Diplomarbeit

Grafische Darstellung des Indexraumes von Anweisungen in Schleifenprogrammen

Autor:
Alexander Wüst

Aufgabenstellung:
Prof. Christian Lengauer, Ph.D.
Lehrstuhl für Programmierung
Universität Passau

Betreuer:
Martin Griebl

17. März 1997

UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Zusammenfassung

DisPo2 (*Dis*play system for *Po*lyhedra *2*) ist ein interaktives Tool für die grafische Ausgabe des Indexraumes verschachtelter Schleifen im Rahmen des LooPo-Projektes (*Loo*p parallelization in the *Po*lytope model) am Lehrstuhl Lengauer zur automatischen Schleifenparallelisierung auf Basis des Polytop-Modells.

Der folgende Text ist in vier Abschnitte gegliedert:

Abschnitt 1 stellt sowohl LooPo, als auch DisPo2 kurz vor.

Abschnitt 2 stellt die mathematischen Grundlagen der Grafik und des Polytop-Modells zur Verfügung und geht auf die Parallelisierung im Polytop-Modell ein.

Abschnitt 3 enthält eine detaillierte Betrachtung der Möglichkeiten, die DisPo2 bietet, und beschäftigt sich mit den Prinzipien ihrer Implementierung.

In Abschnitt 4 werden schließlich Vorteile und Einschränkungen von DisPo2 zusammengefaßt, sowie ein Ausblick auf mögliche Verbesserungen und Erweiterungen gegeben.

Danksagung
Mein Dank gilt dem gesamten LooPo-Team, für ergiebige Diskussionen, wertvolle Hinweise auf Fehler (und Kaffee und Kuchen) - besonders aber Herrn Martin Griebl, der immer Zeit und ein offenes Ohr für Probleme hatte, und Herrn Prof. Lengauer, für das Korrekturlesen dieser Arbeit.

Inhaltsverzeichnis

1 Das LooPo-Projekt und DisPo2

In den folgenden beiden Kapiteln werden sowohl LooPo, als auch DisPo2 kurz vorgestellt, um einen Überblick über die Funktionsweise und die Zielsetzungen dieses umfangreichen Softwarepaketes zu vermitteln. Bis zum jetzigen Zeitpunkt haben über ein Dutzend Studenten im Rahmen von Diplomarbeiten oder Praktika an der Entstehung von LooPo mitgewirkt (siehe [Loop96] für Details zu LooPo, und [Wüst95] für weitere Erläuterungen zu DisPo2).

1.1 Was ist LooPo ?

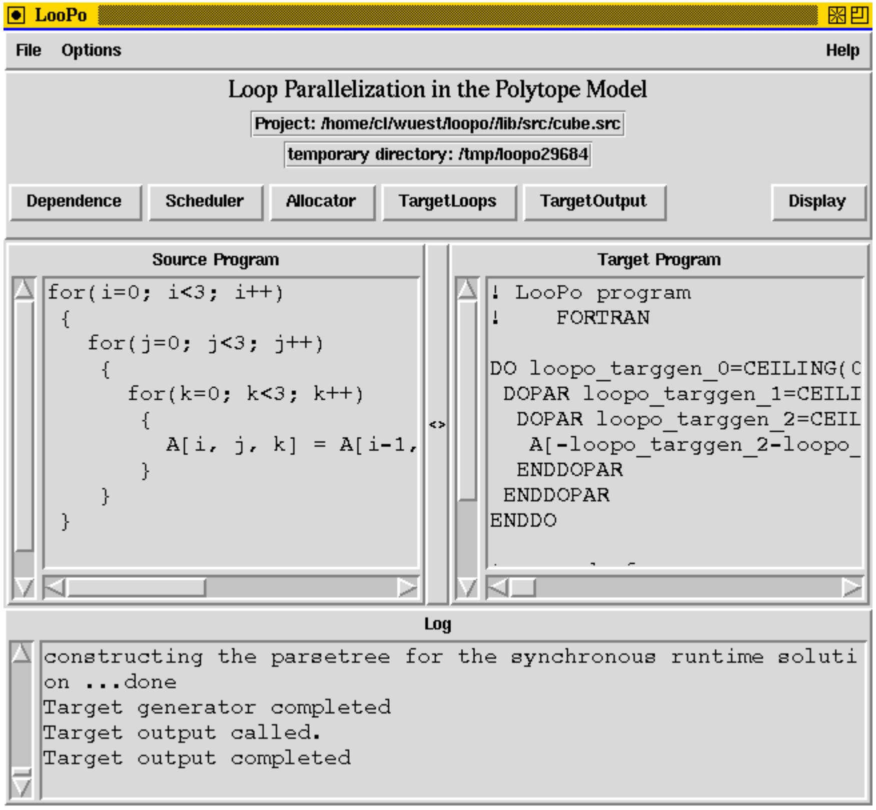


Abbildung 1: LooPo's Hauptfenster

Ziel des LooPo-Projektes ist die automatische Parallelisierung eines sequentiellen Quellprogramms mit Hilfe des Polytop-Modells (zu Einzelheiten zum Polytop-Modell siehe Kap. 2.2). Dieses Quellprogramm ist in einer von mehreren zur Auswahl stehenden Programmiersprachen einzugeben.
Zur Parallelisierung wird das Quellprogramm, das im wesentlichen aus verschachtelten Schleifen und Anweisungen besteht, als eine Menge von Polytopen modelliert. Diese sog. Quellpolytope werden unter Berücksichtigung der vorhandenen Abhängigkeiten, die durch Lese- und Schreibzugriffe verschiedener Anweisungen innerhalb des Schleifensatzes auf gemeinsame Variablen entstehen, in Zielpolytope transformiert. Ziel der Transformation ist es, unabhängige Anweisungen möglichst parallel auf verschiedenen Prozessoren auszuführen, um so eine Verkürzung der Ausführungszeit des ursprünglichen Quellprogramms zu erreichen. Zum Schluß er-

folgt die Erstellung eines parallelen Zielprogramms aus dem Zielpolytop.
Dieser Parallelisierungsvorgang ist durch eine Vielzahl von Parametern und Optionen steuerbar, wie z.B. der Auswahl verschiedener Methoden zur zeitlichen Anordnung der Anweisungen, oder zur Zuordnung der Anweisungen zu Prozessoren.
Abbildung 1 zeigt LooPo's Hauptfenster, von dem aus der Parallelisierungsvorgang gesteuert wird.

1.2 Was ist DisPo2 ?

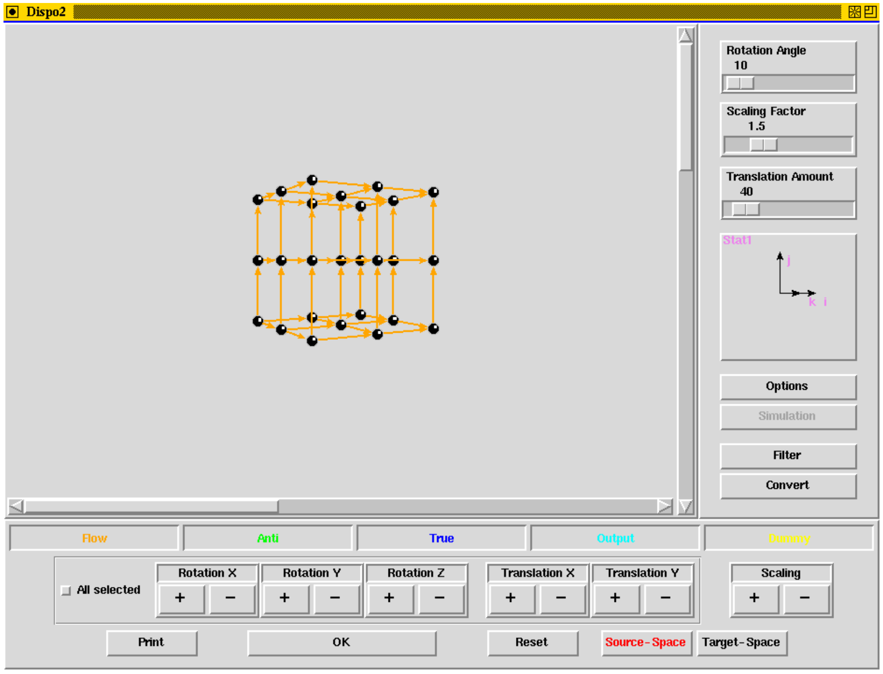


Abbildung 2: DisPo2's Hauptfenster

DisPo2 ist Teil des LooPo-Projektes und als Nachfolger von DisPo1 [Zwick93] zuständig für die grafische Ausgabe der Indexräume von Anweisungen in verschachtelten Schleifensätzen und der Abhängigkeiten zwischen den Anweisungen, sowie für deren interaktive Manipulation durch den Benutzer.
Neben den elementaren Manipulationsmöglichkeiten wie Vergrößerung, Verkleinerung, Rotation und Verschiebung stellt DisPo2 eine Vielzahl von weiteren Optionen und Betrachtungsvarianten zur Verfügung. Beispielsweise können die Indexräume transformiert und einzelne Anweisungen, Abhängigkeiten oder Schleifen bei der Darstellung ausgeblendet werden, oder es kann eine perspektivische Ansicht gewählt werden. Diese Menge von Möglichkeiten wird durch eine übersichtliche Oberfläche (siehe Abbildung 2) überschaubar gehalten, die leicht verständlich und einfach mit der Maus zu bedienen ist.
Anders als der Vorgänger DisPo1 basiert DisPo2 jedoch nicht mehr auf X-Bibliotheken, sondern auf Tcl/Tk [Oust94], was Unabhängigkeit vom jeweils gewählten Betriebssystem garantiert.

2 Theoretische Grundlagen

In diesem Kapitel werden zunächst einige mathematische Grundlagen der Grafikprogrammierung und zum Verständnis der Polyedertheorie vermittelt. Anschließend wird kurz das Verfahren zur Parallelisierung im Polytop-Modell vorgestellt.

2.1 Grundlagen der Grafik

Bei der Darstellung dreidimensionaler Objekte auf einem Bildschirm stößt man in der Regel auf zwei grundlegende Probleme. Zum einen kann der Computerbildschirm, bedingt durch seine Abmessungen, nur einen begrenzten Ausschnitt der realen Welt wiedergeben, und zum anderen ist der Bildschirm nur zweidimensional. Es fehlt also eine Dimension zur Darstellung.
Zur Lösung dieser Probleme geht man im Prinzip in drei Schritten vor (siehe Abbildung 3):

Schritt 1: Einschränkung der realen Welt auf einen anzuzeigenden Bereich (*sichtbarer Bereich*).
Schritt 2: Projektion der Objekte auf eine zweidimensionale Ebene (Reduktion der Dimension um 1).
Schritt 3: Transformation der Weltkoordinaten in Bildschirmkoordinaten.

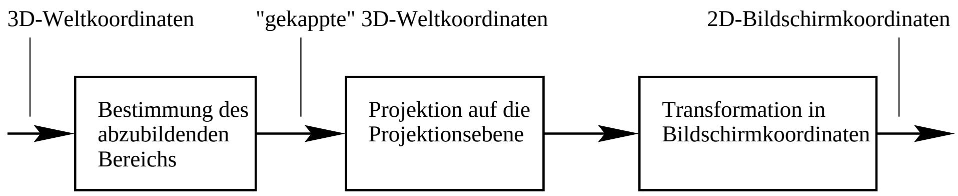


Abbildung 3: Prinzip der Darstellung

Der erste Schritt bereitet im Falle von DisPo2 keine Probleme, da vonvornherein klar ist, was zur Anzeige kommen soll, nämlich die berechneten Polytope.
Auch der dritte Schritt, die Transformation der projezierten Weltkoordinaten (dreidim.) in Bildschirmkoordinaten (zweidim.), kann in kanonischer Weise erfolgen, denn bei DisPo2 sind das Bildschirmkoordinatensystem und die XY-Ebene des Weltkoordinatensystems, die zugleich als Projektionsebene dient, identisch. Deswegen können die ersten beiden Koordinaten eines Punktes auf der Projektionsebene einfach als Bildschirmkoordinaten interpretiert werden und die dritte Koordinate, die ja aufgrund der Lage der Projektionsebene nach der Projektion immer gleich Null ist, bleibt unberücksichtigt. Unter- oder überschreiten die resultierenden Koordinaten die für Bildschirmkoordinaten zulässigen Werte, so liegen sie außerhalb des sichtbaren Fensters und sind deshalb nicht zu sehen (siehe Abbildung 4).

Eine Fülle von Möglichkeiten eröffnen sich jedoch für den zweiten Schritt, mit dem sich das folgende Kapitel eingehend befaßt.

2.1.1 Projektionen

Projektionen dienen dazu, die Dimensionalität von Objekten zu verringern, d.h. sie werden in der Computergrafik in der Regel dafür eingesetzt, dreidimensionale Objekte auf einem zweidimensionalen Bildschirm darzustellen.

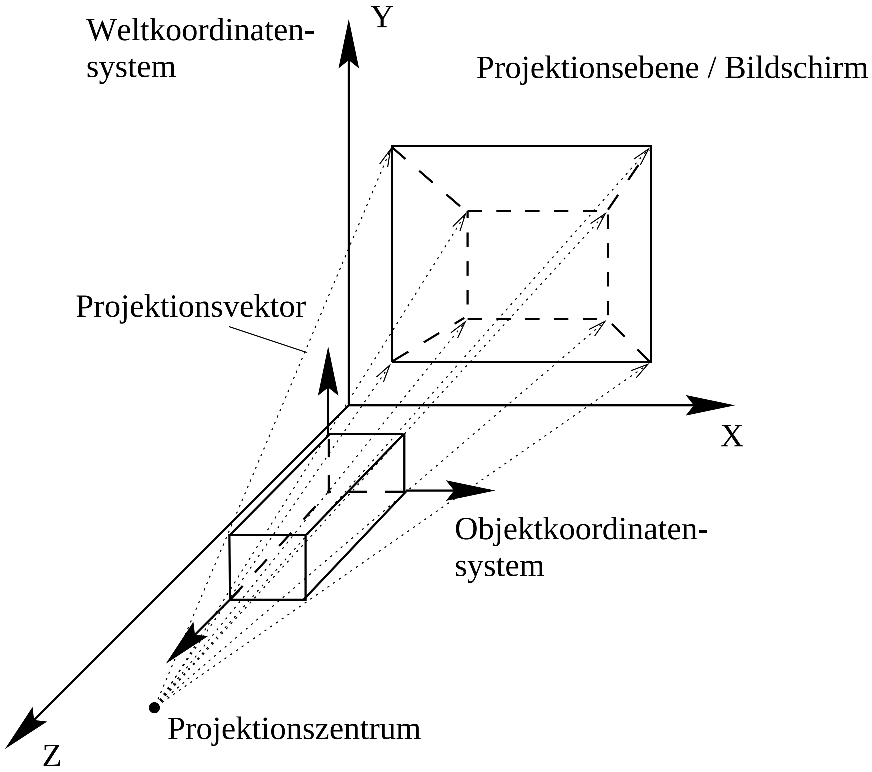


Abbildung 4: Prinzip der Projektion

Wenn in dieser Arbeit von Projektionen die Rede ist, so sind immer *planare geometrische Projektionen* gemeint, d.h. die Projektion erfolgt auf eine Ebene, die *Projektionsebene PE*, und die *Projektionsvektoren PV* sind gerade. Die Projektionsvektoren geben die Richtung der Projektion an und verlaufen vom *Projektionszentrum PZ*, das das Auge des Betrachters darstellt, durch den zu projezierenden Punkt. Der Schnittpunkt des Projektionsvektors mit der Projektionsebene gibt die Lage der Projektion des betreffenden Punktes an.
In der Kartographie existieren auch Projektionen, die nicht planar und/oder geometrisch sind, d.h. die Projektion erfolgt auf eine gekrümmte Oberfläche und/oder die Projektionsvektoren sind gekrümmt. Derartige Abbildungen sollen aber hier nicht betrachtet werden, sondern wir beschränken uns planare geometrische Projektionen. Diese können gemäß Abbildung 5 in parallele und perspektivische Projektionen aufgeteilt werden. Unterscheidungsmerkmal ist hierbei die Entfernung des Projektionszentrums von der Projektionsebene. Diese befindet sich bei den folgenden Ausführungen immer in der XY-Ebene des Weltkoordinatensystems, wenn nicht ausdrücklich auf eine andere Lage hingewiesen wird.

Perspektivische Projektionen

Bei den perspektivischen Projektionen hat *PZ* einen endlichen Abstand zu *PE*. Deswegen sind die Projektionsvektoren mit denen unterschiedliche Punkte eines Objektes projeziert werden nicht identisch, sondern jeder Punkt wird mit einem für seine Lage charakteristischen Projektionsvektor projeziert. Als Folge davon treffen sich die Projektionen paralleler Linien, die nicht parallel zu *PE* verlaufen, in einem sog. *Fluchtpunkt* (siehe Abbildung 6). Parallele Linien, die auch parallel zu

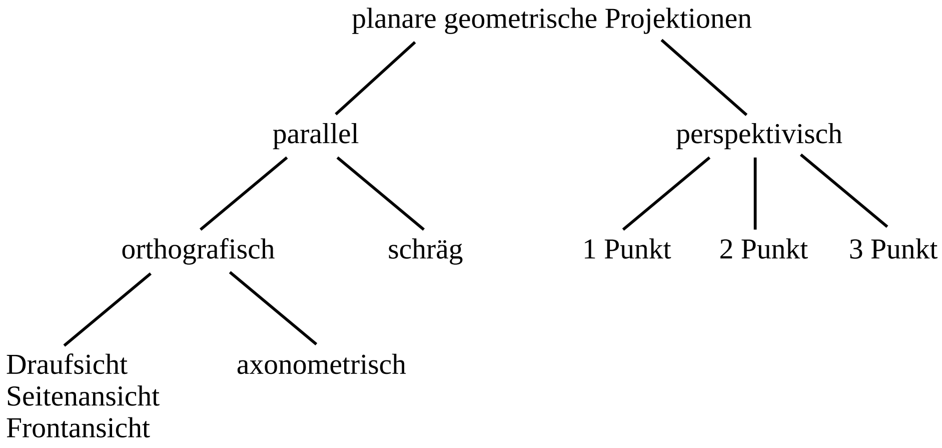


Abbildung 5: Hierarchie der Projektionen

PE verlaufen, bleiben dagegen auch nach der Projektion parallel zueinander und besitzen daher keinen Fluchtpunkt. Der Fluchtpunkt, in dem sich die Projektionen von Linien parallel zu einer Objektkoordinatenachse schneiden, heißt *Achsenfluchtpunkt* der betreffenden Koordinatenachse. Da bei einer Rotation des betrachteten Objektes im Weltkoordinatensystem das Objektkoordinatensystem jeweils mitgedreht wird, ist es möglich, daß *PE* eine, zwei oder drei Objektkoordinatenachsen schneidet. Je nachdem wie viele Objektkoordinatenachsen von *PE* geschnitten werden, existieren ein, zwei oder drei Achsenfluchtpunkte und man unterscheidet somit 1-, 2- und 3-Punkt perspektivische Projektionen.

