFunction und SetTFunction vtkImplicitFunction-Objekte zugewiesen, die als Funktionen dienen, um aus den Input-Punktkoordinaten neue Werte für die x-, y- und z-Richtung zu berechnen. Die neu errechneten Werte werden dann auf den Bereich (0,1) skaliert und verschoben.

Schließlich werden diese zuletzt erhaltenen Werte, den Output-Punkten als Texturkoordinaten (Attributwerte TCoords) übergeben.

Für die vorliegende Aufgabe kann die hier beschriebene Klasse nicht der Erzeugung der Texturkoordinaten dienen. Die Unterschiede zu der benötigten Funktionalität sind mit folgenden Punkten dargestellt. Bei der Bildprojektion auf das Gelände sind die Texturkoordinaten direkt aus den Gelände-Punktkoordinaten zu berechnen. D.h. die Gelände-Punktkoordinaten können abgefragt und auf den Bereich [0;1] transformiert werden. Die neu berechneten Werte sind die Texturkoordinaten und werden den Geländepunkten als Attributdaten TCoords zugewiesen.

Die Transformation der Koordinaten wird in dem folgenden Abschnitt erläutert.

11.3 Berechnung der Texturkoordinaten aus den Punktkoordinaten

11.3.1 Skalierungs- und Verschiebungsimplementierung zur Texturkoordinatenberechnung in der Klasse vtkImplicitTextureCoords

In der Funktion RequestData werden die Texturkoordinaten tCoord[i] für jeden Punkt und i aus [0;Texturkoordinatendimension[nach Gleichung (10) berechnet.

$$tCoord[i] = 0{,}5 + scale[i] \times tc[i] \tag{10}$$

tc[i] bezeichnet dabei, auf die vorliegende Aufgabe der Bildprojektion auf das Gelände angewendet, die Geländekoordinaten eines Punktes in i-Richtung. Scale[i] wird nach Gleichung (11) berechnet. Seien min[i] der minimale Geländekoordinatenwert über alle Geländepunkte in Richtung i, und max[i] der entsprechend maximale Geländekoordinatenwert. An dieser Stelle wird die Berechnung beispielhaft für (-min[i])>max[i] erläutert.

$$scale[i] = \frac{-0{,}499}{\min[i]} \tag{11}$$

Die geschilderte Berechnung ist in Abb. 29 schematisch dargestellt.

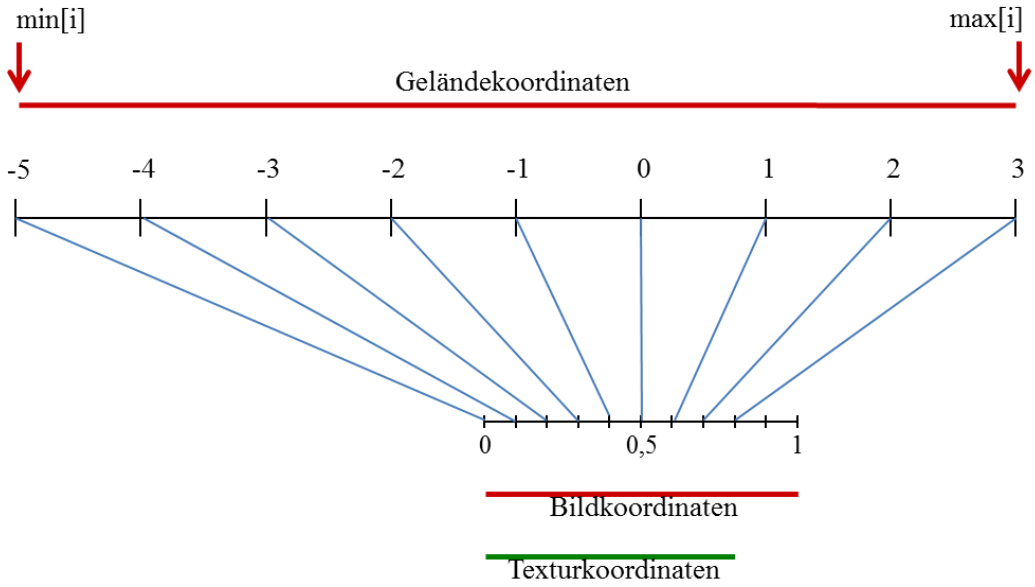


Abb. 29 Transformation der Gelände-Punktkoordinaten zu Texturkoordinaten wie in der Klasse vtkImplicitTextureCoords implementiert. Schematische Darstellung für Punktkoordinaten [-5,3] in eine Richtung i.

Das Prinzip der Berechnung beruht auf der Gleichsetzung der Geländekoordinaten 0 und des Texturkoordinatenwertes 0,5. Der, von (-min[i]) und max[i], größere Wert wird zur Bestimmung der Anzahl der Unterteilungen des Bereiches [0;0,5] bzw. [0,5;1] der Bildkoordinaten herangezogen. Daraus ergibt sich Gleichung (11) für den in Abb. 29 gewählten Fall (-min[i])>max[i]. Scale[i] ergibt damit die Länge einer der Unterteilungen der Bildkoordinaten wie in Abb. 29 auf der Bildkoordinaten-Skala eingezeichnet. Gleichung (10) ist in Abb. 29 durch die blauen Transformationsstriche dargestellt. Die Folge der in dieser Klasse gewählten Berechnung sind die in Abb. 29 grün markierten Texturkoordinaten. Diese stellen nur einen Teilbereich der rot markierten Bildkoordinaten dar, die mit den Bildrändern des als Textur verwendeten Bildes übereinstimmen. Das Ergebnis kann bildlich interpretiert werden, als würde das Gelände auf den grün markierten Bereich (Texturkoordinaten) des rot markierten Bereiches (Bildkoordinaten) des Texturbildes projiziert.

Eine Anwendung dieser Berechnungsmethode für die Zuweisung der Texturkoordinaten als TCoords Attributwerte der Geländepunkte ergibt bei der Darstellung ein Mappen des in Abb. 29 grün markierten Bereiches/Ausschnittes des Texturbildes auf die Geländeoberfläche. Die *render window*-Ausgabe ist in Abb. 30 wiedergegeben. Der Code dazu ist in Anhang 5 gegeben und kommentiert.

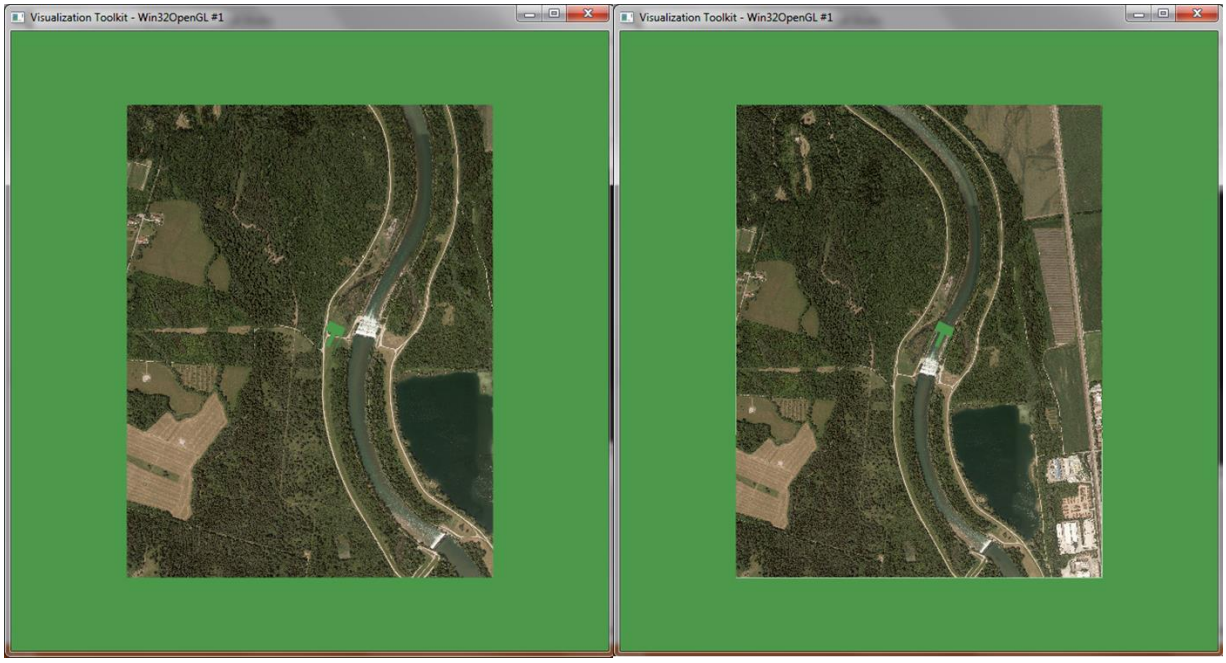


Abb. 30 Projektion eines Bildes auf eine Geländeoberfläche (Ansicht von oben) mit der Transformation der Geländepunktkoordinaten zu Texturkoordinaten links wie in vtkImplicitTextureCoords implementiert (Bildausschnitt) und rechts zum Mappen des gesamten Bildes

11.3.2 Skalierungs- und Verschiebungsimplementierung zur Texturkoordinatenberechnung für die Projektion eines gesamten Bildes auf eine Oberfläche

Die, zur Erzielung der gewünschten Texturierung der Geländeoberfläche mit dem TIF-Bild, durchzuführende Koordinaten-Transformation ist eine Anpassung der in 11.3.1 beschriebenen Implementierung und wird in diesem Abschnitt beschrieben.

Die Grundlage der implementierten Berechnung ist die Transformation der Geländekoordinaten min[i] zu den Bildkoordinaten 0, und die Transformation von max[i] zu 1. Damit ist sichergestellt, dass die Texturkoordinaten den gesamten Bereich der Bildkoordinaten [0;1] abdecken. Die Berechnung der Texturkoordinaten tCoord[i] erfolgt nach Gleichung (12).

$$tCoord[i] = tCoordVal0[i] + scale[i] \times tc[i] \qquad (12)$$

tCoordVal0[i] beschreibt den Texturkoordinatenwert, bei dem die entsprechenden Geländekoordinaten gleich 0 sind. Dieser Wert wird nach Gleichung (13) berechnet.

$$tCoordVal0[i] = 1 - \frac{\max[i]}{\max[i] - \min[i]} \tag{13}$$

Scale[i] ergibt sich, für beliebige max[i] und min[i] Werte, aus einer gleichmäßigen Verteilung der Geländewerte auf die Bildskala [0;1]. Daraus ergibt sich Gleichung (14).

$$scale[i] = \frac{1}{\max[i] - \min[i]} \tag{14}$$

Abb. 31 zeigt eine schematische Darstellung der beschriebenen Berechnung.

