JÜLICH
FORSCHUNGSZENTRUM

VTK

API zur Visualisierung von (wissenschaftlichen) Daten

10. Juli 2017

Dr. Helmut Schumacher

VTK

- www.vtk.org
- Distribution als Source-Code mit Public License
- unter www.kitware.com gibt es Literatur und Support zu kaufen
- Installationshinweise später

Eigenschaften von VTK

- geschrieben in C++
- Schnittstelle zu Qt
- Wrapper für Java, TCL/TK und Python
- setzt auf OpenGL auf
- Visualisierungs-Pipeline
- Daten- und Prozess-Objekte

Motivation VTK statt OpenGL

- OpenGL kann 3D-Objekte in einer beleuchteten Szene rendern
- Objekte bestehen i.A. aus Flächen (Triangle-Strips) und Eigenschaften
  - Größe, Position, Lichtreflektion
- Daten liegen i.A. an diskreten Punkten vor
- VTK erzeugt aus Daten renderbare Objekte

Installation

- Sourcen und Cmake (von cmake.org) holen
- Cmake kompilieren
- VTK mithilfe von Cmake konfigurieren
- VTK kompilieren
- für Java vtkutils.jar von Ilias holen
- Pfade setzen und in Umgebung einbinden
- Beispieldaten und Dokumentation holen

Prozess-Objekte

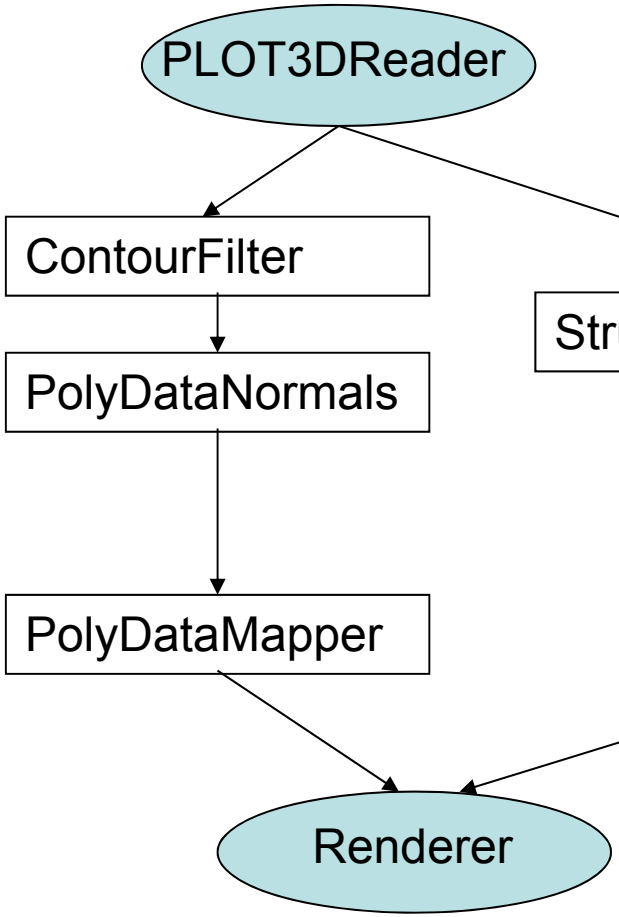
3 Kategorien

- Source objects
  - *prozedural – z.B. SphereSource*
  - *Reader - lesen Dateien*
  - *Importer - holen Daten von Simulationen*
- Filter
  - *überführen Input- in evtl. veränderte Output-Daten*
- Mapper
  - *erzeugen graphische Primitive für Renderer*
  - *Writer - schreiben z.B. Bild- oder Film-Dateien*
  - *Exporter - geben Daten an andere Anwendungen*

Prozess-Objekte (2)

- abgeleitet von (vtk)Algorithm – *das vorangestellte vtk im Namen wird im Folgenden weg gelassen*
- Input (außer Source-Objekte)
- Output (außer Mapper und Renderer)
- Parameter
- aus Daten-Objekten des Inputs und den Parametern werden die Daten-Objekte des Outputs erzeugt

Visualisierungs-Pipeline



bestehend aus

- mindestens einem Source-Objekt
- einer beliebigen Anzahl Filter
- mindestens einen Mapper
- oft einem oder mehreren Renderer

Ausführung der Pipeline

- Wenn das letzte Prozess-Objekt Daten benötigt (Neuzeichnen des Fensters), werden die Erzeuger der Daten rekursiv zum Update aufgefordert.
- Ist der Output veraltet, wird er neu erzeugt
- Sind die Outputs eines Objekts neuer als alle Inputs und Parameter, erfolgt KEIN Update

