Institut für Parallele und Verteilte Systeme
Simulation großer Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3502

Auswertungsunabhängige PUM-Visualisierung

Matthias Kostka

Studiengang:	Softwaretechnik
Prüfer:	Jun.-Prof. Dr. rer. nat. Dirk Pflüger
Betreuer:	M.Sc. Sa Wu Dr. rer. nat. Stefan Zimmer
begonnen am:	04. März 2013
beendet am:	03. September 2013
CR-Klassifikation:	F.2.1, G.1.1, G.1.2, G.1.3, J.2

Kurzfassung

In dieser Diplomarbeit wird ein Plug-in (CraSSVis, Crack Simulation Software Visualization) für ParaView unter Linux entwickelt, welches den Lösungsraum der „Partition of Unity“-Methode (PUM) auswertungsunabhängig visualisiert.

Die PUM berechnet Koeffizienten für Basisfunktionen, die gemeinsam den Lösungsraum bestimmen. Obwohl durch PUM sich numerisch exakte Lösungen ergeben, ist eine Darstellung des Lösungsraums nicht möglich. Zu dessen partieller Darstellung werden diskrete Punkte ausgewertet. Bei der Weiterverarbeitung der sich ergebenden Werte kann nicht mehr auf die Koeffizienten und Basisfunktionen zurückgegriffen werden. Ein Zugriff auf Werte zwischen den zur Auswertung ausgewählten Punkten ist nicht mehr möglich. Der damit einhergehende Genauigkeitsverlust führt dazu, dass die Genauigkeit der Auswertung nicht mehr derjenigen des Lösungsraums der PUM entspricht. Somit besitzt die Visualisierung der Werte eine entsprechend geringere Genauigkeit als die des Lösungsraums der PUM.

CraSSVis ermöglicht die Darstellung des Lösungsraums der PUM auf einem Dreiecks- oder Tetraedergitter. Es integriert die Koeffizienten und Basisfunktionen, die den Lösungsraum der PUM bestimmen. CraSSVis ermöglicht eine partielle Darstellung des Lösungsraums über das Ausschneiden eines Teilbereichs des Gitters durch eine Bounding Box und das automatische Generieren von zusätzlichen Auswertungspunkten auf dem Gitterausschnitt. An allen erzeugten Punkten ist eine Auswertung des Lösungsraums möglich. Da immer weitere Punkte generiert werden können, sind eine Approximation an die Genauigkeit des von PUM erzeugten Lösungsraums und eine entsprechend genaue Darstellung möglich.

CraSSVis ermöglicht eine Weiterverwendung der Koeffizienten und Basisfunktionen, die den Lösungsraum der PUM bestimmen. Dadurch ergibt sich die Bedingung der Möglichkeit für hoch aufgelöste Abbildungen des Lösungsraums, die PUM nicht besitzt.

Inhaltsverzeichnis

1 Einleitung

Ziel dieser Diplomarbeit ist es, ein Plug-in – CraSSVis – für ParaView unter Linux zu entwickeln, welches ermöglicht, den Lösungsraum der „Partition of Unity“-Methode (PUM) hoch aufgelöst zu visualisieren.

Die PUM ist als numerisches Verfahren zur approximierten Lösung einer partiellen Differentialgleichung eine Erweiterung der Finite-Elemente-Methode (FEM). Die PUM benötigt keine aufwendige Erstellung eines Gitters wie die FEM und besitzt trotzdem eine entsprechend hohe Lösungsgenauigkeit. Mit der PUM werden Koeffizienten für Basisfunktionen, die gemeinsam den Lösungsraum bestimmen, berechnet. Die Darstellung des Lösungsraums ist jedoch nicht möglich. Diese Arbeit versucht ein Plug-in zur Lösung des Problems zu entwickeln. Zur Problemlösung werden grundlegende mathematische Voraussetzungen erarbeitet. Der Lösungsraum der PUM wird näher beschrieben. Das Gitter, welches für die Auswertung nötig ist, wird definiert. Das Herausschneiden und Verfeinern des Gitters wird beschrieben, und die für die Auswertung nötigen Basisfunktionen werden näher bestimmt. Die Verwendung der Koeffizienten, die Aufstellung der lokalen Basisfunktionen auf einem Referenzelement und die damit zusammenhängenden Transformationen werden erläutert.

Die Architektur von ParaView und seiner Verwendungsmöglichkeit werden vorgestellt und die Voraussetzungen für die Entwicklung eines Plug-ins näher analysiert. Die Datenstrukturen für die Verwendung des Gitters und der Koeffizienten werden ausführlich beschrieben. Das Plug-in soll ein automatisches Lesen von Koeffizienten, eine nutzergesteuerte Datenerzeugung sowie die Spezifikation und Einbindung beliebiger weiterer Basisfunktionen ermöglichen. Dazu werden die Möglichkeiten des Ladens einer Datei, der Verfeinerung eines Gitters und das Anlegen einer dynamischen Bibliothek untersucht. Die Analyse führt zu einer Spezifikation des zu entwickelnden Plug-ins, seines Entwurfs und der Implementierung. In der Spezifikation werden die Funktionalitäten, die das Plug-in umsetzt, näher bestimmt. Im Entwurf werden die Komponenten des Plug-ins aufgebaut. Bei der Implementierung werden dessen Funktionalitäten realisiert. Von CraSSVis werden eine allgemeine und eine speziell auf den Lösungsraum der PUM zugeschnittene Form entwickelt. CraSSVisGlobal beinhaltet alle Grundfunktionalitäten und wertet globale explizite Funktionen aus. Mit ihm werden die für eine hoch auflösende Auswertung von Basisfunktionen nötigen Funktionen umgesetzt. CraSSVisLocal baut mit seiner Funktionalität auf CraSSVisGlobal auf und wertet lokale Basisfunktionen aus. Mit der Entwicklung von CraSSVisLocal soll das Ziel einer Auswertung des Lösungsraums der PUM mit hoch aufgelöster Visualisierung ermöglicht werden.

Das zu entwickelnde Plug-in CraSSVis soll unter Verwendung von Koeffizienten und Basisfunktionen, die den Lösungsraum der PUM bestimmen, eine Abbildung des Lösungsraums der PUM ermöglichen.

2 Mathematische Grundlagen

Grundlegende mathematische Voraussetzungen für diese Diplomarbeit werden in diesem Kapitel vorgestellt.

2.1 „Partition of Unity“-Methode

Die „Partition of Unity“-Methode (PUM) [Scho8] ist ein gitterfreies Verfahren zur effizienten Lösung von partiellen Differenzialgleichungen mit Randbedingungen auf einer beliebigen Geometrie. Die zu lösende Differentialgleichung mit dem symmetrischen elliptischen Differentialoperator L und gegebenenfalls zu berücksichtigenden Neumann- B_N und Dirichlet-Randbedingungen B_D ist gegeben durch:

$$\begin{aligned} Lu &= f, \quad \text{in } \Omega \subset \mathbb{R}^n \\ B_N u &= g_N, \quad \text{in } \Gamma_N \subseteq \partial\Omega \\ B_D u &= g_D, \quad \text{in } \Gamma_D = \partial\Omega \setminus \Gamma_N. \end{aligned} \tag{2.1}$$

Die Lösung der Differentialgleichung bildet Koeffizienten c_i zu lokalen Basisfunktionen φ_i, die zusammen mit weiteren Funktionen η_{ij} den Lösungsraum

$$u = span\langle \varphi_i, \eta_{ij} \rangle = \sum_{i=0} \varphi_i \left(\sum_{j=0} c_{ij} \eta_{ij} \right) \tag{2.2}$$

aufspannen. Zur Lösung der Differentialgleichung wird die Geometrie mit einem Cover C_Ω überdeckt. Im Gegensatz zur FEM handelt es sich hierbei um ein grobes Gitter. Die PUM ermöglicht dennoch die gleiche Qualität der Lösung wie die FEM mit einem feinen Gitter. Der entstandene Lösungsraum wurde bisher üblicherweise an vielen diskreten Punkten ausgewertet. Die sich ergebenden skalaren Werte wurden anschließend zur Weiterverarbeitung verwendet. Das bisherige Verfahren bietet nicht die Genauigkeit, die von dem berechneten Lösungsraum ausgehend möglich wäre. Die Hinzunahme von neuen Auswertungspunkten soll dieses Problem beheben.

2.2 Gitter

Die Auswertung des durch die PUM erzeugten Lösungsraums hat $\mathbb{R}^2$ oder $\mathbb{R}^3$ (2D oder 3D Gebiet) als Voraussetzung. Das Gebiet besteht aus einer Menge von Punkten, die durch Kanten miteinander verbunden sind. Die Punkte sind durch kartesische Koordinaten x, y, z bestimmt. Das Gebiet setzt sich aus Dreiecken (im 2D) oder Tetraedern (im 3D) zusammen.

Definition 1 (Dreieck)
Ein Dreieck wird über drei nicht kollineare Punkte $P_i(x_i, y_i, z_i = 0)^T,\ i = 1, \ldots, 3$ und den drei Verbindungsstrecken (Kanten genannt) zwischen den Punkten beschrieben.

Anmerkung: Die Notation $z_i = 0$ zeigt an, dass die z-Koordinate stets Null beträgt und das Dreieck somit in der x,y-Ebene liegt.

Definition 2 (Tetraeder)
Ein Tetraeder wird durch vier nicht komplanare Punkte $P_i(x_i, y_i, z_i)^T,\ i = 1, \ldots, 4$ und den sechs Kanten zwischen den Punkten beschrieben.

Definition 3 (Primitiv)
Ein Dreieck (im 2D) oder ein Tetraeder (im 3D) wird als Primitiv bezeichnet.

Definition 4 (Gitter)
Ein Gitter besteht aus endlich vielen gleichartigen Primitiven $P_1, \ldots, P_r$: $\bigcup_{i=1}^{r} P_i$. Die Zerlegung ist zulässig für alle $i \neq j$, $P_i \cap P_j$. Es sind dabei zwei Fälle zu unterscheiden:

- *2D: leer, ein gemeinsamer Eckpunkt oder eine gemeinsame Kante*
- *3D: leer, ein gemeinsamer Eckpunkt, eine gemeinsame Kante oder eine gemeinsame Fläche*

[DR08].

Für die Auswertung des Lösungsraums eines bestimmten Gitterbereichs wird dieser Teilbereich aus dem Gitter herausgeschnitten (nachfolgend Gitterschnitt genannt, siehe Kapitel 2.2.1) und weiter verwendet.

2.2.1 Gitterschnitt

Durch einen Gitterschnitt mit einer Bounding Box wird die genauere Betrachtung eines Teilgitters ohne die Beachtung des restlichen Gitters ermöglicht.

Die Definition erfolgt im Hinblick auf die Erzeugung einer nicht achsenparallelen Bounding Box in ParaView, bei der sechs Punkte und sechs Normalen benötigt werden. Es wird dabei nicht zwischen 2D und 3D unterschieden.

Definition 5 (Bounding Box)
Eine Bounding Box ist ein Quader, der sich über einen Punkt auf seinen jeweiligen Seitenflächen und seiner zugehörigen Normalen definiert.

Definition 6 (Gitterschnitt)
Der Schnitt $A \cap B$ eines beliebigen Gitters A (2D oder 3D) mit einer Bounding Box B heißt Gitterschnitt.

Aus einem Gitter (vgl. Abbildung 2.1 A) entstehen nach dem Schnitt beliebig geartete Polygone (Polyeder) (siehe Abbildung 2.1 B). Da sie in dieser Form nicht weiter verarbeitet werden können, ist eine Zerlegung der einzelnen Polygone (Polyeder) in eine Menge von Primitiven notwendig. Diese Zerlegung wird als Triangulierung (siehe Kapitel 2.2.2) bezeichnet.

Der komplette Gitterschnittalgorithmus existiert bereits in ParaView und wurde nicht selbst entwickelt.

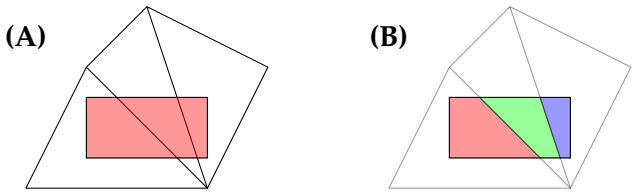


Abbildung 2.1: Schnitt eines Gitters mit der Bounding Box und die daraus resultierenden Polygone
(A) Schnitt mit der Bounding Box (rot) **(B)** Polygone nach dem Gitterschnitt

2.2.2 Triangulierung

Die Verfeinerung des Gitters (vgl. Kapitel 2.2.1, S. 3) erfordert die Zerlegung der aus dem Gitterschnitt resultierenden Polygone (Polyeder) mittels einer Triangulierung (vgl. Abbildung 2.1 B) in Primitive.

Definition 7 (Triangulierung)
Die Zerlegung eines Polygons (Polyeders) P ohne Loch in Primitive durch eine maximale Menge schnittfreier Diagonalen heißt Triangulierung von P. Die Diagonale in einem Polygon (Polyeder) ist ein offenes Liniensegment, das zwei verschiedene Eckpunkte von P verbindet und vollständig in P liegt [Lei12].

Bei einer Triangulierung werden keine neuen Punkte eingefügt, sondern existierende Punkte mit neuen Kanten verbunden. Abbildung 2.2 zeigt beispielhaft die Triangulierung der in Abbildung 2.1B dargestellten Polygone.

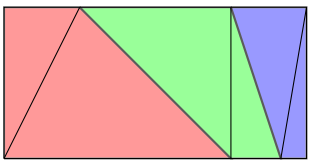


Abbildung 2.2: Beispiel einer Triangulierung für die aus der Abbildung 2.1 B nach dem Gitterschnitt entstandenen Polygone

2.2.3 Verfeinerung des Gitters

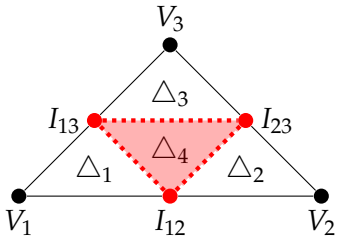
Durch neu hinzukommende Gitterpunkte, also durch eine Verfeinerung des Gitterausschnitts, wird eine hoch auflösende Auswertung ermöglicht. Die Verfeinerung des Gitters erzeugt neue Dreiecke oder neue Tetraeder.

Über die Qualität q, welche den Flächeninhalt (respektive das Volumen) in Relation zu den Kantenlängen setzt, kann die Güte der Verfeinerung sowohl für Dreiecke als auch für Tetraeder ermittelt werden.

Dreieck

Aus der Verbindung der Kantenmittelpunkte I_{12}, I_{13}, I_{23} ergeben sich im ursprünglichen Dreieck V_1, V_2, V_3 vier kleine, zueinander kongruente neue Dreiecke $\triangle_1, \triangle_2, \triangle_3, \triangle_4$:

$\triangle_1(V_1, I_{12}, I_{13})$,
$\triangle_2(V_2, I_{23}, I_{12})$,
$\triangle_3(V_3, I_{13}, I_{23})$,
$\triangle_4(I_{12}, I_{23}, I_{13})$.



Die Qualität für Dreiecke beschreibt die Metrik [Mat]

$$q = \frac{4a\sqrt{3}}{\sum_{1 \le i \le 3} h_i^2} \tag{2.3}$$

mit dem Flächeninhalt a und den Kantenlängen h_i. Ein Dreieck besitzt eine gute Qualität, wenn $q > 0.6$ ist. Ein gleichseitiges Dreieck hat die Qualität $q = 1$. Neu entstandene Dreiecke besitzen immer die gleiche Qualität wie das ursprüngliche Dreieck. Das Verhältnis der Fläche zu ihren Kantenlängen bleibt gleich (vgl. Abbildung 2.3, S. 5).

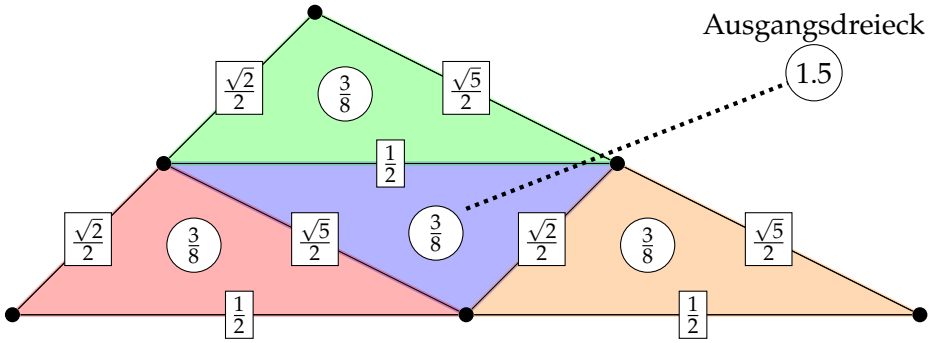
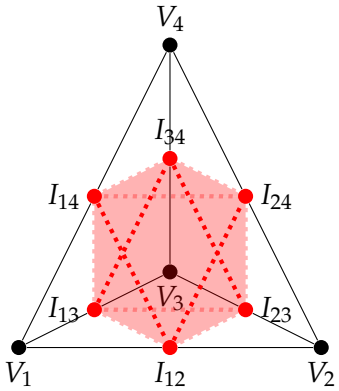


Abbildung 2.3: Qualität eines verfeinerten Dreiecks (Kantenlängen (Rechtecke) in cm; Flächen (Kreise) in cm^2)

Tetraeder

Aus der Verbindung der Kantenmittelpunkte I_{ij} ergeben sich im ursprünglichen Tetraeder $V_1, \ldots, V_4$ acht kleine neue Tetraeder $T_1, \ldots, T_8$. Tetraeder $T_1, \ldots, T_4$ und Tetraeder $T_5, \ldots, T_8$ sind untereinander identisch:

$\mathcal{T}_1(V_1, I_{12}, I_{13}, I_{14})$,
$\mathcal{T}_2(I_{12}, V_2, I_{23}, I_{24})$,
$\mathcal{T}_3(I_{13}, I_{23}, V_3, I_{34})$,
$\mathcal{T}_4(I_{14}, I_{24}, I_{34}, V_4)$,
$\mathcal{T}_5(I_{12}, I_{24}, I_{14}, I_{13})$,
$\mathcal{T}_6(I_{12}, I_{23}, I_{24}, I_{13})$,
$\mathcal{T}_7(I_{34}, I_{13}, I_{23}, I_{24})$,
$\mathcal{T}_8(I_{34}, I_{14}, I_{13}, I_{24})$.



Die Verfeinerung eines Tetraeders erweist sich als bedeutend schwieriger als die Verfeinerung eines Dreiecks. Bei der Verbindung aller Kantenmittelpunkte entstehen vier gleiche Tetraeder mit der Qualität des Ausgangstetraeders und ein Oktaeder, der in vier gleiche Tetraeder aufgeteilt werden kann, deren Qualität geringer als die des Ausgangstetraeders ist. Zur Bestimmung der Qualität eines Tetraeders gibt es mehrere Ansätze [Ahl03]. Hier wird als Metrik die in [LJ95] beschriebene Formel

$$q = \frac{12 \cdot (3a)^{2/3}}{\sum_{1 \le i < j \le 4} h_{ij}^2} \tag{2.4}$$

mit Volumen a und Kantenlängen h_{ij} verwendet. Ein Tetraeder besitzt eine gute Qualität, wenn $q > 0.6$ ist. Ein regelmäßiger Tetraeder hat die Qualität $q = 1$. Die Zerlegung des Oktaeders wird der Einfachheit halber ohne Optimierung durchgeführt. Dies hat einen Qualitätsverlust zur Folge. Nach der verwendeten Metrik sinkt die Qualität von $\mathcal{T}_5, \ldots, \mathcal{T}_8$ durch die Verfeinerung eines regelmäßigen Tetraeders mit den Punkten $(8,6,1), (5,11,9), (0,3,6), (9,2,10)$ von $q = 1$ auf $q = \frac{6}{7}$. Dies entspricht einem Qualitätsverlust von ca. 14,29%. Trotz der nicht optimierten Verfeinerung entstehen qualitativ gute Tetraeder. Jeder weitere Verfeinerungsschritt ergibt eine zusätzliche Verschlechterung.

2.3 Funktionen

Auf dem Gitter werden zwei Funktionen ausgewertet: „Globale explizite Basisfunktionen“ und „Lokale Basisfunktionen mit/ohne komplexe(r) Beschreibung“. Ein Koeffizient c_l kommt zu jedem der beiden Basisfunktionstypen hinzu. Die Auswertung findet auf dem verfeinerten Gitter über die Gitterpunkte statt. Im 2D werden die Basisfunktionen an den Punkten $P_i(x_i, y_i, z_i = 0)^T$ und im 3D an den Punkten $P_i(x_i, y_i, z_i)^T$ ausgewertet. Nachfolgend werden x, y, z als Variablen für die Funktionen verwendet.