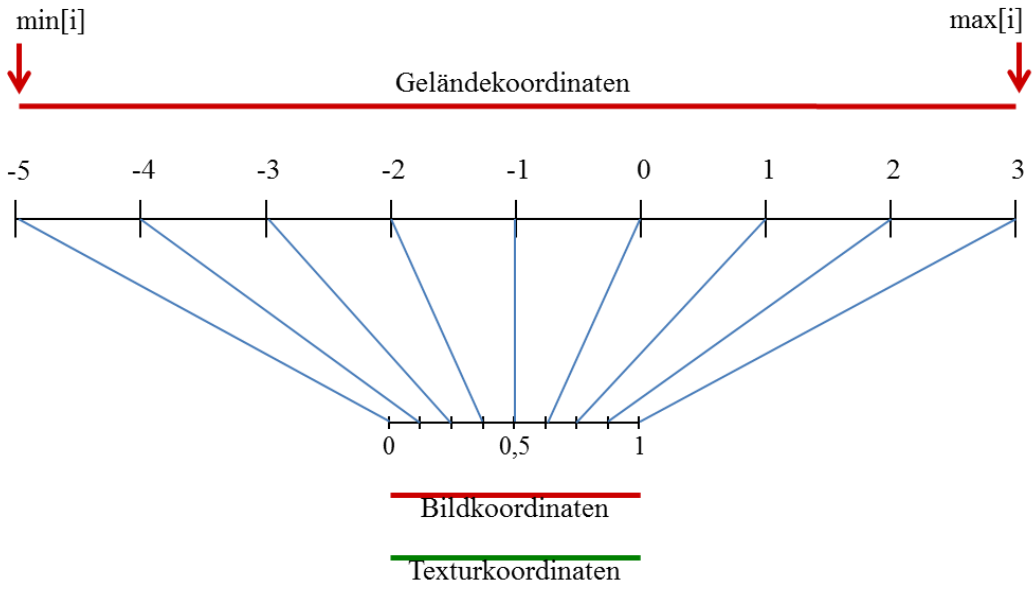


Abb. 31 Transformation der Gelände-Punktkoordinaten zu Texturkoordinaten für die Texturierung einer Oberfläche mit einem Bild (Gesamter Bildbereich). Schematische Darstellung für Punktkoordinaten [-5,3] in eine Richtung i.

Wie in Abb. 31 zu erkennen, deckt der Texturkoordinatenbereich (grün markiert) den gesamten Bildkoordinatenbereich [0;1] (rot markiert) ab. Damit wird wie benötigt das gesamte Bild auf die Geländeoberfläche gemappt. Die *render window*-Ausgabe der Darstellung der texturierten Oberfläche mit dieser Texturkoordinatenbestimmung ist in Abb. 30 rechts dargestellt. Der zugehörige Code ist in Anhang 6 wiedergegeben.

Die Berechnung wurde, wie hier beschrieben, implementiert, da das vorhandene Bild, welches der Texturierung dient, ein georeferenziertes Luftbild ist, welches zu dem STL-Geländemodell passt. Die Ränder des Bildes und der Oberfläche stimmen überein. Diese Eigenschaft wurde hier genutzt. Die Georeferenzierung wurde nicht verwendet.

Nachdem die Funktionalitäten zu der Visualisierung der CFD-Daten (Abschnitte 8, 9 und 10) und zu der Geländedarstellung (Abschnitt 11) analysiert wurden, erfolgt schließlich in Abschnitt 12 die Erläuterung der Interaktionsimplementierung.

12 Echtzeit-Rendering

VTK ermöglicht eine Interaktion bzw. Veränderung der Modelldaten während der Laufzeit des Programms, ohne dass der Code verändert und neu erstellt werden muss.

Dazu wird die Klasse vtkRenderWindowInteractor verwendet. Es gibt verschiedene Möglichkeiten um die Interaktionsoptionen festzulegen. Dabei können von VTK definierte Funktionalitäten ausgeführt werden. Es können aber auch dem Programm (der eigenen Anwendung) angepasste Interaktionen implementiert werden. Die Ausführung der definierten Programmteile wird durch so genannte Events ausgelöst. Dabei handelt es sich z.B. um Mausinteraktionen oder das Drücken von Tasten. Ein Event signalisiert, dass in der Software etwas maßgebendes geschehen ist (Schroeder, Martin und Lorensen 2006, 64). Observer sind Objekte dessen Interesse für ein oder mehrere Events registriert ist. Bei dem Auslösen eines Events wird der Observer benachrichtigt und kann die entsprechenden Anweisungen ausführen (Schroeder, Martin und Lorensen 2006, 64). In den folgenden Abschnitten wird weiter darauf eingegangen, wobei zwischen selbstimplementierten Events und von VTK vordefinierten Events unterschieden wird.

12.1 Implementierung selbstdefinierter Events und Prinzip der Interaktion

Zunächst wird das Prinzip der Interaktion kurz geschildert, um dann genauer auf Codeeinheiten einzugehen.

Wie in dem VTK Quellcode in vtkRenderWindowInteractor.h beschrieben, verfolgt vtkRenderWindowInteractor Events durch VTKs „command/observer design pattern“. Das heißt, dass wenn eine der Unterklassen von vtkRenderWindowInteractor ein Plattform-abhängiges Event „sieht“, dieses in ein VTK Event mit der InvokeEvent()-Methode übersetzt wird. In diesem

Zusammenhang ist es wichtig, zuerst die Methode vtkRenderWindowInteractor::Initialize() aufzurufen um den vtkRenderWindowInteractor zu aktivieren (Enable()) und anschließend vtkRenderWindowInteractor::Start(). Letztere Methode startet die Event-Schleife.

Jedem beliebigen Objekt kann mit vtkObject::AddObserver() `unsigned long vtkObject::AddObserver(const char *event,vtkCommand *cmd, float p)` ein Observer zugewiesen werden. Der erste Übergabeparameter bezeichnet das zu beobachtende Event, welches die Ausführung eines bestimmten Code-Teils bewirken soll, z.B. RightButtonPressEvent. Der zweite Übergabeparameter ist ein Zeiger auf ein Objekt einer Unterklasse von vtkCommand cmd (Erläuterungen zu vtkCommand folgen im Laufe dieses Abschnittes.). p bezeichnet die Priorität (priority) falls mehrere Events gleichzeitig entstehen. Diese muss nicht angegeben werden. Wenn nicht anders definiert, wird der zuletzt hinzugefügte Observer/Event als erstes ausgeführt.

In vtkObject.cxx ist eine Klasse vtkSubjectHelper definiert welche eine Liste der Observer verwaltet. Durch vtkObject::AddObserver() wird die Methode vtkSubjectHelper::AddObserver() aufgerufen, wodurch ein Objekt der Klasse vtkObserver erzeugt wird und in die Liste eingefügt wird.

Am Anfang dieses Abschnittes wurde geschildert, dass bei einem Event die Methode InvokeEvent() aufgerufen wird. Diese ist in vtkObject.cxx als Funktion der Klasse vtkSubjectHelper definiert, welche durch vtkObject::InvokeEvent() aufgerufen wird.

An dieser Stelle ist es interessant zu beachten, welche Klassen von vtkObject abgeleitet sind. Dazu gehört zum Beispiel auch vtkRenderWindowInteractor. Damit kann dem RenderWindowInteractor-Objekt zum Beispiel eine Reihe von Observern für verschiedene Events zugeordnet werden, wie es für das in dieser Arbeit erstellte Modell gemacht wurde.

Bevor die InvokeEvent()-Methode weiter erläutert werden kann ist es notwendig zunächst die Klasse vtkCommand zu untersuchen.

Die Klasse vtkCommand wird zur Programmierung der Codeteile, die bei bestimmten Events auszuführen sind, verwendet. Eine Liste der definierten Events ist in vtkCommand.h zu finden. Für das erstellte Modell von Bedeutung sind zum Beispiel „KeyPressEvent“, „LeftButtonPressEvent“ oder „TimerEvent“. vtkCommand ist eine abstrakte Klasse. Um Objekte dieser Klasse erzeugen zu können muss eine Unterklasse von vtkCommand implementiert werden. Dabei

muss die rein virtuelle Funktion Execute() definiert werden. Diese beinhaltet für jedes Event den auszuführenden Code.

Die InvokeEvent()-Methode `int vtkSubjectHelper::InvokeEvent(unsigned long event, void *callData,vtkObject *self)` ruft mit der Codezeile `elem->Command->Execute(self,event,callData);` die Prozedur Execute() auf. Dabei ist elem ein Zeiger auf vtkObserver (Auch die Klasse vtkObserver ist in vtkObject.cxx definiert.) mit dem Attribut Command vom Typ vtkCommand*. Für das Objekt auf welches Command zeigt, d.h. ein Objekt einer Unterklasse von vtkCommand, wird schließlich die Methode Execute() aufgerufen. Der erste Übergabeparameter (self) ist der vtkObject* für den die Anweisung mit AddObserver() implementiert wurde.

Damit wird der für ein Event programmierte Code ausgeführt. In der implementierten Anwendung ist der Code in VisualisierungCFD.cpp zu finden. Die von vtkCommand abgeleitete Klasse ist als vtkInteraktionEchtzeit definiert.

12.2 In VTK vorhandene Interaktionsmöglichkeiten

In Abschnitt 12.1 wurde beschrieben, dass eine Unterklasse der Klasse vtkCommand implementiert werden kann um Funktionalitäten für bestimmte Events Modellspezifisch zu definieren. VTK verfügt auch über vtkInteractorStyle Klassen, welche Methoden für bestimmte Events beinhalten. Für das erstellte Modell wurde diese Art der Interaktionsimplementierung herangezogen, um bestimmte Kamerafunktionalitäten während der Laufzeit des Programms zu erhalten.

12.2.1 Implementierung

Zur Implementierung muss ein Objekt der Klasse vtkInteractorStyle erzeugt werden. Dieses wird dann dem Objekt der Klasse vtkRenderWindowInteractor mit der Memberfunktion SetInteractorStyle() zugewiesen.

Durch die Methode `void vtkRenderWindowInteractor::SetInteractorStyle (vtkInteractorObserver *style)` werden die hier aufgeführten Anweisungen (aus vtkRenderWindowInteractor.cxx) ausgeführt: `this->InteractorStyle = style; this->InteractorStyle->SetInteractor(this);` InteractotStyle ist ein Attribut der Klasse vtkRenderWindowInteractor mit dem Datentyp vtkInteractorObserver*. Wie in Abb. 32 dargestellt ist vtkInteractorStyle eine Unterklasse von vtkInteractorObserver. Für das erstellte Modell wurde in SetInteractorStyle() als

Übergabeparameter ein vtkInteractorStyle übergeben. Daher wird die Methode SetInteractor() aus vtkInteractorStyle.cxx ausgeführt.

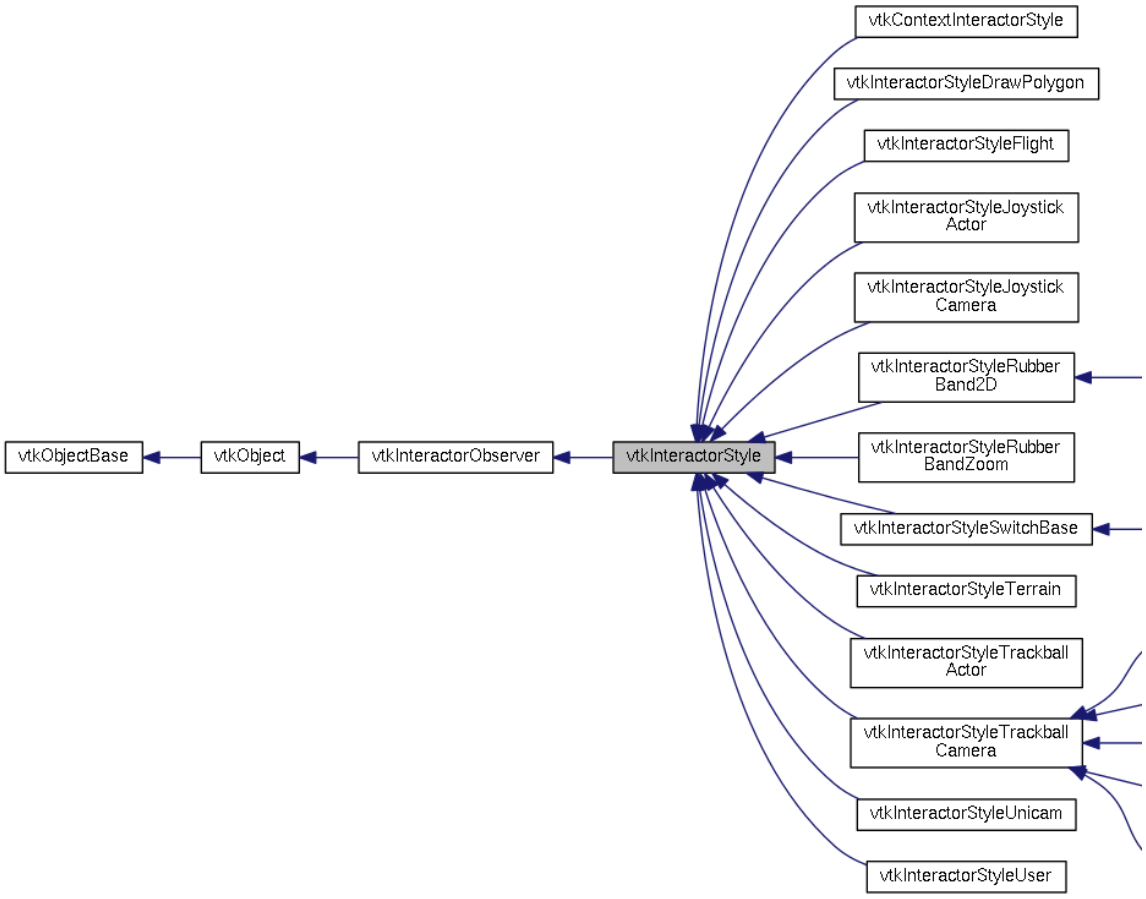


Abb. 32 Vererbung für vtkInteractorStyle (Ausschnitt) nach (vtk.org 2014 g)