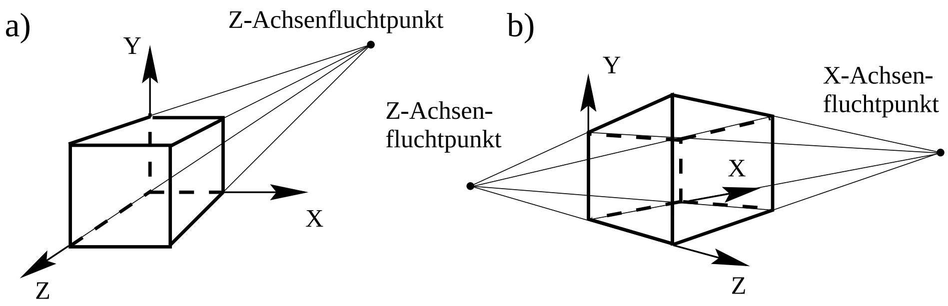


Abbildung 6: a) 1-Punkt, b) 2-Punkt perspektivische Projektion

Der Effekt, daß Objekte, die sich weiter entfernt von *PZ* befinden, kleiner erscheinen als Objekte in der Nähe von *PZ*, heißt *perspektivische Verkleinerung*. Dies entspricht dem visuellen Effekt, den das menschliche Auge produziert und resultiert daher in sehr realistischen Bildern.

Sei $P_p = (x_p, y_p, z_p)$ die Projektion eines Punktes $P = (x, y, z)$. Unter der Annahme, daß die Projektionsebene senkrecht zur Z-Achse bei $z = 0$ liegt und sich das Projektionszentrum bei $PZ = (0, 0, d)$ befindet (siehe Abbildung 7), ist z_p immer null und die restlichen Koordinaten des projezierten Punktes berechnen sich wie folgt:

$$\frac{y_p}{d} = \frac{y}{d - z} \quad ,$$

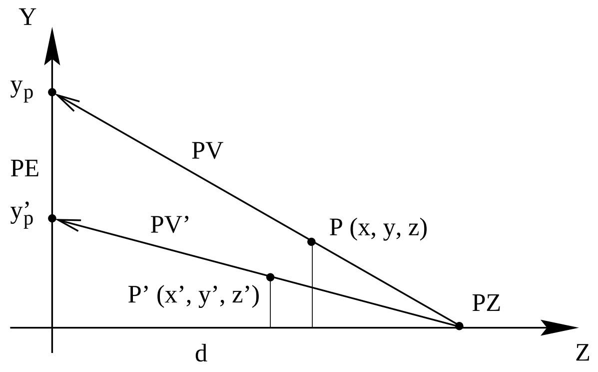


Abbildung 7: Berechnung der perspektivischen Projektion

$$y_p = \frac{d \cdot y}{d - z} = \frac{y}{1 - \frac{z}{d}} \quad .$$

Analog ergibt sich für x_p:

$$x_p = \frac{x}{1 - \frac{z}{d}} \quad .$$

Leider können i.a. wegen der perspektivischen Verkleinerung weder Längen noch Winkel aus der Projektion abgelesen werden.[1] Dieses Manko behebt eine Unterklasse der parallelen Projektionen, die im folgenden untersucht werden sollen. Dabei büßen die Darstellungen allerdings einen Teil ihrer Realitätsnähe ein.

Parallele Projektionen

Bei parallelen Projektionen hat PZ einen unendlichen Abstand zu PE und damit sind alle Projektionsvektoren parallel. Es genügt also, statt des Projektionszentrums den Projektionsvektor anzugeben, der nun für alle zu projezierenden Punkte identisch ist.
Da richtige perspektivische Verkleinerung bei parallelen Projektionen fehlt, erlauben sie das Abmessen von Längen aus der Projektion. Wie bei der perspektivischen Projektion bleiben Winkel jedoch nur auf Ebenen parallel zu PE erhalten.

Abhängig von der Länge von PE und PV zueinander unterscheidet man *orthografische* und *schräge* Projektionen.
Bei der orthografischen Projektion bilden PV und PE einen rechten Winkel. Man spricht auch von Front-, Seitenansicht und Draufsicht, falls die Projektionsebene einer orthografischen Projektion nur eine der Objektkoordinatenachsen schneidet und deswegen auch nur eine Seite des betrachteten Objektes abgebildet wird (d.h. PE ist senkrecht zu einer Objektkoordinatenachse). Eine orthografische Projektion, bei der die Projektionsebene mehrere Objektkoordinatenachsen schneidet und deswegen auch mehrere Seiten des betrachteten Objektes gleichzeitig abgebildet werden, heißt *axonometrische Projektion*.
Bei der schrägen Projektion liegt PE wieder senkrecht zu einer Objektkoordinatenachse, PE und PV bilden jedoch einen Winkel $\beta \neq 90°$. Diese Projektionsart vereint

[1]Winkel auf Ebenen parallel zu PE bleiben jedoch auch nach der Projektion erhalten.

Eigenschaften der Front-, Seitenansicht und der Draufsicht mit denen der axonometrischen Projektion. Einerseits ist PE senkrecht zu einer Objektkoordinatenachse, was Messungen von Längen und Winkeln an Flächen parallel zu PE erlaubt, andererseits werden aber mehrere Seiten des Objektes gleichzeitig abgebildet (siehe dazu Abbildung 8).

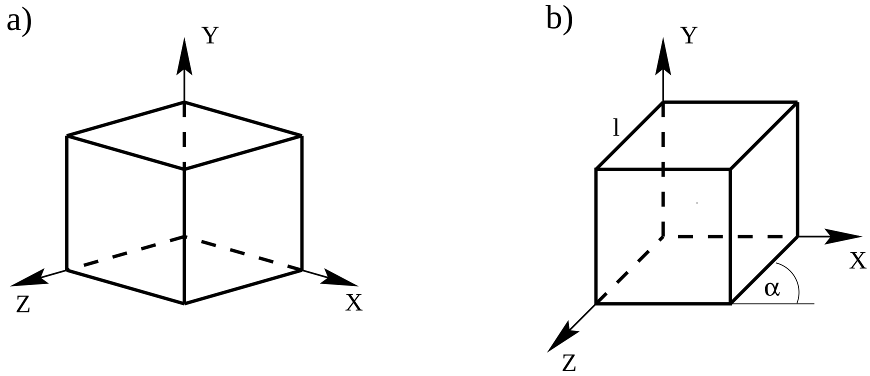


Abbildung 8: a) Orthogonale (hier axonometrische) Projektion,
b) schräge Projektion eines Einheitswürfels

Zur Berechnung der parallelen Projektion wird gemäß Abbildung 9 wieder angenommen, PE sei senkrecht zur Z-Achse, befinde sich bei $z = 0$ und $PV = (x_R, y_R, z_R)$. Die folgenden Ausführungen gelten sowohl für die orthografische, als auch für die schräge Projektion.

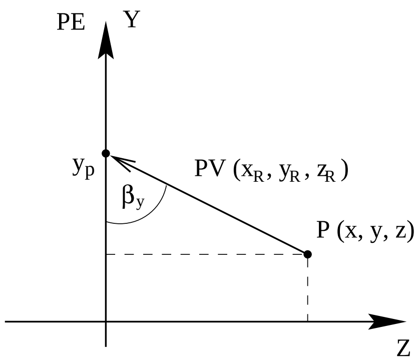


Abbildung 9: Berechnung der parallelen (hier schrägen) Projektion

Man erhält für die parallele Projektion $P_p(x_p, y_p, z_p)$ eines Punktes $P(x, y, z)$:

$$y_p = y + z \cdot \sin(90^\circ - \beta_y) = y - z \cdot \frac{y_R}{z_R} \quad ,$$

$$x_p = x + z \cdot \sin(90^\circ - \beta_x) = x - z \cdot \frac{x_R}{z_R} \quad .$$

z_p ist aufgrund der Lage von PE gleich 0.

Ähnlich wie bei perspektivischen Projektionen gibt es auch bei axonometrischen und schrägen Projektionen perspektivische Verkleinerung. Diese ist jedoch nicht abhängig vom Abstand zum Projektionszentrum (dieses existiert ja auch gar nicht, bzw. nur unendlich weit entfernt), sondern ist für jede Objektkoordinatenachse einheitlich.

Bei axonometrischen Projektionen ist der Verkleinerungsfaktor v abhängig von der Lage der Projektionsebene zum betrachteten Objekt. Bildet eine Achse des Objektkoordinatensystems den Winkel γ mit der Projektionsebene, so beträgt der Verkleinerungsfaktor v entlang dieser Achse $\cos\gamma$. D.h. um einen bestimmten Verkleinerungsfaktor $v, \ \ 0 \leq v \leq 1$, entlang einer Achse des Objektkoordinatensystems zu erreichen, muß diese um den Winkel $\arccos v$ gegenüber der Projektionsebene geneigt sein.

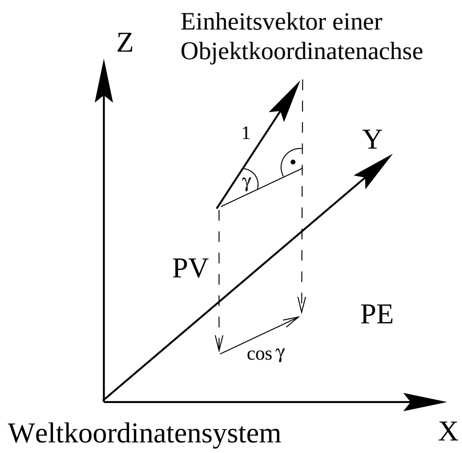


Abbildung 10: Perspektivische Verkleinerung bei axonometrischer Projektion

Abhängig von ihrer Lage sind deswegen alle drei Objektkoordinatenachsen einem (nicht notwendigerweise gleichen) Verkleinerungsfaktor unterworfen. In Abbildung 8a haben z.B. alle Kanten des projezierten Einheitswürfels eine Länge < 1.

Bei schrägen Projektionen ist der Verkleinerungsfaktor $v, \ \ 0 \leq v$ (hier ist sogar eine Vergrößerung möglich), abhängig von der Wahl von PV. Hier unterliegt jedoch nur die Objektkoordinatenachse senkrecht zur Projektionsebene einer Verkleinerung. In Abbildung 8b haben z.B. nur die Kanten des projezierten Einheitswürfels, die in Z-Richtung verlaufen, eine Länge $\neq 1$.
Abbildung 11 (aus [FDFH90]) zeigt, wie bei schrägen Projektionen PV zu wählen ist, um einen bestimmten Verkleinerungsfaktor v zu erreichen.

Die Projektionsebene ist die XY-Ebene des Weltkoordinatensystems, L die Länge der Projektion des Einheitsvektors der Z-Achse (und damit ist L gleich dem gewünschten Verkleinerungsfaktor v) und α der Winkel, den diese Projektion mit der X-Achse einschließt. Bei gegebenem v und α ergibt sich PV zu:

$$PV = P_p - P = (v \cdot \cos\alpha, v \cdot \sin\alpha, 0) - (0, 0, 1) = (v \cdot \cos\alpha, v \cdot \sin\alpha, -1).$$

Verallgemeinerung der Projektionsformeln

Bei der Berechnung der Projektionen sind sowohl bei der parallelen Projektion, als auch bei der perspektivischen Projektion einige Einschränkungen gemacht worden, die im folgenden aufgehoben werden sollen. So wurde z.B. PE immer senkrecht zur Z-Achse bei $z = 0$ angenommen und bei der perpektivischen Projektion lag PZ

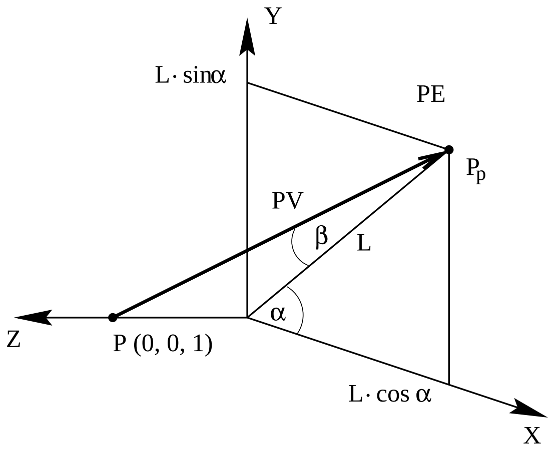


Abbildung 11: Perspektivische Verkleinerung bei schräger Projektion

immer auf der Z-Achse. In [FDFH90] wird ein Ansatz vorgestellt, der nicht nur diese Beschränkungen aufhebt, sondern auch die parallele und perspektivische Projektion in eine Formel integriert.
PE bleibt zwar immer noch senkrecht zur Z-Achse, befindet sich aber nun in einem bestimmten Abstand z_p vom Ursprung. PZ liegt im Abstand Q vom Referenzpunkt $(0, 0, z_p)$, und die Richtung zu PZ wird durch den normalisierten Richtungsvektor (d_x, d_y, d_z) angegeben (siehe Abbildung 12).

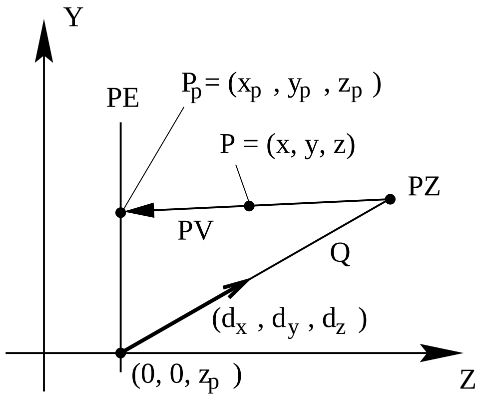


Abbildung 12: Vereinheitlichung der Projektionen

Die parallele Projektion ergibt sich nun als Spezialfall der perspektivischen Projektion, indem $Q = \infty$ gesetzt wird.

Die Koordinaten des projezierten Punktes P_p lassen sich anhand folgender Formeln berechnen.
P_p liegt auf einem Strahl ausgehend von PZ durch P, der wie folgt geschrieben werden kann

$$PZ + t(P - PZ) \ , \qquad t \geq 0.$$

Teilt man diese Formel für einen beliebigen Punkt $P' = (x', y', z')$ auf dem Strahl in die Gleichungen für die einzelnen Koordinaten auf, so erhält man mit $PZ = (0, 0, z_p) + Q(d_x, d_y, d_z)$

$$x' = Qd_x + t(x - Qd_x) \tag{1}$$
$$y' = Qd_y + t(y - Qd_y) \tag{2}$$
$$z' = (z_p + Qd_z) + t(z - (z_p + Qd_z)) \tag{3}$$

P_p liegt im Schnittpunkt des Strahls mit PE, deshalb wird in Gleichung (3) z' durch z_p ersetzt und nach t aufgelöst.

$$t = \frac{z_p - (z_p + Qd_z)}{z - (z_p + Qd_z)} \quad .$$

Nun ersetzen wir in den Gleichungen (1) und (2) den Wert für t und erhalten $x' = x_p$ und $y' = y_p$.

$$x_p = \frac{x - z\frac{d_x}{d_z} + z_p\frac{d_x}{d_z}}{\frac{z_p - z}{Qd_z} + 1} \quad ,$$
$$y_p = \frac{y - z\frac{d_y}{d_z} + z_p\frac{d_y}{d_z}}{\frac{z_p - z}{Qd_z} + 1} \quad .$$

Da z_p bekannt ist, sind somit alle Koordinaten von P_p berechnet.

Realisierung der Projektionen bei DisPo2

Mit DisPo2 können alle bisher vorgestellten Projektionen erzeugt werden, die Projektionsebene befindet sich jedoch immer senkrecht zur Z-Achse des Weltkoordinatensystems bei $z = 0$, ist also die XY-Ebene. Dies bedeutet aber keine Einschränkung, sondern lediglich eine Vereinfachung für den Benutzer, da durch Rotation des dargestellten Objektes trotzdem jede beliebige Seite abgebildet werden kann und keine komplizierte Angabe der Lage der Projektionsebene erforderlich ist.

Die Auswahl zwischen paralleler und perspektivischer Projektion geschieht im Options-Menü, wo auch für die parallele Projektion ein Projektionsvektor und die perspektivische Projektion ein Projektionszentrum angegeben werden können.
Für die Front-, Seitenansicht und Draufsicht, sowie für die axonometrische Projektion ist Parallelprojektion zu wählen und für den Projektionsvektor der Standardwert $(0, 0, -1)$. Die gewünschte Ansicht ist dann durch Rotation des betrachteten Objektes zu erzeugen. Für eine schräge Projektion ist der Projektionsvektor entsprechend zu verändern.
Um die perspektivische Projektion zu erreichen, ist das Feld „perspektivisch“ anzuklicken und ein Projektionszentrum anzugeben. 1-Punkt, 2-Punkt und 3-Punkt Projektionen werden dann durch Rotation des Objektes erzeugt.