Symbole:

- Globale Funktion: f
- Lokale Funktion: u
- Alle Basisfunktionen: g
- Eine Basisfunktion: g_l
- Alle Koeffizienten: c
- Ein Koeffizient (zu g_l gehörend): c_l
- Koordinaten: x, y, z
- Polynomgrad: n

2.3.1 Explizite Form bekannter globaler Funktionen

Die explizite Form einer globalen Funktion ist gegeben mit:

$$f = \sum_{l=0} c_l \cdot g_l. \tag{2.5}$$

In der Funktion werden die Koeffizienten c_l mit den globalen expliziten Basisfunktionen g_l multipliziert. Die Basisfunktion g_l besitzt einen globalen Träger, d. h. sie ist auf dem gesamten Gitter definiert und auswertbar.
Ein Beispiel für eine solche Funktion ist: $f(x, y, z) = 10 \cdot cos(x) + 20 \cdot sin(y) + 5 \cdot cos(z)$.

2.3.2 Lokale Funktionen

Die lokalen Funktionen u

$$u \approx \tilde{u} = \sum_{l=0} g_l \tag{2.6}$$

benötigen ein Gitter aus Primitiven. Die lokale Basisfunktion g_l besitzt einen lokalen Träger, d. h. sie ist nur auf einem Primitiv definiert und kann auch nur auf diesem ausgewertet werden. Zur unabhängigen Entwicklung von der eigentlichen Lösung der PUM werden in dieser Arbeit anstatt den φ_i aus Gleichung 2.2 (S. 2) andere lokale Basisfunktionen g_l verwendet.

Als lokale Basisfunktionen g wird die Polynomfunktion

$$g(x,y,z) = \sum_{\substack{i,j,k\geq 0 \\ i+j+k\leq n}}^{n} c_l x^i y^j z^k \tag{2.7}$$

mit dem Polynomgrad n und dem Koeffizienten c_l verwendet. Die benötigten Koeffizienten c wurden durch die PUM berechnet und sind fest auf die nötigen Stützstellen der Primitive verteilt.

Die lokalen Basisfunktionen g benötigen je nach Höhe ihres Polynomgrades n eine bestimmte Anzahl von Koeffizienten c und Stützstellen. Mit Hilfe von

$$\text{2D: } m = \frac{(n+1)(n+2)}{2} \qquad \text{3D: } m = \frac{(n+1)(n+2)(n+3)}{6} \tag{2.8}$$

lässt sich die Anzahl der Stützstellen und somit auch die der Koeffizienten berechnen. Die Koordinaten der Stützstellen und der dazugehörigen Koeffizienten auf einem Referenzprimitiv werden nach dem folgendem Algorithmus 2.1 festgelegt. Für jedes Primitiv müssen die Koeffizienten in der durch den Algorithmus vorgegebenen Reihenfolge vorliegen.

Definition 8 (Referenzprimitiv)
Das Referenzdreieck ist über die Punkte $(0,0),(1,0),(0,1)$ *und der Referenztetraeder über die Punkte* $(0,0,0),(1,0,0),(0,1,0),(0,0,1)$ *definiert. Referenzdreieck und Referenztetraeder werden als Referenzprimitiv bezeichnet.*

Abbildung 2.4 zeigt die Verteilung der Koeffizienten c_l exemplarisch für ein Dreieck mit Polynomgrad 3.

Algorithmus 2.1 Die Koordinatenberechnung der Stützstellen auf dem Referenzprimitiv

```
procedure StützstellenKoordinaten(Dimension, Polynomgrad)
    for all i ≤ Polynomgrad do
        for all j ≤ Polynomgrad − i do
            if Dimension = 2 then
                Stützstelle_t ← (i/Polynomgrad, j/Polynomgrad, 0)
            else if Dimension = 3 then
                for all k ≤ Polynomgrad − i − j do
                    Stützstelle_t ← (i/Polynomgrad, j/Polynomgrad, k/Polynomgrad)
                end for
            end if
        end for
    end for
end procedure
```

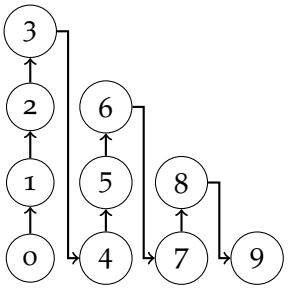


Abbildung 2.4: Die Reihenfolge der Koeffizienten auf dem Referenzdreieck für den Polynomgrad 3

Sowohl die Stützstellen als auch die Polynome beziehen sich auf ein Referenzprimitiv und werden auf diesem aufgestellt. Für jedes Primitiv wird eine Summe von lokalen Basisfunktionen g_l gesucht, die jeweils in einem Eckpunkt den Wert 1 und in den anderen Eckpunkten den Wert 0 besitzen. Dadurch ist eine einfache Verwendung der Koeffizienten c ohne weitere Berechnungen möglich. Es werden lineare Gleichungssysteme (hier der Einfachheit halber in 2D; die Aufstellung der Matrix im 3D folgt analog) durch eine Matrix mit den Polynomfunktionen aus Gleichung 2.7 (S. 8) wie folgt aufgestellt:

$$g_l(x,y) = \begin{pmatrix} 1 & x_1 & y_1 & \dots & x_1^i y_1^j \\ 1 & x_2 & y_2 & \dots & x_2^i y_2^j \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_k & y_k & \dots & x_k^i y_k^j \end{pmatrix} \begin{pmatrix} c_1 \\ \vdots \\ c_k \end{pmatrix} = e_l \tag{2.9}$$

mit $i + j \leq n$ (Polynomgrad), $k =$ Anzahl der Stützstellen, $e_l =$ l-te Einheitsvektor.

Die Lösungen der Gleichungssysteme ergeben die Koeffizienten für Lagrange Finite Elemente vom Typ n. Die Matrix ist aufgrund der linear unabhängigen Terme regulär und somit ist das Gleichungssystem immer eindeutig lösbar. Das Lösen erfolgt mit Hilfe des Gauß'schen Eliminationsverfahrens. Die Matrix ist für die zu erwartenden niedrigen Polynomgrade klein, daher fällt der hohe Aufwand der Berechnung der Lösung mit dem Gauß'schen Eliminationsverfahren von $\mathcal{O}(n^3)$ nicht weiter ins Gewicht. Die Abbildung 2.5 und Abbildung 2.6 (S. 10) zeigen beispielhaft die Polynome auf dem Referenzdreieck für den Grad 1 und 2.

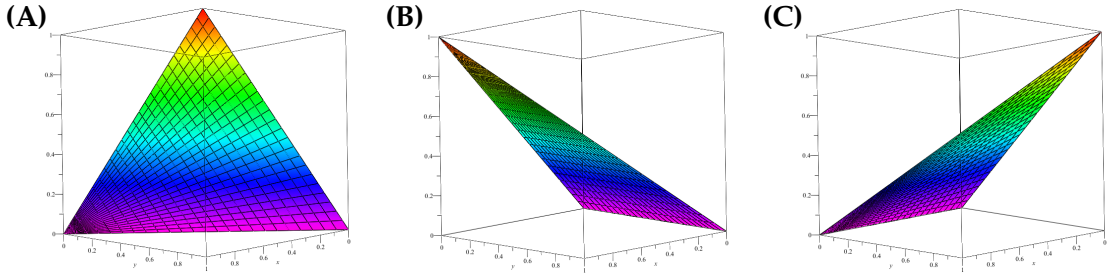


Abbildung 2.5: Lineare Lagrange-Polynome auf einem Referenzdreieck
(A) $g_1(x,y) = 1 - x - y$ **(B)** $g_2(x,y) = x$ **(C)** $g_3(x,y) = y$

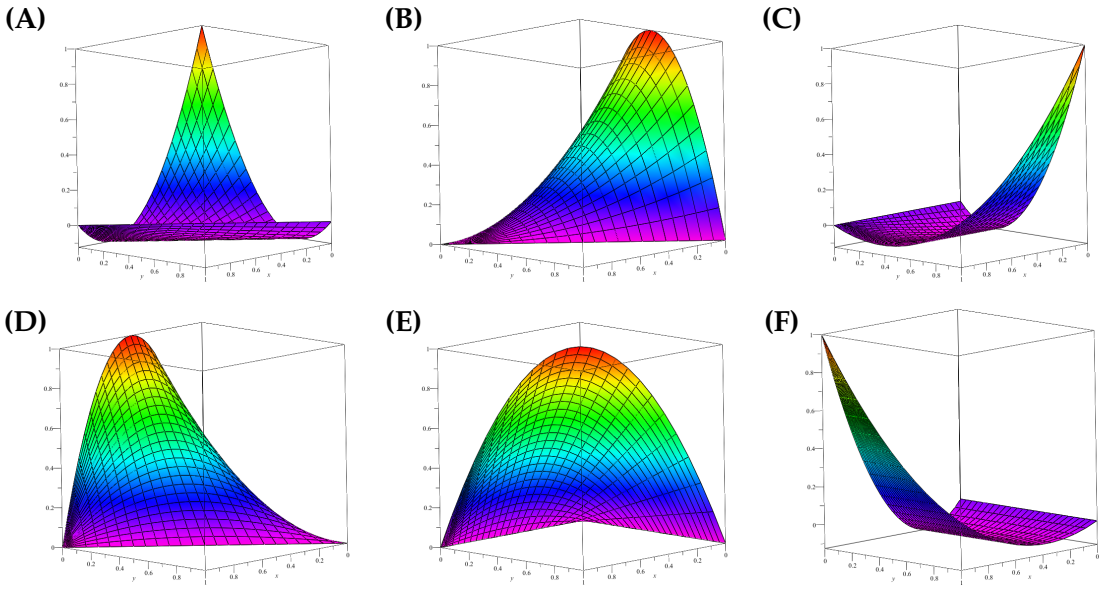


Abbildung 2.6: Quadratische Lagrange-Polynome auf einem Referenzdreieck
(A) $g_1(x,y) = 1 - 3x - 3y + 4xy + 2x^2 + 2y^2$
(B) $g_2(x,y) = -4xy + 4y - 4y^2$ **(C)** $g_3(x,y) = -y + 2y^2$
(D) $g_4(x,y) = 4x - 4xy - 4x^2$ **(E)** $g_5(x,y) = 4xy$ **(F)** $g_6(x,y) = -x + 2x^2$

Die Aufstellung der Lagrange-Polynome erfolgt auf dem nicht verfeinerten Gitter pro Primitiv. Nach dem Gitterschnitt werden für jedes Primitiv, das komplett in der Bounding Box liegt, die Polynomfunktionen aufgestellt. Alle Primitive, die die Bounding Box schneiden, werden nicht berücksichtigt, denn die Veränderung des Primitivs durch den Gitterschnitt hätte eine Veränderung der Koeffizienten zur Folge.

Der Lösungsraum von Gleichung 2.2 (S. 2) kann mit den Lagrange-Polynomfunktionen g_i noch nicht vollständig aufgespannt werden. Bis jetzt ist $u = \sum_i g_i c_i$ möglich.

Zur Aufstellung und Auswertung der Lagrange-Polynome, die auf einem Referenzprimitiv definiert sind, muss das beliebig im Raum liegende Primitiv auf ein Referenzprimitiv transformiert werden. Dies erfolgt über eine affine Transformation.

2.3.3 Affine Transformationen

Referenzdreieck

Das Referenzdreieck D_0 im (δ, ϵ)-System besitzt die Koordinaten $(0,0), (1,0), (0,1)$. Das Referenzdreieck D_0 wird mit der Funktion

$$\boldsymbol{F} = A\vec{x} + \vec{t} = \vec{x}' \text{ mit } A = \begin{pmatrix} (x_2 - x_1) & (x_3 - x_1) \\ (y_2 - y_1) & (y_3 - y_1) \end{pmatrix} \text{ und } \vec{x} = \begin{pmatrix} \delta \\ \epsilon \end{pmatrix} \text{ und } \vec{t} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \tag{2.10}$$

auf ein beliebig gelegenes Dreieck D im (x, y)-System transformiert. Die Eckpunkte des Primitivs werden dabei entsprechend ihrer Reihenfolge $x_1, \ldots, x_3$ und $y_1, \ldots, y_3$ zugeordnet. Die Abbildung 2.7 zeigt die Transformation F. Die Umkehrfunktion zu dieser affinen Transformation lautet:

$$\vec{x} = A^{-1}\vec{x}' - (A^{-1}\vec{t}). \tag{2.11}$$

Mit Hilfe der Gleichung 2.11 wird der auszuwertende Punkt innerhalb eines beliebigen Dreiecks auf das Referenzdreieck transformiert. Anschließend werden auf dem Referenzdreieck die Basisfunktionen aufgestellt bzw. ausgewertet.

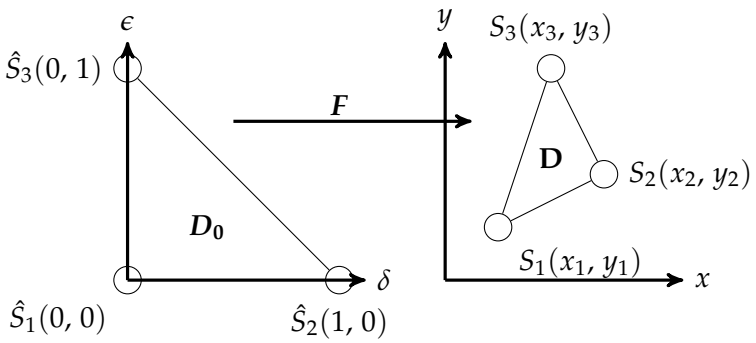


Abbildung 2.7: Transformation des Referenzdreiecks auf ein beliebig gelegenes Dreieck

Referenztetraeder

Der Referenztetraeder T_0 im $(\delta, \epsilon, \zeta)$-System besitzt die Koordinaten $(0,0,0), (1,0,0), (0,1,0), (0,0,1)$. Der Referenztetraeder T_0 wird mit der Funktion

$$A\vec{x} + \vec{t} = \vec{x}'$$
$$\text{mit } A = \begin{pmatrix} (x_2 - x_1) & (x_3 - x_1) & (x_4 - x_1) \\ (y_2 - y_1) & (y_3 - y_1) & (y_4 - x_1) \\ (z_2 - z_1) & (z_3 - z_1) & (z_4 - z_1) \end{pmatrix} \text{ und } \vec{x} = \begin{pmatrix} \delta \\ \epsilon \\ \zeta \end{pmatrix} \text{ und } \vec{t} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \tag{2.12}$$

auf einen beliebig gelegenen Tetraeder T im (x, y, z)-System transformiert. Die Rücktransformation und Zuordnung der Punkte folgt analog der Gleichung 2.11.

2.3.4 Stetigkeit der Koeffizienten auf den Primitiven

Für jedes Primitiv werden die Koeffizienten getrennt spezifiziert und für die Lagrange-Polynome verwendet. Für einen Eckpunkt können theoretisch so viele verschiedene Koeffizienten spezifiziert werden, wie Primitive sich diesen Punkt teilen. Bei einer Kante zwischen zwei angrenzenden Primitiven sind es immer zwei Koeffizienten. Abbildung 2.8 zeigt ein Beispiel für eine Verteilung von Koeffizienten. Dies würde zu einer nicht stetig ausgewerteten Lösung führen. Durch eine Überprüfung der Koeffizienten vor der eigentlichen Auswertung wird dies festgestellt.

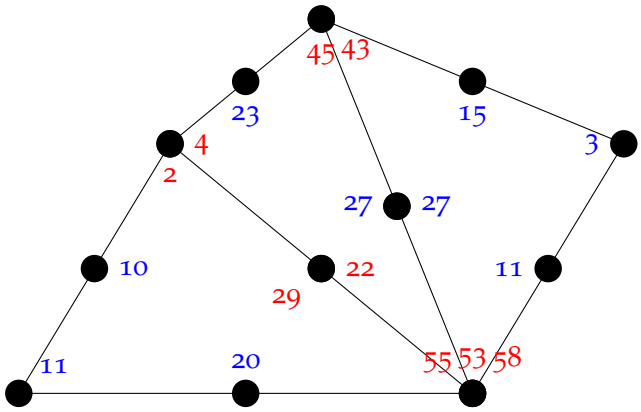


Abbildung 2.8: Verteilung von Koeffizienten auf einem Gitter; rote sind nicht stetig; blaue sind stetig

2.3.5 Enrichments

Die PUM generiert als Lösung nicht nur Koeffizienten für Basisfunktionen sondern auch Enrichments η_{ij}. Wie im Kapitel 2.1 (S. 2) genannt, sind sie ein wichtiger Teil der PUM-Lösung. Sie lassen sich mit einer globalen Funktion abstrahieren und werden als solche behandelt. Die ausgewerteten lokalen Basisfunktionen werden mit den Enrichments verrechnet und bilden gemeinsam den vollständigen Lösungsraum (vgl. Gleichung 2.2, S. 2) der PUM. Für die Verwendung der Enrichments werden weitere Koeffizienten benötigt. Für jede Stützstelle ist eine beliebige Anzahl von Koeffizienten möglich. Es werden für jedes Primitiv die selben Enrichments mit unterschiedlichen Koeffizienten verwendet. Ein konkretes 2D-Beispiel mit Polynomgrad 1 für den vollständigen Lösungsraum mit Lagrange-Polynomen (rot) und Enrichments (blau) ist durch Gleichung 2.13 gegeben.

$$\begin{aligned} u(x,y) =& (1-x-y)(a_{0,0} \cdot cos(x) + a_{0,1} \cdot sin(y)) + \\ & x(a_{1,0}) + y(a_{2,0} \cdot x + a_{2,1} \cdot y + a_{2,2} \cdot xy + a_{2,3} \cdot x^2 y^{-2}). \end{aligned} \tag{2.13}$$

2.3.6 Erweiterung der lokalen Basisfunktionen

Um den Lösungsraum der PUM aufzuspannen und eine Auswertung der Lösung zu ermöglichen, ist eine Ersetzung der Lagrange-Polynome g_i durch andere lokale Basisfunktionen φ_i notwendig. Mit den Enrichments η_{ij} und den lokalen Basisfunktionen φ_i ist eine Auswertung des Lösungsraums u der PUM möglich:

$$u = \sum_{i=0} \varphi_i \left(\sum_{j=0} c_{ij} \eta_{ij} \right). \tag{2.14}$$

Zusammengefasst sind zwei Arten von lokalen Funktionen auswertbar:

$$\begin{aligned} u &= \sum_{i=0} g_i c_i \\ u &= \sum_{i=0} \varphi_i \left(\sum_{j=0} c_{ij} \eta_{ij} \right). \end{aligned} \tag{2.15}$$

3 ParaView

ParaView ist eine Open-Source-Software zur Analyse und Visualisierung großer Datenmengen. Das Programm basiert auf „The Visualization Toolkit“ (VTK) [Kitg]. Für die Entwicklung der Plug-ins wurde die Version 3.98.0 verwendet, die zum Zeitpunkt der Entwicklung der Plug-ins die aktuellste war. Inzwischen (29. August 2013) gibt es die Version 4.0.1. Sie unterscheidet sich kaum von der Vorgängerversion. ParaView läuft unter Linux, Mac OS X und Windows. Das Programm steht unter der eigenen Lizenz –ParaView Lizenz Version 1.2. Diese ist mit einer BSD-Lizenz vergleichbar.