In dieser Methode wird für das Objekt von vtkRenderWindowInteractor (zuvor this) die Funktion AddObserver() für verschiedene Events aufgerufen. Wie in 12.1 beschrieben, wird als erster Übergabeparameter das Event übergeben. An zweiter Stelle wird ein Objektzeiger einer von vtkCommand abgeleiteten Klasse erwartet. Hier ist die Unterklasse vtkCallbackCommand. Das verwendete Objekt dieser Klasse ist in vtkInteractorObserver.h wie folgt deklariert: `vtkCallbackCommand* EventCallbackCommand;` EventCallbackCommand wird in AddObserver() übergeben. Wie in 12.1 wird hiermit eine Observerliste erstellt. Wie zuvor werden auch die Anweisungen vtkRenderWindowInteractor::Initialize() und vtkRenderWindowInteractor::Start() eingebracht.

Der Unterschied besteht in der Implementierung der Codeabschnitte, die bei den Events ausgeführt werden. In 12.1 wurde eine Unterklasse von vtkCommand

erstellt. Jetzt wird die in VTK vorhandene abgeleitete Klasse vtkCallbackCommand verwendet. Diese definiert die Methode vtkCallbackCommand::Execute() als Aufruf von vtkCallbackCommand::Callback() (vtkCallbackCommand.cxx). Mit vtkCallbackCommand::SetCallback() wird eine Methode angegeben, die die Event-Codeabschnitte enthält (vtkCallbackCommand.h).

Diese Methode wird in vtkInteractorStyle.cxx mit dem Konstruktor festgelegt: `this->EventCallbackCommand->SetCallback(vtkInteractorStyle::ProcessEvents);` Die Prozedur vtkInteractorStyle::ProcessEvents() wird schließlich bei der Detektion eines Events ausgeführt.

Wie aus Abb. 32 ersichtlich, sind in VTK eine Reihe von Klassen definiert, die von vtkInteractorStyle abgeleitet sind. Für das erstellte Modell wurde vtkInteractorStyleTrackballCamera verwendet. In den Unterklassen werden die Funktionalitäten der Methode vtkInteractorStyle::ProcessEvents() weiter implementiert. In folgendem Abschnitt werden die Funktionen ausgewählter, von vtkInteractorStyle abgeleiteter Klassen beschrieben. Für die in dieser Arbeit verwendete Klasse werden einige in VTK implementierte Events, und hiermit Interaktionsmöglichkeiten des erstellten Modells, aufgezeigt.

12.2.2 Implementierte Interaktionen und Interaktionsstile

In diesem Abschnitt wird auf die Unterklassen vtkInteractorStyleTrackballCamera und vtkInteractorStyleSwitch eingegangen. Die Klasse vtkInteractorStyleSwitch implementiert den Wechsel zwischen vtkInteractorStyleTrackballCamera, vtkInteractorStyleTrackballActor, vtkInteractorStyleJoystickCamera und vtkInteractorStyleJoystickActor durch Drücken definierter Tasten. Diese Klassen können auch einzeln verwendet werden und implementieren folgende Funktionalitäten.

vtkInteractorStyleJoystickCamera definiert die Mausinteraktionen so, dass die Kamera zum Ansichtswechsel verschoben/gedreht wird. Die Kamera bewegt sich je nach Abstand der Maus zum Zentrum der Szene. Die Verschiebung läuft weiter (z.B. solange die Maustaste gedrückt wird) und die Geschwindigkeit der Mausbewegung bestimmt die Beschleunigung der Kamera (vtkInteractorStyleJoystickCamera.h).

Mit vtkInteractorStyleJoystickActor ist die Art der Funktionen (Rotation, Zoom, etc.) gleich. Hier wird jedoch nicht die Kamera verändert, sondern einzelne Objekte (vtkInteractorStyleJoystickActor.h).

In der Klasse vtkInteractorStyleTrackballActor beziehen sich die Funktionalitäten wieder auf bestimmte, von einander unabhängige, Objekte. Mit der Trackball-Interaktion sind jedoch die Größe der Mausbewegung (z.B. während die linke Maustaste gedrückt wird) und die Actor-Bewegung proportional (vtkInteractorStyleTrackballActor.h).

Die Trackball-Interaktion, mit Bewegung der Kamera statt eines Actors, ist in vtkInteractorStyleTrackballCamera implementiert (vtkInteractorStyleTrackballCamera.h).

Für das erstellte Modell ist es sinnvoll, die Interaktionen mit der Größe der Mausbewegung steuern zu können. Daher ist eine Trackball-Interaktion zu verwenden. Eine Relativverschiebung der Objekte der Szene ist bei dem Ansichtswechsel nicht erwünscht. Hier soll sich die Kamera in der Szene Bewegen. Damit wird für das Modell die Klasse vtkInteractorStyleTrackballCamera verwendet, um die Ansicht während der Laufzeit des Programms zu verändern.

Zusätzlich werden auch mit der in Abschnitt 12.1 geschilderten Methode Modellspezifische Interaktionen definiert (Abschnitt 7 -> Tabelle 1). Dazu zählt zum Beispiel die Veränderung der Wasseroberfläche durch Aktualisierung der verwendeten VTS-Daten. Wie auch in (Kitware 2004, 43f) erwähnt, sollte zur Erstellung eigener Eventmethoden, der standard „interaction style“ deaktiviert werden. Für das Modell wurde überprüft, dass trotz der Kombination, keine Events mehrere verschiedene Funktionen ausführen. Die hinzugefügten Funktionalitäten werden bei den Events *TimerEvent* und *KeyPressEvent* aufgerufen. Die Funktionen, die zu diesen Events in VTK definiert sind werden in vtkInteractorStyle::ProcessEvents() aufgerufen und wurden in vtkInteractorStyle.cxx untersucht. Von Bedeutung ist OnTimer() welche bei den vorhandenen Einstellungen keine Wirkung hat. OnKeyDown(), OnKeyUp(), OnKeyPress() und OnKeyRelease() werden in der Unterklasse von vtkInteractorStyle implementiert. In vtkInteractorStyleTrackballCamera ist das nicht der Fall. Zu beachten ist folglich die Funktion vtkInteractorStyle::OnChar(). Codeteile werden bei dem Drücken bestimmter Tasten ausgeführt. Diese Tasten dürfen nicht mit neuen, selbst definierten Funktionen, assoziiert werden. Die betroffenen Tasten sind: m, M, q, Q, e, E, f, F, u, U, r, R, w, W, s, S, 3, p und P.

In Tabelle 2 werden einige wichtige Interaktionsmöglichkeiten aufgeführt, die das Modell durch Verwendung von vtkInteractorStyleTrackballCamera aufweist. Die bei Events ausgeführten Funktionen (in vtkInteractorStyle::ProcessEvents() implementiert) rufen Methoden auf, die bestimmte Wirkungen implementieren (in

vtkInteractorStyle.cxx wie z.B. OnChar() oder in vtkInteractorStyleTrackballCamera.cxx wie z.B. OnLeftButtonDown()). Der Inhalt letzterer Funktionen kann der Spalte „Wirkung“ entnommen werden (Tabelle 2). Die Prozeduren Rotate(), Spin(), Pan() und Dolly(), die dabei aufgerufen werden, (in vtkInteractorStyleTrackballCamera.cxx programmiert) implementieren schließlich die Kameramethoden zur Ansichtsveränderung (siehe Spalte „Beschreibung der Wirkung“ in Tabelle 2). Diese Kamerabewegungsmethoden sind zur Veranschaulichung in Abb. 33 bildlich dargestellt.

Tabelle 2 Interaktionen zur Echtzeit-Ansichtsveränderung durch Nutzung der Klasse vtkInteractorStyleTrackballCamera

Interaktion (Aufruf eines Events)	**Wirkung**	**Beschreibung der Wirkung**
‘Q‘ / ‘q‘ / ‘e‘ / ‘E‘	Schließen des Programms (exit)	
‘W‘ / ‘w‘	„Wireframe“ Darstellung aller *Actors*, die dem *Renderer* zugewiesen sind	
‘S‘ / ‘s‘	„Surface“ Darstellung aller *Actors*, die dem *Renderer* zugewiesen sind	
LeftButton	vtkInteractorStyle TrackballCamera::Rotate()	vtkCamera::Azimuth() vtkCamera ::Elevation() vtkCamera::OrthogonalizeViewUp()
LeftButton + Shift	vtkInteractorStyle TrackballCamera::Pan()	vtkCamera ::SetFocalPoint() vtkCamera ::SetPosition() „focal point“ und „position“ werden um den gleichen Betrag verschoben
LeftButton + Shift + Ctrl	vtkInteractorStyle TrackballCamera::Dolly()	vtkCamera ::SetPosition() => Bewegung der Kamera hin oder weg von dem *focal point*
LeftButton + Ctrl	vtkInteractorStyle TrackballCamera::Spin()	vtkCamera ::Roll() vtkCamera::OrthogonalizeViewUp()
MiddleButton	vtkInteractorStyle TrackballCamera::Pan()	vtkCamera ::SetFocalPoint() vtkCamera ::SetPosition() *focal point* und *position* werden um den gleichen Betrag verschoben
RightButton	vtkInteractorStyle TrackballCamera::Dolly()	vtkCamera ::SetPosition() => Bewegung der Kamera hin oder weg von dem *focal point*
MouseWheelForward / MouseWheelBackward	vtkInteractorStyle TrackballCamera::Dolly()	vtkCamera ::SetPosition() => Bewegung der Kamera hin oder weg von dem *focal point*

In Tabelle 2 ist zu beachten, dass zur Vereinfachung z.B. LeftButton als Zusammenfassung der Interaktionen LeftButtonDown, MouseMove und LeftButtonUp verwendet wurde. Entsprechend wird z.B. Rotate() durch StartRotate() angefangen und durch EndRotate() beendet. Die Bewegung der Kamera passiert in diesem Beispiel mit MouseMove, d.h. solange die linke Maustaste gedrückt gehalten wird.