Input, -Connection, Output und -Port

- Output ist ein Datensatz
- Input verlangt einen Datensatz
  (z.B. den Output eines anderen Moduls)
- Über eine InputConnection, an die der OutputPort eines anderen Moduls angeschlossen wurde, können Requests erfolgen
- b->SetInput(a->GetOutput()) // veraltet
- b->SetInputConnection(a->GetOutputPort())

Programmierhinweise

- Erzeugung von vtk-Objekten nur durch static member function New()
- Referenz-Counter in jedem vtk-Objekt
- Freigabe durch Delete()
  - dekrementiert den Counter
  - Counter = 0 ⇒ Löschen des Objektes
- Squeeze() gibt überflüssigen Speicher frei

Render-Window

Ein Render-Window

- beinhaltet einen oder mehrere Renderer
- beinaltet einen optionalen RenderWindowInteractor
- eröffnet ein Fenster auf dem Ausgabegerät
- stellt den Renderen einen Bereich dieses Fensters zur Verfügung

Renderer

beinhaltet

- Actors (aus Datenobjekten erzeugt)
- eine Kamera
- Lichtquellen
- Hintergrund (Farbe)

defaults für

- Kamera
- Lichtquellen (je eine; ambient, diffus, specular)
- Hintergrund (schwarz)

RenderWindowInteractor

- erlaubt Interaktion im RenderWindow
- reagiert auf
  - Bewegung des Input-Devices
  - Buttons des Input-Devices
  - Keyboard-Tasten
- initiiert Update des RenderWindows

Reaktion auf Input-Device

Button	Aktion	
	Kamera	Objekt
1	Drehung um den Fokus-Punkt	Drehung um den Ursprung bzgl. des Objekts
2	Translation der Kamera	Translation des Objekts
3	Zoom	Skalierung des Objekts

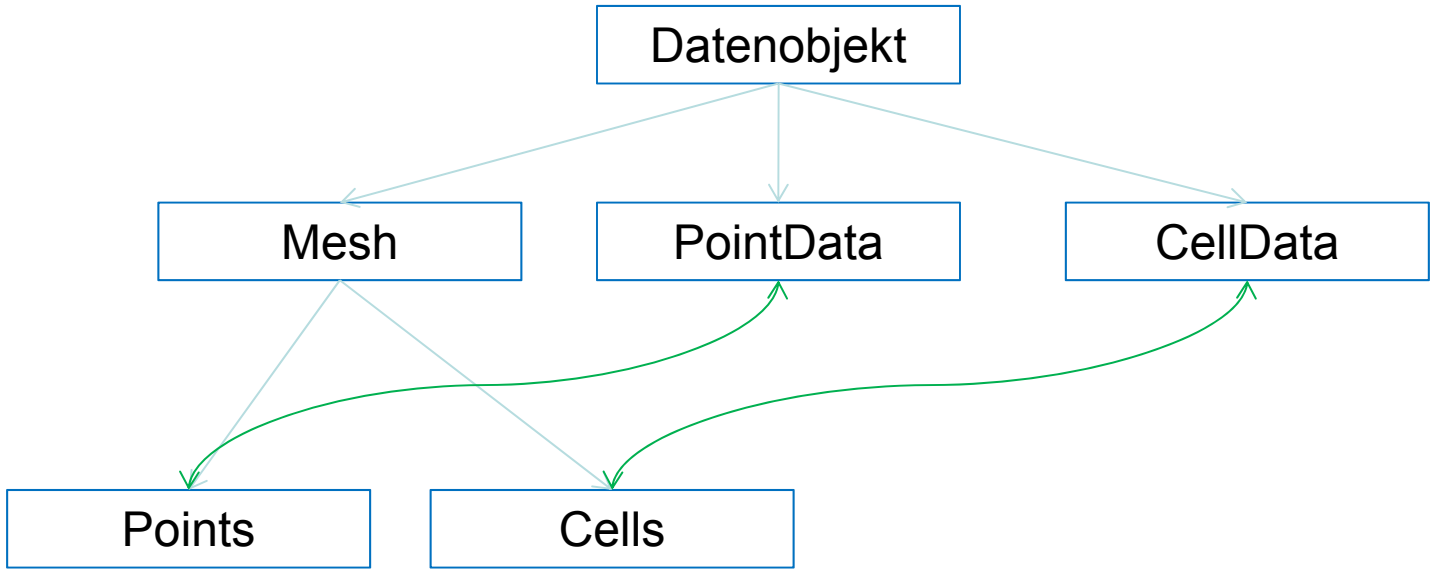
Tastenbefehle Interactor (Auswahl)