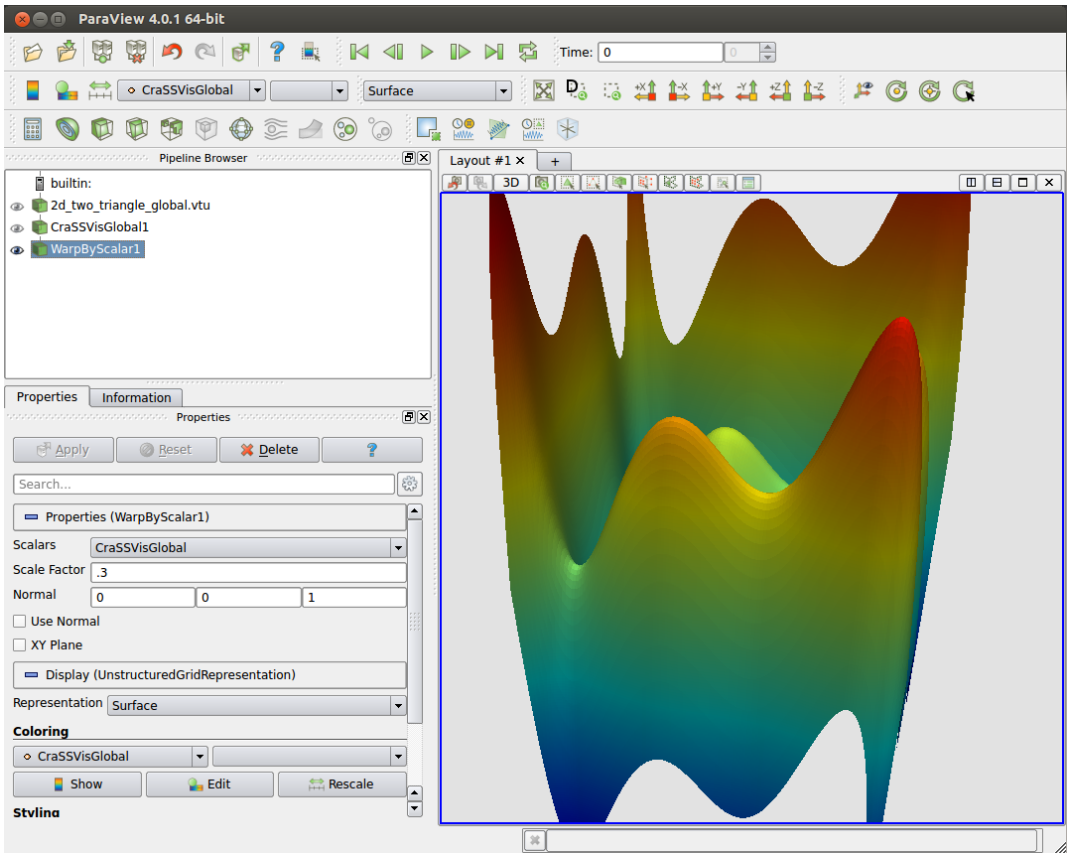


Abbildung 3.1: Screenshot von ParaView 4.0.1

Eine Dokumentation für das Erstellen eines Plug-ins ist unter [Kitb] zu finden. Funktionsmöglichkeiten mit einem Plug-in in ParaView werden dabei vorgestellt. Für die Analyse

der Umsetzung einer Funktion eignen sich die in ParaView vorhandenen Plug-ins besser als diese Dokumentation, weil sie kaum Informationen für die Bearbeitung konkreter Fragestellungen enthält. Die aus der Analyse der Plug-ins gewonnenen Erkenntnisse helfen bei der Entwicklung eines eigenen Plug-ins. Bei offenen Fragen kann auf einen E-Mail-Verteiler zurückgegriffen werden [Kitc].

ParaView hält sich bei der Visualisierung der Daten strikt an die Visualisierungspipeline [SMoo]:
Daten(-gewinnung) → Filter(-bereinigung) → Konvertierung (auf Geometrie und Attribute) → Darstellung.
Die entwickelten Plug-ins sind Filter-Plug-ins, die sich auch auf die Geometrie auswirken. Sie sind in der Pipeline unter „Filter“ und „Konvertierung“ anzusiedeln.

ParaView besitzt eine eigene Repräsentation der Geometrie und den damit zusammenhängenden Attributen. Die für die Plug-ins und zum Verständnis wichtigen Datentypen werden nachfolgend beschrieben.

3.1 Datenstrukturen

Im Kapitel 2.2 wurde ein Gitter mathematisch definiert. In VTK und somit auch in ParaView wird es durch ein unstrukturiertes Gitter dargestellt. Die Koordinaten der Gitterpunkte werden unabhängig von den Punktindizes, die die Primitive repräsentieren, also indirekt gespeichert.

Alle wichtigen Datenstrukturen für ein unstrukturiertes Gitter werden im Folgenden beschrieben. Die interne Repräsentation in C++ wird durch die Schriftart Courier wiedergegeben.

Die Begriffe werden in VTK in dieser Form verwendet und können von der allgemeinen mathematischen Definition abweichen.

ID
: Jedes Element in einer Liste besitzt eine ID `vtkIdType`, über die es identifizierbar ist.

Punkt
: Punkte werden durch ihre Koordinaten x, y, z repräsentiert. Der verwendete Datentyp ist `double[3]`. Alle Punkte im Gitter werden in einer Liste `vtkPoints` mit einer ID gespeichert.

Zelle
: Eine Zelle `vtkCell` (= Primitiv) im Gitter wird über eine Anzahl von IDs, die sich auf Punkte in `vtkPoints` beziehen, festgelegt. Die IDs beziehen sich nicht auf einzelne feststehende Punkte, ein Austausch der Punkte ist ohne weiteres möglich. Alle Zellen im Gitter sind im `vtkCellArray` mit einer ID gespeichert.

Zellentyp
Eine Zelle ist durch eine Anzahl von IDs und ihren Typ, der ihre geometrische Entsprechung beschreibt, definiert. VTK unterscheidet zwischen 69 verschiedenen Typen. Für diese Arbeit sind nur Dreiecke `VTK_TRIANGLE` und Tetraeder `VTK_TETRA` wichtig.

Skalar
Ein Skalar ist ein Datenwert, dem in der Visualisierung ein Farbwert mit Hilfe einer Farbtabelle zugeordnet wird.

Vektor
Ein Vektor ist eine Liste von Skalaren.

Punktdaten
Jedem Punkt im Gitter können Skalare und Vektoren als `vtkPointData` zugeordnet werden. Die Auswertung der hier entwickelten Plug-ins erzeugt neue Punktdaten.

Zellendaten
Jeder Zelle im Gitter können Skalare und Vektoren als `vtkCellData` zugeordnet werden, welche CraSSVisLocal für die Verwendung von Koeffizienten benötigt.

Unstrukturiertes Gitter
Hauptattribute eines unstrukturierten Gitters `vtkUnstructuredGrid` sind die Klassen `vtkPoints`, `vtkCellArray`, `vtkPointData` und `vtkCellData`. Sie werden in dieser Arbeit verwendet.

In Abbildung 3.2 sind die Korrelationen der Datenstrukturen eines unstrukturierten Gitters in VTK dargestellt. Das Gitter wird durch ein „vtkUnstructuredGrid“ (’vtu’ als Dateiendung) beschrieben. Die genaue Dokumentation zu einer vtu-Datei ist unter [Kitd] zu finden.

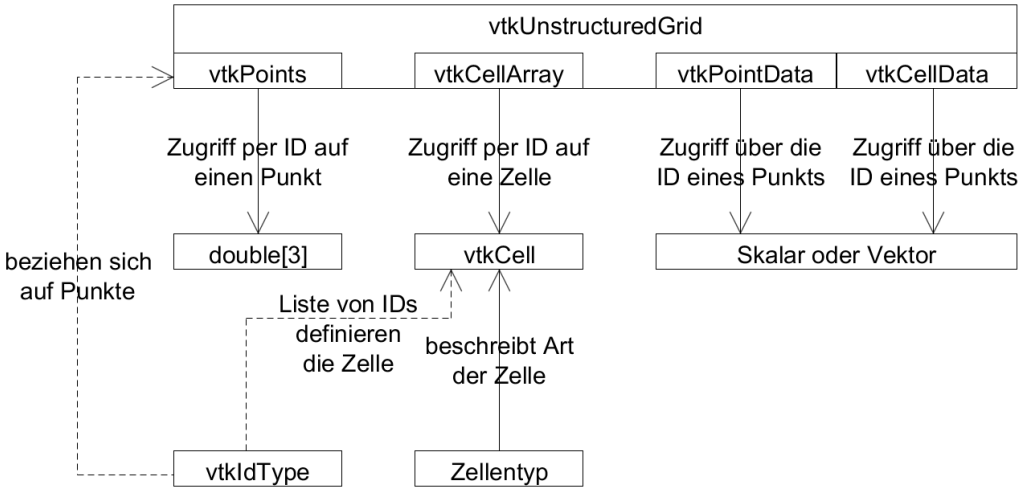


Abbildung 3.2: Korrelationen der Datenstrukturen eines unstrukturierten Gitters in VTK

3.2 Speicherung der Koeffizienten

Die Auswertung der lokalen Basisfunktionen benötigt Koeffizienten, ihre Reihenfolge und Anzahl unterscheidet sich nach den Fällen:

1. $u = \sum_{i=0} g_i c_i$
2. $u = \sum_{i=0} \varphi_i \left(\sum_{j=0} c_{ij} \eta_{ij} \right)$.

In der ersten Gleichung ist die Anzahl der Koeffizienten durch den Polynomgrad (vgl. Gleichung 2.8, S. 8) der Lagrange-Polynome festgelegt. Ihre Reihenfolge ist, wie in Abbildung 2.4 (S. 9) beispielhaft dargestellt, vorgegeben.

In der zweiten Gleichung ist die Anzahl der Koeffizienten durch die Anzahl der Enrichments festgelegt. Der n-te Koeffizient wird dem n-ten Enrichment zugeordnet.

Listing 3.1 zeigt eine Beispiel-vtu-Datei, die alle wichtigen Daten enthält. Ihre Bedeutung wird nachfolgend erklärt. Für die Beispielkoeffizienten in Zeile 11 wurden Lagrange-Polynome vom Grad 1 verwendet.

Listing 3.1 Beispiel für eine vtu-Datei

```
1  <VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">
2    <UnstructuredGrid>
3      <Piece NumberOfPoints="3" NumberOfCells="1">

5        <PointData Scalars="pointScalars" Vectors="pointVectors">
6          <!-- Ergebnis -->
7        </PointData>

9        <CellData Vectors="cellVectors">
10         <DataArray type="Float32" Name="Grad1" NumberOfComponents="3" format="ascii"
               RangeMin="10" RangeMax="12">
11           10 12 11
12         </DataArray>
13       </CellData>

15       <Points>
16         <DataArray type="Float32" Name="Points" NumberOfComponents="3" format="ascii"
               RangeMin="-10" RangeMax="10">
17           -10.0  0.0  0.0
18            10.0  0.0  0.0
19            0.0  10.0  0.0
20         </DataArray>
21       </Points>

23       <Cells>
24         <DataArray type="Int32" Name="connectivity" format="ascii" RangeMin="0"
               RangeMax="2">
25           0 1 2
26         </DataArray>

28         <DataArray type="Int32" Name="offsets" format="ascii" RangeMin="3" RangeMax="3">
29           3
30         </DataArray>

32         <DataArray type="UInt8" Name="types" format="ascii" RangeMin="5" RangeMax="5">
33           5
34         </DataArray>
35       </Cells>

37     </Piece>
38   </UnstructuredGrid>
39 </VTKFile>
```

→ 5-7: Einfügung der Ergebnisse des Plug-ins als neues Datenarray.

→ 10-12: Eintragung der Koeffizienten für CraSSVisLocal. Die Anzahl und Reihenfolge richtet sich nach dem gewünschten Polynomgrad (siehe Abbildung 2.4, S. 9) und der Orientierung der Zelle im Raum.

→ `16-20:` Auflistung der x, y, z-Koordinaten der Gitterpunkte. Eine Anpassung von „type“ (→ `16`) muss je nach Zahlenformat entsprechend vorgenommen werden.
→ `24-26:` Definition der Zellen im Gitter über die Reihenfolge der Indizes der Punkte.
→ `28-30:` Angabe der Anzahl der Indizes einer Zelle (z. B. 3 für Dreiecke) über einen Offset. Die Anzahl wird immer zum vorherigen Offset addiert.
→ `32-34:` Angabe des Zellentyps [Kite] (z. B. 5 für Dreieck).

3.3 Architektur

Die Architektur von ParaView ist in drei strikt getrennte Bereiche aufgeteilt: Render Server, Daten-/Berechnungsserver und Client (vgl. Abbildung 3.3). Die einzelnen Komponenten kommunizieren über TCP/IP und/oder SSH miteinander. Sind mehrere Render Server vorhanden, ist jedem davon ein bestimmter Daten-/Berechnungsserver zugeordnet. Die Parallelität der Server wird über MPI gesteuert. Der Vorteil dieser Architektur ist die Auslagerung der Speicher- und Rechenkapazität auf Maschinen mit großer Leistung oder Rechnercluster. Der Client, der die Steuerung der Visualisierung über die grafische Oberfläche von ParaView übernimmt, kann somit auch für größere Datenmengen auf leistungsschwachen Rechnern betrieben werden.

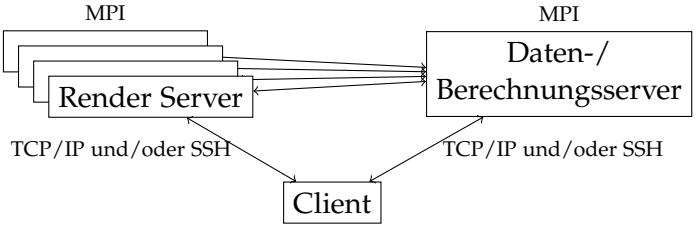


Abbildung 3.3: Schematische Darstellung der Dreikomponenten-Architektur von ParaView

Die strikte Trennung der drei Komponenten wirkt sich auf das Verhalten und die Implementierungsschritte der Plug-ins aus. Die Eingabe von Daten in die grafische Oberfläche übernimmt der Benutzer auf dem Client. Anschließend werden die eingetragenen Daten auf den Daten-/Berechnungsserver geschickt und dort verarbeitet bzw. verändert. Geänderte Daten werden dem Benutzer auf dem Client nicht mehr angezeigt.
Ein Beispiel: Der Benutzer trägt in ein Textfeld eine mathematische Funktion ein. Diese wird anschließend auf dem Server in eine andere Funktion umgewandelt. Diese umgewandelte Funktion kann dem Benutzer nicht mehr im Textfeld angezeigt werden.

4 Technische Spezifikation von CraSSVis

Die Spezifikation beschreibt die Anforderungen, die grafische Oberfläche und die Funktionalität von CraSSVis. Die interne Umsetzung der Funktionen durch die Implementierung wird nicht beschrieben [LL07].

Die Plug-ins ermöglichen eine einfache Auswertung von Basisfunktionen mit ihren dazugehörigen Koeffizienten auf einem vorgegebenen Gitter. Eine benutzerfreundliche Verwaltung der Basisfunktionen und Koeffizienten muss gewährleistet sein. Das Einbinden zusätzlicher Basisfunktionen wird über dynamische Bibliotheken realisiert.

CraSSVis wurde als Filter-Plug-in für ParaView entwickelt. Es ist in ein Plug-in für explizite globale Funktionen (CraSSVisGlobal) und eines für lokale Basisfunktionen mit und ohne komplexer Darstellung (CraSSVisLocal) aufgeteilt.

4.1 Nichtfunktionale Anforderungen

Portabilität
: CraSSVis ist für die ParaView Version $> 3.98.0$ ausgelegt. Eine Abwärtskompatibilität wird nicht gewährleistet. Das Plug-in wurde für Linux entwickelt, aber eine spätere Portierung auf andere Systeme wird nicht eingeschränkt.

Robustheit
: Durch die Programmierung ist die Stabilität von CraSSVis gewährleistet. Vom System werden zu Beginn alle Benutzereingaben überprüft. Falscheingaben können keine Systemabstürze erzeugen und der Benutzer wird über falsche Eingaben informiert.

Benutzer
: Voraussetzung für die Nutzung von CraSSVis ist das Beherrschen der mathematischen Grundlagen im Kapitel 2 (S. 2). Ausgelegt sind die Plug-ins allerdings auf mathematisch versierte Nutzer.

Benutzerfreundlichkeit
: Der Standard ISO-9241 (Abschnitt 10 bis 17) für die Benutzerfreundlichkeit wird im Rahmen von ParaView gewährleistet [EU]. CraSSVis wurde in Deutsch und Englisch entwickelt. Dem Benutzer steht ein deutscher Benutzerleitfaden direkt in ParaView zur Verfügung (siehe Anhang A, S. 55).

Erweiterbarkeit
Eine Erweiterung war zum Zeitpunkt der Durchführung der vorliegenden Arbeit nicht vorgesehen; der Entwurf und die Programmierung lassen dies jedoch zu.

Technologien
CraSSVis verwendet folgende Technologien:

- C++ als Programmiersprache
- Qt [Pro] und VTK durch ParaView
- TinyXML [Tho] zum Lesen der Eingabe-XML-Dateien
- muParser [Ber] zum Parsen der globalen expliziten Funktionen
- googletest [Goo] für die Modultests

Lizenz
Die Plug-ins stehen unter BSD-Lizenz. Die Universität Stuttgart besitzt exklusive uneingeschränkte Nutzungsrechte.

4.2 Funktionale Anforderungen

In diesem Kapitel werden die funktionellen Anforderungen an die Plug-ins CraSSVisGlobal und CraSSVisLocal spezifiziert. Die grafische Benutzeroberfläche, die Möglichkeiten des Benutzers zur Bedienung der Plug-ins und alle Schritte vom Ausgangsgitter bis hin zum visualisierten Ergebnis werden beschrieben.

4.2.1 CraSSVisGlobal als Plug-in für globale Funktionen

Das Plug-in CraSSVisGlobal wertet globale explizite Funktionen (vgl. Gleichung 2.5, S. 7)

$$f = \sum_{i=0} g_i c_i$$

aus. Für die Auswertung müssen folgende Funktionalitäten bereit gestellt werden:

- Funktionseingabe
- Gitterverfeinerung
- Gitterschnitt
- Automatisches Lesen und Einsetzen von Koeffizienten
- Erweiterung des Funktionsumfangs des Parsers

Abbildung 4.1 zeigt den Entwurf für die GUI des Plug-ins.

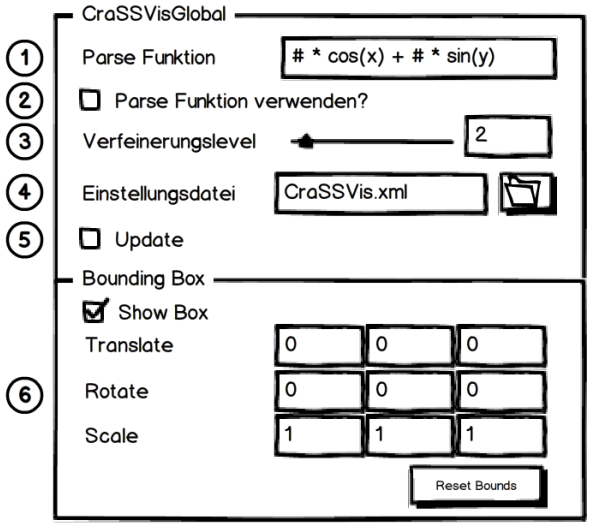


Abbildung 4.1: Entwurf für die GUI des Plug-ins CraSSVisGlobal

Die in der Abbildung 4.1 enthaltenen Funktionen (umrandete Ziffern 1 bis 6) sowie die in diesem Zusammenhang wichtigen Begriffe werden im Folgenden näher beschrieben.

1: Funktion parsen

Dem Benutzer steht ein Textfeld zur Verfügung, in dem er die gewünschte globale explizite Funktion eingeben kann. Als Variablen werden x, y, z für die Koordinaten verwendet. Zusätzlich kann das #-Zeichen durch automatisch geladene Koeffizienten ersetzt werden. Das erste #-Zeichen wird durch den ersten Koeffizienten usw. ersetzt. Sind nicht genügend Koeffizienten vorhanden, werden die #-Zeichen durch eine Eins ersetzt.

2: Parse Funktion verwenden?
Der Benutzer kann die globale Funktion entweder im Textfeld (1) oder in einer Datei spezifizieren. Ist die Checkbox ausgewählt, wird die Funktion aus dem Textfeld verwendet, ansonsten die aus der Datei.

3: Verfeinerungslevel
Das Verfeinerungslevel des Gitters kann durch den Benutzer mit dem Schieberegler zwischen 0 und 10 eingestellt werden. Auch eine direkte Eingabe des Levels im Textfeld ist möglich. Ein Wert kleiner Null oder größer Zehn wird in den entsprechenden Minimal- oder Maximalwert umgewandelt.

4: Einstellungsdatei
Wenn die Checkbox (2) nicht ausgewählt ist, lädt der Benutzer über einen Dateiauswahldialog die **Einstellungs-XML**.

5: Update
Die Checkbox „Update“ hat keine Funktion. Sie dient dazu, ParaView eine Änderung vorzutäuschen, die in einer XML-Datei manuell vorgenommen wurde. Eine erneute Ausführung wird somit möglich. Da die Verwendung eines Knopfs in einem Filter-Plug-in umfangreicher ist als die Umsetzung der Funktionalität durch eine Checkbox, liegt ein Workaround vor.

6: Bounding Box
Die Funktion und die grafische Umsetzung der Bounding Box wird von ParaView zur Verfügung gestellt. Der Benutzer kann mit der Maus einen Bereich für die Verfeinerung auswählen. Über die Textfelder kann die Größe, Position und Drehung der Box beeinflusst werden. Ein ohne erkennbare Funktion vorgegebener „Reset Bounds‘’-Knopf ist wahrscheinlich eine Fehlprogrammierung in ParaView.