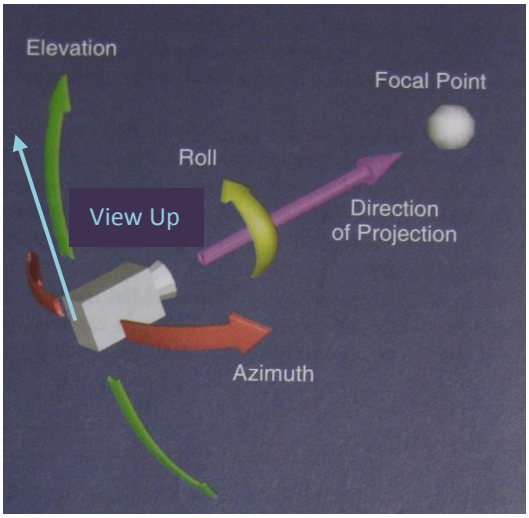


Abb. 33 Darstellung ausgewählter Attribute und Methoden der Klasse vtkCamera nach (Schroeder, Martin und Lorensen 2006, 45)

13 Fazit und Ausblick

Durch die Nutzung von VTK konnte eine Anwendung implementiert werden, welche eine interaktive Visualisierung von realistisch eingebundenen CFD-Ergebnissen (Wasseroberflächen-Geschwindigkeitswerten) modelliert. Auf die Anpassungsfähigkeit des Programms zur Visualisierung weiterer Standorte wurde geachtet, wobei die Anwendung für einen Schachtkraftwerks-Standort am Lech fertiggestellt wurde.

VTK bietet dazu eine große Auswahl an Visualisierungsmethoden und Funktionalitäten. Das Softwaresystem ermöglicht zudem eine Kombination verschiedener Datenformate (Import und Export), sodass die Nutzung verschiedener Programme zu der Lösung der Aufgabe beitragen konnte. Auch existierende, mit VTK implementierte Anwendungen (wie ParaView), können bei der Implementierung zu Hilfe genommen werden.

Im Vergleich zu Software wie Blender ist es bei VTK etwas schwieriger Dokumentation zu bestimmten Funktionalitäten im Netz zu finden. Jedoch können durch Verwendung verschiedener Medien (auf die in dieser Arbeit verwiesen wurde) Anwendungen effektiv entwickelt werden.

In der Anwendung CFDVis/VisualisierungCFD wurden in dieser Arbeit zwei Wasseroberflächen aus CFD-Ergebnissen erzeugt, interaktiv visualisiert und ausgetauscht. Diese decken beispielhaft einen Teilbereich des Kraftwerkes ab. Größere Datenmengen konnten nicht eingelesen werden. Im Gespräch mit den Betreuern hat sich herausgestellt, dass der Wechsel von der 32-bit zu einer 64-bit-Version von VTK vermutlich zur Lösung des Problems beitragen würde. Die mit 32-bit kompilierte Anwendung weist tatsächlich auf 64-bit Windows-Systemen eine Speicher-Grenze von 2 GB auf (msdn.microsoft.com 2014).

Die Möglichkeiten der implementierten Anwendung könnten durch Einlesen kompletter und einer größeren Anzahl an CFD-Datensätzen (in ihrer Funktionalität) besser ausgeschöpft werden.

Das in dieser Arbeit erstellte Modell ist die Grundlage zur Visualisierung wasserbaulich geprägter Standorte. Weitere Funktionalitäten könnten mit VTK hinzugefügt werden. Dazu zählt z.B. die Möglichkeit, Flussbereiche im Schnitt (Querschnitt) darzustellen (Filter „Clip“ in ParaView in Kombination mit (Kitware 2004, 95)). Die, durch die Texturierung der Wasseroberflächen, wiedergegebenen Werte der CFD-Berechnung können ausgetauscht werden, um z.B. Druckvisualisierungen zu ermöglichen.

Im Gespräch mit den Betreuern wurden Aufgabenbereiche skizziert, die im Wasserbau immer mehr an Bedeutung gewinnen könnten.

In diesem Sinne wäre es interessant Modellierungstools aus Blender zur Erstellung von in sich beweglichen Objekten zu nutzen. Die Darstellung von Fischen könnte z.B. vorgenommen werden (Stiller 2011, 48ff und 319ff). Dabei wäre im Voraus zu überprüfen, ob und in welcher Form die Übernahme solcher Elemente mit VTK in der Anwendung unterstützt wird.

Eine interessante Visualisierungsaufgabe könnte weiter darin bestehen, Zell- und Punktattributdaten dazu zu verwenden, um bestimmte Pfade festzulegen, die die Position einzelner Modellobjekte vorgeben. Damit würden z.B. Fische je nach Geschwindigkeit der Strömung auf definierten Pfaden im Fluss bewegt. So könnten verschiedene Szenarien in Abhängigkeit der CFD-Daten dargestellt werden. Ansätze zur Bestimmung der Pfade könnten folgende Aspekte in Betracht ziehen.

Mit definierten Filtern werden Zell- und Punktdaten eines Bereiches extrahiert, z.B. für einen Flussquerschnitt. Daraus könnten Maximalwerte gesucht werden und die Koordinaten derer Zellen zu der Platzierung von Objekten genutzt werden. Andere Filter wie vtkThreshold (Anhang 2) können zur Extraktion von Zellen herangezogen werden, deren Attributwerte in bestimmten Bereichen liegen. Damit werden Regionen geortet, in denen bestimmte Strömungseigenschaften vorherrschen, welche für die Position betrachteter Objekte von Bedeutung sind.

Literatur

Breymann, Ulrich. *C++ Einführung und professionelle Programmierung.* München Wien: Carl Hanser Verlag, 2007.

Feller, David. *Digital Painting. Digitale Kunstwerke mit Adobe Photoshop erstellen.* Heidelberg, München, Landsberg, Frechen, Hamburg: mitp-Verlag, 2010.

Kitware, Inc. *The VTK User's Guide. Install, Use and Extend The Visuualization Toolkit (VTK 4.4).* United States of America: Kitware, Inc., 2004.

Maschke, Thomas. *Digitale Bildbearbeitung. Bildbearbeitung, Farbmanagement, Bildausgabe.* Berlin Heidelberg: X.media.press / Springer-Verlag, 2004.

msdn.microsoft.com. 2014. http://msdn.microsoft.com/en-us/library/windows/desktop/aa366778(v=vs.85).aspx#memory_limits (Zugriff am 12. März 2014).

paraview.org. 2014. http://www.paraview.org/ (Zugriff am 11. März 2014).

Schroeder, Will, Ken Martin, und Bill Lorensen. *The Visualization Toolkit. An Object-Oriented Approach to 3D Graphics.* Colombia: Kitware, Inc., 2006.

Stiller, Heiner. *Blender 2.5. 3D-Modellierung und -Animation.* Poing: Franzis Verlag GmbH, 2011.

Tu, J., G. Yeoh, und C. Liu. *Computational Fluid Dynamics. A Practical Approach.* Oxford: Butterworth-Heinemann publications, 2013.

vtk.org. 2014 g. http://www.vtk.org/doc/nightly/html/classvtkInteractorStyle.html (Zugriff am 11. Februar 2014).

vtk.org. 2014 a. http://www.vtk.org/ (Zugriff am 4. März 2014).

vtk.org. 2014 b. http://www.vtk.org/doc/nightly/html/classvtkSmartPointer.html (Zugriff am 4. März 2014).

vtk.org. 2014 c. http://www.vtk.org/Wiki/VTK/Tutorials/SmartPointers (Zugriff am 4. März 2014).

vtk.org. 2014 d. http://www.vtk.org/VTK/project/license.html (Zugriff am 5. März 2014).

vtk.org. 2014 e. www.vtk.org/Wiki/VTK/Examples/Cxx (Zugriff am 9. März 2014).

vtk.org. 2014 f. www.vtk.org/doc/nightly/html/classvtkLookupTable.html (Zugriff am 9. März 2014).

vtk.org. 2014 h. http://www.vtk.org/VTK/resources/applications.html (Zugriff am 11. März 2014).

wiki.blender.org. 2014 a. http://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts/Import-Export/STL (Zugriff am 11. März 2014).

wiki.blender.org. 2014 b. http://wiki.blender.org/index.php/Doc:2.6/Manual/Extensions/Python/Add-Ons (Zugriff am 11. März 2014).

wiki.blender.org. 2014 c. http://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts/Import-Export/Wavefront_OBJ (Zugriff am 11. März 2014).

wirtschftslexikon.gabler.de. 2014. http://wirtschaftslexikon.gabler.de/Definition/softwaresystem.html#erklaerung (Zugriff am 4. März 2014).

www2.le.ac.uk. 2014. http://www2.le.ac.uk/departments/engineering/research/thermofluids/research/computational-fluid-dynamics-1 (Zugriff am 10. März 2014).

Abbildungsverzeichnis

Abb. 1 Darstellung eines CFD-Ergebnis-Datensatzes mit ParaView. Färbung der Zellen durch f [0;1] 15
Abb. 2 Visualisierungs-Pipeline / Ausführung des Netzwerkes als „implicit execution process“ wie es in VTK implementiert ist (Schroeder, Martin und Lorensen 2006, 104) 17
Abb. 3 Fehlermeldung bei fehlender Einbeziehung von Header-Dateien 20
Abb. 4 Färbung eines VTS-Datensatzes mit ParaView. Anpassung des Wertebereiches der Farbskala 23
Abb. 5 Ausschnitte eines *Python script* aus ParaView (oben) und dessen Ausgabe (unten) 25
Abb. 6 Renderansicht / Ausgabe von CFDVis/VisualisierungCFD. Implementierte Modellelemente 29
Abb. 7 Beispielansicht der Textausgabe des Programmes CFDVis/VisualisierungCFD 30
Abb. 8 VisualisierungCFD.cpp: Bereich zur Eingabe der für das Modell zu verwendenden Daten 31
Abb. 9 vtkStructuredGrid Darstellung mit ParaView 38
Abb. 10 Textverarbeitungsprogramm-Anzeige einer VTS-Datei ohne Attributdaten 38
Abb. 11 Textverarbeitungsprogramm-Anzeige einer VTS-Datei mit Attributdaten 39
Abb. 12 Konvertierung von Zell- zu Punktdaten mit vtkCellDataToPointData 40
Abb. 13 Klassen, die bei der Berechnung der Punktattributdaten aus Zellattributdaten mit VTK einbezogen werden (Inhalte aus VTK Quellcode ermittelt). Mit farbiger Schrift sind einzelne Inhalte der darüber angegebenen Methoden wiedergegeben. Die gestrichelten Pfeile stellen die Einbeziehung verschiedener Klassen dar. 41
Abb. 14 Wireframe-Ansicht der VTS-Daten für die Beispielberechnung aus ParaView 43
Abb. 15 Ausgabe der Zell- (links) und der Punktattributwerte (rechts) in ParaView nach Berechnung der Punktattributwerte aus den Zellattributwerten 43
Abb. 16 Färbung der VTS-Daten in ParaView durch f-Punktattributwerte, die aus den Zellwerten berechnet wurden. (Darstellung der Zellgrenzen durch blaue Striche) 44
Abb. 17 Darstellung der VTS-CellData f (links) und der VTS-PointData f (rechts) als Färbung in ParaView / Auswirkung der Konvertierung mit vtkCellDataToPointData 45
Abb. 18 Färbung einer Oberfläche durch Geschwindigkeitsattributdaten: oben mit Zelldaten, unten mit Punktdaten / Renderansicht zu einer Implementierung mit VTK 45
Abb. 19 Darstellung von VTS-Daten mit Färbung durch Punktattributdaten f und Erstellung einer Isosurface (f=0,5) durch den Filter Contour mit ParaView 46
Abb. 20 Färbung des Objektes in Abhängigkeit von den Zellskalarwerten (hier Geschwindigkeitswerte v) in ParaView. Die Skala zeigt die Zuweisung der Farbwerte für definierte Geschwindigkeiten. 47
Abb. 21 Ansicht aus ParaView wie in Abb. 20 für die Färbung durch Punktskalarwerte. 48
Abb. 22 Farbbestimmung mit Blender 49
Abb. 23 Bestimmung eines Bildfarbwertes mithilfe von Blender 50
Abb. 24 Konvertierung von HSV- zu RGB-Farbwerten (nach vtkMath.cxx) 51
Abb. 25 Zusammenstellung wichtiger Attribute und Methoden der Klasse vtkLookupTable (->vtkLookupTable.h und vtkLookupTable.cxx). (Die Darstellung ist keine vollständige Aufführung der Klasse und weist keine korrekte Anwendung der UML-Notation auf. Die Abbildung dient hier dem Verständnis und soll einen Überblick über den Abschnitt erleichtern.) 53