Tasten	Aktion
r	Reset
j, t	Bewegung nach Joystick- bzw. Trackball-Modell
s, w	Alle Objekte durch • Oberflächen (Surfaces) bzw. • Drahtgittermodelle (Wireframes) darstellen
i	Interaktion starten bzw. beenden
p	Picken eines Objekts
f	Punkt unter dem Zeiger wird Fokus-Punkt
3	Stereomodus ein- bzw. ausschalten
e	Programm beenden

Beispielprogramm bumpy

// Initialisierung
// Daten erzeugen
// in ImageData als Punktdaten packen; actor erzeugen
// ImageData durch GeometryFilter *warpen*; actor erzeugen
// in ImageData als Zellendaten packen; actor erzeugen
// gekrümtes Objekt direkt erstellen; actor erzeugen
// darstellen und Interactor starten
// aufräumen

Datenobjekte



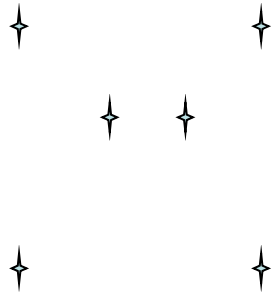
Datenobjekte

- Daten sind
  - Punkten, die durch (x,y,z) repräsentiert werden, oder
  - Zellen, die durch (Eck-)Punkte definiert sind,

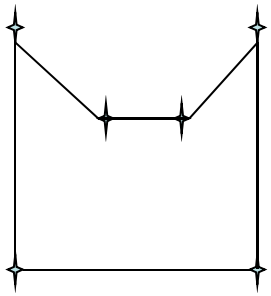
  zugeordnet
- Es gibt je nach Sichtweise 12, 15 bzw. 20 Zell-Typen
- Daten bestehen aus Komponenten, die Vektoren oder Skalare sein können

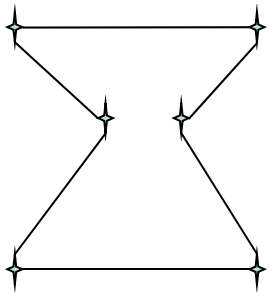
Zellen

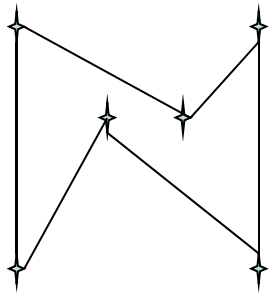
- Basis für Interpolationen
- Durch Eckpunkte nicht eindeutig definiert
- Aufgabe: Bestimmen Sie die 6 Kanten, die einander nicht schneiden und die gesuchte Zelle festlegen

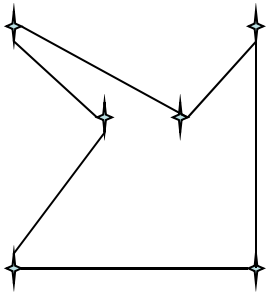


Einige Beispiele









DataArray

- eindimensional
- Länge l
- aufgeteilt in n Tupel
- jedes Tupel hat k Komponenten
- l = n * k
- Point hat 3 Komponenten

Erzeugung von vtk<*type*>Array

SetNumberOfTuples(vtkIdType n)

vtkIdType ist ein großer ganzzahliger Typ

SetNumberOfComponents(int n)

GetRange(double out[2], int component)

Liefert Minimum und Maximum in *out*

SetTupleValue(vtkIdType i, const <type> *value)

InsertTupleValue(vtkIdType i, const <type> *value)

Legt ggf. zusätzlichen Speicher an

InsertNextTupleValue(const <type> *value)

Legt ggf. zusätzlichen Speicher an

Squeeze()

Gibt überflüssigen Speicher frei

Topologie und Geometrie

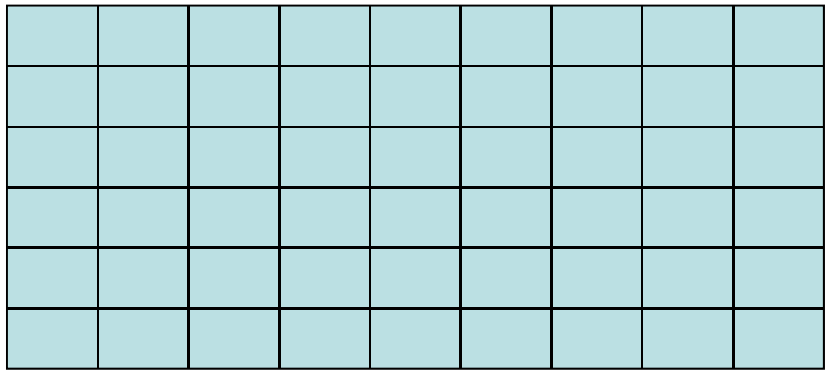
- Die Struktur der Topologie ist die Anordnung der Daten
- Die Struktur der Geometrie ist die Anordnung der Punkte
- Die Zuordnung der Zellen zu den Punkten kann durch implizite Nutzung der Topologie erleichtert werden
- 5 Typen zur Repräsentation der Daten