Einstellungs-XML
Die Einstellungs-XML enthält die Pfade zur **Koeffizientendatei**, zu den **dynamischen Bibliotheken** und zur **globalen Funktionsdatei**. Wenn ein Pfad zu einem Ordner angegeben wird, werden zusätzlich beschriebene Dateien in diesem Ordner, andernfalls im Ordner der Einstellungs-XML gesucht. Gänzlich davon abweichende Dateipfade werden unverändert beibehalten.
Wenn Checkbox (2) nicht ausgewählt wurde, sind alle Inhalte optional. Andernfalls muss die **globale Funktionsdatei** enthalten sein.

Koeffizientendatei
Die Koeffizienten werden im *Matlab-Format* („m“ als Dateiendung) definiert. In dieser Datei wird pro Zeile ein Koeffizient im englischen Zahlenformat spezifiziert.

Dynamische Bibliothek
Die dynamischen Bibliotheken erweitern die globalen Funktionen um weitere während der Laufzeit hinzufügbare Funktionen.

Globale Funktionsdatei
Zur leichteren Eingabe besteht die Möglichkeit, die globale Funktion in einer Datei zu spezifizieren. Auch hier werden #-Zeichen durch geladene Koeffizienten ersetzt.

Funktionsweise von CraSSVisGlobal

Die globale Funktion wird über folgende Schritte, die vom Benutzer bzw. vom System ausgeführt werden, ausgewertet:

- Benutzer:
  1. Laden einer vtu-Datei mit einem unstrukturierten Gitter
  2. Eingeben der auszuwertenden Funktion
  3. Einstellen des Verfeinerungslevels
  4. Auswählen der Einstellungs-XML, wenn die Funktion aus dem Textfeld verwendet wird (sonst optional)
  5. Bestimmen des Verfeinerungsbereichs mit der Maus
  6. Drücken von „Apply“
- System:
  1. Überprüfen der Struktur der Daten
  2. Schneiden des Ausgangsgitters mit der Bounding Box
  3. Verfeinern des Auswahlbereichs
  4. Integrieren aller Einstellungen
  5. Laden der Bibliotheken und ihre Integration in den Funktionsparser
  6. Auswerten der Basisfunktionen pro Gitterpunkt

4.2.2 CraSSVisLocal als Plug-in für lokale Basisfunktionen

Je nachdem ob Lagrange-Polynome verwendet, oder diese durch eine dynamische Bibliothek ersetzt werden, wertet das Plug-in CraSSVisLocal lokale Basisfunktionen mit/ohne komplexe(r) Darstellung (vgl. Gleichung 2.15, S. 13) unterschiedlich aus:

1. Verwendung von Lagrange-Polynomen:

   $$u = \sum_{i=0} g_i c_i \quad \text{(vgl. Gleichung 2.6, S. 7)}$$

2. Ersetzen von Lagrange-Polynomen durch eine dynamische Bibliothek:

   $$u = \sum_{i=0} \varphi_i \left(\sum_{j=0} c_{ij} \eta_{ij} \right) \quad \text{(vgl. Gleichung 2.2, S. 2)}$$

Eine Verrechnung von u mit weiteren globalen Basisfunktionen ist möglich. Für die Auswertung müssen folgende Funktionalitäten bereit gestellt werden:

- Funktionseingabe
- Auswahl der gewünschten Basisfunktionen und des Polynomgrads
- Gitterverfeinerung
- Gitterschnitt
- Automatisches Lesen und Einsetzen von Koeffizienten

- Funktionserweiterung des Parsers
- Verwendung anderer Basisfunktionen

Für die Berechnung werden die benötigten Koeffizienten vorausgesetzt, wie sie im Kapitel 2.3.5 (S. 12) beschrieben wurden.

Abbildung 4.2 zeigt den Entwurf für die GUI des Plug-ins.

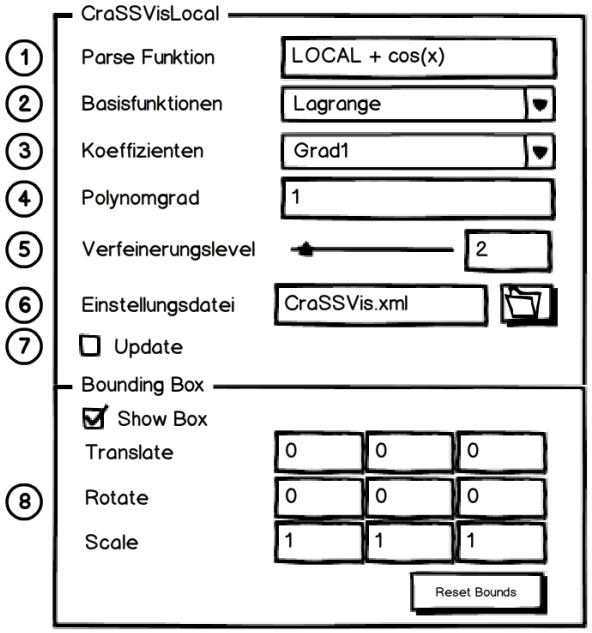


Abbildung 4.2: Entwurf für die GUI des Plug-ins CraSSVisLocal

Die in der Abbildung 4.2 enthaltenen Funktionen (umrandete Ziffern 1 bis 8) sowie die in diesem Zusammenhang wichtigen Begriffe werden im Folgenden näher beschrieben.

1: Parse Funktion

Die auszuwertende lokale Funktion kann in diesem Textfeld mit globalen Basisfunktionen verrechnet werden. „LOCAL“ dient als Platzhalter für das Ergebnis der Auswertung der lokalen Funktion. Wie in CraSSVisGlobal können Koeffizienten ersetzt werden. Wenn das Textfeld leer ist, wird die lokale Funktion unverändert ausgewertet.

2: Basisfunktionen

Aus der Combo-Box kann der Benutzer die gewünschten Basisfunktionen auswählen. Zur Verfügung stehen nur die Lagrange-Polynome. Die Lagrange-Polynome werden zur Auswertung als lokale Basisfunktion verwendet. Die Combo-Box wird verwendet, um eine potentielle Erweiterung zusätzlicher Basisfunktionen zu ermöglichen.

3: Koeffizienten
In der Combo-Box werden alle in der vtu-Datei verfügbaren Cell-Daten als mögliche Koeffizienten für die lokalen Basisfunktionen zur Auswahl durch den Benutzer angezeigt.

4: Polynomgrad
Der gewünschte Polynomgrad für die lokalen Basisfunktionen kann durch den Benutzer eingegeben werden. Standardmäßig ist Polynomgrad 1 eingetragen.

5: Verfeinerungslevel
Das Verfeinerungslevel des Gitters kann durch den Benutzer mit dem Schieberegler zwischen 0 und 10 eingestellt werden. Auch eine direkte Eingabe des Levels im Textfeld ist möglich. Ein Wert kleiner Null oder größer Zehn wird in den entsprechenden Minimal- oder Maximalwert umgewandelt.

6: Einstellungsdatei
Der Benutzer lädt optional über einen Dateiauswahldialog die **Einstellungs-XML**.

7: Update
Die Checkbox „Update“ hat keine Funktion. Sie dient dazu, ParaView eine Änderung vorzutäuschen, die in einer XML-Datei manuell vorgenommen wurde. Eine erneute Ausführung wird somit möglich. Da die Verwendung eines Knopfs in einem Filter-Plug-in umfangreicher ist als die Umsetzung der Funktionalität durch eine Checkbox, liegt ein Workaround vor.

8: Bounding Box
Die Funktion und die grafische Umsetzung der Bounding Box wird von ParaView zur Verfügung gestellt. Der Benutzer kann mit der Maus einen Bereich für die Verfeinerung auswählen. Über die Textfelder kann die Größe, Position und Drehung der Box beeinflusst werden. Ein ohne erkennbare Funktion vorgegebener „Reset Bounds‘‘-Knopf ist wahrscheinlich eine Fehlprogrammierung in ParaView. Es können nur Primitive, die vollständig in der Bounding Box enthalten sind, für die Auswertung der lokalen Basisfunktionen herangezogen werden.

Einstellungs-XML
Die Einstellungs-XML enthält die Pfade zur **Koeffizientendatei** und zur **dynamischen Bibliothek** als lokale Basisfunktionenalternative. Wenn ein Pfad zu einem Ordner angegeben wird, werden zusätzlich beschriebene Dateien in diesem Ordner, andernfalls im Ordner der Einstellungs-XML gesucht. Gänzlich davon abweichende Dateipfade werden unverändert beibehalten. Alle Inhalte sind optional.

Koeffizientendatei
Die Koeffizienten werden im *Matlab-Format* („m“ als Dateiendung) definiert. In dieser Datei wird pro Zeile ein Koeffizient im englischen Zahlenformat spezifiziert.

Dynamische Bibliothek
Die dynamische Bibliothek ersetzt die Lagrange-Polynome.

Funktionsweise von CraSSVisLocal

Die lokale Funktion wird über folgende Schritte, die vom Benutzer bzw. vom System ausgeführt werden, ausgewertet:

- Benutzer:
  1. Laden einer vtu-Datei mit einem unstrukturierten Gitter
  2. Auswählen der Koeffizienten
  3. Eingeben des Polynomgrads
  4. Einstellen des Verfeinerungslevels
  5. Auswählen der Einstellungs-XML (optional)
  6. Bestimmen des Verfeinerungsbereichs mit der Maus
  7. Drücken von „Apply“
- System:
  1. Überprüfen der Struktur der Daten und der Stetigkeit der Koeffizienten
  2. Schneiden des Ausgangsgitters mit der Bounding Box
  3. Verfeinern des Auswahlbereichs
  4. Integrieren aller Einstellungen
  5. Laden der Bibliotheken und ihre Integration in den Funktionsparser
  6. Aufstellen der lokalen Basisfunktionen
  7. Auswerten der Basisfunktionen pro Gitterpunkt

4.2.3 Spezifikation der Eingabedaten

In diesem Kapitel wird der formale Aufbau der Eingabedaten beschrieben. Um die Plug-ins ausführen zu können, ist mindestens eine vtu-Datei mit einem unstrukturierten Gitter notwendig. Optional können XML-Dateien verwendet werden. Die Spezifikation der XML-Dateien erfolgt im XSD-Format. Die Kompatibilität der Einstellungs-XML zu CraSSVisGlobal und CraSSVisLocal ist gewährleistet, eine Einstellungs-XML somit in beiden Plug-ins verwendbar. Eine beispielhafte Darstellung der vtu-Datei befindet sich im Listing 3.1 (S. 18).

Einstellungs-XML: *.xml

Listing 4.1 Einstellungs-XSD-Spezifikation für CraSSVisGlobal

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
       attributeFormDefault="unqualified">

4    <xs:element name="CraSSVis" type="CraSSVis_t" />
5    <xs:element name="FunctionLib" type="FunctionLib_t" />

7    <xs:complexType name="CraSSVis_t">
8      <xs:sequence>
9        <xs:element name="FolderPath" type="xs:string" minOccurs="0" maxOccurs="1"/>
10       <xs:element name="CoefficientsMatlabPath" type="xs:string" minOccurs="0"
             maxOccurs="1"/>
11       <xs:element name="BasesFunctionXMLPath" type="xs:string" minOccurs="0"
             maxOccurs="1"/>
12       <xs:element name="LocalFunctionLibPath" type="xs:string" minOccurs="0"
             maxOccurs="1"/>
13       <xs:element ref="FunctionLib" minOccurs="0" maxOccurs="unbounded"/>
14     </xs:sequence>
15   </xs:complexType>

17   <xs:complexType name="FunctionLib_t">
18     <xs:sequence>
19       <xs:element name="Path" type="xs:string" minOccurs="1" />
20     </xs:sequence>
21   </xs:complexType>

23 </xs:schema>
```

- → 9: Vollständiger Pfad zu einem Ordner, in dem die XML-Dateien liegen. Beim Fehlen dieses Elements wird der Ordner dieser XML-Datei verwendet. Vollständige Dateipfade werden nicht beeinflusst.
- → 10: Pfad/Name zur Koeffizientendatei.
- → 11: Pfad/Name zur Funktions-XML. Wird in CraSSVisLocal nicht verwendet.
- → 12: Pfad/Name zur dynamischen Bibliothek für die alternativen lokalen Basisfunktionen. Wird in CraSSVisGlobal nicht verwendet.
- → 13: Liste mit Pfaden/Namen (→ 19) zu den einzelnen dynamischen Bibliotheken, die den Parser erweitern.

Alle XML-Elemente sind optional.

Koeffizientendatei (Matlab): *.m

Listing 4.2 Spezifikation der Koeffizientendatei

```
1 <Variablen Name> = [
2         Zahl1
3         ZahlN
4 ]
```

→ 1: „<Variablen Name>“ steht für den Bezeichner einer Variablen mit dessen Inhalt zwischen den eckigen Klammern. Dieser Bezeichner wird von Matlab verwendet und hat für CraSSVis keine Bedeutung.

→ 2-3: Beliebig viele Zahlen (jeweils eine Zahl pro Zeile) im englischen Zahlenformat.

Funktions-XML: *.xml

Listing 4.3 Funktions-XSD-Spezifikation für CraSSVisGlobal

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
       attributeFormDefault="unqualified">

4    <xs:element name="BaseFunctions" type="BaseFunctions_t" />
5    <xs:element name="Function" type="Function_t" />

7    <xs:complexType name="BaseFunctions_t">
8      <xs:sequence>
9        <xs:element ref="Function" minOccurs="0" maxOccurs="unbounded" />
10     </xs:sequence>
11   </xs:complexType>
12   <xs:complexType name="Function_t">
13     <xs:sequence>
14       <xs:element name="Function" type="xs:string" minOccurs="1" />
15     </xs:sequence>
16   </xs:complexType>

18 </xs:schema>
```

→ 9: Eine Liste mit → 14 Basisfunktionen (z. B. # * $cos(x)$)

4.2.4 Spezifikation der Ausgabedaten

Das von beiden Plug-ins erzielte Ergebnis ist ein weiteres *Double-Array* (siehe Listing 3.1, S. 18 → 6). Das „CraSSVisGlobal“ bzw. „CraSSVisLocal“ genannte Array, welches für jeden ausgewerteten Punkt einen Wert enthält, wird dem Benutzer automatisch als aktives Array in ParaView präsentiert.

5 Entwurf von CraSSVis

Das Entwurfsmuster für CraSSVis wird durch ParaView vorgegeben. Seine Drei-Komponentenarchitektur wirkt sich auf die Kompilation (siehe Kapitel 5.1) aus. CraSSVis-Global und CraSSVisLocal sind strikt getrennte Komponenten, welche auf der gemeinsamen Komponente „CraSSVisGeneral“ aufbauen, in welcher die Funktionalitäten beider Plug-ins versammelt sind.

Im Fließtext werden Klassennamen fett, Methoden in Klassen in Courier und Klassenvariablen kursiv hervorgehoben. Die Syntax und Semantik der Pfeilsymbole in den verwendeten Klassendiagrammen (vgl. Abbildung 5.1) unterscheidet sich von der Definition eines UML-Diagramms, Sichtbarkeiten von Variablen sind allerdings gleich. Dies soll ein intuitives schnelles Verständnis der Diagramme ermöglichen.

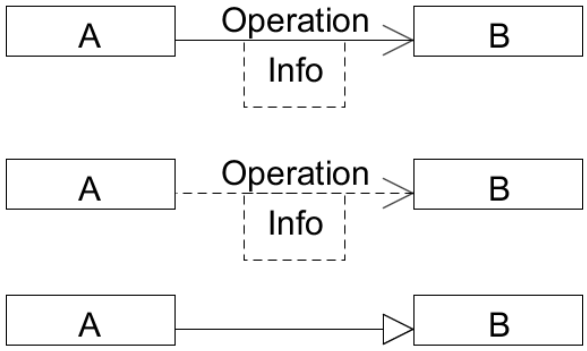


Abbildung 5.1: Syntax und Semantik der Klassendiagramme

$\rightarrow$: Klasse **A** führt eine `Operation` mit der Klasse **B** durch. Im Zusammenhang mit der Operation stehende *Informationen*, die sich auf Klasse **B** beziehen, stehen unterhalb des Pfeils.
Beispiel: Klasse **Matrix** `transformiert` Klasse **Punkt** *auf das Einheitsprimitiv*.

$\dashrightarrow$: Klasse **A** verwendet Klasse **B**, um eine `Operation` durchzuführen. Im Zusammenhang mit der Operation stehende *Informationen*, die sich auf Klasse **B** beziehen, stehen unterhalb des Pfeils.
Beispiel: Klasse **Dreiecktransformation** `benötigt` Klasse **Matrix** *zur Transformation*.

$-\triangleright$: Klasse **A** erbt von Klasse **B**.

Die Struktur der Klasse, die die Daten verarbeitet, ist vorgegeben und wird von der Basisklasse **vtkDataSetAlgorithm** und der Spezifikations-XML-Datei bestimmt, da CraSSVis als ein ParaView-Filter entwickelt wurde. Der eigentliche Filtervorgang wird in der überschriebenen Methode `RequestData(vtkInformation, vtkInformationVector, vtkInformationVector)` durchgeführt, wodurch die Eingangsdaten in Ausgangsdaten umgewandelt werden. In ParaView wird die Methode nach dem Drücken des „Apply"-Knopfs aufgerufen. Auf den speziellen Aufbau der Klasse wird später eingegangen.

5.1 Kompilation eines Plug-ins für ParaView

ParaView schreibt einen genau definierten Ablauf für die Kompilation eines Plug-ins vor. Die C++-Dateien enthalten die Funktionalitäten für ein Plug-in, die XML-Dateien legen die Eingabedaten und die Benutzeroberfläche fest. Beide Dateitypen werden in einem CMake-Script (im Folgenden Build-Script genannt) kompiliert und ergeben das Plug-in [Kita]. Eine ausführliche Anleitung [Kitf] für die Kompilation ist auf der Homepage von ParaView zu finden.

Build-Script

Die Build-Scripte für alle Plug-ins sind im Aufbau identisch. Sie unterscheiden sich durch spezifische Dateien jedoch in ihren Inhalten. Ein Build-Script ist exemplarisch in Listing 5.1 dargestellt. Die Dateinamen spezifischer Teile wurden durch Beispieldateinamen ersetzt und nur die wichtigen Stellen des Scripts werden aufgeführt. Konfigurationsmöglichkeiten, die für die hier entwickelten Plug-ins nicht benötigt wurden, werden nicht aufgeführt.

Listing 5.1 Build-Script für die Plug-ins

```
1  ADD_PARAVIEW_PLUGIN(DasPlugin "1.0"
2    DOCUMENTATION_DIR
3      "doc"
4    SERVER_MANAGER_XML
5      DasPlugin.xml
6    SERVER_MANAGER_SOURCES
7      plugin.cpp
8    GUI_RESOURCE_FILES
9      DasPluginGUI.xml)
```

→ 1: Initialisierung mit Name und Version des Plug-ins.
→ 2: Vollständiger Pfad zum Dateiordner der html-Benutzerdokumentation des Plug-ins.
→ 4: Beschreibung der Komponenten des Plug-ins, die auf dem Server zur Berechnung durch ParaView benötigt werden, durch eine XML-Datei.
→ 6: Gesamtheit aller auf dem Server ausgeführten C++-Dateien.
→ 8: XML-Datei zur Beschreibung rein GUI-spezifischer Elemente (z. B. Menüeinträge).

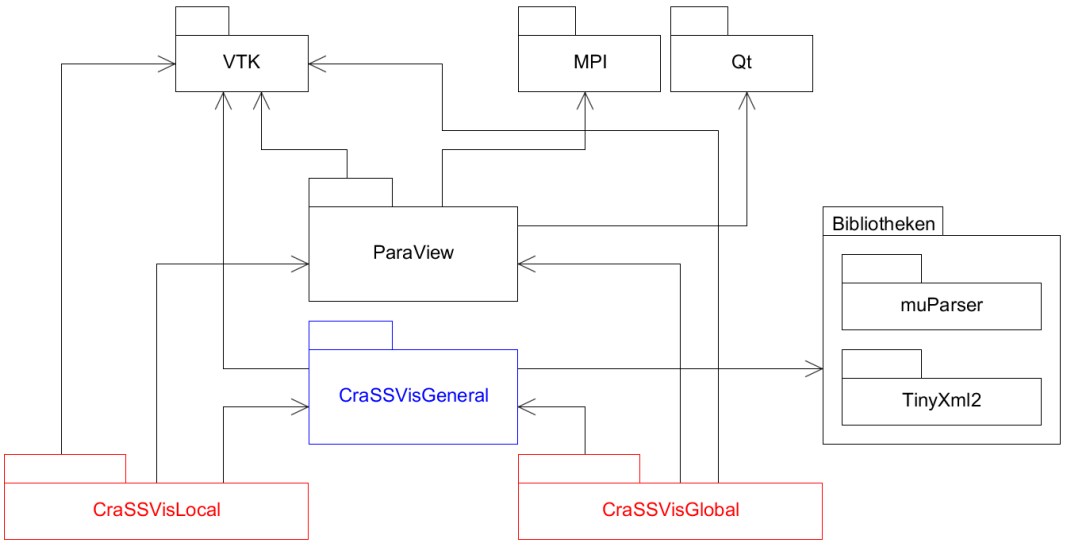


Abbildung 5.2: Komponentendiagramm für die Entwicklung der Plug-ins

Abbildung 5.2 bietet einen Überblick über den Aufbau der Komponenten der Plug-ins.