Abb. 26 Darstellung der in ForceBuild() berechneten Farben der vtkLookupTable für die drei Interpolationsfunktionen am Beispiel des Rotkanals. Der Farbverlauf geht von schwarz bis weiß, mit 256 Farben in der Tabelle. 59
Abb. 27 Unterschiede der Texturierung einer Oberfläche mit einer vtkLookupTable zwischen linearer, *SQRT* und *S-Curve* Interpolation zur Berechnung der Farbwerte der Tabelle / Renderansicht zu einer Implementierung mit VTK 60
Abb. 28 Graphische Darstellung der Methode vtkLookupTable::GetIndex(double v) (vtkLookupTable.cxx) zur Berechnung des Index der Tabellenfarbe für einen gegebenen skalaren Wert v 62
Abb. 29 Transformation der Gelände-Punktkoordinaten zu Texturkoordinaten wie in der Klasse vtkImplicitTextureCoords implementiert. Schematische Darstellung für Punktkoordinaten [-5,3] in eine Richtung i. 66
Abb. 30 Projektion eines Bildes auf eine Geländeoberfläche (Ansicht von oben) mit der Transformation der Geländepunktkoordinaten zu Texturkoordinaten links wie in vtkImplicitTextureCoords implementiert (Bildausschnitt) und rechts zum Mappen des gesamten Bildes 67
Abb. 31 Transformation der Gelände-Punktkoordinaten zu Texturkoordinaten für die Texturierung einer Oberfläche mit einem Bild (Gesamter Bildbereich). Schematische Darstellung für Punktkoordinaten [-5,3] in eine Richtung i. 68
Abb. 32 Vererbung für vtkInteractorStyle (Ausschnitt) nach (vtk.org 2014 g) 72
Abb. 33 Darstellung ausgewählter Attribute und Methoden der Klasse vtkCamera nach (Schroeder, Martin und Lorensen 2006, 45) 76

Anhang 1 Einlesen der VTS-Daten mit VTK

Der folgende Code kann eigenständig ausgeführt werden und zeigt die Ein- und Ausgabe von VTS-Daten. Nach dem Code wird die Ausgabe beispielhaft dargestellt.

```
#include <vtkSmartPointer.h>
#include <vtkXMLStructuredGridReader.h>
#include <vtkXMLStructuredGridWriter.h>
#include <vtkStructuredGrid.h>

int main()
{
        //EINLESEN DER VTS-DATEN:

//Übergabe des Dateinamen (.vts) an die Variable inputFilename.
//Die Datei muss in dem Ordner liegen, in dem die EXE-Datei des Projektes zu finden ist.
std::string inputFilename = "output_0.vts";
//Einlesen der VTS-Datei mit einem Objekt der Klasse vtkXMLStructuredGridReader:
//Dabei ist die Konvertierung von inputFilename zu einem const character array mit c_str() wichtig.
vtkSmartPointer<vtkXMLStructuredGridReader> reader =
  vtkSmartPointer<vtkXMLStructuredGridReader>::New();
reader->SetFileName(inputFilename.c_str());
reader->Update();
//Mit reader->getoutput() wird die StructuredGrid zurückgegeben.
//Welche Attributdaten (CellData und PointData) eingelesen werden, wird wie folgt definiert.

//Einlesen bestimmter, ausgewählter Attributdaten:
//SetCellArrayStatus(name,status): name gibt den Datensatz (CellData) an.
        //Damit können einzelne Attributdatensätze gewählt werden.
        //name ist in der VTS-Datei unter CellData als z.B. "Name="f"" angegeben.
        //status kann gleich 1 oder gleich 0 gesetzt werden:
        //Falls status=1, werden die Zellattribute name eingelesen.
        //Falls status=0, werden sie nicht eingelesen.
//Für die Punktdaten (PointData) gibt es entsprechend die Funktion SetPointArrayStatus.
int nbCellArr=reader->GetNumberOfCellArrays();
for (int j=0;j<nbCellArr;j++)
{
  reader->SetCellArrayStatus(reader->GetCellArrayName(j),1); //Einlesen der gesamten VTS-Datei.
  std::cout<<"cellName "<<j+1<<": "<<reader->GetCellArrayName(j)<<std::endl;
  std::cout<<"cellStatus "<<j+1<<": "<<reader->GetCellArrayStatus(reader-
>GetCellArrayName(j))<<std::endl;
}
reader->Update();
//Schließlich gibt reader eine StructuredGrid zurück, die die definierten Punkt- und Zelldaten
//enthält.

//Zur Veranschaulichung können die Daten des reader in einer VTS-Datei ausgegeben werden.
//Dies wird mit einem Objekt der Klasse vtkXMLStructuredGridWriter durchgeführt:
vtkSmartPointer<vtkXMLStructuredGridWriter> writer =
  vtkSmartPointer<vtkXMLStructuredGridWriter>::New();
//Die Daten werden in eine VTS-Datei mit dem in der nächsten Anweisung gegebenen Namen gespeichert.
//Diese erzeugte Datei wird in dem Ordner, der die EXE-Datei des Projektes enthält, gespeichert.
writer->SetFileName("OutputReadVTSFile.vts");
writer->SetInputData(reader->GetOutputDataObject(0));
writer->Write();

//Bestätigung der CellArrayStatus-Ausgabe:
std::system("PAUSE");
        return 0;
}
```

Ausgabe der CellArrayStatus

Da alle vorhandenen Attributdaten eingelesen wurden, sind die wie folgt angegebenen Werte der eingelesenen VTS-Datei `"output_0.vts"` und der Ausgabedatei `"OutputReadVTSFile.vts"` identisch.

```
<?xml version="1.0"?>
<VTKFile type="StructuredGrid" version="0.1"
byte_order="LittleEndian" compressor="vtkZLibDataCompressor">
  <StructuredGrid WholeExtent="0 110 0 160 0 25">
    <Piece Extent="0 110 0 160 0 25">
      <PointData>
      </PointData>
      <CellData>
        <DataArray type="Float32" Name="f" format="appended"
RangeMin="0"                    RangeMax="1"
offset="0"                   />
        <DataArray type="Float32" Name="p" format="appended"
RangeMin="-9348.9248047"        RangeMax="83394.40625"
offset="33508"               />
        <DataArray type="Float32" Name="bac" format="appended"
RangeMin="0"                    RangeMax="5.1626081467"
offset="92924"               />
        <DataArray type="Float32" Name="vf" format="appended"
RangeMin="0"                    RangeMax="1"
offset="308804"              />
        <DataArray type="Float32" Name="ijkmob" format="appended"
RangeMin="0"                    RangeMax="3"
offset="386264"              />
        <DataArray type="Float32" Name="v" NumberOfComponents="3"
format="appended" RangeMin="0"
RangeMax="13.060861912"         offset="401164"              />
      </CellData>
      <Points>
        <DataArray type="Float32" Name="Points"
NumberOfComponents="3" format="appended" RangeMin="225384.6521"
RangeMax="226008.96227"         offset="686564"              />
      </Points>
```

```
    </Piece>
  </StructuredGrid>
  <AppendedData encoding="base64">
```

Textverarbeitungsprogramm-Anzeige der Ausgabedatei "OutputReadVTSFile.vts"

Anhang 2 Visualisierung eines definierten VTS-Daten Ausschnittes mit vtkThreshold

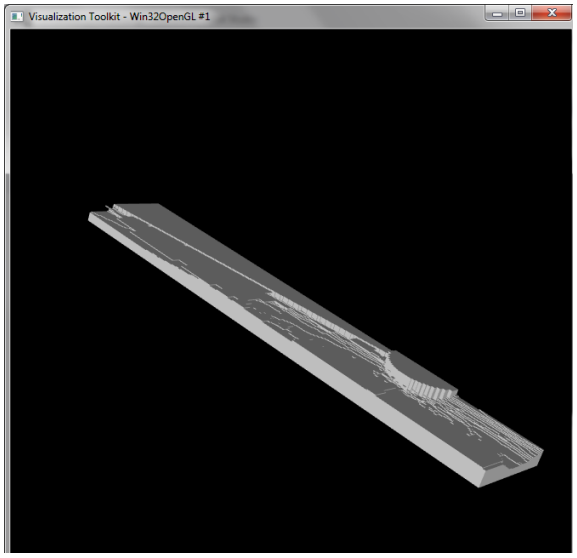
Mit diesem Anhang soll die Möglichkeit, Daten nach einem bestimmten Attributwert zu filtern und darzustellen, erläutert werden. Mit einem Objekt der Klasse vtkThreshold werden bestimmte Zellen der eingelesenen Daten extrahiert, die eine definierte Bedingung erfüllen. Hier werden als Kriterium zwei Werte für einen der Attributdatensätze angegeben, zwischen denen die Attributdaten liegen sollen. Die Ausgabe ist eine vtkUnstructuredGrid die schließlich visualisiert werden kann.

```
#include <vtkSmartPointer.h>
#include <vtkPolyDataMapper.h>
#include <vtkXMLStructuredGridReader.h>
#include <vtkRenderWindow.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkRenderer.h>
#include <vtkThreshold.h>
#include <vtkUnstructuredGrid.h>
#include <vtkGeometryFilter.h>

int main()
{
        //VISUALISIERUNG EINES DEFINIERTEN VTS-DATEN AUSSCHNITTES:

//Einlesen der VTS-Datei wie in Anhang1:
std::string inputFilename = "output_0.vts";
vtkSmartPointer<vtkXMLStructuredGridReader> reader =
  vtkSmartPointer<vtkXMLStructuredGridReader>::New();
reader->SetFileName(inputFilename.c_str());
reader->Update();

int nbCellArr=reader->GetNumberOfCellArrays();
for (int j=0;j<nbCellArr;j++)
{
  reader->SetCellArrayStatus(reader->GetCellArrayName(j),1);
}
reader->Update();


//Der folgende, auskommentierte Abschnitt zeigt eine
//Möglichkeit, die Attributdaten, die eingelesen werden sollen,
//durch eine Abfrage des Attributnamens zusätzlich einzustellen.
//Dabei ist es wichtig zu beachten, dass durch die Nutzung von Pointern
//der Vergleich der Zellattributnamen pro character vorgenommen werden muss.

/*//Abfrage welche Attributdaten eingelesen werden sollen, und Abfrage
//der Länge der Eingabe für den Vergleich des CellArray Namen:
  std::string DataName;
  std::cout<<"Name der CellData, die dargestellt werden sollen: ";
  std::cin>>DataName;
  const char*DataN=DataName.c_str();

  int nD=0;
  std::cout<<"Anzahl der Buchstaben des Namens der Daten: ";
  std::cin>>nD;
```

```
  bool TF; //Variable (true/false) für den Vergleich der einzelnen char
  int Sum=1; //Variable (true/false) für den Vergleich der CellArrayName

  for(int j=0;j<nbCellArr;j++)
  {
        Sum=1;
        //Da die for-Schleifen über wenige Elemente laufen
        //wurde an dieser Stelle nicht auf eine effizientere
        //Lösung zur if-Abfrage geachtet.
        for (int k=0;k<(nD+1);k++)
        {
           TF=(reader->GetCellArrayName(j))[k]==DataN[k];
           if(TF==0)
           {
                   Sum=0;
                   break;
           }
        }
        if(Sum==0)
        {
                  reader->SetCellArrayStatus(reader->GetCellArrayName(j),0);
        }
  }
  reader->Update();
//Ausgabe der CellArrayStatus, um die Auswahl der Daten die eingelesen werden
//sollen zu veranschaulichen:
for (int j=0;j<nbCellArr;j++)
  {
  std::cout<<"cellName "<<j<<": "<<reader->GetCellArrayName(j)<<std::endl;
  std::cout<<"cellStatus "<<j<<": "<<reader->GetCellArrayStatus(reader-
>GetCellArrayName(j))<<std::endl;
  }*/

//Extraktion der Zellen für CellData Name=p und p zwischen 0 und 1000:
vtkSmartPointer<vtkThreshold> threshold =
   vtkSmartPointer<vtkThreshold>::New();
//Angabe des gewollten Datenbereiches:
threshold->ThresholdBetween(0.0,1000.0);
//Input der eingelesenen Daten, von denen Zellen extrahiert werden sollen
threshold->SetInputConnection(reader->GetOutputPort());
//Bei der folgenden Anweisung wird als letzter Übergabeparameter der
//Name der Attributdaten eingegeben, deren Skalarwerte zur Auswahl
//der zu extrahierenden Zellen dienen.
//vtkDataObject::FIELD_ASSOCIATION_CELLS gibt an, dass es sich
//hier um CellData handelt. Entsprechend gibt es für
//PointData vtkDataObject::FIELD_ASSOCIATION_POINTS.
threshold->SetInputArrayToProcess(0, 0, 0, vtkDataObject::FIELD_ASSOCIATION_CELLS, "p");
threshold->Update();

//Übergabe des threshold-Outputs vtkUnstructuredGrid
vtkSmartPointer<vtkUnstructuredGrid> thresholdedCells =
   vtkSmartPointer<vtkUnstructuredGrid>::New();
thresholdedCells =threshold->GetOutput();

//vtkGeometryFilter kann herangezogen werden um eine
//Konvertierung zu vtkPolyData zu erreichen.
//Hier: Konvertierung von vtkUnstructuredGrid zu vtkPolyData
//um die Daten anschließend zu visualisieren
vtkSmartPointer<vtkGeometryFilter> FilterUnstr =
   vtkSmartPointer<vtkGeometryFilter>::New();
FilterUnstr->SetInputData(thresholdedCells);
FilterUnstr->Update();
vtkSmartPointer<vtkPolyData> thresholdpolydata =
   vtkSmartPointer<vtkPolyData>::New();
thresholdpolydata = FilterUnstr->GetOutput();

//Visualisierung:
vtkSmartPointer<vtkPolyDataMapper> mapper =
   vtkSmartPointer<vtkPolyDataMapper>::New();
mapper->SetInputConnection(FilterUnstr->GetOutputPort());

vtkSmartPointer<vtkActor> actor =
   vtkSmartPointer<vtkActor>::New();
actor->SetMapper(mapper);
```