2.1.2 Homogene Koordinaten

In der Computergrafik existieren drei verschiedene grundlegende Transformationen, denen man Punkte oder Punktmengen unterwerfen kann, nämlich Skalierung, Rotation um eine Koordinatenachse und Translation. Diese werden oft als Matrixmultiplikation realisiert, d.h. die Koordinaten eines transformierten Punktes ergeben sich aus der Multiplikation einer 3×3 Matrix, der sog. *Transformationsmatrix* M, mit den Koordinaten des zu transformierenden Punktes (es wird an dieser Stelle von einem dreidimensionalen Punkt ausgegangen, bei zweidimensionalen Punkten hat die Transformationsmatrix entsprechend nur eine Größe von 2×2).

$$P_{neu} = M \cdot P_{alt}$$

Z.B. bei der Rotation eines Punktes um die Z-Achse um den Winkel φ ist

$$M = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Leider läßt sich die Translation nicht so einfach als Matrixmultiplikation darstellen. Es wäre also notwendig, eine besondere Behandlung der Translation vorzusehen. Um dies zu verhindern geht man von der dreidimensionalen Darstellung eines dreidimensionalen Punktes über zu einer vierdimensionalen Darstellung. Der Punkt $(x, y, z)^T$ entspricht dann dem Punkt mit den *homogenen Koordinaten* $(x, y, z, 1)^T$ und damit kann nun auch die Translation als Matrixmultiplikation geschrieben werden.

$$\begin{pmatrix} x_{neu} \\ y_{neu} \\ z_{neu} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \Delta_x \\ 0 & 1 & 0 & \Delta_y \\ 0 & 0 & 1 & \Delta_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta_x \\ y + \Delta_y \\ z + \Delta_z \\ 1 \end{pmatrix}.$$

Die Realisierung der Transformationen mit homogenen Koordinaten und Matrixmultiplikation bietet zwei entscheidende Vorteile:

1. Es können alle drei Transformationen gleich behandelt werden.
2. Als Folge davon können mehrere Transformationen durch Multiplikation der Transformationsmatrizen zu einer Transformationsmatrix zusammengefaßt werden.

Diese Darstellungsweise ist bei DisPo2 jedoch *nicht* zur Anwendung gekommen, da der Hauptgrund für die Verwendung von homogenen Koordinaten, die Zeitersparnis durch Zusammenfassung mehrerer Transformationen, hier nicht zum tragen kommt. Der Benutzer kann immer nur eine Transformation nach der anderen mit der Maus auswählen, so daß keine Möglichkeit zur Zusammenfassung und damit auch keine Notwendigkeit zur Gleichbehandlung der Transformationen besteht. Die Einführung von homogenen Koordinaten würde in diesem Falle sogar eine Verlangsamung der Ausführungsgeschwindigkeit nach sich ziehen, da z.B. die Translation eines dreidimensionalen Punktes mit homogenen Koordinaten 16 Multiplikationen und 12 Additionen benötigt, obwohl die kanonische Methode mit drei Additionen auskommt.
Statt dessen wurden drei separate Prozeduren implementiert, die die Rotation, Skalierung und Translation eines Punktes bewirken.

2.2 Grundlagen des Polytop-Modells

In den folgenden Kapiteln wird versucht, die mathematischen Grundlagen zum Verständnis der Polyedertheorie zu vermitteln, wobei allerdings einige fundamentale Kenntnisse der linearen Algebra vorausgesetzt werden. Sind diese nicht vorhanden, können sie etwa in [Lip77] nachgelesen werden.
Desweiteren werden die Modellierung eines Schleifensatzes als Polytop, sowie einige Besonderheiten von WHILE-Schleifen betrachtet.

2.2.1 Mathematische Grundlagen

Dieses Kapitel dient dazu, dem Leser einen groben Überblick über die im Polytop-Modell benötigte Mathematik zu verschaffen. Detailliertere Informationen und teilweise auch Beweise zu den zitierten Sätzen können in [BaFl82] nachgelesen werden.

Polytope

Bevor der Terminus des Polytops eingeführt werden kann, müssen einige andere mathematische Grundbegriffe definiert werden.

Definition (Konvexe Menge): Sei $M \subseteq \mathbb{R}^n$.

$$M \text{ heißt } \textit{konvex} \Leftrightarrow (\forall p, q : p, q \in M : [p,q] = \{(1-\lambda)p + \lambda q \mid \lambda \in [0,1]\} \subseteq M).$$

D.h: Eine Menge M heißt genau dann *konvex*, wenn die Verbindungsstrecke zweier beliebiger Punkte der Menge ebenfalls Teil der Menge ist.

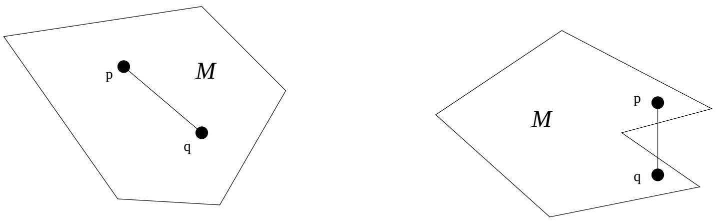


Abbildung 13: Konvexe Menge, nicht-konvexe Menge

Satz:
- Alle Vektorräume und damit der $\mathbb{R}^n$ sind konvex.
- Der endliche Durchschnitt konvexer Mengen ist konvex.

Definition (Lineare Ungleichung): Seien $\alpha_i, \chi_i, \beta \in \mathbb{R}$.
Eine *lineare Ungleichung* ist von der Form:

$$\alpha_1\chi_1 + \ldots + \alpha_n\chi_n \leq \beta \ , \qquad \text{oder}$$

$$a\,x \leq \beta \ ,$$

mit $a = (\alpha_1, \ldots, \alpha_n),\ x = (\chi_1, \ldots, \chi_n) \in \mathbb{R}^n$.

Genauso läßt sich mit Hilfe einer $m \times n$ - Matrix A ein System von n linearen Ungleichungen beschreiben:

$$A\,x \leq b \ ,$$

mit $A = \begin{pmatrix} \alpha_{11} & \dots & \alpha_{1n} \\ \vdots & & \vdots \\ \alpha_{m1} & \dots & \alpha_{mn} \end{pmatrix}$, $\alpha_{ij} \in \mathbb{R},\ x \in \mathbb{R}^n,\ b \in \mathbb{R}^m$.

Definition (Halbraum):

$$H = \{x \in \mathbb{R}^n \mid a\,x \leq \beta\} \text{ heißt } \textit{abgeschlossener Halbraum.}$$

$$H = \{x \in \mathbb{R}^n \mid a\,x < \beta\} \text{ heißt } \textit{offener Halbraum.}$$

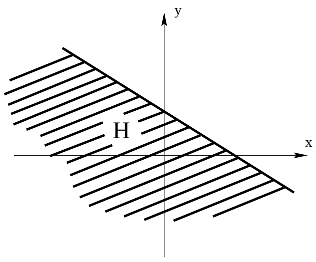


Abbildung 14: Abgeschlossener Halbraum

Satz:
Halbräume sind konvex.

Definiton (Hyperebene):

$$HP = \{x \in \mathbb{R}^n \mid a\,x = \beta\} \text{ heißt } \textit{Hyperebene.}$$

Satz:
Sei $HP \subseteq \mathbb{R}^n$ eine Hyperebene, dann gilt

$$dim(HP) = n - 1.$$

Definiton (Polyeder): Sei $P \in \mathbb{R}^n$, und $\mathcal{M}$ die Menge aller $m \times n$ - Matrizen.

$$P \text{ heißt } \textit{Polyeder} \Leftrightarrow (\exists A : A \in \mathcal{M} : (\exists b : b \in \mathbb{R}^m : P = \{x \in \mathbb{R}^n \mid A\,x \leq b\})).$$

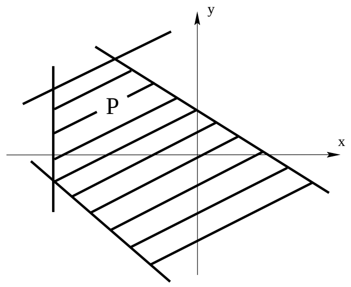
$$P \text{ heißt } \textit{beschränkt} \text{ oder } \textit{Polytop} \Leftrightarrow$$
$$(\forall x : x = (x_1, \dots, x_n) \in P : (\exists w : w \in \mathbb{R}_+ : (\forall j : 1 \leq j \leq n : -w \leq x_j \leq w))).$$

Definiton (Gültige Ungleichung): Eine lineare Ungleichung $a\,x \leq \beta$ heißt *gültig* für einen Polyeder P, wenn sie von allen Punkten in P erfüllt wird.

Definiton (Seite): Sei $a\,x \leq \beta$ gültig für P.

$$S = \{x \in P \mid a\,x = \beta\} \text{ heißt } \textit{Seite} \text{ von } P.$$

Bemerkung:
Alle Ungleichungen, die leere Seiten beschreiben, können weggelassen werden.



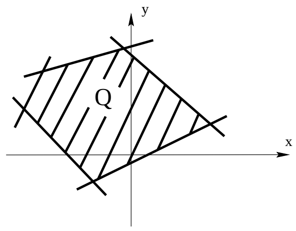


Abbildung 15: Polyeder, Polytop

Definiton (Facette): Sei S eine Seite von P.

$$S \text{ heißt } \textit{Facette} \Leftrightarrow dim(S) = dim(P) - 1.$$

Satz:
- Ein Polyeder hat endlich viele Facetten.
- Jede Facette kann durch eine einzige Ungleichung repräsentiert werden.

Bemerkung:
Jeder Polyeder ist als endlicher Durchschnitt von Halbräumen konvex.

Bei DisPo2 erfolgt die Repräsentation von Polytopen in einer Matrixschreibweise. Sei ein zweidimensionales Polytop durch folgende Ungleichungen bestimmt:
(x und y sind die beiden Koordinaten, k ein fester Parameter)

$$\begin{aligned} &x \geq 1 \ , \\ &y \leq 2 + k \ , \\ &y \geq x \ . \end{aligned}$$

Die Matrixrepräsentation dieses Ungleichungssystems lautet:

$$M = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & -1 & 1 & 2 \\ -1 & 1 & 0 & 0 \end{pmatrix}$$

.

Mit Hilfe dieser Matrix kann das Ungleichungssystem jederzeit mit

$$M \cdot \begin{pmatrix} \textit{Koordinaten} \\ \textit{Parameter} \\ 1 \end{pmatrix} \geq 0$$

wiederhergestellt werden.
Koordinaten bezeichnet den Vektor der (variablen) Koordinaten, hier $\begin{pmatrix} x \\ y \end{pmatrix}$, und *Parameter* den Vektor der (konstanten) Parameter, hier nur k.

Transformationen

Neben den weiter oben beschriebenen Transformation wie Rotation, Skalierung und Translation, die für die räumliche Betrachtung notwendig sind, ist für DisPo2 noch eine weitere Transformation wichtig. Nämlich die Transformation, die aus einem Quellpolytop das Zielpolytop generiert (siehe Kap. 2.3).
Diese ist ganz allgemein eine lineare Funktion $f : \mathbb{Z}^n \rightarrow \mathbb{Z}^m$. Definitionsbereich ist eine Teilmenge des $\mathbb{Z}^n$, das Quellpolytop, und Wertebereich eine Teilmenge des $\mathbb{Z}^m$, das Zielpolytop.

Z.B. $f : \mathbb{Z}^2 \rightarrow \mathbb{Z}^3$

$$f\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y \\ x+y \end{pmatrix}.$$

Diese Transformation kann genauso gut als $m \times n$ - Matrix geschrieben, und die Koordinaten des Bildpunktes durch Matrixmultiplikation berechnet werden.

$f : 3 \times 2$ Matrix, $\qquad f = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}$

$$\begin{pmatrix} x_{neu} \\ y_{neu} \\ z_{neu} \end{pmatrix} = f \cdot \begin{pmatrix} x_{alt} \\ y_{alt} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_{alt} \\ y_{alt} \end{pmatrix} = \begin{pmatrix} x_{alt} \\ y_{alt} \\ x_{alt} + y_{alt} \end{pmatrix}.$$

Da diese Transformationen variabel sind, wird bei DisPo2 hier, im Gegensatz zur Rotation, Skalierung und Translation, die Matrixdarstellung verwendet.

Eine Eigenschaft von Transformationsmatrizen, die später bei der Parallelisierung eine Rolle spielen wird, ist die der Unimodularität.

Definiton (unimodular): Eine Matrix M heißt genau dann *unimodular*, wenn für ihre Determinante $det(M)$ gilt:

$$det(M) = \pm 1.$$

Bemerkung:
- Unimodulare Matrizen sind invertierbar und die Inverse ist ganzzahlig.
- Unimodulare Transformationen erhalten Konvexität.

2.2.2 Schleifensätze im Polytop-Modell

Wie gelangt man nun von einem Schleifensatz, der ja in der Regel in einer bestimmten Programmiersprache notiert ist, zu einem Polytop ?

Sei folgender Schleifensatz gegeben:

```
FOR i = 1 TO 5 STEP 1
   FOR j = i TO MIN(i+2, 5) STEP 1
      A(i, j);
```

Die Anweisung im Schleifenrumpf ist höchstens von den Schleifenvariablen der die Anweisung umgebenden Schleifen abhängig, deren Werte besonderen Begrenzungen, den *Schleifengrenzen*, unterworfen sind. Für die Schleifengrenzen sind affine Ausdrücke erlaubt, die wiederum abhängig sein können von Schleifenvariablen umgebender Schleifen. Außerdem sind bei LooPo für die Untergrenze einer Schleife Maximums- und für die Obergrenze Minimums-Ausdrücke erlaubt.
Der Schleifenrumpf, der in der Regel aus mehreren Anweisungen besteht, jedoch wie eine einzige Anweisung behandelt werden kann, wird für jede Kombination zulässiger (i, j) - Werte einmal ausgeführt. Wenn nun bei jeder Ausführung der Anweisung der Punkt mit den zugehörigen (i, j) - Koordinaten in einem Koordinatensystem markiert wird, so erhält man ein Polytop, den *Indexraum* der betrachteten Anweisung (es genügt, die ganzzahligen Punkte innerhalb des Polytops zu betrachten, da ja auch die Schleifenvariablen nur ganzzahlige Werte annehmen können).

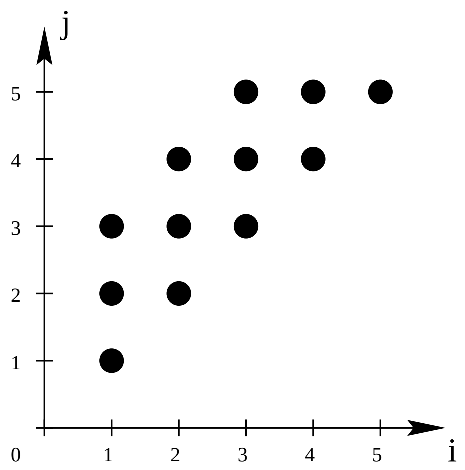


Abbildung 16: Indexraum des FOR-Schleifensatzes

Dieses Verfahren funktioniert jedoch nur bei Schleifensätzen mit einer Schrittweite von +1, da der Indexraum einer Anweisung in einem Schleifensatz mit Schrittweite $\neq 1$ „Löcher“ bekäme und damit kein Polytop mehr darstellen würde. Dies ist jedoch keine echte Beschränkung, da Schleifensätze mit negativer Schrittweite, bzw. einer Schrittweite $\neq 1$ leicht umgeformt werden können.

$$\begin{array}{l} \mathsf{FOR\ i = OG\ TO\ UG\ STEP\ -ST} \\ \quad \mathsf{A(i);} \\ \qquad\qquad \Updownarrow \\ \mathsf{FOR\ i = UG\ TO\ OG\ STEP\ ST} \\ \quad \mathsf{A(OG - i);} \\ \qquad\qquad \Updownarrow \\ \mathsf{FOR\ i = UG\ TO} \left\lfloor \dfrac{\mathsf{OG}}{\mathsf{ST}} \right\rfloor \mathsf{STEP\ 1} \\ \quad \mathsf{A(OG - ST \cdot i);} \end{array}$$

2.3 Parallelisierung im Polytop-Modell

Nachdem nun alle notwendigen Begriffe soweit eingeführt sind, soll im folgenden auf den Parallelisierungsvorgang selbst eingegangen werden. Dieser läuft im wesentlichen in drei Schritten ab:

1. Gewinnung des Quellpolytops aus dem Quellprogramm.
2. Transformierung des Quellpolytops in das Zielpolytop.
3. Generierung des Zielprogramms aus dem Zielpolytop.

Dieses grundlegende Verfahren wird nun exemplarisch an einem einfachen Beispiel aus [Leng93] vorgestellt. Auf existierende Verallgemeinerungen gehen wir im Anschluß ein.

2.3.1 Quellprogramm und Quellpolytop

Sei folgender Schleifensatz gegeben:

```
FOR i = 0 TO N STEP 1
    FOR j = 0 TO N STEP 1
        A(i, j): S[i, j] = S[i–1, j+1];
```

Daraus läßt sich laut Kapitel 2.2.2 das *Quellpolytop* in Abbildung 17 ableiten.

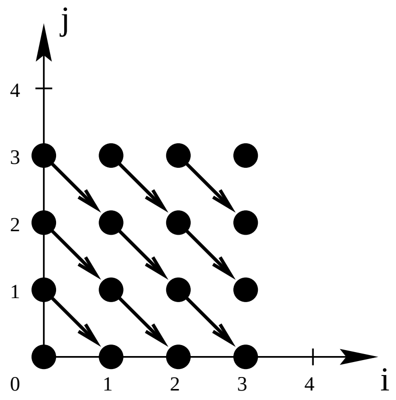


Abbildung 17: Quellpolytop für $N = 3$

Noch nicht besprochen ist jedoch die Ableitung der Abhängigkeitsvektoren aus dem Quellprogramm, die ebenfalls in Abbildung 17 zu sehen sind.