MPI und Qt
: MPI und Qt werden für das Funktionieren der Plug-ins nicht benötigt, allerdings für die Funktion von ParaView bzw. VTK. Sie sind nur der Vollständigkeit halber aufgeführt.

VTK
: Die VTK-Version wurde eigens auf ParaView zugeschnitten. Eine Integration neuerer VTK-Versionen ist mit wenig Aufwand möglich.

ParaView
: Der Einsatz von ParaView ist durch die Verwendung automatisch generierter Klassen des Build-Scripts für die Plug-ins notwendig. Spezielle Klassen werden nicht verwendet.

TinyXML2
: In der TinyXML2-Bibliothek werden die XML-Dateien gelesen.

muParser
: In der muParser-Bibliothek werden textuelle globale Funktionen geparst und ausgewertet.

CraSSVisGeneral
: CraSSVisGeneral sammelt die von den zwei Plug-ins benötigten gemeinsamen Klassen (z. B. zur Fehlerbehandlung oder für Gitteroperationen).

CraSSVisGlobal
: CraSSVisGlobal enthält alle Klassen des Plug-ins zur Auswertung globaler expliziter Funktionen.

CraSSVisLocal
: CraSSVisLocal enthält alle Klassen des Plug-ins zur Auswertung lokaler Basisfunktionen mit/ohne komplexe(r) Beschreibung.

5.2 Entwurf der CraSSVisGeneral-Komponente

In CraSSVisGeneral wird eine Vielzahl der von beiden Plug-ins benötigten gemeinsamen Klassen gesammelt. Seine wichtigsten Aufgaben sind Gitteroperationen, Transformationen auf das Referenzprimitiv, die Verwaltung der XML-Dateien sowie das Diagnostizieren und Anzeigen von Fehlern. CraSSVisGeneral besteht aus insgesamt 15 Klassen.

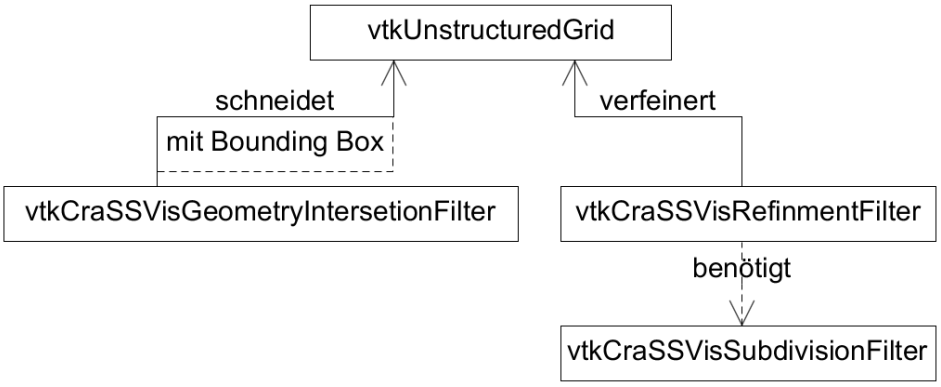


Abbildung 5.3: Gitteroperationen in der CraSSVisGeneral-Komponente

Jede Klasse der Gitteroperationen erhält als Eingabegitter ein **vtkUnstructuredGrid** und schneidet und verfeinert dieses (vgl. Abbildung 5.3).

vtkCraSSVisGeometryIntersectionFilter
: Mit der Klasse **vtkCraSSVisGeometryIntersectionFilter** wird das Gitter mit der Bounding Box, welche als **vtkImplicitFunction** repräsentiert ist, geschnitten. Die **vtkImplicitFunction** wird in eine geometrische Repräsentation umgewandelt, die die Basisklasse **vtkBoxClipDataSet** verwendet. Dafür wird ein Quader erzeugt und mit Hilfe der ImplicitFunction (in **vtkImplicitFunction** enthalten) transformiert. Der transformierte Quader wird mit dem Gitter geschnitten.

vtkCraSSVisRefinmentFilter
: Mit der Klasse **vtkCraSSVisRefinmentFilter** wird die Klasse **vtkCraSSVisSubdivisionFilter** n-Mal (je nach Verfeinerungslevel) aufgerufen. Für eventuell später hinzuzufügende Primitivtypen dient **vtkCraSSVisRefinmentFilter** als Puffer.

vtkCraSSVisSubdivisionFilter
: Mit der Klasse **vtkCraSSVisSubdivisionFilter** wir das Gitter verfeinert. Das Gitter wird von Grund auf neu erzeugt, wobei gegebenenfalls alle Primitive außer VTK_TRIANGLE und VTK_TETRA ausgeschieden werden. Die Verfeinerung erfolgt wie in Kapitel 2.2.3 (S. 5) beschrieben.

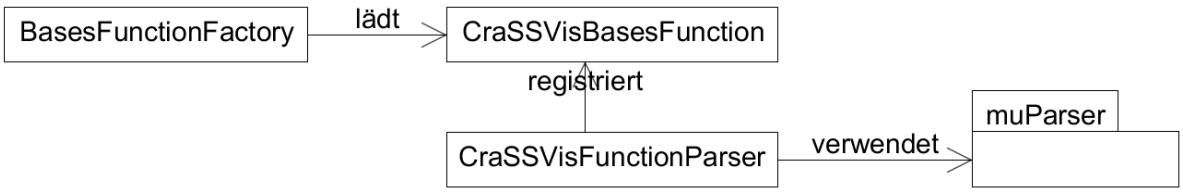


Abbildung 5.4: Parser in der CraSSVisGeneral-Komponente

Der Parser wandelt die textuelle Repräsentation der mathematischen Funktion in eine Darstellung um, die von einem Plug-in ausgewertet wird. Die Erweiterung des Parsers erfolgt über die **muFunctionParserWrapper**-Klasse (siehe Kapitel 5.5.1, S. 40). Abbildung 5.4 zeigt den Aufbau des Parsers in CraSSVisGeneral.

BasesFunctionFactory
Die Klasse **BasesFunctionFactory** lädt die dynamische Bibliothek, die den Parser erweitert.

CraSSVisBasesFunction
Die Klasse **CraSSVisBasesFunction** beinhaltet den Dateipfad einer dynamischen Bibliothek.

CraSSVisFunctionParser
Die Klasse **CraSSVisFunctionParser** wertet eine mathematische Funktion in Textform unter Parsen des Textes und Registrierung der Bibliotheken für einen Gitterpunkt aus. Das Parsen übernimmt der Parser aus dem Package muParser.

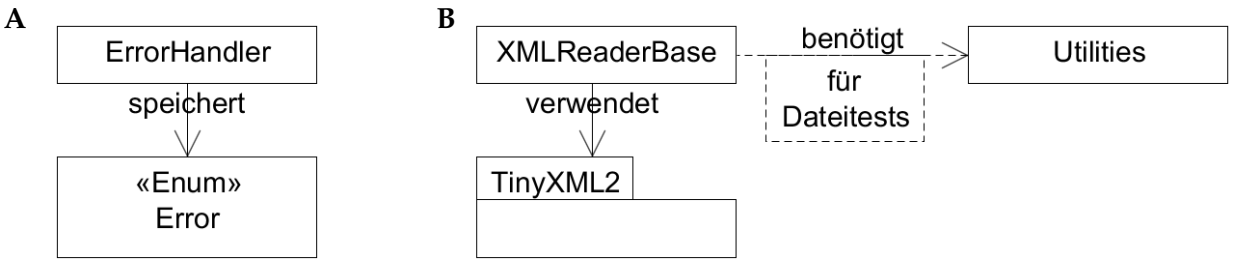


Abbildung 5.5: Fehlerbehandlung und XML-Komponente in CraSSVisGeneral
(A) Fehlerbehandlung **(B)** XML-Komponente

Abbildung 5.5 zeigt den Aufbau der Fehlerbehandlung und der XML-Komponente.

ErrorHandler
Die Klasse **ErrorHandler** speichert und lokalisiert eventuell auftretende Fehler. Die möglichen Fehler werden über das Enum *Error* gespeichert.

XMLReaderBase
Die Klasse **XMLReaderBase** ist ein Wrapper für den Zugriff auf TinyXML2. Während des Lesens einer XML-Datei wird seine Korrektheit überprüft.

Utilities
Die statische Klasse **Utilities** stellt u. a. Methoden zur Dateipfadprüfung oder Dimensionsberechnungen des Gitters zur Verfügung.

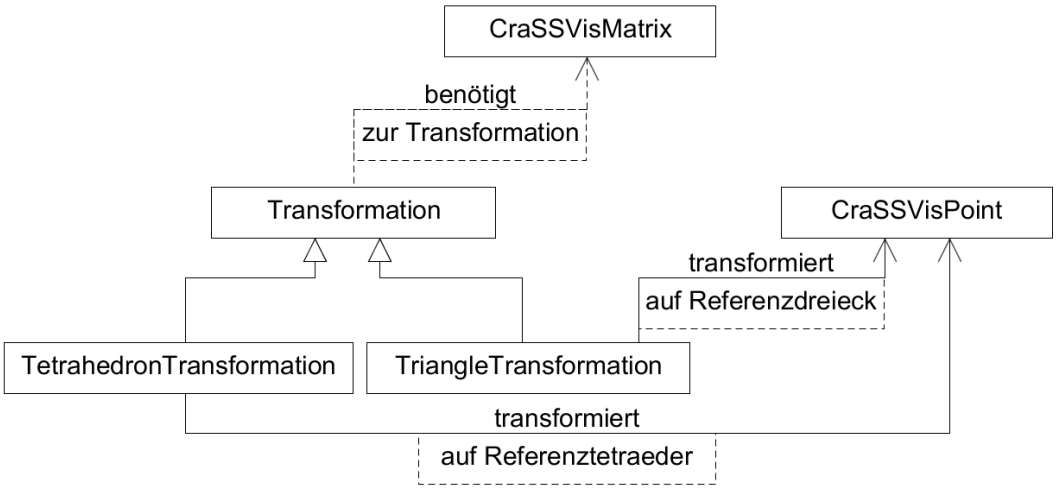


Abbildung 5.6: Vererbungshierarchie und Aufbau der Transformationen in CraSSVisGeneral

CraSSVisLocal benötigt mehrere Transformationen, die ein beliebig gelegenes Primitiv auf das Referenzprimitiv transformieren. Da aber Lagrange-Polynome durch eine dynamische Bibliothek ersetzt werden können, sind sie ein Teil von CraSSVisGeneral. Die Beziehung der Transformationsklassen wird in der Abbildung 5.6 aufgezeigt.

CraSSVisMatrix
Die Klasse **CraSSVisMatrix** repräsentiert eine $n \times n$-Matrix, die grundlegende Funktionen wie Invertieren, Lösen oder Multiplizieren einer Matrix zur Verfügung stellt. Als Lösungsalgorithmus wird das standardmäßige Gauß'sche-Eliminationsverfahren verwendet.

CraSSVisPoint
Die Klasse **CraSSVisPoint** repräsentiert einen 3D Punkt und bietet einen vereinfachten Umgang mit der double[3]-Punktrepräsentation von VTK.

Transformation
Die Basisklasse **Transformation** transformiert ein Primitiv auf ein Referenzprimitiv und dieses auf ein beliebig gelegenes Primitiv.

TetrahedronTransformation
Die von **Transformation** erbende Klasse **TetrahedronTransformation** transformiert (siehe Kapitel 2.3.3, S. 11) einen 3D Gitterpunkt auf den Referenztetraeder.

TriangleTransformation
Die von **Transformation** erbende Klasse **TriangleTransformation** transformiert (siehe Kapitel 2.3.3, S. 11) einen 2D Gitterpunkt auf das Referenzdreieck.

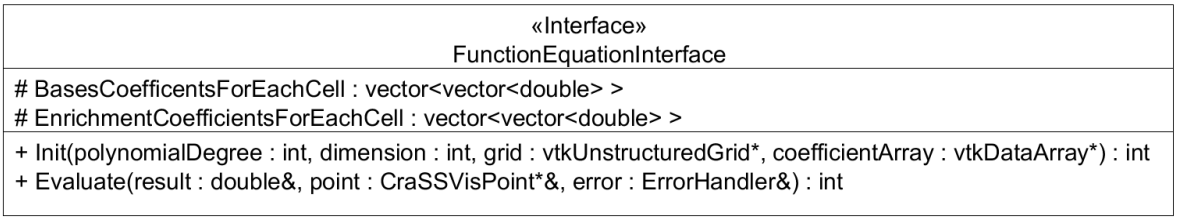


Abbildung 5.7: Aufbau des Interface für lokale Basisfunkionen in CraSSVisGeneral

Die Klasse **FunctionEquationInterface** für die Auswertung der lokalen Basisfunktionen wird in Abbildung 5.7 beschrieben. Die im Interface enthaltenen abstrakten Methoden werden in Kapitel 5.5.2, S. 41 beschrieben und sind hier noch nicht aufgeführt. Das Interface **FunctionEquationInterface** wertet die lokalen Basisfunktionen aus und bildet die Schnittstelle zwischen CraSSVisLocal und der dynamischen Bibliothek. Das Interface stellt dem Entwickler einer dynamischen Bibliothek Transformationen auf das Referenzprimitiv zur Verfügung.

Init

In der Methode `Init` wird die Klasse **FunctionEquationInterface** über den Polynomgrad und die Gitterdimension informiert. Beim Fehlen von Enrichments werden die Koeffizienten der Lagrange-Polynome eines jeden Primitivs in der Variablen *BasesCoefficentsForEachCell*, andernfalls in *EnrichmentCoefficientsForEachCell* gespeichert.

Evaluate

Die Methode `Evaluate` der Klasse **FunctionEquationInterface** wertet die Funktion u unterschiedlich aus:

1. Ohne Enrichments:

$$u = \sum_{i=0} g_i c_i$$

2. Mit Enrichments:

$$u = \sum_{i=0} \varphi_i \left(\sum_{j=0} c_{ij} \eta_{ij} \right)$$

5.3 Entwurf von CraSSVisGlobal

Die Klasse **vtkCraSSVisGlobal** in der CraSSVisGlobal-Komponente, die die Funktionalität des Plug-ins (CraSSVisGlobal) enthält, wird im Klassendiagramm der Abbildung 5.8 dargestellt. Die Komponente besteht aus zwei Klassen:

- vtkCraSSVisGlobal (Auswertung der Funktion, Schnitt und Verfeinerung des Gitters)
- XMLReaderGlobal (Lesen der XML-Dateien).

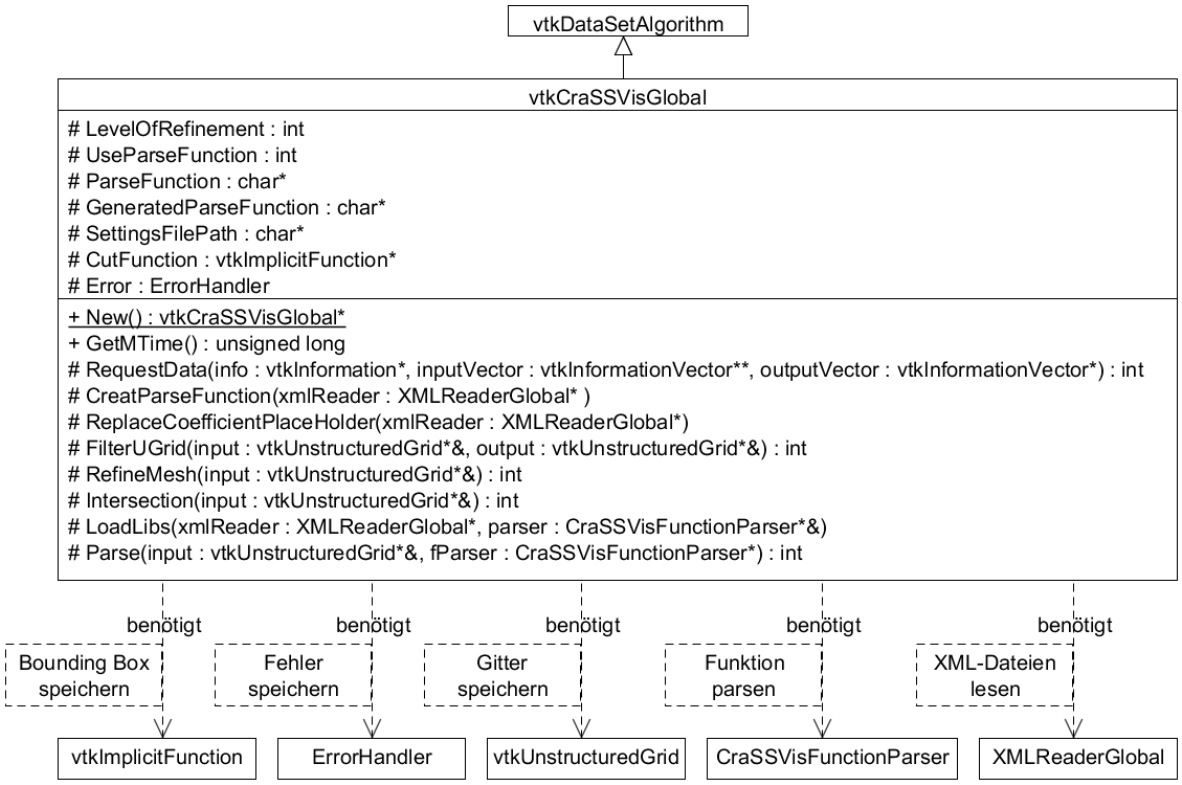


Abbildung 5.8: Klassendiagramm von vtkCraSSVisGlobal

Im Klassendiagramm werden „Getter“ und „Setter“ der Übersichtlichkeit halber nicht berücksichtigt. Die einzelnen Methoden der **vtkCraSSVisGlobal**-Klasse werden im Folgenden detailliert beschrieben:

GetMTime

Jedes Objekt in VTK besitzt einen Zeitstempel, der einen Änderungszeitpunkt genau festlegt. Die Bounding Box ist durch *CutFunction* repräsentiert und kann in der GUI vom Benutzer bearbeitet werden. Der Zeitstempel der Bounding Box wird nicht automatisch verändert, darum ist eine explizite Bearbeitung über die Methode `GetMTime` notwendig.

RequestData
In der Methode `RequestData` findet die eigentliche Funktion des Filters statt. Zuerst wird das Eingabegitter überprüft. Bei Vorliegen eines **vtkUnstructuredGrid** wird `FilterUGrid` aufgerufen. Die Prüfung des Gittertyps erlaubt nachträgliche Gittertyperweiterungen.

FilterUGrid
In der Methode `FilterUGrid` wird das Gitter mit der Bounding Box geschnitten (`Intersection`). Das entstandene Gitter wird weiter verfeinert (`RefineMesh`) und die Parsefunktion erstellt (`GenerateParseFunction`), die #-Zeichen werden ersetzt (`ReplaceCoefficientPlaceHolder`) und die Bibliotheken geladen (`LoadLibs`). Zum Schluss wird die Funktion ausgewertet (`Parse`).

Intersection
In der Methode `Intersection` wird das Gitter mit der Bounding Box über die Klasse **vtkCraSSVisGeometryIntersection** geschnitten.

RefineMesh
In der Methode `RefineMesh` wird über die Klasse **vtkCraSSVisRefinment** das Gitter verfeinert.

CreateParseFunction
In der Methode `CreateParseFunction` wird die Parsefunktion erzeugt und in die Variable *GeneratedParseFunction* gespeichert.

ReplaceCoefficientPlaceHolder
In der Methode `ReplaceCoefficientPlaceHolder` werden alle #-Zeichen in der Parsefunktion durch die geladenen Koeffizienten ersetzt. Alle #-Zeichen, die nicht durch geladene Koeffizienten ersetzt werden können, erhalten den Wert 1.

LoadLibs
In der Methode `LoadLibs` wird jede dynamische Bibliothek mit Hilfe der Klasse **BasesFunctionFactory** geladen und im Parser registriert.