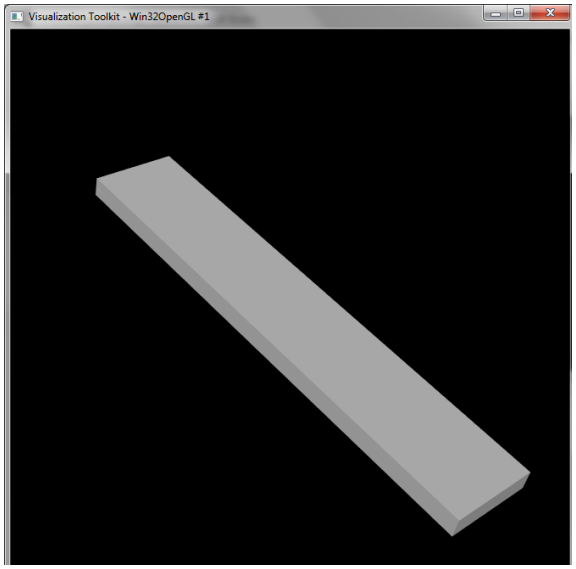
```
vtkSmartPointer<vtkRenderer> renderer =
   vtkSmartPointer<vtkRenderer>::New();
vtkSmartPointer<vtkRenderWindow> renderWindow =
   vtkSmartPointer<vtkRenderWindow>::New();
renderWindow->AddRenderer(renderer);
vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor =
   vtkSmartPointer<vtkRenderWindowInteractor>::New();
renderWindowInteractor->SetRenderWindow(renderWindow);

renderer->AddActor(actor);

renderWindow->Render();
renderWindowInteractor->Start();

  return 0;
}
```

Die Ausgabe der extrahierten Zellen für p zwischen 0 und 1000 (wie in dem aufgeführten Code) ist wie folgt.



Ausgabe des dargestellten Codes (vtkThreshold mit p zwischen 0 und 1000)

Zum Vergleich kann der Datenbereich für p in der VTS-Datei nachgelesen werden und bei `threshold->ThresholdBetween(0.0,1000.0);` eingegeben werden. Damit wird eine Darstellung aller Zellen der VTS-Daten erreicht. Folgende Darstellung wird ausgegeben.



Ausgabe der Darstellung aller Zellen der VTS-Daten

Anhang 3 Attributdaten-Konvertierung mit vtkCellDataToPointData

Um VTS-Daten optimal weiterverarbeiten zu können ist es essentiell das gewünschte Datenformat zu verwenden. Dazu wird der Code für die Erzeugung von PointData aus CellData erläutert.

```
#include <vtkSmartPointer.h>
#include <vtkXMLStructuredGridReader.h>
#include <vtkCellDataToPointData.h>
#include <vtkXMLStructuredGridWriter.h>

int main()
{
        //ATTRIBUTDATEN-KONVERTIERUNG MIT vtkCellDataToPointData:

//Einlesen der VTS-Daten wie in Anhang1:
std::string inputFilename = "output_0.vts";
vtkSmartPointer<vtkXMLStructuredGridReader> reader =
   vtkSmartPointer<vtkXMLStructuredGridReader>::New();
reader->SetFileName(inputFilename.c_str());
reader->Update();

int nbCellArr=reader->GetNumberOfCellArrays();
for (int j=0;j<nbCellArr;j++)
 {
 reader->SetCellArrayStatus(reader->GetCellArrayName(j),1);
 }
reader->Update();

//Mit dem Objekt der Klasse vtkCellDataToPointData werden die
//vorhandenen CellData der eingelesenen VTS-Datei zu
//PointData konvertiert.
vtkSmartPointer<vtkCellDataToPointData> readerPoint =
   vtkSmartPointer<vtkCellDataToPointData>::New();
//Übergabe der eingelesenen VTS-Daten deren CellData zu
//PointData konvertiert werden sollen.
readerPoint->SetInputConnection(reader->GetOutputPort());
//Mit folgender Anweisung werden die CellData dem Output übergeben, sodass
//zusätzlich zu den erzeugten PointData in dem Output auch die Input
//CellData vorhanden sind.
readerPoint->PassCellDataOn();
//Mit readerPoint->PassCellDataOff(); wären nur die erzeugten PointData in
//dem Output vorhanden.
```

```
//Wie in Anhang1 kann die Wirkung von vtkCellDataToPointData
//durch die Ausgabe als VTS-Datei veranschaulicht werden.
vtkSmartPointer<vtkXMLStructuredGridWriter> writer =
   vtkSmartPointer<vtkXMLStructuredGridWriter>::New();
writer->SetFileName("OutputPointData.vts");
writer->SetInputConnection(readerPoint->GetOutputPort());
writer->Write();

        return 0;
}
```

Die ausgegebene VTS-Datei `"OutputPointData.vts"` mit `readerPoint->PassCellDataOn();` zeigt die Zelldaten und die zugehörigen berechneten Punktdaten.

```
<?xml version="1.0"?>
<VTKFile type="StructuredGrid" version="0.1"
byte_order="LittleEndian" compressor="vtkZLibDataCompressor">
  <StructuredGrid WholeExtent="0 110 0 160 0 25">
    <Piece Extent="0 110 0 160 0 25">
      <PointData>
        <DataArray type="Float32" Name="f" format="appended"
RangeMin="0"                    RangeMax="1"
offset="0"                    />
        <DataArray type="Float32" Name="p" format="appended"
RangeMin="-5230.6489258"        RangeMax="81539.078125"
offset="77792"                />
        <DataArray type="Float32" Name="bac" format="appended"
RangeMin="0"                    RangeMax="3.4194049835"
offset="226128"               />
        <DataArray type="Float32" Name="vf" format="appended"
RangeMin="0"                    RangeMax="1"
offset="556120"               />
        <DataArray type="Float32" Name="ijkmob" format="appended"
RangeMin="0"                    RangeMax="3"
offset="714456"               />
        <DataArray type="Float32" Name="v" NumberOfComponents="3"
format="appended" RangeMin="0"
RangeMax="12.555208646"         offset="743404"               />
      </PointData>
      <CellData>
        <DataArray type="Float32" Name="f" format="appended"
RangeMin="0"                    RangeMax="1"
offset="1349972"              />
        <DataArray type="Float32" Name="p" format="appended"
RangeMin="-9348.9248047"        RangeMax="83394.40625"
offset="1383480"              />
        <DataArray type="Float32" Name="bac" format="appended"
RangeMin="0"                    RangeMax="5.1626081467"
offset="1442896"              />
        <DataArray type="Float32" Name="vf" format="appended"
RangeMin="0"                    RangeMax="1"
offset="1658776"              />
        <DataArray type="Float32" Name="ijkmob" format="appended"
RangeMin="0"                    RangeMax="3"
offset="1736236"              />
```

```
        <DataArray type="Float32" Name="v" NumberOfComponents="3"
format="appended" RangeMin="0"
RangeMax="13.060861912"         offset="1751136"             />
      </CellData>
      <Points>
        <DataArray type="Float32" Name="Points"
NumberOfComponents="3" format="appended" RangeMin="225384.6521"
RangeMax="226008.96227"         offset="2036536"             />
      </Points>
    </Piece>
  </StructuredGrid>
```

Textverarbeitungsprogramm-Anzeige der Ausgabedatei "OutputPointData.vts" mit PassCellDataOn()

Folgende Abbildung veranschaulicht die Wirkung der Konvertierung indem die Eingangsdaten dem Output nicht übergeben werden.

```
<?xml version="1.0"?>
<VTKFile type="StructuredGrid" version="0.1"
byte_order="LittleEndian" compressor="vtkZLibDataCompressor">
  <StructuredGrid WholeExtent="0 110 0 160 0 25">
    <Piece Extent="0 110 0 160 0 25">
      <PointData>
        <DataArray type="Float32" Name="f" format="appended"
RangeMin="0"                    RangeMax="1"
offset="0"                   />
        <DataArray type="Float32" Name="p" format="appended"
RangeMin="-5230.6489258"        RangeMax="81539.078125"
offset="77792"               />
        <DataArray type="Float32" Name="bac" format="appended"
RangeMin="0"                    RangeMax="3.4194049835"
offset="226128"              />
        <DataArray type="Float32" Name="vf" format="appended"
RangeMin="0"                    RangeMax="1"
offset="556120"              />
        <DataArray type="Float32" Name="ijkmob" format="appended"
RangeMin="0"                    RangeMax="3"
offset="714456"              />
        <DataArray type="Float32" Name="v" NumberOfComponents="3"
format="appended" RangeMin="0"
RangeMax="12.555208646"         offset="743404"              />
      </PointData>
      <CellData>
      </CellData>
      <Points>
        <DataArray type="Float32" Name="Points"
NumberOfComponents="3" format="appended" RangeMin="225384.6521"
RangeMax="226008.96227"         offset="1349972"             />
      </Points>
    </Piece>
  </StructuredGrid>
```

Textverarbeitungsprogramm-Anzeige der Ausgabedatei "OutputPointData.vts" mit PassCellDataOff()