Abhängigkeiten

Definition (Abhängigkeitsvektor): Seien $\vec{v_1}$ und $\vec{v_2}$ die Vektoren der umgebenden Schleifenvariablen.
Eine Abhängigkeit zwischen zwei Anweisungen $A_1(\vec{v_1})$ und $A_2(\vec{v_2})$ entsteht dann,

wenn diese Anweisungen auf dieselbe Speicherzelle zugreifen und zwischen den Anweisungen diese Speicherzelle nicht referenziert wird. Wird $A_1(\vec{v_1})$ vor $A_2(\vec{v_2})$ ausgeführt, so ergibt sich der *Abhängigkeitsvektor* $\vec{d}$ zwischen diesen beiden Anweisungen zu

$$\vec{d} = \vec{v_1} - \vec{v_2}.$$

Im Beispiel schreibt Anweisung $A(i,j)$ in Speicherzelle $S(i,j)$, bevor Anweisung $A(i+1, j-1)$ aus Speicherzelle $S(i,j)$ liest. Der Abhängigkeitsvektor ergibt sich also zu

$$\vec{d} = \begin{pmatrix} i+1 \\ j-1 \end{pmatrix} - \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

Je nach Art und Reihenfolge der Zugriffe unterscheidet man vier verschiedene Abhängigkeitstypen:

1. Schreiben vor Schreiben: *Outputabhängigkeit*
2. Lesen vor Lesen: *Inputabhängigkeit*
3. Schreiben vor Lesen: *Flußabhängigkeit*
4. Lesen vor Schreiben: *Antiabhängigkeit*

Diese Abhängigkeiten spielen eine entscheidende Rolle für das nun folgende Scheduling.

2.3.2 Zieltransformation

Die Transformationsmatrix $T = \begin{pmatrix} \lambda \\ \sigma \end{pmatrix}$, die das Quellpolytop QP in das Zielpolytop ZP überführt, besteht aus zwei Teilen, dem Schedule λ und der Allokation σ, die im folgenden definiert werden sollen.

Definition (Schedule): Sei D die Menge aller (x, x'), mit $x, x' \in QP \;\wedge\; x'$ ist abhängig von x.
Ein *Schedule* ist eine Funktion $t : QP \to \mathbb{Z}$, die jedem Punkt $x \in QP$ einen Zeitschritt zuordnet, zu dem die korrespondierende Anweisung auszuführen ist. Er teilt das Polytop in Hyperebenen ($\widehat{=}$ Zeitschritte), d.h. Punkte auf derselben Hyperebene werden gleichzeitig ausgeführt.
Ein Schedule heißt *gültig*, wenn er die vorhandenen Abhängigkeiten berücksichtigt. Es muß also gelten:

$$(\forall x, x' \in QP \wedge (x, x') \in D : \;\; t(x) < t(x')).$$

Ein Schedule, der alle Punkte $x \in QP$ dem frühestmöglichen Zeitpunkt zuordnet, heißt *freier Schedule*.

Abbildung 18 zeigt einen zeitminimalen gültigen Schedule $\lambda = \begin{pmatrix} 1 & 0 \end{pmatrix}$ für den Beispielschleifensatz.

Nun zum zweiten Teil der Transformationsmatrix, der Allokation.

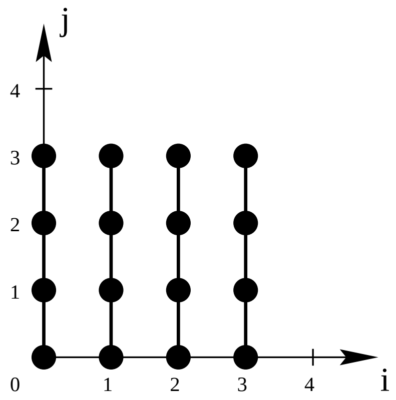


Abbildung 18: Schedule

Definition (Allokation): Sei r die Dimension des Quellpolytops QP.
Eine *Allokation* ist eine Funktion $a : QP \to \mathbb{Z}^{r-1}$, die jedem Punkt $x \in QP$ einen Prozessor zuordnet, auf dem die korrespondierende Anweisung auszuführen ist. Sie teilt das Polytop in Linien ($\widehat{=}$ Prozessornummern), d.h. Punkte auf derselben Linie werden vom selben Prozessor ausgeführt.
Eine Allokation heißt *gültig* im Bezug auf Schedule t, wenn gilt:

$$(\forall x, x' \in QP : \quad t(x) = t(x') \Rightarrow a(x) \neq a(x')).$$

D.h. zwei Anweisungen, die zur gleichen Zeit ausgeführt werden sollen, dürfen nicht demselben Prozessor zugeordnet werden.

Eine weitere Anforderung an die Allokation ist, daß T unimodular sein muß. Damit ist T auch invertierbar und die Inverse ganzzahlig. Allerdings werden wir bei einer Erweiterung der Theorie sehen, daß diese Forderung nicht beibehalten werden muß (siehe Kap. 2.3.4).

Abbildung 19 zeigt eine mögliche Allokation $\sigma = \begin{pmatrix} 1 & 1 \end{pmatrix}$ für den Beispielschleifensatz, die allerdings nicht prozessorminimal ist.

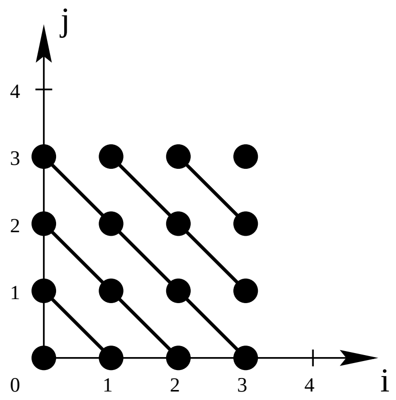


Abbildung 19: Allokation

Die Zieltransformation und ihre Inverse sehen in diesem Fall wie folgt aus:

$$T = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \qquad T^{-1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}.$$

2.3.3 Zielpolytop und Generierung des Zielprogramms

Ausgehend vom Quellpolytop $A\,x \leq b$ und der Zieltransformation T läßt sich das *Zielpolytop* beschreiben durch

$$(A\,T^{-1})\,y \leq b \;, \qquad \text{mit } \; y = T\,x.$$

Abbildung 20 zeigt das resultierende Polytop.

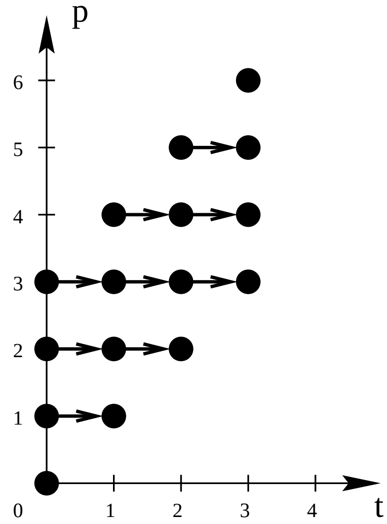


Abbildung 20: Zielpolytop für $N = 3$

Als letzter Schritt muß nun noch aus dem Zielpolytop das Zielprogramm generiert werden. Dazu wandelt man die Scheduling-Dimension in eine sequentielle, und die Allokationsdimension in eine parallele FOR-Schleife um. Dabei besteht die Möglichkeit, die sequentielle Schleife ganz nach außen oder ganz nach innen zu legen. Im ersten Fall erhält man synchrone, im zweiten asynchrone Parallelität.
Im Schleifenrumpf müssen nun noch die Punkte des Quellpolytops aufgezählt werden. Dies erfolgt jedoch im Koordinatensystem des Zielpolytops, weswegen eine einfache Rückrechung erforderlich ist:

$$\begin{pmatrix} i \\ j \end{pmatrix} = T^{-1} \begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} t \\ p - t \end{pmatrix}.$$

Das (synchrone) Zielprogramm ergibt sich somit zu

```
FOR t = 0 TO N STEP 1
    PARFOR p = t TO t + N STEP 1
        A(t, p – t)
```

2.3.4 Erweiterungen

Obwohl das Polytop-Modell aufgrund seiner algebraischen Basis schon sehr leistungsfähig ist, gibt es dennoch einige Ansätze zur Erweiterung und Verbesserung. Hier sollen zwei denkbare Vorgehensweisen kurz vorgestellt werden.

Zulässige Programme

Bisher nahmen wir einige Restriktionen an, denen ein Programm unterworfen werden muß, um es parallelisieren zu können:

1. Der Schleifensatz muß perfekt verschachtelt sein.
2. Es dürfen nur FOR-Schleifen vorkommen.
3. Es dürfen im Schleifenrumpf keine bedingten Anweisungen vorkommen.
4. Schleifengrenzen und Arrayindizes müssen affine Ausdrücke sein.
5. Einzige erlaubte Datenstrukturen sind Arrays.

Jede dieser Einschränkungen ist möglicher Ansatzpunkt für eine Erweiterung der Klasse zulässiger Programme. Die drei ersten sollen im folgenden etwas genauer betrachtet werden, da sie für LooPo keine Gültigkeit mehr besitzen und deswegen auch gewisse Anforderungen an DisPo2 nach sich ziehen.

1. Die Forderung nach perfekter Verschachtelung des betrachteten Schleifensatzes ist bei LooPo aufgehoben. Es sind beliebig verschachtelte Schleifenprogramme zulässig, was zur Folge hat, daß Anweisungen mit verschiedenen, ja sogar verschieden dimensionierten Indexräumen auftreten können. Dies ist natürlich auch bei ihrer Darstellung mit DisPo2 zu berücksichtigen, besonders aber bei der Darstellung der Abhängigkeiten zwischen verschiedenen Indexräumen.

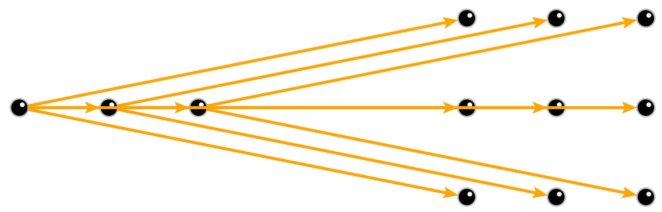


Abbildung 21: Indexräume unterschiedlicher Dimensionalität

2. Bisher sind nur Schleifensätze betrachtet worden, die ausschließlich aus FOR-Schleifen bestanden. Diese Einschränkung hat für LooPo ebenfalls keine Gültigkeit mehr, da es hier auch möglich ist, Schleifensätze mit WHILE-Schleifen zu spezifizieren. Diese unterscheiden sich in zwei ganz wesentlichen Punkten von den bisher untersuchten FOR-Schleifen:

   a. Die Indexräume von Anweisungen innerhalb von WHILE-Schleifen sind potentiell unbeschränkt und in der Regel nicht mehr konvex.

b. WHILE-Schleifen besitzen außerdem keine ausgezeichneten Schleifenvariablen mehr.

Dies macht das Hantieren mit WHILE-Schleifen sehr kompliziert, weswegen man zu einer Modellierung mit FOR-Schleifen übergeht, die schon recht gut verstanden sind.
WHILE-Schleifen werden jeweils durch FOR-Schleifen ersetzt, es werden zwei neue Schleifenvariablen eingeführt und die Anzahl der Durchläufe jeder Schleife wird im voraus mit OG festgelegt.

```
WHILE a < 3
    WHILE b > 2a
        A(a, b);
```

$$\Updownarrow$$

```
FOR i = 1 TO OG1(i)
    FOR j = 1 TO OG2(i,j)
        A(a, b);
```

Abb. 22 zeigt einen möglichen Indexraum für obigen Schleifensatz.

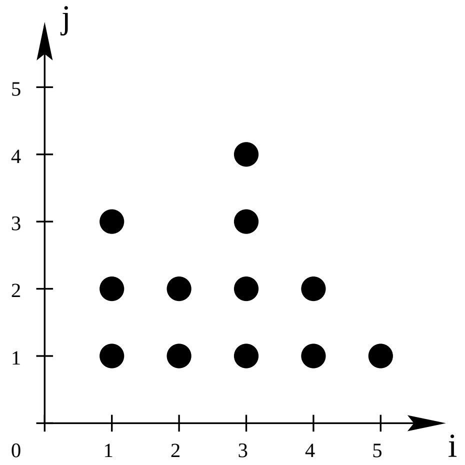


Abbildung 22: Indexraum des WHILE-Schleifensatzes

Zu Einzelheiten zur Behandlung von WHILE-Schleifen siehe [LeGr95] und zur Implementierung in LooPo siehe [Geig97].

Um diese für WHILE-Schleifen charakteristischen „Kämme" darzustellen, genügt es nicht mehr, den Indexraum der entsprechenden FOR-Schleife zu zeichnen, weil dies immer einen konvexen Indexraum zur Folge hätte. Es muß also zwischen „normalen" FOR-Schleifen und FOR-Schleifen, die aus der Umwandlung einer WHILE-Schleife entstanden sind, unterschieden werden. Im zweiten Fall muß die Anzahl der Durchläufe jeder WHILE-Schleife im voraus festgelegt werden, so daß diese bei der Berechnung des Indexraumes berücksichtigt werden kann. Eine genaue Betrachtung der Realisierung der Darstellung von WHILE-Schleifen erfolgt in Kap. 3.1.2.

3. Auch das Verbot von bedingten Anweisungen hat für LooPo keine Gültigkeit mehr. Es sind jedoch nur IF-THEN-Anweisungen zulässig, d.h. es darf kein

ELSE-Zweig angegeben werden. Das IF-THEN-ELSE ist jedoch leicht durch zwei einfache IF-THEN-Anweisungen mit negierter Bedingung zu modellieren.

Die Punkte 4 und 5 behalten auch für LooPo ihre Gültigkeit, weshalb sie hier nicht weiter betrachtet werden sollen. Dies heißt jedoch nicht, daß sie keine einschneidenden Einschränkungen für die Parallelisierung darstellen. Sicherlich wäre es hinsichtlich der wachsenden Objektorientiertheit in der Welt des Programmierens besonders wünschenswert, auch andere Datentypen als nur Arrays zuzulassen.

Zulässige Zieltransformationen

Ein weiterer Ansatzpunkt zur Erweiterung ist die Zieltransformation, an die bisher auch eine Reihe von Anforderungen gestellt ist:

1. Schedule und Allokation müssen affine Funktionen sein.
2. T muß volldimensional sein, d.h. die Zeit nimmt eine Dimension in Anspruch, die restlichen Dimensionen sind Prozessordimensionen.
3. T muß unimodular sein.
4. T ist für alle Anweisungen einheitlich.

1. Eine Verbesserungsmöglichkeit besteht darin, keine Affinität mehr für Schedule und Allokation zu fordern, denn z.B. ist der zeitminimale freie Schedule nicht immer affin.

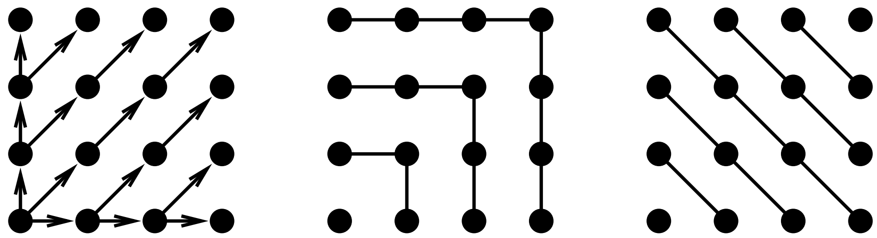


Abbildung 23: Abhängigkeitsgraph, Freier Schedule, kürzester affiner Schedule

   Deswegen ist man bei LooPo dazu übergegangen, auch stückweise-affine Transformationen zuzulassen. Noch allgemeinere Funktionen als stückweise-affine sind jedoch nicht verwendbar, da sich nicht-affine Funktionen nicht mehr mittels einer Matrix darstellen lassen. Es wäre also ein völlig neues Konzept zur Duchführung der Zieltransformation notwendig, die ja bisher mit Hilfe einer Transformationsmatrix realisiert wird. Ob sich der Mehraufwand für solche allgemeinen Transformationen überhaupt lohnen würde, ist mehr als fraglich, weil in vielen Fällen das optimale oder ein nur unwesentlich schlechteres Ergebnis auch mit stückweise-affinen Transformationen erreicht wird.

2. Bei der Parallelisierung des obigen Beispielschleifensatzes wurde nur auf den volldimensionalen Fall eingegangen. Dieser ermöglicht einerseits zwar den größten Geschwindigkeitszuwachs, andererseits ist ein Austausch von Prozessor- und Zeitdimensionen durchaus denkbar und auch erwünscht. So ist es z.B. technisch überhaupt nicht machbar, beliebig hoch dimensionierte Prozessorgitter herzustellen. Auch kann es notwendig sein aus Kostengründen

und um Prozessoren einzusparen, eine oder mehrere Dimensionen vom Raum an die Zeit zu verschieben. Man benötigt dann ein weniger hoch dimensionales Prozessorfeld und erhält dafür eine mehrdimensionale Zeit. Eine zweidimensionale Zeit ist dann z.B. wie die Einteilung der eindimensionalen Zeit in Minuten und Sekunden zu behandeln. In diesem Falle zählt eine Zeit-Dimension die Minuten und die andere die Sekunden.
Dieser Austausch ist bei LooPo möglich und stellt auch für DisPo2 kein Problem dar, da es für die Transformation zunächst unerheblich ist, welche Dimensionen zur Zeit und welche zum Raum gehören.
Das ändert sich erst bei der in DisPo2 möglichen Simulation des Zielprogramms, die die zu einem bestimmten Zeitpunkt jeweils aktiven Indexpunkte hervorhebt (siehe Kap. 3.1.2). Abgesehen von der notwendigen Unterscheidung zwischen synchronem und asynchronem Zielprogramm, bei dem sich die Zeitschleifen jeweils ganz außen oder ganz innen befinden, ist dann auch die genaue Anzahl der Zeitdimensionen von Interesse, um die Bearbeitungsreihenfolge der einzelnen Indexpunkte festzustellen.

3. Zusätzlich kann auch noch auf die Unimodularität der Zieltransformation verzichtet werden. Dies hat zwar zur Folge, daß die Konvexität des Zielpolytops verloren geht und $(A\,T^{-1})\,y \leq b$ nur noch die konvexe Hülle des Zielpolytops beschreibt. Dies ist aber durch geeignete Wahl der Schleifenschrittweiten und Anpassung der Schleifengrenzen zu kompensieren. Es ist sogar nicht einmal mehr notwendig, die Umkehrbarkeit der Zieltransformation zu fordern [Wetz95].

$$T = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}, \qquad det(T) = 2.$$

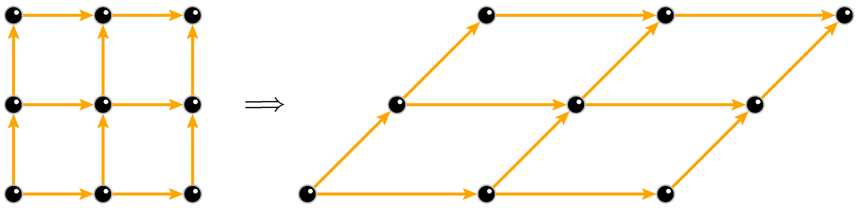


Abbildung 24: Nicht unimodulare Transformation

So wurde bei LooPo auf diese Forderung verzichtet und auch DisPo2 zeigt nicht-unimodulare Transformationen problemlos an.