Parse
In der Methode `Parse` wird die Parsefunktion für jeden Gitterpunkt ausgewertet und als *PointData* mit dem Namen „CraSSVisGlobal“ an das Gitter angehängt.

5.4 Entwurf von CraSSVisLocal

Die Klasse **vtkCraSSVisLocal** in der CraSSVisLocal-Komponente, die die Funktionalität des Plug-ins (CraSSVisLocal) enthält, wird im Klassendiagramm der Abbildung 5.9 dargestellt. Die Komponente besteht aus:

- vtkCraSSVisLocal (Auswertung der lokalen Basisfunktionen, Schnitt und Verfeinerung des Gitters)
- CraSSVisCheckCoefficients (Überprüfung der Koeffizienten auf Stetigkeit)
- CraSSVisXMLReaderLocal (Lesen der XML-Dateien)
- Lagrange (Aufstellung der Lagrange-Polynome)
- LagrangeBasesFunctions (Auswertung der Lagrange-Polynome)
- LocalFunctionFactory (Laden der dynamischen Bibliothek für die alternativen lokalen Basisfunktionen)

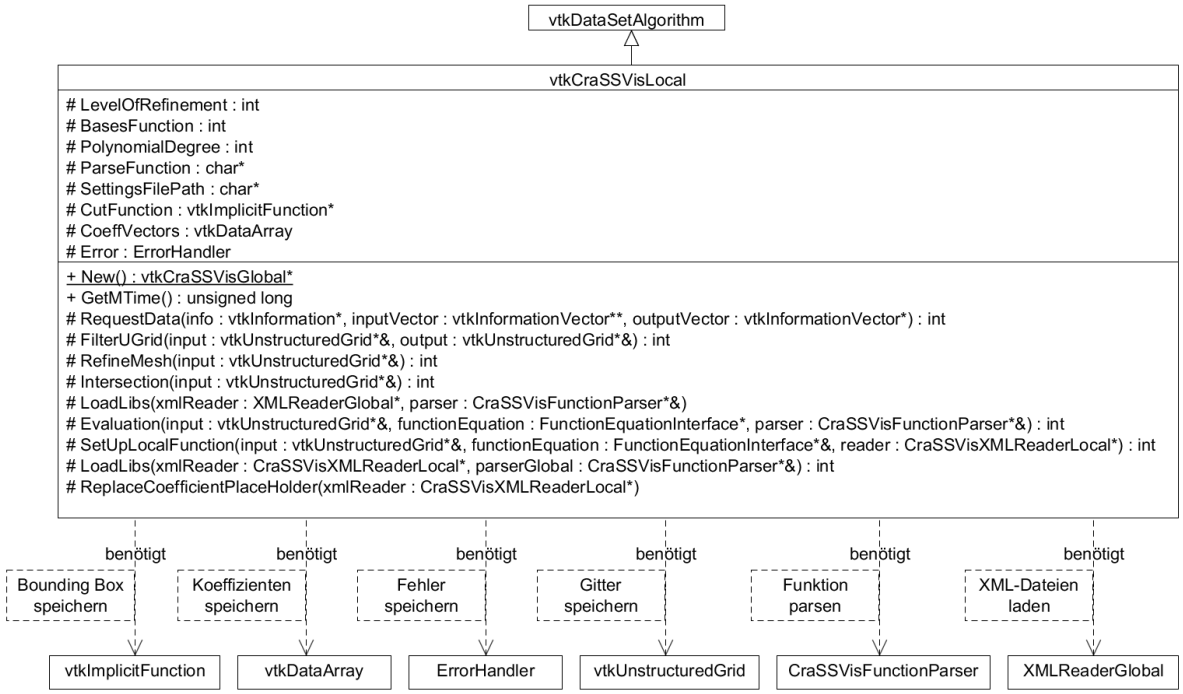


Abbildung 5.9: Klassendiagramm von vtkCraSSVisLocal

Im Klassendiagramm werden „Getter“ und „Setter“ der Übersichtlichkeit halber nicht berücksichtigt. Die einzelnen Methoden der **vtkCraSSVisLocal**-Klasse werden im Folgenden detailliert beschrieben. Einige Methoden sind identisch zu denen in **vtkCraSSVisGlobal** (siehe Abbildung 5.3, S. 37). Sie werden nicht erneut beschrieben.

SetUpLocalFunction
In der Methode `SetUpLocalFunction` wird entschieden, ob die Lagrange-Polynome oder die lokalen Basisfunktionen aus der Bibliothek zur Berechnung herangezogen werden. Nach der Entscheidung werden die Basisfunktionen aufgestellt.

Evaluation
In der Methode `Evaluation` werden für jeden Gitterpunkt die lokalen Basisfunktionen ausgewertet, mit der geparsten Funktion verrechnet und als *PointData* mit dem Namen „CraSSVisLocal“ an das Gitter angehängt.

5.5 Dynamische Bibliotheken

Die Plug-ins können mit parsererweiternden Bibliotheken und einer lokalen Basisfunktionsbibliothek erweitert werden. Die lokale Basisfunktionsbibliothek dient später zur Auswertung des Lösungsraums der PUM in CraSSVisLocal.

5.5.1 Parsererweiternde Bibliotheken

Parsererweiternde Bibliotheken sind dynamische Bibliotheken. Sie erweitern den Funktionsumfang des Funktionsparsers und müssen dazu beim Parser registriert werden. Hierfür implementiert jede dieser Bibliotheken die Methode `RegisterParser` des Interface **FunctionWrapper**. Die Registrierung am Parser erfolgt über einen „FunctionPointer“ durch die Methode `DefineFun`. Der „FunctionPointer“ referenziert auf die Methode, die die Berechnung vornimmt. Zum Erzeugen und Löschen der Klasse werden `CreateClass` und `DestroyClass`, als extern ”C” deklariert, benötigt. Der Aufbau einer dynamischen Bibliothek für den Parser ist in Abbildung 5.10 dargestellt.

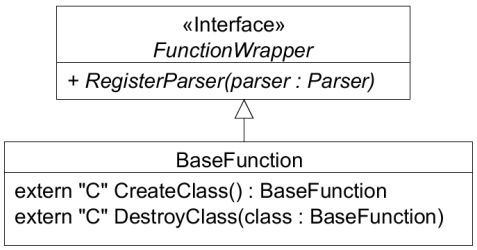


Abbildung 5.10: Klassendiagramm der dynamischen Bibliotheken für den Parser

5.5.2 Lokale Basisfunktionenbibliothek

Jede neue lokale Basisfunktionenklasse muss über das Interface **FunctionEquationInterface** die Methoden `SetUpAllFunctions`, `CapFunction`, `Enrichment` u. `GetNumberOfEnrichments` implementieren. Um einen größtmöglichen Freiraum zu gewährleisten, sind die nötigen Methoden so abstrakt wie möglich gehalten. In CraSSVisLocal wird zuerst `SetUpAllFunctions` aufgerufen. Der Aufbau des Interface ist in Abbildung 5.11 dargestellt.

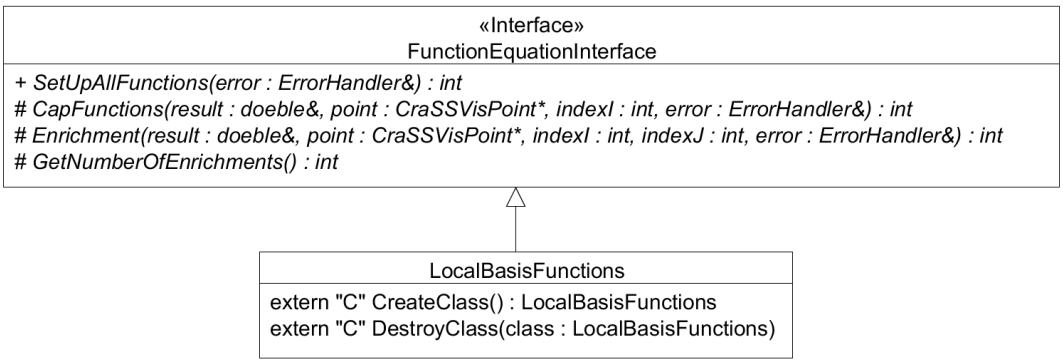


Abbildung 5.11: Klassendiagramm der dynamischen Bibliotheken für lokale Basisfunktionen

SetUpAllFunctions
: In der Methode `SetUpAllFunctions` werden die lokalen Basisfunktionen erzeugt.

CapFunctions
: In der Methode `CapFunctions` wird das Ergebnis der i-ten lokalen Basisfunktion für einen Gitterpunkt berechnet.

Enrichment
: In der Methode `Enrichment` wird das Ergebnis des ij-ten Enrichments für einen Gitterpunkt berechnet.

GetNumberOfEnrichments
: In der Methode `GetNumberOfEnrichment` wird die Anzahl der verwendeten Enrichments für jedes Primitiv angegeben.

6 Implementierung

6.1 Codequalität

Als Styleguide wurde der in ParaView-Projekten angewandte verwendet. Aus Gründen der besseren Lesbarkeit sind jedoch geschweifte Klammern in Extrazeilen (siehe Listing 6.1) aufgeführt.

Listing 6.1 Änderungen des ParaView-Styleguide

```
1  // Falsch
2  if (code) {
3  } else {
4  }

6  // Richtig
7  if (code)
8  {
9  }
10 else
11 {
12 }
```

Die Lesbarkeit des Codes wurde durch aussagekräftige Variablenbezeichnungen und Kommentare verbessert. Der Code kann bei einer späteren Weiterentwicklung als verständliche und nachvollziehbare Vorlage dienen.

6.2 Teststrategie und Dokumentation

Für das systematische Testen der einzelnen Methoden wird auf Modultests zurückgegriffen. Alle weiteren Funktionalitäten – vor allem in Bezug auf korrekte Datenerzeugung – werden mit Hilfe eines Systemtests überprüft. Mit den Modultests wurde eine 90%-ige Überdeckung der Komponente CraSSVisGeneral (siehe Abbildung 5.2, S. 32) erreicht. Für die Systemtests steht eine große Anzahl an Testdaten zur Verfügung, die eine Vielfalt von Anwendungsszenarien abdecken.

Zur Dokumentation des Codes dienten vor allem die mit Doxygen kompatiblen Kopfkommentare der einzelnen öffentlichen Methoden der verschiedenen Klassen. Zusätzlich werden implementierungsspezifische Einschränkungen durch ParaView im Code beschrieben.

6.3 Converter

CraSSVisLocal verwendet für die Koeffizienten in der Eingabedatei ein CellData-Array. Die Reihenfolge der Koeffizienten ist vom Referenzprimitiv abhängig. Die manuelle Erstellung einer korrekten Koeffizientenabfolge für Lagrange-Polynome (z. B. zu Testzwecken) ist sehr zeitaufwendig und fehleranfällig. Das Ersetzen der manuellen Erstellung durch eine erweiterte Darstellung in der Eingabedatei und eine anschließende Konvertierung in ein CellData-Array mit dem Plug-in CraSSVisLocalConverter führt zu einer schnellen und sicheren Erstellung der Koeffizientenabfolge. Jeder einzelne Koeffizient wird als PointData mit einer explizit definierten Stützstelle in der Datei verknüpft. Als Ausgabe wird ein CellData-Array mit dem Namen „CraSSVisLocalConverter“ erzeugt. Dadurch müssen Koeffizienten nicht mehrmals für einen Punkt definiert werden und die Stetigkeit ist von vornherein gegeben. Allerdings wird eine große Anzahl von Stützstellen benötigt – bei höherem Polynomgrad schnell mehrere tausend. Die Darstellung erleichtert das Einlesen der Koeffizienten für die Auswertung der Gleichung 2.6 (S. 7). Mit der Konvertierung erhält man die nötige Eingabedarstellung für CraSSVisLocal. Im Listing 6.2 wird ein Dreieck, auf dem die Funktion $f(x, y) = 0.8x^5 + 0.8y^5 - 5x^3 - 5y^3 + 4x + 4y$ mit Polynomgrad 2 ausgewertet wird, als Beispiel herangezogen. Zur Darstellung in ParaView wird jede Stützstelle als Vertex angegeben.

Zur manuellen Auswertung einer Funktion an den Stützstellen der zugrunde liegenden Primitive wurde ein kleines weiteres Plug-in entwickelt. Dieses wandelt eine textuelle mathematische Funktion in Stützstellen mit Koeffizienten für den gewünschten Polynomgrad im CraSSVisLocalConverter-Format um.

Beide Plug-ins sind sowohl für 2D- als auch für 3D-Gitter ausgelegt. Enrichments werden nicht unterstützt. Die erzeugten Koeffizienten können nur für die Auswertung der Gleichung 2.6 (S. 7) verwendet werden.

Listing 6.2 Beispieldatei zur Eingabe für den CraSSVisLocalConverter

```
1  <VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">
2    <UnstructuredGrid>
3      <Piece NumberOfPoints="6" NumberOfCells="7">
4        <PointData Scalars="CraSSVisLocalCoefficientGenerator">

6          <DataArray type="Float64" Name="CraSSVisLocalCoefficientGenerator" format="ascii"
               RangeMin="-20" RangeMax="20">
7            -20 0 20 -10 10 0
8          </DataArray>

10       </PointData>
11       <CellData>
12       </CellData>
13       <Points>

15         <DataArray type="Float32" Name="Points" NumberOfComponents="3" format="ascii"
               RangeMin="0" RangeMax="3.5355339059">
16           -2.5 -2.5 0
17           0 0 0
18           2.5 2.5 0
19           0 -2.5 0
20           2.5 0 0
21           2.5 -2.5 0
22         </DataArray>

24       </Points>
25       <Cells>
26         <DataArray type="Int64" Name="connectivity" format="ascii" RangeMin="0"
               RangeMax="5">
27           0 5 2
28           0 1 2 3 4 5
29         </DataArray>
30         <DataArray type="Int64" Name="offsets" format="ascii" RangeMin="3" RangeMax="9">
31           3
32           4 5 6 7 8 9
33         </DataArray>
34         <DataArray type="UInt8" Name="types" format="ascii" RangeMin="1" RangeMax="5">
35           5
36           1 1 1 1 1 1
37         </DataArray>
38       </Cells>
39     </Piece>
40   </UnstructuredGrid>
41 </VTKFile>
```

→ 7: Ein Koeffizient wird einer Stützstelle (→ 16-21) zugeordnet.

→ 16-21: Gesamtheit der für den verwendeten Polynomgrad notwendigen Stützstellen.

→ 27 u. 35: Die Indizes erzeugen unter Hinzunahme des Zellentyps das Dreieck.

→ 28 u. 36: Die Erzeugung eines Vertex erlaubt die Darstellung eines jeden Punkts in seiner Rohform.

7 Bewertung der Leistungsfähigkeit von CraSSVis

Im folgenden Kapitel wird die Leistungsfähigkeit von CraSSVisLocal und CraSSVisGlobal analysiert.

Die Performanz der Plug-ins wurde nach den Kriterien „Qualität der Auswertung“, „Länge der Rechenzeit“ und „notwendiger Speicherbedarf“ bewertet. Das Kriterium „Qualität der Auswertung“ wird über die optische Glattheit der dargestellten Funktion beurteilt. Der gesamte zeitliche Rechenaufwand für die Auswertung auf einem Verfeinerungslevel ergibt die Länge der Rechenzeit. Der Speicherbedarf errechnet sich aus den Speicherkapazitäten, die für jedes Verfeinerungslevel benötigt werden.

Die Plug-ins sind auf einen Prozessorkern ausgelegt. Daher wurden keine Tests mit mehreren Threads durchgeführt. Die Bewertung bezieht sich auf die im Folgenden beschriebenen Hard- und Softwarekonfigurationen:

Tabelle 7.1: Verwendete Hard- und Softwarekomponenten für die Bewertung

Betriebssystem	Ubuntu Linux (12.04) 64bit mit Kernel 3.5.0-17-generic
Prozessor	Intel i7-2600K @ 3.40GHz
Arbeitsspeicher	8GB
ParaView	4.0.1
GCC	4.7.2

7.1 Referenzfunktion für die Bewertung

Als Referenzfunktion für die Bewertung dient die Funktion:
$f(x,y) = 0.8x^5 + 0.8y^5 - 5x^3 - 5y^3 + 4x + 4y$. Abbildung 7.1 zeigt die Funktion in den Bereichen $x = -2.5,\ldots,2.5$ und $y = -2.5,\ldots,2.5$, graphisch dargestellt mit Hilfe von Maple 15. Beide Plug-ins verwenden zur Auswertung der Funktion dasselbe Gitter. Das Ausgangsgitter besteht aus Dreiecken $\left(\triangle_1\left(\begin{smallmatrix}-2.5\\2.5\end{smallmatrix}\right),\left(\begin{smallmatrix}2.5\\-2.5\end{smallmatrix}\right),\left(\begin{smallmatrix}2.5\\2.5\end{smallmatrix}\right)\right.$ und $\left.\triangle_2\left(\begin{smallmatrix}-2.5\\2.5\end{smallmatrix}\right),\left(\begin{smallmatrix}2.5\\2.5\end{smallmatrix}\right),\left(\begin{smallmatrix}-2.5\\2.5\end{smallmatrix}\right)\right)$.
Zur graphischen Übereinstimmung der Auswertungen werden die erzeugten Werte der Plug-ins mit dem „Warp By Scalar“-Filter um den Faktor 1 in z-Richtung verschoben. Das eben dargestellte Verfahren lässt sich vom 2D- auf den 3D-Fall übertragen.

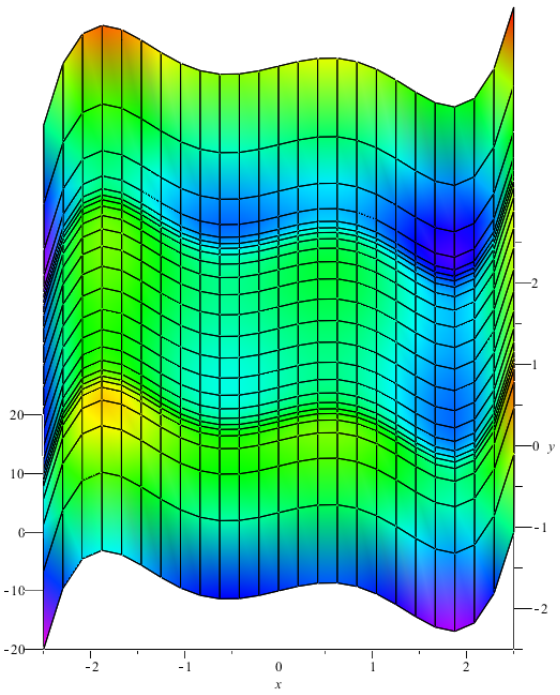


Abbildung 7.1: Referenzfunktion für die Bewertung in Maple

7.2 Qualitätsanalyse der Auswertungen mit CraSSVisGlobal

Da das Plug-in CraSSVisGlobal globale Basisfunktionen verwendet, wird die Referenzfunktion für jeden Gitterpunkt exakt ausgewertet. Bei einer genügend hohen Gitterverfeinerung kann die Referenzfunktion daher glatt dargestellt werden. In Abbildung 7.2 wird die Auswertung der Referenzfunktion mit CraSSVisGlobal auf verschiedenen Verfeinerungsleveln mit der entsprechenden Glattheit der Referenzfunktion dargestellt. Eine ausreichende optische Glattheit ist ab Verfeinerungslevel Fünf (siehe Abbildung 7.2 D) erreicht. Dies entspricht der Anzahl von 2048 Dreiecken und einer damit verbundenen Anzahl von 1089 Auswertungspunkten.