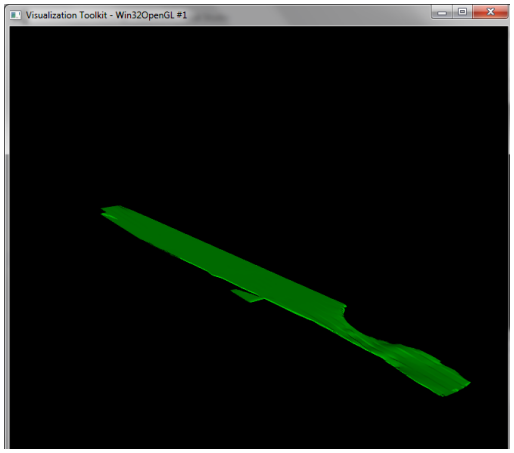
Anhang 4 Erzeugung einer Oberfläche aus einem vtkStructuredGrid mit der Klasse vtkContourFilter

Aufbauend auf dem Code der zuvor präsentierten Anhänge, werden die eingelesenen Daten in diesem Abschnitt, zur Erzeugung einer Oberfläche definierter Höhe genutzt.

```
#include <vtkSmartPointer.h>
#include <vtkPolyDataMapper.h>
#include <vtkXMLStructuredGridReader.h>
#include <vtkRenderWindow.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkRenderer.h>
#include <vtkCellDataToPointData.h>
#include <vtkContourFilter.h>

int main()
{
        //ERSTELLUNG EINER OBERFLÄCHE MIT vtkContourFilter:

//Einlesen der VTS-Daten wie in Anhang1:
std::string inputFilename = "output_0.vts";
vtkSmartPointer<vtkXMLStructuredGridReader> reader =
   vtkSmartPointer<vtkXMLStructuredGridReader>::New();
reader->SetFileName(inputFilename.c_str());
reader->Update();

int nbCellArr=reader->GetNumberOfCellArrays();
for (int j=0;j<nbCellArr;j++)
 {
 reader->SetCellArrayStatus(reader->GetCellArrayName(j),1);
 }
reader->Update();

//Konvertierung der CellData zu PointData wie in Anhang3:
vtkSmartPointer<vtkCellDataToPointData> readerPoint =
   vtkSmartPointer<vtkCellDataToPointData>::New();
readerPoint->SetInputConnection(reader->GetOutputPort());
readerPoint->PassCellDataOn();

//Erstellung einer Oberfläche für die gilt: PointData f=0,5.
//Dazu wird ein Objekt der Klasse vtkContourFilter erzeugt.
vtkSmartPointer<vtkContourFilter> Oberfl =
   vtkSmartPointer<vtkContourFilter>::New();
//Die VTS-Daten mit Punktattributwerten werden übergeben.
Oberfl->SetInputConnection(readerPoint->GetOutputPort());
Oberfl->SetNumberOfContours(1);
//Die folgende Anweisung dient der Festlegung des
//Attributdaten-Wertes bei dem die Fläche erzeugt werden soll.
//Hier 0,5 für den Wert, den f für die Fläche aufweisen soll.
//(zweiter Übergabeparameter).
//Der erster Übergabeparameter ist die Contour-Nummer.
//(Diese ist zwischen 0 und NumberOfContours nicht inbegriffen. )
Oberfl->SetValue(0,0.5);
//Wie in Anhang2 bei dem Objekt threshold, wird wie folgt
//angegeben, dass f der Name der PointData ist, die für die
//Erzeugung der Fläche maßgebend sind.
Oberfl->SetInputArrayToProcess(0, 0, 0, vtkDataObject::FIELD_ASSOCIATION_POINTS, "f");
Oberfl->Update();

//Die erzeugte Oberfläche kann dann visualisiert werden, was einen Vergleich
//mit einer mit ParaView erzeugten Oberfläche (Filters->Contour) ermöglicht.
vtkSmartPointer<vtkPolyDataMapper> mapper =
   vtkSmartPointer<vtkPolyDataMapper>::New();
mapper->SetInputConnection(Oberfl->GetOutputPort());

vtkSmartPointer<vtkActor> actor =
   vtkSmartPointer<vtkActor>::New();
actor->SetMapper(mapper);
```

```
vtkSmartPointer<vtkRenderer> renderer =
   vtkSmartPointer<vtkRenderer>::New();
vtkSmartPointer<vtkRenderWindow> renderWindow =
   vtkSmartPointer<vtkRenderWindow>::New();
renderWindow->AddRenderer(renderer);
vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor =
   vtkSmartPointer<vtkRenderWindowInteractor>::New();
renderWindowInteractor->SetRenderWindow(renderWindow);

renderer->AddActor(actor);

renderWindow->Render();
renderWindowInteractor->Start();

         return 0;
}
```

Die Ausgabe der visualisierten Oberfläche ist wie folgt. Die Farbgebung der Oberfläche wird an dieser Stelle nicht berücksichtigt.



Mit vtkContourFilter aus VTS-Daten erstellte Oberfläche

Anhang 5 Texturierung einer Oberfläche mit einem Bild / Berechnung der Texturkoordinaten nach vtkImplicitTextureCoords

Wie in 11.3.1 beschrieben, konnte die hier aufgeführte Implementierung nicht für das Mappen des gesamten vorhandenen Luftbildes auf die gesamte Geländeoberfläche (Randübereinstimmung) angewendet werden. Für andere Aufgaben, zur Anpassung an andere Vorgaben und zum Verständnis, sind der Code und der Aufbau der Implementierung jedoch an dieser Stelle exemplarisch dargestellt. Dazu wurde die Geländekoordinatentransformation nach vtkImplicitTextureCoords durchgeführt, um die Projektion des Luftbildes auf das Geländemodell darzustellen. Das Ergebnis ist in Abb. 30 links dargestellt.

```
#include <vtkPolyData.h>
#include <vtkSTLReader.h>
#include <vtkSmartPointer.h>
#include <vtkPolyDataMapper.h>
#include <vtkActor.h>
#include <vtkRenderWindow.h>
#include <vtkRenderer.h>
```

```
#include <vtkRenderWindowInteractor.h>
#include <vtkTIFFReader.h>
#include<vtkTexture.h>
#include<vtkFloatArray.h>
#include<vtkPointData.h>

int main ()
{
        //PROJEKTION EINES BILDES AUF EINE OBERFLÄCHE
        //MIT SKALIERUNG DER TEXTURKOORDINATEN NACH
        //vtkImplicitTextureCoords.cxx

//Einlesen des Luftbildes welches als Textur verwendet werden soll
vtkSmartPointer<vtkTIFFReader> readerLB =
   vtkSmartPointer<vtkTIFFReader>::New();
readerLB->SetFileName ( "4421055_5352737.tif" );

//Einlesen des Geländemodells als zu texturierende Oberfläche
std::string inputFilenameGel = "Gelaende.stl";

vtkSmartPointer<vtkSTLReader> readerGel =
   vtkSmartPointer<vtkSTLReader>::New();
readerGel->SetFileName(inputFilenameGel.c_str());
readerGel->Update();

vtkSmartPointer<vtkPolyData> GelPoly=
   vtkSmartPointer<vtkPolyData>::New();
GelPoly=readerGel->GetOutput();

int nGelPts=GelPoly->GetNumberOfPoints();

//Berechnung der Texturkoordinaten und Speicherung dieser als
//Geländepunkt-Attributdaten.
//Dazu Skalierung und Verschiebung der Geländekoordinaten mit
//Berechnung nach Implementierung in vtkImplicitTextureCoords.
  //START CODE aus vtkImplicitTextureCoords.cxx kopiert und verändert:

  vtkIdType ptId;
  vtkSmartPointer<vtkFloatArray> newTCoords =
     vtkSmartPointer<vtkFloatArray>::New();
  double min[3], max[3], scale[3];
  double tCoord[3], tc[3], x[3];
  int i;

// Allocate
//tCoord[3] dient der Speicherung der Geländekoordinaten für einen Punkt (3D)
  tCoord[0] = tCoord[1] = tCoord[2] = 0.0;

//newTCoords dient der Speicherung von tCoord[3] für das gesamte
//Gelände (nGelPts Punkte)
    newTCoords->SetNumberOfComponents(3);
    newTCoords->Allocate(3*nGelPts);

//(Funktionsberechnungen wurden herausgenommen.) Stattdessen werden die
//Punktkoordinaten aufgerufen und als vorläufige Texturkoordinaten
//in newTCoords eingefügt.
//Bestimmung der minimalen (min[i]) und maximalen (max[i])
//Geländekoordinatenwerte für jede der 3 Dimensionen
  for (i=0; i<3; i++) //initialize min/max values array
    {
    min[i] = VTK_DOUBLE_MAX;
    max[i] = -VTK_DOUBLE_MAX;
    }

//Bestimmung von min[i] und max[i] (3D)
  for (ptId=0; ptId<nGelPts; ptId++)
    {
    GelPoly->GetPoint(ptId, x);
    tCoord[0] = x[0];
    tCoord[1] = x[1];
    tCoord[2] = x[2];
```

```
    for (i=0; i<3; i++)
      {
      if (tCoord[i] < min[i])
        {
        min[i] = tCoord[i];
        }
      if (tCoord[i] > max[i])
        {
        max[i] = tCoord[i];
        }
      }

    newTCoords->InsertTuple(ptId,tCoord);
    }

//Transformation der Geländekoordinaten zu Texturkoordinaten
//in dem Bereich (0,1).
//Punktkoordinaten 0,0 werden zu dem Texturkoordinatenwert 0,5 transformiert.
  for (i=0; i<3; i++)
    {
    scale[i] = 1.0;
    if ( max[i] > 0.0 && min[i] < 0.0 ) //have positive & negative numbers
      {
      if ( max[i] > (-min[i]) )
        {
        scale[i] = 0.499 / max[i]; //scale into 0.5->1
        }
      else
        {
        scale[i] = -0.499 / min[i]; //scale into 0->0.5
        }
      }
    else if ( max[i] > 0.0 ) //have positive numbers only
      {
      scale[i] = 0.499 / max[i]; //scale into 0.5->1.0
      }
    else if ( min[i] < 0.0 ) //have negative numbers only
      {
      scale[i] = -0.499 / min[i]; //scale into 0.0->0.5
      }
    }

  for (ptId=0; ptId<nGelPts; ptId++)
    {
//Speicherung der realen Geländekoordinaten in tc für den gegebenen Punkt
     newTCoords->GetTuple(ptId, tc);
    for (i=0; i<3; i++)
      {
      tCoord[i] = 0.5 + scale[i] * tc[i];
      }
//Speicherung der transformierten Geländekoordinaten (0,1) in newTCoords
//für alle Geländepunkte.
    newTCoords->InsertTuple(ptId,tCoord);
    }

//Speicherung der erzeugten Texturkoordinaten newTCoords als
//Punkt-Attributdaten TCoords des vtkPolyData GelPoly:
  GelPoly->GetPointData()->SetTCoords(newTCoords);

  //END CODE aus vtkImplicitTextureCoords.cxx kopiert und verändert.

//Erzeugung der Textur
vtkSmartPointer<vtkTexture> textureGel =
   vtkSmartPointer<vtkTexture>::New();
textureGel->SetInputConnection(readerLB->GetOutputPort());

//Visualisierung/Ausgabe
vtkSmartPointer<vtkPolyDataMapper> mapperGel =
   vtkSmartPointer<vtkPolyDataMapper>::New();
mapperGel->SetInputData(GelPoly);
mapperGel-
>SetInputArrayToProcess(0,0,0,vtkDataObject::FIELD_ASSOCIATION_POINTS,vtkDataSetAttributes::TCOORDS
);
```

```
vtkSmartPointer<vtkActor> actorGel =
   vtkSmartPointer<vtkActor>::New();
actorGel->SetMapper(mapperGel);
actorGel->SetTexture(textureGel);

vtkSmartPointer<vtkRenderer> renderer =
   vtkSmartPointer<vtkRenderer>::New();
renderer->AddActor(actorGel);
renderer->SetBackground(.3, .6, .3); // Background color green
renderer->ResetCamera();

vtkSmartPointer<vtkRenderWindow> renderWindow =
   vtkSmartPointer<vtkRenderWindow>::New();
renderWindow->AddRenderer(renderer);
vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor =
   vtkSmartPointer<vtkRenderWindowInteractor>::New();
renderWindowInteractor->SetRenderWindow(renderWindow);

renderWindow->Render();

renderWindowInteractor->Start();

  return 0;
}
```