4. Zuletzt soll die Einschränkung einer einheitlichen Transformationsmatrix für alle Anweisungen aufgehoben werden. Bei LooPo ist es möglich, jede einzelne Anweisung mit einer eigenen Transformationsmatrix zu transformieren, wobei es auch vorkommt, daß aus einer Quellanweisung mehrere Zielanweisungen entstehen, die sogar unterschiedliche Dimensionalität besitzen können. Dies erfordert von DisPo2 eine Behandlung der Zieltransformation, die weit über die Transformation eines Indexraumes mit Hilfe einer Transformationsmatrix hinausgeht.

Soweit zu den Grundlagen, die zum Verständnis der Arbeitsweise von DisPo2 notwendig sind. Da nun alle notwendigen Anforderungen bekannt sind, kann eine detaillierte Betrachtung dieses Tools in Angriff genommen werden.

3 Realisierung

Dieses Kapitel befaßt sich mit dem Realisierungskonzept von DisPo2 und mit Einzelheiten seiner Implementierung. Außerdem werden am Schluß zwei einfache Beispielprogramme und ihre Darstellung bei DisPo2 vorgestellt.

3.1 Darstellung von Polytopen mit DisPo2

3.1.1 Arbeitsweise

Um DisPo2 starten zu können, muß LooPo's Abhängigkeitsanalyse abgeschlossen sein, da DisPo2 deren Ergebnisse als Eingabe benötigt.
Im folgenden soll die Vorgehensweise zur Darstellung von Indexräumen und Abhängigkeiten erläutert werden.

Vom Quellprogramm zum Polytop

Um ausgehend von einem Quellprogramm den Indexraum einer oder mehrerer Anweisungen darstellen zu können, geht DisPo2 in mehreren Arbeitsschritten vor, die in Abbildung 25 aufgeführt sind.

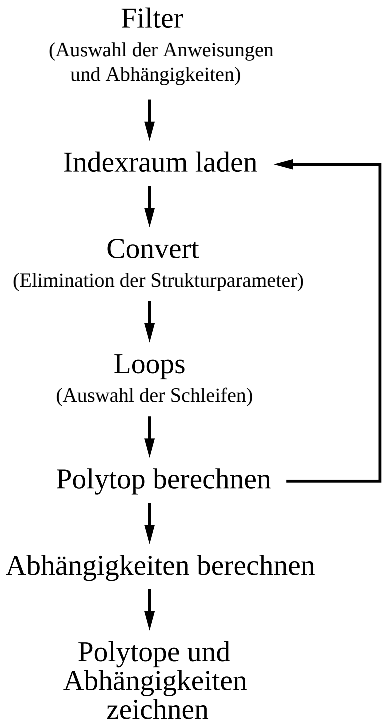


Abbildung 25: Arbeitsschritte zum Zeichnen der Indexräume

Zunächst werden im sog. „Filter“ die Anweisungen ausgewählt, deren Indexräume zur Anzeige kommen sollen. Dabei ist es möglich eine oder mehrere Anweisungen aus beliebigen (auch unterschiedlichen) Gruppen (Gruppe $\hat{=}$ Menge von Anweisungen, die von denselben Schleifen umgeben werden) zu wählen. Außerdem werden bereits im Filter auch die Abhängigkeiten selektiert, die später dargestellt werden sollen.

Es sind allerdings immer nur diejenigen Abhängigkeiten wählbar, die Anweisungen betreffen, die vorher markiert wurden.

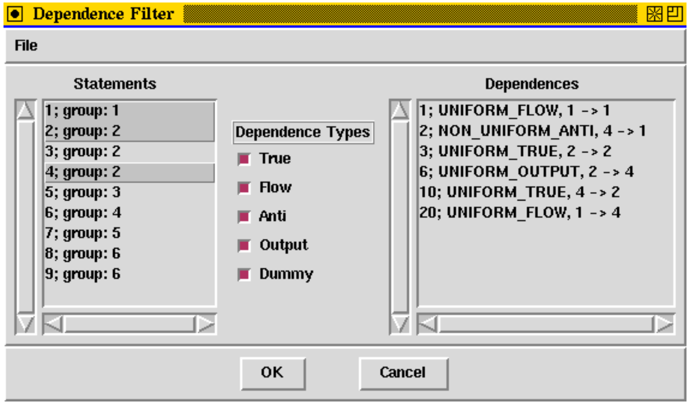


Abbildung 26: DisPo2's Filter-Fenster

Abbildung 26 zeigt das Filter-Fenster zur Auswahl der Anweisungen und Abhängigkeiten.

Beim Schließen des Filter-Fensters werden die Ergebnisse der jeweiligen Auswahl in einem temporären Verzeichnis gespeichert.
Danach wird der Indexraum für die erste der selektierten Anweisungen, der bei der Abhängigkeitsanalyse berechnet wird, geladen.
Im dritten Schritt werden im Indexraum evtl. vorhandene Strukturparameter eliminiert. Strukturparameter sind Variablen, die einen beliebigen aber festen, vom Benutzer einzugebenden Wert erhalten. Im folgenden Beispiel stellt N einen Strukturparameter dar.

```
FOR i = 0 TO N STEP 1
   A(i, j)
```

Die Werte für die Strukturparameter sind in dem in Abbildung 27 gezeigten Fenster einzugeben. Dann werden die Strukturparameter durch die angegebenen Werte ersetzt und der Indexraum neu berechnet. Es entsteht ein neuer Indexraum, der nur noch von Schleifenvariablen und nicht von Strukturparametern abhängt.

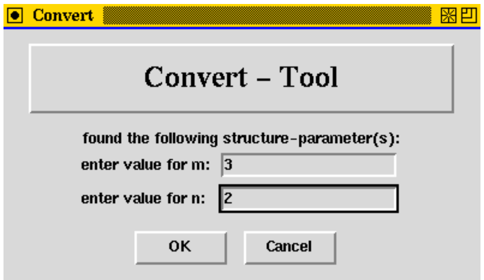


Abbildung 27: DisPo2's Convert-Fenster

Nun ist es aber einerseits nicht zweckmäßig, Polytope darzustellen, deren Dimension größer als 3 ist, und andererseits sind Schleifenprogramme mit mehr als drei ineinander verschachtelten Schleifen durchaus keine Seltenheit. Es muß daher eine Möglichkeit geschaffen werden, Schleifen „auszublenden“ bzw. die Schleifen anzugeben, die dargestellt werden sollen. Dies geschieht, wie Abbildung 28 zeigt, in einem vierten Schritt.

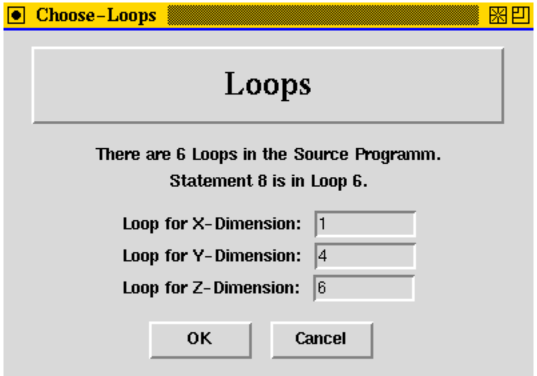


Abbildung 28: DisPo2's Loops-Fenster

Hier kann für jede der drei Raumrichtungen die zu zeichnende Schleife angegeben werden. Diese müssen in aufsteigender Reihenfolge angegeben werden und die letzte Schleife muß diejenige sein, die der zum Indexraum gehörenden Anweisung unmittelbar übergeordnet ist. Werden dabei Schleifen ausgeblendet, von denen zu zeichnende Schleifen abhängig sind, so sind für die Schleifenvariablen der ausgeblendeten Schleifen feste Werte anzugeben.

Nun kann aus dem Ungleichungssystem, das den Indexraum der Anweisung darstellt, das Polytop berechnet werden. In diesem Schritt werden allerdings nur die Schleifenkoordinaten der Indexpunkte berechnet und gespeichert, da die tatsächliche Plazierung der Indexräume am Bildschirm erst zu einem späteren Zeitpunkt, beim Zeichnen erfolgt. Erst dann sind alle Indexräume berechnet und ihre Gestalt und Größe stehen fest. An dieser Stelle erfolgt ein Rücksprung zum Laden eines neuen Indexraumes, falls im Filter mehrere Anweisungen ausgewählt wurden, und es wiederholen sich alle Schritte bis hierher.

Falls nur eine Anweisung gewählt oder der letzte Indexraum gerade bearbeitet wurde, schließt sich nun die Berechnung der Abhängigkeiten an. Wie diese Berechnung und Darstellung funktioniert, wird später in Kap. 3.1.2 behandelt.

Zum Schluß werden die berechneten Indexräume im Tk-Canvas, das mit einem zweidimensionalen Zeichenbrett vergleichbar ist, ausgegeben. D.h. sie werden je nach Größe und Anzahl plaziert und gezeichnet. Dazu ist es notwendig zu wissen, wie das Canvas Objekte (in unserem Fall Indexpunkte), die in ihm dargestellt sind, verwaltet. Die Objekte werden in der Reihenfolge ihrer Entstehung in die sog. „Display-Liste“ eingetragen, d.h. neue Objekte werden hinten an die Display-Liste angehängt. In dieser Reihenfolge gibt sie das Canvas aus, was zur Folge hat, daß später gezeichnete Objekte (hinten in der Display-Liste) früher gezeichnete (vorne in der Display-Liste) verdecken.
Um nun eine dreidimensional richtige Darstellung der Polytope zu erreichen, müssen die einzelnen Punkte eines Indexraumes nach aufsteigender Z-Koordinate sortiert werden. In dieser Reihenfolge können sie dann gezeichnet und in die Display-Liste

eingetragen werden, wobei Indexpunkte mit größerer Z-Koordinate solche mit kleinerer Z-Koordinate überdecken.

3.1.2 Features

In diesem Kapitel soll ein Überblick über die Vielzahl von Möglichkeiten gegeben werden, die DisPo2 bietet. Angefangen von den verschiedenen Betrachtungsweisen eines Polytops, bis hin zur Simulation der zu einem bestimmten Zeitpunkt aktiven Indexpunkte.

Geometrische Transformationen

Mit geometrischen Transformationen sind zum einen diejenigen Transformationen gemeint, die die Lage und Größe eines Indexraumes verändern, sprich die Rotation, Translation und Skalierung, und zum anderen die verschiedenen Möglichkeiten einer perspektivischen Betrachtung, die schon ausführlich diskutiert wurden.
Der größte Teil der Bedienungselemente im DisPo2-Hauptfenster befaßt sich mit geometrischen Transformationen. Zum Vergleich noch einmal das DisPo2-Fenster.

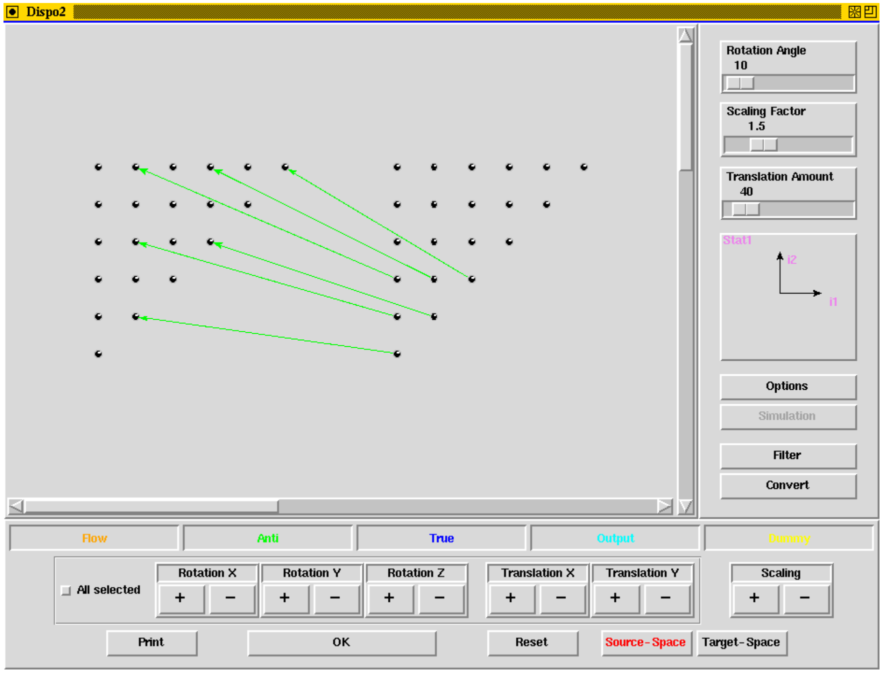


Abbildung 29: DisPo2's Hauptfenster

Die Bedienelemente für die Rotation sind im linken unteren Teil des DisPo2-Fensters angeordnet, und es ist eine Rotation um alle drei Raumachsen sowohl im, als auch gegen den Uhrzeigersinn vorgesehen. Um die Drehachse zu ermitteln, wird der Mittelpunkt des jeweiligen Indexraumes berechnet und eine Gerade in der betreffenden Raumrichtung hindurchgelegt. Durch das „Hintereinanderschalten" von Drehungen um die Raumachsen können auch Drehungen um beliebige Achsen erreicht werden.

Rechts neben den Knöpfen für die Rotation befinden sich die für die Translation. Hier ist nur eine Verschiebung in X- und Y-Richtung vorgesehen, da eine Verschiebung in Z-Richtung keine sichtbaren Auswirkungen hätte, außer bei perspektivischer Betrachtung, aber derselbe Effekt ist auch mit Skalierung zu erzielen.

Ganz rechts befinden sich schließlich die Bedienelemente für die Skalierung, mit denen Indexräume beliebig vergrößert und verkleinert werden können. Zentrum der Skalierung ist immer der Mittelpunkt des gerade gezeigten Bildausschnitts, der durch die Rollbalken zu den Seiten des Canvas verändert werden kann.

Die Skalierung bezieht sich jedoch, anders als die Rotation und Translation, immer auf alle angezeigten Indexräume. Dies ist notwendig um einigen unschönen Effekten bei der Darstellung von Abhängigkeiten zwischen verschiedenen Indexräumen aus dem Wege zu gehen.
Rotation und Translation beziehen sich in der Regel nur auf ein bestimmtes Polytop, das mit der Maus angewählt werden kann. Auch die Anzeige des Koordinatenkreuzes im rechten Teil des DisPo2-Fensters bezieht sich auf das jeweils angewählte (aktive) Polytop. Ist der Kontroll-Knopf im linken unteren Eck des DisPo2-Fensters aktiviert, können alle Polytope gleichzeitig auch rotiert und verschoben werden. Die Anzeige des Koordinatenkreuzes bezieht sich dann auf das zuletzt aktive Polytop.

Die Werte für den Rotationswinkel, den Verschiebungswert und den Skalierungsfaktor sind mit Hilfe der Bedienelemente rechts oben im DisPo2-Fenster beliebig einstellbar. Dabei können die Werte mit der Maus und der linken Taste unmittelbar eingestellt werden, oder man kann sie mit der mittleren Maustaste um eins erniedrigen, bzw. mit der rechten Maustaste um eins erhöhen.

Nun zu den verschiedenen Projektionsmöglichkeiten, die DisPo2 bietet.
Wie schon erwähnt, kann für die Betrachtung eines Indexraumes sowohl parallele, als auch perspektivische Projektion gewählt werden. Wird nichts angegeben, so verwendet DisPo2 als Standard die Parallelprojektion mit dem Projektionsvektor $(0, 0, -1)$, d.h. die Projektion erfolgt senkrecht auf die XY-Ebene des Bildschirmkoordinatenysystems. Wird hingegen perspektivische Projektion gewünscht, muß diese im Options-Fenster (siehe Abbildung 30) aktiviert werden.

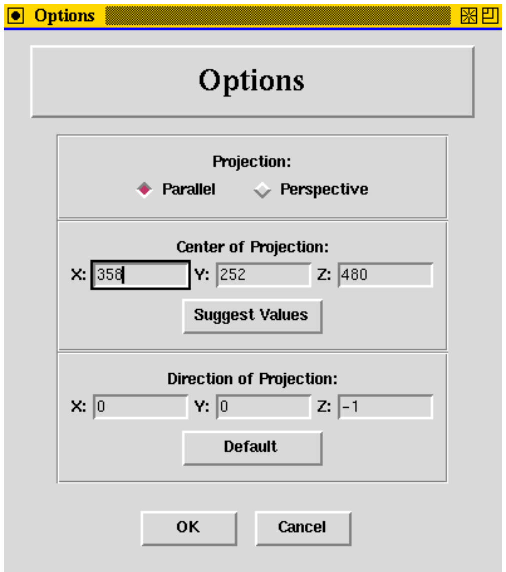


Abbildung 30: DisPo2's Options-Fenster

Hier ist es auch möglich, ein beliebiges Projektionszentrum für die perspektivische Projektion und einen beliebigen Projektionsvektor für die Parallelprojektion anzugeben.
Bei der Findung eines brauchbaren Projektionszentrums leistet der „Suggest Values“-Knopf wertvolle Arbeit. Mit dessen Hilfe kann man sich einen geeigneten Punkt als Projektionszentrum vorschlagen lassen, um den dann direkt zu verwenden, oder ihn nach Bedarf zu variieren. Der vorgeschlagene Punkt liegt immer in der Mitte des gerade angezeigten Ausschnitts des Canvas und in Z-Richtung in geeigneter Entfernung vor den angezeigten Indexräumen („Max. Z-Koordinate“ + 400). Der „Default“-Knopf dagegen setzt den Projektionsvektor für die Parallelprojektion nur auf den Standardwert $(0, 0, -1)$ zurück. Hier ist keine weitere Hilfestellung notwendig, da es beim Projektionsvektor, im Gegensatz zum Projektionszentrum, nicht auf die genaue Lage gegenüber dem betrachteten Polytop ankommt.

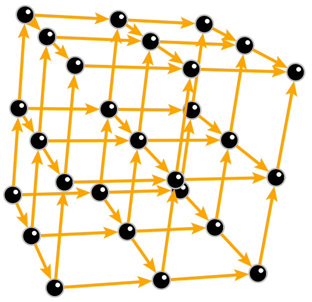


Abbildung 31: 3-Punkt Projektion eines Würfels

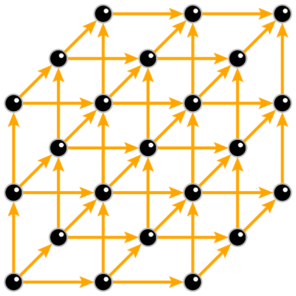


Abbildung 32: Schräge Projektion eines Würfels

Darstellung von Abhängigkeiten

Wie schon weiter oben beschrieben, werden im Filter zunächst einmal diejenigen Abhängigkeiten, die gezeigt werden sollen, selektiert und gespeichert. Diese Abhängigkeiten werden dann später bei der Berechnung der einzelnen Abhängigkeitspfeile wieder eingelesen. Dazu werden zuerst für jede Abhängigkeit der Indexraum des Quellstatements und der Indexraum des Zielstatements der Abhängigkeit ermittelt. Dann wird auf jeden Punkt des Zielindexraumes die h-Transformation ([Feau91] und [Keim97]) angewandt. Die h-Transformation ist eine affine Funktion, die bei gegebenem Zielpunkt die Quelle eines möglichen Abhängigkeitspfeils liefert. Ist der so berechnete Punkt Element des Quellindexraumes, so handelt es sich bei dem Paar (Quellpunkt, Zielpunkt) um einen gültigen Abhängigkeitspfeil, und dieser wird zusammen mit dem Abhängigkeitstyp gespeichert.

Nachdem die Abhängigkeitspfeile berechnet sind, müssen sie dargestellt werden. Dies geschieht unmittelbar nach dem Zeichnen der Indexpunkte. Dabei tritt ein Problem auf, das hauptverantwortlich für den Geschwindigkeitsverlust von DisPo2 bei einer steigenden Anzahl von Abhängigkeiten ist.
Anders als den Indexpunkten kann einem Abhängigkeitspfeil nicht immer eine eindeutige Z-Koordinate zugewiesen werden, da sein Start- und Endpunkt ja unterschiedliche Z-Koordinaten besitzen können. Wie Abbildung 33 zeigt, ist es möglich, daß ein Abhängigkeitspfeil einen Indexpunkt mit einer bestimmten Z-Koordinate verdeckt, aber von einem anderen mit kleinerer Z-Koordinate verdeckt wird. Damit läßt er sich nicht mehr so einfach in die Display-Liste des Canvas einsortieren, die ja nach der Z-Koordinate ihrer Elemente geordnet ist. Unter Umständen wird sogar

eine Umsortierung der Elemente der Display-Liste notwendig.

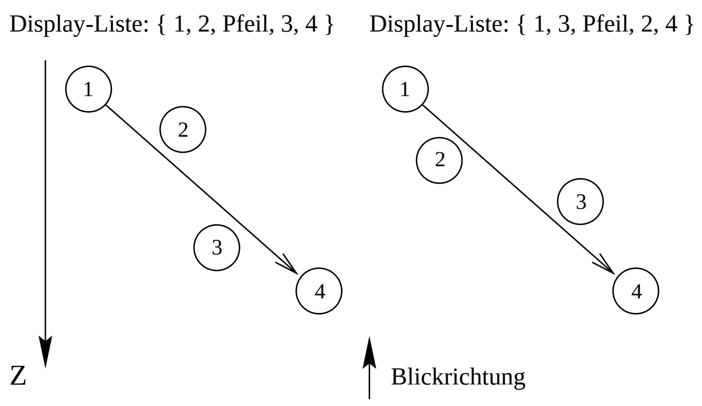


Abbildung 33: Einfügen eines Pfeils in die Display-Liste

DisPo2 löst dieses Problem, indem es den Pfeil unmittelbar hinter dem Quell- bzw. Zielpunkt (je nachdem, welcher die kleinere Z-Koordinate besitzt) in die Display-Liste einträgt und dann die Indexpunkte, die sich in der Liste zwischen dem Quell- und Zielpunkt befinden, vor den Pfeil verschiebt, falls dieser den Indexpunkt überdeckt. Bei dieser Verschiebung eines Indexpunktes nach vorne in der Display-Liste ist allerdings darauf zu achten, daß er keine anderen Indexpunkte überspringt, die er eigentlich überdecken sollte. Indexpunkte, für die dies der Fall ist, müssen dann natürlich mitverschoben werden. Die Ordnung der Indexpunkte nach der Z-Koordinate geht innerhalb der Display-Liste dadurch selbstverständlich verloren.

Die verschiedenen Abhängigkeitstypen bei DisPo2 sind anhand ihrer Farbgebung, die vom Benutzer frei wählbar ist, zu unterscheiden. Unmittelbar unter dem Canvas befinden sich Bedienelemente, auf denen die Zuordung von Abhängigkeitstypen zu Farben angegeben ist. Dort läßt sich auch die Anzeige der Abhängigkeiten getrennt nach ihrem Typ beliebig an- und abschalten.

Darstellung des Zielpolytops

Beim Start von DisPo2 werden zunächst die Quellindexräume der gewählten Anweisungen angezeigt. Um nun zu den beim Parallelisierungsvorgang beschriebenen Zielindexräumen zu gelangen, existieren grundsätzlich zwei verschiedene Möglichkeiten:

1. Die gewünschte Transformation kann von Hand eingegeben werden. Dies geschieht in dem in Abbildung 34 gezeigten Eingabefenster, das nach einem Druck auf den „Target Space“-Knopf erscheint. Hier ist nun für jede angezeigte Anweisung gesondert eine Transformationsmatrix anzugeben. Nach der Angabe der letzten Matrix werden die entsprechenden Zielindexräume angezeigt.

2. Es können die von LooPo errechneten Transformationsmatrizen verwendet werden. In diesem Falle entfällt die Eingabe der einzelnen Matrizen und die Zielindexräume werden sofort nach dem Druck auf den „Target Space“-Knopf angezeigt.

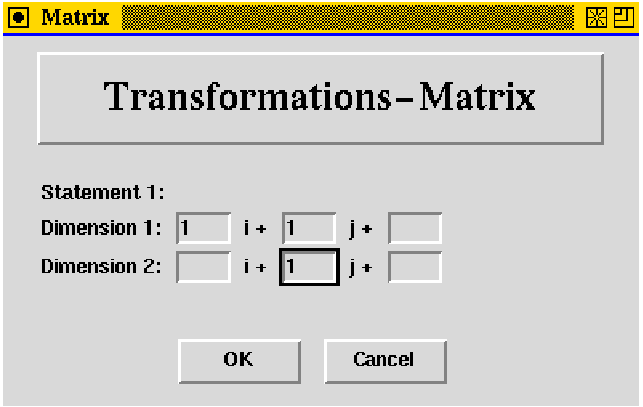


Abbildung 34: Eingabe der Transformationsmatrix

Solange die Berechnungen des Schedulers, des Allokators und des Target-Generators noch andauern, ist ausschließlich die Eingabe der Transformationsmatrizen von Hand möglich. Sind diese Arbeiten jedoch abgeschlossen, kann im Options-Menü frei zwischen den beiden Möglichkeiten ausgewählt werden.
Ein Druck auf den „Source Space"-Knopf dagegen bringt wieder die ursprünglichen Quellindexräume zur Anzeige.

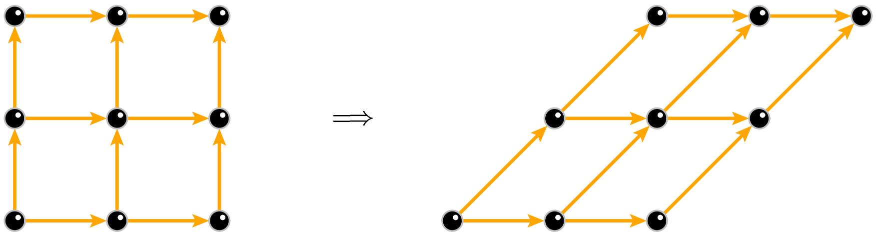


Abbildung 35: Zieltransformation

Abbildung 35 zeigt die Transformation eines zweidimensionalen Indexraumes mit der Transformationsmatrix $T = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$.

Anzeige von WHILE-Schleifen

Wie schon in Kap. 2.3.4 angedeutet, entstehen bei der Anzeige des Indexraumes von WHILE-Schleifen verschiedene Probleme, die eine Umwandlung in FOR-Schleifen notwendig machen. Diese „Normalisierung" [Geig97] eines WHILE-Schleifensatzes geschieht bei LooPo folgendermaßen:

```
WHILE COND_1
    WHILE COND_2
        WHILE COND_3
            Anweisung;
```

$$\Downarrow$$

```
FOR loopo_nl_wii_1 = 1 TO loopo_nl_wubb_1
    FOR loopo_nl_wii_2 = 1 TO loopo_nl_wubb_2
        FOR loopo_nl_wii_3 = 1 TO loopo_nl_wubb_3
            Anweisung;
```

Der normalisierte Schleifensatz enthält nun nur noch FOR-Schleifen und es wurden außerdem drei neue Strukturparameter „loopo_nl_wubb_X“ (X ist eine eindeutige Nummer zur Identifizierung des Parameters) eingeführt, um die Anzahl der Schleifendurchläufe festzulegen.

Dieser Schleifensatz unterscheidet sich nun durch nichts mehr von einem normalen FOR-Schleifensatz und könnte infolgedessen auch als solcher behandelt werden, d.h. die Strukturparameter würden im Convert-Fenster durch feste Werte ersetzt. Da sich die Werte der Strukturparameter jedoch nicht von Schleifendurchlauf zu Schleifendurchlauf verändern, wäre das Ergebnis ein konvexer Indexraum. Die Darstellung der für die WHILE-Schleifen charakteristischen „Kämme“ (siehe Kap. 2.3.4) wäre so nicht möglich. Um dies dennoch zu ermöglichen, muß DisPo2 erkennen, ob es sich bei einem gegebenen Schleifensatz um einen normalisierten WHILE-Schleifensatz oder um einen regulären FOR-Schleifensatz handelt. Dies geschieht anhand der charakteristischen Namen, die die neu erzeugten Strukturparameter bei der Normalisierung erhalten. Werden solche Strukturparameter erkannt, so werden sie nicht, wie die übliche Vorgehensweise wäre, in „Convert“ ersetzt, sondern ihr Wert wird für jedem Schleifendurchlauf neu festgelegt. Dazu müssen die einzelnen Grenzen in einer Datei im Lib-Verzeichnis abgespeichert werden, die für unser dreidimensionales Beispiel etwa folgende Formate haben könnte:

3							
3							
3	2	4		1			
3			oder	4			
2	2	4		2	3	1	3
3							
3	2	3					

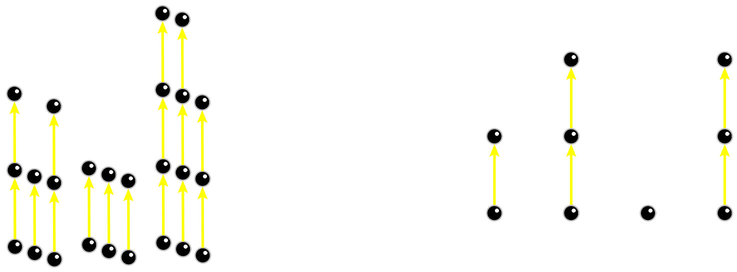


Abbildung 36: Wirkung unterschiedlicher Grenzdateien

Abbildung 36 zeigt die resultierenden, nicht-konvexen Indexräume für die unterschiedlichen Grenzangaben. Der Einfluß der Grenzdatei auf das Aussehen des Indexraumes ist also ganz erheblich. Eine genaue Beschreibung des Aufbaus dieser Grenzdatei befindet sich im Anhang.

Simulation

Ziel der Simulation ist es, diejenigen Punkte eines Indexraumes farblich hervorzuheben, die zu einem ganz bestimmten Zeitpunkt aktiv sind. Dies ist sowohl bei der Anzeige der Zielindexräume, als auch bei der Anzeige der Quellindexräume sinnvoll. Um diese Möglichkeit nutzen zu können, muß allerdings LooPo's Scheduling, Allokation und Generierung der Zielschleifen abgeschlossen sein.

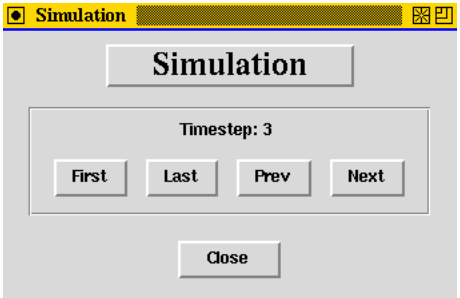


Abbildung 37: DisPo2's Simulations-Fenster

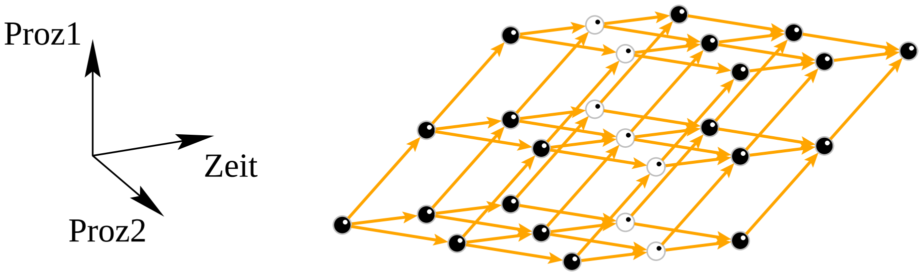


Abbildung 38: Simulation des Zielprogramms

Die Simulation wird durch einen Druck auf den „Simulation“-Knopf aktiviert, wonach das Simulationsfenster (siehe Abbildung 37) am Bildschirm erscheint. Zu Beginn werden die Indexpunkte weiß gekennzeichnet, die zuallererst ausgeführt werden, d.h. die dem kleinsten vorkommenden Zeitschritt zugeordnet sind. Mit Hilfe der Bedienelemente im Simulationsfenster ist es möglich, die einzelnen Zeitschritte der Reihe nach durchzugehen, oder sofort zum erst- bzw. letztmöglichen Zeitschritt zu springen. Diese Zeitschritte, sowie der jeweils nächste bzw. vorhergehende, werden von DisPo2 automatisch ermittelt, wobei auch eine mögliche mehrdimensionale Zeit, die beim Scheduling auftreten kann, berücksichtigt wird.
Die Simulation bezieht sich immer auf alle angezeigten Indexräume. Es ist also nicht möglich die Simulation auf einen isolierten Indexraum anzuwenden.

3.2 Implementierung

Das folgende Kapitel behandelt kurz die Besonderheiten der Implementierung von DisPo2.

3.2.1 Systemvoraussetzungen

Software

DisPo2 ist in C++ programmiert mit einer Oberfläche in Tcl/Tk. Bei der Entwicklung wurde darauf geachtet, daß nur frei erhältliche Tools verwendet wurden, so daß diesbezüglich keine besonderen Voraussetzungen zum Betrieb von DisPo2 erforderlich sind. Dabei kamen Gnu gcc 2.7.2, Tcl 7.4 und Tk 4.0 zum Einsatz.

Hardware

Auch bei der Hardware gibt es keine besonderen Voraussetzungen für die Installation von DisPo2. Es läuft problemlos sowohl auf Unix und Solaris Workstations, als auch auf Linux PCs.
Obwohl nicht zwingend erforderlich, wird der Einsatz eines Farbmonitors empfohlen, da die Unterscheidung der einzelnen Abhängigkeitstypen farblich erfolgt. Diese Unterscheidung wird durch einen Graustufenmonitor in den meisten Fällen zwar nicht unmöglich, jedoch erheblich erschwert.

3.2.2 Ein-/Ausgabe

Der Aufruf von DisPo2 erfolgt per Mausklick im LooPo-Hauptfenster. Dabei werden automatisch <Loopodir>, das Verzeichnis in dem LooPo installiert ist, und <Tempdir>, das Verzeichnis in dem LooPo zur Laufzeit temporäre Dateien ablegt, an DisPo2 übergeben. Diese beiden Kommandozeilenparameter sind aber nicht die einzigen Eingaben, die DisPo2 benötigt. Es wird darüber hinaus erwartet, daß die Abhängigkeitsanalyse eines Projektes vor dem Start von DisPo2 abgeschlossen ist. Dies alles sind Eingaben, die DisPo2 in jedem Fall benötigt. Wie schon beschrieben, müssen für die Darstellung des Zielpolytops bzw. die Darstellung von WHILE-Schleifen zusätzlich die Ergebnisse des Target-Generators bzw. die Datei „while_borders“ im Lib-Verzeichnis vorhanden sein.

Die Ausgabe von DisPo2 erfogt im Prinzip ausschließlich grafisch auf dem Bildschirm. Auf Wunsch können die angezeigten Indexräume inklusive der Abhängigkeiten jedoch auch ausgedruckt werden. Der Ausdruck erfolgt in eine Postscript-Datei, die im Lib-Verzeichnis abgelegt wird.

3.2.3 Datenstruktur

Abbildung 39 zeigt die Hierarchie der Elemente von DisPo2's Datenstruktur. Im Anhang befindet sich eine detaillierte Betrachtung der Datenstruktur (siehe Seite 49).

Die Basis der Datenstruktur ist die „Project“-Struktur. Sie enthält globale Daten, sowie Zeiger auf Strukturen vom Typ „stmntinfo“, „deps“ und „PolyList“.
„stmntinfo“ enthält Informationen zu allen Anweisungen im Quellprogramm, wie z.B. ihre Nummer und Schachtelungstiefe. Außerdem beinhaltet sie einen Zeiger

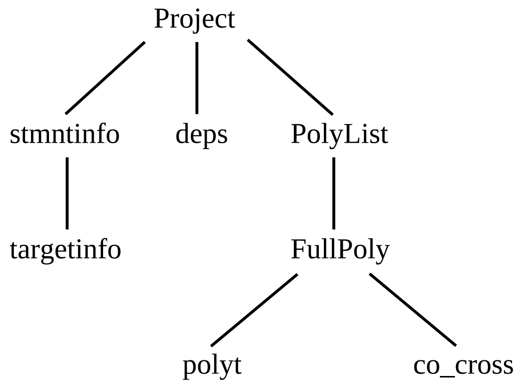


Abbildung 39: DisPo2's Datenstruktur

auf die Struktur „targetinfo“, die die Zuordnung von Anweisungen aus dem Quellprogramm zu Anweisungen aus dem Zielprogramm enthält. Dies ist wichtig für die Darstellung des Zielpolytops (siehe Kap. 3.1.2).
„deps“ speichert Informationen über die Abhängigkeiten zwischen und innerhalb von Indexräumen.
„PolyList“ ist eine verkettete Liste von Indexräumen. Jeder Knoten der Liste enthält einen Zeiger auf die Struktur „FullPoly“, die alle erforderlichen Daten über den Indexraum, sowie einen Zeiger auf „polyt“, das eigentliche Polytop, eine verkettete Liste, deren Knoten jeweils genau einen Indexpunkt repräsentieren, beinhaltet. Außerdem enthält „FullPoly“ einen Zeiger auf „co_cross“, das zugehörige Koordinatenkreuz.

3.2.4 Äußeres Erscheinungsbild

Mit „äußerem Erscheinungsbild“ ist vor allem die Größe der einzelnen Indexkugeln und ihr Abstand zueinander gemeint, aber auch die Standardwerte für Rotation, Translation und Skalierung können an die eigenen Erfordernisse angepaßt werden. Außerdem können die Farben der verschiedenen Abhängigkeiten vom Benutzer frei gewählt werden.

Die Definition der Initialwerte für Rotation, Translation und Skalierung befindet sich am Anfang der Datei „dispo2_win.cc“ und kann nach Belieben verändert werden.

```
ROT_ANGLE_INIT    = 10,
TRA_AMOUNT_INIT = 40,
SCA_FACTOR_INIT  = 1.5;
```

Auch die Initialwerte für den Radius und den Abstand der einzelnen Indexpunkte sind beliebig veränderbar und bei der Initialisierung der „Project“-Datenstruktur am Beginn von „dispo2.cc“ anzugeben.

```
PROJECT->node_radius    = 4;
PROJECT->node_distance = 40;
```

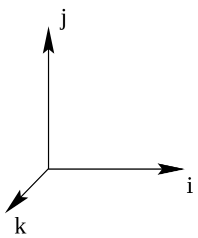
Abschließend sei noch auf die Definition der Farben für die Abhängigkeitspfeile am Anfang von „deps.cc“ hingewiesen. Auch diese Farben sind frei wählbar.

UNIFORM_TRUE	: Blue
UNIFORM_FLOW	: Orange
UNIFORM_ANTI	: Green
UNIFORM_OUTPUT	: Cyan
UNIFORM_DUMMY	: Yellow
NON_UNIFORM_TRUE	: Blue
NON_UNIFORM_FLOW	: Orange
NON_UNIFORM_ANTI	: Green
NON_UNIFORM_OUTPUT	: Cyan
NON_UNIFORM_DUMMY	: Yellow

3.3 Ausgewählte Beispiele

Zum Schluß noch einige mit DisPo2 erstellte Bilder von ausgewählten Beispielen aus der Literatur.

Das Koordinatensystem ist für die Indexräume der folgenden Beispiele immer gleich und deshalb nur einmal an dieser Stelle angegeben.



3.3.1 Matrixmultiplikation

Ein Beispiel, das in der Literatur zur Schleifenparallelisierung immer wieder auftaucht, ist die Multiplikation zweier Matrizen. Es ist übersichtlich, perfekt verschachtelt und enthält nur Arrays als Datenstruktur, womit es selbst die restriktivsten Anforderungen an ein zu parallelisierendes Programm erfüllt.
Wir wollen zwei verschiedene Programmierstile zur Lösung dieses Problems betrachten, nämlich den „imperativen" und den „systolischen".

$$C = A \cdot B \quad , \qquad \text{mit}$$

$$A = \begin{pmatrix} a_{00} & \dots & a_{02} \\ \vdots & & \vdots \\ a_{20} & \dots & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{00} & \dots & b_{02} \\ \vdots & & \vdots \\ b_{20} & \dots & b_{22} \end{pmatrix}, \quad C = \begin{pmatrix} c_{00} & \dots & c_{02} \\ \vdots & & \vdots \\ c_{20} & \dots & c_{22} \end{pmatrix}$$

Imperativ

Der imperative Ansatz ist der einfachste Lösungsweg, den man in einer gängigen (imperativen) Programmiersprache wählen würde. Er enthält sowohl mehrfache Schreib-, als auch Lesezugriffe auf Variablen.

```
FOR(i=0; i<3; i++)
    FOR(j=0; j<3; j++)
        FOR(k=0; k<3; k++)
            c[i, j] = c[i, j] + a[i, k] · b[k, j];
```

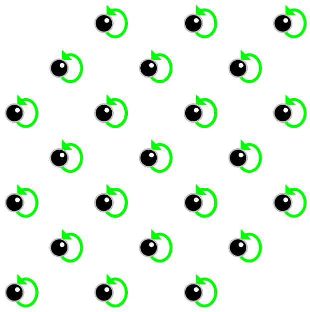


Abbildung 40: Matrixmultiplikation imperativ

Systolisch

Der systolische Ansatz erfüllt zusätzlich die Anforderung, daß jede Variable nur einmal gelesen oder geschrieben werden darf. Außerdem ist nur nachbarschaftliche Kommunikation erlaubt. Deshalb muß der Zugriff auf die Arrayelemente von a[ ] und b[ ] entsprechend modifiziert werden.

```
FOR(i=0; i<3; i++)
    FOR(j=0; j<3; j++)
        FOR(k=0; k<3; k++)
        {
            1: a[i, j, k] = a[i–1, j, k];
            2: b[i, j, k] = b[i, j–1, k];
            3: c[i, j, k] = c[i, j, k–1] + a[i, j, k] · b[i, j, k];
        }
```

Die Abbildungen 41 und 42 zeigen jeweils die Indexräume der drei Anweisungen, einmal ohne und einmal mit den Abhängigkeiten zwischen den Anweisungen. Der obere Indexraum gehört jeweils zu Anweisung 2 und die beiden darunter (von links nach rechts) zu den Anweisungen 1 und 3.

3.3.2 Polynomprodukt

Ein weiteres Beispiel, das in der Literatur oft zitiert wird, ist das Produkt zweier Polynome. Es hat die gleichen Vorzüge wie die Matrixmultiplikation, ist aber noch etwas einfacher, da es sich hierbei nur um zwei verschachtelte Schleifen handelt. Hier wollen wir jedoch zusätzlich zu dem imperativen und systolischen Ansatz noch einen funktionalen Lösungsweg betrachten.

$$c_4 \cdot x^4 + \ldots + c_1 \cdot x + c_0 = (a_2 \cdot x^2 + \ldots + a_1 \cdot x + a_0) \cdot (b_2 \cdot x^2 + \ldots + b_1 \cdot x + b_0)$$

Imperativ

```
FOR(i=0; i<3; i++)
    FOR(j=0; j<3; j++)
        c[i+j] = c[i+j] + a[i] · b[j]
```

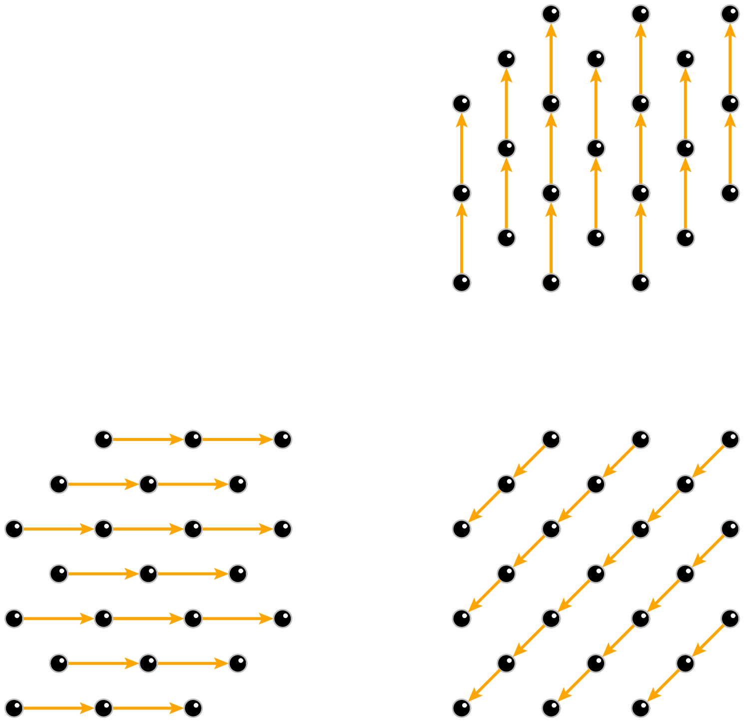


Abbildung 41: Matrixmultiplikation systolisch

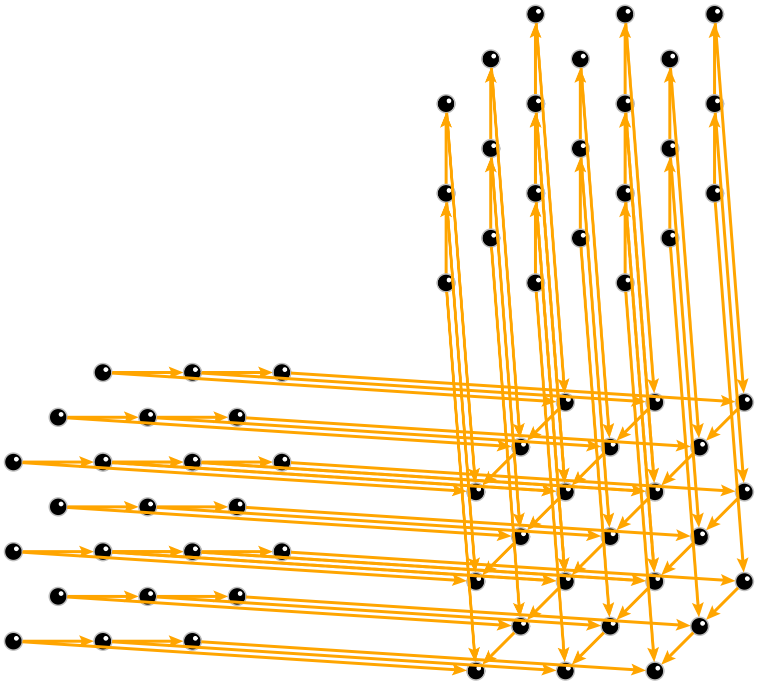


Abbildung 42: Matrixmultiplikation systolisch mit allen Abhängigkeiten

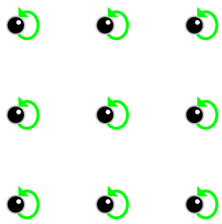


Abbildung 43: Polynomprodukt imperativ

Funktional

Beim funktionalen Ansatz existieren immer noch mehrfache Lesezugriffe auf a[i] und b[i], aber die mehrfachen Schreibzugriffe auf c[i, j] sind eliminiert, d.h. die „single assignment“-Bedingung ist erfüllt.

```
FOR(i=0; i<3; i++)
    FOR(j=0; j<3; j++)
        c[i, j] = c[i-1, j+1] + a[i] · b[j];
```

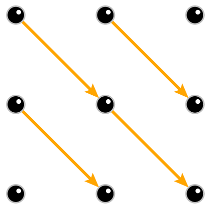


Abbildung 44: Polynomprodukt funktional

Systolisch

```
FOR(i=0; i<3; i++)
    FOR(j=0; j<3; j++)
    {
        1: a[i, j] = a[i, j+1]
        2: b[i, j] = b[i-1, j]
        3: c[i, j] = c[i-1, j+1] + a[i, j] · b[i, j]
    }
```

Auch in Abbildung 45 gehört der obere Indexraum zu Anweisung 2, die beiden unteren zu 1 und 3.

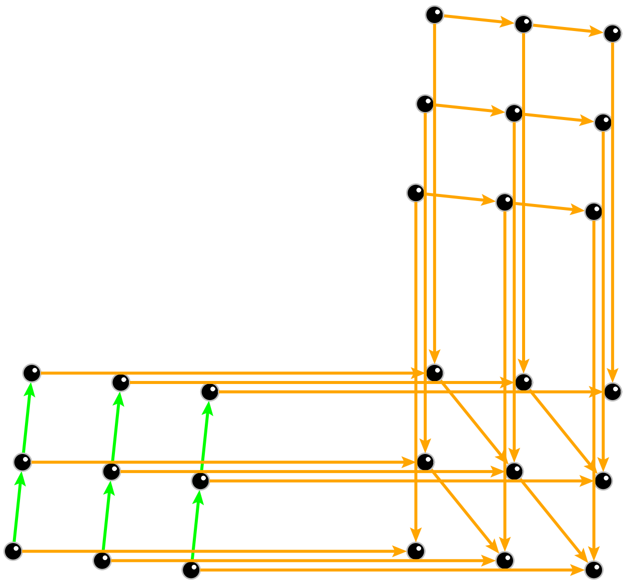


Abbildung 45: Polynomprodukt systolisch

4 Fazit

4.1 Einschränkungen

Vorab sei gesagt, daß alle hier angesprochenen Probleme für DisPo2 nicht grundsätzlich unlösbar sind. Lediglich die Verarbeitungsgeschwindigkeit nimmt bei zunehmender Größe der Indexräume ab.

4.1.1 Höherdimensionale Räume

Es liegt in der Natur der Sache, daß eine anschauliche Darstellung von Räumen mit einer Dimension größer als 3 überhaupt Schwierigkeiten mit sich bringt, insbesondere die Darstellung auf einem zweidimensionalen Bildschirm. Es wäre nun ohne weiteres möglich gewesen, jeden beliebigen Raum, also auch solche mit einer Dimension größer als 3, auf eine zweidimensionalen Ebene zu projezieren. Die Anschaulichkeit hätte auf jeden Fall darunter gelitten. So erscheint die Lösung, einzelne Dimensionen „auszublenden“, als die beste, auch wenn das Ergebnis nicht immer sofort intuitiv verständlich scheint.

4.1.2 Anzeige großer Indexräume bzw. vieler Abhängigkeiten

Im Prinzip ist die Größe eines Indexraumes unerheblich für die Darstellung mit DisPo2. Die Praxis zeigt jedoch, daß Indexräume, die nicht mehr komplett am Bildschirm dargestellt werden können, schnell unübersichtlich werden. Bis zu einem gewissen Grad schafft hier die Skalierung auf eine kleinere Größe Abhilfe. Aber auch dem sind Grenzen gesetzt, da kleine Details bei einer weiteren Verkleinerung natürlich schnell verschwinden.
Auch leidet die Geschwindigkeit der Anzeige unter einer steigenden Anzahl von Indexpunkten nicht unerheblich. Dazu kann aber angemerkt werden, daß in vielen Fällen ein „kleines“ Beispiel ausreichend ist, um die Charakteristika des Indexraumes einer bestimmten Anweisung darzustellen.

Der Hauptgrund für die Verlangsamung von DisPo2 bei größeren Indexräumen ist jedoch, wie schon erwähnt, die große Zahl von Abhängigkeiten, die solche Indexräume mit sich bringen, und ihre korrekte Darstellung. Es empfiehlt sich daher, in diesem Falle z.B. vor einer Rotation die Anzeige der Abhängigkeiten zu deaktivieren und nachdem der Indexraum die gewünschte Lage erreicht hat, wieder zu aktivieren.

4.2 Erweiterungen

An sich ist DisPo2 bereits ein äußerst leistungsfähiges Tool. Trotzdem sind einige Ansätze zur Erweiterung und Verbesserung denkbar, von denen im folgenden einige erläutert werden sollen.

4.2.1 Anzeige von Schedule und Allokation

Neben der Simulation auf dem Zielindexraum einer Anweisung wäre es manchmal wünschenswert, bereits am Quellraum die Auswirkungen des Schedulings und der Allokation überprüfen zu können. Dazu wäre eine Darstellung der Scheduling-Hyperebenen und der Allokationslinien im Quellpolytop notwendig.

4.2.2 Verbesserung der Druckausgabe

Ein zweiter Punkt wäre die Verbesserung der Druckausgabe von DisPo2. Der bestehende Ausdruck enthält momentan lediglich den Inhalt des Canvas, was jedoch nicht immer dem Wunsch des Benutzers entspricht. So wäre es z.B. sinnvoll neben den Indexräumen auch die zugehörigen Koordinatenkreuze und Anweisungsnummern auszugeben, oder überhaupt eine Auswahl auszudruckender Indexräume zuzulassen. Evtl. wäre auch eine vom Benutzer einzugebende Überschrift von einigem Vorteil.

4.2.3 Verbesserung der Verarbeitungsgeschwindigkeit

Der letzte und ganz entscheidende Punkt ist die Verarbeitungsgeschwindigkeit von DisPo2. Diese ist zwar für kleine Indexräume durchaus ausreichend, erreicht aber bei größer werdenden Indexräumen, vor allem bei steigender Anzahl von Abhängigkeiten, schnell ihre Grenzen. Hauptverantwortlich dafür sind, wie gesagt, die Darstellung der Abhängigkeiten und das Zeichnen der Indexkugeln.
Eine Möglichkeit, die Darstellung der Abhängigkeitspfeile zu beschleunigen, wäre der Einsatz einer Tk-Erweiterung, die ein dreidimensionales Canvas zur Verfügung stellt. Die korrekte Anzeige der Abhängigkeitspfeile geschähe dann automatisch. Ob der dabei gewonnene Geschwindigkeitszuwachs allerdings nennenswert wäre, bleibt fraglich.
Einen erheblichen Geschwindigkeitsvorteil beim Zeichnen der Indexkugeln würde jedoch der Einsatz von Bitmaps erzielen. Diese lassen sich im Mittel doppelt so schnell darstellen, wie die frei gezeichneten Kreise, allerdings zu Lasten der beliebigen Skalierbarkeit. Dies ist der Hauptgrund, warum bei der Implementierung von DisPo2 darauf verzichtet wurde.

4.3 Zusammenfassung

Abschließend läßt sich sagen, daß mit DisPo2 ein umfangreiches und mächtiges Tool zur Darstellung von Anweisungsindexräumen bei der Schleifenparallelisierung im Rahmen des LooPo-Projektes zur Verfügung steht. Es entspricht den Anforderungen der Schleifenparallelisierung durch die Anzeige des Quell- und Zielindexraumes der Anweisungen, sowie der Simulation des Zielprogramms und der Darstellung der Abhängigkeiten. Außerdem ermöglicht es die verschiedensten Betrachtungweisen der Indexräume durch diverse Projektionsmöglichkeiten und Manipulationen ihrer Lage und Größe. Trotz seiner Fülle von Möglichkeiten zeichnet sich DisPo2 durch eine hohe Benutzerfreundlichkeit aus und trägt so hoffentlich zu einer weiten Verbreitung von LooPo bei.

A DisPo2's Dateien

Im folgenden die Liste aller Dateien, die zu DisPo2 gehören, aufgeteilt nach den Verzeichnissen, in denen sie sich befinden.

<Loopodir>/dispo2:

aux.cc	Einge allgemeine Hilfsfunktionen, z.B. Typumwandlungen von „int“ nach „String“ usw...
canvas.cc	Funktionen, die mit dem Zeichnen der Indexräume im Canvas befaßt sind.
convert.cc	Funktionen zur Elimination der Strukturparameter und zur Auswahl von Schleifen.
convert_win.cc	Fensterdefinition und Callbacks des Convert-Fensters (Elimination von Strukturparametern).
dep_filter.cc	Fensterdefinition und Callbacks des Filter-Fensters (Auswahl von Anweisungen und Abhängigkeiten).
deps.cc	Funktionen zur Darstellung der Abhängigkeiten.
dispo.cc	Hauptfile von DisPo2. Es initialisiert die Datenstruktur und erledigt den Aufruf sämtlicher Komponenten.
dispo2*	Ausführbare Datei
dispo2.dep	Include-Abhängigkeiten, wird nur vom Makefile benötigt.
dispo2_win.cc	Fensterdefinition und Callbacks des Hauptfensters.
groups.cc	Funktionen, die sich mit der Gruppeneinteilung der Anweisungen befassen.
index_func.cc	Funktionen zur Berechnung des Indexraumes aus einem Ungleichungssystem.
index_struct.h	Datenstruktur
loops_win.cc	Fensterdefinition und Callbacks des Loops-Fensters (Auswahl von Schleifen).
Makefile	Makefile für die Compilation.
trafomat.cc	Funktionen zur Berechnung des Zielindexraumes.

<Loopodir>/lib:

Convert.tcl	Tcl/Tk-Code für das Convert-Fenster.
Dispo2.tcl	Tcl/Tk-Code für DisPo2's Hauptfenster.
Filter.tcl	Tcl/Tk-Code für das Filter-Fenster.
Loops.tcl	Tcl/Tk-Code für das Loops-Fenster.
while_borders	Enthält die Anzahl der Schleifendurchläufe von WHILE-Schleifen.

<Tempdir>:

<Projekt>.sta	Liste der im Filter ausgewählten Anweisungen.
<Projekt>.fil	Liste der im Filter ausgewählten Abhängigkeiten.

Die Dateien im <Tempdir> sind nur zur Laufzeit von DisPo2 existent und nur für den internen Gebrauch bestimmt. Ihr Format ist wie folgt:

<Projekt>.sta:

2	Anzahl der gewählten Anweisungen
1	Nummer der Anweisung
3	Nummer der Anweisung

<Projekt>.fil:

1	Anzahl der Abhängigkeiten
3	Anzahl der Variablen
0	Anzahl der Parameter
0	Anzahl der Pseudo-Parameter
1	Nummer der Abhängigkeit
2	Typ der Abhängigkeit
4	Anzahl der Zeilen
1	Hauptnenner
0 0 -1 2	
0 1 0 0	Quellraum der Abhängigkeit
0 -1 0 2	
1 0 0 0	
4	Anzahl der Zeilen
1	Hauptnenner
0 0 -1 2	
0 1 0 0	Zielraum der Abhängigkeit
0 -1 0 2	
1 0 0 0	
1	Nummer der Quellanweisung der Abhängigkeit
1	Nummer der Zielanweisung der Abhängigkeit
3	Anzahl der Zeilen
1	Hauptnenner
1 0 0 -1	
0 1 0 0	h-Transformation
0 0 1 0	

Und zum Schluß noch das Format der für die Darstellung von WHILE-Schleifen notwendigen Datei „while_borders“ (hier ein Beispiel für einen dreifach verschachtelten WHILE-Schleifensatz):

while_borders:

2	Anzahl der Durchläufe der äußersten Schleife
3	Anzahl der Durchläufe der mittleren Schleife (1. Durchlauf der äußersten)
1 2 3	Anzahl der Durchläufe der innersten Schleife
2	Anzahl der Durchläufe der mittleren Schleife (2. Durchlauf der äußersten)
1 2	Anzahl der Durchläufe der innersten Schleife

B Programmtext

Hier einige wesentliche Stellen aus dem Programmtext, für die ein paar zusätzliche Bemerkungen hilfreich erscheinen.

B.1 Datenstruktur

```
#include "../general/lmaths.h"
```
Enthält u.a. die mathematischen Datentypen „Ineq_system" und „Matrix".

```
#include "../general/dep_stat.h"
```
Enthält alle Datentypen, die mit Abhängigkeiten zu tun haben, wie z.B. „dependence_type".

```
#include "../scanpars2/symtab.h"
```
Enthält den Datentyp für die Symboltabelle „Symbol_Table".

Dies ist die Struktur für das Koordinatenkreuz. Sie enthält die Nummer der Anweisung, zu der es gehört, und die Dimension des Kreuzes. Die Dimension der Anweisung muß nicht mit der Dimension des Kreuzes übereinstimmen, da bei Anweisungen mit einer Dimension größer als 3, einige Dimensionen gestrichen werden müssen. Danach folgen für jede der drei möglichen Achsen die Beschriftung und die Koordinaten der Spitze.

```
typedef struct _cross
{
    int     nr_stat;
    int     cr_dim;

    char*   x_name;
    float   x_x_val;
    float   x_y_val;
    float   x_z_val;

    char*   y_name;
    float   y_x_val;
    float   y_y_val;
    float   y_z_val;

    char*   z_name;
    float   z_x_val;
    float   z_y_val;
    float   z_z_val;
}
co_cross;
```

Die nächste Struktur speichert Informationen über Zielanweisungen. Jeder Knoten der verketteten Liste enthält als erstes die Nummer der Zielanweisung, dann die Transformationsmatrix, mit deren Hilfe die Zielindexpunkte aus den Quellindexpunkten berechnet werden, und die Bedingung, unter der genau diese Transformation zur Anwendung kommt. Zum Schluß kommen noch die gezeichneten Dimensionen des Zielraums, da auch hier die volle Dimension größer als 3 sein kann.

```
typedef struct _targetinfo
{
    int                   trg_stat;

    Ineq_system*          condition;
    Matrix*               trafo;

    int                   dim1;
    int                   dim2;
    int                   dim3;

    struct _targetinfo*   next;
}
targetinfo;
```

Diese Datenstruktur speichert als Gegenstück zur vorhergehenden die Informationen über die Quellanweisungen. Sie ist ebenfalls eine einfach verkettete Liste und jeder Knoten enthält die Nummer der Anweisung, die der Knoten repräsentiert. Außerdem speichert er deren Gruppennummer und einen Zeiger auf die Struktur mit den aus der Quellanweisung entstandenen Zielanweisungen.

```
typedef struct _stmntinfo
{
    int                   statement_nr;
    int                   group_nr;

    targetinfo*           target_stats;

    struct _stmntinfo*    next;
}
stmntinfo;
```

Die nächste Struktur ist die Basis des Indexraums einer Anweisung. Sie besteht wiederum aus einer verketteten Liste, deren Knoten jeweils einen einzelnen Indexpunkt eines Polytops enthalten. Außer den Schleifen-, Welt- und Bildschirmkoordinaten dieses Indexpunktes beinhaltet jeder Knoten eine eindeutige Nummer zur Identifizierung der entsprechenden Indexkugel im Canvas und das Aktiv-Signal für die Simulation. Aufgrund dieses Eintrags wird der entsprechende Indexpunkt bei der Simulation normal oder hervorgehoben gezeichnet.
Gehört der Indexpunkt zu einem Quellindexraum, so zeigt das nächste Feld auf den zugehörigen transformierten Zielindexpunkt und das „untransformed_node"-Feld hat keine Bedeutung. Ist er dagegen Teil eines Zielindexraum, so ist das „transformed_node"-Feld leer und das nächste zeigt auf den entsprechenden ursprünglichen Quellindexpunkt.

```
typedef struct _polyt
{
    int             i_loop;
    int             j_loop;
    int             k_loop;

    float           x_val;
    float           y_val;
    float           z_val;

    float           x_screen;
    float           y_screen;

    int             id;
    Boolean         active;

    struct _polyt*  transformed_node;
    struct _polyt*  untransformed_node;

    struct _polyt*  next;
}
polyt;
```

Folgende Struktur speichert Abhängigkeiten. Jeder Knoten der Liste repräsentiert genau einen Abhängigkeitspfeil und enthält Anweisungsnummer und Indexpunkt sowohl der Quelle, als auch des Zieles der Abhängigkeit. Außerdem beinhaltet er den Abhängigkeitstyp und die Identifikation des Pfeiles im Canvas.

```
typedef struct _deps
{
    int               source_stat;
    polyt*            source_node;

    int               dest_stat;
    polyt*            dest_node;

    dependence_type   dep_type;
    int               id;

    struct _deps*     next;
}
deps;
```

In der nächsten Datenstruktur wird ein Indexraum mit allen notwendigen Informationen gespeichert. Zuerst kommen die Dimension des Indexraumes und welche davon gezeichnet werden, dann der Skalierungsfaktor, anschließend minimale und maximale Bildschirm- und Weltkoordinaten, dann die Mitte des Indexraumes in X-, Y- und Z-Richtung. Am Schluß befinden sich Zeiger auf die eigentlichen Indexpunkte und das zugehörige Koordinatenkreuz.

```
typedef struct _FullPoly
{
    int         dimension;
    int         first_loop;
    int         second_loop;
    int         third_loop;

    float       old_size;

    float       x_min, x_max;
    float       y_min, y_max;
    float       z_min, z_max;
    float       x_world_min, x_world_max;
    float       y_world_min, y_world_max;
    float       x_axis, y_axis, z_axis;

    polyt*      polytope;

    co_cross*   cross;
}
FullPoly;
```

„Poly_List“ ist eine Liste der gezeichneten Indexräume. Jeder Knoten enthält die Nummer der korrespondierenden Anweisung, das Ungleichungssystem, das den Indexraum beschreibt, zwei Zeiger auf den Quell- und Zielraum, sowie zwei Signale, ob der betreffende Indexraum mit der Maus ausgewählt wurde und ob der Quell-, oder der Zielraum gezeichnet werden muß.

```
typedef struct _PolyList
{
    int             statement;

    Ineq_system*    src_index_room;
    FullPoly*       source_polytope;
    FullPoly*       target_polytope;

    Boolean         selected;
    Boolean         transformed;

    _PolyList*      next;
}
PolyList;
```

Folgende Struktur ist die oberste Schicht der Datenstruktur von DisPo2. Sie enthält vor allem Daten, die für alle Programmteile zugänglich sei müssen. Nämlich die Anzahl der gezeichneten Indexräume, der Projektname, das temporäre Verzeichnis, die Symboltabelle, Informationen zu den Quellanweisungen und Abhängigkeiten, Anzahl der Variablen und Strukturparameter, sowie die Werte für die Strukturparameter und zum Schluß Angaben über das Aussehen der Indexkugeln.

```
typedef struct _Project
{
    int             nr_of_polytopes;

    char*           project;
    char*           dir;

    SymbolTable*    symtab;

    int             nr_stats;
    stmntinfo*      stat_info;

    int             nr_deps;
    deps*           dependences;

    int             vars, pars;
    Matrix*         param_values;

    float           node_radius, node_distance;
    float           size;

    PolyList*       polys;
}
Project;
```

B.2 Richtung der Y-Achse

Aus der Orientierung der Y-Achse im Tk-Canvas ergibt sich ein Problem. Da sich der Koordinatenursprung des Canvas in der linken oberen Ecke befindet, ist diese nach unten gerichtet. Deshalb werden Y-Werte nach unten positiv gezählt, was nicht der Erfahrung des Benutzers entspricht, der eine nach oben orientierte Y-Achse gewohnt ist. Es ist daher notwendig, die Y-Achse des Canvas „umzudrehen", so daß z.B. bei einer Verschiebung in Y-Richtung ein positiver Verschiebungswert auch in einer Verschiebung nach oben resultiert. Der Benutzer gibt wie gewohnt den Verschiebungswert ein, der dann intern invertiert wird.
Als Beispiel möge die folgende Routine zur Verschiebung eines Indexraumes dienen.

```
FullPoly* translate(FullPoly* Fpolytope, float X_tra, float Y_tra)
{
    polyt* tmp = Fpolytope->polytope;

    while (tmp != NULL)
        {
            tmp->x_val    += X_tra;
            tmp->x_screen = tmp->x_val;
            tmp->y_val    -= Y_tra;
            tmp->y_screen = tmp->y_val;

            tmp = tmp->next;
        }

    return Fpolytope;
}
```

Abbildungsverzeichnis

Literatur

[BaFl82] M. Barner, F. Flohr: „*Analysis II*“, Walter de Gruyter & Co., 1982.

[BaLe92] M. Barnett, C. Lengauer: „*Loop parallelisation and unimodularity*“, Parallel Processing Letters, 1992.

[Bane93] U. Banerjee: „*Loop Transformations For Restructuring Compilers, The Foundations*“, Kluwer Academic Publishers, 1993.

[Bane94] U. Banerjee: „*Loop Parallelization*“, Kluwer Academic Publishers, 1994.

[FDFH90] J. Foley, A. van Dam, S. Feiner, J. Hughes: „*Computer Graphics: Principals and Practice*“, Addison-Wesley Publishing Company, 1990.

[FDFHP94] J. Foley, A. van Dam, S. Feiner, J. Hughes, R. Phillips: „*Introduction to Computer Graphics*“, Addison-Wesley Publishing Company, 1994.

[Feau91] P. Feautrier: „*Dataflow analysis of array and scalar references*“, Int. J. Parallel Programming, 20(1), 1991.

[Geig97] M. Geigl: „*Parallelization of Loop Nests with General Bounds in the Polyhedron Model*“, Diplomarbeit, Fakultät für Mathematik und Informatik der Universität Passau, in Bearbeitung.

[GrLe94a] M. Griebl, C. Lengauer: „*On Scanning Space-Time Mapped WHILE Loops*“, CONPAR'94, Lecture Notes in Computer Science, Springer-Verlag, Seiten 677-688, 1994.

[GrLe94b] M. Griebl, C. Lengauer: „*On the Space-Time Mapping of WHILE Loops*“, Parallel Processing Letters vol. 4 nr. 3, Seiten 221-232, 1994.

[GrLe96] M. Griebl, C. Lengauer: „*The Loop Parallelizer LooPo - Announcement*“, Lecture Notes in Computer Science, Springer-Verlag, 1996.

[Keim97] H. Keimer: „*Datenabhängigkeitsanalyse zur Schleifenparallelisierung: Vergleich und Erweiterung zweier Ansätze*“, Diplomarbeit, Fakultät für Mathematik und Informatik der Universität Passau, 1997.

[Leng93] C. Lengauer: „*Loop parallelization in the polytope model*“, CONCUR'93, Lecture Notes in Computer Science, Springer-Verlag, Seiten 398-416, 1993.

[LeGr95] C. Lengauer, M. Griebl: „*On the Parallelisation of Loop Nests containing WHILE Loops*“, Proc. pAs'95, IEEE Computer Society Press, Seiten 10-18, 1995.

[Loop96] LooPo-Team: „*LooPo, Loop parallelisation in the Polytope model*“, Programmdokumentation, Fakultät für Mathematik und Informatik der Universität Passau, 1996. http://www.uni-passau.de/loopo/.

[Lip77] S. Lipschutz: „*Theorie and Problems of Linear Algebra*“, McGraw-Hill Book Company GmbH, 1977.

[NeSp86] W. Newman, R. Sproull: „*Grundzüge der interaktiven Computergrafik*“, McGraw-Hill Book Company GmbH, 1986.

[Oust94] J. K. Ousterhout: „*Tcl and the Tk Toolkit*“, Addison-Wesley Publishing Company, 1996.

[Schir92] C. Schirmer: „*Die Programmiersprache C. Die ausführliche Beschreibung aller Sprachelemente.*“, Carl Hauser Verlag, 1992.

[Str87] B. Stroustrup: „*Die C++ Programmiersprache*“, Addison-Wesley Publishing Company, 1987.

[Wetz95] S. Wetzel: „*Automatic Code Generation in the Polytope Model*“, Diplomarbeit, Fakultät für Mathematik und Informatik der Universität Passau, 1995.

[Wien95] C. Wieninger: „*Automatische Methoden zur Parallelisierung im Polyedermodell*“, Diplomarbeit, Fakultät für Mathematik und Informatik der Universität Passau, 1995.

[Wüst95] A. Wüst: „*DisPo2, grafische Darstellung der Indexräume vernesteter Schleifen*“, Programmierpraktikum, Fakultät für Mathematik und Informatik der Universität Passau, 1995.

[Zwick93] A. Zwicklbauer: „*Grafische Polyederdarstellung*“, Diplomarbeit, Fakultät für Mathematik und Informatik der Universität Passau, 1993.

Eidesstattliche Erklärung
Hiermit erkläre ich eidesstattlich, daß ich diese Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe, und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Die Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, 17. März 1997