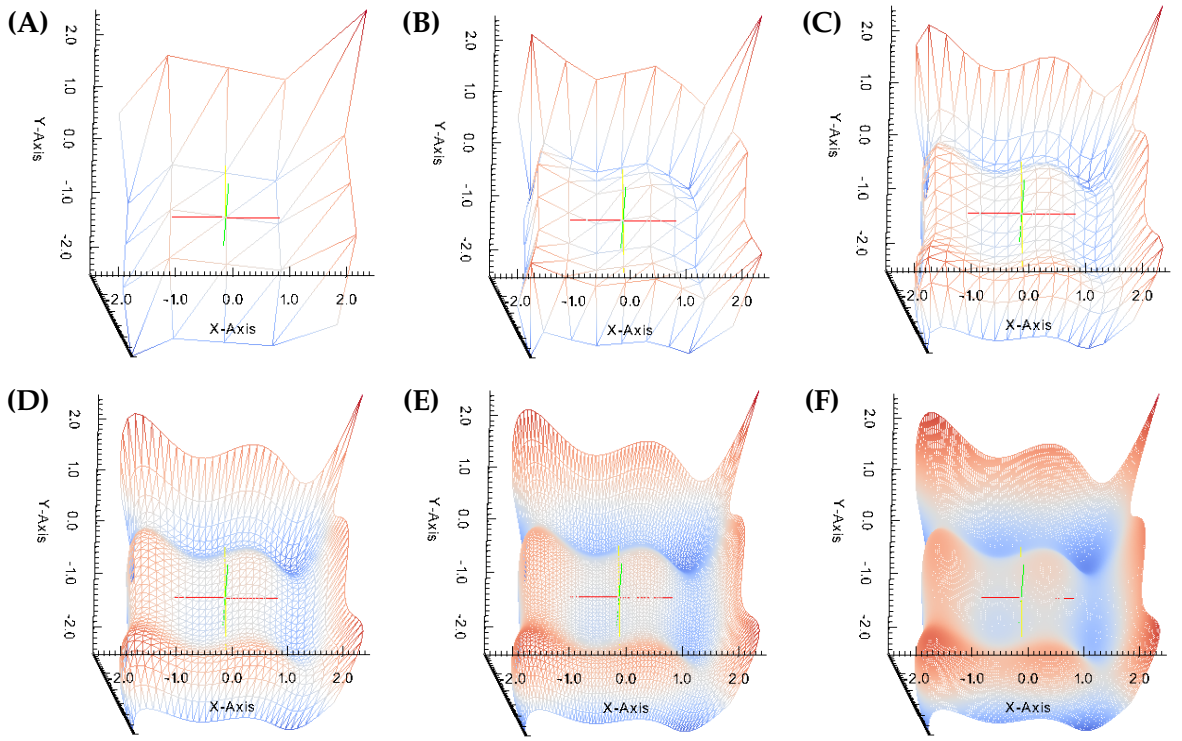


Abbildung 7.2: Auswertung der Referenzfunktion mit CraSSVisGlobal auf verschiedenen Verfeinerungsleveln
(A) Verfeinerungslevel 2 **(B)** Verfeinerungslevel 3 **(C)** Verfeinerungslevel 4
(D) Verfeinerungslevel 5 **(E)** Verfeinerungslevel 6 **(F)** Verfeinerungslevel 7

7.3 Qualitätsanalyse der Auswertungen mit CraSSVisLocal

Das Plug-in CraSSVisLocal wertet für jeden Gitterpunkt die jeweils nötigen Basisfunktionen ohne Enrichments aus. Für die optische Glattheit spielt nur der Polynomgrad eine Rolle, nicht aber das Verfeinerungslevel. Die Referenzfunktion wird auf Verfeinerungslevel 7 mit Lagrange-Polynomen des Grads 1 bis 6 ausgewertet. Mit steigendem Polynomgrad wird die Auswertung der Referenzfunktion immer exakter (vgl. Abbildung 7.3). Ab Polynomgrad 5 (Abbildung 7.3 E) ist die Referenzfunktion bestmöglich ausgewertet, da die Funktion Grad 5 besitzt.

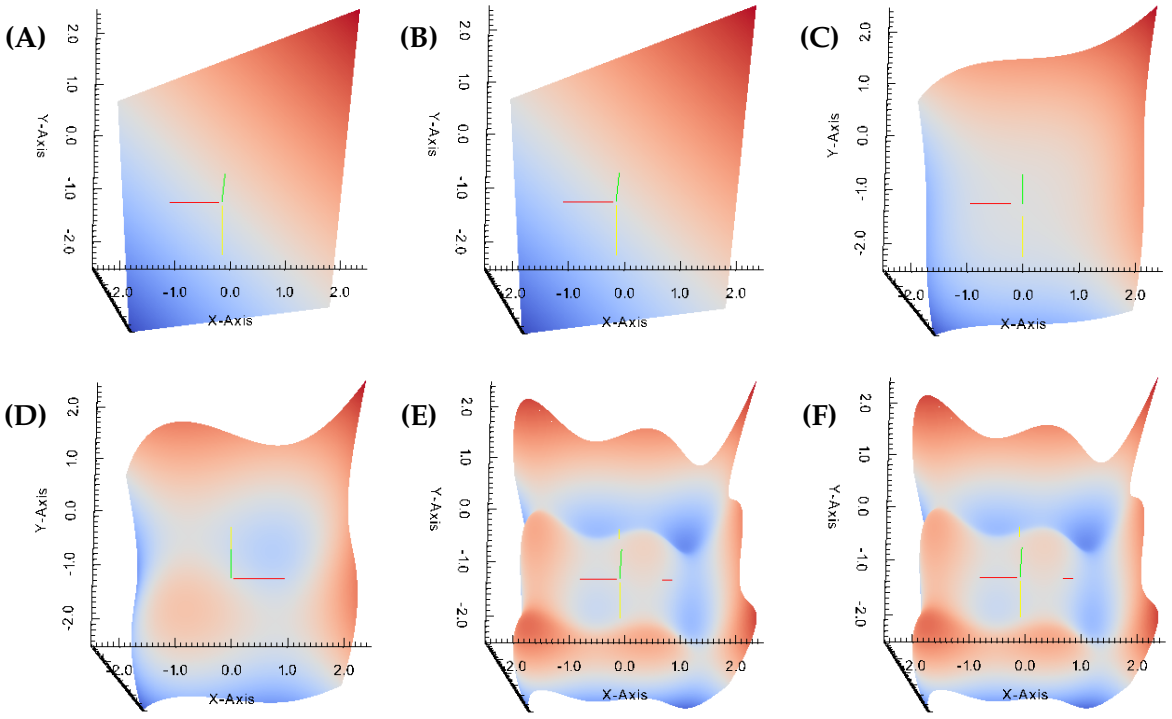


Abbildung 7.3: Auswertung der Referenzfunktion mit CraSSVisLocal bei verschiedenen Polynomgraden
(A) Polynomgrad 1 **(B)** Polynomgrad 2 **(C)** Polynomgrad 3
(D) Polynomgrad 4 **(E)** Polynomgrad 5 **(F)** Polynomgrad 6

Ab Verfeinerungslevel 5 werden selbst Funktionen mit einem hohen Polynomgrad optimal ausgewertet. Ab Polynomgrad 5 besteht zwischen der Auswertung einer globalen und einer lokalen Funktion kein Unterschied.

7.4 Analyse des Zeitaufwands der Auswertung mit CraSSVisGlobal

Die Auswertungszeiten auf Level 1 bis 10 sind durch den Graphen einer Exponentialfunktion (vgl. Abbildung 7.4) dargestellt. Diese Exponentialfunktion wird stark von der Anzahl an Verfeinerungen bestimmt. Die aufgewendete Zeit steigt mit $\mathcal{O}(n)$ zur Anzahl der Auswertungspunkte. Sie ist für die Gesamtdauer der Auswertung der Referenzfunktion irrelevant. Selbst für ca. 1 Million Auswertungspunkte werden nicht mehr als ca. 1.5 Sekunden für die Auswertung benötigt.

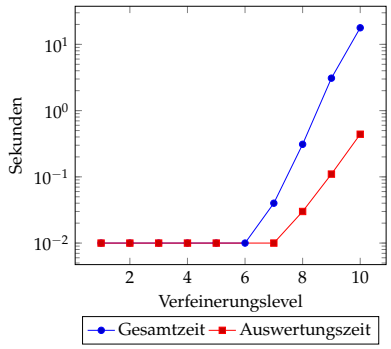


Abbildung 7.4: Gesamtzeit der Auswertung von CraSSVisGlobal auf verschiedenen Verfeinerungsleveln

7.5 Analyse des Zeitaufwands der Auswertung mit CraSSVisLocal

In Abbildung 7.5 wird der Zusammenhang zwischen der Gesamtzeit der Auswertung und der reinen Auswertungszeit der Basisfunktionen mit CraSSVisLocal für die Polynomgrade 1 bis 5 auf einem einzigen Verfeinerungslevel dargestellt.

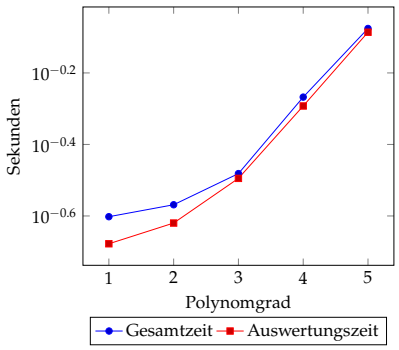


Abbildung 7.5: Gesamtzeit der Auswertung der Basisfunktionen in CraSSVisLocal mit verschiedenen Polynomgraden auf einem einzigen Verfeinerungslevel

Der Vergleich der Kurven zeigt, dass die Auswertung der Basisfunktionen die zeitaufwändigste Operation ist. Im Gegensatz zu CraSSVisGlobal steigt die Auswertungszeit bei CraSSVisLocal aufgrund der exponentiell wachsenden Länge der Lagrange-Polynome nun selbst exponentiell an. Die Erzeugung der Polynome selbst benötigt nur eine minimale Zeit.

7.6 Rechenzeit- und Speicherbedarfsanalyse der Gitteroperationen

Die Gitteroperationen, vor allem die Gitterverfeinerung, benötigen die meiste Rechenzeit. In diesem Abschnitt wird genauer auf die Rechenzeit für die Gitteroperationen und deren Speicherbedarf eingegangen.

Für jedes Verfeinerungslevel n werden pro Dreieck vier neue Dreiecke erzeugt. Daraus folgt ein exponentieller Anstieg der Anzahl der Dreiecke mit 4^n. Abbildung 7.6 A zeigt ein Gitter mit 8 Dreiecken (rote Kanten) auf Level 1. Auf dem Verfeinerungslevel 2 wird das Gitter durch $8 \cdot 4^2 = 128$ Dreiecke (orangene Kanten) gebildet (etc...). Für die Speicherung eines jeden Dreiecks sind drei im Arbeitsspeicher liegende *double*-Koordinaten und drei Indizes mit jeweils 8 Byte Länge nötig, also ergibt sich ein Speicherbedarf von 96 Byte pro Dreieck. Auf Verfeinerungslevel 10 ergibt sich ein Speicherbedarf von 96 MB.

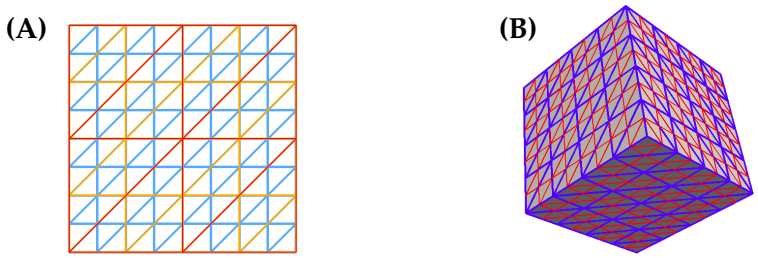


Abbildung 7.6: Verfeinerungen auf verschiedenen Leveln und Dimensionen
(A) Zwei 2D Dreiecke: Level 1 (rot); Level 2 (orange); Level 3 (blau)
(B) Sechs 3D Tetraeder: Level 2 (blau); Level 3 (rot)

In 3D ist der Speicherbedarf um den Faktor $\frac{2}{3}2^{x+2}$ höher als der in 2D. Für die Speicherung eines Tetraeders werden bei 4 Koordinaten und 4 Indizes 128 Byte Arbeitsspeicher benötigt. Der Speicherbedarf auf Level 10 berechnet sich auf ca. 129 GB. In Abbildung 7.7 ist der Speicherbedarf für Dreiecke und Tetraeder in Abhängigkeit vom Verfeinerungslevel dargestellt.

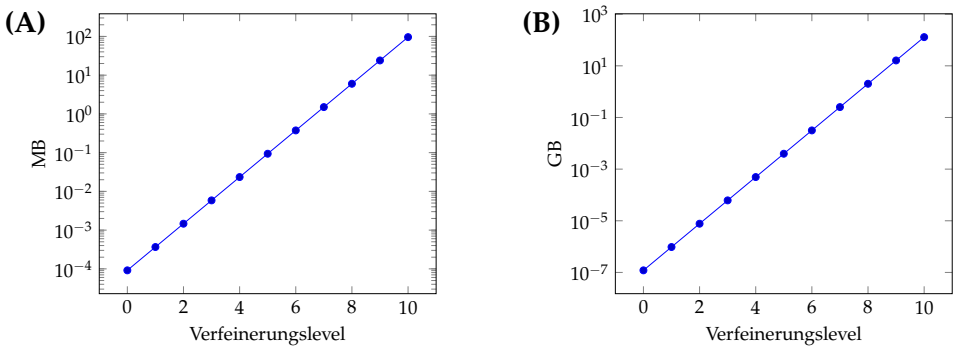


Abbildung 7.7: Speicherbedarf pro Verfeinerungslevel beginnend mit einem Primitiv
(A) Speicherbedarf für Dreiecke **(B)** Speicherbedarf für Tetraeder

Mit jedem Verfeinerungslevel steigt nicht nur der Speicherbedarf sondern auch die für die Verfeinerung benötigte Zeit. Der Zeitaufwand für die Verfeinerung in Abhängigkeit vom Verfeinerungslevel ist für Dreiecke in Abbildung 7.8, für Tetraeder in Abbildung 7.9 dargestellt. Wegen des hohen Speicherbedarfs konnte die Auswertung bei Tetraedern nur bis Level 8 durchgeführt werden.

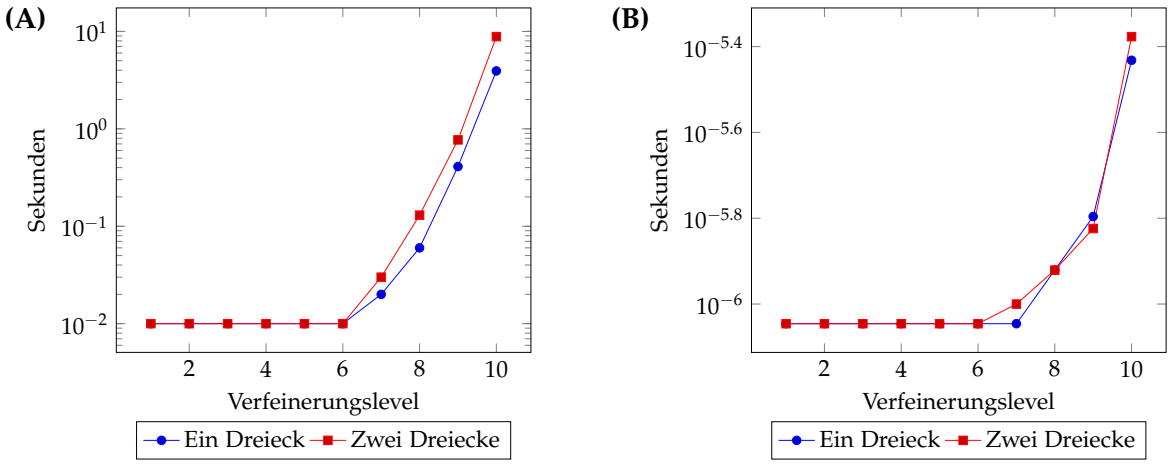


Abbildung 7.8: Zeitaufwand in Abhängigkeit von den Verfeinerungsleveln für Dreiecke
(A) Gesamtzeit **(B)** Zeit pro Dreieck

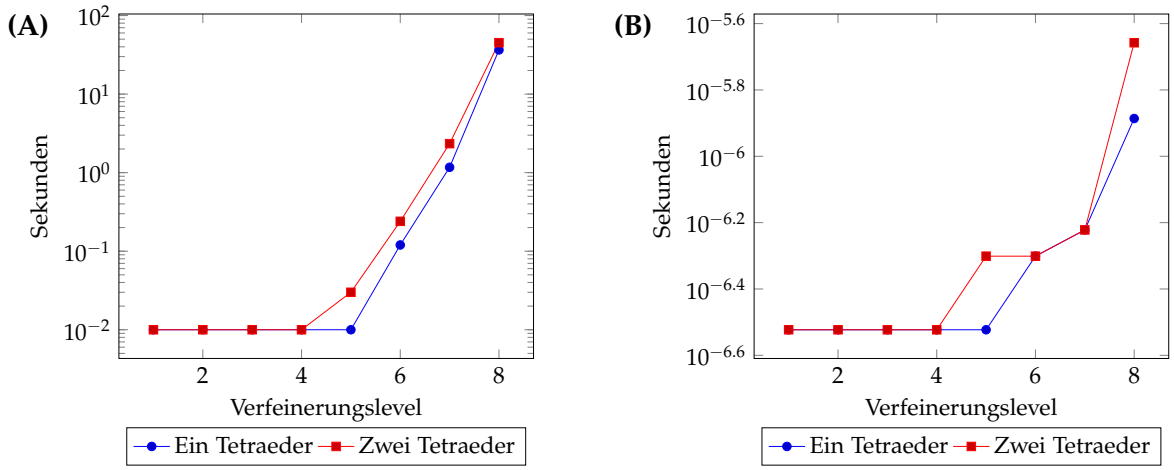


Abbildung 7.9: Zeitaufwand in Abhängigkeit von den Verfeinerungsleveln für Tetraeder
(A) Gesamtzeit **(B)** Zeit pro Tetraeder

Die Rechenzeit für den Gitterschnitt ist vernachlässigbar gering. Sie beträgt selbst bei großen Gittern nicht mehr als 1 bis 2 Sekunden.

Die Zunahme der Rechenzeit und des Speicherbedarfs zeigt bei Gitteroperationen einen exponentiellen Verlauf. Ab einem höheren Level ist daher entweder die Wartezeit des Benutzers unvertretbar oder die Kapazität des Rechners erschöpft. Eine adäquate Verfeinerung relevanter Teilgitter durch mehrmaliges Verwenden der Plug-ins erscheint als ein guter Kompromiss, um möglichst gleichwertige Ergebnisse auf dem gesamten Gitter zu erzielen.

8 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde ein Plug-in – CraSSVis – für ParaView entwickelt, das eine auswertungsunabhängige Visualisierung des Lösungsraums der „Partition of Unity“-Methode (PUM) auf einem Dreiecks- oder Tetraedergitter erlaubt. CraSSVis ermöglicht eine Weiterverwendung der Koeffizienten und Basisfunktionen, die den Lösungsraum der PUM bestimmen. Dadurch ergibt sich die Bedingung der Möglichkeit für hoch aufgelöste Abbildungen des Lösungsraums, die PUM nicht besitzt.

CraSSVis wurde als Filter-Plug-in entwickelt. Sein Entwurfsmuster ist durch ParaView vorgegeben, dessen Drei-Komponentenarchitektur sich auf die Kompilation der Plug-ins auswirkt. Es integriert die Koeffizienten und Basisfunktionen, die den Lösungsraum der PUM bestimmen. CraSSVis ermöglicht eine Darstellung des Lösungsraums auf dem ausgeschnittenen Gitter. An allen von CraSSVis erzeugten Punkten ist eine Auswertung des Lösungsraums möglich. Eine Approximation an die Genauigkeit des von PUM erzeugten Lösungsraums und eine entsprechend genaue Darstellung ist möglich. CraSSVis besitzt eine benutzerfreundliche Verwaltung der Basisfunktionen und Koeffizienten. Zusätzliche Basisfunktionen werden über dynamische Bibliotheken eingebunden. CraSSVisLocal verwendet für die Koeffizienten in der Eingabedatei ein CellData-Array. Durch eine erweiterte Darstellung in der Eingabedatei und eine anschließende Konvertierung in ein CellData-Array ist eine sichere Erstellung der Koeffizientenabfolge möglich. Bei der Rechenzeit sind Verbesserungen möglich. Der notwendige Speicherbedarf ist durch die interne Speicherung vorgegeben, kann also nicht verändert werden. Bis zu mittleren Verfeinerungslevel ergibt sich eine gute Performanz.

Die Performanz könnte durch die Verwendung der Parallelität bei Gitteroperationen verbessert werden. Weil jedes Primitiv unabhängig von allen anderen verfeinert werden kann, könnte das Verfahren zur Gitterverfeinerung stark beschleunigt und die Rechenzeit für jeden hinzukommenden Prozess nahezu halbiert werden. Der Einsatz eines optimierten Algorithmus' könnte bei der Tetraederverfeinerung bessere Ergebnisse liefern.

Beide Formen von CraSSVis bieten eine Grundfunktionalität zur Darstellung des Lösungsraums der PUM. Diese Funktionalität könnte mit einem adaptiven Auswertungsverfahren oder einer allgemeineren Galerkin-Lösung mit hierarchischen Ansatzfunktionen erweitert werden. Die Verwendung von Rechtecken oder Quadern und der damit verbundenen Anpassung der Basisfunktionen könnte die Anzahl der unterstützten Primitive erweitern. Dadurch würden die Möglichkeiten, die PUM bietet, besser ausgenützt. Eine Erweiterung auf „vtkPolyData“ als Eingabeformat für die Plug-ins würde eine bessere nachträgliche Filterung ermöglichen.