Anhang 6 Texturierung einer Oberfläche mit einem Bild / Berechnung der Texturkoordinaten zur Überlagerung der Ränder des Bildes und der Oberfläche

Die in 11.3.2 beschriebene Berechnung zur Geländekoordinatentransformation zu Texturkoordinaten wird hier für die Projektion des Luftbildes auf die Geländeoberfläche angewendet und, wie in folgendem Code ersichtlich, implementiert. Die Ausgabe der Visualisierung ist in Abb. 30 rechts dargestellt.

```
#include <vtkPolyData.h>
#include <vtkSTLReader.h>
#include <vtkSmartPointer.h>
#include <vtkPolyDataMapper.h>
#include <vtkActor.h>
#include <vtkRenderWindow.h>
#include <vtkRenderer.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkTIFFReader.h>
#include<vtkTexture.h>
#include<vtkFloatArray.h>
#include<vtkPointData.h>

int main ()
{
        //TEXTURIERUNG/PROJEKTION EINES BILDES AUF EINE OBERFLÄCHE
        //MIT TRANSFORMATION DER TEXTURKOORDINATEN ZUR ÜBEREINSTIMMUNG
        //DER RÄNDER DES BILDES UND DER OBERFLÄCHE

//Einlesen des Luftbildes welches als Textur gemappt wird
vtkSmartPointer<vtkTIFFReader> readerLB =
   vtkSmartPointer<vtkTIFFReader>::New();
readerLB->SetFileName ( "4421055_5352737.tif" );

//Einlesen des Geländes als zu texturierende Oberfläche
std::string inputFilenameGel = "Gelaende.stl";

vtkSmartPointer<vtkSTLReader> readerGel =
   vtkSmartPointer<vtkSTLReader>::New();
readerGel->SetFileName(inputFilenameGel.c_str());
readerGel->Update();
```

```
vtkSmartPointer<vtkPolyData> GelPoly=
   vtkSmartPointer<vtkPolyData>::New();
GelPoly=readerGel->GetOutput();

int nGelPts=GelPoly->GetNumberOfPoints();

//START CODE aus vtkImplicitTextureCoords.cxx kopiert und verändert:

  vtkIdType ptId;
  vtkSmartPointer<vtkFloatArray> newTCoords =
     vtkSmartPointer<vtkFloatArray>::New();
  double min[3], max[3], scale[3], tCoordVal0[3];
  double tCoord[3], tc[3], x[3];
  int i;

// Allocate
//tCoord[3] dient der Speicherung der Geländekoordinaten für einen Punkt (3D)
  tCoord[0] = tCoord[1] = tCoord[2] = 0.0;

//newTCoords dient der Speicherung von tCoord[3]
//für das gesamte Gelände (nGelPts Punkte)
    newTCoords->SetNumberOfComponents(3);
    newTCoords->Allocate(3*nGelPts);

//Speicherung der Gelände-Punktkoordinaten -> Einfügen in
//newTCoords als Anfangs-Textur-Koordinaten.
//Bestimmung der minimalen und maximalen Geländekoordinatenwerte
//für jede der 3 Dimensionen.
  for (i=0; i<3; i++) //Initialisierung min/max values array
    {
    min[i] = VTK_DOUBLE_MAX;
    max[i] = -VTK_DOUBLE_MAX;
    }
  for (ptId=0; ptId<nGelPts; ptId++)
    {
    //Speicherung der Geländepunktkoordinaten in x[3]
    //und Übergabe an tCoord[3] zur Speicherung in newTCoords
    GelPoly->GetPoint(ptId, x);
    tCoord[0] = x[0];
    tCoord[1] = x[1];
    tCoord[2] = x[2];

    for (i=0; i<3; i++)
      {
      if (tCoord[i] < min[i])
        {
        min[i] = tCoord[i];
        }
      if (tCoord[i] > max[i])
        {
        max[i] = tCoord[i];
        }
      }

    newTCoords->InsertTuple(ptId,tCoord);
    }
//Skalierung der Geländekoordinaten damit diese in [0,1] liegen.
//tCoordVal0[3] bezeichnet den Texturkoordinatenwert, bei dem die
//Geländekoordinaten gleich 0 sind.
//Die folgende Berechnung der Texturkoordinaten [0,1] transformiert
//die Geländekoordinaten so, dass min[i]=0 und max[i]=1.
  for (i=0; i<3; i++)
    {
    scale[i] = 1.0;
    scale[i] = 1.0 / (max[i]-min[i]);
        tCoordVal0[i] = 1.0 - (max[i] / (max[i] - min[i]) );
    }
```

```
  for (ptId=0; ptId<nGelPts; ptId++)
    {
     //Speicherung der realen Geländekoordinaten in tc für den gegebenen Punkt
     newTCoords->GetTuple(ptId, tc);
    for (i=0; i<3; i++)
      {
      tCoord[i] = tCoordVal0[i] + scale[i] * tc[i];
      }
    //Speicherung der skalierten Geländekoordinaten [0,1] in newTCoords
    //für alle Geländepunkte.
    newTCoords->InsertTuple(ptId,tCoord);
    }

//Speicherung der erzeugten Geländekoordinaten [0,1] newTCoords als
//Punkt-Attributdaten TCoords des vtkPolyData GelPoly:
  GelPoly->GetPointData()->SetTCoords(newTCoords);

//END CODE aus vtkImplicitTextureCoords.cxx kopiert und verändert.

//Erzeugung der Textur
vtkSmartPointer<vtkTexture> textureGel =
   vtkSmartPointer<vtkTexture>::New();
textureGel->SetInputConnection(readerLB->GetOutputPort());

//Visualisierung/Ausgabe
vtkSmartPointer<vtkPolyDataMapper> mapperGel =
   vtkSmartPointer<vtkPolyDataMapper>::New();
mapperGel->SetInputData(GelPoly);
mapperGel->SetInputArrayToProcess(0,0,0,vtkDataObject::FIELD_ASSOCIATION_POINTS,
vtkDataSetAttributes::TCOORDS);

vtkSmartPointer<vtkActor> actorGel =
   vtkSmartPointer<vtkActor>::New();
actorGel->SetMapper(mapperGel);
actorGel->SetTexture(textureGel);

vtkSmartPointer<vtkRenderer> renderer =
   vtkSmartPointer<vtkRenderer>::New();
renderer->AddActor(actorGel);
renderer->SetBackground(.3, .6, .3); // Background color green
renderer->ResetCamera();

vtkSmartPointer<vtkRenderWindow> renderWindow =
   vtkSmartPointer<vtkRenderWindow>::New();
renderWindow->AddRenderer(renderer);
vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor =
   vtkSmartPointer<vtkRenderWindowInteractor>::New();
renderWindowInteractor->SetRenderWindow(renderWindow);

renderWindow->Render();

renderWindowInteractor->Start();

  return 0;
}
```

Anhang 7 URL-Sammlung verwendeter Codebeispiele (nach Themen geordnet)

Einlesen von STL-Dateien:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/IO/ReadSTL

Einlesen von TIFF-Dateien:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/IO/ReadTIFF

Einlesen von PNG-Dateien:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/IO/PNGReader

Einlesen von VTS-Dateien:

http://www.paraview.org/Wiki/VTK/Examples/Cxx/IO/ReadStructuredGrid

Ausgabe als VTS-Datei:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/IO/XMLStructuredGridWriter

Weiteres / Hilfestellung:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/IO/SimplePointsReader

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/DataBounds

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/Silhouette

http://www.paraview.org/Wiki/VTK/Examples/Cxx/Meshes/CapClip

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/PointInsideObject

http://www.vtk.org/Wiki/VTK/Examples/Cxx/VolumeRendering/SmartVolumeMapper

Texturierung:

http://www.cmake.org/Wiki/VTK/Examples/Cxx/Visualization/TextureMapImageData

http://www.vtk.org/Wiki/VTK/Examples/Cxx/Visualization/TextureMapQuad

http://www.new-npac.org/projects/sv2all/sv2/vtk/graphics/examplesPython/motor.py

http://vtk.org/gitweb?p=VTK.git;a=blob;f=Common/DataModel/Testing/Tcl/quadricCut.tcl

Färbung eines Actors:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/Visualization/ColorAnActor

Zugriff auf Punktkoordinaten:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/PolyDataGetPoint

Extraktion des Randes eines Objektes:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/ExternalContour

http://www.cmake.org/Wiki/VTK/Examples/Cxx/Meshes/BoundaryEdges

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/ExtractSelectedIds

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/ExtractSelection

Datenkonvertierung zu vtkPolyData:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/GeometryFilter

Threshold:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/ThresholdingCells

http://www.vtk.org/Wiki/VTK/Examples/Cxx/PolyData/ThresholdingPoints

Schatten:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/Visualization/Shadows

Zugriff auf einen Ordner (zum Einlesen der CFD-Daten verwendet):

http://www.vtk.org/Wiki/VTK/Examples/Cxx/Utilities/DirectoryFileExtensions

Vektor von Actors (zur Speicherung der Wasseroberflächen verwendet):

http://www.vtk.org/Wiki/VTK/Examples/Cxx/Visualization/VectorOfActors

Implementierung der Benutzerinteraktion mit vtkCommand:

http://www.vtk.org/Wiki/VTK/Examples/Cxx/Utilities/Animation

(Zugriffsdatum: 11.2013 - 03.2014)

Anhang 8 Allgemeine Bemerkungen

- Bei Verweisen auf Methoden im Text wurden häufig nach dem Namen leere Klammern verwendet. Die Übergabeparameter wurden zur Vereinfachung oft nicht angegeben, wenn diese in dem gegebenen Kontext zum Verständnis nicht notwendig waren.

- Die in dieser Arbeit entwickelte Anwendung wurde in der schriftlichen Ausarbeitung auch als implementiertes Modell oder mit dem Namen CFDVis/VisualisierungCFD bezeichnet.

- Struktur der implementierten Anwendung auf den beiliegenden DVDs:

Die DVD „CFDVisVersion1.0“ ist die Anwendung (EXE), die programmunabhängig verwendet werden kann. Die DVD beinhaltet einen Ordner „CFDVisVersion1.0“ in dem die Modellanwendung „VisualisierungCFD.exe“ liegt. Weiter sind die eingelesenen Daten und die zum Laufen des Programms benötigten DLL-Dateien in diesem Ordner enthalten.

Die DVD „CFDVisSource“ beinhaltet den gesamten Quellcode und ermöglicht eine Weiterentwicklung der Anwendung. In dem Ordner „CFDVisSource“ ist ein Ordner zu VTK „VTK6.0.0“ vorhanden in dem u.a. der VTK-Quellcode (header- und CXX-Dateien) vorhanden sind. Der Ordner „CFDVisSource\CFDVis“ beinhaltet die Projektdatei „VisualisierungCFD.sln“ und den „Debug“-Ordner, der dem Ordner „CFDVisVersion1.0“ entspricht (EXE + Inputdaten+DLL). Der Pfad zu dem im Rahmen dieser Arbeit implementierten Code der Anwendung ist CFDVisSource\CFDVis\VTSLoad\VisualisierungCFD.cpp.