Koordinaten und Indizes

- x, y und z sind immer Koordinaten in der 3-dimensionalen Szene
- i, j und k sind Indizes bezogen auf das Datenarray
- dabei ist i der Index, der seinen Wert immer ändert (innerste Schleife)

```
for (int k = 0; k < dim[2]; ++k)
        for (int j = 0; j < dim[1]; ++j)
                for (int i = 0; i < dim[0]; ++i)
                        int addr = (k * dim[1] + j) * dim[0] + i;
```

ImageData



- regelmäßige Topologie
- Zellen sind Voxel bzw. Pixel
- Angabe der Dimensionen, des Spacings (beides pro Achse) und des Ursprungs
- Einfach und unflexibel

Erzeugung von ImageData

SetDimensions(int i, int j, int k)

Dimensionen des Datensatzes

SetDimensions(const int dims[3])

Andere Parameterübergabe – wird im Folgenden weggelassen

SetOrigin(double x, double y, double z)

World-Coordinates des Ursprungs

SetExtent(int minI,int maxI,int minJ,int maxJ,int minK,int maxK)

Grenzen der Indizes – keine Auswirkung auf Datenzugriff

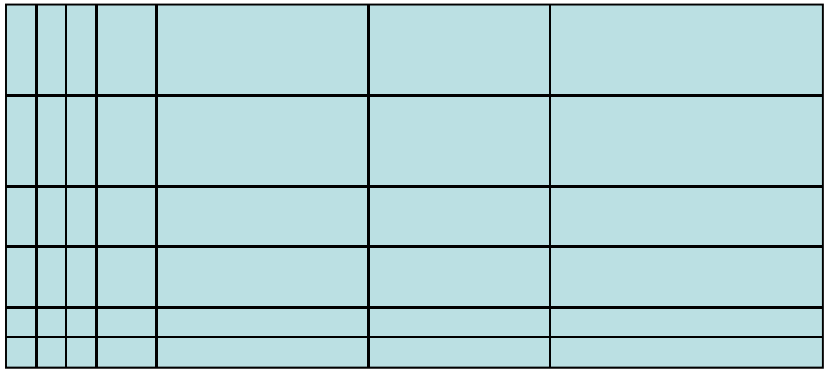
SetSpacing(double ix, double, jy, double kz)

Spacing bei der Berechnung der Koordinaten

implizite Koordinaten

- mit Hilfe der Vektoren *origin* und *spacing* lassen sich die Koordinaten aus den Indizes berechnen
- $(x,y,z)^T = \text{spacing} \; (\; (i,j,k)^T - \text{origin}^T \;)$
- die $\prod$ (dimension[i] – 1) Zellen lassen sich ebenfalls implizit bestimmen

RectilinearGrid



- regelmäßige Topologie
- Zellen sind Voxel bzw. Pixel
- Angabe der Dimensionen und pro Achse ein Array der skalaren Koordinaten-Werte

Koordinatenberechnung

- Die Koordinaten des Punktes, dem die Daten an der Stelle i,j,k im Array zugeordnet sind, ergeben sich als
- $(x[i],y[j],z[k])^T$
- Die Zellen lassen sich analog zu ImageData implizit bestimmen
- Etwas aufwendiger und flexibler

Erzeugung von RectilinearGrid

SetDimensions(int i, int j, int k)

Dimensionen des Datensatzes

SetOrigin(double x, double y, double z)

World-Coordinates des Ursprungs

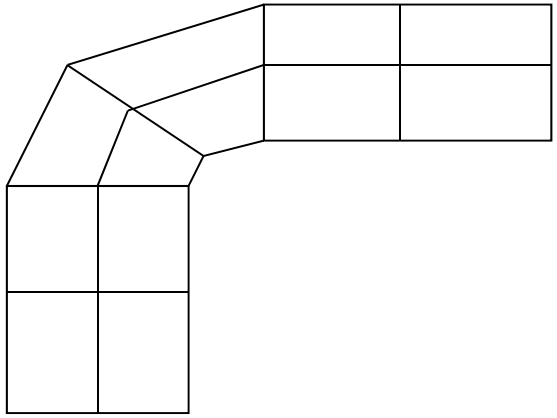
SetExtent(int minI,int maxI,int minJ,int maxJ,int minK,int maxK)

Grenzen der Indizes – keine Auswirkung auf Datenzugriff

Set[X,Y,Z]Coordinates(vtkDataArray *coords)

Koordinaten-Array

StructuredGrid



- weiteres Beispiel ist eine Kugeloberfläche
  Angabe der Längen- und Breitengrade
- reguläre Topologie
- Zellen sind Quads bzw. Hexahedrons
- Koordinaten für alle Punkte angeben
- Relativ aufwendig; ziemlich flexibel

Erzeugung von vtkPoints

SetNumberOfPoints(vtkIdType n)

GetBounds(double out[6])

Liefert Boundary in *out*

SetPoint(vtkIdType i, const <type> values[3])

InsertPoint(vtkIdType i, const <type> values[3])

Legt ggf. zusätzlichen Speicher an

InsertNextPoint(const <type> values[3])

Legt ggf. zusätzlichen Speicher an

SetData(vtkDataArray *coords)

DatenArray mit Koordinaten setzen

Squeeze()

Gibt überflüssigen Speicher frei

Erzeugung von StructuredGrid

SetDimensions(int i, int j, int k)

Dimensionen des Datensatzes

SetOrigin(double x, double y, double z)

World-Coordinates des Ursprungs

SetExtent(int minI,int maxI,int minJ,int maxJ,int minK,int maxK)

Grenzen der Indizes – keine Auswirkung auf Datenzugriff

SetPoints(vtkPoints *coords)

Koordinaten-Array

UnstructuredGrid

- irreguläre Topologie
- alle Zelltypen möglich
- inhomogene Kombinationen von Zelltypen erlaubt
- Koordinaten aller Punkte angeben
- Alle Zellen angeben
- Sehr aufwendig und flexibel

Erzeugung von CellArray

Prinzipiell ein IntegerArray der Form:

n1, pId11, … pId1n1, n2, pId21, ... pId2n2, … nx, … pIdxnx

InsertNextCell(vtkIdType npts, const vtkIdType *ptIds)

Zelle mit npts Punkten.

ptIds enthält die Id‘s (Indizes) der Punkte

Erzeugung von UnstructuredGrid

SetPoints(vtkPoints *coords)

Koordinaten-Array

InsertNextCell(vtkIdType npts, const vtkIdType *ptIds)

Zelle mit npts Punkten.

ptIds enthält die Id‘s (Indizes) der Punkte

SetCells(int type, vtkCellArray *cells)

Homogene Zellen des Typs *type*

PolyData

- prinzipiell ein UnstructuredGrid
- vier Zelltypen nebeneinander
  - Punkte (Punkt, Punktmenge)
  - Linien (Linie, PolyLine)
  - PolyLines (Triangle, Pixel, Quad, Polygon)
  - TriangleStrips
- alle anderen Zelltypen müssen durch TriangleStrips dargestellt werden
- Output der Mapper
- Input der Actors

Erzeugung von PolyData

SetPoints(vtkPoints *coords)

Koordinaten-Array

InsertNextCell(vtkIdType npts, const vtkIdType *ptIds)

Zelle mit npts Punkten.

ptIds enthält die Id‘s (Indizes) der Punkte

SetVerts(vtkCellArray *cells)

SetLines(vtkCellArray *cells)

SetPolys(vtkCellArray *cells)

SetStrips(vtkCellArray *cells)

0-, 1- und 2-dimensionale lineare Zelltypen

0-dimensional
- Vertex (Punkt); Polyvertex (Punktmenge)

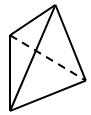
1-dimensional
- Line; Polyline

2-dimensional
- Triangle; Triangle-Strip (immer in einer Ebene)
- Quadrilateral (Quad), beliebiges Viereck
- Pixel (achsenparalleles Rechteck)
- (geschlossenes) Polygon

blaue Zelltypen müssen Ebenenbedingung erfüllen

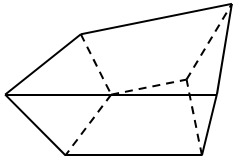
hier 6 bzw. 9 Zelltypen

3-dimensionale lineare Zelltypen

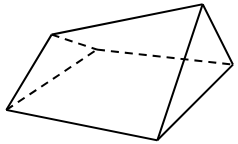


Tetrahedron

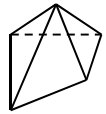
(Tetra)



Hexahedron



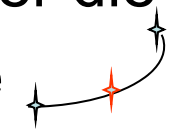
Wedge



Pyramid

- Ein Voxel ist ein achsenparalleles Hexahedron
- ConvexPointSet ist der generellste 3D-Zelltyp
- Somit kennen wir nun 12 bzw. 15 Zelltypen

nicht-lineare Zelltypen

- bisher verbinden lineare Kanten die Eckpunkte
- Einfügen des Mittelpunktes einer Kante
- quadratische Interpolation über die Kante
- QuadraticEdge ist eine Kurve 
- außerdem QuadraticTriangle, QuadraticQuad, QuadraticTetra und QuadraticHexahedron
- somit 12 lineare geometrische + 3 Mengentypen + 5 nicht-lineare = 20 Zelltypen

Exkurs quadratische Interpolation

gegeben $t \in [0, 1]$; $x(0)$, $x(½)$ und $x(1)$

gesucht $f(t) = at^2 + bt + c = x(t)$

$c = x(0)$

$b = x(1) – a – c$

$4\,(a¼ + b½ + c) = 4\,x(½) \Leftrightarrow$

$a + 2\,x(1) – 2\,a + 2\,c = 4\,x(½) \Leftrightarrow$

$a = 2\,x(0) + 2\,x(1) – 4\,x(½)$

$b = 4\,x(½) – x(1) – 3\,x(0)$

für $x(½) = ½(x(0) + x(1))$ gilt

$a = 0$ und $b = x(1) – x(0)$ (lineare Interpolation)

Point- bzw. CellData

Beinhaltet:

- Daten der Punkte bzw. Zellen
- Das sind unter anderem
  - Einen skalaren Datensatz
  - Einen Vektor-Datensatz
  - Die Normalen
  - Weitere Datensätze, die über Namen angesprochen werden
- Datensätze können leer sein

Einbringen von Daten in Point- bzw. CellData

AddArray(vtkDataArray *data)

SetScalars(vtkDataArray *data)

SetVectors(vtkDataArray *data)

SetNormals(vtkDataArray *data)

…

vtkDataArray *GetArray(const char *name)

SetActiveScalars(const char *name)

…

Beispiel: bumpy

Algorithmen und DataSets

- Eine Vielzahl von Algorithmen akzeptieren mehrere Typen von Datasets als Input und sind dadurch vielfach einsetzbar.
- Einige Algorithmen (z.B. volumeRenderer) verarbeiten ausschließlich ImageData, da sie die Struktureigenschaften benötigen
- Meist ist der Output vom gleichen Typ wie der Input oder PolyData

generell ⇔ effizient

- werden Algorithmen generell gehalten, um vielfältig eingesetzt zu werden, muss man auf die Nutzung der Struktureigenschaften verzichten und verliert Performance.
- Algorithmen, die Struktureigenschaften nutzen, gewinnen Performance auf Kosten der Vielfalt der Einsatzmöglichkeiten
- ➔ DataSet als Input und anhand des Subtyps verzweigen

Shading

- Liegen lediglich die Punktdaten vor, so werden die Farbwerte für die Zellen interpoliert
- Zur Berechnung der Lichtreflektion werden die natürlichen Normalen der Flächen genommen
- Ein Shading (Gouraud bzw. Phong) findet nicht statt
- Die Zellen sind erkennbar

Punkt-Normalen

- Flächen der Zellen sind nur eine Näherung der eigentlichen Oberfläche
- Verfügt der Datensatz über Normalen für jeden Punkt, so kann das Shading die Normalen bei der Berechnung der Lichtreflektion interpolieren
- Diese Darstellung wird „Phong-Shading“ genannt
- Es entstehen „gekrümmte“ Darstellungen der Zellen
- Zellen können optisch kontinuierlich ineinander übergehen
- „Gouraud-Shading“ interpoliert die Farbwerte der Punkte und ist daher schneller

Shading der Kugel

- Bei der Einheitskugel um den Ursprung ist die Normale in jedem Punkt der Kugeloberfläche gleich den Koordinaten des Punktes
- ➔ einfache Erweiterung der Kugel, so dass die Zellen nicht mehr erkennbar sind

Konturen

- Häufig geht es darum, Konturen im Datensatz sichtbar zu machen
- 2 Ansätze
  - (n-1)-dimensionale Strukturen gleichen Wertes extrahieren (Iso-Linien bzw. –Flächen)
  - Transparenz nutzen, um wichtige Daten hervorzuheben (Volume-Rendering)
- Eventuell Kombination dieser Ansätze
- Problem:
  - Ausblendung ⇔ Verdeckung
  - ➔ Wahl geeigneter Parameter

Iso-Linien und -Flächen

- Die bekannteste Anwendung von Iso-Linien sind Höhenlinien-Darstellungen in Landkarten
- Generell werden in einem n-dimensionalen Datensatz die Punkte (auch durch Interpolation) bestimmt, an denen ein vorgegebener Isowert vorliegt.
- Aus einem 3D-Volumen wird eine Iso-Fläche extrahiert
- Aus einem 2D-Datensatz wird eine Iso-Linie extrahiert
- Bekanntestes Verfahren (in 3D) ist „Marching Cube“

Besonderheiten in Java

- vtkCanvas und vtkPanel sind Java-Widgets, die gleichzeitig renderWindows sind, die
  - einen renderer beinhalten
  - einen renderWindowInteractor beinhalten
  - in awt- oder swing-Strukturen einbindbar sind
- vtkPanel kann ein eigenständiges Fenster sein oder über einen VtkPanelContainer in Java-Frames eingebunden sein
- vtkCanvas bzw. vtkPanel müssen als erste VTK-Objekte erzeugt werden, da sie das gesamte VTK initialisieren
- VM-Option gegen Bug:
  -Dsun.java2d.ddoffscreen=false -Dsun.java2d.gdiblit=false

ContourFilter

- Neben einem beliebigen Datensatz werden ein oder mehrere Isowerte angegeben
- Zu jedem Isowert wird eine Kontur als Output erzeugt
  - SetNumberOfContours(n)
  - SetValue(i, v) – i $\in$ [0..n-1]
- Kann selbst mehrere äquidistant verteilte Isowerte erzeugen
  - GenerateValues(n, low, high)
- Kann Normalen berechnen
  - ComputeNormalsOn()

VolumeRendering

Renderer addVolume(v)

Volume v benötigt

- VolumeMapper m – SetMapper(m)
- VolumeProperty p – SetProperty(p)

VolumeMapper erhält als Input ImageData vom Typ UnsignedChar

VolumeProperty benötigt

- ColorTransferFunction
- PiecewiseFunction opacityTransfer

Interaktion

- Parameter der Module als Reaktion eines GUI-Elementes (z.B. Slider)
  - plane.SetOrigin(orig);
  - iso.panel.update(getGraphics());
- VTK-Widgets (später)

Clipping, Cutting, Subsampling

- Clip- und ExtractGeometry-Filter schneiden den Datensatz an einer ImplicitFunction (z.B. Plane) oder bezüglich eines skalaren Thresholds
- Cutter liefert den Schnitt zwischen den Daten und einer ImplicitFunction (Slice)
- ExtractGrid extrahiert das „Volume Of Interest“ (VOI) und kann dabei das Grid unterabtasten (subsampling). Der Input muss strukturiert sein
- Während beim Clipping und Cutting die Geometry (xyz-Raum) bearbeitet wird, wird das VOI anhand der Topologie (ijk-Indizes) bestimmt

ClipPolyData, ClipVolume

- SetValue(double value) – setzt den Threshold
- SetClipFunction(vtkImplicitFunction *) – setzt die Clip-Geometrie
- InsideOutOn/Off() – tauscht außen und innen
- GenerateClippedOutputOn/Off() – erzeugt zusätzlich die entfernte Geometrie
- GetClippedOutput() – falls generiert

Cutter

- SetValue(double value) – setzt den Threshold
- wie ContourFilter – GenerateValues
- SetCutFunction(vtkImplicitFunction *) – setzt die Cut-Geometrie

ExtractGrid

- SetVOI(int minI, int maxI,
  int minJ, int maxJ,
  int minK, int maxK) – Minima und Maxima der Indizes
- SetSampleRate(int divI, int divJ, int divK) – Unterabtastung
- IncludeBoundaryOn/Off() – Fixiert den Rand

Widgets

- erlauben Interaktion
- vom Benutzer manipulierbare ImplicitFunctions
- ein-/ausschalten per Programm oder Eingabe („i“ im RenderWindowInteractor)
- Reaktion durch Callbacks
- hier: Generierung der Startpunkte für Streamlines

Callback in C++

```
vtkActor *actor = vtkActor::New();
vtkPolyData *planeGeo = vtkPolyData::New();
vtkPlaneWidget *plane = vtkPlaneWidget::New();

void processEvents(vtkObject *caller, unsigned long event, void *clientData,
                   void *callData) {
    switch (event) {
    case vtkCommand::StartInteractionEvent:
                   actor->VisibilityOn();
                   return;
    case vtkCommand::EndInteractionEvent:
                   actor->VisibilityOff();
                   return;
    case vtkCommand::InteractionEvent:
                   plane->GetPolyData(planeGeo);
                   return;
    default:
                   return;
    }
```

Einbindung in C++

```
vtkCallbackCommand *cmd = vtkCallbackCommand::New();
cmd->SetCallback(processEvents);
plane->
    AddObserver(vtkCommand::StartInteractionEvent, cmd, 1.);
plane->AddObserver(vtkCommand::InteractionEvent, cmd, 1.);
plane->
    AddObserver(vtkCommand::EndInteractionEvent, cmd, 1.);
cmd->Delete();
```

Callback in Java

```
public class Callback {
    vtkActor actor;
    vtkPlaneWidget plane;
    vtkPolyData planeGeo;
    public void work() {
        plane.GetPolyData(planeGeo);
    }
    public void start() {
        actor.VisibilityOn();
    }
    public void stop() {
        actor.VisibilityOff();
    }
}
```

Einbindung in Java

```
Callback cb = new Callback(actor, plane, planeGeo);
plane.AddObserver("StartInteractionEvent", cb, "start");
plane.AddObserver("InteractionEvent", cb, ”work");
plane.AddObserver(„EndInteractionEvent", cb, "stop");
```

Die in Java übliche Vorgehensweise per „Listener“-Interface wird leider nicht verwendet

Glyphen

- Jeder Datensatz (auch Punktmenge) als Input möglich
- Source-Objekt wird an jeden Punkt des Input-Datensatzes gesetzt
- Source-Objekt wird angepasst in
  - Orientierung
  - Farbe
  - Skalierung

Vektorfelder und Glyphen

- Daten sind (3-dimensionale) Vektoren, die als Kraft bzw. Geschwindigkeit interpretiert werden
- Glyphen bilden eine Möglichkeit der Darstellung
- Meist werden Pfeile (Arrows) als Sourcen verwendet (Hedgehog)
- Darstellung entspricht dem sichtbar machen von Magnetfeldern mittels Eisenspänen

Streamer, Streamlines und Streampoints

- Streamer bildet die Basis
- Source ist eine Punktmenge
- diese wird durch das Vektorfeld integriert.
- Integrationsmethode wählbar (z.B. Runge-Kutta)
- Streampoints liefert Punktmenge in mehreren Zeitschritten
- Streamlines liefert für jeden Punkt eine Linie, die die Koordinaten des Punktes zu verschiedenen Zeitpunkten verbinden

Picking

- Durch „p“ im RenderWindowInteractor vom Benutzer ausgelöst
- Callback als Observer angeben
- World-Koordinaten im Picker verfügbar
- je nach Art des Pickers auch Point-Id bzw. Cell-Id abfragbar

Smoothing

- „Rauschen“ in den Daten
  ➔
  - sehr viele Dreiecke
    ➔
    - *“unsauberes” Bild*
    - *großer Rechenaufwand*
- Glättung durch Reduzierung der Anzahl der Dreiecke
- Aber: große Anzahl an Dreiecken ist oft ein Fehlerindikator
- Netzvereinfachung
- Diplomarbeit von Maik Boltes

Texturen

- Realistische Flächen (Spiele)
- Ausnutzen der Transparenz von Texturen
  - Ausblenden von Teilen von Objekten
  - Erzeugen von Löchern mit Rand in Objekten
  - Fake-VolumeRendering

Bildverarbeitung

viele ImageXXXFilter

z.B.

- FastFourierTransformation (FFT)
- EdgeDetection
- uvm.

2-dimensionale XY-Plots

- Achsen mit Labels
- Legende
- usw.

Properties von Aktoren

- für ambientes, diffuses und spekulares Licht kann gesetzt werden
  - Koeffizient
  - Farbe
- Repräsentation (z.B. Wireframe)
- Shading
- Linienbreite
- …

Lichtquellen

- Punkt- oder gerichtete Lichtquelle
- nach Position und Bewegung
  - HeadLight (von der Kamera zum Fokuspunkt)
  - CameraLight (Position relativ zur Kamera)
  - SceneLight (beliebige Position)
- Stärke und Farbe der Lichtarten definierbar

Kamera

- Im Wesentlichen definiert durch
  - Position
  - Fokuspunkt
  - Up-Vektor
  - Viewport
- Perspektivische oder Parallelprojektion
- Methoden zur Rotation
  - Azimuth
  - Elevation
  - Roll

Beschriftung

- Text als „normale“ 3D-Objekte
- Text als Billboard
  (immer der Kamera zugewandt; in Entwicklung)
- mit „visuellem Zeiger“ auf den Referenzpunkt eines Objektes
- ScalarBarWidget (Zuordnung Farbe und Titel)

Bilder speichern

- WindowToImageFilter
  - konvertiert den Inhalt des Fensters in ein Image
  - muss explizit durch die Methode „Modified“ zum Update des Bildes angestossen werden
- vtk<XXX>Writer
  - erzeugt aus dem Input-Image eine Datei im <XXX>-Format
  - der Input kann z.B. der Output eines WindowToImageFilter sein