CraSSVis bietet Nutzern heute schon die Möglichkeit, den Lösungsraum der PUM hoch aufgelöst darzustellen und kann für Erweiterungen bei zukünftigen Projekten dienen.

A Anhang

A.1 Benutzerleitfaden zu CraSSVisGlobal und CraSSVisLocal

Der Benutzerleitfaden ist für jedes der beiden Plug-ins in ParaView in der „Hilfe“ zu finden. Einzig und allein die Formatierung ist unterschiedlich. Die Leitfäden enthalten in aller Kürze die Beschreibung der Funktionen der Oberfläche und die nötigen Daten, die für die Anwendung des Plug-ins nötig sind. Die beiden Leitfäden werden folgend beschrieben.

Hilfe zu CraSSVisGlobal

Allgemeine Beschreibung

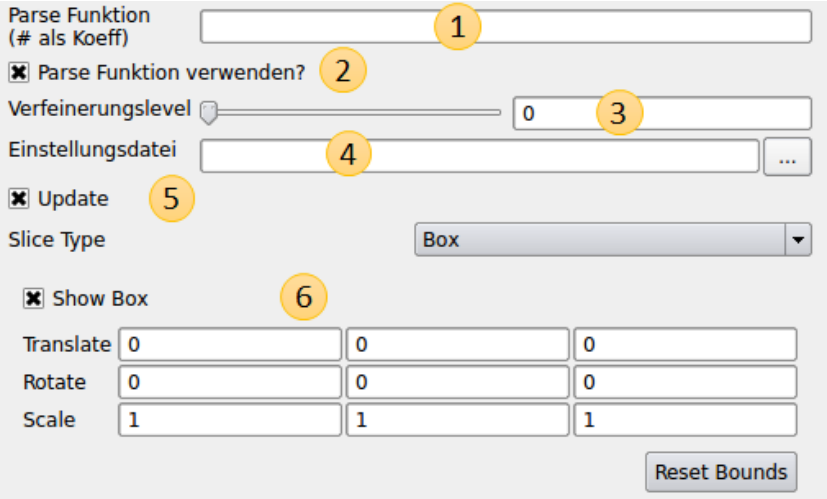
Dieses Filter-Plug-in dient der Auswertung von globalen expliziten Funktionen auf Basis der Gitterpunkte:

$$f = \sum_{i=0} g_i c_i$$

g_i : globale Basisfunktion und c_i : Koeffizienten

Aus einem Dreiecks- oder Tetraedergitter kann ein Bereich ausgeschnitten und verfeinert werden. Anschließend wird eine textuell eingegebene mathematische globale explizite Funktion an den Gitterpunkten ausgewertet. Zusätzlich können Koeffizienten (durch #-Zeichen repräsentiert) aus einer Datei geladen werden und automatisch in die Funktion eingesetzt werden. Der Funktionsparser, der die textuelle Funktion auswertet, kann durch dynamische Bibliotheken erweitert werden.

Funktion und Beschreibung der Filterelemente



1. **Parse Funktion (#-Zeichen als Koeff)**
   Dieses Textfeld dient der Eingabe der expliziten globalen Funktion. Koeffizienten können hier entweder direkt oder indirekt durch ein #-Zeichen verwendet werden. Die #-Zeichen werden anschließend durch die Koeffizienten aus einer Datei ersetzt. Ist die Anzahl der #-Zeichen größer als die der Koeffizienten in der Datei, werden die #-Zeichen zu Einsen ausgewertet. Die Variablen der Punktkoordinaten sind: x, y, z. Beispiel: #*cos(x)+#*cos(y)+#*cos(z)
2. **Parse Funktion verwenden?**
   Hier kann entschieden werden, ob die Funktion aus 1. ausgewertet wird, oder eine Funktion, die in einer Datei spezifiziert ist. Ist das Häkchen gesetzt, wird 1. verwendet.
3. **Verfeinerungslevel**
   Über den Schieberegeler, oder alternativ über das Textfeld kann das Verfeinerungslevel des Gitters festgelegt werden. Mögliche Level sind 0 bis 10. Hier ist vor allem bei einem 3D-Gitter auf den hohen Arbeitsspeicherbedarf zu achten.
4. **Einstellungsdatei**
   Über den Dateiauswahldialog kann der komplette Dateipfad zur Einstellungs-XML-datei ausgewählt werden. Der Aufbau dieser Datei wird weiter unten erklärt.
5. **Update**
   Diese Checkbox dient nur der Aktivierung des „Apply“-Knopfs von ParaView. Eine mögliche Änderung in der Einstellungsdatei kann ParaView somit manuell mitgeteilt werden.
6. **Bounding Box**
   Mit Hilfe der Maus oder wahlweise den Textfeldern kann der Benutzer den Bereich des Gitters zur Verfeinerung und Auswertung definieren.

Definitionen der benötigen Dateien

Einstellungsdatei

In der Einstellungsdatei werden die Dateipfade zur Koeffizientendatei, der globalen Funktionsdatei und den dynamischen Bibliotheken angegeben. Jede dieser Angaben sind optional und können beliebig kombiniert werden. Eine vollständige XML sieht folgendermaßen aus:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <CraSSVis>
3   <FolderPath></FolderPath>
4   <CoefficientsMatlabPath></CoefficientsMatlabPath>
5   <BasesFunctionXMLPath></BasesFunctionXMLPath>
6   <FunctionLib>
7     <Path></Path>
8   </FunctionLib>
9 </CraSSVis>
```

- **FolderPath:** Wenn ein Pfad zu einem Ordner angegeben wird, werden zusätzlich beschriebene Dateien in diesem Ordner, andernfalls im Ordner der Einstellungs-XML gesucht. Gänzlich davon abweichende Dateipfade werden unverändert beibehalten.
  Beispiel: /home/CraSSVis
- **CoefficientsMatlabPath:** Kompletter Dateipfad zur der Koeffizientendatei. Beispiel: /home/CraSSVis/Koeffizienten.m
- **BasesFunctionXMLPath:** Vollständiger Dateipfad zu der globalen Funktionsdatei. Beispiel: /home/CraSSVis/Funktion.xml
- **FunctionLib - Path:** Vollständiger Dateipfad zu den dynamischen Bibliotheken, die den Funktionsparser erweitern. Beispiel: /home/CraSSVis/libParser.so

Koeffizientendatei

In dieser Datei werden die Koeffizienten aufgelistet. Die Dateiendung muss hier 'm' lauten. Jeder Koeffizient wird in einer extra Zeile im englischen Zahlenformat (z. B. 2.23) definiert. In Anlehnung an das Matlab-Dateiformat sieht eine Beispieldatei folgendermaßen aus:

```
1 var = [
2   2.0
3   2.12
4   1.0
5 ]
```

Globale Funktionsdatei

In dieser Datei werden die Basisfunktionen der globalen expliziten Funktion definiert. Jeder der Basisfunktionen werden dabei addiert. Sie können entweder mit einem #-Zeichen einen automatisch geladenen Koeffizienten erhalten, oder er wird explizit mit in die Datei geschrieben. Eine Beispieldatei sieht folgendermaßen aus:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <BasesFunctions>
3   <Function>cos(x)</Function>
4   <Function>sin(100*y)</Function>
5   <Function>tan(\#*z)</Function>
6 </BasesFunctions>
```

- **Function:** Eine Basisfunktion in beliebiger Länge.

Hilfe zu CraSSVisLocal

Allgemeine Beschreibung

Dieses Filter-Plug-in dient der Auswertung von Basisfunktionen mit lokalem Träger auf Basis der Gitterpunkte:

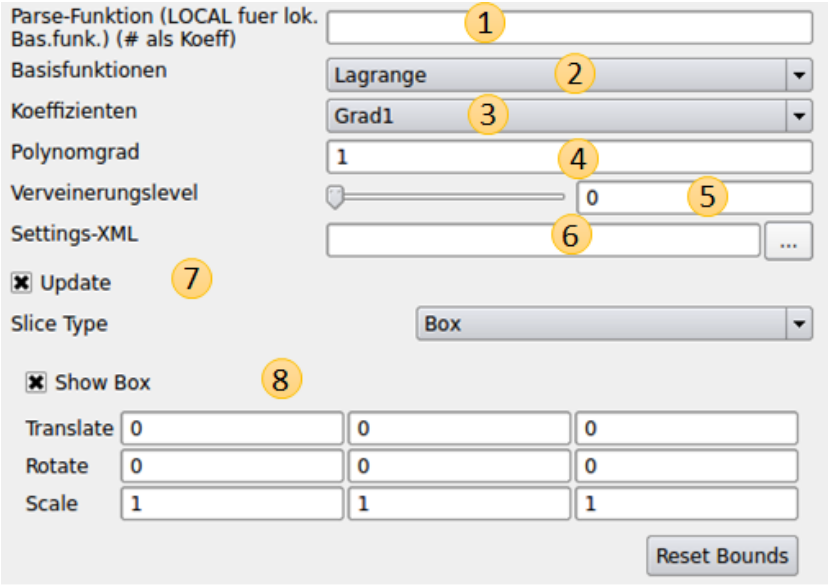
$$u = \sum_{i=0} g_i c_i \quad \text{oder} \quad u = \sum_{i=0} \varphi_i \left(\sum_{j=0} c_{ij} \eta_{ij} \right)$$

g_i : Lagrange-Polynome, c_i : Koeffizienten, η_{ij} : Enrichments und

φ_i : lokale Basisfunktionen

Aus einem Dreiecks- oder Tetraedergitter kann ein Bereich ausgeschnitten und verfeinert werden. Die lokalen Basisfunktionen bilden entweder die Lagrange-Funktionen oder komplexe Basisfunktionen ohne explizite Darstellung, die durch eine dynamische Bibliothek eingebunden werden können. Diese Funktionen können mit beliebigem Polynomgrad ausgewertet werden. Die Koeffizienten für die Basisfunktionen werden als CellData-Array aus der vtu-Datei geladen. Anschließend kann das Ergebnis der lokalen Basisfunktionen mit einer textuell eingegebenen mathematischen Funktion verändert werden. Zusätzlich können Koeffizienten (durch #-Zeichen repräsentiert) aus einer Datei geladen werden und automatisch in die textuelle Funktion eingesetzt werden. Der Funktionsparser, der die textuelle Funktion auswertet, kann durch dynamische Bibliotheken erweitert werden.

Funktion und Beschreibung der Filterelemente



1. **Parse Funktion (LOCAL für lok.Bas.funk.)(#-Zeichen als Koeff)**
   Dieses Textfeld dient der Eingabe der expliziten globalen Funktion zur Verrechnung des Ergebnisses der Auswertung der lokalen Basisfunktionen. Für das Ergebnis der lokalen Basisfunktionen wird der Platzhalter 'LOCAL' verwendet. Koeffizienten können hier entweder direkt oder indirekt durch ein #-Zeichen verwendet werden. Die #-Zeichen werden anschließend durch die Koeffizienten aus einer Datei ersetzt. Ist die Anzahl der #-Zeichen größer als die der Koeffizienten in der Datei, werden die #-Zeichen zu Einsen ausgewertet. Die Variablen der Punktkoordinaten sind: x, y, z. Beispiel: LOCAL+#*cos(x)+#*cos(y)+#*cos(z)
2. **Basisfunktionen**
   Zur Auswahl der verwendeten Basisfunktionen stehen nur die Lagrage-Funktionen zur Verfügung. Alternativ können komplexe Basisfunktionen ohne explizite Darstellung aus einer dynamischen Bibliothek verwendet werden.
3. **Koeffizienten**
   Der Benutzer kann die gewünschten Koeffizienten aus der vtu-Datei auswählen. Es werden nur Koeffizienten angezeigt, die als CellData-Array in der Datei vorhanden sind.
4. **Polynomgrad**
   Die Auswahl des gewünschten Polynomgrades der lokalen Basisfunktionen muss sich nach der Anzahl der verfügbaren Koeffizienten in der vtu-Datei richten.
5. **Verfeinerungslevel**
   Über den Schieberegeler, oder alternativ über das Textfeld kann das Verfeinerungslevel des Gitters festgelegt werden. Mögliche Level sind 0 bis 10. Hier ist vor allem bei einem 3D-Gitter auf den hohen Arbeitsspeicherbedarf zu achten.

6. **Einstellungsdatei**
   Über den Dateiauswahldialog kann der komplette Dateipfad zur Einstellungs-XML-datei ausgewählt werden. Der Aufbau dieser Datei wird weiter unten erklärt.
7. **Update**
   Diese Checkbox dient nur der Aktivierung des „Apply“-Knopfs von ParaView. Eine mögliche Änderung in der Einstellungsdatei kann ParaView somit manuell mitgeteilt werden.
8. **Bounding Box**
   Mit Hilfe der Maus, oder wahlweise den Textfeldern, kann der Benutzer den Bereich des Gitters zur Verfeinerung und Auswertung definieren. Hierbei ist zu beachten, dass nur Dreiecke oder Tetraeder, die vollständig in der Bounding Box enthalten sind, für die Auswertung der lokalen Basisfunktionen herangezogen werden können.

Definitionen der benötigten Dateien

Einstellungsdatei

In der Einstellungsdatei werden die Dateipfade zur Koeffizientendatei, der globalen Funktionsdatei und den dynamischen Bibliotheken angegeben. Jede dieser Angaben sind optimal und können beliebig kombiniert werden. Eine vollständige XML sieht volgendermaßen aus:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <CraSSVis>
3   <FolderPath></FolderPath>
4   <CoefficientsMatlabPath></CoefficientsMatlabPath>
5   <LocalFunctionLibPath></LocalFunctionLibPath>
6   <FunctionLib>
7     <Path></Path>
8   </FunctionLib>
9 </CraSSVis>
```

- **FolderPath:** Wenn ein Pfad zu einem Ordner angegeben wird, werden zusätzlich beschriebene Dateien in diesem Ordner, andernfalls im Ordner der Einstellungs-XML gesucht. Gänzlich davon abweichende Dateipfade werden unverändert beibehalten.
  Beispiel: /home/CraSSVis
- **CoefficientsMatlabPath:** Kompletter Dateipfad zur der Koeffizientendatei. Beispiel: /home/CraSSVis/Koeffizienten.m
- **BasesFunctionXMLPath:** Vollständiger Dateipfad zu der globalen Funktionsdatei. Beispiel: /home/CraSSVis/Funktion.xml
- **LocalFunctionLibPath:** Kompletter Dateipfad zu der dynamischen Bibliothek, die die Lagrange-Funktionen ersetzt. Als Beispiel: /home/CraSSVis/libFunktion.so
- **FunctionLib - Path:** Vollständiger Dateipfad zu den dynamischen Bibliotheken, die den Funktionsparser erweitern. Beispiel: /home/CraSSVis/libParser.so

Koeffizientendatei

In dieser Datei werden die Koeffizienten aufgelistet. Die Dateiendung muss hier 'm' lauten. Jeder Koeffizient wird in einer extra Zeile im englischen Zahlenformat (z. B. 2.23) definiert. In Anlehnung an das Matlab-Dateiformat sieht eine Beispieldatei folgendermaßen aus:

```
1  var = [
2    2.0
3    2.12
4    1.0
5  ]
```

A.2 muParser

Der Parser besitzt eine Reihe von vordefinierten Funktionstermen (siehe Tabelle 1), eine Menge von Operatoren (siehe Tabelle 2, S. 63) und ein paar konstante Werte (siehe Tabelle 3, S. 63).

Tabelle 1: Vorhandene Funktionen im muParser

Name	**Var.**	**Beschreibung**
sin	1	Sinus-Funktion
cos	1	Kosinus-Funktion
tan	1	Tangens-Funktion
asin	1	Arkussinus-Funktion
acos	1	Arkuskosinus-Funktion
atan	1	Arkustangens-Funktion
atan2	2	Arkustangens2-Funktion
sinh	1	Sinus Hyperbolicus-Funktion
cosh	1	Kosinus Hyperbolicus-Funktion
tanh	1	Tangens Hyperbolicus-Funktion
asinh	1	Arkussinus Hyperbolicus-Funktion
acosh	1	Arkuskosinus Hyperbolicus-Funktion
atanh	1	Arkustangens Hyperbolicus-Funktion
log2	1	Logarithmus zur Basis 2
log10	1	Logarithmus zur Basis 10
log	1	Logarithmus zur Basis 10
ln	1	Natürlicher Logarithmus
exp	1	Exponentialfunktion (e^x)
sqrt	1	Wurzel-Funktion
sign	1	Vorzeichenfunktion (-1 falls $x < 0$; 1 falls $x > 0$)
rint	1	Ganzzahlig runden
abs	1	Absolutbetrag
min	n	Minimum aller Variablen
max	n	Maximum aller Variablen
sum	n	Summe aller Variablen
avg	n	Durchschnitt aller Variablen

Tabelle 2: Vorhandene Operatoren im muParser

Operator	Priorität	Bedeutung
=	-1	Zuweisung
&&	1	Logisches Und
||	2	Logisches Oder
<=	4	Kleiner oder gleich
>=	4	Größer oder gleich
!=	4	Nicht gleich
==	4	Gleich
>	4	Größer als
<	4	Kleiner als
+	5	Addition
-	5	Subtraktion
*	6	Multiplikation
/	6	Division
^	7	Potenz
?:	-1	If-Then-Else (C++ Syntax)

Tabelle 3: Vorhandene Konstanten im muParser

Name	Wert	Beschreibung
_pi	3.14159265358979323846264 3	π
_e	2.71828182845904523536028 7	e

Literaturverzeichnis

[Ahl03] J. Ahlmann. Qualitäts-Metriken und Optimierung von Tetraeder-Netzen, 2003. (Zitiert auf Seite 6)

[Ber] I. Berg. muParser. URL http://muparser.beltoforion.de/. (Zitiert auf Seite 21)

[DR08] W. Dahmen, A. Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. 2 Auflage, 2008. (Zitiert auf Seite 3)

[EU] EU. EU ISO 9241-10. URL http://www.interactive-quality.de/site/DE/int/pdf/ISO_9241-10.pdf. (Zitiert auf Seite 20)

[Goo] Google. googletest. URL https://code.google.com/p/googletest/. (Zitiert auf Seite 21)

[Kita] Kitware. CMake. URL http://www.cmake.org. (Zitiert auf Seite 31)

[Kitb] Kitware. ParaView. URL http://www.paraview.org/Wiki/ParaView/Plug-in_HowTo. (Zitiert auf Seite 14)

[Kitc] Kitware. ParaView. URL http://the-unofficial-paraview-developers-forum.34153.x6.nabble.com. (Zitiert auf Seite 15)

[Kitd] Kitware. ParaView. URL http://www.vtk.org/VTK/img/file-formats.pdf. (Zitiert auf Seite 16)

[Kite] Kitware. ParaView. URL http://www.vtk.org/doc/nightly/html/vtkCellType_8h.html. (Zitiert auf Seite 19)

[Kitf] Kitware. ParaView. URL http://www.paraview.org/Wiki/ParaView:Build_And_Install. (Zitiert auf Seite 31)

[Kitg] Kitware. Visualization Tool Kit (VTK). URL http://www.vtk.org/. (Zitiert auf Seite 14)

[Lei12] H. Leitte. Algorithmische Geometrie – 5. Triangulierung von Polygonen, 2012. URL http://www.iwr.uni-heidelberg.de/groups/CoVis/Teaching/AG_SS12/AG_5_PolygonTriangulation.pdf. (Zitiert auf Seite 4)

[LJ95] A. Liu, B. Joe. Quality Local Refinement Of Tetrahedral Meshes Based On 8-Subtetrahedron Subdivision. *SIAM J. Sci. Comput*, 16:1269–1291, 1995. (Zitiert auf Seite 6)

[LL07] J. Ludewig, H. Lichter. *Software Engineering – Grundlagen, Menschen, Prozesse, Technik*. 2007. (Zitiert auf Seite 20)

[Mat] MathWorks. Dreiecksqualitätsmessung. URL http://www.mathworks.de/de/help/pde/ug/pdetriq.html. (Zitiert auf Seite 5)

[Pro] Q. Project. Qt. URL http://qt-project.org. (Zitiert auf Seite 21)

[Sch08] M. A. Schweitzer. *Meshfree and Generalized Finite Element Methods*. Dissertation, Rheinischen Friedrich–Wilhelms–Universität Bonn, 2008. (Zitiert auf Seite 2)

[SM00] H. Schumann, W. Müller. *Visualisierung: Grundlagen Und Allgemeine Methoden*. Springer, 2000. (Zitiert auf Seite 15)

[Tho] L. Thomason. TinyXML. URL http://www.grinninglizard.com/tinyxml. (Zitiert auf Seite 21)

Alle URLs wurden zuletzt am 29. August 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschift